DIPLOMARBEIT

Ein objektorientierter Entwurf von wiederverwendbarer Scheduling Software

ausgeführt am Institut für Informationssysteme
Abteilung für Datenbanken und Expertensysteme
der Technischen Universität Wien

unter der Anleitung von
o.Univ.Prof. Dr. Georg Gottlob

und

Univ.Ass. Dr. Wolfgang Slany
als verantwortlich mitwirkendem Universitätsassistenten

durch

Witold Hendrysiak
Matr.Nr. 8727099
Hauptstr. 115 a
A-2372 Gießhübl

Wien, im Mai 1995

Abstract

DomFLIP++ is the knowledge engineering module of the *FLIP++ (pronounce: StarFlipPlusPlus) project. *FLIP++ is a tool for optimizing multiple criteria problems. It uses fuzzy constraints to model optimizing criteria and applies algorithms such as Tabu search or genetic algorithms to the problems. DomFLIP++ is a C++ library. It allows the definition of new optimization problems. It helps a domain engineer to design the structure of a new problem at hand. Methods and datastructures are specific to the new problem. However, there is a domain independent interface to other *FLIP++ modules such as OptiFLIP++, DynaFLIP++, and InterFLIP++. After each iteration in the optimization process, the considered instantiations of the problem are evaluated. Each evaluation produces a list of violated constraints. For each constraint, a list of repair steps is defined that can be used to increase the score of this constraint in further iterations of the optimization. A domain can be fine-tuned through modification of constraints, through editing their repair lists, and through change in the optimizing parameters. A well-tuned domain can be successfully applied for optimization. Object-oriented design and implementation makes this module easy extendable and reuseable. Definition of new domains, system extensions with new optimizing algorithms, and definition of specific domain-dependent repair steps can be done efficiently. DomFLIP++ is tested on a real-world example, namely scheduling the steel plant LD3 in Linz, Austria.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Diese Arbeit ist eine Weiterentwicklung eines Projektes, welches das Institut für Informationssysteme an der Technischen Universität in Wien und das Christian Doppler Labor für Expertensysteme begonnen hat [6].

1.1 Entwicklung des Projektes

In dem Projekt wurde gemeinsam mit dem Alcatel ELIN Forschungszentrum ein Expertensystem entwickelt, welches die Planung der optimalen Reihenfolge von Schmelzvorgängen im Stahlwerk der Böhler Ges.m.b.H. durchführt bzw. den menschlichen Experten dabei unterstützt [5]. Das Expertensystem ist seit Anfang 1993 im Einsatz.

In dem Projekt wurden verschiedene Optimierungsalgorithmen implementiert, getestet und verglichen.

1.1.1 Optimierungsalgorithmen

Die Arbeit von Mario Girsch [10] beschreibt die Stärken und Schwächen von Genetischen Algorithmen. Er zeigt, daß ein Genetischer Algorithmus bei einem komplexen Schedulingproblem, bei dem die verschiedensten Einschränkungen zu beachten sind, ohne großen Aufwand implementiert werden kann. Ein Vorteil, der sich bei den Testläufen gezeigt hat, ist, daß der Genetische Algorithmus nicht in lokalen Maxima hängenbleibt, wie es beim Suchalgorithmus vorkommen kann. Als Nachteil wurde die Tatsache erwähnt, daß der Algorithmus nur für dieses spezielle Problem implementiert wurde und deswegen für andere Schedulingprobleme nicht anwendbar ist.

Günther Skele in seiner Diplomarbeit [24] beschreibt einen Tabu Suche Algorithmus. Dieser wird mit dem Random Repair Algorithmus verglichen. Weiters wurden auch andere Optimierungsalgorithmen verglichen.

Genetischer Algorithmus

Mit Genetischen Algorithmen [11] versucht man die Evolution der Natur nachzubilden. Durch Mutationen und Kreuzungen entwickeln sich Lebewesen weiter, und manche sich so ergebenden Neuschöpfungen sterben, weil sie den natürlichen Lebensbedingungen nicht gewachsen sind. Die Idee beim Einsatz von Genetischen Algorithmen zur Optimierung von Lösungen ist, daß eine Menge von Kandidatenlösungen in der sogenannten Population betrachtet werden. Elemente der Menge sind Chromosomen, repräsentiert durch Zeichenketten. Ein Chromosom enthält eine geordnete Liste von Genen, die im einfachsten Fall Binärwerte sind. Ein Gen kann aber auch eine Zahl oder ein Symbol sein. Mit Hilfe einer Fitnessfunktion wird ein Chromosom evaluiert.

Die Anfangspopulation wird zufällig erzeugt und entwickelt sich dann in Iterationen (Generationen) fort. In jeder Generation werden die Chromosomen genetischen Operatoren und einem Auswahlmechanismus unterworfen. Durch die Operatoren werden Informationen über Generationen weitergegeben, und die Auswahl mit Hilfe der Fitnessfunktion erlaubt das Überleben der besten Chromosome.

Es gibt eine Vielzahl von Genetischen Operatoren, die in der Literatur vorgestellt wurden. Bei den Mutationen werden einzelne Gene eines Chromosoms zufällig geändert. Sind die Gene Binärwerte, dann kann z.B. ein zufälliges Gen den inversen Wert annehmen. Beim gegenseitigen Austausch (crossover) werden von zwei Eltern ein oder mehrere Kinder durch Austausch von Genen erzeugt. Dieses Modell wurde mehrfach für Schedulingprobleme untersucht. Durch die Anwendung ergibt sich wieder eine Art iterativer Suche nach einem optimalen Schedule. Dabei werden meist spezielle Operatoren eingeführt, die entsprechend der Form des Schedules formal-korrekte neue Schedules erzeugen.

In der einfachsten Form wird die Reihenfolge der Jobs bzw. Operationen auf einer Maschine als Zeichenkette (Chromosom) repräsentiert. Jede Operation ist ein Gen. Durch eine Mutation können nun die Positionen von zwei zufällig ausgewählten Genen vertauscht werden. Ein anderer Mutationsoperator wählt zufällig zwei Gene aus und stellt das zweite vor das erste. Beim Austausch werden bei einem Gen wieder zufällig zwei Positionen ausgewählt, und die Gene an diesen Positionen werden mit dem zweiten Chromosom ausgetauscht [14].

Tabu Suche

Tabu Suche *(tabu search)* ist ein allgemeines Problemlösungsverfahren um Optimierungsprobleme zu lösen [4]. Man geht hier davon aus, daß man bereits eine zulässige Lösung für ein gegebenes Problem gefunden hat und diese iterativ verbessert. Dazu muß eine Evaluierungsfunktion existieren, die einer Lösung einen Wert zuweist und so einen Vergleich von zwei zulässigen Lösungen erlaubt. In jeder Iteration wird nun durch eine lokale heuristische Suche eine neue Lösung gesucht, deren Evaluierung besser ist als die alte. Üblicherweise wird davon ausgegangen, daß eine explizite Funktion existiert, die alle zulässigen Nachbarlösungen einer gegebenen Lösung liefert.

Bei einem Schedulingproblem kann diese Nachbarschaftsrelation durch das Vertauschen von zwei Jobs oder zwei Operationen des Schedules definiert sein. Vertauschen wir also zwei Jobs J_i und J_j im Schedule S, dann ergibt sich ein neuer Schedule S', der ähnlich dem alten ist. Die Zulässigkeit dieses neuen Schedules muß dann jedoch noch explizit überprüft werden. Existieren noch nicht eingeplante Jobs, dann können diese auch mit Jobs des Schedules vertauscht werden. Die bisherigen Ausführungen laufen auf eine sogenannte hill climbing Strategie hinaus. Wenn aber bei der lokalen Suche nur bessere Lösungen akzeptiert werden, dann kann es passieren, daß die globale Suche nach einer optimalen Lösung in einem lokalen Maximum steckenbleibt. Deswegen werden bei der Tabu Suche auch Verschlechterungen akzeptiert. Wenn Verschlechterungen akzeptiert werden, ist aber nicht mehr automatisch gesichert, daß die Suche sich nicht in Schleifen durch den Problemlösungsraum bewegt. Es können also neue Lösungen (Schedules) erzeugt werden, die schon einmal generiert wurden. Um Endlosschleifen zu verhindern wird eine Tabu Liste eingeführt, die bestimmte Lösungen sperrt. Im allgemeinen Fall würde man ganze Lösungen, zu denen man nicht mehr zurückkehren möchte, in dieser Liste halten. Meist wird das zu aufwendig sein und die Lösung wird nur durch bestimmte Charakteristika beschrieben. Wenn ein Schedule als Liste repräsentiert ist und der Job J_i bzw. J_j an der k-ten bzw. l-ten Stelle stehen, dann steht J_i im neuen Schedule an der l-ten Stelle. In der Tabu Liste könnte nun stehen, daß es verboten ist, Job J_i wieder zur k-ten Stelle zu positionieren. Eine andere, weniger strenge Formulierung wäre das Verbot, beide Jobs wieder zu vertauschen. Da bei einer Tabu Suche kein Backtracking zu einer alten Lösung vorgesehen ist, wird die Länge der Tabu Liste eingeschränkt. Das bedeutet, nach einer Anzahl von t Iterationen kann man zu einer Lösung zurückkehren und von dort einen anderen Weg bei der Suche verfolgen. Das heißt, eine Tabu Liste ist eine Art Ringpuffer der Länge t, und ein neues Element überschreibt immer das älteste.

Es kann nun natürlich passieren, daß Schleifen der Größe t entstehen. Um dies zu verhindern, wird noch ein generelles Abbruchkriterium für die Tabu Suche spezifiziert. Man merkt sich, in welcher Iteration die bisher beste Lösung gefunden wurde und gibt eine Schranke an, wieviele Iterationen erlaubt sind, bis wieder eine Verbesserung gefunden werden muß.

Simulated Annealing

Simulated Annealing ist eine Methode, die für viele Optimierungsprobleme angewendet wird und seine Wurzeln in der Simulation von physikalischen Prozessen hat [12]. Dort wurde mit dieser Simulation das Abkühlen eines Festkörpers auf seinen Grundzustand, d.h. den Zustand mit minimaler Energie modelliert.

Simulated Annealing kann auch zur Suche nach besseren Schedules verwendet werden. Dabei wird wieder davon ausgegangen, daß für einen Schedule ein Evaluierungswert berechnet werden kann. Von einem gegebenen Schedule S_i mit dem Evaluierungswert c_i werden die Nachbarschedules S_j erzeugt und bewertet. Wenn der Wert c_j eines Schedules S_j kleiner ist als c_i , dann wird der Schedule als neuer bester Schedule ausgewählt, und es wird versucht, diesen weiter zu verbessern. Um nun aus lokalen Maxima herauszukommen, wird ein Wahrscheinlichkeitsmodell benutzt, mit dem auch schlechtere Schedules akzeptiert werden. Die Wahrscheinlichkeit P, mit der ein schlechterer Schedule akzeptiert, wird, ist wie folgt:

$$P = e^{-\frac{c_j - c_i}{t}}$$

Der Wert t stellt einen Temperaturparameter dar, der die Suche steuert und während der Suche verringert wird. Die Wahrscheinlichkeit, daß ein schlechterer Schedule akzeptiert wird, verringert sich so im Laufe der Suche. Außerdem ist die Wahrscheinlichkeit, daß große Verschlechterungen akzeptiert werden, kleiner als die von kleinen Veränderungen.

1.2 Übersicht über das *FLIP++

Auf diese Weise gewonnenes Wissen will man jetzt so verwenden, daß die bereits im Projekt entwickelten und getesteten Teile für ein Domänenunabhängiges Schedulingtool verwendet werden können.

So entstand auch die Frage was man machen muß, um bereits existierenden Optimierungsalgorithmen für andere Problemstellungen verwenden zu können.

Ich versuche in dieser Diplomarbeit eine Antwort auf diese Frage zu geben. Die Antwort ist aber nicht so einfach und kann nie endgültig sein. Es ist viel mehr eine Entscheidung zwischen:

- Einem allgemeinen Modell, das aber für konkrete Fälle keine optimale Lösung des Problems liefert, dafür aber eine große Unterstüzung im Bereich Domain Engineering.
- Einem Modell mit vielen Restriktionen, das leider nicht für alle Probleme anwendbar ist, aber dafür die meisten Probleme sehr gut lösen kann.

In diesen Fall kann man nicht über eine Wiederverwendung, wie in [21] [20] diskutiert wird, reden. Bei einem so kleinen Projekt ist der Aufwand, den man braucht viel zu groß, um ein Repository zu entwickeln, das die Ressourcen wiederverwenden läßt. Viel vernünftiger erscheint es in dem Fall eine Schnittstelle zu allen domänenunabhängigen Modulen genau zu spezifizieren, und maximale Unterstützung bei der Entwicklung von den domänenspezifischen Teilen zu gewährleisten. Der Domainengineer muß die bereits entwickelten Module kennen und verstehen, um bei seiner Arbeit maximal entlastet zu werden.

Das *FLIP++ [STAR FLIP PLUS PLUS] Projekt wurde in sechs Module aufgeteilt. Diese wurden so modeliert, daß der Grad der Abhängigkeit zwischen den Modulen (*coupling*) möglichst schwach bleibt, der Grad der Abhängigkeit innerhalb der Module (*cohesion*) aber möglichst stark wird. Die Abbildung 1.1 zeigt die Module des neuen Projektes.

- OptiFLIP++ (Optimize Algorithms)
  Das OptiFLIP++ Modul enthält die Optimierungsalgorithmen. Bis jetzt wurde der Tabu Search Algorithmus integriert. Die Arbeiten an dem genetischem Algorithmus sind in der Endphase.
- DomFLIP++ (Domain Knowledge Engineering Tool)
  Das DomFLIP++ Modul besteht aus Datenstrukturen und Methoden, die es erlauben, eine neue Domäne zu definieren. Die Schedulingobjekte dieser Domäne können beliebig definiert werden. Die Schnittstelle zu den anderen Modulen ist aber genau spezifiziert und ändert sich nicht.
- DynaFLIP++ (Dynamic Constraints Generator)
  Das DynaFLIP++ Modul baut mit der Hilfe von Informationen über das Schedulingobjekt die in DomFLIP++ definiert wurden, sowie auch mit Einschränkungen, die mittels ConFLIP++ definiert wurden, dynamische Bäume auf, die eine Evaluierung der Domäne ermöglichen.

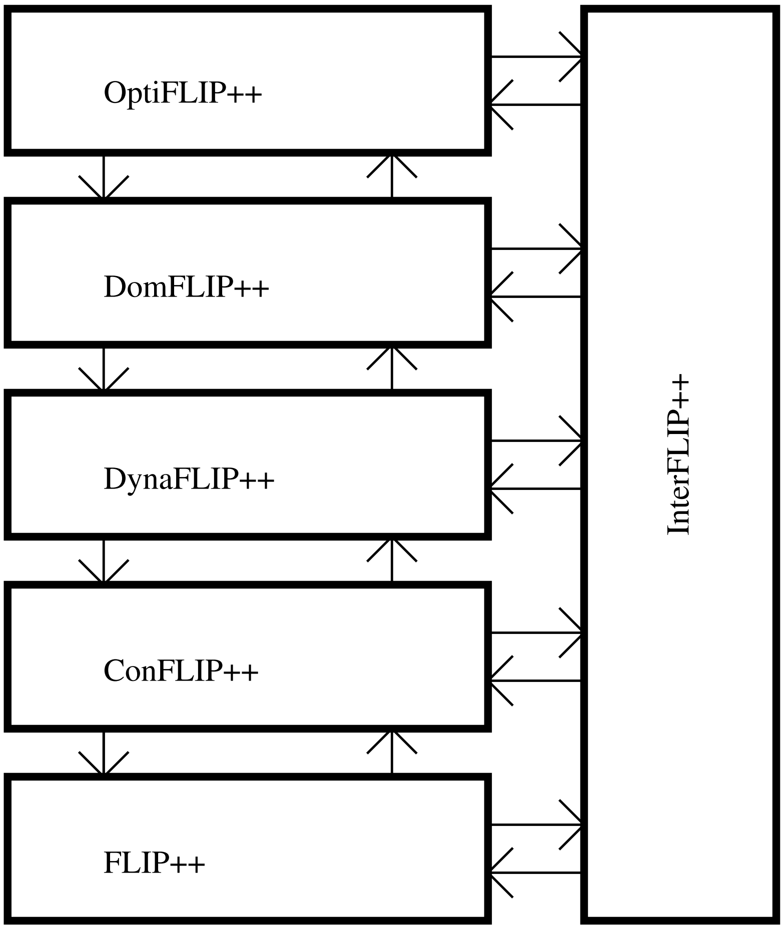

Abbildung 1.1: Module von *FLIP++

- ConFLIP++ (Fuzzy Constraints Tool)
  Dieses Modul erlaubt es, neue Daten und Regeln für die Bearbeitung mit FLIP++ zu modellieren. Hier werden alle für die Evaluierung benötigten Einschränkungen definiert.

- FLIP++ (Fuzzy Logic Inference Processor in C++)
  Dieses Modul führt die Berechnungen mit unsicheren Daten verschiedener Wichtigkeit an Hand von speziell für diese Zwecke definierten

Regeln durch. Es werden Teilstrukturen der Datenhierarchie evaluiert, die für eine Bewertung von Bedeutung sind. Das Ergebnis der Evaluierung wird in zwei Formen geliefert. Einerseits ein fixer Wert, der die Güte der Bewertung bestimmt, und andererseits eine Liste der membership Funktionen.

- InterFLIP++ (User Interface)
  Das InterFLIP++ Modul stellt eine einheitliche Benutzerschnittstelle für alle FLIP Module zur Verfügung.

Ich werde in Kapitel 2 kurz die Problematik der Softwarewiederverwendung charakterisieren. Kapitel 3 stellt eine Einführung in die Begriffswelt von Scheduling dar. Kapitel 4 beschreibt die von mir ausgewählte Lösung und das danach folgende Kapitel 5 stellt eine Fallstudie vor.

Kapitel 2

Software Wiederverwendung

In allen Herstellungsprozessen stellt die Wiederverwendung einen sehr wichtigen Aspekt dar. In der kommerziellen Herstellung von Hardware wird schon seit Jahren die Wiederverwendung bereits vorhandener Bauteile in großem Rahmen zur Erhöhung der Produktivität im Produktionsprozeß verwendet.

Durch das Sinken der Hardwarekosten steigen prozentuell gesehen die Kosten der Software. Demgegenüber steht eine jährliche Produktivitätssteigerung in der Softwareentwicklung von lediglich 3-8% innerhalb der letzten 30 Jahre. Außerdem herrscht ein Engpaß an qualifiziertem Entwicklungspersonal. Das alles führt zu einer Softwarekrise.

2.1 Ziele

Um diesen Mißverhältnis Herr zu werden, bedarf es neuer Methoden der Softwareentwicklung. Ein zielführender Weg ist die Wiederverwendung nicht nur schon einmal implementierter Teile von Code, sondern jeder Art von Information, die bei der Softwareentwicklung entsteht.

Wir sehen im Abbildung 2.1 [27] wie wichtig die Wiederverwendung bei Softwareentwicklung ist, wenn wir uns die Kostenverteilung im Software-Life-Cycle anschauen.

Die Wartung von Software Systemen beansprucht 67% der Kosten. Betrachtet man die aufgewendete Zeit für die einzelnen Aktivitäten während der Software Wartung (Abbildung 2.2), so läßt sich daraus die besondere Bedeutung des Verstehens des Programmcodes für eine erfolgreiche Software

Wartung ableiten. Durch die Software Wiederverwendung erhofft man sich eine signifikante Steigerung der Produktivität im Wartungsprozeß.

Die Bestrebungen der Qualitätssicherung im Software-Entwicklungsprozeß basierend auf internationalen Normen (z.B. ISO 9000) erhöht zusätzlich die Rolle von Software Wiederverwendung.

Entwicklungsphase	**Prozentanteil**
Anforderungsanalyse	3%
Spezifikation	3%
Design	5%
Codierung	7%
Modul Test	8%
Integrationstest	7%
Wartung	*67%*

Abbildung 2.1: Kostenverteilung im Software-Life-Cycle

Aktivitäten in der Software-Wartung	**Prozentanteil**
Verstehen des Source Codes	*50 %*
Änderungen testen	*25 %*
Änderungen planen	*10 %*
Änderungen durchführen	*10 %*
Änderungen dokumentieren	*5 %*

Abbildung 2.2: Aktivitäten in der Software-Wartung

2.2 Geschichtliche Entwicklung

Software Engineering hat, wie auch andere Disziplinen, die praktische Bedeutung von Wiederverwendung schon früh in seiner Geschichte erkannt. Aber im Gegensatz zu anderen Disziplinen schritt diese Entwicklung nur sehr langsam voran. Generische Algorithmen, Makros, Funktionen, Subroutinen und Objekte sind Beispiele für wiederverwendbare Software-Artefakte. Aufgrund der leichten Änderbarkeit von Software werden Software-Bausteine fortwährend an neue Anforderungen angepaßt. Diese Tatsache verhinderte lange Zeit die Entstehung von Standards für die Software-Entwicklung.

Erst in jüngster Zeit werden durch das zunehmende Interesse an Standards sogenannte Software-Architekturen entworfen. IAA (Insurance Application Architecture) von IBM [9] ist ein typischer Vertreter einer solchen Architektur, der die Schnittstellen von Software-Komponenten für die Versicherungsdomäne beschreibt.

Im Jahre 1968 wurde von Dough McIlroy in den Artikel *mass-produced software components* [18] die Notwendigkeit von standardisierten und katalogisierten Software-Bausteinen für eine erfolgreiche Software Wiederverwendung dargestellt.

Die erste *Software Factory* wurde von der Systems Development Corp. (SDC) Mitte der 70er Jahre eingerichtet. SDC definierte eine Menge von Prozeduren für einen konsistenten und nachvollziehbaren Software-Entwicklungsprozeß. Reuse war trotzdem kein eigenständiger Prozeß, sondern die Programmierer waren angewiesen, alles Mögliche (vom Source Code bis hin zu Entwicklungserfahrung) wiederzuverwenden.

Die erste Integration des Wiedervervendungprozesses auf organisatorischer Ebene wurde von Lanergan und Poynton im *Raytheon Missile Division Projekt* durchgeführt [13]. In diesem Projekt wurde festgestellt, daß Design und Implementierung von 60% aller kommerziellen Applikationen redundant waren und ein Potential zur Standardisierung beinhalteten. Sie stellten auch fest, daß den meisten kommerziellen Applikationen drei logische Strukturen (edit, update, report) zugrunde liegen. Diese logischen Strukturen wurden standardisiert und eine Datenbank erstellt, die alle klassifizierten Komponenten zur Verfügung stellt. Die Programmierer wurden speziell für die Benutzung dieser Bibliothek und hinsichtlich des Einsatzes der logischen Strukturen ausgebildet.

Das erste universitäre Forschungsprojekt wurde 1976 von Freeman an der University of California, Irvine, eingerichtet. Eines der ersten hervorragenden Ergebnisse war die Arbeit von Neighbors über Draco [19] [21] [20]. Draco fügt vordefinierte Software-Komponenten mittels Transformationen, Verfeinerungen und Integration zu einem neuen Software-System zusammen und arbeitet dabei mit eigenen Spezifikationssprachen für diese Transformationen.

Systematische Softwarewiederverwendung wurde von Matsumoto [16] 1980 mittels einer Software Workbench bei Toshiba realisiert. Diese Software Workbench war eine Zusammenstellung von Tools und Methoden zur Massenproduktion von Software. Prototyping, Wiederverwendung und Programm-Generierung wurden mit Management-Methoden kombiniert. Der Erfolg wurde in [17] vorwiegend auf die Wiederverwendung auf unterschiedlichen Abstraktionsniveaus (Design, Dokumentation, Testfälle) zurückgeführt.

2.3 Klassifizierung

Der Begriff Wiederverwendung *(Reusability)* wird von IEEE [Std 610.12-1990] wie folgt genormt:

> *"Reusability ist der Grad zu dem ein Software Modul oder ein anderes Entwicklungsdokument in mehr als einem Computer Programm oder Software System verwendet werden kann."*

Software Wiederverwendung *Software Reuse* wird in [22] durch Prieto-Diaz wie folgt definiert:

> *"Unter Software Wiederverwendung versteht man die Verwendung existierender Software Komponenten für die Erstellung neuer Software Systeme."*

Die Wiederverwendung bezieht sich auf alle Dokumente, die im Zuge des Software Life-Cycles entstehen. Die Arten von Software Wiederverwendung können wir aus verschiedenen Blickpunkten kategorisieren:

- **Produkt:**
  (Welche Produkte werden wiederverwendet?)

  1. **Source Code Reuse**
     stellt die am weitesten verbreitete Art von Wiederverwendung. Die in Standard-Bibliotheken zu Verfügung gestellten Prozeduren und Funktionen werden bei der Implementierung von neuen Programmen sehr oft angewendet.

2. **Design Reuse**
   bietet einen höheren Nutzen in der Wiederverwendung und kann weitgehend durch objekt-orientierte Methoden genützt werden. Ein Beispiel für einen Prototypen stellt IDeA (Intelligent Design Assistent) [15] dar, das die Wiederverwendung von Design mittels Design Schemata unterstützt.

3. **Specification Reuse**
   wird meistens gemeinsam mit dem zugehörigen Design und der entsprechenden Implementierung wiederverwendet. Dabei versucht man vor allem aus Spezifikationen automatisch Programme generieren zu lassen.

4. **Object Reuse**
   Bei der Wiederverwendung von Objekten versuchen die derzeitigen Methoden wie z.B. OOA (Object Oriented Analysis [3]), OMT (Object Modeling Technique [23]) oder OSA (Object-Oriented Systems Analysis [7]) Phasen des Software Life-Cycles zu integrieren, um das große Wiederverwendungspotential von Objekten optimal ausnützen zu können.

5. **Text Reuse**
   Die Integration von Text mit anderen Entwicklungsdokumenten zielt vorwiegend auf die Erstellung von Hypertext-Systemen für Reuse ab.

6. **Architecture Reuse**
   stellen die größte Einheit für die Wiederverwendung dar. Anwendungsdomänen werden untersucht, um generische Teile zu identifizieren, die als sogenannte Templates für die Erstellung neuer Software verwendet werden können.

- **Absicht**
  (Will man Komponenten ohne Änderungen, oder nach einer Adaptierung verwenden?)

  1. **Black-box Reuse**
     beschreibt die Wiederverwendung von Software, ohne dabei irgendwelche Modifikationen durchzuführen. Diese Art von Reuse garantiert zwar höhere Qualität und Zuverlässigkeit, die Erzeugung solcher für die Black-box Reuse geeigneten Komponenten ist aber um einiges aufwendiger und schwieriger. Ada und bis zu einem bestimmten Grad auch objekt-orientierte Programmierung verwenden dieses Black-box Prinzip für modulare, unabhängige Komponenten.

2. **White-box Reuse**
   bedeutet die Wiederverwendung von Komponenten durch Modifikation und Adaptierung. Die meisten Reuse-Programme, speziell Software Factories, verwenden White-Box Reuse. Nach und nach wird auch diese Art der Wiederverwendung zunehmends formalisiert, was sich in Parametrisierungen und eingebauter Adaptierbarkeit einer Komponente widerspiegelt. Auch hierbei spielt die Domänenanalyse eine wichtige Rolle in der Unterscheidung von Systemfamilien und somit in der Parametrisierung.

- **Technik**
  (Entweder sammelt man die wiederverwendbaren Bausteine, um aus diesen eine neue Anwendung zu bauen, oder man stellt Generatoren zu Verfügung.)

  1. **Component Reuse**
     ist die Wiederverwendung von existierenden Software-Komponenten als *building blocks* für neue Systeme. Die Basis dafür sind effiziente Bibliotheksysteme, repräsentative Ansammlungen von Komponenten und standardisierte Schnittstellen der einzelnen Komponenten. Das Zusammenfügen solcher Bausteine wird in der Praxis zumeist auf der Source-Code Ebene durchgeführt. Die Schwerpunkte der Forschung in diesem Bereich konzentrieren sich auf die Selektion und Wiederauffindung in Bibliotheken, das Verständnis und die Adaptierung, sowie die Integration dieser Komponenten mit neu entwickelten Software-Teilen.

  2. **Generative Reuse**
     verkörpert die Wiederverwendung auf Spezifikationsebene mittels Applikations-Generatoren. Diese Art von Reuse verspricht den größten Vorteil, demgegenüber steht die extrem schwierige Aufgabe, diese Technologie umzusetzen. Man versucht domänenabhängige Spezifikationssprachen formal zu repräsentieren und Grammatiken und Transformationsregeln zu definieren.

- **Modus**
  (Systematische Verwendung von wiederverwendbaren Komponenten oder informale Verwendung von Komponenten aus allgemeinen Software-Bibliotheken)

  1. **Systematic Reuse**
     wird in Software Factories eingesetzt, wo die Systematik und

formale Anwendung ebenso wie die zugehörigen Richtlinien und Metriken definiert sind. Geplantes Reuse benötigt eine Änderung der herkömmlichen Software-Entwicklung und erfordert Disziplin und eine Art Reuse-Kultur vom Software-Entwickler.

2. **Ad-hoc Reuse**
   ist eine informale Praxis der Software-Entwickler, wobei Komponenten aus allgemeinen Software-Bibliotheken selektiert werden. Diese in der Praxis sehr häufig anzutreffende Art von Reuse bewegt sich auf einer individuellen Ebene des einzelnen Programmierers und wird nicht von ganzen Projektteams eingesetzt. Die in diesen allgemeinen Software-Bibliotheken gespeicherten Software Komponenten sind auch nicht speziell für Reuse adaptiert.

- **Bereich**
  (Sind die Komponenten domänenunabhängig oder domänenspezifisch?)

  1. **Vertical Reuse**
     ist Wiederverwendung innerhalb desselben Anwendungsbereiches. Ein Ziel ist die Definition von generischen Komponenten für die Erstellung ähnlicher Systeme. Dieser Bereich schließt zahlreiche Arbeiten im Bereich der Domänen-Analyse und Domänen-Modellierung ein.
  2. **Horizontal Reuse**
     bezeichnet die Wiederverwendung von generischen Komponenten in unterschiedlichen Anwendungsbereichen. Der Schwerpunkt liegt dabei auf der Erstellung von Software Bibliotheken, der Katalogisierung und Klassifizierung von Software Komponenten.

- **Substanz**
  (Was wird wiederverwendet?)

  1. **Concept Reuse**
     schließt die Wiederverwendung von Konzepten (z.B. für die generelle Lösung einer Klasse von Problemen) ein. Typische Beispiele sind generische Algorithmen.
  2. **Component Reuse**
     Software Factories versuchen in diesem Bereich, spezielle Kataloge von Komponenten zu erarbeiten. Diese Wiederverwendung von Software-Teilen konzentriert sich auch auf die Qualität und die Zuverlässigkeit und somit die Zertifizierung von Komponenten. In

diesem Bereich werden verstärkt domänenspezifische Kollektionen angestrebt.

3. **Procedure Reuse**
   konzentriert sich auf den Bereich der Formalisierung und Kapselung von Software-Entwicklungsprozeduren.

2.4 Konzepte und Technologien

Freeman [8] stellt eine Hierarchie von unterschiedlichen Arten von Information auf, die für die Wiederverwendung benötigt werden.

- **Code Fragmente**
  Die Wiederverwendung von ausführbaren Code-Teilen zählt zu den ursprünglichsten Formen von Reuse und hat als Ziel, die Anzahl der zu entwickelnden Source-Code Zeilen für ein neues Software-System zu reduzieren. Im allgemeinen werden für die Realisierung Software-Bibliotheken verwendet.

- **Logische Strukturen**
  Diese Art von Information besteht sowohl aus den Prozessen wie auch den Datenstrukturen. Logische Strukturen werden durch ihre Teile (Module, Daten) und deren Beziehungen untereinander (Aufrufe, Parameterübergabe) charakterisiert. Diese Teile werden zwar durch ihre Funktionalität und deren Semantik, nicht aber aufgrund einer zugehörigen Implementierung beschrieben. Bei logischen Strukturen handelt es sich meist um das interne Design der Software.

- **Funktionale Architekturen**
  Das externe Design von Software umfaßt jene Aspekte, die von den Benutzern eines Systems gesehen werden. Diese Art von Information beinhaltet eine Beschreibung der Funktionen und Daten, ohne konkrete Realisierungsdetails. Funktionale Architekturen werden in **funktionale Kollektionen** und **generische Systeme** unterteilt.

  Eine **wiederverwendbare funktionale Kollektion** ist eine Menge von Funktionen, die eine Einheit darstellen (*package*), einem speziellen Anwendungsbereich zuordenbar sind, einen ganz bestimmten Teil dieses Anwendungsbereiches abdecken und eigenständig verwendbar sind (z.B. mathematische Funktionspakete).

Generische Systeme werden dahingehend charakterisiert, daß sie aus einer Menge unterstützender Funktionen bestehen, die einer gemeinsamen Funktion zugehören, und ohne Modifikation des Source Codes in vielen unterschiedlichen Situationen einsetzbar sind.

- **Externes Wissen**
  Dabei handelt es sich um Informationen, die aus dem Umfeld außerhalb der technischen Realisierung der Software stammt. Hierbei werden zwei Arten von Information unterschieden:
  Information aus dem **Anwendungsbereich** und
  Information aus dem **Entwicklungsprozeß**.

  Information aus dem **Anwendungsbereich** umfaßt dabei die Information aus der Domäne. Diese Information liegt jedoch in den meisten Fällen nur informal vor, und ist somit schwierig wiederzuverwenden.

  Information aus dem **Entwicklungsprozeß** schließt Life-Cycle Modelle, Beschreibungen von Arbeitsprodukten, Testpläne und Checklisten für die Qualitätssicherung ein. Im allgemeinen wird erst gar nicht versucht, diese Art von Information wiederzuverwenden. Mit der zunehmenden Formalisierung des Reuse Prozesses gewinnt aber auch diese Information an Bedeutung.

- **Informationen aus der Umgebung**
  Informationen aus der Umgebung einer Anwendung stellen die abstrakteste Form von Information in dieser Hierarchie dar. Wissen über die Verwendung besteht aus der Information, wie ein Software-System praktisch eingesetzt wird. Der Unterschied zum Wissen über den Anwendungbereich ist fließend; beide Arten werden gemeinsam verwendet.

Biggerstaff und Richter beschreiben die vorhandenen Reuse Technologien. Die Technologien, die im Reuse Bereich Anwendung finden, können in zwei große Gruppen eingeteilt werden, die von der Natur der wiederzuverwendenden Komponente abhängen. Diese beiden Gruppen sind **Kompositions-Technologien** und **Generator-Technologien**.

In der **Kompositions-Technologie** sind die wiederverwendbaren Komponenten weitestgehend atomar und werden idealerweise ohne Modifikation wiederverwendet. Komponenten müssen jedoch u.U. an die Anforderungen des Reusers angepaßt werden. Dennoch sind diese Komponenten weitgehend passiv, wie z.B. Code-Skeletons, Subroutinen, Funktionen, oder Programme. Neue Programme entstehen aus existierenden Building Blocks, die nach speziellen

Prinzipien der Komposition abgeleitet werden. Der Unix Pipe-Mechanismus ist ein klassisches Beispiel dafür.

Generator-Technologien sind schwerer zu charakterisieren. Die wiederverwendbaren Komponenten sind Patterns, keine konkreten, abgekapselten Entitäten, die in einen Generator gegeben werden. Die resultierenden Strukturen sind nur mehr bedingt mit den ursprünglichen Patterns verwandt.

Kapitel 3

Scheduling

Scheduling wird von Baker [1] definiert als:

> *Zuteilung von Mitteln in der Zeit um eine Sammlung von Aufgaben durchzuführen oder zu lösen*

Unter dem Scheduling von Produktionsprozessen wird eine Tätigkeit verstanden, bei der ausgehend von einer Menge von Aufträgen ein zeitlicher Plan *(schedule)* aufgestellt wird, der festhält, wann, bzw. in welcher Reihenfolge die Aufträge fertiggestellt und die dafür benötigten Operationen durchgeführt werden.

In einem Produktionsprozeß ist ein Auftrag ein zu fertigendes Produkt, an das bestimmte Anforderungen hinsichtlich der Funktion, der Qualität oder des Aussehens gestellt werden. Manchmal werden mehrere gleiche oder ähnliche Aufträge entweder aus produktionstechnischen Gründen oder zur Verringerung der Komplexität beim Planen zusammengefaßt. Zur Erledigung eines Auftrags müssen bestimmte Operationen unter Verwendung von Ressourcen durchgeführt werden.

Eine Ressource kann eine benötigte Maschine, aber auch benötigtes Personal, Raum, Material, Energie, eine Befestigung, ein Transportmittel oder ein bestimmtes Werkzeug sein. Häufig unterscheidet man zwischen erneuerbaren, nur beschränkt zur Verfügung stehender Ressourcen wie z.B. den Maschinen und den nicht erneuerbaren aber dafür prinzipiell unbeschränkt verfügbaren Ressourcen wie benötigtem Rohmaterial oder Energie.

Sind zur Erfüllung eines Auftrags mehrere Operationen notwendig, können sie zu einem Prozeßplan *(prozess)* zusammengefaßt werden. Dieser enthält eine Beschreibung der Ressourcen, die für die einzelnen Operationen benötigt werden und möglicherweise eine zeitliche Reihenfolge der Operationen. Wenn die Art der Operationen erst noch geplant werden muß, wird von einer Prozeßplanung gesprochen. Wenn nur noch geplant wird, welche Maschine für eine Operation des Prozeßplans benutzt werden soll, wird auch vom Routing gesprochen.

Wenn der Prozeßplan als ganzer eingeplant wird, wird von einem Job gesprochen. Ein Job wird in der Produktion als logistische Einheit betrachtet. Ein Job kann manchmal auch eine Zusammenfassung von mehreren gleichartigen Aufträgen sein, die gleichzeitig oder unmittelbar hintereinander durchgeführt werden sollen. In der flexiblen Fertigung wird diese Zusammenfassung häufig Los *(lot)* genannt.

Der Begriff der Maschinenbelegung *(machine allocation)* oder Ressourcenbelegungsplanung *(resource allocation)* wird auch synonym zum Begriff des Scheduling benutzt. Die Generierung von Schichtplänen, Stundenplänen oder Fahrplänen sind auch Anwendungsgebiete des Scheduling.

3.1 Typen von Schedulingproblemen

Wir unterscheiden drei Typen von Schedulingproblemen.

3.1.1 Flow-Shop Scheduling

Sind in einer Anwendung die Operationen aller Prozeßpläne zeitlich genau hintereinander gereiht, und damit deren Reihung nicht mehr Bestandteil der Aufgabe, dann wird von einem Flow-Shop gesprochen. Bei einem Flow-Shop Scheduling ist die Reihenfolge von Operationen gleich für alle Jobs in Schedule. Zum Beispiel alle Jobs werden zuerst durch die Maschine A, dann durch die Maschine B und schließlich durch die Maschine C bearbeitet. Typische Flow-Shops treten z.B. bei der Chargen- und Fließfertigung in der Lebensmittelproduktion, der Verfahrensindustrie und der Metallerzeugungsindustrie auf.

3.1.2 Permutation Scheduling

Bei einem Permutation Scheduling sind mit der Reihenfolge der Jobs gleichzeitig auch die Reihenfolge der n-ten Operation jedes Jobs festgelegt. Die Reihenfolge von Jobs ist gleich bei jeder Maschine. Zum Beispiel Maschinen A, B und C führen zuerst die Operationen von Job 2, dann von Job 1 und Job 3 durch. Chargenprozesse (z.B. Stahlerzeugung) sind meist Flow-Shops, bzw. Permutation Flow-Shops.

3.1.3 Job-Shop Scheduling

Es ist möglich, daß für jeden Job die verschiedenen Operationen auf immer anderen Maschinen eingeplant werden. Bei einem Job-Shop sind neben den notwendigen Operationen und Ressourcen meist Präzedenzgraphen gegeben, die eine Teilordnung vorgeben. Bei Montageprozessen ist das häufig ein baumartiger Graph. Die Unterteilung ergibt sich meist aus den physikalischen Gegebenheiten des Produktionsprozesses. Job 1 wird durch die Operationen auf den Maschinen A, B und C, Job 2 aber durch die Operationen auf den Maschinen C und B definiert. Für einen Stückgutprozeß (z.B. Leiterplattenfertigung) oder Montageprozeß ist meist ein Präzedenzgraph bekannt, so daß diese durch einen Job-Shop modelliert werden.

3.2 Typen von Scheduling

Das Scheduling kann auf verschiedene Weise klassifiziert werden.

3.2.1 Scheduling versus Planung

Das Scheduling von Produktionsprozessen liegt in einem Unternehmen unterhalb der Ebene der Produktionsplanung. Ein sogenanntes Produktionsplanungssystem (PPS) generiert meist eine Auftragsliste aufgrund von geschätzten Kapazitäten und Lieferzeiten, ohne daß eine mögliche Reihung der Aufträge bekannt ist.

So werden bei der Planung aus einer Liste von Aufträgen *(pool)* die ausgewählt, für die ein Schedule erstellt werden soll.

Die Menge der einzuplanenden Aufträge kann aber auch von einem weiterverarbeitenden Betrieb kommen, der als Auftraggeber fungiert.

Unter Berücksichtigung von Auftragspräferenzen und von technischen Einschränkungen in der Produktion muß dann eine Planung der Reihenfolge *(sequencing)* und möglicherweise eine Festlegung von Zeiten für Operationen auf Maschinen *(scheduling)* geschehen.

3.2.2 Dispatching

Die einfachste Variante zur Lösung des Schedulingproblems betrachtet die Belegung der Maschinen und wählt einen neuen Auftrag aus der Auftragsliste aus, sobald eine Maschine frei wird. Man spricht hier meist von Dispatching.

3.2.3 Predictive Scheduling

In vielen Produktionsprozessen sind die Produktionabläufe sehr ähnlich. Das läßt auf die in der Zukunft notwendigen Ressourcen schließen. Das predictive Scheduling verwendet bei der Planung auch das Wissen das sich zu dieser Zeit nur voraussagen lassen kann. Ein vorausschauendes Scheduling ist oft sinnvoll, weil Engpässe *(bottlenecks)* in der Produktion besser berücksichtigt werden.

3.2.4 Preemptive Scheduling

Bei Produktionsprozessen wird häufig angenommen, daß Operationen, die auf einer Maschine begonnen haben, nicht mehr unterbrechbar sind, um eine andere Operation vorzuziehen. Beim Preemptive Scheduling wie z.B. bei Computerprozessen kann eine Operation unterbrochen werden, um eine andere zu bearbeiten.

3.2.5 Reactive Scheduling

In vielen Produktionsprozessen wird die Ausführung der Operationen von der Planung abweichen. Es können Maschinen in der Produktion ausfallen, Produkte fehlerhaft gefertigt werden, dringende Aufträge vorgezogen werden oder eine Operation dauert länger als geplant. Beim reaktiven Scheduling *(reactive scheduling)* wird versucht, diese Unwägbarkeiten zu berücksichtigen, indem eine Rückkopplung zum Produktionsprozeß realisiert wird. Anstatt einer Neuplanung wird dann versucht, den verletzten Plan zu reparieren, um möglichst wenig Änderungen am Plan zu machen.

3.3 Einschränkungen

Die Lösung des Schedulingproblems unterliegt diversen Einschränkungen *(constraints)*. Man unterscheidet meist zwischen harten Einschränkungen *(hard constraints)*, die unbedingt einzuhalten sind, und weichen Einschränkungen *(soft constraints)*. Die Erfüllung der letzteren wird oft graduell bestimmt und dient zur Optimierung der erzeugten Pläne. Die meisten Einschränkungen an Ressourcen *(resource constraints)* sind unbedingt einzuhalten, weil sie physikalischer Natur sind.

So werden bestimmte Arten von Ressourcen für die Durchführung einer Operation gefordert, und eine Bearbeitungszeit *(processing time)* ist vorgegeben. Unter Umständen können allerdings alternative Ressourcen mit unterschiedlichen Bearbeitungszeiten existieren, so daß die Einschränkung sich nicht auf eine ganz spezielle Ressource bezieht. Auch eine Beeinflußung der Bearbeitungszeit ist oft noch möglich. Ressourceneinschränkungen, wie z.B. die Verfügbarkeit von Personal, das normalerweise nur während gegebener Schichten arbeitet, aber in Sonderfällen einmal länger arbeiten kann, sind auch nicht so strikt.

Eine harte Einschränkung sind oft Rüstzeiten *(set-ups)*. Wenn z.B. zwei unterschiedliche Operationen hintereinander auf einer multifunktionalen Maschine mit unterschiedlichen Werkzeugen eingeplant werden, kann es notwendig sein, die Maschine umzurüsten. Da das Umrüsten Zeit und wahrscheinlich auch sonstige Kosten verursacht, kann es günstiger sein, die Operationen nicht hintereinander einzuplanen, sondern eine andere einzuplanen. Manchmal tritt auch der Fall auf, daß zwei Operationen nur in einer Reihenfolge hintereinander ausgeführt werden dürfen und die umgekehrte Reihenfolge verboten ist.

In vielen Produktionsprozessen müssen auch vorbeugende Wartungsarbeiten für Maschinen berücksichtigt werden. Diese können in äquidistanten Abständen nötig oder aber auch abhängig von den auf den Maschinen durchgeführten Operationen sein. Oft kann nur eine Ausfallswahrscheinlichkeit einer Maschine gegeben werden, so daß die Zeit, wann die Wartung durchgeführt werden sollte, nicht eindeutig ist und damit als weiche Einschränkung betrachtet werden kann.

Die vorgeschriebenen Reihenfolgen von Operationen in einem Prozeßplan, die von der Herstellungstechnologie gegeben sind und deswegen oft auch technische Einschränkungen *(technical constraints)* genannt werden, sind harte Einschränkungen. Für nicht zwingend vorgeschriebene Reihenfolgen können natürlich auch Präferenzen angegeben werden.

Neben produktionstechnischen Einschränkungen werden meist noch weitere, von außen vorgegebene, auftragsabhängige Einschränkungen betrachtet. So werden häufig von der Produktionsplanung Fertigstellungstermine *(due dates)* vorgegeben, die aber meist schwächere Einschränkungen darstellen als die Einschränkungen an Ressourcen oder die technischen Einschränkungen.

Oft werden auch Einlastzeiten *(release dates)* vorgegeben, die verhindern sollen, daß mit der Produktion begonnen wird, obwohl benötigte Materialien noch nicht vorhanden sind. Aufträge können auch mit Prioritäten versehen sein, um ihre Erledigung zu beschleunigen.

3.4 Evaluierung

Erzeugte Pläne, die alle harten Einschränkungen erfüllen, werden als zulässig *(feasible schedule)* bezeichnet. Die weichen Einschränkungen können häufig nicht vollständig erfüllt werden. Neben der Suche nach zulässigen Plänen wird beim Scheduling deswegen häufig noch die Optimierung der erzeugten Pläne anhand der weichen Einschränkungen gewünscht. Das Kriterium dafür kann durch quantifizierbare Funktionen vorgegeben werden oder durch eine explizite Darstellung von Präferenzen zwischen Planalternativen. Oberstes Ziel der Produktion ist natürlich ein möglichst großer Gewinn für das Unternehmen. Was das im einzelnen aber bedeutet, ist nicht immer klar. Eine maximale Produktionsmenge erscheint im ersten Augenblick einleuchtend, aber die Genauigkeit mit der Lieferzeiten eingehalten werden, kann für mögliche Folgeaufträge noch wichtiger sein. Die Folgen einer Terminverletzung können aber nicht so einfach quantifiziert werden. Oft existieren verschiedene Optimierungskriterien, die sich teilweise widersprechen.

In der Literatur existieren verschiedene quantitative Optimierungs- oder Evaluierungsfunktionen für Schedulingprobleme, deren Anwendbarkeit natürlich auch von der Art der Anwendung abhängt. Mit Hilfe dieser Kriterien kann einem erzeugten Plan ein Wert zugewiesen werden, und zwei alternative Pläne können verglichen werden. Damit ist es auch theoretisch möglich, bei abzählbar vielen möglichen Schedules den besten zu finden.

3.4.1 Fertigstellungszeit

Die Fertigstellungszeit *(completion time)* eines Jobs ist der Zeitpunkt zu dem die Bearbeitung des Jobs beendet wird. Die maximale Fertigstellungszeit *(maximum completion time)* ist der Zeitpunkt der Fertigstellung des letzten Jobs. Bei der Minimierung der maximalen Fertigstellungszeit wird für eine gegebene Menge von Jobs ein Plan gesucht, bei dem die Zeit für die Durchführung aller notwendigen Operationen minimal ist. Die Zielfunktion, die insbesondere in vielen theoretischen Veröffentlichungen dominiert, basiert auf der Idee, daß bei einer minimalen Gesamtfertigstellungszeit die Produktionsmenge maximal ist. Folge einer solchen Strategie kann es sein, daß relativ große Zwischenlager benötigt werden. Außerdem ist es natürlich nicht sicher, daß eine minimale Gesamtfertigstellungszeit für die betrachtete Menge von Jobs für einen größeren Zeitraum auch wieder optimal ist. Diese Evaluierungsfunktion scheint insbesondere für eine Massenfertigung geeignet. Für einen fortlaufenden Betrieb erscheint eine Minimierung der mittleren Fertigstellungszeit *(mean completion time)* geeigneter, weil hier bei dem optimalen Plan vielleicht einige wenige Jobs mit hoher Fertigstellungszeit noch bearbeitet werden müssen, aber der Großteil der Maschinen bereits wieder frei ist für neue Jobs.

3.4.2 Fließzeit

Die Fließzeit *(flow time)* eines Jobs ist die Zeitspanne zwischen Einlastzeit und Fertigstellungszeit. Wenn alle Einlastzeiten 0 sind, dann ist die Minimierung der maximalen Fließzeit *(maximum flow time)* identisch mit der Minimierung der maximalen Fertigstellungszeit. Bei der Minimierung der mittleren Fließzeit *(mean flow time)* werden Pläne besser bewertet, in denen keine großen Wartezeiten zwischen den einzelnen Operationen eines Jobs auftreten. Diese Evaluierungsfunktion ist jedoch identisch mit der Minimierung der maximalen Fertigstellungszeit.

3.4.3 Auslastung

Oft wird die Evaluierung auch bezüglich der Ressourcen bzw. Maschinen untersucht. Eine Möglichkeit ist die Minimierung der momentanen Arbeit *(work-in-progress* oder *work-in-process)*. Hier soll die mittlere Anzahl der Aufträge, die zu einem Zeitpunkt in Bearbeitung sind, minimiert werden. Der Sinn dabei ist die Minimierung von Zwischenlagern.

3.4.4 Termintreue

Wenn Fertigstellungszeiten berücksichtigt werden müssen, stellt die Minimierung der mittleren Terminverletzung *(mean lateness)* ein wichtiges Maß für die Optimalität eines Plans dar. Dabei kann es sinnvoll sein, größere Terminverletzungen stärker zu bestrafen. Wenn ein Auftrag früher als gefordert fertig wird, erscheint diese Differenz als negativer Wert in der Berechnung auf. Diese Evaluierungsfunktion ist insbesondere für eine auftrags-orientierte Produktion mit kleinen Losgrößen wichtig und zielt auf eine Verbesserung der Kundenakzeptanz. Die Einhaltung dieser Termine kann natürlich auch von großer Bedeutung für Folgebetriebe sein, die ihre Planung auf den Terminzusagen aufgebaut haben. Manchmal wird auch die Minimierung der maximalen Terminverletzung *(maximum lateness)* betrachtet. Wenn Fertigstellungszeiten gefordert sind, macht es oft auch Sinn eine zu frühe Fertigstellung zu vermeiden, da sonst das Produkt lagern muß und unnötiges Kapital im Produkt gebunden ist. Da die Lagerkosten meist unabhängig von dem aktuellen Lagerbestand sind, ist das Hauptkriterium für diese Funktion die Kapitalbindung. Wenn also die Kosten für die benötigten Materialien hoch sind, sollte eine zu frühe Fertigstellung vermieden werden. Während bei der Berechnung der Terminverletzung negative Werte mit in die Evaluation einfließen, tun sie dies bei der Minimierung der mittleren Verspätung *(mean tardiness)* nicht. Manchmal wird auch die maximale Verspätung *(maximum tardiness)* minimiert. Auch die Minimierung der mittleren Verfrühungen *(mean earliness)* oder der maximalen Verfrühungen *(maximum earliness)* kann als eigenständige Evaluierungsfunktion betrachtet werden.

Kapitel 4

Systemdesign

Wie bereits im Kapitel 1 dargestellt wurde, sollte das Modul DomFLIP++ eine Schnittstelle zwischen den Modulen OptiFLIP++ und DynaFLIP++ bilden.

4.1 Einleitung

Zu den Aufgaben von DomFLIP++ zählen:

- Unterstützung des Domainengineers bei der Definition einer neuen Domäne
- Anbindung einer neudefinierten Domäne an das bestehende System
- Erfassung und Verwaltung aller Daten in der Domäne
- Weitergabe der Domänenstruktur für die Evaluierung an das DynaFLIP++
- Unterstützung der Optimierungsalgorithmen

In der Entwurfsphase hat mich die Literatur von Coad und Yourdon [3], Booch [2] und Rumbaugh [23] inspiriert. Bei der Implementierung habe ich von den Stroustrup C++ Bücher [25] [26] gelernt.

Die Objekte des DomFLIP++ Moduls können wir in zwei Gruppen teilen. Die anwendungsunabhängigen Objekte ändern für neu definierte Domänen nur ihre Daten. Die zweite Gruppe von Objekten - die anwendungsabhängigen Objekte müssen für jede neue Domäne neu definiert werden.

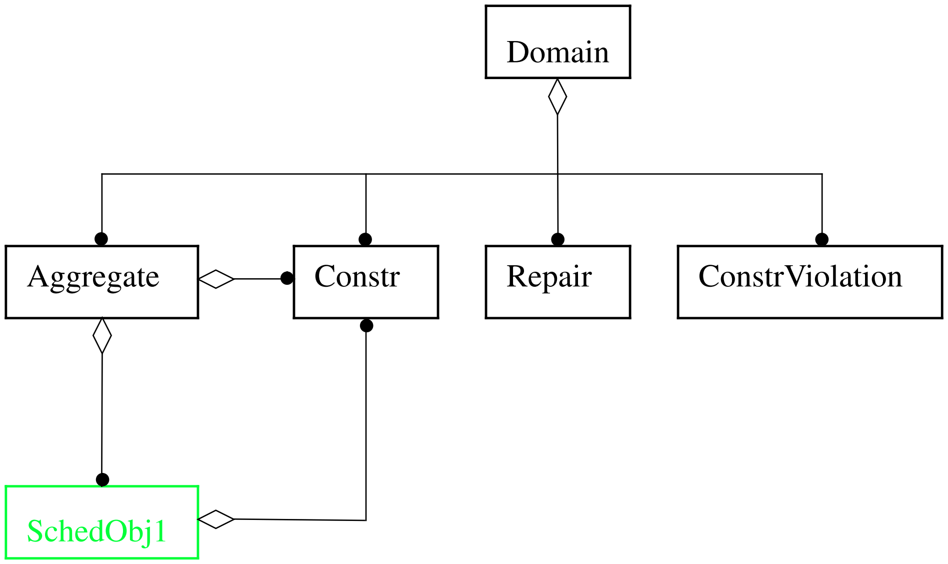

Abbildung 4.1: Objektstruktur von DomFLIP++

Im Bild 4.1 sehen wir die Objektstruktur von DomFLIP++. Das graugezeichnete Objekt *SchedObj1* stellt ein Beispiel für ein Schedulingobjekt dar. Es gehört zu der Gruppe der anwendungsabhängigen Objekte und kann bei der Definition einer neuen Domäne als ein neues Objekt definiert werden. Ein Objekt der Klasse *Aggregate* ist anwendungsunabhängig und verwaltet mehrere Schedulingobjekte. Ein Objekt der Klasse *Domain* verwaltet mehrere Objekte der Klasse *Aggregate*.

Das Modul DomFLIP++ dient als Werkzeug, das eine Domäne definieren und optimieren läßt. Bei der Definition einer neuen Domäne sind folgende Schritte nötig:

1. Wahl des Schedulingobjekts
   Die Reihenfolge von welchen Objekten in unserer Domäne wird bewertet? Läßt sich durch die Bewegungen von Schedulingobjekten eine Optimierung durchführen? Das Schedulingobjekt sollte ein atomares Objekt bei unserer Problemstellung definieren.

2. Definition der Attribute des Schedulingobjekts.
   Man soll hier alle Eigenschaften des Objektes beschreiben. Besonders wichtig ist hier die Definition von einem Schlüsselattribut. Dieses sollte

ein Schedulingobjekt in einer Domäne eindeutig identifizieren. Der Name von diesem Attribut kann beliebig gewählt werden. Der Zugriff auf das Schlüsselattribut erfolgt durch die Methode *key* der Klasse *ListOfEntr*, die dieses Attribut liefern sollte.

3. Definition der Einschränkungen für jedes Schedulingobjekt.
   Für alle bei der Evaluierung gebrauchten Attribute sollten auch Regeln definiert werden, die eine Bewertung ermöglichen. Das Programm unterstützt die Definitionen von fuzzy-sets und speziellen Regeln um diese evaluieren zu können.

4. Definition der Aggregate.
   Jede Domäne besteht aus mehreren Aggregaten, die Schedulingobjekte enthalten.

5. Definition der Einschränkungen für die Aggregate.
   Genauso wie bei den Schedulingobjekten werden auch für Aggregate Regeln für die Evaluierung festgelegt. Die richtige Modelierung der Aggregate hat einen wesentlichen Einfluß auf die Erfolge der Optimierung.

6. Definition der Einschränkungen für die Domäne.
   Auf der Domänenebene werden Regeln definiert, die Zusammenhänge zwischen den Aggregaten definieren.

7. Definition von Reparaturschritten für alle Einschränkungen.
   Bei der Definition von *Domain* wird für jede Einschränkung auch eine Liste von Reparaturschritten definiert, die man durchführen soll, wenn man eine Verletzung dieser Einschränkung minimieren will. Werden spezielle Reparaturschritte gebraucht, so besteht auch die Möglichkeit sie zu definieren.

8. Auswahl der Daten für die Bewertung.
   Nun kann man die Daten der Domäne zum Einlesen vorbereiten. Das Format ist durch die Struktur des Schedulingobjekts eindeutig.

9. Wahl der Optimierungsmethode.
   Für verschiedene Probleme eignen sich bestimmte Optimierungsalgorithmen. Es wird vorgeschlagen mit den Tests jeweils die am besten funktionierende Optimierungsmethode zu finden. Die Einstellung der Parameter für die Optimierungsalgorithmen erfordert eine Erfahrung, die nach einer gewissen Anzahl von Tests vorhanden sein sollte.

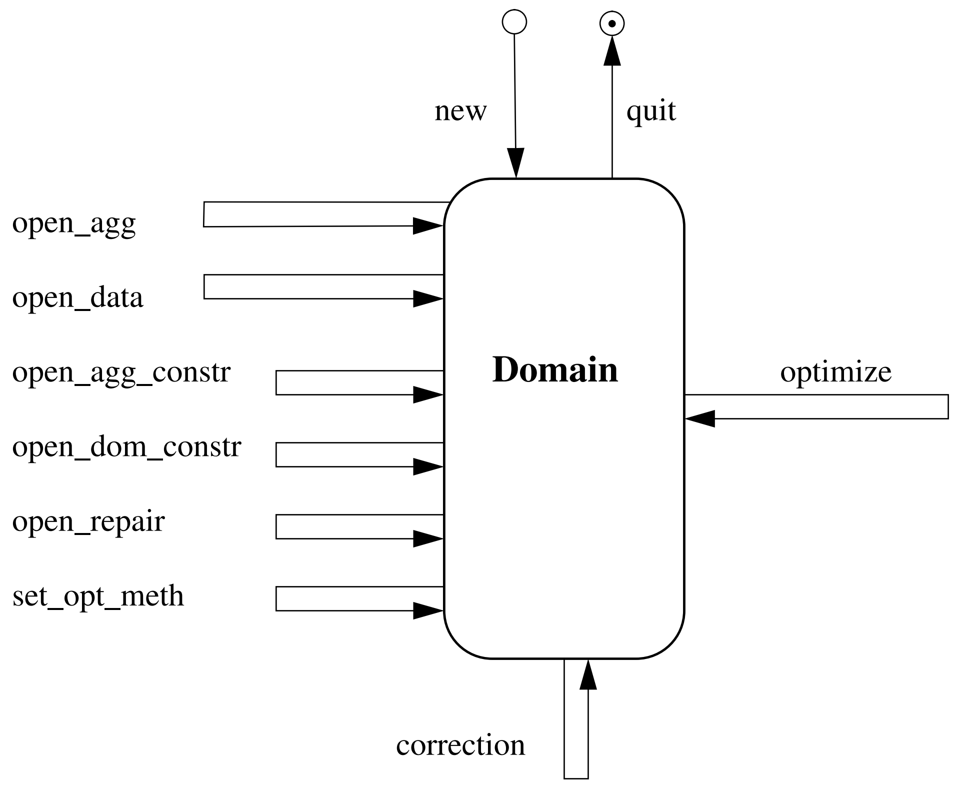

Abbildung 4.2: Funkzionalität von DomFLIP++

10. Abstimmung von Einschränkungen und Reparaturschritten
    Die durchgeführten Tests mit überschaubaren Menge von Daten erlauben eine immer bessere Definition von Einschränkungen und Reparaturschritten. Die Änderungen können während eines Programmlaufes beliebig oft wiederholt werden.

Das Bild 4.2 zeigt den Ablauf der Anwendung von DomFLIP++ in der Abstimmungphase. Neue *Domain* wird mit *new* geöfnet. Die Daten werden mit *open_* eingelesen. Eine Optimierungsmethode wird mittels *set_opt_meth* ausgewählt. Schließlich wird die Optimierung (*optimize*) durchgeführt. Das Verhalten während der Optimierung wird analisiert und die Einschränkungen, Reparaturschritten und Parameter der Optimierungsmethode werden besser abgestimmt (im Bild durch *correction* gekennzeichnet). Hat das Modell die benötigte Reife erreicht, so kann das System endlich für die Optimierung von reallen Daten angewendet werden.

4.2 Schedulingobjektebene

Im *FLIP++ Projekt wird eine allgemeine Listenklasse *Genlist* verwendet. Alle Listenklassen von DomFLIP++ wurden auch von dieser Klasse abgeleitet. Die Klasse *Genlist* wird besonders genau beschrieben, weil ihre Methoden von vielen vererbten Klassen verwendet werden.

4.2.1 Klasse GenList

Die Knoten der Liste *Genlist* wurden als Objekten der Klasse *Node1d* definiert.

Node1d	
item	: void*
pred	: Node1d*
succ	: Node1d*

Abbildung 4.3: Klasse Node1d

GenList ist eine doppelt verkettete Liste. *GenList* hat drei Attribute:

- Einen Zeiger auf den ersten Knoten der Liste - den *root* Knoten. Alle Knoten der Liste sind von Type *Node1d*. Objekte von Type *Node1d* bestehen aus drei Attributen: einen Zeiger auf den nächsten Knoten der Liste (*succ*), einen Zeiger auf den vorherigen Knoten (*pred*), und einen Zeiger vom Typ void* auf das Element der Liste (*item*). Der *root* Knoten hat die Position 0, sein *item* Zeiger hat immer den Wert NULL. Nach der Initialisierung zeigen die *succ* und *pred* Zeiger auf sich selbst. Erst wenn ein Objekt in die Liste eingefügt wird, wird ein neuer Nachfolger vom *root* Knoten erzeugt, der dann mit dem *item* Zeiger auf das eingefügte Objekt zeigt.

- Der Integerwert *no_of_items* gibt an, aus wievielen Elementen die Liste besteht.

- Das Attribut *flag* informiert über den Zustand der Liste. Wurde die Liste gerade initialisiert so ist *flag* gleich 0 (lf_init). Ist *flag* gleich 1 (lf_insert), so darf die Klasse *GenSled* ein neues Objekt in die Liste einschließen. Ist

GenList	
no_of_items	: int
flag	: GenListFlag
root *	: Node1d
del(void*)	: void
key(void*)	: void*
look(void*, void*)	: bool
cmp(void*, void*)	: bool
cpy(void*&, void*)	: void
rd(ifstream&, void*&)	: bool
wr(ofstream&,void*)	: bool
init(void)	: void
insert(Node1d*,void*,InfoCopy): bool	
kick_out(Node1d*)	: bool
lookup(void*)	:Node1d*
add(void*,InfoCopy)	: bool
kick_out(void*)	: bool
get_info(Node1d*)	: void
clear(void)	: void
copy(GenList &)	: void
operator =(GenList &)	: GenList&
operator =(GenSled &)	: GenList&
read(ifstream &)	: bool
write(ofstream &)	: bool
get_how_full(void)	: int
get_flags(void)	: GenListFlag
set_flags(int)	: void
clr_flags(int)	: void
show(void)	: void

Abbildung 4.4: Klasse GenList

flag gleich 2 (lf_kick_out) so darf die Klasse *GenSled* ein Objekt aus der Liste löschen.

Die Klasse *GenList* unterstützt folgende Aktionen:

- **add** - das Einfügen von einem neuen Knoten am Ende der Liste.
- **clear** - das Löschen aller Elemente der Liste.
- **copy** - das Kopieren einer Liste.
- **get_how_full** - die Abfrage der Anzahl der Elemente in der Liste.
- **insert** - das Einfügen von einen neuen Knoten in die Liste an einer bestimmten Stelle.
- **kick_out** - das Löschen von einen bestimmten Knoten aus der Liste.
- **lookup** - das Suchen nach einen bestimmten Knoten in der Liste.
- **read** - das Einlesen einer Liste von einer angegebenen Datei.
- **show** - die Ausgabe der Struktur der Liste an Standardausgabe.
- **write** - das Schreiben der Struktur der Liste in eine angegebene Datei.

Diese Methoden wurden als virtuelle Methoden definiert und werden von den abgeleiteten Klassen überschrieben, wo der Typ des Objektes der Liste feststeht.

4.2.2 Klasse GenSled

Klasse *GenSled* operiert auf einem Objekt der Klasse *GenList*. Sie erleichtert die Verwaltung der Elemente in der Liste. Die im Bild 4.5 dergestellten Methoden sind selbsterklärend und werden daher nicht näher beschrieben. Die Klassen *GenList* und *GenSled* wurden als *friends* deklariert. So können beide auf die privaten Attribute und Methoden der anderen Klasse zugreifen.

GenSled	
ptr	: Node1d*
list_ptr	: GenList*
index	: int
operator++(void)	: void*
operator++(int)	: void*
operator--(void)	: void*
operator--(int)	: void*
operator[](int)	: void*
operator+=(int)	: void*
operator-=(int)	: void*
operator()(void)	: void*
isvalid(void)	: bool
islast(void)	: bool
isfirst(void)	: bool
first(void)	: void*
last(void)	: void*
position(void)	: int
lookup(void*)	: void*
insert(void*,InfoCopy)	: bool
add(void*,InfoCopy)	: bool
kick_out(void)	: bool
get_how_full(void)	: int

Abbildung 4.5: Klasse GenSled

Klasse *Gensled* besteht aus drei Attributen:

- Einem Zeiger *list_ptr* auf die Liste *GenList*.

- Einem Zeiger *ptr* der auf einen beliebigen Knoten in der Liste zeigt.
- Einem Integerwert *index* der die Stelle in der Liste angibt auf die mit dem *ptr* Zeiger gezeigt wird.

Das Bild 4.6 zeigt die Klassen *GenList* und *GenSled* noch einmal.

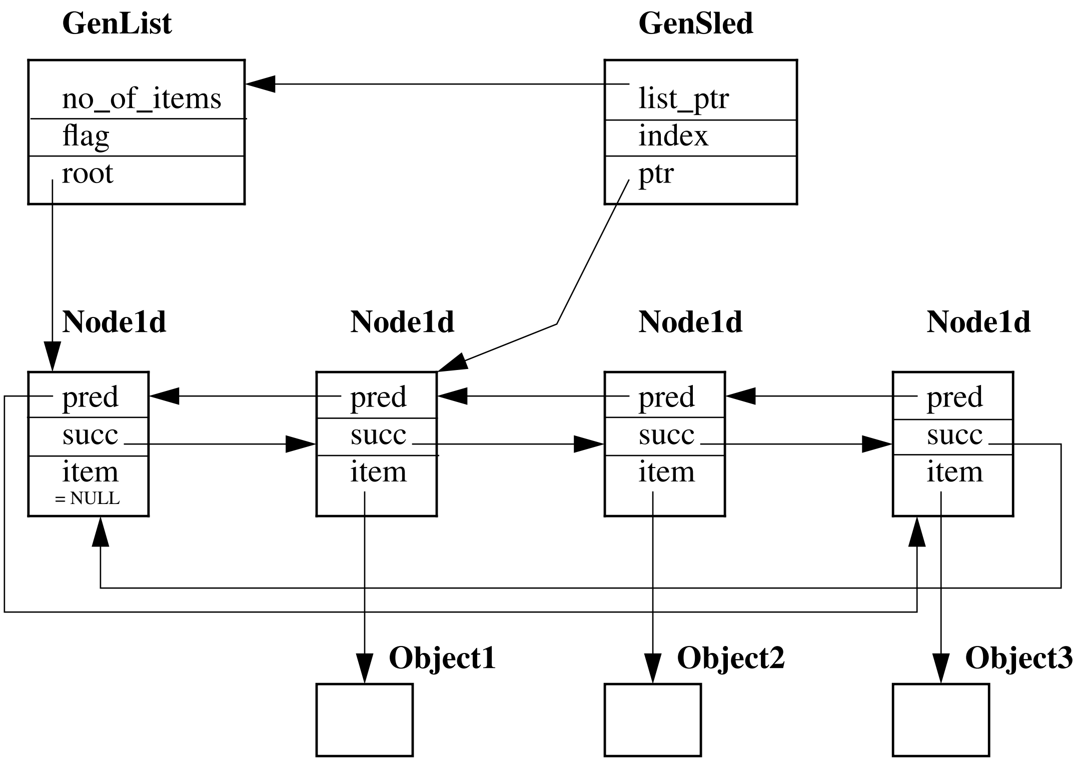

Abbildung 4.6: Klassen GenList und GenSled

4.2.3 Klasse Schedulingobjekt

Diese Klasse wird bei der Definition der Domäne spezifiziert.

Wenn man eine neue Klasse definiert, legt man zuerst fest welche Attribute ein Objekt dieser Klasse haben soll. Zu jedem Attribut werden die Daten für die Evaluierung (Bewertung) in Form einer Liste (*ListOfConstr*) dem Schedulingobjekt zugewiesen.

Das Bild 4.7 zeigt ein Beispiel für eine solche Klasse.

SchedObj1	
att1	: Type1
att2	: Type2
item_constr	: ListOfConstr
init(void)	: void
copy(SchedObj1 &)	: void
write(ofstream &, bool)	: bool
read(ifstream &, bool)	: bool
get_key(void)	: int
get_att1(void)	: Type1
get_att2(void)	: Type2
set_att1(Type1)	: void
set_att2(Type2)	: void

Abbildung 4.7: Klasse Schedulingobjekt

Jede Klasse muß dann außer dem Konstruktor und Destruktor eine Initialisierungsmethode **void init(void)** besitzen, die die Werte der Attribute initialisiert. Es muß auch eine Methode **void copy(SchedObj1 &)** definiert werden, die das ganze Objekt kopiert, weiters eine Methode **bool write(ofstream &, bool)**, die alle Attribute der Klasse (samt ihrer definierten Einschränkungen) auf die vorgegebene Ausgabe schreibt. Analog zu der Methode *write* arbeitet die Methode **bool read(ifstream &, bool)**, die alle Attribute einschließlich der Einschränkungsliste einliest. Nun wird noch eine Methode **int**

get_key(void) benötigt, die das Schlüsselattribut der Klasse umwandelt und als einen Integerwert zurückliefert. Diese Methode wird bei einer der Optimierungsmethoden (dem Tabu Search Algorithmus) verwendet, um die Hash Funktion zu berechnen. Nun braucht man noch **Type get_attribute(void)** und **void set_attribute(Type)** um die Attribute lesen und ändern zu können.

Die Attribute und Methoden dieser Klasse sind genau spezifiziert und werden analog zu den schon existierenden Klassen definiert. Für die Zukunft ist es geplant die Definition der Klasse zu unterstützen. Alles was man dafür braucht ist ein Programm, das die Information über alle Attribute der Klasse (Namen, Type und Initialisierungswerte) sammelt und dann durch einfache Textersetzung die richtige Definition erzeugt. Das Programm arbeitet dann genau so wie das Klassengenerierungswerkzeug genclass von Gnu Compiler, mit dem man auch aus zwei definierten Klassen eine neue definieren kann.

4.2.4 Klasse ListOfConstr

Die Klasse *ListOfConstr* soll für jedes Schedulingobjekt eine Liste von Einschränkungen für dieses Objekt verwalten können. Die Einschränkungen werden mit DynaFLIP++ definiert. Das Modul DynaFLIP++ braucht diese Einschränkungen bei der Evaluierung von jedem Schedulingobjekt.

Auf der Liste von Einschränkungen sind die gleichen Operationen wie für die Klasse *GenList* erlaubt.

4.3 Aggregatebene

Die Klasse *Aggregate* beschreibt die Attribute und Methoden von Objekten, die eine Liste von Schedulingobjekten verwalten. Unter einem solchen Objekt können wir uns eine Maschine vorstellen, auf der verschiedene Aufträge eingeplant werden.

Bei der Evaluierung von einem Aggregat wird die Bewertung von allen Schedulingobjekten in diesem Aggregat durchgeführt. Zusätzlich werden auch die Einschränkungen ausgewertet, die sich auf die Zusammenhänge zwischen den Schedulingobjekten beziehen. Alle Einschränkungen, die auf der Aggregatebene bewertet werden, sind in Form einer Liste gespeichert. Diese Liste ist genauso wie bei dem Schedulingobjekt von Type *ListOfConstr*.

Es werden zwei Typen von Aggregaten unterschieden.

- Ein **pool** wird als Lager für die Schedulingobjekte definiert. Hier werden die Schedulingobjekte gesammelt und so lange aufbewahrt, bis sie eingeplant werden. Die Reihenfolge der Schedulingobjekten hat hier keine Bedeutung.
- Ein **plan** enthält Schedulingobjekte in einer bestimmten Reihenfolge, die für die Bewertung von sehr großen Bedeutung ist. Bei der Optimierung von einem Aggregat dieses Typs wird es mit den Schedulingobjekten aus dem pool-Aggregat gefüllt, und die Reihenfolge der Objekte wird vertauscht. Die Schedulingobjekte werden innerhalb eines Aggregats und zwischen den Aggregaten bewegt.

Der Zugriff auf die Schedulingobjekte des Aggregats erfolgt über eine Instanz der Klasse SledForEntr. Auf diese Weise wird der Zugriff sowohl auf die ganze Liste von Schedulingobjekten (*ListOfEntr*) als auch auf ein bestimmtes Objekt dieser Liste (z.B. *SchedObj1*) möglich.

4.3.1 Klasse ListOfEntr

Die Klasse *ListOfEntr* wird von *GenList* abgeleitet. Sie ist domänenabhängig und muß für ein bestimmtes Schedulingobjekt definiert werden. Die Objekte werden in Form einer Liste verwaltet. Die genau spezifizierten Methoden der Klasse *ListOfEntr* und *SledForEntr* bilden eine Schnittstelle zwischen den domänenunabhängigen und domänenabhängigen Modulen. Ist die Klasse des Schedulingobjekts definiert, so ist es auch möglich die Klassen *ListOfEntr* und *SledForEntr* automatisch zu generieren.

Die Klasse *ListOfEntr* hat die gleichen Methoden wie die Klasse *GenList*.

4.3.2 Klasse SledForEntr

Zu den neuen Mehoden von *SledForEntr* zählen:

- **copy_entr** - das von *SledForEntr* ausgehend die Liste von Schedulingobjekten kopiert
- **get_entr_key** - das das Schlüsselattribut des gefragten Objektes auf Integerwert umwandelt und liefert

4.3.3 Klasse Aggregate

Aggregate	
name	: char[LENGTH_OF_NAMES]
type	: agg_type
agg_evaluation	: float
agg_s_ent	: SledForEntr
agg_constr	: ListOfConstr
init(void)	: void
clear(void)	: void
copy(Aggregate &)	: void
operator =(Aggregate &)	: Aggregate
operator ==(Aggregate &)	: bool
read(ifstream&,bool)	: bool
read_constr(ifstream&,bool)	: bool
read_data(ifstream&,bool)	: bool
get_nr_of_entries(void)	: int
get_entry(int)	: int
get_evaluation(void)	: float
evaluate_plan(void)	: float
test_eval(void)	: float
get_name(void)	: char*
set_name(char *)	: bool
get_type(void)	: agg_type
set_type(int)	: bool
get_agg_constr(void)	: ListOfConstr
get_agg_s_entr(void)	: SledForEntr*

Abbildung 4.8: Klasse Aggregate

Die Klasse *Aggregate* hat folgende Attribute:

- *name* erlaubt, einem Aggregat einen Namen zu vergeben und es später mit dem Namen zu identifizieren. In einer Domäne dürfen gleiche Namen nicht vergeben werden.

- *type* beschreibt den Typ vom Aggregat. Es wird zwischen einen *pool* und einem *plan* unterschieden. Die Optimierungsalgorithmen behandeln die zwei Typen von Aggregaten anders.
- *agg_evaluation* gibt die aktuelle Bewertung des Aggregates an.
- *agg_s_ent* speichert die Information, wo sich alle Schedulingobjekte im Aggregat befinden. Gleichzeitig wird aber vermerkt, welches Schedulingobjekt gerade bearbeitet wird.
- *agg_constr* enthält eine Liste von Aggregat Einschränkungen. Sie beziehen sich auf die Zusammenhänge zwischen den Objekten im Aggregat oder auf den Platz, den sie im Aggregat belegen.

Zu den neuen Methoden von *Aggregate* gehören:

- **get_entry** - liefert das Schlüsselattribut vom angegebenen Schedulingobjekt aus der Liste.
- **get_nr_of_entries** - liefert die Anzahl der Schedulingobjekte von diesem Aggregat.
- **read_constr** - liest Name und Typ von einem Aggregat, und falls diese gefunden werden, setzt sie für das Aggregat die Liste der Einschränkungen gleich der im angegebenen File.
- **read_data** - liest Name und Typ von einem Aggregat und falls diese gefunden werden, setzt sie für das Aggregat die Liste der Schedulingobjekte gleich der im angegebenen File.
- **write_agg** - die wie *write* funktioniert, schreibt aber nur den Namen und Typ des Aggregats.
- **write_constr** - die außer den Namen und Typ auch die Einschränkungen des Aggregats schreibt.
- **write_data** - die außer den Namen und Typ des Aggregats auch alle Objekte mit ihrem Attributen und Einschränkungen schreibt.

4.4 Domänenebene

Die Klasse *Domain* repräsentiert die Information über eine bestimmte Domäne. Die Methoden der Klasse *Domain* erlauben es eine Domäne zu bewerten und die Plazierung der Schedulingobjekte zu optimieren. Ein Objekt dieser Klasse enthält außer einer Liste von Aggregaten auch eine Liste mit Einschränkungen der Domäne, eine Liste, die die wichtigsten Verletzungen der letzten Evaluierung festhält und eine Liste mit Reparaturvorschlägen.

Das Bild 4.9 zeigt die wichtigsten Funktionen auf dieser Ebene.

Die Methoden *open, export* und *clear* verwalten die Daten der Domäne. Sie erlauben entsprechende Teilstrukturen von einer Datei zu laden, sie zu ändern und falls erwünscht sie wieder in einer Datei zu speichern.

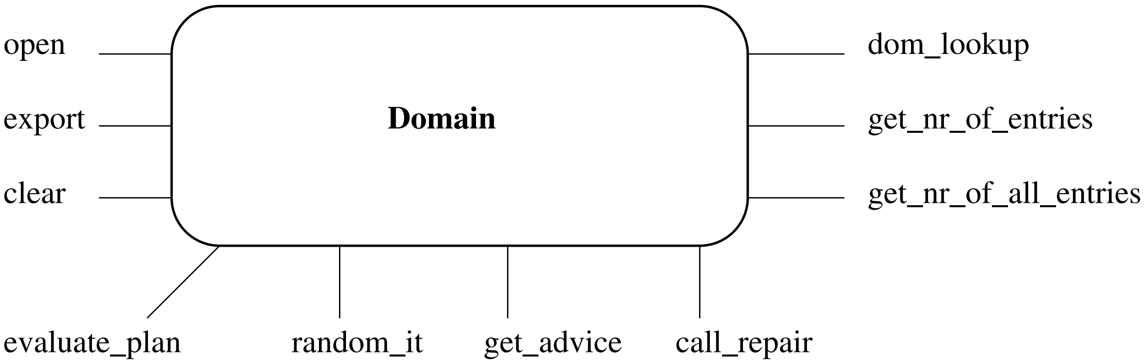

Abbildung 4.9: Modulschnittstellen

Die Methode *evaluate_plan* führt eine Evaluierung der Domäne durch. Diese Aufgabe wird zu dem Modul DynaFLIP++ weitergeleitet. Sie stellt eine Schnittstelle zu dem Modul dar.

Alle anderen auf diesem Bild dargestellten Methoden werden von den Optimierungsalgorithmen verwendet. Sie bauen eine Schnittstelle zu dem Modul OptiFLIP++ auf.

4.4.1 Klasse ListOfAgg

Auf der Domänenebene werden mehrere Aggregate in Form einer Liste verwaltet. Die Klasse *ListOfAgg* wird von *GenList* abgeleitet.

Zusätzlich werden für diese Klasse Methoden zu Verfügung gestellt, die es erlauben, nur einen Teil der Struktur von der Klasse *Aggregate* zu verändern. Zu diesen zählen:

- **clr_cons** und **clr_dat** löschen die entsprechenden Teilstrukturen der Liste.
- **write_agg**, **write_constr** und **write_data** schreiben die entsprechenden Teilstrukturen der Liste.

4.4.2 Klasse SledForAgg

Die Klasse *SledForAgg* hat Zugriff auf alle Schedulingobjekte der Domäne. Diese Klasse führt auch tatsächlich die Bewegungen der Schedulingobjekte durch.

Die Methode **bool call_repair(int,int,ind)** führt einen Reparaturschritt durch. Die Koordinaten sind von Type *ind*. Drei Integerwerte beschreiben, um welches Aggregat es sich handelt, um welches Schedulingobjekt in diesem Aggregat und wieviele Objekte betroffen sind.

Grundsätzlich gibt es zwei Typen von Reparaturschritten:

- **change** Vertauscht eine Gruppe von Schedulingobjekten. Haben wir zum Beispiel ein *Aggregat1* mit Schedulingobjekten *1 2 3 4 5 6* und führen wir ein *change((1,2,1),(1,5,1))* durch, so werden die Schedulingobjekte im Aggregat in der *1 5 3 4 2 6* Reihenfolge geordnet.
- **move** Verschiebt eine Gruppe von Schedulingobjekten. Führen wir auf dem *Aggregat1* ein *move((1,2,1),(1,5,1))* durch, so werden die Schedulingobjekte im Aggregat in der *1 3 4 5 2 6* Reihenfolge geordnet.

Alle anderen Reparaturschritte erzeugt man nun, indem man die Methoden *change* und *move* entsprechend einsetzt. Man kann die Liste von Reparaturschritten beliebig erweitern. Um neue Reparaturschritte zu definieren, sind nur zwei Schritte notwendig.

1. In der Methode *bool SledForAgg::call_repair(int f,int pos,ind i)* den neuen Reparaturschritt implementieren. Diese Methode befindet sich in der Datei *repstepsdef.h*.

2. In der Datei *common.h* wird jedem Reparaturschritt ein eindeutiger Integerwert zugewiesen. Diese Zuweisung muß auch für die neuen Reparaturschritte definiert werden.

Die Methoden **bool change(ind i1,ind i2)** und **bool move(ind i1,ind i2)** liefern einen boolschen Wert zurück, der informiert, ob die Veränderung erfolgreich durchgeführt worden ist. Der Wert *false* wird in folgenden Fällen zurückgegeben:

- Das angegebene Aggregat *i1.agg* oder *i2.agg* existiert nicht .

- Die Anzahl der zu bewegenden Objekte vom ersten Parameter (*i1.num*) ist ungleich der vom zweitem Parameter (*i2.num*).

- Die Position des Objektes ist kleiner als 1.

- Es wird versucht, mehr Objekte zu bewegen als sich im Aggregat befinden. *pla* + *num* kann nicht größer als die Anzahl der Objekte im Aggregat sein.

- Man versucht ein *change* innerhalb eines Aggregats durchzuführen, und die Bereiche überlappen sich.

Wird eine Gruppe mit der Methode **bool move(ind i1,ind i2)** Verschoben, so soll man als Zielplatz den Platz angeben, an dem das erste Objekt aus der Gruppe liegen soll. Wird auf dem *Aggregat1*, der die Schedulingobjekte *1 2 3 4 5 6 7 8* enthält, eine Operation *move((1,1,3),(1,3,3))* durchgeführt, so enthält nach der Ausführung das *Aggregat1* die Schedulingobjekte in folgender Reihenfolge: *4 5 1 2 3 6 7 8* .

So wird auch ein *move((1,1,3),(1,2,3))* erfolgreich durchgeführt und liefert als Ergebnis einen Aggregat mit Schedulingobjekten *4 1 2 3 5 6 7 8* , ein *change((1,1,3),(1,2,3))* hingegen liefert den Wert *false*.

Methode **void show_sched(void)** ist für Testzwecke gedacht und erlaubt die Ausgabe der Reihenfolge von Schedulingobjekten und den Evaluierungswert vom aktuellen Aggregat.

4.4.3 Klasse ConstrViolation

Die Klasse *ConstrViolation* verwaltet die Information über die verletzten Einschränkungen.

ConstrViolation	
type	: constr_type
name	: char[LENGTH_OF_NAMES]
place	: ind
init(void)	: void
clear(void)	: void
copy(ConstrViolation &)	: void
operator=(ConstrViolation&)	: ConstrViolation
write(ofstream &,bool)	: bool
read(ifstream &,bool)	: bool
get_name(void)	: char*
get_type(void)	: int
get_place(void)	: ind

Abbildung 4.10: Klasse ConstrViolation

Zu den Attributen der Klasse gehören:

- *type* beschreibt den Typ der Einschränkung. Es werden drei Typen von Einschränkungen unterschieden - sie ordnen die Einschränkung einer bestimmten Ebene zu. So gibt es *item constraints* - die Einschränkungen auf der Schedulingobjektebene, *agg constraints* - die Einschränkungen auf der Aggregatebene und *dom constraints* - die Einschränkungen auf der Domänenebene.
- *name* identifiziert eine Einschränkung. In einer Domäne dürfen für zwei Einschränkungen nicht gleiche Namen vergeben werden. Der Name der Einschränkung bestimmt auch eindeutig die Regeln für die Evaluierung.

- *place* wird bei der Evaluierung gesetzt und beschreibt, wo das Schedulingobjekt liegt, das zur Verletzung von dieser Einschränkung geführt hat. Dieses Attribut besteht aus drei Integerwerten. *place.agg* bestimmt das Aggregat und *place.pla* die Position des Schedulingobjektes in diesem Aggregat. Der letzte Wert *place.num* beschreibt, wieviele Objekte bei dem Reparaturschritt bewegt werden und bleibt defaultmäßig auf 1 gesetzt.

Zu den Methoden der Klasse *ConstrViolation* gehören die, die für einen Knoten der Liste definiert werden müssen.

4.4.4 Klasse ListOfCviol

Diese Klasse verwaltet eine Liste von verletzten Einschränkungen. Sie wurde aus der Klasse *GenList* abgeleitet. Die Liste enthält Zeiger auf Objekte der Klasse *ConstrViolation*. Sie wird analog zu den schon vorgestellten Listenklassen definiert, und deswegen wird hier auf eine detaillierte Beschreibung verzichtet.

4.4.5 Klasse SledForCviol

Die Klasse *SledForCviol* erweitert die Methoden der Klasse *ListOfCviol*. Sie wird von der *GenSled* Klasse abgeleitet und erbt auch alle Methoden der Basisklasse.

4.4.6 Klasse Repair

Die Klasse *Repair* verwaltet die Information über die Reparaturschritte. Bei der Definition von *Domain* wird für jede Einschränkung auch eine Liste von Reparaturschritten definiert, die man durchführen soll, wenn man eine Verletzung dieser Einschränkung minimieren will.

Repair	
name	:,char[LENGTH_OF_NAMES]
rep_steps	: ListOfInt
init(void)	: void
clear(void)	: void
copy(Repair &)	: void
operator =(Repair &)	: Repair
write(ofstream &,bool)	: bool
read(ifstream &,bool)	: bool
get_name(void)	: char*
get_rep_steps(void)	: ListOfInt*

Abbildung 4.11: Klasse Repair

Zu den Attributen der Klasse gehören:

- *name* ist ein Name einer Einschränkung für die eine Liste von Reparaturschritten definiert wird.
- *rep_steps* ist eine Liste mit Integerwerten. Jedem Reparaturschritt wird ein eindeutiger Integerwert zugewiesen. Diese Werte werden hier in einer Liste gespeichert und repräsentieren jeweils einen Reparaturschritt.

Zu den Methoden der Klasse *Repair* gehören die, die für einen Knoten der Liste definiert werden müssen. Zusätzlich wurden auch Methoden zur Verfügung gestellt, die den Zugriff auf die Attribute der Klasse erlauben.

4.4.7 Klasse ListOfRepairs

Um eine Liste von Reparaturschritten für jede Einschränkung definieren zu können, brauchen wir die Klasse *ListOfRepairs*. Diese Klasse, wie auch alle Listenklassen in diesem Modul, wurde von *GenList* abgeleitet.

4.4.8 Klasse SledForRepairs

Die von *GenSled* vererbte Klasse *SledForRepairs* erleichtert die Verwaltung der Elemente der Liste. Diese Klasse verwendet die von *GenSled* vererbten Methoden.

4.4.9 Klasse Domain

Domain	
name	:char[LENGTH_OF_NAMES]
evaluation	: float
optmet	: optmethod
adv_idx	: ind
rep_step	: int
aggs	: SledForAgg
dom_constr	: ListOfConstr
c_violl	: ListOfCviol
repairl	: ListOfRepairs

Abbildung 4.12: Attribute der Klasse Domain

Zu den Attributen der Klasse *Domain* zählen:

- *name* erlaubt einer Domäne einen Namen zu geben und sie später mit dem Namen zu identifizieren.
- *evaluation* gibt die aktuelle Bewertung der Domäne (d.h. aller Aggregate) an.
- *optmet* gibt die ausgewählte Methode der Optimierung für die Domäne an. Wurde keine Methode ausgewählt, so wird die Tabu Suche angewandt.
- *adv_idx* wird während der Optimierung vor dem Aufruf eines Reparaturschritts verwendet. Dieses Attribut beinhaltet die Koordinaten des Schedulingobjektes (Aggregat, Platz), das für die Verletzung der aktuell bearbeiteten Einschränkung verantwortlich ist.
- *rep_step* empfiehlt, welchen Reparaturschritt man im nächsten Optimierungschritt durchführen soll, um die besten Resultate zu erzielen.
- *aggs* speichert alle Schedulingobjekte die sich in der Domäne befinden. Gleichzeitig wird vermerkt, welches Aggregat gerade bearbeitet wird.

- *dom_constr* enthält eine Liste von Domäneneinschränkungen. Sie beziehen sich auf die Zusammenhänge zwischen den Aggregaten in der Domäne.

- *c_violl* speichert die Ergebnisse der letzten Bewertung. Bei jeder Bewertung entsteht eine Liste mit der Information, welche Einschränkungen wie stark verletzt wurden. Ein Element der Liste informiert über den Typ der Einschränkung, Name und Koordinaten des Schedulingobjektes, das zu dieser Bewertung geführt hat. Die Liste ist so sortiert, daß sich die Einschränkungen mit den größten Verletzungen am Anfang der Liste befinden.

- *repairl* - die Liste mit den Reparaturschrittvorschlägen wird von dem Domainengineer erstellt. Gleich nach der Definition einer Einschränkung sollte man eine Liste von Reparaturschritten festlegen, die bei der Verletzung dieser Einschränkung durchgeführt werden sollen, um diese Verletzung zu minimieren.

Die Methoden der Klasse *Domain* verwalten die Daten der Domäne.

- **bool call_repair(int)** erlaubt, einen Reparaturschritt durchzuführen. Für die Reparatur nötigen Parameter werden von den Attributen *rep_step* und *adv_idx* der Domäne gelesen. Die Reparatur wird auf die Ebene *SledForAgg* ausgeführt.

- Die **clear** Methoden für einzelne Attribute der Domäne initialisieren entsprechende Attribute und erlauben, sie mit einem neuem Inhalt zu füllen.

- **ind dom_lookup(void *)** erlaubt, in der Domäne nach einem Schedulingobjekt zu suchen. Der (void*) Zeiger sollte auf das Schlüsselattribut des Objekts zeigen. Der *ind* Wert gibt die Position des Objektes an. Falls das Objekt nicht gefunden wird, wird die Default-Position (0,0,1) zurückgegeben.

- **float evaluate_plan(void)** evaluiert alle Aggregate der Domäne. Die Evaluierung wird durch DynaFLIP++ durchgeführt. Das Ergebnis der Evaluierung wird in dem Attribut *evaluation* gespeichert. Zusätzlich wird auch eine Liste mit den am meisten verletzten Einschränkungen *c_violl* erzeugt.

- Die **export** Methoden für einzelne Attribute der Domäne schreiben die aktuelle Struktur in eine Datei. Auf diese Weise gesicherte Daten können später wieder mit entsprechenden *open* Methoden eingelesen werden.

- **bool get_advice(int i, int j)** erarbeitet einen Vorschlag für den nächsten Reperaturschritt. Für die i-te Einschränkung aus der Liste mit verletzten Einschränkungen von der letzten Bewertung (*c_violl*) wird der j-te Reperaturvorschlag angenommen. Wurde die Operation erfolgsreich durchgeführt und die Parameter *adv_idx* und *rep_step* neu gesetzt so wird *true* geliefert.

- **int get_nr_of_all_entries(void)** gibt die Anzahl der Schedulingobjekte in der Domäne an.

- **int get_nr_of_entries(void)** gibt die Anzahl der Schedulingobjekte im aktuellen Aggregat an.

- Die **open** Methoden für einzelne Attribute der Domäne füllen die entsprechenden Attribute mit Daten.

- **int pos_aggs(void)** liefert die Position des Aggregates, das aktuell bearbeitet wird.

- **void random_it(void)** wählt eine zufällige Position in einem Aggregat. Diese Methode wird verwendet, um auf zufällige Schedulingobjekte zugreifen zu können.

4.5 Fehlerbehandlung

Für die Zwecke der Fehlerbehandlung wurde die Klasse *DomflipError* definiert. Sie enthält alle Typen von Fehlermeldungen.

Die Methode **void set(error_no, long, char*)** erlaubt es eine bestimmte Fehlermeldung zu aktivieren.

Die Methode **char *get_txt_eval(void)** gibt die zuletzt gesetzte Fehlermeldung aus. Diese Trennung von Fehlerursache und Fehlermeldung ermöglicht eine leichte Änderung der Sprache in der die Meldungen ausgegeben werden.

Es ist anzustreben, daß die Fehlerbehandlungsstrategie bei allen Modulen gleich modeliert wird.

4.6 Wiederverwendung

Die Anforderung entscheidet wie die Softwarewiederverwendung in unserem Fall realisiert werden kann. Im Kapitel 2 wurden die Arten der Softwarewiederverwendung bereits näher beschrieben. Analog zu den im Abschnitt 2.3 dargestellten Blickpunkten werden die Probleme und Erfolge der Softwarewiederverwendung bei dieser konkreten Anwendung vorgestellt.

1. **Produkt**

   - **Source Code Reuse**
     Diese Art von Wiederverwendung wird in diesem Fall unterstützt. Es werden nicht nur die Standard Bibliotheken (wie z.B. stdlib.h, stdio.h, string.h, fstream.h) verwendet, sondern auch die eigendefinierten Klassen (wie z.B GenList - eine allgemeine Listenklasse).
   - **Design** und **Specification Reuse**
     Bei der Definition von einer neuen Domäne wird auch eine Unterstützung angeboten. Der Domainengineer definiert das Objekt aus welchem das Schedule besteht. Eine Klasse, die eine Liste von diesen Objekten definiert, sollte dann automatisch generiert werden.
   - **Object Reuse**
     Es ist wünschenswert, daß die hier definierten Klassen in anderen Modulen auch verwendet werden. Die Objekte werden nicht kategorisiert und auf eine besondere Weise zu Verfügung gestellt. Es handelt sich hier aber um Anwendungen in einem bestimmten Bereich. Dadurch sollte das Finden der entsprechenden Klasse nicht so problematisch sein.

2. **Absicht**

   - **Black-box Reuse**
     wird durch den objektorientierten Lösungsansatz unterstützt. Die einmal definierten Klassen mit ihren Attributen und Methoden stehen für die Wiederverwendung bereit. Da es sich bei der Schedulingsanwendung oft um sehr ähnliche Probleme handelt, finden sich auch die bereits definierten Klassen wieder.

3. **Technik**

   - **Generative Reuse**
     wird hier angestrebt. Die Schnittstele zwischen der domänenspezifischen und domänenunabhängigen Struktur wurde genau spezifiziert. Es können jetzt Programme entstehen, die den Domainengineer bei der Definition von neuen Schedulingobjekten unterstützen, und diese dann an das System automatisch anbinden. Die Domänen- und Aggregatstruktur bleibt vom Schedulingobjekt unabhängig, sie enthält nur andere Daten.

4. **Modus**

   - **Systematic Reuse**
     wäre hier wünschenswert, der *Ad-hoc Reuse* liegt aber der Wirklichkeit näher. Der geplante Reuse erfordert eine Systematik, die nur durch die Anwendung von CASE Werkzeugen möglich wird. Im Rahmen dieses Projekts war es leider nicht möglich.

5. **Bereich**

   - **Vertical Reuse**
     ist für eine bereichspezifische Anwendung typisch. Es wird hier eine Unterstützung für den Bereich "Optimierung von Schedulingproblemen" geboten. Die festgelegten Schnittstellen ermöglichen eine leichte Erweiterung des Programms für eine neudefinierte Domäne. Es ist möglich, diesen Schritt zu automatisieren. Die generischen Komponenten würden dann den Quellcode für neuerstellte Schedulingobjekte selbst generieren.

4.7 Zusammenfassung

Ich befasse mich in dieser Arbeit nicht mit den wichtigsten Problemen der Software Wiederverwendung wie z.B. die Kategorisierung und Suche nach wiederverwendbaren Objekten. Diese Arbeit beschäftigt sich vielmehr mit der Problematik eines erweiterbareren Entwurfes. Die Bereitstellung der generischen Funktionen und Standardisierung der Schnittstellen zu anderen Modulen zählen hier zu den wichtigsten Problemen. Der objektorientierte Ansatz bringt hier die erwünschten Resultate. Gute Dokumentation, gleiche Standards und Konventionen bei der Programmierung und ein optimales Verhältnis zwischen Code und Kommentaren erleichtern das Verständnis. Gute Strukturierung, vollständige, allgemeingehaltene und gut getestete Klassen führen die Wiederverwendung zum Erfolg.

Kapitel 5

Produktionsplanung

Als eine Fallstudie wurde die Produktionsplanung im Stahlwerk LD3 der VOEST Alpine Stahl AG in Linz herangezogen.

5.1 Einleitung

Stahl ist eine Gruppe von Eisenlegierungen, deren Eigenschaften (Härte, Festigkeit, Zähigkeit, chemische Beständigkeit, ...) sich durch Änderungen der Legierungszusammensetzung und durch Wärmebehandlung in weitem Maße variieren läßt. Durch Wärmebehandlung und ein anschließendes rasches Abkühlen läßt sich der Stahl härten. Durch eine kontrollierte Erstarrung lassen sich noch besonders hohe Qualitäten erreichen. Die Härte des Stahls kann auch durch Legierungselemente beeinflußt werden. Außer nach den von den Legierungselementen abhängigen Zusammensetzungen lassen sich Stähle nach dem Reinheitsgrad in Massen-, Qualitäts- und Edelstähle, nach der chemischen Beständigkeit in nichtrostende Stähle (Chrom $>$ 12,5%), säurebeständigem Stahl (Chrom 12 - 19%, Nickel $<$ 8%) usw. und nach den technischen Eigenschaften unterteilen (z.B. selbsthärtender Stahl mit bis zu 20% Wolfram).

Kohlenstoff ist ein Element, das die Eigenschaften des Stahls sehr stark beeinflußt. Das Roheisen, das im Hochofen erzeugt wird, hat noch einen zu hohen Kohlenstoffanteil, der im Stahlerzeugungsprozeß durch das sogenannte Auffrischen verringert wird. Dabei werden die im Roheisen gelösten unerwünschten Begleitelemente wie Kohlenstoff aber auch Mangan, Silizium, Phosphor und Schwefel durch Oxidieren mit Luft oder reinem Sauerstoff in Form von Schlacke oder gasförmigen Verbindungen ganz oder teilweise entfernt. Bei dem sogenannten LD-Verfahren wird das Roheisen im Konverter durch Aufblasen von reinem Sauerstoff (unter Zusatz von schlackenbildenden Substanzen) durch die Sauerstofflanze gefrischt und so gleichzeitig die erforderliche Temperatur erreicht.

Der flüssige Stahl wird dann durch den Abstich aus dem Konverter oder Elelektroofen in eine Pfanne geleert. Diese Pfanne wird als Transportmittel in der Sekundärmetallurgie benutzt. In der Sekundärmetallurgie werden nun weitere Feinlegierungsprozesse durchgeführt. Beim Beruhigen des Stahls wird der gelöste Sauerstoff, der noch durch das Auffrischen im Stahl ist, mittels Aluminium und Silizium gebunden und durch das Bodenspülen mit Argon abgeschieden. Obwohl das Roheisen meist schon entschwefelt wird, müssen manche Stähle in der Sekundärmetallurgie weiter entschwefelt werden. Stähle, die z.B. einen sehr niedrigen Kohlenstoffgehalt ($<$ 0,01%) oder Wasserstoffgehalt ($<$ 3 ppm) haben sollen, müssen einer Vakuumbehandlung unterzogen werden.

Der letzte Prozeß im Stahlwerk ist das Gießen des Stahls in Formen, in denen er erstarren kann. Das alte Verfahren ist das Gießen des Stahls in Blöcke (ingot). Dazu werden die Kokillen (mould) aufgestellt und mittels eines Gießkrans wird der Stahl aus der Pfanne in die Kokillen gegossen. Die Blöcke müssen dann eine gewisse Zeit stehenbleiben bis der Stahl in den Kokillen erstarrt ist. Danach werden die Kokillen entfernt und die erstarrten Blöcke abtransportiert.

Das modernere und meist kostengünstigere Verfahren zum Gießen des Stahls ist das kontinuierliche Gießen mittels einer Stranggußanlage. Die kontinierlichen Stränge werden in Brammen zerschnitten. Es gibt im wesentlichen zwei Arten von Stranggußanlagen: die horizontal und die vertikal gießenden Anlagen. Aus produktionstechnischen Gründen besteht bei den Stranggußanlagen das Problem, daß der Stahl kontinuierlich an die Anlage geliefert werden muß. Anderenfalls muß die Anlage neu gerüstet werden, was einen unerwünschten Produktionsstillstand zur Folge hat.

5.2 Domänenbeschreibung

Im Stahlwerk LD3 der VOEST Alpine Stahl AG in Linz werden hauptsächlich Qualitätsstähle hergestellt. Roheisen wird diekt vom Hochofen geliefert und in Konvertern (hier Tiegel genannt) behandelt. Der gesamte Stahl wird mit zwei Stranggußanlagen gegossen und an das Warmwalzwerk geliefert. Der Dispatcher des Stahlwerks erhält täglich Auftragslisten vom Arbeitsvorbereiter des Warmwalzwerks. Der Dispatcher ist für die Planung, die Überwachung der Ausführung und die Reaktion auf unvorhergesehene Ereignisse zuständig. Hauptkriterium bei der Planung ist die effiziente Nutzung der Stranggußanlagen unter Berücksichtigung diverser Einschränkungen.

5.2.1 Aggregate

Im folgenden werden die Produktionsaggregate im Stahlwerk LD3 mit ihren Funktionen beschrieben. Jeder Auftrag aus dem Auftragspool enthält Informationen (Produktionsvorschriften) darüber, welche Aggregatfunktionen in der Produktion benötigt werden. Die einzelnen Aggregate des Stahlwerks werden in Richtung des Materialflusses aufgezählt. Der Materialtransport zwischen den Aggregaten wird über Pfannen von Kränen bzw. Pfannenwägen durchgeführt.

Vom Hochofen kommt das flüssige Roheisen mit Rohrpfannen bzw. Torpedopfannen entweder direkt (ca. 40%) oder über die Roheisenentschwefelungsstation (ca. 60%) in das Stahlwerk. Dort wird es in Einleerpfannen umgefüllt, abgeschlackt und in einen Tiegel oder in den 2000 t fassenden Roheisenmischer geleert. Der Mischer dient als Zwischenbehälter für Roheisen. Wenn das Roheisen in den Tiegeln gebraucht wird, wird es wiederum in Einleerpfannen geleert und von einem Kran in den jeweiligen Tiegel eingesetzt.

In den Tiegeln wird der sogenannte LD-Prozeß mit Bodenspülen durchgeführt. Schrott wird gemeinsam mit dem gelieferten Roheisen geschmolzen und der Kohlenstoffgehalt wird durch Aufblasen von Sauerstoff auf unter 0,02-0,1% gebracht. Der Kohlenstoff verläßt in Form von CO und CO_2 den Tiegel. Dabei erhöht sich die Temperatur während des Blasens von der Roheisentemperatur von ca. 1300 $^\circ C$ auf die gewünschte Rohstahltemperatur von 1650-1700 $^\circ C$. Um den durch Temperatur und Analyse definierten Endzustand des Stahls im Tiegel mit hoher Genauigkeit zu erreichen, wird über die Sublanze bereits während des Blaseprozesses der Zustand der Schmelze (Temperatur, C-Gehalt, O_2-Aktivität) gemessen. Es gibt aber auch einige Begleitelemente des Eisens, die nicht so leicht entfernt werden können. Schwefel ist in den meisten Stahlsorten sehr unerwünscht. Er wird durch Vorbehandlung des Roheisens, z.B. mit CaC_2, oder durch eine Nachbehandlung des Stahls in der Gießpfanne, z.B. $CaSi$, auf niedrige Werte gebracht. Es existieren die drei Tiegel 7, 8 und 9. Der Produktionsablauf ist so organisiert, daß (wenn nicht durch Störungen verhindert) jeweils zwei Tiegel produzieren. Ein Tiegel faßt bis zu 145 t Stahl. Die Blasezeit beträgt etwa 13 Minuten und die gesamte Tiegelzeit 35 Minuten. Ein Tiegel liefert seinen Rohstahl an den Konditionierstand, der andere Tiegel liefert den Rohstahl an den Pfannenofen. Sie können aber auch an die Vakuumanlage liefern. Der freie Tiegel kann optional für die Gießerei produzieren. Wenn der Tiegel 7 oder 9 in Betrieb ist, können Teilabstichschmelzen eingeplant werden, die direkt in die Gießerei geliefert werden und daher die restlichen Aggregate nicht in Anspruch nehmen.

Hauptaufgabe der Sekundärmetallurgie ist es, flüssigen Stahl mit exakter Analyse und Temperatur zum richtigen Zeitpunkt an die Stranggußanlagen zu übergeben. Der Pfannenofen dient vor allem zum Aufheizen von zu kalten Schmelzen und als Puffer zwischen der schnellen zweisträngigen Stranggußanlage *CC4* und dem langsameren Tiegel. Die Aufheizgeschwindigkeit beträgt 3-5 $^oC/min$. Es besteht auch die Möglichkeit, durch Einblasen von Entschwefelungsmitteln den Schwefelgehalt des Stahls oder durch die Beruhigung den Sauerstoffgehalt zu reduzieren. Auf dem Pfannenofen können zwei Chargen gleichzeitig behandelt werden. Dabei ist eine Charge in der eigentlicher Behandlung und die andere in Warteposition. Alle drei bis vier Minuten wechseln sich die Chargen ab. Der Gesamtaufenthalt einer Charge ist bis zu 30 Minuten. Da elektrisch über Elektroden geheizt wird, ändert sich dabei die Analyse des Stahls nur sehr wenig.

Im Konditionierstand wird der Stahl feinlegiert und für den Gießvorgang aufbereitet. Er verfügt über die Funktionen Konditionieren, Beruhigen und Entschwefeln.

Auf der Vakuumanlage (Synonym RH-Anlage) werden vor allem Stähle mit niedrigem Kohlenstoff- und Wasserstoffgehalt behandelt. Sie kann außerdem für die Funktionen Konditionieren und Beruhigen benutzt werden. Nach vier bis fünf Chargen ist für den Rüssel auf dieser Anlage eine Wartung von etwa zwei Stunden einzuplanen.

Weiters sind nach einer Anzahl von Chargen Rüssel- und Gefäßwechsel vorzunehmen. Die Rüst- bzw. Wechselzeitpunkte werden mit den jeweiligen Vorarbeitern vereinbart. Die Vakuumanlage wird von beiden Produktionswegen benutzt. Der in der Sekundärmetallurgie aufbereitete Stahl wird mittels Gießpfannen zum Gießbereich geliefert. Die Pfannendrehtürme bilden jeweils einen drehbaren Zwischenspeicher für die Pfannen mit gießbarem Stahl vor den Stranggußanlagen. Etwa fünf Minuten vor Gießende einer Charge muß die nachfolgende Pfanne bereits an der Stranggußanlage abgeliefert werden.

Die Stranggußanlagen, sogenannte Kreisbogenanlagen, gießen den aufbereiteten Stahl zu Stahlsträngen. Der Stahl fließt durch einen Schieber im Pfannenboden und durch das Schattenrohr in den Verteiler, der als Puffer dient, um ein kontinuierliches Gießen auch bei Pfannenwechseln zu erlauben.

Vom Verteiler wird der Stahl durch das Gießrohr in eine oszillierende, wassergekühlte Kokille geleitet (Primärkühlung). An den Wänden dieser Kokille erstarrt der Stahl zu einer dünnen Schale. Durch die Oszillation und das Aufgeben von Gießpulver als Schmiermittel wird das Ankleben und Aufreißen der Schale verhindert. Nach Verlassen der Kokille wird die Strangschale durch Rollen gestützt und geführt und durch Aufblasen eines Luft-Wassergemisches weiter abgekühlt (Sekundärkühlung).

Die Kühlintensität richtet sich nach der Stahlmarke und nach der Gießgeschwindigkeit und ist zusammen mit der Wahl des richtigen Gießpulvers entscheidend für die Leistung, für die Brammenqualität und für die Durchbruchsfreiheit der Stranggußanlagen. Der Strang wird beim Durchlaufen der Brennschneidmaschine in Brammen von 5 bis 12 m unterteilt und danach dem Warmwalzwerk übergeben.

Die beiden Stranggußanlagen *CC3* und *CC4* besitzen zwei unterschiedliche Arten von Verteilern. Der Verteiler der *CC4*-Anlage ist ein V-Verteiler um zweisträngig gießen zu können und der der *CC3*-Anlage ist ein E-Verteiler, wobei ein Strang produziert wird.

Die Verteiler müssen regelmäßig gewartet werden. Dafür wird nach einer gewissen Zeit der sogenannte Verteilerwechsel durchgeführt. Während ein Verteiler benutzt wird, wird der nächste vorbereitet.

Beim Gießrohr unterscheidet man nach Gießrohrtyp und Gießrohrhaltbarkeit. Das Gießrohr hat eine eingeschränkte Lebensdauer (geschätzt 240 Minuten), die von der Gießzeit und der Qualität der Charge abhängt. Das Gießrohr ist zur Zeit der bestimmende Faktor für Verteilerwechsel.

Die Kokille ist ein quaderförmiger Trichter durch den der Stahl gegossen wird und die damit das Format des Stahls bestimmt. Kokillen unterscheiden sich in Gießdicke (210 mm, 285 mm) und vier Kokillentypen.

Beide Anlagen erlauben Gießbreiten von 740 bis 1650 mm. Weil der Durchsatz auf der zweisträngigen Anlage höher ist, wird die *CC3*-Anlage vorwiegend zur Produktion kleinerer Breiten 740-1100 mm und kleinerer Aufträge mit weniger Brammen verwendet.

Auf den Stranggußanlagen werden folgende vier Gießgeschwindigkeiten als Sollrichtwerte verwendet: 1,2 m/min, 1,3 m/min, 1,4 m/min, 2,0 m/min. Meist wird eine relativ hohe Geschwindigkeit angegeben. Diese kann immer bis zu einer Mindestgeschwindigkeit verringert werden. Wird eine Pfanne nun nicht rechtzeitig geliefert, kann durch eine Verringerung der Geschwindigkeit eine kurze Verspätung toleriert werden.

5.2.2 Aufträge und Auftragslisten

Der Arbeitsvorbereiter des Walzwerks stellt vormittags das tägliche Walzwerksprogramm aus Gießaufträgen zusammen. Das Programm besitzt eine dreistellige Nummer, die teilweise auf die Art des Programms schließen läßt. Die Reihenfolge der Aufträge, die insbesondere bei Warmdurchsätzen wichtig ist, läßt sich an der eindeutigen Zeilennummer erkennen. Das Walzwerk versucht diese Reihung schon an die Bedingungen des Stahlwerks anzupassen und berücksichtigt, wie lang auf einem Verteiler gegossen werden kann und ob eine Qualitätstrennung vorgenommen werden muß.

Ein Gießauftrag enthält mehrere Aufträge für Brammen. Die Daten eines Gießauftrags beginnen mit der Nummer des Walzwerkprogramms, zu dem er gehört und der Zeilennummer innerhalb des Programms. Für jede Bramme wird angegeben, zu welcher Werksmarke sie gehört. Die Werksmarke ist eine verschlüsselte Beschreibung der Analyse eines Auftrags, wobei entweder Mittelwerte der Mengen der einzelnen Elemente oder Obergrenzen angegeben sind. Für Elemente, für die eine Untergrenze ungleich Null angegeben ist (z. B. Kohlenstoff) nur Mittelwerte, für andere (z.B. Phosphor und Schwefel) wird die Obergrenze angegeben. Allerdings sind nicht alle Elemente in der Werksmarke verschlüsselt.

Für jede Bramme wird die zweistellige Begleitelementnummer angegeben, die in verschlüsselter Form Begleitelemente angibt. Danach wird das Format jeder Bramme angegeben. In der Reihenfolge Länge, Breite, Dicke und Masse werden die Größen in Millimeter bzw. in Kilogramm angegeben. Danach wird die Anzahl der Brammen dieses Formats angegeben.

Da auf der Stranggußanlage *CC4* zwei parallele Stränge beschickt werden, faßt man immer zwei Aufträge zu einer Charge zusammen. Wenn diese Charge dann auf der Stranggußanlage *CC4* gegossen wird, wird je ein Auftrag auf einem Strang gegossen. Wird die Charge auf der Stranggußanlage *CC3* gegossen, dann werden die Aufträge hintereinander gegossen.

Der Stahlwerkspool enthält Aufträge für die nächsten drei Tage. Dieser Pool wird im Walzwerk jeden Tag auf den aktuellen Stand gebracht und dem Stahlwerk per Rechner übermittelt. Zur Zeit enthält der Stahlwerkspool Aufträge über etwa 30.000 Tonnen Stahl. Diese Menge ändert sich aber auch je nach Unternehmensstrategie. Sie soll den Dispatchern einerseits eine Vorausschau auf kommende Walzwerksprogramme geben und andererseits ihnen die Möglichkeit bieten, bestimmte Aufträge vorzuziehen. Solche Änderungen müssen allerdings mit dem Arbeitsvorbereiter des Walzwerks abgesprochen werden. Dem Stahlwerkspool werden auch übriggebliebene Chargen wieder zugeführt.

Der Walzwerkspool ist eine Menge von Aufträgen, die in den nächsten zehn Tagen gegossen werden sollten und etwa 100.000 Tonnen Stahl ausmachen. Auf diese Aufträge haben die Dispatcher des Stahlwerks allerdings keinen direkten Zugriff.

5.2.3 Einschränkungen

Bei der Erstellung eines Gießplans muß der Dispatcher bestimmte Einschränkungen einhalten. Neben Einschränkungen bezüglich der benutzten Aggregate, müssen hauptsächlich Kompatibilitäten verschiedener Stähle berücksichtigt werden. Diese Kompatibilität muß einerseits bei der Zusammenfassung von Aufträgen zu Chargen und andererseits bei der Reihenfolgebildung berücksichtigt werden. Die Kompatibilität bezieht sich auf die chemische Zusammensetzung der Stähle, der benutzten Beruhigung und des Formats, das auf der Stranggußanlage gegossen werden soll.

Aggregate

Die Benutzung der Vakuumanlage ist ein Kriterium, das bei der Planung Berücksichtigung finden muß. Der Punkt ist deshalb wichtig, weil einerseits nach etwa vier bis fünf Chargen auf der Vakuumanlage Wartungsarbeiten von etwa zwei Stunden anfallen und andererseits nicht gleichzeitig für beide Stranggußanlagen Chargen eingeplant werden können, die die Vakuumanlage benötigen.

Eine Charge muß auf der *CC4*-Anlage vergossen werden, falls die Untergrenze des Kohlenstoffanteils größer als 0,75% ist oder die Untergrenze des Kohlenstoffanteils größer als 0,5% ist und der von Mangan größer als 1,7% ist.

Wenn der Aluminiumanteil größer 0,12% oder der Phosphoranteil größer 0,1% ist, muß die Charge auch auf der *CC4*-Anlage gegossen werden.

Rüsten kann nach einer Charge vorgeschrieben sein. Solche Chargen sollen als letzte einer Sequenz eingeplant werden. Stahl, der mehr als 1% Mangan enthält, muß auf den Pfannenofen.

Paarungseinschränkungen

Damit zwei Aufträge zu einer Charge gepaart werden dürfen, müssen die chemischen Analysewerte kompatibel sein. Im Prinzip muß ein Stahl produziert werden, der die Analysewerte von beiden Aufträgen erfüllt.

Die Analysewerte für einen Auftrag werden durch ein Intervall [Untergrenze - Obergrenze] angegeben. Wenn beide Aufträge unlegiert sind, dann sind beide paarbar.

Ein Auftrag wird als legiert bezeichnet, wenn die Untergrenze für Aluminium größer 0,08%, von Phosphor oder Schwefel größer als 0,05% oder für andere Elemente größer als null ist. Wenn für beide Aufträge die Werte gleich sind, können sie natürlich auch gepaart werden. Auch wenn das Intervall für einen Auftrag im anderen vollständig enthalten ist, sind die Aufträge paarbar.

Chemisches Element	Überschneidung	Schwellenwert	Verkürzte Überschneidung
Kohlenstoff (C)	0,01		
Silizium (Si)	0,04		
Mangan (Mn)	0,08	0,6	0,04
Titan (Ti)	0,01	0,03	0,005
Vanadium (V)	0,005		
Molybden (Mo)	0,03		
Nickel (Ni)	0,03		
Chrom (Cr)	0,08	0,6	0,04
Aluminium (Al)	0,01		
Kupfer (Cu)	0,03		
Niob (Nb)	0,005		
Stickstoff (N)	0,001		
Zirkonium (Zr)	0,02		

Abbildung 5.1: Paarungskriterien für Aufträge

Wenn sich die Intervalle gegenseitig jedoch nur teilweise überschneiden, wird angegeben, wie groß diese Überschneidung sein muß. Diese Überschneidung ist abhängig vom chemischen Element, weswegen diese Größen in der Tabelle 5.1 angegeben werden. Dabei gibt es für einige Elemente zwei Angaben. Wenn die Obergrenze beider Werte unterhalb eines Schwellenwert liegt, dann muß die Überschneidung nicht so groß sein. Für Bor und Kadmium gibt es diese Bedingung nicht. Bei beiden müssen sich die Intervalle vollständig überschneiden.

Kompatibilitätseinschränkungen

Die vorgeschriebene Qualität für Aufträge ist grundsätzlich einzuhalten. Es existiert zwar ein gewisser Toleranzbereich, ist aber die Analyse außerhalb der Toleranz, so kann nur mehr versucht werden, diese Qualität für einen anderen existierenden Auftrag zu nutzen.

Wenn zwei Chargen hintereinander auf einer Stranggußanlage gegossen werden, müssen die Chargen kompatibel sein. Dazu müssen beide Chargen aus der gleichen Gruppe von Beruhigungsarten, die chemische Kompatibilität erfüllt sein und gewisse Formateinschränkungen berücksichtigt werden.

Chemisches Element	Überschneidung	Schwellenwert	Verlängerte Überschneidung
Chrom (Cr)	0,0	0,06	0,04
Titan (Ti)	0,0	0,05	0,01
Silizium (Si)	0,0		
Mangan (Mn)	0,0	0,06	0,04
Kadmium (Cd)	0,0		
Bor (B)	0,0		

Abbildung 5.2: Erlaubte Sequenzen auf der Stranggußanlage

Die dreizehn Beruhigungsarten (U, A, B, C, D, E, F, G, H, J, K, L, N) können teilweise zu kompatiblen Gruppen zusammengefaßt werden, Die Arten B, L und N, sowie die Arten C, D, E, F, G, H, J sind untereinander kompatibel.

Für alle nicht in der Tabelle 5.2 angeführten Elemente gelten bei der Reihung von Chargen die gleichen Werte wie bei der Paarung. Bei der Reihung wird bei einigen Elementen eine größere Überschneidung der Analyseintervalle gefordert, wenn die Obergrenze beider Chargen unter einem gewissen Schwellenwert liegt. Eine Überschneidung 0,0 bedeutet, daß die Intervalle mindestens aneinanderstoßen müssen.

Die erlaubte Verstellung der Breite des gegossenen Strangs auf den Stranggußanlagen ist bei jedem Auftrag mitangegeben. Es gibt vier Möglichkeiten, wobei bei Formatverstellungen über 100 mm eine S-Verstellung durchgeführt wird:

- eine Formatverstellung ist in beide Richtungen erlaubt
- eine Verringerung des Formats ist erlaubt
- eine Verbreiterung ist erlaubt
- keine Formatverstellung ist erlaubt

Qualitätstrennungen

Bei kleineren Verletzungen der Kompatibilität reicht eine Qualitätstrennung, bei der eine Stahlplatte zwischen zwei Schmelzen eingefügt wird. Dabei wird eine Schrottbramme produziert, die nicht für die Aufträge verwendet werden kann. Die Größe der Bramme ist abhängig von der Qualität. Typischerweise ist sie etwa 13 Meter lang. Eine Qualitätstrennung kann nicht durchgeführt werden, wenn die Pfannentemperatur zu niedrig ist. Da es meist nicht erwünscht ist, einen Verteilerwechsel allein aus Kompatibilitätsgründen durchzuführen, wird oft versucht eine ansonsten nicht mögliche Reihung mit einer Qualitätstrennung zu erlauben.

Zwei Chargen dürfen hintereinander mit Qualitästrennung und ohne Verteilerwechsel gegossen werden, wenn für Kohlenstoff, Mangan und Silizium die Differenz der Analysemittelwerte kleiner als die Werte in der Tabelle 5.3 ist:

Wenn eine Qualitätstrennung nicht möglich ist, muß ein Verteilerwechsel durchgeführt werden. Diese Methode ist nach momentaner Einschätzung

Chemisches Element	Erlaubte Differenz der Mittelwerte
Kohlenstoff (C)	0,08
Mangan (Mn)	0,40
Silizium (Si	0,20

Abbildung 5.3: Erlaubte Sequenzen mit Qualitätstrennung

teurer als eine Qualitätstrennung. Werden jedoch Verteilerwechsel wegen der ablaufenden Nutzungsdauer und Qualitätsgründen gemacht, so kann ein Verteilerwechsel billiger als eine Qualitätstrennung sein.

Rüsten der Stranggußanlage ist die aufwendigste Methode um Qualitätsrestriktionen einzuhalten, da ein längerer Produktionsstillstand entsteht. Diese Methode kann beispielsweise bei extrem hochkohligen Chargen notwendig sein, und zwar sowohl vor der Charge, als auch nach der Charge (da der Gießstrang wegen der Dauer des Verteilerwechsels erstarren würde).

5.2.4 Optimierungskriterien

Für das Stahlwerk sind lokal einige Optimierungen möglich. Diese beziehen sich vor allem auf die Maximierung des Durchsatzes bei Minimierung des Einsatzes von Ressourcen.

Nach einer gewissen Anzahl von gegossenen Chargen müssen Wartungsarbeiten an den Verteilern vorgenommen werden. Deswegen wird ein neuer Verteiler genommen und der sogenannte Verteilerwechsel durchgeführt. Die Verteilerlebensdauer beträgt im Moment schätzungsweise 240 Minuten. Die Menge der Chargen, die zwischen zwei Verteilerwechseln gegossen werden kann, heißt im Stahlwerksjargon "Verteiler". Die Anzahl der Chargen hängt von der Gießzeit und der Qualität der Chargen ab. Im Moment liegt der Engpaß bei der Lebensdauer des Gießrohrs.

Wenn der Verteilerwechsel immer rechtzeitig durchgeführt wird und keine Kompatibilitätsprobleme auftreten, wird ein Reißen des Strangs und damit ein Rüsten der Stranggußanlage vermieden. Die Mindestgießzeit, die benötigt wird, um den nächsten Verteiler vorzubereiten und dadurch Rüsten zu vermeiden beträgt für die *CC3*-Anlage 80 Minuten und für die *CC4*-Anlage 100 Minuten. Wenn ein Rüstvorgang notwendig ist, werden jedoch direkt zwei Verteiler vorbereitet, sodaß die Gießdauer des ersten Verteilers auch kürzer sein kann.

Bei der zweisträngigen Stranggußanlage sollten beide Stränge etwa gleichzeitig fertigwerden, da sonst bei dem Strang, der früher fertig wird, der Strang reißt. Es ist deshalb zu versuchen, daß die beiden Aufträge, die zu einer Charge zusammengefaßt werden, etwa gleich lange gegossen werden. Dies kann über Gewicht, Format und die Gießgeschwindigkeit der beiden Aufträge oder über die Länge aller Brammen und der Gießgeschwindigkeit erkannt werden.

Zu große Differenzen der beiden Aufträge können durch Geschwindigkeitsanpassungen der einzelnen Stränge abgefangen werden, indem der Strang, der eigentlich nicht so lange benötigt, langsamer gegossen wird. Da dies der Qualität keinen Abbruch tut, kann dies meist angewendet werden. Andererseits wird dadurch aber der Durchsatz der Stranggußanlage verringert. Deswegen versuchen die Dispatcher durch abwechselndes Beschicken der beiden Stränge die Differenz zwischen beiden Strängen zu verringern. Wenn beispielsweise der eine Auftrag aus drei kleinen, schnell zu gießenden Brammen besteht, und der andere aus drei langen, langsam zu gießenden Brammen besteht, dann können, vorausgesetzt die Beschränkungen für die Gießformate werden nicht verletzt, die beiden mittleren Brammen ausgetauscht werden. Nichtsdestotrotz ist es besser, wenn von Anfang an die Aufträge, die zu einer Charge zusammengefaßt werden, gleich lange Gießzeiten erfordern.

5.2.5 Planungsstrategien

Bei der Planung des Programms wird die Grobplanung des Arbeitsvorbereiters aus dem Walzwerk in eine Feinplanung umgearbeitet. Es wird dabei die genaue Reihenfolge der einzelnen Aufträge, Chargen und Verteiler pro Stranggußanlage festgelegt. Der Arbeitsvorbereiter des Walzwerks versucht zwar schon eine Reihenfolge einzuplanen, aber da er über viele Details des aktuellen Zustands im Stahlwerk kein Wissen besitzt, muß die Reihenfolge vom Dispatcher oft noch umgeplant werden.

Im Allgemeinfall faßt der Dispatcher zuerst einzelne Aufträge zu Chargen zusammen. Dann teilt er die einzelnen Chargen Gruppen zu, die im weiteren Verteiler genannt werden, und ohne Verteilerwechsel gegossen werden können. Er ordnet diese Verteiler den Stranggußanlagen zu und reiht dann die Verteiler auf den Stranggußanlagen. Außerdem kann es dann noch sein, daß er Versuche in diese Reihung einbindet. Versuche sind keine zusätzlichen Chargen, sondern die Versuche werden auf eingeplanten Chargen ausgeführt.

Da die Komplexität der Reihung sehr groß ist, ist es weiters notwendig, daß die Dispatcher Strategien anwenden. Diese werden benutzt, um einerseits überhaupt eine gültige Lösung zu finden und andererseits die Ressourcen im Stahlwerk möglichst optimal einzusetzen. Der Dispatcher kann vorsichtig oder riskant bezüglich der vorgegebene Zeiten, Anzahl der Chargen pro Verteiler, als auch auf hintereinander zu gießende Qualitäten planen.

Um bei etwaigen Fehlschmelzen (entweder Qualität oder Format falsch) noch die Möglichkeit eines Umplanens zu haben, werden Chargen mit höherer Qualität und breiterem Format möglichst früh eingeplant.

Warmdurchsatz

Das Warmwalzwerk, das als Auftraggeber für das Stahlwerk fungiert, hat eigene Kriterien für die Reihung von Aufträgen. Am liebsten wäre es dem Arbeitsvorbereiter des Warmwalzwerks, wenn das Stahlwerk genau diese Reihenfolge produzieren würde. Da dies aber unmöglich ist, wird als eine Art Kompromiß versucht, eine Teilmenge der Aufträge in der gleichen Reihenfolge im Stahlwerk zu gießen, wie sie im Walzwerk weiterverarbeitet werden. Diese Teilmenge wird als Warmdurchsatz bezeichnet. Diese Aufträge werden dem Stahlwerk genau in der Reihenfolge mitgeteilt, wie sie gewalzt werden, und das Stahlwerk sollte in dieser Reihenfolge die Schmelzen auch gießen.

Beim Normaldurchsatz können im Vergleich zum Warmdurchsatz die Brammen vor der Weiterverarbeitung auskühlen und damit die Reihenfolge der Weiterverarbeitung geändert werden. Es gibt außerdem Aufträge, bei denen die Brammen abkühlen müssen.

Beim Warmdurchsatz werden die heißen Brammen sofort dem Warmwalzwerk geliefert und nur möglichst kurz in den Warmhalteboxen zwischengelagert. Damit die Brammen nicht zu stark auskühlen und das Walzwerk sich auf den Warmdurchsatz vorbereiten kann, sollte der Warmdurchsatz innerhalb von zwei Stunden vom vorgesehenen Zeitpunkt fertiggestellt werden. Dieser Termin ist weniger wichtig wie die korrekte Reihenfolge.

Gießformate auf den Stranggußanlagen

Auf der Stranggußanlage *CC4*, deren Kapazität durch die zwei Stränge größer ist als die der Stranggußanlage *CC3*, wird versucht, einen möglichst hohen Durchsatz zu erreichen. Das wird unter anderem dadurch erreicht, daß kleinere Formate im Bereich von 740-1100 mm auf der *CC3*-Anlage eingeplant werden. Formate über dieser Breite werden normalerweise auf der *CC4*-Anlage vergossen.

Roheisenstand

Der momentane, sowie der zu erwartende Roheisenstand sind ebenfalls wichtige Punkte, die zu beachten sind. Er beeinflußt die Planung indirekt, indem er bestimmt, welche Chargen überhaupt gegossen werden können. Ist wenig Roheisen vorhanden, so wird der Dispatcher versuchen, möglichst kleine Formate mit langsamen Gießzeiten einzuplanen, oder Sonderchargen, die nur einzeln oder als Zwilling produziert werden, einzufügen. Ein Rüsten der Stranggußanlage ist in solch einer Situation kein großes Problem, da ja sowieso nicht viel produziert werden kann. Ist hingegen genügend Roheisen vorhanden, so wird man bemüht sein, dieses auch zu verwerten und daher großformatige Brammen produzieren. Beim Roheisen wird zwischen entschwefeltem und nicht-entschwefeltem Roheisen unterschieden. Normalerweise dürfte vom Hochofen nur entschwefeltes Roheisen kommen. Durch verschiedene Einflüsse, die für die Planung aber uninteressant sind, kann es vorkommen, daß das Roheisen nicht entschwefelt ist. Diese Tatsache kann aber bei manchen Qualitäten sehr wohl die Planung beeinflussen, da dann z. B. die Vakuumanlage zum Herabsetzen des Schwefelgehalts eingesetzt werden muß.

5.2.6 Überwachung und reaktive Planung

Die meiste Zeit verbringt der Dispatcher nicht mit der beschriebenen Neuplanung, sondern mit der Überwachung der Ausführung seiner Pläne und mit Reaktionen auf Abweichungen zu seinen Plänen. Abweichungen entstehen dann, wenn

- Aggregate ausfallen (Störungen)
- Fehlschmelzen entstehen
- sonstige Ereignisse

Während des normalen Produktionsprozesses wird laufend Kontakt mit dem Wassersteuermann, der Stranggußanlage und dem Schmelzbetrieb (Tiegel) gehalten. Die auszutauschenden Informationen werden dabei hauptsächlich per Telefon oder mündlich weitergegeben. Ist aufgrund einer Abweichung eine Programmänderung notwendig, so bekommen wieder die angeführten Stellen und natürlich der Arbeitsvorbereiter des Walzwerks das neue Programm.

Eine Änderung des Schedules sollte nur bei solchen Abweichungen durchgeführt werden, die eine große Verletzung einer Einschränkung verursachen. Wenn eine Charge länger als geplant braucht, ist das normalerweise kein Grund neuzuplanen. Eventuell sollten dem Walzwerk die neu prognostizierten Lieferzeiten mitgeteilt werden. Wenn aber eine Charge nicht geliefert werden kann, weil ihre Analyse verfehlt ist, dann wird eine Neuplanung notwendig, weil der nicht erfüllte Auftrag neu eingeplant werden muß. Ziel einer Neuplanung ist es dann, unter Berücksichtigung der Produktionseinschränkungen des Walzwerks und des Stahlwerks den Auftrag einzuplanen und den existieren Schedule so wenig wie möglich zu verändern, da jede Änderung Umplanungen in anderen Teilgebieten verursacht.

Störungen

Tritt eine Situation ein, in der ein Abstellen der Stranggußanlage notwenig wird, so spricht man von einer Störung. Demzufolge ist ab diesem Zeitpunkt auch die Produktion unterbrochen. Betroffen von dieser Unterbrechung sind Warmwalzwerk, Hochofen, Tiegel und Stranggußanlage. Abhängig davon, wieviel Roheisen zur Verfügung steht und welche Anlage ausfällt, wird eine Umplanung vorgenommen. Insbesondere, wenn durch den Ausfall Aufträge eines Warmdurchsatzes betroffen sind, wird man versuchen, diese Aufträge auf der anderen Stranggußanlage zu produzieren.

Die Stranggußanlagen verfügen über eine Reihe von Produktionsparametern, die während des Gießvorgangs unabhängig voneinander geändert werden können und unter Umständen eine Störung vermeiden helfen. Während des Gießvorgangs veränderbar sind die Gießbreite und die Gießgeschwindigkeit. Außerdem ist eine S-förmige Formatverstellung und ein Gießpulverwechsel möglich.

Jede dieser Aktionen ist mit unterschiedlichem Verlust und unterschiedlichen Vorschriften verbunden, sodaß sie sehr sorgfältig zu planen sind. Z. B. werden Chargen aufeinander gegossen, die einer Breitengrundrichtung gehorchen (d. h. deren Format in möglichst kleinen Schritten auf- bzw. absteigend sortiert ist), um den Verlust durch große Formatverstellungen zu minimieren.

Fehlschmelzen

Fehlschmelzen sind Schmelzen, deren vorgegebene Analyse nicht erreicht wurde oder deren Format nicht dem Vorgegebenen entspricht. Sie werden im Fehlschmelzenbuch mit dem Grund der Fehlschmelze registriert und eine neue Charge muß unter Umständen eingeplant werden.

Kann der Auftrag auf einen anderen Auftrag umgelegt werden, so spricht man vom Umleger. Dabei ist es unerheblich, ob der Auftrag aus dem vorhandenen Pool des Dispatchers oder nach Rücksprache mit dem Arbeitsvorbereiter aus dem Pool des Arbeitsvorbereiters kommt. Umleger werden im Gegensatz zu Anhängern nicht im Fehlschmelzenbuch eingetragen. Kann die Schmelze weder auf einen Auftrag aus dem Stahlwerkspool des Dispatchers, noch auf einen Auftrag aus dem Pool des Arbeitsvorbereiters (Walzwerkspool) umgelegt werden, werden die Brammen dieser Schmelze auf Lager gelegt. Kann nur weniger als die Hälfte der Brammen für einen Auftrag produziert werden, so ist dieser Auftrag nicht erfüllt. Es muß daher noch eine Schmelze angehängt werden. Ein Auftrag kann z.B. nicht erfüllt werden, wenn der Tiegelinhalt nicht der gewünschten Menge entspricht.

Sonstige Ereignisse

Sonstige Ereignisse können auch zu einer Programmänderung führen, werden aber nicht als Störung bezeichnet werden. Wenn zu wenig Roheisen verfügbar ist, kann es sinnvoll sein, schwierige Aufträge, die unter Umständen ein Rüsten der Stanggußanlage erforderlich machen, vorzuziehen. Da das starken Einfluß auf die Arbeitsreihenfolge im Walzwerk hat, muß hier natürlich eine Abstimmung mit dem Walzwerk erfolgen.

Das Walzwerk kann auch versuchen einzelne wichtige Aufträge vorzuziehen. Das Stahlwerk versucht so weit wie möglich solche Wünsche zu erfüllen. Im einfachsten Fall kann man diesen Auftrag mit einem anderen vertauschen, der nicht so dringend ist. Im Extremfall kann das aber bedeuten, daß eine neue Planung mit anderen Verteilerwechseln aus diesem Auftrag folgt.

Neben der Stanggußanlage können auch andere Produktionsaggregate ausfallen. Sie werden hier nicht als Störung interpretiert, weil die Auswirkung oft einfacher abgefangen werden kann und auch seltener auftritt. Wenn ein Tiegel ausfällt, kann der dritte Tiegel benutzt werden. Wenn ein Aggregat der Sekundärmetallurgie ausfällt, muß neugeplant werden, da nicht alle Aufträge wie gewünscht produziert werden können, aber unter Uständen kann dann trotzdem auf beiden Stranggußanlagen weitergegossen werden.

Bei solchen Ereignissen und Ausnahmesituationen wird der Arbeitsvorbereiter rückgerufen, da dessen Pool an Aufträgen größer ist, und er somit leichter einen Auftrag ausfindig machen kann, der in der gegebenen Situation am besten ist und trotz Ausfall noch produziert werden kann. Bei Programmänderungen sind natürlich genauso die Vorschriften und Regeln zu berücksichtigen.

5.3 Entwurf

Dieses Job-Shop Schedulingsproblem wird als predictive, nonpreemtive und reactive Scheduling modelliert. Will man eine neue Problemdomäne definieren, so müßen vorher Antworten auf paar wichtige Fragen gefunden werden.

5.3.1 Schedulingobjektebene

Die erste Frage, mit der wir uns befassen sollten, lautet: Von welchem Objekt soll eine Ordnung hergestellt werden, die dann bewertet und optimiert wird?

Das Stahlwerk erhält täglich mehrere Aufträge zur Produktion von bestimmten Arten und Formaten von Stahl. Diese Aufträge werden dann auf speziellen Aggregaten eingeplant mit dem Ziel maximalen Durchsatz bei der Stahlproduktion zu erzielen, und dabei die qualitativen Anforderungen nicht verletzen.

In diesem Fall ist die Suche nach dem geeignetem Objekt einfach. Wir definieren einen Auftrag (*Order*) als ein Schedulingobjekt. Neue Aufträge warten in einem Auftragspool auf die Einplanung. Wird ein Auftrag für die Produktion eingeplant, so wird er aus dem Pool in das entsprechende Aggregat bewegt.

Die nächste Frage zwingt uns ein Schlüsselattribut für ein Objekt der Klasse *Order* festzulegen. Dieses Attribut soll ein Objekt der Klasse *Order* eindeutig identifizieren.

Im Stahlwerk wird jeder Auftrag mit einem eindeutigen Nummer versehen. Diese Nummer eignet sich perfekt für ein Schlüsselattribut. In einer Domäne dürfen für zwei Aufträge nicht gleiche Schlüßelattribute vergeben werden.

Die dritte Frage lautet: Welche Attribute beschreiben das Schedulingsobjekt?

Zu diesen zählen in unserem Fall:

- Anlage (*plant*)
  gibt an, ob eine bestimmte Anlage bei der Erzeugung der Brammen ausgewählt werden muß.
- Werkmarke (*plantmark*)
  eine verschlüsselte Beschreibung der Analyse eines Auftrags.

Order	
onumber	: long
plant	: char*
plantmark	: char*
ko_throughput	: char
weight	: long
format	: int
thickness	: int
number_slabgroup	: int
slabgroup	: slab[accomp_max]
accomp_elem	: extreme[element_max]
speed	: extreme
change_format	: char
desulphurization	: char*
kill_meth	: char
delivery_date	: char*
item_constr	: ListOfConstr

Abbildung 5.4: Attribute der Klasse Order

- Durchsatzart (*ko_throughput*)
  bestimmt, in welcher Form die Bramme an den Kunden geliefert werden soll (Warmdurchsatz oder Normaldurchsatz).
- Gewicht (*weight*)
  gibt das Gesamtgewicht aller Brammen an.
- Format (*format*)
  gibt das Format der breitesten Bramme an.
- Dicke (*thickness*)
  bestimmt die Dicke für alle Brammen eines Auftrags.
- Anzahl der Brammengruppen (*number_slabgroup*)
  gibt die Anzahl der Brammengruppen aus der ein Auftrag besteht an.
- Brammengruppe (*slabgroup*)
  beschreibt eine Brammengruppe.

- Begleitelement (*accomp_elem*)
  gibt eine ausführliche Beschreibung der chemischen Zusammensetzung. In 17 Feldern werden jeweils die Obergrenze und Untergrenze für ein bestimmtes chemisches Element angegeben. (Vergleiche Abbildung 5.1)

- Gießgeschwindigkeit (*speed*)
  gibt die maximale und minimale Gießgeschwindigkeit in cm/min an, sodaß sich ein Durchsatz von 3,8 t $Stahl/min$ ergibt.

- Formatverstellung (*change_format*)
  gibt die erlaubte Formatverstellung an.

- Entschwefelung (*desulphurization*)
  gibt die Notwendigkeit der Entschwefelung an.

- Beruhigungsart (*kill_meth*)
  legt fest, ob der Stahl beruhigt wird, und wenn ja welche chemischen Elemente (Aluminium, Silizium) zur Beruhigung des Stahls zugegeben werden.

- Termin (*delivery_date*)
  gibt an, wann der Auftrag die Produktion verlassen sollte.

Welche Methoden die Klasse *Order* haben soll wurde im Kapitel 4.2.3 am Beispiel der Klasse *SchedObj1* bereits erklärt. Eine genauere Betrachtung ist nur für die Methode *get_key* erforderlich. Das Schlüsselattribut der Klasse - *onumber* hat den Typ *long*, die Methode *get_key* soll aber einen Integerwert liefern, der für die berechnung der Hashfunktion dienen soll. Die entsprechende Umwandlung muß hier durchgeführt werden.

5.3.2 Aggregatebene

Haben wir die Attribute des Schedulingobjekts festgelegt, so können wir auch die Klassen *ListOfEntr* und *SledForEntr* definieren.

Die Klasse *ListOfEntr* definiert eine Liste von Schedulingobjekten. Sie verbindet eine Liste von domänenspezifischen Schedulingobjekten mit einer domänenunabhängigen Struktur.

Alle Methoden dieser Klasse wurden bereits im Kapitel 4.3.1 vorgestellt. Genauso wurde auch die Klasse *SledForEntr* bereits im Kapitel 4.3.2 beschrieben.

ListOfEntr	
del(void*)	: void
key(void*)	: void*
look(void*, void*)	: bool
cpy(void*\&, void*)	: void
rd(ifstream &, void *&)	: bool
wr(ofstream &, void*)	: bool
operator=(ListOfEntr &)	: ListOfEntr&
operator=(SledForEntr&)	: ListOfEntr&
add(Order*, InfoCopy)	: bool
kick_out(char*)	: bool
lookup(char*)	: Order*

Abbildung 5.5: Klasse ListOfEntr

Nach der Definition von diesen drei Klassen kann man sie in das System einbinden und zur Modellierung der Domäne übergehen.

Als erstes kann man die Einschränkungen des Schedulingobjektes mit der Unterstützung von DynaFLIP++ definieren. Mit den Einschränkungen werden auch die Regeln für die Evaluierung definiert.

Jetzt kann man alle Aggregate, in denen die Schedulingobjekte gesammelt werden (die von Typ pool), definieren. In unserem Fall definieren wir nur einen Pool *order_pool*, der die Aufträge der Domäne sammelt.

Weiters werden alle Aggregate definiert, auf die die Schedulingobjekte eingeplant werden sollen (die von Typ plan). Für das Stahlwerk LD3 definieren wir drei Aggregate: *CC3* für die Stranggußanlage mit dem V-Verteiler, *CC4* für die Stranggußanlage mit dem E-Verteiler und *vak* für eine Vakuumanlage.

Nun werden auch die Einschränkungen für jedes Aggregat modelliert. Diese beziehen sich zum Beispiel auf den Platz, den ein Objekt in einem Aggregat besetzt. Beschreibt z.B. das Attribut *delivery_date* wann spätestens dieser Auftrag bearbeitet sein sollte, so werden die Regeln mittels DynaFLIP++ definiert, wie die Einplanung von diesem Auftrag auf einen bestimmten Platz in diesem Aggregat bewertet werden soll.

Die Einschränkungen beziehen sich auch auf die Nachbarobjekte im Aggregat. Haben mehrere Aufträge ähnliche Parameter (chemische Kompatibilität, Formatverstellung, u.s.w.) so werden sie auch zusammengegossen. Der Optimierungsalgorithmus muß bei Aufträgen mit geringerem Gewicht, die deswegen schlecht Bewertet sind, danach streben, kompatible Aufträge zu finden um diese zusammenfassung zu ermöglichen.

Die dritte Art der Einschränkung bezieht sich auf das Aggregat selbst. Jeder Verteiler einer Stranggußanlage muß nach einer Anzahl von gegossenen Chargen gewartet werden. Deswegen wird ein neuer Verteiler genommen und der sogenannte Verteilerwechsel durchgeführt. Die Verteilerlebensdauer beträgt schätzungsweise 240 Minuten. Die Einschränkungen der Aggregate *CC3* und *CC4* sollten den Verteilerwechsel optimal einplanen.

5.3.3 Domänenebene

Einschränkungen für die Domäne beschreiben die Abhängigkeiten zwischen den Aggregaten. Wird zum Beispiel eine Charge an der Vakuumanlage fertig, so muß sie innerhalb von kurzer Zeit gegossen werden (das heißt auf der *CC3* oder *CC4* Anlage eingeplant werden). Diese Bedienung erzwingt für jeden Auftrag auf der Vakuumanlage ein schnelles Einplanen an der Stranggußanlage.

Für alle in dieser Domäne definierten Einschränkungen werden auch Reparaturschritte festgelegt. Man kann dabei die bereits implementierten Reperaturschritte verwenden, oder auch domänenspezifische definieren. Zu den bereits implementierten Reparaturschritten zählen:

- **MOVE_FWD** bewegt ein Schedulingobjekt um eine Stelle zum Listenanfang hin.
- **MOVE_BWD** bewegt ein Schedulingobjekt um eine Stelle zum Listenende hin.
- **MOVE_GRP_FWD** bewegt eine Gruppe von Schedulingobjekten zum Listenanfang hin.
- **MOVE_GRP_BWD** bewegt eine Gruppe von Schedulingobjekten zum Listenende hin.
- **CHANGE** vertauscht zwei Schedulingobjekte.
- **INSERT** verschiebt ein Schedulingobjekt von einem Aggregat vom Typ *pool* ins Aggregat vom Typ *plan*.

- **REMOVE** verschiebt ein Schedulingobjekt von einem Aggregat von Typ *plan* ins Aggregat von Typ *pool*.
- **MOVE_TO_END** bewegt ein Schedulingobjekt zum Ende der Liste.
- **MOVE_TO_BEG** bewegt ein Schedulingobjekt zum Anfang der Liste.

Jetzt sollte man alle Daten einlesen und eine Optimierungsmethode auswählen. Es ist sinnvoll, mit einer kleinen - überschaubaren Menge von Daten anzufangen und dabei die Modellierung der Einschränkungen zu verfeinern, noch bessere Optimierungsschritte zu definieren und das Verhalten der Optimierungsalgorithmen zu lernen. Hat das Modell die benötigte Reife erreicht, so kann das System endlich mit der Bearbeitung der realen Daten beginnen.

Literaturverzeichnis

[1] K. Baker. *Introduction to Sequencing and Scheduling.* John Wiley and Sons, Inc., 1974.

[2] G. Booch. *Object Oriented Design.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[3] P. Coad and E.Yourdon. *Object Oriented Analysis.* Yourdon Press Computing Series. Prentice Hall, Inc, 1991.

[4] D. de Werra and A. Hertz. Tabu search techniques: A tutorial and an application to neural networks. *OR Spectrum*, 11:131–141, 1989.

[5] J. Dorn and R. Shams. An expert system for scheduling in a steelmaking plant. In *Proceedings of the 1st World Congress on Expert Systems*, pages 395–404, Orlando Fla., December 1991. Pergamon Press.

[6] J. Dorn, W. Slany, W. Snopek, Ch. Stary, W. Steindl, and K. Stohl. Aufgabenananlyse der Dispatcher im Stahlwerk LD3. CD-Studie 93/11, Christian Doppler Laboratory for Expert Systems, Technische Universität Wien, 1993.

[7] D. Embley, K. Kurtz, and W. Woodfield. *Object-Oriented Systems Analysis.* Prentice-Hall, Inc., Englewood Cliffs, 1992.

[8] P. Freeman. Reusable software engineering: Concepts and research directions. In *ITT Proceedings of the Workshop on Reusability in Programming*, pages 129–137, 1983.

[9] H. Gall and R. Klösch. *Software Wiederverwendung.* Technische Universität Wien, 1994.

[10] M. Girsch. Optimierung von schedules mit genetischen algorithmen und interativer vertiefung. Master's thesis, Technische Universität Wien, 1994.

[11] J. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[12] P. Laarhoven, E. Aarts, and L. Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, 40:113–125, 1992.

[13] R. G. Lanergan and B. A. Poynton. Reusable code: The application development technique of the future. In *Proc. of the IBM SHARE/GUIDE Software Symposium*, pages 88–98, Monterey, CA, October 1979. IBM.

[14] G. Lawton. Genetic algorithms for schedule optimization. *AI Expert*, pages 23–27, 1992.

[15] M. Lubars. A knowledge-based design aid for the construction of software systems. Report nr. uiucdcs-r-86-1304, Department of Computer Science, University of Illinois, Urbana, 1986.

[16] Y. Matsumoto. *Software Engineering Environments.* North-Holland, New York, 1980. Seiten 305-318.

[17] Y. Matsumoto. Management of industrial software production. *IEEE Computer*, 17(2):59–72, February 1984.

[18] M. D. McIlroy, J. M. Buxton, P. Naur, and B. Randell. Mass-produced software components. In *Software Engineering Concepts and Techniques*, pages 88–98. 1968 NATO Conf. Software Engineering, 1976.

[19] J. M. Neighbors. *Software Construction Using Components.* PhD thesis, University of California, Irvine, CA, 1981.

[20] J. M. Neighbors. The draco approach to constructing software from reusable components. *IEEE Trans. on Software Engineering*, SE-10(5):564–574, September 1984.

[21] J. M. Neighbors. Draco: A method for engineering reusable software systems. *ACM Press Software Reusability*, 1:99–122, 1989.

[22] R. Prieto-Diaz. Status report: Software reusability. *IEEE Software*, 10(3):61–66, May 1993.

[23] J. Rumbaugh, M. Blaha, W. Premerlani, and F. Eddy. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., Englewood Cliffs, 1991.

[24] G. Skele. Repair based scheduling mit tabu search. Master's thesis, Technische Universität Wien, 1994.

[25] B. Stroustrup. *C++ Programming Language.* Addison-Wesley Publishing Company, 1986.

[26] B. Stroustrup and M. A. Ellis. *C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.

[27] M. V. Zelkowitz, A. C. Shaw, and J. D. Gannon. *Principles of Software Engineering and Design.* Prentice-Hall, Inc., Englewood Cliffs, 1989.