Technische Universität Dresden
Fakultät Elektrotechnik und Informationstechnik
Institut für Aufbau- und Verbindungstechnik der Elektronik
PD Dr.-Ing. G. Weigert

Ringpraktikum
Mikrorechentechnik II

Simulation und Optimierung von Fertigungsprozessen

Mai 2015

Inhalt

1 Zielstellung **4**

2 Allgemeine Grundlagen **5**

2.1 Diskrete Fertigungsprozesse und -systeme 5
2.2 Simulation von Fertigungsabläufen 7
2.3 Optimierung von Fertigungsabläufen 10

3 Programmierung **13**

3.1 Listen 13
3.2 Das Simulationssystem ROSI 14
3.3 Warteschlangen im Simulationssystem 16

4 Aufgabenstellung **17**

5 Literaturhinweise **19**

6 Arbeits- und Brandschutzhinweise **19**

7 Anhänge **20**

1 Zielstellung

Die Zielstellung dieses Praktikumsversuchs besteht darin, den Umgang mit dynamischen Datenstrukturen (Listen) in der Programmiersprache C++ zu üben bzw. zu festigen. Darüber hinaus dient der Versuch auch dazu, Grundlagen der Produktionsplanung und –steuerung sowie Methoden der ereignisdiskreten Simulation und der heuristischen simulationsgestützten Optimierung kennenzulernen. Gegenstand des Versuchs ist das Simulationssystem *ROSI*, in dem eine eigene Methode zur Vertauschung von Jobs in einer Warteschlange implementiert werden soll. Der Test der Methode erfolgt anhand von Experimenten zur Optimierung von Reihenfolgen.

Schwerpunkte des Praktikumsversuchs sind:

- Objektorientiertes Programmieren in C++
- Dynamische Objektverwaltung in Listen
- Modulare Programmgestaltung
- Arbeiten mit Microsoft Visual C++
- Simulation und Optimierung diskrete Fertigungsprozesse

Organisatorische Hinweise:

Die Leistung im Praktikum wird mit maximal 10 Punkten bewertet. Davon entfallen 6 Punkte auf den Eingangstest, 2 Punkte auf die Praktikumsdurchführung und 2 Punkte auf das Protokoll. Der Eingangstest wird als individuelle Leistung, die Praktikumsdurchführung und das Protokoll werden als kollektive Leistung bewertet. Das Protokoll ist spätestens eine Woche nach dem Praktikumstermin abzugeben. Beachten Sie bitte auch die **Hinweise zur Abfassung des Protokolls** (Abschnitt 4). Aus Zeitgründen ist es dringend angeraten, das Programm bereits in Vorbereitung zum Praktikum zu erstellen und den Programmcode in elektronischer Form zum Versuchstermin mitzubringen!

Der Praktikumsversuch beginnt mit einem schriftlichen Eingangstest (30-45 Minuten Dauer, mündlicher Eingangstest vorbehalten). Für die Praktikumsteilnahme sind mindestens 3 Punkte erforderlich. Als Vorbereitung für den Eingangstest empfiehlt sich ein gründliches Studium dieser Praktikumsanleitung. Jedes Mitglied der Praktikumsgruppe sollte in der Lage sein, elementare Listenoperationen in C++ selbst zu schreiben bzw. entsprechenden Programmcode zu interpretieren.

Schwerpunkte im Eingangstest sind:

- Objekte und deren Verwaltung (Klassen, Methoden, …)
- Zeigeroperationen in C++ (z.B. e->Next=First; First=e;)
- Einfach und doppelt verkettete Listen (z.B. Schreiben Sie eine Methode zur Ermittlung der Länge der Liste, zum Einfügen oder Entfernen von Listenelementen!)
- Interpretation des im Anhang (Abschnitt 7) veröffentlichten Programmcodes (Welcher Listentyp? Was passiert konkret beim Ausführen der Methoden?)
- Sie sollten selbstverständlich in der Lage sein, Ihren eigenen vorbereiteten Quelltext zu erklären!
- Diskrete Fertigungsprozesse und ihre Abbildung im Simulationssystem, Funktion und Bedeutung von Warteschlangen
- Reihenfolgen und ihre Anwendung bei der Steuerung und Optimierung von Fertigungsprozessen

Konkrete Hilfestellung zur Versuchsdurchführung finden Sie in den einzelnen Abschnitten dieser Anleitung jeweils unter „**Hinweise …**“:

- **Hinweise für die Durchführung von Simulationsläufen**
  (Abschnitt 2.2)

- **Hinweise für die Durchführung von Optimierungsläufen**
  (Abschnitt 2.3)
- **Hinweise zur Versuchsdurchführung**
  (Abschnitt 4)
- **Hinweise zur Abfassung des Protokolls**
  (Abschnitt 4)

Eine vollständige Bedienungsanleitung für das Simulationssystem *ROSI*, einschließlich der Beschreibung aller Befehle, finden Sie in der Online-Hilfe /2/:

http://www.avt.et.tu-dresden.de/rosi/hlp_rosi/contents.htm

Weitere Informationen u.a. auch zu dem kommerziellen Simulationssystem simcron MODELLER finden Sie auch auf unserer Homepage:

http://www.avt.et.tu-dresden.de/rosi/

Die im Abschnitt 5 angegebene Zusatzliteratur kann für die Erweiterung und Vertiefung der Kenntnisse genutzt werden, ist aber für die Durchführung des Praktikumsversuchs nicht unbedingt erforderlich.

2 Allgemeine Grundlagen

2.1 Diskrete Fertigungsprozesse und -systeme

Elektronikproduktion findet im Allgemeinen als diskreter Fertigungsprozess statt. Das trifft z.B. in der Halbleiterindustrie aber auch bei der Herstellung von Leiterplatten sowie bei der Fertigung elektronischer Baugruppen und Geräten zu. Diskret bedeutet hier, dass sowohl die Produktionsmittel (z.B. Maschinen, Werkzeuge, Transport- und Lagereinrichtungen) als auch die Produkte (z.B. Wafer, Leiterplatten, Geräte) separate Objekte sind und der Fertigungsprozess aus zeitlich und funktional abgegrenzten einzelnen Arbeits- bzw. Prozessschritten besteht. Da jeder Fertigungsprozess Zeit und Ressourcen erfordert, kommt der Planung und Optimierung der Fertigungsabläufe eine große Bedeutung zu.

Die Theorie der Ablaufplanung - auch als Scheduling-Theorie bezeichnet - hat sich inzwischen zu einer eigenständigen, stark mathematisch geprägten Wissenschaft entwickelt. Bei der Ablaufplanung werden im Allgemeinen sehr abstrakte Modelle des jeweiligen Fertigungssystems verwendet. Alle Produkte, unabhängig von ihrer Art, werden unter dem Begriff Job zusammengefasst und die Produktionsmittel als allgemeine Ressourcen aufgefasst. Die einzelnen Arbeitsschritte werden lediglich auf ihren Zeitbedarf reduziert, d.h., es ist nicht von Bedeutung, was in dem betreffenden Arbeitsschritt geschieht, sondern nur, wie lange der Arbeitsschritt dauert. Jobs und Ressourcen verhalten sich komplementär zueinander, ähnlich wie Verbraucher (=Jobs) und Anbieter (=Ressourcen). Wir setzen im Folgenden voraus, dass für jeden Arbeitsschritt eine oder mehrere Ressourcen in Anspruch genommen werden.

In der Regel produziert ein Fertigungssystem nicht nur ein Produkt, sondern eine Vielzahl unterschiedlicher Produkte gleichzeitig. Dadurch kommt es immer wieder zu Konflikten, etwa wenn die Kapazität einzelner Maschinen begrenzt ist. In diesen Fällen kann es geschehen, dass ein Job warten muss, bis ein anderer Job die benötigte Maschine frei gegeben hat. Diese ablaufbedingten Wartezeiten sind schwer vorhersagbar, haben aber großen Einfluss auf die Effizienz des Fertigungsprozesses. Wartezeiten können durch geeignete Maßnahmen, wie etwa Prioritätsregeln, minimiert werden.

Fertigungssysteme, insbesondere in der Halbleiterproduktion, können sehr komplexe Strukturen und Zusammenhänge aufweisen. So werden z.B. mehrere Maschinen zu so genannten Cluster-Tools mit speziellen Ablaufregeln zusammengefasst. Zwischen dem Prozessschritt „Nasschemie“ (Ätzschritt) und dem nachfolgenden „Ofenprozess“ (Passivierungsschritt) dürfen bestimmte Wartezeiten nicht überschritten werden (Zeitkopplung). Der Ofenprozess selbst ist als Batch-Prozess ausgelegt, d.h., der Prozessschritt wird erst gestartet, wenn eine ausreichende Anzahl Jobs bereitsteht. Oft muss auch noch eine Zusatzzeit (Rüstzeit) eingeplant werden, um die Maschine oder Anlage auf ein anderes Produkt umzustellen (z.B. Werkzeugwechsel oder Einmessen der Anlage). Bereits diese kurze Aufzählung zeigt, was bei der Planung realer Fertigungsprozesse alles beachtet werden muss.

Im Folgenden werden nur einfache Fertigungssysteme betrachtet, an denen sich auch ohne tiefere Kenntnis der Scheduling-Theorie die Grundprinzipien der Fertigungsplanung und –steuerung zeigen lassen. Die hier behandelten Modelle enthalten nur zwei Typen von Ressourcen: Maschinen und Lager (Warteschlangen). Es wird weiter angenommen, dass der Job in jedem Arbeitsschritt genau eine Ressource beansprucht. Die erforderliche Bearbeitungszeit an jeder Maschine ist technologisch bedingt und wird als determiniert angenommen. Zwischen den einzelnen Arbeitsschritten kann der Job in einer Warteschlange gelagert werden, wobei die Wartezeit unbestimmt (≥ 0) ist. Die Jobs werden in einem Eingangslager bereitgestellt und nach Prozessende in einem Ausgangslager abgelegt. Die Aufnahmekapazität der Warteschlangen wird als unendlich angenommen, während eine Maschine stets nur einen Job gleichzeitig aufnehmen kann. Modelle, bei denen jedem Arbeitsschritt genau eine Ressource und genau ein Job zugeordnet ist, wobei Arbeitsschritte, die denselben Job und/oder dieselbe Ressource enthalten, nicht gleichzeitig ausgeführt werden können, werden in der Scheduling-Theorie auch als Shop-Modelle bezeichnet. Sie bilden die Grundlage für viele Optimierungsverfahren.

Abbildung 1: Die drei Grundbausteine Warteschlange, Maschine und Job (von links nach rechts). Die Warteschlange ist mit 2, die Maschine mit einem Job belegt.

> Frage 1:
>
> Finden Sie Beispiele aus der Praxis, die sich nicht als einfaches Shop-Modell beschreiben lassen!

Um funktionsfähige Produkte zu erhalten, müssen nicht nur bestimmte Arbeitsschritte ausgeführt werden, sondern es ist wichtig, dass die richtigen Vorgänger-Nachfolger-Beziehungen zwischen den Arbeitsschritten eingehalten werden. So kann man z.B. bei der Herstellung einer elektronischen Baugruppe den Arbeitsschritt „Löten“ nicht vor dem Arbeitsschritt „Bestücken“ ausführen. Im einfachsten Fall lassen sich diese Beziehungen durch eine technologische Reihenfolge beschreiben. Jeder Job besitzt eine eigene technologische Reihenfolge in Gestalt einer Maschinen-Reihenfolge, z.B. Job1: Maschine1 → Maschine2 → Maschine4. Komplementär zur technologischen Reihenfolge existiert auch eine organisatorische Reihenfolge, die die Job-Reihenfolge beschreibt, die man im Verlauf des Fertigungsprozesses an einer Maschine beobachten kann. Jede Maschine besitzt eine eigene organisatorische Reihenfolge, z.B. Maschine1: Job1 → Job3 → Job2. Ist für alle Jobs in einem Fertigungssystem nur eine technologische Reihenfolge zugelassen, bezeichnet man dieses Fertigungssystem als Flow Shop bzw. Fertigungslinie. Da die Bearbeitungszeiten an den einzelnen Maschinen des Flow Shops für jeden Job verschieden sein können, sollte man besser von einer asynchronen Fertigungslinie sprechen. Eine (synchrone) Taktstraße ist somit ein Sonderfall des Flow Shops. Werden dagegen Jobs mit unterschiedlichen technologischen Reihenfolgen in einem Fertigungssystem bearbeitet, spricht man von einem Job Shop. Der Job Shop ist nach dem Werkstattprinzip organisiert, d.h., es gibt unterschiedliche Kundenaufträge, die aber jeweils nach fest vorgegebenen technologischen Regeln abgearbeitet werden. Für die Bearbeitungszeiten gelten selbstredend keine Einschränkungen. Somit kann der Flow Shop als Sonderfall des Job Shops aufgefasst werden. Sind keine technologischen Reihenfolgen vorgegeben, handelt es sich um die allgemeinste Form der hier aufgezählten Fertigungssysteme, einen so genannten Open Shop. /1/

> Frage 2:
>
> Können Sie ein Beispiel (nicht notwendig aus der Produktion) für einen Open Shop nennen?

2.2 *Simulation von Fertigungsabläufen*

Für die Modellierung diskreter Fertigungsprozesse haben sich ereignisorientierte Simulationssysteme als unentbehrliche Werkzeuge etabliert. Das liegt daran, dass es keine allgemeingültigen Formeln gibt, mit denen man ein reales Fertigungssystem, unter Berücksichtigung aller Nebenbedingungen, hinreichend genau analysieren kann. Dazu sind die im System ablaufenden Prozesse viel zu kompliziert und unübersichtlich. Das Problem kann durch Prozessschleifen (in der technologischen Reihenfolge eines Jobs kommt ein und dieselbe Maschine mehrfach vor) oder Blockierungen (die Freigabe einer Maschine wird durch die Maschine des nachfolgenden Arbeitsschritts verhindert) weiter erschwert werden.

> Frage 3:
>
> Wie lassen sich Blockierungen im Fertigungssystem verhindern?

Für den Praktikumsversuch liegen zwei Simulationsmodelle vor. Das erste Modell (`flowshop.mc`) ist ein Flow Shop, bestehend aus 2 Maschinen und 9 Jobs. Die technologische Reihenfolge ist gegeben durch M1 → M2. Das Technologie-Objekt T1 enthält die technologische Reihenfolge einschließlich der determinierten Bearbeitungszeiten als Liste (Abbildung 2). Da die technologische Reihenfolge aller Jobs im Flow Shop die gleiche ist, reicht ein Technologie-Objekt zur Beschreibung der Routen aus. Die unterschiedlichen Bearbeitungszeiten der einzelnen Jobs werden nur spezifiziert (Abbildung 2, linke Spalte: Spezifikator 1 … 9). Abbildung 2 zeigt also die Bearbeitungszeiten für den Job1. Man beachte, dass in T1 auch die Lager als „Arbeitsschritt“ abgebildet sind!

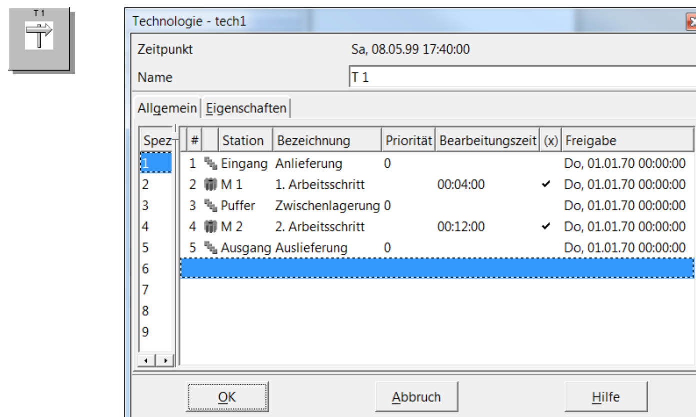

Abbildung 2: Technologische Reihenfolge im Modell `flowshop.mc`

Beide Maschinen sind durch ein Pufferlager entkoppelt. Im Anfangszustand liegen alle Jobs im Eingangslager. Nach Beendigung des Simulationslaufs sollten alle Jobs im Ausgangslager liegen. Eine Besonderheit dieses Flow Shops besteht darin, dass sich die Jobs im System nicht gegenseitig überholen können, d.h., die organisatorischen Reihenfolgen sind an jeder Station gleich. Dieser Spezialfall wird auch als Permutations Flow Shop bezeichnet.

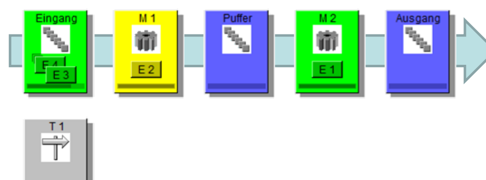

Abbildung 3: Flow Shop (M1 → M2) mit Pufferlager

Frage 4:

Wie wirkt sich die Entfernung des Pufferlagers auf den Fertigungsablauf aus?

Das zweite Modell (`jobshop.mc`) ist ein Job Shop. Es handelt sich dabei um ein klassisches Referenzmodell, das bereits in den 1960iger Jahren von Muth und Thompson für den Test von Scheduling-Algorithmen entwickelt wurde. Es besteht aus insgesamt 10 Maschinen und 10 Jobs. Jeder Job besitzt eine eigene technologische Reihenfolge, die in den Technologie-Objekten T1 bis T10 abgelegt ist. Das Modell ist so angelegt, dass jeder Job jede Maschine genau einmal besetzt. Die Maschinen erhalten ihren Job entweder direkt aus dem Eingangslager (erster Arbeitsschritt) oder aus dem zentralen Pufferlager. Die letzte Maschine in der jeweiligen technologischen Reihenfolge legt den Job schließlich im Ausgangslager ab.

Frage 5:

Warum ist das Pufferlager hier unbedingt notwendig? Was könnte im schlimmsten Fall passieren, wenn man das Pufferlager entfernt?

Überlegen Sie, wie viele Jobs Sie höchstens im Modell anlegen könnten, wenn keine zwei Jobs die gleiche technologische Reihenfolge haben sollen?

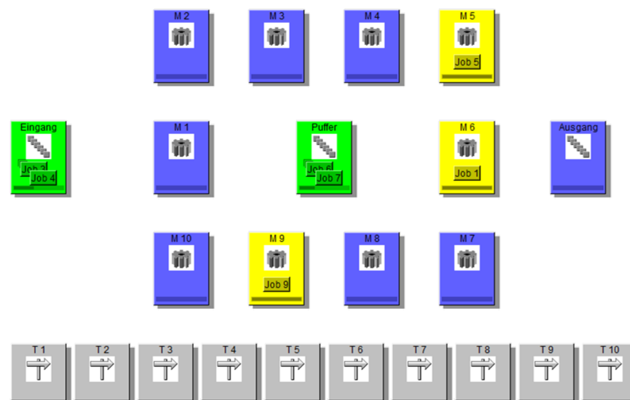

Abbildung 4: Job Shop des Modell `jobshop.mc`

Hinweise für die Durchführung von Simulationsläufen:

Es ist nicht möglich, einzelne Simulationsschritte rückgängig zu machen (Keine Undo-Funktion!). Wenn Sie ein Modell noch einmal starten wollen, laden Sie es zuvor neu (*Datei* → *Neu laden* oder - Symbol in der Symbolleiste). Beim Schließen oder „*Neu laden*" eines Modells werden Sie gefragt, ob Sie das aktuelle Modell speichern wollen. Diese Frage ist in der Regel mit „*Nein*" zu beantworten. Anderenfalls überschreiben Sie das Originalmodell evtl. ungewollt mit einem neuen Zustand.

Der Simulationslaufs kann mit dem Menü-Punkt *Simulation* → … oder direkt über die Symbolleiste gesteuert werden:

(… → *Start*)	Startet den Simulationslauf und endet mit Stopp-Ereignis, animiert
(… → *Schritt*)	Führt einen einzelnen Simulationsschritt aus
(… → *Blitzsimulation*)	Wie *Start*, aber ohne Animation
(… → *Stopp*)	Unterbricht den Simulationslauf

Der Zustand der Maschinen- und Warteschlangen-Bausteine wird während des Simulationslaufs animiert. Den einzelnen Zuständen sind folgende Farben zugeordnet:

Blau – Die Station ist leer.

Gelb – Die Station ist besetzt und arbeitet.

Grün – Die Station ist besetzt, der oder die Jobs sind aber fertig und können abgegeben werden.

> Frage 6:
>
> Beobachten Sie während des Simulationslaufs die Farben der Bausteine. Welche Unterschiede können Sie zwischen Maschinen und Warteschlangen erkennen?

Sie können den Transfer der Jobs von Station zu Station auch animieren (*Animation* → *Ereignisse*, anschließend *Transfer* aktivieren). Die Ereignisfolge und die Simulationsuhr lassen sich zusätzlich einblenden (*Ansicht* → *Monitor*). Die Simulationszeit ab Beginn des Simulationslaufs wird sowohl in [Tag, Stunden:Minuten:Sekunden] als auch in Sekunden angezeigt. Nach dem regulären Ende des Simulationslaufs (Stopp-Ereignis) zeigt die Simulationszeit die so genannte Zykluszeit an. Die Zykluszeit ist die Zeitspanne zwischen dem Beginn des ersten Arbeitsschritts und dem Abschluss des letzten Arbeitsschritts im Fertigungsprozess. Sie wird wesentlich durch die Wartezeiten der Jobs in den Warteschlangen beeinflusst. In realen Fertigungsprozessen beträgt die Wartezeit oft ein Vielfaches der aktiven Bearbeitungszeit.

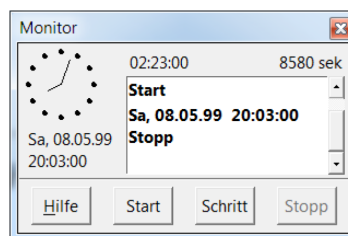

Abbildung 5: Simulations-Monitor (*Ansicht* → *Monitor*)

Einen Überblick über den Simulationsablauf können Sie sich auch mittels Maschinenbelegungsdiagramm beschaffen (*Analyse* → *Gantt* → *Stationen*). Die einzelnen Arbeitsschritte werden hier durch farbige Balken unter der Zeitachse dargestellt. Die Farben der Balken entsprechen den oben erwähnten Zustandsfarben der Bausteine. Durch Anklicken eines beliebigen Balkens im Diagramm markieren Sie alle Arbeitsschritte des zugehörigen Jobs (Abbildung 6). Diese Diagramme werden auch nach seinem Erfinder, dem amerikanischen Ingenieur Henry Laurence Gantt (1861–1919), bezeichnet. Falls Sie eine Job-orientierte Sicht wünschen, öffnen Sie das entsprechende Gantt-Diagramm (*Analyse* → *Gantt* → *Jobs*). In diesem Diagramm lassen sich alle Arbeitsgänge markieren, die zu einer Maschine gehören.

> Frage 7:
>
> Überlegen Sie, wie Sie die organisatorische Reihenfolge an einer Station ermitteln können. Vergleichen Sie die verschiedenen Möglichkeiten.

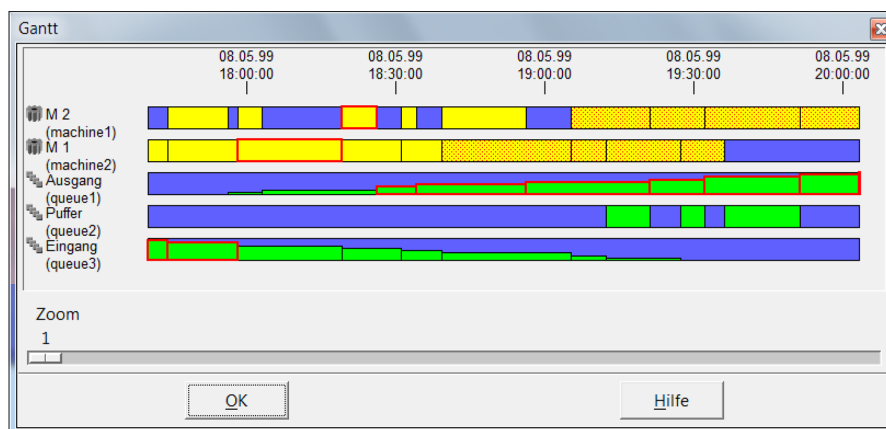

Abbildung 6: Gantt-Diagramm des Flow Shops mit markiertem Job (*Analyse* → *Gantt* → *Stationen*)

2.3 *Optimierung von Fertigungsabläufen*

Mit Hilfe der Simulation lassen sich Fertigungsprozesse analysieren und bewerten. Ausgewählte Kenngrößen wie Zykluszeit, Durchlaufzeit, Liegezeit, Maschinenauslastung oder Termintreue können so virtuell im Voraus bestimmt werden. Da ein Simulationslauf wesentlich weniger Zeit benötigt als der reale Fertigungsprozess und außerdem keine realen Ressourcen verbraucht, kann er wiederholt werden. So kann man mit verschiedenen Varianten des Fertigungsprozesses experimentieren und die Auswirkungen von Steuerstrategien auf den Fertigungsablauf studieren. Damit lassen sich schließlich Entscheidungen für eine möglichst effektive Steuerstrategie treffen. Dennoch ist die Anwendung der Simulation allein noch keine Optimierung.

Um einen Fertigungsprozess zu optimieren, muss man sich zunächst Klarheit über die Ziele verschaffen, die erreicht werden sollen. Meist wird man dafür konkrete Kenngrößen definieren, deren Werte jeweils minimiert oder maximiert werden sollen. Anschließend ist zu entscheiden, welche Möglichkeiten bestehen, auf den Fertigungsprozess Einfluss zu nehmen. Es geht also darum, Ziel- und Einflussvariable festzulegen. Dabei ist es wichtig, beide nicht miteinander zu verwechseln! Als Zielvariable kommen u.a. die bereits eingangs aufgeführten Kenngrößen in Frage. Einflussvariable können bei einem diskreten Fertigungsprozess z.B. Prioritäten für Jobs oder einzelne Arbeitsschritte, Bearbeitungszeiten, Maschinen- und Lagerkapazität oder Reihenfolgen sein. Oft muss man noch Nebenbedingungen definieren, die sowohl als harte oder weiche Forderung in Erscheinung treten können. So kann z.B. gefordert sein, das ein bestimmter Job einen vorgegebenen Termin keinesfalls (harte Nebenbedingung) oder möglichst wenig (weiche Nebenbedingung) überschreitet. Eine Lösung des Optimierungsproblems in den von den Nebenbedingungen gezogenen Grenzen kann nicht immer garantiert werden.

In der Praxis sollen meist mehrere Zielvariable zugleich optimiert werden. In diesem Fall handelt es sich um ein multikriterielles Optimierungsproblem, für das eine Kompromisslösung gefunden werden muss. Ähnliches gilt für die Einflussvariablen, die oft zu einem Steuervektor zusammengefasst werden können. Besonders geeignet als Einflussvariable sind organisatorische Reihenfolgen. Im Gegensatz zu technologischen Reihenfolgen sind sie nicht durch die jeweilige technologischen Anforderungen festgelegt und sie lassen sich in der Regel auch leicht verändern, ohne dabei zusätzliche Kosten zu verursachen. Als Werte kann eine solche Variable verschiedene Permutationen annehmen. Diese Art der Optimierungsprobleme wird auch als Reihenfolgeproblem bezeichnet.

> Frage 8:
>
> Wie viele unabhängige organisatorische Reihenfolgen (Einflussvariable) könnte man jeweils in den Modellen `flowshop.mc` und `jobshop.mc` anlegen?

Man spricht von diskreten Optimierungsproblemen, wenn eine oder mehrere Einflussvariable nur diskrete Werte annehmen können. Das trifft auf die Reihenfolgeprobleme zu, wobei der Wertevorrat darüber hinaus noch endlich ist. Diskrete Optimierungsprobleme mit endlichem Wertevorrat lassen sich prinzipiell auch auf einfache Weise lösen, indem man für jeden Wert der Einflussvariablen die zugehörigen Zielgrößen berechnet oder simuliert und anschließend durch Vergleich die optimale Lösung findet. Man nennt dieses Vorgehen auch vollständige Enumeration (Auszählung). In vielen Fällen, so auch bei Reihenfolgevariablen, steigt die Anzahl der diskreten Werte jedoch exponentiell mit der Problemgröße, d.h. in unserem Fall mit der Anzahl der Jobs. Die meisten Reihenfolgeprobleme gehören zur Klasse der NP-schweren Optimierungsprobleme, für die es keine effizienten Lösungsalgorithmen gibt /1/. Die Methode der vollständigen Enumeration lässt sich in diesem Fall schon bei verhältnismäßig kleinen Problemen nicht mehr praktisch anwenden.

> Frage 9:
>
> Wie viele verschiedene Permutationen lassen sich mit den 9 Jobs des Modells `flowshop.mc` bzw. den 10 Jobs des Modells `jobshop.mc` bilden, wenn man nur die organisatorische Reihenfolge in der Eingangswarteschlange als Einflussvariable zulässt?

Ein simulationsgestütztes Optimierungssystem arbeitet nach dem Prinzip „Suchen und Bewerten“ (siehe Abbildung 7). Im Suchschritt werden konkrete Werte für die Einflussvariablen festgelegt, was einer möglichen Lösung des Problems entspricht. Im anschließenden Simulationsschritt wird die potentielle Lösung bewertet, indem die zugehörigen Zielgrößen ermittelt werden. Dieser Optimierungszyklus wird so oft wie-

derholt, bis ein Abbruchkriterium erreicht oder der Zyklus manuell unterbrochen wird. Das wiederholte experimentieren mit unterschiedlichen Simulationsläufen für ein und dasselbe Modell wird quasi automatisiert.

Der Vorteil der simulationsgestützte Optimierungssysteme besteht in ihrer Flexibilität. Durch Austausch des Simulationsmodells kann das System praktisch an jede Optimierungsaufgabe angepasst werden. Mathematische Formeln oder Gleichungssysteme sind nicht erforderlich. Für die Suchalgorithmen kommen unterschiedliche, meist jedoch heuristische Verfahren zum Einsatz. Stellvertretend seien hier Genetische Algorithmen, Simulierte Abkühlung oder Schwellenwertverfahren (z.B. Threshold Accepting) genannt.

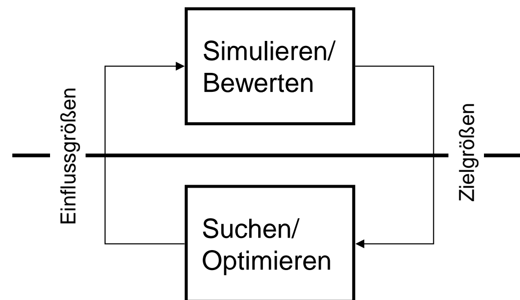

Abbildung 7: Prinzip der simulationsgestützten heuristischen Optimierung

Die meisten Heuristiken enthalten eine stochastische Komponente für die Suche, verwenden aber auch in unterschiedlichem Umfang Informationen aus vergangen Simulationsläufen, um so die Konvergenzgeschwindigkeit zu verbessern. Oft werden beim Entwurf einer neuen Heuristik Erscheinungen in der Natur kopiert, die in irgendeiner Weise mit Optimierungsvorgängen zu tun haben (z.B. Evolution → Die überlebensfähigsten Individuen setzen sich durch., Abkühlungsprozesse → Bei der Erstarrung einer Schmelze wird das Energieniveau des Kristallgitters minimiert.). Anschließend wird eine formale Korrespondenz zwischen den natürlichen Objekten (z.B. Individuen einer Art) und den Einflussvariablen (z.B. Permutationen einer organisatorischen Reihenfolge) hergestellt.

Der Nachteil simulationsgestützter heuristischer Optimierungsverfahren besteht darin, dass es keine Garantie für den Erfolg der gewählten Strategie gibt. Suchstrategie und Simulationsmodell sind zwar unabhängig voneinander, müssen aber dennoch aufeinander abgestimmt sein. Zudem sind die Simulationsläufe oft sehr zeitintensiv, so dass diese Verfahren für zeitkritische Anwendungen (z.B. Online-Optimierung von Fertigungsprozessen) nicht immer geeignet sind, insbesondere dann, wenn der Optimierungszyklus sehr oft wiederholt werden muss. Wurde schließlich eine bessere Lösung gefunden, fehlen Informationen darüber, wie weit diese Lösung vom Optimum tatsächlich entfernt ist. Simulationsgestützte heuristische Optimierungsverfahren sind daher überall dort angeraten, wo es nicht auf das Optimum an sich, sondern lediglich auf eine verbesserte Lösung ankommt, wo ausreichen Zeit vorhanden ist und wo das Simulationsmodell auch noch für andere Aufgaben genutzt werden kann.

Für die beiden Simulationsmodelle `flowshop.mc` und `jobshop.mc` soll jeweils die Zykluszeit (Zielgröße) minimiert werden. Als Einflussgröße steht in beiden Modellen die Job-Reihenfolge in der Eingangswarteschlange zur Verfügung. Beide Variablen sind in den Modellen bereits angelegt. Für die heuristische Suche stehen mehrere Algorithmen zur Verfügung, von denen aber nur die so genannte Blinde Suche verwendet werden soll. Durch diese Heuristik wird die Job-Reihenfolge in jedem Optimierungszyklus stochastisch verändert, ohne dass Informationen aus den vorangegangenen Optimierungszyklen ausgewertet werden. Die Veränderung der Reihenfolge geschieht durch mehrfaches Vertauschen zweier Jobs aus der Eingangswarteschlange. Sowohl die Anzahl der Vertauschungen als auch die jeweiligen Tauschpartner sind zufällig. Beim Vertauschen von Jobs wird auf die zu programmierende Methode `ExchJobs` zurückgegriffen!

Hinweise für die Durchführung von Optimierungsläufen:

Laden Sie das Modell `flowshop.mc` neu und öffnen Sie den Optimierungsmonitor (*Optimierung* → *Optimierung*). Starten Sie die Optimierung mit dem *Start*-Button. Es werden 100 Optimierungszyklen ausgeführt. Jeder Optimierungszyklus führt einen vollständigen Simulationslauf aus, der jedoch nicht animiert wird. Mit Hilfe des *Weiter*-Buttons kann die Optimierung fortgesetzt und mit dem *Stopp*-Button auch vorzeitig abgebrochen werden.

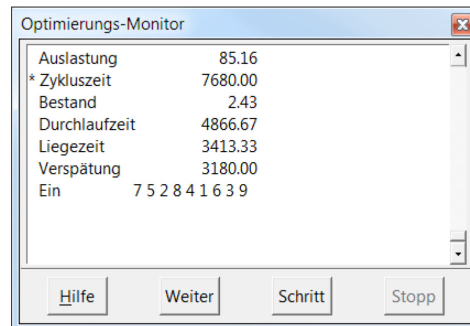

Abbildung 8: Optimierungsmonitor (*Optimierung* → *Optimierung*)

Die Ergebnisse für die Zykluszeit werden im Optimierungs-Graph angezeigt (Abbildung 9). Die stark ausgezeichnete blaue Kurve zeigt den „ewigen" Minimalwert (Optimierungsziel), die rote Kurve den „ewigen" Maximalwert der Zykluszeit als relative Werte. Die Absoluten Werte, gemessen in Sekunden, sind rechts unten im Optimierungs-Graph zu finden. Die dünnen Kurven stehen für die aktuellen Werte. Alle Kurven sind sowohl zeitlich als auch wertmäßig normiert. Der Optimierungslauf kann abgebrochen werden, sobald sich die Zielwerte längere Zeit (etwa 100 oder mehr Optimierungszyklen) nicht mehr ändern.

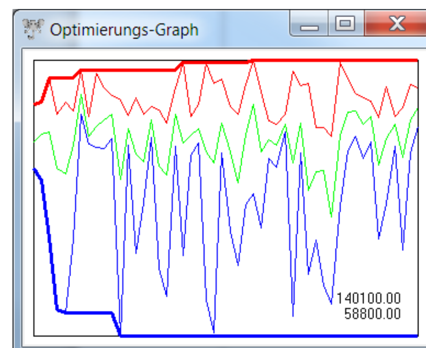

Abbildung 9: Optimierungs-Graph (öffnet sich automatisch)

Nach Abschluss der Optimierung bleibt die Reihenfolge der Jobs in der Eingangswarteschlange zunächst unverändert. Um die optimale Reihenfolge in das Simulationsmodell zu übernehmen, wählen Sie im Menü *Optimierung* → *Ergebnis* und beantworten den Dialog mit „ *Ja* ". Falls Sie das optimierte Modell später reproduzieren möchten, speichern Sie es unbedingt vor dem Simulationslauf ab! Verwenden Sie für das Modell einen eigenen Namen, um später auch noch auf das Originalmodell zurückgreifen zu können.

Die optimale Job-Reihenfolge lässt sich aus der Dialogbox der Eingangswarteschlange, dem Gantt-Diagramm oder aus dem animierten Simulationslauf selbst entnehmen. Wiederholen Sie den Test mit dem Modell `jobshop.mc`.

Frage 10:

Schätzen Sie die Zeit, die ein Optimierungszyklus für die beiden Modelle jeweils benötigt. Wie viel Zeit müsste man für die vollständige Enumeration der Modelle einplanen? Wie viel Zeit würde man für das Job-Shop-Modell benötigen, wenn man die Zahl der Jobs verdoppeln würde?

3 Programmierung

3.1 Listen

Listen sind dynamische Datenstrukturen. D.h., die Länge einer Liste wird nicht im Quellcode des Programms, sondern erst während der Laufzeit festgelegt bzw. verändert. Abbildung 10 zeigt die einfachste Form. Der Listenkopf verwaltet die Liste über den Zeiger `First`, der das erste Listenelement referenziert. Die Listenelemente selbst enthalten einen Zeiger, der jeweils das nächste Element referenziert. Die Aufnahme oder Entnahme eines neuen Elements erfolgt immer über den Kopf der Liste:

```
// Neues Element in die Liste aufnehmen
//------------------------------------
NewElement->Next = First;
First = NewElement;

// 1. Element aus der Liste entfernen
//------------------------------------
Element = First;
First = Element->Next;
Element->Next = NULL;
```

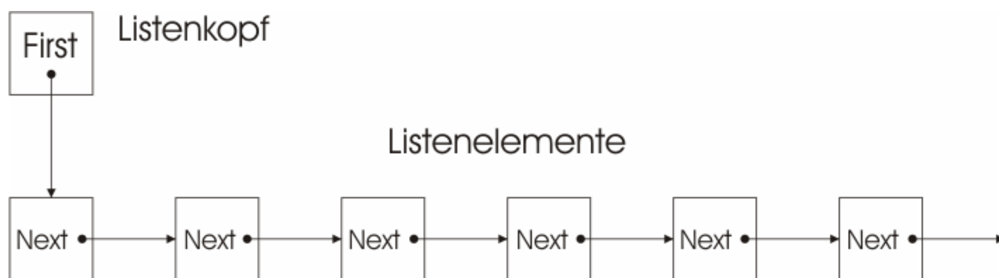

Abbildung 10: Einfach verkettete Liste

Einfach verkettete Listen werden auch als Stapel- oder Kellerspeicher bezeichnet, da sie streng nach dem *lifo*-Prinzip organisiert sind (*lifo* = *last in first out*). Für deren Verwaltung ist nur ein einziger Zeiger erforderlich. Sie werden vor allem dort angewendet, wo ohnehin nur auf das erste Element zugegriffen werden soll (z.B. Parameter-Stack beim Aufruf von Prozeduren) oder wenn es ohne Bedeutung ist, an welcher Stelle der Liste das Element steht (z.B. Halde für gelöschte Objekte). Soll auf ein bestimmtes Element der Liste zugegriffen werden, kann der Aufwand – abhängig von der Länge der Liste – jedoch erheblich sein. Eine doppelt verkettete Liste (Abbildung 11), in der jedes Listenelement zusätzlich auf seinen Vorgänger verweist, ist hier im Vorteil. Die Suche nach bestimmten Elementen kann auch dadurch beschleunigt werden, dass dem Verwalter noch weitere Zeiger hinzugefügt werden (z.B. Zeiger auf den letzten Zugriff in der Liste). Natürlich kann die Verwaltung der Liste auch noch durch andere Parameter, wie z.B. die aktuelle Länge der Liste (`length`), erleichtert werden.

> Frage 11:
>
> Ist es sinnvoll, den Verwalter einer einfach verketteten Liste um einen Zeiger `Last` auf das letzte Element zu erweitert? Welche Vorteile hat demgegenüber eine doppelt verkettete Liste?

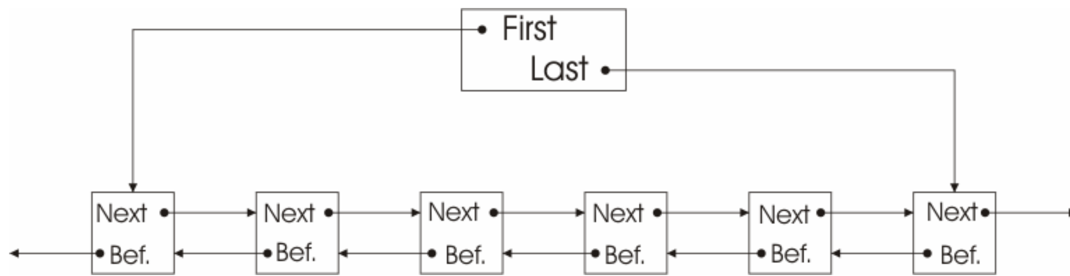

Abbildung 11: Doppelt verkettete Liste

Sowohl die Liste selbst als auch ihre Elemente sind Objekte, die zur Laufzeit des Programms mit Hilfe des `new`-Operators angelegt werden. Objekte, die nicht mehr benötigt werden, sollten unbedingt gelöscht (`delete`-Operator) oder anderweitig aus dem Verkehr gezogen werden. Der Zugriff auf Objekte erfolgt mittels Zeiger. Der Programmierer ist immer selbst dafür verantwortlich, dass die Verbindung zu einem Objekt nicht verloren geht. Vor der Vernichtung eines Objektes sollte zudem immer geprüft werden, ob noch andere Referenzen auf das Objekt gerichtet sind. Andernfalls kann es im späteren Programmablauf zu den gefürchteten „Ausnahmefehlern" kommen, falls versucht wird, auf nicht mehr vorhandene Objekte zuzugreifen. Im Simulationssystem ***ROSI*** werden die Objekte im Regelfall nicht vernichtet, sondern in einer so genannten Freispeicherliste (die oben erwähnte Halde) gesammelt, aus der sie bei Bedarf wieder recycelt werden können. Ausführlichere Informationen finden Sie u.a. in /3/ und /4/.

Frage 12:

Überlegen Sie sich, welche Vor- und Nachteile eine Freispeicherliste besitzt!

3.2 Das Simulationssystem ROSI

Die Programmarchitektur des Simulationssystems *ROSI* folgt einer Schalenstruktur (Abbildung 12), bestehend aus einem inneren Kern und einer äußeren Schale, der graphischen Oberfläche (GUI = Graphical User Interface). Zwischen diesen beiden Teilen des Programms vermittelt eine Kommando-Shell, die auch mittels Konsole manuell bedient werden kann (*Ansicht* → *Konsole*). Der eigentliche Zweck der Kommando-Shell besteht jedoch darin, mit anderen Programmen, z.B. Produktionssteuerungssystemen oder Datenbanken, zu kommunizieren und Daten auszutauschen. Dies geschieht dann natürlich ohne manuelle Eingriffe und ohne dass die Konsole geöffnet werden muss.

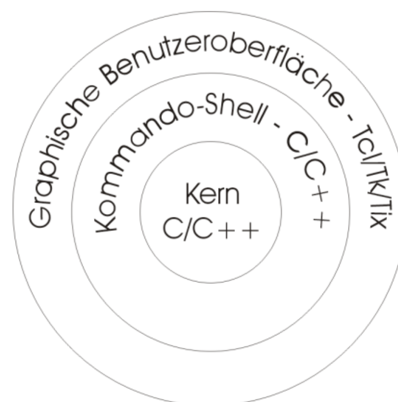

Abbildung 12: Architektur des Simulationssystems ***ROSI***

Das Optimierungssystem wird zwar aus dem Menü des Simulationssystems heraus gestartet (*Optimierung* → *Optimierung*), ist aber eigentlich eine eigenständige Anwendung. Der Austausch der Einflussgrößen (vom Optimierungssystem zum Simulationssystem) und der Zielgrößen (vom Simulationssystem zum Optimierungssystem) läuft ausschließlich über die Kommando-Shell (siehe Abbildung 7). Beide Programme sind somit völlig unabhängig voneinander, sie müssen lediglich über einen gemeinsamen Befehlsvorrat verfügen.

Der Kern ist in der Programmiersprache C++ programmiert. Er enthält alle für die Simulation diskreter Fertigungsprozess notwendigen Bausteine wie Maschinen, Warteschlangen, Jobs oder Technologien in Form von Klassen. Die Verwaltung der Objekte und die Ablaufsteuerung erfolgt ebenfalls im Kern. Jeder Objektklasse wird in der Regel ein eigenes Modul zugeordnet, das aus einer (öffentlichen) Header-Datei und der eigentlichen (nicht öffentlichen) Quell-Datei besteht. Die Objektklasse `CQueue*` (synonym mit `QUEUE`) wird z.B. in der Datei `queue.cpp` definiert. Die zugehörige Header-Datei ist `queue.h`. Zusätzlich enthält der Kern natürlich noch eine Vielzahl von untergeordneten Objekten, wie etwa Listen oder Listenelemente, die von anderen Objekten benötigt werden, aber nicht an der Oberfläche des Simulators in Erscheinung treten. Der Kern enthält bereits das vollständige Simulationssystem, allerdings müsste man, um den Simulator in dieser Form verwenden zu können, ein eigens C/C++-Programm schreiben, um den Simulator anwenden zu können.

Die Funktionen des Datenaustauschs und der Steuerung übernimmt die Kommando-Shell. Für die Kommando-Shell wird ein Tcl-Interpreter genutzt, der um einen eigenen simulationsspezifischen Befehlssatz erweitert wurde /5/. Tcl (Tool command language) ist eine Skriptsprache, die im Wesentlichen mit Zeichenketten operiert. Sie ist vergleichbar mit anderen bekannten Skriptsprachen, wie etwa Pearl, Python oder Java-Skript. Auch die verschiedenen Unix-Shells und die Batch-Skripte unter Windows sind typische Skriptsprachen. Der Tcl-Interpreter selbst ist aus Portabilitätsgründen in C programmiert, unter Einhaltung wesentlicher Prinzipien des objektorientierten Programmierens. Die Quellen sind frei verfügbar (aktuellere Informationen findet man darüber hinaus auch zahlreich im Internet, siehe z.B. /6/).

Während im Kern jedes Objekt letztlich eindeutig durch einen Zeiger referenziert wird, geschieht das in der Kommando-Shell durch eine eindeutige Zeichenkette – dem Objekt-Bezeichner. In der Regel setzt sich der Objektbezeichner aus dem Objekttyp und einer (fortlaufenden) Nummerierung zusammen. Für jedes Objekt wird automatisch ein vollständiger objektspezifischer Befehlssatz angelegt, über den man Parameter des Objekts abfragen oder verändern kann. So lässt sich die Länge der Warteschlange mit dem Objekt-Bezeichner `queue1` mit Hilfe des Befehls `queue1 length` abfragen. Der gleiche Befehl existiert auch für alle anderen Warteschlangen, z.B. `queue2 length`. Der Name eines Jobs z.B. kann mit dem Befehl `job5 name` abgefragt oder mit dem Befehl `job5 name „LP 1"` in `LP 1` geändert werden. Den vollständigen Befehlssatz findet man in der Online-Hilfe zum Simulationssystem *ROSI* (siehe /2/).

Fester Bestandteil von Tcl ist das so genannte Toolkit (Tk = Tool kit), weshalb man auch meist von Tcl/Tk spricht. Das Toolkit ist ein Bausteinkasten für die Entwicklung plattformunabhängiger graphischer Oberflächen, für das es auch zahlreiche nützliche Erweiterungen gibt (z.B. Tix). Dieser Bausteinkasten wurde genutzt, um die graphische Oberfläche des Simulationssystems zu entwickeln. D.h., die gesamte GUI des Simulators wurde in der Skriptsprache Tcl geschrieben. Diese Tatsache hat aber für den Praktikumsversuch keine weitere Bedeutung. Wichtig ist, dass zwischen den drei Schichten des Simulationssystems ein enger Zusammenhang besteht. Ein einmal angelegtes Objekt existiert real nur im Kern, wird aber, wenn gewünscht, in allen Schichten dargestellt. In der Kommando-Shell wird das Objekt mittels eindeutiger Zeichenkette und auf der graphischen Oberfläche mittels Bausteinsymbol referenziert. Objekte können sowohl auf der graphischen Oberfläche (*Doppelklick → Dialogbox*) als auch in der Kommando-Shell angelegt, gelöscht oder verändert werden. Die Informationen werden stets an die anderen Schichten weitergereicht (siehe Abbildung 13). Der Simulator kann übrigens auch ohne graphische Oberfläche gestartet werden und ist in diesem Modus voll funktionsfähig. *ROSI* kann dadurch in übergeordnete Programmsysteme eingebettet und auch komplett durch diese gesteuert werden – eine wichtige Voraussetzung für den industriellen Einsatz.

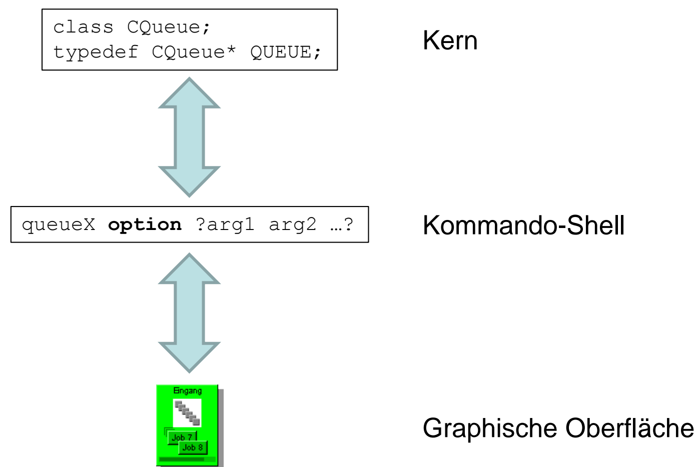

Abbildung 13: Objektdarstellung in den verschiedenen Schalen: Zeiger, Zeichenkette, graphisches Symbol

3.3 Warteschlangen im Simulationssystem

Warteschlangen werden im Simulationssystem *ROSI* durch doppelt verkettete Listen realisiert. Dabei beinhaltet die Klasse `QUEUE` (Modul `queue.cpp/h`) lediglich die reine Listenverwaltung, nicht jedoch die Listenelemente selbst. Die Klasse `JOB` (Modul `job.cpp/h`) definiert auch die Zeiger `Before` und `Next` (beide vom Typ `JOB`), so dass Jobs direkt miteinander verknüpft werden können. Die Job-Objekte übernehmen damit zugleich die Funktion der Listenelemente. Abbildung 14 zeigt den prinzipiellen Aufbau einer Warteschlange, wobei nur die für die Verknüpfungen wesentlichen Bestandteile der Objekte dargestellt sind.

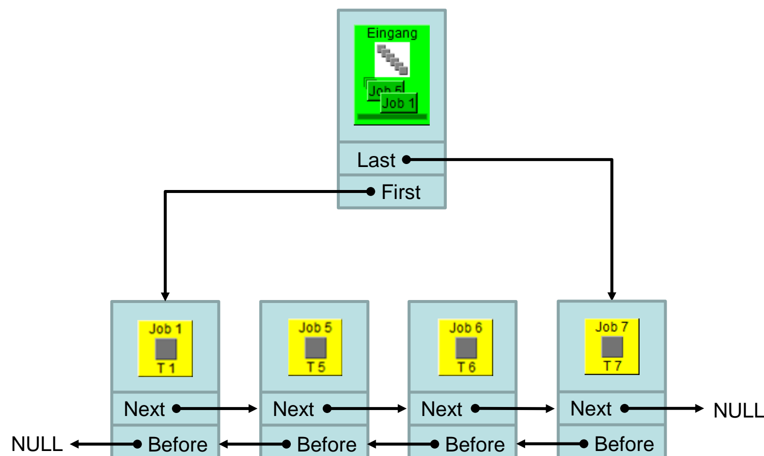

Abbildung 14: Warteschlange mit direkt verknüpften Jobs

Das Vertauschen von Jobs ist eine Funktion der Listenverwaltung, gehört also in das Modul `queue.cpp/h`. Da die Klassen aber weitgehend gekapselt sind (beachten Sie die Einteilung in `private` und `public`), kann man aus der Klasse `QUEUE` nicht direkt auf die Zeiger `Before` und `Next` (Klasse `JOB`) zugreifen. Dafür werden von der Klasse `JOB` die entsprechenden `Put`- und `Get`-Methoden bereitgestellt (siehe Anhang 3, Auszug aus Datei `job.h`). Um einen neuen Job an die Spitze der Warteschlange zu stellen, kann man also nicht schreiben `NewJob->Next=First`, sondern:

```
NewJob->PutNext(First)
```

Frage 13:
Welche Vorteile bringt die Kapselung von Klassen?

Achtung, der Listenindex im Simulationssystem *ROSI* startet stets bei 1! Der Ausdruck `GetJob(1)` gibt also den ersten Job aus der Warteschlange zurück. In C/C++ beginnt der Listenindex dagegen immer bei 0.

Um nicht die gesamte Datei `queue.cpp` veröffentlichen zu müssen, wurde der für das Praktikum relevante Teil in die Datei `queuex.cpp` ausgelagert (siehe Anhang 4). Für die Programmierung der beiden Varianten der Methode `ExchJobs` können alle veröffentlichten Methoden, insbesondere aus dem Modul `job.h`, verwendet werden. Es ist ratsam, sich zunächst die bereitgestellten Methoden anzusehen, bevor man eine eigene programmiert, Sie sparen so Aufwand und vermeiden zusätzliche Fehlerquellen.

4 Aufgabenstellung

Die Aufgabenstellung besteht aus zwei Teilen:

1. Programmieren und testen Sie die Methode `ExchJob`. Es sind beide Varianten, sowohl die objektbasierte als auch die indexbasierte Variante, zu implementieren. Dabei ist es ratsam, mit der objektbasierten Methode zu beginnen. Die Eingangsparameter lassen sich später in Indices umwandeln, so dass die indexbasierte Methode nicht von Grund auf neu entworfen werden muss.
2. Führen Sie Optimierungsexperimente mit einem Flow Shop und einem Job Shop durch. Dokumentieren Sie die Ergebnisse der Optimierungsläufe (optimale Reihenfolge und Zykluszeit). Die Optimierung setzt das korrekte Funktionieren der indexbasierten Methode `ExchJob` voraus.

Hinweise zur Versuchsdurchführung:

Unter „`Eigene Dateien`" finden Sie das Unterverzeichnis `ROSI`. In diesem Verzeichnis sind alle Dateien enthalten, die Sie für das Praktikum benötigen. Im Einzelnen enthält das Verzeichnis `ROSI` folgende Unterverzeichnisse:

> `.\bin` – ausführbare Dateien Bibliotheken (`*.exe`, `*.dll`)
> `.\dat` – Simulationsmodelle (`*.mc`, ...)
> `.\lib` – Tcl-Dateien für die graphische Oberfläche u.a.
> `.\src` – Quell- und Objektdateien (`*.cpp`, `*.h`, `*.obj`)

Die Datei `.\src\queuex.cpp` ist der öffentliche Teil des Moduls `queue.cpp` (Beschreibung der Klasse `CQueue` alias `QUEUE`) und enthält die beiden Methoden für die Vertauschung von Jobs in der Warteschlange, jedoch mit leerem Prozedurkörper.

Starten Sie das Microsoft Visual C++ Studio und laden Sie anschließend das Projekt `ct_prakt.dsw` aus dem Verzeichnis `ROSI\src` (*Datei* → *Arbeitsbereich öffnen*). Der Editor des Studios zeigt die Datei `queuex.cpp`. Kopieren Sie nun Ihren Programmtext für die Methoden an die dafür vorgesehenen Stellen (evtl. Texteditor verwenden). Versuchen Sie <u>nicht</u>, Ihre eigene Datei in das Projekt einzubinden!

Starten Sie die Übersetzung (Empfehlung: *Erstellen* → *Alles neu erstellen*). Im Nachrichten-Fenster werden die einzelnen Schritte des Vorgangs angezeigt. Das Ergebnis der Projekterstellung ist eine neue Datei `ROSI\bin\rosi25.dll`, die die alte Datei überschreibt. Ein erfolgreich abgeschlossenes Projekt muss mit folgender Zeile im Nachrichten-Fenster enden:

```
rosi25.dll  -  0 Fehler, 0 Warnung(en)
```

<u>Wichtiger Hinweis</u>: Das Simulationsprogramm *ROSI* muss vor der Projekterstellung geschlossen werden, da sonst `rosi25.dll` nicht überschrieben werden kann (Fehlermeldung im Nachrichten-Fenster des Studios beachten!).

Sind keine syntaktischen Fehler mehr vorhanden, können Sie Ihr Programm testen. Starten Sie zunächst das Simulationsprogramm *ROSI* und laden das Modell `flowshop.mc` aus dem Verzeichnis `ROSI\dat`. In der Eingangswarteschlange dieses Modells befinden sich bereits 9 Jobs (`job1`, `job2`, ... `job9`), die Sie mit Hilfe Ihrer Methoden tauschen können. Der Test muss mittels Interpreter erfolgen, da die Operation über die graphische Oberfläche die Exchange-Methoden umgeht. Für den Interpreter öffnen Sie eine Konsole (*Ansicht* → *Konsole*). Sie können nun alle Objekte des Simulationsmodells über deren Objektbezeichner ansprechen. Den Objektbezeichner finden Sie im oberen Rahmen des Dialogfensters. Dort steht der Typ des Objekts, gefolgt von dessen Objektbezeichner. Für das Beispiel in Abbildung 15 ist der Objektbezeichner für die Warteschlange mit dem Namen „Eingang" die Zeichenkette `queue3`.

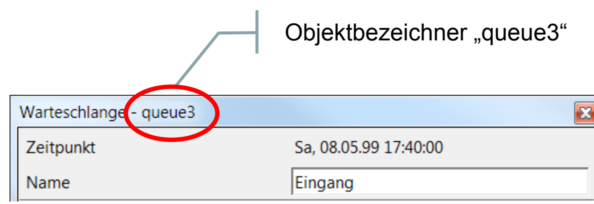

Abbildung 15: Dialogfenster der Warteschlange mit Objektbezeichner

Den vollständigen Befehlssatz für alle Objekte enthält die Online-Hilfe des Simulationsprogramms /2/. Folgende Befehle sind wichtig für das Praktikum:

`queue3 length`
gibt die Anzahl der Jobs zurück

`queue3 job`
gibt die Liste der wartenden Jobs zurück

`queue3 exchange 2 5`
tauscht den 2. mit dem 5. Job (Zählung beginnt bei 1)

`queue3 job exchange job2 job5`
tauscht die Objekte `job2` mit `job5`

Für die Fehlersuche existiert ein nicht dokumentierter Befehl, mit dem man die Zeigerbeziehungen in einer Warteschlange auf einfache Weise sichtbar machen kann (selbstverständlich darf auch der Debugger des Studios genutzt werden).

```
queue3 list

First -> job1
NULL <- job1 -> job2
job1 <- job2 -> job3
job2 <- job3 -> job4
job3 <- job4 -> job5
job4 <- job5 -> job6
job5 <- job6 -> job7
job6 <- job7 -> job8
job7 <- job8 -> job9
job8 <- job9 -> NULL
Last -> job9
```

Testen Sie sowohl die index- als auch die objektorientierte Methode gründlich. Berücksichtigen Sie dabei auch die kritischen Fälle wie unmittelbare Nachbarn und ersten und/oder letzten Job. Wenn keine Fehler mehr auftreten, könne Sie mit der Optimierung beginnen. Dabei werden, wie bereits weiter oben beschrieben, sehr viele Vertauschungen ausgeführt. Sollte Ihr Programm trotz erfolgreich verlaufenem Test noch Fehler enthalten, zeigen sie sich spätestens hier. Häufig auftretende Fehler sind:

1. Das Programm reagiert nicht mehr
2. Die optimale Zykluszeit ist zu kurz

Ursache für Fehler 1 könnte ein „verlorener“ Zeiger sein. Überprüfen Sie, ob immer alle Zeiger korrekt gesetzt werden. Fehler 2 entsteht, wenn die Liste abgeschnitten wird, z.B. durch fehlerhaften oder vergessenen `Last`-Zeiger.

Hinweise zur Abfassung des Protokolls:

Dokumentieren Sie ausführlich Ihren Programmtext (Quellcode). Bedienen Sie sich dabei nach Möglichkeit auch der üblichen Werkzeuge (Flussdiagramm, Struktugramm, …) oder anderer anschaulicher Mittel. Gehen Sie auf Fehler in Ihrem Programm ein. Was könnte man verbessern?

Werten Sie die beiden Optimierungsergebnisse aus. Welche minimale Zykluszeit wurde jeweils ermittelt und welche Jobreihenfolge gehört dazu? Die Verwendung von Screen-Shots ist nicht zwingend, erleichtert aber oft die Erklärung der Ergebnisse.

Gehen Sie bei der Abfassung des Protokolls auch auf die in der Praktikumsanleitung gestellten Fragen/Aufgaben ein (gerahmte Textstellen).

5 Literaturhinweise

/1/ Domschke, W.; Scholl, A.; Voß, S.: ***Produktionsplanung: Ablauforganisatorische Aspekte***. Springer, 1997.

/2/ Weigert, G.; Werner, S.: ***ROSI-Online-Hilfe***. Dresden, 2001 http://www.avt.et.tu-dresden.de/rosi/hlp_rosi/contents.htm.

/3/ Achtert, W.: ***Das große Buch zu C++***. Data Becker, Düsseldorf, 1995.

/4/ Willms, G.: ***C++ - Das Grundlagenbuch***. Data Becker, Düsseldorf, 2001

/5/ Ousterhout, J.K.: ***Tcl und Tk***. Addison-Wesley, 1995.

/6/ ***Tcl Developer Xchange*** http://www.tcl.tk/

6 Arbeits- und Brandschutzhinweise

Vorbeugende Maßnahmen:

- Die Praktikumsteilnehmer haben sich so zu verhalten, dass Gefahrensituationen und Unfälle vermieden werden.
- Die Befugnis zum Bedienen und Nutzen von Geräten ist auf den zugewiesenen Praktikumsplatz beschränkt.
- Eingriffe in die zum Praktikumsaufbau gehörenden Geräte sind nicht erlaubt.
- Der Anschluss und der Betrieb privater Geräte in den Praktikumsräumen sind verboten.
- Defekte an Geräten oder Gebäudeeinrichtungen sind unverzüglich dem Betreuer mitzuteilen. Betroffene Geräte sind außer Betrieb zu nehmen. Andere Personen sind vor Gefahren zu warnen.
- Den Anweisungen der Praktikumsbetreuer bzw. anderer aufsichtsführender Personen ist unbedingt Folge zu leisten.
- Betriebsfremde dürfen sich nicht in den Praktikumsräumen aufhalten.
- Rauchen und Umgang mit offenem Feuer ist nicht gestattet.
- Das Einnehmen von Speisen und Getränken im Computerkabinett ist nicht gestattet.
- Nach Ende des Praktikums ist der Arbeitsplatz sauber und aufgeräumt zu hinterlassen.
- Außergewöhnliche Ereignisse bzw. besondere Vorkommnisse sind umgehend dem Betreuer oder den aufsichtsführenden Personen zu melden.

Verhalten im Falle eines Brandes:

- Beachten der richtigen Reihenfolge:
  MELDEN – RETTEN – LÖSCHEN
- FEUER MELDEN

  - Telefonische Brandmeldung
  - Deutliche, genaue und vollständige Angaben:
    - **Wo** brennt es?
    - **Was** brennt?
    - Angaben zu verletzten oder gefährdeten **Personen.**
    - **Wer** meldet?
- PERSONEN RETTEN

  - Erste Hilfe leisten
  - Weitere Hilfe organisieren, medizinische Hilfe anfordern
  - Gefahrenbereich räumen; Fluchtwege benutzen, keine Aufzüge
  - Andere Personen warnen, Sammelplatz (Platz vor Turmeingang zum Barkhausenbau) aufsuchen.
  - Behinderten und älteren Personen helfen.
- LÖSCHVERSUCH UNTERNEHMEN

  - Feuerlöscher verwenden (Standorte: Gänge des Barkhausenbaues), dabei nicht selbst gefährden.
  - Fenster und Türen schließen, aber nicht abschließen.
  - Möglichst elektrische Verbraucher abschalten.

Rufnummern für Notfälle:

Rettungsdienst:	**112**
Polizei:	**110**
TUD-Notruf:	**HA 34515**
Betriebsärztin Dr. Römer:	**HA 36255**

7 Anhänge

- **Anhang 1**
  Definition der Klasse `CQueue` alias `QUEUE` (Datei `queue.h`)
- **Anhang 2**
  Beispielmethoden der Klasse `CQueue` (Auszug aus der Datei `queue.cpp`)
- **Anhang 3**
  Definition der Klasse `CJob` alias `JOB` (Auszug aus der Datei `job.h`)
- **Anhang 4**
  Datei `queuex.cpp`
  Enthält nur die beiden Methoden `ExchJob` der Klasse `CQueue`.
  Diese Methoden sind im Praktikum zu implementieren!

Anhang 1, Datei queue.h

```
//------------------------------------------------------------------
// ROSI, class CQueue
//------------------------------------------------------------------

#ifndef  QUEUE_H
#define  QUEUE_H

#include  "def.h"
#include  "job.h"
#include  "stoch.h"
#include  "order.h"
#include  "shift.h"

class CQueue;
typedef CQueue * QUEUE;

extern CAdmin *QueueAdmin;      // Verwalter der Queue-Objekte

// Create-defaults
#define  QUEUE_Input       pctrl
#define  QUEUE_Output      all
#define  QUEUE_Space       -1
#define  QUEUE_Create      -1
#define  QUEUE_Rest        NullTime
#define  QUEUE_Time        NullTime
#define  QUEUE_MaxTime     InfTime
#define  QUEUE_Kill        False
#define  QUEUE_STContr     True
#define  QUEUE_Power       1     // eingeschaltet
#define  QUEUE_MinLoad     -1    // min. Ladung, neue
                                 // Warteschlange!

#define  QUEUE_MaxLoad     -1    // max. Ladung, neue
                                 // Warteschlange!

#define  QUEUE_Integral    0.0   // Zeit x Ladung

extern void QueueRecalcLoad();
extern void QueueSynchronizePlan();

class CQueue:   public CObject
{
private:

     // In der Warteschlange befindliche Jobs als doppelt
     // verkettete Liste
     // -----------------------------------------------------
     JOB            First;          // erster Job
     JOB            Last;           // letzter Job

     PERMUT         Permut;         // Permutation der Jobs

     ADJUST         AdjustSpace;    // Steuerung der
                                    // Kapazitaet
```

```
      ADJUST          AdjustInput;    // Steuerung des Input
                                      // Operators

      InputStrategy   Input;          // Input-Strategie

      OutputStrategy  Output;         // Output-Strategie
                                      // all
                                      // first

      JOB        Sample;              // Musterjob

      TIME       RestCreate;          // Restzeit bis zur
                                      // Erzeugung

      STOCHASTIC Stoch;               // Stochastik fuer
                                      // Restzeit

      PLAN       Plan;                // Zeitplan fuer Queue
      long       Create;              // Erzeugung von Jobs
      bln        Kill;                // Vernichtung von Jobs

      bln        STContr;             // Ueberwachung der
                                      // Liegezeit

      int        Power;               // eingeschaltet
                                      // (Schichtobjekt)

      // Belegungszustand der Queue, kann nur abgefragt
      // werden!
      // -----------------------------------------------------
      long BusyLength;
      long ReadyLength;
      long Busy;
      long Ready;

public:
      COffer               Offer;     // Logistisches
                                      // Angebot der
                                      // Queue

      CStationMonitor Monitor;        // Logistische
                                      // Ueberwachung der
                                      // Queue

public:
      CQueue();
      ~CQueue();
      void CleanUp();

      // Gibt 1 zurueck, wenn Jobs vernichtet wurden, sonst 0
      // -----------------------------------------------------
      int        Enter (JOB);

      int        CheckLeave (JOB);    // Kann der Job die
                                      // Queue verlassen?

      JOB        Leave (JOB);
```

```
JOB        CreateJob();
int        Go (TIME);
void       RecalcLoad();        // nach Zeitspruengen

// Link stellt eine Verbindung zwischen einem Job
// und der durch den Pass-Zeiger des Jobs bezeichneten
// Queue her. Im Unterschied zu Enter werden keine
// Zustandsgroessen des Jobs oder der Queue veraendert.
// Link gibt 0 zurueck, wenn die Verbindung gelungen
// ist.
// -------------------------------------------------------
int   Link (JOB);

// Loest eine (bestehende) Verknuepfung zwischen einem
// Job und einer Queue. Die Technologie und die
// Arbeitsgangzeiger sowie der Zustand der Maschine
// werden dabei nicht veraendert.
// -------------------------------------------------------
JOB   Unlink (JOB);

// Entfernt alle evtl. vorhandenen Jobs aus der Queue.
// -------------------------------------------------------
int Clear();

// Hat die Queue ausreichend Platz (Space), um diesen
// Job bereitzustellen und ist weder ausgefallen noch
// ausgeschaltet?
// 1: ja, 0: nein
// -------------------------------------------------------
int CheckSupply (JOB);

// Hat die Queue ausreichend Platz (Space), um diesen
// Job aufzunehmen und ist weder ausgefallen noch
// ausgeschaltet?
// 1: ja, 0: nein
// -------------------------------------------------------
int CheckEnter (JOB);

// Hat die Queue ausreichend Platz (Space), um diesen
// Job aufzunehmen?
// 1: ja, 0: nein
// -------------------------------------------------------
int CheckSpace (JOB);

// Die Queue stellt den Job bereit oder nimmt ihn
// zurueck. Die Belegung der Queue wird dabei geaendert
// ROSI_OK: ausgefuehrt
// sonst: nicht ausgefuehrt
// -------------------------------------------------------
int SupplyJob (JOB);
int UnSupplyJob (JOB JPtr);
```

```
// Gibt den naechsten wartenden Job zurueck.
// Eingabeparameter = NULL: erster wartender Job.
// Wenn kein wartender Job mehr folgt, wird NULL
// zurueckgegeben. Wenn der Ausgabeoperator nur den
// ersten Job beruecksichtigt, folgen auf den ersten
// Job (First) keine weiteren wartenden Jobs mehr
// (NULL).
// -------------------------------------------------------
JOB NextWaitingJob (JOB);

// Ordnet gleichzeitig die Jobs in der Queue um!
// -------------------------------------------------------
int   PutPermut (PERMUT);
int   PutAdjustSpace (ADJUST);
int   PutAdjustInput (ADJUST);

// Eine evtl. mit der Queue verknuepftes Permut-Objekt
// wird zunaechst auf Laenge 0 gesetzt
// (order:clear_order) und anschliessend durch Anfuegen
// von Index-Elementen (order:append_order) auf die
// aktuelle Laenge der Warteschlange verlaengert.
// -------------------------------------------------------
int   ResetPermut();
int   MakePermut();

// Ordnen der Jobs nach Einlagerungsstrategie
//--------------------------------------------------------
int   SortJobs(InputStrategy);
int   InverseJobs();

// Methoden zur Parametereinstellung
// -------------------------------------------------------
int   PutCreate (long);
int   PutSample (JOB Sample);
int   PutRestCreate (TIME Rest);
int   PutStoch (STOCHASTIC Stoch);
int   PutKill (bln);
int   PutStcontr (bln);
int   PutInput (InputStrategy);
int   PutOutput (OutputStrategy);
int   PutPowerState (int);
int   PutPlanObj (PLAN);

// Methoden zur Parameterabfrage
// -------------------------------------------------------
PERMUT          GetPermut();
ADJUST          GetAdjustSpace();
ADJUST          GetAdjustInput();
long            GetCreate();
JOB             GetSample();
TIME            GetRestCreate();
STOCHASTIC      GetStoch();
bln             GetKill();
bln             GetStcontr();
OutputStrategy  GetOutput();
InputStrategy   GetInput();
int             IsOff();
```

```
    PLAN            GetPlanObj ();
    TIME            GetRestPower ();
    State           GetState();
    BasicState      GetBasicState();
    long            GetPowerState ();
    long            GetDownState ();
    long            GetBusyState ();
    long            GetReadyState ();

    // Die folgenden Methoden dienen der Abfrage von
    // Zustandsgroessen, die keinen eigenen Parameter
    // besitzen. Es gibt daher keine zugehoerige
    // Put-Methode.
    // -----------------------------------------------------
    long GetLoad();
    long GetLength(State = nostate);
    long GetAvailable(int *infinty);
    JOB  FirstJob();
    JOB  LastJob();
    JOB  GetJob (long No);     // no-ter Job aus der
                               // Warteliste

    long Full();               // 1 - voll, sonst 0
    long Empty();              // 1 - leer, sonst 0


    // Diese Methoden sind im Praktikum zu erstellen!
    // Modul: queuex.cpp
    // -----------------------------------------------------
    int  ExchJobs (long JobAIx, long JobBIx);
    int  ExchJobs (JOB JobA, JOB JobB);

};
#endif   // QUEUE_H
```

Anhang 2, Auszug aus der Datei `queue.cpp`

```
//----------------------------------------------------------------
// Gibt Anzahl der Jobs zurueck
//----------------------------------------------------------------
long  CQueue::GetLength (State state)
{
     if (!this)            return 0;  // keine Queue: 0
     if (state == mready)  return ReadyLength;
     if (state == busy)    return BusyLength;
                           return     ReadyLength + BusyLength;
}

//----------------------------------------------------------------
int  CQueue::Link (JOB JPtr)
{
     long Size;
     int inf;

     if (!this)                 return  ROSI_ERROR;
     if (!JPtr)                 return  ROSI_NOP;

     if (!First) {
          First = JPtr;
     }
     else {
          Last->PutNext (JPtr);
          JPtr->PutBefore (Last);
     }
     Last = JPtr;
     JPtr->PutStation ((CObject *)this);
     Size = JPtr->GetSize(&inf);
     ReadyLength++;
     JPtr->TurnReady(True);

     // nur bereitgestellte Jobs belegen Platz
     if (JPtr->IsSupplied()) {
          Ready = Ready + Size;
     }
     JPtr->IncRef();

                                return  ROSI_OK;
}
```

Anhang 3, Auszug aus der Datei `job.h`

```
//--------------------------------------------------------------
// ROSI, class CJob (Auszug)
//--------------------------------------------------------------
#ifndef  JOB_H
#define  JOB_H

class CJob;
typedef CJob* JOB;
extern CAdmin* JobAdmin;   // Verwalter der Job-Objekte

class CJob:     public CObject
{
private:
     . . .
     . . .
     . . .
     JOB  BeforeJob;       // Verwendung in Queue
     JOB  NextJob;         // Verwendung in Queue
     . . .
     . . .
     . . .
public:
     CJob ();
     ~CJob ();
     . . .
     . . .
     . . .

// Gibt naechsten Job in der Liste zurueck
//--------------------------------------------------------------
     JOB    GetNext ();
// Gibt vorhergehenden Job in der Liste zurueck
//--------------------------------------------------------------
     JOB    GetBefore ();

// Verkettung mit naechstem Job
//--------------------------------------------------------------
     int    PutNext (JOB);

// Verkettung mit vorhergehenden Job
//--------------------------------------------------------------
     int    PutBefore (JOB);

     . . .
     . . .
     . . .

};
#endif   // JOB_H
```

Anhang 4, Datei queuex.cpp

```
 #include  "queue.h"

//--------------------------------------------------------------
// Vertauscht den Job A der Warteschlange mit dem Job B
// Rueckgabewert:
// erfolgreich ausgefuehrte Operation:     ROSI_OK
// keine Operation ausgefuehrt:            ROSI_NOP
// Fehler, keine Operation ausgefuehrt:    ROSI_ERROR
//--------------------------------------------------------------
int CQueue::ExchJobs (JOB A, JOB B)
{
     // Hier Programmtext einfügen!
}

//--------------------------------------------------------------
// Vertauscht den Job mit Listenindex Ax (= 1, 2, 3, . . .) mit
// dem Job mit Listenindex Bx (= 1, 2, 3, . . .).
// Rueckgabewert:
// erfolgreich ausgefuehrte Operation:     ROSI_OK
// keine Operation ausgefuehrt:      ROSI_NOP
// Fehler, keine Operation ausgefuehrt:    ROSI_ERROR
//--------------------------------------------------------------
int CQueue::ExchJobs (long Ax, long Bx)
{
     // Hier Programmtext einfügen!
}
```