kassel
university
press

Vom Zeitmechanismus bis zur Anlagensteuerung

Objektorientierte Modellierungsansätze für die Materialflußsimulation

Matthias Wiegand

Dissertation zur Erlangung des akademischen Grades eines Doktor-Ingenieurs (Dr.-Ing.) am Fachgebiet Produktionssysteme im Institut für Produktionstechnik und Logistik der Universität Gesamthochschule Kassel.

Die Deutsche Bibliothek - CIP-Einheitsaufnahme
Wiegand, Matthias
Vom Zeitmechanismus bis zur Anlagensteuerung : Objektorientierte Modellierungsansätze für die Materialflusssimulation / Matthias Wiegand. - Kassel : kassel univ. press, 2000. - 271 S. : Ill.

Zugl.: Kassel, Univ., Diss. 1999
ISBN 3-933146-50-X

Umschlaggestaltung: 5 Büro für Gestaltung, Kassel
Druck und Verarbeitung: Zentraldruckerei der Universität Gesamthochschule Kassel
Printed in Germany

Vorwort zur veröffentlichten Fassung

Mit dem vorliegenden Werk publiziere ich meine am Fachgebiet Produktionssysteme im Institut für Produktionstechnik und Logistik entstandene Dissertation.

Da die Suche nach einem geeigneten Partner für die Veröffentlichung doch langwieriger war als ich gedacht hatte, freue ich mich umso mehr, diesen mit dem Verlag *Kassel University Press* nunmehr gefunden zu haben.

An dieser Stelle möchte ich mich daher ausdrücklich bei Frau B. Bergner (Geschäftsführerin), Herrn H. Begemann (Technischer Support) sowie auch bei allen anderen Mitarbeiterinnen und Mitarbeitern des Verlags, die zur Veröffentlichung des vorliegenden Werkes beigetragen haben bedanken.

Kassel, im November 2000

Matthias Wiegand

Vorwort

Im Rahmen meiner Tätigkeit als Wissenschaftlicher Mitarbeiter am Fachgebiet Produktionssysteme im Institut für Produktionstechnik und Logistik entstand die vorliegende Arbeit, die ich einer Prüfungskommission des Fachbereichs Maschinenbau der Universität Gesamthochschule Kassel im Rahmen meiner Promotion als Dissertation vorlege.

Für die Bereitschaft zur Übernahme der Begutachtung möchte ich mich an dieser Stelle bei den Herren Univ.-Prof. Dipl.-Ing A. Reinhardt und Univ.-Prof. Dr.-Ing. F. Tikal bedanken. Mein besonderer Dank gilt dabei Herrn Univ.-Prof. Dipl.-Ing A. Reinhardt für die Betreuung der Arbeit.

Ausdrücklich und lobend möchte ich auch die übrigen Mitarbeiter des Fachgebiets erwähnen, die große Projekte ebenso wie kleine Alltagsgeschäfte gemeinsam mit mir nicht nur engagiert und kooperativ bearbeiteten, sondern auch in vielen Gesprächen mit Kritik und Anregungen zum Gelingen dieser Arbeit beitrugen.

Last, but not least danke ich auch meiner Familie für die große Geduld und Unterstützung mit der sie mir in der Zeit der Entstehung dieser Arbeit zur Seite stand.

Kassel, im Juni 1999

Matthias Wiegand

Zusammenfassung

Die Simulation hat sich seit ihren Anfängen in den 50er Jahren bis heute zu einem anerkannten und regelmäßig genutzten Hilfsmittel bei Planung und Betrieb von Materialflußsystemen entwickelt.

Die Erfahrungen des Autors aus der Durchführung von Simulationsstudien zeigen, daß sich die Anforderungen an solche Untersuchungen und die bei ihrere Erarbeitung eingesetzten Werkzeuge in der letzten Zeit deutlich verändern: Die betrachteten Anlagen werden zunehmend komplexer, die geforderte Ergebnisqualität nimmt zu, die Untersuchungen sind in kürzerer Zeit durchzuführen und es besteht ein Trend zur verbesserten Integration mit anderen Verfahren und Werkzeugen.

Die Bewahrung und eventuelle Vergrößerung der heutigen Bedeutung der Materialflußsimulation bedingen daher die Weiterentwicklung der einschlägigen Werkzeuge. Es ist das Ziel dieser Arbeit, neue und modifizierte Konzepte für die Materialflußsimulation zu entwickeln und vorzustellen, mit denen den gestiegenen Anforderungen begegnet werden kann.

Offensichtlich erfordert dies zunächst eine Analyse ebendieser Anforderungen. Wegen der zunehmend anzutreffenden ganzheitlichen Ansätze bei Planung und Betrieb materialflußtechnischer Systeme wird dabei das Umfeld der Simulation einbezogen, da ein zu eng gewählter Fokus implizieren würde, daß abgeleitete Konzepte nicht über Detailverbesserungen hinausgehen könnten.

Die Arbeit beginnt daher mit einem Abschnitt, der sich mit der allgemeinen Situation produzierender Unternehmen beschäftigt, da in diesem Bereich wohl die meisten materialfluß-technischen Systeme betrieben werden. Um den Umfang der Ausführungen zu begrenzen, werden Unternehmen aus anderen Bereichen wie z.B. Handel und Dienstleistungen nicht behandelt. Zur Einführung wird zunächst ein Abriß der Entwicklung der industriellen Produktion gegeben. Daran anschließend werden die heutigen Verhältnisse auf den Absatzmärkten und die daraus resultierenden Anforderungen an die Produkte dargestellt. Das letzte Kapitel dieses Abschnitts befaßt sich mit Strategien und Strategieelementen zur Zukunftssicherung der Unternehmen im gegebenen dynamischen Umfeld.

Der zweite Abschnitt behandelt die Produktion im engeren Sinne. Er beginnt mit Ausführungen zur unternehmensstrategischen Bedeutung von Fabriken und zum Zielsystem der Produktion. Weitere Kapitel behandeln den Rechnereinsatz in der Produktion und die (Produktions-) Logistik, da die Bedeutung dieser Aspekte immer mehr zunimmt.

Gegenstand des dritten Abschnitts ist die Materialflußsimulation. Zunächst wird ein ausführlicher Überblick gegeben, daran anschließend werden zukünftig zu erwartende Entwicklungen diskutiert. Besonderen Raum nimmt dabei die Darstellung der veränderten Anforderungen an Simulationsstudien und die bei ihrer Durchführung eingesetzten Softwarewerkzeuge ein. Daneben wird auch die konzeptuelle und technologische Basis für zeitgemäße solche Werkzeuge erörtert.

Im vierten und letzten Abschnitt wird zunächst eine Reihe von Teilkonzepten für die Abbildung von und den Umgang mit materialflußtechnischen Systemen in Simulationswerkzeugen vorgestellt. Abschließend werden diese Konzepte im Hinblick auf ihre Eignung für die erweiterte Nutzung damit erstellter Simulationsmodelle in der Anlagensteuerung untersucht.

Inhalt

I Produzierende Unternehmen heute

1 Die Entwicklung der industriellen Produktion

Die industrielle Güterproduktion hat einen sehr großen Einfluß auf die Verhältnisse in unserer Gesellschaft. Sie beeinflußt unseren Alltag in vielfältiger Weise. Dies kommt beispielsweise darin zum Audruck, daß oft davon gesprochen wird, daß wir in einer "Industriegesellschaft" leben [1].

Die Güter, die wir konsumieren, werden von produzierenden Unternehmen in Fabriken hergestellt. In den weiteren Abschnitten dieser Arbeit wird die Planung und Steuerung dieser Fabriken behandelt. Um die Anforderungen an die heutigen Fabriken und das diesbezügliche Verhalten der sie besitzenden und betreibenden Unternehmen einordnen zu können, erscheint eine Darstellung der Entwicklung der industriellen Produktion von Gütern sinnvoll. Sie soll daher in diesem Kapitel gegeben werden.

1.1 Die Industrielle Revolution

In der Zeit vor 1750 erfolgte die Produktion von Gütern in Hand- und Heimarbeit. Die industrielle Produktion, wie wir sie kennen, hat sich dann in den Jahren zwischen 1750 und 1850 im Zuge der sogenannten "Industriellen Revolution" entwickelt.

---

[1] Heute wird auch zunehmend der Begriff "Informationsgesellschaft" benutzt. Wie noch gezeigt werden wird, ist dies vielleicht angemessener und hat jedenfalls seine Berechtigung.

Diese Veränderungen nahmen ihren Ausgang von Großbritannien. Dort führte die auftretende Verknappung von Holz als Brenn- und Konstruktionsmaterial zum Einsatz von Steinkohle als Energieträger und von Eisen, das mit immer besseren mechanischen Eigenschaften hergestellt werden konnte, als Werkstoff. Gleichzeitig kam es aufgrund steigender Nachfrage zu Produktionsengpässen in der damals bedeutenden Textilindustrie, die mit den bestehenden Techniken und Organisationsformen nicht behoben werden konnten [1].

Durch das Zusammentreffen dieser Umstände entstand ein Bedarf an neuen Verfahren, Maschinen und Werkzeugen, die der Forderung nach gesteigerter Produktivität sowie höherer und gleichmäßigerer Qualität der Erzeugnisse genügten. Dies führte zur Entwicklung mechanischer Spinnmaschinen und Webstühle. Auch die Entstehung von Drehbänken, Fräs- und Bohrmaschinen fällt in diese Zeit. Der Bedarf an Antriebsenergie konnte mit der von James Watt erfundenen Dampfmaschine (Patenterteilung 1769) gedeckt werden.

In den entstehenden Fabriken, in denen die neuen Maschinen eingesetzt wurden, gab es in der Regel eine zentrale Kraftquelle. Von dieser wurden die übrigen Maschinen über Transmissionen angetrieben. Mithin bestimmte die Energieerzeugung und -verteilung den Aufbau der Fabriken.

Auch außerhalb der Produktionsstätten ergaben sich weitreichende Veränderungen. Für den Transport der Rohstoffe und Fertigprodukte mußten die Verkehrswege ausgebaut und neue Transporttechniken entwickelt werden. Beispielsweise wurde auch die Eisenbahn in dieser Zeit (und ebenfalls in Großbritannien) erfunden [2]. Als weitere Folge ergaben sich daraus erhebliche Veränderungen in Bauwesen und Städtebau.

Die meisten Neuerungen dieser Zeit entstanden aus der Arbeit von Erfindern und Konstrukteuren, die sich die notwendige Kenntnisse meist autodidaktisch erarbeiteten. Die akademische Wissenschaft wandte sich der Technikentwicklung nur langsam zu, beispielsweise erreichten die entstehenden Ingenieurwissenschaften erst zu Beginn des 20. Jahrhunderts ihre allgemeine Anerkennung.

Da die Umwälzungen in Wirtschaft und Technik im Vergleich zur davorliegenden Zeit in rasantem Tempo abliefen, wurde später der Begriff “Industrielle Revolution” geprägt.

---

[1] vgl. Warnecke: *Die Fraktale Fabrik*; S. 28 ff.

[2] Die erste Eisenbahnlinie wurde im Jahre 1825 zwischen Stockton und Darlington eröffnet.

Im Blick auf den Zusammenhang dieser Arbeit können folgende Eckpunkte festgehalten werden:

- Der beherrschende Trend der Industriellen Revolution war die Mechanisierung.
- Die Schlüsselindustrien (soweit davon schon gesprochen werden kann) dieser Zeit waren die Grundstoff- (Bergbau, Energieerzeugung) und die Textilindustrie.
- Der Aufbau der ersten Fabriken war durch die Energieerzeugung und -verteilung bestimmt.

1.2 Die zweite industrielle Revolution

Etwa um die Jahrhundertwende begann die nächste große Welle von Veränderungen in der Produktionstechnik. Sie nahm ihren Ausgang von den USA.

Die Erfindungen der Dynamomaschine durch Werner von Siemens im Jahre 1866 und des Verbrennungsmotors durch Rudolf Diesel im Jahre 1892 ermöglichte eine Dezentralisierung der Antriebe. Damit entfiel die Notwendigkeit der Transmissionen, die benötigte Antriebskraft kann seitdem lokal an jedem Arbeitsplatz erzeugt werden. In der Folge konnten immer mehr Bearbeitungsvorgänge mechanisiert werden, so daß die Produktivität der Arbeitskraft erheblich zunahm. Weiterhin ergab sich aus der nahezu beliebigen Verfügbarkeit von Antriebsenergie die Möglichkeit, die Einrichtungen in den Fabriken entsprechend dem Produktionsablauf anzuordnen. Dadurch konnten zuvor verteilte Bearbeitungsvorgänge zu produktbezogenen Prozeßketten zusammengefügt werden.

Die von Henry Ford noch vor dem ersten Weltkrieg in Detroit errichtete Automobilfabrik war in dieser Hinsicht von epochaler Bedeutung. Die “Tin Lizzie” (offiziell “Model T”) wurde auf einer Fertigungsstraße gebaut. Das Produkt wurde den einzelnen Arbeitsstationen in einem zuvor errechneten Rhythmus so zugeführt, daß zum einen an allen Stationen gleichzeitig und permanent gearbeitet werden konnte und sich zum anderen die geringstmöglichen Betriebskosten ergaben [1].

Da der Fahrzeugtransport auf Fords Fertigungsstraße ohne Eingriff der Arbeiter ablief, markiert sie auch den Beginn der Automatisierung. Im Laufe der Zeit wurden mehr und mehr Werkzeuge und Hilfsmittel zur Steuerung und Regelung der Produktionsprozesse entwickelt.

---

[1] vgl. Warnecke: *Die Fraktale Fabrik*; S. 31

Diese basierten zunächst noch auf ständigen menschlichen Eingriffen, arbeiteten aber zumindest im regulären Betrieb zunehmend ohne daß Bedieneraktivitäten notwendig waren.

In der chemischen Industrie verbreiteten sich solche weitgehend automatisierten Systeme besonders schnell, im Laufe der Zeit griff die Automatisierung jedoch auf alle Branchen über.

Neben den geschilderten technischen und organisatorischen Veränderungen in den Fabriken war es für die zweite industrielle Revolution ebenso kennzeichnend wie für ihren Erfolg ausschlaggebend, daß es gelang, das durch den Produktivitätszugewinn erschlossene Potential sowohl in sinkende Kosten und Preise als auch in höhere Einkommen für die Erwerbstätigen umzusetzen.

Fords Arbeiter waren mit ihrem Einkommen in der Lage, sich das von ihnen selbst gebaute Produkt kaufen zu können, was dem Unternehmen wiederum einen riesigen Absatzmarkt eröffnete. Nach diesem Prinzip wurde die Stückzahl der gefertigten Produkte zum bestimmenden Parameter. Die Massenfertigung erlebte ihre höchste Blüte [1].

Im Laufe der Jahre vervielfachten sich die volkswirtschaftlichen Leistungen und damit einhergehend nahm der Wohlstand des einzelnen zu. Der kaum begrenzten Verbrauch wertvoller Ressourcen wie beispielsweise der nicht erneuerbaren Energieträger, die Bedrohung ökologischer Gleichgewichte und sowohl das gesellschaftliche wie auch das globale Wohlstandsgefälle zeigen allerdings, daß auch die zweite industrielle Revolution ihre Schattenseiten hatte (und noch hat).

Insgesamt können die folgenden wesentlichen Punkte festgehalten werden:

- Die beherrschenden Trends der zweiten industriellen Revolution waren die weitere Mechanisierung und insbesondere die Automatisierung.

- Die Schlüsselindustrien dieser Zeit waren zunächst die Stahl- und später die Automobilindustrie.

- Der Aufbau der Fabriken war durch die Produkte und die zu ihrer Erzeugung nötigen Vorgänge bestimmt.

[1] vgl. Warnecke: *Die Fraktale Fabrik*; S. 31

1.3 Die dritte industrielle Revolution

Die Zeit seit Beginn der 70er Jahre dieses Jahrhunderts wird inzwischen zunehmend als dritte industrielle Revolution bezeichnet. Die unter diesem Begriff zusammengefaßten Entwicklungen nahmen ihren Ursprung wiederum von den USA, aber auch von Japan.

Aus technischer Sicht legte die (allerdings bereits füher erfolgte) Entwicklung des Computers als frei programmierbarer Maschine die Grundlagen für die erfolgten und noch andauernden Veränderungen. Mit seiner Hilfe gelang es, die mentale Leistungsfähigkeit des Menschen zu vervielfachen. Der Beitrag des Computers besteht dabei darin, Berechnungen mit enormer Geschwindigkeit ausführen und sehr große Mengen von Informationen speichern zu können. Der Bereich der Kreativität wird allerdings noch auf absehbare Zeit, wenn nicht für immer, dem Menschen vorbehalten bleiben [1].

Mit Hilfe von Computern können mehr und mehr Vorgänge und ganze Prozesse automatisiert werden. In der Folge nimmt die Industrie eine der Landwirtschaft vergleichbare Entwicklung: Die Produktion der benötigten Güter kann durch immer weniger Arbeitskräfte erfolgen. Obwohl die Anzahl der Arbeitsplätze im tertiären Sektor (Dienstleistungsbereich) derzeit ständig zunimmt, finden heute nicht mehr (fast) alle Menschen einen Arbeitsplatz. Die resultierende strukturelle Arbeitslosigkeit stellt unsere (und vergleichbare) Gesellschaften vor erhebliche Probleme.

Von immenser Bedeutung ist im Zusammenhang mit Computern auch deren zunehmende lokale und globale Vernetzung (Stichwort Internet). Dadurch ist es (prinzipiell) möglich, jede beliebige Information zu jedem Zeitpunkt an jedem Ort der Erde verfügbar zu haben.

Ein anderes wichtiges Merkmal der dritten industriellen Revolution ist die Geschwindigkeit des Wandels, die ein zuvor nie gekanntes Maß erreicht hat. Beispielsweise verringern sich die Produktlebenszyklen ständig, ganze Berufsbilder verändern sich in nur wenigen Jahren grundlegend, verschwinden völlig oder entstehen neu.

---

[1] vgl. Warnecke: *Die Fraktale Fabrik*; S. 34

Die nachfolgenden Kapitel stellen weitere Aspekte der dritten industriellen Revolution vor. Für den Augenblick können die folgenden Punkte festgehalten werden:

- Der beherrschende Trend der dritten industriellen Revolution ist die Verbreitung der Informationsverarbeitung mit dem Computer.
- Die Schlüsselindustrie unserer Zeit ist die Hard- und Softwareindustrie [1].
- Der Aufbau der Fabriken ist durch das Vorhandensein der für den jeweils anstehenden Vorgang erforderlichen Kenntnisse bestimmt [2].

1.4 Zusammenfassung

Es ist festzuhalten, daß sich die industrielle Produktion seit ihrer Entstehung ständig verändert und weiterentwickelt hat und daß sie dies bis zum heutigen Tag tut.

Quer über die beschriebenen Veränderungsschübe hinweg ist eine Tendenz erkennbar: Der überall stattfindende Übergang von zentralen zu dezentralen Strukturen. Beispielhaft seien hier die Übergänge von der (zentralen) Dampfmaschine zum (dezentralen) Elektromotor und vom Mainframe zum Arbeitsplatzrechner (Workstation bzw. PC) genannt.

---

[1] Im Unterschied zu den Schlüsselindustrien früherer Phasen ist sie allerdings nicht mehr der nach Zahl der Beschäftigten bedeutendste Arbeitgeber.

[2] vgl. hierzu die Kapitel 2.1 und 4.1

2 Märkte und Produkte

Die Einkünfte produzierender Unternehmen entstammen dem Verkauf der von ihnen hergestellten Güter auf Märkten. Die Verhältnisse auf den Märkten haben daher weitreichende Auswirkungen auf die Produkte und in der Folge auch auf die Unternehmen selbst. In diesem Kapitel sollen daher zunächst wesentliche heute gültige Aspekte von Märkten und anschließend die wichtigsten daraus resultierenden Anforderungen an die Produkte dargestellt werden.

2.1 Märkte

Märkte sind Orte des Handels, auf denen Anbieter Waren oder Dienstleistungen an Käufer (Nachfrager) verkaufen. Märkte sind nicht notwendigerweise an geographische Lokalitäten gebunden, für den Handel mit Industriegütern gilt viel eher, daß die einschlägigen Märkte einen virtuellen Charakter haben [1].

[1] Es scheint, als ob die meisten Menschen im Umgang mit dieser Virtualität gewisse Schwierigkeiten haben. Sie ist nämlich nicht recht in der Lage, den Wunsch der Käufer nach einer (vielleicht nur vermuteten) Absicherung durch Inaugenscheinnahme des Kaufgegenstands (und des Verkäufers) vor dem Erwerb zu befriedigen. Daher lokalisieren sich virtuelle Märkte von Zeit zu Zeit z.B. auf Messen, die unter anderem aus diesem Grund veranstaltet werden.

Auf Märkten kommen Geschäfte (d.h. Verkaufsabschlüsse) dadurch zustande, daß ein Angebot die Wünsche eines Nachfragers befriedigt, so daß dieser das angebotenen Gut gegen Zahlung eines Entgelts erwirbt.

Die auf einem Markt angebotenen Güter unterscheiden sich in der Regel ebenso voneinander wie die Wünsche der verschiedenen Kaufinteressenten. Dennoch lassen sich, wenn auch mit einer gewissen Unschärfe, Kriterien angeben, die für einen Geschäftsabschluß von Bedeutung sind. Dies sind:

- Produkteigenschaften
  Das angebotene Gut muß die Anforderungen des Käufers an seine technischen und funktionalen Eigenschaften erfüllen. Sie müssen insbesondere dem Käufer die Nutzung des Produktes für die vorgesehenen und alle weiteren üblichen Zwecke ermöglichen.

- Qualität
  Die qualitativen Eigenschaften des Produktes müssen dem Käufer die Nutzung für die vorgesehenen Zwecke über die gesamte erwartete Gebrauchszeit erlauben. Die während des Gebrauchs auftretenden Ausfälle oder Nutzungseinschränkungen dürfen das für die jeweilige Art des Produktes übliche Maß nicht überschreiten.

- Service
  Der Käufer muß die Erwartung gewinnen können, daß der Anbieter auch nach Abschluß des Geschäfts mindestens in der für Güter der jeweiligen Art üblichen Weise und im üblichen Umfang dafür einstehen wird, daß die Nutzung des Gutes durch den Käufer nicht unangemessen eingeschränkt wird. Er muß sich weiter darauf verlassen können, daß der Anbieter alle hierfür nötigen Handlungen zuverlässig und in angemessenen Fristen unternehmen wird.

- Liefertermin
  Das Produkt muß dem Käufer in einer angemessenen Frist nach dem Kauf zur Verfügung stehen. Er muß darauf vertrauen können, daß ein vereinbarter Liefertermin eingehalten wird.

- Preis
  Der für das Gut geforderte Preis und die zugehörigen Zahlungsbedingungen müssen dem Kunden als Gegenwert für das Produkt und seine Nutzung angemessen erscheinen.

Produkteigenschaften, Qualität, Service und Liefertermin werden üblicherweise insgesamt auch als Leistung bezeichnet.

Im Allgemeinen wird davon ausgegangen, daß sich das Geschehen auf Märkten aus dem Zusammenspiel von Angebot und Nachfrage heraus selbst regelt. In der Realität unterliegen jedoch alle Märkte weiteren Einflüssen. Insbesondere sind Märkte keine rechtsfreien Räume, auf die Geschäfte finden vielmehr alle am Ort des jeweiligen Vertragsabschlusses geltenden juristischen Vorschriften Anwendung. Diese umfassen zum einen das allgemeine Handelsrecht [1] und zum anderen alle für das Inverkehrbringen und Benutzen von Gütern der jeweiligen Art geltenden Reglementierungen.

Die von Staat zu Staat bestehenden Unterschiede in diesen Vorschriften führen dazu, daß die Märkte regionalisiert sind. Dem entgegen stehen allerdings die Bemühungen von Staaten und Staatengruppen (z.B. der Europäischen Union (EU)) sowie von internationalen Organisationen (z.B. der Welthandelsorganisation WTO) um Vereinheitlichung und Deregulierung.

Die Vereinheitlichung der Rechtsvorschriften führt zusammen mit der in Kapitel 1.3 dargestellten Tendenz zur weltweiten informationstechnischen Vernetzung zu einer Globalisierung der Märkte. Jedes Angebot und jede Nachfrage sind prinzipiell weltweit bekannt oder können bekannt gemacht werden. Da dies auch für das technische Wissen über die Produkte und ihre Herstellung gilt, können auch Unternehmen aus weit entfernten Ländern entsprechende Güter herstellen und um die einschlägige Nachfrage weltweit konkurrieren. Damit steht jedes produzierende Unternehmen prinzipiell im globalen Wettbewerb.

Die allgemein hohe und insbesondere in den sogenannten Schwellenländern noch kräftig ansteigende Arbeitsproduktivität [2] führt zu einer Zunahme des Angebots auf den Märkten. Diese wandeln sich in der Folge von Wachstums- zu Verdrängungsmärkten. Beide Begriffe beschreiben in ihrer ursprünglichen Bedeutung das jeweilige Marktvolumen, das im Falle von Wachstumsmärkten zunimmt (eben wächst) und bei Verdrängungsmärkten stagniert oder sogar zurückgeht. Von größerer praktischer Bedeutung und heute auch üblicher ist es jedoch, mit diesen Begriffen das auf Märkten bestehende je typische Verhältnis von Angebot und Nachfrage zu charakterisieren. Wachstumsmärkte sind danach Märkte, auf denen ein Nachfrageüberhang besteht, während auf Verdrängungsmärkten das Angebot größer ist als die Nachfrage [3].

---

[1] Dieses kann allerdings Vereinbarungen über die Anwendung eines anderen Rechtssystems durch die Vertragsparteien zulassen.

[2] vgl. hierzu die Kapitel 1.2 und 1.3.

[3] Die nur noch geringe Relevanz der ursprünglichen Definitionen ist u.a. daran erkennbar, daß auch Märkte mit (noch) wachsendem Volumen (z.B. der für Personal-Computer) berechtigterweise als Verdrängungsmärkte angesehen werden.

Bezogen auf die oben dargestellten Kriterien für Geschäftsabschlüsse bedeutet dies, daß auf Wachstumsmärkten eher die Käufer zu Kompromissen hinsichtlich der erhaltenen Leistung und des dafür zu zahlenden Preises bereit sein müssen, während auf Verdrängungsmärkten der Druck auf den Anbietern liegt.

Da die produzierenden Unternehmen im wesentlichen auf Verdrängungsmärkten anbieten, stehen sie vor der Notwendigkeit, jedem potentiellen Käufer tatsächliche und erkennbare Vorteile gegenüber den Wettbewerbern in Bezug auf die genannten Kriterien bieten zu müssen. Können sie dies nicht, werden sie aus dem Markt verdrängt, wodurch wiederum ihre Existenz insgesamt bedroht sein kann.

Die Sicherung und womöglich der Ausbau der Marktposition sowie die Erreichung und Bewahrung von Wettbewerbsvorteilen sind damit essentielle Voraussetzungen für den wirtschaftlichen Erfolg und das Fortbestehen produzierender Unternehmen. Da sich die Märkte wie beschrieben dynamisch verändern, sind aber alle diesbezüglichen Maßnahmen nur von zeitlich begrenzter Wirkung.

Unternehmenserfolg ist heute und in der absehbaren Zukunft nur durch ständige Anpassung nicht nur der Produkte sondern des gesamten Unternehmens zu erreichen. Längere Phasen ohne Veränderungen sind nicht mehr unbedingt Kennzeichen beruhigender Stabilität sondern vielleicht Indiz für verlorengegangene Veränderungsfähigkeit.

2.2 Produkte

Die von produzierenden Unternehmen hergestellten Güter entstehen auf der Basis und als Ergebnis des in dem Unternehmen vorhandenen Know-hows. Diese Summe von Kenntnissen findet in den Produkten gewissermaßen ihren materiellen Ausdruck. Das in einem Unternehmen vorhandene Know-how läßt sich in die Bereiche Produkt- und Produktions-Know-how unterteilen [1].

Der Begriff Produkt-Know-how umfaßt mit der Entwicklung und Konstruktion der Produkte zusammenhängende Aspekte, also im wesentlichen die der Fertigung vorgelagerten Aktivitäten. Hier geht es um die Frage, ob das Unternehmen zur Konzeption marktgerechter Produkte in der Lage ist.

[1] vgl. Warnecke: *Die Fraktale Fabrik*; S. 99

Der Begriff Produktions-Know-how zielt dagegen auf den Fertigungsprozeß. Die Produkte müssen selbstverständlich die vorgesehenen Leistungen erbringen können und die gewünschte Qualität aufweisen. Darüberhinaus müssen sie zu möglichst geringen Kosten und in möglichst geringer Zeit hergestellt werden können.

Für produzierende Unternehmen sind natürlich Kenntnisse in beiden Bereichen notwendig. Allerdings lassen sich üblicherweise, je nach Unternehmen und Produkt, Präferenzen angeben, auf die dann erforderlichenfalls die verfügbaren Ressourcen konzentriert werden können. So wird beispielsweise bei einem Hersteller wissenschaftlicher Instrumente im allgemeinen größeres Gewicht auf das Produktwissen gelegt werden, wohingegen Großserienhersteller aus der Fahrzeug- oder der Hausgeräteindustrie auf führendes Produktionswissen, beispielsweise in der Automatisierung, unbedingt angewiesen sind.

Der Erhalt und der Ausbau dieses Wissens ist für die Unternehmen von zentraler Bedeutung. Diese Aufgabe wird allerdings zunehmend komplexer. Die Ursache hierfür ist der enorme Anstieg des verfügbaren Wissens der Menschheit in unserer Zeit.

Die für die Verdoppelung dieses Wissens erforderliche Zeitspanne beträgt derzeit nur noch ca. sechs Jahre. Die wesentlichen Gründe hierfür sind, daß ca. 90 Prozent aller jemals forschenden Wissenschaftler in der Gegenwart leben und daß ihnen sehr leistungsfähige Hilfsmittel zur Verfügung stehen. Der Austausch von Wissen durch Publikationen, auf Tagungen und über das Internet wurde ausgeweitet und beschleunigt. Heute erscheinen auf der Welt ca. 15 bis 20 Millionen wissenschaftliche Veröffentlichungen jährlich [1].

Zusammen mit den in Kapitel 2.1 dargestellten Veränderungen der Absatzmärkte ergibt sich für Unternehmen zwangsläufig die Notwendigkeit zur Definition neuer Strategien zur Zukunftssicherung, die im folgenden dargestellt werden.

[1] vgl. Warnecke: *Die Fraktale Fabrik*; S. 97 ff.

3 Strategien zur Zukunftssicherung der Unternehmen

Die in den vorangegangenen Kapiteln dargelegten Veränderungen der Absatzmärkte und die sich ständig weiter entwickelnden Anforderungen an die Produkte erfordern von Seiten der produzierenden Unternehmen die Definition und Umsetzung neuer Strategien zur besseren und permanenten Anpassung an die Verhältnisse.

Die nähere Beschäftigung mit solchen Strategien zur Sicherung des Unternehmenserfolgs führt allerdings ebenso wie der Blick auf Untersuchungen erfolgreicher Unternehmen zu der Erkenntnis, daß es kein Patentrezept für den Unternehmenserfolg gibt. Der Erfolg ist zwar meßbar, er beruht jedoch auf dem Zusammenwirken vieler Faktoren und entzieht sich damit einfachen griffigen Regeln [1].

Trotz dieser grundlegenden Schwierigkeit werden bei erfolgreichen Unternehmen immer wieder einige gemeinsame Charakteristika beobachtet [2]:

- Sie konzentrieren sich typischerweise auf relativ wenige Produkte und auf die wichtigsten Kunden.

- Durch Konzentration auf die eigenen operativen Stärken werden die Fertigungstiefe verringert und gleichzeitig die Beziehungen zu den Lieferanten verstärkt.

---

[1] vgl. Warnecke: *Die Fraktale Fabrik*; S. 71

[2] vgl. Schulz: *Warum sind erfolgreiche Unternehmen erfolgreich?*

- Die Produktentwicklung ist auf die rasche Umsetzung auch kleinerer Innovationsschritte ausgerichtet. Der Gedankenaustausch mit den Kunden spielt dabei eine wichtige Rolle.

- Standorte und Logistik sind an die jeweiligen Produkte und Märkte angepaßt.

- Computertechnologien werden zur Verbesserung der Qualität, zur Senkung der Herstellkosten und zur Verringerung der Durchlaufzeiten eingesetzt.

- Das Ausbildungsniveau der Mitarbeiter ist in allen Bereichen relativ hoch.

- Erfolgreiche Unternehmen haben einfache und eher dezentrale Organisationsstrukturen mit teilweise den Produktgruppen zugeordneten Zentralfunktionen.

Wenn sich auch, wie gesagt, keine übertragbaren Erfolgsrezepte angeben lassen, so können doch aus den dargestellten Überlegungen und Beobachtungen einige Punkte isoliert werden, die sich als Elemente oder Bausteine im Rahmen zu entwickelnder individueller Strategien einsetzen lassen. Dies sind:

- Marktorientierung
  Die Aktivitäten eines Unternehmens müssen sich an den Anforderungen und Wünschen der Kunden orientieren. Dies erfordert die Bildung eines entsprechenden Bewußtseins bei allen Beschäftigten. Ein bedeutender Nachteil der im übrigen sinnvollen und notwendigen Arbeitsteilung ist der Verlust des Kundenbezugs. Die einzelnen Mitarbeiter denken und handeln losgelöst vom eigentlichen Zweck des Unternehmens, über Nutzen für den Kunden einen Mehrwert zu schaffen. Abhilfe kann hier dadurch geschaffen werden, daß auch innerhalb des Unternehmens, z.B. zwischen verschiedenen Bereichen oder Abteilungen Kunden-Lieferanten-Beziehungen geschaffen werden. Darüber können dann Forderungen und Möglichkeiten verdeutlicht, abgestimmt und gemeinsam weiterentwickelt werden.

- Qualität
  Es gilt, im Unternehmen einen erweiterten Qualitätsbegriff durchzusetzen, wonach sich die Qualität eines Produkts nicht in seiner Fehlerfreiheit erschöpft [1]. Es sind vielmehr alle Aspekte der Beziehungen zu den Kunden einzubeziehen, da diese immer weniger bereit sind, Qualitätsmängel, schlechten Service und Terminüberschreitungen zu akzeptieren.

---

[1] vgl. Warnecke: *Die Fraktale Fabrik*; S. 76

- Konzentration auf die eigenen Stärken
  Durch den ständigen Zuwachs an verfügbarem Wissen ist es nicht mehr möglich, auf allen Gebieten führend zu sein. Daher ist es notwendig, diejenigen Bereiche des Unternehmens zu identifizieren, die das Kerngeschäft ausmachen und in denen die besonderen operativen Stärken des Unternehmens zu finden sind. Die Aktivitäten und Ressourcen sind dann auf diese Bereiche zu konzentrieren, um hier auch weiterhin im Wettbewerb erfolgreich sein und nach Möglichkeit die Entwicklung mitbestimmen zu können.

- Innovation
  Einmal erzielte Wettbewerbsvorsprünge werden von der Konkurrenz immer schneller wieder aufgeholt. Um neue Vorteile zu erreichen, müssen Innovationen bei den Produkten und in der Produktion immer schneller umgesetzt werden (vgl. Bild 1). Andererseits muß wegen der bestehenden Marktbedingungen der Erfolg von Innovationen von vornherein feststehen. Das neue Produkt muß bei der Einführung bereits besser sein als das alte, da “Kinderkrankheiten” und Anlaufzeiten vom Markt kaum noch toleriert werden.

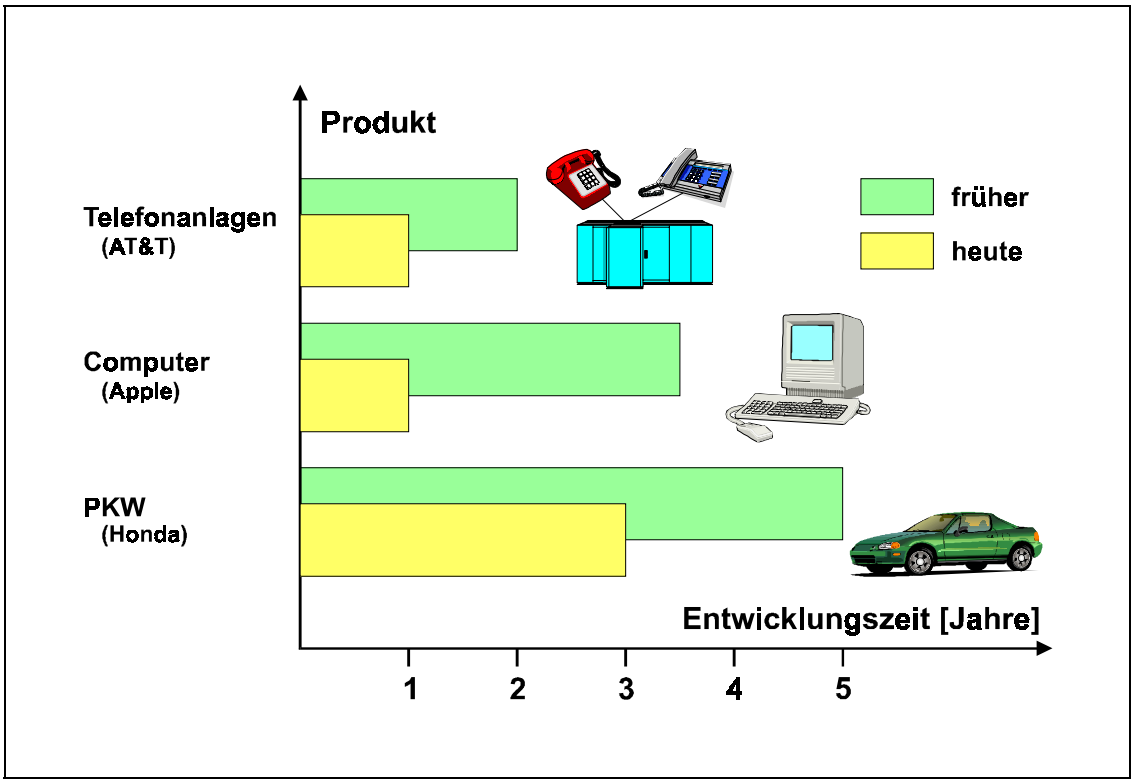

Bild 1: Entwicklung der Innovationszyklen beispielhafter Produkte

- Menge oder Vielfalt
  Es ist festzustellen, ob im Unternehmen eher viele verschiedene Produkte in je geringer Stückzahl oder wenige Produkte in großen Mengen herzustellen sind. Dementsprechend sind die Prioritäten zwischen Erwerb, Erhalt und Entwicklung von Produkt- bzw. Produktions-Know-how zu setzen. Oft fällt die Antwort für verschiedene Unternehmensbereiche unterschiedlich aus. Dann ist normalerweise davon auszugehen, daß die Produktion und ihr Management nicht nach unternehmensweit einheitlichen Prinzipien gestaltet werden können.

- Organisatorische Strukturen
  Die Produktion kommt kaum noch in einen stabilen Zustand, sondern unterliegt einem ständigen Wandel durch Anpassung an sich verändernde Rahmenbedingungen. Das Produktionsmanagement muß diesen Wandel begleiten und steuern und sich dabei auch selbst verändern. Hierarchische und über lange Zeiten stabile Organisationsstrukturen sind dazu kaum in der Lage. Ein großer Teil des Informationsflusses in solchen Strukturen erfolgt in vertikaler Richtung, d.h. über Hierarchieebenen. Die Orte, an denen Informationen entstehen und benötigt werden, sind aber eher auf einer horizontalen Ebene zu finden, so daß der Informationsfluß zweckmäßiger über Bereichsgrenzen, d.h. mit bzw. gegen den Produktionsfortschritt zu organisieren wäre. Unnötige Umwege im Informationsfluß aber kosten Zeit, die immer weniger zur Verfügung steht.

- Kleine Einheiten
  Die herkömmlichen großen Organisationseinheiten können auf veränderte Bedingungen nicht schnell genug reagieren, da die erforderlichen Abläufe zu langsam sind. Typischerweise erfolgt die Gliederung eines Unternehmens heute statt nach Funktionen zunehmend nach Produkten bzw. Produktgruppen. Diesen Einheiten werden auch die entsprechenden Teile zentraler Funktionsbereiche zugeordnet. Je nach Einzelfall sind sie als Cost-Center, Profit-Center oder sogar als formal selbständiges Unternehmen organisiert. Beispielsweise stellt der Automobilhersteller Ford seine gesamten Automobilaktivitäten um von einer regionalen Struktur (Nordamerika, Europa, ...) auf fünf nach Produktgruppen (kleine PKW, mittlere PKW, ...) gegliederte Bereiche, die jeder für sich weltweit operieren [1].

[1] vgl. Ford Motor Company: *Ford Around The World*

- Wertewandel
  Die Umstrukturierung von Unternehmen hin zu kleineren selbständigeren Einheiten führt für diese zu mehr Verantwortung, aber auch zu größeren Handlungsspielräumen. Der insbesondere in den Industrieländern zu beobachtende allgemeine Wertewandel, wonach die Erwerbstätigkeit neben der Existenzsicherung auch mehr Selbstverwirklichung ermöglichen soll, läßt sich daher mit den veränderten Anforderungen an die Unternehmen sinnvoll verknüpfen und kann bei geeigneter Strukturbildung zu einer erheblichen Erhöhung der Motivation der Beschäftigten führen.

Konkrete Strategien für einzelne Unternehmen, die auf diesen und vielleicht weiteren Bausteinen aufbauen müssen und werden natürlich unterschiedlich aussehen. Nach Meinung einer Expertenrunde amerikanischer Führungskräfte müssen erfolgreiche Strategien jedoch fünf Basiselemente aufweisen [1]. Sie müssen demnach

- eine langfristige Vision vorgeben,
- konkrete Schritte auf dem Weg dorthin aufzeigen,
- die benötigten und die vorhandenen Ressourcen einbeziehen und vergleichen,
- einen strategischen Plan zur Ressourcenentwicklung umfassen und
- ein durchgreifendes Programm zur Schulung und Weiterbildung der Beschäftigten vorsehen.

Die dargestellten Punkte können gut mit den Aussagen dieser Experten zusammengefaßt werden. Insgesamt propagieren diese das “agile Unternehmen”, dessen drei wesentliche Merkmale der ständige Wandel, die schnelle Reaktion und ein erweiterter Qualitätsbegriff sind.

[1] vgl. Warnecke: *Die Fraktale Fabrik*; S. 75

II Aspekte moderner Produktion

4 Ausrichtung der Produktion

Die Fabrik als Ort der Produktentstehung ist für produzierende Unternehmen von herausragender Wichtigkeit. Die Beherrschung der Abläufe in der Produktion entscheidet darüber, ob das Unternehmen seine Produkte in der erforderlichen Menge und Qualität und zum geforderten Zeitpunkt liefern kann. Davon ist wiederum die Erzielung von Erlösen und damit der Bestand des Unternehmens insgesamt abhängig.

Aufgrund der wechselseitigen Abhängigkeiten von Märkten, Produkten und Fabriken und der zentralen Bedeutung ihrer Beherrschung kann die Gestaltung der Produktionsstätten eines Unternehmens sinnvoll nur im Rahmen eines integrierten Gesamtkonzepts erfolgen. Es hat ein Zielsystem vorzugeben, innerhalb dessen dann konkrete lang-, mittel- und kurzfristige Aufgaben auszuführen sind [1].

Die langfristigen Aufgaben umfassen in erster Linie das klassische Feld der Fabrikplanung. Dabei geht es um die Entwicklung und Pflege von Perspektivenplänen, deren Horizont meist über die Nutzungsdauer der Produktionsanlagen hinausreicht. Hierzu gehören insbesondere die Standortwahl und die Einführung neuer Technologien auf breiter Basis.

Auf dieser Basis werden in mittelfristigen Zeitabständen Maßnahmen durchgeführt, um die Fabriken an veränderte Gegebenheiten und Zielvorstellungen anzupassen. Hierbei steht bereits die Realisierung, d.h. die Planung und Umsetzung konkreter Veränderungen im Vordergrund.

---

[1] vgl. Rudnig: *Grundlagen der Fabrikplanung*

Die kurzfristig auszuführenden Tätigkeiten umfassen die Vornahme von Ersatzinvestitionen, die Einführung (partieller) technologischer Verbesserungen, die Durchführung von Rationalisierungsmaßnahmen und die Umsetzung anderer lokal wirkender Veränderungen die beispielsweise aus der Tätigkeit von Qualitätszirkeln resultieren.

4.1 Die Fabrik als strategisches Instrument

Nicht zuletzt aufgrund des erheblichen Publikums- bzw. Medieninteresses tritt der Einfluß strategischer Überlegungen auf die Gestaltung der Produktion bei der Durchführung der langfristigen Planungsaktivitäten, und hier insbesondere bei der Standortwahl, sicherlich am deutlichsten zu Tage. Daher werden auch alle mit diesen Aktivitäten in Zusammenhang stehenden Entscheidungen (fast) immer auf der höchsten Führungsebene der Unternehmen getroffen.

Eine große Vielfalt von Faktoren beinflußt die Technologie- und Standortwahl. In jedem Fall muß die Verkehrs-, Informations-, Ver- und Entsorgungsinfrastruktur eines ins Auge gefaßten Standorts den Anforderungen von vorn herein genügen oder entsprechend ausgebaut werden können. Hier kann beispielsweise die Anbindung an Verkehrs- und Kommunikationsnetze ebenso bedeutsam sein wie die Verfügbarkeit von Strom oder Wasser.

Als weiterer Einflußfaktor sind die Marktbeziehungen des Unternehmens zu nennen. Die Nähe zu Kunden und Lieferanten bestimmt nicht nur den anfallenden Transportaufwand maßgeblich, sie kann auch z.B. die wechselseitige Einbindung in Just-In-Time-Konzepte erst ermöglichen oder ungünstigenfalls verhindern. Durch eine entsprechende Standortwahl kann ein Unternehmen für sich unter Umständen Märkte erschließen, die ihm anderenfalls aufgrund bestehender Importzölle oder -beschränkungen verschlossen blieben.

Ein (in diesem Sinne) ungeeigneter Standort kann auch die Umsetzung bestimmter organisatorischer Konzepte in der Produktionssteuerungsverfahren ausschließen. Beispielsweise kann eine Fertigung auf Kundenbestellung nur dann realisiert werden, wenn die Transportzeiten (zusammen mit den Durchlaufzeiten) von der Fabrik zum Kunden nicht zu Lieferfristen führen, die potentielle Kunden nicht akzeptieren.

Von großer Bedeutung ist auch der Faktor Arbeit. Die für den Betrieb der Fabrik benötigten Arbeitskräfte müssen in ausreichender Zahl vorhanden sein und sie müssen die erforderlichen Qualifikationen haben. Natürlich müssen auch die Personalkosten beachtet werden, wobei auch die am Standort allgemein übliche und erreichbare Arbeitsproduktivität zu berücksichti-

gen ist. Auch das Bildungsniveau und das Schul- und Ausbildungssystem sind durchaus von Interesse, sie bestimmen mit über die Entwicklungsmöglichkeiten des Standorts.

Bestehende und geplante Genehmigungs- und Betriebsvorschriften beeinflussen die langfristigen Planungsaktivitäten ebenfalls. Sie können z.B. den Einsatz bestimmter Technologien verhindern oder soweit erschweren oder verzögern, daß sie an einem Standort faktisch nicht in Frage kommen.

Wirtschaftliche Aspekte bilden einen weiteren Einflußkomplex. Hierzu gehören die abzuführenden Abgaben wie Steuern auf Grundbesitz, Umsätze und Gewinne ebenso wie die konkreten Möglichkeiten des Kapitalverkehrs (z.B. Gewinntransferierung zur Muttergesellschaft). Weiter sind in diesem Zusammenhang mögliche Devisenrisiken zu nennen, die beispielsweise daraus resultieren können, daß Kosten und Einnahmen in verschiedenen Währungen entstehen.

Abschließend sei an dieser Stelle noch auf übergeordnete Aspekte wie die Stabilität der wirtschaftlichen und politischen Verhältnisse an einem möglichen Standort und das mit ihm vielleicht verbundene Prestige hingewiesen.

Zusammenfassend ist festzustellen, daß durch die Vielfalt der konkreten Einflußfaktoren und die vorhandenen Gestaltungsmöglichkeiten die Fabrik zu einem strategischen Instrument wird, das über den Unternehmenserfolg entscheidend mitbestimmt.

4.2 Das Zielsystem der Produktion

Nicht nur für die Neu- und Umgestaltung von Produktionsanlagen, sondern auch für den laufenden Betrieb und damit für die Produktionsplanung und -steuerung, resultieren aus den in Kapitel 2.1 dargestellten Veränderungen der Märkte neue Zielsetzungen und Anforderungen.

Das sich ergebende Zielsystem der Produktion ist selbstverständlich dem wichtigsten Ziel des Unternehmens untergeordnet, nämlich der Erwirtschaftung von Gewinnen bzw. der Erzielung einer möglichst hohen Rendite.

In der Produktion hat es für das Erreichen eines wirtschaftlichen Gesamtoptimums schon immer mehrere angepaßte Zielgrößen gegeben [1]. Vom Markt bzw. von den Kunden werden kurze Lieferzeiten und pünktliche Lieferung verlangt. Von der betrieblichen Seite werden

[1] vgl. Wiendahl: *Belastungsorientierte Fertigungssteuerung*; S. 17

dagegen möglichst niedrige Bestände und eine hohe und gleichmäßige Auslastung der Betriebsmittel angestrebt. Bild 2 veranschaulicht das sich ergebende Zielsystem.

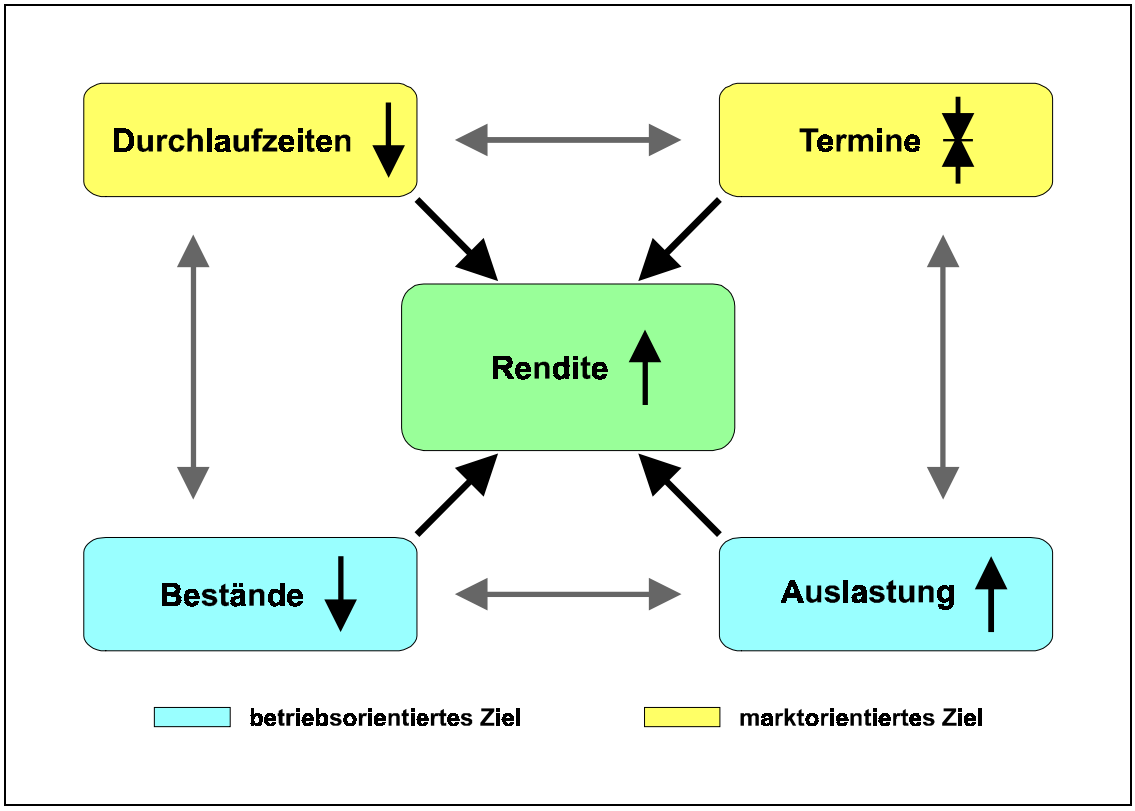

Bild 2: Das Zielsystem der Produktion

Aufgrund der geschilderten Marktentwicklung ist in jüngerer Zeit eine Verschiebung der Gewichtung der Teilziele zu beobachten. Während früher die hohe Auslastung der Maschinen und Anlagen im Vordergrund stand, werden heute kurze Durchlaufzeiten, niedrige Bestände und die Termintreue deutlich stärker gewichtet.

Kurze Durchlaufzeiten und damit kurze Lieferfristen sind heute ein wichtiger Faktor im Wettbewerb [1]. Darüberhinaus mindern sie für das Unternehmen das Änderungsrisiko angearbeiteter Teile und Produkte. Eine hohe Termintreue ist ebenfalls ein Verkaufsargument. Sie zeigt außerdem im Betrieb die Auswirkung, daß wesentlich “ruhiger” und damit kostengünstiger produziert werden kann, weil die Durchführung geplanter Abläufe seltener durch “Feuerwehraktionen” gestört wird [2].

Der Wunsch nach niedrigen Beständen resultiert zum einen aus der damit verbundenen Reduzierung des Umlaufvermögens. Zum anderen ist er Ergebnis der sich durchsetzenden

---

[1] vgl. Kapitel 2.1

[2] vgl. Wiendahl: *Belastungsorientierte Fertigungssteuerung*; S. 17 ff.

Erkenntnis, daß hohe Bestände viele Unzulänglichkeiten wie z.B. Qualitätsmängel und Prozeßanfälligkeiten verdecken und dabei lange Durchlaufzeiten verursachen [1].

Zwischen den Zielgrößen des Zielsystems der Produktion bestehen Zusammenhänge, die es unmöglich machen, eine der Größen zu verändern ohne andere zu beeinflussen. Diese Zusammenhänge sollen im folgenden am Beispiel eines Arbeitsplatzes diskutiert werden.

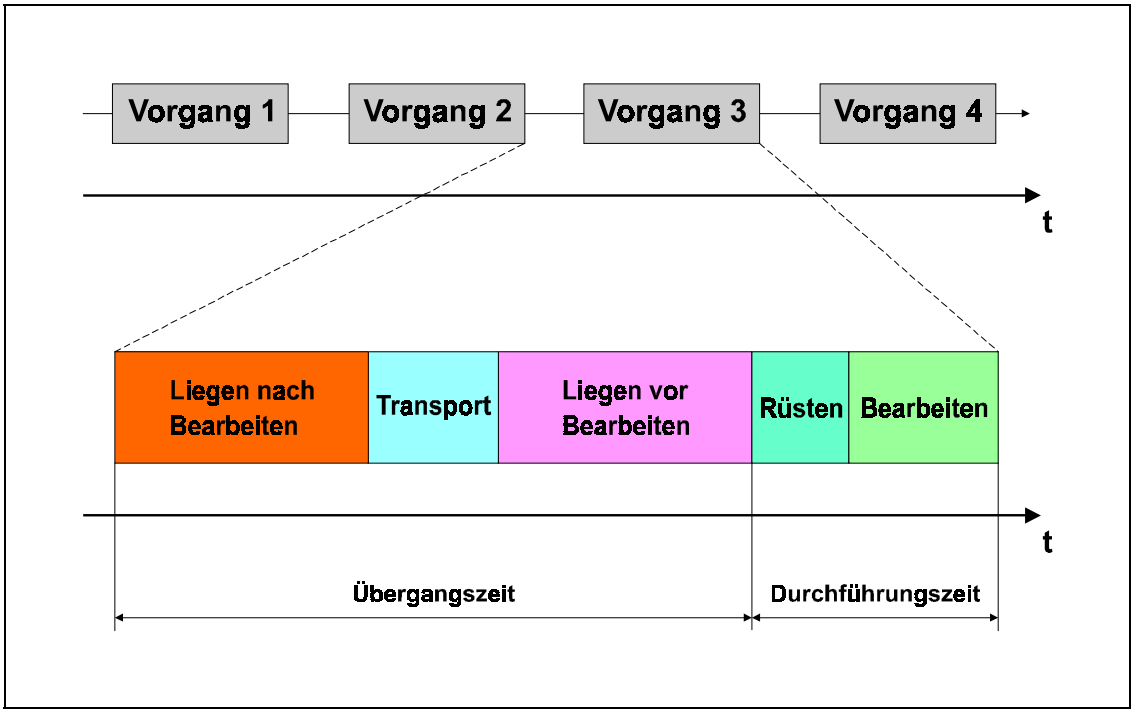

Bild 3: Aufbau eines Durchlaufelements

Die Durchlaufzeit eines Auftrags bzw. Loses durch die Fertigung setzt sich aus den Zeiten für die Durchführung der einzelnen Arbeitsvorgänge zusammen. Diese werden als Arbeitsvorgangs-Durchlaufzeiten oder Durchlaufelemente bezeichnet. Die Abgrenzung der Vorgänge bzw. Durchlaufelemente gegeneinander und ihre Untergliederung in die fünf weiteren Komponenten Liegen nach Bearbeiten, Transportieren, Liegen vor Bearbeiten, Rüsten und Bearbeiten zeigt Bild 3 [2].

Die Bearbeitungszeit ist auf das gesamte Los bezogen, sie ist also die Summe der Bearbeitungszeiten aller Teile des Loses. Weiter werden die Zeiten für Rüsten und Bearbeiten zusammen als Durchführungszeit und die übrigen Zeiten im Durchlaufelement als Übergangszeit bezeichnet.

---

[1] vgl. Wiendahl: *Belastungsorientierte Fertigungssteuerung*; S. 18 ff.

[2] ebd.; S. 51 ff. Wie Bild 3 zeigt, ordnet Wiendahl im Gegensatz zu anderen die Liegezeiten nach Bearbeitung am Vorgänger-Arbeitsplatz und die Transportzeiten jeweils dem Folgearbeitsplatz zu.

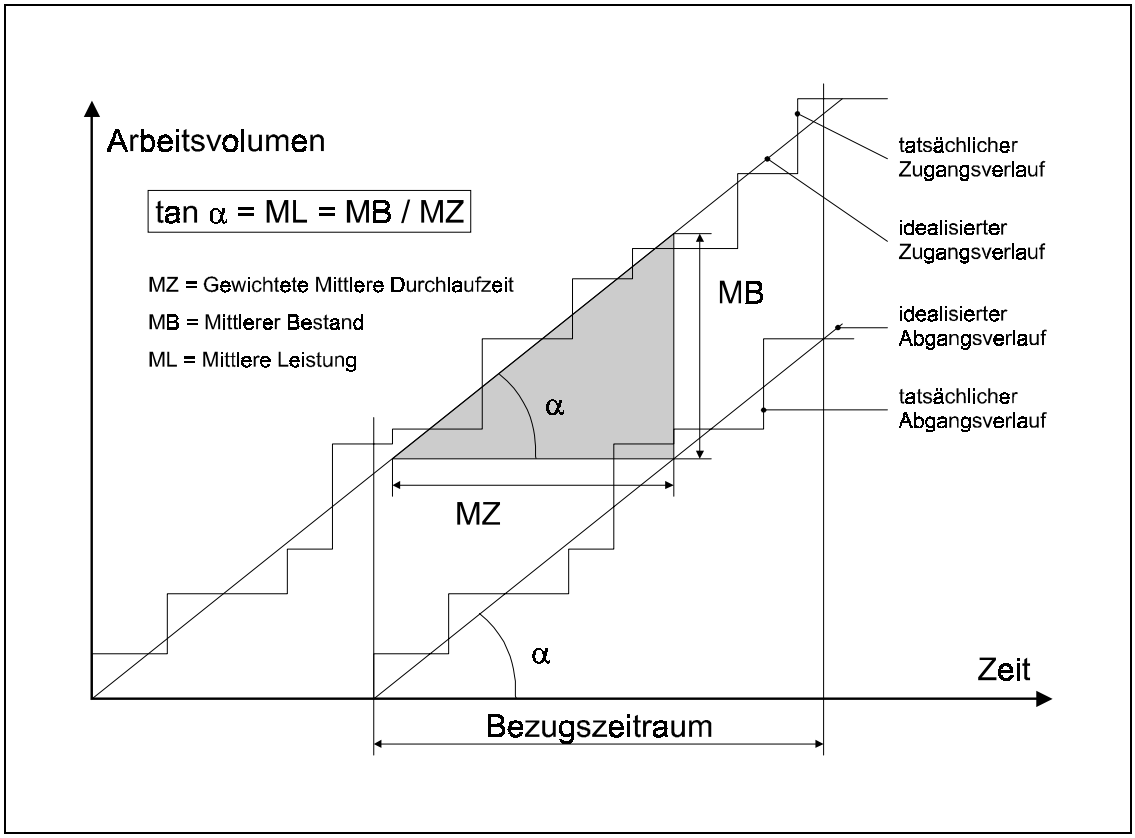

Bild 4: Durchlaufdiagramm eines Arbeitsplatzes

Aufbauend auf diese Definitionen kann das Durchlaufdiagramm für einen Arbeitsplatz entwickelt werden. Hierzu werden die Arbeitsinhalte der in einem Bezugszeitraum an dem Arbeitsplatz tatsächlich zu- bzw. abgegangenen Fertigungsaufträge bzw. Lose über der Zeit als Treppenkurve aufgetragen. Für beide Verläufe werden zusätzlich idealisierende Geraden eingetragen. Bild 4 zeigt ein so entstandenes Durchlaufdiagramm für einen beispielhaften Arbeitsplatz.

Aus dem Diagramm kann weiter abgelesen werden, daß es, eine im Bezugszeitraum gleichbleibende mittlere Leistung des Arbeitsplatzes vorausgesetzt, einen Zusammenhang zwischen dem in Arbeitsvolumen gemessenen Bestand an einem Arbeitsplatz und der sich einstellenden Durchlaufzeit dahingehend gibt, daß sich die Durchlaufzeit proportional zum Bestand verhält.

Dieser Zusammenhang wurde empirisch nachgewiesen und wird allgemein als "Trichterformel" bezeichnet [1]. Sie gilt exakt allerdings nur, wenn im Bezugszeitraum immer ein Bestand vorhanden ist (was bei genauer Betrachtung des Diagramms offensichtlich ist) und keine Reihenfolgevertauschungen stattfinden. Je weniger diese Voraussetzungen gegeben sind, desto eher ist nach Wiendahl der Einsatz von Hilfsmitteln wie z.B. der Simulation angeraten.

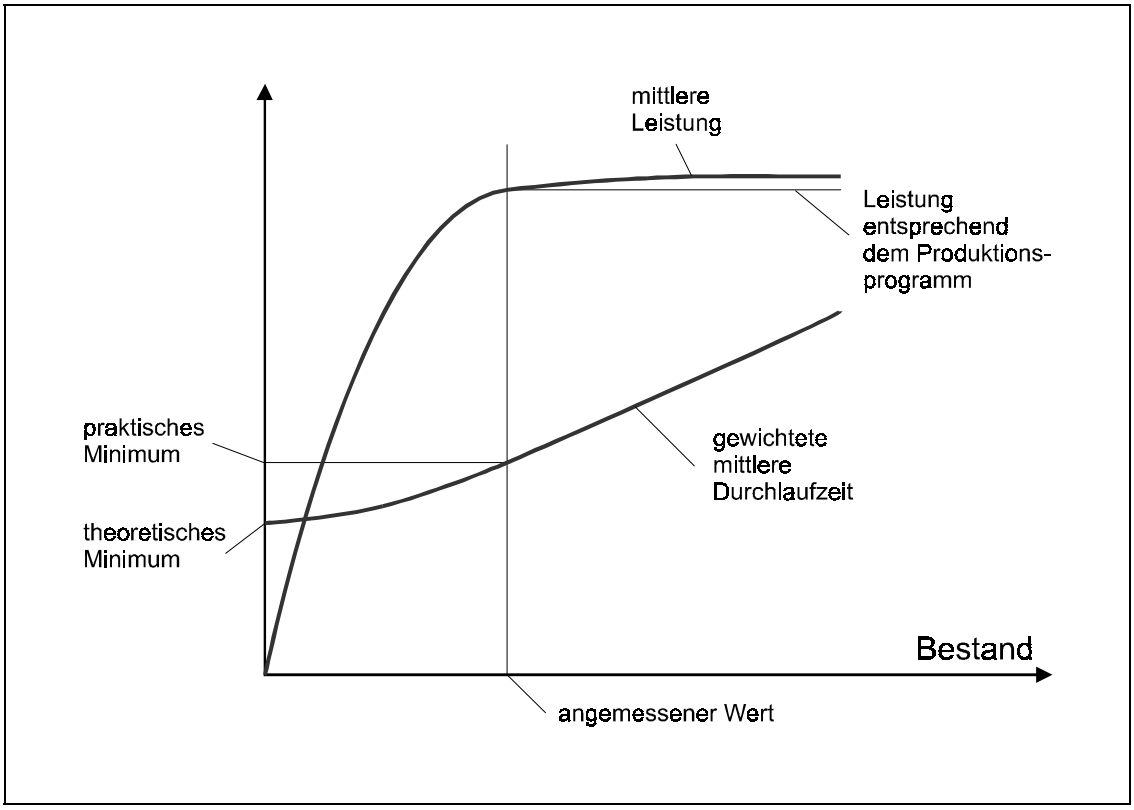

Bild 5: Prinzipieller Verlauf einer Betriebskennlinie

Aus der Veränderung des Bestands an einem Arbeitsplatz in weiten Grenzen lassen sich sogenannte Betriebskennlinien entwickeln [2], deren prinzipiellen Verlauf Bild 5 zeigt. Aus ihnen wird die Veränderung der mittleren Leistung und der mittleren Durchlaufzeit in Abhängigkeit vom mittleren Bestand deutlich.

Die Leistung verändert sich oberhalb eines "angemessenen" Bestands nicht mehr wesentlich, weil es nicht mehr zu Arbeitsunterbrechungen aufgrund fehlender Aufträge kommt, wie es unterhalb des angemessenen Bestands der Fall ist. Weiter ist oberhalb des angemessenen Bestands die Durchlaufzeit proportional zum Bestand, hier gilt die "Trichterformel". Unterhalb des angemessenen Bestands sinkt die Durchlaufzeit nur noch bis auf die mittlere Durchführungszeit zuzüglich der mittleren Transportzeit ab.

---

[1] vgl. Wiendahl: *Belastungsorientierte Fertigungssteuerung*; S. 123 ff.

[2] ebd.; S. 207

Neben den bisher erörterten Abhängigkeiten zwischen Beständen, Durchlaufzeiten und Leistung bzw. Auslastung besteht zwischen Durchlaufzeiten und Termintreue zusätzlich ein Zusammenhang dahingehend, daß die Terminsicherheit mit sinkenden Durchlaufzeiten ansteigt [1]. Dies soll jedoch hier nicht weiter ausgeführt werden.

Zusammenfassend folgt aus den vorstehenden Ausführungen, das im Rahmen des Zielsystems der Produktion die Bestände die wichtigste Führungsgröße sind. Im übrigen ist festzustellen, daß eine optimale Abstimmung der Produktion aufgrund der komplexen und vielfältigen Randbedingungen immer eine Aufgabe ist, die nur unternehmens- und situationsbezogen gelöst werden kann. Die Verhältnisse in der Produktion müssen ebenso wie die Ausrichtung des Unternehmens insgesamt in einem kontinuierlichen Prozeß an die sich verändernden Anforderungen und Gegebenheiten angepaßt werden.

[1] vgl. Wiendahl: *Belastungsorientierte Fertigungssteuerung*; S. 18

5 Rechnerintegrierte Produktion

Die Herstellung eines Produktes durch ein Unternehmen erfordert die koordinierte Ausführung einer ganzen Reihe technischer und betriebswirtschaftlich-planerischer Aktivitäten. Diese werden heute in der Regel computerunterstützt, d.h. unter Nutzung geeigneter Hard- und Softwaresysteme durchgeführt.

Die folgende Darstellung dieser Systeme beschränkt sich im Hinblick auf den Gesamtzusammenhang dieser Arbeit auf die innerbetrieblichen Tätigkeiten von der Auftragsannahme bis zur Auslieferung an den Kunden. Nicht betrachtet werden also unternehmensstrategische und -planerische Tätigkeiten [1] und andere Pre- und Post-Sales-Aktivitäten (z.B. Marketing und Service) [2], deren Bedeutung für das Unternehmen insgesamt damit aber keineswegs in Abrede gestellt werden soll.

In je einem Abschnitt wird kurz auf die Motivation zur Entwicklung solcher Systeme, auf die wichtigsten von ihnen und auf den zu beobachtenden Trend zu ihrer Integration eingegangen. Abschließend werden die sich entwickelnden Rechnerstrukturen in der Fertigung betrachtet.

---

[1] Auf diese Aspekte wurde in den Kapiteln 3 und 4 eingegangen.

[2] Ihre Bedeutung wurde in Kapitel 2.1 angedeutet.

5.1 Motivation

Das aus den heute gegebenen Markt- und Produktanforderungen [1] abgeleitete Zielsystem der Produktion [2] führt zu erhöhten Anforderungen an alle Bereiche des Unternehmens. Um die Marktposition des Unternehmens zu sichern und womöglich zu verbessern, muß nicht nur in der Fertigung, sondern ebenso in den anderen Bereichen die Produktivität, also das Verhältnis von erbrachter Leistung zu erforderlichem Mitteleinsatz, erhöht und die für die Leistungserbringung erforderliche Zeit (Durchlaufzeit) reduziert werden.

Gleichzeitig steigen auch die Anforderungen an die Qualität aller Prozesse, die sich nicht mehr in Fehlerfreiheit erschöpfen kann, sondern unter Berücksichtigung einer zunehmenden Fülle von Kriterien die Erreichung eines aufgabenbezogenen Optimums erfordert.

Diese erhöhten Anforderungen konnten mit den hergebrachten Arbeitstechniken und Informationsmedien nicht mehr erfüllt werden. Der schon früh gesuchte Ausweg fand sich in der Anwendung rechnerbasierter Systeme und daran angepaßter Arbeitstechniken, deren Entwicklung durch die zunehmende Verfügbarkeit leistungsfähiger Computerhard- und -software zu (leistungsbezogen) fallenden Preisen ermöglicht wurde.

5.2 Rechnerunterstütze Systeme

Die wohl bekannteste Art hier zu nennender Systeme sind CAD-Systeme (“Computer-Aided-Design”) zur Unterstützung der Konstruktion. In ihren Anfängen halfen sie nur bei der Zeichnungserstellung (z.B. bei Bemaßungen). Das Leistungsspektrum erweiterte sich schnell, z.B. um bessere Unterstützung der Variantenkonstruktion durch parametrierbare Elemente und um Stücklistengeneratoren. Die Ankopplung von Datenbanken erlaubte die geordnete Zeichnungsverwaltung und den Rückgriff auf Normteilebibliotheken.

Heute ist der Übergang zum räumlichen Konstruieren (3D-CAD) weitgehend vollzogen. Moderne CAD-Systeme generieren z.B. Schnitt-, Explosions- und Zusammenbauzeichnungen weitgehend automatisch und integrieren Berechnungsverfahren wie z.B. FEM (s.a. Bild 6).

---

[1] vgl. Kapitel 2

[2] vgl. Kapitel 4.2

Der CAD-Einsatz zielte zunächst auf eine quantitative Rationalisierung durch Zeiteinsparung. Heute ist die qualitative Verbesserung des Konstruktionsprozesses ein gleichberechtigtes Ziel [1].

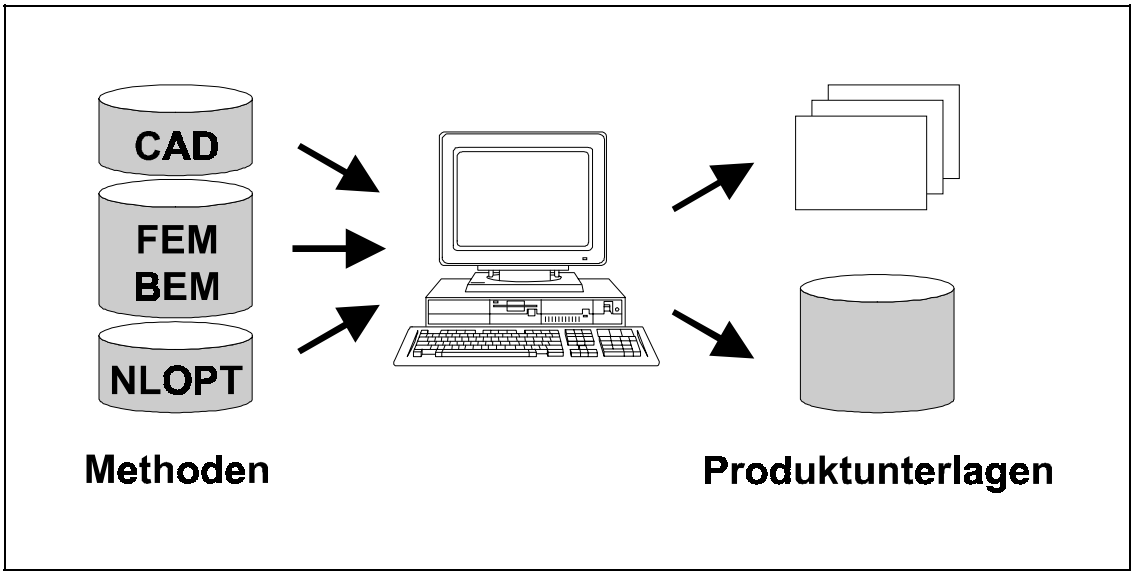

Bild 6: CAD-System mit integrierten Berechnungsverfahren

Ebenso wie CAD unterstützt auch CAM ("Computer-Aided-Manufacturing") technische Aktivitäten bei der Produktherstellung. Unter dem Begriff CAM werden Hard- und Softwaresysteme für den Einsatz in der Fertigung zusammengefaßt. Die ersten solchen Systeme waren numerische Steuerungen für Werkzeugmaschinen (NC-Steuerungen). Diese Technik hat sich von den ursprünglichen fest verdrahteten bis zu den heutigen frei programmierbaren und vernetzten Steuerungen (CNC bzw. DNC) entwickelt.

Daneben zählen zum Bereich des CAM die in der Fertigung eingesetzten Speicherprogrammierbaren Steuerungen (SPS) für Lager- und Transportsysteme, die Steuerungen von Handhabungsgeräten bzw. Robotern, Zellenrechner und Anlagenleitstände sowie Betriebs- bzw. Maschinendatenerfassungssysteme (BDE bzw. MDE).

CAM ermöglicht Produktivitätssteigerungen vor allem durch hoch automatisierte und damit bedienungsarme oder bedienungslose Fertigungs- und Transportsysteme.

[1] vgl. Geitner: *CIM Handbuch*; S. 199

Auch für den Bereich der Arbeitsvorbereitung sind computerunterstützte Systeme verfügbar, die unter dem Begriff CAP ("Computer-Aided-Planning") zusammengefaßt werden. Ausgehend von Informationen aus der Konstruktion über Werkstückgeometrien, Qualitätsanforderungen und Materialeigenschaften können unter Zuhilfenahmen ergänzender Daten (z.B. über Werkstoffe und Betriebsmittel) Arbeitspläne, Montageanweisungen und NC-Programme erstellt werden [1].

Am weitesten fortgeschritten ist die Generierung von NC-Programmen, die zumindest teilweise automatisch durchgeführt werden kann. Typischerweise werden die Programme zunächst in einer neutralen Hochsprache (APT, EXAPT o.ä.) erzeugt und dann durch Postprozessoren an die jeweilige Maschine bzw. ihre Steuerung angepaßt. Die automatische Generierung von Arbeits- und Montageplänen hat noch keine breite Anwendung in der Praxis gefunden. Als Stand der Technik ist die Arbeitsplanverwaltung anzusehen [2]. Sie unterstützt in der Regel immerhin die Änderung bzw. Anpassung im Dialog am Rechner und erlaubt damit zumindest die effiziente Erzeugung von Variantenplänen.

Der Einsatz von CAP-Systemen ermöglicht Produktivitätszugewinne vor allem durch schnellere Erzeugung von Arbeits- und Montageplänen sowie von NC-Programmen und durch die automatische Anpassung letzterer an verschiedene Maschinen (bei verfügbarem Postprozessor).

Schließlich seien hier noch die computerunterstützten Systeme für die Qualitätssicherung genannt, die unter dem Begriff CAQ ("Computer-Aided-Quality-Assurance") zusammengefaßt werden. Sie dienen insbesondere der Erfüllung von Aufgaben aus den Bereichen Qualitätsplanung, -prüfung und -lenkung indem sie beispielsweise bei der Ausarbeitung von Prüfplänen, der Durchführung von Prüfungen und bei der Erfassung und Auswertung von Prüfergebnissen helfen.

Wie bei der Qualitätssicherung allgemein, lassen sich die Produktivitätswirkungen von CAQ-Systemen nur schwer ermessen. Auch wenn geeignete Systeme sicherlich Möglichkeiten für Effizienzsteigerungen in der Qualitätssicherung selbst bieten, ist der üblicherweise intendierte Nutzen eher indirekter Natur. Auf die Bedeutung einer hohen und gleichmäßigen Produktqualität für den Markterfolg des Unternehmens im allgemeinen wurde bereits hingewiesen [3]. Daneben erleichtern oder ermöglichen CAQ-Systeme die Qualitätsnachweisführung bei der Anwendung eines unternehmensübergreifend standardisierten

---

[1] vgl. Geitner: *CIM-Handbuch*; S. 249

[2] ebd.; S. 274

[3] vgl. Kapitel 2.1

Qualitätsmanagements [1] ebenso wie im Falle von Rechtsstreitigkeiten. Aufgrund der Komplexität der Verfahren und der großen Datenmengen wird auch die Durchführung aufwendiger statistischer Analysen zur Identifikation von Fehlerquellen durch die kombinatorische Untersuchung von Prozeß- und Qualitätsdaten de facto erst mit Computerhilfe möglich.

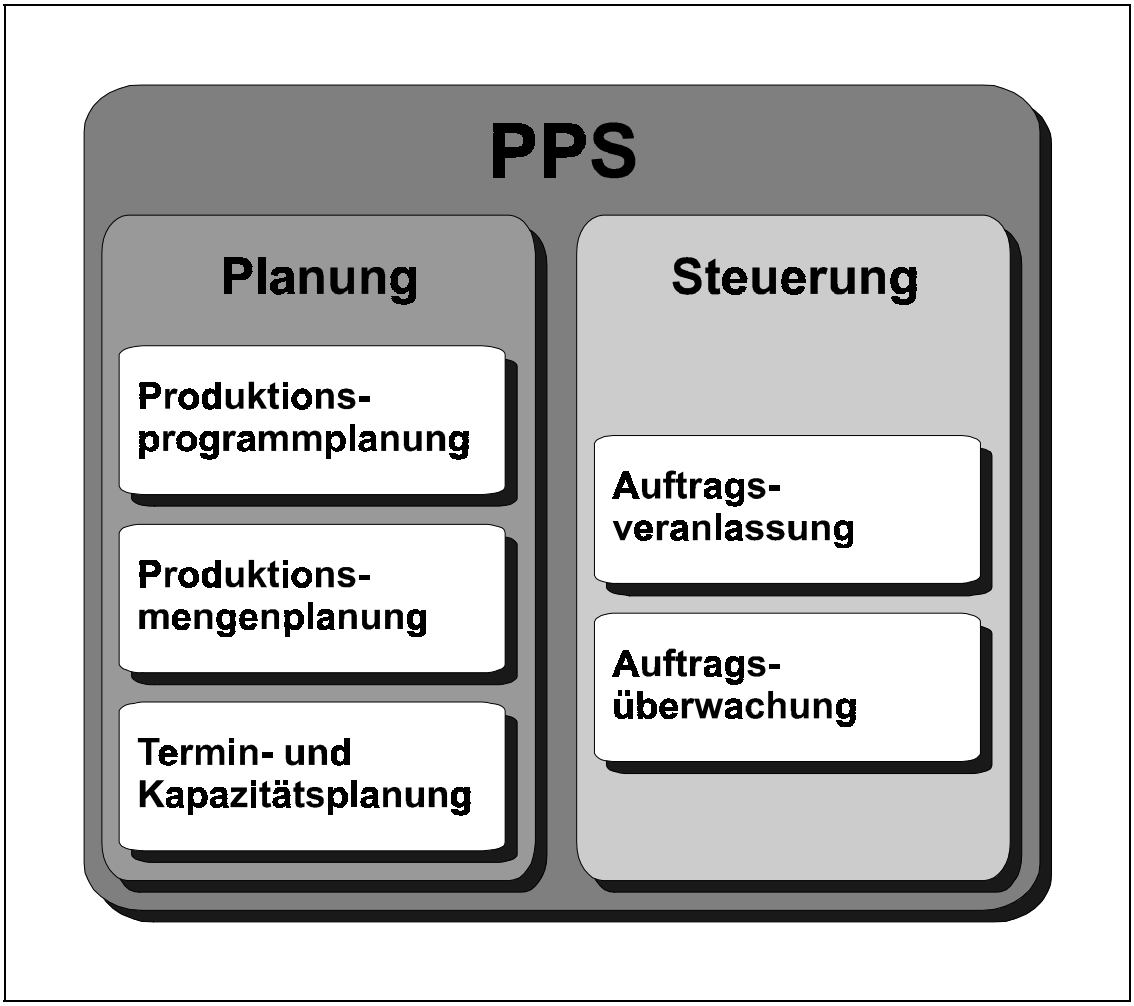

Bild 7: Funktionen eines PPS-Systems

Produktionsplanungs- und -steuerungssysteme (PPS) erfüllen im Gegensatz zu den bisher genannten Systemen statt technischer primär betriebswirtschaftlich-planerische Funktionen. Sie greifen dabei auf ein als Datenbank angelegtes auftrags- und bewegungsneutrales Datengerüst zurück. Die darin enthaltenen Informationen können in Stamm-, Struktur- und Zuordnungsdaten eingeteilt werden und beschreiben u.a. Teile, Stücklisten, Arbeitspläne, Arbeitsplätze, Betriebsmittel, Kostenstellen, Personal, Kunden und Lieferanten [2].

---

[1] z.B. nach DIN EN ISO 9000 ff.

[2] vgl. Geitner: *CIM-Handbuch*; S. 54 ff. Dort wird die PPS-Datenbasis ausführlicher diskutiert.

Wie in Bild 7 dargestellt, umfaßt die Produktionsplanung und -steuerung die Funktionen Produktionsprogramm- und -mengenplanung, Termin- und Kapazitätsplanung sowie Auftragsveranlassung und -überwachung, die PPS-Systeme auf der Basis des oben beschriebenen Datengerüsts durchführen bzw. unterstützen [1].

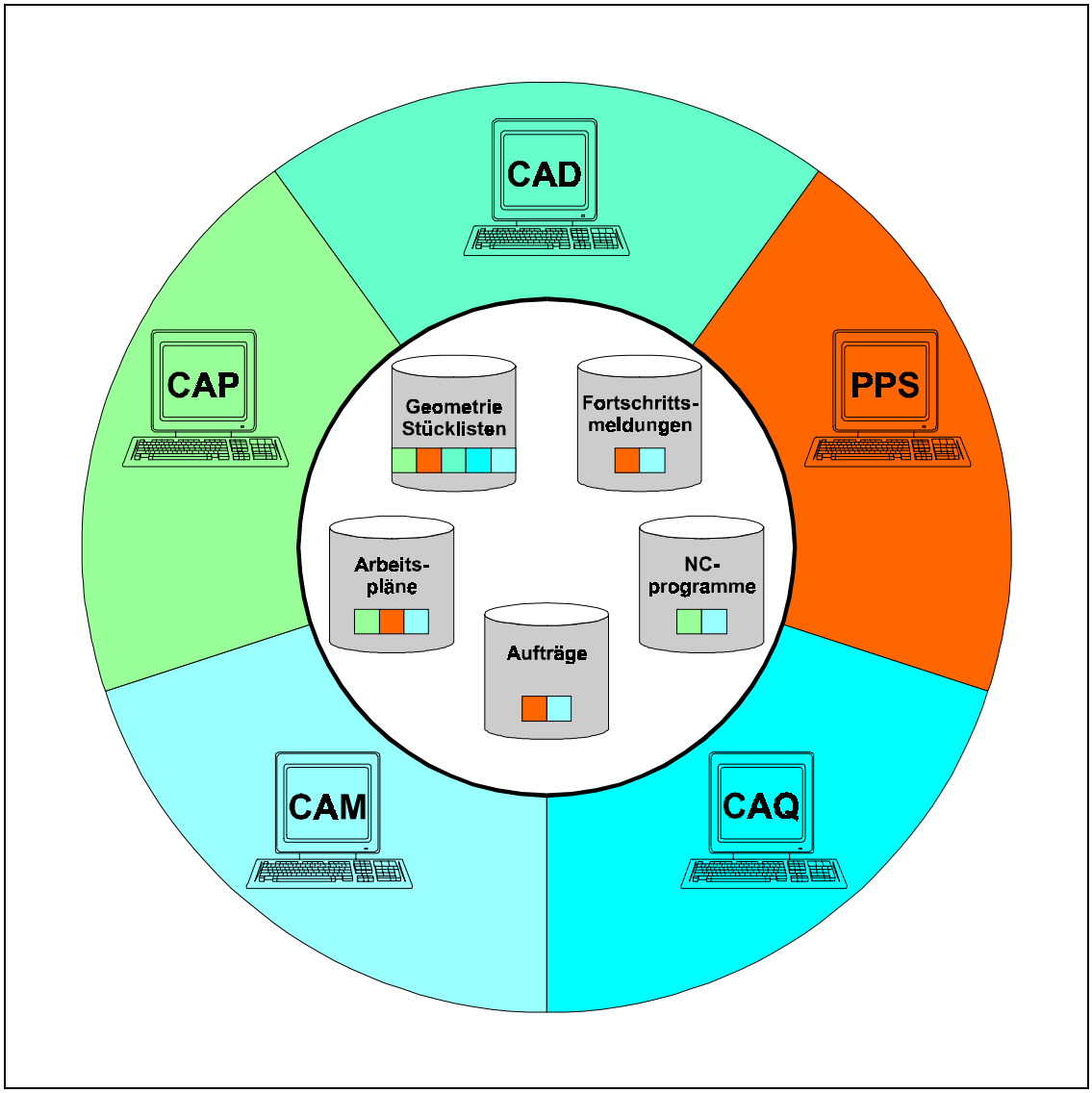

Bild 8: Mehrfache Datennutzung durch betriebliche Informationssysteme

Die Vorteile von PPS-Systemen liegen vor allem in der damit möglichen rechtzeitigen und fundierten Grobplanung und in der konsequenten Auftragsverfolgung. Aufbauend auf eine funktionierende Betriebsdatenerfassung bieten sie ein realistisches Abbild des Betriebsgeschehens und ermöglichen so beispielsweise ein effizientes Störungsmanagement.

---

[1] Auf die Funktionen soll hier nicht im einzelnen eingegangen werden. Ausführlichere Darstellungen bieten z.B. Geitner: *CIM-Handbuch*; S. 45 ff. und Eversheim: *Organisation in der Produktionstechnik; Band 1 Grundlagen*; S. 59 ff.

5.3 Systemintegration

Die oben beschriebenen Systeme wurden in den Betrieben zumeist als Einzellösungen eingeführt, da damals nur abgegrenzte, überschaubare Systeme beherrschbar erschienen. Es entstanden informationstechnische Inseln.

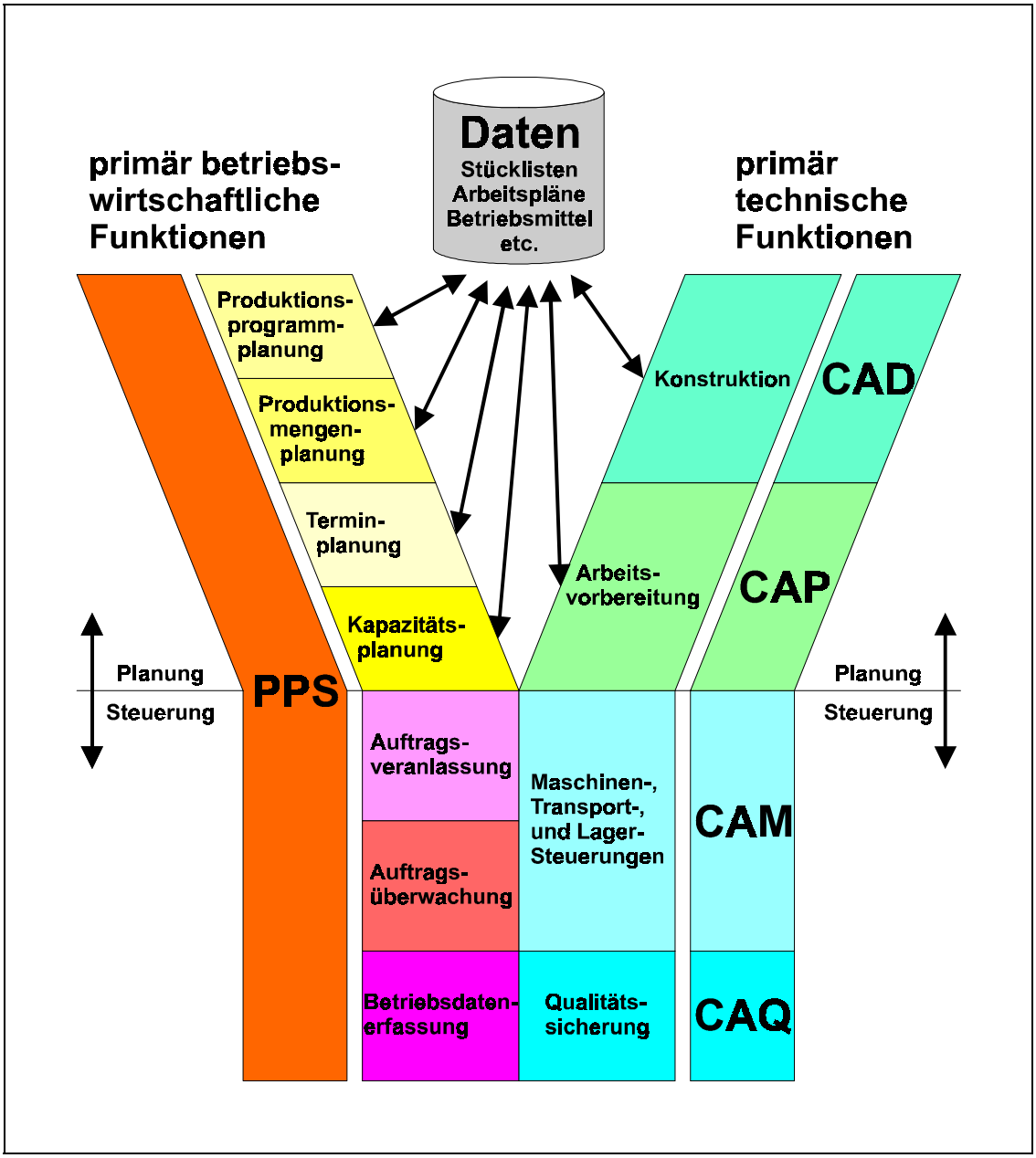

Bild 9: CIM-Modell nach Scheer

Im täglichen Einsatz ließen sich die erwarteten Produktivitätsfortschritte nicht im erwarteten Umfang dauerhaft realisieren. Die Ursache hierfür waren die erforderlichen Aufwendungen für die Anlage und Pflege der Datenbestände der Einzelsysteme. Insbesondere waren (und sind zum Teil bis heute) Daten, die in mehreren Abteilungen bzw. von deren Softwaresystemen benötigt werden (s.a. Bild 8), mehrfach einzugeben, da die Einzelsysteme keine Informationen austauschen konnten.

Es wurde klar, daß informationstechnische Systeme in den Unternehmen ihre volle Wirtschaftlichkeit erst entfalten können, wenn sie Daten miteinander austauschen können [1]. Die wirtschaftlichen Vorteile, die sich beispielsweise aus der Übergabe von Konstruktionsdaten aus CAD-Systemen an Systeme der Arbeitsvorbeitung und der direkten Übertragung dort erstellter Programme an numerisch gesteuerte Maschinen ergeben, sind so offensichtlich und bedeutend, daß die Integration der betrieblichen Informationssysteme ins unternehmerische Blickfeld rückte.

Einschlägige Lösungen firmieren unter dem Begriff CIM ("Computer-Integrated-Manufacturing). Der Integrationsgedanke ist das verbindende Element der vorliegenden CIM-Modelle, von denen ein bekanntes in Bild 9 dargestellt ist. Mehrfachaufwand für Dateneingabe und -pflege und die damit verbundenen Fehlermöglichkeiten sollen durch den Aufbau einer unternehmensweiten Datenbasis vermieden werden. Auf diese Datenbasis, die technisch eine entsprechend erweiterte PPS-Datenbank ist, können dann alle Systeme zugreifen. Sie werden dann als CIM-Komponenten bezeichnet.

Aus der bisherigen Betrachtung blieben die Informationssysteme der kaufmännischen Verwaltung ausgespart. Für sie ist der Begriff CAO ("Computer-Aided-Office") geprägt. Ihre Integration mit den CIM-Komponenten (insbesondere mit PPS-Systemen) wird ebenfalls betrieben. Hierfür steht der Begriff CAI ("Computer-Aided-Industry") [2].

Einige Vorschläge versuchen sich darüberhinaus an der Integration von CIM-Lösungen mehrerer Unternehmen zu einem "CIM-Verbund" unter dem Titel CIB ("Computer-Integrated-Business") [3]. Das heute noch weit entfernte Ziel solcher Konzepte ist die rechnerintegrierte Gesamtwirtschaft.

---

[1] vgl. Geitner: *CIM-Handbuch*; S. 3

[2] ebd.; S. 6

[3] ebd.; S. 10

5.4 Rechnerstrukturen in der Fertigung

Die Rechnerunterstützung in der Fertigung (CAM) umfaßt, wie oben dargestellt, die Teilfunktionen Bearbeiten, Handhaben, Transportieren und Lagern. Auch hier wurde bald erkannt, daß ein signifikanter Nutzen z.B. durch verringerte Durchlaufzeiten und Bestände nicht durch entkoppelte Teillösungen, sondern nur durch Integration zu erreichen ist.

Die ersten Schritte in diese Richtung wurden mit der Installation flexibler Fertigungssysteme (FFS) [1] unternommen. In der Bundesrepublik Deutschland wurde das erste derartige System im Jahre 1971 in Betrieb genommen, ihre erwartete Verbreitung fanden sie aber erst mit Beginn der 80er Jahre [2].

Das Ausmaß der mit flexiblen Fertigungssystemen erzielbaren wirtschaftlichen Verbesserungen wird zu einem Großteil von der Leistungsfähigkeit des übergeordneten Informationssystems bestimmt. Diesem obliegt die Steuerung und Überwachung sämtlicher Bearbeitungsabläufe und Materialbewegungen. Sowohl im Normalbetrieb wie auch bei Störungen muß es in der Lage sein, die Flexibilität des Gesamtsystems voll zu nutzen, um die Erreichung der Produktivitätsziele unabhängig vom aktuellen Auftragsmix zu gewährleisten.

Der Funktionsumfang moderner Informationssysteme umfaßt sowohl die Verwaltung von Werkzeugen, Vorrichtungen sowie Lager- und Bereitstellplätzen als auch Dispositionsaufgaben wie Auftragsverfolgung und Maschinenbelegungsplanung. Aus Gründen der Verfügbarkeit und zur Gewährleistung der erforderlichen Reaktionszeiten erfordern bereits kleinere Systeme den Aufbau einer hierarchischen Rechnerstruktur mit einer Verteilung von zeitkritischen und -unkritischen Aufgaben [3].

Auf der nächsthöheren Realisierungsstufe der rechnerintegrierten Fertigung werden aus mehreren Fertigungszellen, -inseln und -systemen sowie aus verbleibenden Einzelmaschinen und manuellen Verrichtungsplätzen sogenannte flexible Fertigungsverbundsysteme aufgebaut [4]. Sie bilden rechnergeführte logistische Einheiten mit übergeordneter Auftragsablauf-

---

[1] vgl. Kapitel 6.3.1

[2] vgl. Geitner: *CIM-Handbuch*; S. 334

[3] ebd.; S. 342

[4] ebd.; S. 358

steuerung, Werkstück- und Werkzeugversorgung. Ihr wesentliches und kennzeichendes Merkmal ist die Zusammensetzung als Kombination autonomer Subsysteme mit je eigener Steuerungseinheit, die durch ein offenes Kommunikationsnetz gekoppelt sind. Aufgrund der modularen Struktur des Gesamtsystems läßt sich so eine hochflexibel automatisierte Fertigung realisieren.

Durch die Einbeziehung der betrieblichen Planungsebene können solche Systeme zu geschlossenen CIM-Strukturen erweitert werden. Die Durchführung aller einbezogenen betrieblichen Abläufe erfolgt in so strukturierten rechnerintegrierten Fabriken zweckmäßig durch Verteilung der Aufgaben auf mehrere hierarchisch aufgebaute Steuerungs- und Überwachungsebenen [1]. Die identifizierbaren Ebenen sind

- die Planungsebene,
- die Leitebene,
- die Zellenebene,
- die Steuerungsebene und
- die Aktor- und Sensorebene.

Zum Grundprinzip dieses in Bild 10 dargestellten Ebenenmodells gehört, daß die Systeme jeder Ebene immer so lange eigenverantwortliche Steuerungsaktivitäten durchführen kann, wie die Vorgaben der nächsthöheren Ebene eingehalten werden.

Auf der untersten, der Aktor- und Sensorebene, finden die eigentlichen Bearbeitungs-, Transport- und Lagervorgänge statt.

NC-Steuerungen, SPS, Robotersteuerungen u.ä. bilden die Steuerungsebene, die die erste Rechnerebene ist. Beide Ebenen sind durch E-/A-Systeme oder Feldbusse gekoppelt.

Darüber liegt die Zellenebene. Sie umfaßt z.B. Zellen- oder Inselrechner, die entsprechende Teilsysteme steuern. Zu den weiteren Aufgaben dieser Ebene gehören die Protokollierung von Prozeßdaten und die Weitergabe von Meldungen an die übergeordnete Ebene.

Auf der Leitebene ist jeweils eine Gruppe von Fertigungsinseln, -zellen oder -systemen zu koordinieren. Die hierfür eingesetzten Fertigungsleitrechner stellen die notwendigen Verbindungen zu den Systemen der Zellenebene her. Sie müssen den Auftragsdurchlauf unter Berücksichtigung der aktuellen Zustände aller Subsysteme steuern und überwachen. Weiter sind Produktionsdaten zu erfassen und Betriebsstatistiken zu führen.

---

[1] vgl. Geitner: *CIM-Handbuch*; S. 363

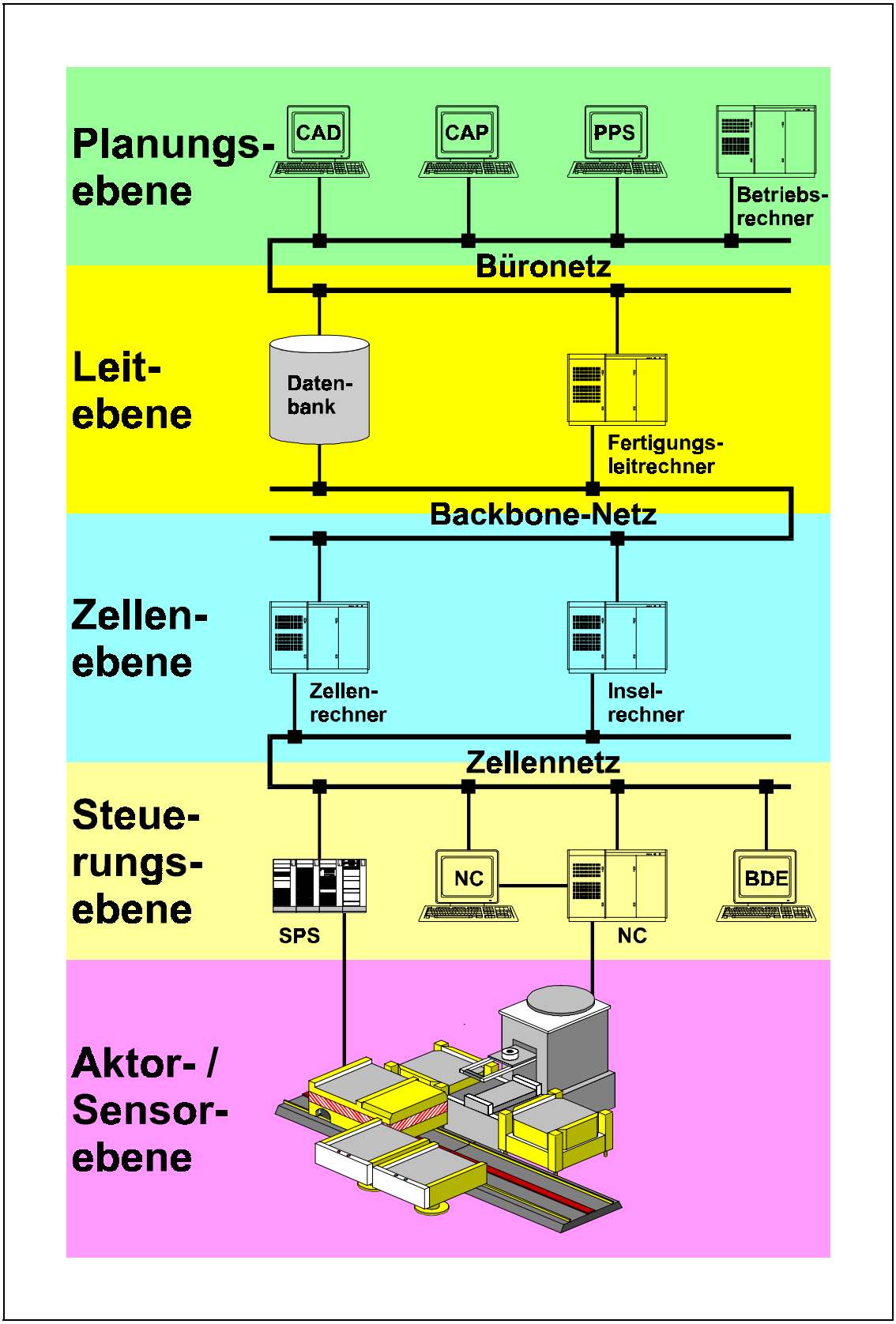

Bild 10: Strukturebenen der rechnerintegrierten Fertigung

Die oberste Ebene ist die Planungsebene, die die komplette Produktionsstätte umfaßt. Wiederum sind die unterlagerten Ebenen zu koordinieren. Außerdem sind hier die übergeordnete Produktionsplanung und -steuerung (PPS-Systeme) und die technischen Planungssysteme wie CAD und CAP angebunden.

Die zentrale Datenbank ist so in das Gesamtsystem eingeordnet, daß von allen Führungsebenen auf die dort gespeicherten Informationen zugegriffen werden kann.

Der auf und zwischen den Hierarchieebenen abzuwickelnde Datenverkehr unterscheidet sich zum einen hinsichtlich Umfang und Häufigkeit des Informationsaustauschs und zum anderen durch die verschiedenen Echtzeitanforderungen [1].

Die Echtzeitanforderungen sind auf den unteren prozessnahen Ebenen sehr streng und nehmen zu höheren Ebenen hin ab.

Die zu übertragenden Datenmengen sind auf den unteren Ebenen eher gering und nehmen zu höheren Ebenen hin zu. Mit zunehmender Hierarchie überwiegt dabei die Übertragung von Dateien (Files) gegenüber der von Nachrichten bzw. Signalen. Die Frequenz des Datenaustauschs schließlich ist auf den unteren Ebenen im allgemeinen höher als auf den oberen.

[1] vgl. Geitner: *CIM-Handbuch*; S. 363

6 Logistik

Die von produzierenden Unternehmen angebotenen Güter werden, wie in den vorangegangenen Abschnitten dargelegt, in Fabriken hergestellt. Die Zahl und Reihenfolge der zur Herstellung eines Produktes notwendigen Schritte werden üblicherweise in Form von Arbeits- und Montageplänen beschrieben. Diese enthalten Informationen über die wertschöpfenden Elemente des Herstellungsprozesses, d.h. über die Bearbeitungs- und Montagevorgänge. Zur Durchführung eines solchen Vorgangs an einem bestimmten Arbeitsplatz müssen dort die nötigen Teile, die erforderlichen Werkzeuge und Hilfsmittel und Bearbeitungsinformationen vorhanden sein. Alle hierfür erforderlichen Operationen und Vorkehrungen werden unter dem Begriff Logistik subsumiert.

6.1 Ziele, Aufgaben und Umfang

Gegenstand der Logistik ist die Gestaltung des Materialflußes vom Lieferanten bis zum Kunden. Sie umfaßt weiterhin die Gestaltung des Informationsflußes, der zur Steuerung des Materialflußes notwendig ist.

"*Ziel ist, die 'sechs R der Logistik' zu realisieren, d.h.*

- *die richtige Menge,*
- *die richtigen Objekte,*

- *am richtigen Ort,*
- *zum richtigen Zeitpunkt,*
- *in der richtigen Qualität,*
- *zu den richtigen Kosten*

zur Verfügung zu stellen [1]."

Die Logistik ist also ein wesentlicher Teil der Leistungserstellung eines Unternehmens. Im Gegensatz zu Bearbeitung und Montage erfolgt jedoch in der Logistik in der Regel keine Änderung der Produkteigenschaften [2]. Logistikleistungen sind daher Dienstleistungen.

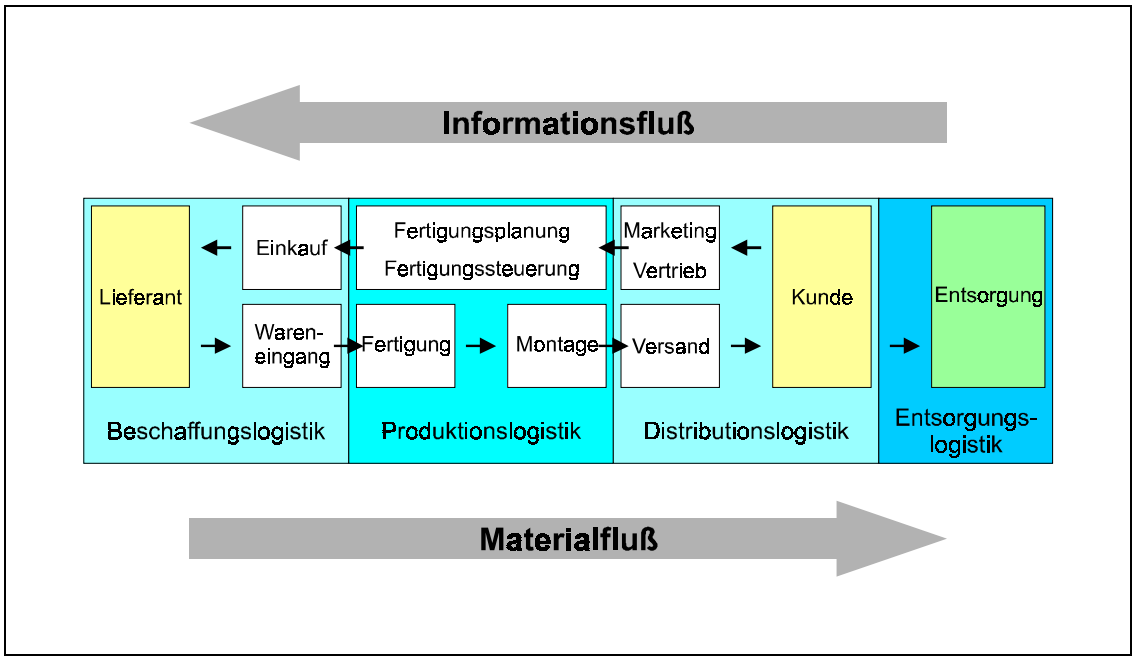

Bild 11: Die logistische Kette

Die für die betriebliche Leistungserstellung erforderlichen Logistikleistungen lassen sich in die Bereiche Beschaffungs-, Produktions-, Distributions- und Entsorgungslogistik einteilen. Sie bilden insgesamt die in Bild 11 dargestellte logistische Kette.

Die Beschaffungslogistik umfaßt die organisatorischen und technischen Abläufe zur Zulieferung von Teilen. Dazu gehören die Lieferantenauswahl, die Bestellabwicklung und die Festlegung und Durchführung von Lieferungen nach Mengen und Terminen.

---

[1] Koether: *Technische Logistik*; S. 1

[2] Grenz- und Ausnahmefälle sind z.B. Reifevorgänge.

Die Produktionslogistik umfaßt die Planung und Abwicklung des innerbetrieblichen Material- und Informationsflußes. Hierzu gehören im wesentlichen die Auswahl und Steuerung von Fördertechnik, Puffern und Lagern einschließlich der zugehörigen Hilfsmittel, die Behandlung von Ausfällen von Fertigungseinrichtungen und Logistiksystemen und die Festlegung von Losgrößen, Transporteinheiten und Fertigungsablaufprinzipien.

Die Distributionslogistik ist gewissermaßen das Gegenstück zur Beschaffungslogistik. Sie umfaßt daher wiederum die Abwicklung von Bestellungen und Lieferungen. Hinzu können noch die Planung und Durchführung von Schulungen, Wartungen und Instandhaltungen (auch beim Kunden), die Vorhaltung von Ersatzteilen sowie Montage und Inbetriebnahme von Produkten beim Kunden kommen.

Zur Entsorgungslogistik rechnen die Abholung, Verwertung und/oder Beseitigung von Verpackungsmaterial, Rest- und Abfallstoffen. Dies umfaßt auch die Abholung und Behandlung ausgemusterter Produkte von Kunden.

6.2 Wirtschaftliche Aspekte

Wie die Bearbeitung erfordert die Logistik Aufwendungen und bringt dafür Nutzen. Eine ausführlicher Definition der Rendite als

$$\textit{Rendite} = \frac{\textit{Gewinn}}{\textit{Kapitaleinsatz}} = \frac{\textit{Umsatz} - (\textit{fixe Kosten} + \textit{variable Kosten})}{\textit{Anlagevermögen} + \textit{Umlaufvermögen}} \quad (1)$$

schlüsselt die beteiligten Größen genauer auf [1]. Die Logistik beeinflußt Teile aller dieser Größen.

Als Teil der Leistungserstellung produzierender Unternehmen beeinflußt die Logistik den Umsatz. Auf der höchsten Betrachtungsebene ermöglicht sie, aufgrund der Zuständigkeit für die Distribution, erst die Lieferung von Produkten an Kunden. Auf unteren Ebenen stellt sie die Durchführung von Arbeitsvorgängen durch die Bereitstellung von Material, Werkzeugen und Bearbeitungsinformation am jeweiligen Arbeitsplatz sicher.

[1] vgl. Koether: *Technische Logistik*; S. 1

Variable Logistikkosten entstehen beispielsweise für die Durchführung von Transporten. Fixe Logistikkosten entstehen z.B. für Abschreibungen auf Lager- und Transporteinrichtungen. Diese sind Teil des Anlagevermögens des Unternehmens. Das in den Materialien, Halbfabrikaten und Fertigprodukten gebundene Kapital rechnet zum Umlaufvermögen. Diese Bestände sind der eigentliche Verantwortungsbereich der Logistik.

Da das in den Beständen gebundene Kapital verzinst werden muß, verursachen diese Kosten. Die Höhe des Umlaufvermögens bzw. die Menge der Bestände ist damit für die Logistik von herausragender Bedeutung. Die Durchlaufzeiten bestimmen dabei die Dauer der Kapitalbindung. Neben den bereits angesprochenen Rückwirkungen auf die Lieferfähigkeit und damit die Marktchancen des Unternehmens [1] haben sie daher, wie Bild 12 erläutert, auch erheblichen Einfluß auf die Bestandskosten.

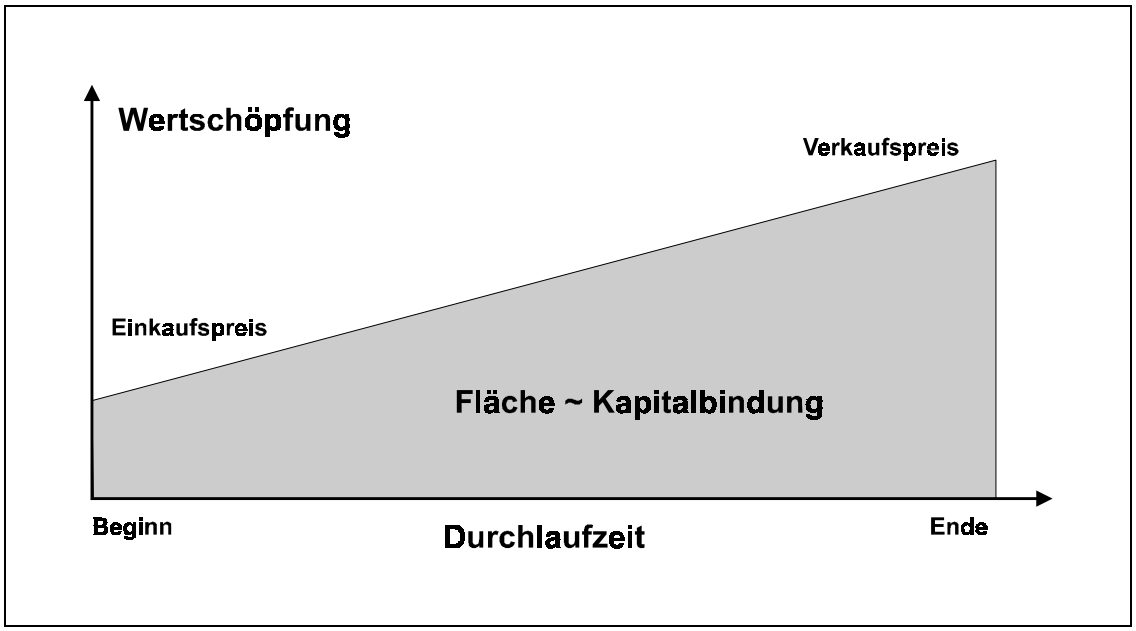

Bild 12: Einfluß der Durchlaufzeit auf das Umlaufvermögen

Für die Logistik ergibt sich daraus als Ziel, möglichst geringe Bestände an möglichst gering bewerteten Teilen und auch diese nur über möglichst kurze Zeiträume vorzuhalten. Wegen der damit verbundenen Kosten (s.o.) wird außerdem angestrebt, Lager- und Transporteinrichtungen knapp zu dimensionieren und die vorhandenen Anlagen hoch auszulasten. Schließlich ist auch eine hohe Termintreue ein Ziel der Logistik, wie vor allem in der Distribution sichtbar wird, wo die rechtzeitige Erledigung von Transportaufträgen offensichtliche Voraussetzung für die Einhaltung von Lieferterminen ist.

[1] vgl. Kapitel 2.1 und 4.2

Insgesamt zeigt sich, daß das in Kapitel 4.2 dargestellte Zielsystem der Produktion auch für die Logistik gültig ist. Da sie als integraler Teil der betrieblichen Leistungserstellung eingeordnet wurde, entspricht dies den Erwartungen.

6.3 Logistikgerechte Fertigung

Die im Rahmen der Logistik anfallenden Aufgaben lassen sich in technische und organisatorische Teile untergliedern [1]. Die technischen Aufgabenanteile umfassen dabei im wesentlichen die Auswahl, Beschaffung und Aufstellung bzw. Anordnung von Betriebsmitteln, beispielsweise von Lager- oder Fördertechnik. Zum organisatorischen Teil gehören insbesondere die zum Betrieb der Anlagen eingesetzten Verfahren, beispielsweise die Systeme zur Steuerung der Anlagen und zur Fertigungsplanung und -steuerung.

Gelegentlich werden auch andere Kategorisierungen verwendet. So ist es ebenfalls üblich, planerische und dispositive bzw. fertigungsvorbereitende und -begleitende Aufgaben zu unterscheiden. In jedem Fall aber können alle Aufgabenteile nur im Zusammenhang betrachtet werden, da sie sich wechselseitig beeinflussen. So ist die Fertigungsplanung und -steuerung offensichtlich auf die Nutzung der von den Fertigungsanlagen gegebenen Möglichkeiten (z.B. hinsichtlich der verfügbaren Kapazitäten) beschränkt.

Ausgangspunkt aller Überlegungen zur Gestaltung von Fertigungsanlagen ist stets das vorgesehene Produktionsprogramm, d.h. das herzustellende Teilespektrum und das zugehörige Mengengerüst. Darauf aufbauend ist zum einen die Fertigungsstruktur festzulegen und zum anderen sind Maßnahmen zur Kontrolle von Beständen und Durchlaufzeiten zu entwickeln und umzusetzen.

6.3.1 Fertigungsstruktur

Die Fertigungsstruktur beschreibt die Anordnung der Maschinen, Anlagen und Fertigungsbereiche [2]. Sie legt damit den Durchlauf der Fertigungsaufträge durch die Produktion fest. Aus der Fertigungsstruktur ergeben sich bestimmende Rückwirkungen auf die zu erwartende Kapazitätsauslastung und die Durchlaufzeiten der Fertigungsaufträge.

---

[1] vgl. Koether: *Technische Logistik*; S. 1

[2] ebd.; S. 105. Weitere gängige Begriffe hierfür sind Fertigungsprinzip oder Fertigungsablaufprinzip.

Die Fertigungsstruktur kann sich an den eingesetzten Fertigungstechnologien und am Durchlauf der Aufträge orientieren. Technologieorientierte Fertigungsstrukturen bauen auf dem Verrichtungsprinzip auf, durchlauforientierte Strukturen dagegen folgen dem Fließprinzip [1].

In technologieorientierten Fertigungsstrukturen sind Maschinen gleicher oder ähnlicher Technologie in einer Abteilung zusammengefaßt. Typisches Beispiel ist eine Werkstattorganisation mit (beispielsweise) Dreherei, Schleiferei und Oberflächenbearbeitung. Die Anlagen innerhalb dieser Abteilungen können sich weitgehend ersetzen, so daß Nachfrageschwankungen zwischen verschiedenen Aufträgen bzw. Teilen innerhalb der Bereiche ausgeglichen werden können. In einer technologieorientierten Fertigungsstruktur können die Kapazitäten der Anlagen daher tendenziell hoch ausgelastet werden [2].

In durchlauforientierten Fertigungsstrukturen werden die Maschinen und Bereiche entsprechend dem Fertigungsfortschritt angeordnet. In einer Abteilung wird also beispielsweise das Teil A und in einer zweiten das Teil B hergestellt. Nachfrageschwankungen zwischen den Teilen A und B können daher innerhalb einer Abteilung nicht ausgeglichen werden, so daß zum Abdecken von Bedarfsspitzen eine höhere Kapazität erforderlich ist, die im Normalfall kaum ausgelastet werden kann. Da andererseits zwischen den Teilen keine Konkurrenz um die Maschinen besteht, sind die Durchlaufzeiten der Aufträge tendenziell geringer als in technologieorientierten Fertigungsstrukturen [3].

Die beiden grundsätzlichen Alternativen technologie- bzw. durchlauforientierter Fertigungsstrukturen lassen sich für den praktischen Einsatz weiter aufschlüsseln. Es werden, hier geordnet nach zunehmender Durchlauf- und abnehmender Technologieorientierung, die folgenden Ablaufprinzipien unterschieden [4]:

- Baustellenfertigung,
- Fertigungsanlage,
- Werkstattfertigung,
- Flexible Fertigungszelle,
- Fertigungsinsel,
- Zentrallager-Fertigung,
- Flexibles Fertigungssystem,

---

[1] vgl. REFA: *Methodenlehre des Arbeitsstudiums, Teil 3*

[2] vgl. Koether: *Technische Logistik*; S. 105

[3] ebd.

[4] ebd.; S. 106 ff.

- Flexible Fertigungslinie und
- Fließlinie.

Die wesentlichen Kriterien bei der Festlegung der Fertigungsstruktur sind die Kapitalbindung im Anlage- und Umlaufvermögen, die Durchlaufzeit als Faktor für die Kapitalbindung und den Markterfolg [1] und die Flexibilität, mit der auf Änderungen in Teilespektrum und Mengengerüst reagiert werden kann bzw. muß [2].

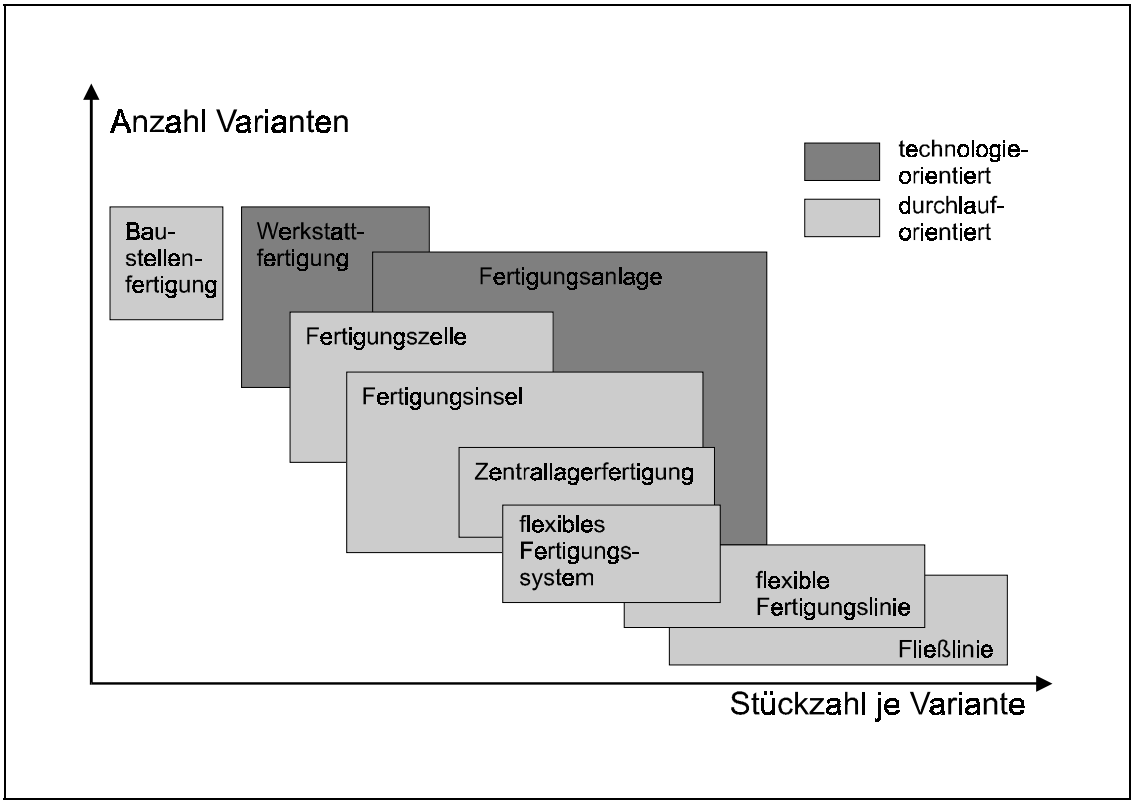

Bild 13: Einsatz verschiedener Fertigungsstrukturen

Wie Bild 13 zeigt, sind technologieorientierte Fertigungsstrukturen besonders bei vielen Varianten und geringen Stückzahlen wirtschaftlich, während durchlauforientierte Prinzipien um so wirtschaftlicher sind, je geringer die Variantenvielfalt und je größer die Stückzahlen sind.

Die heute zunehmende Bedeutung kurzer Liefer- und damit Durchlaufzeiten [3] führt zu erhöhten Anforderungen an die dispositive Logistik, die in technologieorientierten

---

[1] vgl. Kapitel 2.1

[2] vgl. Koether: *Technische Logistik*; S. 116

[3] vgl. Kapitel 2.1 und 4.2

Fertigungsstrukturen auch bei Nutzung moderner Produktions-Planungs- und Steuerungssysteme (PPS) normalerweise nicht hinreichend erfüllt werden können [1]. In der Folge werden auch in der Teilefertigung zunehmend durchlauforientierte Strukturen wie Fertigungszellen oder Fertigungsinseln eingerichtet.

6.3.2 Bestände und Durchlaufzeiten

In Kapitel 4.2 wurde die große und zunehmende Bedeutung niedriger Bestände und kurzer Durchlaufzeiten diskutiert. Um diese Ziele zu erreichen, sind technische und organisatorische Maßnahmen möglich.

In Frage kommende organisatorische Maßnahmen sind die Gewinnung zuverlässiger Informationen über den Auftragsdurchlauf, z.B. durch Betriebs- oder Maschinendatenerfassung (BDE bzw. MDE) und der Einsatz eines geeigneten Systems zur Fertigungsplanung und -steuerung, das insbesondere eine reaktionsfähige Feinsteuerung bietet [2].

Mögliche technische Maßnahmen sind die Verringerung der Fertigungsstufen, die Reduzierung der Losgrößen, der Aufbau durchlauforientierter Fertigungsstrukturen, die Gewährleistung einer hohe Prozeßsicherheit und die gleichzeitige, parallele Fertigung der Baugruppen eines Produkts z.B. in einer Just-In-Time-Fertigung [3]. Auf diese möglichen Maßnahmen soll in den folgenden Abschnitten jeweils kurz eingegangen werden.

6.3.2.1 Verringerung der Fertigungsstufen

Eine Fertigungsstufe umfaßt eine Bearbeitung oder eine Folge von Bearbeitungen mit einer gemeinsamen Eingangswarteschlange. Dieser Vorpuffer entkoppelt die Fertigungsstufe von (im Ablauf) vorherliegenden und gleicht Verfügbarkeitsschwankungen aus.

Liegezeiten in Vorpuffern sind mit Anteilen von 75% und mehr an den Durchlaufzeiten der diese bestimmende Faktor [4]. Je weniger Fertigungsstufen ein Produkt durchläuft, um so geringer werden demzufolge die Aufenthaltszeiten in den Warteschlangen sein. Ziel ist deshalb, ein Teil möglichst komplett in wenigen Fertigungsstufen zu bearbeiten.

---

[1] vgl. Koether: *Technische Logistik*; S. 117

[2] ebd.; S. 119 ff.

[3] ebd.

[4] vgl. Wiendahl: *Belastungsorientierte Fertigungssteuerung*; S. 48 ff.

Die Komplettbearbeitung kann im Kostenvergleich trotz der typischerweise komplexeren und daher teureren Maschine schon deswegen günstiger sein, weil gegenüber der konventionellen Bearbeitung Nebenzeiten z.B. für mehrfaches Auf- und Abspannen der Werkstücke eingespart werden können [1]. Ihr wesentlicher Vorteil ist aber die Reduzierung der Durchlaufzeit auf etwa 25% des Vergleichswertes der konventionellen Fertigung [2]. Die durch den zugrundeliegenden Fertigungsauftrag bedingte Kapitalbindung reduziert sich im gleichen Maß [3].

Auch wenn eine Komplettbearbeitung nicht möglich ist, kann die Zahl der wirksamen Fertigungsstufen reduziert werden. Werden beispielsweise Maschinen so verkettet, daß die Werkstücke direkt weitergegeben werden, so wirkt der verkettete Bereich insgesamt wie eine Fertigungsstufe. Dies kann bis zur Fließlinie gehen. Entscheidend ist, daß der verkettete Bereich aus nur einem Eingangspuffer gespeist wird.

6.3.2.2 Reduzierung der Losgrößen

Da immer nur ein Teil aus einem Fertigungslos bearbeitet wird, während alle anderen warten, wachsen die Wartezeiten der Teile mit den Losgrößen. Diese hängen allerdings wesentlich von den Rüstzeiten bzw. -kosten ab und sind daher nicht beliebig reduzierbar: Je höher die anfallenden Rüstkosten und je höher die erforderlichen Rüstzeiten sind, desto größer ist die wirtschaftliche Losgröße (s.u.).

Kürzere Rüstzeiten ermöglichen also kleinere Losgrößen, wodurch die Bestände (und damit auch die Durchlaufzeiten) sinken. Darüberhinaus setzen kürzere Rüstzeiten direkt Fertigungskapazität frei.

Eine Verkürzung der Rüstzeiten kann einerseits durch organisatorische Maßnahmen wie z.B. die bessere Koordination der Rüstaktivitäten und unterschiedliche Pausenregelungen für Maschinenbediener und Einrichter und andererseits durch technische Maßnahmen wie z.B. hauptzeitparalleles Rüsten und die Verwendung von Rüsthilfen und Schnellwechseleinrichtungen erreicht werden [4].

---

[1] Aus der Bearbeitung in nur einer Aufspannung resultiert außerdem in der Regel eine bessere Maßhaltigkeit der Teile, wodurch engere Fertigungstoleranzen realisiert werden können.

[2] vgl. Koether: *Technische Logistik*; S. 122

[3] vgl. Kapitel 6.2

[4] vgl. Koether: *Technische Logistik*; S. 124

Die Berechnung der sogenannten wirtschaftlichen Losgröße erfolgt in der Industrie auch heute noch weitgehend nach der aus Überlegungen von Harris und Andler abgeleiteten "Andlerschen Losgrößenformel" [1]. In der Praxis wird allerdings vom Ergebnis der Formelanwendung im Interesse eines schnelleren Teiledurchlaufs häufig abgewichen. Das Verfahren stößt zunehmend auf Kritik, da es den Einfluß der Losgröße auf die Bestände und damit auf die Durchlaufzeiten weder für das betrachtete noch für andere um die Kapazitäten konkurrierende Lose berücksichtigt [2].

Deshalb wird vielfach die Forderung "Losgröße = 1" erhoben, die aber, zumal in technologieorientierten Fertigungsstrukturen, häufig nicht zu erfüllen ist, da auf Rüstvorgänge nicht immer verzichtet werden kann. Im übrigen entstünden gegenüber der Losfertigung so auch wesentlich mehr Fertigungsaufträge, die zu steuern sind.

Als pragmatische Schlußfolgerung ergibt sich, daß außer der mittleren Durchführungszeit [3] auch die Spannweite dieser Zeiten begrenzt werden sollte [4]. Dies bedeutet, daß die Harmonisierung der Arbeitsinhalte der Fertigungslose anzustreben ist, um insgesamt reduzierte Bestände und Durchlaufzeiten zu erreichen.

6.3.2.3 Durchlauforientierte Fertigungsstrukturen

Unter dem Aspekt minimaler Bestände und Durchlaufzeiten ist die Fließlinie die optimale Fertigungsstruktur. In einer solchen Linie werden die Werkstücke sofort nach dem Ende jeder Bearbeitung an die folgende Station weitergegeben. Bestände zwischen den Arbeitsgängen entstehen nicht mehr, so daß die Fließlinie einer Fertigungsstufe entspricht.

Darüber hinaus werden die Arbeitsinhalte durch Abtaktung auf die einzelnen Stationen möglichst gleichmäßig verteilt. Die Durchlaufzeit entspricht dem Produkt aus der Anzahl der Stationen und der Taktzeit (Bearbeitungszeit).

Die beiden zugrundeliegenden Prinzipien der Harmonisierung der Durchlaufzeiten durch die einzelnen Stationen bzw. Fertigungsstufen einerseits und der Weitergabe kleiner Transportlose andererseits, die durch eine Verkettung oder wenigstens durch kurze

---

[1] vgl. Wiendahl: *Betriebsorganisation für Ingenieure*; S. 293 ff. Dort wird auch eine ausführlichere Darstellung der Zusammenhänge gegeben.

[2] ebd.; S. 227 ff.

[3] vgl. Kapitel 4.2

[4] vgl. Wiendahl: *Belastungsorientierte Fertigungssteuerung*; S. 278 ff.

Entfernungen zwischen den beteiligten Maschinen erleichtert wird, gelten in abgeschwächter Form für alle durchlauforientierten Fertigungsstrukturen [1]. Daher sind in diesen Fällen kürzere Durchlaufzeiten und geringere Bestände zu erwarten als in technologieorientierten Fertigungsstrukturen.

6.3.2.4 Hohe Prozeßsicherheit

Sicherheitsbestände in den Vorpuffern einzelner Fertigungsstufen dienen dem Ausgleich von Qualitätsmängeln und von Schwankungen der Verfügbarkeit dieser und direkt vorgelagerter Stufen, d.h. der Störungskompensation. Um solche Sicherheitsbestände reduzieren oder ganz abschaffen zu können, müssen also Qualitätsmängel und Störungen vermieden werden.

Die Verfügbarkeit von Fertigungsanlagen und damit die Prozeßsicherheit kann z.B. durch eine in die Produktion integrierte Qualitätssicherung und eine vorbeugende und nach Möglichkeit produktionssynchrone Wartung und Instandhaltung erhöht werden [2].

6.3.2.5 Just-in-time-Fertigung

Die Just-in-time-Fertigung zielt auf die bestandsminimale Zulieferung von Teilen. Solche Systeme existieren hauptsächlich in Montagebereichen, da hier die Teile bereits weitgehend veredelt sind und daher die Kapitalbindung am größten ist. Die benötigten Teile werden gerade rechtzeitig (eben just in time), also weder zu früh noch zu spät, zum Verbraucher bzw. zur Einbaustation geliefert [3].

Eine Just-in-time-Fertigung erfordert eine entsprechende technische und organisatorische Gestaltung. Im Fertigungsfluß wird ein Leitprodukt definiert (in einer Automobilmontage z.B. die Karosserie). Dieses hat im einfachsten Fall einen exakt definierten Fertigungsdurchlauf ohne Verzweigungen, so daß seine Durchlaufzeit determiniert ist.

Da sich die Leitprodukte nicht überholen, kann der Bedarfszeitpunkt für ein bestimmtes Einbauteil aus der Durchlaufzeit und der Reihenfolge der Leitprodukte bestimmt werden. Die Lieferanten bekommen eine Meldung über die Reihenfolge und Bedarfszeitpunkte der Teile und liefern diese entsprechend zu.

---

[1] vgl. Kapitel 6.3.1

[2] vgl. Koether: *Technische Logistik*; S. 137 ff. und 159 ff.

[3] ebd.; S. 126 ff.

Im Zeitraum zwischen Reihenfolgemeldung und Einbau müssen die Zulieferteile gefertigt, transportiert und bereitgestellt werden. Um geringfügige Störungen in Teilbereichen nicht auf das Gesamtsystem durchschlagen zu lassen, entkoppeln kleine Puffer (im Minutenbereich) die Flüsse des Leitprodukts und der Zulieferteile. Da die Vorlaufzeiten kurz sind, müssen bei einer Just-in-time-Fertigung die Fertigungssysteme sowohl des Leitprodukts wie der Zulieferteile zuverlässig und mit hoher Qualität produzieren.

Auch die Transporte der Zulieferteile müssen mit hoher Zuverlässigkeit durchgeführt werden können. Wenn die erforderlichen Transportwege lang sind und durch den öffentlichen Verkehrsraum führen, müssen Just-in-time-Konzepte in der Regel auf Teile beschränkt werden, die leicht nachrüstbar sind, wie z.B. Autositze. Teile, die in der zur Verfügung stehenden Zeit nicht (vollständig) hergestellt werden können, müssen bis zu einem geeigneten Grad vorgefertigt und innerhalb der Vorlaufzeit dann fertiggestellt werden. Beispielsweise werden in der Automobilindustrie die Motoren häufig erst kurz vor dem Einbau in das Fahrzeug im Motorenendmontagewerk komplettiert [1].

Sofern bei einer Just-in-time-Fertigung vom einfachen Grundfall abgewichen wird, so daß Reihenfolgeänderungen beim Leitprodukt oder den Zulieferteilen auftreten können, muß der Aufwand zur Systemgestaltung weiter erhöht werden. Beispielsweise könnten Vorlaufzeiten verkürzt oder Leitprodukte oder Baugruppen umsortiert werden. Da eine Reihenfolgeänderung eine Vielzahl von Umsortierungen erfordern kann, werden sie nur in Fällen zugelassen, in denen sie unvermeidlich sind.

6.4 Logistik und Produktgestaltung

Die Logistik steht immer in Wechselwirkung mit anderen Bereichen des Unternehmens und kann nicht losgelöst von diesen betrachtet werden. Großen Einfluß auf die Logistik haben neben den bereits in Kapitel 4.1 behandelten Aspekten der Kunden- und Lieferantenbeziehungen und der Wahl des Fertigungsstandorts vor allem die Gestaltung der einzelnen Teile, Baugruppen und Produkte, die Produktgliederung und die Anzahl der angebotenen Varianten.

Bei der Konstruktion einzelner Teile wird heute großer Wert auf eine fertigungsgerechte Gestaltung gelegt. So werden z.B. “Near-Net-Shape”-Teile konzipiert, die bereits nach der Erzeugung durch Ur- oder Umformen kaum noch von der Endkontur abweichen, so daß in

[1] Dies war z.B. bei den Anlagen der Fall, die im Rahmen der Simulationsstudien “Triebwerkaufrüstung inclusive Hochzeitsbereich” und “Motorenendmontage ...” (vgl. Anhang Projekte) untersucht wurden.

der Fertigbearbeitung nur noch geringe Aufmaße abzunehmen sind [1]. Weiter wird angestrebt, die gesamte spanende Formgebung in nur einem Schnitt oder wenigstens in einer Aufspannung durchführen zu können [2]. Besonders in Kombination mit dem Einsatz moderner Werkzeuge läßt sich so die Wirtschaftlichkeit der Fertigung deutlich steigern. Gleichzeitig ergeben sich Änderungen in den Anforderungen an die Logistik. Hier sind vor allem höhere Frequenzen bei der Ver- und Entsorgung von Arbeitsstationen und mögliche Reduzierungen der Anzahl der Fertigungsstufen [3] zu nennen.

Veränderte Anforderungen an die Produktgestaltung gibt es auch aus dem Bereich der Montage. Die montagegerechte Produktgestaltung zielt vor allem darauf ab, durch die Minimierung von Montagezeiten (vor allem bei manuellen Tätigkeiten) und den Einsatz einfacher und zuverlässiger Betriebsmittel für automatisierte Montageschritte wirtschaftliche Fortschritte in der Montage zu erreichen [4]. Häufig wird auch eine ergonomischere Gestaltung der Montagetätigkeiten angestrebt [5]. Die sich als Konsequenz ergebenden veränderten Anforderungen an die Logistik sind en eben genannten vergleichbar.

Die transport- und lagergerechte Gestaltung der Produkte mit guter Platznutzung, geringer Empfindlichkeit und der Möglichkeit zur Verwendung von Standardbehältern trägt ebenfalls wesentlich zur Reduzierung des Logistikaufwands bei.

Hinsichtlich der Produktgliederung ist aus Logistiksicht eine Baukastenkonstruktion wünschenswert. Die dabei bestehende Gleichheit von Funktions- und Baugruppen ermöglicht, die Fertigung analog zum Produkt zu strukturieren und zu modularisieren. Technologische Änderungen von Produkt oder Prozess schlagen sich dann in der Regel nur in einer Baugruppe und in einem Fertigungsbereich nieder.

Varianten dienen der Anpassung der Produkte an individuelle Kundenanforderungen. Aus der Sicht von Marketing und Vertrieb bieten Varianten Möglichkeiten zur Umsatzsteigerung durch die Gewinnung zusätzlicher Kunden. Diesem Nutzen muß jedoch der anfallende höhere Logistikaufwand (z.B. durch die Verwaltung zusätzlicher Teile) gegenübergestellt werden.

---

[1] vgl. Tikal: *Produktionstechnik 2*; S. 74

[2] ebd.

[3] vgl. Kapitel 6.3.2.1

[4] vgl. Eversheim: *Organisation in der Produktionstechnik; Bd. 4 Fertigung und Montage*; S. 144 ff.

[5] vgl. Tikal et al.: *Alternative Montagekonzepte am Beispiel einer PKW-Tür*; S. 4

6.5 Planung von Fertigungsbereichen

Die Neu- oder Umplanung von Fertigungsbereichen ist eine Aufgabe, in die alle bisher diskutierten Aspekte der Logistik Eingang finden. Da die Planung für den späteren Betrieb viele wesentliche und üblicherweise nur schwer veränderliche Rahmenbedingungen vorgibt, kommt ihr eine grundlegende Bedeutung für die Wirtschaftlichkeit der Fertigung in Bezug auf das Zielsystem der Produktion [1] zu. So wie auf der Produktseite ein wesentlicher Teil der Kosten durch die Konstruktion festgelegt wird, gilt auch auf der Produktionsseite, daß die der eigentlichen Fertigung vorgelagerten Aktivitäten erheblichen Einfluß auf die Herstellungskosten haben.

Da Fertigungseinrichtungen immer für die Herstellung bestimmter Produkte in (wenigstens ungefähr) festgelegten Mengen geplant werden, kann eine konkrete Planung nicht unabhängig von den Produkten betrachtet oder durchgeführt werden, sondern muß mit der Festlegung eines Produktionsprogramms beginnen.

Im Zuge des weiteren Vorgehens sind dann die Fertigungstechnologien, d.h. die zum Einsatz kommenden Bearbeitungsverfahren auszuwählen. Die dementsprechend erforderlichen Maschinen sind auszuwählen und zu Fertigungsstufen bzw. auf einer höheren Ebene zu einer Fertigungsstruktur zusammenzufassen.

Diese Schritte werden in der Praxis iterativ und mit zunehmender Detailgenauigkeit durchgeführt. Häufig werden zunächst auch mehrere Varianten betrachtet, von denen schließlich eine ausgewählt wird. Um zu einer fundierten Entscheidung zu gelangen und wegen ihrer erheblichen Bedeutung kommen dabei oft aufwendige Verfahren wie z.B. die Nutzwertanalyse zum Einsatz [2].

Wenn der Planungsprozeß den erforderlichen Reifegrad erreicht hat, beginnt die Layoutplanung. Dabei sind verschiedene Teilpläne, z.B. für Gebäude und Flächen, Materialfluß, Ver- und Entsorgung und Sozialflächen in einem abgestimmten Vorgehen zu entwickeln und zu integrieren [3]. Auch die Layoutplanung wird typischerweise iterativ mit zunehmender Genauigkeit durchgeführt. Bereits auf groben Stufen können dabei wichtige Erkenntnisse

---

[1] vgl. Kapitel 4.2

[2] vgl. Eversheim: *Organisation in der Produktionstechnik, Bd. 4 Fertigung und Montage*; S. 296 ff.

[3] vgl. Koether: *Technische Logistik*; S. 174 ff.

erzielt werden. Beispielsweise können aus Materialflußmatrix und Blocklayout lagegerechte Mengenflußbilder [1] erzeugt werden, die Aufschluß über zu erwartende innerbetriebliche Verkehrsströme geben.

Der Computereinsatz ist in der Fabrikplanung im Vergleich zur Produktentwicklung (noch) weniger verbreitet. Im Rahmen der Layoutplanung werden jedoch häufig CAD-Systeme und spezielle Layoutplanungssysteme eingesetzt. Sie unterstützen die Koordination und Integration der o.a. Teilplanungen, können Datenredundanzen verhindern und unterstützen die Bildung und Bewahrung einer einheitlichen Planungsgrundlage.

Die eingesetzten herkömmlichen Berechnungsverfahren sind üblicherweise statischer Natur oder operieren mit gemittelten oder in ähnlicher Weise zeitlich geglätteten Werten z.B. für Mengenströme [2]. Da Fabrikanlagen jedoch dynamische Systeme mit zeitabhängigem Zustand und Verhalten sind, stoßen solche Verfahren schnell an Grenzen. Aus diesem Grund ist die Simulation, auf die in Kapitel 7 näher eingegangen wird, heute ein verbreitetes und anerkanntes Hilfsmittel bei der Planung von Fabrikanlagen und anderen Materialfluß- und Logistiksystemen.

Aufgrund allgemein sinkender Produktlebensdauern und sich verkürzender Innovationszyklen [3] werden auch die für die Planung und Inbetriebnahme von Fertigungsanlagen zur Verfügung stehenden Zeiten immer geringer. Da die Produkte und Anlagen außerdem immer komplexer werden, erhöhen sich auch die Investitionssummen und -risiken, was wiederum zu gesteigerten Anforderungen an die Planungssicherheit führt.

Diesen Rahmenbedingungen kann mit der herkömmlichen konsekutiven Planung von (erst) Produkten und (danach) Fertigungsanlagen nicht mehr Rechnung getragen werden. Sie erfordern den Übergang zum Simultaneous Engineering, also zur gleichzeitigen und integrierten Entwicklung von Produkt und Fabrik.

---

[1] z.B. nach VDI-Richtlinie 3300: *Materialfluß-Untersuchungen*

[2] vgl. z.B. VDI-Richtlinie 3300: *Materialfluß-Untersuchungen* und Großeschallau: *Materialflußrechnung*

[3] vgl. Kapitel 3

III Materialflußsimulation

7 Übersicht

Dieses Kapitel gibt einen Überblick über die Materialflußsimulation. Insbesondere werden ihre Nutzungsmöglichkeiten, das Vorgehen bei ihrem Einsatz und die Charakteristika entsprechender Werkzeuge vorgestellt. Die Darstellung orientiert sich, insbesondere hinsichtlich der verwendeten Begriffe, an der VDI- Richtlinie 3633 [1].

7.1 Definitionen und Leitsätze

"*Simulation ist das Nachbilden eines Systems mit den darin ablaufenden dynamischen Prozessen in einem experimentierfähigen Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind* [2]."

Zur Simulation gehören Vorbereitung, Durchführung und Auswertung gezielter Experimente an Simulationsmodellen, die vereinfachte Nachbildungen geplanter oder existierender Systeme sind. Sie unterscheiden sich hinsichtlich der untersuchungsrelevanten Eigenschaften nur innerhalb eines vom Untersuchungsziel abhängigen Toleranzrahmens vom Original [3].

---

[1] *VDI 3633 Simulation von Logistik-, Materialfluß- und Produktionssystemen*

[2] ebd.; S. 3

[3] *ebd.*; S. 3

Simulationsexperimente werden mit Hilfe von Simulatoren durchgeführt, die Softwarewerkzeuge sind. Sie machen die Modelle lauf- und nutzungsfähig, indem sie die darin ablaufenden Prozesse und damit das dynamische Verhalten des Systems über die Zeit nachbilden.

Für den Einsatz der Simulation gelten die folgenden Leitsätze:

- Simulation sollte stets vor einer eventuellen Investition eingesetzt werden.
- Simulation setzt die vorherige Definition von Untersuchungszielen und eine Aufwandsabschätzung voraus.
- Vor der Simulation sollten die für den jeweiligen Anwendungsfall zur Verfügung stehenden analytischen Methoden ausgeschöpft werden.
- Simulation ist kein Ersatz für Planung.
- Die Abbildungsgenauigkeit der Modelle sollte nur so groß sein, wie es zur Zielerfüllung erforderlich ist.
- Simulationsergebnisse sind nur so "gut" wie die bei der Modellerstellung verwendeten Eingangsdaten. Eine fehlerhafte Datenbasis macht Simulationsergebnisse wertlos oder irreführend.

7.2 Nutzungsmöglichkeiten der Simulation

Die Simulation kann auf jeden Abschnitt der logistischen Kette [1] angewendet werden. Sie ist ein allgemein anerkanntes Hilfsmittel bei Planung, Realisierung und Betrieb technischer Systeme. Ursprünglich wurde sie vorwiegend zur Planungsabsicherung eingesetzt. Wie Bild 14 zeigt, wird sie mittlerweile durchgängig in allen Phasen des Planungs- und Realisierungsprozesses genutzt und findet zunehmend auch Anwendung in der Prozeßsteuerung während des Betriebs [2]. Die folgenden Abschnitte stellen die Nutzungsmöglichkeiten der Simulation in den verschiedenen Lebenszyklusphasen von Anlagen kurz dar [3].

---

[1] vgl. Kapitel 6.1

[2] *VDI-Richtlinie 3633 Blatt 1*; S. 2

[3] Die Darstellung orientiert sich an der ausführlicheren Behandlung dieser Thematik in der *VDI-Richtlinie 3633 Blatt 1*; S. 4 ff.

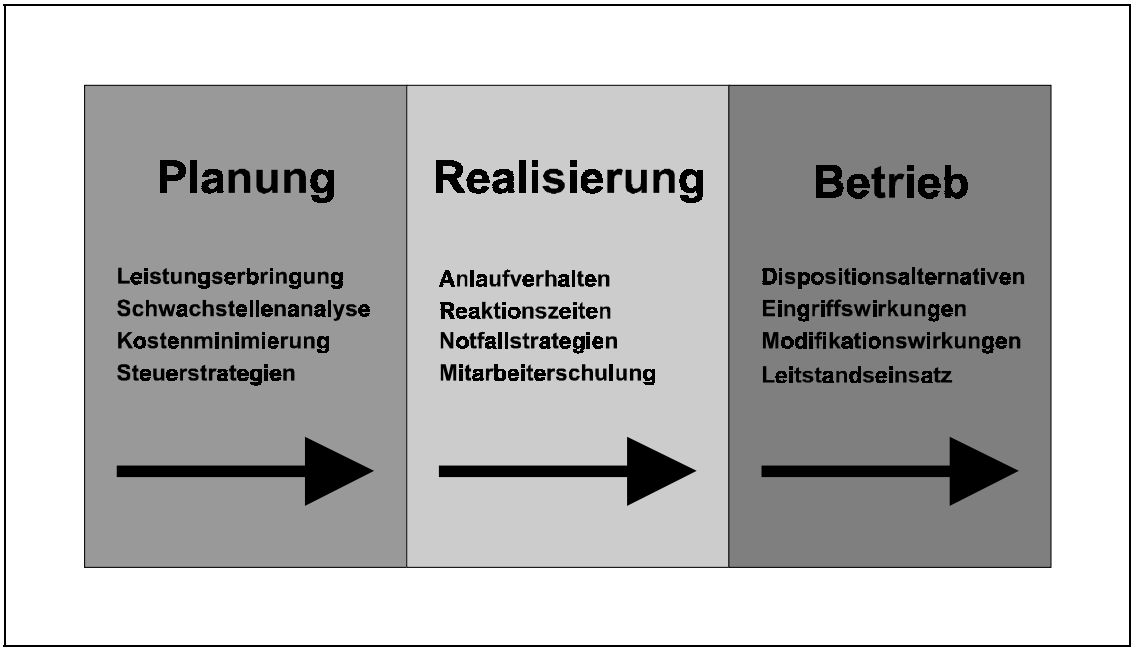

Bild 14: Simulationseinsatz im Lebenszyklus technischer Systeme

7.2.1 Simulation in der Planungsphase

In der Planungsphase helfen Simulationsstudien, die Planung von Systemen und Prozessen abzusichern. Der Detaillierungsgrad der Simulationsmodelle wird dabei (außer durch die Untersuchungsziele) vor allem durch die erreichte Planungstiefe bestimmt.

Typische Aufgabenstellungen bei vorhandenen Anlagen sind Vorabuntersuchungen von Modifikationen und die Ermittlung von Schwachstellen und von Grenzwerten des Durchsatzes. Die Untersuchungen zielen oft darauf, Veränderungen von Organisationsformen, Layout oder Führungsstrategien zu beurteilen, die beispielsweise durch Kapazitäts- oder Produktänderungen ausgelöst wurden.

Simulationsuntersuchungen neu geplanter, noch nicht existierender Anlagen zielen typischerweise auf die Erbringung von Funktionsnachweisen, die Leistungsabsicherung und die Projektkostenminimierung. Betrachtet werden vor allem die Anlagendimensionierung und die vorgesehenen Steuerungsstrategien, um Leistungsgrenzen und Engpässe sowie Durchsatz, Durchlaufzeiten und Bestandsentwicklung zu ermitteln und mit Planungsdaten zu vergleichen. Oft werden dabei mehrere Alternativen überprüft.

7.2.2 Simulation in der Realisierungsphase

Die Simulation ermöglicht die Ermittlung des Anlaufverhaltens von Anlagen entsprechend den vorgesehenen Realisierungsstufen und die Präsentation funktionaler Übersichten und Zusammenhänge, die die Mitarbeiterschulung an der Anlage und das Training von Leitstandspersonal sinnvoll ergänzen können.

Typische Aufgabenstellungen von Simulationsuntersuchungen in dieser Phase sind Leistungstests bei schrittweiser Einsteuerung der verschiedenen Produktvarianten oder bei stufenweiser Kapazitätsausweitung z.B. durch abschnittsweise Inbetriebnahmen, die Überprüfung der Auswirkungen von Anforderungsveränderungen oder von Problemen, die während der Realisierung aufgetreten sind, und die Untersuchung neuer oder veränderter Steuerungsstrategien.

7.2.3 Simulation in der Betriebsphase

Die Simulation erlaubt die vergleichende Bewertung kurzfristiger und situationsabhängiger Ablaufvarianten, so daß beispielsweise die Wirkung von Notfallstrategien und anderer Reaktionen auf Veränderungen während des Betriebs ohne Eingriff in die Produktion vorausschauend untersucht werden können.

Typische Aufgabenstellungen von Simulationsuntersuchungen in dieser Phase sind die Bewertung von Dispositionsalternativen in der Fertigungssteuerung, die Beurteilung von Varianten operativer Entscheidungen z.B. hinsichtlich Auftragsreihenfolgen, Losgrößen und Personaleinsatz, das Überprüfen von Notfallstrategien und die Ermittlung der Auswirkungen von Veränderungen beispielsweise der Produkte, des Auftragsmix oder der Arbeitszeitmodelle.

7.3 Durchführung von Simulationsstudien

Die Durchführung von Simulationsstudien umfaßt die Phasen Vorbereitung, Durchführung und Auswertung, die jeweils mehrere Teilschritte umfassen. Bild 15 gibt eine Übersicht über die Phasen und Teilschritte, die im folgenden kurz vorgestellt werden [1].

---

[1] s. hierzu auch *VDI-Richtlinie 3633 Blatt 1*; S. 9 ff.

7.3.1 Vorbereitung

Ausgehend von einer vorliegenden Planungsaufgabe ist zunächst zu entscheiden, ob ein Simulationseinsatz erfolgen soll. Die dabei zu berücksichtigenden Gesichtspunkte umfassen z.B. die Komplexität der Aufgabe (Vielzahl der Einflüsse und ihre Abhängigkeiten, die Notwendigkeit der Berücksichtigung paralleler Prozesse), bestehende Unsicherheiten hinsichtlich der Ergebnisse analytischer Methoden oder des Einflusses der mathematischen Verteilungstreuender Werte auf die Ergebnisse und die Möglichkeit, das Simulationsmodell auch zur Bearbeitung weiterer Aufgaben zu verwenden.

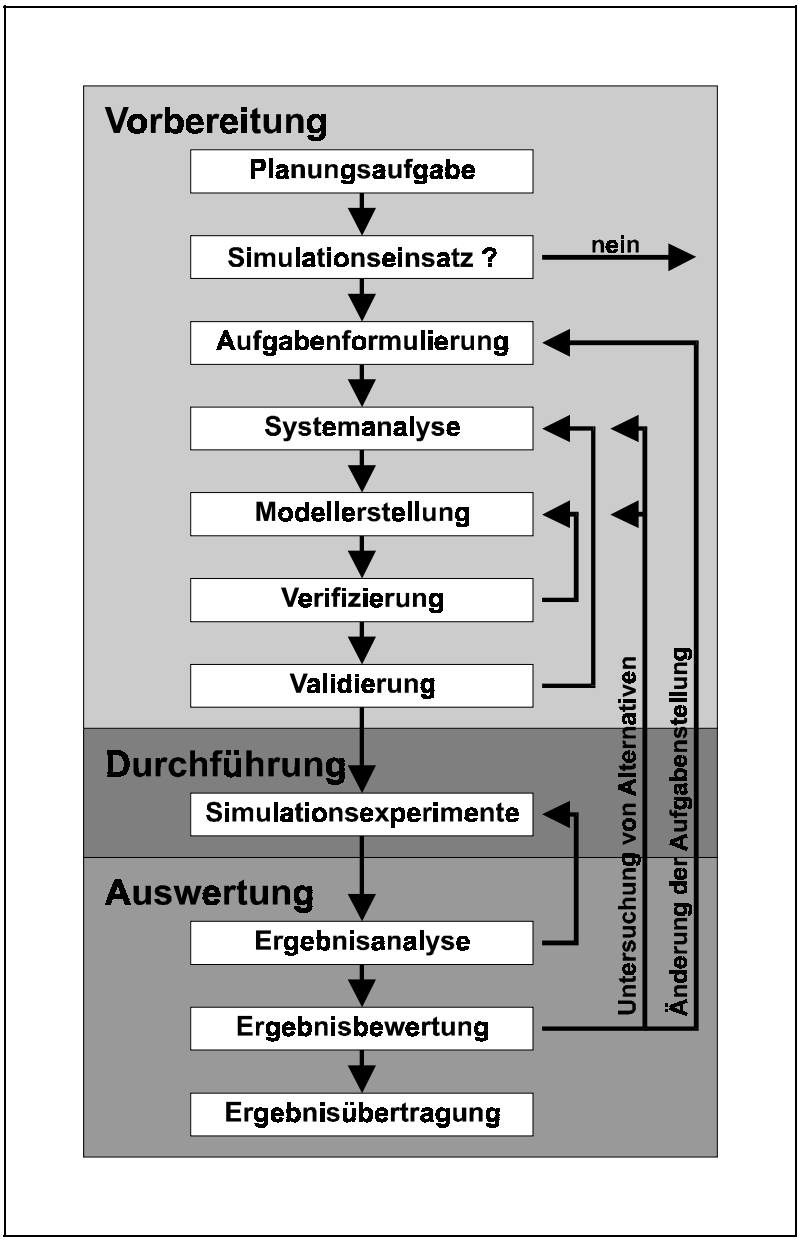

Bild 15: Phasen und Schritte einer Simulationsstudie

Eine weitere wesentliche Entscheidungsgrundlage ist der für die Simulation erforderliche Aufwand. Hier ist zunächst zu klären, welche Unterstützung seitens der involvierten Abteilungen erforderlich ist, welche Kosten für die Simulation anfallen und wann Unter-

suchungsergebnisse zur Verfügung stehen können [1]. Außerdem ist festzulegen, ob die Simulation durch Dienstleister oder mit eigenen Kräften durchgeführt werden soll, wobei letzteres natürlich das Vorhandensein des notwendigen Wissens und der erforderlichen technischen und personellen Kapazitäten voraussetzt.

Dem Aufwand ist der erwartete Nutzen aus der Simulation gegenüberzustellen. Neben dem direkten Nutzen aus der Lösung der Aufgabenstellung, der zumindest in einigen Fällen schon schwer zu quantifizieren ist, sind dabei auch mögliche indirekte und kaum quantifizierbare Effekte einzubeziehen, die sich z.B. aus der Nutzbarmachung von Know-how und Kreativität von Dienstleistern für die Bearbeitung des Gesamtprojekts ergeben.

Sofern eine Entscheidung für den Simulationseinsatz getroffen wurde, ist eine präzise Aufgabenstellung für die Simulationsstudie zu formulieren. Hier sind vor allem die grundlegenden Gestaltungsziele vorzugeben. Typischerweise bilden sie ein Zielsystem aus miteinander in Wechselwirkung stehenden Teilzielen, die in irgendeiner Weise aus dem übergeordneten Zielsystem der Produktion [2] abgeleitet sind.

Weiter ist festzulegen, welche Kenngrößen des System ermittelt werden sollen und in welcher Weise sie zu präsentationsfähigen Ergebnissen aufzubereiten sind. Anhand dieser Ergebnisse kann im weiteren Verlauf der Studie die Zielerreichung überprüft werden.

Ein aufgabenangepaßtes Zielsystem und geeignete Kenngrößen als Grundlage für die Beurteilung der Zielerreichung sind notwendige Voraussetzungen für einen wirtschaftlichen und effektiven Simulationseinsatz und eine sinnvolle Ergebnisinterpretation.

Im nächsten Schritt ist die Datenbasis für die Simulation durch eine Systemanalyse aufzubauen. Jedes zu untersuchende technische System wird durch eine Menge von Daten beschrieben, die sich in technische, Organistions- und Systemlastdaten gliedern lassen. Sie beschreiben die Komponenten (Leistung, Verfügbarkeit), die Struktur (Layout, Fertigungsstufen, Materialfluß), die Last (Stücklisten, Arbeitspläne, Aufträge) und das Verhalten (Ressourcen-zuordnung, Strategien, Arbeitszeitmodelle, Störfallmanagement) des Systems.

Die benötigten Daten sollten möglichst nicht in Papierform, sondern in gängigen Formaten auf Datenträgern oder im Direktzugriff zur Verfügung stehen, um den Aufwand für Erfassung und Aufbereitung zu begrenzen,. Der Umfang der zu erfassenden Daten hängt offensichtlich

---

[1] Sofern die betrachtete Anlage bereits früher Gegenstand von Simulationsuntersuchungen war, ist zu prüfen, ob die in Frage stehende Simulation mit entsprechenden Zeit- und Kostenvorteilen als Weiterführung der früheren Studie beauftragt werden kann.

[2] vgl. Kapitel 4.2

von der Komplexität des betrachteten Systems ab. Wesentlichen Einfluß hat aber auch die Aufgabenstellung, da sie die erforderliche Abbildungsgenauigkeit bestimmt.

Die Daten der Simulationsdatenbasis sind prinzipiell Bestandteil jeder Planung und Betrachtung technischer Systeme. Allerdings sind die Anforderungen an die Datenqualität im Falle der Simulation typischerweise deutlich höher als bei Anwendung anderer Planungsmethoden. Die gewonnenen Daten sind in jedem Fall auf Vollständigkeit und Plausibilität zu prüfen und gegebenenfalls zu ergänzen bzw. zu korrigieren, um Fehler oder Mehraufwendungen in nachfolgenden Schritten zu vermeiden.

Auf die Systemanalyse folgt die Modellerstellung. Sie umfaßt das Umsetzen des betrachteten Systems in ein experimentierfähiges Modell. Die Informationen der Simulationsdatenbasis sind als statische (nicht experimentierfähige) Teilmodelle anzusehen, die bei der Modellerstellung in ein integriertes dynamisches Softwaremodell überführt werden.

Zur Modellerstellung gehört die Abstraktion, d.h. die Reduktion und Idealisierung des betrachteten Systems zur Herausarbeitung seiner aufgabenrelevanten Eigenschaften. Da bereits kleine Anlagen sehr komplex sein können, ist es von erheblicher Bedeutung, die Abbildungsgenauigkeit auf das erforderliche Maß zu beschränken, um so den Zeit- und Kostenaufwand für die nachfolgenden Schritte in möglichst engen Grenzen zu halten.

Im Hinblick auf die nachfolgenden Arbeitsschritte, eventuelle nachträglich einzubringende Änderungen und den möglichen späteren Einsatz für weiterführende Untersuchungen ist das Modell in angemessener Weise zu dokumentieren.

In der Arbeitsfolge schließen sich zwei Prüfschritte an. Deren erster ist die Verifizierung, d.h. die Überprüfung der logischen Konsistenz des Modells. Dabei wird auch kontrolliert, ob das Modell mit der Simulationsdatenbasis übereinstimmt. Beispielsweise müssen alle Systemkomponenten richtig parametriert sein, alle Störungen im Rahmen ihrer mathematischen Verteilungen auftreten und alle Aufträge nach den richtigen Arbeitsplänen vollständig abgearbeitet und zu den vorgesehenen Zeiten bereitgestellt werden.

Der zweite Prüfschritt ist die Validierung, d.h. die Überprüfung der Gültigkeit des Modells. Es ist sicherzustellen, daß Modell und Originalsystem hinreichend übereinstimmen. Die Validierung ist ein iterativer Prüf- und Korrekturprozeß, für den keine festen Regeln existieren. Sie muß vielmehr aufgabenspezifisch vorgenommen werden. Bei der Simulation bestehender Systeme liegt es nahe, aus Testläufen gewonnene Ergebnisse mit bekannten Ist-Daten zu vergleichen. Bei Neuplanungen wird die Validierung typischerweise in Abstimmung mit Projektbeteiligten, die die erforderlichen Kenntnisse haben durchgeführt.

Aufgrund der bei der Modellerstellung vorgenommenen Abstraktion ist die Übereinstimmung von Modell und Original niemals vollständig, sondern nur innerhalb einer für den jeweiligen Fall als akzeptabel anzusehenden Toleranz möglich. Grundsätzlich bedarf es einer hinreichend großen Anzahl von Testläufen, um zuverlässige Aussagen über das Verhalten und die Qualität des Modells machen zu können.

Mit der Validierung werden das System- und Modellverständnis erhöht und damit die Grundlage für die Aussagefähigkeit der späteren Simulationsergebnisse gelegt.

7.3.2 Durchführung

Die Durchführung der Simulationsexperimente erfolgt in direkter Abhängigkeit von der jeweiligen Aufgabenstellung. In der Regel sind eine ganze Reihe von Simulationsläufen notwendig, bei denen Modellparameter, Lastdaten und Ablaufregeln systematisch variiert werden, so daß die gefundenen Ergebnisse vergleichend bewertet werden können.

Bei der Planung einzelner Experimente sind im wesentlichen die jeweiligen Werte variiender Eingangsgrößen so festzulegen, daß die Untersuchungsziele nach möglichst wenigen Simulationsläufen erreicht werden. Da Simulatoren die Modelle nicht selbst optimieren, sind oft regelmäßige Zwischenprüfungen der schon vorliegenden Ergebnisse erforderlich.

Die endgültige Durchführung von Simulationsstudien hängt in hohem Maß von der Erfahrung der Beteiligten ab und läßt sich nie vollständig vorherbestimmen. Sie ist typischerweise ein systematisiertes Probieren, bei dem sich Experimente oft erst aus den Resultaten vorangegangener Simulationsläufe ergeben.

7.3.3 Auswertung

Die Auswertung beginnt mit der Ergebnisanalyse, bei der die gewonnenen Simulationsergebnisse in geeigneter Form aufbereitet werden, um eine Interpretation zu ermöglichen. Dazu sind sie mit den mathematischen Mitteln der Statistik zu den von der Aufgabenstellung geforderten Kenngrößen zu verdichten, die das Systemverhalten beschreiben.

Simulatoren bieten hierfür üblicherweise auch Schnittstellen an, über die Simulationsergebnisse in externe Auswerteprogramme (z.B. Präsentationsgrafik- oder Tabellenkalkulationsprogramme) übernommen und dort weiterverarbeitet werden können. Daraus ergibt sich die Möglichkeit, Auswertungen und Ergebnisdarstellungen von Simulationsuntersuchungen an die Bedürfnisse und Standards des Auftraggebers anzupassen.

Wie oben bereits angedeutet, führt die Analyse von Simulationsergebnissen oft zur Durchführung zusätzlicher Experimente, z.B. weil erwartete Effekte mit den eingestellten Parametern (noch) nicht auftreten Es gibt jedoch auch Fälle, in denen die Ergebnisse ein Zurückgehen bis in die Vorbereitungsphase erfordern. Dabei kann es notwendig werden, zur Untersuchung von Alternativen das Modell und eventuell auch die Simulationsdatenbasis zu verändern. Es besteht auch die Möglichkeit, daß sogar die Aufgabenstellung angepaßt werden muß.

Der letzte Schritt einer Simulationsstudie ist die Ergebnisübertragung. Die gewonnenen Erkenntnisse, z.B. geeignete Parametereinstellungen, sind im Originalsystem umzusetzen.

7.4 Vorteile der Simulation

"*Auf Grund der Vielzahl von zeit- und zufallsabhängigen Systemgrößen sowie der über einfache Wechselwirkungen der Systemelemente hinausgehenden Wirkzusammenhänge stoßen mathematisch-analytische Methoden bei der Untersuchung derartiger Systeme schnell an Grenzen. Mit Hilfe der Simulation hingegen kann das zeitliche Ablaufverhalten komplexer technischer Systeme untersucht und beurteilt werden.*

Signifikante Vorteile der Simulation liegen in der Möglichkeit zur Untersuchung

- *real (noch) nicht existierender Systeme,*
- *real existierender Systeme ohne direkten Betriebseingriff,*
- *mehrerer Gestaltungsvarianten bei geringem Arbeitsaufwand,*
- *des Systemverhaltens über lange Zeiträume hinweg (Zeitraffung),*
- *von Anlaufvorgängen, Einschwingphasen und Übergängen zwischen definierten Betriebszuständen* [1]."

Ein weiterer Vorteil kann sich ergeben, wenn der (durchaus aufwendige) Aufbau der Simulationsdatenbasis gründlich und konsequent betrieben wird. Im Zuge des Zusammentragens und der Kontrolle dieser Daten werden nämlich eventuell bestehende Unzulänglichkeiten und Unvollständigkeiten der Planungsdaten des Projekts und mögliche Abweichungen zwischen den Informationen verschiedener Projektbeteiligter deutlich. Der Aufbau der Simulationsdatenbasis kann so die Planungsqualität insgesamt verbessern und damit dem Projekterfolg förderlich sein.

---

[1] *VDI-Richtlinie 3633 Blatt 1*; S. 2 ff.

Da die Simulation insbesondere bei der Datensammlung, der Validierung und der Auswertung typischerweise die Mitarbeit mehrerer Projektbeteiligter erfordert, fördert ihr Einsatz den allgemeinen Informationsaustausch im Projektteam durch die Vorgabe einer gemeinsamen Diskussionsgrundlage. Weil in Simulationsmodelle in der Regel Aspekte verschiedener Teilplanungen einfließen, regt die Simulation die Beteiligten auch zu einer ganzheitlichen Betrachtungsweise an, lange bevor diese, z.B. im Zuge der Inbetriebnahme, als (u.U. hart erfahrene) Notwendigkeit auftritt.

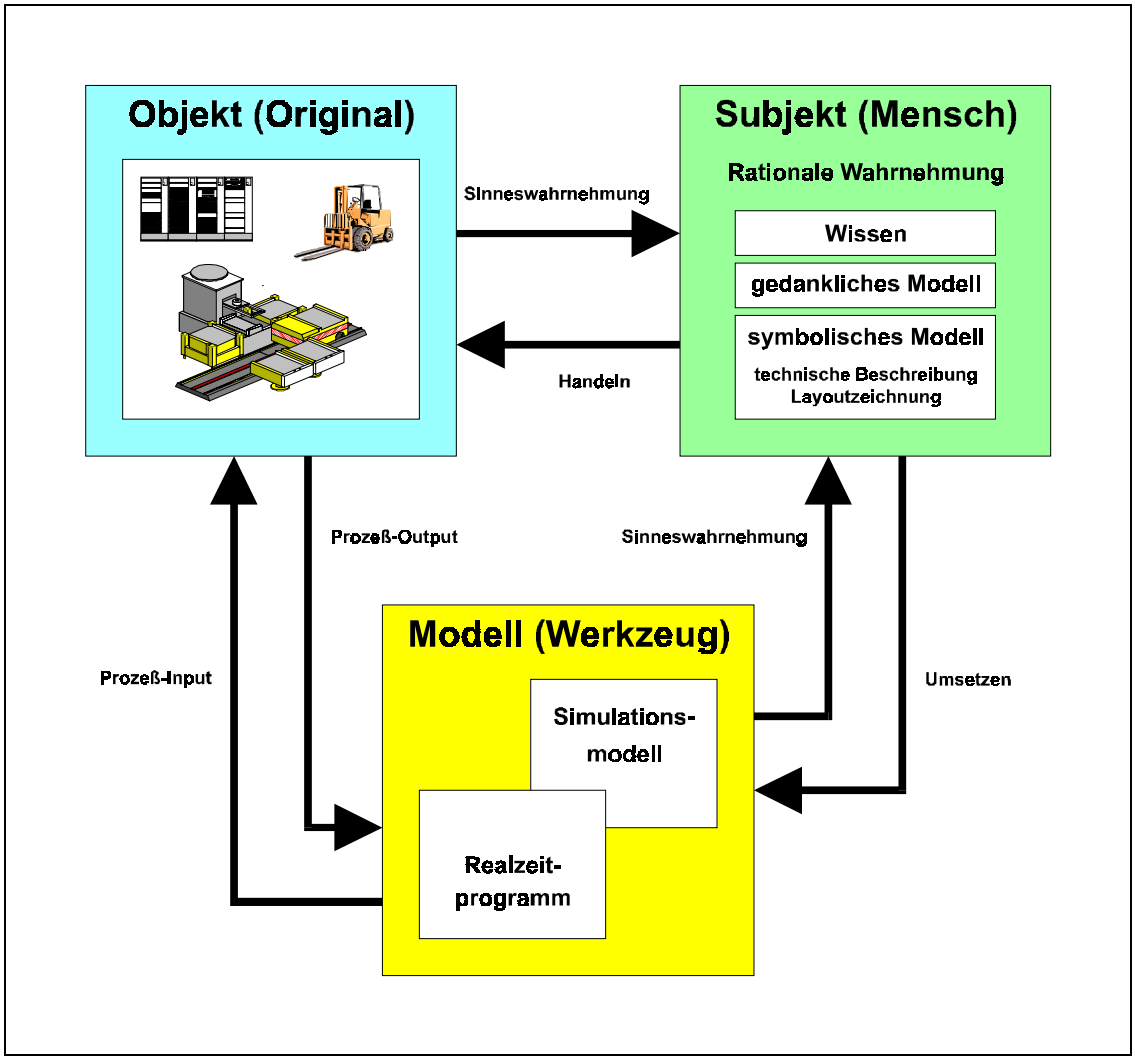

Bild 16: Werkzeugentwicklung in der Objekt-Subjekt-Modell-Relation

7.5 Simulatoren und Simulationsmodelle

Softwarewerkzeuge wie z.B. Simulatoren entstehen aus der Erfahrung von Experten, enthalten das erworbene Wissen dieser Menschen und sind dadurch bei der Lösung von Aufgaben aus dem jeweiligen Gegenstandsbereich einsetzbar [1].

Simulationsexperten haben die erforderlichen Fähigkeiten, um die Grundprinzipien technischer Systeme, die darin ablaufenden Prozesse sowie die eingesetzten Organisations- und Steuerstrategien unter Nutzung gedanklicher und symbolischer Modelle als Zwischenschritte im Medium Software als experimentierfähige Modelle formulieren zu können (s. Bild 16). Dadurch sind sie befähigt, Werkzeuge für die Durchführung von Simulationsstudien zur Verfügung zu stellen, die sich mit angemessenem Aufwand an Zeit und Wissen flexibel an Aufgabenstellungen unterschiedlicher Komplexität anpassen lassen.

Die verwendeten gedanklichen und symbolischen Modelle sind keine simulationstechnischen Artefakte, sondern spezifische Teilmodelle des betrachteten Systems, die auch für andere Aufgaben im Rahmen der Planung und Steuerung des Systems verwendet werden. Sie beschreiben die Anlage, ihre Steuerung und die sie durchlaufenden Produkte bzw. Teile. Aus der Sicht der Simulation bildet die Summe (der Inhalte) der Teilmodelle die bei der Systemanalyse aufzubauende Simulationsdatenbasis [2].

Ein Simulationsmodell faßt diese Teilmodelle, wie in Bild 17 dargestellt, zu einem integrierten Gesamtmodell zusammen. Die über die einfache Addition von Teilmodellen hinausgehende Qualität dieses Gesamtmodells liegt in der Einbeziehung der im System ablaufenden Prozesse, d.h. in der Nachbildung des Systemverhaltens über die Zeit. Das Gesamtmodell wird so zu einem dynamischen Fabrikmodell.

---

[1] *VDI-Richtlinie 3633 Blatt 1*; S. 5

[2] vgl. Kapitel 7.3.1

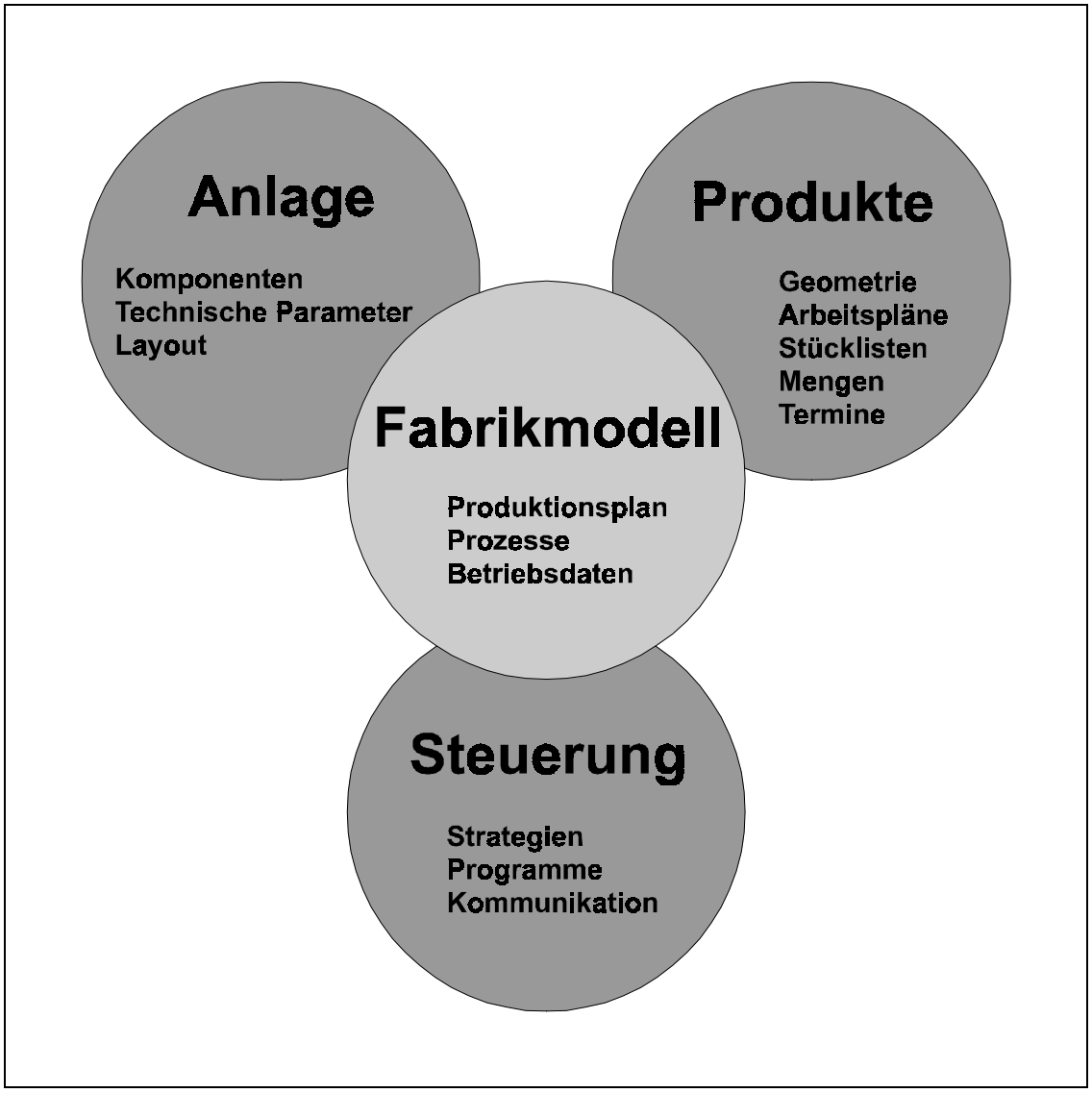

Bild 17: Integration von Teilmodellen zum dynamischen Fabrikmodell

7.5.1 Aufbau von Simulatoren

Die am Markt angebotenen Simulationswerkzeuge bestehen, wie in Bild 18 dargestellt, typischerweise aus den vier Hauptbestandteilen

- Simulatorkern,
- Datenverwaltung,
- Bedienoberfläche und
- Schnittstellen zu externen Datenbeständen [1].

---

[1] *VDI-Richtlinie 3633 Blatt 1*; S. 7 ff.

Der Simulatorkern ist der Teil, der Modellelemente sowie Meß- und Protokolliereinrichtungen bereitstellt. Im Kern jedes Simulators ist auch ein Zeitmechanismus [1] implementiert, der die chronologische Abarbeitung der vielen Einzelschritte veranlaßt, die zur Abbildung der im System ablaufenden Prozesse erforderlich sind. Der Simulatorkern verkörpert damit die zentrale Ablaufsteuerung, die die verschiedenen ablaufenden Prozesse miteinander verknüpft.

Die Datenverwaltung hält die Eingangs-, Zustands- und Resultatdaten eines Modells vor. Die Eingangsdaten werden vom Nutzer bereitgestellt. Sie sind zumeist exakt festgelegt, wie z.B. Geschwindigkeiten von Fördermitteln, können jedoch auch, wie z.B. Stördaten (Dauern und Abstände), durch mathematische Verteilungen beschrieben sein.

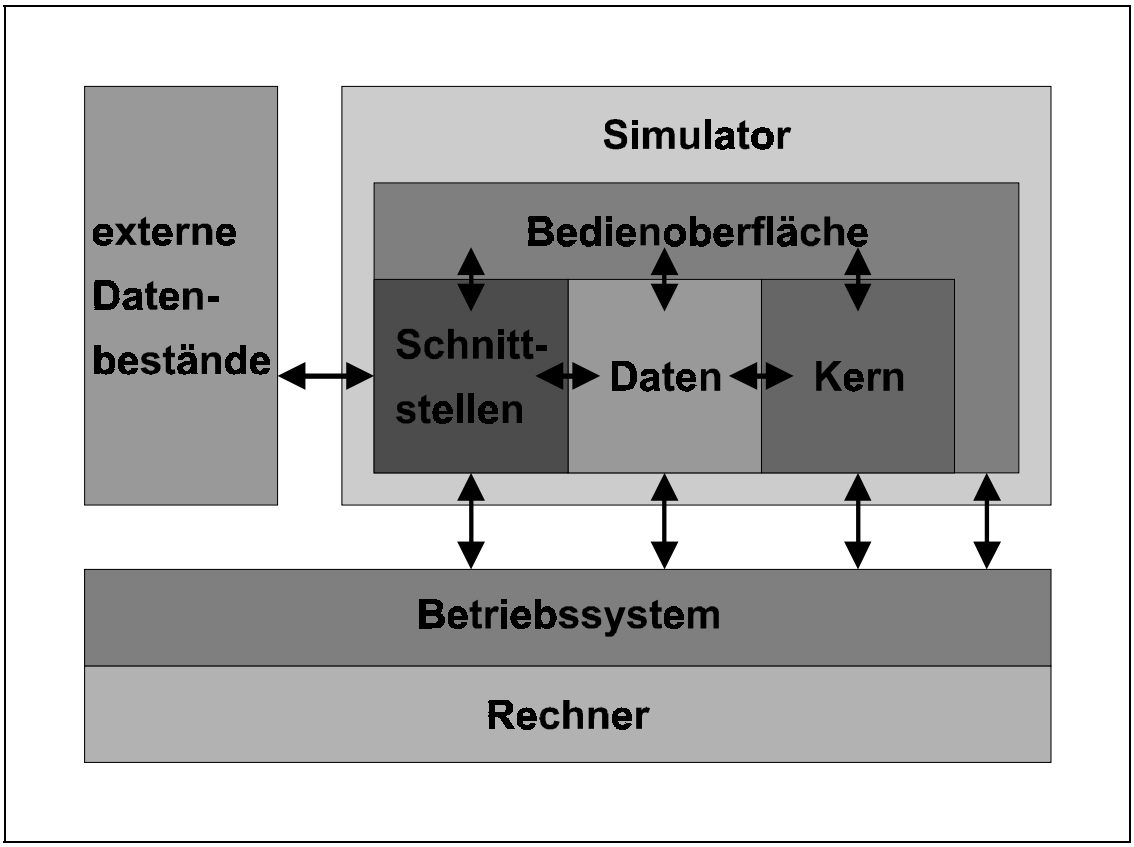

Bild 18: Aufbau eines Simulators

Die Zustandsdaten werden während der Durchführung von Experimenten durch die im System ablaufenden Prozesse verändert und haben also dynamischen Charakter. Sie beschreiben den Zustand des Systems zu jedem Zeitpunkt eines Experiments. Zu diesen Daten gehören beispielsweise Informationen über die Positionen von Transportmitteln.

[1] vgl. Kapitel 7.5.3.3

Resultatdaten sind alle Informationen, die zur Ermittlung und Bereitstellung der gewünschten Ergebnisse während der Experimente gesammelt, ausgegeben oder für die weitere Verarbeitung gespeichert werden.

Die Bedienoberfläche bietet Funktionen, die den Modellaufbau und die Dateneingabe einerseits und die Experimentbeobachtung und Ergebnisdarstellung andererseits unterstützen. Sie erlaubt das Editieren von Simulationsmodellen und ihre Darstellung auf dem Bildschirm. Sie ermöglicht die anschauliche Darstellung von Simulationsläufen und ihrer Resultate. Bei Simulatoren mit interaktiven Eingriffsmöglichkeiten, können während der Simulationsläufe z.B. Parameter modifiziert und Zustandsdaten abgefragt werden.

Im Simulator enthaltene Schnittstellen zu externen Datenbeständen ermöglichen den Import von Daten z.B. aus Datenbanken in das Simulationsmodell und ihren Export in umgekehrter Richtung. Dies erlaubt zum einen, vorhandene Datenbestände wie beispielsweise Auftragsinformationen oder CAD-Zeichnungen in Simulationsmodelle einzubringen und zum anderen, Ergebnisse aus Simulationsexperimenten anderen Softwarepaketen zur Auswertung, Visualisierung oder anderweitigen Verarbeitung zur Verfügung zu stellen.

7.5.2 Implementierung aufgabenbezogener Simulatoren

Ein aufgabenbezogener Simulator ist eine Software, die ein Simulationsmodell und die in Kapitel 7.5.1 vorgestellten Bestandteile eines Simulators umfaßt. Die Implementierung eines aufgabenbezogenen Simulators entspricht damit der Erstellung eines experimentierfähigen Modells [1].

Sie kann erfolgen unter Verwendung

- einer Programmiersprache,
- einer Simulatorentwicklungsumgebung oder
- eines bausteinbasierten Simulators [2].

Der Vorteil der Verwendung einer Programmiersprache ist die damit verbundene große Flexibilität hinsichtlich der möglichen Anwendungsgebiete. Diesem Vorteil stehen die Nachteile eines vergleichsweise großen Aufwands und einer hohen Fehleranfälligkeit bei der Modellerstellung und insbesondere bei nachträglichen Modifikationen gegenüber.

---

[1] vgl. Kapitel 7.3.1

[2] *VDI-Richtlinie 3633, Blatt 1*; S. 16 ff. Anstelle von bausteinbasierter Simulator wird dort der Begriff " *Modellwelt eines Simulators*" benutzt.

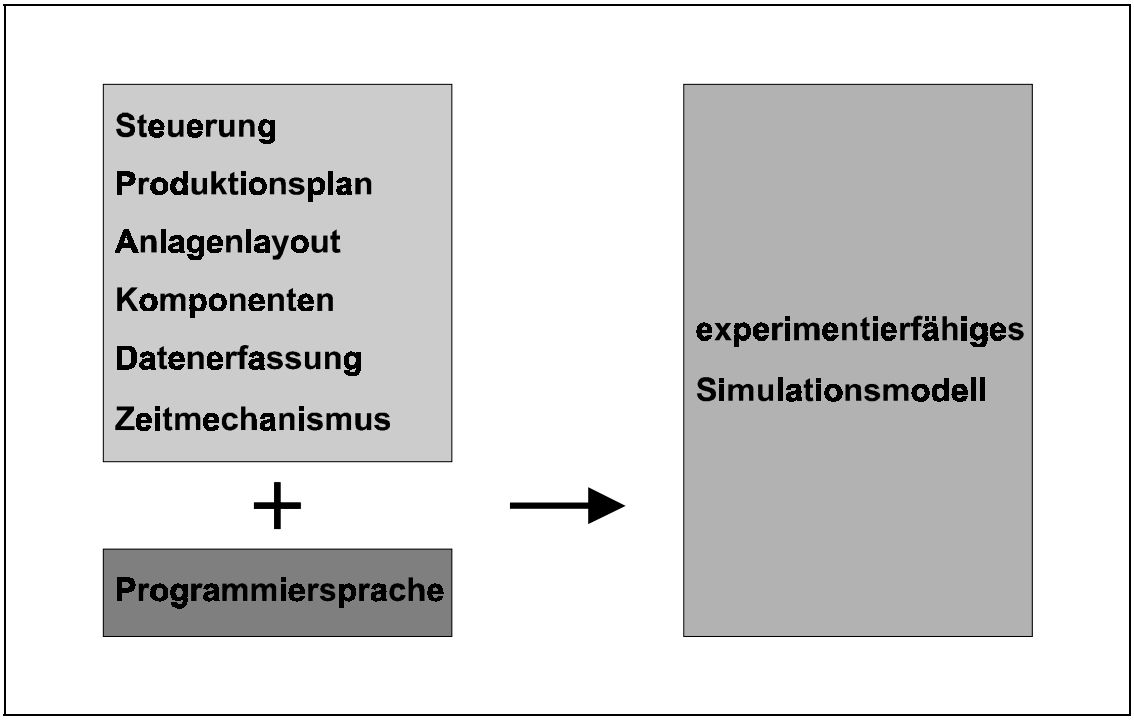

Bild 19: Simulationsmodell auf Basis einer universellen Programmiersprache

Bei der Verwendung universeller Programmiersprachen sind neben dem Modell auch alle erforderlichen Simulatorbestandteile zu programmieren (s. Bild 19). Im Umfang spezieller Simulationssprachen sind dagegen typische Elemente eines Simulatorkerns wie z.B. ein Zeitmechanismus und Meß- und Protokolliereinrichtungen enthalten. Die Implementierung von Simulatoren unter Verwendung von Programmiersprachen erfordert weitgehende Kenntnisse der Grundlagen der Simulationstechnik und bleibt daher Simulationsexperten vorbehalten [1].

Eine Simulatorentwicklungsumgebung stellt alle erforderlichen Simulatorbestandteile als Sofware-Paket zur Verfügung (s. Bild 20) und bietet häufig die Möglichkeit zum graphisch-interaktiven Editieren des Modells unter Verwendung relativ abstrakter Modellelemente. Mit ihrer Hilfe können Modelle ohne direkte Programmierung erstellt werden. Die Simulator-entwicklungsumgebung überführt die Eingaben und die übrigen Bestandteile in einen aufgabenbezogenen Simulator.

Verglichen mit der Verwendung einer Programmiersprache sind Aufwand und Fehleranfälligkeit reduziert. Die Flexibilität wird kaum eingeschränkt, da die angebotenen Modellelemente relativ allgemeinen Charakter haben. Die Verwendung von Simulatorentwicklungsumgebungen ermöglicht die Implementierung auch komplexerer Modelle [2].

---

[1] *VDI-Richtlinie 3633, Blatt 1*; S. 17

[2] ebd.

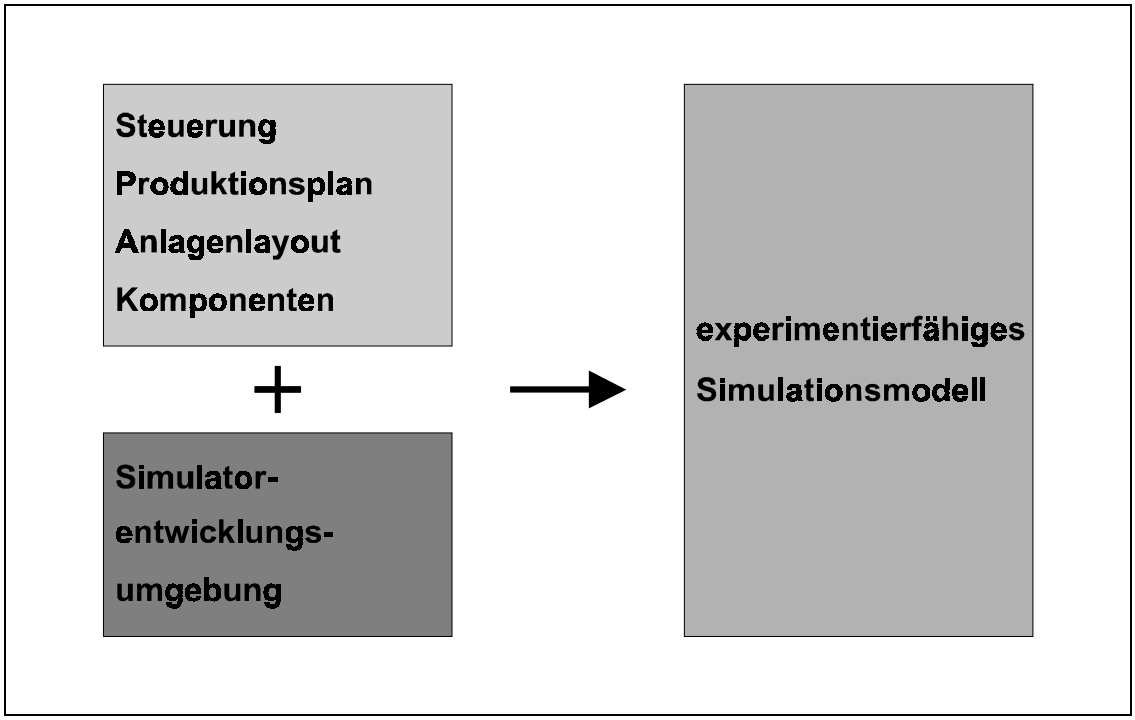

Bild 20: Simulationsmodell auf Basis einer Simulatorentwicklungsumgebung

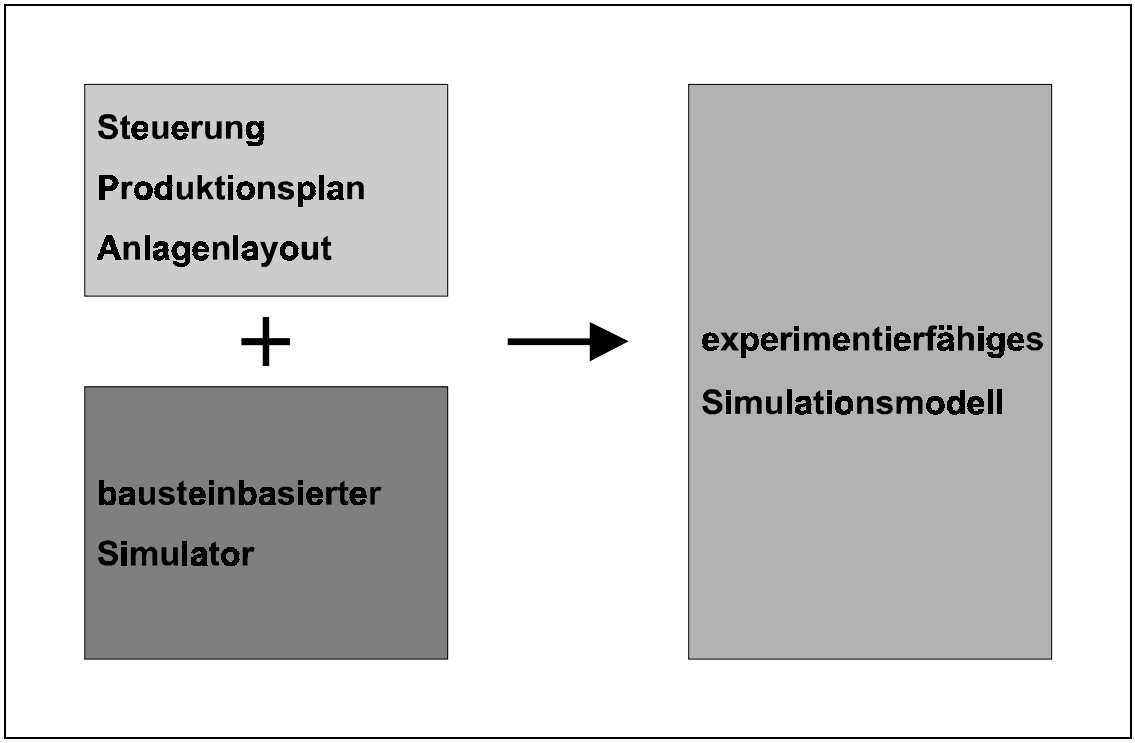

Bild 21: Simulationsmodell auf Basis eines bausteinbasierten Simulators

Ein bausteinbasierter Simulator stellt ebenso wie eine Simulatorentwicklungsumgebung alle erforderlichen Simulatorbestandteile als Software-Paket zur Verfügung (s. Bild 21) und bietet in der Regel auch die Möglichkeit zum graphisch-interaktiven Editieren des Modells unter Verwendung von Modellelementen. Diese sind jedoch typischerweise viel spezialisierter und komplexer als die Modellelemente von Simulatorentwicklungsumgebungen und entsprechen häufig direkt den Komponenten realer Systeme. Mit Hilfe bausteinbasierter Simulatoren können ohne Programmierung Modelle erstellt werden, die direkt experimentierfähig sind.

Die Spezialisierung der Modellelemente führt zu einer Begrenzung der möglichen Anwendungsgebiete. Andererseits verringern sich die Komplexität der Anwendung, der erforderliche Aufwand für die Modellerstellung und -änderung und die Fehleranfälligkeit. Da die Modellerstellung vollständig im Vorstellungs- und Begriffssystem des Anwendungsgebiets durchgeführt wird, kann sie ohne die Hilfe eines Simulationsexperten erfolgen. Die Verwendung bausteinbasierter Simulatoren ermöglicht die Implementierung komplexer und umfangreicher Modelle [1].

7.5.3 Merkmale verfügbarer Simulationswerkzeuge

Eine Übersicht über die verfügbaren Simulationswerkzeuge geben Noche et al. [2]. Sie nennen über 50 für den Einsatz in der Materialflußsimulation angebotene Werkzeuge, die je etwa zur Hälfte den sprach- bzw. bausteinbasierten Simulatoren zuzurechnen sind [3]. Nach Noche et al. ist davon auszugehen, "*daß alle auf dem Markt erhältlichen Instrumente in der industriellen Praxis ihr Einsatzfeld gefunden haben* [4]".

Reinhardt et al. stellen Lösungsansätze für eine materialflußtechnische Aufgabenstellung aus dem Bereich der Fertigung vor, die mit fünf verschiedenen Simulationswerkzeugen erarbeitet wurden [5]. Eine weitere Gegenüberstellung von auf mehreren Simulationsmodellen bzw. -werkzeugen basierenden Lösungen einer fertigungstechnischen Simulationsaufgabe initiierte Krauth [6].

---

[1] vgl. *VDI-Richtlinie 3633, Blatt 1*; S. 17

[2] Noche et al.: *Simulationsinstrumente im Überblick*

[3] vgl. Kapitel 7.5.2

[4] Noche et al.: *Simulationsinstrumente im Überblick*; S. 267

[5] Reinhardt et al.: *Simulationsinstrumente - Modellierung und Implementierung*

[6] Krauth: *Comparison 2: Flexible Assembly System*. Bis heute werden Lösungen zu dieser Aufagbenstellung in EUROSIM - Simulation News Europe veröffentlicht.

Diese Quellen zeigen, daß sich die verfügbaren Werkzeuge für die Materialflußsimulation durch die unterschiedliche Ausprägung einer ganzen Reihe von Merkmalen teilweise erheblich unterscheiden. Die wichtigsten dieser Merkmale aus Anwendungssicht sind

- die Rechnerplattform,
- die Möglichkeiten bei der Modellbeschreibung,
- die Konzeption des Zeitmechanismus,
- die Möglichkeiten zur Experiment- und Ergebnisdarstellung,
- die Eingriffsmöglichkeiten in laufende Experimente und
- die Möglichkeiten zur Abbildung spezieller Ablauflogiken.

7.5.3.1 Rechnerplattformen

Unter einer Rechnerplattform ist hier eine Kombination von Rechnerhardware, Betriebssystem und Basisgrafiksystem zu verstehen. Für Simulationswerkzeuge spielen nur zwei solche Plattformen eine Rolle. Dies sind zum einen Workstations unter verschiedenen UNIX-Derivaten, wobei als Basisgrafiksystem überwiegend die X-Window-Implementierungen der jeweiligen Betriebssystemhersteller zum Einsatz kommen. Die zweite verbreitete Plattform sind PCs unter MS-Windows (3.x und ´95) und (selten noch) DOS. Andere Plattformen kommen nur in Einzelfällen zum Einsatz.

Die meisten Simulationswerkzeuge sind nur auf einer Plattform implementiert. Einige sind jedoch auf mehreren Plattformen verfügbar, so daß die damit aufgebauten Modelle z.B. durch Übertragung und Rekompilierung portiert werden können. Die Netzwerkfähigkeiten der Simulationswerkzeuge sind in der Regel auf die vom jeweiligen Betriebssystem gebotenen Möglichkeiten beschränkt.

7.5.3.2 Möglichkeiten der Modellbeschreibung

Die von Simulationswerkzeugen gebotenen Möglichkeiten der Modellbeschreibung können nach den dabei verwendbaren Datenstrukturen und nach ihrer grundsätzlichen Art unterschieden werden. Es sind drei Arten der Modellbeschreibung möglich [1].

Die sprachorientierte Modellbeschreibung wird vor allem bei der Verwendung von Simulationssprachen als Werkzeug verwendet. Sie erlaubt die Nachbildung beliebiger Systeme und Prozesse mit Hilfe der jeweils vorgegebenen Sprachkonstrukte.

---

[1] *VDI-Richtlinie 3633, Blatt 1*; S. 6

Die parameterorientierte Modellbeschreibung setzt spezielle, nur parametrisch veränderbare Modelle voraus. Sie ist, da dieser Fall nur selten vorkommt, von geringer Bedeutung.

Die grafikorientierte Modellbeschreibung verwendet standardisierte oder nutzerdefinierte Symbole oder Grafiken als Bausteine zum Aufbau und zur Darstellung der Struktur des Modells. Diese können durch Parametereingaben ergänzt bzw. angepaßt werden. Zusätzliche sprachliche Beschreibungen ermöglichen die Erweiterung um nutzerdefinierte Baustein- und Systemeigenschaften und die aufgabenbezogene Anpassung der programmierten Abläufe.

Die verwendbaren Datenstrukturen können entweder auf fest vorgegebenen Standardklassen von Modellelementen beschränkt oder zusätzlich um anwendungsbezogene Klassen erweiterbar sein.

7.5.3.3 Zeitmechanismen

Jeder Simulator enthält notwendigerweise einen Zeitmechanismus, der die chronologische Abarbeitung der im System ablaufenden Prozesse steuert [1]. Alle Systemveränderungen sind bei diesem Mechanismus zusammen mit dem Zeitpunkt ihres Eintretens zu registrieren. Der Mechanismus seinerseits veranlaßt dann ihre Realisierung in nach aufsteigender Zeit geordneter Reihenfolge. Jedes Simulationsmodell umfaßt also ein von der realen Zeit unabhängiges virtuelles Zeitsystem in dem die aktuelle Zeit (d.h. die “Uhr”) diskret fortschreitet. Als mögliche Ausprägungen werden takt-, ereignis-, prozeß- und transaktionsorientierte Zeitmechanismen unterschieden [2].

Auf die Eigenschaften und Unterschiede zwischen diesen Typen von Zeitmechanismen wird an dieser Stelle nicht weiter eingegangen, zumal der für die Modellerstellung eingesetzte Zeitmechanismus auf hohen, aufgabennahen Beschreibungsebenen des Systems ohne Belang ist, da die Prozesse und Aktivitäten des Systems auf diesen Ebenen ohne Beachtung ihrer programmtechnischen Umsetzung diskutiert und formuliert werden. Auf niedrigeren, lösungsnäheren Ebenen kommt dem gewählten Zeitmechanismus dagegen große Bedeutung zu.

Jeder Zeitmechanismus impliziert eine bestimmte Weltsicht, d.h. eine bestimmte Sicht auf Systeme und die in ihnen ablaufenden Prozesse und Aktivitäten. Sie drückt sich auf allgemeiner Ebene darin aus, welche Systemelemente als Aktivitätsträger, also als aktive, handelnde und welche als passive, also Aktivitäten hinnehmende Einheiten angesehen werden. Die implizite

---

[1] vgl. die Erläuterungen zum Simulatorkern in Kapitel 7.5.1

[2] *VDI-Richtlinie 3633, Blatt 1*; S. 6

Weltsicht eines Simulationswerkzeugs bzw. seines Zeitmechanismus ist also eine Philosophie [1]. Sie gibt ein Begriffsystem vor, daß seinen konkreten Ausdruck in den von dem Werkzeug zur Verfügung gestellten Klassen (Datenstrukturen und Algorithmen) findet. Deren Umfang und Ausgestaltung erzwingt jeweils eine bestimmte Systemstrukturierung [2].

7.5.3.4 Experiment- und Ergebnisdarstellung

Unter Experimentdarstellung ist die dynamische Anzeige des Prozeßfortschritts am Bildschirm zu verstehen. Nach dem Zeitpunkt der Ausgabe werden Offline- und Onlinedarstellungen unterschieden. Offlinedarstellungen erfolgen nach Beendigung des Simulationslaufs, wohingegen Onlinedarstellungen während des Simulationslaufs synchron zum Experimentfortschritt erfolgen [3].

Die Möglichkeiten zur Experimentdarstellung umfassen typischerweise Ausgaben von Zustandsgrößen wie beispielsweise Auslastungen, Belegungs- und Durchlaufzeiten, die in alphanumerischer oder grafischer Form dargestellt werden können, sowie Animationen, d.h. grafische Darstellungen qualitativer und quantitativer Zustandsänderungen und der Ortsveränderungen bewegter Objekte [4].

Ein Schwerpunkt des Einsatzes von Experimentdarstellungen sind die Phasen der Modellprüfung (Verifizierung und Validierung) [5]. Sie ermöglichen hierbei u.a. die Beobachtung des Systemverhaltens, die effiziente Prüfung seiner Korrektheit, die schnelle Aufdeckung im Modell enthaltener Fehler und die einfache Identifikation und genaue Betrachtung kritischer Systembereiche, wie z.B. Engpässe und Kreuzungsstellen.

Auch bei Präsentationen von Simulationsstudien bzw. -modellen sind Experimentdarstellungen gut einsetzbar. Insbesondere Animationen unterstützen nach Beobachtung des Autors die Akzeptanz der konkreten Untersuchung wie der Simulation als Methode ungemein. Aufgrund ihrer hohen Anschaulichkeit [6] sind sie eine gute Grundlage für konstruktive Diskussionen.

---

[1] Sie entspricht (natürlich) dem Querschnitt der (bewußten und unbewußten) philosophischen Grundüberzeugungen seiner Erschaffer.

[2] Eine ausführlichere Diskussion der verschiedener Typen von Zeitmechanismen enthält Kapitel 12.

[3] *VDI-Richtlinie 3633, Blatt 1*; S. 8 ff.

[4] ebd.

[5] vgl. Kapitel 7.3.1

[6] vgl. die Definition in Kapitel 8.1.2

Während der Durchführung von Simulationsläufen werden Ergebnisdaten ausgegeben, die die Bewertung des Systemverhaltens in Bezug auf die jeweilige Aufgabenstellung ermöglichen. Diese Daten werden entweder bereits während der Experimente oder danach mit den mathematischen Mitteln der Statistik aufbereitet und so interpretierbar gemacht.

Die eigentliche Auswertung und Ergebnisdarstellung erfolgt nach dem Simulationslauf. Neben Übersichten allgemeinerer Natur können dabei spezielle Auswertungen beispielsweise für einzelne Systemkomponenten oder Produkte erstellt und die Ergebnisse mehrerer mit unterschiedlichen Parametern durchgeführter Simulationsläufe in gemeinsame Darstellungen zusammengefaßt und so verglichen werden.

Für die Ergebnisdarstellung werden heute zunehmend externe Auswerteprogramme eingesetzt [1]. Sie bieten im allgemeinen über die Fähigkeiten des eingesetzten Simulationswerkzeugs hinausgehende Darstellungsmöglichkeiten und vereinfachen die Weiterverwendung der erstellten Grafiken z.B. in Berichten oder als Folien für Präsentationen.

7.5.3.5 Interaktionsmöglichkeiten

Die von den verfügbaren Simulationswerkzeugen gebotenen Interaktionsmöglichkeiten während der Laufzeit unterscheiden sich ebenfalls erheblich voneinander [2].

Einige Werkzeuge erlauben ausschließlich die Stapelverarbeitung, d.h. die jeweils vollständige Ausführung eines Experiments, nach dessen Beendigung die ausgegebenen Ergebnisse untersucht werden können. Sie bieten also de facto keine Interaktionsmöglichkeiten.

Eine zweite Gruppe gestattet die Ein- und Ausschaltung verschiedener Sichten auf die Modelldaten (Experimentdarstellungen) während der Simulationsläufe.

Die dritte Gruppe bilden Simulationswerkzeuge, die während der Simulationsläufe Eingriffe in das Modell zulassen, so daß beispielweise Parameter verändert oder Systemkomponenten gesperrt werden können. Nur diese Werkzeuge sind im eigentlichen Sinne interaktiv, da sich vorgenommene Eingriffe unmittelbar auf den Fortgang des Experiments auswirken.

Umfangreiche Interaktionsmöglichkeiten unterstützen ebenso wie Experimentdarstellungen insbesondere die Modellerstellung und -prüfung sowie Präsentationen. Vor allem in Kombination miteinander bieten diese beiden Features großen Nutzen. Beispielsweise können

---

[1] vgl. Kapitel 7.3.3

[2] *VDI-Richtlinie 3633, Blatt 1*; S. 6

durch interaktive Modelleingriffe bestimmte Situationen wie z.B. Ausfälle von Systemkomponenten gezielt hergestellt und in Experimentdarstellungen das Ansprechen und die Wirkungen von Notfallstrategien überprüft, beobachtet und vorgeführt werden.

7.5.3.6 Abbildung spezieller Ablauflogiken

Bei der Erstellung von Simulationsmodellen ist es oft erforderlich, im abgebildeten System vorhandene spezielle Ablauflogiken bzw. Steuerungsalgorithmen in das Modell zu integrieren. Die verfügbaren Simulationswerkzeuge unterscheiden sich auch in den hierfür gebotenen Möglichkeiten. Als prinzipielle Methoden werden Entscheidungstabellen, spezielle Beschreibungssprachen oder Schnittstellen zur Integration von in (allgemeinen) Programmiersprachen formuliertem Code eingesetzt [1].

Unabhängig von der Methode erfordert die Abbildung spezieller Ablauflogiken vom Simulationswerkzeug die Bereitstellung ausreichender Möglichkeiten zur Gewinnung von Informationen aus dem Modell und zur Einflußnahme auf die Abläufe darin. Reicht der hier gebotene Leistungsumfang nicht aus, kann z.B. die Modellierung der Ablauflogik des in Kapitel 12.1 (Bild 29) vorgestellten Anlagenausschnitts mit Teileübergabe von Palette an EHB zu einer nur schwer lösbaren Aufgabe werden.

[1] Noche et al.: *Simulationsinstrumente im Überblick*; S. 283 ff.

8 Zukünftige Entwicklungen

In Kapitel 7.2 wurde ausgeführt, daß die Materialflußsimulation ein Hilfsmittel bei Planung, Realisierung und Betrieb technischer Systeme ist. Die Anforderungen an diese Systeme bzw. an ihre Planung und Steuerung führen konsequenterweise zu entsprechenden Anforderungen an die Simulationswerkzeuge.

Im folgenden werden die resultierenden erweiterten Anforderungen an Simulationswerkzeuge, zu erwartende Entwicklungen sowie die Basis für ihre Umsetzung dargestellt.

8.1 Neue Anforderungen

Die heute bestehenden und für den Zusammenhang dieser Arbeit wesentlichen Anforderungen an die Produktion wurden in den Kapiteln 4 bis 5 dargestellt. Die Ausführungen können wie folgt zusammengefaßt werden:

- Die veränderte Gewichtung der Teilziele des Zielsystems der Produktion resultiert in Produktionsanlagen heute im Einsatz aufwendiger Steuerungsmechanismen zur Kontrolle und Minimierung von Beständen und Durchlaufzeiten.

- Die aufgrund wachsender Produkt- und Anlagenkomplexität steigenden Investitionssummen und -risiken erhöhen die Anforderungen an die Planungssicherheit.

- Infolge verkürzter Innovationszyklen werden auch die für die Planung und den Bau von Fabriken zur Verfügung stehenden Zeiten immer geringer.

- Ständige Veränderungen an den Produkten zwingen zur fortlaufenden Anpassung der Produktion bis hin zur permanenten Fabrikplanung.

Diese Entwicklungen führen, wie eingangs gesagt, zu erweiterten Anforderungen an die Materialflußsimulation. Die wichtigsten hier zu nennenden Punkte sind:

- Die höhere Anlagenkomplexität führt zu komplexeren Simulationsmodellen.

- Die gestiegenen Anforderungen an die Planungssicherheit bedingen eine Steigerung der Ergebnisqualität von Simulationsuntersuchungen.

- Die verkürzten Planungs-, Bau- und Anpassungszeiten von Anlagen erfordern, daß auch Simulationsuntersuchungen in kürzerer Zeit durchführbar sein müssen.

Diese Punkte sollen in den nachfolgenden Abschnitten genauer erläutert werden.

8.1.1 Komplexere Simulationsmodelle

Der in Simulationsmodellen nach den Erfahrungen des Autors zu beobachtende und auch weiter zu erwartende Komplexitätszuwachs läßt sich auf mehrere Faktoren zurückführen.

Zunächst sind hier die aufwendigeren Steuerungsmechanismen realer Anlagen zu nennen. Entwicklungen wie die stärkere Gewichtung niedriger Bestände und Durchlaufzeiten, der damit zusammenhängende häufigere Aufbau durchlauforientierter Fertigungsstrukturen und die zunehmende Verbreitung von Just-in-time-Konzepten führen zur Realisierung immer aufwendigerer, auch anlagenübergreifender Steuerungssysteme. Der Trend geht dabei dahin, Bestände durch die Bildung und Nutzung von Information zu ersetzen.

Simulationsmodelle solcher komplexen Anlagen müssen natürlich auch deren Steuerungsmechanismen angemessen abbilden, um verwertbare Ergebnisse liefern zu können. Konkret äußert sich dies beispielsweise darin, daß Auftragsfreigaben statt in starren Intervallen abhängig von der Prozeßdynamik, z.B. von bestimmten Pufferbeständen oder anderen Auslöseimpulsen durchgeführt werden müssen, was im Simulationsmodell die Implementierung entsprechender Algorithmen anstelle statischer Vorgaben erfordert.

Eine andere Ursache für die zunehmende Komplexität von Simulationsmodellen ist, daß sie nach Anzahl der System- bzw. Modellelemente größer werden. Dies hängt u.a. sicherlich mit den oben genannten Aspekten zusammen. Beispielsweise erfordert die Simulation einer in ein Just-in-time-Konzept eingebundenen Anlage in der Regel die Einbeziehung der zufließenden Teileströme und ihrer Quellen ebenso so wie die des Abflusses bis zu den Verbauorten. Erst die entsprechende Verschiebung der Systemgrenzen ermöglicht die Auslegung eventuell erforderlicher Puffer zum Ausgleich von Stromschwankungen, die aber andererseits wesentlichen Einfluß auf die Funktionserfüllung der Anlage haben.

Schließlich erfordert die Senkung des Bestandsniveaus in der Regel auch das Vorsehen von Notfallstrategien und anderen Maßnahmen, die z.B.bei Betriebsmittelausfällen die eben nicht mehr über Bestände gewährleistete Lieferfähigkeit sicherstellen sollen. Simulationsuntersuchungen sollen typischerweise auch Aussagen über die Wirksamkeit solcher Maßnahmen liefern, so daß diese in das Modell einbezogen werden müssen.

Aus diesen Beobachtungen ergeben sich zwei konkrete Anforderungen an Simulationswerkzeuge. Zum einen darf es keine praktisch wirksamen Grenzen hinsichtlich der Modellgröße geben, d.h. alle möglichen Modellelemente müssen (ohne zahlenmäßige Beschränkung) beliebig oft verwendbar sein. Zum anderen muß es möglich sein, beliebige Steuerungsmechanismen abzubilden.

8.1.2 Höhere Ergebnisqualität

Die Qualität der Ergebnisse von Simulationsuntersuchungen kann sich in Zukunft ebensowenig in der Fehlerfreiheit erschöpfen wie die der Produkte [1]. Auch beim Simulationseinsatz wird sich ein erweiterter Qualitätsbegriff durchsetzen (müssen). Auftraggeber von Simulationsstudien sind als Kunden anzusehen, die (der Simulation) nur erhalten bleiben, wenn sie für ihre Aufwendungen einen überzeugenden Nutzen erhalten. Simulationsanbieter müssen sich als Dienstleister begreifen, die ihren Kunden umfassendere, über Selbstverständlichkeiten hinausgehende Leistungen bieten, indem sie von der einfachen Berechnung mehr und mehr zur kompetenten Beratung übergehen.

Nach den Beobachtungen des Autors äußern sich die gestiegenen Anforderungen an die Ergebnisqualität zunächst einmal darin, daß in Simulationsexperimenten heute typischerweise mehr Daten erfaßt werden als noch vor einigen Jahren und daß diese auch mit aufwendigeren Verfahren aufzubereiten sind.

---

[1] vgl. Kapitel 3

Die resultierenden Ergebnisdarstellungen müssen für die Auftraggeber in hohem Maße anschaulich sein, um ein besseres Verständnis für das betrachtete System und die erzielten Ergebnisse zu ermöglichen. Anschaulichkeit bedeutet in diesem Zusammenhang, daß die Darstellungen die Wahrnehmung ihrer Inhalte durch die Adressaten fördern und unterstützen müssen.

Das ursprünglich der Erkenntnistheorie entstammende Bild 16 (Seite 84) verdeutlicht die bei der Erkenntnisgewinnung wirkenden Mechanismen: Menschen nehmen ihre Umwelt zunächst mit den Sinnen (Sehen, Hören, ...) wahr (Sinneswahrnehmung) und verarbeiten die rezipierten Eindrücke dann unter Nutzung von Wissen und Erfahrungen, die sie im Verlauf von Sozialisation und Ausbildung erworben haben (rationale Wahrnehmung).

Die Volksweisheit "Ein Bild sagt mehr als tausend Worte" ist Ausdruck der Tatsache, daß das Sehen der bestausgebildete und meistgenutzte Sinn des Menschen ist. Folglich sollten (auch) die Ergebnisse von Simulationsstudien zweckmäßig als Grafiken dargestellt werden. Deren jeweilige Ausprägung (Tortendiagramm, Balkendiagramm, o.ä.) muß dabei ebenso wie ihre formale Gestaltung (Beschriftungen, Farben, etc.) den bei den Adressaten für die Darstellung der jeweiligen Inhalte üblichen Gepflogenheiten entsprechen [1].

Für Simulationswerkzeuge ergibt sich daraus die Anforderung, Darstellungen mit diesen Qualitäten entweder selbst erzeugen zu können oder ihre Erzeugung in weiteren Verarbeitungsschritten angemessen zu unterstützen. Werkzeugspezifische Darstellungen werden zukünftig nur noch als Extras ("Goodies") und auch dies nur bei frappanter Qualität akzeptiert werden.

Auf die zunehmende Integration von Steuerungsmechanismen und Notstrategien in Simulationsmodelle wurde in Kapitel 8.1.1 bereits eingegangen. Wie alle Regeln lassen sie sich nicht selbst, sondern nur in Form ihrer Wirkungen messen. Dabei treten jedoch oft typische Schwierigkeiten auf. So kann eine beobachtbare Wirkung auf mehrere Mechanismen zurückgehen oder ein Mechanismus kann mehrere, u.U. komplex zusammenhängende Wirkungen haben, unter denen gelegentlich auch unbeabsichtigte und vielleicht ungewollte sind [2].

Die Darstellung solcher Zusammenhänge ist ebenfalls nicht immer einfach. Zum einen kann die schiere Anzahl der relevanten Faktoren die Fähigkeiten menschlicher Wahrnehmung

---

[1] Es ist wesentlich, zu erkennen, daß sich diese Gepflogenheiten mit der Zeit ändern. Beispielsweise führten erst die verbesserten Möglichkeiten der Computergrafik und das Aufkommen von Präsentationsgrafikprogrammen dazu, daß die vormals üblichen Zeitreihen (Böswillige nennen sie heute "Zahlenfriedhöfe") inzwischen überwiegend durch Grafiken abgelöst wurden.

[2] Schon das Erkennen der Zusammenhänge von Ursachen(n) und Wirkunge(n) kann dabei ein nichttriviales Problem darstellen.

schlicht überfordern [1], zum anderen lassen sich mehrstufige Ursache-Wirkungsbeziehungen und dynamische Erscheinungen wie z.B. ein zeitlicher Versatz bis zum Eintreten einer Wirkung mit Hilfe von Grafiken oft nur unzureichend verdeutlichen.

Eine angemessene Veranschaulichung kann in solchen Fällen nur das Simulationsmodell selbst (bzw. seine Beobachtung) liefern. Es muß dazu dieselben Anforderungen an die Anschaulichkeit wie die Ergebnisdarstellungen erfüllen. Um die Prozeßdynamik sichtbar (sic!) machen zu können, ist also die Möglichkeit zur Animation [2] erforderlich.

Wie die der Ergebnisdarstellungen muß sich auch die Gestaltung von Animationen an den im Adressatenkreis üblichen Gepflogenheiten orientieren. Die meisten dieser Personen haben eine technische Ausbildung (typischerweise ein Ingenieurstudium) durchlaufen und dabei technische Zeichnungen kennen- und interpretieren gelernt. Daher sind (2D-) Layoutzeichnungen eine verbreitete Darstellungsform technischer Systeme wie z.B. Fabrikanlagen. Da sie die Wahrnehmung der Rezipienten folglich gut unterstützen, werden “animierte Layouts” andere werkzeugspezifische Animationsformen zunehmend in die Rolle von Extras (s.o.) verdrängen. Übliche, z.B. von CAD-Systemen gewohnte, ergänzende Features wie dynamisches Zoomen werden dabei selbstverständlich erwartet werden.

Bei der Animation größerer Modelle kann es dazu kommen, daß nicht alle, beispielsweise für die Beurteilung der Wirkung einer Notstrategie relevanten Anlagenkomponenten gleichzeitig in der erforderlichen Detailgenauigkeit visualisiert werden können, weil kein geeigneter Ausschnitt setzbar ist [3]. Für diesen Fall ist es wünschenswert, daß Animationen in mehreren Ansichten bzw. Fenstern gleichzeitig dargestellt werden können, in denen (unabhängig voneinander) verschiedene Ausschnitte setzbar sind.

Im Bereich der Animation steht m.E. noch eine weitere Entwicklung bevor: Nicht alle Adressaten von Simulationsstudien haben eine technische Ausbildung. Vielmehr sind auch z.B. Wirtschaftswissenschaftler zunehmend häufiger anzutreffen [4]. In der Folge ist die Layoutzeichnung nicht mehr immer die gemeinsame “Sprache” der Beteiligten. Layoutzeichnungen sind außerdem recht abstrakte Modelle, da Menschen ihre Umwelt eben nicht zweidimensional und linienhaft, sondern als dreidimensional und aus Körpern bestehend wahrnehmen. Weiter sind wir es gewohnt, durch diese Welt (also z.B. durch eine Fabrikanlage)

---

[1] vgl. Miller, G.: *The Magical Number Seven ...*

[2] vgl. Kapitel 7.5.3.4

[3] Beispielsweise könnten sich die relevanten Komponenten an gegenüberliegenden Enden des Layouts befinden.

[4] Die steigenden Investitionssummen und -risiken könnten der Grund hierfür sein.

wandern zu können, wobei sich die Körper, die wir sehen, verändern, indem sie sich perspektivisch verzerren, sich gegenseitig ganz oder teilweise verdecken oder (zeitweise) aus unserem Blickfeld verschwinden [1].

Da sie der menschlichen Wahrnehmung mehr als die üblichen Formen entsprechen, werden 3D-Animationen von "einfachen" Flächenmodellen bis hin zu VR-Visualisierungen ("Virtual Reality") zunehmende Verbreitung finden. Damit werden virtuelle Spaziergänge ("Virtual Walkthrough") durch das Simulationsmodell möglich. Ein Blick in benachbarte Disziplinen zeigt, daß die beschriebene Entwicklung dort bereits im Gange ist. So ist es im Bereich der Architektur nicht mehr ungewöhnlich, VR-Modelle geplanter großer oder bedeutender Gebäude im Zuge von Genehmigungsverfahren und als verbesserte Entscheidungsgrundlage z.B. für Investoren zu verwenden [2].

Im Zusammenhang mit dem Einsatz von Simulationen zur Veranschaulichung der Wirkung von Steuerungsmechanismen sei hier noch auf einen weiteren Aspekt hingewiesen. Spezielle solche Mechanismen, wie z.B. Notstrategien, greifen ihrer Anlage nach nur in besonderen Situationen. Um ihre Wirkungen beobachten und beurteilen zu können, ist es notwendig, die auslösenden Situationen künstlich herstellen zu können. Hierfür sind Möglichkeiten zum interaktiven Eingriff [3] in laufende Experimente erforderlich. Zur zweifelsfreien Erkennbarkeit des Zusammenhangs zwischen Ursache bzw. Auslösung und den eintretenden Wirkungen bedingt dies weiter, daß die zur Beobachtung eingesetzten Animationen online, also in direkter Kopplung an den Experimentfortschritt arbeiten.

Die Ausführungen zu Animationen lassen sich dahingehend zusammenfassen, daß Simulationswerkzeuge Möglichkeiten für interaktive Eingriffe in laufende Experimente und online-Animationen in 2D ("animiertes Layout") und in 3D bieten sollten.

Als letzter Aspekt in diesem Abschnitt soll noch die Frage der Darstellung der Steuerungsmechanismen selbst angeschnitten werden [4]. Es ist sicher unstreitig, daß sie wegen ihrer Bedeutung für die Ergebnisse festgehalten werden müssen und also im Rahmen der Dokumentation einer Studie darzustellen sind. Dies gilt natürlich in besonderer Weise, wenn ihre Entwicklung zu den Zielen der Untersuchung gehörte und sie weiterverwendet, also beispielsweise in die Steuerungsoftware der Anlage übernommen werden sollen.

---

[1] Genau genommen verändern sich die Körper natürlich nicht, wir nehmen sie, bedingt durch die Spezifika menschlichen Sehens, nur verändert wahr.

[2] vgl. z.B. Sperlich, Bauer: *Künstliche Welten*

[3] vgl. Kapitel 7.5.3.5

[4] Bisher wurde nur die Frage der Darstellung ihrer Wirkungen behandelt.

Für die Dokumentation von Steuerungsalgorithmen gilt natürlich wiederum die Forderung nach Anschaulichkeit der Darstellung. Weiterhin ist Eindeutigkeit zu fordern, um Mißverständnisse und Fehler z.B. durch abweichende Interpretationen auszuschließen. Das Spektrum möglicher Darstellungen umfaßt textuelle Beschreibungen, Wiedergaben in formalen Sprachen (z.B. Programmiersprachen) und die Verwendung (spezieller) abstrakter (und formaler) Notationen (z.B. Datenflußdiagramme oder Struktogramme [1]).

Da textuelle Beschreibungen die Anforderungen an Eindeutigkeit und Anschaulichkeit nicht hinreichend erfüllen [2], werden sie sich in der Zukunft sicher nicht durchsetzen. Die Wiedergabe in einer Programmiersprache setzt voraus, daß die Adressaten dieser Sprache mächtig sind, es wird sich also um eine "verbreitete" Programmiersprache handeln müssen [3]. Am interessantesten und vielversprechendsten ist wohl die Verwendung formaler Notationen. Diese nutzen im allgemeinen grafische Symbole. Sie sprechen daher die visuelle Wahrnehmung des Menschen an und nutzen deren besondere Leistungsfähigkeit.

Für Simulationswerkzeuge ergibt sich daraus die Anforderung, die in Simulationsmodellen verwendeten Steuerungsalgorithmen unter Verwendung formaler Notationen oder in verbreiteten (s.o.) Programmiersprachen darstellen zu können. Natürlich können solche Darstellungen auch indirekt erzeugt werden, z.B. indem Ausgaben oder Programmteile mit Hilfe von Zusatzsoftware konvertiert werden.

8.1.3 Kürzere Untersuchungszeiten

Der beobachtbare Druck, Simulationsuntersuchungen in immer kürzeren Zeiträumen durchführen zu können, führt in der Regel zunächst zu dem Wunsch, "schnellere" Simulationswerkzeuge zur Verfügung zu haben, die die für die Durchführung einzelner Experimente erforderliche Zeit reduzieren. Diese Sichtweise betont allerdings m.E. zu einseitig die Softwareperformance und greift damit zu kurz.

Natürlich können die Vorteile schnellerer Experimentdurchführung nicht bestritten werden. Allerdings ist, wie in Kapitel 7.3 dargestellt, die Durchführung von Experimenten nur eine der Phasen einer Simulationsstudie. Dabei werden oft vergleichende Läufe mit nur geringfügig variierenden Eingangsdaten durchgeführt. Da diese Läufe (eben wegen der Vergleichbarkeit)

---

[1] z.B. nach DIN 66001.

[2] Aus diesem Grunde werden formale Notationen ja gerade entwickelt.

[3] Dies läßt zunächst einigen Interpretationsspielraum. Im Einzelfall wird Auswahl aber recht einfach sein: Was die Adressaten nicht kennen und daher nicht verstehen, werden sie nicht akzeptieren.

typischerweise ohne Eingriffe und damit bedienungslos durchgeführt werden, lassen sich Zeitvorteile in dieser Phase auch ohne schnellere Software einfach dadurch erreichen, daß mehrere Rechner eingesetzt werden, auf denen die Experimente im “Batch-Betrieb” über Nacht oder über das Wochenende laufen.

Eine ganzheitliche Betrachtung einer Simulationsstudie mit allen ihren Phasen und den dafür typischerweise aufzuwendenden Zeiten führt schnell zu der Erkenntnis, daß signifikante Potentiale für Zeiteinsparungen vor allem in der Vorbereitungs- und in der Auswertungsphase vorhanden sind, eben dort, wo die Durchführenden den größten Teil ihrer Arbeitszeit aufwenden müssen.

Auf die Anforderungen an die Auswertungsphase bzw. an die Ergebnisdarstellung und mögliche Ansätze zu ihrer Erfüllung wurde in Kapitel 8.1.2 eingegangen. Eine hohe Zeiteffizienz läßt sich dabei erreichen, wenn alle Verfahrensschritte von der Datenermittlung bis zu den fertigen Grafiken in einem aufgaben- und werkzeugangepaßten Prozeß möglichst weitgehend automatisiert werden. Für Darstellungen, die auf Daten mehrerer Simulationsläufe aufbauen und bei Verwendung von Präsentationsgrafikprogrammen ist dies nicht allein durch Vorkehrungen in den Simulationswerkzeugen selbst zu erreichen. Die an diese zu stellenden Anforderungen zur Unterstützung des Verarbeitungsprozesses beschränken sich daher sinnvollerweise darauf, flexibel nutzbare Möglichkeiten zur frei formatierbaren Ausgabe von Experimentdaten z.B. in Dateien bereitzustellen.

Ansatzpunkte für Zeiteinsparungen in der Vorbereitungsphase einer Simulationsstudie finden sich m.E. vor allem bei den Teilaufgaben Systemanalyse und Modellerstellung einerseits und bei der Modellprüfung, d.h. bei der Verifizierung und Validierung andererseits. Wegen der damit verbundenen Erhöhung der Anschaulichkeit auch für die Durchführenden kommt auch in diesem Bereich dem verstärkten Einsatz grafischer Hilfsmittel große Bedeutung zu.

Am weitesten fortgeschritten ist diese Entwicklung sicherlich bei der Layouterstellung. Eine ganze Reihe von Simulationswerkzeugen erlaubt es, Modelle in einer grafischen Umgebung aufzubauen, indem entsprechende Darstellungen von Modellelementen ausgewählt, positioniert, miteinander verbunden und mit Attributwerten (z.B. technischen Parametern) versehen werden. Ihre konsequente Fortsetzung finden solche Features in der Möglichkeit zur Übernahme von Layouts aus CAD-Zeichnungen.

Bei der Eingabe produktbezogener Daten wie Arbeitsplänen, Stücklisten, Mengen und Terminen ist die von Simulationswerkzeugen gebotene grafische Unterstützung im allgemeinen deutlich weniger weit entwickelt. Da jedoch einerseits grafische Darstellungen beispielweise von Stücklisten als Bäume bekannt und verbreitet sind und andererseits offenkundig einfache, anschauliche und schnelle grafische Editierverfahren wie z.B. die Eingabe der Bearbeitungs-

stationen eines Arbeitsplans durch "Anklicken" im Layout vorstellbar sind, ist die Realisierung entsprechender Features eine an zeitgemäße Simulationswerkzeuge heute zu stellende Anforderung.

Weiteres Potential für die Aufwandsreduzierung bei der Eingabe derartiger Daten in Simulationsmodelle birgt ihre direkte Übernahme aus externen Quellen wie z.B. Datenbanken. Da diese Datenquellen in der Praxis sehr unterschiedlich ausgebildet und nicht standardisiert sind, ist eine feste Integration entsprechender Schnittstellen in Simulationswerkzeuge nur in Einzelfällen sinnvoll, da sie entweder die Einsetzbarkeit des Werkzeugs beschränkt oder wenigstens in allen Fällen, in denen sie nicht benötigt wird, einen unnötigen Ballast darstellt. Um die Möglichkeiten zur Effizienzsteigerung durch direkte Datenübernahme dennoch nutzen zu können, bietet sich der Weg über ein neutrales und verbreitetes Zwischenformat wie z.B. ASCII-Dateien an. Von Simulationswerkzeugen ist daher zu fordern, daß sie (möglichst beliebig gestaltete) solche Zwischenformate auswerten können. Deren Erzeugung kann aufgabenbezogen und bedarfsweise z.B. durch die Generierung eines Reports aus einer Datenbank geschehen.

Im Zusamenhang mit der Modellerstellung ist auch die Eingabe von Steuerungsmechanismen zu erwähnen. Die Forderung, beliebige solche Mechanismen in Simulationsmodellen abbilden zu können, wurde bereits in Kapitel 8.1.1 erhoben. An dieser Stelle ist ergänzend darauf hinzuweisen, daß, insbesondere beim Simulationseinsatz in der Betriebsphase, diese Mechanismen in der Steuerung des betrachteten Systems bereits implementiert sind, so daß eine direkte Übernahme auch hier erhebliche Zeitersparnis bedeuten könnte. Obwohl die in der Praxis auftretenden Hindernisse (Steuerung auf viele Rechner unterschiedlichen Typs (SPS, PC, etc.) verteilt, allenfalls Binärcode verfügbar, usw.) dies heute in der Regel wohl noch verhindern, ist die Übernahme von Steuerungsmechanismen sicherlich dann am ehesten realisierbar, wenn Simulationswerkzeuge entsprechende Algorithmen aus formalen Notationen oder verbreiteten Programmiersprachen [1] übernehmen oder letztere direkt verwenden können.

Neben der Modellerstellung bietet auch die Modellprüfung Ansatzpunkte zur Zeinteinsparung. Die in Kapitel 7.3.1 beschriebenen Prüfschritte umfassen neben (statischen) Kontrollen (z.B. Vollzähligkeit der Systemkomponenten) vor allem die Durchführung von Testläufen, bei denen eventuelle Fehler im Modell erkannt und behoben werden müssen.

[1] vgl. die Ausführungen zur Dokumentation solcher Algorithmen in Kapitel 8.1.2

Selbstverständlich ist es wünschenswert, Fehler zu vermeiden, also gar nicht erst auftreten zu lassen. Beim aktuellen Stand der Softwaretechnik sind aber Fehler in Simulationsmodellen wie allgemein in jeder Software unvermeidlich [1]. Praxisgerechte Software- und damit auch Simulationswerkzeuge unterstützen daher die effiziente Fehlererkennung und -behebung.

Zu Beginn eines Experiments ist ein Simulationsmodell in den meisten Fällen leer, d.h. im betrachteten System sind noch keine Teile bzw. Aufträge in Bearbeitung. Während einer Anlaufphase wird das System durch Einschleusungen gefüllt, bis es schließlich einen eingeschwungenen Zustand erreicht. Die Anlaufphase birgt, z.B. durch falsche oder ganz fehlende Startwerte für Zustandsdaten [2] (sog. Initialisierungsfehler) oder verfrüht greifende Notstrategien, einige spezielle Fehlermöglichkeiten.

Im allgemeinen ist aber die Betrachtung des eingeschwungenen Zustands, auf den sich in der Regel auch alle erhobenen Ergebnisdaten beziehen, von größerem Interesse. Bei der Behebung eines Fehlers ist es also zu Erfolgskontrolle erforderlich, das Experiment mindestens einmal neu zu starten [3]. Da die Fehlererkennung und -behebung den Einsatz menschlicher Arbeitskraft erfordert, liegt das offensichtlichste Potential für die Aufwandsreduzierung bei der Modellprüfung in der Vermeidung oder wenigstens Verkürzung dieser Turn-Around-Zyklen.

Vermieden werden können diese Zyklen, wenn die ergriffenen Maßnahmen erfolgreich sind. Dies erfordert, daß die Ursache eines Fehlers sicher und richtig erkannt wird. Simulationswerkzeuge können dies dadurch unterstützen, daß Experimente an beliebiger Stelle unterbrechbar, alle Zustandsdaten abfragbar und fehlerverursachende Situationen gezielt herbeiführbar sind und daß sie das Modell insgesamt anschaulich darstellen können. Offensichtlich führt dies wiederum auf die in Kapitel 8.1.2 schon aus anderen Gründen erhobenen Forderungen nach interaktiven Eingriffsmöglichkeiten in Verbindung mit online-Animationen.

Zur Verkürzung der Turn-Around-Zyklen könnte die erforderliche Zeit zum (Wieder-) Erreichen relevanter Konstellationen zum einen dadurch reduziert werden, daß Teilfunktionen der Software zu beliebigen Zeitpunkten während der Experimente zu- und abschaltbar sind. So könnten z.B. Datenerfassungen und Animationen nach Bedarf eingeschaltet bzw. eben nicht eingesetzt werden, woraus sich wegen der Vermeidung der als (vergleichsweise) langsam

---

[1] vgl. Booch: *Object-Oriented Analysis and Design ...*; S. 278 ff.

[2] vgl. Kapitel 7.5.1

[3] Die Erfahrung des Autors zeigt (wie die anderer Softwarentwickler), daß typischerweise sogar einige Neustarts erforderlich sind, da nicht jede versuchte Fehlerbehebung sofort zum Erfolg führt.

bekannten Datei- und Grafikoperationen relevante Zeiteinsparungen ergeben könnten [1]. Simulationswerkzeuge sollten entsprechende Funktionalität bieten.

Zeitgewinne gerade bei der Modellprüfung brächte auch die Aufhebung der in vielen Simulationswerkzeugen anzutreffenden Trennung zwischen Entwurf und Experiment, also ihrer Teilung in (mindestens) zwei Programme. Je nach Realisierung der Integration lassen sich durch sie die erforderlichen Zeiten für wiederholtes Laden der (Einzel-) Programme und die Zeiten für das Initialisieren und Sichern des Modells zumindest teilweise einsparen. Da auch in anderen Softwaresystemen die Integration aller zugehörigen Funktionalitäten unter einer gemeinsamen Oberfläche heute üblich ist [2], ist dies auch für Simulationswerkzeuge zu fordern.

Eine weitere Möglichkeit zur Verkürzung der Turn-Around-Zyklen bei der Fehlerbehebung bestünde darin, ein Simulationsmodell in einem beliebigen (eingeschwungenen) Zustand einfrieren bzw. abspeichern und neue Läufe auf diesen Zustand aufsetzend starten zu können. Entsprechenden Ansätzen stehen jedoch zwei Schwierigkeiten entgegen. Deren eine ist, daß neben der durchaus vorstellbaren Sicherung des Zustands der Systemkomponenten und der in Arbeit befindlichen Teile und Aufträge auch der Zustand aller Steuerungsalgorithmen einzufrieren (und wiederherzustellen) wäre. Dies ist mindestens im Zusammenhang mit der zweiten Problematik kaum zu realisieren. Diese besteht darin, daß in der Regel eben nicht das gleiche sondern ein (leicht) modifiziertes Experiment durchgeführt werden soll. Die Veränderungen können einzelne Parameter oder, und dies ist bei der Modellprüfung typisch, gerade die Steuerungsalgorithmen betreffen.

Im Ergebnis erscheint für die Speicherung und Wiederherstellung beliebiger Modellzustände eine allgemeingültige Vorgehensweise nicht realisierbar zu sein, so daß die Forderung eines entsprechenden Features von Simulationswerkzeugen nicht zu begründen ist. Modellspezifische Lösungen sind jedoch sicherlich in vielen Fällen möglich [3].

Zum Abschluß dieses Kapitels soll noch auf die Bedeutung der Bedienoberfläche von Simulationmodellen bzw. -werkzeugen auf die Untersuchungszeiten eingegangen werden.

---

[1] Offensichtlich ist besonders die Möglichkeit, ohne Animationen zu experimentieren auch für die Durchführung “bedienungsloser” Serienexperimente sinnvoll.

[2] Beispielsweise vereinigen moderne Softwareentwicklungswerkzeuge unter einer gemeinsamen Oberfläche Softwaretools für die Eingabe (Editor), die Übersetzung in ausführbaren Code (Compiler und Linker) und den Test (Debugger) von Programmen.

[3] Eine solche wurde beispielsweise unter Mitarbeit des Autors im Rahmen des Projekts “Grundlagen der rechnerintegrierten Fabrik” (s.a. Quellenverzeichnis) realisiert.

Wahrscheinlich wird es keinen Menschen geben, der mit Simulationswerkzeugen arbeitet und nicht gleichzeitig andere Softwaresysteme benutzt [1]. Obwohl sich die Bedienoberflächen dieser Systeme in vielen Punkten unterscheiden (immerhin dienen die Programme ja auch verschiedenen Zwecken), ist in den letzten Jahren eine Tendenz zu einer gewissen Vereinheitlichung der Oberflächen zu beobachten.

Von einem modernen PC-Programm erwarten seine Benutzer beispielsweise, daß es eine grafische Bedienoberfläche hat, in der sich die Funktionen des Programms mit der Maus auslösen lassen und daß sich am oberen Rand des Hauptfensters ein Menü befindet, über dessen am weitesten links angeordneten Eintrag sich (mindestens) eine Funktion zum Abbrechen des Programms erreichen läßt. Sie erwarten weiter, daß sich z.B. in einer Textverarbeitung in Texten enthaltene Grafiken und in einem Zeichenprogramm Zeichnungsobjekte einheitlich dadurch verschieben lassen, daß sie mit der Maus markiert ("angeklickt") und bei gedrückt gehaltenener linker Maustaste an eine andere Position verschoben werden.

Derartige Benutzererwartungen bilden zusammen mit über die Jahre gewachsenen sogenannten de-facto-Standards, den für die Bedienoberfläche des Betriebssystems (GUI - Graphical User Interface) von dessen Entwicklern vorgegebenen Gestaltungsrichtlinien (Style Guides) [2] und (allgemeineren) Erkenntnissen über Mensch-Maschine-Schnittstellen und Softwareergonomie [3] ein Gerüst von Anforderungen und Vorgaben, an denen sich die Bedienoberflächen von Softwaresystemen orientieren sollten.

Obwohl die Nichteinhaltung solcher Standards (natürlich) nicht sanktioniert wird, führt sie mindestens dazu, daß Anwender programmabhängig verschiedene Schritte zur Durchführung an sich gleicher Handlungen abzuarbeiten haben [4]. Da die Anwender (auch nur) Menschen sind, werden sie daraufhin bei der Bedienung nicht standardkonformer Programms Fehler begehen, die günstigstenfalls Zeit kosten, u.U. aber auch andere unangenehme Folgen (z.B. teilweise Datenverluste) haben können [5].

---

[1] Es wurde bereits erwähnt, daß im Zuge der Durchführung von Simulationsstudien z.B. Präsentationsgrafiksysteme (zur Ergebnisdarstellung) eingesetzt werden. Daneben wird zur Dokumentationserstellung sicherlich ein Textverarbeitungs- und vielleicht auch noch ein Zeichenprogramm genutzt.

[2] vgl. z.B. Apple: *Macintosh Human Interface Guidelines*, IBM: *Systems Application Architecture ...* und Microsoft: *The Windows Interface ...*

[3] vgl. z.B. Balzert: *Software-Ergonomie*

[4] Allen Standardisierungsbemühungen zum Trotz wird wahrscheinlich heute noch kein PC-Besitzer große Mühe haben, einschlägige Beispiele bei den auf seinem Rechner installierten Softwaresystemen zu finden.

[5] Faktisch wird sich die Situation noch ungünstiger entwickeln. Die Anwender werden die Benutzung des nicht standardkonformen Programms (auch unbewußt) so weit wie möglich zu vermeiden versuchen. In der Folge sind sie im Umgang mit dem betroffenen Programm (immer) weniger geübt.

Aus allen diesen Gründen sollten auch die Bedienoberflächen von Simulationswerkzeugen den für das (die) Betriebssystem(e) üblichen Standards entsprechen.

8.1.4 Zusammenfassung

Die in den vorangegangenen Abschnitten dargelegten neuen Anforderungen an Simulationswerkzeuge seien hier noch einmal zusammengefaßt aufgeführt. Folgende Punkte wurden angesprochen:

- Um Modelle beliebiger Größe untersuchen zu können, dürfen keine Beschränkungen der Anzahl zulässiger Systemelemente (gleich welcher Art) existieren.

- Der Entwurf von Modellen und die Durchführung von Experimenten sollten unter einer Oberfläche integriert sein.

- Simulationswerkzeuge sollten über eine an die Gestaltungsrichtlinien des jeweiligen Betriebssystems angepaßte grafische Bedienoberfläche verfügen, die interaktive Eingriffe in laufende Experimente ermöglicht.

- Sie sollten Möglichkeiten zu ihrer Integration mit anderen Softwaresystemen bieten, um mit diesen Daten austauschen zu können. Es sollten z.B. Layoutinformationen (aus CAD-Systemen), Aufträge und Arbeitspläne (aus Datenbanken) und Steuerungsalgorithmen übernommen und Simulationsergebnisse und Steuerungsalgorithmen übergeben werden können. Wegen der mangelnden Standardisierung der entsprechenden Datenformate könnte der Austausch ohne direkte Kopplung unter Nutzung neutraler Zwischenformen (z.B. ASCII-Dateien) erfolgen.

- Simulationswerkzeuge sollten die online-Animation von Modellen in 2D (“animiertes Layout”) und 3D (bis hin zu VR-Darstellungen) ermöglichen. Mehrere gleichzeitige Animationen (z.B. in mehreren Fenstern) sollten möglich sein. In den Animationen sollten beliebige Ausschnitte setzbar sein (Zoom).

- Für die Prozeßnachbildung nicht immer erforderliche Funktionalitäten (z.B. Datenerfassungen und Animationen) sollten beliebig an- und abkoppelbar sein.

Läßt sich die Benutzung des Programms einmal nicht umgehen, kommt es daraufhin vermehrt zu Fehlern, die wiederum die Abneigung gegen das Programm verstärken. Diese Spirale pflanzt sich somit immer weiter fort.

- In Modellen sollten beliebige Steuerungsmechanismen abbildbar sein. Diese sollten in verbreiteten Programmiersprachen und in speziellen Notationen (Flußdiagramme o.ä.) dokumentiert werden können.

- Ergebnisse sollten in Form von Grafiken beliebiger Ausprägung (Tortendiagramme, Balkendiagramme, etc.) und beliebiger Gestaltung (Beschriftung, Farben, etc.) dokumentiert werden können. Der Erstellungsprozeß sollte (weitgehend) automatisierbar sein (gleiche Grafik aus verschiedenen Experimenten). Um die Weiterverarbeitung zu vereinfachen, sollten die Grafiken in verbreiteten Datenformaten vorliegen oder in solche überführbar sein.

8.2 Neue Werkzeuge

Die Realisierung der im vorigen Kapitel dargestellten Anforderungen erfordert offensichtlich die Weiter- oder Neuentwicklung von Simulationswerkzeugen. Da die Anforderungsliste umfangreich und, jedenfalls in einigen der aufgeführten Punkte, anspruchsvoll ist, wird die Weiterentwicklung vorhandener Werkzeuge wohl nicht immer möglich sein.

Umfassende Redesigns oder vollständige Neuentwicklungen müssen sicherlich am ehesten dann erwogen werden, wenn das entsprechende Simulationswerkzeug bzw. seine Grundkonzeption bereits ein gewisses Alter erreicht hat. Auch bei anderen Softwaresystemen kann beobachtet werden, daß in dieser Situation (und allgemein nach einigen Produktgenerationen) Neuimplementierungen erforderlich sind (und bei erfolgreichen Produkten auch vorgenommen werden). Neben dem Einbau neuer Funktionalitäten können beispielsweise die Anpassung an neue oder veränderte unterlagerte Betriebssysteme oder die Nutzung neuer Programmiertechniken Gründe für Neuauflagen von Softwaresystemen sein [1].

Von erheblicher Bedeutung in diesem Zusammenhang ist auch, daß die Simulation als Methode heute einen gewissen Reifegrad erreicht hat, allgemein anerkannt ist und vielfach eingesetzt wird [2]. Bei Software wie bei anderen technischen Systemen rückt mit dem Erreichen eines solchen Reifezustands der Aspekt grundlegender Funktionserfüllung zugunsten höherer

---

[1] So wurden beispielsweise praktisch alle PC-Textverarbeitungssysteme mit dem Betriebssystemübergang von DOS zu MS-Windows grundlegend überarbeitet. Systeme, deren Anbieter die Überarbeitung nicht, nicht umfassend genug oder verspätet durchführten, sind inzwischen vom Markt verschwunden. Bei den Textverarbeitungen kann das zu DOS-Zeiten recht verbreitete System WordStar als einschlägiges (Negativ-) Beispiel angeführt werden.

[2] vgl. Kapitel 7.2

Ansprüche an Gebrauchsnutzen und Anwendungsqualität in den Hintergrund. Als einschlägige Beispiele seien hier zum einen das Auto und zum anderen (aus dem Softwarebereich) wiederum Textverarbeitungssysteme genannt.

Autokäufer sind heute zunehmend weniger mit der Möglichkeit, nach individuellem Bedarf von A nach B gelangen zu können zufrieden, sondern wünschen immer öfter eine Klimaanlage, um dabei auch im Sommer nicht mehr schwitzen zu müssen. Auch bei den Textverarbeitungen hat sich die Aufmerksamkeit von den zuerst vorhandenen Möglichkeiten wie (gegenüber der Schreibmaschine) eleganterer Fehlerkorrektur und Montage von Texten aus Textbausteinen auf erweiterte Features wie umfangreiche Formatierungsmöglichkeiten, WYSIWYG ("What You See Is What You Get", d.h. druckbildentsprechende Textdarstellung schon am Bildschirm) und Einbindungsmöglichkeiten z.B. für Grafiken verlagert, ohne die keine Textverarbeitung mehr Markterfolge erzielen kann.

Für das Vorliegen eines Reifezustands der Simulation spricht neben der Selbstverständlichkeit der Anwendung auch, daß seit dem Erscheinen der Sprache Simula [1], also seit etwa 30 Jahren, keine grundlegend neue Technik zur Lösung der simulationstechnischen Grundaufgabe der Abbildung (zeit-) paralleler Aktivitäten mehr entwickelt wurde. Auch die Tatsache, daß die oben dargestellten Anforderungen an Simulationswerkzeuge keine elementaren Neuerungen sondern Qualitätsverbesserungen in der Anwendung intendieren kann als Hinweis in diese Richtung gedeutet werden.

Wie an den Beispielen Textverarbeitungssysteme und Autos dargestellt, wird sich in der Zukunft auch die Simulationstechnik in Richtung auf höhere Anwendungsqualität und verbesserten Gebrauchsnutzen entwickeln, was sich natürlich in den einschlägigen Werkzeugen niederschlagen wird. Ihre bisherigen (Basis-) Funktionen werden dabei erhalten (und wichtig) bleiben, deren fehlerfreies und problemloses Funktionieren wird allerdings von der zentralen Leistung zur selbstverständlich vorausgesetzten Grundlage für erweiterte Features, die schließlich allein über Erfolg oder Mißerfolg des Werkzeugs entscheiden werden.

8.2.1 Entwicklungslinien

Wie in Kapitel 7.5.2 ausgeführt wurde, ist die Erstellung eines experimentierfähigen Simulationsmodells gleichbedeutend mit der Implementierung eines aufgabenbezogenen Simulators. Diese kann wie dort dargelegt auf der Basis einer Programmiersprache, einer Simulatorentwicklungsumgebung oder eines bausteinbasierten Simulators erfolgen.

---

[1] vgl. Nygaard et al.: *The Development of the Simula Languages*

Als eine Folge der veränderten Anforderungen an Simulationswerkzeuge ist m.E. zu erwarten, daß experimentierfähige Modelle zukünftig zumeist auf der Grundlage bausteinbasierter Simulatoren implementiert werden. Entwicklungsumgebungen und Programmiersprachen (insbesondere dedizierte Simulationssprachen) werden dagegen eher als Fundament für die Programmierung bausteinbasierter Simulatoren eingesetzt werden. Für die Annahme einer derartigen Entwicklung sprechen eine ganze Reihe von Gründen.

Zunächst ist festzustellen, daß die Erstellung eines Simulationsmodells immer die Überwindung der Kluft zwischen den letztlich auf die Aufgabenstellung zurückgehenden Anforderungen und den Fähigkeiten des eingesetzten Werkzeugs impliziert. Bausteinbasierte Werkzeuge bieten im Vergleich zu Entwicklungsumgebungen und insbesondere zu Programmiersprachen höher integrierte Modellelemente als Entwurfsgrundlage [1]. Bei ihrem Einsatz ist also die erwähnte Kluft und damit die zu leistende Entwicklungsarbeit am geringsten. Offensichtlich ist gerade die derartige Aufwandsbegrenzung durch den Einsatz leistungsfähiger Werkzeuge ein probates Mittel, um dem zunehmenden Zeitdruck bei der Durchführung von Simulationsuntersuchungen [2] begegnen zu können [3].

Ein weiteres Argument folgt aus der Annahme, daß das für die Implementierung jeweils eingesetzte Werkzeug ausgereift, d.h. von verläßlicher Qualität und fehlerfrei ist [4]. Da andererseits Implementierungsfehler in Simulationsmodellen (wie in jeder Software) unvermeidlich sind [5] folgt, daß in Simulationsmodellen die Fehler in der Regel dort zu finden sind, wo das Werkzeug im Zuge der Modellerstellung parametriert, angepaßt oder erweitert wurde. Wegen der höheren Integration der Modellelemente ist dieser Anwendungsbereich des Werkzeugs bei bausteinbasierten Simulatoren im Vergleich zu Entwicklungsumgebungen und Programmiersprachen konzeptuell am kleinsten [6]. Die damit einhergehende stärkere Eingrenzung möglicher Fehlerorte verringert tendenziell den Zeitaufwand für die Modellprüfung und damit für die Untersuchung insgesamt.

---

[1] vgl. Kapitel 7.5.2

[2] vgl. Kapitel 8.1

[3] Auch die oben erwähnten leistungsfähigeren Textverarbeitungssysteme werden eingesetzt, um den gestiegenen Anforderungen an die (formale) Qualität damit erzeugter Texte genügen zu können.

[4] Zur Qualität von Software s.a. Booch: *Object-Oriented Analysis and Design ...*; S. 278 ff. Der Einsatz nicht ausgereifter Werkzeuge ist natürlich im Interesse des Projekterfolgs unbedingt zu vermeiden.

[5] vgl. Kapitel 8.1.3

[6] In Kapitel 7.5.2 ist in den Bildern 19 bis 21 (Seiten 89, 90 und 90) dieser Anwendungsbereich für den jeweiligen Werkzeugtyp türkis dargestellt.

Last, but not least ist auch hier der Aspekt der Anschaulichkeit zu nennen. In Kapitel 7.5.2 wurde ausgeführt, daß die Modellelemente bausteinbasierter Simulatoren häufig direkt den Komponenten der abzubildenden Systeme entsprechen. Wie dort dargelegt kann dadurch die Modellerstellung im Begriffssystem des Anwendungsgebiets erfolgen.

Die Termini dieses Begriffssystems sind im Unterschied zu solchen der Simulation als Methode oder des eingesetzten Werkzeugs auch den Adressaten einer Untersuchung und damit allen Beteiligten bekannt. Sie bilden ein transparentes Projektvokabular dessen durchgängige Verwendung es ermöglicht, alle im Rahmen einer Studie erforderlichen Gespräche auf einem aufgabennahen Niveau zu halten. Damit ist eine gute Grundlage für das wechselseitige Verständnis der Beteiligten gegeben, die die weitgehende Durchdringung der Aufgabenstellung fördert und die Erschließung kreativer Potentiale unterstützt. Offensichtlich verbessern sich so die Chancen, gute Lösungen zu finden.

Bausteinbasierte Simulatoren bieten also die besten Voraussetzungen, um die erweiterten Anforderungen an Simulationswerkzeuge erfüllen zu können. Die restlichen Abschnitte dieser Arbeit konzentrieren sich daher auf die Entwicklung bausteinbasierter Simulatoren.

8.2.2 Technologische Basis

Da Simulationswerkzeuge Software sind, müssen sie auf verfügbarer Computerhard- und -software implementiert werden. Die wesentliche Grundlage ist eine zu wählende Rechnerplattform, d.h. eine Kombination aus Hardware, Betriebs- und Basisgrafiksystem. Wegen ihrer Verbreitung bei potentiellen Anwendern von Simulationswerkzeugen kommen hierfür nach heutigem Stand eigentlich nur die bereits in Kapitel 7.5.3.1 angesprochenen Kombinationen Workstation/UNIX/X-Window bzw. PC/MS-Windows in Frage.

Bei der Softwareimplementierung ist es heute in der Regel nicht mehr erforderlich, Betriebssystemfunktionen oder gar die Rechnerhardware aus Anwendungen direkt anzusprechen. Während einige der hier zu nennenden Funktionalitäten bereits (weitgehend) plattformunabhängig auf der Ebene der Programmiersprachen standardisiert sind (z.B. Interaktionen mit dem Dateisystem), werden andere (z.B. Kommunikation mit der Bedienoberfläche) von der jeweiligen Plattform typischerweise als (Software-) Bibliotheken zur Verfügung gestellt, die über entsprechende Programmierschnittstellen (API - Application Programming Interface) angesprochen werden können. Ihr großer Leistungsumfang gestatten Anwendungen die Realisierung auch komplexer Aufgaben (z.B. die Erzeugung von und die Interaktion mit einem Dialogfenster) mit relativ einfachen Mitteln.

Obwohl funktional weitgehend vergleichbar sind diese Programmierschnittstellen plattform- bzw. betriebssystemabhängig. Bei der Realisierung plattformunabhängiger (oder jedenfalls auf mehreren Plattformen einsetzbarer) Werkzeuge ist dies eine ernstzunehmende Erschwernis, die im allgemeinen zu mehrfacher Programmierung der betroffenen Teile einer Software zwingt. Dem kann jedoch u.U. durch die Verwendung sogenannter Middleware begegnet werden. Dabei handelt es sich um auf verschiedenen Plattformen verfügbare Softwarebibliotheken, die für die von ihnen gebotenen Funktionalitäten zur Seite von Anwendungen hin systemunabhängige Schnittstellen bieten und so Plattformspezifika verbergen. Ein Beispiel eines solchen Systems zur Realisierung von Bedienoberflächen ist Tcl/Tk [1].

Für die Grafikprogrammierung haben solche Middleware-Systeme eine erhebliche Bedeutung. Sie implementieren einschlägige “offene”, d.h. plattformunabhängige Standards. Im Bereich der 3D-Grafik z.Zt. am wichtigsten ist wohl OpenGL [2]. In der näheren Zukunft könnte sich auch das noch recht junge VRML [3] weiter verbreiten. Von beiden Systemen sind Implementierungen verfügbar. Bei der Entwicklung von Simulationswerkzeugen können sie vor allem zur Realisierung von (grafischen) Entwurfs- und Animationskomponenten eingesetzt werden. Die Vorteile ihres Einsatzes liegen außer in der Plattformunabhängigkeit auch darin, daß sie durch ihre Leistungsfähigkeit die Entwicklung von Anwendungen wie z.B. Simulationswerkzeugen auf einem hohen, aufgabennahen Niveau unterstützen. Die so vermiedene Kleinteiligkeit in der Programmierung fördert tendenziell die Qualität (und die Produktivität) der Anwendungsentwicklung.

Ein wesentliches Element der technologischen Basis bei der Entwicklung von Softwaresystemen bildet die verwendete Programmiersprache. Neben den je spezifischen Möglichkeiten dieser Sprachen und den vorhandenen Kenntnissen der Entwickler stellen bei der Auswahl vor allem die Verbreitung einer Sprache und die Nutzbarkeit der oben angesprochenen APIs und Middlewaresysteme beachtenswerte Aspekte dar. Die Verbreitung bestimmt sich in der Praxis vor allem über die (auch perspektivisch zu erwartende) ausreichende Verfügbarkeit ausgereifter Compiler bzw. Entwicklungssysteme.

Eine Bewertung legt derzeit, gerade wegen der beiden letztgenannten Aspekte, die Verwendung von C bzw. C++ als Programmiersprache nahe. In der überschaubaren Zukunft kann sich wohl allenfalls Java [4] zu einer realistischen Alternative entwickeln.

---

[1] vgl. Osterhout: *Tcl und Tk: Entwicklung grafischer Benutzerschnittstellen für das X Window System*

[2] vgl. OpenGL Architecture Review Board: *OpenGL Specification & Manual Pages*

[3] VRML Consortium: *The Virtual Reality Modeling Language*

[4] vgl. Sun Microsystems: *Java Platform Documentation*

8.2.3 Schlüsselelemente

Jede Simulationsmodell ist ein Abbild eines (geplanten oder existierenden) technischen Systems. Es integriert anlagen-, produkt- und steuerungsbeschreibende Teilmodelle zu einem dynamischen Fabrikmodell [1]. Die in Bild 17 (Seite 86) angegebenen Elemente dieser Teilmodelle und des Gesamtmodells sind damit die zentralen Termini zur Beschreibung technischer Systeme und der sie abbildenden Simulationsmodelle.

Es ist offensichtlich erforderlich, daß diese Elemente in Simulationswerkzeugen direkte Entsprechungen haben. So wie z.B. Arbeitspläne in PPS-Systemen verwaltet werden können, muß dies auch in Simulationswerkzeugen bzw. -modellen möglich sein. Arbeitspläne müssen eingegeben und modifiziert werden können und sie müssen bei der Durchführung von Experimenten verwendet werden können [2]. Vergleichbare Anforderungen bestehen auch für die übrigen in Bild 17 angegebenen Modellelemente. Diese sind damit die wesentlichen Schlüsselelemente für das Design von Simulationswerkzeugen.

Auf der Basis so aufgebauter Simulationswerkzeuge entwickelte Simulationsmodelle können als eine Menge derartiger Elemente aufgefaßt und beschrieben werden, die in definierter Weise miteinander interagieren.

8.2.4 Programmierparadigma

Die Entwicklung einer Software impliziert, bewußt oder unbewußt, immer die Anwendung eines Programmierparadigmas. Dies ist definiert als "*eine Art, Programme auf der Grundlage eines konzeptuellen Programmiermodells und einer passenden Programmiersprache aufzubauen, um den Programmen einen klaren Stil zu geben* [3]."

Ein Programmierparadigma impliziert eine bestimmte Sichtweise auf das der Programmieraufgabe zugrundeliegende System. Diese Sichtweise bestimmt die Art der Abstraktionen, die in dem System als wesentlich identifiziert werden und die für die Anwendung wichtigen

---

[1] vgl. Kapitel 7.5

[2] Die Existenz von Arbeitsplänen und vergleichbaren logischen Einheiten auch in Simulationsmodellen ist darüberhinaus eine wichtige Voraussetzung, um sie (wie in Kapitel 7.5.1 dargelegt) über Schnittstellen aus externen Datenbeständen übernehmen zu können.

[3] Bobrow, Stefik: *Perspectives on Artificial Intelligence Programming*

Details des Systems beschreiben. Die Verwendung einer zu einem Programmierparadigma "passenden" Programmiersprache erlaubt durch die Bereitstellung entsprechender Konstrukte die (möglichst) direkte Umsetzung der gefundenen Abstraktionen

Die folgende Tabelle zeigt verbreitete Programmierparadigmen zusammen mit den verwendeten Abstraktionen und Beipielen passender Programmiersprachen [1].

Name	Abstraktionen	Programmiersprachen
Prozedurorientiert	Algorithmen	FORTRAN, Pascal, C
Objektorientiert	Klassen und Objekte	Smalltalk, C++, Java
Regelorientiert	Wenn-Dann-Regeln	Prolog

Wegen des grundlegend unterschiedlichen Charakters der verwendeten Abstraktionen ist keines der aufgeführten Paradigmen für alle Arten von Softwaresystemen gleichermaßen geeignet. Die regelorientierte Programmierung beispielsweise ist gut geeignet für Wissensdatenbanken. Für das Design von rechenintensiven Systemen dagegen ist sicher die Prozedurorientierte Programmierung am ehesten einsetzbar. Das objektorientierte Paradigma scheint für die breitgefächertste Palette von Anwendungen geeignet zu sein [2].

In Kapitel 8.2.3 wurden als Schlüsselelemente und damit als wesentliche Abstraktionen in Simulationsmodellen im wesentlichen solche identifiziert, die Objekte bzw. Klassen sind. Aus diesem Grund ist für die Entwicklung von Simulationswerkzeugen die objektorientierte Programmierung [3] ein geeignetes Paradigma. Es wird daher im folgenden angewandt.

[1] nach Booch: *Object-Oriented Analysis and Design with Applications*; S. 40

[2] ebd.; S. 40

[3] Eine ausführliche Darstellung der objektorientierten Programmierung bietet z.B. Booch: *Object-Oriented Analysis and Design with Applications*.

IV Modellierungsansätze für Simulationswerkzeuge

9 Einführung

Gegenstand der noch folgenden Kapitel dieser Arbeit ist die Vorstellung von Modellierungsansätzen für die Realisierung von Softwarewerkzeugen zur Simulation von Materialflußsystemen. Die Ansätze entstanden als Ergebnis einer Vorgehensweise, die sich an den von Booch [1] beschriebenen Entwurfsprozeß objektorientierter Software anlehnte [2]. Sie ergeben insgesamt ein objektorientiertes Design, das die Abbildung von Materialflußsystemen in experimentierfähigen Simulationsmodellen beschreibt und ihre Bearbeitung in einem entsprechenden Softwarewerkzeug skizziert.

In der Summe bieten diese Ansätze alle Möglichkeiten, um mit auf sie aufbauenden Simulatoren alle in Kapitel 8.1.4 genannten Anforderungen an solche Werkzeuge erfüllen zu können. Beim Entwurf wurden vier wesentliche Ziele verfolgt. So sollen für den schnellen Modellaufbau Bausteine zur Verfügung stehen, die aufgabennah und leistungsfähig genug sind, um die zu überwindende Kluft zwischen Aufgabenstellung und Werkzeugfähigkeiten klein zu halten [3]. Weiter soll die Geometrie der zu untersuchenden Anlagen vollständig und richtig dreidimensional abbildbar sein, um ein ausreichende Grundlage für 3D-Animationen, VR-Visualisierungen und die Datenübernahme aus (3D-) CAD-Systemen zu

---

[1] Booch: *Object-Oriented Analysis and Design with Applications*; S. 229 ff.

[2] Die Abweichungen von der Booch-Methode ergaben sich, weil die vorliegende Arbeit von einer Einzelperson als Konzept mit teilweiser und prototypischer Implementierung erstellt wurde, während Booch die industrielle Softwareentwicklung im Team adressiert.

[3] vgl. Kapitel 8.2.1

bieten. Drittens sollen beliebige Steuerungsalgorithmen in die Modelle integriert werden können. Zwar soll die Steuerung aller Modellelemente und ihres Zusammenwirkens bei Bedarf vollständig übernommen werden können, doch sollen einfache Standardalgorithmen die schnelle Entwicklung prototypischer Modelle unterstützen. Schließlich sollen die entworfenen Modelle bei Anwendung hierfür üblicher Vorgehensweisen auch in der Anlagensteuerung einsetzbar sein.

Neben diesen Zielen waren vor allem die grundlegenden Eigenschaften von Materialflußsystemen Ausgangspunkt der Überlegungen. Die vorgenommene Verteilung der Gesamtlösung auf die verschiedenen Ansätze hängt mit diesen Eigenschaften eng zusammen. Die Architektur der modellierten Systeme erlaubt es, die enthaltenen Typen und Komponenten (weitgehend) unabhängig voneinander zu betrachten, zu verstehen und zu modellieren. Jeder Ansatz behandelt dementsprechend die Abbildung eines bestimmten Teils der in Materialflußsystemen anzutreffenden Typen und Komponenten. Der folgende Abschnitt stellt die entwickelten Ansätze und die Verteilung ihrer Beschreibungen auf die weiteren Kapitel vor.

9.1 Übersicht

Wie vermutlich alle Designs nichttrivialer Softwaresysteme so stützt sich auch dieser Entwurf auf eine Reihe von auf konzeptuell niedriger Ebene angesiedelten grundlegenden Modulen und Subsystemen. Ihr Sinn besteht darin, regelmäßig wiederkehrende einfache Teilaufgaben in einheitlicher Weise zu erledigen und ihre Nutzer von der Beachtung komplexer Details zu befreien, um ihnen das Arbeiten auf einer höheren Abstraktionsebene zu ermöglichen. Die im Rahmen dieser Arbeit relevanten grundlegenden Module und Subsysteme ermöglichen die einheitliche Behandlung auszugebender Meldungen, die Abbildung von Zeit und von Zufallszahlenströmen, die Verwaltung von Objekten in dynamischen Datenstrukturen, die Identifikation von Objekten sowie die Abbildung und Behandlung mathematischer Objekteigenschaften aus dem Bereich der Geometrie. Diese Grundlagenmodule und -subsysteme werden im nachfolgenden Kapitel vorgestellt.

Der Ansatz für die wichtige Teilaufgabe der Abbildung der Objektgeometrie und der Visualisierung wird in Kapitel 11 vorgestellt.

Die Vorstellung eines Zeitmechanismus zur Abbildung (zeitlich) paralleler Vorgänge ist Gegenstand von Kapitel 12.

Daran anschließend wird dargelegt, wie die in Materialflußsystemen anzutreffende Vielfalt von Fördertechnikkomponenten durch Komposition aus wenigen Grundelementen zweckmäßig

abgebildet werden kann. Mit dem hierfür entwickelten Ansatz befaßt sich Kapitel 13. Die materialflußtechnische Verkettung dieser Komponenten bestimmt die Wege, die Teile auf ihrem Weg durch das System nehmen können. Daher wird auch ein Konzept zur Abbildung von Verkettungen und der darauf gestützten Wegesuche behandelt.

Die Abbildung der ein Materialflußsystem durchlaufenden Teile stellt Kapitel 14 vor.

Vor allem in produktionstechnischen Anlagen haben die durchlaufenden Teile oft einen zugeordneten Arbeitsplan. Arbeitspläne legen insbesondere eine Folge von Bearbeitungsvorgängen fest, die an den Teilen durchzuführen sind und bestimmen daher mit über den Weg der Teile durch das System. Die Abbildung von Arbeitsplänen behandelt Kapitel 15.

Die Vorstellung eines erweiterten Einheitenkonzepts ist Gegenstand von Kapitel 16. Es führt über die Fördertechnikkomponenten hinaus weitere Typen statischer Systemkomponenten ein. Alle diese Typen werden in eine gemeinsame Hierarchie eingeordnet.

Mit der Modellierung von Lagern und der Ein- bzw. Auslagerung von Teilen in solche Einheiten befaßt sich Kapitel 17.

Das Kapitel 18 stellt ein Konzept zur Abbildung Störungen und Ankunftsströmen vor.

Kapitel 19 beschreibt die Programmierung der Abläufe in Systemen bzw. Modellen. Es werden diejenigen Situationen behandelt, die für den Gesamtzusammenhang bzw. -ablauf am bedeutendsten sind. Neben den bestehenden Möglichkeiten zur Umsetzung anlagen- bzw. aufgebenbezogener Steuerungen durch individuelle Programmierung wird auch die Standardbehandlung der jeweiligen Situation vorgestellt.

Einen Ansatz zur Speicherung und zum (Wieder-) Einlesen von Modellen behandelt Kapitel 20.

Die Grundzüge eines zu den gefundenen Ansätzen passenden Simulationswerkzeugs stellt Kapitel 21 vor.

Abschließend wird in einem letzten Kapitel die Eignung der beschriebenen Ansätze für den Einsatz von Simulationsmodellen in der Anlagensteuerung untersucht.

9.2 Notation

Alle in den folgenden Kapiteln dargestellten Codefragmente und Programmierbeispiele sind in C [1] bzw. C++ [2] geschrieben. Diese Sprachen sind durch internationale Normen standardisiert und stellen derzeit den de-facto-Standard in vielen Applikationsbereichen dar, so daß viele Menschen sie lesen und verstehen können.

Da die im folgenden vorgestellten Teilsysteme auch in C++ implementiert sind, ergibt sich als weiterer Vorteil der Verwendung dieser Sprachen zur Dokumentation, daß Übertragungsverluste aus der Übersetzung in eine andere Notation ausgeschlossen sind. Im übrigen erlaubt dieses Vorgehen auch, auf die Entwicklung und (vor allem) Beschreibung einer eigenen Notation an dieser Stelle verzichten zu können.

Die dargestellten Codefragmente und Programmierbeispiele sind auf das im jeweiligen Sachzusammenhang unbedingt notwendige Mindestmaß beschränkt. Sie sollen die Ausführungen im Text der Arbeit illustrieren und deren Verständnis unterstützen. Im Interesse von Kürze, Übersichtlichkeit und Aussagekraft wurde daher auf eine "kugelsichere" Ausgestaltung verzichtet.

Außer auf Codefragmente wird zur Dokumentation der entworfenen Teilsysteme in den folgenden Kapiteln auch auf Elemente der von Booch et al. entwickelten "Unified Modeling Language" [3] zurückgegegriffen. Diese Notation für die Beschreibung von Softwaresystemen definiert eine Vielzahl von Diagrammtypen zur Dokumentation von Softwaredesigns. Von diesen werden hier vor allem Klassen-, Objekt- und Zustandsdiagramme zur Erläuterung des Aufbaus und der Funktionsweise der entwickelten Ansätze verwendet.

---

[1] vgl. Schildt: *The Annotated ANSI C Standard*

[2] vgl. Stroustrup: *Die C++ Programmiersprache*

[3] vgl. Booch et al.: *The Unified Modeling Language - UML Notation Guide*

10 Grundlagenmodule

10.1 Meldungen

In umfangreichen Softwaresystemen wie es Materialflußsimulatoren sind, besteht regelmäßig die Notwendigkeit, bei Eintritt von Inkonsistenzen wie der Verletzung einzuhaltender Bedingungen Warnungen oder Fehlermeldungen an Anwender auszugeben. Während die eigentliche Ausgabe über die (wie auch immer gestaltete) Bedienoberfläche der Software zu geschehen hat, ist der Ort, an dem die Inkonsistenz festgestellt wird, häufig dem "logischen Kern" des Softwaresystems zuzurechnen. Es sind also Teilsysteme involviert, die, vom Standpunkt des Programmentwurfs betrachtet, idealerweise "nichts voneinander wissen", also softwaretechnisch vollständig entkoppelt sind.

Der Zweck des hierfür entworfenen Moduls besteht darin, diese Entkopplung aufrechterhalten zu können. Es bietet dazu eine Schnittstelle, über die die Ausgabe von Warnungen und Fehlermeldungen veranlaßt werden kann, ohne daß Aufrufer irgendwelche Kenntnisse über die Art und Weise haben müssen, in der die Ausgaben Anwendern präsentiert werden.

Über andere Teile der Modulschnittstelle, die sinnvollerweise von den die Bedienoberfläche realisierenden Teilsystemen einer Software angesprochen werden, kann die Ausgabe gesteuert werden. Dabei ist neben der Ausgabe in einen Standardausgabekanal des Programms und

der Umlenkung der Meldungen in Dateien [1] auch die Angabe einer "Rückruffunktion" (Callback) möglich, die dann für jede Meldungsausgabe aufgerufen wird und die ihrerseits beispielsweise ein Bildschirmfenster öffen kann. Durch diese Lösung ist auch das Meldungsmodul selbst von der Bedienoberfläche der nutzenden Software unabhängig.

10.2 Abbildung von Zeit

Ein wesentlicher Nutzen und Vorteil der Materialflußsimulation besteht darin, das Verhalten des betrachteten Systems über lange Zeiträume untersuchen zu können [2]. Die Realisierung der hierfür erforderlichen Zeitraffung impliziert, daß Simulationsexperimente in einem virtuellen Zeitsystem ablaufen, dessen technische Umsetzung Aufgabe und Bestandteil des verwendeten Simulationswerkzeugs ist.

Für den im Simulatorkern enthaltenen Zeitmechanismus [3] und für die Erfassung von Daten wie Auslastungen oder Störzeiten o.ä. ist folglich die Berechnung und Speicherung von Zeiten (Zeitpunkten und -spannen) elementare Grundlage, um beispielsweise die Eintrittszeitpunkte von Ereignissen oder die Dauer von Aktivitäten berechnen zu können.

Solche Zeiten müssen also im Simulationsmodell bzw. -werkzeug als Werte, d.h. als Variablen oder Objekte abbildbar sein. Hierfür ist zunächst eine Basiseinheit (z.B. Sekunden oder Minuten) modellabhängig zu wählen oder fest (werkzeugabhängig) vorzugeben [4]. Ansonsten ist für die Zeiten zu fordern, daß

- negative Werte zulässig sind [5],
- die erforderlichen Operationen Zuweisung, Addition, Subtraktion und Vergleiche durchführbar sind,

---

[1] Diese Funktionalität ist gut für die Durchführung bedienungsloser Simulationsexperimente beispielsweise im Batch-Betrieb geeignet.

[2] vgl. Kapitel 7.4

[3] vgl. Kapitel 7.5.1

[4] Dabei ist zu bedenken, daß die getroffene Wahl Nebeneffekte haben kann. Bei Verwendung von Minuten als Basiseinheit kann beispielsweise ein elementarer Wert wie 1 Sekunde nicht exakt abgebildet werden, da 1/60 periodisch ist. Die feste Vorgabe der Basiseinheit ist sinnvoll, wenn das Werkzeug von der Zeiteinheit abhängige Daten wie z. B. Geschwindigkeiten direkt verarbeiten kann, da sonst die Parametrierung von Modellen u.U. umfangreiche und entsprechend fehlerträchtige Umrechnungen erfordert.

[5] Dies vereinfacht Operationen wie z.B. die Berechnung von Terminabweichungen.

- der Wertebereich hinreichend groß ist um auch Experimente mit langen Laufzeiten durchführen zu können,
- die Auflösung bzw. Genauigkeit hoch und womöglich anpassbar ist und
- die Rechengenauigkeit über den gesamten Wertebereich konstant ist.

Implementierung **Anforderungen**	**int** **(4 Byte)**	**double** **(8 Byte)**	**class SimTime** **(int Sec, int Frac)**
Operationen: =, +, +=, -, -=, ==, !=, <, <=, >, >=	vorhanden	vorhanden	Programmierung erforderlich (Aufwand)
möglichst großer Wertebereich	-0,68a < x < 0,68a (Auflösung 1/100s)	Wechselwirkung mit Genauigkeit	-68a < x < 68a
hohe Auflösung (anpassbar)	Wechselwirkung m. Wertebereich	Wechselwirkung mit Genauigkeit	Bruchteile anpassbar (bis 10^{-9})
konstante Rechengenauigkeit über Wertebereich	ok	betragsabhängig, nicht alle mögl. Werte darstellbar	ok

Bild 22: Mögliche Implementierungen von Zeit im Vergleich mit den Anforderungen

Für die Implementierung von Zeiten kommen als Datentyp von der verwendeten Programmiersprache oder dem eingesetzten Compiler vorgegebene oder ein selbstdefinierter Typ bzw. eine solche Klasse in Frage. Die vorstehende Tabelle stellt die in C++ prinzipiell bestehenden Möglichkeiten den Anforderungen gegenüber [1].

Die grau hinterlegten Felder der Tabelle weisen auf Probleme hin, die auftreten, wenn Zeiten auf Basis vordefinierter Datentypen implementiert werden. Um diese zu umgehen, wurde das Modul *SimTime* entworfen, das zur Abbildung von Zeiten eine gleichnamige Klasse enthält. Die interne Darstellung nutzt zwei Variable, in denen die in der jeweiligen Zeit enthaltenen Sekunden (*Sec*) bzw. Sekundenbruchteile (*Frac*) gespeichert sind.

[1] "Kleinere" ganzzahlige Typen als int (z.B. short) haben auch einen kleineren Wertebereich. Der Typ long long int ist nicht standardisiert (vgl. Schildt: *The Annotated ANSI C Standard*; Kap. 6.1.2.5) und damit nicht auf jedem Compiler verfügbar. Die Genauigkeit des reellen Typs float ist mit nur 6 Dezimalstellen so gering, daß bei einer Auflösung von 1/100 s Berechnungen mit Werten größer 1000 s (entsprechend ca. 16,7 min.) nicht mehr genau sind.

Die Klasse ist mit Operatoren und anderen Funktionen so ausgestattet, daß Nutzer sie wie einen vordefinierten Typ verwenden können. Durch eine Vielzahl von Ausgabe- und Umwandlungsfunktionen können *SimTime*-Zeiten Anwendern in gewohnten Darstellungen (z.B. in der Form “hh:mm:ss”) präsentiert und auch so entgegengenommen werden.

10.3 Zufallszahlenströme

Die Erzeugung von Zufallszahlenströmen ist eine essentielle Grundlage für die Implementierung von Simulationswerkzeugen und vielen anderen Softwaresystemen [1]. Entsprechende Verfahren wurden daher bereits früh entwickelt und sind heute ausgereift und gut dokumentiert [2].

Für die Erzeugung von Zufallszahlenströmen wurde ein Subsystem entwickelt, das zwei Hierarchien von Klassen umfaßt. Die Instanzen der Klassen der Hierarchie *DNG* (Distributed Number Generator) repräsentieren jeweils einen Zahlenstrom, dessen Einzelwerte einer vorgegebenen Verteilung genügen. Sie erzeugen die einzelnen Zahlen unter Rückgriff auf je eine Instanz einer Klasse der Hierarchie *RNG* (Random Number Generator). Diese Objekte repräsentieren ebenfalls Zahlenströme. Sie liefern jedoch ausnahmslos Zahlen, die über einen bestimmten Bereich gleichverteilt sind.

Die Erzeugung einer Folge (beispielsweise) normalverteilter Zahlen geschieht also hier in einem zweistufigen Verfahren. Der wesentliche Vorteil dieses Vorgehens ist, daß bei der Programmierung einer beliebigen Klasse aus einer der Hierarchien des Subsystems jeweils ein eindeutiger Fokus vorgegeben ist. Bei den Klassen der Hierarchie *RNG* ist dies die Erzeugung möglichst gut gleichverteilter Zahlen. Klassen aus der *DNG*-Hierarchie andererseits können sich darauf beschränken, auf eine vorliegende Gleichverteilung die jeweils gewünschte Zielverteilung aufzuprägen.

Die Klassen der Hierarchie *RNG* implementieren je ein bestimmtes Verfahren zur Gewinnung gleichverteilter Zufallszahlen. Die Instanzen der Klassen repräsentieren also je einen

---

[1] Genau genommen sind maschinell (also mit dem Computer) erzeugte Zufallszahlenströme nie wirklich “zufällig”, da sie (notwendigerweise) von exakten Algorithmen berechnet werden. Die Reproduzierbarkeit von Zufallszahlen ist für die Implementierung von Simulationswerkzeugen jedoch (anders als z.B. bei kryptografischen Anwendungen) durchaus willkommen, da sie erst die unumgänglich notwendige Reproduzierbarkeit von Simulationsexperimenten ermöglicht.

[2] Der erste entsprechende Algorithmus wurde von John von Neumann schon 1946 vorgestellt. Eine ausführliche Übersicht gibt z.B. Knuth: *The Art of Computer Programming; Vol. 2: Seminumerical Algorithms*. Auf diese Darstellung stützen sich auch die hier implementierten Verfahren.

(unabhängigen) Zufallszahlenstrom. Sie liefern die jeweils nächste Zufallszahl entweder als ganzzahligen Wert (unsigned) zwischen 0 und einem (klassenspezifischen) Maximum (exakt: $0 \leq x \leq Max$) oder als rellen Wert (double) zwischen 0.0 und 1.0 (exakt: $0.0 \leq x \leq 1.0$). Die verfügbaren Klassen bzw. Verfahren sind der Systemgenerator (Verwendung der *rand ()*-Funktion der C-Standardbibliothek), die Quadrat-Mitten-Methode nach v. Neumann, die Lineare und die Multiplikative Kongruenzmethode nach Lehmer, eine quadratische Methode nach Coveyou, die Nutzung von Fibonacci-Zahlen sowie eine additive Methode nach Mitchell und Moore [1].

Die Klassen der Hierarchie *DNG* implementieren je eine bestimmte Verteilung, die sie auf einen (von einem *RNG*-Objekt erzeugten) gleichverteilten (Basis-) Zahlenstrom aufprägen. Die Parameter der Verteilungen sind bei der Erzeugung anzugeben und somit frei wählbar. Die verfügbaren Klassen bzw. Verteilungen sind Konstantwert (eine degenerierte Form einer Verteilung), Gleichverteilung, Negativ-Exponentialverteilung, Gauss- oder Normalverteilung, Erlangverteilung und Poissonverteilung [2].

10.4 Dynamische Datenstrukturen

Dynamische Datenstrukturen wie Felder, Listen und ähnliche Einheiten von Objekten werden als Container bezeichnet. Zur Laufzeit eines Programms können Objekte in solche Einheiten beispielsweise eingeordnet, aus ihnen entfernt oder darin gesucht werden. Sie ermöglichen also die Anordnung und Verwaltung von Objekten in komplexeren Konfigurationen. Auf der Benutzungsebene ist so ein höheres Abstraktionsniveau erzielbar.

Obwohl entsprechende Bibliotheken von Softwareherstellern verfügbar sind, wurde der Weg über eine eigene Implementierung von Containernklassen vorgezogen. Es entstand das Subsystem *Structures* als plattformunabhängige Lösung, deren Leistungsumfang auf die Bedürfnisse der benutzenden Teilsysteme zugeschnitten ist.

Um die Homogenität aller von Nutzern erzeugten Container zu gewährleisten, umfaßt das Subsystem ausschließlich generische Klassen, die für die Verwendung instanziiert werden

---

[1] Alle genannten Methoden beschreibt und bewertet Knuth: *The Art of Computer Programming; Vol. 2: Seminumerical Algorithms*; S. 3ff., so daß auf eine ausführliche Darstellung hier verzichtet wird. Ergänzend ist anzumerken, daß die Fibonacci- und die Quadrat-Mitten-Methode nur als Zitate klassischer Verfahren und trotz unbestreitbarerer qualitativer Mängel aufgenommen wurden.

[2] Die genannten Verteilungen und ihre Parameter sind in der Stochastik hinreichend beschrieben, so daß hier wiederum auf eine ausführliche Darstellung verzichtet wird.

müssen [1]. Hierfür fand der template-Mechanismus von C++ Verwendung. Alle realisierten Containerklassen arbeiten nur mit Verweisen auf die enthaltenen Objekte, so daß diese weder beim Einordnen kopiert noch beim Austragen gelöscht werden.

Das Subsystem stellt Stacks, Queues, Deques, Listen, Arrays, dynamische Arrays (*DynArray)* und Maps zur Verfügung. Da derartige Containerklassen in der Softwaretechnik allgemein bekannt sind und für jeden genannten Typ eine hinreichend eindeutige Semantik definiert ist [2], wird an dieser Stelle auf weitergehende Erläuterungen verzichtet.

10.5 Identifikation von Objekten

Die Identität eines Objektes unterscheidet es von allen anderen (auch gleichartigen) Objekten [3]. Die Bewahrung der Identität von Objekten in einer Software wie z.B. einem Simulationsmodell bzw. -werkzeug erfordert das Vorhandensein eines (eindeutigen) Identifikationsmerkmals als Teil des Objektzustands.

Als Identifikationsmerkmal werden herkömmlicherweise (von Hand oder automatisch vergebene) ganzzahlige Schlüsselnummern verwendet, um das betreffende Objekt in einer Verwaltungsstruktur (beipielsweise einer Liste) auffinden zu können. Die Verwendung von Namen anstelle von Nummern zur Objektidentifizierung ist wünschenswert (und beim heutigen Stand der Technik auch realisierbar [4]), da sie die Transparenz des Modells bzw. aller mit einem Simulationswerkzeug bearbeiteten Modelle erhöht [5].

---

[1] vgl. z.B. Booch: *Object-Oriented Analysis and Design with Applications*; S. 131 f.

[2] ebd.; S. 330 ff.

[3] ebd.; S. 91 ff.

[4] Der klassische Einwand gegen die Verwendung von Namen ist der höhere Laufzeitaufwand, der daraus resultiert, daß statt schneller Zahlenvergleiche aufwendige string-Vergleiche zum Auffinden eines Objekts erforderlich sind. Dieser Einwand greift nicht mehr, da heute (jedenfalls in C / C++-Bibliotheken) effiziente Vergleichsfunktionen für strings verfügbar sind, die zudem seltener benötigt werden, da unter den heute üblichen grafischen Bedienoberflächen die Identifikation von Objekten statt über die (manuelle) Eingabe des Identifikationsmerkmals in eine Maske zunehmend durch "Anklicken" mit der Maus erfolgt.

[5] Die Verwendung von Namen erlaubt, bei der Modellimplementierung dichter an der verbalen bzw. textuellen Systembeschreibung (und damit mehr auf der Anwendungsebene) zu bleiben. Existiert im System beispielsweise ein "Heber" genanntes Fördertechnikelement, so kann es auch im Modell unter diesem Namen angelegt werden. Die Verwendung von Nummern oder anderen Schlüsseln erfordert dagegen die Bildung einer modellbezogenen Übersetzungstabelle.

Hierfür wurde ein Modul implementiert, das im wesentlichen eine Klasse *Identity* zur Verfügung stellt. Alle Klassen von Objekten einer Anwendung, die über Namen identifizierbar sein sollen, sind aus dieser Klasse durch Ererbung abzuleiten.

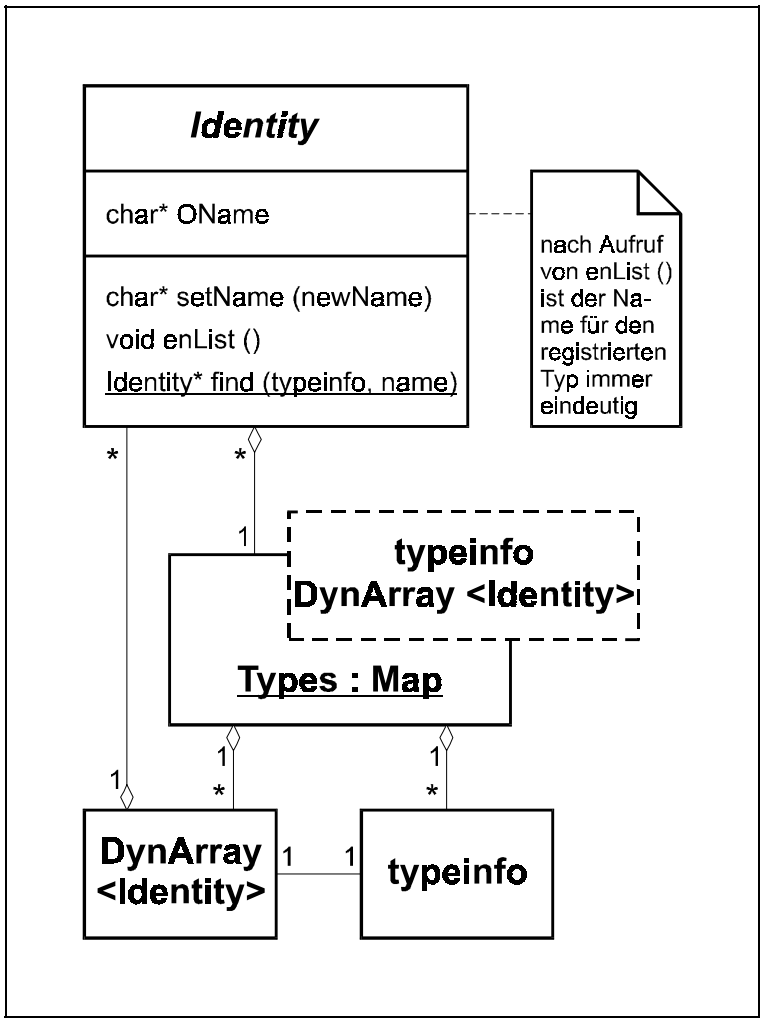

Bild 23: Struktur von *Identity*-Objekten

Die softwaretechnische Struktur von *Identity*-Objekten zeigt Bild 23. Sie enthalten im wesentlichen den identifizierenden Namen, der jederzeit geändert werden kann.

Bei Bedarf können die Objekte abgeleiteter Klassen (durch Aufruf der Funktion *enList ()*) in einer Verwaltung registriert werden in der sie nach ihrem Typ in Listen eingeordnet sind [1]. Andersherum können registrierte Objekte durch Angabe ihres Typs und ihres Namens in der Verwaltung gesucht werden.

Weiterhin ist für registrierte Objekte garantiert, daß kein anderes Objekt dieses Typs mit dem selben Namen existiert, so daß der Name als eindeutiges Identifikationsmerkmal dienen kann. Erforderlichenfalls wird hierfür bei der Registrierung bzw. Namensänderung der Name des betroffenen Objekts um ein die Eindeutigkeit herstellendes Suffix erweitert.

[1] Die (automatische) Feststellung des Objekttyps stützt sich auf den *runtime type identification*-Mechanismus von C++.

10.6 Mathematik für geometrische Berechnungen

Zur Unterstützung der geometrischen Modellierung von Körpern und ihrer Bewegungen wurde das Modul *XMath* geschaffen. Es enthält Klassen zur Abbildung von Winkeln, Vektoren und Matrizen.

Die Winkelklasse *Angle* bietet im wesentlichen drei Funktionalitäten. Da die Winkel intern als ganzzahlige Werte gespeichert werden, stellt sie zunächst eine gleichbleibende und hohe Genauigkeit bei Winkelberechnungen sicher. Weiterhin fängt sie Ungenauigkeiten der trigonometrischen Funktionen der mathematischen Standardbibliothek von C bzw. C++ ab [1] und verhindert so die Fehlerfortpflanzung in komplexen Berechnungen. Schließlich verbirgt sie die häufig notwendigen Umrechnungen zwischen der Winkeldarstellung im Bogenmaß (erforderlich für die Nutzung der trigonometrischen Funktionen der mathematischen Standardbibliothek) und der von Anwendern gewohnheitsmäßig genutzten Darstellung in Grad.

Die Vektorklassen *Vec3D* und *Vec2D* bilden (wie die Namen nahelegen) zwei- bzw. dreidimensionale Vektoren ab. Die Instanzen der Matrizenklassen *Hmatrx3D* und *Hmatrx2D* sind harmonische Matrizen in jeweils passender Dimensionalität. Die Klassen bieten alle für die Durchführung geometrischer Berechnungen erforderlichen Funktionalitäten entsprechend den einschlägigen mathematischen Konventionen [2].

---

[1] Beispielsweise gibt sin (x) in vielen Implementierungen von C / C++ nicht immer eine exakte 0 zurück, wenn das Argument x zwar (mathematisch) den Wert 0 hat, es aber als (komplexer) Ausdruck angegeben wird, der zuerst ausgewertet werden muß.

[2] Grundzüge der Funktionalität und Anwendung dieser Klassen werden wegen des Sachzusammenhangs in Kapitel 11 dargestellt.

11 Objektgeometrie und Visualisierung

Die als Entwurfsziel vorgegebene [1], ursprünglich in Kapitel 8.1.4 erhobene Forderung, Möglichkeiten zur online-Animation von Simulationsmodellen bis hin zu VR-Darstellungen zu bieten, erfordert offensichtlich, daß die räumliche Lage und Gestalt aller (sichtbaren) Modellelemente in irgendeiner Weise Bestandteil des Modells ist.

Die ergänzenden Forderungen nach mehreren gleichzeitig sichtbaren Animationsfenstern und Abschaltbarkeit der Animation sowie die Notwendigkeit, die interne Struktur des Simulationswerkzeugs bzw. -modells möglichst einfach zu halten, bedingen weiterhin eine klare softwaretechnische Trennung zwischen der "Modelllogik" (die Veränderungen der räumlichen Lage und evtl. auch der Gestalt von Objekten verursacht) einerseits und allen Aspekten der sichtbaren Darstellung(en) des Modells (durch die Anwender die Gestalt und Lage der Objekte sehen) andererseits.

Im Rahmen dieser Arbeit wurde zusammen mit Herrn J. Bernhard ein Konzept für die geometrische Objektrepräsentation und Visualisierung entwickelt, daß die beschriebene Trennung von Logik einerseits und Sichtbarmachung andererseits durch die Verteilung der entsprechenden Zuständigkeiten auf Instanzen von verschiedenen Klassen realisiert [2].

---

[1] vgl. Kapitel 9

[2] Hier nicht dargestellte Teile dieses Konzepts und seine Implementierung sind in Bernhard: *Entwicklung einer Geometrie- und Animationsbibliothek ...* beschrieben.

Das Konzept ist auf den drei grundlegenden Abstraktionen “Geometrieobjekt”, “Raum” und “Ansicht” aufgebaut, die im folgenden vorgestellt werden. Ihre konkreten Realisierungen sind Instanzen entsprechender Klassen, die die Gesamtfunktionalität durch Kooperation untereinander auf weitgehend transparente Weise erreichen.

11.1 Geometrieobjekte

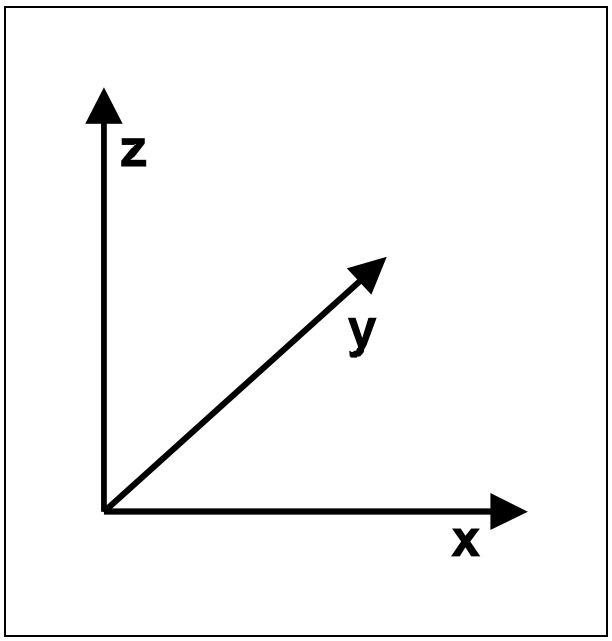

Bild 24: Kartesisches R-Koordinatensystem

Die (mathematische) Beschreibung der Lage von Objekten in dreidimensionalen Räumen erfordert zunächst die Vereinbarung eines globalen oder Weltkoordinatensystems für eben diesen Raum. Angenehmerweise verwenden die gängigen 3D-Grafiksysteme hierfür kartesische R-Koordinatensysteme wie das in Bild 25 beipielhaft gezeigte [1], deren Handhabung Programmierern wie Anwendern von Simulationswerkzeugen wegen ihrer gewohnheitsmäßigen Verwendung sicherlich am leichtesten fällt.

Die Lage eines Punktes ist darin durch seinen Ortsvektor *P [x, y, z]* vollständig beschrieben. Die Beschreibung komplexerer Objekte erfordert dagegen einen größeren Aufwand.

11.1.1 Struktur

Zunächst ist festzustellen, daß komplexe Objekte sinnvollerweise in Bezug auf ein eigenes lokales Koordinatensystem definiert werden, da sie typischerweise wiederum aus Teilobjekten bestehen, deren Lage innerhalb des Objekts zweckmäßig relativ zu einem Referenzpunkt angegeben wird, der dadurch zum Ursprung des lokalen Koordinatensystems wird. Im übrigen

[1] vgl. Watt: *3D Computer Graphics*; S. 2

existieren derartige Objekte in einer Szene (bzw. in einem Simulationsmodell) häufig mehrfach, so daß ihre Definition in einem eigenen Bezugssystem eine sinnvolle Möglichkeit des effizienten Umgangs mit dieser Situation darstellt [1].

Solche komplexen Objekte sind Ausprägungen der Abstraktion "Geometrieobjekt" und werden im hier dargestellten Konzept als Instanzen der Klasse *GeoObject* abgebildet. Die vollständige Beschreibung der Lage und Gestalt eines *GeoObjects* erfordert also einerseits die Aufzählung aller enthaltenen Teilobjekte und andererseits Angaben über die Lage des Objektkoordinatensystems im übergeordneten Koordinatensystem [2].

Letzteres leistet eine Kombination aus drei Vektoren, die die Position des Ursprungs sowie die Richtung der x- und der z-Achse des lokalen im übergeordneten Koordinatensystem definieren [3]. Solche als Lageinformation verwendete Vektorkombinationen werden als Instanzen der Klasse *Location* abgebildet, von denen *GeoObjects* daher je eine enthalten.

Die in einem *GeoObject* enthaltenen Teilobjekte lassen sich in zwei Gruppen einteilen. Die eine Gruppe umfaßt diejenigen Teile, deren (Aussehen und) Lage relativ zu dem enthaltenden *GeoObject* unveränderlich ist [4]. Hierzu gehören insbesondere Konturelemente, also beispielsweise alle Teile des Gehäuses einer (als *GeoObject* abgebildeten) Maschine. Solche Elemente werden als Instanzen der Klasse *Pattern* abgebildet. *GeoObjects* enthalten also eine Liste von *Pattern*.

Aus der hier zugrundeliegenden eher logischen Sicht auf *Pattern* ist nur von Bedeutung, daß sie zum einen Informationen über ihre Geometrie (d.h. ihren Aufbau aus geometrischen Primitiven) sowie Darstellungsattribute (Farbe, etc.) enthalten und zum anderen eine Hülle besitzen, die wie diejenige von *GeoObjects* definiert ist (s.u.) [5].

---

[1] vgl. Watt: *3D Computer Graphics*; S. 2

[2] Dies kann sowohl das globale Koordinatensystem wie auch das (lokale) Koordinantensystem eines anderen Objekts sein, in dem das beschriebene Objekt eingeordnet ist.

[3] Diese Lösung wurde der alternativen Definition der Achsenrichtungen durch Winkelangaben vorgezogen, da sich die im folgenden vorgestellten Operationen auf dieser Basis einheitlicher implementieren lassen.

[4] Diese Unveränderlichkeit gilt genau genommen nur während der Durchführung eines Simulationsexperiments. Beim Modellentwurf dagegen muß vollständiges Editieren aller Objekte selbstverständlich möglich sein, um ihre Eigenschaften ändern und Objekte löschen oder neue erzeugen zu können.

[5] Eine ausführlichere Darstellung gibt Bernhard: *Entwicklung einer Geometrie- und Animationsbibliothek ...*

Pattern werden in allen gleichartigen *GeoObjects* wiederverwendet. Das heißt, daß zwei gleiche Maschinen dasselbe (Gehäuse-) *Pattern* benutzen, wodurch ausgedrückt wird, daß die beiden Maschinen distinkte Objekte sind, während sich ihre Gehäuse gleichen. *Pattern* sind also (wie der Name nahelegt) Muster. *GeoObjects* enthalten dementsprechend die zu ihnen gehörigen *Pattern* nicht direkt (als Objekte), sondern nur Referenzen auf diese.

Aus dieser Struktur ergeben sich zwei Konsequenzen. Zum einen definieren *Pattern* kein eigenes (lokales) Koordinatensystem [1], zum anderen (und hier liegt der Vorteil dieses Aufbaus) können die *Pattern* für die Anzeige in Computergrafiksystemen vorbereitet (z.B. in OpenGL als Display-Liste vorkompiliert) werden, so daß jede Verwendung nur noch die Referenzierung der vorbereiteten Daten erfordert und so die mehrfache Verarbeitung der enthaltenen Grafikinformation erspart [2].

Die zweite Gruppe in *GeoObjects* enthaltener Teilobjekte umfaßt diejenigen Elemente, deren Lage sich relativ zu dem enthaltenden *GeoObject* verändern kann, wie beispielsweise die Gabel eines Staplers, oder die sogar während der Beobachtung (d.h. hier während der Durchführung eines Simulationslaufs) in den durch das enthaltende *GeoObject* gebildeten Bezugsraum eintreten oder diesen verlassen können, wie z.B.ein von einem Stapler transportiertes Teil. Diese Elemente werden wiederum als Instanzen der Klasse *GeoObject* abgebildet.

Die Eigenschaft eines *GeoObjects*, andere *GeoObject*s enthalten zu können wird dadurch ausgedrückt, daß jedes *GeoObject* eine Instanz der Klasse *Parent* ist. Hier bedeutet dies, daß die Klasse *GeoObject* von der Klasse *Parent* erbt. *Parents* enthalten im wesentlichen eine Liste von *GeoObjects*. Andersherum enthalten *GeoObjects* einen Zeiger auf das *GeoObject* (bzw. *Parent*-Objekt), in dem sie enthalten sind.

Dies erlaubt den Aufbau hierarchischer Objektstrukturen. Ihr wesentlicher Nutzen besteht darin, daß die Bewegung eines *GeoObjects*, wie beispielsweise des angeführten Staplers die (logische und sichtbare) Mitbewegung aller zum jeweiligen Zeitpunkt enthaltenen Teilobjekte, also beispielsweise der Gabel des Staplers und eventuell darauf befindlicher Teile impliziert, ohne daß das benutzende Programm hierfür Aufwendungen machen muß, die über das Ein- bzw. Ausordnen der jeweiligen *GeoObjects* ineinander hinausgehen.

Schließlich hat jedes *GeoObject* noch ein Hülle. Dies ist eine Kombination zweier Vektoren, die die gegenüberliegenden Eckpunkte eines Quaders angeben, der mathematisch als der

---

[1] Dies würde nämlich die Angabe von Lageinformationen (d.h. einer *Location*) für jede Verwendung eines *Patterns* in einem *GeoObject* und damit eine deutlich komplexere Objektstruktur erfordern.

[2] vgl. Bernhard: *Entwicklung einer Geometrie- und Animationsbibliothek* ...; S. 72 ff.

kleinste mögliche Quader definiert ist, der alle im jeweiligen *GeoObject* enthaltenen Teilobjekte (seien es *Pattern* oder wiederum *GeoObjects*) einschließt und dessen Kanten parallel zu den Grundachsen des Koordinatensystems des *GeoObjects* sind. Solche Objekthüllen sind als Instanzen der Klasse *WrapBox* abgebildet, von denen folglich jedes *GeoObject* eine enthält.

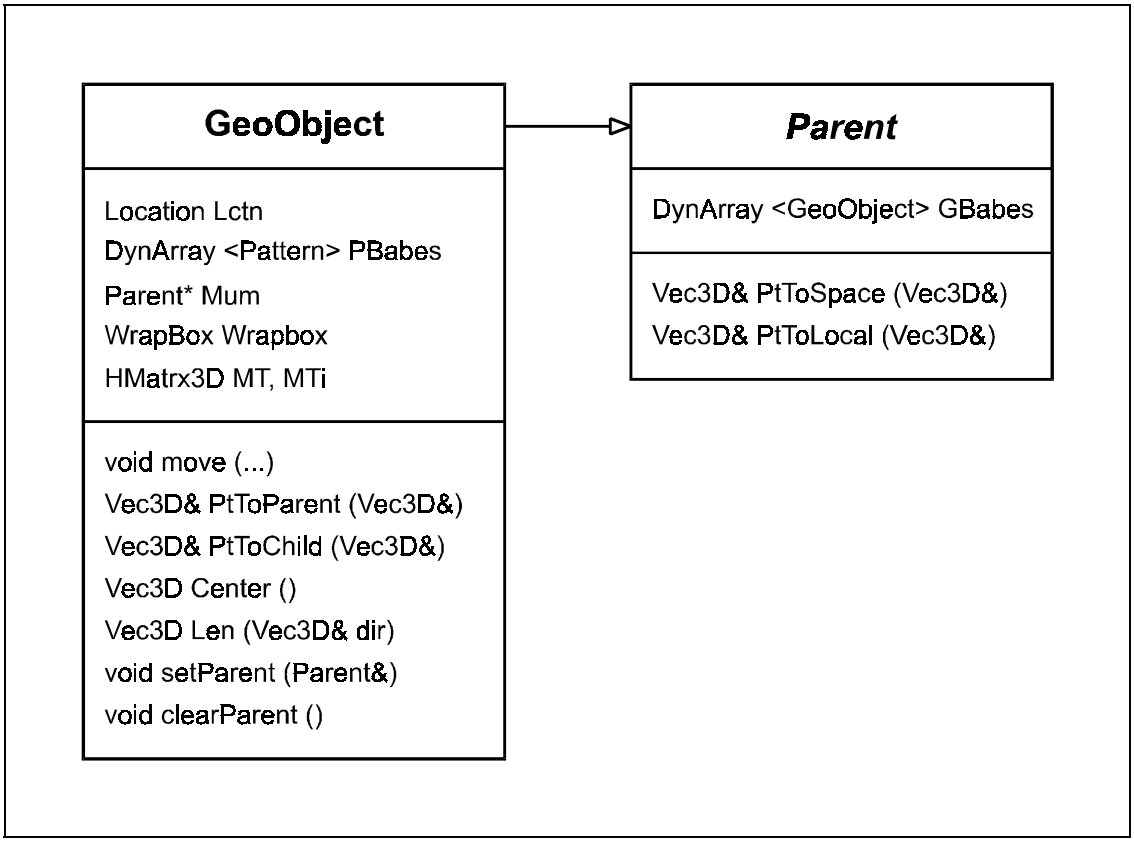

Bild 25: Struktur von *GeoObjects*

Bild 25 zeigt die softwaretechnische Struktur von *GeoObjects* mit ihren wesentlichen Eigenschaften und Operationen, auf die im folgenden eingegangen wird.

11.1.2 Transformationen

Die beim Entwurf eines Simulationswerkzeugs für Materialflußsysteme zu berücksichtigenden, in Animationen sichtbaren Bewegungsoperationen auf Modellelementen bzw. *GeoObjects* als deren grafische Repräsentationen sind Translation und Rotation. Sie bilden mit der Skalierung und der (im hier diskutierten Zusammenhang nicht relevanten) Scherung die Menge der dreidimensionalen affinen Transformationen [1].

[1] Watt: *3D Computer Graphics*; S. 2

Die neue Lage eines Punktes *P* als Resultat seiner Translation um einen Vektor *T* ist die Summe seines (ursprünglichen) Ortsvektors und des Verschiebevektors.

$$\boldsymbol{P'} = \boldsymbol{P} + \boldsymbol{T} \tag{2}$$

Die neue Lage eines Punktes *P* als Resultat seiner Rotation um eine beliebige Achse und einen beliebigen Winkel ist das Produkt seines (ursprünglichen) Ortsvektors und einer Rotationsmatrix *R*.

$$\boldsymbol{P'} = \boldsymbol{P}\,\boldsymbol{R} \tag{3}$$

Ein neuer Punkt *P'* als Ergebnis der Skalierung eines Vektors *P* ist das Produkt aus *P* und einer Skalierungsmatrix *S*.

$$\boldsymbol{P'} = \boldsymbol{P}\,\boldsymbol{S} \tag{4}$$

Um Translation, Rotation und Skalierung einheitlich und in Kombination behandeln zu können, werden, wie in Computergrafiksystemen üblich, homogene Koordinaten benutzt [1]. Dieser Übergang führt (mathematisch) eine weitere Dimension ein. Ein Vektor *V [x, y, z]* wird dabei als *V [x, y, z, 1]* behandelt [2]. Translation, Rotation und Skalierung eines Vektors *P* lassen sich dann einheitlich durch Multiplikation des (homogenen) Vektors *P* mit einer (ebenso homogenen) Transformationsmatrix *M* der Dimension 4 x 4 berechnen.

$$\boldsymbol{P'}\,[\boldsymbol{x'},\ \boldsymbol{y'},\ \boldsymbol{z'},\ \mathbf{1}] = \boldsymbol{P}\,[\boldsymbol{x},\ \boldsymbol{y},\ \boldsymbol{z},\ \mathbf{1}]\,\boldsymbol{M} \tag{5}$$

---

[1] Watt: *3D Computer Graphics*; S. 3

[2] In diesem Text werden Vektoren als Zeilenvektoren dargestellt. M.E. erhöht dies die Übersichtlichkeit, da eine entsprechend (12) ermittelte Gesamttransformationsmatrix M_{12} die Einzeltransformationen (erwartungskonform) so zusammenfaßt, als ob sie in der Reihenfolge der Notation (von links nach rechts) nacheinander ausgeführt würden. Die Tatsache, daß die *XMath*-Bibliothek Vektoren entsprechend der mathematischen Konvention als Spaltenvektoren abbildet, ist für Benutzer nur bei direktem Kontakt mit den Matrizen der *XMath*-Bibliothek von Bedeutung und bleibt bei der (ausschließlichen) Nutzung des hier vorgestellten Visualisierungskonzepts völlig transparent.

M ist dabei entweder eine Rotationsmatrix *R* der Form

$$\boldsymbol{R} = \begin{vmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \qquad (6),$$

eine Skalierungsmatrix *S* der Form

$$\boldsymbol{S} = \begin{vmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \qquad (7)$$

oder eine Translationsmatrix *T* der Form

$$\boldsymbol{T} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{vmatrix} \qquad (8)$$

worin die t_i die Komponenten des Verschiebevektors sind.

Die Verwendung homogener Koordinaten erlaubt die Zusammenfassung mehrerer einzelner Transformationen zu einer Gesamttransformation. Wenn beispielsweise

$$\boldsymbol{P'}\ [x',\ y',\ z',\ 1] = \boldsymbol{P}\ [x,\ y,\ z,\ 1]\ \boldsymbol{M_1} \qquad (9)$$

und

$$P'' [x'', y'', z'', 1] = P' [x', y', z', 1] M_2 \quad (10)$$

gelten, dann können die Transformationsmatrizen M_1 und M_2 entsprechend (11) zu einer Matrix M_{12} zusammengefaßt werden, wobei dann auch (12) gilt [1].

$$M_{12} = M_1 M_2 \quad (11)$$

$$P'' [x'', y'', z'', 1] = P [x, y, z, 1] M_{12} \quad (12)$$

Diese Konkatenierungsmöglichkeit wird im hier vorgestellten Visualisierungskonzept ausgenutzt. Jedes *GeoObject* enthält eine Matrix, die durch multiplikative Aneinanderreihung (entsprechend (12)) der Matrizen aller Transformationen errechnet wird, die das betreffende Objekt im Laufe seiner Existenz erfahren hat. Diese Matrix kann für die Anzeige des *GeoObjects* in Computergrafiksystemen wie OpenGL direkt verwendet werden. Sie reduziert den Aufwand bei der Neuanzeige des Objekts auf eine einzige Operation, die die Ausrichtung der Grundachsen des Objektkoordinatensystems und seine Positionierung im übergeordneten Koordinatensystem zusammenfaßt.

Die Klasse *GeoObject* stellt eine ganze Reihe von Operationen zur Verfügung, mit deren Hilfe Instanzen der Klasse verschoben und rotiert werden können [2]. Weitere Operationen erlauben die (absolute) Neufestlegung der Position, der x- und der z-Achse sowie der gesamten Lageinformation. Diese Operationen sind in der Regel doppelt vorhanden, wobei sich die beiden Versionen jeweils darin unterscheiden, daß die anzugebenden Parameter (Verschiebevektoren bzw. Rotationsachsen) einmal als auf das lokale Koordinatensystem des *GeoObjects* selbst bezogen und einmal als auf das übergeordnete Koordinatensystem in das das *GeoObject* eingeordnet ist bezogen interpretiert werden.

---

[1] Dabei ist allerdings zu beachten, daß nur Translationen (untereinander) kommutativ sind, Rotationen und gemischte Transformationen dagegen nicht. Im allgemeinen gilt also $M_1 M_2 \neq M_2 M_1$. Praktisch bedeutet dies, daß die Reihenfolge von Transformationen für das Ergebnis bedeutsam ist.

[2] In Bild 25 ist stellvertretend für alle diese Operationen nur die Funktion *move ()* aufgeführt.

11.1.3 Weitere Operationen

Neben der erwähnten Gesamttransformationsmatrix ist in jedem *GeoObject* auch deren Inverse gespeichert. Beide bilden zusammen die Grundlage für die effiziente Implementierung von Operationen zur Transformation von Vektoren (Ortsvektoren und Richtungen) und Lageinformationen (*Location*-Objekten) zwischen verschiedenen Koordinatensystemen einer *GeoObject*-Hierarchie, die die Klasse bereitstellt. Die zwei vorhandenen Gruppen solcher Umrechnungsoperationen wandeln zum einen zwischen dem lokalen und dem übergeordneten Koordinatensystem und zum anderen zwischen dem lokalen und dem Weltkoordinatensystem [1] um.

Auf Basis der enthaltenen Hülle sind Funktionen implementiert, die den Mittelpunkt des *GeoObjects* und seine Länge in eine anzugebende Richtung ermitteln. Schließlich umfaßt die Klasse noch Operationen, die die Hierarchie von (Strukturen aus) *GeoObjects* modifizieren. Diese Operationen erlauben das Einfügen und Entfernen von *GeoObjects* in bzw. aus anderen *GeoObjects* oder Räumen.

11.2 Räume

Im hier vorgestellten Konzept sind Räume Aufenthaltsorte beliebig vieler *GeoObjects* mit (implizitem) dreidimensionalem kartesischen R-Koordinatensystem. Aus der Sicht der direkt oder indirekt (als Teilobjekte anderer Objekte) enthaltenen *GeoObjects* ist das Koordinatensystem des Raumes das globale oder Weltkoordinatensystem.

Räume werden als Instanzen der Klasse *Space* abgebildet und sind damit explizit zu handhabende Objekte. Die Fähigkeit, *GeoObjects* enthalten bzw. als deren übergeordnete Koordinatensysteme verwendet werden zu können, erhalten *Spaces* ebenso wie *GeoObjects* durch Ableitung aus der Klasse *Parent* [2].

Die zweite wesentliche Eigenschaft von *Spaces* ist, daß jedem von ihnen eine beliebige Anzahl von Ansichten (s.u.) zugeordnet sein kann, die wiederum in einer Liste verwaltet werden.

---

[1] Unter Weltkoordinatensystem ist in diesem Konzept immer das Koordinatensystem des Raums zu verstehen, in den ein (selbst hierarchisch aufgebautes) *GeoObject* eingeordnet ist.

[2] vgl. Kapitel 11.1

Da es letztlich die Ansichten sind, die Objekte anzeigen und damit sichtbar machen, erfordert die Visualisierung von Objekten in diesem Konzept also insgesamt die Modellierung der Objekte als *GeoObjects* und ihre Einordnung in einen *Space*, dem wenigstens eine Ansicht zugeordnet ist.

Die Modellierung von Räumen als explizite Objekte hat dabei zur Konsequenz, daß jede Anwendung sinnvollerweise mindestens ein *Space*-Objekt erzeugen muß. Im Falle eines Simulationswerkzeugs repräsentiert dieser *Space* das betrachtete Materialflußsystem als geometrischen Raum, in dem (als *GeoObjects* abgebildete) Fördertechnikelemente zum Gesamtlayout angeordnet sind [1].

11.3 Ansichten

Ansichten sind Komponenten, die die in einem Raum enthaltenen Objekte sichtbar machen, also visualisieren. Aus konzeptueller Sicht umfassen sie jeweils eine vollständige Verarbeitungskette von einer (in einen Raum gerichteten) Kamera bis zur Ausgabe des von dieser erfaßten Bildes auf irgendein Medium.

Das naheliegendste Ausgabemedium einer Ansicht ist ein Bildschirmfenster, in dem durch Nutzung eines Grafiksystems wie z.B. OpenGL ein (dynamisches) Abbild der in dem beobachteten Raum von der Kamera erfaßten Objekte angezeigt wird. Andere mögliche Ausgabemedien sind Dateien (Trace-Files), in die Bewegungsprotokolle aufgezeichnet, oder Netzwerkverbindungen über die Objekt- und Bewegungsdaten an entfernte Rechner zur dortigen Visualisierung übertragen werden können (“Remote-Ansicht”).

Während also auf der Ausgabeseite Vielfalt sinnvoll und wünschenswert ist, muß auf der Eingabe- bzw. Kameraseite eine einheitliche Schnittstelle die Grundlage dafür bieten, daß der beobachtete Raum mit jeder ihm zugeordneten Ansicht unabhängig von ihrem Ausgabemedium in gleicher Weise interagieren kann. Diese Anforderung kann dadurch erfüllt werden, daß Ansichten als Instanzen von Klassen abgebildet werden, die im wesentlichen die Bedienung des jeweiligen Ausgabemediums kapseln. Beispiele solcher Klassen könnten *GLView* (OpenGL-basierte Ausgabe in ein Bildschirmfenster) und *TraceView* (Schreiben eines Trace-Files) sein.

---

[1] Das Erzeugen mehrerer *Spaces* ist ohne weiteres möglich. Dies könnte in Simulationswerkzeugen genutzt werden, um Funktionalitäten wie die Anzeige von Betriebsdaten oder das Editieren von Arbeitsplänen grafisch zu unterstützen.

Die Ableitung aller dieser Klassen aus der (gemeinsamen) abstrakten Basisklasse *View* sorgt dabei für die einheitliche Schnittstelle zum beobachteten Raum bzw. zu dem diesen repräsentierenden *Space*-Objekt. Es kann alle Ansichten einheitlich als *View*-Objekte behandeln.

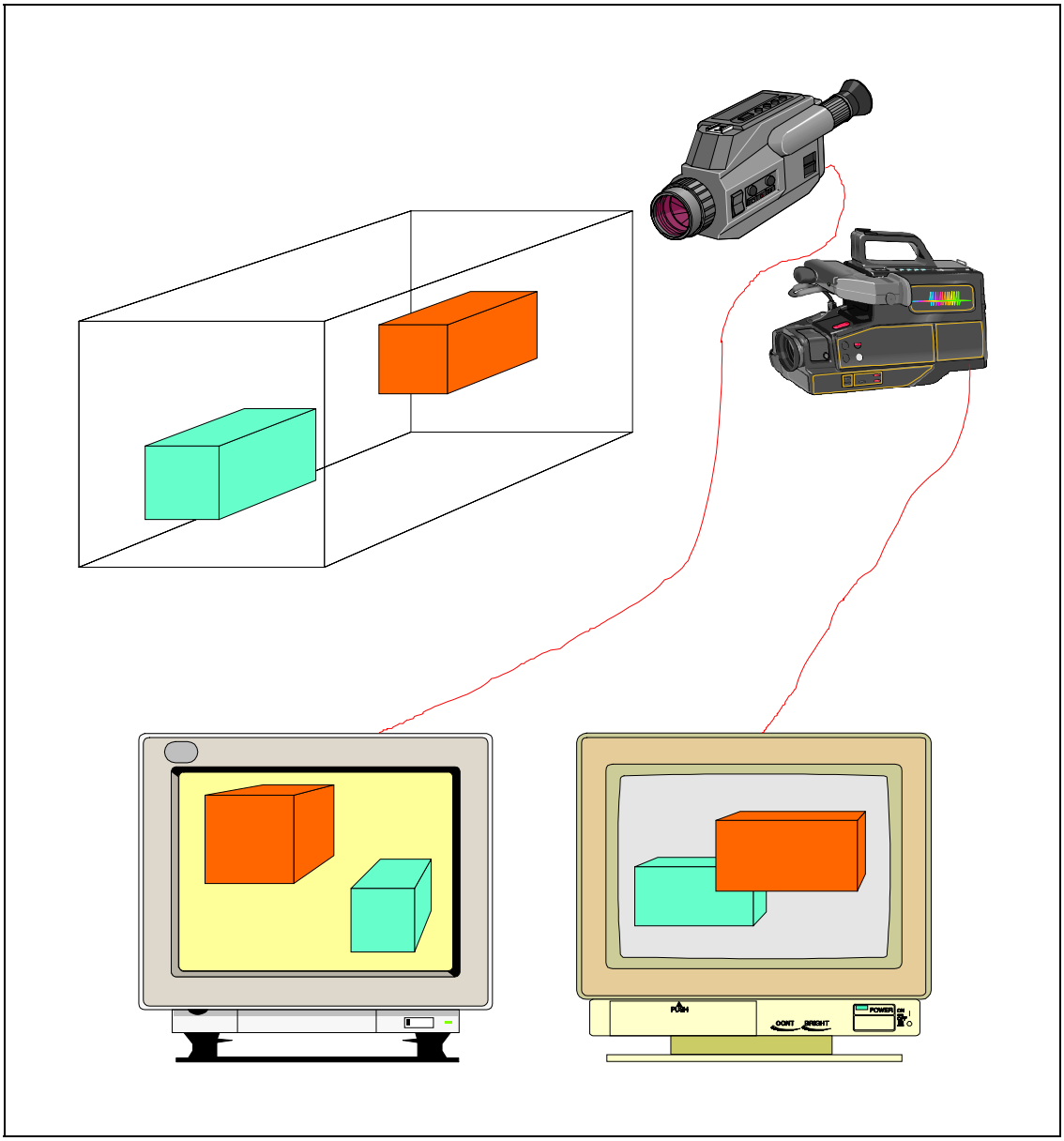

Bild 26: Zwei Ansichten in einen zwei Geometrieobjekte enthaltenden Raum

11.4 Zusammenfassung

Das vorgestellte Konzept für die geometrische Objektrepräsentation und Visualisierung baut auf den drei Abstraktionen “Geometrieobjekt”, “Raum” und “Ansicht” auf.

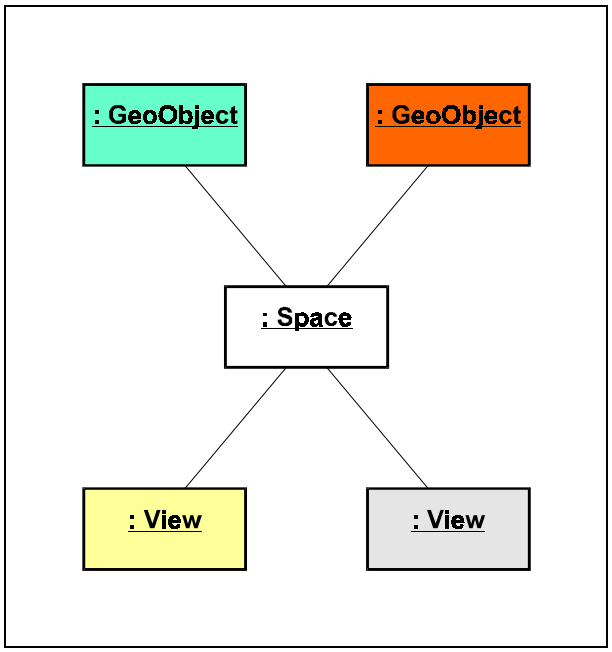

Bild 27: Objekbeziehungen in Bild 26

Geometrieobjekte bilden die dreidimensionale Geometrie beliebiger Objekte wie beispielsweise der Elemente von Materialflußsystemen vollständig ab und definieren eigene (lokale) Koordinatensysteme. Sie können hierarchisch aufgebaut und in Räume eingeordnet sein. Räume sind globale Koordinatensysteme. Jedem Raum kann eine beliebige Anzahl Ansichten zugeordnet werden, die (typischerweise auf der Basis von Computergrafiksystemen) die in dem Raum enthaltenen (Geometrie-) Objekte darstellen bzw. visualisieren.

Bild 26 zeigt eine beispielhafte Konstellation: Zwei Ansichten sind an einen Raum gebunden und zeigen jeweils die beiden in diesem enthaltenen Geometrieobjekte. Bild 27 zeigt die in der erzeugenden Anwendung entstehenden Objektbeziehungen.

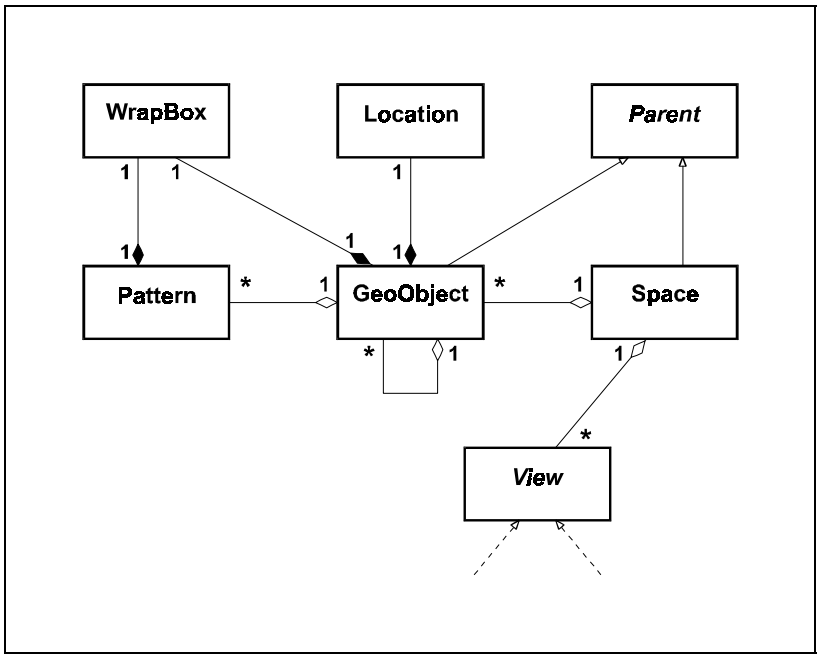

Bild 28: Klassenbeziehungen im Visualisierungskonzept

Eine Übersicht über die (wichtigsten) Klassen des dargestellten Konzepts und ihre Beziehungen untereinander gibt Bild 28. Das Konzept versetzt Anwendungen in die Lage die in Kapitel 8.1 abgeleiteten Forderungen nach mehreren unabhängigen und abschaltbaren online-Animationen zu erfüllen.

12 Zeitmechanismus

In Kapitel 7.5.3.3 wurde ausgeführt, daß jedes Simulationswerkzeug einen Zeitmechanismus enthält, dessen Aufgabe die Steuerung der chronologischen Abarbeitung der im betrachteten System ablaufenden Prozesse ist. Als mögliche Ausprägungen wurden takt-, ereignis-, prozeß- und transaktionsorientierte Zeitmechanismen genannt.

In den nachfolgenden Unterabschnitten werden die Wechselwirkungen zwischen den verschiedenen Typen von Zeitmechanismen und der in Kapitel 8.1.4 erhobenen Forderung nach Abbildbarkeit beliebiger Steuerungsmechanismen in Simulationsmodellen behandelt [1]. Zum Schluß dieses Kapitels wird der im Rahmen dieser Arbeit konzipierte Zeitmechanismus vorgestellt, dessen Entwurf auf den zuvor dargestellten Überlegungen basiert.

12.1 Ereignisorientierte Zeitmechanismen

Bei Verwendung ereignisorientierter Zeitmechanismen werden die im System ablaufenden parallelen Vorgänge als eine Folge von Ereignissen modelliert. Zu jedem Ereignis gehören

---

[1] Taktgetriebene Zeitmechanismen werden hier nicht weiter betrachtet, da sie (je nach technischer Realisierung) als Variante einer der drei übrigen Ausprägungen angesehen werden können. Der Vollständigkeit halber sei daher erwähnt, daß bei diesen die Modellzeit in konstanten Schritten fortgeschaltet wird. Dieses Konzept ist eng verwandt mit den Prinzipien der (quasi-) kontinuierlichen Simulation (vgl. *VDI-Richtlinie 3633, Blatt 1*; S. 6)

im wesentlichen ein Eintrittszeitpunkt und eine Programmfunktion (d.h. ein Algorithmus), der die durch das Ereignis ausgelösten Systemveränderungen beschreibt. Sie wird beim Eintreten des Ereignisses vollständig abgearbeitet und ist also nicht unterbrechbar.

Zeitverbrauchende Aktivitäten wie die Bewegung einer Palette auf einem Förderer müssen daher auf mehrere Ereignisse verteilt werden. Typischerweise wird je ein Anfangs- und ein Endeereignis vorgesehen. Ereignisse führen oft zu Folgeereignissen, die aus den Funktionen eingeplant werden. Beispielsweise wird das Anfangsereignis der Palettenbewegung das zugehörige Endeereignis für den Zeitpunkt nach Ablauf der Bewegungszeit einplanen.

Der Zeitmechanismus verwaltet alle eingeplanten Ereignisse in einem nach aufsteigenden Eintrittszeitpunkten geordneten "Terminkalender" [1]. Er veranlaßt immer wieder die Abarbeitung der zu dem Ereignis mit dem frühesten Eintrittszeitpunkt gehörenden Funktion. Dieser Zyklus (und damit das Experiment) wird beendet, wenn entweder keine weiteren Ereignisse eingeplant sind oder eine vorgegebene Zeitschranke erreicht ist [2].

Die Wechselwirkungen zwischen ereignisorientierten Zeitmechanismen und Steuerungsalgorithmen soll der in Bild 29 dargestellte beispielhafte Ausschnitt aus einem Materialflußsystem veranschaulichen. Darin werden auf Paletten (grau) befindliche Teile (gelb) über einen Linearförderer (grün) zu einem Übergabepunkt bewegt. Dort werden sie von der Palette abgenommen und an Elektrohängebahnfahrzeuge (EHB, magenta) angehängt, auf denen sie weitergefördert werden. Jede Übergabe erfordert also das Vorhandensein einer Palette (mit Teil) und einer EHB im Übergabepunkt. Weiter sei unterstellt, daß die Paletten- und EHB-Ströme Schwankungen unterliegen, so daß nicht vorhersagbar ist, wann und in welcher Reihenfolge Palette und EHB im Übergabepunkt eintreffen.

Offensichtlich erfordert die Steuerung (und damit auch die Modellierung) dieses Anlagenausschnitts die Synchronisation von EHB- und Palettenstrom im Übergabepunkt. Bild 30 zeigt für die Steuerung in Frage kommende beispielhafte Algorithmen, die auf den Übergabepunkten (hier als PalettenPunkt bzw. EHBPunkt bezeichnet) für jede Palette bzw. EHB einmal abgearbeitet werden (müssen) [3].

---

[1] Die Frage der Reihenfolge "gleichzeitiger" Ereignisse wird an dieser Stelle bewußt vernachlässigt.

[2] Eine klassische Implementierung eines ereignisorientierten Zeitmechanismus ist die Sprache Simscript (vgl. Kiviat et al.: *The Simscript II Programming Language*). Auch der am Fachgebiet Produktionssysteme entwickelte Simulator SIMFLEX/2 (s. Anhang S. 261) arbeitet ereignisorientiert.

[3] In den Algorithmen unterbricht die Anweisung *warte ()* die Ausführung der aufrufenden Funktion, so daß die nachfolgenden Anweisungen (zunächst) nicht mehr ausgeführt werden. Die Anweisung *aktivieren (...)* veranlaßt (als Gegenstück dazu) die Fortsetzung der Ausführung einer auf dem angesprochenen Übergabepunkt eventuell unterbrochenen Funktion als nächstes Ereignis.

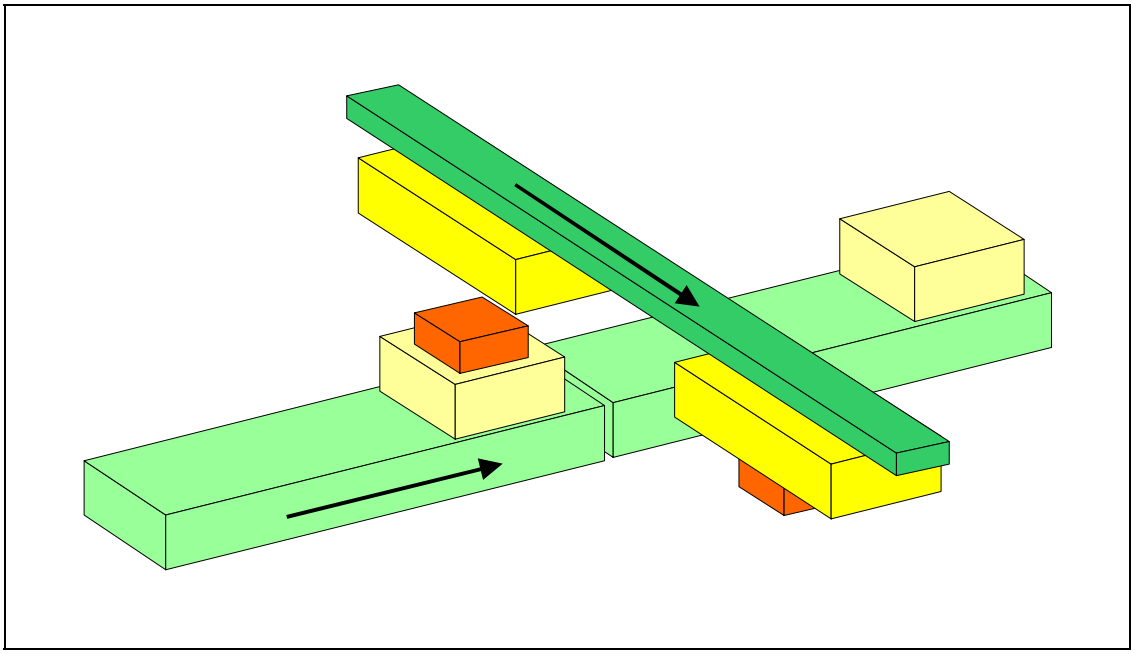

Bild 29: Anlagenausschnitt mit Teileübergabe von Palette an EHB

Bei der Verwendung ereignisorientierter Zeitmechanismen bereitet die softwaretechnische Umsetzung dieser Steuerungsalgorithmen einige Schwierigkeiten. Der konkrete Problempunkt ist die notwendige Unterbrechung (und spätere Fortsetzung) der Abarbeitung in der Anweisung *warte ()*, durch die der enthaltende Algorithmus nicht wie oben für Ereignisfunktionen gefordert, zu einem Zeitpunkt vollständig abgearbeitet werden kann. Offensichtlich birgt jede Anweisung die einen Zeitverbrauch impliziert diese Problematik.

```
int Pal_ok;

void Palette () {
  Pal_ok = 1;
  aktivieren (EHBPunkt);
  while (EHB_ok == 0)  warte ();
  Pal_ok = 0;
  abtransportieren ();
}
```

```
int EHB_ok;

void EHB () {
  EHB_ok = 1;
  aktivieren (PalettenPunkt);
  while (Pal_ok == 0)  warte ();
  EHB_ok = 0;
  abtransportieren ();
}
```

Bild 30: Steuerungsalgorithmen zur Teileübergabe von Palette an EHB

Da die Unterbrechung von Algorithmen also mit der “reinen Lehre” ereignisorientierter Zeitmechanismen nicht vereinbar ist, stellt sich die Frage, ob auf zeitverbrauchende Anweisungen verzichtet werden kann. Die in Bild 31 dargestellten, entsprechend angepaßten Steuerungsalgorithmen des Beispiels zeigen, daß dies technisch durchaus möglich ist. Dabei

nehmen allerdings Anschaulichkeit und Übersichtlichkeit der Algorithmen (im Vergleich mit denen aus Bild 30) deutlich ab [1]. Es ist sicher vorstellbar, daß bei der Umsetzung noch komplexerer (Teil-) Steuerungen von Materialflußsystemen Anschaulichkeit und Übersichtlichkeit der gefundenen Algorithmen immer mehr verlorengehen, wenn in dieser Weise gearbeitet wird (bzw. werden muß), bis die gefundenen Lösungen schließlich (sogar für ihre Erschaffer) kaum noch durchschaubar sind [2].

```
int Pal_ok;

void Palette () {
  Pal_ok = 1;
  if (EHB_ok == 1) {
    EHBEnde ();
    PaletteEnde ();
  }
}
void PaletteEnde () {
  Pal_ok = 0;
  abtransportieren ();
}
```

```
int EHB_ok;

void EHB () {
  EHB_ok = 1;
  if (Pal_ok == 1) {
    PaletteEnde ();
    EHBEnde ();
  }
}
void EHBEnde () {
  EHB_ok = 0;
  abtransportieren ();
}
```

Bild 31: Steuerungsalgorithmen zur Teileübergabe von Palette an EHB (2. Version)

Es ist also unbedingt wünschenswert, bei der Formulierung von Steuerungsalgorithmen in einem Simulationswerkzeug auf zeitverbrauchende Anweisungen zurückgreifen zu können, um die Umsetzung auch komplexer Steuerungen in übersichtliche und anschauliche Algorithmen zu ermöglichen [3].

---

[1] Auf den ersten Blick ist die reduzierte Anschaulichkeit vor allem die Folge davon, daß die Steuerungsalgorithmen hier auf nun vier (statt zwei) Funktionen verteilt wurden. Der eigentliche (und eben nicht behebbare) Mangel dieses Ansatzes ist aber, daß das (unumgänglich notwendige) wechselseitige Warten der Übergabepunkte aufeinander in den Algorithmen nicht mehr explizit (als Anweisung), sondern nunmehr implizit (und damit versteckt) enthalten ist.

[2] Die geringere Übersichtlichkeit führt tendenziell zu einer höheren Fehlerquote bei der Implementierung und diese weiter zu erhöhtem Zeitbedarf für die Modellprüfung bzw. -korrektur.

[3] Sicherlich ist es softwaretechnisch machbar, eine "Hülle" um den Kern eines ereignisorientierten Zeitmechanismus zu programmieren, auf den aufsetzend zeitverbrauchende Anweisungen realisiert werden können. Die konzeptuelle Kluft zwischen ihnen und den Prinzipien ereignisorientierter Zeitmechanismen bleibt aber bestehen.

12.2 Transaktionsorientierte Zeitmechanismen

Die zentrale Abtraktion in auf transaktionsorientierten Zeitmechanismen basierenden Simulationssystemen sind (eben) Transaktionen. Sie repräsentieren temporäre Objekte, die entstehen, für eine Zeit im betrachteten System verweilen, es schließlich wieder verlassen und dabei gelöscht werden. Beispiele für Transaktionen sind das System durchlaufende Teile oder Störungen, die die Verfügbarkeit von Komponenten unterbrechen.

Zu jeder Transaktion gehört ein Algorithmus, der die Aktivitäten der Transaktion im System beschreibt. Diese Algorithmen enthalten (u.a.) zeitverbrauchende Anweisungen, die bewirken, daß der aufrufende Algorithmus in der Anweisung unterbrochen und später an der Unterbrechungsstelle fortgesetzt wird. Der Zeitverbrauch einer solchen Anweisung kann entweder explizit angegeben sein (z.B. zur Nachbildung von Bearbeitungs- oder Störzeiten) oder sich implizit aus dem Zustand (eines Teils) des Systems ergeben (z.B. beim Versuch eines Teils, einen Förderer zu belegen, der gerade gestört ist).

Der Zeitmechanismus verwaltet alle vorhandenen Transaktionen in einem nach aufsteigenden Fortsetzungszeitpunkten geordneten “Terminkalender”. Er veranlaßt immer wieder die Fortsetzung der Abarbeitung des zu der Transaktion mit dem frühesten Fortsetzungszeitpunkt gehörenden Algorithmus [1]. Dieser Zyklus (und damit das Experiment) wird beendet, wenn entweder keine Transaktionen mehr vorhanden sind oder eine vorgegebene Zeitschranke erreicht ist [2].

Der Grundsatz, daß in transaktionsorientierten Zeitmechanismen die Dynamik des Systems ausschließlich durch Transaktionen beschrieben wird, ist Ausdruck der Annahme, daß alle Aktivitäten in einem System von temporären Systembestandteilen ausgeht. Sofern die explizite Modellierung von Speicherprogrammierbaren Steuerungen (SPS), Lager-verwaltungs- oder Steuerungsrechnern für Transportsysteme (z.B. FTS) erforderlich ist, müssen also spezielle (Dummy-) Transaktionen eingeführt werden, da die genannten Fälle Beispiele aktiver, dabei aber permanenter (also über den gesamten Betrachtungszeitraum existierender) Systembestandteile sind. Es existiert also wiederum eine konzeptuelle Kluft, diesmal zwischen den Prinzipien transaktionorientierter Zeitmechanismen und der Abbildung aktiver permanenter Systembestandteile.

---

[1] Die Behandlung blockierter Transaktionen wie z.B. des o.a. Teils, das eine gestörten Förderer zu belegen versuchte, wird hier nicht erörtert, da die Darstellung auf die fundamentalen Grundlagen der Mechanismen begrenzt bleiben soll.

[2] Die klassische Implementierung eines transaktionsorientierten Zeitmechanismus ist die Sprache GPSS (vgl. Gordon: *The Application of GPSS V to Discrete Systems Simulation*).

12.3 Prozeßorientierte Zeitmechanismen

Der Ansatz prozeßorientierter Zeitmechanismen ist, gerade am zuletzt angesprochenen Punkt, noch etwas anders. Prozesse, die hier die Aktivitätsträger sind, kennzeichnet, daß sie ein Eigenverhalten haben. Sowohl temporäre (Teile, etc.) als auch permanente Einheiten (Förderer, Steuerungen, etc.) können als Prozesse abgebildet werden.

Zu jedem Prozeß gehört wiederum ein Algorithmus, der sein (dynamisches) Verhalten im System beschreibt. Wie bei transaktionsorientierten Mechanismen können diese Algorithmen zeitverbrauchende Anweisungen enthalten, die bewirken, daß der aufrufende Algorithmus unterbrochen und später an der Unterbrechungsstelle fortgesetzt wird.

Der Zeitmechanismus verwaltet alle aktiven Prozesse in einem nach aufsteigenden Fortsetzungszeitpunkten geordneten "Terminkalender". Er veranlaßt immer wieder die Fortsetzung der Abarbeitung des Prozesses mit dem frühesten Fortsetzungszeitpunkt. Dieser Zyklus (und damit das Experiment) wird wiederum beendet, wenn entweder keine Prozesse mehr aktiv sind oder eine vorgegebene Zeitschranke erreicht ist [1].

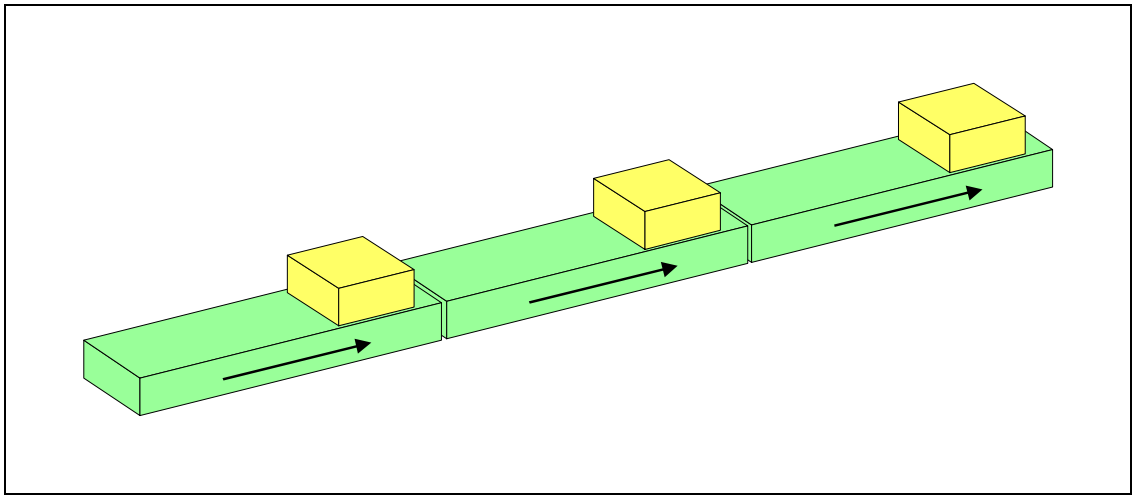

Bild 32: Beispielsystem mit Linearförderern

Prozeßorienierte Zeitmechanismen implizieren also keine Annahmen bzw. Festlegungen über den Charakter der aktiven Objekte und gestatten die Formulierung übersichtlicher Algorithmen zur Beschreibung der Prozeßdynamiken. Für die Implementierung von Simulationswerkzeugen,

[1] Die klassische Implementierung eines prozeßsorientierten Zeitmechanismus ist die Sprache Simula (vgl. Nygaard, et al.: *The Development of the Simula Languages*).

die die in Kapitel 8.1.4 genannten Anforderung der Abbildbarkeit beliebiger Steuerungsmechanismen scheinen sie also am geeignetsten. Es sind jedoch weitere Aspekte zu bedenken, die anhand eines neuen Beispiels behandelt werden sollen.

```
class Foerderer {
  Foerderer *Eingang, *Ausgang;
  Prozess *Wartend;
  int Frei;
  void foerdern (Teil* teil) {
    Wartend = AktuellerProzess ();
    if (Frei == 0)        warte ();
    Wartend = NULL;
    Frei = 0;
    warte (Aufnahmezeit);
    Eingang->Frei = 1;
    if (Eingang->Wartend != NULL)     Eingang->Wartend->schedule ();
    warte (Transportzeit);
  }
};
class Teil : public Prozess {
  Foerderer *Ort;
  void durchlaufen () {
    for (Ort = &Foerderer1; Ort != NULL; Ort = Ort->Ausgang)
      Ort->foerdern (this);
  }
};
```

Bild 33: Programmcode zum Linearfördererbeispiel

Bild 32 zeigt das betrachtete System, in dem Teile (grau) über eine Reihe hintereinandergeschalteter Linearförderer (grün) bewegt werden. Die Förderer seien untereinander so verkettet, daß ihre Ausgänge jeweils den in Förderrichtung nachfolgenden und ihre Eingänge jeweils den vorangehenden Förderer referenzieren. Der Eingang des ersten und der Ausgang des letzten Förderers seien nicht (d.h. auf "NULL") gesetzt. Es sei weiter angenommen, daß die Teile einen Förderer erst dann belegen dürfen, wenn das voranlaufende Teil diesen vollständig verlassen hat. Jedes Teil ist (auch) ein Prozeß, der die Funktion *durchlaufen ()* ausführt. In Bild 33 sind die wesentlichen Teile des zugehörigen Programmcodes dargestellt [1].

---

[1] In den Algorithmen unterbricht die Anweisung *warte ()* die Ausführung des aktuellen Prozesses für die angegebene Zeit bzw. beliebig lange, wenn keine Zeit angegeben wurde. Die Anweisung *schedule ()* veranlaßt die Fortsetzung des angesprochenen Prozesses.

Ein Mangel dieses Ansatzes besteht sicher darin, daß sich jeder Förderer mit vielen Details seines Eingangs befaßt. Insgesamt wichtiger ist aber m.E., daß die Zeit, die jedes Teil (wie verlangt) mit dem Warten darauf verbringt, daß das voranlaufende den zu belegenden Förderer vollständig verlassen hat, in den Algorithmen nicht explizit nachgebildet ist. Diese Bedingung wird hier (implizit) dadurch eingehalten, daß die Förderer später (zum richtigen Zeitpunkt) von ihrem Ausgang als frei markiert werden. Damit ist die Beschreibung der pro Teil auf einem Förderer stattfindenden Abläufe verteilt.

```
class Foerderer {
  Foerderer *Ausgang;
  Prozess *Wartend;
  int Frei;
  void foerdern (Teil* teil) {
    Wartend = AktuellerProzess ();
    while (Frei == 0)   warte ();
    Wartend = NULL;
    Frei = 0;
    warte (Aufnahmezeit);
    if (teil->Alt != NULL)      teil->Alt->schedule ();
    warte (Transportzeit);
    teil->Alt = AktuellerProzess ();
    if (Ausgang != NULL)      starteProzess (Ausgang, foerdern, teil);
    warte ();    // wartet bis das Teil den Förderer verlassen hat
    Frei = 1;
    if (Wartend != NULL)      Wartend->schedule ();
  }
};
class Teil {
  Prozess *Alt;
  void durchlaufen {
    starteProzess (&Foerderer1, foerdern, this);
  }
};
```

Bild 34: Programmcode zum Linearfördererbeispiel (2. Version)

Offensichtlich ließe sich die Übersichtlichkeit steigern, wenn alle Abläufe in der Reihenfolge, in der sie auch stattfinden, vollständig in der Funktion *fördern ()* des jeweils durchlaufenen Förderers beschrieben wären. Bild 34 zeigt einen entsprechend überarbeiteten Programmcode, der die gewünschte Verbesserung bringt. Allerdings sind nun für jedes Teil während des Übergangs von einem Förderer zum nächsten sogar zwei Prozesse aktiv. Die Modellierungsphilosophie des verwendeten Zeitmechanismus muß diesen Ansatz natürlich angemessen unterstützen.

In der Sprache Simula [1] werden aktive Objekte wesentlich dadurch beschrieben, daß ihre Klasse aus der (vordefinierten) Klasse *process* abgeleitet wird. Zwischen den Klassen besteht damit eine Ererbungsbeziehung, die ausdrückt, daß jedes aktive Objekt (genau) ein *process*-Objekt ist. Andere prozeßorientierte Simulationsbibliotheken [2] arbeiten mit der gleichen Philosophie [3]. Der Ansatz, auch zwei (oder mehr) einem aktiven Objekt zugeordnete Prozesse zuzulassen, wird so nicht angemessen unterstützt.

12.4 Die Klasse *Activity*

Als Konsequenz aus den dargestellten Überlegungen wurde im Rahmen dieser Arbeit ein Zeitmechanismus entwickelt, der anstelle der Ererbungs- eine Aggregationsbeziehung zwischen aktiven Objekten und verhaltensbeschreibenden Algorithmen verwendet. Aktive Objekte kennzeichnet dabei, daß sie (beliebig viele) zugehörige Aktivitäten "haben", deren Ausführung sie kontrollieren. Jeder solche Aktivität ist eine Instanz der Klasse *Activity* und führt die (parallele) Abarbeitung eines Algorithmus (bzw. einer Funktion) durch. Dieser Ansatz entspricht im Grunde den Multithreading-Funktionalitäten moderner Betriebssysteme, allerdings mit dem Unterschied, daß diese die zur Verfügung stehende Rechen- bzw. Prozessorzeit zwischen den aktiven Threads auf der Basis eines Zeitscheibenalgorithmus verteilen (sog. preemptives Multithreading), während die Klasse *Activity* ein kooperatives Multithreading realisiert [4].

Sie verwaltet dazu (intern) einen "Terminkalender", in dem analog zu prozeßorientierten Zeitmechanismen alle aktiven Threads nach aufsteigenden Fortsetzungszeitpunkten geordnet eingetragen sind. Der so realisierte Zeitmechanismus veranlaßt immer wieder die Fortsetzung der Abarbeitung des Threads mit dem frühesten Fortsetzungszeitpunkt. Das Umschalten zwischen den Threads findet dabei dadurch statt, daß der gerade ausgeführte Thread die Kontrolle abgibt, indem er sich entweder explizit (durch Aufruf einer zeitverbrauchenden Funktion) oder implizit (z.B. durch Ausführung einer ihn blockierenden Funktion) von der ersten Position im "Terminkalender" entfernt.

---

[1] vgl. Nygaard et al.: *The Development of the Simula Languages*

[2] vgl. z.B. Ahrens et al.: *Objektorientierte Prozeßsimulation in C++*

[3] In vergleichbarer Weise (und mit gleichem Ergebnis) existiert auch in transaktionsorientierten Zeitmechanismen (z.B. in GPSS) eine 1-1-Beziehung zwischen je einem aktiven Objekt (also einer Transaktion) und einem zugehörigen verhaltensbeschreibenden Algorithmus (vgl. Kapitel 12.2).

[4] Die als *Activity*-Objekte abgebildeten Threads unterscheiden sich von den Threads der Betriebssysteme vor allem dadurch, daß sie in einer eigenen abgeschlossenen Umgebung arbeiten, die insgesamt in einem Betriebssystem-Thread läuft.

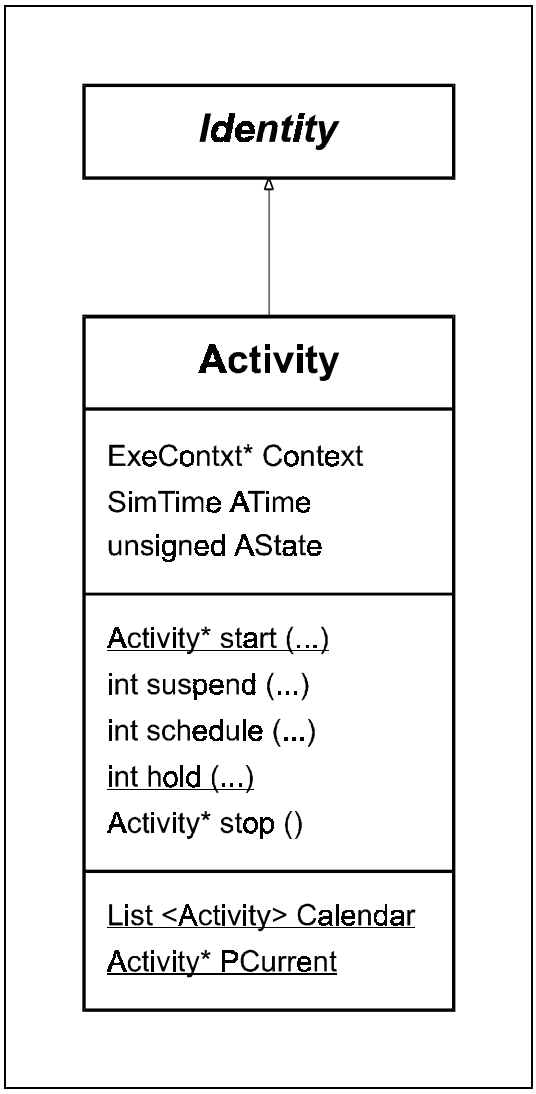

Bild 35: Struktur von *Activity*-Objekten

Das Vorgehen bei der Unterbrechung und späteren Fortsetzung von Threads orientiert sich an den entsprechenden Funktionalitäten von Betriebssystemen und an Vorbildern aus anderen Simulationsbibliotheken [1]. Der Ausführungszustand eines Threads ist auf den für die Implementierung von Simulationswerkzeugen in Frage kommenden Rechnerplattformen [2] durch den Aufbau des zugehörigen Stacks und den Inhalt der Prozessorregister vollständig beschrieben. Stack und Register sind also bei der Unterbrechung eines Threads zu sichern und zu seiner (späteren) Fortsetzung wiederherzustellen. Diese Funktionalität wird hier von der Klasse *ExeContxt* bereitgestellt. Jedes *Activity*-Objekt enthält daher (einen Zeiger auf) eine Instanz dieser Klasse. Für Anwendungen ist ihre Existenz allerdings vollständig transparent.

Für die Kontrolle des dynamischen Verhaltens von Threads stellt die Klasse *Activity* eine Reihe von Funktionen zur Verfügung, durch die Threads für eine anzugebende Zeit unterbrochen (*suspend (...)*), zu einem späteren Zeitpunkt oder relativ zu (vor bzw. nach) anderen Threads (wieder) eingeplant (*schedule (...)*), passiv gesetzt und damit (vorübergehend) ganz aus dem Terminkalender entfernt (*hold (...)*) oder beendet werden können (*stop ()*). Der Funktionsumfang entspricht damit dem von prozeßorientierten Zeitmechanismen für die Behandlung von Prozessen gebotenen [3]. Bild 35 zeigt die resultierende softwaretechnische Struktur von *Activity*-Objekten mit ihren wesentlichen Eigenschaften und Operationen.

[1] vgl. z.B. Ahrens et al.: *Objektorientierte Prozeßsimulation in C++* und Sun: *AT&T Lang. System*

[2] vgl. Kapitel 8.2.2

[3] vgl. z.B. Nygaard et al.: *The Development of the Simula Languages*

Threads bzw. *Activity*-Objekte können nicht explizit erzeugt oder gelöscht werden [1]. Sie entstehen vielmehr dadurch, daß die parallele Ausführung eines Algorithmus (bzw. einer Funktion) veranlaßt wird. Dies geschieht durch Aufruf der Funktion *start (...)*, der (ein Zeiger auf) die auszuführende Funktion zu übergeben ist. Sie erzeugt ein neues *Activity*-Objekt, gliedert es in den Terminkalender ein und gibt einen Zeiger darauf zurück. Über diesen Zeiger kann der Thread im weiteren Verlauf kontrolliert werden. Threads werden beendet, wenn "ihr" Algorithmus vollständig abgearbeitet oder abgebrochen wurde. Das zugehörige *Activity*-Objekt wird dabei automatisch gelöscht.

Der in der Klasse *Activity* implementierte Zeitmechanismus wird durch die Klassenhierarchie *Blocker* abgerundet. Die Instanzen ihrer Klassen erlauben die einheitliche und übersichtliche Behandlung von Situationen, in denen Threads Aktionen unternehmen (wollen), die aufgrund relevanter Aspekte des Systemzustands zum aktuellen Zeitpunkt unmöglich sind.

Ein Beispiel einer solchen Situation ist der Versuch eines Teils im Beispielsystem aus Bild 32, einen Linearförderer zu belegen (bzw. sich darüber transportieren zu lassen), den das voranlaufende Teil noch nicht verlassen hat. Der notwendige Aufwand zur Behandlung dieser Situation trägt erheblich zur mangelnden Übersichtlichkeit der in den Bildern 33 bzw. 34 dargestellten Steuerungsalgorithmen des Beispielsystems bei.

```
class Blocker {
  List <Activity> Waiting;
  block () {
    Waiting.append (Activity::Current ());
    Activity::hold ();
  }
  unblock () {
    Activity *a = Waiting.removeFirst ();
    if (a != NULL)      a->schedule ();
  }
  wakeup () {
    for (Activity *a; (a = Waiting.removeLast ()) != NULL; a->schedule ());
  }
};
```

Bild 36: Grundzüge des Programmcodes der Klasse *Blocker*

[1] Dies wird dadurch verhindert, daß der öffentliche Teil der Schnittstelle der Klasse *Activity* keine Konstruktoren und keinen Destruktor enthält. Diese Maßnahme setzt das Konzept, aktive Objekte nicht als Instanzen sondern als Eigentümer (Aggregate) von Threads abzubilden, in Anwendungen wirksam durch.

Allgemein kennzeichnet solche Situationen, daß ein Thread blockiert bzw. unterbrochen werden muß, wenn eine bestimmte Bedingung (zunächst) nicht erfüllt ist und fortgesetzt werden soll, wenn diese Bedingung zu einem späteren Zeitpunkt erfüllt ist.

Die zur Behandlung derartiger Situationen in der Klassenhierarchie *Blocker* umgesetzte Lösung basiert darauf, daß solche kritischen Bedingungen als Objekte (einer Klasse der Hierarchie) abgebildet werden. Deren wesentliche in der (abstrakten) Basisklasse *Blocker* angesiedelte Funktionalität ist die Möglichkeit, Threads blockieren zu können. Für abgeleitete Klassen stehen dementsprechend Funktionen zum blockieren und fortsetzen von Threads zur Verfügung. Blockiert wird dabei immer der aktuelle (d.h. vom Zeitmechanismus gerade ausgeführte) Thread. Fortgesetzt werden kann entweder der erste (durch *unblock ()*) oder alle wartenden Threads (durch *wakeup ()*). Bild 36 zeigt die wesentlichen Teile des Programmcodes der Klasse *Blocker*.

```
class Flag : public Blocker {
  int Val;
  reset () { Val = 0; }
  set () {
    Val = 1;
    unblock ();
  }
  wait () {
    if (Val == 0)        block ();
  }
};
```

Bild 37: Grundzüge des Programmcodes der Klasse *Flag*

Eine abgeleitete Klasse ist die Klasse *Flag*. Ihre Instanzen sind die einfachsten konkreten Blocker. Gemäß der üblichen Vorstellung können *Flags* gesetzt oder nicht gesetzt sein, entsprechende Funktionen ermöglichen die Änderung. Hier am wichtigsten ist jedoch die Funktion *wait ()*. Wird sie bei gesetztem *Flag* aufgerufen, so kehrt sie sofort zurück und der aktuelle Thread läuft weiter. Bei nicht gesetztem *Flag* unterbricht bzw. blockiert ihr Aufruf dagegen den aktuellen Thread und trägt ihn in eine Warteliste ein. Das (spätere) Setzen des Flags veranlaßt die Eintragung des ersten wartenden Threads in den Terminkalender und damit dessen Entblockierung und Fortsetzung. Bild 37 zeigt die wesentlichen Teile des Programmcodes der Klasse *Flag*. Bild 38 zeigt zusammenfassend die softwaretechnische Struktur von *Blocker*- und *Flag*-Objekten mit ihren wesentlichen Eigenschaften und Operationen.

Eine die Klasse *Flag* nutzende dritte Version des Programmcodes zum Linearförderbeispiel zeigt Bild 39. Der Algorithmus hat jetzt deutlich an Übersichtlichkeit gewonnen und es wird vorstellbar, durch Zusammenfassung einiger Anweisungen zu weiteren Funktionen (was hier aber nicht mehr demonstriert werden soll) dahin zu kommen, daß der Algorithmus nur noch einfachste Anweisungen transporttechnischer Art (aufnehmen, transportieren, verlassen) und Steuerungsanweisungen (hier die Behandlung des Flags Frei) enthält.

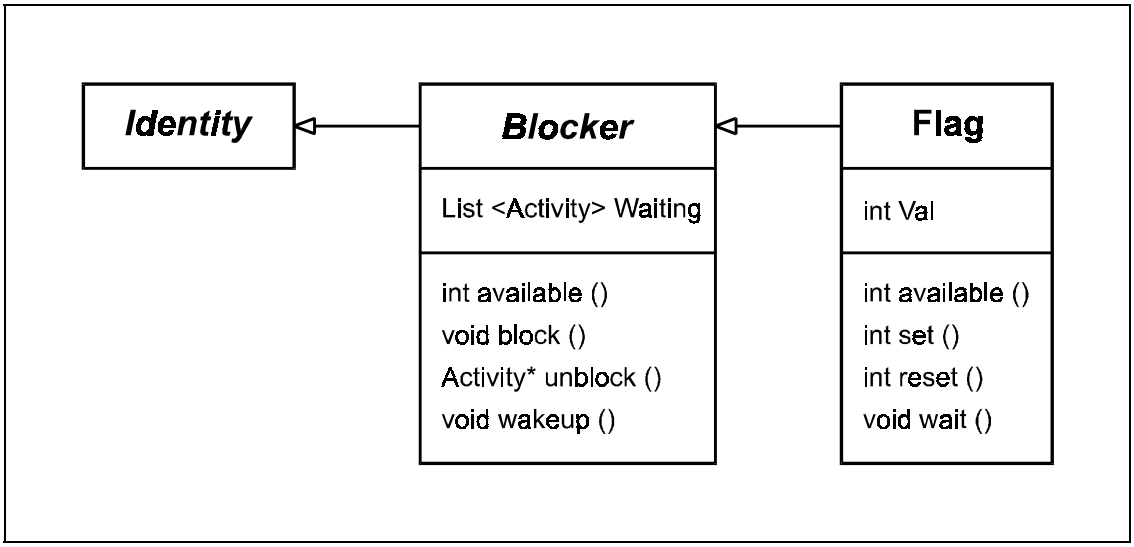

Bild 38: Struktur von *Blocker*- und *Flag*-Objekten

```
class Foerderer {
  Foerderer *Ausgang;
  Flag Frei;
  foerdern (Teil* teil) {
    Frei.wait ();
    Frei.reset ();
    Activity::hold (Aufnahmezeit);
    teil->HatVerlassen.set ();
    Activity::hold (Transportzeit);
    Activity::start (Ausgang->foerdern (teil));
    teil->HatVerlassen.reset ();
    teil->HatVerlassen.wait ();
    Frei.set ();
  }
};

class Teil {
  Flag HatVerlassen;
  durchlaufen {
    Activity::start (Foerderer1.foerdern (this));
  }
};
```

Bild 39: Programmcode zum Linearfördererbeispiel mit Nutzung der Klasse *Flag*

Durch Modifikation (bzw. durch Weglassung) dieser Steuerungsanweisungen und damit durch Anpassungen in einer einzigen Funktion könnte dann ohne Veränderung oder Beeinflussung anderer Details oder gar anderer Förderer die Charakteristik eines Förderers von Einzeltransport (wie hier) auf echtes Stauverhalten oder andere Varianten umgestellt werden.

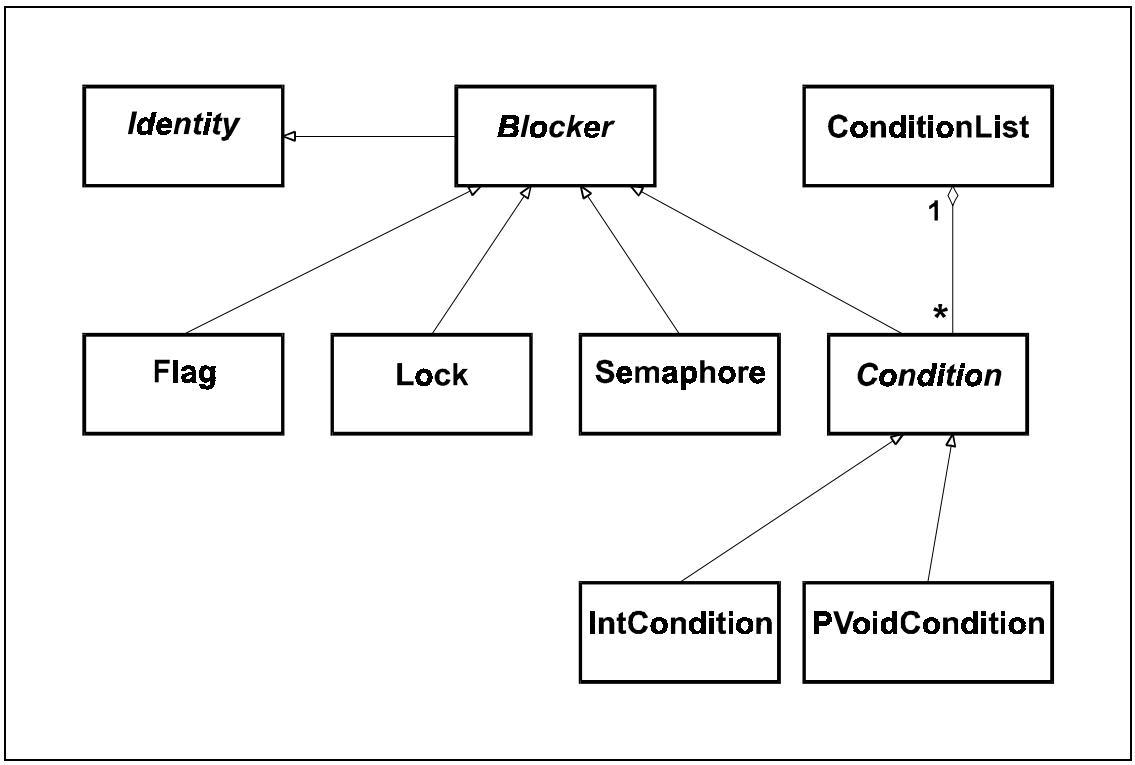

Bild 40: Klassenhierarche *Blocker*

Neben der Klasse *Flag* umfaßt die Klassenhierarchie *Blocker* die Klassen *Semaphore* und *Lock* sowie die untergeordnete Klassenhierarchie *Conditions*. Bild 40 gibt einen Überblick über die Struktur der Hierarchie und die Ererbungsbeziehungen der enthaltenen Klassen, die im folgenden beschrieben werden.

Semaphore-Objekte arbeiten in der aus der Systemprogrammierung bekannten Weise. Sie dienen zur Kontrolle und Sequentialisierung von Zugriffen auf zugriffsbeschränkte Ressourcen. Sie enthalten im wesentlichen einen Zähler, dessen Startwert die Anzahl gleichzeitig möglicher Zugriffe auf die kontrollierte Ressource angibt.

Durch Aufruf der Funktion *get ()* wird eine Semaphore (und damit ein Zugriffsrecht) "erworben". Dabei wird der Zähler dekrementiert und der aktuelle Thread blockiert, falls zum Aufrufzeitpunkt alle vorhandenen Zugriffsrechte vergeben sind. Nach Beendigung eines Zugriffs wird die Semaphore durch Aufruf der Funktion *release ()* wieder freigegeben. Dabei wird der Zähler wieder inkrementiert und erforderlichenfalls der erste auf den Zugriff wartende Thread fortgesetzt.

Locks regeln den Zugriff auf Objekte, die aus mehreren voneinander unabhängigen Gründen gesperrt sein können und die nur benutzt werden dürfen, wenn keine Sperre aktiv ist. Sperren können mit der Funktion *lock ()* gesetzt und mit der Funktion *unlock ()* wieder entfernt werden. Vor Zugriffen auf das kontrollierte Objekt ist die Funktion *wait ()* aufzurufen, die den aktuellen Thread erforderlichenfalls blockiert, bis alle Sperren entfernt wurden.

Die Instanzen der Klassen der (Teil-) Hierarchie *Conditions* repräsentieren Bedingungen allgemeinerer Art, die zur Fortsetzung eines Threads erfüllt sein müssen. Konkrete Bedingungen sind in Anwendungen als C-Funktionen zu codieren, die entweder einen ganzzahligen Wert (Klasse *IntCondition*) oder einen allgemeinen Zeiger (void*; Klasse *PvoidCondition*) zurückgeben. Eine Bedingung gilt entsprechend gängiger C-Konvention als erfüllt, wenn die zugehörige Funktion ein von 0 (bzw. NULL) verschiedenes Ergebnis liefert. Threads, die auf das Eintreten einer Bedingung warten (müssen), rufen die Funktion *wait ()* des entsprechenden *Condition*-Objekts auf, die sie blockiert, falls die Bedingung nicht erfüllt ist. Die Überprüfung einer Bedingungen wird durch Aufruf der Funktion *check ()* veranlaßt. Ist sie dabei erfüllt, werden alle wartenden Threads fortgesetzt.

Die Instanzen der Klasse *ConditionList* schließlich verwalten eine beliebige Anzahl zusammmenhängender (z.B. zum selben Objekt in Beziehung stehender) Bedingungen in einer Liste. Neben Funktionen zum Ein- und Ausordnen von Bedingungen bieten sie vor allem eine Funktion *check ()* an. Ihr Aufruf veranlaßt die Überprüfung aller in der Liste enthaltenen Bedingungen (durch Aufruf von deren *check ()*-Funktion). Erfüllte Bedingungen werden dabei gelöscht (und eventuell darauf wartende Threads fortgesetzt).

Der vorgestellte Zeitmechanismus erlaubt durch die Verfügbarkeit zeitverbrauchender Anweisungen die Formulierung übersichtlicher Algorithmen zur Beschreibung der Dynamik von Modellkomponenten. Da beliebige Funktionen parallel ausgeführt werden können, macht er weder Vorgaben hinsichtlich des Charakters aktiver Objekte noch begrenzt er die Zahl der von diesen ausgehenden zeitgleichen Aktivitäten.

Mit den Klassen der Hierarchie *Blocker* stehen ergänzend leistungsfähige Mittel zur kompakten und übersichtlichen Implementierung von Steuerungs- und Kontrollmechanismen zur Verfügung.

13 Fördertechnikkomponenten

Bei der Besichtigung von Materialflußsystemen und bei der Lektüre einschlägiger Fachzeitschriften oder -bücher [1], wird die beeindruckende und ständig wachsende Vielfalt der in der Praxis eingesetzten Fördersysteme deutlich. Eine wesentliche Teilaufgabe beim Entwurf bausteinbasierter Simulationswerkzeuge besteht darin, geeignete Modellelemente (als Bausteine) zur Abbildung solcher Fördersysteme in Modellen bereitzustellen [2]. Die Vielfalt realer Systeme muß dabei (durch Abstraktion) auf eine begrenzte Menge von Typen reduziert werden. Die Festlegung eines angemessenen Abstraktionsniveaus (und damit der resultierenden Typenmenge) unter Beachtung aller Anforderungen [3] ist allerdings keine einfache Aufgabe, wie schon daraus folgt, daß sich existierende Simulationswerkzeuge in diesem Punkt erheblich unterscheiden [4].

Nach Erkenntnissen aus der Psychologie, fühlen sich Menschen in Entscheidungssituationen überfordert, wenn sie mehr als (etwa) sieben Alternativen zur Auswahlen haben [5]. Weiter neigen sie dazu, in Wiederholungen solcher Situationen diejenigen Alternativen zu bevorzugen, die sie besser kennen, weil sie sie bereits früher gewählt hatten. Diese Effekte verstärken

---

[1] vgl. z.B. Koether: *Technische Logistik*; S. 15 ff.

[2] vgl. Kapitel 7.5.2

[3] vgl. Kapitel 8.1.4

[4] vgl. Kuhn, et al.: *Simulationsanwendungen in Produktion und Logistik*; S. 323 ff.

[5] vgl. Miller, G.: *The Magical Number Seven ...*

sich noch, wenn die Entscheidungen unter Streß (z.B. unter Zeitdruck) getroffen werden müssen [1].

Die Übertragung dieser Erkenntnisse auf die Frage der Festlegung der von einem Simulationswerkzeug bereitzustellenden Typenmenge legt es nahe, diese eher klein zu halten. Die gegebene Vielfalt von Fördersystemen führt dagegen (zumal in Verbindung mit dem Wunsch nach geometrisch richtiger Modellierung als Grundlage realistischer (online-) Visualisierungen [2]) tendenziell zu einer großen Typenzahl. Um angesichts dieses Widerspruchs zu einem geeigneten Kompromiß, also zu einer überschaubaren und beherrschbaren Typenmenge zu gelangen, ist zu klären, welche Aspekte von Typen bei Anwendern zu Schwierigkeiten im Umgang mit ihrer Anzahl führen können.

Der geradlinigste Ansatz beim objektorientierten Entwurf eines bausteinbasierten Materialflußsimulators ist sicher, alle in Materialflußsystemen vorkommenden Komponententypen wie Maschinen, Linearförderer, Drehtische, Regalförderzeuge, Lagerplätze, Kräne, etc. in je einer Klasse zu beschreiben. Jede dieser Klassen implementiert ein komplexes und abgeschlossenes Verhaltensmuster, das sich von dem aller anderen Klassen unterscheidet (sonst wäre die Klasse überflüssig). Die Komplexität ist unvermeidlich, da die abgebildeten realen Fördertechnikelemente ebenfalls ein komplexes Verhalten haben [3]. Abgeschlossenheit bedeutet, daß die Klasse alle Aspekte des Verhaltens beschreibt und daß bei ihrem Einsatz keiner dieser Aspekte ausgeschaltet bzw. weggelassen und kein neuer hinzugefügt werden kann [4]. Die Abgeschlossenheit resultiert daraus, daß das Werkzeug bei der Anwendung als ausführbare Software eingesetzt wird, in der die Klassen bzw. Verhaltensmuster als Algorithmen im Programmcode enthalten sind, die nicht (mehr) verändert, sondern (durch Erzeugen von Instanzen) verwendet werden.

Nach den Projekterfahrungen des Autors ist es eben diese Vielfalt teilweise nur im Detail abweichender Verhaltensmuster, die den Umgang mit einer großen Typenmenge schwierig macht. Die mit der Entscheidung, eine reale Systemkomponente in einem Modell als Instanz einer bestimmten Klasse (bzw. als Baustein eines bestimmten Typs) abzubilden verbundene Festlegung eines komplexen Verhaltensmusters geht an den Schlüsselstellen des Modells

---

[1] Dies beschränkt die Auswahlmöglichkeiten de facto auf eine individuell überschaubare Menge.

[2] vgl. Kapitel 8.1.4

[3] Beispielsweise ist für viele von ihnen die Realisierung einer aufwendigen Steuerung (u.U. auf einem dedizierten Rechner) eine notwendige Einsatzvoraussetzung.

[4] Auf Versuche, solche Verhaltensmuster durch aufgabenbezogene Programmierbarkeit oder die Vorgabe von Strategien adaptierbar zu machen, wird weiter unten eingegangen.

zu selten reibungsfrei vonstatten. Häufig stören einzelne Aspekte, während sich andere Dinge gar nicht oder nur sehr umständlich erreichen lassen [1].

Der klassische Ansatz, mit dieser keineswegs neuen Situation umzugehen, ist die Einführung von Strategien, durch die das Verhalten einer Klasse für Instanzen individuell angepaßt werden kann. Dieser Ansatz hat jedoch zwei gewichtige Nachteile. Zum einen müssen die Strategien ebenso wie das grundlegende Verhaltensmuster Bestandteil des Programmcodes sein, so daß ihre Anzahl begrenzt bleiben muß [2]. Mindestens in Situationen in denen es viele mögliche Lösungen gibt, wie z.B. bei der Festlegung der Abarbeitungsreihenfolge bei einem Transportsystem vorliegender Aufträge, ist die Chance groß, daß die "passende" Strategie in der Klasse des Transportsystems nicht vorhanden ist. Zum anderen können Strategien nur auf den Zustand der zu steuernden Komponente (und vielleicht auf den ihrer unmittelbaren Nachbarschaft) aufgebaut sein [3]. Im o.a. Fall der Reihenfolgefestlegung verhindert dies z.B. die Einbeziehung des Füllstands eines entfernten Puffers.

Ablaufsteuerungen, die neben der lokalen Situation auch relevante Aspekte der Gesamtanlage berücksichtigen, finden aber heute wegen des in Kapitel 4.2 dargestellten veränderten Zielsystems der Produktion zunehmende Verbreitung. Ihre angemessene Abbildung in Simulationsmodellen ist also durch die Bereitstellung von Strategien (allein) nicht möglich. Das eingesetzte Softwarewerkzeug muß darüberhinaus, wie bereits in Kapitel 8.1.4 gefordert, eine Schnittstelle für die anlagen- bzw. aufgabenbezogene Programmierung bieten.

Ein derartiges Feature eines Simulationswerkzeugs steht in enger Beziehung zur hier diskutierten Abbildung fördertechnischer Elemente, da jede Ablaufsteuerung ja gerade auf die Beinflussung des Verhaltens des Modells bzw. seiner Komponenten abzielt. Die Klasse(n), in denen das Verhalten dieser Komponenten beschrieben ist, können dieses also nicht vollständig und unveränderlich festlegen, sondern müssen ihrerseits eine Schnittstelle für die Einflußnahme durch Ablaufsteuerungen bieten [4].

Ausgehend von diesen Überlegungen wurde im Rahmen dieser Arbeit ein Konzept entwickelt, daß anstelle fertiger komplexer Fördertechnikkomponenten ein Baukastensystem bereitstellt,

---

[1] Der Umgang mit der je Typ bzw. Klasse unterschiedlichen Struktur, die sich in der Anwendung vor allem in je spezifischen Parametersätzen äußert, geht dagegen meistens recht leicht von der Hand.

[2] Sollte es für eine Klasse einmal viele Strategien geben, liegt wiederum eine der Typwahl vergleichbare Auswahlproblematik vor.

[3] Der Grund hierfür ist, daß zum Zeitpunkt der Implementierung einer Strategie keine Annahmen über Art und Identität entfernter Modellkomponenten möglich sind. Es kann noch nicht einmal angenommen werden, daß bestimmte Komponenten (wie z.B. ein Lager) überhaupt vorhanden sind.

[4] Andere Aspekte der aufgabenbezogenen Programmierung werden in Kapitel 12 diskutiert.

aus dessen Elementen reale Komponenten entsprechend ihres Aufbaus und ihrer Fähigkeiten zusammengesetzt werden können. Dabei werden sowohl die physikalische als auch die logische Struktur (also das Verhalten) der Komponenten montiert. Die Grundelemente des Baukastens sind Klassen, die in die zwei Hierarchien *Places* und *Movers* angeordnet sind. Diese werden in den beiden folgenden Abschnitten vorgestellt.

Jede zusammengesetzte Kombination von Grundelementen wird in einer Instanz der Klasse *SingleUnit* abgelegt, die die jeweilige Fördertechnikkomponente als Einheit im Modell repräsentiert. Alle Komponenten haben damit nach außen eine identische Schnittstelle. Sie bietet einen Vorrat von Operationen z.B. für die Behandlung von Teilen an. Die Klasse *SingleUnit* wird im folgenden ebenfalls vorgestellt. Daran anschließend wird der praktische Einsatz des Baukastensystems an einigen Beispielen realer Fördertechnikkomponenten demonstriert. Den Abschluß dieses Kapitels bildet ein Abschnitt, der die Abbildung materialflußtechnischer Verkettungen von Komponenten und die Wegesuche in Modellen bzw. Anlagen behandelt.

13.1 Die Klassenhierarchie *Places*

Die Klassen der Hierarchie *Places* bilden Plätze ab. Darunter werden hier diejenigen Teile von Förderern verstanden, die mit den geförderten Teilen direkt in Berührung stehen, wo also die Relativbewegung von Teilen und Förderer stattfindet. Bild 41 gibt einen Überblick über die Struktur der Hierarchie und die Beziehungen der enthaltenen Klassen.

Alle Klassen der Hierarchie sind (direkt oder indirekt) aus der abstrakten Basisklasse *Place* abgeleitet. Sie gibt eine einheitliche Schnittstelle vor, über die alle Plätze angesprochen werden können und legt weiter fest, daß jeder Platz eine Instanz der Klasse *GeoObject* [1] enthält, so daß er ein eigenes Koordinatensystem definiert und die Möglichkeit bietet, ihm Konturelemente für die Visualisierung zuzuordnen.

Zur Nachbildung des Transports von Teilen über Plätze stehen Funktionen zum Aufnehmen eines Teils von einem materialflußtechnisch vorgelagerten Platz (*take ()*), zum Abgeben eines Teils an eine nachgelagerten Platz (*give ()*) und zum Bewegen eines Teils über den Platz (*transport ()*) zur Verfügung. Diese Funktionen werden durch die Teile aufgerufen [2]. Die Transportcharakteristik jedes Platzes kann (auch im laufenden Experiment) zwischen Stau-

[1] vgl. Kapitel 11.1

[2] vgl. Kapitel 14

(wie z.B. bei einer Rollenbahn) und Kettenbetrieb (wie z.B. bei einem Gurtband) durch den Aufruf einer entsprechenden Funktion umgeschaltet werden.

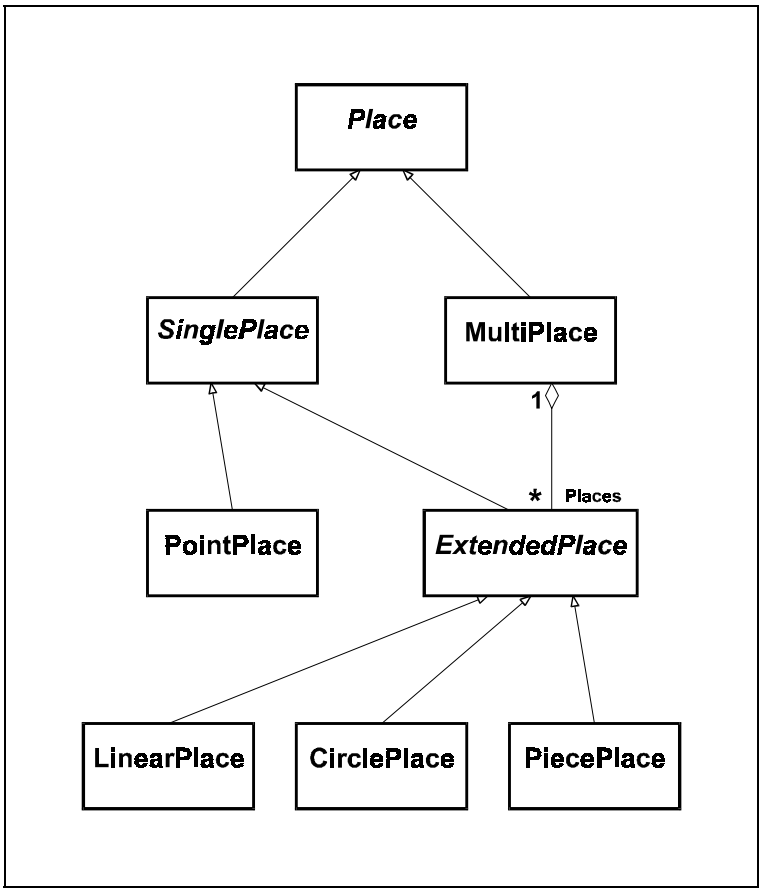

Bild 41: Klassenhierarchie *Places*

Die weiteren Funktionen *moveTo (...)*, *rotTo (...)* und *placeTo (...)* ermöglichen das Auslösen von Bewegungen des Förderers, zu dem der angesprochene Platz gehört (einschließlich eventuell darauf befindlicher Teile). Diese Funktionen reichen die jeweilige Bewegungsanforderung an "ihren" Förderer (bzw. dessen *PlaceList*-Objekt) weiter [1].

Schließlich enthält jedes *Place*-Objekt einen Motor, der eine Instanz der Klasse *Lock* [2] ist und über Funktionen der Schnittstelle angehalten und gestartet werden kann. Der geeignete Einsatz dieser Funktionen bietet die Möglichkeit, zur Nachbildung von Störungen alle Transportbewegungen auf dem jeweiligen Platz zu unterbrechen.

Die ebenfalls abstrakte Klasse *SinglePlace* beschreibt Einzelplätze (im Gegensatz zu den weiter unten behandelten *MultiPlaces*). Jede Instanz enthält zwei Vektoren als Eingangs- (*InPt*) bzw. Ausgangspunkt (*OutPt*) des Platzes. Weitere Vektoren legen die Förderrichtung (*Dir*) und die Richtung "oben" fest (*Up*, wird wie Bild 43 zeigt als Projektion der z-Achse des Platzkoordinatensystems in die Ebene senkrecht zur Förderrichtung errechnet), die für die Positionierung von Teilen auf Plätzen benötigt werden. Schließlich verwaltet jede Instanz noch eine Liste der jeweils auf dem Platz befindlichen Teile. Bild 42 zeigt die Beziehungen zwischen einem *SinglePlace* und darauf befindlichen Teilen.

---

[1] vgl. Kapitel 13.2 und 13.3

[2] vgl. Kapitel 12.4

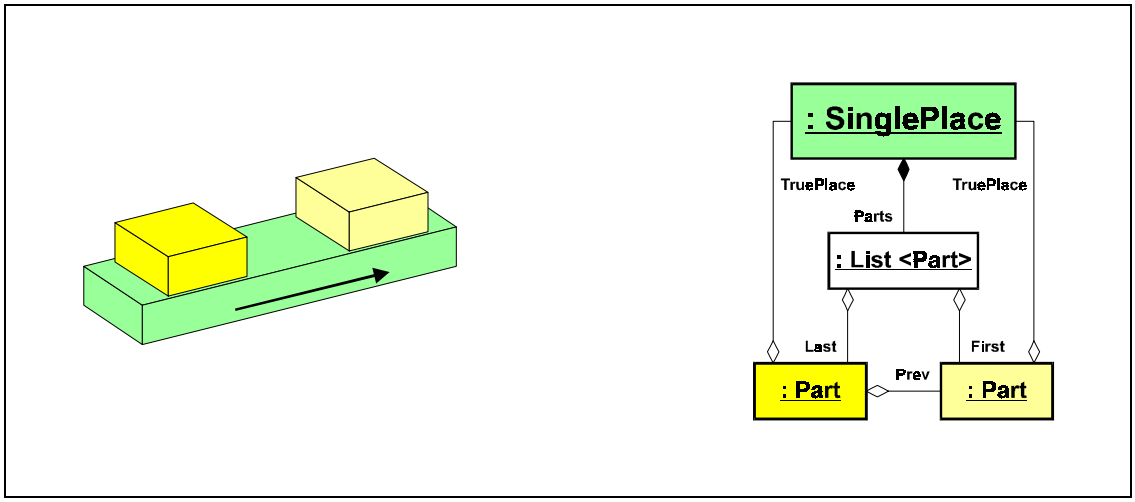

Bild 42: Objektbeziehungen auf einem *SinglePlace*

Die Instanzen der Klasse *PointPlace* bilden Plätze ab, die als punktförmig idealisiert werden können. Auf *PointPlaces* kann jeweils nur ein Teil vorhanden sein, das beim Aufnehmen auf einen Bezugspunkt zentriert wird. Dieser wird als Mittelpunkt der Strecke vom Eingangs- zum Ausgangspunkt des Platzes berechnet. Als weitere Parameter können Zeiten für die Dauer der Übernahme und Übergabe von Teilen von bzw. an andere Plätze vorgegeben werden. Als *PointPlaces* können z.B. der Haken eines Krans, die Gabel eines Staplers sowie Plätze in Lagern betrachtet werden [1].

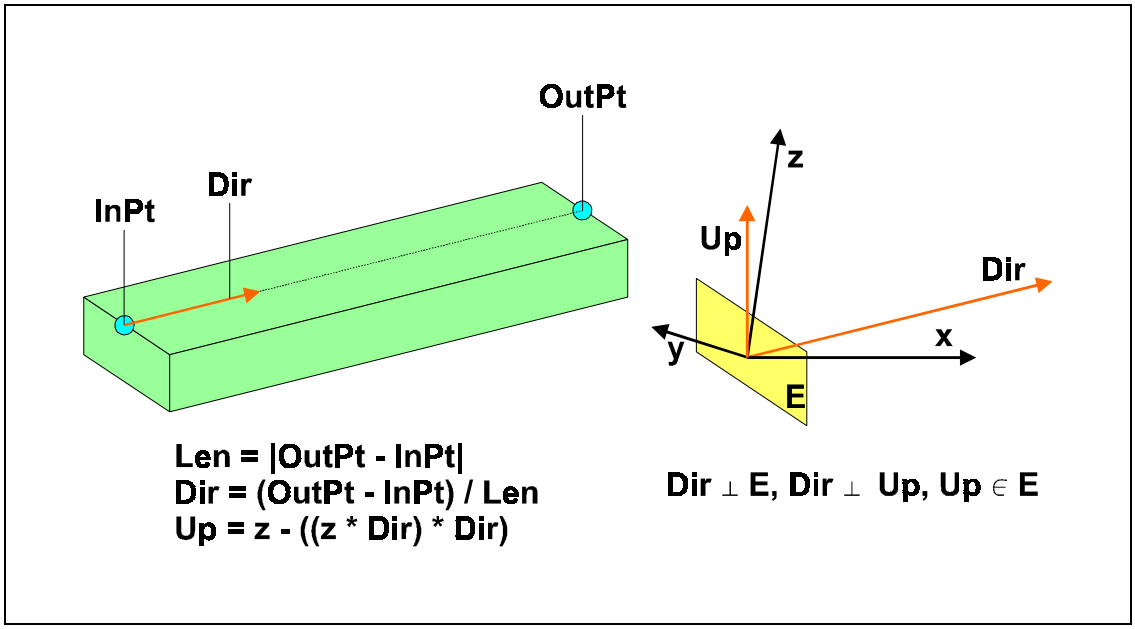

Bild 43: Geometrie eines *LinearPlace*

[1] In der Realität sind solche Plätze natürlich durchaus räumlich ausgedehnt. Ihre Abbildung als *PointPlaces* ist jedochwegen ihrer Eigenschaften (meistens) zweckmäßig, da sie in der Regel je ein Teil aufnehmen, dessen Zentrierung auf dem Platz außerdem einfachstes Handling erlaubt.

Die abstrakte Klasse *ExtendedPlace* ist die Basis für weitere Klassen, die räumlich ausgedehnte Plätze mit linienförmigem Materialfluß abbilden. Sie enthält deren jeweilige Länge und die Fördergeschwindigkeit.

Die Instanzen der Klasse *LinearPlace* bilden Plätze ab, über die die Teile entlang einer Geraden vom Eingangs- zum Ausgangspunkt gefördert werden. Bild 43 zeigt die Geometrie eines solchen Platzes.

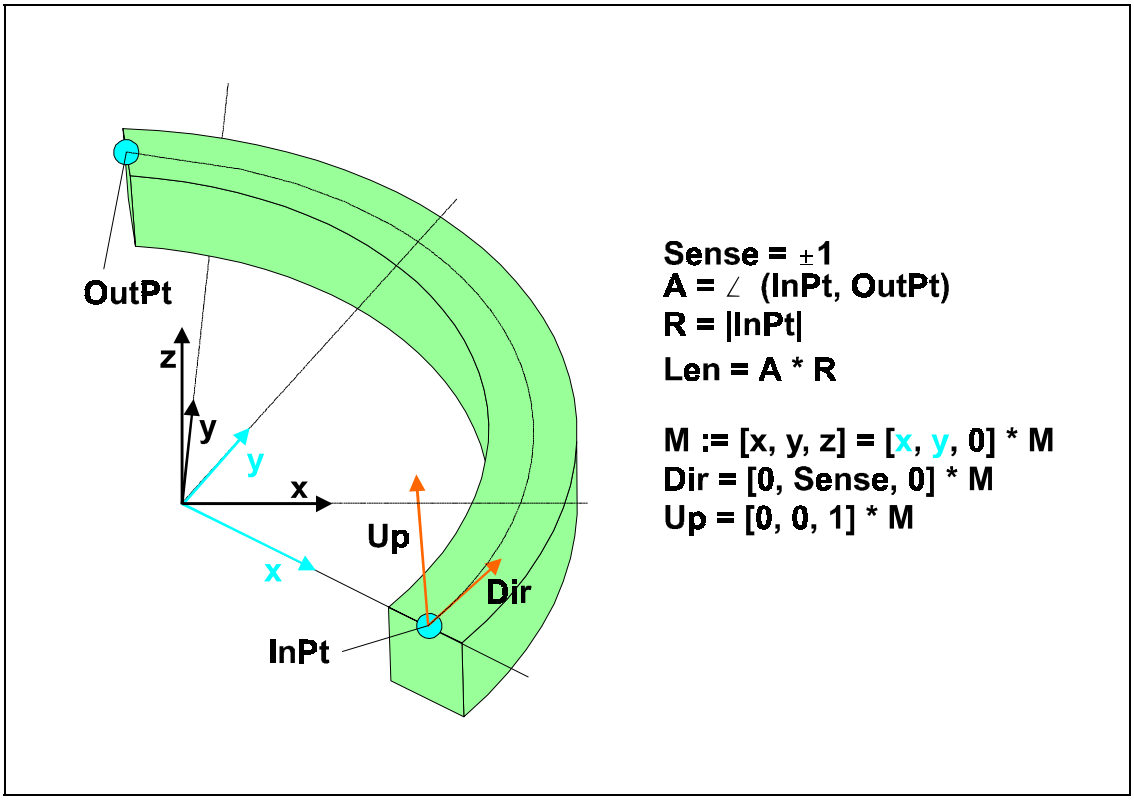

Bild 44: Geometrie eines *CirclePlace*

Die Instanzen der Klasse *CirclePlace* bilden Plätze ab, über die die Teile entlang einer (ebenen) Kreisbahn vom Eingangs- zum Ausgangspunkt gefördert werden. Als Mittelpunkt des Bogens wird der Ursprung des Platzkoordinatensystems verwendet, der Drehsinn (Links- bzw. Rechtsbogen) ist bei der Erzeugung anzugeben. Zur Vereinfachung der Positionsberechnung von Teilen auf einem *CirclePlace* wird intern ein virtuelles Koordinatensystem verwendet, in dem der Bogen in der x,y-Ebene verläuft (im Bild blau dargestellt). Bild 44 zeigt die Geometrie eines solchen Platzes.

Zur Demonstration des Potentials der Klassenhierarchie *Places* wird hier noch die Klasse *PiecePlace* vorgestellt. Ihre Instanzen bilden Plätze ab, über die die Teile vom Eingangspunkt über eine beliebige Anzahl weiterer anzugebender Stützpunkte zum Ausgangspunkt gefördert werden. Als Bahn werden z. Zt. Geradenstücke vom Stützpunkt zu Stützpunkt errechnet.

Es ist jedoch möglich, aufwendigere Interpolationen (z.B. Bézier-Splines) zu verwenden ohne (abgesehen vielleicht vom Konstruktor) die Schnittstelle der Klasse ändern zu müssen. So könnten (fast) beliebige räumliche Bahnen abgebildet werden.

Von zentraler Bedeutung im Hinblick auf die einheitliche Abbildung der verschiedensten Fördertechnikkomponenten und damit für die Vereinfachung des Modellentwurfs ist, daß alle Plätze (nach außen) eine identische Schnittstelle bieten und ein gleichartiges Verhalten zeigen. Dies wird hier dadurch erreicht, daß sich alle *ExtendedPlaces* [1] nur in den (internen) Bahnbeschreibungsdaten und den (von der Bahn abhängigen) Algorithmen zur (Neu-) Berechnung der Position von Teilen nach Bewegungen unterscheiden, während alle übrigen Daten und Algorithmen in den Basisklassen enthalten und damit identisch sind.

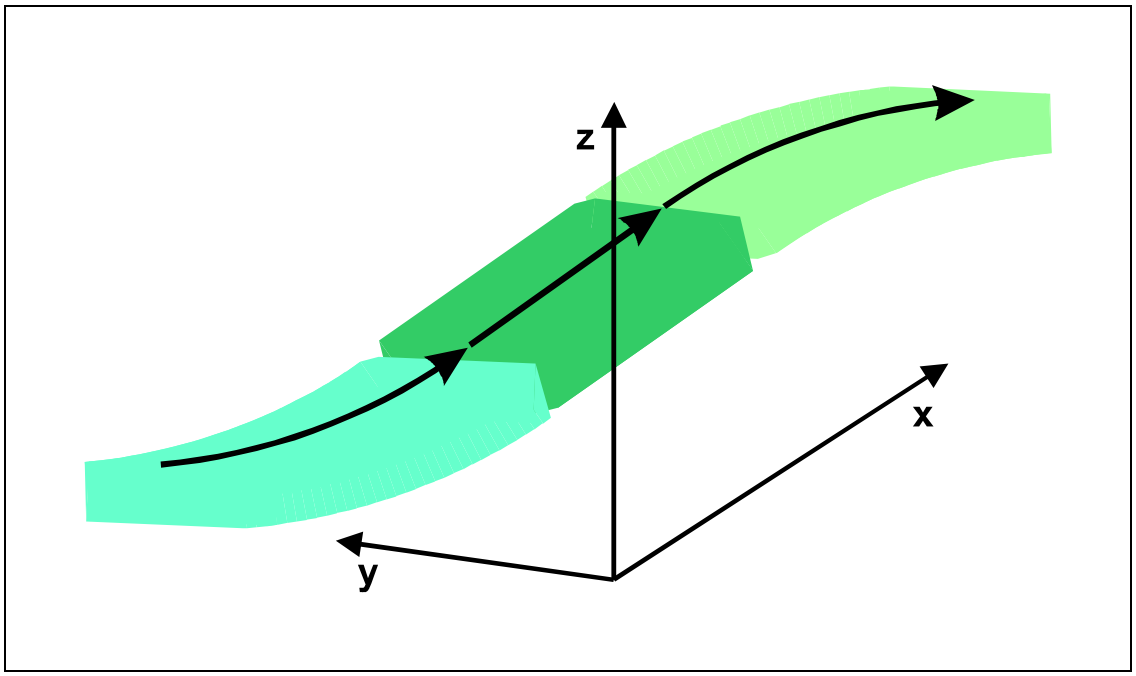

Bild 45: Steigungsstrecke als *MultiPlace*

Aus der Hierarchie *Places* verbleibt zur Beschreibung noch die Klasse *MultiPlace*. Ihre Instanzen ermöglichen die Zusammenfassung mehrerer *ExtendedPlaces* zu einer Einheit, die nach außen als ein Platz behandelt wird. Bild 45 zeigt ein Beispiel in dem eine (als *LinearPlace* abgebildete) Steigungsstrecke mit den zugehörigen (als *CirclePlaces* abgebildeten) ein- und ausleitenden Übergängen zu einem einzigen Platz zusammengefaßt wurde.

MultiPlace-Objekte haben aufgrund der Ableitung aus der Klasse *Place* ein eigenes Koordinatensystem. In dieses sind die Koordinatensysteme der zugehörigen *ExtendedPlaces*

[1] Dies sind alle Instanzen der aus *ExtendedPlace* abgeleiteten Klassen *LinearPlace*, *CirclePlace* und *PiecePlace*.

eingeordnet, so daß *MultiPlaces* als Gruppe positioniert werden können. Sie sind aber nicht nur geometrische sondern auch logische Einheiten, so daß durch einfache Kapazitätsanpassung (der enthaltenden *SingleUnit*) ohne weitere Programmierung z.B. erreicht werden kann, daß jeweils nur ein Teil auf der Gesamtstrecke vorhanden ist [1]. Teile, die über einen *MultiPlace* gefördert werden, laufen nacheinander über alle zugehörigen *ExtendedPlaces*. Bild 46 zeigt die Beziehungen zwischen einem *MultiPlace*, den enthaltenen Plätzen und darauf befindlichen Teilen.

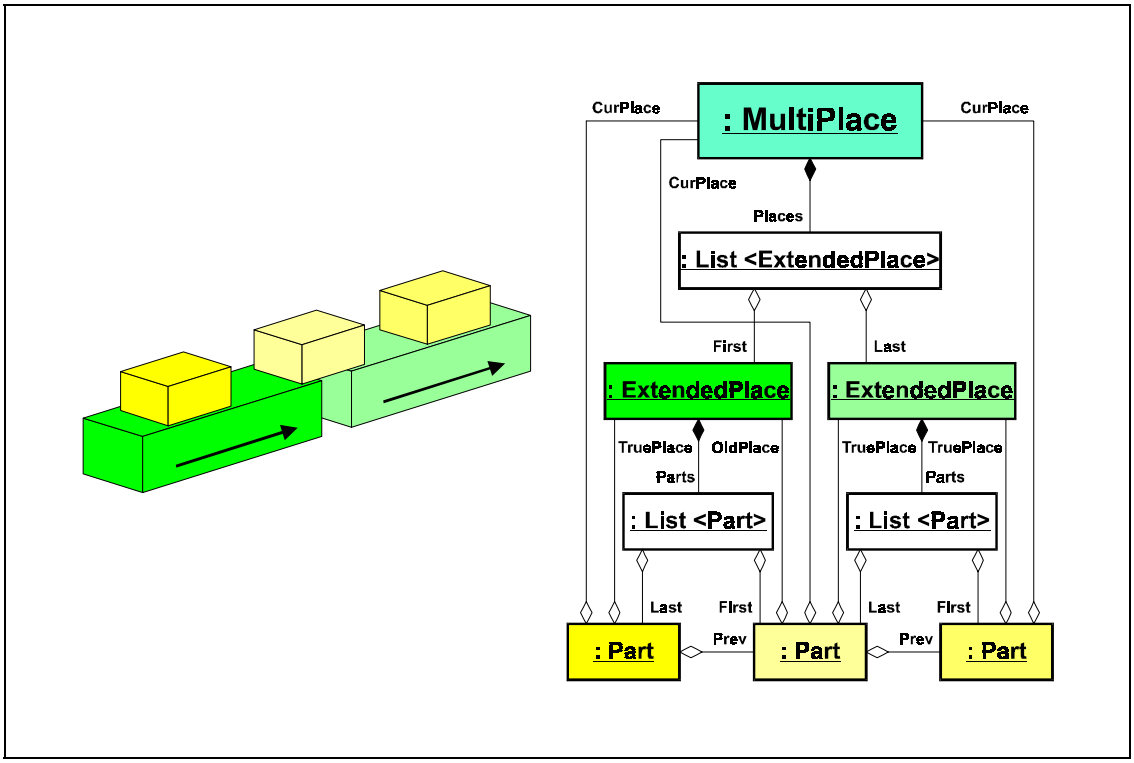

Bild 46: Objektbeziehungen auf einem *MultiPlace*

13.2 Die Klassenhierarchie *Movers*

Die Klassen der Hierarchie *Movers* bilden diejenigen Teile von Förderern ab, die die Bewegung von Förderern relativ zur Umgebung realisieren. Trotz der erwähnten Vielfalt heutiger Fördertechnik kommen als grundsätzliche Bewegungsarten nur Verschiebungen (Translationen) und Drehungen (Rotationen) vor. Auch die komplexesten Handhabungsgeräte kombinieren in ihrem Bewegungsapparat nur translatorische und rotatorische Achsen. Im übrigen kommen (jedenfalls in den relevanten Anwendungsfällen) auch keine anderen Kombinationen als Aneinanderreihungen bzw. Aufeinanderstapelungen von Bewegungsachsen vor.

---

[1] Nach den Erfahrungen des Autors kommt diese Anforderung gerade bei Steigungsstrecken oft vor.

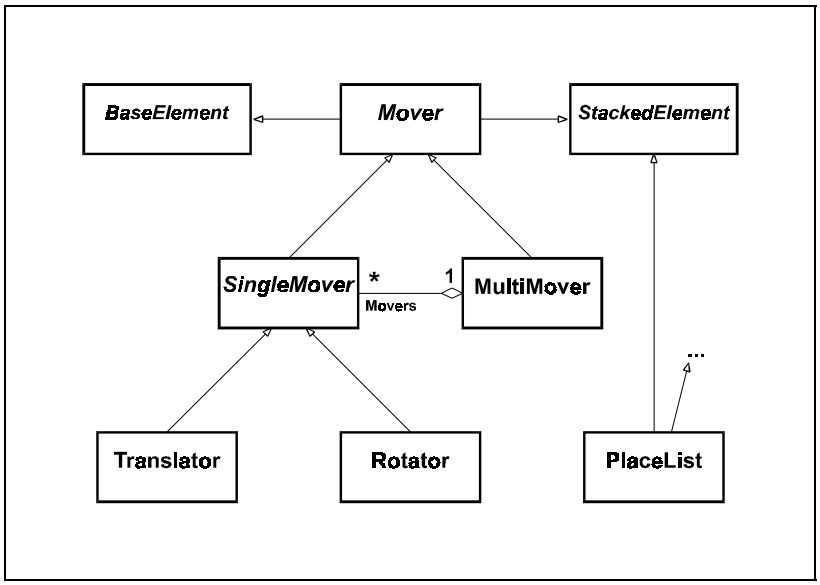

Bild 47: Klassenhierarchie *Movers*

Die Klassenhierarchie *Movers* bildet diese Bewegungsarten und ihre Kombinationsmöglichkeiten ab. Bild 47 gibt einen Überblick über die Struktur der Hierarchie und die Beziehungen der enthaltenen Klassen, die im folgenden vorgestellt werden.

Die beiden abstrakten Klassen *BaseElement* und *StackedElement* unterstützen die Aufeinanderstapelung von Bewegungsachsen. Ein Objekt, das ein *BaseElement* ist (dessen Klasse also von der Klasse *BaseElement* abgeleitet ist), kann (als Basis) unter ein anderes Objekt (dessen Klasse von der Klasse *StackedElement* abgeleitet sein muß) eingeordnet werden. Umgekehrt kann jedes *StackedElement* auf ein *BaseElement* aufgestapelt werden. Die Beschränkung auf je ein über- bzw. untergeordnetes Objekt verhindert Verzweigungen im Stapel der Bewegungselemente. Jedes *StackedElement*-Objekt enthält außerdem eine Instanz der Klasse *GeoObject* [1], wodurch es ein eigenes Koordinatensystem definiert und darüberhinaus die Möglichkeit bietet, ihm Konturelemente für die Visualisierung zuzuordnen.

Die ebenfalls abstrakte Klasse *Mover* ist die Basisklasse für alle “echten” [2] Bewegungselemente bzw. -klassen. Sie ist aus den Klassen *BaseElement* und *StackedElement* abgeleitet. Da alle Bewegungselemente von Förderern (von außen) als *Mover* behandelt werden, ist die Schnittstelle dieser Klasse der Zugang zur Auslösung aller Bewegungen von Förderern. Sie bietet dazu im wesentlichen die Funktionen *moveTo (...)* (reine Verschiebung), *rotTo (...)* (reine Drehung) und *placeTo (...)* (kombinierte Verschiebung und Drehung), denen jeweils eine Position und ein Ziel zu übergeben ist. Darüber hinaus bietet die Schnittstelle Funktionen, die es erlauben, *Mover* in entsprechende Stapel einzuordnen oder daraus zu entfernen.

---

[1] vgl. Kapitel 11.1

[2] Es wird noch gezeigt, daß zur Verzahnung der Bewegungselemente mit den übrigen Teilen eines abzubildenden Förderers die Einführung weiterer von *BaseElement* bzw. *StackedElement* abgeleiteter Klassen sinnvoll ist, die jedoch keine Bewegungen audführen können.

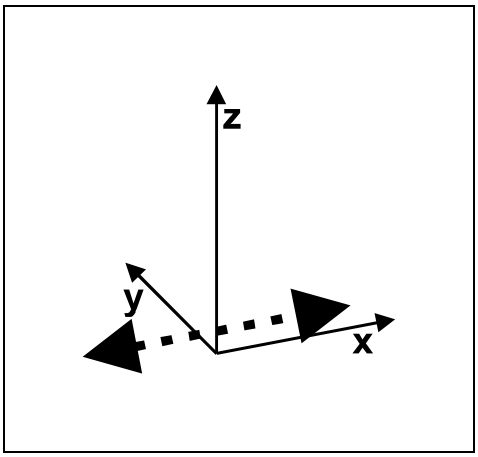

Bild 48: Bewegung eines *Translators*

Bei Aufruf einer der Bewegungsfunktionen versucht jeder *Mover* zunächst, den Auftrag an den in der Struktur unterhalb eingeordneten *Mover* weiterzugeben. Existiert kein solcher oder hat dieser seinen Teil der Bewegung erledigt, ermittelt der *Mover*, ob er mit seinen Bewegungsmöglichkeiten die angegebene Position näher an das Ziel bewegen kann. Falls dies möglich ist, führt er die Bewegung aus. Danach (oder falls er sich nicht sinnvoll bewegen konnte) gibt er den Auftrag an den Aufrufer zurück. Aufeinandergestapelte *Mover* zerlegen eine Bewegung also in nacheinander ausgeführte Einzelbewegungen. Die erste Bewegung führt der dem Fundament der jeweiligen Komponente zunächst liegende *Mover* aus, dann pflanzt sich die Bewegung nach oben fort [1].

Die nächste (ebenfalls abstrakte) Klasse *SingleMover* umfaßt einzelne Bewegungselemente. Sie wurde eingeführt, um die Aufbaumöglichkeiten von *MultiMovern* zu beschränken (s.u.) und hat sonst keine Funktion.

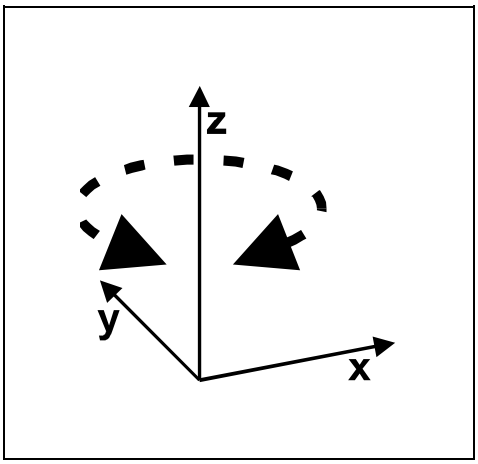

Bild 49: Bewegung eines *Rotators*

Die Klasse *Translator* beschreibt Bewegungselemente, die Verschiebungen ausführen können. Wie Bild 48 zeigt, erfolgt die Bewegung entlang der x-Achse des lokalen Koordinatensystems des *Translators* [2]. Die Bewegungsgeschwindigkeit ist individuell einstellbar.

Die Klasse *Rotator* beschreibt Bewegungselemente, die Drehungen ausführen können. Die Bewegung erfolgt, wie Bild 49 zeigt, um die z-Achse des lokalen Koordinatensystems [3]. Die Drehgeschwindigkeit ist für jeden *Rotator* individuell einstellbar.

---

[1] vgl. auch Kapitel 13.4.4 für genauere Erläuterungen der Konsequenzen dieses Verhaltens.

[2] Diese Beschränkung vereinfacht die internen Berechnungsalgorithmen. De facto können *Translator* Verschiebungen in jede Richtung ausführen, indem ihr Koordinatensystem (in dem ihm übergeordneten) so ausgerichtet wird, daß die x-Achse in die gewünschte Bewegungsrichtung zeigt.

[3] Die Gründe für diese Beschränkung und die faktischen Möglichkeiten wurden bei *Translator*-Objekten erläutert.

Die Instanzen der Klasse *MultiMover* ermöglichen die Zusammenfassung mehrerer (übereinander angeordneter) *SingleMover* zu einer Einheit, die nach außen als ein *Mover* behandelt wird. Die Bewegungen der in einen *MultiMover* eingeordneten Bewegungselemente werden statt nacheinander parallel ausgeführt. Damit kann also die gleichzeitige Bewegung mehrerer Achsen eines Förderes nachgebildet werden. Die für eine Bewegung aufgewendete Zeit entspricht jeweils der längsten Zeit, die einer der enthaltenen *SingleMover* für den von ihm ausgeführten Anteil an der Bewegung benötigt.

Aus der Hierarchie *Movers* verbleibt zur Beschreibung noch die Klasse *PlaceList*. Je eine Instanz dieser Klasse wird in nach dem hier dargestellten Konzept modellierten Fördertechnikkomponenten als Verbindung zwischen den (unterhalb angeordneten) Bewegungselementen (*Movern*) und den (oberhalb befindlichen) Plätzen verwendet. Zur technischen Realisierung ist die Klasse *PlaceList* einerseits aus der Klasse *StackedElement* (s.o.) abgeleitet, so daß *PlaceList*-Objekte auf *Mover* aufgestapelt werden können. Andererseits sind *PlaceList*-Objekte (durch eine in Bild 47 nicht ausführlich dargestellte private Ererbung) auch Listen von Plätzen [1]. Für die Verwaltung der enthaltenen Plätze bietet die Schnittstelle der Klasse entsprechende Funktionen, die z.B. das Hinzufügen und Entfernen von Plätzen ermöglichen.

Die Funktionen *moveTo (...)*, *rotTo (...)* und *placeTo (...)* [2] erlauben Plätzen (bzw. darauf befindlichen Teilen) die Anforderung von Bewegungen ohne Kenntnis des individuellen Bewegungsapparates der jeweiligen Fördertechnikkomponente. *PlaceList*-Objekte reichen Bewegungsanforderungen an die unterhalb eingeordneten *Mover* weiter. Dabei erfüllen sie zusätzlich die Aufgabe, erforderlichenfalls mehrere vorliegende Bewegungsanforderungen zu sequentialisieren und diese gegen gleichzeitige Übernahme- und Übergabeoperationen auf den Plätzen zu verriegeln. Die Koordinatensysteme der *PlaceList*-Objekte selbst werden dabei nicht verändert (verschoben oder rotiert), so daß sie in der Regel mit dem jeweils übergeordneten Koordinatensystem identisch sind. Sie werden lediglich zur Vereinfachung der Manipulation des Aufbaus von *SingleUnits* (s.u.) eingesetzt.

Schließlich umfaßt die Klasse *PlaceList* noch Funktionen, die im Gange befindliche Bewegungen unterbrechen bzw. unterbrochene Bewegungen fortsetzen. Sie bieten eine Schnittstelle, über die Störungen von Fördertechnikkomponenten nachgebildet werden können. *PlaceList*-Objekte unterbrechen dabei sowohl die Bewegungen unterhalb eingeordneter *Mover* wie auch alle Transportbewegungen auf den enthaltenen Plätzen.

---

[1] Erläuterungen zur Klasse *List* gibt Kapitel 10.4

[2] s.a. die Beschreibung der Klasse *Mover* weiter vorn in diesem Kapitel.

13.3 Die Klasse *SingleUnit*

Jede Fördertechnikkomponente eines Materialflußsystems wird im hier vorgestellten Konzept als eine Instanz der Klasse *SingleUnit* modelliert. Jede *SingleUnit* verwaltet den internen Aufbau der jeweiligen Komponente, also den aus *Movern* individuell zusammengesetzten Bewegungsapparat und die zugehörigen Plätze. Die Klasse *SingleUnit* drückt die Gleichartigkeit aller Fördertechnikkomponenten aus, indem sie für jede von ihnen die gleiche Schnittstelle definiert, über die sie, trotz ihres unterschiedlichen Aufbaus, im Modell auf einheitliche Weise behandelt werden können. Bild 50 zeigt wesentliche Teile der softwaretechnischen Struktur von *SingleUnit*-Objekten.

Im internen Aufbau repräsentiert die *SingleUnit* das Fundament der Komponente. Sie erbt dazu von der Klasse *BaseElement* [1], so daß ein als Kombination von *Movern* aufgebauter Bewegungsapparat auf das Fundament aufgestapelt werden kann. Aus Gründen, die in Kapitel 16.1 erläutert werden, ist die Klasse *SingleUnit* auch aus der Klasse *Unit* abgeleitet. Sie enthält daher eine Instanz der Klasse *GeoObject* [2], wodurch sie zum einen ein Koordinatensystem für die gesamte Komponente definiert und darüberhinaus die Möglichkeit bietet, dieser Konturelemente für die Visualisierung zuzuordnen.

Um materialflußtechnische Verkettungen von Fördertechnikkomponenten abbilden und verwalten zu können, ist die Klasse *SingleUnit* schließlich noch aus der Klasse *Linker* abgeleitet, die in Kapitel 13.5 vorgestellt wird.

Jede *SingleUnit* enthält ein *PlaceList*-Objekt, in dem die Plätze der Komponente verwaltet werden und dessen Koordinatensystem in das der *SingleUnit* eingeordnet ist. Der (als *Mover*-Stapel abgebildete) Bewegungsapparat der Komponente wird in diesen Aufbau zwischen die *SingleUnit* und ihre *PlaceList* eingeordnet (dies gilt insbesondere auch für die jeweiligen Koordinatensysteme). Die Schnittstelle der Klasse *SingleUnit* bietet hierzu Funktionen, die das Ein- und Ausordnen von *Movern* und *Places* übernehmen [3].

---

[1] vgl. Kapitel 13.2

[2] vgl. Kapitel 11.1

[3] Die Behandlung von Plätzen wird dabei an das enthaltene *PlaceList*-Objekt “durchgereicht”.

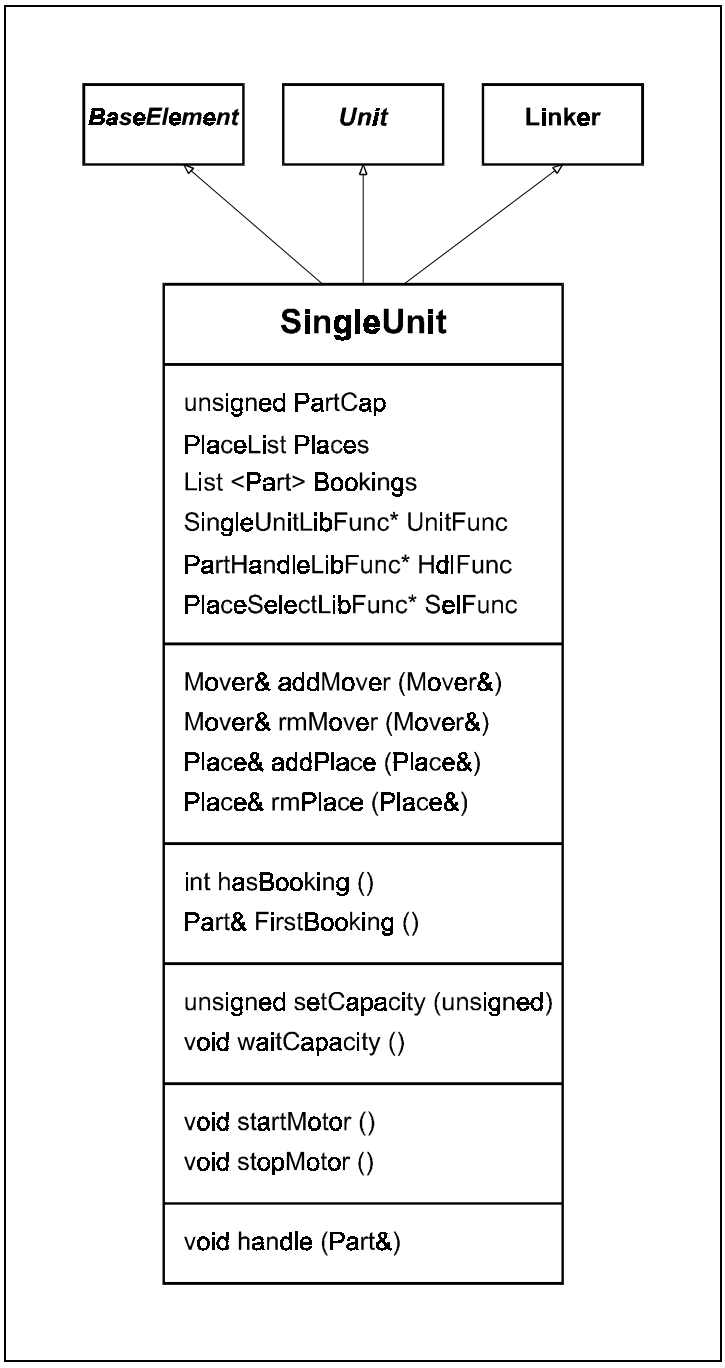

Bild 50: Vereinfachte Struktur einer *SingleUnit*

Die Teile, die über eine *SingleUnit* bewegt werden sollen, müssen bei dieser zunächst angemeldet werden [1]. Alle angemeldeten Teile werden in einer Liste (*Bookings*) verwaltet, auf die mit einer Reihe von Funktionen zugegriffen werden kann. So ist es möglich, zu prüfen, ob Anmeldungen vorliegen (*hasBooking ()*) oder auf das Eintreffen einer Anmeldung zu warten (*FirstBooking ()*) und den aktuellen Thread bis dahin zu unterbrechen [2].

SingleUnits haben eine Kapazität, die abgefragt und verändert (*setCapacity ()*) werden kann. Sie legt fest, wieviele Teile maximal gleichzeitig auf der Komponente behandelt werden können [3]. Die Funktion *waitCapacity ()* unterbricht den aktuellen Thread, bis auf der Komponente freie Kapazität vorhanden ist.

---

[1] Diese Anmeldung nimmt das Teil selbst vor, wenn es von der materialflußtechnisch vorangehenden Komponente abgegeben werden soll (vgl. Kapitel 14).

[2] Beispiele für den Zugriff auf die Anmeldungen einer *SingleUnit* zeigen die in Kapitel 19.2.2 vorgestellten Steuerungsfunktionen.

[3] Bei Fördertechnikkomponenten, die nur *PointPlaces* (vgl. Kapitel 13.1) als Plätze enthalten, kann die Kapazität höchstens gleich der Anzahl der Plätze sein.

Um Störungen von Fördertechnikkomponenten nachbilden zu können, bietet die Klasse *SingleUnit* Funktionen, die durch Nutzung der entsprechenden Funktionalität der enthaltenen Platzliste alle auf der Komponente im Gange befindliche Bewegungen unterbrechen (*stopMotor()*) bzw. unterbrochene Bewegungen fortsetzen (*startMotor ()*).

Weiter enthalten *SingleUnits* je (einen Zeiger auf) eine Steuerungs- (*UnitFunc*), eine Teilebehandlungs- (*HdlFunc*) und eine Platzwahlfunktion (*SelFunc*). Diese Funktionen ermöglichen die modell- bzw. aufgabenabhängige Anpassung des Verhaltens von Fördertechnikkomponenten [1]. Die Klasse *SingleUnit* stellt ergänzend dazu (in Bild 50 nicht dargestellte) Operationen bereit, mit denen diese Funktionen ausgewechselt werden können. In diesem Zusammenhang sei schließlich noch die Funktion *handle ()* erwähnt. Sie wird typischerweise aus der Steuerungsfunktion der jeweiligen *SingleUnit* aufgerufen und veranlaßt die (einmalige) Ausführung der Teilebehandlungsfunktion zur Bewegung eines Teils über die Komponente in einem (parallelen) Thread.

Zusammenfassend kann festgestellt werden, daß das vorgestellte Konzept die geometrisch richtige Abbildung (fast) beliebiger Fördertechnikkomponenten ermöglicht. Die zu Anfang dieses Kapitels problematisierte Notwendigkeit, sich bei der Abbildung einer Komponente auf einen bestimmten Typ (bzw. eine Klasse) und damit auf eine festgelegte Struktur und ein vordefiniertes Verhalten festlegen zu müssen, besteht hier nicht. Alle Komponenten sind Instanzen einer Klasse und damit strukturell und in ihrem Verhalten gleich oder (wegen eventuell unterschiedlicher Bewegungsapparate) wenigstens sehr ähnlich.

Anstatt Entscheidungen von grundlegender Tragweite treffen zu müssen sind hier nur Adaptionen eines Grundgerüsts erforderlich: Für jede Komponente sind lediglich ein oder mehrere passende Plätze zuzuordnen und der Bewegungsapparat realitätsentsprechend zusammenzusetzen. Da (wie in Kapitel 14 noch dargestellt wird) auch zur Abbildung von Systemgrenzen, Lagern und Bearbeitungsstationen keine besonderen Komponenten erforderlich sind, ist das Auftreten von Folgeproblemen, die aus unglücklichen Typfestlegungen beim Modellentwurf resultieren, hier implizit ausgeschlossen.

Das vorgestellte Konzept hat allerdings den Nachteil, daß “gebrauchsfertige” Komponenten zum Aufbau von Modellen nicht direkt zur Verfügung stehen, sondern jeweils erst zusammengesetzt werden müssen. Dies kann jedoch durch Möglichkeiten zum Kopieren von Komponenten aus einem Modell in ein anderes aufgefangen werden, wie sie beispielsweise in dem in Kapitel 21 vorgestellten Konzept für ein Simulationswerkzeug gegeben sind.

---

[1] Sie werden (wegen des thematischen Zusammenhangs) in den Kapiteln 19.2.1 bis 19.2.3 ausführlich behandelt.

13.4 Anwendungsbeispiele

Um Einsatz des dargestellten Modellierungskonzepts zu veranschaulichen und die vorangegangenen, eher abstrakten Erläuterungen zu den Konzeptbestandteilen transparenter zu machen, sollen an dieser Stelle einige Beispiele für den Aufbau von Abbildungen realer Fördertechnikkomponenten vorgestellt werden.

Wie zuvor erläutert, ist jede Komponente eine Instanz der Klasse *SingleUnit*, die implizit immer eine Platzliste enthält. Da Teile nur über Plätze bewegt werden können, erfordert die Abbildung einer realen Komponente immer die Zuordnung mindestens eines Platzes. Wird hierfür ein *PointPlace* verwendet, so zentriert die entstandene Komponente Teile beim Aufnehmen auf dem Platz, übergeht alle Transport- und Bewegungsanforderungen und gibt das Teil bei einer entsprechenden Anforderung wieder ab. So könnte z.B. eine Maschine abgebildet werden.

Von diesem wohl einfachsten Fall ausgehend werden nachfolgend zunehmend komplexere Komponenten entworfen. Die farbigen Hinterlegungen in den Strukturdarstellungen deuten dabei jeweils eine zweckmäßige Zuordnung von Konturbestandteilen zu den verschiedenen Koordinatensystemen (bzw. *GeoObjects*) der Komponenten an.

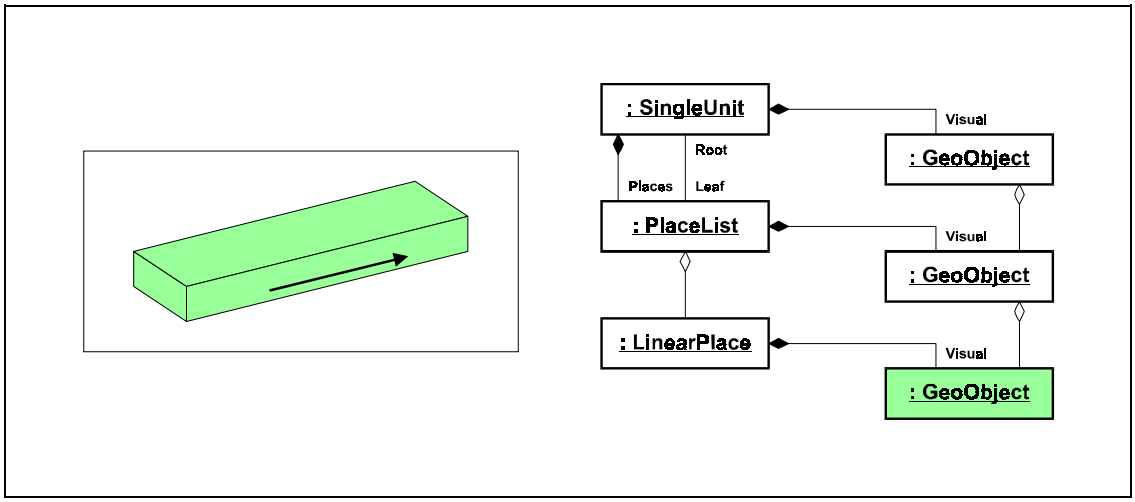

Bild 51: Struktur der Abbildung eines Linearförderers

13.4.1 Linearförderer

Die Modellierung erfordert lediglich die Verwendung eines *LinearPlaces* anstelle des *PointPlaces* aus dem Grundfall. Bild 51 zeigt die resultierende Struktur der Komponente. Diese wird Teile beim Aufnehmen soweit auf den Platz fördern, bis ihre Kontur vollständig

innerhalb der des Platzes ist. Transportanforderungen fördern die Teile über den Platz, während Bewegungsanforderungen übergangen werden. Bei einer entsprechenden Anforderung werden die Teile wieder abgegeben.

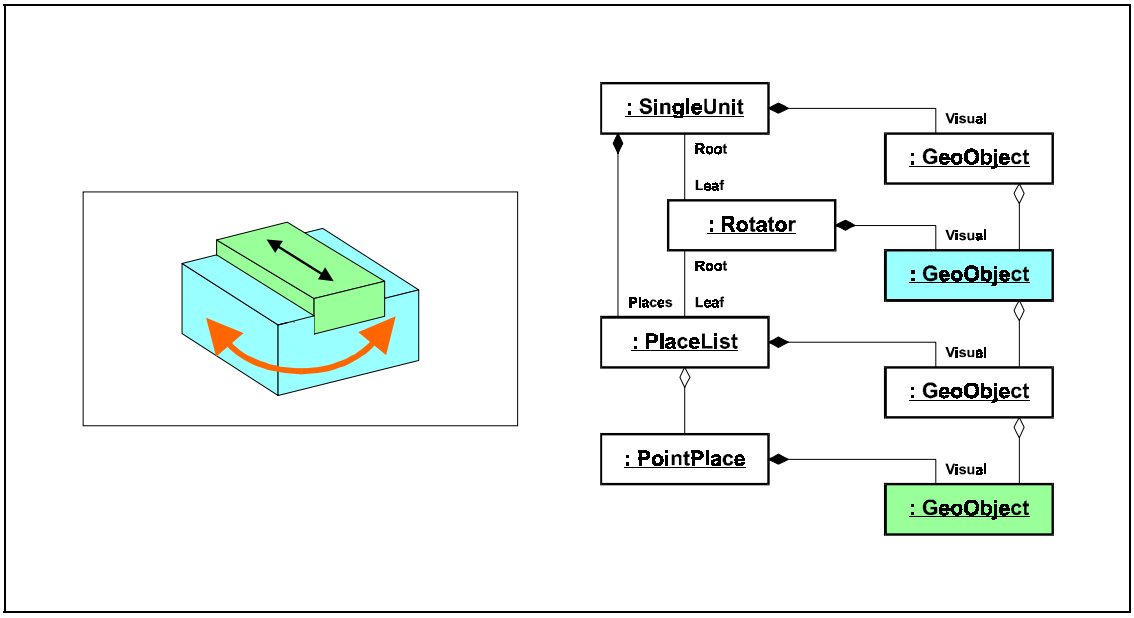

Bild 52: Struktur der Abbildung eines Drehtischs

13.4.2 Drehtisch

Die Modellierung entspricht wieder im wesentlichen dem Grundfall. Es ist jedoch zusätzlich ein *Rotator* erforderlich, der zwischen Fundament und Platzliste eingeordnet wird. Bild 52 zeigt die resultierende Struktur der Komponente. Diese wird Teile beim Aufnehmen auf dem Platz zentrieren. Transportanforderungen werden übergangen, während Bewegungsanforderungen in Drehungen des Tischs (bzw. des *Rotators*) resultieren. Bei einer entsprechenden Anforderung werden die Teile wieder abgegeben.

13.4.3 Verteilfahrzeug

Die Modellierung entspricht im wesentlichen dem Drehtisch. Es wird jedoch anstelle des *Rotators* ein *Translator* verwendet. Bild 53 zeigt die resultierende Struktur der Komponente. Das Verhalten entspricht dem des Drehtischs mit der Ausnahme, daß Bewegungsanforderungen (natürlich) nicht zu Drehungen sondern zu Verschiebungen des Fahrzeugs (bzw. des *Translators*) führen.

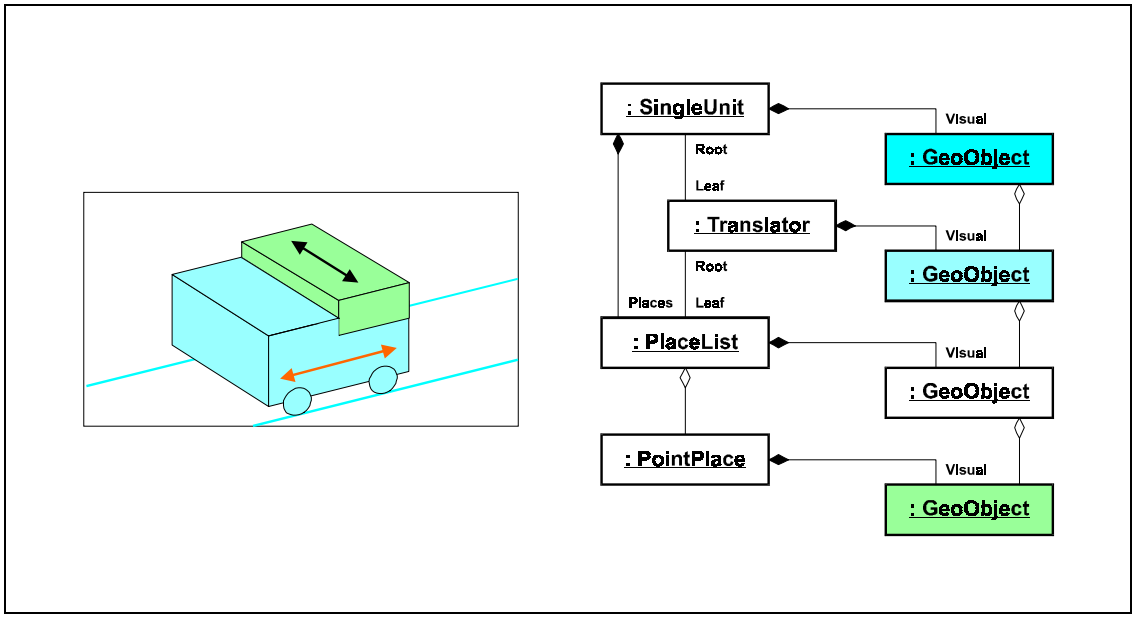

Bild 53: Struktur der Abbildung eines Verteilfahrzeugs

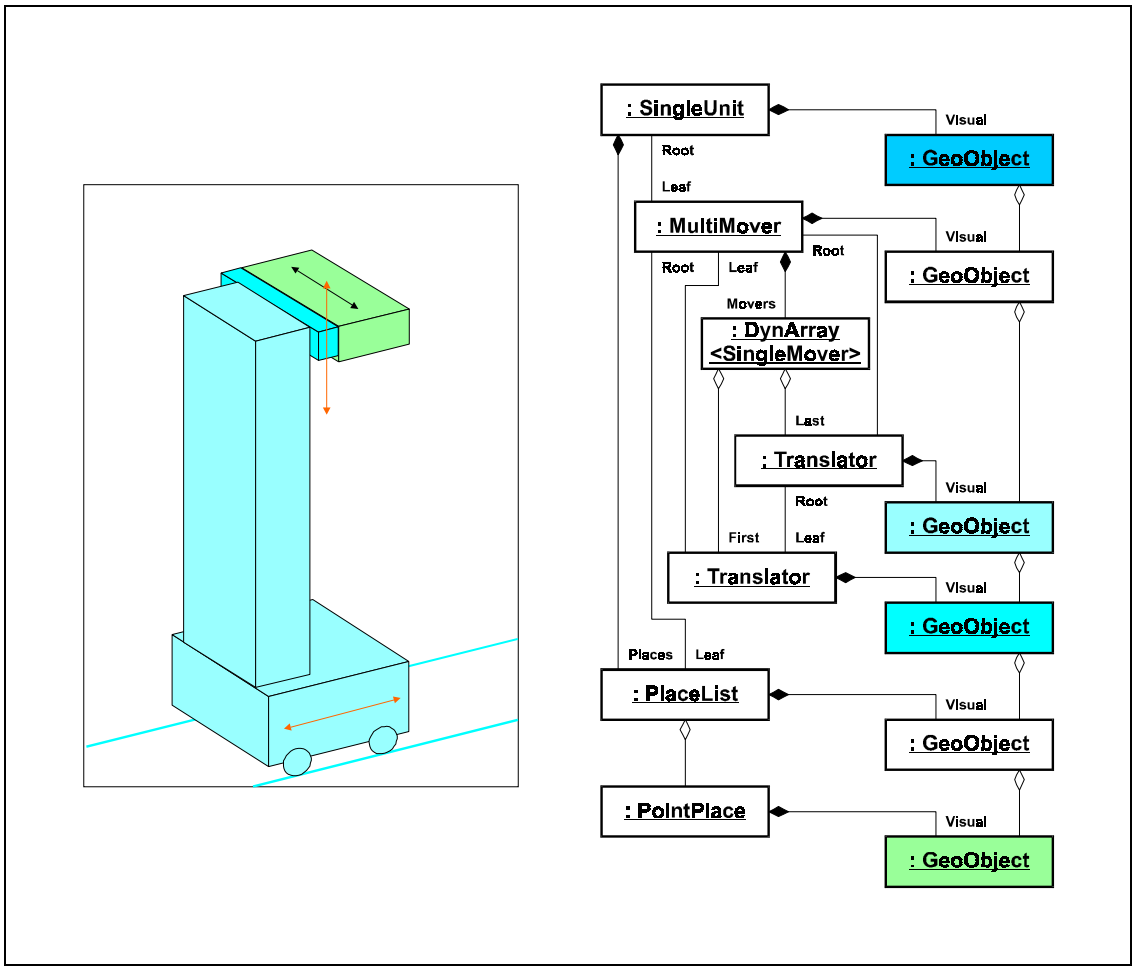

Bild 54: Struktur der Abbildung eines Regalbediengeräts

13.4.4 Regalbediengerät

Die Modellierung entspricht im wesentlichen dem Verteilfahrzeug. Es muß jedoch ein weiterer *Translator* für die Vertikalbewegung eingebaut werden. Der einfache Ansatz, diesen zweiten *Translator* direkt einzufügen, würde dazu führen, daß die beiden Achsen des Regalbediengeräts jeweils (mit insgesamt erhöhtem Zeitbedarf) nacheinander verschoben würden. Da in der Realität beide Achsenbewegungen zeitgleich erfolgen, erfordert die wirklichkeitsgetreue Abbildung einen *MultiMover*, der zwischen Fundament und Platzliste eingeordnet wird und für die zeitgleiche Achsenbewegung sorgt. In diesen werden dann die beiden *Translators* eingefügt. Bild 54 zeigt die resultierende Struktur der Komponente. Das Verhalten entspricht trotz des zusätzlichen *Movers* für die Vertikalbewegung vollständig dem des Verteilfahrzeugs.

13.5 Verkettungen und Wegesuche

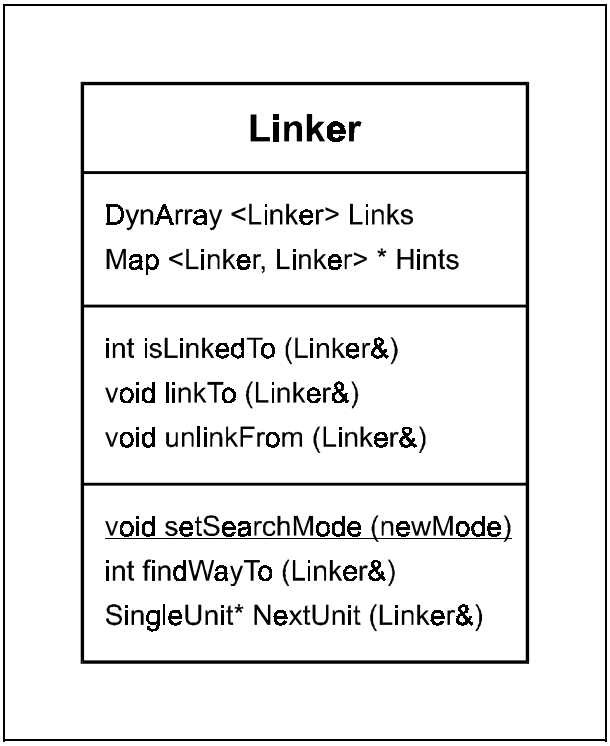

Bild 55: Struktur von *Linker*-Objekten

In Anlagen bzw. Modellen existieren Fördertechnikkomponenten nicht isoliert. Sie sind vielmehr mit benachbarten Komponenten materialflußtechnisch verkettet. Durch solche Ein- und Ausgangsbeziehungen entstehen Transportstrecken und insgesamt eine Materialflußstruktur. Zur Verwaltung von Verkettungen und zur Lösung der darauf aufsetzenden Aufgaben der Wegesuche dient im hier dargestellten Konzept (wie in Kapitel 13.3 bereits erwähnt) die Klasse *Linker*, die im folgenden vorgestellt wird. Da die Klasse *SingleUnit* von der Klasse *Linker* erbt, ist jede Fördertechnikkomponente auch ein *Linker*-Objekt, so daß sie mit anderen Komponenten verkettet werden kann.

Linker verwalten die hier Links genannten Verkettungen “ihrer” Komponente in einer Liste. Durch Funktionen in der Schnittstelle der Klasse können neue Links eingerichtet (*linkTo ()*)

und bestehende gelöscht werden (*unlinkFrom ()*). Außerdem kann getestet werden, ob eine Komponente mit einer bestimmten anderen verkettet ist (*isLinkedTo ()*).

Die ein Materialflußsystem durchlaufenden Teile haben stets ein Ziel, d.h. eine bestimmte Komponente, zu der sie transportiert werden sollen. Dieses Ziel kann z.B. ein Lagerplatz oder die nächste Bearbeitungsstation sein. Haben Teile ihr Ziel erreicht, erhalten sie in der Regel ein neues Ziel. Um Teile dorthin transportieren zu können, muß ein Weg vom Standort des Teils zum Ziel gefunden werden. In diesem Sinne ist ein Weg eine Folge von Komponenten, über die das Teil gefördert werden muß und deren letzte das Ziel ist.

Zur Festlegung solcher Wege kommen zwei Ansätze in Frage. Zum einen können sie im Zuge des Modellentwurfs festgelegt und mit der (übrigen) Modellbeschreibung gespeichert werden und zum anderen können sie während der Experimentdurchführung vom Simulationswerkzeug dynamisch ermittelt werden.

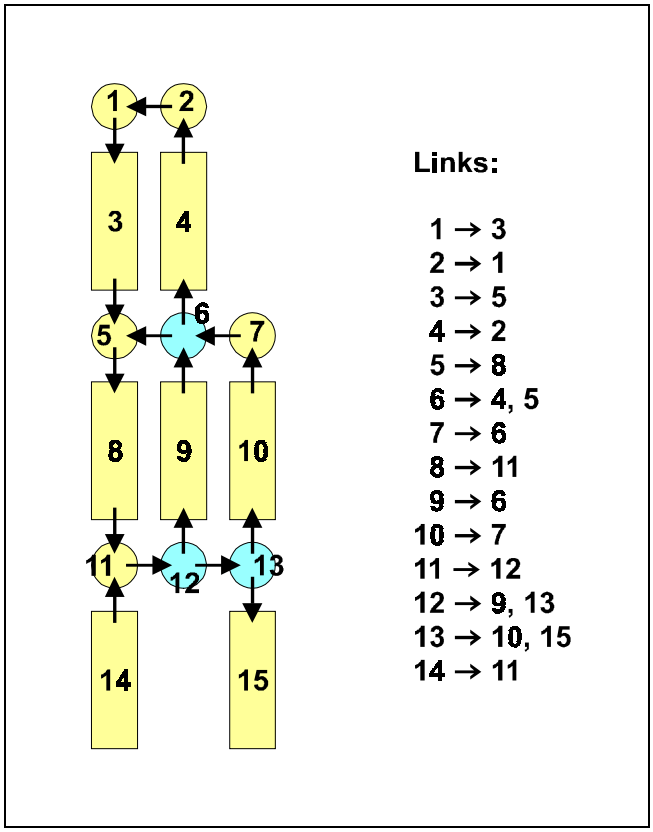

Bild 56: Materialflußsystem mit Verkettungen

Der erste Ansatz kann vom Simulationswerkzeug nur unzureichend unterstützt werden. Es muß bei der automatischen Erzeugung von Wegen auf andere Teile der Modellbeschreibung zurückgreifen. Wege, deren Notwendigkeit sich daraus nicht unmittelbar ergibt, wird es nicht erzeugen. Dies gilt z.B. für Wege zum Rücktransport von Prüfplätzen zu (vorgelagerten) Bearbeitungsstationen (bzw. ganz allgemein bei Sprüngen in der Abarbeitung von Arbeitsvorgangsfolgen) und bei "Umleitungen", die durch Notfallstrategien erfolgen.

Der einzige Vorteil der Wegefestlegung im Zuge des Modellentwurfs ist die dabei bestehende Möglichkeit, bestimmte Wege leichter erzwingen zu können. Durch die Vorgabe von Pseudozielen als Zwischenstation kann dies allerdings auch bei dynamischer Wegesuche erreicht werden. Im Rahmen des hier vorgestellten Konzepts werden daher Verfahren zur dynamischen Wegesuche eingesetzt, die nachfolgend erläutert werden. Grundlage der

Diskussion ist die beispielhafte Aufgabe, in dem in Bild 56 schematisch dargestellten verketteten Materialflußsystem einen Weg von der Komponente 14 zur Komponente 3 zu suchen [1].

Die eingesetzten Verfahren bauen auf die Graphentheorie [2] auf. Aus dieser Sicht sind Materialflußsysteme Graphen, in denen die Komponenten (bzw. *Linker*-Objekte) die Knoten und die Verkettungen die Kanten sind. Jede Komponente ist mit nur wenigen anderen verkettet, der Graph ist also "licht" [3].

Da es oft (wie auch im Beispiel) mehrere mögliche Wege gibt, kann die Aufgabe dahingehend erweitert und präzisiert werden, daß nicht (nur) irgendein, sondern der bestmögliche Weg zu suchen ist. Die Suche nach Wegen in Graphen ist eine bekannte und wohluntersuchte Aufgabenstellung für deren Lösung ausgereifte Algorithmen bekannt sind [4]. Hier wurden zwei Verfahren implementiert.

Das erste Verfahren liefert als besten Weg denjenigen, der über die wenigsten Komponenten führt (dies entspricht der Suche in einem ungewichteten Graphen) und adaptiert dazu einen Algorithmus von Moore [5]. Er benutzt eine Liste von (Teil-) Wegen und erfordert in jedem Knoten eine Marke, die gesetzt wird, wenn er in einen Weg eingeordnet wird [6]. Zur Vorbereitung wird die Marke des Startknotens gesetzt und ein Weg in die Liste eingetragen, der nur den Startknoten enthält. Der Algorithmus arbeitet dann wie folgt:

- Solange noch Wege in der Liste sind
    - Greife den letzten Knoten des ersten Weges in der Liste.
    - Für alle Ausgänge dieses Knotens
        - Wenn der Ausgang das gesuchte Ziel ist,
          dann breche den Algorithmus ab.
        - Wenn die Marke des Ausgangs gesetzt ist,
          dann setze mit dem nächsten Ausgang fort.
        - Setze die Marke des Ausgangs.

---

[1] Im folgenden wird auf die ständige Wiederholung des Wortes Komponente verzichtet und nur noch "von 14 nach 3" geschrieben.

[2] vgl. z.B. Perl: *Graphentheorie: Grundlagen und Anwendungen*

[3] Dies ist sicherlich charakteristisch für die Verkettungen in Materialflußsystemen.

[4] Ein Überblick gibt z.B. Rose: *Kritische Analyse von Routensuchverfahren*

[5] Moore: *The Shortest Path Through a Maze*

[6] *Linker*-Objekte enthalten hierfür ein entsprechendes Feld.

- Kopiere den ersten Weg in der Liste
  und hänge den Ausgang an diesen neuen Weg an.
- Hänge den neuen Weg an die Wegeliste an.
- Lösche den ersten Weg aus der Wegeliste.

Wenn die Wegeliste nach Beendigung des Algorithmus nicht leer ist, dann ist der erste Weg darin der beste. Endet der Algorithmus dagegen mit einer leeren Wegeliste, dann existiert kein Weg vom gegebenen Start- zum Zielknoten.

Im Beispielfall werden also nacheinander folgende Schritte ausgeführt [1]:

- Die Vorbereitung erzeugt einen Weg mit dem Startknoten 14.
  Beim Start ist die Wegeliste also { [14] }.

- Schritt 1 bearbeitet Knoten 14. Er kopiert den ersten Weg in der Liste, hängt an die Kopie den (einzigen) Ausgang von 14 an und löscht den ersten Weg.
  Die Wegeliste ist also nun { [14, 11] }.

- Schritt 2 bearbeitet Knoten 11. Da er nur einen Ausgang hat, entsteht auch nur ein neuer Weg durch Anhängen von 12 an den ersten Weg. Dieser wird dann gelöscht.
  Die Wegeliste ist danach { [14, 11, 12] }.

- Schritt 3 bearbeitet Knoten 12. Es werden zwei neue Wege durch Anhängen der Ausgänge von 12 an Kopien des ersten Weges erzeugt, der dann gelöscht wird.
  Die Wegeliste ist jetzt { [14, 11, 12, 9], [14, 11, 12, 13] }.

- Schritt 4 ist so einfach wie Schritt 2, da der bearbeitete Knoten 9 wie schon 11 nur einen Ausgang hat. So entsteht auch hier nur ein neuer Weg.
  Die Wegeliste ist dann { [14, 11, 12, 13], [14, 11, 12, 9, 6] }.

- Schritt 5 bearbeitet Knoten 13. Es entstehen zwei neue Wege durch Anhängen von 10 bzw. 15 an den ersten Weg, der schließlich gelöscht wird.
  Die Wegeliste ist { [14, 11, 12, 9, 6], [14, 11, 12, 13, 10], [14, 11, 12, 13, 15] }.

- Schritt 6 erzeugt mit den Ausgängen von Knoten 6 wiederum zwei neue Wege.
  Die Wegeliste ist danach { [14, 11, 12, 13, 10], [14, 11, 12, 13, 15], [14, 11, 12, 9, 6, 4], [14, 11, 12, 9, 6, 5] }.

---

[1] Es wird hier unterstellt, daß die Ausgänge in der Reihenfolge absteigender Numerierung getestet werden. Der Algorithmus selbst ist reihenfolgeunabhängig, was natürlich auch erforderlich ist.

- Schritt 7 bearbeitet Knoten 10 und ist so einfach wie Schritt 4.
  Die neue Wegeliste ist { [14, 11, 12, 13, 15], [14, 11, 12, 9, 6, 4], [14, 11, 12, 9, 6, 5], [14, 11, 12, 13, 10, 7] }.

- Schritt 8 bearbeitet Knoten 15. Da dieser keine Ausgänge hat, entstehen keine neuen Wege, es wird lediglich der erste gelöscht.
  Die Wegeliste ist anschließend { [14, 11, 12, 9, 6, 4], [14, 11, 12, 9, 6, 5], [14, 11, 12, 13, 10, 7] }.

- Die Schritte 9 und 10 sind wiederum so einfach wie Schritt 4.
  Die Wegeliste ist nach Schritt 10 { [14, 11, 12, 13, 10, 7], [14, 11, 12, 9, 6, 4, 2], [14, 11, 12, 9, 6, 5, 8] }.

- Schritt 11 bearbeitet Knoten 7. Da dessen einziger Ausgang 6 bereits als benutzt markiert ist, da er schon (in Schritt 6) erreicht wurde, wird kein neuer Weg erzeugt sondern nur der erste gelöscht.
  Die Wegeliste verkürzt sich zu { [14, 11, 12, 9, 6, 4, 2], [14, 11, 12, 9, 6, 5, 8] }.

- Schritt 12 bearbeitet Knoten 2 und ist ebenso einfach wie Schritt 4.
  Die Wegeliste ist danach { [14, 11, 12, 9, 6, 5, 8], [14, 11, 12, 9, 6, 4, 2, 1] }.

- Schritt 13 bearbeitet Knoten 8, dessen einziger Ausgang 11 bereits (in Schritt 2) erreicht wurde. Wie in Schritt 11 wird also lediglich der erste Weg gelöscht.
  Die Wegeliste verkürzt sich also zu { [14, 12, 9, 6, 4, 2, 1] }.

- Schritt 14 bearbeitet Knoten 1. Da dessen (einziger) Ausgang 3 das gesuchte Ziel ist, bricht der Algorithmus ohne Veränderung der Wegeliste ab.

Das Beispiel zeigt, daß der Algorithmus mögliche Fehler durch (längere) parallele Wege (Schritt 11), Umschlüsse [1] (Schritt 13) und Sackgassen (Schritt 8) sicher vermeidet und die betroffenen Wege nicht weiterverfolgt.

Wegen der Markierung wird jeder Knoten höchstens einmal bearbeitet, die Zeitkomplexität des Verfahrens (bzw. der Suchaufwand) ist also direkt proportional zur Anzahl der Knoten und damit nach dem Erkenntnisstand der Graphentheorie optimal [2]. Am Ende dieses Kapitels wird im übrigen noch gezeigt, wie der Suchaufwand durch Berücksichtigung bereits gefundener Wege von anderen Startknoten zum gewünschten Ziel weiter reduziert werden kann.

---

[1] Im Vokabular der Graphentheorie wird dies als “Kreis” bezeichnet.

[2] vgl. Rose: *Kritische Analyse von Routensuchverfahren*

Das zweite konzipierte Suchverfahren liefert als besten Weg denjenigen mit der geringsten Länge (dies entspricht der Suche in einem gewichteten Graphen) und adaptiert dazu einen Algorithmus von Dijkstra [1], der den oben beschriebenen von Moore geringfügig variiert. Als Länge eines Weges wird die Summe der Mittelpunktsabstände der darin enthaltenen Knoten bzw. Fördertechnikkomponenten verwendet. Da jede Komponente eine Instanz der Klasse *SingleUnit* ist und damit ein (als *GeoObject* abgebildetes) Koordinatensystem hat [2], können die ihre Mittelpunkte durch einfachen Funktionsaufruf ermittelt werden.

In Abänderung des ersten Verfahrens wird erstens die Länge jedes Weges in diesem gespeichert. Sie wird jeweils als Länge des Weges aus dem er kopiert wurde zuzüglich des Mittelpunktsabstands des bearbeiteten und des neu angefügten Knotens berechnet [3]. Die zweite Änderung besteht darin, daß neu erzeugte Wege nicht mehr einfach ans Ende der Wegeliste angehängt werden, sondern in diese so eingeordnet werden, daß die Elemente der Wegeliste zu jedem Zeitpunkt nach aufsteigender Länge geordnet sind.

Im übrigen arbeitet der Algorithmus wie der des ersten Verfahrens, so daß an dieser Stelle auf eine ausführlichere Darstellung verzichtet wird. Zeitkomplexität (bzw. Suchaufwand) sind auch bei diesem Verfahren im wesentlichen proportional zur Anzahl der Knoten. Wegen der erforderlichen Sortierung der Wegeliste ist der Aufwand jedoch etwas größer als beim ersten Verfahren. Für die gegebene Aufgabenstellung ist das Verfahren nach dem Erkenntnisstand der Graphentheorie dennoch optimal [4].

Von den beiden vorgestellten Verfahren wird jeweils eines genutzt. Mit der Funktion *setSearchMode ()* kann zwischen den Verfahren umgeschaltet werden. Bei jeder Umschaltung werden alle zuvor gefundenen Wege gelöscht, da sie nach einer Änderung des Bewertungsverfahrens u.U. nicht mehr optimal sind. Die eigentliche Suche nach einem Weg führt die Funktion *findWayTo (...)* aus. Sie sucht einen Weg vom angesprochenen *Linker*-Objekt (d.h. von der aufrufenden Komponente) zur (als Parameter) übergebenen Zielkomponente. Die Funktion *NextUnit ()* liefert jeweils (einen Zeiger auf) die vom angesprochenen *Linker* aus nächste Komponente auf dem Weg zum übergebenen Ziel.

Als weiterer wichtiger Punkt im Zusammenhang mit der Wegesuche ist festzulegen, wie (über die Beantwortung der aktuellen Suchanfrage hinaus) mit einmal gefundenen Wegen umgegangen werden soll. Hierzu können drei Vorüberlegungen angestellt werden. Erstens

---

[1] Dijkstra: *A note on two problems in connexion with graphs*

[2] vgl. Kapitel 13.3

[3] Als Länge des bei der Vorbereitung erzeugten Weges wird 0 gesetzt.

[4] vgl. Rose: *Kritische Analyse von Routensuchverfahren*

ist es sicher ineffizient, keinerlei Informationen über einmal gefundene Wege zu speichern, da in einer Anlage typischerweise immer wieder dieselben Wege benutzt werden. Zweitens führen verschiedene Wege häufig über gleiche Teilwege [1] und drittens finden sich oft verkettete Abschnitte ohne Verzweigungen, so daß darin befindliche Teile immer zunächst bis zur nächsten Verzweigung gefördert werden müssen bevor sie überhaupt unterschiedliche (Teil-) Wege nehmen können [2].

Die Betrachtung des Beispielsystems unter dem letztgenannten Aspekt zeigt, daß darin nur bei drei der fünfzehn Komponenten die Möglichkeit unterschiedlichen Weitertransports besteht. Diese drei (die in Bild 56 blau eingefärbt sind) kennzeichnet, daß nur sie mit mehr als einer (Ausgangs-) Komponente verkettet sind [3]. Offensichtlich sind nur diese Verzweigungsstellen bei der Speicherung von Informationen über Wege von Interesse.

Dies erinnert an eine Autobahnfahrt z.B. von Kassel nach Hamburg. Auch dabei ist die Vorbeifahrt an (u.a.) Bockenem oder Evendorf kaum von Belang. Wichtig sind vielmehr die Verzweigungsstellen (Autobahndreiecke und -kreuze), wo jeweils die richtige Fortsetzungsstrecke zu wählen ist. Dort findet sich auch eine gute Lösung für die Speicherung der richtigen Menge von Wegeinformation: Es sind die dort aufgestellten Hinweisschilder.

Das in der Klasse *Linker* implementierte Konzept arbeitet genauso. Zur Markierung des Weges von (z.B.) 14 nach 3 werden an den auf dem Weg liegenden Verzweigungsstellen Hinweise angebracht, welche Komponente auf dem Weg zu einem bestimmten Ziel als nächstes zu benutzen ist. Im Beispiel wird also (u.a.) bei Komponente 12 der Hinweis “nach 3 über 9" angebracht. Diese Hinweise werden von *Linker*-Objekten in einer Map [4] verwaltet, die jedoch nur an Verzweigungsstellen angelegt wird. Die Eintragung von Hinweisen erfolgt automatisch im Zuge der Wegesuche.

---

[1] In dem in Bild 56 gezeigten Materialflußsystem stimmen beispielsweise die Wege von 14 nach 3 und von 14 nach 8 bis zur Komponente 6 (also etwa bis zur Hälfte) überein.

[2] In dem in Bild 56 gezeigten Materialflußsystem müssen beispielsweise alle auf 4 befindlichen Teile zunächst nach 5 gefördert werden, bevor die Möglichkeit besteht, daß sie auf unterschiedlichen Wegen weitertransportiert werden können.

[3] Nach den Projekterfahrungen des Autors ist dies trotz des Beispielcharakters des gezeigten Systems keine untypisch niedrige Quote.

[4] vgl. Kapitel 10.4

Hinweise können ungültig werden, wenn Verkettungen gelöscht werden. Im Beispielsystem führt das Löschen der Verkettung von 2 nach 1 dazu, daß 1 und 3 überhaupt nicht mehr erreichbar sind. Es müssen also an allen Verzweigungsstellen eventuell vorhandene Hinweise zu den Zielen 1 und 3 entfernt werden. Beim Löschen von Verkettungen werden daher alle dadurch ungültig gewordenen Hinweise automatisch entfernt.

Insgesamt arbeitet die Klasse *Linker* so, daß alle Wegeinformation (bzw. Hinweise) automatisch erzeugt und erforderlichenfalls auch wieder gelöscht werden. Da von außen auch keine Zugriffsmöglichkeiten auf die Hinweise gegeben sind, ist die Verwaltung (und die Existenz) der Wegeinformation für die Anwendung völlständig transparent.

Abschließend sei noch erwähnt, daß die Hinweise intern zur weiteren Verbesserung der Suchverfahren benutzt werden. Wenn bei der Bearbeitung eines Verzweigungsknotens festgestellt wird, daß bei diesem bereits ein Hinweis für das gesuchte Ziel eingetragen ist, dann wird im betreffenden Schritt statt eines Weges pro Ausgang nur noch ein einziger Weg mit der in dem Hinweis vermerkten Fortsetzungsstation erzeugt und damit die Anzahl der Suchschritte reduziert.

Zur Verdeutlichung diene der Fall, daß in dem in Bild 56 dargestellten System ein Weg von 8 nach 3 zu suchen ist, nachdem zuvor der Weg von 14 nach 3 gefunden und markiert wurde. Da somit bei 12 und bei 6 bereits Hinweise auf 3 existieren, verfolgt die Suche dort jeweils nur noch den richtigen Weg, ohne andere Ausgänge zu prüfen. Statt in 14 Schritten führt die Suche so in den minimal möglichen 8 Schritten zum Ziel.

14 Teile

Materialflußsysteme werden in der Regel entworfen und betrieben, um darin Teile beispielsweise lagern, bearbeiten oder sortieren zu können. Die durchlaufenden Teile haben also für Materialflußsysteme elementare Bedeutung. Dies gilt natürlich auch für Simulationsmodelle solcher Systeme.

Im folgenden wird der im Rahmen dieser Arbeit entworfene Ansatz zur Modellierung von Teilen erläutert. Die grundlegende Bedeutung von Teilen im hier dargestellten Gesamtkonzept wird jedoch im Zusammenhang mit der in Kapitel 19 diskutierten Programmierung von Ablaufsteuerungen für Modelle deutlich.

Teile werden hier als Instanzen der Klasse *Part* abgebildet. Die wesentlichen Elemente ihrer softwaretechnische Struktur zeigt Bild 57. Jedes *Part* enthält u.a. eine Instanz der Klasse *GeoObject* [1], die seine Geometrie beschreibt. Sie definiert ein Koordinatensystem für das Teil und ermöglicht die Zuordnung von Konturelementen für die Visualisierung.

Für die Erzeugung von Teilen stehen zwei Versionen der Funktion *create ()* zur Verfügung [2]. Beiden Versionen muß eine Fördertechnikkomponente übergeben werden, auf der das neue Teil erzeugt werden soll. Außerdem kann ein Konturelement angegeben werden. In den weiteren Parametern unterscheiden sich die beiden Versionen. Während eine die Angabe

---

[1] vgl. Kapitel 11.1

[2] Diese Lösung basiert auf dem in C++ möglichen *function overloading*.

eines Förderziels (s.u.) erlaubt, ermöglicht die andere, dem neuen Teil einen Arbeitsplan und einen als nächstes auszuführenden Vorgang daraus zuzuordnen (s.u.). Mit der Funktion *dispose ()* können Teile gelöscht werden.

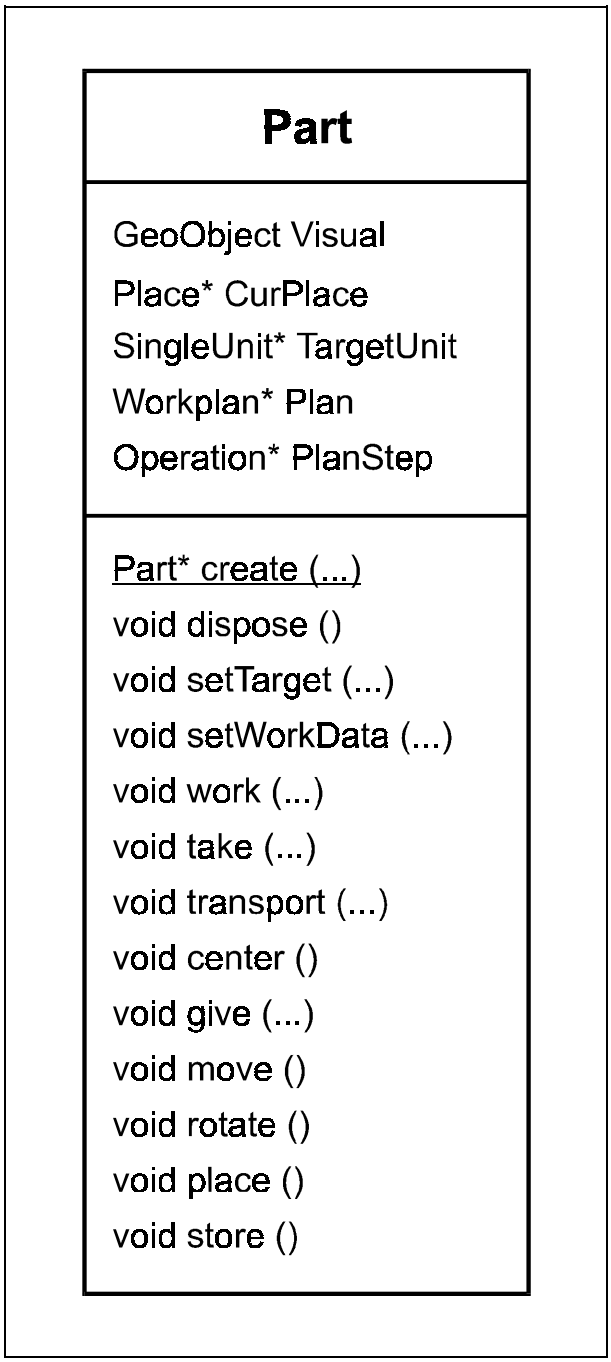

Bild 57: Struktur eines *Parts*

Jedes Teil hat (einen Zeiger auf) ein *Place*-Objekt [1] (*CurPlace)*, der angibt, wo im Modell sich das Teil gerade befindet. Dieser Standort ist also ein Platz auf einer Fördertechnikkomponente.

Weiter hat jedes *Part*-Objekt ein Ziel (*TargetUnit)*, zu dem es gefördert werden soll. Dieses Ziel ist (ein Zeiger auf) ein *SingleUnit*-Objekt [2], also eine Fördertechnikkomponente. Teile, die ihr Ziel erreicht haben, verlassen entweder das System oder erhalten ein neues Ziel, was hier jederzeit durch Aufruf der Funktion *setTarget ()* erreicht werden kann, der das neue Ziel zu übergeben ist.

Teile können einem Arbeitsplan [3] zugeordnet sein, der die an dem Teil auszuführenden Arbeitsvorgänge spezifiziert. Für diesem Fall enthalten *Parts* zwei Zeiger, die auf den Arbeitsplan bzw. auf den nächsten auszuführenden Vorgang daraus verweisen. Vorgang und Arbeitsplan können durch Aufruf der Funktion *setWorkData ()* geändert werden.

[1] vgl. Kapitel 13.1

[2] vgl. Kapitel 13.3

[3] vgl. Kapitel 15

Die Bearbeitung von Teilen wird mit der Funktion *work ()* nachgebildet. Bei *Parts*, die einem Arbeitsplan zugeordnet sind, wird als Vorgabe für die Bearbeitungszeit die Durchführungszeit des aktuell auszuführenden Vorgangs verwendet, wenn das Teil sich auf der Zielstation dieses Vorgangs befindet. In diesem Fall wird nach der Bearbeitung auch der auszuführende Vorgang weitergeschaltet. In allen anderen Fällen ist die Vorgabe für die Bearbeitungszeit 0. Die Vorgabe kann durch explizite Angabe einer Bearbeitungszeit beim Aufruf von *work ()* angepaßt werden.

Zur Bewegung von Teilen auf bzw. über ihren Standort stehen weitere Funktionen zur Verfügung. Das Aufnehmen wird durch *take ()* nachgebildet, *transport ()* bewegt das Teil über den Platz und *center ()* zentriert es auf diesem. Alle diese Funktionen berücksichtigen die technischen Daten des Platzes (z.B. seine Bahnkurve), mögliche Behinderungen durch voranlaufende Teile sowie weitere Randbedingungen (z.B.eventuelle Störungen). Die Funktion *give ()* veranlaßt die Übergabe des Teils an die auf seinem Weg zum Ziel folgende Fördertechnikkomponente. Insbesondere wird das Teil dabei bei dieser Komponente angemeldet [1]. Normalerweise kehrt *give ()* zum Aufrufer zurück, wenn das Teil von der nächsten Komponente bzw. dem nächsten Platz (vollständig) aufgenommen wurde. Durch Angabe eines entsprechenden Parameters kann jedoch die Rückkehr bereits bei Beginn der Übernahme erreicht werden.

Durch Aufruf der Funktionen *move ()*, *rotate ()* oder *place ()* wird die Bewegung des Standorts veranlaßt. Die Funktionen rufen dazu ihrerseits die Funktionen *moveTo ()*, *rotTo ()* bzw. *placeTo ()* des Platzes auf, auf dem sich das Teil befindet [2]. Wurde das Teil bereits auf diesen Platz aufgenommen (d.h. *take ()* bereits aufgerufen), so wird er (zur Vorbereitung der Abgabe) auf die nachfolgende Komponente zubewegt, anderenfalls wird er (zur Vorbereitung der Aufnahme) auf die Position des Teils zubewegt [3].

Schließlich besteht noch die Möglichkeit, das Teil durch Aufruf der Funktion *store ()* auf dem Standort einzulagern. Das Teil bleibt dann auf seiner aktuellen Position liegen bis es durch eine Bestellung wieder aktiviert wird [4].

Durch eine geeignete Aneinanderreihung der Funktionen der Klasse *Part* ergibt sich der genaue Ablauf bei der Benutzung einer Fördertechnikkomponente durch ein Teil. Bild 58 zeigt die

---

[1] Es wird dazu in deren *Booking*-Liste eingetragen (vgl. Kapitel 13.3).

[2] vgl. Kapitel 13.1

[3] In diesem Fall befindet sich das Teil physikalisch noch auf der vorangegangenen Komponente. Der angesprochen Platz wurde jedoch bereits für den Weitertransport des Teils ausgewählt.

[4] vgl. Kapitel 17

hierfür standardmäßig verwendete Funktion [1]. Sofern keine anlagenbezogenen Steuerungsanweisungen erforderlich sind, kann sie verwendet werden, um Teile über die unterschiedlichsten Komponenten wie z.B. Maschinen, Linearförderer, Drehtische oder Regalförderzeuge zu bewegen.

```
void defaultPartHandleFunc (Part& part, SingleUnit&) {
  part.place ();          // Standort zur Aufnahmeposition bewegen
  part.take ();           // Teil aufnehmen
  part.center ();         // Teil zentrieren
  part.work ();           // Teil gegebenenfalls nach Arbeitsplan bearbeiten
  part.transport ();      // Teil zum Ende des Platzes transportieren
  part.place ();          // Standort zur Abgabeposition bewegen
  part.give ();           // Teil abgeben
}
```

Bild 58: Standardfunktion zur Teilebehandlung auf einer Einzelkomponente

Bild 59 zeigt eine einfache Funktion für die Teilebehandlung auf Lagerkomponenten. Da Lagerplätze im allgemeinen unbewegliche Stellplätze ohne Antriebe für die Teilebewegung sind und dort auch keine Bearbeitungen stattfinden, kann hier auf Bewegungs- und Bearbeitungsoperationen verzichtet werden [2].

```
void samplePartStorageFunc (Part& part, SingleUnit&) {
  part.take ();           // Teil aufnehmen
  part.store ();          // Teil einlagern
                          //   (auslagern durch Bestellung von anderer Stelle)
  part.give ();           // Teil abgeben
}
```

Bild 59: Teilebehandlungsfunktion für Lagerkomponenten

[1] vgl. Kapitel 19

[2] Allerdings könnte ein Aufruf von *part.work ()* (vor *part.store ()*) u.U. zweckmäßig sein, weil sich damit auf einfachste Weise eine Mindestlagerzeit erzwingen ließe.

15 Arbeitspläne

Zu den Materialflußsystemen, die mit Hilfe von Simulationswerkzeugen experimentell untersucht werden sollen, zählen auch produktionstechnische Anlagen, in denen Teile, Baugruppen und Endprodukte gefertigt, d.h. bearbeitet und montiert werden. In vielen solchen Anlagen erfolgt die Teilebearbeitung anhand von Arbeitsplänen.

Im hier vorgestellten Konzept werden Arbeitspläne als Instanzen der Klasse *Workplan* modelliert. Da sie unterscheidbar und identifizierbar sein müssen, ist die Klasse *Workplan* aus der Klasse *Identity* [1] abgeleitet. Dadurch erhält jedes *Workplan*-Objekt einen eindeutigen Namen als identifizierendes Merkmal.

Für die Simulation produktionstechnischer Anlagen besteht die Bedeutung von Arbeitsplänen im wesentlichen darin, daß sie eine Folge von Bearbeitungsvorgängen festlegen, die an allen Teilen durchzuführen sind, die nach einem bestimmten Arbeitsplan bearbeitet werden. Hier wurde eine Ererbungsbeziehung etabliert, so daß ein Arbeitsplan eine Folge von Beabeitungsvorgängen ist. Dazu ist die Klasse *Workplan* aus der Klasse *DynArray* <*Operation*> abgeleitet (die ihrerseits aus der generischen Klasse *DynArray* [2] instanziiert ist).

Bild 60 zeigt den softwaretechnischen Aufbau von *Workplan*-Objekten. Die Klasse erweitert die Funktionalität ihrer Basisklassen nur um Funktionen, die (einen Zeiger auf) den einem

---

[1] vgl. Kapitel 10.5

[2] Ein *DynArray* ist eine spezielle (speicheroptimierte) Implementierung einer Liste (vgl. Kapitel 10.4).

anzugebenden Bearbeitungsvorgang nachfolgenden (*NextOperation ()*) bzw. vorangehenden (*PrevOperation ()*) Vorgang sowie einen Vorgang mit einem angegebenen Namen (*findOperation ()*) zurückliefern.

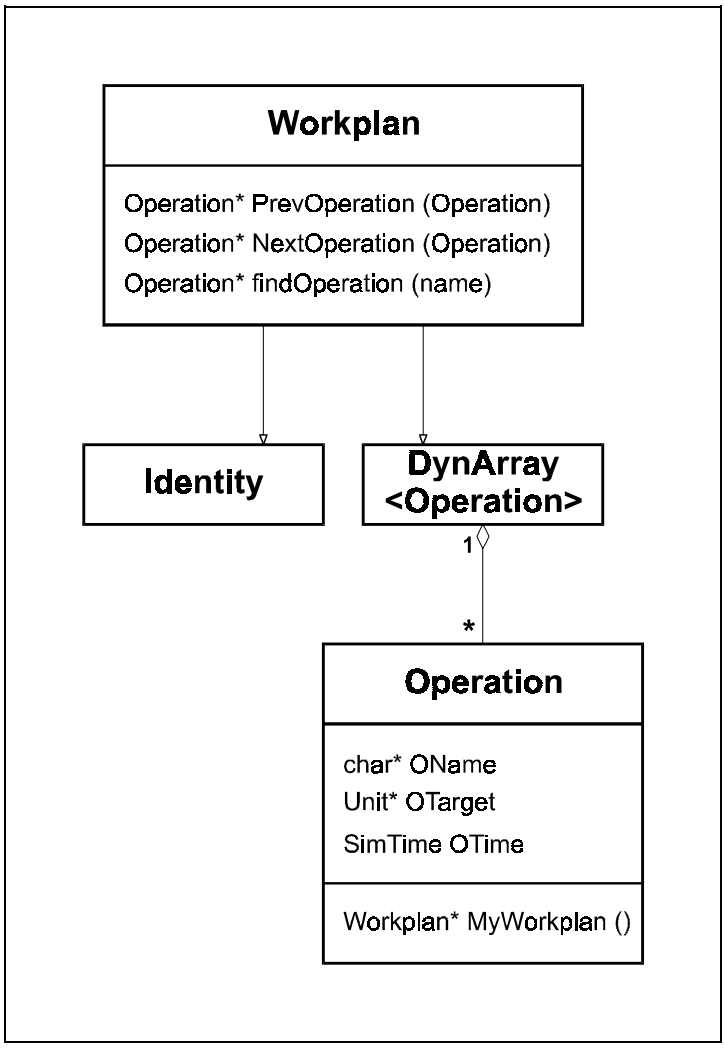

Bild 60: Struktur von *Workplan*- und *Operation*-Objekten

Die einzelnen Bearbeitungsvorgänge werden als Instanzen der Klasse *Operation* abgebildet, deren softwaretechnische Struktur ebenfalls in Bild 60 dargestellt ist. *Operation*-Objekte haben im wesentlichen einen Namen, eine Bearbeitungszeit (Vorgabezeit) und einen Bearbeitungsort, der eine Systemkomponente bezeichnet, auf der die Bearbeitung durchzuführen ist. Alle diese Eigenschaften können durch (in Bild 60 nicht dargestellte) Funktionen der Klasse *Operation* abgefragt und modifiziert werden. Zusätzlich kann durch Aufruf der Funktion *MyWorkplan ()* derjenige Arbeitsplan identifiziert werden, zu dem ein angesprochener Bearbeitungsvorgang gehört.

Während zu Name und Bearbeitungszeit von Bearbeitungsvorgängen sicherlich keine weiteren Erläuterungen erforderlich sind, bedarf es an dieser Stelle noch ergänzender Ausführungen zum Bearbeitungsort. Das in Kapitel 13 diskutierte Teilkonzept würde hierfür einzig die Verwendung von *SingleUnit*-Objekten erlauben. Dadurch müßte als Bearbeitungsort jeweils eine einzelne Systemkomponente, also beipielsweise eine ganz bestimmte Maschine festgelegt werden. Der Blick in die betriebliche Praxis zeigt jedoch, daß als Durchführungsort von Arbeitsvorgängen häufig Maschinengruppen oder Kostenstellen angegeben werden [1].

[1] vgl. z.B. Wiendahl: *Betriebsorganisation für Ingenieure*; S. 152 ff.

Dadurch bleibt die Auswahl einer konkreten Maschine für einen Auftrag (sinnvollerweise) der Kapazitätsplanung vorbehalten und kann bei entsprechender Planungsdurchführung bis in die Feindisposition innerhalb der Maschinengruppe verschoben werden.

Um diese Sachverhalte angemessen abbilden zu können, bedarf das Einheiten- bzw. Komponentenkonzept aus Kapitel 13 der Ergänzung. Im folgenden Kapitel wird es daher nochmals aufgegriffen und (u.a.) um Möglichkeiten zur Modellierung von Komponentengruppen erweitert.

16 Statische Systemkomponenten

In Kapitel 13 wurde ein Konzept zur Abbildung der Komponenten von Materialflußsystemen beschrieben. Die Instanzen der dort vorgestellten Klasse *SingleUnit* ermöglichen die Abbildung derjenigen Komponenten, die mit den das System durchlaufenden Teilen direkt in Berührung kommen.

Dieses Konzept wird im folgenden um zusätzliche Klassen zur Modellierung weiterer Typen von Systemkomponenten erweitert, mit denen beispielsweise Maschinengruppen und Steuerungsrechner geeignet abgebildet werden können. Die resultierende Klassenmenge wird abschließend in eine Hierarchie eingeordnet, in der die strukturellen Gemeinsamkeiten aller Systemkomponenten ausgedrückt werden.

16.1 Komponentengruppen

Die in Kapitel 15 dargestellten Eigenschaften von Arbeitsplänen erfordern, daß mit einem erweiterten Komponentenkonzept wenigstens zwei Sachverhalte abbildbar sein müssen. Zum einen müssen Maschinengruppen gebildet werden können und zum zweiten müssen solche Gruppen ebenso wie Einzelkomponenten z.B. als Bearbeitungsorte in Vorgängen verwendbar sein.

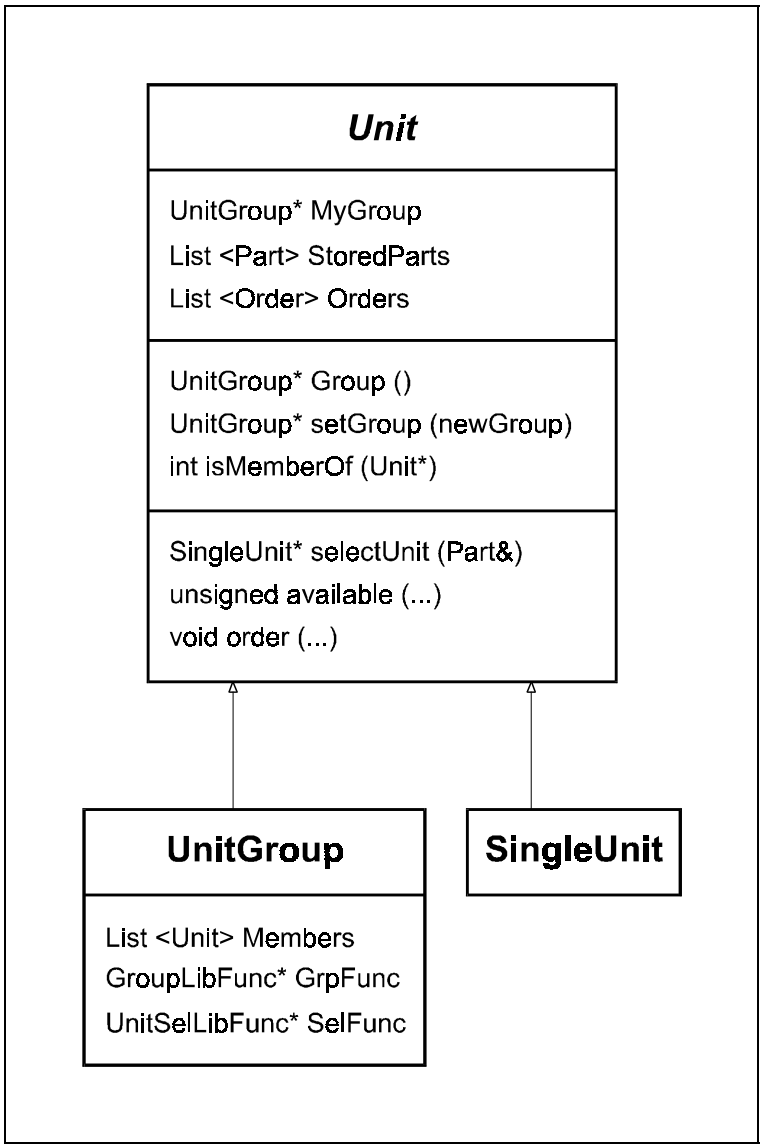

Bild 61: Struktur von *Unit*- und *UnitGroup*-Objekten

Im bisherigen Einheitenkonzept sind alle Komponenten einheitlich als Instanzen der Klasse *SingleUnit* abgebildet, so daß Maschinen von anderen Komponenten wie z.B. Förderern softwaretechnisch nicht unterscheidbar sind. Da die einheitliche Abbildung aus guten Gründen vorgesehen wurde [1], soll davon nicht abgewichen werden. Außerdem wird noch gezeigt, daß die Zusammenfassung von Komponenten zu Gruppen nicht nur zur Abbildung von Maschinengruppen bzw. Kostenstellen, sondern auch bei der Modellierung von Lagern zweckmäßig ist [2].

Daher können Gruppen hier beliebige Komponenten als Mitglieder enthalten [3]. Jede Gruppe wird als eine Instanz der Klasse *UnitGroup* abgebildet, deren softwaretechnische Struktur in Bild 61 dargestellt ist. Da als Typ der Member-Objekte (also der Gruppenmitglieder) nicht die Klasse *SingleUnit* sondern deren Basisklasse *Unit* (s.u.) verwendet wird, ist es sogar möglich, eine Gruppe zum Mitglied einer anderen zu machen und so rekursive Gruppenstrukturen aufzubauen.

---

[1] Diese Gründe wurden in Kapitel 13 dargelegt. Die de facto natürlich bestehenden Unterschiede in Funktion und Verhalten verschiedener Komponenten ergeben sich, wie dort ausgeführt, implizit aus dem strukturellen Aufbau und der Verwendung im Modell.

[2] Diese Thematik wird in Kapitel 17 behandelt.

[3] Die Verantwortung für die Sinnhaftigkeit einer Gruppe wird damit natürlich (mehr als bei anderen denkbaren Ansätzen) auf die Anwendung und ihre Benutzer und Benutzerinnen verschoben.

Weiter enthalten *UnitGroups* je (einen Zeiger auf) eine Gruppensteuerungs- (*GroupFunc*), und eine Komponentenwahlfunktion (*SelFunc*). Diese Funktionen ermöglichen die modell- bzw. aufgabenabhängige Anpassung des Verhaltens von Komponentengruppen [1]. Die Klasse *UnitGroup* stellt ergänzend dazu (in Bild 61 nicht dargestellte) Operationen bereit, mit denen diese Funktionen ausgewechselt werden können.

Die zweite eingangs genannte Anforderung, sowohl Einzelkomponenten wie auch Gruppen als Bearbeitungsorte in Vorgängen verwenden zu können, erfordert im Objektorientierten Design den Ausdruck dieser gemeinsamen Eigenschaft durch Ableitung der beiden Klassen *SingeUnit* und *UnitGroup* aus einer (gemeinsamen) Basisklasse. Dies ist hier die in Bild 61 ebenfalls dargestellte abstrakte Klasse *Unit*. Allgemein drücken die so etablierten Ererbungsbeziehungen aus, daß sowohl Einzelkomponenten (*SingleUnits*) als auch Komponentengruppen (*UnitGroups*) Einheiten (*Units*) von Materialflußsystemen sind.

Die angesprochene Anforderung kann damit durch die Verwendung eines (Zeigers auf ein) *Unit*-Objekt als Bearbeitungsort in der Klasse *Operation* realisiert werden, wie in Bild 60 [2] im Vorgriff auf die hier gemachten Ausführungen bereits zu sehen war.

Zur Durchführung eines Bearbeitungsvorgangs ist es erforderlich, aus einer als Bearbeitungsort angegebenen Maschinengruppe (einer *UnitGroup*) eine konkrete Einzelmaschine (eine *SingleUnit*) auszuwählen. Dies leistet die Funktion *selectUnit ()*. Immer wenn einem Teil ein neues Ziel zugewiesen wird (z.B. weil sein aktueller Bearbeitungsvorgang weitergeschaltet wurde), wird sie von der dazu benutzten Funktion *setTarget ()* [3] aufgerufen. Wenn das angesprochene Ziel eine *UnitGroup* ist, führt *selectUnit ()* deren Komponentenwahlfunktion (s.o.) aus, anderenfalls ist das Ziel eine *SingleUnit* und bleibt unverändert.

Die Bedeutung der von der Klasse *Unit* bereitgestellten Funktionen *available (...)* und *order(...)* sowie der Listen *StoredParts* und *Orders* wird in Kapitel 17 erläutert. Die übrigen in Bild 61 dargestellten Eigenschaften und Funktionen der Klassen *Unit* und *UnitGroup* dienen der Verwaltung der Gruppenstrukturen. Sie werden hier nicht weiter erläutert, da sie sicherlich selbsterklärend sind.

---

[1] Sie werden (wegen des thematischen Zusammenhangs) in den Kapiteln 19.2.6 bzw. 19.2.4 ausführlich behandelt.

[2] s. Seite 190

[3] vgl. Kapitel 14

16.2 Steuerungen

Nach den Projekterfahrungen des Autors [1] ist es häufig sinnvoll (oder wenigstens wünschenswert), in Simulationsmodellen in der untersuchten Anlage vorhandene oder mit ihr in Beziehung stehende Steuerungsrechner mit abzubilden. Beispielsweise wurden im Projekt "Grundlagen der rechnerintegrierten Fabrik" die Speicherprogrammierbaren Steuerungen der Anlage explizit modelliert. Im Projekt "Motorenendmontage ..." wurde die zyklisch durchgeführte Reihenfolgeplanung (die in der Realität von übergeordneten Rechnern übernommen wird) in das entstandene Modell integriert.

Die explizite Abbildung von Rechnern oder Rechnerfunktionalitäten in Softwaremodellen von Materialflußsystemen mag auf den ersten Blick überraschen, im Grunde ist sie aber nur konsequent, da die Modelle ja die Realität abbilden. Wie in Kapitel 5.4 ausgeführt wurde [2], übernehmen vernetzte Rechner in modernen Produktionsanlagen eine solche Fülle von Aufgaben, daß sie aus den Anlagen und Unternehmen nicht mehr wegzudenken sind. Für nicht der Produktion dienende Materialflußsysteme gilt dies in gleicher Weise.

Im hier vorgestellten Komponentenkonzept können Rechner bzw. Steuerungen als Instanzen der Klasse *Control* abgebildet werden. Die (einzige) wesentliche Eigenschaft von *Control*-Objekten ist, daß sie einen Thread ausführen und kontrollieren können. Dieser Thread kann eine beliebige anlagen- bzw. aufgabenbezogene Funktion abarbeiten. *Controls* enthalten hierfür je (einen Zeiger auf) eine Steuerungskontrollfunktion [3]. Ergänzend dazu steht eine Operation bereit, mit der diese Funktion ausgewechselt werden kann.

16.3 Gebäude

Die Instanzen der Klasse *Building* repräsentieren im hier vorgestellten Konzept Gebäude oder Etagen. Sie ermöglichen die geometrische Strukturierung beliebiger Komponenten von Anlagenmodellen dadurch, daß diese in Gebäude eingeordnet werden können.

---

[1] vgl. Anhang Projekte, S. 264

[2] s.a. Bild 10 auf Seite 55

[3] Sie wird (wegen des thematischen Zusammenhangs) in Kapitel 19.2.7 ausführlich behandelt.

Die Klasse *Building* ist aus der Klasse *Component* abgeleitet. *Building*-Objekte enthalten damit jeweils ein *GeoObject* [1]. Zum einen können ihnen damit Konturelemente zugeordnet werden und zum anderen definieren sie so ein Koordinatensystem, in das die Koordinatensysteme der enthaltenen Komponenten eingeordnet werden. Zusätzlich verwalten *Buildings* die enthaltenen Komponenten in einer Liste.

Im Zusammenhang von Modellen erfüllen *Buildings* im wesentlichen zwei Funktionen. Einerseits ermöglichen sie die direkte Übernahme der Koordinaten von (z.B.) Förderern oder Maschinen in Fällen, in denen diese als gebäude- oder hallenbezogene Werte vorgegeben sind. Die Hallen oder Gebäude können dann als *Buildings* abgebildet werden, in die die Förderer und Maschinen eingeordnet werden [2].

Andererseits ermöglichen *Buildings* die einfache Manipulation von Modellen zur Erzeugung übersichtlicher 2D-Ansichten und -Animationen [3] aus 3D-Modellen. Beispielsweise können Anlagen, die sich über mehrere Etagen erstrecken nur unzulänglich in (übersichtlichen) 2D-Draufsichten dargestellt werden. In solchen Fällen erlaubt es die Modellierung der Etagen als *Buildings*, diese schnell und einfach nebeneinander (statt übereinander) anzuordnen (wobei alle jeweils enthaltenen Komponenten wegen der Koordinatensystemhierarchie mitwandern), so daß 2D-Draufsichten anschaulicher werden.

16.4 Komponenten

Unter dem Begriff Komponenten werden in diesem Konzept alle Elemente von Materialfluß-systemen zusammengefaßt, die jeweils über die gesamte Experimentlaufzeit vorhanden sind (permanente Einheiten) und die beim einem Durchgang durch die Anlage sichtbar sind oder wenigstens sein können. Zu den Komponenten zählen damit Einzelkomponenten (*SingleUnits*, z.B. Förderer und Maschinen), Komponentengruppen (*UnitGroups*, die durch Anordnung oder Beschilderung sichtbar werden können), Steuerungsrechner (*Controls*) und Gebäude (*Buildings*). Dagegen sind z.B. Teile und Arbeitspläne keine Komponenten.

---

[1] vgl. Kapitel 11.1

[2] Hallen- bzw. gebäudebezogene Koordinatenangaben für die Anordnung von Anlagenkomponenten sind beispielsweise beim Einsatz von Layoutplanungssystemen für ganze Standorte üblich.

[3] Trotz des unverkennbaren Trends zur 3D-Visualisierung (vgl. Kapitel 8.1) sind 2D-Animationen und -Ansichten heute noch weit verbreitet. Da die Abstraktion von 3D nach 2D die Übersichtlichkeit steigern kann, bleiben sie wohl auch in (der überschaubaren) Zukunft gefragt.

Die Gemeinsamkeiten aller Komponenten werden durch Ableitung ihrer Klassen aus der Basisklasse *Component* ausgedrückt. Es entsteht die in Bild 62 dargestellten Klassenhierarchie *Components*. Dabei wurden die Klassen *SingleUnit* und *UnitGroup* nicht direkt, sondern durch Ableitung ihrer Basisklasse *Unit* aus der Klasse *Component* abgeleitet.

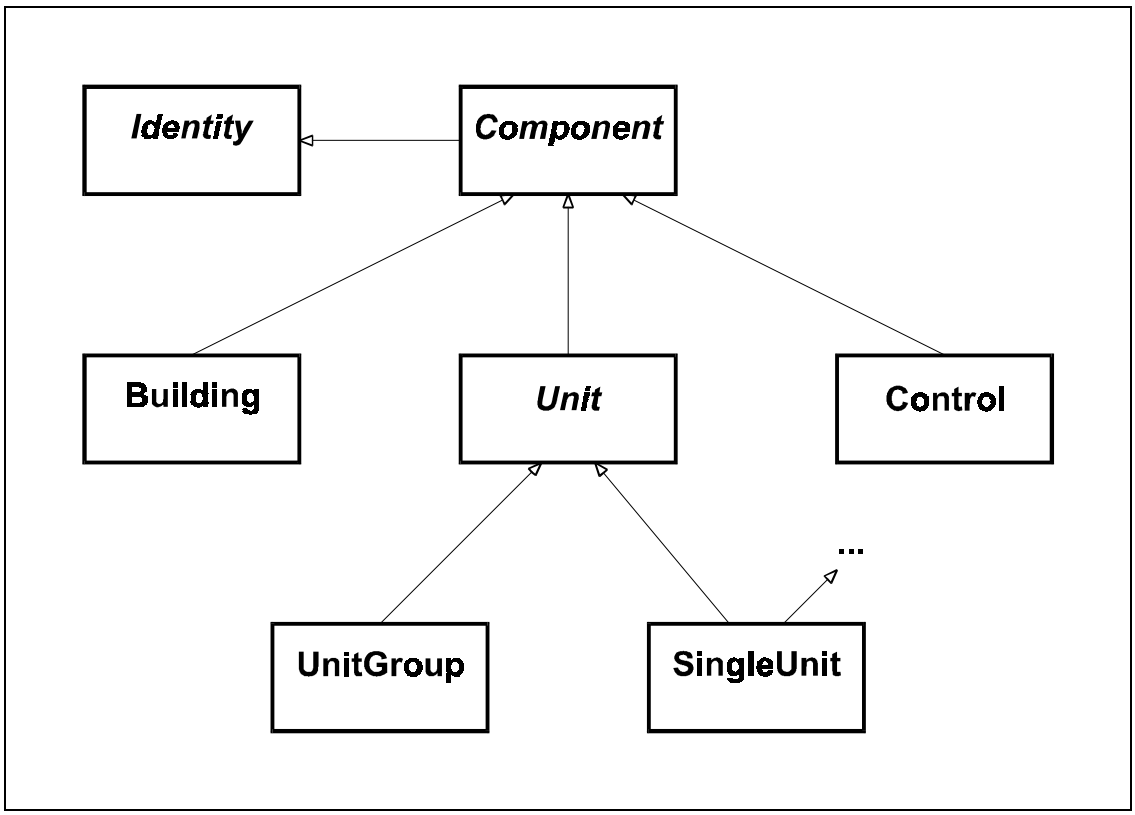

Bild 62: Klassenhierarchie *Components*

Die Ererbungsbeziehungen implizieren, daß z.B. ein Gebäude (ein *Building*-Objekt) gleichzeitig eine Komponente (ein *Component*-Objekt) ist. Die Eigenschaft möglicher Sichtbarkeit wird dadurch abgebildet, daß jede *Component*-Instanz ein *GeoObject* enthält und so ein Koordinatensystem mit der Möglichkeit der Zuordnung von Konturelementen definiert.

Da Komponenten unterscheidbar und eindeutig identifizierbar sein müssen, ist die Klasse *Component* aus der Klasse *Identity* [1] abgeleitet. Als wichtige Eigenschaft von Komponenten ist noch die Möglichkeit zu erwähnen, sie in Gebäude (*Buildings*) einordnen zu können [2]. Die Klasse *Component* stellt Funktionen für die Verwaltung solcher Einordnungsbeziehungen bereit.

---

[1] vgl. Kapitel 10.5

[2] vgl. Kapitel 16.3

17 Lager und Puffer

Viele Materialflußsysteme sind reine Lagersysteme. Aber auch in Anlagen, in denen die Lagerung von Teilen nicht der Hauptzweck des Systems ist, wie z.B. in Produktionsanlagen, sind Lager- und Pufferplätze ein häufig anzutreffendes und oft notwendiges Element. Dementsprechend ist die Bestimmung erforderlicher Lager- bzw. Pufferkapazitäten oft eine wichtige Teilaufgabe bei der Durchführung von Simulationsstudien. Die Beantwortung derartiger Fragestellungen erfordert offensichtlich die Abbildung der Lager bzw. Puffer und ihrer Bedienung in Simulationsmodellen. Im folgenden wird dargestellt, wie dies im Rahmen des hier diskutierten Gesamtkonzepts geschehen kann.

Die Modellierung von Lagern oder Puffern umfaßt zwei Aspekte. Zum einen ist die Struktur, also die eigentlichen Lager- bzw. Pufferplätze, abzubilden und zum anderen ist ihre Bedienung bzw. Steuerung einzurichten.

Die Abbildung der Struktur geschieht durch Erzeugen von Anlagenkomponenten (d.h. *SingleUnits* [1]), denen die vorgesehenen Plätze zugeordnet werden. Beispielsweise könnte ein Palettenwechsler vor einer Maschine als eine *SingleUnit* mit zwei gegenüberliegenden Plätzen und einem *Rotator* angelegt werden. Die so aufgebaute Komponente nimmt dann den Palettentausch durch je eine Drehung um 180 Grad vor.

[1] vgl. Kapitel 13

Die Modellierung eines Hochregallagers ist dagegen aufwendiger. Ein typisches Beispiel ist ein Lager mit mehreren Gassen, in denen je ein ihnen fest zugeordneten Regalbediengerät (RBG) operiert. Die RBG können als *SingleUnits* wie in Kapitel 13.4.4 gezeigt abgebildet werden. Alle links und rechts an einer Gasse angeordneten Regalplätze werden zweckmäßigerweise je einer *SingleUnit* zugeordnet [1]. Die materialflußtechnischen Beziehungen werden eingerichtet, indem jeweils der zu einer Gasse gehörende Lagerbaustein mit dem in dieser Gasse fahrenden RBG über Kreuz verkettet wird [2]. Die Zusammenfassung des Lagers zu einer Einheit geschieht durch Zuordnung aller Lagerbausteine zu einer Komponentengruppe [3]. Diese kann dann z.B. als Förderziel einzulagernder Teile angegeben werden und repräsentiert so das Lager insgesamt nach außen.

Die Bedienung bzw. Steuerung so modellierter Puffer und Lager ist in diesem Konzept auf Ein- und Auslagervorgänge verteilt.

Einlagerungen beginnen damit, daß einem Teil ein Puffer oder Lager als Förderziel zugewiesen wird [4]. Ist dieses Förderziel eine Komponentengruppe (wie im Beispiel des Hochregallagers), so wird dabei deren Funktion *selectUnit ()* [5] zur Festlegung einer Einzelkomponente als Fahrziel aufgerufen (im Hochregallagerbeispiel wird dadurch ein Lagerbaustein und damit die anzusteuernde Gasse ausgewählt). Nachdem das Teil dorthin gefördert wurde, wird die Platzwahlfunktion [6] der Einzelkomponente zur Festlegung des Lagerplatzes aufgerufen. Die Einlagerung wird durch Aufruf der Funktion *store ()* aus der Steuerungs- oder Teilebearbeitungsfunktion der Einzelkomponente [7] abgeschlossen.

Das Teil liegt nun auf dem ausgewählten Platz, der kontrollierende Thread ist unterbrochen

---

[1] Diese *SingleUnits* enthalten jeweils nur die entsprechenden Regalplätze. Da sich die Regalplätze nicht bewegen (können), sind keine Bewegungselemente (*Mover*) erforderlich.

[2] Durch diese Verkettungen (s.a. Kapitel 13.5) wird ausgedrückt, daß jedes Teil, das auf einem Regalplatz ein- bzw. ausgelagert werden soll, von dem in dessen Gasse operierenden RBG dorthin bzw. von dort weg transportiert werden muß. Aus der festen Zuordnung der RBG zu je einer Gasse folgt auch die Notwendigkeit, auch die Lagerplätze gassenweise zusammenzufassen. Die (technisch durchaus mögliche) Zuordnung aller Regalplätze zu nur einer *SingleUnit* und deren Verkettung mit allen RBG würde implizit bedeuten, daß jedes RBG jeden Regalplatz bedienen kann.

[3] vgl. Kapitel 16.1

[4] Dies kann entweder dadurch geschehen, daß das Lager (bzw. der Puffer) der Bearbeitungsort des aktuellen Vorgangs im Arbeitsplan des Teils ist (vgl. Kapitel 15) oder dadurch, daß das Lager (bzw. der Puffer) durch Aufruf von *setTarget ()* explizit als Förderziel gesetzt wird (vgl. Kapitel 14).

[5] vgl. Kapitel 16.1 und 19.2.4

[6] vgl. Kapitel 19.2.3

[7] vgl. Kapitel 19.2.2 bzw. 19.2.1

und wird erst fortgesetzt, wenn das Teil wieder ausgelagert wird. Zusätzlich ist jedes eingelagerte Teil in eine durch die Liste *StoredParts* [1] implementierte Verwaltungsstruktur eingeordnet. In der Regel ist dies die Liste derjenigen Einzelkomponente (*SingleUnit*), auf der das Teil eingelagert ist. Wenn diese Komponente jedoch Mitglied einer Komponentengruppe (*UnitGroup*) ist, wird die Liste der Gruppe verwendet.

Bei Auslagerungen wird auf diese Verwaltungsstruktur zurückgegriffen. Sie werden durch Bestellungen ausgelöst, die durch Aufruf einer der *order ()*-Funktionen einer Systemkomponente (d.h. einer Gruppe oder einer Einzelkomponente) erzeugt werden [2]. Diese Funktionen prüfen, ob die Bestellung von auf der Komponente eingelagerten Teilen erfüllt wird (s.u.). In diesem Fall werden die betreffenden Teile ausgelagert, indem sie aus der Liste eingelagerter Teile entfernt werden und die Fortsetzung der sie kontrollierenden Threads veranlaßt wird. Bestellungen, die nicht (vollständig) ausgeführt werden können, werden in die Liste *Orders* der Komponente [3] eingeordnet und bleiben so erhalten [4], während augeführte Bestellungen gelöscht werden.

Zu jeder *order ()*-Funktion bietet die Klasse *Unit* eine passende *available ()*-Funktion. Statt Teile zu bestellen, d.h. auszulagern, stellen diese Funktionen lediglich jeweils fest, wieviele Teile für eine gleichartige Bestellung zum Zeitpunkt des Aufrufs verfügbar sind und geben diesen Wert zurück.

Intern werden die Bestellungen als Instanzen einer der Klassen der Hierarchie *Orders*, also als Objekte abgebildet. Obwohl die Existenz dieser Objekte und ihrer Klassen für Anwendungen vollständig transparent ist, wird die Hierarchie im folgenden vorgestellt, da dabei die verschiedenen Möglichkeiten, Bestellungen anforderungsgerecht zuzuschneiden verdeutlicht werden können. Bild 63 zeigt die softwaretechnische Struktur der Klassen der Hierarchie. Mit jeder (instanziierbaren) Klasse der Hierarchie korrespondiert eine der verschiedenen *order ()*-Funktionen, die für *Units* aufgerufen werden können.

Die Klasse *Order* ist die (abstrakte) Basisklasse der Hierarchie. Jedes *Order*-Objekt enthält einen Zähler (*Count*), in dem die Anzahl (noch) zu bestellender Teile verwaltet wird. Die (in dieser Klasse rein virtuelle) Funktion *try ()* wird für jedes auf einer Systemkomponente eingelagerte Teil aufgerufen, um zu prüfen, ob es eine Bestellung erfüllt.

---

[1] vgl. Kapitel 16.1

[2] ebd.

[3] ebd.

[4] Konsequenterweise wird bei jeder Einlagerung geprüft, ob bereits eine “passende” Bestellung vorliegt. Ist dies der Fall, so wird das betroffene Teil unmittelbar wieder ausgelagert.

Die erste instanziierbare Klasse der Hierarchie ist die Klasse *SimpleOrder*. Ihre Instanzen repräsentieren unspezifizierte Bestellungen, die von allen eingelagerten Teilen erfüllt wer-den. Dies ermöglicht das einfache Auslagern (bzw. Weiterbewegen) von Teilen in Puffern.

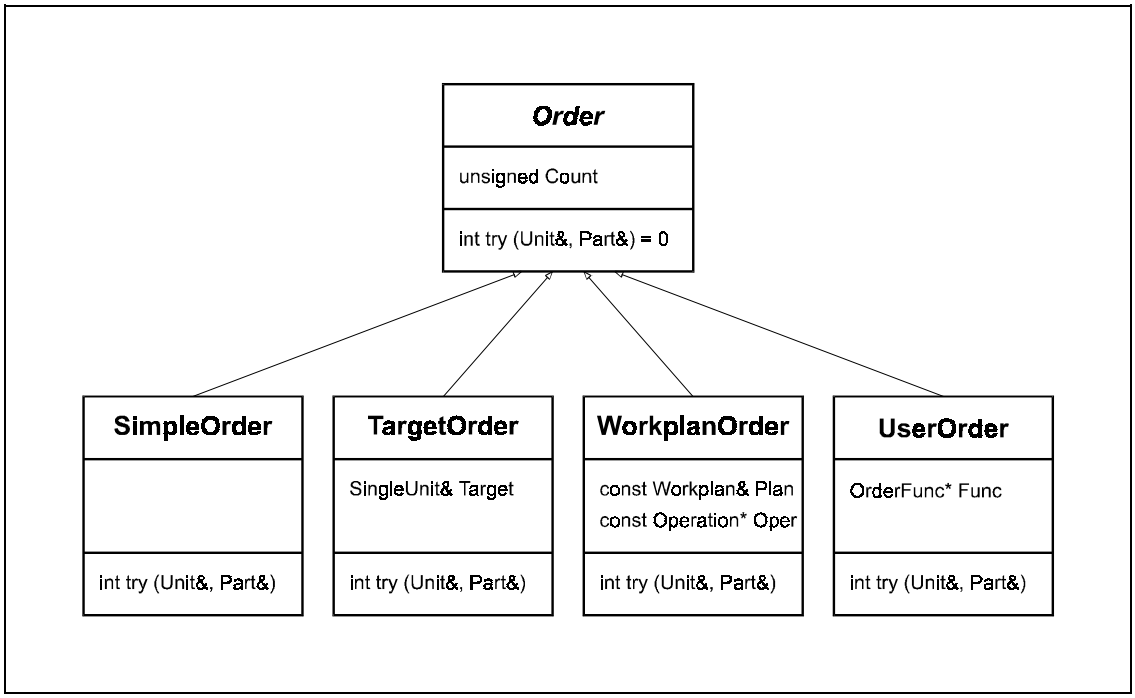

Bild 63: Struktur der Klassen der Hierarchie *Orders*

Die Instanzen der Klasse *TargetOrder* repräsentieren Bestellungen, die nur von Teilen erfüllt werden, die ein bestimmtes Förderziel haben. Daher ist bei der Erzeugung der Bestellung (eine Referenz auf) dieses Ziel, d.h. auf eine Einzelkomponente anzugeben.

Die Instanzen der Klasse *WorkplanOrder* repräsentieren Bestellungen, die nur von Teilen erfüllt werden, die nach einem bestimmten Arbeitsplan bearbeitet werden. Daher ist bei der Erzeugung der Bestellung (eine Referenz auf) diesen Arbeitsplan anzugeben. Zusätzlich kann ein (Zeiger auf einen) Vorgang angegeben werden. Ist dies der Fall, dann werden nur Teile ausgelagert, deren nächster Bearbeitungsvorgang dem angegebenen entspricht.

Die Instanzen der Klasse *UserOrder* repräsentieren Bestellungen, die nur von Teilen erfüllt werden, die anderen modellabhängigen Bedingungen genügen. Daher ist bei der Erzeugung der Bestellung der Name einer Bestellfunktion anzugeben, die die Prüfung durchführt [1].

[1] Sie werden (wegen des thematischen Zusammenhangs) in Kapitel 19.2.5 ausführlich behandelt.

18 Störungen und Ankunftsströme

In Modellen von Materialflußsystemen sind häufig Vorkommnisse abzubilden, die während der Durchführung von Experimenten wiederholt aber nicht regelmäßig auftreten. Zu diesen Ereignissen gehören z.B. der Eintritt von Teilen in das System (Ankunftsströme) und der Ausfall von Systemkomponenten (Störungen).

Das verbindende und kennzeichnende Merkmal derartiger Ereignisse ist ihr statistischer Charakter [1]. Die Dauer von Störungen und die Abstände zwischen aufeinanderfolgenden Störungen oder Systemeintritten von Teilen (Zwischenankunftszeiten) sind in der Regel nicht konstant, sondern genügen stochastischen Verteilungen, deren Art und Parameter beim Systementwurf vorzugeben sind.

Im folgenden wird ein Konzept zur Modellierung von Störungen und Ankunftsströmen vorgestellt. Diese werden darin als Objekte abgebildet, deren Klassen in der Hierarchie *Repeaters* angeordnet sind. Bild 64 zeigt die softwaretechnische Struktur der Klassen dieser Hierarchie.

Die Klasse *Repeater* ist die (abstrakte) Basisklasse der Hierarchie. Sie ist aus der Klasse *Identity* [2] abgeleitet, um verschiedene Störungen und Ankunftsströmen unterscheiden und eindeutig identifizieren zu können. Jede *Repeater*-Instanz ist ein aktives Objekt, dem ein Thread

---

[1] vgl. *VDI-Richtlinie 3633, Blatt 1*; S. 8

[2] vgl. Kapitel 10.5

(d.h. ein *Activity*-Objekt [1]) zugeordnet ist, den es über den Zeiger *Act* kontrolliert. Daneben enthält jeder *Repeater* (einen Zeiger auf) einen Zufallszahlenstrom (d.h. ein *DNG*-Objekt [2]). Die aus diesem Zahlenstrom gezogenen Werte bestimmen die Zeitabstände zwischen den Wiederholungen des abgebildeten Ereignisses. Ein *Repeater* (bzw. der ihm zugeordnete Thread) suspendiert sich für den gezogenen Zeitabstand und ruft bei seiner Reaktivierung durch den Zeitmechanismus die (in dieser Klasse rein) virtuelle Funktion *RepeatFunc ()* auf, in der die abgeleiteten Klassen die durch das abgebildete Ereignis bewirkten Systemveränderungen beschreiben. Dieser Zyklus wiederholt sich immer wieder.

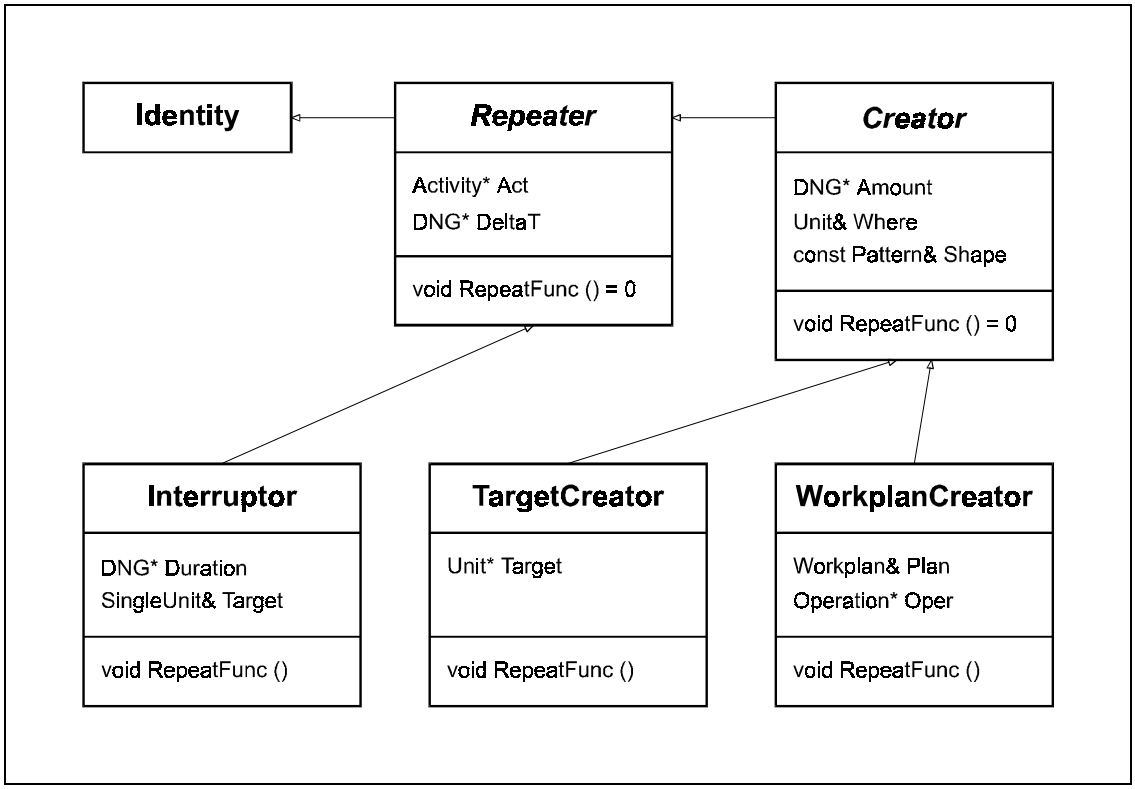

Bild 64: Struktur der Klassen der Hierarchie *Repeaters*

Die erste abgeleitetete Klasse der Hierarchie ist die Klasse *Interruptor*. Ihre Instanzen beschreiben Störungen von Einzelkomponenten des Systems. Neben einem (Zeiger auf einen) weiteren Zufallszahlenstrom, dessen Werte die jeweilige Stördauer bestimmen, enthalten sie eine Referenz auf die von den Störungen betroffene Einzelkomponente (d.h. auf eine Instanz der Klasse *SingleUnit* [3]). Die Wiederholfunktion *RepeatFunc ()* dieser Klasse startet jeweils einen weiteren Thread. Dieser unterbricht zunächst alle auf der zu störenden Komponente

[1] vgl. Kapitel 12.4

[2] vgl. Kapitel 10.3

[3] vgl. Kapitel 13.3

ablaufenden Aktivitäten, suspendiert sich selbst für die jeweilige Stördauer und veranlaßt schließlich die Fortsetzung der unterbrochenen Aktivitäten bevor er sich wieder beendet [1]. Zur Behandlung der Aktivitäten werden die Funktionen *stopMotor ()* bzw. *startMotor ()* der Klasse *SingleUnit* benutzt.

Ebenfalls aus der Klasse *Repeater* abgeleitet ist die wiederum abstrakte Klasse *Creator*. Sie wird als Basisklasse für die Abbildung von Ankunftsströmen genutzt. *Creator*-Objekte enthalten (einen Zeiger auf) einen weiteren Zufallszahlenstrom, dessen Werte die jeweils zu erzeugende Anzahl von Teilen angeben, eine Referenz auf die Systemkomponente (bzw. Komponentengruppe), auf der die Teile zu erzeugen sind sowie eine Referenz auf eine Instanz der Klasse *Pattern* [2], die den erzeugten Teilen als Kontur mitgegeben wird.

Die Klasse *TargetCreator* ist aus der Klasse *Creator* abgeleitet. Die Instanzen dieser Klasse erzeugen Teile, die keinen zugeordneten Arbeitsplan haben. Es ist jedoch möglich, den erzeugten Teilen unmittelbar eine Systemkomponente als Förderziel zuzuweisen, indem ein entsprechender Zeiger in den *TargetCreator*-Objekten bei deren Generierung gesetzt wird. Die Wiederholfunktion *RepeatFunc ()* dieser Klasse erzeugt die Teile durch Aufruf der passenden *create ()*-Funktion der Klasse *Part* [3].

Auch die Klasse *WorkplanCreator* als letzte Klasse der Hierarchie ist aus der Klasse *Creator* abgeleitet. Die Instanzen dieser Klasse erzeugen Teile mit zugeordnetem Arbeitsplan. Sie enthalten dazu eine Referenz auf eine Instanz der Klasse *Workplan* [4]. Es ist weiter möglich, den nächsten auszuführenden Bearbeitungsvorgang der erzeugten Teile festzulegen, indem ein entsprechender Zeiger in den *WorkplanCreator*-Objekten bei deren Generierung gesetzt wird. Die Wiederholfunktion *RepeatFunc ()* dieser Klasse erzeugt die Teile wiederum durch Aufruf der passenden *create ()*-Funktion der Klasse *Part*.

---

[1] Da die Abstände aufeinanderfolgender Störungen in der Praxis nicht als “lichte Weite” (störungsfreie Zeit) sondern als Abstände des Eintritts von Störungen (“Spitze-Spitze”) gegeben sind, lassen sich durch die Verwendung zusätzlicher Threads für die Störungsdurchführung Verfälschungen der Störabstände durch die Stördauern (bzw. entsprechende Korrekturrechnungen) auf einfache Weise vermeiden.

[2] vgl. Kapitel 11.1

[3] vgl. Kapitel 14

[4] vgl. Kapitel 15

19 Programmierung

In Kapitel 8.1.4 wurde gefordert, daß Simulationswerkzeuge die Möglichkeit bieten sollten, beliebige Steuerungsalgorithmen in Modelle integrieren und diese in verbreiteten Programmiersprachen und speziellen Notationen dokumentieren zu können. Um diese Anforderung zu erfüllen, müssen Simulationswerkzeuge über eine Programmierschnittstelle verfügen, die die aufgabenbezogene Anpassung der Abläufe in Modellen ermöglicht.

Das Benutzermodell von Simulationswerkzeugen impliziert, daß die (zunächst beim Anwender liegende) Kontrolle über den Programmablauf mit dem Start eines Experiments (überwiegend [1]) auf den Zeitmechanismus des Werkzeugs übergeht. Dieser steuert die chronologische Abarbeitung der im System ablaufenden Prozesse, indem er die Ausführung von Algorithmen bzw. Funktionen kontrolliert, die die Systemveränderungen beschreiben. Diese Funktionen, zu denen auch vom Benutzer hinzugefügte aufgabenbezogene Algorithmen gehören, werden also vom Zeitmechanimus “zurückgerufen”.

Aus dieser Sicht bedeutet aufgabenbezogene Ablaufanpassung bzw. Programmierung also die Bereitstellung einer Menge von (Rückruf-) Funktionen. Die Programmierschnittstelle eines Simulationswerkzeugs kann sich daher nicht auf die Bereitstellung von Möglichkeiten z.B. zur Abfrage und Änderung von Parametern u.ä. beschränken. Sie muß zusätzlich Situationen definieren, an die Funktionen “angehängt” werden können und dabei präzise festgelegen, wann diese aufgerufen werden.

---

[1] Die Einschränkung erfolgt, da normalerweise jederzeit Dialogeingriffe (oder wenigstens der Abbruch des Experiments) über die Bedienoberfläche des Werkzeugs möglich sind.

In diesem Kapitel wird die im Rahmen des hier diskutierten Gesamtkonzepts entworfene Programmierschnittstelle vorgestellt. Sowohl ihre technische Realisierung als auch die für die Anbindung von Funktionen gegebenen Ansatzpunkte werden erläutert.

Beim Umgang mit Programmierschnittstellen von Softwaresystemen gleich welcher Art besteht selbstverständlich auch die Möglichkeit, auf die Nutzung der Schnittstellen ganz zu verzichten. Gerade bei bausteinbasierten Simulationswerkzeugen wie dem hier konzipierten wird dieser Fall häufig vorkommen, da sie ja gerade darauf abzielen, den Aufwand bei der Modellerstellung durch die Bereitstellung komplexer und spezialisierter Modellelemente zu begrenzen [1].

In allen Fällen, in denen auf die Nutzung der Programmierschnittstelle eines Softwaresystems verzichtet wird, muß dieses ein Standardverhalten zeigen, das mindestens die Fortsetzung der Programmausführung ermöglicht. Für Simulationswerkzeuge heißt dies, daß immer wenigstens das “Anlaufenlassen” von Experimenten möglich sein muß [2].

Auch im hier vorgestellten Konzept werden Standardalgorithmen verwendet, wenn keine benutzerdefinierten vorhanden sind. Diese sind in der gleichen Weise wie benutzerdefinierte implementiert. Dadurch ist es weitgehend möglich, das vorgegebene Standardverhalten durch ein benutzerdefiniertes Standardverhalten zu ersetzen. Wo dies zutrifft, werden bei den Erläuterungen zur jeweiligen Anwendungssituation entsprechende Hinweise gegeben.

19.1 Technische Realisierung

Die hier vorgestellte Programmierschnittstelle ist für die Codierung modellspezifischer Steuerungsalgorithmen als Funktionen in der Programmiersprache C++ ausgelegt. Es werden Definitionsdateien in dieser Sprache (also Header- bzw. .h-Dateien) bereitgestellt, in denen Klassen, Objekte und Funktionen definiert sind, die die Manipulation der Abläufe im Modell unterstützen bzw. durchführen. Mit ihrer Hilfe ist es z.B. möglich, Parameter von Systemkomponenten, Teilen u.ä. abzufragen oder zu ändern und Aktionen wie die Bewegung von Systemkomponenten oder den Transport von Teilen auszulösen.

---

[1] vgl. Kapitel 7.5.2

[2] Diese Forderung ist bewußt moderat, da Standardsteuerungsalgorithmen nicht grundsätzlich ausschließen können, daß Ausnahmesituationen (wie z.B. Deadlocks) auftreten die dann dazu führen, daß das Experiment über das “Anlaufen” nicht hinauskommt.

Die zunächst als (C++-) Quellcode vorliegenden modellabhängigen Steuerungsfunktionen müssen dann in ausführbaren Maschinencode übersetzt (compiliert) und zu einer "shared library" [1] gebunden (gelinkt) werden. Dieser Weg wurde gewählt, da Techniken existieren, die es erlauben, shared libraries erst zur Laufzeit eines Programmes zu identifizieren, zu laden und dann zu benutzen [2]. Shared libraries müssen also zum Zeitpunkt der Erzeugung des benutzenden Programmes noch gar nicht existieren. Da bei der Erzeugung eines Simulationswerkzeugs modellspezifische Steuerungsalgorithmen natürlich ebenfalls noch nicht existieren, ist dieser Weg gut geeignet, um sie dennoch verwenden zu können.

Shared libraries mit modellabhängigen Funktionen werden im folgenden als Modellprogrammbibliotheken bezeichnet. Sie werden jeweils beim Laden eines Modells in das Simulationswerkzeug gesucht und (falls vorhanden) geöffnet [3]. Die einzelnen Algorithmen bzw. Funktionen darin werden dann bei Bedarf gesucht und ausgeführt. Anwenderinnen werden weiter unterstützt, indem die Modellprogrammbibliotheken vor dem Öffnen aktualisiert, d.h. bei Bedarf neu übersetzt und gebunden werden [4].

19.2 Anwendungssituationen

In den nachfolgenden Unterabschnitten werden die für eine modellabhängige Programmierung gegebenen Ansatzpunkte vorgestellt. Für je eine Anwendungssituation wird dargestellt, wann und zu welchem Zweck benutzerdefinierte Funktionen zur Übernahme der Kontrolle in diesen Situationen aufgerufen werden, welche formalen Anforderungen diese Funktionen erfüllen müssen und wie das Standardverhalten in der jeweiligen Situation ist, falls keine modellspezifischen Funktionen vorhanden sind. Wo dies möglich ist, wird schließlich noch erläutert, wie dieses Standardverhalten angepaßt werden kann.

---

[1] Der Begriff "shared library" entstammt der Unix-Welt (in der Windows-Welt werden sie DLLs ("dynamic link library") genannt) und bezeichnet ursprünglich eine zu einer Softwarebibliothek (library) zusammengefaßte Menge ausführbarer Routinen bzw. Funktionen, die in den Arbeitsspeicher eines Computers geladen und deren Bestandteile von mehreren Programmen (quasi) gleichzeitig genutzt werden können. Vgl. hierzu auch Simon: *Windows 95 WIN32 programming API Bible*; S. 1341 ff.

[2] Diese Technik wird als "runtime dynamic linking" bezeichnet. Vgl. wiederum Simon: *Windows 95 WIN32 programming API Bible*; S. 1341 ff.

[3] vgl. Kapitel 21

[4] ebd.

19.2.1 Teilebehandlung auf Einzelkomponenten

Jeder Einzelkomponente (*SingleUnit* [1]) eines Modells ist eine Teilebehandlungsfunktion zugeordnet, in der die Aktionen beschrieben sind, die ausgeführt werden müssen, um ein Teil über die Komponente zu bewegen. Die Teilebehandlung wird durch Aufruf der Funktion *handle ()* der jeweiligen *SingleUnit* gestartet. Diese Funktion wird im allgemeinen aus der Steuerungsfunktion [2] der *SingleUnit* aufgerufen. Ihr ist (eine Referenz auf) das zu behandelnde Teil als Parameter zu übergeben. Sie veranlaßt dann ihrerseits die Ausführung der Teilebehandlungsfunktion für das Teil in einem eigenen Thread.

Als Teilebehandlungsfunktionen können alle Funktionen verwendet werden, deren Typ *void f (Part&, SingleUnit&)* ist. Die Teilebehandlungsfunktion einer *SingleUnit* kann bei der Erzeugung oder durch Aufruf ihrer Funktion *setPartHandleFunc ()* gesetzt werden [3]. Dabei ist jeweils der Name der gewünschten Teilebehandlungsfunktion anzugeben.

Einzelkomponenten, deren Teilebehandlungsfunktion nicht explizit gesetzt wurde, benutzen bei Bedarf die in Bild 58 [4] gezeigte Standardteilebehandlungsfunktion. Für jedes Modell kann eine eigene Standardteilebehandlungsfunktion definiert werden. Dies erfordert lediglich die Integration einer Funktion des o.a. Typs mit dem Namen *defaultPartHandleFunc* in die Modellprogrammbibliothek.

Beim Aufruf der Teilebehandlungsfunktion werden dieser (Referenzen auf) das zu behandelnde Teil und die jeweilige Einzelkomponente als Parameter übergeben.

Wie eingangs dieses Abschnitts gesagt, sind in den Teilebehandlungsfunktionen die Aktionen beschrieben, die ausgeführt werden müssen, um ein Teil über die jeweilige Einzelkomponente zu bewegen. Der Blick auf die Standardfunktion zeigt, daß dazu im wesentlichen die Bewegungsfunktionen der Klasse *Part* [5] für das jeweilige Teil in geeigneter Reihenfolge aufgerufen werden (müssen).

---

[1] vgl. Kapitel 13

[2] vgl. Kapitel 19.2.2

[3] s. hierzu auch Kapitel 19.2.9

[4] s. Seite 188 in Kapitel 14

[5] vgl. Kapitel 14

Die Standardfunktion enthält eine Obermenge von Operationen, die in den meisten Fällen ausreichen wird, um die Teilebehandlung auf Komponenten wie Maschinen, Linearförderern, Drehtischen oder Regalbediengeräten zu steuern [1]. Die Leistungsfähigkeit der Standardfunktion endet, wenn Teile gelagert (bzw. die sie kontrollierenden Threads unterbrochen) werden müssen [2] oder wenn modell- bzw. komponentenabhängige Besonderheiten zu berücksichtigen sind.

Um einen Eindruck von den Einsatz- und Gestaltungsmöglichkeiten für Teilebehandlungsfunktionen zu geben, sollen hier noch einige Beispiele (bzw. Verweise darauf) angeführt werden.

```
Flag PalReady;                          Flag EHBReady;

void PalF (Part& p, SingleUnit&)        void EHBF (Part& p, SingleUnit&)
{                                       {
  p.take ();                              p.take ();
  p.transport ();                         p.transport ();
  PalReady.set ();                        EHBReady.set ();
  EHBReady.wait ();                       PalReady.wait ();
  PalReady.reset ();                      EHBReady.reset ();
  p.give ();                              p.give ();
}                                       }
```

Bild 65: Teilebehandlungsfunktionen zur Teileübergabe von Palette an EHB

Zur Demonstration der Behandlung modellabhängiger Besonderheiten wird hier nochmals auf den Anlagenausschnitt mit Teileübergabe von Palette an EHB eingegangen, der in Bild 29 [3] vorgestellt wurde. Wenn Linearförderer und EHB-Strecke als *SingleUnits* und Paletten und EHB-Fahrzeuge als *Parts* modelliert wurden, können mit Hilfe von *Flags* [4] die in Bild

---

[1] Auf den meisten Komponenten werden wohl einige Aufrufe ignoriert werden. So wird der Aufruf von *work ()* nur bei der Behandlung eines Teils mit zugeordnetem Arbeitsplan auf der Zeilkomponente des aktuellen Bearbeitungsvorgangs eine Aktion bzw. einen Zeitverbrauch bewirken. Für Komponenten, die sich nicht selbst bewegen können (z.B. Linearförderer), sind die beiden *place ()*-Aufrufe entbehrlich und auf Komponenten mit punktförmigen Plätzen (z.B. Drehtisch) bleiben die Anweisungen *center ()* und *transport ()* ohne Wirkung.

[2] Die Einlagerung bzw. Unterbrechung des Kontrollthreads wird durch Aufruf von *store ()* erreicht. Da jeder solche Aufruf eine (passende) Bestellung erfordert, um das Teil wieder auszulagern (bzw. den Kontrollthread fortzusetzen (vgl. Kapitel 17), ist ein *store ()*-Aufruf in der Standardfunktion nicht sinnvoll.

[3] s. Seite 145 in Kapitel 12.1

[4] vgl. Kapitel 12.4

65 gezeigten Teilebehandlungsfunktionen zur Synchronisation von EHB- und Palettenstrom formuliert werden (die wirklich übersichtlich bleiben).

Eine Teilebehandlungsfunktion zur Benutzung auf Lagerkomponenten zeigt Bild 59 [1]. In der in Bild 70 [2] dargestellten beispielhaften Steuerungsfunktion für eine Abfüllmaschine wird auf die Benutzung einer Teilebehandlungsfunktion ganz verzichtet.

19.2.2 Steuerung von Einzelkomponenten

(Einzel-) Komponenten von Materialflußsystemen wie Förderer, Maschinen u.ä. werden im hier vorgestellten Gesamtkonzepts als Instanzen der Klasse *SingleUnit* [3] abgebildet. Jede *SingleUnit* ist ein aktives Objekt, dessen Verhalten im Modell durch eine Steuerungsfunktion kontrolliert wird.

Die Steuerungsfunktion wird als eigener Thread durch ein der *SingleUnit* zugeordnetes *Activity*-Objekt [4] ausgeführt. Dieser Thread wird zu Beginn eines Experiments gestartet und bei dessen Beendigung gestoppt. Die Steuerungsfunktion selbst enthält (typischerweise) eine Dauerschleife [5], um den Kontrollthread nicht vorzeitig enden zu lassen.

```
void defaultSingleUnitFunc (SingleUnit& self, LCursor<Part>&) {
  for (;;)        self.handle (self.FirstBooking ());
}
```

Bild 66: Standardsteuerungsfunktion für Einzelkomponenten

Als Steuerungsfunktionen können alle Funktionen verwendet werden, deren Typ *void f (SingleUnit&, LCursor<Part>&)* ist. Die Steuerungsfunktion einer *SingleUnit* kann bei der Erzeugung oder durch Aufruf ihrer Funktion *setSingleUnitFunc ()* gesetzt werden [6]. Dabei ist jeweils der Name der gewünschten Steuerungsfunktion anzugeben.

---

[1] s. Seite 188 in Kapitel 14

[2] s. Seite 216 in Kapitel 19.2.2

[3] vgl. Kapitel 13

[4] vgl. Kapitel 12

[5] vgl. die nachfolgenden Beispiele.

[6] s. hierzu auch Kapitel 19.2.9

SingleUnits, deren Steuerungsfunktion nicht explizit gesetzt wurde, benutzen die in Bild 66 gezeigte Standardsteuerungsfunktion. Für jedes Modell kann eine eigene Standardsteuerungsfunktion definiert werden. Dies geschieht durch Integration einer Funktion des o.a. Typs mit dem Namen *defaultSingleUnitFunc* in die Modellprogrammbibliothek.

Beim Aufruf der Steuerungsfunktion werden dieser (Referenzen auf) "ihre" *SingleUnit* und ein *LCursor*-Objekt [1] zur Bearbeitung des Inhalts der Anmeldeliste der Komponente als Parameter übergeben.

```
 1 void RBGUnitFunc (SingleUnit& self, LCursor<Part> cPart) {
 2   Vec3D rbgPos, aPos, bestPos;                  // 3 Hilfsvektoren
 3   Part *bestPart;                               // (bisher) bestes Teil
 4   GeoObject& geo (self.LastPlace ()->Shape ()); // Platzkoordinatensystem
 5
 6   for (;;) {                                    // Dauerschleife
 7     self.FirstBooking ();                       // auf Anmeldung warten
 8     cPart.First ();                             // Cursor auf 1. Teil
 9     bestPart = cPart ();                        // bestes Teil = 1. Teil
10     GeoObject& g = bestPart->Shape ();          // Koordinatensys. 1. Teil
11     bestPos = g.Pos ();                         // beste Pos = 1. Pos
12     g.PosToGlobal (bestPos);                    // in globale Koordinaten
13     rbgPos = geo.Pos ();                        // aktuelle Position RBG
14     geo.PosToGlobal (rbgPos);                   // in globale Koordinaten
15
16     while (++cPart) {                           // solange Anmeldungen
17       GeoObject& aGeo = cPart ()->Shape ();     // Koordinatensys. Teil
18       aPos = aGeo.Pos ();                       // Position des Teils
19       aGeo.PosToGlobal (aPos);                  // in globale Koordinaten
20       if ((aPos - rbgPos) < (bestPos - rbgPos)) {  // wenn Abstand kleiner
21         bestPos = aPos;                         // beste Pos merken
22         bestPart = cPart ();                    // bestes Teil merken
23         }
24       }
25
26       self.handle (*bestPart);                  // bestes Teil behandeln
27       self.waitCapacity ();                     //freie Kapazität erwarten
28     }
29   }
```

Bild 67: Eine Steuerungsfunktion für ein RBG

[1] *LCursor* sind Iteratoren zum Bearbeiten von Listen (vgl. Kapitel 10.4).

Wie eingangs dieses Abschnitts gesagt, kontrolliert die Steuerungsfunktion das Verhalten der Komponente. Ihre wichtigste Aufgabe ist es, die Behandlung angemeldeter Teile zu veranlassen (oder selbst durchzuführen).

Die Standardfunktion zeigt den einfachsten Weg zur Erledigung dieser Aufgabe. In einer Dauerschleife holt die Komponente durch Aufruf der Funktion *FirstBooking ()* eine Referenz auf das jeweils erste Teil aus seiner Anmeldeliste und veranlaßt dann dessen Behandlung (d.h. die Ausführung der Teilebehandlungsfunktion [1]) durch Aufruf der Funktion *handle ()*, der sie (die Referenz auf) das Teil übergibt. Dabei unterbricht *FirstBooking ()* den aktuellen Thread (also hier den Kontrollthread der Komponente) implizit bis zum Eintreffen einer Anmeldung falls zunächst keine vorhanden ist.

Die Leistung der Standardfunktion wird wohl zur Steuerung der meisten Fälle ausreichen. Sie endet jedoch, wenn die Teile nicht mehr in der Reihenfolge der Anmeldung (also nach der FIFO-Regel) behandelt werden sollen oder wenn andere modell- bzw. komponentenabhängige Besonderheiten zu berücksichtigen sind.

Um einen Eindruck von den Gestaltungsmöglichkeiten für Steuerungsfunktionen zu geben, sollen hier noch zwei Beispiele vorgestellt werden.

Bild 67 zeigt eine mögliche Steuerungsfunktion für das in Kapitel 13.4.4 entworfene Regalbediengerät (RBG). Die angemeldeten Teile werden dabei nicht in FIFO-Reihenfolge befördert, es wird vielmehr jeweils das der aktuellen Position des RBG am nächsten liegende Teil transportiert. Die Steuerungsfunktion beschränkt sich auf die Teileauswahl. Zur eigentlichen Teilebehandlung wird die Standardteilebehandlungsfunktion benutzt.

Als zweites Beispiel soll eine Steuerungsfunktion für die in Bild 68 schematisch dargestellte Maschine vorgestellt werden, die z.B. eine Getränkeabfüllmaschine sein könnte. Die Maschine rotiert ständig mit konstanter Geschwindigkeit im Uhrzeigersinn. Sie hat acht kreisförmig angeordnete Plätze. Von der grünen Zufuhrstrecke nimmt die Maschine jedesmal auf den vorbeikommenden Platz (im Bild Platz 0) ein Teil auf. Während der weiteren Drehung werden die Teile bearbeitet. Auf die (ebenfalls) grüne Abfuhrstrecke gibt die Maschine jedesmal das auf dem vorbeikommenden Platz (im Bild Platz 6) liegende Teil ab. Der zwischen Abfuhr- und Zufuhrstrecke liegende Platz (im Bild Platz 7) ist immer leer. Die Maschine rotiert auch dann weiter, wenn keine Teile am Eingang vorhanden sind, so daß Lücken bzw. leere Plätze vorkommen können. Da Aufnahme und Abgabe “on-the-fly” geschehen sollen, sind die entsprechenden Zeitparameter der Plätze auf 0 zu setzen.

[1] vgl. Kapitel 19.2.1

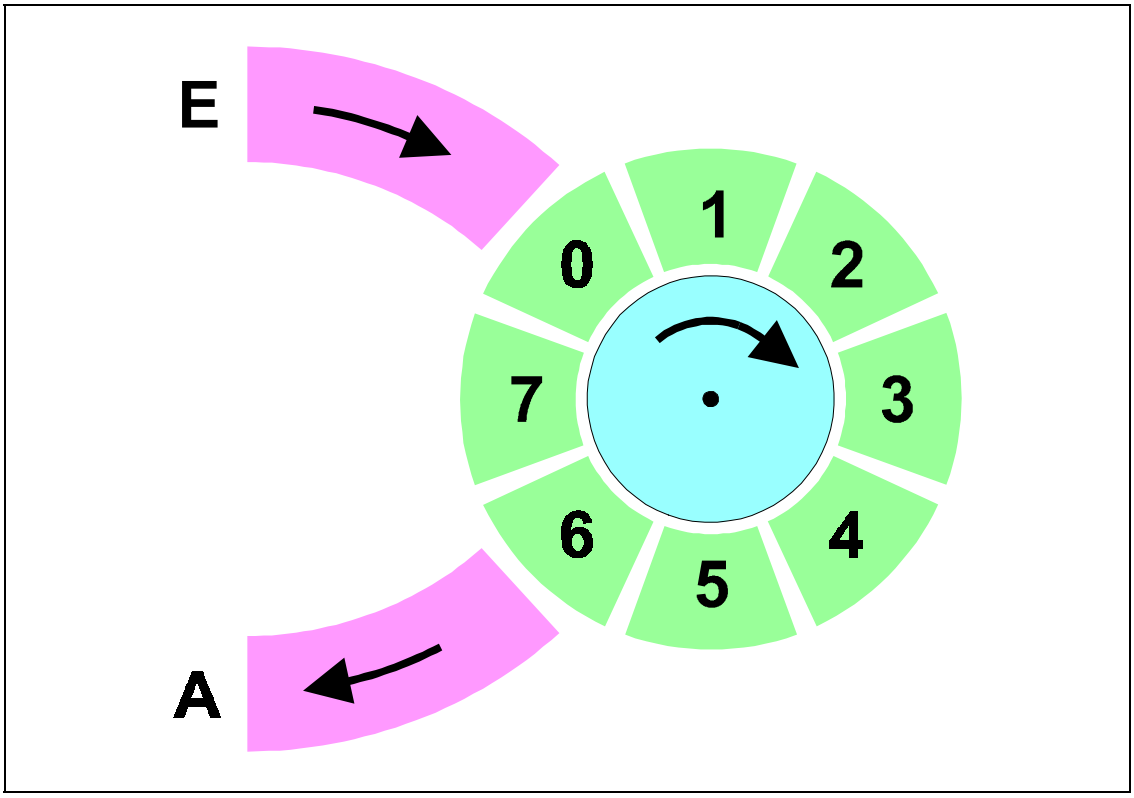

Bild 68: Schema einer Abfüllmaschine

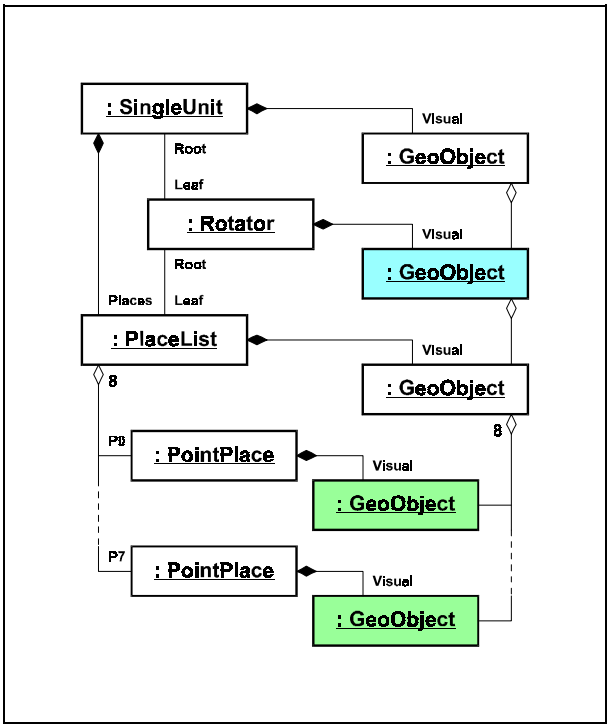

Bild 69: Struktur der Abfüllmaschine

Bild 69 zeigt ergänzend die Struktur der Abbildung der Abfüllmaschine in der aus Kapitel 13.4 bekannten Weise. Der Unterschied zu dem in Kapitel 13.4.2 dargestellten Drehtisch besteht strukturell lediglich darin, daß die Abfüllmaschine acht *PointPlaces* als Plätze hat gegenüber nur einem beim Drehtisch.

Die Steuerungsfunktion dazu zeigt Bild 70. Darin werden zunächst ein Feld mit Zeigern auf Teile und zwei Indizes für den Zugriff darauf vereinbart. Weiter wird ein *Lcursor* erzeugt, der immer den aktuellen Eingangsplatz referenziert. Schließlich wird noch dessen *Location* als Rotationsziel gespeichert.

Danach tritt der Algorithmus in eine Dauerschleife ein. Darin wird jeweils zuerst geprüft, ob ein Teil abzugeben ist und diese Operation erforderlichenfalls ausgelöst. Danach wird geprüft, ob ein Teil aufzunehmen ist und diese Operation bei Bedarf durchgeführt. Dabei werden die Indizes weitergesetzt und das Belegungsfeld aktualisiert. Schließlich wird der Eingangsplatz (d.h. der *LCursor*) entgegen dem Uhrzeigersinn eine Position weitergesetzt und dieser neue Eingangsplatz auf das Rotationsziel gedreht.

```
 1 void AbfuellUnitFunc (SingleUnit& self, LCursor<Part>&) {
 2   Part* part [8];                          // Belegungsfeld
 3   for (int i =0; i < 8; part [i++] = NULL); // Belegungsfeld initialisieren
 4   int iEin = 0, iAus = 6;                  // 2 Indizes
 5   LCursor<Place> cEin (self.Places ());    // aktueller Eingangsplatz
 6   Location loc (cEin ()->Shape ().Loc ()); // loc ist Ziel für Rotation
 7   cEin ()->Shape ().LocToGlobal (loc);     // in globalen Koordinaten
 8
 9   for (;;) {                               // Dauerschleife
10     if (part [iAus]) {                     // wenn Teil am Ausgang
11       part [iAus].give ();                 // Teil abgeben
12       part [iAus] = NULL;                  // Belegung zurücksetzen
13     }
14     if (--iAus == 0) iAus = 7;             // Index aktualisieren
15
16     if (self.hasBooking ()) {              // wenn Teil am Eingang
17       part [iEin] = self.FirstBooking ();  // Belegung setzen
18       part [iEin].take (ein ());           // Teil nehmen auf Eingangsplatz
19     }
20     if (--iEin == 0) iEin = 7;             // Index aktualisieren
21
22     if (--cEin == NULL) cEin.Last ();      // Eingangsplatz aktualisieren
23     cEin ()->rotTo (loc);                  // Eingangsplatz an Ziel drehen
24   }
25 }
```

Bild 70: Die Steuerungsfunktion der Abfüllmaschine

Die Besonderheit dieser Steuerungsfunktion ist, daß sie die Teilebehandlung mit übernimmt, wobei sie sich aber auf das unumgänglich notwendige Aufnehmen und Abgeben beschränkt (Zeilen 18 bzw. 11). Eine explizite Teilebehandlungsfunktion wird nicht benutzt. Auch dieser Ansatz bewegt sich im Rahmen der gegebenen Gestaltungsmöglichkeiten. Er schließt hier z.B. konkurrierende Bewegungsanforderungen aus parallel ausgeführten Teilebehandlungsfunktionen (bzw. -threads) implizit aus und verhilft der Lösung durch die Behandlung aller Aspekte in einem Algorithmus zu größtmöglicher Transparenz.

19.2.3 Platzwahl auf Einzelkomponenten

Wie in Kapitel 13 dargestellt, kann jede Einzelkomponente (*SingleUnit*) eine beliebige Anzahl von Plätzen enthalten. Ein Beispiel einer Komponente mit mehreren Plätzen ist die im vorigen Abschnitt entworfene Abfüllmaschine. Auch die in Kapitel 17 vorgeschlagene Vorgehensweise zur Modellierung von Hochregallagern führt zu Komponenten mit mehreren Plätzen (Alle links und rechts einer Lagergasse angeordneten Plätze wurden dort je einer die Gasse als Ganzes repräsentierenden *SingleUnit* zugeordnet). Weitere Beispiele sind Palettenwechsler vor Bearbeitungsmaschinen und Maschinen mit mehreren Plätzen.

Die Plätze solcher Komponenten stehen logisch nebeneinander. Die Teile, die auf einer solchen Komponente behandelt werden, benutzen also jeweils nur einen der Plätze. Daher muß für jedes Teil ein Platz zur Benutzung ausgewählt werden. Hierzu ist jeder Komponente eine Platzwahlfunktion zugeordnet, deren Aufgabe darin besteht, den Platz für ein Teil auszuwählen und (eine Referenz auf diesen) zurückzugeben

Die Platzwahlfunktion wird für jedes Teil einmal aufgerufen. Der Aufruf erfolgt aus der Ablaufsteuerung der von dem Teil zuvor belegten *SingleUnit*, also noch vor der Anmeldung des Teils bei der eigentlich betroffenen Komponente. Der genaue Aufrufzeitpunkt hängt von der Struktur und der Ablaufsteuerung der vorangehenden *SingleUnit* ab. Wenn diese sich nicht selbst bewegen kann [1] oder aus ihrer Ablaufsteuerung keine Bewegungsanforderungen erfolgten, wird die Platzwahlfunktion für die nachfolgende Komponente erst beim Abgeben des Teils aus der Funktion *give ()* aufgerufen. Anderenfalls erfolgt der Aufruf bei der ersten Bewegungsanforderung nachdem das Teil auf der vorgelagerten Komponente aufgenommen wurde [2].

Als Platzwahlfunktionen können alle Funktionen verwendet werden, deren Typ *Place& f (Part&, SingleUnit&, Lcursor<Place>&)* ist. Die Platzwahlfunktion einer *SingleUnit* kann bei der Erzeugung oder durch Aufruf der Funktion *setPlaceSelectFunc ()* gesetzt werden [3]. Dabei ist jeweils der Name der gewünschten Platzwahlfunktion anzugeben.

---

[1] Dies ist dann der Fall, wenn in der Struktur einer Komponente keine Bewegungselemente (*Mover*, vgl. Kapitel 13.2) enthalten sind, also z.B. bei dem in Kapitel 13.4.1 dargestellten Linearförderer.

[2] Exakt erfolgt die Platzwahl bei Aufruf einer der Funktionen *move ()*, *rotate ()* oder *place ()* für das Teil aus der Ablaufsteuerung der vorangehenden Komponente nachdem dieses Teil dort (durch Aufruf der Funktion *take ()*) aufgenommen wurde.

[3] s. hierzu auch Kapitel 19.2.9

Einzelkomponenten, deren Platzwahlfunktion nicht explizit gesetzt wurde, benutzen die in Bild 71 gezeigte Standardplatzwahlfunktion. Für jedes Modell kann eine eigene Standardplatzwahlfunktion definiert werden. Dies geschieht durch Integration einer Funktion des o.a. Typs mit dem Namen *defaultPlaceSelectFunc* in die Modellprogrammbibliothek.

```
Place& defaultPlaceSelectFunc (Part&, SingleUnit&, Lcursor<Place>& c) {
  return *c ();
}
```

Bild 71: Standardplatzwahlfunktion für Einzelkomponenten

Bei jedem Aufruf der Platzwahlfunktion werden dieser (Referenzen auf) das zu behandelnde Teil, die jeweilige Einzelkomponente und ein *LCursor*-Objekt zur Bearbeitung des Inhalts der Platzliste der Komponente als Parameter übergeben.

Wie weiter oben gesagt, besteht die Aufgabe der Platzwahlfunktionen vor allem darin, für jedes Teil einen Platz festzulegen, den es bei seiner Behandlung auf der jeweiligen Einzelkomponente benutzt. Der Blick auf die Standardfunktion zeigt, daß diese immer den ersten Platz der Komponente auswählt. Die Leistungsfähigkeit der Standardfunktion reicht daher in der Regel nur für Komponenten aus, die nur einen Platz haben [1].

```
Place& LagerPlatzSelectFunc (Part& p, SingleUnit&, Lcursor<Place>& c) {
  for ( ; c; c++)                          // solange noch Plätze ungeprüft
    if (p.Len () < c ()->FreeLength ()) // wenn Teillänge < freie Länge auf Platz
      return *c ();                        // dann nimm diesen Platz
  return *c.First ();                      // Ersatzauswahl: nimm ersten Platz
}
```

Bild 72: Beispielhafte Platzwahlfunktion für Lagerkomponenten

Um einen Eindruck von den Gestaltungsmöglichkeiten für Platzwahlfunktionen zu geben, wird in Bild 72 noch ein Beispiel für eine Platzwahlfunktion zur Benutzung auf nach dem Konzept aus Kapitel 17 gestalteten Lagerkomponenten vorgestellt. Die Funktion wählt den ersten Platz aus der Platzliste aus, auf dem genügend freier Platz für das Teil vorhanden ist. Falls kein solcher Platz gefunden wird, wählt sie ersatzweise den ersten Platz [2].

---

[1] Dies dürfte allerdings auf die überwiegende Zahl der Einzelkomponenten zutreffen.

[2] Diese Ersatzauswahl sei hier gestattet, da es sich um ein Beispiel handelt. In einem konkreten Anwendungsfall bestünde hier wohl Anlaß zur Besorgnis. Es wäre zu prüfen, ob diese Situation (dem Teil wurde zuvor eine voll belegte Lagergasse zugewiesen) überhaupt auftreten dürfte.

19.2.4 Komponentenwahl aus Komponentengruppen

Wie in Kapitel 16.1 dargestellt wurde, können Komponentengruppen (*UnitGroups*) gebildet werden, in die Einzelkomponenten (*SingleUnits*) oder andere Gruppen eingeordnet werden können. Komponentengruppen können z.B. Maschinengruppen oder ganze Lager (als Zusammenfassung mehrerer Lagergassen) repräsentieren.

UnitGroups können wie Einzelkomponenten als Bearbeitungsorte von Vorgängen in Arbeitsplänen verwendet werden [1]. Dabei treten sie, ebenso wie bei ihrer (ebenfalls zulässigen) Übergabe als Parameter der Funktion *setTarget ()* [2], als Förderziel von Teilen auf. Da Komponentengruppen im hier vorgestellten Gesamtkonzept logische Elemente sind, die mit Teilen nicht in Berührung kommen (können), muß beim Auftreten einer Gruppe als Förderziel eines Teils jeweils ein Mitglied der Gruppe als tatsächliches Förderziel ausgewählt werden [3]. Hierzu ist jeder *UnitGroup* eine Komponentenwahlfunktion zugeordnet, deren Aufgabe darin besteht, die für ein Teil zu benutzenden Mitgliedskomponente auszuwählen und (eine Referenz auf diese) zurückzugeben. Die Komponentenwahlfunktion wird für jedes Teil einmal aufgerufen, wenn dem Teil als Förderziel eine Komponentengruppe zugewiesen werden soll. Der Aufruf erfolgt aus der Funktion *setTarget ()* des Teils.

```
Unit& defaultUnitSelectFunc (Part&, UnitGroup&, Lcursor<Unit>& c) {
  return *c ();
}
```

Bild 73: Standardkomponentenwahlfunktion für Komponentengruppen

Als Komponentenwahlfunktionen können alle Funktionen verwendet werden, deren Typ *Unit& f (Part&, UnitGroup&, Lcursor<Unit>&)* ist. Bei der Erzeugung oder durch Aufruf der Funktion *setUnitSelectFunc ()* [4] kann die Komponentenwahlfunktion einer *UnitGroup* gesetzt werden, wobei jeweils der Name der gewünschten Funktion anzugeben ist.

---

[1] vgl. Kapitel 15

[2] vgl. Kapitel 14

[3] Falls eine Gruppe, wie oben und in Kapitel 16.1 als Möglichkeit dargestellt, eine andere Gruppe als Mitglied enthält, kann es also zu einem mehrstufigen rekursiven Auswahlprozeß kommen.

[4] s. hierzu auch Kapitel 19.2.9

Komponenten, deren Komponentenwahlfunktion nicht explizit gesetzt wurde, benutzen bei Bedarf die in Bild 73 gezeigte Standardkomponentenwahlfunktion. Für jedes Modell kann eine eigene Standardkomponentenwahlfunktion definiert werden. Dies erfordert lediglich die Integration einer Funktion des o.a. Typs mit dem Namen *defaultUnitSelectFunc* in die Modellprogrammbibliothek.

Bei jedem Aufruf der Komponentenwahlfunktion werden dieser (Referenzen auf) das zu behandelnde Teil, die jeweilige Komponentengruppe und ein *LCursor*-Objekt zur Bearbeitung des Inhalts der Mitgliederliste der Gruppe als Parameter übergeben.

Wie oben gesagt, besteht die Aufgabe der Komponentenwahlfunktionen darin, für jedes Teil eine Mitgliedskomponente festzulegen, die es als tatsächliches Förderziel anstelle der Gruppe benutzt. Der Blick auf die Standardfunktion zeigt, daß diese immer das erste Mitglied der Gruppe auswählt. Die Leistungsfähigkeit der Standardfunktion reicht daher in der Praxis wohl nur für Testzwecke oder Sonderfälle aus.

```
Unit& LagerUnitSelectFunc (Part&, UnitGroup&, Lcursor<Unit>& c) {
  for ( ; c; c++)                  // solange noch Mitglieder ungeprüft
    if (c ()->Capacity () >        // wenn Kapazität >
        c ()->PartCount ())        //    Belegung
      return *c ();                // dann nimm dieses Mitglied
  return *c.First ();              // Ersatzauswahl: nimm erstes Mitglied
}
```

Bild 74: Beispielhafte Komponentenwahlfunktion für Lager

Um einen Eindruck von den Gestaltungsmöglichkeiten für Komponentenwahlfunktionen zu geben, wird in Bild 74 noch ein Beispiel für eine Komponentenwahlfunktion zur Benutzung auf nach dem Konzept aus Kapitel 17 gestalteten Lagern vorgestellt. Die Funktion wählt jeweils die erste Komponente aus der Mitgliederliste aus, deren Kapazität größer als die auf ihr vorhandene Anzahl Teile ist. Falls keine solche Komponente gefunden wird, wählt sie ersatzweise die erste Komponente [1].

[1] Diese Ersatzauswahl sei hier gestattet, da es sich um ein Beispiel handelt. In einem konkreten Anwendungsfall wäre zu prüfen, ob diese Situation (das Teil soll in ein voll belegtes Lager transportiert werden) überhaupt auftreten dürfte. Eventuell müßte die Kapazität des Lagers erhöht werden. Ebenso wäre zu erwägen, den aktuellen Thread (der die Komponentenwahlfunktion aufgerufen hat) zu unterbrechen, bis freie Kapazität vorhanden ist, wodurch das nachfragende Teil implizit auf seiner aktuellen Position festgehalten würde.

19.2.5 Bestellungen

Wie in Kapitel 17 dargestellt wurde, müssen Teile, die auf Einzelkomponenten durch Aufruf der Funktion *store ()* eingelagert wurden, durch explizite Bestellungen ausgelagert werden, die durch Aufruf einer der *order ()*-Funktionen einer Einzelkomponente oder Komponentengruppe erzeugt werden.

Eine der dabei bestehenden Möglichkeiten ist die Angabe (des Namens) einer modell- bzw. aufgabenbezogenen Prüf- oder Bestellfunktion, mit der die auf der Komponente eingelagerten Teile unter Anwenderkontrolle daraufhin untersucht werden können, ob sie gegebenen Kriterien genügen und also ausgelagert werden sollen. Wie in Kapitel 17 beschrieben, wird diese Bestellfunktion solange immer wieder für die eingelagerten Teile aufgerufen, bis die Bestellung vollständig erfüllt ist.

Als Bestellfunktionen können alle Funktionen verwendet werden, deren Typ *int f (Unit&, Part&)* ist. Bei jedem Aufruf einer Bestellfunktion werden dieser (Referenzen auf) die Komponente, bei der die Bestellung erzeugt wurde und das zu prüfende Teil als Parameter übergeben. Wenn die Bestellfunktion einen von 0 verschiedenen Wert zurückliefert, wird das geprüfte Teil anschließend ausgelagert, anderenfalls bleibt es eingelagert.

Da keine allgemeingültigen Bestellkriterien angegeben werden könnten, die nicht durch andere Bestellmöglichkeiten abgedeckt würden, gibt es keine Standardbestellfunktion. Damit entfällt auch die Möglicheit, eine modellspezifische Standardbestellfunktion definieren zu können.

```
int sampleOrderFunc (Unit&, Part& p) {
  const SingleUnit* t (p.ActTarget ());        // Förderziel des Teils
  if (t->Capacity () > t->PartCount ())        // wenn Kapazität > Belegung
      return 1;                                // dann Teil auslagern
  else   return 0;                             // sonst Teil liegenlassen
}
```

Bild 75: Beispielhafte Bestellfunktion

Um einen Eindruck von den Gestaltungsmöglichkeiten für Bestellfunktionen zu geben, wird in Bild 75 noch ein Beispiel für eine solche Funktion vorgestellt. Diese Funktion veranlaßt die Auslagerung derjenigen geprüften Teile, bei deren (aktuellem) Förderziel zum Prüfzeitpunkt die Kapazität größer als die dort vorhandene Anzahl Teile ist [1].

---

[1] Die Benutzung dieser Bestellfunktion ist nur sinnvoll, wenn das Förderziel aller zu prüfenden Teile vor ihrer Einlagerung durch Aufruf von *store ()* bereits auf das nächste Ziel weitergeschaltet wurde.

19.2.6 Steuerung von Komponentengruppen

Wie in Kapitel 16.1 dargestellt, können Einzelkomponenten von Materialflußsystemen in Komponentengruppen zusammengefaßt werden, die als Instanzen der Klasse *UnitGroup* abgebildet werden. Jede *UnitGroup* kann zu einem aktiven Objekt gemacht werden indem ihr eine Gruppensteuerungsfunktion zugeordnet wird, die ihr Verhalten im Modell kontrolliert. *UnitGroups* ohne Steuerungsfunktion bleiben passiv.

Die Gruppensteuerungsfunktion wird als eigener Thread durch ein der *UnitGroup* zugeordnetes *Activity*-Objekt [1] ausgeführt. Dieser Thread wird zu Beginn eines Experiments gestartet und bei dessen Beendigung gestoppt. Die Gruppensteuerungsfunktion selbst enthält (typischerweise) eine Dauerschleife [2], um den Kontrollthread nicht vorzeitig enden zu lassen.

Als Gruppensteuerungsfunktionen können alle Funktionen verwendet werden, deren Typ *void f (UnitGroup&)* ist. Die Gruppensteuerungsfunktion einer *UnitGroup* kann bei der Erzeugung oder durch Aufruf ihrer Funktion *setGroupFunc ()* gesetzt werden [3]. Dabei ist jeweils der Name der gewünschten Gruppensteuerungsfunktion anzugeben.

Beim Aufruf der Gruppensteuerungsfunktion wird dieser (eine Referenz auf) "ihre" *UnitGroup* als Parameter übergeben.

Da keine allgemeingültigen Aufgaben für Gruppensteuerungsfunktionen angegeben werden können und die Existenz einer Gruppensteuerungsfunktion für eine *UnitGroup* nicht zwingend erforderlich ist, gibt es keine Standardgruppensteuerungsfunktion. Konsequenterweise entfällt auch die Möglicheit, eine modellspezifische Standardgruppensteuerungsfunktion definieren zu können.

Um einen Eindruck von den Gestaltungsmöglichkeiten für Gruppensteuerungsfunktionen zu geben, wird in Bild 76 noch ein Beispiel für eine solche Funktion vorgestellt. Diese Funktion könnte für ein Lager, d.h. für die Gruppe der zugehörigen Lagerkomponenten verwendet werden. Nachdem sie ihre weitere Arbeit vorbereitet hat, tritt sie in eine Dauerschleife ein, in der sie im Abstand von 10 Minuten prüft, ob der Bestand einer bestimmten Art von Teilen (identifiziert durch die Zugehörigkeit zu einem bestimmten Arbeitsplan) unter einen

---

[1] vgl. Kapitel 12

[2] vgl. das nachfolgende Beispiel.

[3] s. hierzu auch Kapitel 19.2.9

Schwellwert gefallen ist. In diesem Fall sorgt die Funktion für die Wiederauffüllung des Lagers, indem sie eine festgelegte Menge neuer Teile dieser Art auf einer anderen Einzelkomponente (deren Name “Eingang” ist) erzeugt. Alle diese Teile benutzen das *Pattern* [1] “TeileGestalt” als geometrische Repräsentation.

```
void sampleGroupFunc (UnitGroup& self) {
  SingleUnit* ein = SingleUnit::find (“Eingang”);     // Systemeingang / Quelle
  Pattern *geo = Pattern::find (“TeileGestalt”);      // Geometrie d. Teile
  Workplan *plan = WorkPlan::find (“TeilePlan”);      // Arbeitsplan
  unsigned schwelle = 7, menge = 25;                  //Bestellschwelle, -menge

  for (;;) {                                          // Dauerschleife
    Activity::hold (600);                             // warte 600s (= 10 min.)
    if (self.available (*plan) <= schwelle)           // Teilezahl <= Schwelle
      Part::create (*ein, *geo, *plan, 0, menge);     // dann neu erzeugen
  }
}
```

Bild 76: Beispielhafte Gruppensteuerungsfunktion

19.2.7 Kontrolle von Steuerungen

Wie in Kapitel 16.2 dargestellt, gestattet es das hier entwickelte Komponentenkonzept, in Materialflußsystemen vorhandene oder mit ihnen in Beziehung stehende Steuerungsrechner in Modellen explizit als Instanzen der Klasse *Control* abzubilden. Jedes *Control*-Objekt ist in der Regel ein aktives Objekt, dessen Verhalten im Modell durch eine Kontrollfunktion bestimmt wird. *Control*-Objekte ohne Kontrollfunktion bleiben passiv.

Die Steuerungskontrollfunktion wird als eigener Thread durch ein jeder *Control*-Instanz zugeordnetes *Activity*-Objekt [2] ausgeführt. Dieser Thread wird zu Beginn eines Experiments gestartet und bei dessen Beendigung gestoppt. Die Steuerungskontrollfunktion selbst enthält (typischerweise) eine Dauerschleife, um den Kontrollthread nicht vorzeitig enden zu lassen.

Als Steuerungskontrollfunktionen können alle Funktionen verwendet werden, deren Typ *void f (Control&)* ist. Die Steuerungskontrollfunktion einer *Control* kann bei der Erzeugung oder

---

[1] vgl. Kapitel 11.1

[2] vgl. Kapitel 12

durch Aufruf ihrer Funktion *setControlFunc ()* gesetzt werden [1]. Dabei ist jeweils der Name der gewünschten Steuerungskontrollfunktion anzugeben.

Beim Aufruf der Steuerungskontrollfunktion wird dieser (eine Referenz auf) "ihre" Steuerung als Parameter übergeben.

Da keine allgemeingültigen Aufgaben für Steuerungskontrollfunktionen angegeben werden können und die Existenz einer Steuerungskontrollfunktion für *Control*-Objekte nicht zwingend erforderlich ist, gibt es keine Standardsteuerungskontrollfunktion. Konsequenterweise entfällt auch die Möglicheit, eine modellspezifische Standardsteuerungskontrollfunktion definieren zu können.

Um einen Eindruck von den gebotenen Gestaltungsmöglichkeiten für Steuerungskontroll-funktionen zu geben, wird hier noch ein Beispiel einer solchen Funktionen vorgestellt.

```
void sampleControlFuncA (Control& self) {
  cout << "ControlFunc A ist aktiv" << endl;          // Meldung ausgeben
  Activity::hold (600);                               // warte 600s (= 10 min.)
  self.setControlFunc ("sampleControlFuncB");         // ControlFunc umsetzen
  self.reset ();                                      // abbrechen + neu starten
}

void sampleControlFuncB (Control& self) {
  cout << "ControlFunc B ist aktiv" << endl; // Meldung ausgeben
  Activity::hold (600);                               // warte 600s (= 10 min.)
  self.setControlFunc ("sampleControlFuncA");         // ControlFunc umsetzen
  self.reset ();                                      // abbrechen + neu starten
}
```

Bild 77: Beispielhafte Steuerungskontrollfunktionen

Die in Bild 77 dargestellten Funktionen erfüllen keine vorstellbaren anlagenbezogenen Aufgaben. Sie illustrieren vielmehr die in Kapitel 19.2.9 folgenden Ausführungen bezüglich des Wechsels von Steuerungsfunktionen während der Experimentdurchführung. Beide Funktionen arbeiten prinzipiell gleich und verzichten insbesondere auf die sonst übliche Dauerschleife. Sie geben zu Beginn eine Meldung aus und warten dann 10 Minuten. Danach installieren sie die jeweils andere Funktion als Steuerungskontrollfunktion des *Control*-Objekts, dessen Verhalten sie kontrollieren. Der abschließende Aufruf von *reset ()* bricht den aktuellen Kontrollthread ab und startet einen neuen (der dann also die jeweils andere Funktion ausführt).

---

[1] s. hierzu auch Kapitel 19.2.9

Das sich ergebende nach außen sichtbare Verhalten der kontrollierten Steuerung besteht also lediglich in einem ständigen Taskwechsel im Zyklus von 10 Minuten, der durch die Ausgabe einer Meldung publik gemacht wird.

19.2.8 Initialisierung und Abschluß

Bei der Benutzung einer Programmierschnittstelle wie der hier vorgestellten wird es nach den Projekterfahrungen des Autors sicherlich dazu kommen, daß die Programmbibliothek eines Modells eigene, vom Werkzeugkern unabhängige Objekte erzeugt und verwendet. Diese könnten z.B. der aufgabenbezogenen Datenerfassung und -ausgabe dienen.

Solche Objekte müssen zu Beginn eines Experiment erzeugt und initialisiert und bei dessen Ende wieder gelöscht werden, wobei spätestens dann erfaßte Daten beispielsweise in Dateien ausgegeben werden müssen. Zur Unterstützung bei der Ausführung solcher Aufgaben wurden hier spezielle Möglichkeiten vorgesehen.

Wenn die Modellprogrammbibliothek eine Funktion *initModelLibrary ()* enthält, sorgt das Werkzeug dafür, daß diese Funktion zwischen der Initialisierung des Modells und dem Start des Experiments aufgerufen wird. In dieser Funktion können also modellabhängige Objekte erzeugt und initialisiert werden.

In gleicher Weise wird eine in der Modellprogrammbibliothek eventuell vorhandene Funktion *closeModelLibrary ()* zwischen der Beendigung eines Experiments und der Löschung des Modells (aus dem Speicher) aufgerufen, .

Das Vorhandensein dieser beiden Funktionen ist jedoch keine notwendige Bedingung. Falls eine von ihnen (oder beide) nicht in der Modellprogrammbibliothek enthalten ist, wird der entsprechende Aufruf schlicht unterdrückt.

19.2.9 Wechsel von Kontrollfunktionen

In den vorangegangenen Abschnitten wurde eine Reihe von Situationen diskutiert, in denen durch die Integration entsprechender Funktionen in die Modellprogrammbibliothek das Standardverhalten des hier konzipierten Simulationswerkzeugs aufgabenabhängig angepaßt werden kann. Es wurde ausgeführt, daß in den meisten Fällen durch Aufruf einer entsprechenden Funktion *set...Func ()* das Verhalten der jeweils betroffenen Systemkomponenten in den jeweiligen Situationen (auch während der Experimentdurchführung) geändert werden kann, indem die entsprechende Steuerungsfunktion ausgewechselt wird. Da solche

Umschaltvorgänge unerwünschte Nebeneffekte haben können, sind zu diesem Thema noch einige Präzisierungen erforderlich, die im folgenden dargestellt werden.

Grundsätzlich können die angesprochenen Situationen bzw. die zugehörigen Kontrollfunktionen unter dem Aspekt des Verhaltens- bzw. Funktionswechsels in zwei Gruppen eingeteilt werden. Zur ersten Gruppe gehören dabei die Teilebehandlungs-, Platzwahl- und Komponentenwahlfunktionen [1]. Die zweite Gruppe umfaßt die Steuerungs- bzw. Kontrollfunktionen von Einzelkomponenten, Komponentengruppen und Steuerungen [2].

Die Funktionen der ersten Gruppe erfüllen im Rahmen des vorgestellten Konzepts Aufgaben, die ihrer Natur nach zeitlich begrenzt sind [3]. Unter dem hier diskutierten Aspekt unterscheidet sie von den Funktionen der zweiten Gruppe formal vor allem, daß sie in der Regel keine Dauerschleifen enthalten. Das Auswechseln der Funktionen dieser ersten Gruppe findet daher naheliegenderweise so statt, daß alle "neuen" Aufrufe, die nach dem Wechsel erfolgen, die "neue" Funktion benutzen, während die zum Zeitpunkt des Wechsels aktiven Aufrufe mit der "alten" Funktion zu Ende geführt werden.

Die Funktionen der zweiten Gruppe erfüllen dagegen zeitlich nicht limitierte Aufgaben, die während der gesamten Experimentlaufzeit wahrgenommen werden müssen. Sie enthalten daher typischerweise eine Dauerschleife und sind zum Wechselzeitpunkt aktiv.

Das Auswechseln einer Funktion die sich "in Arbeit" befindet, d.h. gerade von einem Thread ausgeführt wird, ist aber nicht ohne weiteres möglich. Zum einen kann aus dem aktuellen Ausführungszustand der "alten" Funktion im allgemeinen nicht auf einen Fortsetzungspunkt in der "neuen" Funktion geschlossen werden, da beide beliebig unterschiedlich sein können, so daß nur die Möglichkeit bleibt, die "neue" Funktion "von vorne" beginnen zu lassen und die alte abzubrechen. Zum anderen kann die "alte" Funktion lokale Objekte erzeugt haben, die (mangels Zugriffsmöglichkeiten) beim Funktionswechsel nicht gelöscht werden können, was jedoch im allgemeinen angemessen wäre.

In der Folge erfordert der Wechselvorgang bei diesen Funktionen zwei (von außen durchzuführende) Schritte. Zunächst ist, wie dargestellt, die neue Funktion durch Aufruf von

---

[1] vgl. Kapitel 19.2.1, 19.2.3 bzw. 19.2.4

[2] vgl. Kapitel 19.2.2, 19.2.6 bzw. 19.2.7

[3] Die Teilebehandlungsfunktionen beschreiben die Behandlung eines Teils auf einer Einzelkomponente, ihre Aufgabe endet mit der Abgabe des Teils an die nachfolgende Komponente. Die Platz- und Komponentenwahlfunktionen sollen Plätze bzw. Komponenten spezifizieren, die von Teilen benutzt bzw. angesteuert werden, sie erledigen diese Aufgabe in der Regel sofort und ohne jeden Zeitverbrauch.

set...Func () zu setzen. Danach muß auf diese umgeschaltet werden. Hierzu umfassen die betroffenen Klassen *SingleUnit*, *UnitGroup* und *Control* jeweils die Funktionen *stop ()* (stoppt den jeweiligen Kontrollthread), *start ()* (startet einen Kontrollthread mit der aktuellen Funktion sofern noch keiner aktiv ist) und *reset ()* (Kombination der beiden anderen). Das Umschalten erfordert also einen *reset ()*-Aufruf oder eine *stop ()*/*start ()*-Kombination. Die genauen Umschaltzeitpunkte müssen modellabhängig festgelegt und dabei so gewählt werden, daß unerwünschte Nebeneffekte ausgeschlossen sind.

Bei den Erläuterungen zu Steuerungskontrollfunktionen wurde in Bild 77 ein Beispiel vorgestellt, daß eine solche Umschaltung mittels *reset ()* vornimmt.

19.3 Zusammenfassung und Bewertung

Die vorgestellte Programmierschnittstelle bietet durch die Verwendung der Programmiersprache C++ praktisch unbegrenzte Möglichkeiten zur Programmierung und Integration aufgabenabhängiger Steuerungsalgorithmen in Simulationsmodelle. Dabei ist es, auch aus laufenden Experimenten heraus, prinzipiell möglich, Kopplungen mit anderen Softwaresystemen wie Datenbankservern o.ä. zu realisieren, um z.B. Daten auszutauschen.

Für die Erzeugung des C++-Codes der Modellprogrammbibliothek können externe Softwarewerkzeuge wie Editoren, Compiler oder Integrierte Entwicklungsumgebungen (IDEs) benutzt werden, die Anwendern und Anwenderinnen oft bereits bekannt sind. Im übrigen entfällt so auch die Notwendigkeit, entsprechende Funktionalitäten als Teil des Simulationswerkzeugs bereitstellen (und zuvor programmieren) zu müssen.

Wegen der Verbreitung von C / C++ sind auch die Anforderungen an die Dokumentation gut zu erfüllen, da genügend Programme verfügbar sind, die C++-Quellcode in symbolische Notationen verschiedener Art umsetzen.

Die definierten Ansatzpunkte erlauben die Übernahme der Steuerung der Abläufe in einem Simulationsmodell bis ins Detail. Da nicht nur punktuelle Entscheidungen getroffen, sondern vollständige Abläufe kontrolliert werden können, ist nicht nur beeinflußbar, was zu geschehen hat, sondern auch, wann welche Festlegungen getroffen werden. Damit erhöht sich die erreichbare Qualität der Steuerungsalgorithmen und ihrer Dokumentation.

Insgesamt erfüllt die konzipierte Programmierschnittstelle alle in Kapitel 8.1 erhobenen einschlägigen Anforderungen.

20 Speicherung von Modellen

Der Entwurf eines Simulationsmodells findet typischerweise nicht in einem Zug, sondern über mehrere Sitzungen verteilt statt, in denen das Modell schrittweise aufgebaut wird. Die sich anschließende Modellprüfung wird meist wiederum mehrere Sitzungen erfordern, bis alle erforderlichen Korrekturen eingearbeitet sind und das Modell validiert ist. Auch zur Durchführung der vorgesehenen Experimente wird das verwendete Simulationswerkzeug üblicherweise mehrfach gestartet und das Modell ebensooft geladen. Dies gilt auch bei Nutzung mehrerer Rechner für die Experimente und bei der Präsentationen des Modells.

Diese Arbeitsweise erfordert, daß Simulationsmodelle nicht nur in flüchtiger Form im Speicher eines Rechners existieren. Simulationswerkzeuge müssen vielmehr die Möglichkeit bieten, erzeugte Modelle (z.B. auf Festplatte) speichern und später wieder laden zu können. Es müssen dazu Beschreibungen der Modelle erzeugt und ausgegeben und die Modelle durch Einlesen der Beschreibungen im Rechner bzw. Werkzeug wiederhergestellt werden können. Nicht zuletzt wird dadurch auch ein gewisser Schutz gegen Daten- und Zeitverluste aufgrund (nicht auszuschließender) Rechner- oder Programmabstürze erreicht.

Als allgemeine Eigenschaften, die solche Modellbeschreibungen aufweisen sollten, nennt Eschenbacher u.a. die Lesbarkeit auch für Menschen und die Austauschbarkeit zwischen mehreren Anwendern bzw. Nutzern auch über verschiedene Rechnerplattformen [1].

---

[1] Eschenbacher: *Entwurf und Implementierung einer formalen Sprache zur Beschreibung dynamischer Modelle*; S. 16 ff.

Für die Speicherung von Modellbeschreibungen kommt damit de facto nur die Verwendung eines Systems von Dateien im ASCII-Format in Frage. Die Alternativen, Dateien in binären Formaten oder Datenbanksysteme zu benutzen, scheiden aus, weil Modellbeschreibungen in diesen Formen weder menschenlesbar noch ohne weiteres portierbar sind, da sich die Binärformate verschiedener Rechner oftmals unterscheiden und konkrete Datenbanksysteme häufig nicht auf mehreren Rechnerplattformen verfügbar sind.

Da im ASCII-Format vorliegende Modellbeschreibungen auch (mal) mit einem gewöhnlichen Editor geändert werden können, ergibt sich als weiterer Vorteil, daß Serienexperimente mit jeweils nur leicht veränderten Parametern einfach automatisierbar sind: Ein passendes Skript paßt zunächst die Parameter an (z.B. unter Verwendung eines Stream-Editors), startet dann das Experiment und wiederholt diesen Zyklus so oft wie nötig.

Das (Wieder-) Einlesen einer Modellbeschreibung durch das Simulationswerkzeug setzt wegen der erforderlichen Maschinenlesbarkeit voraus, daß die abgelegten Modellbeschreibungen festzulegende formale Konventionen (d.h. eine Syntax) einhalten [1], so wie der Quellcode eines Programms der Syntax der verwendeten Programmiersprache genügen muß. Eine formal (d.h. syntaktisch) richtige Beschreibung ermöglicht wegen ihrer Eindeutigkeit auch die Prüfung der Vollständigkeit und Korrektheit eines Modells [2], was sinnvollerweise beim Einlesen ebenfalls geschieht.

Im allgemeinen Fall umfaßt die Beschreibung eines Simulationsmodells die Definition und Parametrierung aller enthaltenen Modellelemente sowie die Festlegung ihrer Struktur und ihres Verhaltens. Wesentlich ist dabei, daß die Menge unterschiedlicher Typen von Modellelementen nicht begrenzt ist. Wegen der Beschränkung auf die Simulation von Materialflußsystemen konnte diese Menge der Modellelementtypen im hier vorgestellten Gesamtkonzept jedoch begrenzt werden [3]. Struktur und Verhalten dieser Modellelementtypen sind (als Bestandteil des Werkzeugs) vorgegeben [4] und müssen daher nicht mit der Modellbeschreibung gespeichert werden. Diese umfaßt damit hier nur (noch) die Definition und Parametrierung der im Modell enthaltenen Instanzen.

---

[1] Eschenbacher: *Entwurf und Implementierung einer formalen Sprache zur Beschreibung dynamischer Modelle*; S. 90

[2] ebd.

[3] Diese Menge der Modellelementtypen umfaßt hier die in den Kapiteln 13 bis 18 vorgestellten *Places*, *Movers*, *SingleUnits*, *Workplans*, etc.

[4] Das Verhalten kann (wie in Kapitel 19 beschrieben) aufgabenbezogen angepaßt werden.

Zur Handhabung von Modellbeschreibungen definieren allgemeine Simulationswerkzeuge typischerweise eine Beschreibungssprache und stellen einen passenden Compiler bereit [1]. Wegen der erreichten Vereinfachung der Modellbeschreibungen wurde hier von dieser Vorgehensweise abgewichen und statt dessen ein Konzept zur Beschreibung der Modelle in einem speziellen ASCII-Format, nämlich in Form von C++-Quellcode entwickelt. So erübrigt sich bei der Werkzeugimplementierung die Definition einer Beschreibungssprache und die Programmierung des zugehörigen Compilers. Aus Benutzersicht entfällt das Erlernen der Beschreibungssprache [2]. Auch gegenüber dem bei bausteinbasierten Simulationswerkzeugen verbreiteten Ansatz, ein spezielles Datenformat festzulegen und passende Einleseroutinen vorzuhalten, ergibt sich ein Vorteil, da die Einhaltung der Formatkonventionen beim Einlesen nicht geprüft werden muß. Im hier entworfenen Konzept wird dies implizit durch den (vorgeschalteten) C++-Compiler garantiert.

Die Realisierung des gefundenen Ansatzes zur Speicherung von Modellbeschreibungen erfordert für alle enthaltenen permanenten Objekte die Verfügbarkeit von Operationen (bzw. Funktionen) zum Ausschreiben und Einlesen (Wiederherstellen). Die beim Speichern des Modells benutzten Schreibfunktionen erzeugen C++-Code, der im wesentlichen aus einer Folge von Aufrufen der Lesefunktionen besteht, die ihrerseits aus den jeweiligen Aufrufparametern je ein Modellelement erzeugen. Um die Ausführung zu ermöglichen und den Code (und damit die Modellbeschreibung) angemessen zu strukturieren, sind die genannten Aufrufe wiederum in Funktionen zusammengefaßt. Jeder dieser Funktionen obliegt die Erzeugung aller Modellelemente eines bestimmten Typs. So enthält z.B. die Funktion *initSingleUnits ()* die Aufrufe zur Erzeugung aller Einzelkomponenten eines Modells und die Funktion *initWorkplans ()* die Aufrufe zur Erzeugung aller Arbeitspläne. Anhand dieser beiden Funktionen soll das Verfahren im folgenden erläutert werden.

Zunächst soll die Funktion *initSingleUnits ()*, d.h das Abspeichern bzw. Wiederherstellen von Einzelkomponenten diskutiert werden. Als Beispiel sei ein Modell angenommen, das lediglich ein entsprechend der Beschreibung aus Kapitel 13.4.3 modelliertes Verteilfahrzeug enthält. Bild 78 zeigt die (Version der) Funktion *initSingleUnits ()*, die beim Speichern dieses Modells erzeugt würde. Die Ausführung der Funktion *initSingleUnits ()* beim Laden bzw. Wiederherstellen des Modells läßt das Verteilfahrzeug neu entstehen. Dies geschieht in drei Schritten.

---

[1] So entstanden Simulationssprachen wie z.B. GPSS oder Simula und auch Eschenbachers Modellbeschreibungssprache MDL.

[2] Die statt dessen benötigten C++-Kenntnisse sind für den Umgang mit den in Kapitel 19 beschriebenen Modellprogrammbibliotheken ohnehin erforderlich und implizieren daher keinen zusätzlichen Lernaufwand.

Im ersten Schritt wird die Funktion *buildSingleUnit ()* aufgerufen. Sie erzeugt eine Einzelkomponente (d.h. ein *SingleUnit*-Objekt [1]) mit dem Namen “Wagen” (1. Parameter).

```
void initSingleUnits ()
 {
 buildSingleUnit ("Wagen", /* Building */ NULL,
                    Location (/*Pos*/ Vec3D (1, 1, 1),
                              /* X */ Vec3D (1, 0, 0),
                              /* Z */ Vec3D (0, 0, 1)),
                    /* Capacity */ 1, "defaultSingleUnitFunc",
                    "defaultPlaceSelectFunc", "defaultPartHandleFunc",
                    /* Pattern */ 1, “WagenSchienen");
 buildTranslator ("Wagen", /* inMulti */ 0, 1 /* m/s */,
                   Location (/*Pos*/ Vec3D (2, 0, 0),
                             /* X */ Vec3D (1, 0, 0),
                             /* Z */ Vec3D (0, 0, 1)),
                   /* Pattern */ 2, “WagenKasten", “WagenRaeder”);
 buildPointPlace ("Wagen", /* ChainMode */ 0,
                    Location (/*Pos*/ Vec3D (0, 0, 0),
                              /* X */ Vec3D (0, 1, 0),
                              /* Z */ Vec3D (0, 0, 1)),
                    /* PStart */ Vec3D (-1, 0, 0), /* PEnd */ Vec3D (1, 0, 0),
                    /* TakeTime (s) */ 4, /* GiveTime (s) */ 0,
                    /* Pattern */ 1, “WagenPlatz”);
 } // void initSingleUnits ()
```

Bild 78: Funktion *initSingleUnits ()* zum Beispielmodell mit einem Verteilfahrzeug

Dieser Name dient als Identifikationsmerkmal das diese Einzelkomponente von allen anderen unterscheidet [2]. Der Wagen ist nicht in ein Gebäude [3] eingeordnet (2. Parameter, anderenfalls wäre hier der Name des Gebäudes angegeben), seine Position und Ausrichtung (d.h. die als 3. Parameter angegebene (aus drei Vektoren bestehende) *Location* [4]) sind also auf das Modell- bzw. Weltkoordinatensystem bezogen. Der beschriebene Wagen kann zu jedem Zeitpunkt nur ein Teil behandeln, er hat die Kapazität 1 (4. Parameter). Danach sind die Namen der zur Kontrolle der Einzelkomponente zu verwendenden Steuerungsfunktion (5. Parameter), Platzwahlfunktion (6. Parameter) und Teilebehandlungsfunktion (7.Parameter)

---

[1] vgl. Kapitel 13.3

[2] vgl. Kapitel 16.4

[3] vgl. Kapitel 16.3

[4] vgl. Kapitel 11.1.1

angegeben [1], im Beispiel werden also die jeweiligen Standardfunktionen benutzt. Die folgende 1 (8. Parameter) spezifiziert die Anzahl der der Einzelkomponente (bzw. ihrem *GeoObject*) zugeordneten Konturelemente (*Pattern*) [2], deren Namen anschließend (9. und evtl. folgende Parameter) aufgeführt sind. Im Beispiel ist dem Wagen also einzig das Konturelement "WagenSchienen" zugeordnet.

Im zweiten Schritt wird die Funktion *buildTranslator ()* aufgerufen. Sie erzeugt ein *Translator*-Objekt [3]) und fügt dieses in den Bewegungsapparat der *SingleUnit* mit dem Namen "Wagen" (1. Parameter) ein. Der beschriebene *Translator* ist nicht in einen *MultiMover* [4] eingeordnet (2. Parameter, anderenfalls wäre hier eine 1 angegeben). Er führt seine Bewegungen mit einer Geschwindigkeit von 1 m/s aus (3. Parameter). Seine Position und Ausrichtung (d.h. die als 4. Parameter angegebene *Location*) sind auf das übergeordnete Koordinatensystem, d.h. hier auf das der *SingleUnit* "Wagen" bezogen. Die folgende 2 (5. Parameter) spezifiziert die Anzahl der dem *Translator* (bzw. seinem *GeoObject*) zugeordneten Konturelemente, deren Namen anschließend (6. und evtl. folgende Parameter) aufgeführt sind. Im Beispiel sind dem *Translator* also die Konturelemente "WagenKasten" und "WagenRaeder" zugeordnet.

Im abschließenden dritten Schritt wird die Funktion *buildPointPlace ()* aufgerufen. Sie erzeugt einen punktförmigen Platz (d.h. ein *PointPlace*-Objekt [5]) und ordnet diesen in die Platzliste der Einzelkomponente mit dem Namen "Wagen" (1. Parameter) ein. Der Platz wird nicht in Ketten- sondern in Staucharakteristik betrieben (2. Parameter, anderenfalls wäre hier eine 1 angegeben). Seine Position und Ausrichtung (d.h. die als 3. Parameter angegebene *Location*) sind auf das übergeordnete Koordinatensystem, d.h. hier auf das der Platzliste bzw. des "obersten" *Movers* im Bewegungsapparat der *SingleUnit* "Wagen" bezogen. Die Förderrichtung des beschriebenen Platzes ist hier also quer zur Bewegungsrichtung des (in der Struktur unterhalb eingeordneten) *Translators* (entsprechend der Darstellung in Bild 53). Anschließend sind Eingangs- und Ausgangspunkt des Platzes (4. und 5. Parameter) als Vektoren relativ zum Platzkoordinatensystem angegeben. Als technische Parameter werden danach die Übernahme- bzw. Übergabezeit des Platzes in Sekunden gesetzt (6. und 7. Parameter). Die folgende 1 (8. Parameter) spezifiziert die Anzahl der dem Platz (bzw. seinem *GeoObject*) zugeordneten Konturelemente, deren Namen anschließend (9. und evtl. folgende Parameter) aufgeführt sind. Im Beispiel ist dem Platz also nur die Kontur "WagenPlatz" zugeordnet.

---

[1] vgl. Kapitel 19.2.2, 19.2.3 bzw. 19.2.1

[2] vgl. Kapitel 11.1.1

[3] vgl. Kapitel 13.2

[4] vgl. Kapitel 16.3

[5] vgl. Kapitel 13.1

Nach der Ausführung dieser Schritte (bzw. Funktionen) ist das Verteilfahrzeug vollständig aufgebaut und “betriebsbereit”. In der Funktion *initSingleUnits ()* könnte sich die Erzeugung der nächsten Einzelkomponente anschließen.

In Ergänzung der erwähnten *build... ()*-Funktionen existieren weitere, die die Erzeugung bzw. Wiederherstellung von *Rotators* und *MultiMovers* sowie von *LinearPlaces*, *CirclePlaces*, *PiecePlaces* und *MultiPlaces* beschreiben bzw. durchführen. Mit diesen Funktionen kann die Struktur beliebig aufgebauter *SingleUnit*-Objekte gespeichert und wiederhergestellt werden.

Neben der Funktion *initSingleUnits ()* zur Erzeugung der Einzelkomponenten werden beim Speichern eines Modells vergleichbare Funktionen *initUnitGroups ()*, *initControls ()* und *initBuildings ()* ausgeschrieben, die beim Laden des Modells die Erzeugung der darin enthaltenen Komponentengruppen, Steuerungen bzw. Gebäude besorgen.

```
void initWorkplans ()
  {
  buildWorkplan (“Arbeitsplan A”);
  buildOperation (“Arbeitsplan A”, “Aufspannen”, “Spannplatz”, 12.5 /*s*/);
  buildOperation (“Arbeitsplan A”, “Bohren”, “Gruppe Bohren 1”, 90 /*s*/);
  buildOperation (“Arbeitsplan A”, “Prüfen”, “Prüfplatz Bohren”, 25 /*s*/);
  buildOperation (“Arbeitsplan A”, “Abspannen”, “Spannplatz”, 12.5 /*s*/);

  buildWorkplan (“Arbeitsplan B”);
  buildOperation (“Arbeitsplan B”, “Aufspannen”, “Spannplatz”, 12.5 /*s*/);
  buildOperation (“Arbeitsplan B”, “Fräsen”, “Gruppe Fräsen 1”, 180 /*s*/);
  buildOperation (“Arbeitsplan B”, “Prüfen”, “Prüfplatz Fräsen”, 60 /*s*/);
  buildOperation (“Arbeitsplan B”, “Abspannen”, “Spannplatz”, 12.5 /*s*/);
  }  // void initWorkplans ()
```

Bild 79: Funktion *initWorkplans ()* zum Beispielmodell mit zwei Teilen “A” bzw. “B”

Als zweites Beispiel wird im folgenden die Funktion *initWorkplans ()* vorgestellt, die das Speichern bzw. Wiederherstellen von Arbeitsplänen besorgt. Im angenommenen Modell werden zwei Teile “A” bzw. “B” nach entsprechenden Arbeitsplänen bearbeitet, die entsprechend der Beschreibung aus Kapitel 15 modelliert wurden. An den Teilen “A” sind Bohrungen anzubringen, die Teile “B” sind zu fräsen. Alle Teile sind zunächst aufzuspannen, nach der Bearbeitung folgt je ein Prüfvorgang bevor die Teile wieder abgespannt werden. Bild 79 zeigt die (Version der) Funktion *initWorkplans ()*, die beim Speichern dieses Modells erzeugt würde. Die Ausführung der Funktion *initWorkplans ()* beim Laden bzw. Wiederherstellen des Modells läßt die Arbeitspläne neu entstehen. Dies geschieht jeweils in mehreren Schritten.

Im ersten Schritt wird die Funktion *buildWorkplan ()* aufgerufen. Sie erzeugt einen Arbeitsplan (d.h. ein *Workplan*-Objekt [1]) mit dem Namen "Arbeitsplan A". Dieser Name dient als Identifikationsmerkmal das diesen Arbeitsplan von allen anderen eindeutig unterscheidet. Im zweiten Schritt wird die Funktion *buildOperation ()* aufgerufen. Sie erzeugt ein *Operation*-Objekt [2]) und fügt dieses (am Ende) in die Bearbeitungsvorgangsfolge des Arbeitsplans mit dem Namen "Arbeitsplan A" (1. Parameter) ein. Der Bearbeitungsvorgang hat den Namen "Aufspannen" (2. Parameter). Der Bearbeitungsort ist die Komponente [3] mit dem Namen "Spannplatz" (3. Parameter). Die als 4. Parameter angegebene Bearbeitungszeit beträgt hier 12.5 s.

In weiteren Schritten wird wiederum jeweils die Funktion *buildOperation ()* zur Erzeugung der weiteren Bearbeitungsvorgänge des Arbeitsplans "A" aufgerufen. Nach der Ausführung dieser Schritte ist der Arbeitsplan vollständig aufgebaut und benutzbar. In der Funktion *initWorkplans ()* schließt sich die Erzeugung des nächsten Arbeitsplans an. Der "Arbeitsplan B" wird in gleicher Weise wie der "Arbeitsplan A" erzeugt.

Neben den oben angesprochenen Funktionen *initSingleUnits ()*, *initUnitGroups ()*, *initControls ()* und *initBuildings ()* zur Erzeugung der statischen Systemkomponenten [4]und der hier vorgestellten Funktion *initWorkplans ()* werden beim Speichern eines Modells vergleichbare Funktionen zur Erzeugung bzw. Wiederherstellung der übrigen Modellelemente ausgeschrieben. So besorgen z.B. die Funktion *initRepeaters ()* die Wiederherstellung aller Ankunftsströme und Störmodelle [5] und die Funktion *initPatterns ()* die Wiederherstellung aller in Teilen oder Komponenten verwendeten Konturelemente [6].

Es liegt damit eine vollständige Modellbeschreibung in Form von C++-Quellcode vor. Vor ihrer Nutzung zum Laden bzw. Wiederherstellen des Modells muß sie lediglich in ausführbaren Maschinencode übersetzt und zu einer "shared library" [7] gebunden werden.

---

[1] vgl. Kapitel 15

[2] ebd.

[3] vgl. Kapitel 16.4

[4] vgl. Kapitel 16

[5] vgl. Kapitel 18

[6] vgl. Kapitel 11.1.1

[7] Es wird also zur technischen Realisierung der gleiche Weg wie bei der Realisierung der Modellprogrammbibliothek beschritten (vgl. Kapitel 19.1).

Die eine Modellbeschreibung enthaltende shared library im folgenden als Modellstrukturbibliothek bezeichnet. Sie wird jeweils beim Laden eines Modells in das Simulationswerkzeug gesucht und geöffnet [1]. Die einzelnen Funktionen wie *initSingleUnits ()* und *initWorkplans ()* darin werden dann gesucht und ausgeführt und damit das Modell wiederhergestellt. Wie bei der Modellprogrammbibliothek kann das Simulationswerkzeug seine Benutzer weiter unterstützen, indem es die Modellstrukturbibliothek vor dem Öffnen aktualisiert, d.h. bei Bedarf neu übersetzt und bindet [2].

Im übrigen ist bei der Ausführung der einzelnen Funktionen der Modellstrukturbibliothek auf die Einhaltung einer vorgegebenen Reihenfolge zu achten, die sich aus den wechselseitigen Abhängigkeiten der Modellelemente ergibt. Beispielsweise müssen die statischen Systemkomponenten vor den Arbeitsplänen wiederhergestellt werden (d.h. die Funktionen *initSingleUnits ()* und *initUnitGroups ()* sind vor der Funktion *initWorkplans ()* aufzurufen), damit beim Anlegen der Bearbeitungsvorgänge geprüft werden kann, ob die als Bearbeitungsorte angegebenen Systemkomponenten im Modell auch wirklich existieren.

Insgesamt wird durch den vorgestellten Ansatz erreicht, daß beim Einlesen bzw. Wiederherstellen eines Modells vom Werkzeug nur noch die semantische Korrektheit geprüft werden muß. Die Prüfung auf syntaktische Fehlerfreiheit wird vom Compiler bei der Erzeugung bzw. Aktualisierung der Modellstrukturbibliothek durchgeführt. Dies vereinfacht die Implementierung der Einlesefunktionen erheblich.

Da die technische Realisierung in der gleichen Weise wie die der Modellprogrammbibliothek erfolgt, wird außerdem weder das Werkzeug (intern) noch seine Handhabung verkompliziert. Nach außen ist die Arbeitsweise an beiden Punkten gleich und bei der Implementierung können die entsprechenden Programmteile wiederverwendet werden.

---

[1] vgl. Kapitel 21

[2] ebd.

21 Integrierendes Softwarewerkzeug

In den Kapiteln 10 bis 20 wurden (Teil-) Konzepte für das logische Modell eines objektorientierten Simulationswerkzeugs diskutiert. Die Kapitel 19 und 20 berührten mit der Bündelung der modellabhängigen Steuerungsalgorithmen in der Modellprogrammbibliothek bzw. der Zusammenfassung der Modellbeschreibung in der Modellstrukturbibliothek zusätzlich Aspekte der Modulstruktur und damit des physikalischen Modells.

Im folgenden wird ein Konzept für ein integrierendes Softwarewerkzeug zum Umgang mit Modellen als Ganzes vorgestellt. Dabei wird unterstellt, daß die Modelle in der in den vorangegangenen Kapiteln entwickelten Weise aufgebaut sind. Die Darstellung bleibt auf allgemeiner Ebene und verzichtet auf die Erörterung von Details beispielsweise der Benutzeroberfläche. Ausgehend von grundsätzlichen Überlegungen zur Vorgehensweise bei der Erledigung alltäglicher Aufgaben im Umgang mit Simulationsmodellen werden vielmehr die grundlegenden Beziehungen zwischen dem Softwarewerkzeug und den Modellen entwickelt und daraus ein Komponentenkonzept als weiteres Element des physikalischen Modells eines Simulationswerkzeug abgeleitet.

In Kapitel 8.1.3 wurde für Simulationswerkzeuge eine grafische Bedienoberfläche gefordert. Sie bildet die Schnittstelle, über die Anwenderinnen mit den Modellen (bzw. dem Werkzeug) interagieren und bietet alle Funktionalitäten, um Modelle entwerfen, modifizieren, speichern, laden und mit ihnen experimentieren zu können.

Statt also den Aufbau des Werkzeugs an Funktionen wie Entwurf und Experiment zu orientieren (und womöglich auf mehrere Programme zu verteilen), wird so eine “modell-

zentrierte" Arbeitsweise unterstützt bzw. erst ermöglicht, die vergleichbaren Ansätzen, wie beipielsweise der Arbeitsweise in modernen Textverarbeitungssystemen entspricht. Dabei besteht auch die Möglichkeit, mehrere Modelle gleichzeitig zu bearbeiten, ebenso wie in einer Textverarbeitung mehrere Dokumente gleichzeitig bearbeitet werden können. Aus der Anwenderperspektive gesehen erlaubt dies unter anderem, in einfachster Weise auch komplexe Objekte wie z.B. Anlagenkomponenten von einem Modell in ein anderes zu verschieben oder zu kopieren [1].

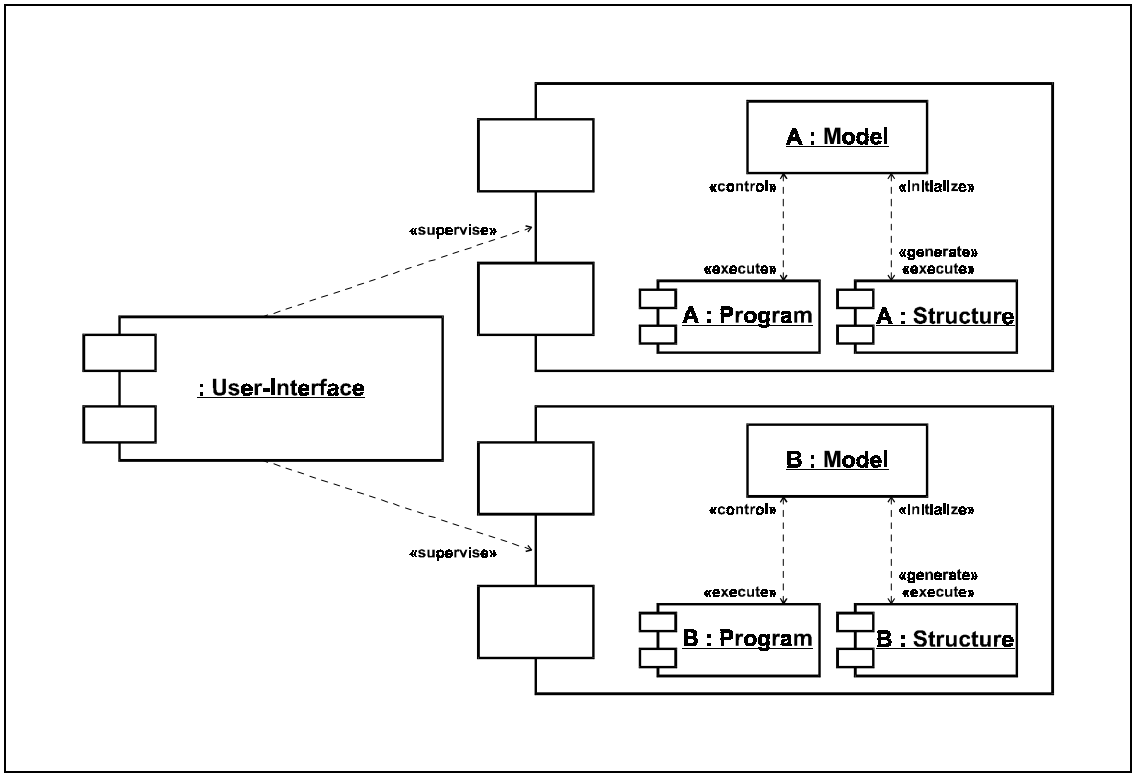

Bild 80: Laufzeitkomponenten des Simulationswerkzeugs (2 Modelle)

Die Bearbeitung mehrerer Modelle schließt offensichtlich auch den Fall der gleichzeitigen Durchführung mehrerer Experimente ein. Hierfür ist es notwendig, daß jedes im Werkzeug in Bearbeitung befindliche Modell eine eigenständige Laufzeitkomponente ist. Dies impliziert, daß alle relevanten Datenstrukturen (als Beispiel sei der Terminkalender des Zeitmechanismus [2] genannt) nicht nur einmal im Werkzeug, sondern einmal je Modell vorhanden sind.

Um eine wechselseitige Beeinflussung auszuschließen, ist es weiter notwendig, daß jedes Experiment bzw. Modell in einem eigenen (Betriebssystem-) Thread ausgeführt wird, so daß

---

[1] Durch diese Möglichkeit kann z.B. das in Kapitel 13 vorgestellte Konzept zur Modellierung von Fördertechnikkomponenten erst praktisch sinnvoll eingesetzt werden (vgl. Kapitel 13.3).

[2] vgl. Kapitel 12

ihre Abarbeitung (wie gewünscht) unabhängig voneinander ist. Da jedoch alle Threads eines Programms im selben Adreßraum laufen, sind die oben beschriebenen Kopier- und Verschiebemöglichkeiten für Objekte einfach zu realisieren [1].

Da das Werkzeug auch ohne Modell gestartet werden kann (beispielsweise um ein neues zu erzeugen), ist die Oberfläche von den Modellen ebenfalls (weitestgehend) unabhängig [2]. Daher ist auch sie eine getrennte Laufzeitkomponente mit eigenem Thread.

Aus diesen Überlegungen ergibt sich insgesamt das in Bild 80 dargestellte grobe Komponentenkonzept für ein Simulationswerkzeug. Hinsichtlich der Beziehungen zwischen den dargestellten Komponenten ist zu erkennen, daß (wie auch im Text dargelegt wurde) die Oberfläche die Modelle "überwacht", während diese voneinander unabhängig sind. Weiter sind im Bild noch die innerhalb der Modelle bestehenden Beziehungen zwischen dem Modell als Ganzem und den in vorangegangenen Kapiteln vorgestellten physikalischen Designeinheiten Modellstruktur- und Modellprogrammbibliothek dargestellt.

Wie in Kapitel 20 erläutert, enthält die Modellstrukturbibliothek die Beschreibung des Modells in Form von C++-Funktionen. Diese bauen das Modell auf, wenn sie beim Laden ausgeführt werden. Zuvor wird die Modellstrukturbibliothek vom Werkzeug erforderlichenfalls aktualisiert, d.h. neu übersetzt und gebunden. Hierzu können allgemein verfügbare Softwaretools wie z.B. *make* benutzt werden. Die Aktualisierung der Modellstrukturbibliothek aus dem Werkzeug heraus verbessert die Benutzerunterstützung, da dieser nicht selbst (und rechtzeitig) an diesen Vorgang denken muß. Beim Speichern des Modells werden die Funktionen der Modellstrukturbibliothek neu geschrieben.

In gleicher Weise übernimmt das Werkzeug die Aktualisierung der Modellprogrammbibliothek. Wie in Kapitel 19 ausgeführt, enthält sie Steuerungsalgorithmen, die, abhängig von ihrem jeweiligen Verwendungszweck, zu bestimmten Zeitpunkten während der Experimentdurchführung aufgerufen werden. Diese Algorithmen kontrollieren das Verhalten des Modells bzw. seiner Elemente. Die Aktualisierung der Modellprogrammbibliothek findet dementsprechend jeweils beim Start eines Experiments statt.

---

[1] Die alternative Möglichkeit, je Modell einen Prozeß bzw. ein Programm vorzusehen, impliziert das Vorhandensein getrennter und durch das Betriebssystem geschützter Adreßräume, wodurch sich das Kopieren bzw. Verschieben von Objekten erheblich verkompliziert.

[2] Da die Oberfläche letztlich zur Bearbeitung der Modelle dient, ist die Unabhängigkeit nicht vollständig.

22 Einsatz in der Anlagensteuerung

Der Einsatz der Simulation in der Betriebsphase von Anlagen ist bis dato auf wenige Fälle beschränkt obwohl die VDI-Richtlinie zur Materialflußsimulation einige entsprechende Nutzungsmöglichkeiten nennt [1]. Das durchaus vorhandene Interesse an diesem Bereich resultiert aus dem naheliegenden Wunsch, die mit erheblichem Aufwand entwickelten und viel Know-how enthaltenden Simulationsmodelle intensiver und vielfältiger zu nutzen und damit weitere Kostenpotentiale zu erschließen.

In diesem Kapitel wird dargestellt, wie sich in der Literatur beschriebene Ansätze zur Nutzung der Simulation in Verbindung mit anderen Systemen der Anlagensteuerung auf der Basis der in den vorangegangenen Kapiteln entwickelten Konzepte umsetzen lassen. Da anderenfalls keine neue Qualität eingeführt würde, richtet sich der Blick dabei nur auf Ansätze, in denen Simulationsmodelle bzw. -werkzeuge mit anderen Bestandteilen der Anlagensteuerung integriert sind, d.h. mit diesen in irgendeiner Form Daten austauschen.

In den nachfolgenden Abschnitten wird je ein Ansatz behandelt. Neben einer kurzen Einführung werden jeweils Beispiele für auszutauschende Daten gegeben. Weiter werden die möglichen Realisierungen des Datenaustauschs und sonstige besondere Erfordernisse angesprochen. Schließlich wird je eine mögliche Umsetzung vorgestellt.

[1] vgl. *VDI-Richtlinie 3633 Blatt 1*; Bild 3 u. S. 5

22.1 PPS- bzw. Leitstandsunterstützung

Möglichkeiten zum Einsatz der Simulation als "decision support tool" für Produktionsplaner und Leitstandspersonal beschreiben Harder und Ortmann [1]. Ihre Motivation ist die Erkenntnis, daß die in PPS-Systemen und Leitständen implementierten Verfahren der Auftragsreihenfolgeplanung den heutigen Anforderungen nicht mehr genügen [2].

Diese Verfahren rechnen statisch unter Verwendung vorgegebener (mittlerer) Durchlaufzeiten für die Ressourcen (Komponenten, Kostenstellen, etc.) und eventuell noch (ebenfalls vorgegebener) Übergangszeiten (aus Transportzeitmatrizen) [3]. Da die realen Kapazitäten der Ressourcen unberücksichtigt bleiben, können Engpässe nicht gezielt, sondern nur pauschal behandelt werden. Die Folge sind verfrühte Materialbereitstellungen, wachsende Warteschlangen vor Engpaßkapazitäten und damit insgesamt zu hohe Bestände [4]. Dies führt weiter zu höheren Durchlaufzeiten und Terminverzögerungen [5].

Zum Wesen der Simulation gehört es, die in den Anlagen ablaufenden Prozesse dynamisch nachzubilden [6]. Sie berücksichtigt also reale Kapazitäten und Transportzeiten und ist daher geeignet, die Schwächen der statischen Verfahren zu überwinden. Im hier behandelten Ansatz wird daher ein Simulationsmodell verwendet, um ausgehend vom Ist-Zustand und einer vorgesehenen Folge von Auftragsfreigaben Aussagen über die sich der Anlage ergebenden Verhältnisse zu gewinnen. Das Simulationsmodell ist bei jedem Berechnungsdurchgang zunächst mit dem Ist-Zustand der Anlage abzugleichen. Es muß weiter die vorgesehenen Freigaben übernehmen. Als Ergebnisse des Simulationsexperiments können dann z.B. Start- und Endtermine für einzelne Arbeitsgänge sowie Auslastungen von und Bestände vor Ressourcen ermittelt werden. Aus der Gegenüberstellung mehrerer Läufe mit unterschiedlichen Freigabeplänen kann ein "pragmatisch bester" Plan ermittelt und dieser anschließend umgesetzt werden.

---

[1] Harder: *Simulationsgestützer Leitstand* und Ortmann: *Zeitdynamische Simulation - neue Qualität für PPS-Systeme und Leitstände*

[2] vgl. Ortmann: *Zeitdynamische Simulation - neue Qualität für PPS-Systeme und Leitstände*; S. 130

[3] vgl. Harder: *Simulationsgestützer Leitstand*; S. 120

[4] ebd.

[5] vgl. Kapitel 4.2

[6] vgl. Kapitel 7.1

Bild 81 zeigt die Kopplung von Leitstand bzw. PPS-System und Simulationsmodell schematisch. Die Systeme sind offline verbunden. Sie tauschen die Daten also nicht direkt aus, sondern nutzen einen Zwischenspeicher. Hierfür kommen ASCII-Dateien oder PPS- bzw. Unternehmensdatenbanken in Frage [1].

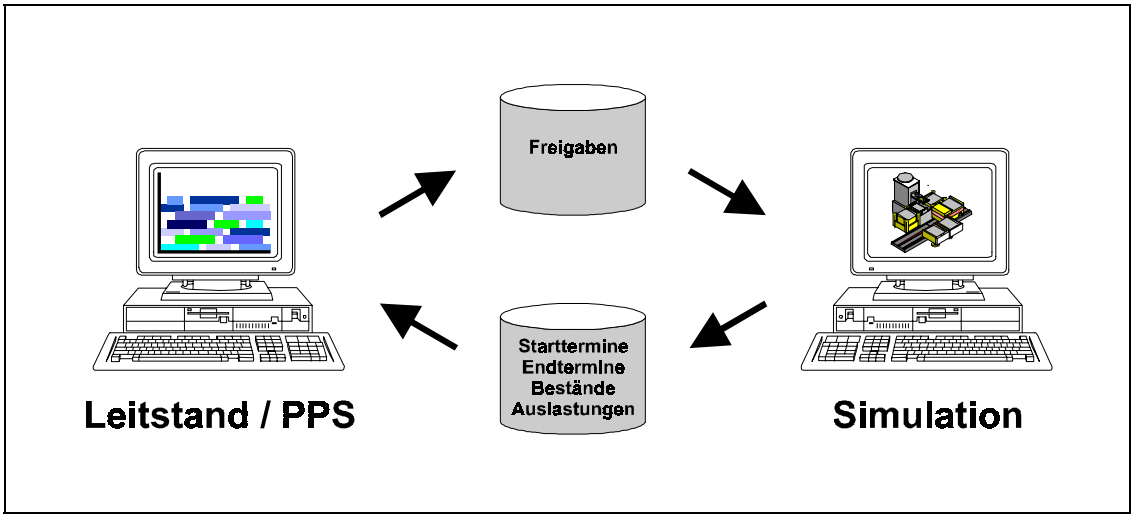

Bild 81: Simulationsmodell als decision support tool - Schema

Die Umsetzung dieses Ansatzes auf der Basis der in den vorangegangenen Kapiteln dargestellten Konzepte ist möglich. Grundlage ist ein zuvor erstelltes Modell der betrachteten Anlage, das die erforderlichen Arbeitspläne und das Layout enthält [2]. Für den zu Experimentbeginn erforderlichen Zustandsabgleich und die Übernahme der Freigabepläne ist eine angepaßte Funktion *initModelLibrary ()* [3] in der Modellprogrammbibliothek erforderlich. Aus je passenden anderen Funktionen dieser Bibliothek heraus können die zurückzugebenden Daten protokolliert werden. Die Standardbliothek der Programmiersprache C++ unterstützt den Zugriff auf ASCII-Dateien, so daß diese als Basis des Datenaustauschs problemlos gelesen und geschrieben werden können. Wegen der Verbreitung von C++ (bzw. C) sind für die als Hardware- bzw. Betriebssystembasis [4] in Frage kommenden Plattformen auch Softwarebibliotheken für den Zugriff auf Datenbanken verfügbar. Die von diesen bereitgestellten Funktionalitäten können aus der Modellprogrammbibliothek heraus genutzt werden, so daß der Datenaustausch auch über eine Datenbank erfolgen kann.

---

[1] vgl. Harder: *Simulationsgestützer Leitstand*; S. 125

[2] Diese Teile des Modells ändern sich typischerweise nur selten (jedenfalls im Vergleich zu den Aufträgen) und werden daher hier als statisch betrachtet.

[3] vgl. Kapitel 19.2.8

[4] vgl. Kapitel 8.2.2

22.2 Leitstandsentwicklung und Schulung

Die Möglichkeit, Simulationsmodelle bei der Entwicklung von Leitstandssoftware und zur Schulung von Leitstandspersonal einzusetzen, beschreibt Noche [1]. Der Nutzen dieses Ansatzes besteht darin, die Funktionalität der Leitstandssoftware bereits vor der Inbetriebnahme der Anlage prüfen (und damit die Inbetriebnahmephase verkürzen) zu können, indem diese in einer Testanordnung durch das Simulationsmodell ersetzt wird.

Sobald der Leitstand funktionsfähig ist, kann an dieser Testanordnung (also ohne die Anlage) die Schulung des Leitstandspersonals beginnen [2]. Das Training an der Testanordnung kann später parallel zum laufenden Betrieb fortgesetzt werden. Auch beim Schulungsaspekt besteht der Nutzen zunächst im Zeitgewinn bis zum Erreichen des Volllastbetriebs. Zusätzlich können gefahrlos und ohne Produktionseinbußen mit dem Personal beipielsweise besondere (vielleicht kritische) Situationen durchgespielt werden.

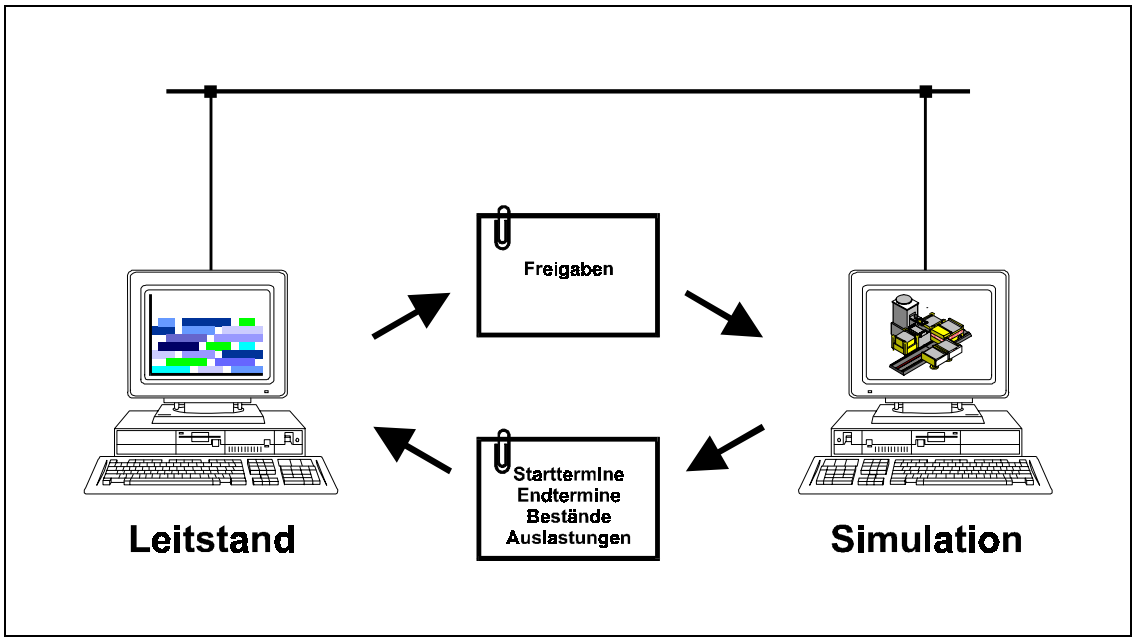

Bild 82: Simulationsmodell für Leitstandstest und Schulung - Schema

[1] Noche: *Kopplung von Simulationsmodellen mit Leitrechnern*

[2] ebd. S. 170 ff.

Bild 82 zeigt die Kopplung von Leitstand und Simulationsmodell schematisch. Die Systeme sind online über ein Netzwerk miteinander verbunden. Sie können Daten direkt austauschen, indem sie sich Nachrichten bzw. “Telegramme” übermitteln.

Aus logischer Sicht erfordert dieser Ansatz vor allem, daß die im Simulationsmodell enthaltene Leitstandsfunktionalität wie in Bild 83 dargestellt softwaretechnisch abgetrennt und damit stillgelegt werden kann. Ihre Aufgaben übernimmt in der vollständigen Testanordnung der “richtige” Leitstand. Stattdessen müssen in das Simulationsmodell Funktionen integriert werden, die es in die Lage versetzen, über das Netzwerk zu kommunizieren, d.h. Nachrichten zu empfangen und zu versenden.

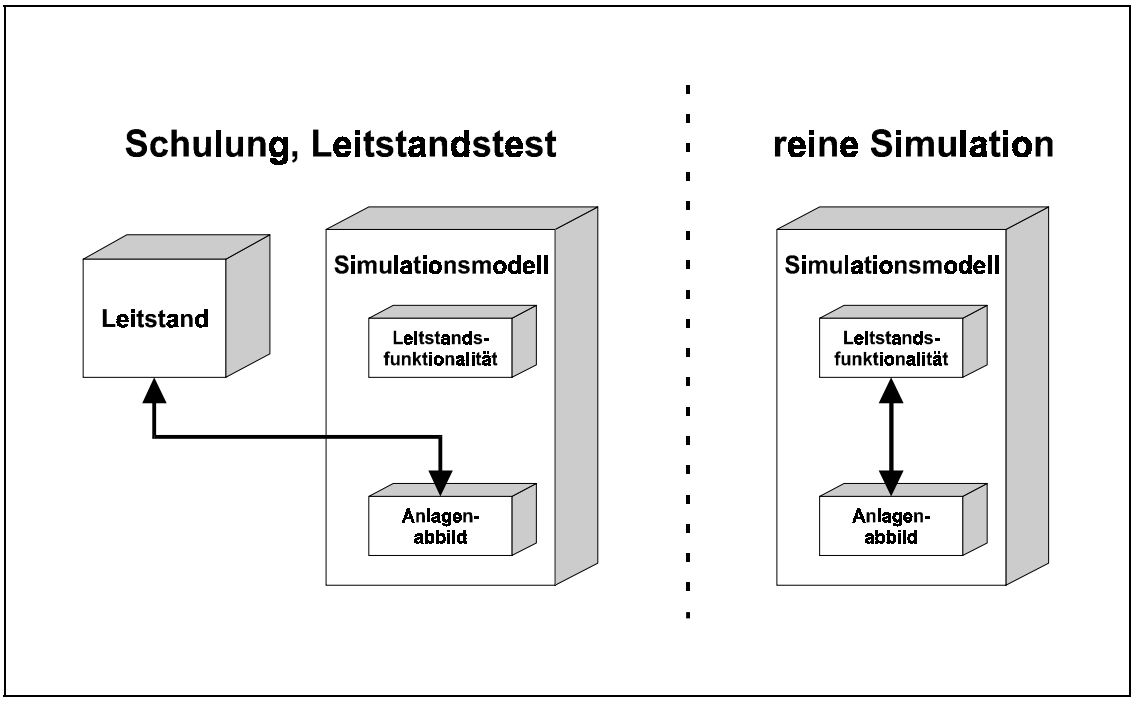

Bild 83: Simulationsmodell für Leitstandstest und Schulung - Logik

Damit können bereits Untersuchungen des Datenverkehrs durchgeführt werden, um beipielsweise sicherzustellen, daß alle verschickten Nachrichten richtig adressiert und aufgebaut sind. Für weitergehende Tests und Schulungen müssen dann zwischen beiden Systemen wie in Bild 82 gezeigt etwa die gleichen Daten wie beim Einsatz des Simulationsmodells als decision support tool ausgetauscht werden. Weiter ist es erforderlich, die virtuelle Zeit, in der das Simulationsmodell arbeitet [1], mit der “echten” Zeit, in der der Leitstand und das daran tätige Personal arbeitet, zu synchronisieren.

[1] vgl. Kapitel 10.2

Auf der Basis der in den vorangegangenen Kapiteln dargestellten Konzepte kann auch dieser Ansatz umgesetzt werden. Die Synchronisation der beiden Zeitsysteme wird zweckmäßig von einer Steuerung, d.h. von einem in das Modell integrierten *Control*-Objekt [1] übernommen. Dessen (in der Modellprogrammbibliothek implementierte) Steuerungskontrollfunktion muß in kurzen Abständen Modell- und Realzeit vergleichen und das Fortschreiten der Modellzeit durch entsprechende Betriebssystemaufrufe (*sleep ()* o.ä.) verzögern, wenn diese der Realzeit vorauseilt.

Der umgekehrte Fall, also das "Nachlaufen" der Modellzeit, ist durch dieses Verfahren nicht korrigierbar. Dann wäre im übrigen die Zeitraffungsqualität des Modells verlorengegangen, was auch seine Eignung für die reine Simulation in Frage stellen würde. Abhilfe könnte z.B. durch Modelländerungen (höheres Abstraktionsniveau) oder durch den Einsatz leistungsfähigerer Hardware geschaffen werden. Angesichts der Leistung heute verfügbarer Rechner darf jedoch davon ausgegangen werden, daß diese Situation in der Praxis nicht auftritt. Dies ist jedenfalls die Erfahrung aus den vom Autor bearbeiteten Projekten.

Die erzielbare Übereinstimmung zwischen den beiden Zeitsystemen ist durch die Auflösung der Systemuhr des verwendeten Rechners determiniert. Da der im Simulatorkern implementierte Zeitmechanismus [2] unverändert bleibt und keine entsprechenden Merkmale aufweist, sind Echtzeitfähigkeiten im Sinne garantierter Reaktionszeiten z.B. auf eingehende Telegramme nicht erreichbar [3]. Für die oben skizzierte Testanordnung sind solche Fähigkeiten allerdings auch nicht erforderlich.

Ein weiteres *Control*-Objekt könnte zyklisch prüfen, ob Nachrichten vom Netzwerk (bzw. Leitstand) eingetroffen sind und erforderlichenfalls deren Auswertung (also z.B. das Erzeugen von Teilen bei Eintreffen einer Freigabe) übernehmen. Aus je passenden anderen Funktionen der Modellprogrammbibliothek heraus können die zurückzugebenden Telegramme verschickt werden.

Wegen der Verbreitung der Programmiersprache C++ (bzw. C) sind für alle verbreiteten Rechnerplattformen Softwarebibliotheken für den Zugriff auf Netzwerke verfügbar. Deren Funktionalitäten können aus der Modellprogrammbibliothek heraus genutzt werden, so daß der Datenaustausch in der oben angedeuteten Weise erfolgen kann.

---

[1] vgl. Kapitel 16.2

[2] vgl. Kapitel 12.4

[3] Sie würden im übrigen ein Echtzeitbetriebssystem voraussetzen (und im Gesamtrahmen ohne ein Netzwerk mit garantierten Laufzeiten wenig Sinn machen).

22.3 Einsatz als Leitstand

Am Fachgebiet Produktionssysteme liegen seit längerem Erfahrungen mit dem Einsatz von Simulationsmodellen als Anlagenleitstand vor [1]. Dieser zunächst in Laborumgebung entwickelte Ansatz wurde inzwischen auch in einem Industrieprojekt umgesetzt und so unter Alltagsbedingungen überprüft [2].

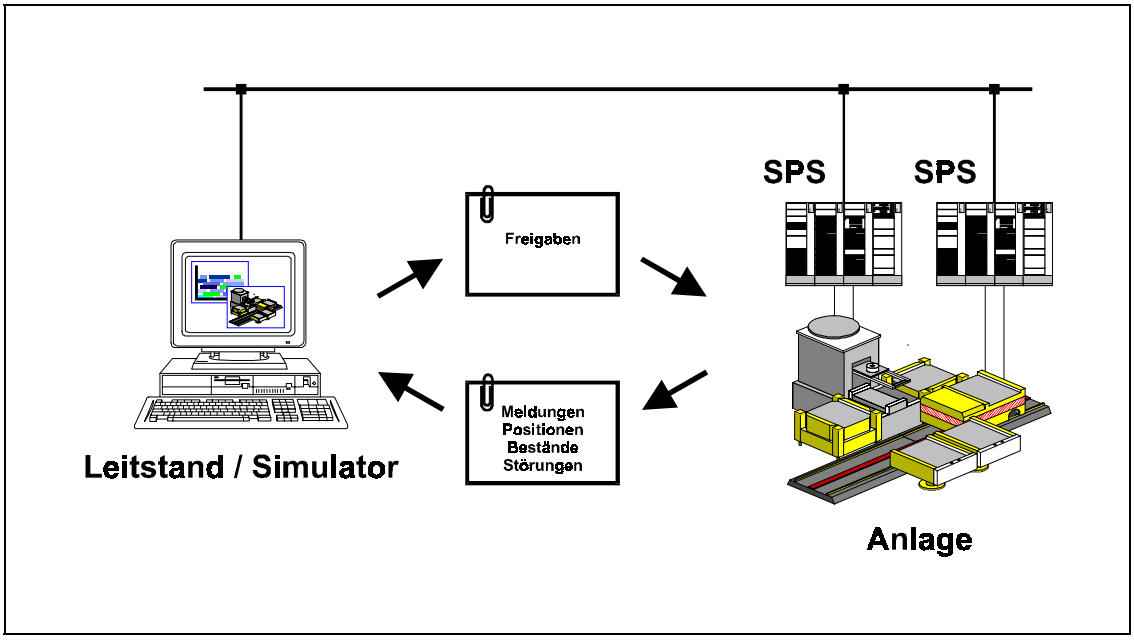

Bild 84: Simulationsmodell als Leitstand - Schema

Der Nutzen des Ansatzes besteht darin, daß das Simulationsmodell bereits eine Prozeßvisualisierung, eine Bedienoberfläche und Funktionalitäten zur Betriebsdatenerfassung und -verarbeitung und damit wesentliche Merkmale einer Leitstandssoftware umfaßt [3]. Auch alle Steuerungsalgorithmen, die in der Planung entwickelt, in das Simulationsmodell integriert und darin getestet wurden, sind bei seiner Verwendung als Leitstand darin vorhanden und unmittelbar einsatzbereit [4].

---

[1] Reinhardt: *Die Kluft zwischen Simulation und Steuerungssoftware*

[2] vgl. “Grundlagen der rechnerintegrierten Fabrik” im Anhang Projekte.

[3] Zunächst fehlen natürlich noch die Funktionen zum Datenaustausch mit der Anlage.

[4] Im “klassischen” Fall getrennter Simulations- und Leitstandssoftware müßten sie neu implementiert werden, wobei Abweichungen aufgrund von Übertragungsverlusten z.B. infolge unzulänglicher Dokumentation auftreten können.

Der Vergleich mit dem in Kapitel 22.2 dargestellten Einsatz des Simulationsmodells zum Test der Leitstandssoftware zeigt, daß im hier vorgestellten Ansatz das Simulationsmodell die dort skizzierte Testanordnung aus Simulator und Leitstand integriert. Sobald das Simulationsmodell funktionsfähig ist, kann es also auch bei diesem Ansatz zur Schulung des Leitstandspersonals vor und während des Betriebs eingesetzt werden.

Für den Einsatz als Leitstand wird das Simulationsmodell mit den Rechnern der Steuerungsebene (also mit Speicherprogrammierbaren Steuerungen, Maschinensteuerungen u.ä.) gekoppelt [1]. Wie in Bild 84 schematisch dargestellt ist, sind die Systeme über ein Netzwerk online miteinander verbunden. Sie können darüber Daten direkt austauschen, indem sie sich Nachrichten bzw. Telegramme übermitteln.

Aus logischer Sicht erfordert dieser Ansatz vor allem, daß, wie bereits angedeutet, in das Simulationsmodell Funktionen integriert werden, die es in die Lage versetzen, über das Netzwerk mit der Anlage bzw. den Rechnern der Steuerungsebene zu kommunizieren, d.h. Nachrichten zu empfangen und zu versenden.

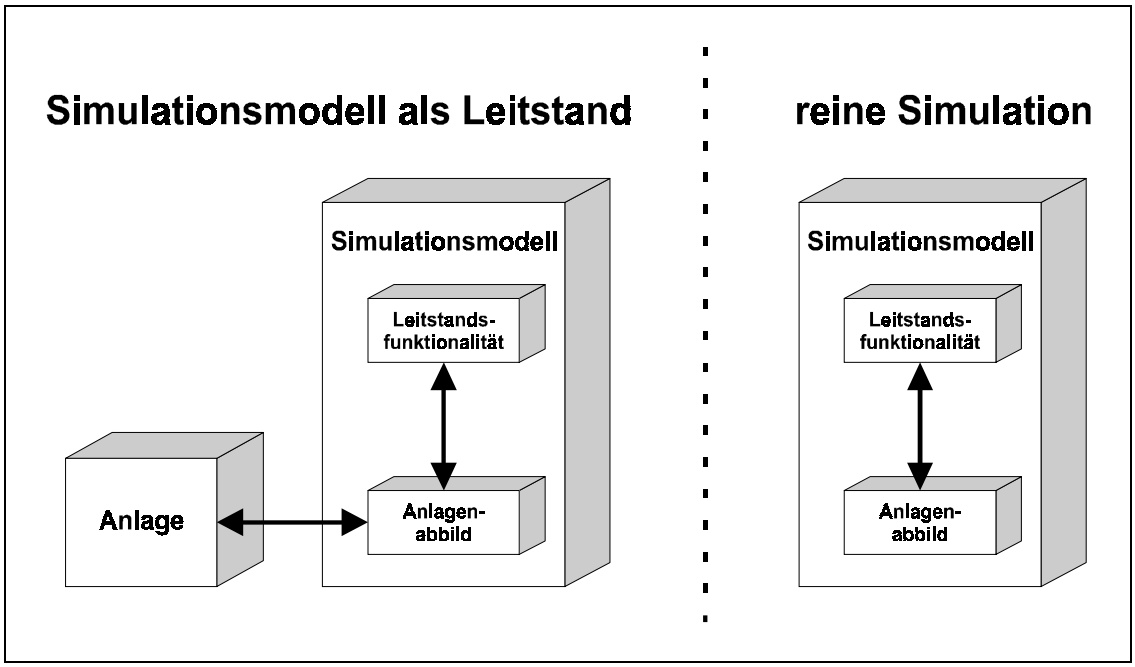

Bild 85: Simulationsmodell als Leitstand - Logik

Für den laufenden Betrieb müssen dann zwischen den Systemen wie in Bild 84 gezeigt wiederum ähnliche Daten wie beim Einsatz des Simulationsmodells zum Test der Leitstandssoftware ausgetauscht werden. Ebenso wie dort ist es erforderlich, die virtuelle Zeit des Simulationsmodells mit der "echten" Zeit (hier der Anlage) zu synchronisieren.

---

[1] vgl. Kapitel 5.4 und darin Bild 10

Auf der Basis der in den vorangegangenen Kapiteln dargestellten Konzepte kann auch dieser Ansatz umgesetzt werden. Die Synchronisation der beiden Zeitsysteme kann wie beim Einsatz des Simulationsmodells zum Test der Leitstandssoftware und unter denselben Randbedingungen erfolgen. Sie wird daher hier nicht weiter beschrieben.

Ein oder mehrere weitere Steuerungsobjekte können zyklisch prüfen, ob Nachrichten vom Netzwerk (d.h. von den Steuerungen) eingetroffen sind und erforderlichenfalls deren Auswertung übernehmen. Beispielsweise kann eine nach Abschluß einer Bearbeitung eintreffende Fertigmeldung an die betroffene Komponente des Simulationsmodells weitergeleitet werden, woraufhin diese die Weiterleitung des Teils zur nächsten Bearbeitungsstation veranlassen kann.

Das Senden von Nachrichten an die Steuerungen kann auch hier direkt aus je passenden Funktionen der Modellprogrammbibliothek heraus erledigt werden.

Wegen der Gleichartigkeit der anzutreffenden Netzwerke können für den Netzwerkzugriff aus den Funktionen der Modellprogrammbibliothek die gleichen Softwarebibliotheken wie beim Einsatz des Simulationsmodells zum Test der Leitstandssoftware genutzt werden.

23 Bewertung und Ausblick

In den vorangegangenen Kapiteln wurde eine Reihe miteinander verbundener Ansätze zur softwaretechnischen Modellierung von Materialflußsystemen und der darin ablaufenden dynamischen Prozesse behandelt.

Die kennzeichnenden Merkmale der dargestellten Ansätze sind

- Modellierung aller Anlagenkomponenten in 3D,
- 3D-Visualisierung auch in mehreren Fenstern mit der Möglichkeit zur Interaktion mit den dargestellten Objekten (z.B. als Grundlage für grafisches Editieren),
- Modellierung beliebiger Fördertechnikkomponenten durch Zusammensetzung aus Grundelementen für Bewegung und Teilebehandlung,
- Trennung von Struktur und Verhalten der Komponenten,
- Schnittstelle für anlagenbezogene Programmierung mit der Möglichkeit der vollständigen Kontrolle des Verhaltens aller Komponenten und
- Offenheit der Schnittstelle durch Verwendung der Programmiersprache C/C++.

Die vorgestellten Konzepte sind insgesamt eine geeignete Basis, um mit darauf aufsetzenden Simulationswerkzeugen die für diese in Kapitel 8.1 genannten Anforderungen erfüllen zu können.

Wesentliche Teile dieser Konzepte wurden in experimentellen Softwaresystemen prototypisch implementiert. Die dabei gewonnenen Erkenntnisse bestätigten die Praxistauglichkeit der diskutierten Ansätze. Sie sind daher zur konzeptuellen Basis des am Fachgebiet Produktionssysteme neu entwickelten Fabriksimulators SIMFLEX/3 geworden.

Für die Zukunft wird die vollständige Implementierung aller beschriebenen Konzepte in einem Simulationswerkzeugs angestrebt, um ihre Leistungsfähigkeit auch in Industrieprojekten in vollem Umfang nutzen zu können.

Quellenverzeichnis

Bildnachweis

Bild 1: Entwicklung der Innovationszyklen beispielhafter Produkte (S. 31)
nach Koether: *Technische Logistik*
Bild 1.4, S. 4

Bild 2: Das Zielsystem der Produktion (S. 40)
nach *VDI-Richtlinie 3633- Blatt 1*
Bild 4, S. 5

Bild 3: Aufbau eines Durchlaufelements (S. 41)
nach Wiendahl: *Belastungsorientierte Fertigungssteuerung*
Bild 3.5, S. 53

Bild 4: Durchlaufdiagramm eines Arbeitsplatzes (S. 42)
nach Wiendahl: *Belastungsorientierte Fertigungssteuerung*
Bild 6.1, S. 206

Bild 5: Prinzipieller Verlauf einer Betriebskennlinie (S. 43)
nach Wiendahl: *Belastungsorientierte Fertigungssteuerung*
Bild 6.2, S. 207

Bild 6: CAD-System mit integrierten Berechnungsverfahren (S. 47)
nach Geitner: *CIM-Handbuch*
o. Nr., S. 201

Bild 7: Funktionen eines PPS-Systems (S. 49)
nach Eversheim: *Organisation in der Produktionstechnik; Band 1*
Bild 3-36, S. 62

Bild 9: CIM-Modell nach Scheer (S. 51)
nach Geitner: *CIM-Handbuch*
Bild 3, S. 52

Bild 10: Strukturebenen der rechnerintegrierten Fertigung (S. 55)
nach Geitner: *CIM-Handbuch*
Bild 17, S. 363

Bild 11: Die logistische Kette (S. 58)
nach Koether: *Technische Logistik*
Bild 1.1, S. 2

Bild 12: Einfluß der Durchlaufzeit auf das Umlaufvermögen (S. 60)
nach Koether: *Technische Logistik*
Bild 1.2, S. 3

Bild 13: Einsatz verschiedener Fertigungsstrukturen (S. 63)
nach Koether: *Technische Logistik*
Bild 5.13, S. 116

Bild 15: Phasen und Schritte einer Simulationsstudie (S. 79)
nach *VDI-Richtlinie 3633 Blatt 1*
Bild 8, S. 9

Bild 16: Werkzeugentwicklung in der Objekt-Subjekt-Modell-Relation (S. 84)
nach *VDI-Richtlinie 3633 Blatt 1*
Bild 5, S. 5

Bild 17: Integration von Teilmodellen zum dynamischen Fabrikmodell (S. 86)
nach *VDI-Richtlinie 3633 Blatt 1*
Bild 6, S. 6

Literatur

Ahrens, K.; Fischer, J.; Witaszek, D.
Objektorientierte Prozeßsimulation in C++
Informatik Preprint 14
FB Informatik d. Humboldt-Universität, Berlin, 1992

Apple Computer (Hrsg.)
Macintosh Human Interface Guidelines
Addison-Wesley, Reading, Massachusetts; 1992

Balzert, H.
Software-Ergonomie
Bericht über die Tagung I/1983 des German Chapter of the ACM
B.G. Teubner, Stuttgart; 1983

Bernhard, J.
Entwicklung einer Geometrie- und Animationsbibliothek für die Simulation in Produktion und Logistik
Diplomarbeit, Fachgebiet Produktionssysteme, Universität Gh Kassel; Juli 1998

Bleul, A.; Loviscach, J.
Programmieren nach Plan
in: c't - Magazin für Computertechnik; Ausgabe 19/98
Verlag Heinz Heise, Hannover; 1998

Bobrow, D.; Stefik, M.
Perspectives on Artificial Intelligence Programming
in: Science, vol.231, S.951; 1986

Booch, G.
Object-Oriented Analysis and Design with Applications
Benjamin/Cummings, Redwood City, California; 2. Aufl., 1994

Booch, G.; Jacobsen, I.; Rumbaugh, J.
The Unified Modeling Language - UML Notation Guide
http://www.rational.com/uml/html/notation
Rational Software Corporation; Version 1.1, 01.09.1997

Coplien, J. O.
Advanced C++ Programming Styles and Idioms
Addison-Wesley, Reading, Massachusetts; 1992

Dahl, O.; Dijkstra, E.W.; Hoare, C. A. R.
Structured Programming
Academic Press, London; 1972

Dijkstra, E.W.
A note on two problems in connexion with graphs
in: Numerische Mathematik 1, Springer, Berlin, 269-271, 1959

DIN Deutsches Institut für Normung e.V.
DIN EN ISO 9000 Qualitätsmanagementsysteme - Grundgedanken und Begriffe
Beuth, Berlin; 1998

DIN Deutsches Institut für Normung e.V.
DIN 66001 Informationsverarbeitung;
Sinnbilder für Datenfluß- und Programmablaufpläne
Beuth, Berlin; 1977

DIN Deutsches Institut für Normung e.V.
DIN 66261 Informationsverarbeitung;
Sinnbilder für Struktogramme nach Nassi-Shneiderman
Beuth, Berlin; 1985

Ellis, M. A.; Stroustrup, B.
The Annotated C++ Reference Manual
Addison-Wesley, Reading, Massachusetts; 1990

Eschenbacher, P.
Entwurf und Implementierung einer formalen Sprache zur Beschreibung dynamischer Modelle
Technische Fakultät der Universität Erlangen-Nürnberg, Erlangen; 1989

Eversheim, W.
Organisation in der Produktionstechnik; Band 1 Grundlagen
VDI, Düsseldorf; 2. Aufl., 1990

Eversheim, W.
Organisation in der Produktionstechnik; Band 4 Fertigung und Montage
VDI, Düsseldorf; 2. Aufl., 1989

Ford Motor Company (Hrsg.)
Ford Around The World
http://www.ford.com/archive/fordhistory.html
ohne Ort und Datum (aufgerufen am 28.01.1998)

Geitner, U. W. (Hrsg.)
CIM Handbuch
Vieweg, Braunschweig Wiesbaden; 2. Aufl., 1991

Gordon, G.
The Application of GPSS V to Discrete Systems Simulation
Prentice-Hall, Englewood Cliffs, New Jersey; 1975

Großeschallau, W.
Materialflußrechnung
Springer, Berlin Heidelberg; 1984

Harder, K.
Simulationsgestützer Leitstand
in: *Fortschritte in der Simulationstechnik*, Band 11; ASIM-Tagungsbericht
Vieweg, Braunschweig Wiesbaden; 1997

IBM (Hrsg.)
Systems Application Architecture, Common User Access
- Guide to User Interface Design -
IBM; 1991

Jackson, M.
Principles of Program Design
Academic Press, Orlando, Florida; 1975

Jackson, M.
System Development
Prentice-Hall, Englewood Cliffs, New Jersey; 1983

Kernighan, B. W.; Ritchie, D. M.
Programmieren in C
Carl Hanser, München Wien / Prentice Hall International, London;
2. Ausg., 1990

Kiviat, P. J.; Markowitz, H.; Villanueva, R.
The Simscript II Programming Language
Prentice-Hall, Englewood Cliffs, New Jersey; 1969

Knuth, D. E.
The Art of Computer Programming; Vol. 2: Seminumerical Algorithms
Addison-Wesley, Reading, Massachusetts; 1981

Koether, R.
Technische Logistik
Carl Hanser, München Wien; 1993

Krauth, J.
Comparison 2: Flexible Assembly System
in: EUROSIM - Simulation News Europe; Nr. 2
ARGESIM, Wien; 1991

Kuhn, A.; Reinhardt, A,; Wiendahl H.-P. (Hrsg.)
Handbuch Simulationsanwendungen in Produktion und Logistik
erschienen als Band 7 der Reihe *Fortschritte in der Simulationstechnik*
Vieweg, Braunschweig Wiesbaden; 1993

Ladkin, P.
Fallstricke auf dem Weg ins All
in: c't - Magazin für Computertechnik; Ausgabe 19/98
Verlag Heinz Heise, Hannover; 1998

Langdon, G.
Computer Design
Computeach Press, San José, California; 1982

Loviscach, J.
Absturzgefahr - Die Bug-Story -
in: c't - Magazin für Computertechnik; Ausgabe 19/98
Verlag Heinz Heise, Hannover; 1998

Meyer, B.
Object-Oriented Software Construction
Prentice-Hall, New York, New York; 1988

Miller, G.
The Magical Number Seven, Plus or Minus Two:
Some Limits on Our Capacity for Processing Information
in: The Psychological Review, vol. 63 (2)

Microsoft (Hrsg.)
The Windows Interface, An Application Design Guide
Microsoft Press; 1992

Moore, E.F.
The Shortest Path Through a Maze
Proceedings of the International Symposium on the Theory of Switching,
The Annals of the Computing Laboratory of Harvard University 30, Part II,
Harvard University Press, 1959

Noche, B.; Bernhard, W.; Krauth, J.; Meyer, R.; Wenzel, S.
Simulationsinstrumente im Überblick
in: *Handbuch Simulationsanwendungen in Produktion und Logistik*
Vieweg, Braunschweig Wiesbaden; 1993

Noche, B.
Kopplung von Simulationsmodellen mit Leitrechnern
in: *Fortschritte in der Simulationstechnik*, Band 11; ASIM-Tagungsbericht
Vieweg, Braunschweig Wiesbaden; 1997

Nygaard, K.; Dahl, O-J.
The Development of the Simula Languages
in: History of Programming Languages;
Academic Press, New York, New York; 1981

OpenGL Architecture Review Board (Hrsg.)
OpenGL Specification & Manual Pages
http://www.opengl.org/Documentation/Specs.html
ohne Ort und Datum (aufgerufen am 14.10.1998)

Ortmann, L.
Zeitdynamische Simulation - neue Qualität für PPS-Systeme und Leitstände
in: *Fortschritte in der Simulationstechnik*, Band 11; ASIM-Tagungsbericht
Vieweg, Braunschweig Wiesbaden; 1997

Osterhout, J. K.
Tcl und Tk:
Entwicklung grafischer Benutzerschnittstellen für das X Window System
Addison-Wesley Deutschland, Bonn; 1995

Perl, J.
Graphentheorie: Grundlagen und Anwendungen
Akademische Verlasgesellschaft, Wiesbaden; 1981

REFA (Hrsg.)
Methodenlehre des Arbeitsstudiums (MLA); Teile 1 bis 3
Hanser, München; 2. Aufl., 1987

Reinhardt, A.
SIMFLEX - Ein Softwaresystem zur interaktiven graphischen Erstellung und Steuerung von Modellen flexibler Fertigungssysteme
in: *Informatik-Fachberichte, Band 11*
Springer, Berlin; 1977

Reinhardt, A.
Entwurf von Materialflußsystemen und Experimentsteuerung mittels graphisch-interaktiver Simulation
in: *Informatik-Fachberichte Band 85: Simulationstechnik*
Springer, Wien; 1984

Reinhardt, A.
Die Kluft zwischen Simulation und Steuerungssoftware
in: Simulationstechnik und Logistik; ASIM-Tagungsbericht
gfmt-Verlag, München; 1988

Reinhardt, A.; Koop, D.; Hanisch, S.; Rabe, M.; Rottbeck, B.
Simulationsinstrumente - Modellierung und Implementierung
in: *Handbuch Simulationsanwendungen in Produktion und Logistik*
Vieweg, Braunschweig Wiesbaden; 1993

Reinhardt, A.; Rudnig, M.; Wiegand, M.
Simulation und Realzeitsteuerung
in: Simulation und Fabrikbetrieb; ASIM-Tagungsbericht
gfmt-Verlag, München; 1993

Rose, M.
Kritische Analyse von Routensuchverfahren
http://www.bauinf.uni-hannover.de/Mitarbeiter/ROSE/
Routensuche/html/Routensuche.html
Studienarbeit, Institut für Verkehrswirtschaft, Strassenwesen und Städtebau, Universität Hannover; 1996

Rudnig, M.
Grundlagen der Fabrikplanung
Skriptum, Universität Gh Kassel; ohne Datum

Rudnig, M.
Ausgewählte Kapitel der Fabrikplanung
Skriptum, Universität Gh Kassel; ohne Datum

Schildt, H.
The Annotated ANSI C Standard
American National Standards for Programming Languages - C
ANSI / ISO 9899 - 1990
McGraw-Hill, Berkeley, California; 1990

Schulz, H.
Warum sind erfolgreiche Unternehmen erfolgreich ?
in: Werkstatt und Betrieb 124 (1991) 11

Simon, R.
Windows 95 WIN32 programming API Bible
Waite Group Press, Corte Madera, California; 1996

Sommerville, I.
Software Engineering
Addison-Wesley, Workingham, England; 2. Aufl., 1985

Sperlich, T.; Bauer, Ch.
Künstlich Welten - Virtual Reality: schneller und echter -
in: c't - Magazin für Computertechnik; Ausgabe 4/96
Verlag Heinz Heise, Hannover; 1996

Stroustrup, B.
Die C++ Programmiersprache
Addison-Wesley (Deutschland), Bonn; 2. Aufl., 1992

Sun Microsystems (Hrsg.)
AT&T Lang. System Library Manual (Task Library)
Sun Microsystems, Part-No. 800-5148-10 Rev. A; 1991

Sun Microsystems (Hrsg.)
Java Platform Documentation
http://java.sun.com/docs/index.html
Stand: 02. Okt. 1998

Tikal, F.
Produktionstechnik 2
Skriptum, Institut für Produktionstechnik und Logistik - Universität Gh Kassel; ohne Datum

Tikal, F.; Vollmer, Ch.
Alternative Montagekonzepte am Beispiel einer PKW-Tür
in: *Einflußfaktoren auf die Montageorganisation der europäischen Automobilindustrie*, Seminar des Instituts für Produktionstechnik und Logistik, Universität Gh Kassel Kassel; 1996

VDI - Verein Deutscher Ingenieure
VDI 3300 Materialfluß-Untersuchungen
Beuth, Berlin, 1973

VDI - Verein Deutscher Ingenieure
VDI 3633 Simulation von Logistik-, Materialfluß- und Produktionssystemen Blatt 1 Grundlagen
Beuth, Berlin, 1993

VRML Consortium (Hrsg.)
The Virtual Reality Modeling Language
International Standard ISO/IEC 14772-1:1997
http://www.vrml.org/Specifications/VRML97
ohne Ort und Datum (aufgerufen am 14.10.1998)

Warnecke, H.-J.
Die Fraktale Fabrik - Revolution der Unternehmenskultur
Rowohlt Taschenbuch, Reinbek, 1996

Watt, A.
3D Computer Graphics
Addison-Wesley, Workingham, England; 2. Aufl., 1993

Wiegand, M.
Ein Simulator als Leitstand - Erfahrungen aus einem Industrieprojekt -
in: *Fortschritte in der Simulationstechnik*, Band 9; ASIM-Tagungsbericht
Vieweg, Braunschweig Wiesbaden; 1994

Wiendahl, H.-P.
Betriebsorganisation für Ingenieure
Carl Hanser, München Wien; 2. Aufl., 1986

Wiendahl, H.-P.
Belastungsorientierte Fertigungssteuerung
Carl Hanser, München Wien; 1987

Wirth, N.
Program Development by Stepwise Refinement
in: Communications of the ACM, vol. 26 (1)

Wirth, N.
Algorithmen und Datenstrukturen
B. G. Teubner, Stuttgart; 3. Aufl., 1983

Der Simulator SIMFLEX/2

SIMFLEX/2 ist ein graphisch-interaktiver Simulator für die Modellierung und Simulation von Materialflußsystemen. SIMFLEX/2 wurde von Univ.-Prof. Dipl.-Ing. A. Reinhardt an der TU Berlin entwickelt [1] und wird heute am Fachgebiet Produktionssysteme des Instituts für Produktionstechnik und Logistik der Universität Gh Kassel weiterentwickelt und gepflegt. Durch meine Einbindung in diese Arbeiten gelangte ich zu Kenntnissen und Erfahrungen, von denen viele in diese Arbeit eingeflossen sind.

Modellstruktur

SIMFLEX/2 ist ein bausteinbasierter Simulator [2], so daß Modelle auf der untersten Ebene aus Bausteinen bestehen. Diese sind als Prototypen in modifizierbaren Bausteinbibliotheken abgelegt. Jeder Prototyp spezifiziert dabei eine Variante eines Basistyps. Die verfügbaren Basistypen können vom Anwender nicht ergänzt werden. Sie repräsentieren materialflußtechnische Komponenten wie Stau- und Kettenförderer, Fahrzeuge, Arbeitsplätze (Maschinen), etc. Der Basistyp spezifiziert das dynamische Verhalten eines Bausteins, das im Simulatorkern fest kodiert ist. Die in einer Bausteinbibliothek enthaltenen Varianten eines Basistyps können sich in den (voreingestellten) technischen und steuertechnischen Parametern und in der grafischen Kontur unterscheiden.

Die Bausteine eines Modells sind Instanzen der in der verwendeten Bausteinbibliothek enthaltenen Varianten. Sie können einzeln parametriert, in ein Modell eingefügt und darin beliebig angeordnet werden. SIMFLEX/2-Modelle sind maßstäbliche Anlagenmodelle, beispielsweise werden die Fahrwege und -zeiten von Fahrzeugen in SIMFLEX/2 aus der Position der zu bedienenden Übergabepunkte im Modell bzw. Layout und der Geschwindigkeit des Fahrzeugs errechnet. In konsequenter Fortführung dieses Prinzips können die (graphische Gestalt der) Bausteine und ihre Positionen im Layout aus CAD-Zeichnungen direkt in SIMFLEX/2-Modelle überführt werden (s.u.).

---

[1] Reinhardt, A.: *Entwurf von Materialflußsystemen und Experimentsteuerung mittels graphisch-interaktiver Simulation*

[2] vgl. Kapitel 7.5.2

Die Bausteine eines Modells werden dann miteinander verkettet, d.h. durch Ein- und Ausgangsbeziehungen miteinander verknüpft, um den Materialfluß in der Anlage zu spezifizieren. In weiteren Schritten wird das Modell nach Bedarf um Arbeitspläne, Aufträge (Mengen und Termine), Störprofile und Meß- und Protokolleinrichtungen erweitert.

Für komplexe Modelle erlaubt die Programmierschnittstelle von SIMFLEX/2, das Verhalten der Arbeitsplätze (Maschinen) aufgabenbezogen zu programmieren. Der Standardablauf Teil aufnehmen, bearbeiten, abgeben kann so beliebig verfeinert werden. Dies ermöglicht u.a. einen zustandsabhängigen Programmablauf, das Synchronisieren von Abläufen, das Bestellen von Teilen aus Lagern und Quellen, das Verschicken von Signalen an andere Bausteine und das Bilden, Nutzen und Protokollieren aufgabenspezifischer Informationen.

Softwarewerkzeuge

Für den Entwurf von Simulationsmodellen und die Durchführung von Experimenten bietet SIMFLEX/2 eine Reihe abgestimmter Softwarewerkzeuge ("Tools") für verschiedene Aufgaben, die dem Anwender die Arbeit im Dialog am Rechner ermöglichen.

Das Layout-Erstellungs-Tool (LET) ist ein Anwendungsprogramm für das CAD-System AutoCAD. Mit ihm können AutoCAD-Zeichnungselemente (Blöcke) in SIMFLEX/2-Bausteinprototypen (Varianten von Basistypen) überführt und um technische und steuertechnische Parameter ergänzt werden. Die Vorkommen der Blöcke in der Ausgangszeichnung (Inserts) können in Bausteine überführt werden. Aus den Prototypen kann eine Bausteinbibliothek erzeugt werden, und die Bausteine (Typ- und Positionsinformation) können in eine SIMFLEX/2-Layout-Datei geschrieben werden, die anschließend mit weiteren Tools zu einem vollständigen Simulationsmodell erweitert werden kann.

Das Graphische Topologieentwurfssystem (GTS) dient der Erstellung von Simulationsmodellen. Wie in einem CAD-System, in dem aus vorhandenen Makros ein Zeichnung erstellt wird, können mit dem GTS Bausteine, die als Prototypen in der verwendeten Bausteinbibliothek vorhanden sind, instanziiert, in einem Layout positioniert und miteinander verkettet werden. Ihre technischen und steuertechnischen Parameter können individuell angepaßt werden. Das so entstandene Layout kann nach Bedarf um Arbeitspläne, Aufträge, Störprofile und Meß- und Protokolliereinrichtungen ergänzt und so zu einem vollständigen SIMFLEX/2-Modell erweitert werden.

Das GSL ist das Werkzeug für die Durchführung von Experimenten. Es besteht aus den Komponenten Graphisches Darstellungssystem (GDS), Statistisches Darstellungssystem (SDS) und Logisches Modellsystem (LMS), die zu einem Programm zusammengefaßt sind. Das

LMS ist der Kern des Simulators, in dem die Bausteine in ihrer Dynamik aktiv sind. Das SDS stellt Betriebsdaten aus dem Experiment auf dem Bildschirm dar. Das GDS stellt das Layout des Modells grafisch (2D) dar und animiert es, indem es die durch die abgebildete Anlage fließenden Teile durch die Grafik bewegt. Die drei Teilsysteme sind online gekoppelt. Da das GSL interaktive Benutzereingriffe erlaubt, können beispielsweise besondere Situationen (z.B. Ausfall eines Fördermittels) gezielt hergestellt und ihre Auswirkungen unmittelbar am Bildschirm beobachtet werden. Zugunsten höherer Geschwindigkeit (z.B. für Langzeitexperimente) können die Darstellungssysteme ganz oder für im Dialog einstellbare (Modell-) Zeiträume (Zeitsprungeffekt) abgeschaltet werden, so daß nur das LMS arbeitet.

Projekte

Während meiner Tätigkeit als Wissenschaftlicher Mitarbeiter am Fachgebiet Produktionssysteme im Institut für Produktionstechnik und Logistik der Gesamthochschule Kassel habe ich an der Bearbeitung der nachfolgend aufgeführten Projekte teilgenommen. Die dabei entstandenen Erkenntnisse und die bei der Bearbeitung gemachten Erfahrungen sind in diese Arbeit eingeflossen. In allen aufgeführten Projekten wurde der Fabriksimulator SIMFLEX/2 (s.o.) eingesetzt.

Da den jeweiligen Auftraggebern Vertraulichkeit zugesichert wurde, ist es bei den meisten Projekten nicht möglich, die Auftraggeber und die exakten Arbeitstitel zu nennen, wofür ich hiermit um Verständnis bitte.

1. Grundlagen der rechnerintegrierten Fabrik (1991 - 1994)
(Auftraggeber: Volkswagen AG, Zentrales Bildungswesen / Fortbildung)

Einsatz des Simulators SIMFLEX/2 als Leitstand einer bei VW bestehenden Modellfabrik zur Mitarbeiterschulung und zur Simulation von Layoutvarianten im Rahmen von Schulungen. Simulations- und Leitstandssoftware sind vom Quellcode bis zum ausführbaren Programm (einschl. aller Laufzeitdateien) identisch. SIMFLEX/2 ist für den Leitstandseinsatz mit den auf unterlagerten Steuerungsebenen eingesetzten industrieüblichen speicherprogrammierbaren Steuerungen (SPS) über Ethernet/SINEC-AP gekoppelt.

Aus dem Projekt entstanden die Veröffentlichungen Reinhardt et al.: *Simulation und Realzeitsteuerung* und Wiegand: *Ein Simulator als Leitstand - Erfahrungen aus einem Industrieprojekt -*.

2. Triebwerkaufrüstung inclusive Hochzeitsbereich (1995)

Simulationsstudie zur Untersuchung einer Produktionsanlage zur Endmontage von PKW-Motoren einschließlich der Zulieferung zum Einbau (Hochzeit). Endmontage und Einbau sind in einem Just-in-time-Konzept integriert. Der Einbau erfolgt im Minutentakt (450 Fzg. / Schicht). Die Motoren sind auf Subpaletten aufgebaut, die in der Montage von Elektro-Hängebahnen (EHB) und beim Einbau auf Montagewagen gefördert werden.

Untersuchungsziele waren der Nachweis der Funktion und der Verfügbarkeit unter Berücksichtigung von Störungen, die Ermittlung der erforderlichen Anzahl von EHB-Fahrzeugen, die Plazierung und Dimensionierung erforderlicher Puffer im EHB-Umlauf und die Entwicklung von Steuerungsalgorithmen für neuralgische Punkte (EHB-Aufzüge, Übergabe der Motoren von EHB-Fahrzeugen an Montagewagen einschließlich der Rücknahme leerer Subpaletten von den Montagewagen (verfügbarer Platz aus baulichen Gründen nur 2 Montagewagen- bzw. 4 EHB-Längen)) und zur Sammlung freier EHBs in einem speziellen Leerpuffer.

3. Motorenendmontage ... (1995 - 1996)

Simulationsstudie zur Untersuchung eines Motorenendmontagewerks einschließlich der LKW-Zulieferung zum Einbau (4 Linien). Motorenendmontage und Einbau sind in einem Just-in-time-Konzept integriert. Die Produktion beträgt ca. 900 Motoren je Schicht (440 min.), wobei 8 Varianten unterschieden werden. Die Motoren werden in der Montage von Elektro-Hängebahnen (EHB) und in den Lager- und Verladebereichen mit konventioneller Fördertechnik (Stauförderer, RFZ, Aufzüge) transportiert. Die Verladung erfolgt auf spezielle Transportdecks zu je 32 Motoren (4 Schächte à 8 Motoren ohne wahlfreien Zugriff) in Einbausequenz. Die Montage erfolgt in 3 Linien in optimierter Reihenfolge (Linienzuordnung der Varianten, Arbeitsinhalte).

Untersuchungsziele waren der Nachweis der Funktion und der Verfügbarkeit unter Berücksichtigung von Störungen (keine Taktausfälle in den Montagelinien, möglichst verzögerungsfreie und vollständige LKW-Beladung), die Ermittlung der zu erwartenden Anzahl von “Taxi-Fahrten” zur Eillieferung verspäteter Motoren zum Einbau, die Ermittlung der erforderlichen Anzahl von EHB-Fahrzeugen, die Plazierung und Dimensionierung erforderlicher Puffer im EHB-Umlauf, die Gestaltung des Nacharbeitsbereichs für optimalen Durchsatz und die Entwicklung von Steueralgorithmen für die Taktregelung verschiedener Teilbereiche.

4. PKW-Montage ... (1996)

Simulationsstudie zur Untersuchung eines Montagewerks neuartiger Konzeption für kleine PKW. Die Montage erfolgt weitestgehend aus vormontierten Modulen (Cockpit, Heck (Motor, Getriebe, Auspuff, Hinterachse), Front (Vorderachse, Scheinwerfer, Stoßfänger)). Die Modulmontage erfolgt am gleichen Standort durch Zulieferer, die in einem Just-in-time-Konzept eingebunden sind. Die Produktion beträgt ca. 200000 PKW p.a., die Taktzeit beträgt 1,7 min., die Durchlaufzeit liegt im Regelfall deutlich unter 8 Stunden. In der Montage gibt es keine Ausschleusmöglichkeiten, sie ist in 4 Teilbereiche (Äste) gegliedert, die durch kleine Puffer (ca. 15 min. Arbeitsinhalt) voneinander entkoppelt sind. Prüfung und Finish sind in 2 weiteren Ästen angeordnet, vor denen jeweils in einen zentralen Nacharbeitsbereich ausgeschleust werden kann.

Untersuchungsziele waren der Nachweis der Funktion und der Verfügbarkeit unter Berücksichtigung von Störungen, die Aufschlüsselung der Stillstandszeiten in den Ästen nach Störung, Modulausfall, Werkereingriff, Mangel (Vorpuffer leer) und Stau, die Ermittlung der Bestände in den Zwischenpuffern und vor den Nacharbeitsplätzen, die Ermittlung der Durchlaufzeiten und die Ermittlung der erforderlichen Anzahl Werker für den Nacharbeitsbereich.

Sonstige Quellen

Wichtige Einflüsse und Anstöße für diese Arbeit entstammen auch einer Vielzahl von Gesprächen, die ich zum einen mit Mitarbeitern der Auftraggebern der o.a. Projekte und zum anderen mit vielen anderen auf dem Gebiet der Simulation tätigen Menschen geführt habe. In besonderer Weise ist hier Herr Dipl.-Ing U. Hartke zu nennen.

Schließlich hatten auch noch einige Fachtagungen und mit diesen jeweils im Zusammenhang durchgeführten Ausstellungen von Simulationswerkzeugen Einfluß auf die vorliegende Arbeit, da sich mir in diesem Rahmen die Möglichkeit zum (allerdings sicher nicht sehr ausführlichen) Kennenlernen einer ganzen Reihe von Simulatoren für Materialflußsysteme bot. Zu nennen sind hier:

Fachtagung “Simulation und Fabrikbetrieb” der
Arbeitsgemeinschaft Simulation (ASIM) in der Gesellschaft für Informatik
Februar 1993, Aachen

9. Symposium Simulationstechnik der
Arbeitsgemeinschaft Simulation (ASIM) in der Gesellschaft für Informatik
Oktober 1994, Stuttgart

Fachtagung “Simulation und Animation für Planung, Bildung und Präsentation” der
Otto-von-Guericke-Universität Magdeburg und der ASIM (u.a.)
29.02. - 01.03.1996, Magdeburg

10. Symposium Simulationstechnik der
Arbeitsgemeinschaft Simulation (ASIM) in der Gesellschaft für Informatik
September 1996, Dresden