Eine Programmiertechnik für Verteilte Systeme von Mikroprozessoren

Abschlußbericht zum DFG-Vorhaben

Ho 824/2

Dr. P. Holleczek

Regionales Rechenzentrum der
Universität Erlangen-Nürnberg

15. August 1986

I N H A L T S V E R Z E I C H N I S

Vorwort

Ein großer Teil der hier vorgestellten Entwicklungen wurde im DFG-Schwerpunktprogramm "Mikroprozessoren" im Vorhaben Ho 824/2 mit dem Thema "Eine Programmiertechnik für Verteilte Systeme von Mikroprozessoren" am Regionalen Rechenzentrum der Universität Erlangen-Nürnberg (RRZE) - Partner in diesem Projekt war das Physikalische Institut, das die praktischen Aspekte abdeckte - erarbeitet. Während das Projekt anfangs eigentlich sein Anwendungsfeld hauptsächlich in der Steuerung und Überwachung technischer Prozesse sah, zeigte sich im Laufe der Arbeit, daß sich die entwickelten Werkzeuge ohne Abstriche auch in anderen Realzeitanwendungen, wie der Kommunikationstechnik, bewährten. Deshalb weitete sich das Thema allmählich zu der Entwicklung einer "Programmiertechnik für Verteilte Systeme von Mikrorechnern in Realzeitanwendungen" aus.

Das Thema des Projekts verlangt im wesentlichen die Entwicklung geeigneter programmiersprachlicher Konstrukte, von zugehöriger Systemsoftware und von sprachorientierten Testverfahren. (Die Ausarbeitungen zu diesen Teilkomplexen finden sich in den Kapiteln 2.2., 3.3.2.) Sie können allerdings nicht isoliert dargestellt, sondern müssen im Zusammenhang mit dem ganzen Software Life Cycle gesehen werden.

Aus diesem Grund wird hier auch über alle begleitenden Untersuchungen berichtet, die sich mit der Abrundung des Forschungsprojekts befassen. Sie wurden hauptsächlich im Rahmen von zahlreichen vom RRZE betreuten Studien- und Diplomarbeiten vorgenommen sowie in Dissertationen, die in Zusammenarbeit mit den Lehrstühlen für Informatik II (Programmiersprachen) und Informatik IV (Betriebssysteme) angefertigt wurden. Desgleichen wird auch über Erfahrungen berichtet, die im Bereich der Experimentalphysik und am RRZE gewonnen wurden.

1. Einleitung

Bei der Lösung von Realzeitaufgaben stellen sich neben der zeitkritischen Verarbeitung von Signalen und der Weitergabe von Ergebnissen oft die Randbedingungen,

- die Lösung kostengünstig vorzunehmen,
- die Lösung bei wachsenden Anforderungen modular zu erweitern,
- mit räumlich verteilten Problemen (z.B. in der Prozeßdatenverarbeitung) zurechtzukommen,
- Vorkehrungen zur Steigerung der Zuverlässigkeit gefährdeter Komponenten zu treffen.

Bedingung ist, daß sich die Lösung so geeignet in Teilkomponenten strukturieren läßt, daß sich der Datenaustausch zwischen den Komponenten in sinnvollen Grenzen hält. Verteilte, lose gekoppelte Systeme von Mikrorechnern stellen eine Lösung dieser Aufgabenklasse dar. Für solche Konfigurationen braucht man eine geeignete Programmiertechnik. Für marktgängige Mikrorechner gibt es zwar sequentielle Hochsprachen, nötig aber ist vielmehr eine Programmiertechnik, die der Problematik der Verteilten Systeme als auch dem Echtzeitcharakter der Aufgabenstellung gerecht wird. Die Entwicklung einer solchen Technik ist Gegenstand dieses Berichtes. Als modellhaftes Erprobungsgebiet der entwickelten Programmiertechnik dient die Experimentiertechnik der Kern- und Teilchenphysik, auf die die genannten Anforderungen zutreffen.

1.1. Aufgabenstellung

Aus dem Wortlaut des Themas "Eine Programmiertechnik für Verteilte Systeme von Mikroprozessoren (in Realzeitanwendungen)" lassen sich wesentliche Merkmale für die Aufgabenstellung ableiten.

Die Entwicklung einer Programmiertechnik legt nahe, daß ein Werkzeug bereitgestellt werden soll, das an möglichst vielen Stellen des Software Life Cycle /KKST79/ angreift, um dem Anwender breite Hilfestellung zu bieten. Eine zentrale Rolle spielen dabei Spezifikationstechnik und Programmiersprache (mit Übersetzer und Betriebssystem). Darüber hinaus sind von Interesse: Entwurfshilfsmittel, Testwerkzeuge und Implementationshilfen.

Die Anwendung der Programmiertechnik für Verteilte Systeme bedeutet, daß der Entwicklungsschwerpunkt auf Rechnerkonfigurationen mit loser Kopplung, weniger auf Multiprozessoren mit gemeinsamem Speicher, gelegt wird. Das soll aber eine Anwendbarkeit des letztlich entstehenden Werkzeugs für diese Konfiguration nicht ausschließen.

Mit Hilfe der Programmiertechnik sollen Programme für Mikrorechner erzeugt werden. Die für unsere Belange wesentlichen Eigenschaften von Mikrorechnern sind, daß sie in der Regel ohne Bedienterminal bzw. ohne Hintergrundspeicher eingesetzt werden. Das hat Einfluß auf das Testsystem und das Übersetzungs- bzw. Betriebssystem.

Als Anwendungsgebiet sind Realzeitaufgaben gedacht, bei denen die physikalische Experimentiertechnik Maßstäbe setzt, was Vielseitigkeit und Flexibilität angeht. Trotzdem geht man auch hier von gewissen Voraussetzungen aus:

- Die Konfiguration ist statisch:
  Der technische Prozeß und die Prozeßperipherie sind stets am gleichen Rechner angeschlossen. Die zugehörigen Programme sind also an einen bestimmten Rechner gebunden.

- Die Kommunikation zwischen den Rechenprozessen orientiert sich am Datenfluß zwischen den Komponenten des technischen Prozesses.

- Die Steuerung technischer Prozesse stellt gewisse Sicherheitsforderungen. So müssen Vorkehrungen getroffen werden können, um den Ausfall von Rechnern, Rechenprozessen oder der Kommunikationseinrichtung zwischen Rechnern feststellen und darauf reagieren zu können.

Diese Forderungen sind nicht unbillig. Sie decken über die genannte Testanwendung hinaus einen großen Bereich von Anwendungen ab. Andererseits sind sie, insbesondere wegen des Verzichts auf dynamische Rekonfiguration, so gewählt, daß eine Implementierung mit begrenztem Aufwand möglich bleibt.

1.2. Die Programmiertechnik im Software Life Cycle

Eine Programmiertechnik kann nur innerhalb des Modells des Software Life Cycle entwickelt werden. Sie stellt, je nach Sichtweise, einen oder mehrere Stufen in diesem Modell dar.

Im weiteren wird vom Software Life Cycle-Modell nach /KKST79/ ausgegangen. Es unterscheidet verschiedene Phasen und zugehörige Abschlußpunkte eines Software-Entwicklungsprojekts (Bild 1).

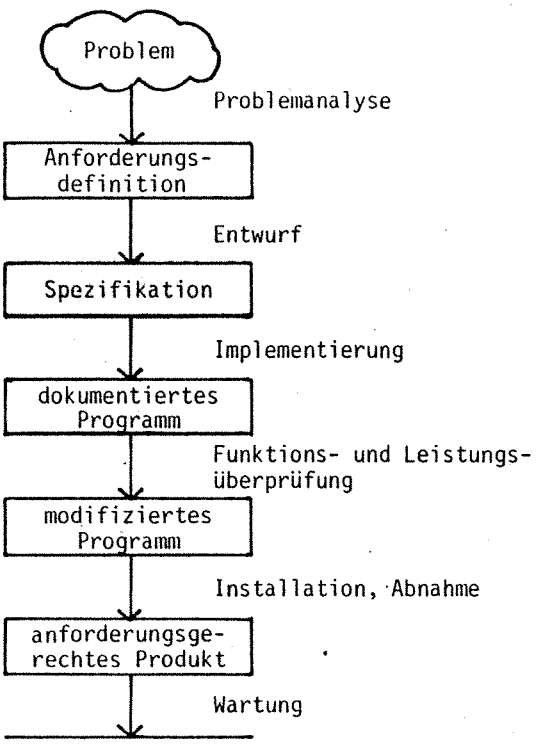

Bild 1: Software Life Cycle nach /KKST79/

Die hier entwickelte Programmiertechnik liefert Hilfsmittel für die Phasen Entwurf, Implementation und Funktions-/Leistungsüberprüfung.

1.3. Zur Gliederung

Die Beschreibung der entwickelten Programmiertechnik folgt den Projekt-Phasen des Software Life Cycle. Die folgende Tabelle gibt einen Überblick über die für die einzelnen Phasen entwickelten Werkzeuge:

Phase	Hilfsmittel
Entwurf	Spezifikationstechnik PASS
Implementierung	Programmiersprache Verteiltes PEARL
Funktions-/ Leistungsüberprüfung	Testverfahren auf Spezifikations-/ Sprachebene

Spezifikationstechnik, Programmiersprache und Testumgebung werden in den Kapiteln 2.1., 2.2. und 2.3. beschrieben. Es folgen Vorschläge zur Handhabung der Werkzeuge in hier betrachteten Phasen (Kapitel 3.).

Der Bericht schließt mit einer Beschreibung von Anwendungen (Kapitel 4.) und einer Auswertung bisheriger Erfahrungen (Kapitel 5.).

2. Entwicklungswerkzeuge für Realzeitprogramme

Die Werkzeuge zur Entwicklung von Realzeitprogrammen umfassen eine Spezifikationstechnik, Programmiersprache und Testumgebung.

Die inhaltlichen Anforderungen, die sowohl an Spezifikationstechnik als auch an Programmiersprache gestellt werden, sind die Formulierungsmöglichkeit von

- parallelen Prozessen,
- gemeinsamen Objekten verschiedener Prozesse,
- Kommunikation und Synchronisation von Prozessen mit ihrer Umwelt,
- Erkennung von Ausnahmefällen,
- Beheben von Fehlern (soweit möglich).

Die Forderungen werden von Spezifikationstechnik und Programmiersprache auf verschieden hohem Abstraktionsniveau gelöst. So ist auf Spezifikationsebene z.B. unbedeutend, ob der Kommunikationspartner eines (Rechen-) Prozesses der Benutzer oder der technische Prozeß ist, da hier der Synchronisationseffekt im Vordergrund steht. Für die Programmiersprache dagegen ist es wichtig zu wissen, ob z.B. Daten in Interndarstellung an andere Prozesse, in Bitmustern an den technischen Prozeß oder formatiert an den Benutzer weitergereicht werden.

2.1. Spezifikationsmethode

Die Spezifikation ist das abschließende Dokument der Entwurfsphase. Es enthält ein Modell der Lösung, ohne eine konkrete Implementierung vorwegzunehmen. Es dient auch dazu /KKST79/, den Nachweis der Korrektheit einer Implementierung und der Übereinstimmung mit der Anforderungsdefinition zu liefern. Ein Hilfsmittel zur Erstellung einer Spezifikation ist das zentrale Werkzeug der Entwurfsphase.

2.1.1. Anforderungen an eine Spezifikationstechnik

Die Anforderungen an Spezifikationssprachen werden aus den geforderten Eigenschaften von Spezifikationen abgeleitet.

Nach /KKST79/ sollen für eine Spezifikation folgende Qualitätsmerkmale gelten:

- Vollständigkeit und Widerspruchsfreiheit
- Minimalität
  Die Spezifikation soll alle zur Lösung des Problems notwendigen Entwurfsentscheidungen enthalten und nur diese.
- Verständlichkeit

Aus diesen Qualitätsmerkmalen werden in /KKST79/ folgende Anforderungen abgeleitet:

- Formalisierbarkeit
  Die Spezifikationssprache muß es zulassen, eine Spezifikation auf Widerspruchsfreiheit und Vollständigkeit zu prüfen. Außerdem soll es möglich sein, die Implementation gegenüber der Spezifikation zu verifizieren.

- Implementationsunabhängigkeit
  Durch die Spezifikation sollen keine speziellen Algorithmen zur Problemlösung vorgegeben werden.
- Konstruierbarkeit und Rezipierbarkeit
  Für einen Programmierer muß es mit vertretbarem Aufwand erlernbar sein, seine Entwurfsideen mit Hilfe einer Spezifikationssprache zu beschreiben.
- Änderbarkeit
  Geringe Änderungen in der Anforderungsdefinition sollten nur (entsprechend) geringe Änderungen in der (entsprechenden) Spezifikation nach sich ziehen.

Die Gewichtung dieser Anforderungen an eine Spezifikationssprache hängt stark vom Anwendungsbereich für ein Programm und von der Ausbildung der Programmentwerfer bzw. Programmierer ab. Einige dieser Anforderungen können dann sogar zueinander in Widerspruch geraten.

Daraus läßt sich die zusätzliche Anforderung ableiten, daß eine Spezifikationssprache verschiedene Grade an Formalisierbarkeit zulassen sollte, vor allem wenn die Spezifikationssprache in Bereichen eingesetzt werden soll, in denen im allgemeinen nicht von reinen Informatikern programmiert wird (kommerzielle Datenverarbeitung, Prozeßautomatisierung usw.).

In der Prozeßautomatisierung liegt die Aufgabenstellung zumeist nur in einer unvollständigen, informellen und u.U. widersprüchlichen Form vor /LUDE81/. Eine Spezifikationstechnik für Prozeßautomatisierungsprogramme muß deshalb unvollständige und zunächst auch nur informelle Spezifikationen zulassen.

2.1.2. Das Spezifikationsmodell PASS

Die hier entwickelte Technik /FL84, AFHHK85/ geht davon aus, daß sich Programme zur Steuerung von technischen Prozessen in parallele (Rechen-) Prozesse gliedern lassen. Man kann sie als Prozeßsystem bezeichnen. Der erste Teil der Spezifikation beschreibt das Zusammenwirken der Prozesse über Botschaften. Dieser Teil nennt sich "Kommunikationsstruktur".

Der zweite Teil beschreibt die Eigenschaften von Prozessen. In der sogenannten "Ablaufsteuerung" wird beschrieben, wie Rechenprozesse auf den Empfang von Botschaften reagieren oder unter welchen Umständen sie Botschaften senden. In der "Kommunikationsmaschine" wird geregelt, ob ein Prozeß Botschaften synchron oder asynchron empfängt. In der "Benutzermaschine" werden seine Operationen, Funktionen und Daten beschrieben. In einer "gemeinsamen" Benutzermaschine wird angegeben, ob und wie Prozesse über gemeinsame Objekte kommunizieren.

Kommunikationsstruktur und Ablaufsteuerung werden graphisch dargestellt. Dadurch lassen sich besonders nicht-sequentielle Vorgänge besser beschreiben.

2.1.2.1. Kommunikationsstruktur

Die Beziehungen zwischen den Prozessen werden in der "Kommunikationsstruktur" aufgezeigt, d.h. dort wird beschrieben, welche Prozesse an einem Prozeßsystem beteiligt sind und welche Nachrichten sie austauschen. Graphisch werden die beteiligten Prozesse durch Rechtecke dargestellt. Ein Pfeil, beschriftet mit dem Namen der Botschaft, zeigt vom sendenden zum empfangenden Prozeß (Bild 2).

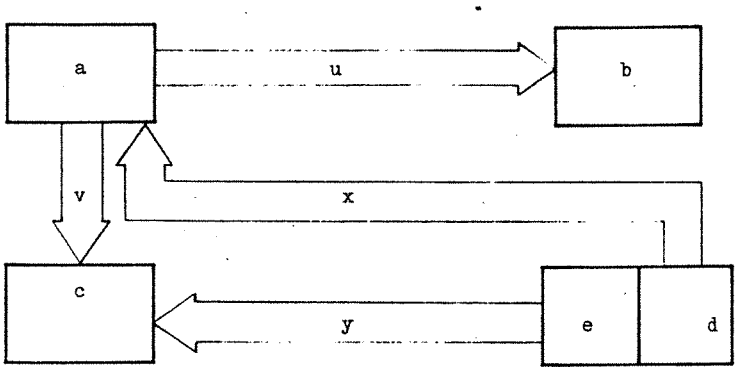

Bild 2: Beispiel einer einfachen Kommunikationsstruktur aus /FL84/

Eine Nachricht kann mit Parametern versehen sein, die Kommunikation zwischen den Prozessen erfolgt über diese Parameter.

Genauer müßte man in der Kommunikationsstruktur von Strukturierungseinheiten statt von Prozessen sprechen. Strukturierungseinheiten können einzelne Prozesse, Prozeßbündel oder Prozeßgruppen sein. Prozeßbündel bzw. Prozeßgruppen wurden eingeführt, um eine zusätzliche Kommunikation über gemeinsame Objekte zu ermöglichen.

Jede Strukturierungseinheit wird in einem zweiten Schritt genauer beschrieben.

2.1.2.2. Strukturierungseinheiten

Welche Arten von Strukturierungseinheiten (Prozessen) das Spezifikationsmodell vorsieht und welche Komponenten sie enthalten, ist in Bild 3 zusammengefaßt dargestellt.

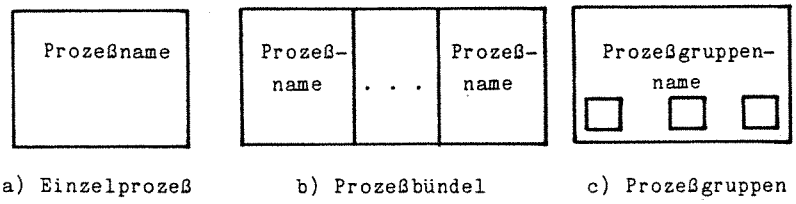

Bild 3: Symbole für die Strukturierungseinheiten aus /FL84/

Prozesse, Prozeßgruppen und Prozeßbündel werden grundsätzlich als ein Prozeßtyp definiert. Der Grund liegt darin, daß es in einem System kooperierender Prozesse oft Prozesse mit gleicher Ablaufsteuerung und gleicher Benutzermaschine, aber verschiedenen Partnern gibt. Objekte, deren Typ auf diese Art eingeführt wurde, müssen im Anschluß an die Definition deklariert werden, wobei dann die konkreten Kommunikationspartner festgelegt werden.

2.1.2.2.1. Einzelprozesse

Die Struktur eines Einzelprozesses umfaßt die Kommunikationsmaschine, die Ablaufsteuerung und die private Benutzermaschine (Bild 4).

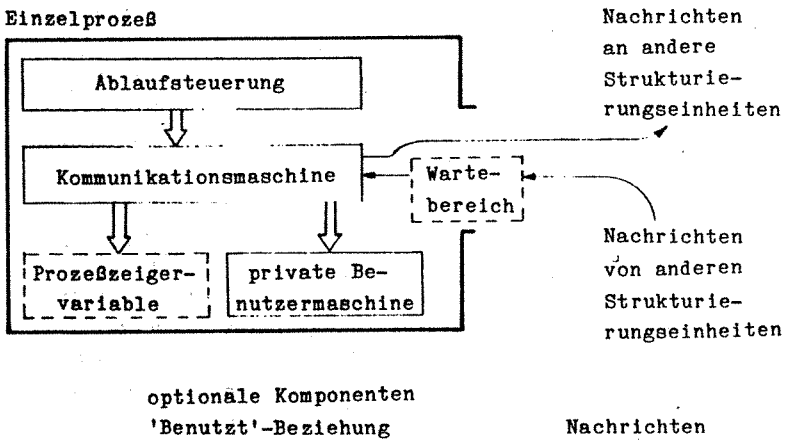

Bild 4 : Struktur von Einzelprozessen

Die "Kommunikationsmaschine" regelt das Senden umd Empfangen von Nachrichten und veranlaßt die Ausführung der entsprechenden Ausgabefunktion bzw. Eingabeoperation (näheres dazu folgt bei der Beschreibung der Benutzermaschine). Die Eigenschaften der Kommunikationsmaschine sind fest vorgegeben. Der Entwerfer des Systems kann lediglich angeben, ob die Botschaften synchron oder asynchron ausgetauscht werden sollen; d.h. es kann spezifiziert werden, ob Nachrichten über einen sogenannten Wartebereich empfangen werden sollen und wie groß dieser sein soll (wieviele Nachrichten darin Platz haben, unabhängig von der Art und Anzahl ihrer Parameter). Ist einem Prozeß ein Wartebereich vorgelagert, so können Nachrichten, die an diesen Prozeß gesendet werden, darin abgelegt werden. Der sendende Prozeß wird nicht blockiert, wenn im Wartebereich zum Ablegen der Nachricht noch ein Platz frei ist (asynchroner Nachrichtenaustausch).

Ist dem empfangenden Prozeß kein Wartebereich vorgelagert, wird der Sender solange blockiert, bis ihm vom Empfänger die Nachricht abgenommen wird (synchroner Nachrichtenaustausch).

In der "Ablaufsteuerung" wird festgelegt, in welcher Reihenfolge ein Prozeß Nachrichten sendet bzw. empfängt. Ein Prozeß tauscht nicht nur Nachrichten mit anderen Prozessen aus, sondern führt auch interne Berechnungen durch. Aufgrund der Ergebnisse solcher Berechnungen kann in verschiedene Teile der Ablaufsteuerung verzweigt werden.

In der Ablaufsteuerung gibt der Programmentwerfer somit an

- wann welche Nachrichten an wen gesendet werden,
- wann welche Nachrichten von wem empfangen werden,
- wann interne Berechnungen durchgeführt werden.

Die Ablaufsteuerung wird graphisch mit den Elementen Knoten und Kanten dargestellt.

Erreicht ein Prozeß in der Ablaufsteuerung einen "Kommunikationsknoten" (Rechtecksymbol), möchte er Nachrichten senden oder empfangen. Sollen Nachrichten gesendet werden, führen von einem Kommunikationsknoten Sendekanten (Doppelpfeile/gestrichelte Pfeile) weg. Die Sendekanten sind jeweils mit dem Adressaten und dem Namen der zu sendenden Nachricht beschriftet. Kann die Nachricht gesendet werden, geht der Prozeß in den Zustand über, zu dem die Sendekante führt.

Ein Prozeß erwartet Nachrichten, wenn er sich in einem Kommunikationsknoten befindet, von dem Empfangskanten (einfache Pfeile) wegführen. Die Empfangskanten sind jeweils mit dem Absender und dem Namen der erwarteten Nachricht beschriftet. Trifft die Nachricht vom angegebenen Quellprozeß (Absender) ein, wird in den Folgezustand übergegangen.

Führen von einem Kommunikationsknoten mehrere Sende- bzw. Empfangskanten weg, so wird alternativ auf Nachrichten gewartet bzw. es werden alternativ Nachrichten gesendet.

In der graphischen Darstellung der Ablaufsteuerung werden interne Berechnungen durch "Internknoten" (Ovale) gekennzeichnet. In diesen Ovalen wird der Name der auszuführenden internen Berechnung eingetragen. Bei internen Berechnungen wird zwischen internen Operationen und internen Funktionen unterschieden. Bei der Ausführung einer internen Operation werden die lokalen Daten eines Prozesses verändert, die internen Funktionen dagegen fragen den Zustand der lokalen Daten ab.

Führen von einem Internknoten sogenannte Operationskanten (Doppelpfeile/gestrichelte Pfeile) weg, wird eine interne Operation ausgeführt. Die Operationskanten sind mit den möglichen Resultaten der Operationsausführung beschriftet. Abhängig von den Resultaten wird in einen Folgezustand übergegangen.

Wird in einem Internknoten eine interne Funktion ausgeführt, so verlassen diesen Zustand sogenannte Funktionskanten (einfache Pfeile). Die Funktionskanten sind mit den möglichen Ergebnissen der internen Funktion beschriftet.

Der Anfangszustand wird in der Ablaufsteuerung durch einen Pfeil gekennzeichnet, dessen Anfangspunkt an keinem Knoten liegt.

Mit der Ablaufsteuerung wird nur festgelegt, wann welche Aktionen ausgeführt werden, aber nicht, wie die einzelnen Aktionen definiert sind. Es gilt noch zu beschreiben, welche Parameter die einzelnen Nachrichten (Anzahl, Typ) haben, wie sich das Empfangen einer Nachricht auf den entsprechenden Prozeß auswirkt (außer, daß in der Ablaufsteuerung in einen anderen Zustand übergegangen wird) bzw. wie beim Senden von Nachrichten der Wert der entsprechenden Parameter bestimmt wird.

Jeder Nachricht, die gesendet wird, wird eine Ausgabefunktion zugeordnet. Diese Funktion ermittelt aus dem internen Zustand eines Prozesses (Belegung der lokalen Variablen eines Prozesses) die Nachrichtenparameterwerte, ohne daß der interne Zustand verändert wird. Jeder Nachricht, die empfangen wird, ist eine Eingabeoperation zugeordnet. Sie beschreibt, wie der Empfang einer Nachricht sich auf den internen Zustand eines Prozesses auswirkt.

Bild 5 zeigt ein Beispiel für eine Ablaufsteuerung.

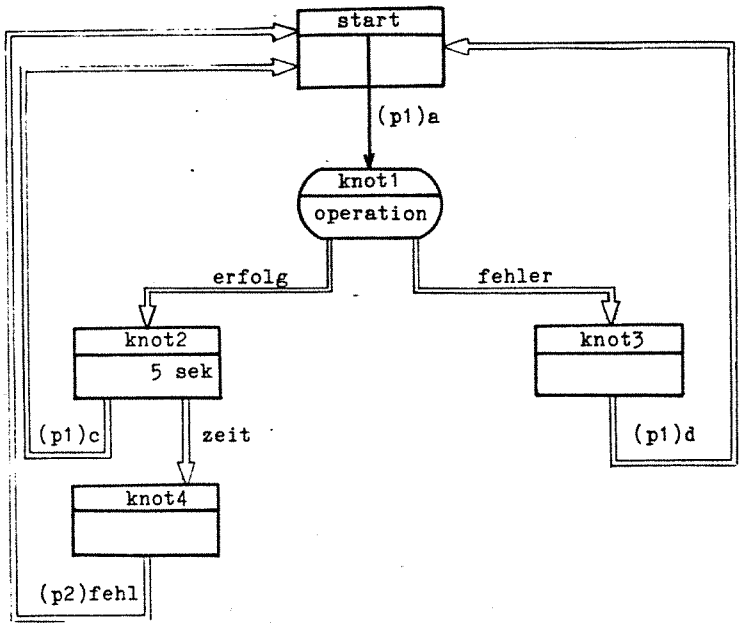

Bild 5: Beispiel für eine Ablaufsteuerung

Insgesamt müssen für eine vollständige Spezifikation eines Prozesses neben der Ablaufsteuerung folgende Funktionen und Operationen definiert werden:

- interne Funktionen,
- interne Operationen,
- Ausgabefunktionen,
- Eingabeoperationen.

Diese Funktionen und Operationen werden in der sogenannten "privaten Benutzermaschine" beschrieben. Die Benutzermaschine enthält außerdem die Daten eines Prozesses.

Die Benutzermaschine kann mit Techniken spezifiziert werden, wie sie in der sequentiellen Programmierung üblich sind (umgangssprachliche Beschreibung, pseudo-programmiersprachliche oder algebraische Spezifikation, Prädikatentransformation etc).

2.1.2.2.2. Prozeßbündel und Prozeßgruppen

Prozeßbündel und Prozeßgruppen bestehen aus mehreren Prozessen, die über gemeinsame Objekte verfügen. Jeder einzelne Prozeß wird mit der eben vorgestellten Methode, bestehend aus Kommunikationsmaschine, Ablaufsteuerung und privater Benutzermaschine, beschrieben. Die gemeinsamen Objekte werden zu einer gemeinsamen Benutzermaschine zusammengefaßt. Der Zugriff auf die gemeinsame Benutzermaschine kann durch ressourcenorientierte Synchronisationskonzepte geregelt werden.

Jeder einzelne Prozeß eines Prozeßbündels kann Nachrichten zu anderen Stukturierungseinheiten senden bzw. von diesen empfangen. Die einzelnen Prozesse einer Prozeßgruppe dagegen können nicht von außen identifiziert werden. Prozesse einer Prozeßgruppe können demnach weder als Absender noch als Adressat einer Nachricht auftreten. Nur der Name der gesamten Prozeßgruppe kann als Absender oder Adressat dienen.

2.2. Programmiersprache

Oft von entscheidender Bedeutung für den Erfolg einer Implementation ist die Wahl der Programmiersprache. Insbesondere beim Einsatz von Verteilten Systemen in der Prozeßautomatisierung werden hohe Anforderungen an die Programmiersprache gestellt.

2.2.1. Anforderungen

Neben den unter 2. genannten, für Spezifikationstechnik und Programmiersprache identischen inhaltlichen Forderungen, sollte durch eine Programmiersprache für Verteilte Systeme eine Asynchronitäts- und eine Prozeßabstraktion erfüllt sein /HOF83/. Unabhängig davon, sollte die Sprache im Kern möglichst weit standardisiert sein, um ihren Einsatz in technischen Prozessen, wo auch andere Standards eingehalten werden müssen, zu erleichtern. Aus dem gleichen Grund sollte sie im Kern einfach sein, um sie für Anwendungsingenieure einsetzbar zu machen.

Alle Forderungen zugleich werden von keiner derzeit bekannten Programmiersprache erfüllt. Die zentrale Eigenschaft der parallelen Prozesse (siehe Kapitel 2.) wird derzeit von einer Reihe neuer Sprachen erfüllt:

- Concurrent PASCAL /HANS75/, DP /HANS78/, ARGUS /LTS83/
- PEARL (DIN 66253)
- Ada (Registered Trademark of the US Government - AJPO)

Davon erheben nur PEARL und Ada den Anspruch, zur Steuerung technischer Prozesse geeignet zu sein. Diesen Anspruch erhebt auch

- Realtime-Fortran /IRTF80/.

Es ist vom (sequentiellen) Sprachkern her allerdings bereits über 20 Jahre alt.

Will man keine neue Programmiersprache entwickeln, liegt nahe, eine der drei letztgenannten Sprachen als Grundlage für weitere Entwicklungen zu verwenden.

Von der zugehörigen Systemsoftware wird verlangt, daß sie Übersetzten Code für herkömmliche Mikrorechner liefert, der, zusammen mit einem Betriebssystem, ablauffähig ist. Wünschenswert ist auch, daß der Compiler selbst auf Mikrorechnern des gleichen Typs zur Verfügung steht. Der Einsatz in physikalisch verteilten Anordnungen verlangt, daß der (von einem Compiler) erzeugte Code über die vorhandenen Kommunikationswege, z.B. mit Hilfe eines Laders, zu dem Zielrechner übertragen werden kann. Das Betriebssystem muß so geartet sein, daß Programme auch ohne Bedienterminal ablauffähig sind.

2.2.2. Die Sprache und ihre Eigenschaften

Von den drei genannten Basissprachen scheiden zwei wegen theoretischer und praktischer Mängel aus.

Real-Time-Fortran weist vom Sprachkern her zahllose, nicht umgehbare Schwächen auf, z.B. die fehlende Deklarationspflicht und die fehlende Blockstrukturierung. Gerade die erstgenannte Schwäche macht es anfällig für nicht entdeckbare Flüchtigkeitsfehler und für einen Einsatz in sicherheitsrelevanten technischen Prozessen ungeeignet.

Die gerade auch zur Programmierung technischer Prozesse nach modernen Gesichtspunkten entwickelte Sprache Ada hat sicher keine Mängel dieser Art. Allerdings bezweifelt man inzwischen angesichts ihrer Größe ihre Handhabbarkeit durch Anwendungsingenieure. Andererseits ist ihr Kommunikationsverfahren über Botschaftsaustausch auf das Rendezvous-Konzept beschränkt und dadurch die Parallelität zwischen Prozessen stark eingeschränkt. Außerdem erlaubt die Sprache keine (Subsets oder) Supersets, so daß sich eine Erweiterung ohnehin verbietet. Compiler für herkömmliche Mikrorechner ließen lange auf sich warten.

Als aussichtsreichster Kandidat einer Basissprache bleibt somit PEARL. PEARL wurde speziell zur Programmierung technischer Prozesse entworfen und verfügt auch über die nötigen Konstrukte für konventionelle (sequentielle) Probleme.

PEARL erlaubt die Formulierung gemeinsamer Objekte und die Zugriffs-Synchronisation über Semaphore. Ansätze für einen Nachrichtenaustausch fehlen allerdings. Ausnahmesituationen können festgestellt, Reaktionen darauf eingeplant werden. Die Handhabung dieser Fehlerbehandlung ist allerdings nicht sehr übersichtlich.

Ein Nachrichtenkonzept läßt sich auf verschiedene Weise verwirklichen, z.B. über Botschaften, Remote-Procedure-Calls und Transaktionen. Während Remote-Procedure-Calls hauptsächlich zur Lösung sequentieller Probleme in Rechnernetzen gedacht sind, stellt die Kommunikation über Botschaften mit eindeutiger Empfänger-Sender-Beziehung das am besten geeignete Mittel zum Datenaustausch bei gekoppelten Rechnern im Echtzeitbetrieb dar.

Es muß folgende Leistungen erbringen:

- Operationen zum Senden und Empfangen von Botschaften,
- Konstrukte zum Vermeiden von Reihenfolgeproblemen,
- Zeitüberwachung von Botschaftsoperationen.

Bedingung ist, die Basissprache homogen zu erweitern, d.h. vorhandene Prinzipien mitzunutzen oder neue Konstrukte deutlich von ihnen abzusetzen. Das hat Konsequenzen z.B. für die Wahl von Schlüsselwörtern, die Einführung von Datentypen usw.

Die vorgenommene Spracherweiterung ist in /FHKK83/ eingehend beschrieben. An dieser Stelle reicht es aus, auf die Prinzipien einzugehen.

2.2.2.1. Botschaftsoperationen

Die konzeptionellen Grundlagen für die Botschaftsoperationen sind identisch mit denen der Spezifikationstechnik. Es muß deswegen nur daran erinnert werden, daß zu Botschaftsoperationen folgende Angaben entnommen werden müssen:

- Blockiertbedingungen,
- Datenformate,
- Empfänger und Absender.

Dies wird im vorliegenden Konzept auf folgende Weise verwirklicht: Es wird davon ausgegangen, daß Botschaften (durch ihren Namen) unterscheidbar sind und einen bestimmten Typ haben. Ihr Typ ist durch die Datentypen der einzelnen Komponenten einer Botschaft festgelegt. Botschaftsname, Typ sowie Adressat bzw. Absender einer Botschaft sind statische Angaben im Vereinbarungsteil eines Prozesses.

Im Vereinbarungsteil wird auch die Größe des Wartebereichs angegeben, d.h. unter welchen Umständen ein Prozeß beim Senden bzw. Empfangen einer Botschaft blockiert werden soll.

Im Ablaufteil eines Prozesses erfolgt lediglich noch der Anstoß (des Sendens bzw. Empfangens) einer Botschaftsoperation.

Dieses Verfahren wurde gewählt, um einem Übersetzungs- und Bindeprogramm zu erlauben, die Übereinstimmung der von einem Prozeß gesendeten und von einem anderen Prozeß empfangenen Botschaftstypen zu überprüfen.

2.2.2.2. Nichtdeterministische Kontrolloperationen

Der Gebrauch von Botschaftsoperationen bringt zweierlei Probleme mit sich:

- Eine sequentielle Abarbeitung von Empfangsanweisungen kann zu Deadlock-Situationen führen, wenn die zugehörigen Botschaften in unerwarteter Reihenfolge eintreffen.

- Wird aufgrund von Übertragungsstörungen eine zu sendende Botschaft vom Empfänger nicht abgenommen oder trifft eine erwartete Botschaft nicht ein, so kann der sendende bzw. empfangende Prozeß auf unbestimmte Zeit blockiert werden.

Um dem ersten Problem zu begegnen, muß eine Möglichkeit geschaffen werden, alle Botschaften anzugeben, auf deren Empfang bzw. Absenden alternativ gewartet werden soll.

Um dem zweiten Problem zu begegnen, muß das Empfangen und Senden von Botschaften zeitüberwacht werden können. Läuft die Zeitüberwachung ab, muß eine Ausnahmereaktion angestoßen werden.

Im vorliegenden Fall wurde vorgezogen, die Botschaftsoperationen nicht zu überladen und alternatives Warten und Zeitüberwachung in einer getrennten Sprachkonstruktion zu verwirklichen. Hierzu bieten sich die von Dijkstra /DIJK75/ eingeführten nichtdeterministischen Kontrolloperationen an.

Dabei lassen sich vor allem folgende vier Typen unterscheiden /Hoar78, Hans78/:

- Guarded Regions,
- Guarded Commands,
- Guarded Loops,
- Guarded Cycles.

Nichtdeterministische Kontrollanweisungen bestehen aus mehreren sogenannten "Guards". Ein Guard besteht aus einer Bedingung und einer beliebigen Anweisungsfolge. Je nach Programmiersprache kann die Bedingung anders aufgebaut sein. In /Hoar78/ kann in der Bedingung das Warten auf eine Nachricht auftreten. Eine solche Nachrichtenbedingung ist "wahr", wenn die erwartete Nachricht von dem entsprechenden Prozeß auch gesendet wird.

Je nachdem, welcher Typ einer nichtdeterministischen Kontrollanweisung nun vorliegt, wird bei der Abarbeitung einer solchen Anweisung unterschiedlich verfahren.

Wie die verschiedenen Konstruktionen im einzelnen wirken, ist in /FHKK83/ im Zusammenhang dargestellt. Es kann deshalb hier davon ausgegangen werden, daß Guarded Regions und Guarded Commands ausreichen, um die eingangs geschilderten Probleme zu lösen.

Ein Prozeß, der eine Guarded Region abarbeitet, prüft, ob die Bedingung eines Guards wahr ist und führt dann die zugehörige Anweisungsfolge aus. Danach gilt die nichtdeterministische Kontrollanweisung als abgeschlossen. Sind mehrere Bedingungen wahr, wird ein beliebiges Guard ausgeführt. Ist keine Bedingung wahr, wird der abarbeitende Prozeß blockiert.

Der Unterschied zwischen einer Guarded Region und einem Guarded Command besteht darin, daß ein Prozeß bei der Abarbeitung eines Guarded Commands nicht blockiert wird, wenn keine der Bedingungen wahr ist. In diesem Fall wirkt ein Guarded Command wie eine Leeranweisung.

Bei der hier vorgenommenen Spracherweiterung wurde im einzelnen so vorgegangen:

Falls innerhalb eines Guards alle Anweisungen ausführbar sind, wird der zugehörige Reaktionszweig abgearbeitet. Ist keines der Guards ausführbar, wird bei einem Guarded Command die Anweisungsfolge nach OUTREACT abgearbeitet; fehlt dieser Zweig, wirkt das Guarded Command wie eine Leeranweisung. Ist in einer Guarded Region keines der Guards ausführbar, wird gewartet, bis entweder eines von ihnen ausführbar wird oder die nach TIMEOUT angegebene Zeitdauer verstrichen ist. Danach wird der zugehörige Reaktionszweig ausgeführt. Fehlt der TIMEOUT-Zweig, wird gewartet bis eines der Guards ausführbar wird.

Dieser Vorschlag weist darüber hinaus zwei Besonderheiten auf:

Die Elemente eines Guards werden im Sinn eines logischen UND verknüpft, d.h. erst wenn alle Elemente wahr sind, wird der Reaktionszweig durchlaufen. Diese Eigenschaft ist von ausschlaggebender Bedeutung für die Lösung folgender Problemklasse: Mehrere Prozesse bearbeiten in sicherheitsrelevanter Umgebung unabhängig von einander die gleiche Aufgabe. Ein anderer Prozeß vergleicht die Ergebnisse der Aufgabe und nimmt denjenigen als richtig an, der innerhalb eines Toleranzbereichs von der Mehrheit der Bearbeitungsprozesse geliefert worden ist.

Mit Hilfe der UND-Verknüpfung von Guards können solche Probleme sehr einfach formuliert werden.

Die zweite Besonderheit ist, daß als Guard-Element nicht nur Botschaftsoperationen auftreten können, sondern auch herkömmliche Synchronisations- und Kommunikationsmittel. Das erlaubt z.B. alternativ auch auf das Freiwerden eines Semaphors, das Eintreffen eines Prozeßinterrupts oder einer Benutzereingabe zu warten. Hiermit lassen sich solche Probleme leicht formulieren, bei denen Prozesse sowohl mit anderen Prozessen als auch z.B. mit einem technischen Prozeß oder einem Benutzer kommunizieren können. An dieser Stelle ist die Integration der Spracherweiterung in die Basissprache weit vorangetrieben.

2.2.2.3. Sprachelemente zur Konfigurationsbeschreibung

Bevor die Sprachmittel beschrieben werden, die dem Benutzer zur Beschreibung der Konfiguration von Prozessen auf Prozessoren zur Verfügung stehen, wird erläutert, welchen Einschränkungen die Konfiguration des betrachteten Verteilten Systems unterliegt.

- Die Verteilung der Prozesse auf die Prozessoren ist statisch. Es wird von keiner (dynamischen) Rekonfigurationsmöglichkeit ausgegangen.

  Der Grund für diesen Ansatz ist der verhältnismäßig hohe Aufwand für eine Rekonfigurierbarkeit: Dynamische Rekonfigurierung heißt, daß bei Auftreten eines schwerwiegenden Fehlers die Aufgaben aller Prozesse auf einem Prozessor von einem Ersatzprozessor wahrgenommen werden müssen. Das verlangt, daß die Peripherie zum technischen Prozeß mehrfach vorhanden sein oder von jedem Prozessor erreichbar sein muß. Eine komfortable Prozeßperipherie dieser Art existiert nur in wenigen Fällen /BHS83/, ist im allgemeinen nicht verfügbar und kann daher keine Basis für ein ingenieurmäßig anwendbares Konzept sein. Geht man jedoch davon aus, daß Ersatzprozessoren auch über die vollständige Prozeßperipherie verfügen, dann ist die dynamische Rekonfigurierbarkeit auf genau diese Prozessoren beschränkt. In diesem Fall liegt wohl eher eine statische Redundanz vor, die mit dem vorliegenden Verfahren zu lösen ist.

- Es wird davon ausgegangen, daß zwischen Prozessoren, deren Prozesse miteinander kommunizieren, eine Verbindung besteht. Prozessoren innerhalb eines Verteilten Systems nehmen also keine Vermittlungsfunktionen wahr. Diese Forderung ist nicht unbillig. Ihr kommt jedes lokale Netz nach. Lediglich bei Netzwerkübergängen können Rechner solche Funktionen übernehmen.

Der Grund für diese Einschränkung liegt in der erheblichen Mehrbelastung eines Prozessors durch solche Aufgaben, die im Echtzeitbetrieb nicht zu vertreten ist. Umgekehrt ist beim Aufbau von Rechnernetzen die Tendenz zu beobachten, daß Aufgaben des Netzwerkzugangs vom Verarbeitungsprozessor getrennt werden.

Nach den vorgenannten Einschränkungen ist es nur nötig, auf Ebene der Programmiersprache eine eindeutige, statische Angabe darüber zu machen, auf welchem Weg ein Prozeß seinen Kommunikationspartner erreichen kann. Dabei ist es unerheblich, ob es sich bei dieser Angabe um eine lokale Anschluß-Nummer eines Prozessors oder um eine Netzwerkadresse handelt, da beide Formen ineinander überführbar sind.

Im vorliegenden Fall wurde folgendes Vorgehen gewählt:

Die Angaben zur Adressierung erfolgen im PEARL-Systemteil. Der Systemteil hat die Aufgabe einer Konfigurationsbeschreibung. Waren bisher nur Geräte (für Ein-/Ausgabe-Operationen) als Kommunikationspartner angebbar, so können jetzt auch Prozesse als Kommunikationspartner (für Botschaftsoperationen) angegeben werden. Im einzelnen wird in einer Systemteilanweisung für jeden Partnerprozeß beschrieben, über welche Anschlußnummer der zugehörige Prozessor erreicht werden kann, auf dem er untergebracht ist.
Eine weitergehende Beschreibung findet sich in /FHKK83/.

2.2.2.4. Verteilung von Programmen auf Prozessoren

Bei der Verteilung von Programmen auf Prozessoren stellt sich die Frage, in welchen Einheiten Programmteile auf verschiedene Prozessoren verteilt werden können. Geht man davon aus, daß in einem Verteilten System auch unterschiedliche Prozessoren auftreten können, müssen solche Einheiten auch übersetzbare Einheiten sein. Übersetzbare Einheiten sind in PEARL Module. Ein Modul besteht aus einem Systemteil (der der Konfigurationsbeschreibung dient) und/oder einem Problemteil (der Prozesse, Prozeduren und Daten enthält).

Somit steht mit dem Modul eine Strukturierungseinheit zur Verteilung von Programmen auf Prozessoren zur Verfügung. Berücksichtigt man die als statisch vorausgesetzte Konfiguration eines Verteilten Systems, sind zur Verteilung von Programmen auf Prozessoren einfache Minimalregeln zu beachten:

- Für jeden Modul muß bekannt sein, auf welchem Typ von Prozessor er laufen soll.
- Prozesse, die auf verschiedenen Prozessoren laufen sollen, müssen in verschiedenen Moduln definiert sein.

Diese Regeln sind notwenig aber nicht hinreichend. Von Bedeutung ist darüber hinaus, daß

- die Konfigurationsangaben in allen Moduln eines Prozessors vollständig,
- die Konfigurationsangaben in allen Moduln des ganzen Verteilten Systems widerspruchsfrei sind.

Die Überprüfung aller dieser Regeln kann zuverlässig nur durch ein Systemprogramm vorgenommen werden, das dann auch den tatsächlichen Ladevorgang kontrolliert.

2.2.3. Systemsoftware

Die Systemsoftware für "Verteiltes PEARL" wurde in zwei Schritten entwickelt. Der erste Schritt war das Programmiersystem für Ein-Rechner-Konfigurationen. Darauf aufbauend wurde es für Verteilte Konfigurationen erweitert.

Das Programmiersystem sollte zwar zunächst für kostengünstige Z80-Mikrorechner eingesetzt werden, prinzipiell aber rechnerunabhängig angelegt werden, um einen späteren Einsatz für andere, leistungsfähigere Rechner nicht auszuschließen. Es lag nahe, Gedankengut /H080/ und teilweise auch fertige Komponenten /H082/ aus einem Projekt zur Entwicklung portabler Systemsoftware für Realzeitaufgaben zu übernehmen. Erfahrungsgemäß läßt sich viel Aufwand sparen, wenn auch Komponenten aus der Hersteller-Software mit verwendet werden können. Bei dem im 8-Bit-Mikrorechnerbereich hauptsächlich vertretenen Betriebssystem CP/M sind das z.B. Assembler, Binder und Lader.

Damit war die Grundstruktur des Programmiersystems vorgegeben:

- Die Übersetzungskomponenten, bestehend aus Compileroberteil, Codegenerator, Assembler, Binder und Lader.
- Die Ablaufkomponenten Laufzeitfunktionen, Betriebssystem, Gerätetreiber.

Um der Modulstruktur von PEARL-Programmen gerecht zu werden, wird zusätzlich zum linearen Binder noch ein modulübergreifender "Modulbinder" benötigt. Die rechnerübergreifende Kommunikation in einem Verteilten System verlangt außerdem eine Netzwerkverwaltung.

Die Übersetzungskomponenten können auf Z80-Mikrorechnern mit 64 KByte Speicher und CP/M-Betriebssystem eingesetzt werden. Zur Beschleunigung des Übersetzungsvorgangs können entweder bis zu zwei RAM-Disks verwendet oder leistungsfähigere Host-Rechner (wie z.B. CYBER 170/800, VAX) mit einbezogen werden.

Die Ablaufkomponenten können den Speicherbereich von 64 KByte des Z80 ausnutzen.

Im folgenden werden die Komponenten erläutert.

2.2.3.1. Hardware

Das Programmiersystem ist zum Einsatz auf einer bestimmten Hardwarekonfiguration gedacht. Abweichende Konfigurationen sind denkbar, erfordern aber Anpassungen.

Die Rechner sind in ECB-Steckkartentechnik aufgebaut. Sie bestehen mindestens aus den Komponenten

- CPU-Karte mit Speicher, zwei seriellen und einer parallelen Schnittstelle,
- Arithmetikprozessor-Karte mit Echtzeituhr,
- Floppy-Controller.

Werden Rechner als Verteiltes System eingesetzt, erfolgt die Kommunikation über serielle Leitungen. Über diese Leitungen können auch Rechner "fern"-geladen werden, so daß in solchen Fällen auf Floppy-Controller (und Laufwerke) verzichtet werden kann.

Zum Anschluß technischer Prozesse können auf Basis des ECB-Busses zahlreiche Interface-Karten zum Einsatz kommen, wie z.B.

- Analog-Ein-/Ausgabe (statisch),
- Digital-Ein-/Ausgabe,
- Zähler,
- impulsgesteuerte Analog-Ein-/Ausgaben
- weitere serielle Schnittstellen.

2.2.3.2. Übersetzungskomponenten

Das rechnerunabhängige Compileroberteil /ESG77/ übersetzt ("Verteilte") PEARL-Moduln in die rechnerunabhängige Zwischensprache CIMIC /ESG80/, einer abstrakten Assemblersprache. Die Übersetzung erfolgt in 11 aufeinander folgenden Phasen und gliedert sich grob in Syntaxanalyse, Namensbuchbearbeitung und Semantikfunktionen. Für die im Verteilten PEARL neu hinzugekommenen Sprachkonstrukte, Botschaftsoperationen und Guarded Statements, wurden neue Verarbeitungsschritte hinzugefügt. Die Abarbeitung der Guarded Statements z.B. entspricht der CASE-Anweisung. Die erzeugte CIMIC-Sprache ist streng formatiert, enthält noch die (aus PEARL) bekannten Datentypen und Betriebssystemaufrufe sowie bereits aufgelöste Kontrollstrukturen und Ausdrücke. Das Compilerteil ist vollständig in FORTRAN geschrieben.

Aus CIMIC erzeugt der Codegenerator Assemblercode. Die Umsetzung erfolgt in zwei Stufen. Die erste Stufe setzt den mnemonischen CIMIC-Code in komprimierten Zahlencode um, die zweite Stufe erzeugt hieraus Assemblercode. Der Codegenerator besteht aus einem festen Gerüst und variablen, rechnerabhängigen Teilen, die für jeden neuen Zielrechner angepaßt bzw. neu geschrieben werden müssen. Der Codegenerator ist in FORTRAN geschrieben.

Aus den Assembler-Quellversionen der Anwendermoduln erzeugt der CP/M-Makroassembler Objektprogramme mit verschiebbarem Code, der noch gebunden werden muß.

Aus allen globalen Daten der Anwendermoduln erzeugt der Modulbinder einen globalen Modul, in dem u.a. die Daten- und Kontrollstrukturen des gesamten Programms enthalten sind. Auch dieser Modul muß noch vom Assembler in ein verschiebbares Objekt umgewandelt werden. Der Modulbinder ist in FORTRAN geschrieben.

Sämtliche erzeugten Objekte werden von einem CP/M-Binder (L80 oder PLINK) zusammen mit Initialisierungsprozeduren, PEARL-Betriebssystem, Treibern und benötigen Laufzeitfunktionen zu einem festadressierten, ladbaren Programm gebunden.

Die Steuerung des gesamten Übersetzungsvorgangs erfolgt mit Hilfe von Kommandoprozeduren.

Bild 6 zeigt das Zusammenwirken der Übersetzungskomponenten.

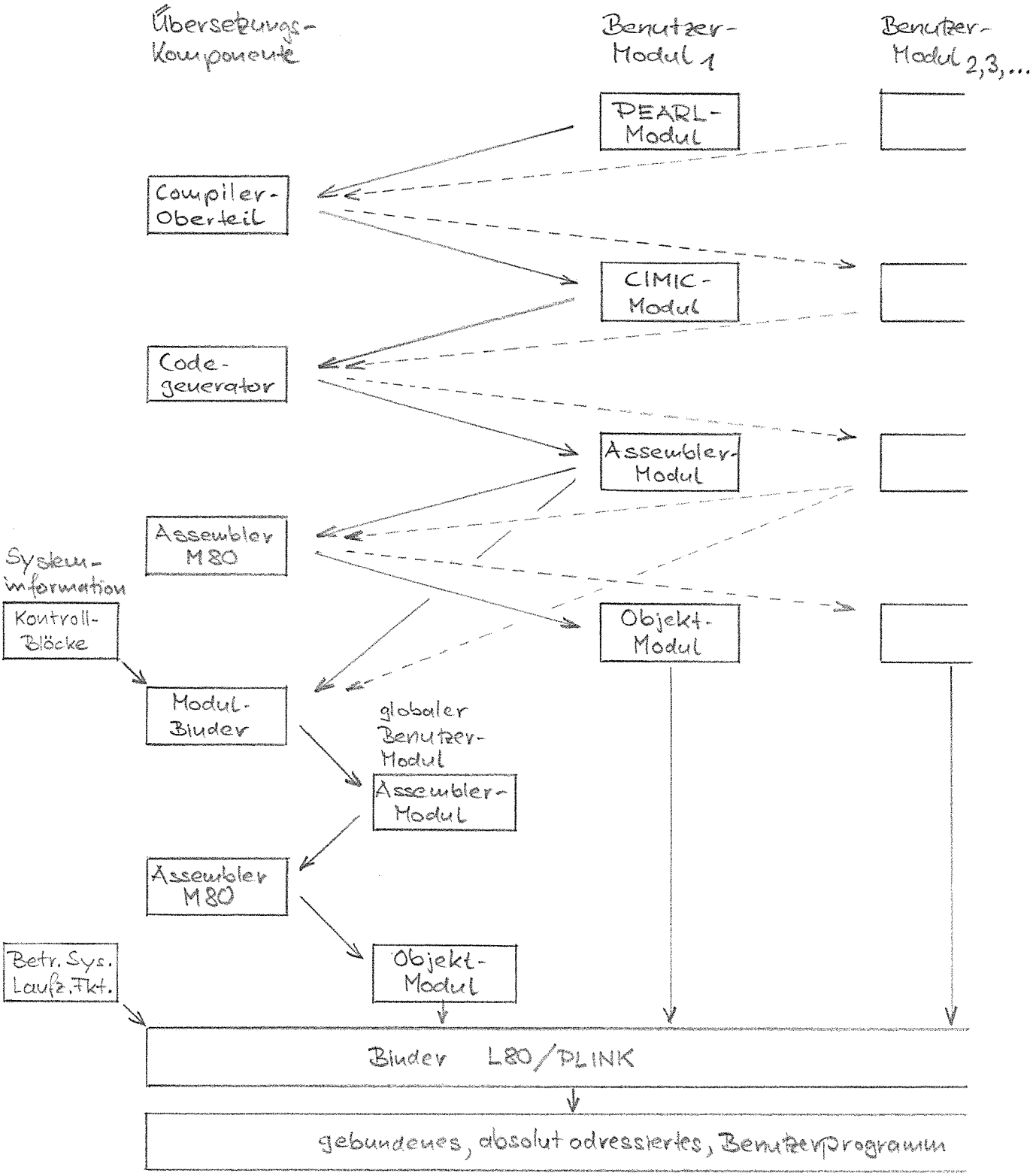

Bild 6: Zusammenwirken der Übersetzungskomponenten

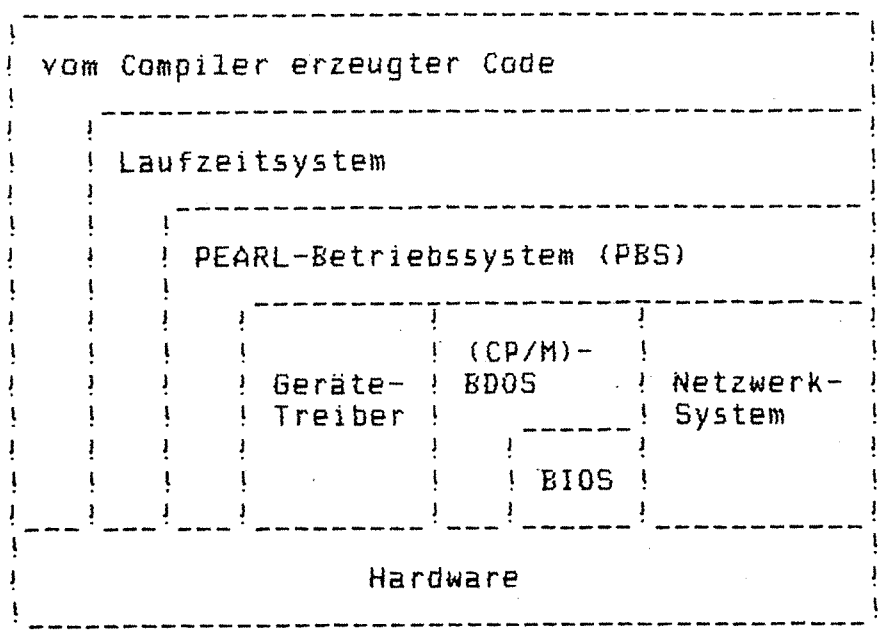

Bild 7: Schichtenmodell der Ablaufkomponenten

2.2.3.3. Ablaufkomponenten

Die Ablaufkomponenten unterteilen sich in die Schichten Laufzeitfunktionen, Betriebssystem, Treiber und Netzwerkverwaltung. Die Laufzeitfunktionen dienen der Abarbeitung der vom Codegenerator nicht aufgelösten Befehlsfolgen und rufen ggf. das Betriebssystem auf. Die Laufzeitfunktionen sind reentrantfähig und erlauben reentrantfähige Anwenderprozeduren. Sie wickeln z.B. die formatierte Ein-/Ausgabe, die Zugriffe auf CP/M-Dateien und die Operationen auf den PEARL-Datentyp ab. Sie sind teils in PEARL (z.B. die Formatierungsroutinen), teils in Assembler geschrieben.

Das PEARL-Betriebssystem (PBS) realisiert herkömmlich die direkte Tasksteuerung, die Synchronisierung, die Abarbeitung von (zeitlichen) Einplanungen und die Geräteverwaltung. Hinzu kommen die Funktionen zum Austausch von Botschaften und zur Verwirklichung von Guarded Statements.

Da nicht jedes Anwendungsprogramm die gleichen Anforderungen an das Betriebssystem stellt, läßt sich das PBS in Abhängigkeit vom Anwenderprogramm konfigurieren. Folgende Varianten sind möglich:

Für Ein-Rechner-Konfigurationen:

- Herkömmliches Betriebssystem,
- Betriebssystem mit Guarded Statements.

Für Verteilte Rechner:

- Betriebssystem mit Botschaften,
- Betriebssystem mit Botschaften und Guarded Statements.

Das PBS ist größtenteils in einem abstrakten Assembler SPASS /R078/, teils in Assembler geschrieben.

Die (Geräte-) Treiber sorgen für Datentransfer und Interruptverarbeitung an den Peripherieschnittstellen. Gerätetreiber gibt es für die unter 2.3.3.1. genannte Standard- und Prozeßperipherie. Sie können - z.B. für spezielle Geräte - von Anwendern selbst erstellt werden. Gerätetreiber sind aus Effizienzgründen in Assembler programmiert.

Die Netzwerkverwaltung vereinigt zwei Funktionen in sich. Sie enthält einen Ladeteil, der von einem willkürlich bestimmbaren Masterprozessor die Satelliten im Verteilten System mit Programmcode versorgt. Die Satelliten benötigen keinen Hintergrundspeicher (Floppy, Diskette etc.). Je nach Qualität des unterlagerten Transportsystems kann eine gesicherte oder ungesicherte Ladeprozedur gewählt werden.

Das Netzwerksystem enthält außerdem Leitungstreiber, die den Austausch der Botschaften zwischen den Betriebssystemen über die Übertragungshardware abwickeln. Die Leitungstreiber werden automatisch an die jeweilige Hardwarekonfiguration des Verteilten Systems angepaßt. Die Informationen über die Konfiguration werden dem PEARL-Systemteil entnommen. Die Leitungstreiber können ebenfalls wahlweise eine Sicherungsprozedur vorsehen. Das Netzwerksystem ist wie das Betriebssystem in SPASS formuliert und weitgehend portabel.

Bild 7 zeigt die Ablaufkomponenten.

2.2.4. Erfahrungen

Die Erfahrungen mit der Programmiersprache lassen sich unter zwei Aspekten betrachten:

- Erfahrungen mit den vorgenommenen Erweiterungen,
- Erfahrungen mit der gesamten Sprache.

Das Konzept der vorgenommenen Erweiterungen ist in der Spezifikationstechnik verankert, es wird also auch dort diskutiert. Hier steht nur die sprachliche Ausprägung zur Diskussion.

Die eingangs geforderte Asynchronitätsabstraktion ist dadurch erfüllt, daß die neu eingeführten Guarded Statements eine Nicht-Blockierungsbedinung enthalten. Die Prozeßabstraktion ist trivialerweise erfüllt.

Bei den Botschaftsoperationen zeigt sich, daß die als Parameter zugelassenen Strukturen eine zu starke Einschränkung darstellen. Hier wären eine Auswahl von Parametern, wie sie auch in Prozeduraufrufen vorkommen, sinnvoller.

Bei den nichtdeterministischen Kontrolloperationen zeigt sich, daß die zur Verfügung stehende Zeitüberwachung zu wenig sensitiv auf grundsätzlich verschiedene Fehlerursachen reagiert. Bei einem Ablauf der Zeitüberwachung weiß man z.B. nicht, ob der angesprochene Prozeß, der Prozessor oder die Verbindung zum angesprochenen Prozessor ausgefallen ist. Hier müßte eine genauere Fehlerdiagnose möglich sein.

Bei der Anwendung der Programmiersprache stellte sich heraus, daß zwar die Hilfsmittel für eine Kommunikation über Botschaften recht komfortabel geworden sind, daß aber bei der Verwirklichung gemeinsamer Objekte zur Zugriffssynchronisation nur Semaphore zur Verfügung stehen. Hier könnten z.B. Monitore in der Sprache eine wirksame Unterstützung darstellen.

Die Erfahrungen mit der Systemsoftware konzentrieren sich auf drei Stellen: Compiler, Betriebssystem, Binder.

Der in Fortran geschriebene, stark modularisierte Compiler erwies sich an anfänglichen Einarbeitungsproblemen als relativ freundlich. Um eine möglichst hohe Anpaßbarkeit an neue Zielmaschinen zu erreichen, wurden allerdings mehrere Zwischenschritte bei der Codeerzeugung eingeführt. Dabei überführt ein Zwischenschritt oft eine Interndarstellung des Programms in eine andere, die sich nur geringfügig von der Ausgangsversion unterscheidet, wodurch sich der Übersetzungsablauf verlangsamt. Hier wäre eine grundsätzliche Überarbeitung des Übersetzungssystems nötig.

Das verteilte PEARL-Betriebssystem ist historisch aus der Ein-Prozessor-Version hervorgegangen. Die Eingriffe bei der Erweiterung waren recht schwerwiegend, so daß das entstandene verteilte Betriebssystem nicht mehr als modular betrachtet werden kann. Eine nochmalige Erweiterung dürfte deshalb auf Schwierigkeiten stoßen. Auch hier bietet sich eine Überarbeitung an.

Das Arbeiten mit einem Verteilten System zwingt zu einer starken Modularisierung der Programme. Fehler im Zusammenwirken von Moduln werden wahrscheinlicher, sie durch Testen zu finden schwieriger und unwirtschaftlicher. Eine automatische Überprüfung von Modulschnittstellen (siehe auch 2.2. und 2.4.) wäre hilfreich und notwendig.

Bei der Lösung von Realzeitaufgaben gibt es allerdings eine Schlüsselfrage, die den praktischen Wert eines Programmiersystems entscheidend mitbestimmt: die Laufzeiteffizienz. Sie wurde insbesondere bei Anwendungen in der Experimentalphysik und in der Kommunikationstechnik zum Prüfstein. Es zeigte sich, daß die Laufzeiteffizienz der vom PEARL-Compiler/Codegenerator im Vergleich zu anderen (z.B. FORTRAN-) Compilern erzeugten Codes stellenweise geringer war. Das lag hauptsächlich daran, daß manche Eigenschaften von modernen Programmiersprachen sich nur aufwendig auf eine 8 bit Struktur, wie die des Z80, aufprägen lassen. Demgegenüber überzeugte das Zeitverhalten von Betriebssystem und Treibern bei der Bearbeitung von Betriebssystemaufrufen und Interrupts. So wurden die Grenzen der Einsetzbarkeit in der Regel durch die Leistung des verwendeten Mikroprozessortyps Z80 festgelegt.

Daß es auch negative Erfahrungen gab, schmälerte angesichts der Flexibilität und Leistungsfähigkeit der Systemsoftware nicht den positiven Gesamteindruck.

2.3. Testumgebung

Methodisches Testen steht bei Software-Entwicklungsprojekten oft im Hintergrund von Spezifikation und Programmierung. Funktions- und Leistungsüberprüfungen werden bei Automatisierungsproblemen historisch auf der niedrigsten Programmierebene, Assembler oder gar Maschinencode vorgenommen. Hier soll der Versuch unternommen werden, für solche Probleme eine Testumgebung anzubieten, die komfortablen Spezifikations- und Programmierungstechniken angepaßt ist.

Beim Testen kann grundsätzlich unterschieden werden, auf welcher Abstraktions- bzw. Konkretheitsebene ein Automatisierungsprogramm untersucht wird (z.B. auf Spezifikations-, Programmiersprachenebene).

Weniger klar ist die Unterscheidung, durch welchen Übergang im Software Life Cycle sich Fehler eingeschlichen haben.

So können sich beim Testen z.B. noch Fehler in der Systemsoftware (Compiler, Betriebssystem, ...) herausstellen. Davon soll hier aber nicht ausgegangen werden: Die Systemsoftware wird als korrekt angenommen. Als wesentliche Fehlerquellen bleiben also:

- Die Implementation: Es sind Fehler beim Übergang von der Spezifikation zum Programm entstanden. Hier kann eine formale, weitgehend automatische Umsetzung Fehlerursachen vermeiden helfen.

- Der Entwurf: Es sind Fehler dadurch entstanden, daß die Anforderungsdefinition unvollständig oder inkonsistent war oder vom Entwerfer mißverstanden wurde. Der einfachste Fall dürfte vorliegen, wenn keiner dieser Gründe vorliegt, der Entwerfer aber eine in sich inkonsistente Spezifikation erstellt hat.

Eine besondere Art von Fehler liegt vor, wenn das letztlich entstandene Automatisierungsprogramm zwar unter gewissen Umständen richtig arbeitet, aber die geforderten Zeitbedingungen nicht einhält. Ob dieser Fehler dem Entwurf oder der Implementation zuzuordnen ist, ist allgemein nicht zu klären.

Letztes grundsätzliches Unterscheidungsmerkmal ist, was als Referenzobjekt zum Test dient. Als Testpartner kann auftreten:

- Der technische Prozeß selbst, was sicher nur der Fall sein darf, wenn das Programm eine gewisse Reife erreicht hat;
- eine Nachbildung von wesentlichen Teilen des technischen Prozesses, z.B. der Sensoren und Stellglieder;
- eine Nachbildung des technischen Prozesses durch entsprechende Rechenprozesse.

Als Entscheidungsmaß, ob ein Fehlverhalten vorliegt, stehen zur Verfügung:

- Die Anforderungsdefinition,
- die Abnahmebedingungen.

Auf welcher Ebene, ob Spezifikations- oder Programmebene, letztlich getestet wird, kann sowohl prinzipiell als auch pragmatisch beantwortet werden.

- Es ist sinnvoll, immer auf der höchstmöglichen Abstraktionsebene zu testen, da man sich ggf. kostspielige Umwege durch Rückkehr zur nächsthöheren Ebene erspart.
- Der Anwender wird sicher der Testmethode den Vorzug geben, die ihm die bequemsten Hilfsmittel bietet.

In den folgenden Abschnitten sollen Testwerkzeuge vorgestellt werden, die dem Benutzer erlauben, sein Programm in einen im Sinne der Anforderungsdefinition korrekten Zustand zu bringen. Unabhängig davon ist zu sehen, daß zwar die Existenz solcher Werkzeuge schon von Wert ist, der Benutzer aber durchaus bei ihrer Handhabung unterstützt werden sollte. Dem ist durch einige methodische Ansätze beim Testen auf Spezifikationsebene Rechnung getragen, eine breitere Hilfe bedarf aber einer eigenen Diskussion (Kapitel 3.3.).

2.3.1. Testen auf Spezifikationsebene

Testen auf Spezifikationsebene bedeutet, ähnlich wie Testen auf Programmiersprachenebene, mit den Konstrukten dieser Ebene den Programmablauf zu analysieren oder in den Ablauf eingreifen zu können.

Es muß zu einer Analyse also möglich sein

- den Zustand eines Prozesses festzustellen, d.h. in welchem Kommunikations- oder Internzustand er sich befindet,
- das Vorhandensein von Botschaften zu überprüfen, d.h. von welchem Absender welche Botschaften vorliegen,
- das Ergebnis einer internen Funktion bzw. Operation feststellen zu können, d.h. den Rückgabewert oder den Erfolg/Mißerfolg mitgeteilt zu bekommen.

Zur Veranschaulichung komplexer Vorgänge wäre es darüber hinaus sinnvoll, eine zustandsorientierte Ablaufverfolgung durchführen zu können.

An Eingriffsmöglichkeiten ist wünschenswert:

- Prozesse gezielt in einen Zustand, z.B. den Initialzustand versetzen zu können.
- Ergebnisse von internen Funktionen bzw. Operationen zu erzeugen.
- Botschaften künstlich erzeugen oder löschen zu können.

Ist das zu einer Spezifikation gehörende Programm vollständig implementiert und ablauffähig, stellt die Verwirklichung dieser Forderungen kein gravierendes Problem dar. Da die Zeit zwischen Erstellen der Spezifikation und dem Vorliegen des Programms aber in der Regel sehr groß ist, hat diese Methode wenig Sinn, um rechtzeitig Fehler in der Spezifikation zu finden.

Um darüber schneller Aussagen treffen zu können, ist es notwendig, aus einer Spezifikation "sofort", dafür unvollständige und trotzdem ablauffähige Objekte zu erzeugen.

Eine Möglichkeit ist, besonders zum Test des Kommunikationsverhaltens eines Prozesses, die Ablaufsteuerung vollständig in ein Programmgerüst umzusetzen, über die internen Operationen und Funktionen aber nur bestimmte Annahmen zu machen. Auch die zu empfangenden Botschaften, die Schnittstellen zum Benutzer und zum technischen Prozeß können auf diese Weise angenähert werden.

Mit Hilfe dieser Methode ist es möglich, einen ersten Eindruck über die Kooperation von Prozessen miteinander und mit ihrer Umwelt zu gewinnen.

2.3.2. Testen auf Programmiersprachenebene

Sprachorientierte Testhilfen für sequentielle Programmiersprachen (ein Rechenprozeß) können im allgemeinen vorausgesetzt werden.

Für quasi-parallele Programmiersprachen (mehrere Rechenprozesse auf einem Prozessor), wie PEARL, beginnen sich Testhilfen durchzusetzen /BMW81, MA81/, für verteilte Programme (mehrere Rechenprozesse auf verschiedenen Prozessoren) sind erste Ansätze bekannt /HS81, LA78/.

Für alle drei Arten von Programmiersprachen kann man folgende (minimale) Testfunktionen voraussetzen:

- Anhalten des Programms durch Setzen von Haltepunkten;
- Kontrollieren und Ändern von Variablenwerten;
- Verfolgen des Kontrollflusses (z.B. Zeilenprotokoll).

Für quasi-parallele und verteilte Programme kommen noch folgende Funktionen (mindestens) hinzu:

- Kontrollieren von Prozeßzuständen und Ändern von Zuständen, wie z.B. durch Starten von Prozessen;
- Eingreifen in das Synchronisationsverhalten, wie z.B. durch das Verändern von Semaphorwerten.

Während die Testfunktionen für sequentielle Programme Allgemeingut sind, gilt das Augenmerk den Testfunktionen für quasi-parallele und verteilte Programme. Sie können unter mehrerlei Aspekten diskutiert werden:

- Welche Funktionen braucht man im einzelnen für die hier vorgeschlagene Programmiersprache?
- Welche Probleme muß man grundsätzlich beim Verwenden von Testhilfen erwarten?
- Wie lassen sich solche Testhilfen implementieren?

2.3.2.1. Testfunktionen und ihre Verwendung

Testfunktionen gliedern sich üblich in Auskunftsfunktionen und manipulierende Funktionen (siehe auch /AFHT85/). Die Objekte, mit denen diese Funktionen arbeiten, sind in der Regel vom Betriebssystem verwaltete Objekte.

An Auskunftsfunktionen bieten sich neben den herkömmlichen sequentiellen Angaben an:

- Status aller Prozesse (Priorität, Zustand),
- Semaphor-Status (aktueller Wert, Warteschlange),
- Geräte-Status (belegender Prozeß, Warteschlange),
- Interrupt-Status (Warteschlange),
- Prozessor-Status (Warteschlange).

Für die Botschaftsmechanismen kommen an Funktionen hinzu:

- Status der Botschaftspuffer
  (Inhalt des Warte- und Blockiertbereichs, Anzahl der Botschaften im Wartebereich des Prozesses).

An Manipulationsmöglichkeiten sind erlaubt:

- Zustandsänderung von einzelnen Prozessen
  (innerhalb des PEARL-Zustandsmodells);
- Anhalten und Fortsetzen des gesamten Prozeßsystems;
- Ändern von Semaphor-Werten;
- Simulieren eines Interrupts.

Zur Ablaufverfolgung gibt es ein zeilenweises Protokoll mit den Funktionen

- Setzen und Löschen von Trace-Bereichen,
- Ein- und Ausschalten des Trace-Vorgangs.

Der Gebrauch mancher Testfunktionen hat bei quasi-parallelen und verteilen Programmen, im Gegensatz zu sequentiellen Programmen, einen starken Einfluß auf das Zeitverhalten und evtl. auch auf das Gesamtverhalten eines Prozeßsystems, wie in /AFHT85/ erläutert wird. Dies trifft selbst für den Gebrauch herkömmlicher sequentieller Testmittel zu.

- Bereits das Kontrollieren von Variablen bewirkt eine Änderung des Zeitverhaltens des zu testenden Programms dadurch, daß dessen Operationen durch Operationen des Testsystems unterbrochen werden. Durch den verzögerten Ablauf des steuernden Programms können zeitbedingte, scheinbar nichtdeterministische Fehler auftreten.

- Die Funktion Zeilentrace bewirkt beim zu testenden Prozeß eine Verlängerung der Programmlaufzeit, da bei jedem ausgeführten Statement eine Meldung an das Testsystem erfolgen muß. Das führt auch zu den oben genannten Konsequenzen.

- Das Ändern von Prozeßzuständen (auch über Semaphore) stellt offensichtlich einen gravierenden Eingriff in ein Prozeßsystem dar und hat z.B. nur dann einen Sinn, nach einer Verklemmung noch Restfunktionen zu testen und so Testzeit zu sparen.

- Das Anhalten eines Prozesses kann zu einer Verklemmung des gesamten Prozeßsystems führen, da über Synchronisationsoperationen bzw. Zeiteinplanungen Querbezüge zwischen den Testobjekten und dazu parallelen, nicht getesteten Prozessen existieren.

- Das Anhalten aller Prozesse auf einem Prozessor vermeidet eine einseitige Verschiebung der Operationen eines Prozesses gegenüber Operationen anderer, dazu paralleler Prozesse. Es tritt nur eine Zeitverschiebung gegenüber externen Prozessen (technischen Prozessen bzw. Prozessen auf anderen Prozessoren) auf. Bei rein quasi-parallelen Programmen ist diese Funktion sehr sinnvoll, da nach dem Anhalten eine konsistente Information über den Zustand des gesamten Prozeßsystems abgefragt werden kann. Bei verteilten Programmen wird das Synchronisationsverhalten gestört.

- Das Anhalten aller Prozesse in einem Verteilten System ist schwierig, allein mit Softwarehilfe nicht sondern nur durch eine Kombination von Software und Hardware zu verwirklichen. Die Bearbeitungszeiten für die Erkennung einer solchen Funktion sind lang gegenüber der Ausführungszeit einer Programmanweisung. Diese Funktion ist somit wertlos, da der zeitliche Zusammenhang zwischen den Operationen im Verteilten System verloren gegangen ist. Lediglich bei verteilten Programmen mit gemeinsamem Speicher ließe sich solch eine Funktion sinnvoll verwenden.

Testhilfen auf Programmiersprachenebene führen also nur dann zu aussagekräftigen Ergebnissen, wenn die "Umgebung" eines Eingriffs untersucht und die Wechselwirkung zum Rest des Prozeßsystems eingegrenzt wurde.

2.3.2.2. Instrumentieren von Testfunktionen

Instrumentieren eines (zu testenden) Programms heißt, eine Verbindung zwischen dem Testsystem und dem (Code des) zu testenden Programms herzustellen, die zur Laufzeit des zu testenden Programms Eingriffe in das Ablaufverhalten des Testobjekts erlaubt. Die gängigen Verfahren zur Codeinstrumentierung sind in /AFHT85/ näher erläutert. Hier genügt deswegen, kurz auf die "dynamische" Instrumentierung einzugehen, die für die vorliegende Aufgabe, durch Crosscompilierung erzeugte Echtzeitprogramme zu testen, ausgewählt wurde.

- Sie verändert das Echtzeitverhalten des Testobjekts zwischen Testpunkten nicht, da nur der ursprüngliche Programmcode durchlaufen wird.

- Zum Testen ist kein eigener Übersetzungslauf nötig, was bei Crosscompilation viel Zeit spart.

- Im Gegensatz zu anderen Instrumentierungstechniken wird der Code des zu testenden Objekts nicht expandiert, was bei einem beschränkten Speicherausbau von Vorteil ist.

Auf Basis dieser Instrumentierungstechnik wurde ein Testsystem entwickelt, das auf Ein- wie Mehrprozessor-Konfigurationen läuft und die erläuterten Testfunktionen zur Verfügung stellt. Es ist in /AFHT85/ näher beschrieben.

Das Testsystem wird in Form von zwei miteinander kommunizierenden Prozessen, einem Dialog-Prozeß und einem Test-Prozeß, realisiert. Bei einer Monoprozessor-Konfiguration bilden beide Prozesse zusammen das Testsystem für diesen Rechner.

In Mehrrechner-Konfigurationen läuft der Dialog-Prozeß auf dem Prozessor des Verteilten Systems, der mit einem Bedienterminal ausgerüstet und mit den anderen Prozessoren verbunden ist. Der Test-Prozeß läuft stets auf dem Prozessor, der das zu testende Programm beherbergt (dies kann auch der Bedienrechner sein).

Diese Konfiguration unterstützt Mikroprozessorsysteme, deren Frontend-Prozessoren üblicherweise über keine Bedienperipherie verfügen.

Der Dialog-Prozeß, bestehend aus Kommandointerpreter und Protokollaufbereitung, interpretiert die vom Anwender abgegebenen Testaufträge (Kommandointerpreter) und komprimiert sie in Form von Botschaften, die dem Test-Prozeß übergeben werden. Der Test-Prozeß führt die Testaufträge durch und gibt die Ergebnisse oder Bestätigungen in Form von Botschaften an den Dialog-Prozeß zurück. Der Dialog-Prozeß entkomprimiert die Ergebnisse (Protokollaufbereitung) und protokolliert sie auf dem Bedienterminal.

Das Testsystem wurde (für Z80-Mikrorechner) in zwei Versionen entwickelt:

- Testsystem für Monorechner-Konfigurationen (quasi-parallele Programme),
- Testsystem für Verteilte Systeme (echt-parallele Programme).

Der Dialogteil des Testsystems ist in PEARL, die Testfunktionen sind wie das PEARL-Betriebssystem in SPASS geschrieben.

2.3.2.3. Erfahrungen

Erfahrungen mit dem Testsystem betreffen hauptsächlich Aspekte der Implementierung.

- Da der Compiler, außer für globale Größen, nur generierte Namen ablegt, ist Kontrolle und Ändern von Variablen nur für globale Größen möglich.

- Der Speicherumfang für das Testsystem ist für Z80-Mikrorechner mit maximal 64 KByte Arbeitsspeicher ziemlich umfangreich. Diese Einschränkung ist für 16-Bit-Mikrorechner bedeutungslos.

3. Handhabung der Entwicklungswerkzeuge

Auch wenn dem Benutzer eine geeignete Spezifikationstechnik und Programmiersprache zur Verfügung stehen, bleibt ihm das Problem, aus einer Anforderungsdefinition ein korrektes Programm zu entwickeln. Normalerweise folgt er bei diesem Schritt seiner Intuition. Das läßt zwar der persönlichen Entfaltung freien Raum, eröffnet aber auch viele Fehlermöglichkeiten. Es kann ein Programm entstehen, das der Anforderungsdefinition nicht entspricht.

Es liegt nahe, dem Anwender beim Entwurfs-, Implementierungsvorgang bzw. Testvorgang, d.h. bei Transformationen, zu unterstützen, ohne ihn aller intuitiver Einflußnahme zu berauben.

Eine Entwicklung solcher Hilfsmittel nimmt, da hier stark Erfahrungen zum Tragen kommen, viel Zeit in Anspruch und steht demgemäß erst am Anfang. Es sollen deshalb nur die Anforderungen und ggf. auch Ansätze vorgestellt werden.

3.1. Entwurf

In der Entwurfsphase soll ein Modell des gesamten Softwaresystems entwickelt werden, das, als ablauffähiges Programm, die gestellten Anforderungen erfüllt /KKST79/. Endprodukt der Entwurfsphase ist die Spezifikation. Verwendet man eine bestimmte Methode zur Erstellung einer Spezifikation, wie z.B. PASS, so kann sie begrifflich und strukturell den Entwurfsvorgang prägen. Es stellt sich die Frage, ob die Denkweise und Strukturierungselemente von PASS auch Grundlage von Hilfsmitteln beim Entwurf von Automatisierungs-/Realzeitprogrammen sein können.

Verschiedene Anforderungen müssen durch ein solches Hilfsmittel erfüllt sein:

- Zur Unterstützung eines ingenieurmäßigen Einsatzes sollte es leicht handhabbare (graphische) Elemente enthalten.

- Es sollte stufenlose Verfeinerungsmöglichkeiten aufweisen, um dem üblichen top-down-Vorgehen entgegen zu kommen. Umgekehrt sollten auch Vergröberungen möglich sein, um entweder Rückgriffe im Entwurfsvorgang zu erlauben oder um zu detaillierte Anforderungen auszugleichen.

- Es sollte unabhängig sein von der Art der Anforderunsdefinition, die in der Regel vom Auftraggeber eines Projekts, außerhalb des Wirkungskreises des Entwicklers, erstellt wird.

Dennoch muß man sich vergegenwärtigen, daß das angestrebte Hilfsmittel lediglich eine Leitlinie darstellen darf. Dem Entwerfer muß genügend Freiraum für Intuition bleiben.

Bei der Entwicklung eines geeigneten Hilfsmittels sollten existierende, konventionelle Ansätze, wie z.B. SADT und PSL/PSA, trotz ihrer bekannten Schwächen mit betrachtet werden.

3.2. Implementation

Implementation ist die Umsetzung der Spezifikation in ein Programm /KKST79/.

Die Merkmale der Spezifikationstechnik PASS und die Formulierungsmöglichkeiten der Programmiersprache "Verteiltes" PEARL sind so weit aufeinander abgestimmt, daß eine solche Umsetzung stark schematische Züge hat.

So finden sich folgende in der Spezifikationstechnik geforderten Elemente in der Programmiersprache wieder:

- parallele Prozesse,
- Kommunikation über Botschaften,
- gemeinsame Objekte,
- nichtdeterministische Kontrolloperationen.

Es ist beabsichtigt, aus der Spezifikation heraus mit Rechnerunterstützung Programmcode erzeugen zu lassen. Dieser Code muß nicht unbedingt vollständig sein, jedoch kritische Passagen unterstützen, d.h. es sollte mindestens ein Coderahmen mit Kommunikationsfunktionen erzeugt werden.

Von einer (auch nur unvollständigen) rechnergestützten Umsetzung kann eine Beseitigung weiterer Fehlerquellen in der Programmierung erwartet werden.

Ziel der Implementierung ist es, ausgehend von der Spezifikation, ein in der gewünschten Systemumgebung lauffähiges Programmpaket herzustellen, das so codiert und dokumentiert ist, daß es übersichtlich, leicht verständlich und Korrektheitsbetrachtungen zugänglich ist /KKST79/.

Ziel des hier verfolgten Ansatzes ist es, durch Überprüfungen auf Spezifikationsebene und durch eine möglichst automatische Umsetzung der Spezifikation in ein PEARL-Programmgerüst die Zuverlässigkeit der zu erstellenden Prozeßsoftware zu erhöhen /KRAG86/.

Durch die automatische Umsetzung in Code (Programmsynthese) soll erreicht werden, daß sich der oder die Entwickler weniger mit der Programmierung als mit der Spezifikation beschäftigen. Darüber hinaus soll der für die Programmentwicklung benötigte Zeitaufwand reduziert werden.

Weitere Anforderungen sind eine Verbesserung der Dokumentation durch Ausgabe der in der Spezifikationsphase gesammelten Informationen. Die Dokumentation kann nach jeder Änderung in der Spezifikation auf Anforderung ausgegeben werden.

Die einheitliche Struktur der automatisch erzeugten PEARL-Programmgerüste mit den darin enthaltenen Kommentaren sollen ein leichteres Ändern und Ergänzen und späteres Warten der Programme ermöglichen.

Die Programmierumgebung wird modular aufgebaut, so daß vom Anwender gegebenenfalls nur die Werkzeuge zur Unterstützung der Spezifikation (ohne Programmsynthese) verwendet werden können. Dies ist vorteilhaft, wenn als Zielsprache nicht PEARL gewünscht wird.

Bei der Umsetzung nach PEARL wird sowohl die Programmierung im Großen, d.h. die Bildung einer Modulstruktur, als auch das Programmieren im Kleinen auf Anweisungsebene unterstützt.

Bei der Synthese von PEARL-Programmen sind drei Aspekte von Bedeutung. Zum einen wird zur Codeerzeugung Information aus den in der Spezifikation erzeugten graphischen Objekten verwendet. Zum anderen liegt der Schwerpunkt bei der Synthese auf Sprachkonstrukten zur zeitüberwachten Kommunikation von Prozessen über Botschaften; d.h. auf Sprachkonstrukten, die in Verteilten Systemen benötigt werden.

3.3. Methodisches Testen

Auch mit komfortablen (und abstrakten) Testwerkzeugen, wie den in 2.3. beschriebenen, ist der Programmentwickler bei Fehler-Entdeckung und -Beseitigung auf gewisse Hilfestellungen angewiesen.

Die Praxis sieht wohl so aus:

- Der Fehler zeigt sich im Betrieb des Programms. Das Testwerkzeug dient lediglich zum Einkreisen und Lokalisieren des Fehlers.

- "Man kennt" typische Fehler (z.B. eines Programmierers, einer Methode) und sucht sie prophylaktisch.

- Werden Programme für einen unter verschiedenen Aspekten immer wiederkehrenden Anwendungsfall geschrieben, gibt es ein anwendungstypisches Fehlverhalten, z.B. Deadlocks in Kommunikationsprotokollen.

Außer beim ersten Fall, wo "Kunden" zur Fehlersuche mißbraucht werden, bedarf es also zahlreicher Versuche oder eines gewissen Maßes an Erfahrung, um Fehler finden zu können. Anhand des Testens auf Spezifikationsebene, der abstrakt höchsten verfügbaren Ebene, sollen zwei Ansätze genannt werden, dem Programmentwickler beim Handhaben seiner Testwerkzeuge zur Seite zu stehen.

- Eine Möglichkeit besteht darin, mit der aus der Spezifikation stammenden Rumpfimplementation gewisse Testreihen (vom Rechner) selbständig durchführen zu lassen. Dazu können z.B. die Ergebnisse von internen Funktionen bzw. Operationen durch Aufrufe eines Zufallszahlengenerators oder durch konstante, wahrscheinliche Werte gesetzt werden. Auch der technische Prozeß und der "Benutzer" wird so simuliert, daß er wahrscheinliche oder zufallsgesteuerte Werte liefert.

  Analysen der Testaufzeichnungen lassen Aussagen über Entwurfsfehler, eventuelle Deadlocks und das Zeitverhalten zu.

  Arbeiten die Tests mit Zufallszahlen, kann es zu "sinnlosen" Testabläufen kommen, die verworfen werden müssen. Andererseits lassen solche Tests Aussagen über die "Robustheit" eines Programms zu.

Der Übergang zwischen Implementation und Test ist fließend: Ist ein Modul fertig implementiert, wird er statt des Testobjekts eingesetzt.

Das Verfahren hat allerdings dort seine Grenzen, wo es, weil "blind", zu zeitaufwendig und unübersichtlich wird.

- Eine andere Möglichkeit besteht darin, das üblich beim Testen angewandte (intuitive) Vorgehen zu untersuchen, das Wissen von erfahrenen Testern zu sammeln und dem Programmentwickler das Wissen zusammen mit den Testwerkzeugen zur Verfügung zu stellen. Ein solches Wissen dürfte in der Regel anwendungs-spezifisch geprägt sein. Bei dem Test von Kommunikationsprotokollen wird von dieser Technik bereits Gebrauch gemacht.

4. Anwendungen

Die hier vorgestellte Programmiertechnik, die Spezifikationstechnik PASS und die Programmiersprache (Verteiltes) PEARL, werden inzwischen in ca. zehn Projekten auf ca. zwanzig Mikrorechnern eingesetzt. Darüber hinaus gibt es auch Anwendungen, in denen die Spezifikationstechnik in Verbindung mit anderen Programmiersprachen (und anderen Rechnern) eingesetzt wird.

Stellvertretend sollen hier einige Projekte skizziert werden, in denen PASS und (Verteiltes) PEARL im Bereich der Universität Erlangen-Nürnberg auf Z80-Mikrorechnern zum Einsatz kommen.

4.1. Programme zur ausfallsicheren Datenerfassung /HHT86/

Die Erfassung unwiederbringlicher Daten erfordert ausfallsichere Rechnerkonfigurationen.

In einer Zeit der billigen Mikrorechner bietet es sich an, die zur Erfassung vorgesehenen Rechner mehrfach (redundant) auszulegen und als Verteiltes System zu betrachten. Das wird am Beispiel der Erfassung von Telefongesprächsdaten an der Universität Erlangen durch ein System aus redundanten Mikrorechnern gezeigt.

Will man die Erfassung wichtiger Daten ausfallsicher machen, heißt das, die für die Datenaufnahme und Zwischenspeicherung vorgesehenen Komponenten durch redundante Mikrorechner vornehmen zu lassen. Eine solche Konfiguration läßt sich zusammen mit dem Verarbeitungsrechner als Verteiltes System betrachten. Die Programmierung Verteilter Systeme läßt sich bequem durch Verteiltes PEARL, unterstützt durch die Spezifikationstechnik PASS, vornehmen.

Das skizzierte Verfahren fand Anwendung bei der ausfallsicheren Erfassung von Telefongesprächsdaten. Zum Einsatz kamen vier gekoppelte Z80-Mikrorechner, von denen zwei unabhängig die Daten (hot-stand-by) aufnehmen und zwischenspeichern. Die beiden anderen dienen zur Bedienerführung bzw. zur Abwicklung der Übertragungsprozeduren zum Verarbeitungsrechner. Es wird untersucht, wo die wesentlichen Eigenschaften und Vorteile der verwendeten Technik für die betrachtete Klasse von Anwendungen liegen. Besondere Erwähnung finden Sprachkonstrukte, die zur einfachen gegenseitigen Überwachung und zum Synchronisieren von physikalisch verteilten Prozessen einsetzbar sind.

4.2. Überwachung von physikalischen Experimenten /BBHHM86/

Z80-Mikrorechner werden in der physikalischen Experimentiertechnik unter verschiedenen Bedingungen eingesetzt. Beispiele sind:

- Überwachung von digitalen Meßwerten in Experimenten am CERN. Aufgabe des Mikrorechners ist die Überwachung von Beschleunigerstrahl, Drahtkammer und Hodoskop sowie Weiterleitung der Daten an den Aufnahmerechner VAX 750. Hinzu kommen einfache Datenaufnahme-Aufgaben, die einen schnellen Überblick über den Stand des Experiments erlauben.

Zum Einsatz kommen dabei 48 "Langzeit"-Zähler und zwei "Kurzzeit"-Zähler, die über 64fach-Multiplexer an verschiedene Meßstellen angeschlossen werden können. Basis aller Zähler sind Z80-CTCs. Die Software ist vollständig in PEARL programmiert und umfaßt zehn Tasks (Prozesse) mit insgesamt ca. 3500 Quellzeilen.

- Überwachung von analogen Meßwerten in Experimenten am SIN (Schweizerisches Institut für Nuklearforschung).
  Aufgabe des Mikrorechners ist die Überwachung von Verstärkern, Detektoren und Nachweiselektronik sowie Weiterleitung der Daten an den Aufnahmerechner PDP 11/34. Außerdem wird der Datenaufnahmerechner bei instabilen Experimentbedingungen vom Experiment getrennt. Zum Einsatz kommt u.a. ein impulsgesteuerter Analog-Digital-Konverter, der wahlweise auf verschiedene (bis zu acht) Meßstellen aufgeschaltet werden kann. Die Software ist vollständig in PEARL programmiert und umfaßt zwei Tasks (Prozesse) mit insgesamt ca. 2000 Quellzeilen.

- Ein Regelsystem für Zeitzweige kernphysikalischer Elektronik.
  Die Zeitbeziehungen kernphysikalischer Ereignisse werden mit konventioneller Koinzidenzelektronik gemessen und die Daten mit einem Mikrorechner erfaßt. Die entstehende Häufigkeitsverteilung der Zeitdifferenzen wird überwacht. Auftretende Schwankungen werden mit einem programmierbaren Delay, das Impulse mittels Kabellängen verzögern kann, automatisch korrigiert. Die Software wurde in PEARL geschrieben und an einem Z80-Mikrorechner auf ECB-Bus-Basis implementiert.

4.3. Steuereinheiten in der Kommunikationstechnik /HHHL84/

Z80-Mikrorechner werden im Kommunikationsnetz des RRZE als billige Remote-Job-Entry-("RJE"-)Station eingesetzt. Zwei Entwicklungen sollen genannt werden:

- Emulation der RJE-Station UT200 der Firma CDC
  Aufgaben des Mikrorechners sind die Abwicklung der Synchron-Prozedur Mode4a gegenüber dem Host-Rechner und der Transfer der Daten von bzw. zu den Geräten Dialogterminal, Drucker und Leseeinrichtungen. Zum Einsatz kommen Z80-SIOs, die synchron bzw. asynchron betrieben werden. Die Software ist vollständig in PEARL programmiert und umfaßt sechs Tasks (Prozesse) mit ca. 1100 Quellzeilen.

- Emulation der RJE-Station T8418
  Aufgaben des Mikrorechners sind die Abwicklung der Synchron-Prozedur MSV2 gegenüber dem Host-Rechner und der Transfer der Daten zu/von den Geräten Drucker und Leseeinrichtung. Zum Einsatz kommen Z80-SIOs, die synchron und asynchron betrieben werden. Die Software ist vollständig in PEARL programmiert und umfaßt fünf Tasks (Prozesse) mit ca. 1100 Quellzeilen.

5. Ergebnis

Da inzwischen eine Vielzahl von Anwendungen vorliegt, läßt sich die entwickelte "Programmiertechnik für Verteilte Systeme von Mikrorechnern" erfreulicherweise anhand von Erfahrungen aus den Anwendungen beurteilen.

Ein Bild über die Programmiertechnik als Ganzes mit allen ihren Werkzeugen ließe sich am besten in dem Projekt zur Erfassung von Telefongesprächsdaten machen. Hier zeigt sich besonders, wie wichtig neben einer komfortablen Programmiersprache eine Spezifikationstechnik ist, die bereits beim Entwurf durch ihre Grundelemente die Synchronisation und Kommunikation in Verteilten Systemen einfach formulieren läßt. Bei der Programmierung erwies sich nicht nur, daß die in /DIJK75/ für Verteilte Systeme vorgeschlagenen nicht-deterministischen Kontrolloperationen tatsächlich in einer Programmiersprache verwirklicht werden können, sondern auch, daß sie von Anwendern beherrscht werden können. Insbesondere die "Verundungsmöglichkeiten" von Botschaftsoperationen lieferten eine einfache Möglichkeit, den Gleichlauf von Prozessen auf verschiedenen Prozessoren zu sichern.

Die als Nebenbedingung ins Auge gefaßte Portabilität der Systemsoftware ließ sich bei der Übertragung des Kerns des Programmiersystems (d.h. Codegenerator, Laufzeitfunktionen, Betriebssystem) auf den Mikroprozessor Motorola 68000 demonstrieren. Hierzu genügten drei Informatik-Studienarbeiten. Der hierzu nötige Aufwand stellt nur einen Bruchteil einer Neuimplementierung dar.

Alle diese rundweg positiven Erfahrungen ermunterten das RRZE und die von ihm betreuten Einrichtungen, künftig an dieser Technik festzuhalten und sie auch in Verbindung mit Mikroprozessoren des Typs Motorola 68000 bei Realzeitaufgaben einzusetzen.

6. Literatur

6.1. Allgemeines

KKST79 R. Kimm, W. Koch, W. Simonsmeier, F. Tontsch: Einführung in Software Engineering; de Gruyter, Berlin, New York, 1979

6.2. Spezifikationstechnik

AFHHK85 C. Andres, A. Fleischmann, P. Holleczek, U. Hillmer, R. Kummer: Eine Methode zur Beschreibung von Kommunikationsprotokollen; Informatik Fachberichte Nr. 95, GI-NTG-Fachtagung "Kommunikation in Verteilten Systemen" 1985; Springer-Verlag, Berlin, 1985

FL84 A. Fleischmann: Ein Konzept zur Darstellung und Realisierung von Verteilten Prozeßautomatisierungssystemen; Dissertation an der Universität Erlangen-Nürnberg, 1984

LUDE81 J. Ludewig: PCSL und ESPRESO - zwei Ansätze zur Formalisierung der Prozeßrechnerspezifikation; Informatik Fachberichte Nr. 39, Fachtagung Prozeßrechner 1981; Springer-Verlag, Berlin, 1981

6.3. Programmiersprache

BHS83 R. Bähre, Ditteger, F. Saenger: Der RDC-Ring, ein fehlertolerantes, dezentral- und ereignisgesteuertes Lichtleiter-Kommunikationsnetz; Informatik-Fachberichte Nr. 60, 1983

DIJK75 E.W. Dijkstra: Guarded Commands, Nondeterminacy and Derivation of Programs; Communication of the ACM, August 1975

ESG77 PEARL Subset for Avionic Applications, Language Description; Electronic System Gesellschaft, München, 1977

ESG80 Avionic PEARL-Subset: Beschreibung der Zwischensprache CIMIC/AV; Electronic System Gesellschaft, München, 1980

FHKK83 A. Fleischmann, P. Holleczek, G. Klebes, R. Kummer: Synchronisation und Kommunikation verteilter Automatisierungsprogramme; Angewandte Informatik 7, 1983

HANS75 P. Brinch-Hansen: The Programming Language Concurrent PASCAL; IEEE Trans. Software Engineering SE-1, 2, 199-206, June 1975

HANS78 P. Brinch-Hansen: Distributed Processes: A Concurrent Programming Concept; Communication of the ACM, November 1978

HOAR78 C.A.R. Hoare: Communicating Sequential Processes; Communication of the ACM, November 1978

HOF83 F. Hofmann: Koordinierung und Verteilte Systeme; PEARL-Tagung 1983; PEARL-Verein, 1983

HO80 P. Holleczek: Erfahrungen mit portabler Systemsoftware bei PEARL-Implementationen für die Prozeßrechner SIEMENS 306 und 310; Berichte des German Chapter of the ACM 4; Tagung "Portable Software" 1980; Teubner, 1980

HO82 P. Holleczek: Ein portables Prozeßprogrammiersystem auf der Basis von PEARL; Bericht KfK-PFT-E8, Kernforschungszentrum Karlsruhe, 1982

IRTF80 Industrial Realtime Fortran, Draft Standard; International Purde Workshop/European Workshop on Industrial Computer Systems, 1980

LIS83 B. Liskov, R. Scheifler: Guardians and Actions: Linguistic Support for Robust, Distributed Programs; ACM TOPLAS 5, 381-404, 3. July 1983

RO78 R. Rössler: PEARL-Betriebssystem für den Z80; Entwicklungsnotiz PDE-E119, Kernforschungszentrum Karlsruhe, 1978

6.4. Testumgebung

AFHT85 C. Andres, A. Fleischmann, P. Holleczek, M. Trautner: Testen von Verteilten Programmen zur Prozeßautomatisierung; Angewandte Informatik 2, 1985

BMW81 P.-J. Brunner, K. Meffert, H. Windauer: PEARL-Testsystem; PEARL-Rundschau, Band 2, Nr. 6, Dezember 1981

HS81 P. Heine, F.v. Stülpnagel: Petsy, ein PEARL-Testsystem zum Austesten von physikalisch verteilten PEARL-Programmen unter Realzeitbedinungen; PEARL-Rundschau, Band 2, Nr. 6, Dezember 1981

LA81 L. Lamport: Time, Clocks and the ordering of events in a destributed system; Communication of the ACM, July 1978

MA81 K. Mangold: Ein quellenbezogenes Testsystem für PEARL auf einem Prozeßrechner; PEARL-Rundschau, Band 2, Nr. 2, Dezember 1981

6.5. Entwicklungswerkzeuge

KRAG86 G. Kragl: Eine Programmierumgebung für Verteiltes PEARL - Codeerzeugung mittels Programmsynthese; Dissertation an der Universität Erlangen-Nürnberg, 1986

6.6. Anwendungen

BBHHM86 R. Baran, R. Besold, A. Hofmann, P. Holleczek, R. Müller: An automatic microcomputer based control system; Nucl. Instr. and Methods, North Holland, to be published

HHHL84 U. Hillmer, G. Höfner, P. Holleczek, P. Lindlein: Z80-Mikrorechner als Remote-Job-Entry-Station; PEARL-Tagung 1984, PEARL-Verein, 1984

HHT86 E. Heilmeier, P. Holleczek, M. Trautner: Verteilte Programme zur ausfallsicheren Datenerfassung mit redundanten Mikrorechnern; Automatisierungstechnische Praxis,