# Testen von verteilten Programmen zur Prozeßautomatisierung

C. Andres, A. Fleischmann, P. Holleczek, M. Trautner*

**Stichworte:** *Testen von Echtzeitprogrammen, verteilte Systeme, PEARL, Prozeßsteuerung*

**Zusammenfassung:** *Die Akzeptanz von höheren Realzeitsprachen in der Proßzeßsteuerung hängt wesentlich von den Testmöglichkeiten ab. Dieser Tatsache wurde bei PEARL bereits vielfach Rechnung getragen. Der Einsatz von Realzeitsprachen in verteilten Systemen und der Test von Programmen auf Front-End-Prozessoren ohne Bedienperipherie bedarf neuer Ansätze. Vorgestellt wird ein PEARL-Testsystem, das den Test physikalisch verteilter Programme erlaubt. Untersucht werden Einflüsse auf das Verhalten von Echtzeitprogrammen, die durch den Einsatz eines solchen Testsystems entstehen. Außerdem werden Methoden zur Programm-Instrumentierung diskutiert, um die Veränderung des Laufzeitverhaltens durch das Testsystem zu minimieren.*

**Testing of distributed programs for process control**

**Key-words:** *realtime software testing, distributed systems, PEARL, process control*

**Abstract:** *The acceptance of higher level real time languages for process control depends largely on the possibilities for testing. This has been realized with PEARL in many cases. The application of real time languages in distributed systems and program testing on front end processors without console make new approaches necessary.*

*A PEARL test system for testing of physically distributed programs is introduced. The influence of such a test system on the performance of real time programs is studied. In addition, methods of program instrumentation are discussed that can minimize the change of performance caused by the test system.*

* Regionales Rechenzentrum der Universität Erlangen-Nürnberg (RRZE) D-8520 Erlangen, Martensstr. 1, W. Mühlbauer, COMSOFT GmbH, D-8520 Erlangen

## 1 Einleitung

Die wachsenden und immer komplexeren Aufgaben, die Rechner zur Überwachung und Regelung von automatisierten technischen Prozessen übernehmen müssen, stellen an die Zuverlässigkeit von Programmsystemen zur Prozeßsteuerung erhöhte Anforderungen.

Da größere Programme fast grundsätzlich Fehler enthalten, die im Rahmen der Prozeßsteuerung eventuell fatale Auswirkungen haben können, erhält die Aktivität bei der Entwicklung von Software erhebliche Bedeutung.

Die Verwendung von höheren Realzeitsprachen, wie z.B. PEARL, stellt bereits einen ersten Schritt zur Verbesserung der Software-Zuverlässigkeit dar. Jedoch auch hier ist dem Testen eine große Bedeutung beizumessen. Die Akzeptanz einer höheren Realzeitsprache hängt darum wesentlich von dem Vorhandensein einer sprachorientierten Testhilfe ab.

In der Literatur werden bereits einige Testsysteme für PEARL beschrieben. *Brunner* et al. [1] sowie *Mangold* [9] beschreiben Testsysteme für den Einsatz von PEARL auf Monorechnersystemen. *Heine* und *Stülpnagel* [6] stellen eine Testhilfe für verteilte Systeme vor.

In diesem Artikel soll nun ein Testsystem für verteilte Systeme von Mikrorechnern eingeführt werden, das er erlaubt, nicht nur die konventionellen Konstrukte zur Prozeßsynchronisation und – Kommunikation, sondern auch solche Konstrukte für physikalisch verteilte Programme (Botschaftsoperationen) zu überwachen und zu manipulieren. Zunächst werden jedoch die Probleme beim Testen von verschiedenen Programmtypen (sequentielle Programme, quasiparallele Programme, echt parallele Programme) diskutiert. Danach wird ein interaktives Testsystem zum Testen von verteilten Automatisierungsprogrammen vorgestellt.

## 2 Probleme beim Testen verschiedener Programmtypen

Testhilfen, wie Dialogtestsysteme, beeinflussen das Verhalten von Programmen zum einen explizit durch die Wirkung der Testfunktionen, zum anderen allein dadurch, daß sie den Prozessor zur Ausführung der Testfunktionen in Anspruch nehmen.



0013-5704/85/2 0069–08 $ 03.00/0 

Die Auswirkungen sind unterschiedlich, je nach Typ des Testobjekts. Es werden drei Typen von Testobjekten unterschieden:

- sequentielle Programme (ein Rechenprozeß),
- quasi-parallele Programme (mehrere Rechenprozesse auf einem Prozessor),
- verteilte Programme (mehrere Rechenprozesse auf verschiedenen Prozessoren).

Es wird davon ausgegangen, daß ein Dialogtestsystem als Minimalforderung folgende Testfunktionen zur Verfügung stellt:

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

Diese Testfunktionen werden zum Testen aller Programmtypen benötigt. Für quasi-parallele und verteilte Programme müssen zusätzlich noch Operationen zur Beeinflussung und Kontrolle des Prozeßverhaltens zur Verfügung stehen:

- Kontrollieren von Prozeßzuständen und Verändern von Zuständen, z.B. durch Starten von Prozessen,
- Eingreifen in das Synchronisationsverhalten, z.B. durch Verändern von Semaphor-Werten.

Im folgenden sollen nun die Programmtypen definiert und die entsprechenden Auswirkungen bei Eingriffen mit Hilfe des Testsystems diskutiert werden.

## 2.1 Programmtypen

Ein Programm kann als Deklaration eines Prozesses aufgefaßt werden, der die Bearbeitung eines Problems gemäß einem Algorithmus übernimmt. Ein Prozeß zerfällt in einzelne Operationen, die zu bestimmten Zeiten ausgeführt werden.

*Sequentielle Programme* sind dadurch ausgezeichnet, daß die Operationen nur in zeitlicher Reihenfolge nacheinander ausgeführt werden [2].

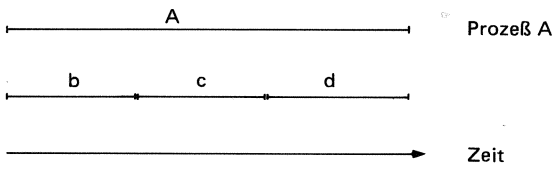


**Bild 1** Ablauf sequentieller Programme

Beispiel:

Der Prozeß A besteht aus den Operationen b, c, d, die nacheinander ausgeführt werden.

*Quasi-parallele Programme* enthalten die Deklaration mehrerer Rechenprozesse, die um das Betriebsmittel Prozessor, das nur einmal vorhanden ist, konkurrieren.

Operationen eines Prozesses werden bei Monoprozessorsystemen überlappt zu Operationen anderer Prozesse ausgeführt. Beschränkt wird diese Fähigkeit aber dadurch, daß nur ein ausführender Prozessor vorhanden ist; auszuführende Operationen werden auf diesem Prozessor sequentiell nach bestimmten Strategien bearbeitet (z.B. prioritätengesteuert).

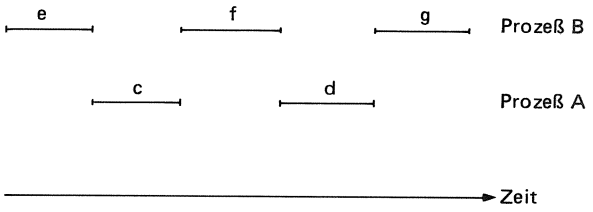


**Bild 2** Ablauf quasi-paralleler Prozesse

Beispiel:

Der Prozeß A, bestehend aus den Operationen c und d, wird quasi-parallel zum Prozeß B mit den Operationen e, f und g abgearbeitet.

Nach außen hin scheinen diese Prozesse jedoch parallel abzulaufen, da über die relativen Bearbeitungszeiten für einzelne Operationen und somit über die Prozeßumschaltzeitpunkte keine Aussagen gemacht werden können.

Über Synchronisationsmechanismen, wie z.B. Semaphor-Operationen, entstehen Querbezüge zwischen den einzelnen Prozessen, so daß durch Sprachkonstrukte im Programm das Ablaufverhalten gesteuert werden kann.

*Verteilte Programme* bestehen wie quasi-parallele aus Rechenprozessen, die parallel zueinander ablauffähig sind. Hier können die Prozesse jedoch echt zeitlich überlappend arbeiten, da mehrere Prozessoren zur Ausführung vorhanden sind.

Je nach Art der verwendeten Kommunikationsmechanismen unterscheidet man in:

- verteilte Programme in lose gekoppelten Systemen (Kommunikation durch Botschaftsaustausch über Leitungen)
- verteilte Programme mit gemeinsamen Speicher (Kommunikation durch gemeinsame Variable)

Beispiel:

Die Prozesse A, B und C sind auf verschiedenen Prozessoren installiert; sie können gleichzeitig ablaufen.

Zwischen den Prozessoren können durch Synchronisationsoperationen Bedingungen für das Ablaufverhalten relativ zueinander geschaffen werden.

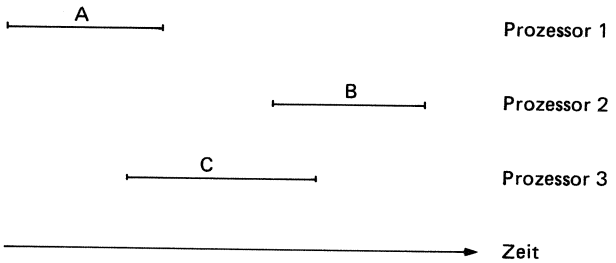


**Bild 3** Ablauf paralleler Prozesse

Auf einem einzelnen Prozessor der Mehr-Rechner-Konfiguration können wiederum parallel ablauffähige Prozesse installiert sein; diese Prozesse sind jedoch nur quasi-parallel.

## 2.2 Einflüsse der Testfunktionen auf das Programmverhalten

Die Funktion *Variable ändern/kontrollieren* bewirkt bei allen Programmtypen eine Veränderung des Zeitverhaltens des zu testenden Prozesses dadurch, daß dessen Operationen durch Operationen des Testsystems unterbrochen werden. Das bedeutet, daß sich Operationen des Prozesses auf der Zeitachse verschieben. Auswirkungen hat dies bei der Steuerung von technischen Prozessen, wo Zeitbedingungen eingehalten werden müssen.

Durch den verzögerten Ablauf des steuernden Programms können zeitbedingte, scheinbar nicht deterministische Fehler auftreten, die durch den Einsatz des Testsystems verursacht werden. Durch die verschiedenen Ausführungszeiten der Testfunktionen entstehen jeweils neue Zeitabhängigkeiten, die zur Verfälschung des Ein-/Ausgabe-Verhaltens des steuernden Programms gegenüber dem technischen Prozeß führen.

Weist man Variablen, die in bedingten Verzweigungen abgefragt werden, per Testsystem Werte zu, ändert sich u.U. der Kontrollfluß im Testobjekt. Dies kann sich bei quasi-parallelen und echt-parallelen Programmsystemen auf alle Prozesse auswirken, die durch Synchronisationsoperationen mit dem zu testenden Prozeß verbunden sind.

Die Funktion *Zeilentrace* bewirkt beim zu testenden Prozeß eine Verlängerung der Programmlaufzeit, da bei jedem ausgeführtem Statement eine Meldung an das Testsystem erfolgen muß. Dies führt, wie oben bereits beschrieben, zu Veränderungen des Zeitverhaltens.

Bei *Haltepunkten* sind die Auswirkungen abhängig von der Art der Haltepunktsfunktion:

- Taskhaltepunkt
  Bei sequentiellen Programmen ist nur diese Form der Haltepunktsfunktion möglich. Sie bewirkt eine erhebliche Veränderung des Zeitverhaltens.

  Bei quasi-parallelen und echt-parallelen Programmen wird durch diese Funktion nur ein einzelner Prozeß (eine Task) angehalten. Die anderen Prozesse bleiben unbeeinflußt.

  Synchronisationsoperationen bzw. Zeiteinplanungen stellen jedoch Querbezüge zwischen Testobjekt und dazu parallelen nicht getesteten Prozessen her, so daß das Anhalten eines Prozesses unter Umständen zu einer Verklemmung des Gesamtsystems führen kann.
- Gesamtstop (auf einem Prozessor)
  Erreicht der zu testende Prozeß einen Haltepunkt, werden alle Prozesse des Programmsystems angehalten.
  Dadurch vermeidet man eine einseitige Verschiebung der Operationen eines Prozesses gegenüber Operationen anderer, dazu paralleler Prozesse. Es tritt nur eine Veränderung des Zeitverhaltens gegenüber externen Prozessen (technischen Prozessen bzw. Prozessen auf anderen Rechnern bei verteilten Programmen) auf.

  Bei quasi-parallelen Programmen ist diese Funktion sehr nützlich, da nach einem Gesamtstop eine konsistente Information über den Zustand des gesamten Programmsystems abgefragt werden kann.

  Bei verteilten Programmen wird wiederum das Synchronisationsverhalten gestört.
- Gesamtstop (verteilt)
  Um ein verteiltes System kontrolliert über Testfunktionen abprüfen zu können, ist ein Haltepunkt notwendig, der auf alle Prozesse des verteilten Systems einwirkt.

  Hier besteht jedoch das Problem der Realisierung einer solchen Funktion bei lose gekoppelten Systemen. Ein absolut synchrones Anhalten aller Prozesse ist durch ein Testsystem, dessen Funktionen softwaremäßig realisiert sind, nicht möglich. Hier spielen die Bearbeitungszeiten für die Ausführung einer solchen Funktion (Erkennen eines Haltepunktes, Rückmeldung an die Testsystemablaufkontrolle, Anhalten der Prozesse auf den verschiedenen Prozessoren) sowie die Übertragungszeit für den Haltepunktaufruf an alle Prozessoren eine wesentliche Rolle. Da diese Zeiten im allgemeinen sehr groß gegenüber der Ausführungszeit einer Programmanweisung sind, ist eine Haltepunktsfunktion wertlos, da der zeitliche Zusammenhang zwischen der Ausführung der einzelnen Aufrufe im Gesamtsystem nicht gewährleistet ist.

  Realisierbar wäre ein systemglobaler Haltepunkt bei verteilten Programmen mit gemeinsamen Speicher. Durch Benutzung einer gemeinsamen Anzeigezelle, die vor jeder Ausführung einer Programmanweisung kontrolliert wird, könnten alle Prozesse gleichzeitig angehalten werden, wenn ein Prozeß den Haltepunkt erreicht hat und die Anzeigezelle gesetzt ist. Der Aufwand zur Synchronisierung und die Abfrage der Zelle bei jedem Programmstatement verändert jedoch das Laufzeitverhalten so stark, daß diese Lösung nicht sinnvoll erscheint.

*Ändert man Prozeßzustände* bzw. führt man Synchronisationsoperationen mit Hilfe von Testfunktionen aus, hat dies erhebliche Auswirkungen auf Prozesse, die untereinander verbunden sind. Wird z.B. ein Prozeß beendet, bevor er eine Semaphoroperation ausführt, durch die ein anderer Prozeß wieder lauffähig wird, so bleibt dieser Prozeß blockiert. Der Einfluß der Testfunktionen läßt sich jedoch durch Analyse des Programmtextes ermitteln, wenn man die entsprechenden Querbezüge untersucht.

In Bild 4 sind die Einflüsse der Testfunktionen auf das Laufzeitverhalten tabellarisch zusammengefaßt.

## 3 Verschiedene Instrumentierungstechniken und deren Einfluß auf die Progammlaufzeit

Zur Ausführung von Haltepunkt- und Traceaufrufen muß zwischen Testsystem und dem zu testenden Programm (Testling) eine Verbindung geschaffen werden, die zur Laufzeit des zu testenden Programms Eingriffe in das Ablaufverhalten des Testobjekts erlaubt. Da diese Verbindung interaktiv zu beliebigen Zeiten an beliebigen Stellen im Testling hergestellt werden soll, muß das Programm „instrumentiert" werden, d.h. es wird Code eingebracht, der die Ablaufkontrolle an das Testsystem übergibt. Dazu gibt es drei Methoden der Codeinstrumen-

| Test-funktion / Programm-typen | Standardtesthilfen | | | Testhilfen für verteilte Programme | |
|---|---|---|---|---|---|
| | Haltepunkte | Zeilen-protokoll | Variable ändern/kontr. | Prozeßstatus ändern/kontr. | Synchronisation ändern/kontroll. |
| Sequentielle Programme | Verändern des Zeitverhaltens | | Verändern des Zeit- u. Ablauf-verhaltens | – – – | |
| Quasi-parallele Programme | Task-Haltepunkt: Verändern der Synchronisation | Verändern der Synch. durch Strecken einer Task | Verän. des Kontrollflusses →Synchronisation | Verändern von Prozeßzuständen und Synchronisationsverhalten unter Benutzerverantwortung | |
| Parallele Programme | Gesamtstop: (verteilt) Realisierung?<br>Gesamtstop: (Prozessor) Synchr. v. Tasks auf verschiedenen Prozessoren<br>Task-Haltepunkt: Verändern der Synchronisation | Verändern des Zeitverhaltens und evtl. verändern der Synchronisation durch Strecken einer Task | Verändern des Kontrollflusses ——> verändern des Synchronisationsverhaltens (lokal und verteilt) | wenn verteilter Gesamtstop fehlt, konsistente Information über den Zustand des Gesamtsystems nur schwer zu gewinnen | |

**Bild 4** Einflüsse von Testfunktionen auf das Programmverhalten

tierung, die hier kurz vorgestellt und kritisch betrachtet werden sollen.

## 3.1 Statische Instrumentierung

In das Testobjekt wird durch den Compiler bei der Übersetzung jeder Anweisungszeile im Quellprogramm zusätzlich ein Aufruf an das Testsystem eincompiliert, der als Parameter die Nummer der Quellzeile, in der die Anweisung steht, besitzt.

Bei Ausführung des Testobjekts verzweigt der Kontrollfluß nach jeder Anweisung in das Testsystem. Dort wird geprüft, ob für die aktuelle Quellzeile ein Auftrag vorliegt und falls vorhanden, der Auftrag (Programm anhalten oder Zeilennummer protokollieren) ausgeführt.

Vorteile des Verfahrens:

Das Verfahren ist rechnerunabhängig, da die Aufrufe (Unterprogrammaufrufe) vom Compiler abgesetzt werden können.

Nachteile des Verfahrens:

Für Testzwecke sind eigene Übersetzungsläufe notwendig, um das zu testende Programm zu instrumentieren, bzw. um Code ohne Instrumentierung zu erhalten.

Die Laufzeit des Programms wird an Stellen, an denen keine Testaufträge vorliegen, unnötig verlängert, da die Verzweigung ins Testsystem nach jeder Quellzeile erfolgt.

Der Code des Testobjekts wird stark aufgebläht, da pro Anweisung Speicherplatz für den Testsystemaufruf notwendig ist.

## 3.2 Dynamische Instrumentierung

Hier wird die Verbindung zum Testsystem nicht statisch in das zu testende Programm eincompiliert, sondern erst nach Übersetzung im lauffähigen Programm, durch Testkommando spezifiziert, dynamisch geschaffen.

An den Stellen im Programm, wo eine Testfunktion aufzurufen ist, wird der Code vom Testsystem gerettet und durch einen Befehl ersetzt, der einen Zustandswechsel des Testobjekts veranlaßt, so daß das Testsystem aktiv werden kann (Codeaustausch).

Läuft der Testling auf diesen Befehl, erfolgt die Verzweigung ins Testsystem; dort wird die gewünschte Funktion ausgeführt.

Soll das Programm fortgesetzt werden, wird der gerettete Befehl ausgeführt und die Kontrolle wieder an das Testobjekt zurückgegeben.

Vorteile des Verfahrens:

Der Code des Testobjekts wird nur an den Testpunkten verändert, so daß eine Laufzeitbeeinflussung nur durch Ausführung der Testfunktionen erfolgt.

Der Speicherplatzbedarf des Testobjekts vergrößert sich nicht; es muß nur zusätzlich der ersetzte Code gespeichert werden.

Um ein lauffähiges Programm herzustellen, ist kein spezieller Übersetzungslauf notwendig.

Nachteile des Verfahrens:

Das Verfahren ist rechnerabhängig, da der Codeaustausch nur mit Kenntnis des Befehlsformats im verwendeten Zielrechner durchgeführt werden kann.

### 3.3 Halbdynamische Instrumentierung

Dieses Verfahren ist eine Mischform der beiden obigen Verfahren. Bei der Übersetzung des Quellprogramms wird vor dem einer Quellzeile entsprechenden Objektcode Platz für einen Aufruf an das Testsystem freigehalten. Hier wird dann durch Definition eines Testpunktes durch den Benutzer dynamisch der konkrete Aufruf eingesetzt.

Vorteile des Verfahrens:

Gegenüber der statischen Instrumentierung ist die Laufzeitbeeinflussung geringer, da bei jeder Quellzeile nur die Leerbefehle zum Platzfreihalten ausgeführt werden müssen und die Nachfrage beim Testsystem, ob ein Auftrag vorliegt, entfällt.

Die Implementierung ist rechnerunabhängig.

Nachteile des Verfahrens:

Gegenüber der dynamischen Instrumentierung ist eine größere Laufzeitbeeinflussung festzustellen, da durch die eingesetzten Leeranweisungen eine Verzögerung entsteht.

Auch hier muß für Testzwecke neu übersetzt werden. Der Code des Testobjekts wird expandiert.

### 3.4 Diskussion der verschiedenen Instrumentierungsarten

Die Auswahl der geeigneten Instrumentierungsmethode wird bestimmt durch den Verwendungszweck des Testsystems.

Im vorliegenden Fall bestand die Aufgabe, ein Testsystem zum Testen von Echtzeitprogrammen im Rahmen einer Cross-Compilerumgebung für Mikro-Rechner zu entwickeln. Daraus ergaben sich zwei wesentliche Kriterien

- Minimale Beeinflussung des Laufzeitverhaltens des Testlings durch das Testsystem,
- Möglichst geringer Speicherplatzbedarf des Testsystems.

Ausgewählt wurde die dynamische Instrumentierung. Sie verändert das Echtzeitverhalten des Testobjekts zwischen Testpunkten nicht, da nur der ursprüngliche Programmcode durchlaufen wird. Die anderen Methoden verlängern generell die Laufzeit des Testobjekts, auch wenn keine Testfunktionen ausgeführt werden. Die Auswirkungen auf das Programmverhalten wurden bereits in Abschnitt 2 dargestellt.

Zum Testen ist kein eigener Übersetzungslauf notwendig. Dies spart bei Cross-Übersetzungssystemen viel Zeit und trägt dadurch erheblich zur Benutzerfreundlichkeit bei.

Der Speicherplatzbedarf der verschiedenen Methoden ist ungefähr gleich. Bei der dynamischen Instrumentierung werden die zur Ausführung der Testfunktionen benötigten Informationen in eigenen Listen abgelegt. Diese Listen können auf Externspeicher ausgelagert und bei Bedarf in Form von Teillisten wieder zurückgeholt werden. Dies ist bei Mikroprozessorsystemen mit beschränktem Arbeitsspeicherausbau besonders wertvoll.

Bei statischer Instrumentierung würde der Code des zu testenden Programms expandiert. Dies macht eventuell ein Overlay-Binden für Testzwecke erforderlich.

## 4 Struktur eines Testsystems für quasi-parallele und verteilte Programme

Die Funktionsweise eines Testsystems kann so spezifiziert werden, daß es sowohl auf Ein- wie Mehr-Rechner-Konfigurationen läuft. Das Testsystem wird in Form von zwei miteinander kommunizierenden Prozessen, einem Dialog-Prozeß und einem Test-Prozeß, realisiert. Bei Monoprozessor-Konfiguration bilden beide Prozesse zusammen das Testsystem für diesen Rechner.

In Mehrrechner-Konfiguration läuft der Dialogprozeß auf dem Prozessor des verteilten Systems, der mit einem Bedienterminal ausgerüstet und mit den anderen Prozessoren verbunden ist.

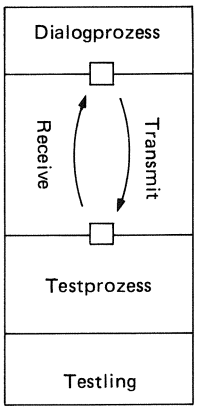


**Bild 5** Testsystem in Mono-Prozessor-Konfiguration

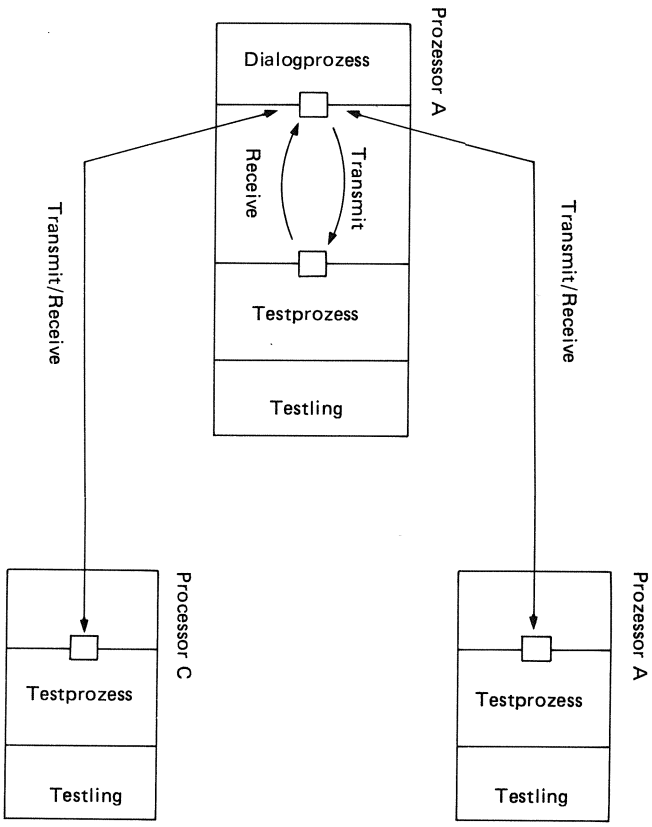


**Bild 6** Testsystem in Mehr-Prozessor-Konfiguration

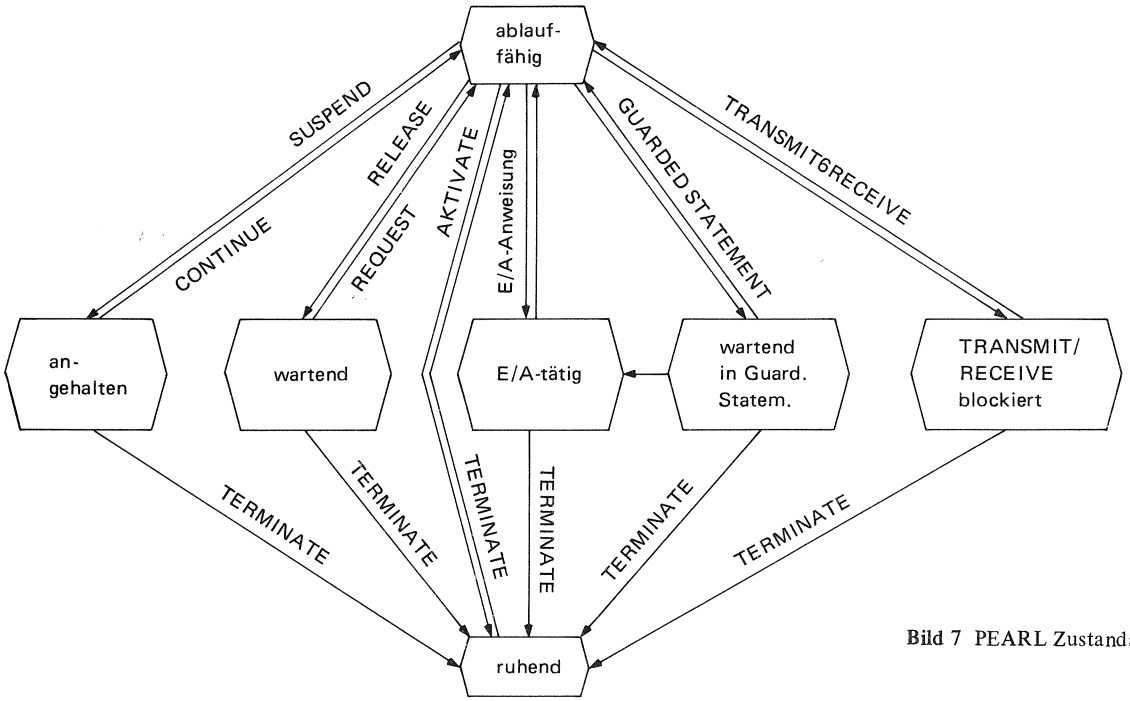


**Bild 7** PEARL Zustandsmodell

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 enkomprimiert die Ergebnisse (Protokollaufbereitung) und protokolliert sie auf dem Bedienterminal.

## 5 Ein sprachorientiertes PEARL-Testsystem für verteilte Systeme von Mikroprozessoren

Für die Programmiersprache PEARL, erweitert um Sprachkonstrukte für verteilte Systeme [4], wurde nach dem obigen Konzept ein Testsystem entwickelt. Im Folgenden sollen die für das Testsystem wesentlichen Aspekte von PEARL kurz vorgestellt werden.

### 5.1 Die Programmiersprache PEARL

PEARL ist eine höhere Programmiersprache, deren Sprachumfang Echtzeitelemente enthält. Dadurch eignet sich PEARL zur Formulierung von Aufgaben der Prozeßautomatisierung und -steuerung. Für Automatisierungsprogramme sind Prozesse ein wesentliches Strukturierungselement, in PEARL verkörpert durch sogenannte Tasks. Tasks sind die aktiven Elemente eines PEARL-Programms. Eine Task ist die Deklaration eines selbständig, parallel zu anderen Prozessen ablauffähigen Rechenprozesses. Tasks synchronisieren sich über Semaphore und können über globale Variable, Prozeduren usw. verfügen.

Zur Kommunikation in verteilten Systemen wurden Botschaftsoperationen eingeführt (TRANSMIT/RECEIVE). Mit diesen Operationen können sich verteilte Prozesse synchronisieren und Nachrichten austauschen.

Die Tasks können sich in verschiedenen Zuständen befinden. PEARL bietet Sprachkonstrukte, mit denen ein Zustandswechsel von Tasks veranlaßt werden kann (siehe Bild 7).

Eine Beschreibung der Erweiterungen von PEARL durch Botschaftsmechanismen und nichtdeterministische Kontrolloperationen findet sich in [4].

### 5.2 Leistungsumfang des Testsystems

Neben den herkömmlichen Auskünften über Prozesse und Variable liefert das Testsystem Informationen über Betriebssystem-Warteschlangen sowie über den Inhalt der Botschaftspuffer.

An Manipulationen sind die üblichen Operationen auf PEARL-Datentypen sowie das Erzeugen von Botschaften erlaubt.

#### 5.2.1 Auskunftsfunktionen

Das Testsystem liefert folgende Informationen:

1) Auskunft über den Status aller Prozesse.
   Ausgabe: – Taskpriorität
   – Taskzustand
2) Auskunft über Variable und Variablenfelder.
   Ausgabe: – Speicherinhalt
3) Auskunft über Semaphore.
   Ausgabe: – aktueller Wert
   – Semaphore-Warteschlange
   (ausgegeben werden die an dem Semaphor blockierten Tasks)

4) Auskunft über den Inhalt der Botschaftspuffer.
   Ausgabe: – Inhalt des Warte- und des Blockiertbereichs der Task
   – Anzahl der Botschaften im Wartebereich der Task

5) Auskunft über Geräte.
   Ausgabe: – belegende Task
   – Geräte-Warteschlange
   (ausgegeben werden die auf die Zuteilung wartenden Tasks)

6) Auskunft über Interrupts.
   Ausgabe: – Interrupt-Warteschlange
   (ausgegeben werden diejenigen Tasks, die auf das Eintreffen des Interrupts warten)

7) Auskunft über die Prozessor-Warteschlangen.
   Ausgabe: – Prozessor-Warteschlange
   (ausgegeben werden die Tasks, die auf die Zuteilung des Prozessors warten)

Dabei ist zu berücksichtigen, daß die Auskünfte überholt sein können, wenn die Prozesse nicht angehalten werden.

#### 5.2.2 Manipulierende Funktionen

Das Testsystem läßt folgende Manipulationen zu:

1) Zustandsänderungen von einzelnen Prozessen:
   – ACTIVATE
   – CONTINUE
   – PREVENT
   – SUSPEND
   – TERMINATE
2) Anhalten und Fortsetzen des Gesamtsystems
4) Ändern von Semaphore-Werten
5) Ändern der Werte von Variablen und Arrays
6) Ein und Ausschalten von Trace:
   – Setzen und Löschen von Tracebereichen
7) Trigger-Interrupt

Dabei ist bei manchen Manipulationen (z.B. Änderung von Semaphorwerten) zu beachten, daß sie zwar gefährliche Seiteneffekte haben können, aber durch Überwinden von Stillständen wertvolle Testzeit sparen helfen.

### 5.3 Implementierung

Für ein PEARL Programmsystem (unter Verwendung von Avionik-PEARL [3] auf Z80 Mikrorechnern wurde das oben beschriebene Testsystem in zwei Versionen entwickelt:

- Testsystem für Monorechnerkonfiguration (quasi-parallele Programme),
- Testsystem für verteilte Systeme (echt-parallele Programme).

Der Dialogteil des Testsystems ist in PEARL, die Testfunktionen sind in SPASS geschrieben. SPASS ist eine abstrakte Assemblersprache, in der auch das benötigte PEARL-Betriebssystem (PBS) realisiert ist.

Insgesamt ergibt sich folgende Speicherbelastung.

| | | |
|---|---|---|
| Eingabe | komfortabel | 15 KBytes |
| Kommandointerpreter | | |
| | kurz | 3 KBytes |
| Ausgabe | | |
| | komfortabel | 7 KBytes |
| Protokollaufbereitung | | |
| | kurz | 3 KBytes |

Testfunktionen:

| | |
|---|---|
| TS Kern (Mono) | 5 KBytes |
| TS Kern (verteilt) | 6,5 KBytes |

Die komfortablen Versionen werden in der Ausbildung eingesetzt, da hier eine umfangreiche Kontrolle der Testkommandos (Kommandointerpreter) bzw. eine besonders ausführliche Darstellung der Testergebnisse vorgesehen ist. Im normalen Anwendungsbetrieb kommen die Kurzausführungen zum Einsatz.

Vom Testsystem werden zur Laufzeit folgende Listen benötigt, um Quellinformation (Quellzeilen-Nummern, Variablennamen, Tasknamen usw.) in Objektadressen umzuwandeln:

- Modulliste
  Zur Realisierung von Haltepunkten und Zeilentrace muß es möglich sein, bei gegebenen Modulnamen und Zeilennummer die zugehörige Adresse bzw. umgekehrt bei gegebener Adresse die Zeilennummer und den Modulnamen zu ermitteln.

  Dazu ist in jedem Modul eine Quellzeilenliste notwendig, die zu jeder Quellzeile die zugehörige Adresse angibt. Die Modulliste enthält die Zuordnung von Modulname zu Quellzeilenliste.

- Taskliste
  In der Taskliste sind alle im Benutzerprogramm vereinbarten Tasks aufgelistet. Neben dem Tasknamen und der dazugehörigen Task-Kontrollblockadresse ist eine Leitungsnummer aufgeführt. Sie wird beim Testen verteilter Programme benötigt, um festzustellen welchem Prozessor die einzelnen Tasks zugeordnet sind.

Ebenso werden Semaphor-, Geräte- und Variablenliste benötigt, die die Zuordnung von Namen zu Kontrollblockadressen bzw. Speicherplatzadressen erlauben.

Erzeugt werden diese Listen vom PEARL-Modulbinder. Seine Aufgabe ist es , die Querbezüge zwischen den einzelnen PEARL-Moduln herzustellen. Als Eingabe erhält der Modulbinder übersetzte, in Assemblersprache vorliegende PEARL-Moduln. Im Assemblertext ist gekennzeichnet, welche Befehlsfolgen einem PEARL-Statement entsprechen. Aus diesen Informationen erzeugt der Modulbinder die Quellzeilenliste und die Modulliste.

Für alle vom Betriebssystem verwalteten Objekte werden Kontrollblöcke erstellt. Die Adressen der Kontrollblöcke stellt der Modulbinder in den Task-, Semaphor- bzw. Gerätelisten zusammen. Die Erzeugung der Listen ist optionell; sie wird durch einen Parameter beim Starten des Binders gesteuert.

Dadurch ist es möglich, testbare PEARL-Programme nur durch einen Bindelauf ohne Neuübersetzung zu erzeugen.

Zur Realisierung von Testfunktionen, die vom Betriebssystem verwaltete Objekte (z.B. Semaphore, Tasks) betreffen, wird die Auftragsschnittstelle des PEARL-Betriebssystems verwendet. Aus seiner Sicht wird das Testsystem wie ein gewöhnlicher Benutzerprozeß behandelt, der mit höchster Priorität läuft.

Für die Auskunftsfunktionen (z.B. Anzeigen eines Semaphorwertes mit Ausgabe der Warteschlange) wurden die benötigten Funktionen in das Betriebssystem integriert.

## 6 Zusammenfassung

Es wurde ein Testsystem für verteilte Systeme von Mikrorechnern (lose gekoppelte Systeme) entwickelt. Das Testsystem ermöglicht ein sprachorientiertes Testen von PEARL-Programmen zur Prozeßautomatisierung.

Die angebotenen Funktionen haben sich im Einsatz als sehr hilfreich erwiesen und bilden die Basis für eine Weiterentwicklung des Testsystems. Hinzugefügt werden sollen noch Funktionen zur Messung von Programmlaufzeiten, um Reaktionszeiten von Programmen zur Prozeßsteuerung erfassen zu können.

Der modulare Aufbau des Testsystems ermöglicht es, nachträglich Funktionen hinzuzufügen, die sich im praktischen Einsatz als notwendig erweisen, bzw. überflüssige oder „gefährliche" Funktionen wegzulassen.

### Literatur


[1] *P.-J. Brunner, K. Meffert, H. Windauer:* PEARL-Testsystem. In: PEARL-Rundschau, Band 2, Nr. 6, Dez. 1981

[2] *P. Brinch Hansen:* Operating Systems Principles. Prentice Hall, Englewood (1973)

[3] PEARL-Subset for Avionic Applications. Language Description. ESG-Elektronik-System-GmbH, München (1977)

[4] *A. Fleischmann, P. Holleczek, G. Klebes, R. Kummer:* Synchronisation und Kommunikation verteilter Automatisierungsprogramme. In: Angewandte Informatik, Heft 7/83

[5] *L. Gmeiner, G. Hommel* (Hrsg.): Testen und Verifizieren von Prozeßrechner-Software. KfK-PDV 179, Dez. 1979, KfK, Karlsruhe

[6] *P. Heine, F. v. Stülpnagel:* Petsy: Ein PEARL-Testsystem zum Austesten von physikalisch verteilten PEARL-Programmen unter Realzeitbedingungen. In: PEARL-Rundschau, Band 2, Nr. 6, Dez. 1981

[7] INFOTECH: Software Testing INFOTECH State of the Art Report. Infotech London 1978

[8] *Lamport. L.:* Time, clocks and the ordering of events in a distributed system. In: CACM, Vol. 21, No. 7, Juli 1978

[9] *K. Mangold:* Ein quellenbezogenes Testsystem für PEARL auf einem Prozeßrechner. In: PEARL-Rundschau Band 2, Nr. 6, Dez. 1981

[10] *Ovenhausen* et al.: PROMOTE, Prozeßorientiertes Modul- und Gesamttestsystem.
PDV 67, Kernforschungszentrum Karlsruhe (1976)