Ein Mechanismus zur Kommunikation für verteilte Systeme in PEARL*)

A. Fleischmann, P. Holleczek, G. Klebes, R. Kummer

0. Einleitung

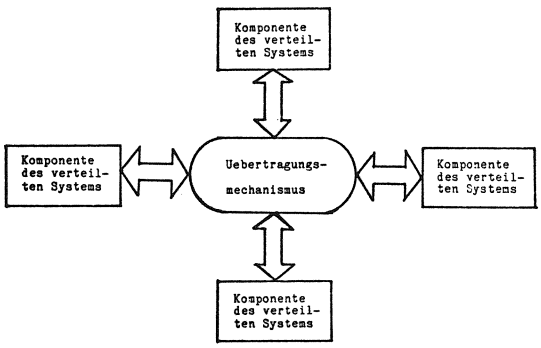
Die Programmierung von verteilten Systemen stellt besondere Anforderungen an die Programmiersprache. Hier soll ein Konzept vorgestellt werden, das sich mit der Kommunikation lose gekoppelter verteilter Systeme beschäftigt. Gemeinsame Objekte zur Kommunikation zwischen Prozessen, die sich auf verschiedenen Prozessoren befinden, werden nicht betrachtet, ebensowenig auch Aspekte, wie Sicherheit, Zuverlässigkeit, Wiederaufsetzverfahren etc.. Transportmechanismen, wie sie z.B. durch Netzwerke geliefert werden, werden vorausgesetzt.
Das hier vorgestellte Konzept basiert auf Botschaftsoperationen und nichtdeterministischen Kontrollstrukturen. Die genannten Operationen sind funktionale und formale Erweiterungen der Basissprache PEARL. Sie wurden in den Compiler und das Betriebssystem integriert.
Durch die Festlegung auf reine Botschaftskommunikation zwischen Prozessen auf verschiedenen Prozessoren und durch die vollständige Integration in Compiler und Betriebssystem unterscheiden sich die vorgeschlagenen Konstrukte von anderen Ansätzen /BONN81/,/AMMA81/.
Im folgenden wird zunächst der derzeitige Stand der Diskussion um Kommunikationsmechanismen skizziert, darauf die vorgeschlagenen Konstrukte erklärt und schließlich die Implementation vorgestellt.

*) gefördert von der DFG im Rahmen des Vorhabens Ho824

1. Kommunikation und Synchronisation in verteilten Anwenderprogrammen

1.1 Begriffsbestimmungen

Ein häufig benutztes Strukturierungskonzept für Programme zur Steuerung technischer Prozesse ist die Aufteilung in einzelne Rechenprozesse. Prozesse sind Berechnungen, die parallel zueinander ausgeführt werden können. Die einzelnen Prozesse können auf den Komponenten eines verteilten Systems laufen.
Ein verteiltes System besteht aus den Systemkomponenten und dem Übertragungsmechanismus /BOCH79/ (siehe B i l d 1).



B i l d 1 : Grundstruktur eines verteilten Systems

Systemkomponenten können Recheneinheiten, intelligente E/A-Geräte, usw. sein. Der Übertragungsmechanismus kann z.B. von einem lokalen Netz oder von einem öffentlichen Netz (z.B. Datex P) angeboten werden. Laufen die Rechenprozesse eines Programmes auf verschiedenen Komponenten eines verteilten Systems, spricht man von einem verteilten Programm.

1.2 Kommunikations- und Synchronisationskonzepte

Obwohl Rechenprozesse unabhängig voneinander ablaufen, müssen sie u.U. untereinander in Verbindung treten bzw. ihre "Arbeit" koordinieren. Außerdem müssen in der Prozeßautomatisierung die Rechenprozesse mit dem technischen Prozeß Steuerinformationen austauschen bzw. sich mit diesem Synchronisieren.
Um die Zusammenarbeit von Prozessen (Synchronisation und Kommunikation) zu organisieren, wurden verschiedene Konzepte vorgeschlagen. Dabei werden zwei Konzeptklassen unterschieden /STAU82/:

- Synchronisations- und Kommunikationskonzepte, die auf gemeinsamen Objekten aufbauen,
- Synchronisations- und Kommunikationskonzepte, die auf Botschaftsmechanismen basieren.

Diese beiden Konzeptklassen schlagen sich auf Spezifikation und Implementation nieder. In der Spezifikation wird ein Programm auf abstrakterer Ebene beschrieben, als dies in der programmiersprachlichen Formulierung der Implementation der Fall ist /KKST79/.
Ein Vergleich von Programmiersprachen mit verschiedenen Synchronisations- und Kommunikationskonzepten findet sich in /STOT82/.

1.2.1 Gemeinsame Objekte

Objekte können z.B. Variable, Dateien, Prozeduren usw. sein. Gemeinsame Objekte sind Objekte, die von mehreren Prozessen benutzt werden können. Eine Datei kann von mehreren Prozessen beschrieben oder gelesen werden. Eine Prozedur, die nicht wiedereintrittsfähig ist, wird von mehreren Prozessen u.U. gleichzeitig aufgerufen.
Um zu gewährleisten, daß der Zustand eines gemeinsamen Objektes konsistent ist, müßten die einzelnen Prozesse sich über dessen Verwendung "absprechen". Die Kommunikation von Prozessen wird als Spezialfall der Synchronisation betrachtet. Prozesse tauschen Daten über eine entsprechendes gemeinsames Objekt aus (i.A. ein Puffer) /KERA82/.
Spezifikationskonzepte für die Beschreibung von parallelen Programmen, die gemeinsame Objekte benutzen, sind z.B. die in /GOEH81/, /KERA79/ und /LAV78/ vorgestellten Modelle. Synchronisationskonzepte, die in Programmiersprachen verwendet werden und auf gemeinsamen Objekten beruhen, sind z.B. Semaphore, Monitore, kritische Abschnitte, Programmiersprachen, die solche Konzepte enthalten, sind z.B. PEARL, CONCURRENT PASCAL, MODULA.
Synchronisationskonzepte, die auf gemeinsamen Objekten basieren, gehen davon aus, daß die Rechenprozesse einen gemeinsamen Speicher benutzen. Sollen deshalb gemeinsame Objekte zur Programmierung verteilter Systeme verwendet werden, so gibt es vor allem folgende Probleme:

- die Implementierung von Synchronisationskonzepten basierend auf gemeinsamen Objekten wird ineffizient /LEVI81/, /KEME79/, und
- es gibt Probleme durch die Fehleranfälligkeit des Übertragungsmechanismus /BONN81/.

Außerdem ist es schwierig, die Kommunikation eines Programmes mit dem technischen Prozeß durch gemeinsame Objekte zu beschreiben. So werden z.B in PEARL zur Synchronisation der Rechenprozesse Semaphore benutzt, für die Kommunikation und Synchronisation mit dem technischen Prozeß indessen Ein/Ausgabeanweisungen und Anweisungen zur Einplanung von Interrupts. Diese Anweisungstypen haben aber große Ähnlichkeit mit Botschaftsoperationen.

1.2.2 Botschaftsmechanismen

Botschaftskonzepte wurden zur Programmierung von verteilten Systemen vorgeschlagen. Da verteilte Systeme in den letzten Jahren erheblich an Bedeutung gewonnen haben, wurden die meisten Botschaftskonzepte auch erst in jüngster Zeit veröffentlicht (/HOAR78/, /HANS78/,

/ADA79/, /BOOS81/, /FELD79/, /BOCH79/, /MAYE80/,/COOK80/).
Diese Konzepte unterscheiden sich vor allem in folgenden Punkten /GENT81/:
- Blockiert und Nichtblockiertbedingungen
- Adreß- und Absenderangaben
- Botschaftsformate
- Verhalten bei Übertragungsfehlern
- Abstraktionsgrad.

Programmiersprachen, in die Botschaftskonzepte eingebettet sind, sind z.B. ADA, *MOD.
Spezifikationssprachen für verteilte Programme werden in /BOCH79/ und /BRR081/ beschrieben.
Botschaftskonzepte sind zur Programmierung von Rechensystemen mit gemeinsamen Speichern nicht sehr effizient. Wird z.B. eine Datei von mehreren Prozessen beschrieben oder gelesen, so wird bei Programmiersprachen, die zur Synchronisation und Kommunikation nur Botschaftsmechanismen besitzen, ein Verwaltungsprozeß vorgeschaltet, der die Lese- und Schreibaufträge für die Benutzerprozesse ausführt und dann die Ergebnisse an diese zurückgibt.
Allerdings läßt sich die Zusammenarbeit technischer Prozeß/Rechenprozesse natürlicher beschreiben. Signale vom technischen Prozeß zu den Rechenprozessen bzw. umgekehrt lassen sich leicht als Botschaften interpretieren.

1.2.3 Nichtdeterministische Kontrollstrukturen

In Zusammenhang mit Botschaftskonzepten werden häufig die von Dijkstra /DIJK75/ eingeführten nichtdeterministischen Kontrollstrukturen verwendet /HOAR78/, /HANS78/, /ADA79/.
Mit dieser Anweisungsart wird es möglich, auf verschiedene Nachrichten von verschiedenen Prozessen gleichzeitig zu warten. In den Sprachen mit Botschaftskonzepten werden unterschiedliche nichtdeterministische Kontrollstrukturen verwendet. Dabei lassen sich vor allem folgende vier Typen unterscheiden:

- 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 /ADA79/ und /HOAR78/ kann in der Bedingung auch eine erwartete 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.
Ein Prozeß, der eine Guarded Region abarbeitet, prüft, ob eine Bedingung eines Guards wahr ist und führt dann die zugehörige Anweisungsfolge aus. Danach gilt die nichtdeterministische Kontrollanweisung als abgeschlossen. Ist keine der Bedingungen wahr, wird der Prozeß solange blockiert, bis eine Bedingung wahr wird. Sind mehrere Bedingungen wahr, wird ein beliebiges Guard ausgeführt.
Bei Guarded Commands wird genauso verfahren. Nur wird hier der Prozeß nicht blockiert, wenn keine der Bedingungen wahr ist. Das Guarded Command wirkt dann wie eine Leeranweisung.
Führt ein Prozeß eine Guarded Loop aus, wird jede Bedingung geprüft und falls sie wahr ist, die dazugehörige Befehlsfolge ausgeführt. Dies wird solange wiederholt, bis keine Bedingung mehr wahr ist. Erst dann gilt die Guarded Loop als beendet. Eine Guarded Loop, bei der keine Bedingung wahr ist, wirkt wie eine leere Anweisung.
Ein Guarded Cycle hat eine ähnliche Wirkung wie eine Guarded Loop. Der Unterschied besteht darin, daß der Guarded Cycle nicht beendet wird, wenn keine Bedingung mehr wahr ist, sondern der abarbeitende Prozeß blockiert wird, daß

bis wieder eine oder mehrere Bedingungen wahr werden. Ein Guarded Cycle wird nie beendet.

2. Erweiterung von (AVIONIC-) PEARL

Die folgenden Sprachkonstrukte sind echte Erweiterungen von AVIONIC-PEARL, d.h. die bereits vorhandenen Sprachelemente blieben unverändert.

2.1 Botschaften

2.1.1 Vereinbarung

In PEARL werden Prozesse durch Tasks dargestellt. Da es sich bei Botschaften um Mechanismen zur Prozeß- (Task-) Kommunikation handelt, scheint es sinnvoll, den Taskkopf um die Kommunikationsstruktur zu erweitern, die angibt, welche Tasks welche Botschaften an welche anderen Tasks schicken bzw. von diesen empfangen.
Die erste Zeile in Bild 2 stellt den ursprünglichen Taskkopf dar.

```
taskname": " "TASK" ["PRIO" prioritaet] ["GLOBAL"] ["RESIDENT"]
        ["TRANSMITS" sendebotschaftsname(n)
                       [parameterbeschreibung]
                     "TO" empfaengername(n)
                       {,sendebotschaftsname(n)
                         [parameterbeschreibung]
                     "TO" empfaengername(n)
                       }... ]
        ["RECEIVES"  empfangsbotschaftsname(n)
                       [parameterbeschreibung]
                     "FROM" sendername(n)
                       {, empfangsbotschaftsname(n)
                         [parameterbeschreibung]
                     "FROM" sendername(n)
                       }... ]
                      ["NOBUFFER" /
                       "BUFFER" wartebereichsgroesse ] ";"
```

Bild 2
Halbformale Darstellung des erweiterten Taskkopfs

Mit der TRANSMITS-Klausel werden die Sendebotschaften vereinbart. Sie besteht aus dem Botschaftsnamen bzw. einer geklammerten Namenliste, gefolgt von einer optionalen Parameterliste. Diese setzt sich zusammen aus dem Parameternamen (bzw. einer geklammerten Namenliste), dem der Parametertyp (ein elementarer PEARL-Datentyp) folgt. Hierauf folgt das Schlüsselwort "TO" und der bzw. die Empfängernamen.
Die RECEIVES-Klausel ist analog der TRANSMITS-Klausel aufgebaut. In ihr werden die Empfangsnachrichten vereinbart. An Stelle des Schlüsselworts "TO" steht das Schlüsselwort "FROM". Anschließend folgt die Wartebereichsvereinbarung (für die Empfangsnachrichten). Bei Angabe von NOBUFFER existiert kein Wartebereich Bei Angabe von BUFFER existiert ein Wartebereich endlicher Größe. Die Größenangabe legt die Zahl der Botschaften fest, die im Wartebereich Platz finden.
Die Bedeutung von Wartebereichen ist unter 2.1.2 beschrieben.
Das zugehörige Anschlußschema wird im Systemteil beschrieben. Es gibt an welche Tasks über welche Verbindungen kommunizieren.

2.1.2 Verwendung

Die Darstellung der Botschaftsoperationen ähnelt den PEARL-E/A-Anweisungen:

Sendeanweisung:

```
"TRANSMIT" "FROM" botschaftsname
"TO" empfängername ";"
```

Mit der TRANSMIT-Anweisung wird diejenige Botschaft, die durch den angegebenen Namen identifiziert wird, mit ihren Parametern gesendet. Letztere haben vorher ihren Wert per Zuweisung erhalten.
Ist der Empfänger nicht empfangsbereit, wird der Sender blockiert, falls der Empfänger nicht über einen Wartebereich verfügt. Ist der Wartebereich des Empfängers voll, wird der Sender ebenfalls blockiert.

Empfangsanweisung:

```
"RECEIVE" "FROM" sendername "TO"
botschaftsname ";"
```

Mit der RECEIVE-Anweisung werden Botschaften aus dem Wartebereich bzw., falls kein Wartebereich vorhanden ist, von anderen Tasks anstehende Botschaften übernommen, d.h. ihre Parameterwerte werden in die Parameter der Empfangsbotschaft kopiert.

Befindet sich die Botschaft nicht im Wartebereich bzw. ist der Sender nicht sendebereit, so wird der Empfänger blokkiert.

2.2 Guarded Regions/Commands

2.2.1 Aufbau und Bedeutung

Der Aufbau eines Guarded Statement ist B i l d 3 zu entnehmen.

```
guarded-statement: "GUARDED"  region / command
                   "GUARDEND" ";"

region: "REGION"
        guards
        ["TIMEOUT" "AFTER" duration-ausdruck
                   "REACT"  anweisungsfolge ]

command: "COMMAND"
         guards
         ["OUTREACT" anweisungsfolge]

guards: "GUARD" guard-element; {guard-element; } ...
              "REACT" aneisungsfolge
        {"GUARD" guar-delement; {guard-element} ...
                 "REACT"  anweisungsfolge } ...

guard-element:  "WHEN" interrupt /
                request-anweisung /
                get-anweisung /
                receive-anweisung /
                release-anweisung /
                put-anweisung /
                transmit-anweisung
```

Bild 3
Halbformale Darstellung der nichtdeterministischen Kontrollanweisungen

Die Guarded Statements werden von oben nach unten, die Guards von links nach rechts abgearbeitet. 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.
Guarded Loops und Guarded Cycles wurden zunächst nicht implementiert.

2.2.2 Verwendung

Einige Beispiele:

- alternatives Warten auf verschiedene Interrupts (B i l d 4):

```
MODULE;
   :
PROBLEM;
SPC (POSI, ENDE) INTERRUPT;
   :
SCHRITTMOTORSTEUERUNG: TASK GLOBAL;
   :
        GUARDED REGION
          GUARD WHEN POSI; REACT /* REGULAERE BEARBEITUNG
                                    FORTSETZEN            */
          GUARD WHEN ENDE; REACT /* FEHLERBEHANDLUNG      */
          TIMEOUT AFTER 1 MIN;
                           REACT /* SCHRITTMOTOR KAPUTT   */
        GUARDEND;
   :
END;
MODEND;
```

Bild 4
Alternatives Warten auf mehrere Interrupts

Ein Prozeß zur Steuerung eines Schrittmotors soll alternativ auf zwei Interrupts warten:

a) die gewünschte Position wird erreicht; dies wird durch einen Interrupt (hier: POSI) angezeigt.
b) einer der beiden Endkontakte wird erreicht; die Rückmeldung hierfür erfolgt durch einen anderen Interrupt (hier: ENDE).

- Zeitüberwachung von Semaphoroperationen (B i l d 5)
  Ein Prozeß soll an einem Semaphor nur eine maximale Zeitdauer blockiert werden können.

```
MODULE;
   :
PROBLEM;
SPC MUTEX SEMA;
   :
ZUGRIFF: TASK GLOBAL;
   :
        GUARDED REGION
          GUARD REQUEST MUTEX; REACT /* ZUGRIFFSAKTION */
          TIMEOUT AFTER 5 SEC; REACT /* AUSWEICHAKTION */
        GUARDEND;
   :
END;
MODEND;
```

Bild 5
Zeitüberwachen von Semaphoreoperationen

- Zeitüberwachung von Eingaben
  Analog der Zeitüberwachung von Semaphoren kann das Warten bei Eingabeoperationen ebenfalls zeitlich begrenzt werden.

- gleichzeitiges Empfangen mehrerer Botschaften (B i l d 6)

```
MODULE;
SYSTEM;
AUSWAHL:<-ASNR*1; /* BOTSCHAFTEN AN AUSWAHL
   :                  GEHEN UEBER LEITUNG 1  */

PROBLEM;
SPC AUSWAHL TASK;
      :
MESS1: TASK GLOBAL
       TRANSMITS MESSW1 (DRUCK FLOAT) TO AUSWAHL;
          :
       TRANSMIT FROM MESSW1 TO AUSWAHL;
          :
END;

/* MESS2 UND MESS3 SIND ANALOG AUFGEBAUT UND BEFINDEN SICH IN
   ANDEREN MODULN AUF ANDEREN PROZESSOREN                    */

MODEND;


MODULE;
SYSTEM;
MESS1:->ASNR*1;
MESS2:->ASNR*2;
MESS3:->ASNR*3;
   :
PROBLEM;
SPC (MESS1,MESS2,MESS3) TASK;
      :
AUSWAHL: TASK GLOBAL
         RECEIVES MESSW1 (DRUCK FLOAT) FROM MESS1,
                  MESSW2 (DRUCK FLOAT) FROM MESS2,
                  MESSW3 (DRUCK FLOAT) FROM MESS3
         NOBUFFER;
            :
         GUARDED REGION
           GUARD RECEIVE FROM MESS1 TO MESSW1;
                 RECEIVE FROM MESS2 TO MESSW2;
                 RECEIVE FROM MESS3 TO MESSW3;
             REACT /* VERGLEICHEN UND AUSWERTEN */

           TIMEOUT AFTER 0.5 SEC;
             REACT /* FEHLERBEHANDLUNG */

         GUARDEND;
            :
END;
MODEND;
```

Bild 6
Gleichzeitiges Warten auf Nachrichten

Drei Prozesse messen unabhängig voneinander die gleiche Prozeßgröße (z.B. Druck, Temperatur, usw.). Ein vierter Prozeß vergleicht die Meßwerte und nimmt denjenigen als richtig an, der (mit gewissen Toleranzen) von mindestens zwei Meßtasks geliefert wurde. Bei Ausfall eines Meßprozesses sollen entsprechende Fehlerbehandlungen eingeleitet werden.
Die Meßtasks sollen erst fortgesetzt werden, wenn alle Meßwerte bei der Auswahltask eingetroffen sind.

3. Implementierung

3.1 Compiler und Codegenerator

Der maschinenunabhängige obere Compiler wurde um die oben genannten Sprachkonstrukte erweitert. Hierfür mußten neue Anweisungen für die Zwischensprache (CIMIC/AV) geschaffen werden. Ausserdem wurde der Codegenerator den neuen Anforderungen angepaßt.

3.2 Betriebssystem

Ein bereits vorhandenes, eigens für das gesamte PEARL-System entwickeltes, portables Betriebssystem mußte den neuen Anforderungen angepaßt werden. Die Änderungen erforderten u.a. die Einführung von sog. "Blockiertbereichen" für jede Empfängertask. Diese dienen zur Speicherung von Sendertask- und Botschaftsnamen, die entweder im Wartebereich keinen Platz fanden oder, falls kein Wartebereich vorhanden ist, die Empfängertask nicht empfangsbereit vorfanden.

4.3 Implementationseinschränkungen

Da die Überlegungen über die Verklemmungsfreiheit der entworfenen Guarded Statements noch nicht abgeschlossen sind (eine kurze Schilderung der Problematik ist zu finden in /MAYE79/), scheint es ratsam, die Mischung von Eingabe- (GET-, RECEIVE-, REQUEST-, WHEN-) und Ausgabe- (PUT-, TRANSMIT-, RELEASE-) Anweisungen (vorerst) nicht zuzulassen.

5. Abschlußbemerkungen

Absicht des voliegenden Artikels ist, einige Sprachkonstrukte zur Programmierung von verteilten Systemen vorzustellen und ihre Implementierbarkeit zu demonstrieren. Nichtsdestoweniger liegen bisher nur wenig praktische Erfahrungen vor. Es sollte deswegen daraufhingewiesen werden, daß einerseits die vorgeschlagene Syntax nur ein erster Ansatz ist, und daß andererseits eine gründliche Diskussion der Semantik der Konstrukte erfolgen sollte.

Literatur

/ADA79/ Ichbiah et. al.
Rationale for design of ADA programming language.
SIGPLAN Not. (ACM) Juni 1979

/AMMA81/ M. Ammann
PEARL für verteilte Systeme
Prozeßrechnertagung 1981
Informatik Fachberichte (39)
Springer Verlag

/BOCH79/ G. Bochmann
Distributed systems
Lecture Notes of Com. Science
Springer Verlag 1979

/BONN81/ G. Bonn, L. Lorenz
Eignung von Mehrrechner-PEARL zur Programmierung paralleler Prozesse, Erfahrung und Folgerungen
Prozeßrechnertagung 1981
Informatik Fachberichte (39)
Springer Verlag 1981

/BOOS81/ J. Boos, R. Plasmeijer, J. Stoet
Process communication based on input specifications
ACM TOPLAS juli 1981

/BRRO81/ P. Brück, J. Robra
Verfahren und Hilfsmittel für die Softwareentwicklung von Nebenstellenanlagen
TE-KA-DE Technische Mitteilungen 1981

/COOK80/ R. B. Cook
*MOD A language for distributed programming
IEEE Trans. Softw. Eng.
November 1980

/DIJK75/ E. W. Dijkstra
Guarded commands, nondeterminacy and derivation of programs
Comm. ACM August 1975

/FELD79/ J. Feldman
High level programming for distributed computing
Comm. ACM Juni 1979

/GENT81/ W. M. Gentlemen
Message passing between sequential processes: the reply primitive and the administrator concept

/GOEH81/ P. Göhner
Spezifikation der Synchronisierung paralleler Rechenprozesse in EPOS
Prozessorrechnertagung 1981
Informatik Fachberichte
Springer 1981

/HANS78/ P. B. Hansen
Distributed processes: A concurrent programming concept
Comm. ACM November 1978

/HOAR78/ C. A. R. Hoare
Communicating sequentiell Processes
Comm. ACM August 1978

/KEME79/ H. Kemen
Zur Programmierung verteilter Systeme
Workshop der GI
Informatik Fachberichte (22)
Springer 1979

/KERA82/ S. Keramidis
Eine Methode zur Spezifikation und korrekten Implementierung von asynchronen Systemen
Habilitationsschrift, IMMD Bericht der Univ. Erlangen 1982

/KKST79/ Kimm et. al.
Einführung in Software Engeneering
de Gruyter Berlin 1979

/LAVE79/ M. S. Laventhal
A constructive approach to reliable synchronisation code
Proc. 4th intern. conf. of software engeneering 1979

/LEVI81/ P. Levi
Betriebssysteme für Realzeitanwendungen
Data Context Verlag Köln 1981

/MAYE80/ T. W. Mao, R. T. Yeh
Communication port: A language concept for concurrent programming
IEEE Tran. Softw. Eng. März 80

/STOT82/ P. D. Stotts
A comparative survey of concurrent programming languages
SIGPLAN (ACM) Oktober 1982