Zur Kommunikation in verteilten Systemen: Lösung des I/O-Guard Problems mit Programmtransformationen

Communication in Distributed Systems: A Transformational Solution to the I/O-Guard Problem

Dieter Zöbel, EWH Rheinland-Pfalz, Koblenz

Die Programmierung verteilter Systeme verlangt nach mächtigen Konzepten zur Kommunikation und Synchronisation. Einige wesentliche sind auf der Grundlage des Sprachmodells CSP festgelegt worden. Zu ihnen zählt das Konzept der gemischten Kommunikationsguards, das eine nichtdeterministische Auswahl zwischen ausführbaren Sende- und Empfangsoperationen definiert und eine abstrakte Spezifikation verteilter Berechnungen zuläßt. Die prinzipiellen Schwierigkeiten bei der Implementierung der gemischten Kommunikationsguards sind als „I/O-Guard Problem" in die Fachliteratur eingegangen. Der vorliegende Artikel löst das I/O-Guard Problem mit einem entkoppelten Verfahren, das selbst wieder in CSP abgefaßt ist und mit Hilfe von Programmtransformationen in jedes beliebige CSP-Programm einbettbar ist.

The I/O-guard problem is one of the paradigmas of distributed programming. It arises in programming constructs which offer nondeterministic choice between input and output activities. Its implementation however requires a neighbourhood consensus to decide which of a set of conflicting communications actually will happen. This paper proposes the methodology of program transformations to solve the I/O-guard problem for any CSP-program. In this way the transformations replace output guards and incorporate a distributed I/O-guard protocol.

1 Einleitung

Parallele Systeme gewinnen im Zuge des Preisverfalls der Hardware immer mehr an Bedeutung. Neben den technischen Aufgabenstellungen, die im Zuge dieser Entwicklung aufgeworfen werden, sind insbesondere die programmiertechnischen Ausdrucksmittel ins Blickfeld der Diskussion gerückt. Aus einer Unmenge von Ansätzen haben sich u.a. solche als wegweisend herausgestellt, die nur noch lokale Daten für aktive Objekte (Prozesse, Prozessoren) zulassen. Für parallele Systeme dieser Gattung gewinnt die Kommunikation zwischen Objekten (im folgenden Prozesse genannt) entscheidende Bedeutung. Mit Hilfe von Nachrichten werden Daten übertragen und Prozesse synchronisiert.

Als einer der wichtigsten Ansätze bietet CSP (Communicating Sequential Processes) [Hoa 78] Sprachelemente für die Nachrichtenübertragung, die Parallelausführung von Prozessen und die nichtdeterministische Ausführung von Anweisungen. Dabei ist CSP keine vollständig definierte Sprache. Sie ist eher als Sprachrahmen zu verstehen, aus dem sich so bedeutende Sprachen wie Ada und Occam entwickelt haben. Erweitert um bekannte Datenstrukturen und Operationen sequentieller Programmiersprachen eignet sich CSP hervorragend für die Spezifikation verteilter Systeme.

An die wenigen Sprachelemente von CSP sind einige programmiertechnisch sehr mächtige Konzepte geknüpft, u.a.:

(a) die verteilte Terminationsbedingung
(b) die gemischten Kommunikationsguards (I/O-Guard Problem).

Die Spezifikation und die Programmierung werden durch diese Konzepte erleichtert, ihre Implementierung hingegen ist schwierig und aufwendig. Während für (a) bereits Lösungen existieren, die selbst wieder in CSP spezifiziert wurden (z.B.: [AptFra 84], [Zöb 86]), hat die Suche nach einer zufriedenstellenden Lösung für (b), formuliert in CSP, gerade erst begonnen. Der vorliegende Artikel nimmt sich deshalb dieser Aufgabenstellung an und beginnt mit einer kurzen Einführung in CSP, bei der zum einen die programmiertechnische Mächtigkeit und zum anderen die Schwierigkeiten der Implementierung der I/O-Guards zutage treten (Abschnitt 2). Grundsätzliche Überlegungen zu einer Lösungsfindung führen zur Definition von zwei Normalformen CSP_{io} und CSP_{in}. In diesem Zusammenhang vereinigt CSP_{io} die für die Spezifikation und Programmierung vorteilhaften Konzepte, während CSP_{in} nur über diejenigen Grundoperationen verfügt, die sich unmittelbar implementieren lassen (Abschnitt 3). Mit Hilfe von Zustandsdiagrammen wird die Lösungsstrategie erläutert und vorab geprüft (Abschnitt 4). Die eigentliche Lösung, d.h. die Überführung von CSP_{io}- in CSP_{in}-Programme, wird schließlich in Form von Programmtransformationen angegeben (Abschnitt 5). Zum Nachweis der Korrektheit der transformierten Programme wird gezeigt, daß genau die ursprünglichen Berechnungen auch im transformierten Programm möglich sind und keine Deadlocks hinzukommen noch verlorengehen (Abschnitt 6). Abschließend wird versucht, die Bedeutung der vorgestellten Programmtransformationen insbesondere im Vergleich mit anderen Lösungen hervorzuheben (Abschnitt 7).

2 I/O-Guards in CSP

CSP-Programme bestehen aus einer beliebigen aber festen Anzahl von Prozessen:

$$P::[P_0 \| \ldots \| P_{N-1}]$$

Diese Notation der Parallelanweisung ist bedeutungsgleich mit der im folgenden verwandten Quantorenschreibweise:

$$P::[\underset{i \in \{0,\ldots,N-1\}}{\|} P_i]$$

Die Parallelanweisung startet N Prozesse und ist beendet, wenn jeder einzelne Prozeß zu Ende ist. Abgesehen von Kommunikationsanweisungen sind die Prozesse während ihrer Abarbeitung unabhängig voneinander. Nur mittels synchroner Nachrichtenübertragung treten Prozesse in Beziehung zueinander. Dabei kann ein Prozeß P_i nur dann mit einem Prozeß P_k kommunizieren, wenn beide gleichzeitig zu einer Nachrichtenübertragung bereit sind: $i,k \in \{0,\ldots,N-1\}$, $i \neq k$

P_i sendet an P_k: $P_k!(t\_exprs)$
P_k empfängt von P_i: $P_i?(t\_vars)$

Semantisch entspricht die Nachrichtenübertragung der Zuweisung eines Vektors von Ausdrücken (t_exprs) an den Vektor von Variablen (t_vars)

$$t\_vars := t\_exprs,$$

wobei die Typen komponentenweise übereinstimmen. In konsistenten Programmen gibt es zu jeder Sendeanweisung eine korrespondierende Empfangsanweisung und umgekehrt. Damit läßt sich jedem konsistenten CSP-Programm in eindeutiger Weise ein Nachrichtengraph $G_p = (V_p, E_p)$ mit $V_p = \{0,\ldots,N-1\}$ und $E_p \subseteq V_p \times V_p$ zuordnen. Dabei ist $(i,k) \in E_p$, falls wie oben eine Nachrichtenübertragung von P_i nach P_k im Programm spezifiziert ist. Das Senden bzw. Empfangen einer Nachricht kann ein auslösendes Ereignis für die Ausführung von Anweisungen sein. Als entsprechendes Sprachelement dient das Guarded Command [Dij 75]:

$$G \rightarrow C,$$

wobei die Guard G aus boole'schen Ausdrücken gegebenenfalls gefolgt von einer Kommunikationsanweisung besteht. Die Anweisungsfolge C darf nur dann ausgeführt werden, wenn die Guard G erfüllt ist, d.h. die Auswertung aller boole'schen Ausdrücke den Wert true ergibt und, sofern eine Kommunikationsanweisung vorhanden ist, die Nachrichtenübertragung bereit ist.

In einer alternativen Anweisung lassen sich mehrere Guarded Commands zusammenfassen:

$$[\ \underset{j \in Alt}{\square}\ G_j \rightarrow C_j\ ]$$

Genau eine Anweisungsfolge C_j wird ausgewählt und ausgeführt, wenn bzw. sobald mindestens eine Guard G_j erfüllt ist. Sind gleichzeitig mehrere Guards erfüllt, so wird nichtdeterministisch eine der zugehörigen Anweisungsfolgen ausgewählt und ausgeführt. Schließlich wird bei der Wiederholungsanweisung

$$*[\ \underset{j \in Alt}{\square}\ G_j \rightarrow C_j\ ]$$

die darin enthaltene alternative Anweisung solange ausgeführt, bis keine der Guards mehr erfüllt ist[1]). Der Umgang mit den strukturierten Anweisungen wird nun an einem Beispiel erläutert. N ($N \geq 2$) ganze Zahlen sollen von N Prozessen, die bidirektional in eine Reihe angeordnet sind, sortiert werden. Von einem Eingabeprozeß E_i erhält jeder Prozeß P_i, $i \in \{0,\ldots,N-1\}$, einen Wert, um nach dem Sortieren an den Ausgabeprozeß A_i einen Sortierwert auszugeben. Die Prozeßstruktur des zugehörigen Programms

$$P::[\underset{i \in \{0,\ldots,N-1\}}{\|} E_i \ \|\ \underset{i \in \{0,\ldots,N-1\}}{\|} P_i \ \|\ \underset{i \in \{0,\ldots,N-1\}}{\|} A_i]$$

ist durch den Nachrichtengraphen (Abb. 1) repräsentiert.

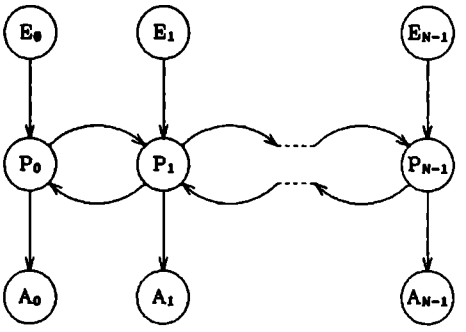


Bild 1. Der Nachrichtengraph des Sortierprogramms.

Die Prozesse E_i und A_i werden nicht weiter spezifiziert. Jeder Prozeß P_i, der einen Eingabewert erhalten hat, versucht seinen aktuellen Wert solange mit denen seiner Nachbarn zu tauschen, bis eine den Prozeßindizes entsprechende, aufsteigende Sortierung erreicht ist. Während der Sortierphase ist jeder Prozeß permanent sensibel, Werte zu empfangen und zu senden. Konkret bedeutet das für einen Prozeß P_i, der einen Wert x besitzt und einen Wert y erhält: Der Wert x ist an den Sender zu geben und y anstelle von x zu behalten, falls entweder

- y vom linken Nachbarn stammt und es gilt: $y > x$
- y vom rechten Nachbarn stammt und es gilt: $y < x$

Damit ergeben sich folgende Spezifikationen für die Prozesse P_i, $i \in \{0,\ldots,N-1\}$ mit $NB_0 = \{1\}$, $NB_{N-1} = \{N-2\}$ und $NB_i = \{i-1, i+1\}$ für $i \in \{1,\ldots,N-2\}$:

$P_i :: E_i(x)$
  $*[\ \underset{j \in NB_i}{\square}\ P_j?(y) \rightarrow$
    $[(i-j)(y-x) > 0 \rightarrow P_j!(x); x := y$
    $\square\ (i-j)(y-x) <= 0 \rightarrow P_j!(y)$
    $]$
  $\underset{j \in NB_i}{\square}\ P_j!(x) \rightarrow P_j?(x)$
  $]$
  $A_i!(x)$

[1]) Die ursprüngliche Definition geht an dieser Stelle weiter, indem die Termination benachbarter Prozesse die Termination der Wiederholungsanweisung beeinflußt [Hoa 78]. Für die folgenden Überlegungen reicht die obige Definition jedoch vollkommen aus.

Neben vielen anderen prinzipiellen Fragestellungen, die sich bei der Diskussion dieses Programms ergeben (z. B. Termination, Fairneß), soll im folgenden ausschließlich der Aspekt der Kommunikation angesprochen werden. Konkret geht es um die Frage, wie innerhalb der Umgebung eines Prozesses P_i zu entscheiden ist, welche von mehreren möglichen Nachrichtenübertragungen stattfinden wird. Der Kern der Problematik liegt darin, daß jeder Prozeß darauf wartet, eine Nachricht zu senden oder zu empfangen. D. h., keiner der Prozesse entscheidet sich definitiv, das eine oder andere zu tun. Denn wenn alle Prozesse zufällig die gleichen Annahmen über das Verhalten von Nachbarprozessen machen, kann nie eine Kommunikation stattfinden, und die Berechnung endet in einem Deadlock.

Grundsätzlich anders sind die programmiertechnischen Möglichkeiten für CSP-Dialekte, die auf Guard-Position nur Eingabeanweisungen erlauben (z. B. wie in CSP_{in} oder Occam). In CSP_{in} erfolgt jedes Senden unbedingt und bindet den Sender, bis seine Nachricht empfangen ist. In dieser Weise werden Prozesse durch den Empfang einer Nachricht angestoßen, während sie sich durch das Senden einer Nachricht synchronisieren. Eine verteilte Implementierung des einfachen Kommunikationsmechanismuses für CSP_{in} ist vergleichsweise trivial. Ziel einer Lösung zum I/O-Guard Problem ist die Angabe eines Protokolls, das zwischen zwei Nachbarprozessen entscheidet, daß eine bestimmte Nachrichtenübertragung und keine der komplementär ebenso möglichen stattfinden wird. Das Protokoll selbst ist so abzufassen, daß nur Eingabe-Guards notwendig sind.

Obwohl das I/O-Guard Problem zuerst im Zusammenhang mit CSP bekannt geworden ist, sind die Lösungen bis auf wenige Ausnahmen in informeller Art dargestellt worden (vergl. [Ber 80], [Sne 81], [BucSil 83] und [Nat 86]). Dabei liegt es doch nahe, dieses Problem, das einer Programmiersprache für verteilte Systeme innewohnt, auf der Grundlage einer implementierungsfreundlichen Untermenge dieser Sprache zu formulieren. Zu diesem Zweck werden Programmtransformationen erklärt, die ein gegebenes CSP-Programm textlich so umformen, daß alle Ausgabe-Guards verschwinden, ohne daß sich die ursprüngliche Bedeutung des Programms verändert.

3 Prinzipielle Überlegungen zum I/O-Guard Problem

Transformationen definieren eine binäre Relation zwischen formalen Sprachen. Sie werden durch Regeln zur Termersetzung verkörpert, die auf syntaktisch bestimmbare Teile eines Programms angewandt werden. Zur Lösung des I/O-Guard Problems bildet die Transformation COM_PROT Schablonen von ursprünglichen Prozessen auf Schablonen von transformierten Prozessen ab. Die ursprünglichen Prozesse müssen dafür gewisse Normalformeigenschaften besitzen, wie durch die folgende Schablone dargestellt wird:

$$
\begin{aligned}
&P_i :: D\\
&\quad *[\ \Box_{j \in B}\ B_j \rightarrow C_j\\
&\qquad \Box_{j \in I}\ B_j;\ I_j \rightarrow C_j\\
&\qquad \Box_{j \in O}\ B_j;\ O_j \rightarrow C_j\\
&\quad]
\end{aligned}
$$

Prozesse in der obigen Normalform CSP_{io} bestehen aus einem Deklarationsteil und einer großen Wiederholungsanweisung. Innerhalb der angegebenen Schablone ersetzen Stellvertreter syntaktisch korrekte CSP-Programmtexte:

- D für die Deklaration lokaler Variablen
- B für eine Folge boole'scher Ausdrücke
- I für eine Eingabeanweisung
- O für eine Ausgabeanweisung
- C für eine Anweisungsfolge, die nur noch aus Zuweisungen besteht.

Die Menge Alt aller der Alternativen teilt sich in die disjunkten Teilmengen B der rein boole'schen Guards, I der Eingabe-Guards und O der Ausgabe-Guards.

Es wurde bereits nachgewiesen, daß sich jedes CSP-Programm konstruktiv in ein semantisch äquivalentes Programm in CSP_{io} transformieren läßt [BalZöb 87]. Mit COM_PROT wird im nächsten Schritt die Transformation in die Normalform CSP_{in} vollzogen, in der nur noch Eingabe-Guards erlaubt sind:

$$
\begin{aligned}
&P_i' :: D\\
&\quad *[\ \Box_{j \in B}\ \mathrm{nsmp};\ B_j \rightarrow C_j\\
&\qquad \Box_{j \in I}\ \mathrm{smp}_j;\ B_j;\ I_j \rightarrow C_j\\
&\qquad \Box_{j \in O}\ \mathrm{smp}_j;\ B_j \rightarrow O_j;\ C_j\\
&\quad]
\end{aligned}
$$

Die Initialisierung und Berechnung der Variablen smp_j ist in der obigen Prozeßschablone noch völlig offen gelassen. Ihrer Bedeutung nach hat smp_j genau dann den Wert true anzunehmen, wenn das Kommunikationsprotokoll entschieden hat, die Nachricht in der Eingabeanweisung I_j, $j \in I$, zu empfangen bzw. in der Ausgabeanweisung O_j, $j \in O$, zu senden, um im Anschluß daran die jeweils zugehörige Anweisungsfolge C_j, $j \in I \cup O$, auszuführen. Deshalb ist smp_j anfänglich false und erhält wieder den Wert false, sobald die zugehörige Nachrichtenübertragung stattgefunden hat. Mit nsmp wird der boole'sche Ausdruck

$$\bigwedge_{j \in I \cup O} \neg \mathrm{smp}_j$$

abgekürzt. Dieser verhindert die Ausführung von Guarded Commands mit rein boole'schen Guards, wenn eine Nachrichtenübertragung bereits entschieden ist.

Für die Berechnung der Variablen smp_j sind noch einige Definitionen notwendig. So soll $d(j)$, $j \in I \cup O$, für jeden Prozeß P_i denjenigen Prozeß bezeichnen, von dem mit I_j, $j \in I$, eine Nachricht empfangen wird bzw. zu dem mit O_j, $j \in O$, eine Nachricht gesandt wird. Im

weiteren gehen wir davon aus, daß die Indizes von Guarded Commands innerhalb eines CSP-Programms eindeutig sind. Damit ist es möglich, in $CORR_{\{i,k\}}$ für je zwei Prozesse P_i und P_k, $i < k$, alle Paare von Indizes von Guarded Commands zu erfassen, die korrespondierende Kommunikationsanweisungen besitzen. $CORR_{\{i,k\}}$ ist durch den Programmtext festgelegt und kann für ein vorgegebenes CSP-Programm P durch den Compiler ermittelt werden.

Bevor nun die eigentliche Strategie zur Lösung des I/O-Guard Problems vorgestellt wird, muß eine grundsätzliche Aussage beachtet werden. Bougé [Bou 86] hat nachweisen können, daß kein symmetrischer Lösungsalgorithmus zum I/O-Guard Problem existiert. Symmetrisch bedeutet in diesem Zusammenhang, daß jede mögliche Berechnung auch dann noch möglich wäre, wenn man die Prozesse nachbarschaftserhaltend permutiert. Bei dem Beweis dieser Aussage wird deutlich, daß symmetrische Lösungen nicht deadlockfrei sein können. Dementsprechend ist die Asymmetrie, in unserem Fall induziert durch die totale Ordnung der Prozeßnummern, als Strategie einzusetzen, um Prozesse unterschiedlich zu behandeln.

4 Strategie zur Lösung des I/O-Guard Problems

Zur Berechnung von smp_j wird zwischen je zwei Prozesse P_i und P_k, $i < k$, ein Vermittlerprozeß $M_{\{i,k\}}$ eingesetzt. Dieser soll die Kommunikationswünsche der transformierten Prozesse Q_i und Q_k annehmen und entscheiden, ob eine Nachrichtenübertragung stattfinden wird. Dadurch verändert sich der ursprüngliche Nachrichtengraph $G_P = (V_P, E_P)$ erheblich:

$$G_Q := (V_P \cup V_E, E_P \cup E_E)$$

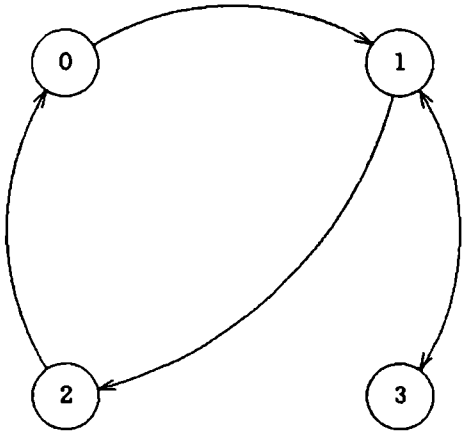
mit

$$V_E := \{e | e = \{i,k\} \text{ mit } (i,k) \in E_P \vee (k,i) \in E_P\}$$

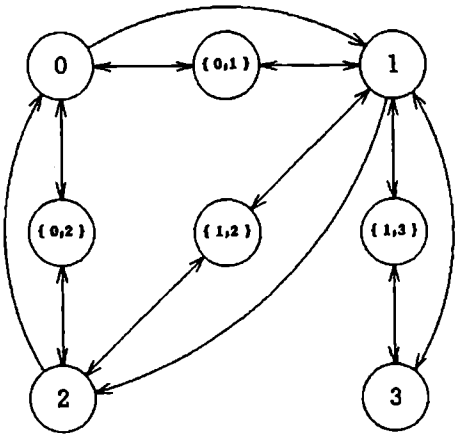
und

$$E_E := \{(e,i) \cup (i,e) \cup (e,k) \cup (k,e) | (i,k) \in E_P, e = \{i,k\}\}$$

Damit wird aus einem gegebenen Nachrichtengraphen G_P:



der Nachrichtengraph G_Q:



Zwischen den Prozessen entspannt sich nun ein Dialog von Signalen[2]), der im Falle einer Entscheidung für eine Nachrichtenübertragung etwa so ablaufen soll:

(Q) Ein Prozeß informiert die in Frage kommenden Vermittler über seine Kommunikationsmöglichkeiten.

(M) Der Vermittler wartet ab, bis zwei korrespondierende Kommunikationswünsche vorhanden sind. Als erster wird derjenige Prozeß mit der höheren Prozeßnummer davon unterrichtet.

(Q) Der Prozeß, der als erster unterrichtet wird, verpflichtet sich vorübergehend. D.h., er bestätigt, daß die offerierte Kommunikationsmöglichkeit noch besteht und wartet auf eine Antwort von diesem Vermittler.

(M) Wenn sich der erste Prozeß verpflichtet hat, wird der Vermittler den zweiten Prozeß, also denjenigen mit niedrigerem Prozeßindex, unterrichten.

(Q) Der Prozeß, der als zweiter unterrichtet wird, teilt dem Vermittler mit, daß sein Kommunikationswunsch noch besteht und ist damit zur Nachrichtenübertragung bereit.

(M) Besteht auch beim zweiten Prozeß noch der Kommunikationswunsch, dann sendet der Vermittler eine Bestätigung an den ersten Prozeß.

(Q) Beim Empfang der Bestätigung wird auch der erste Prozeß für die Nachrichtenübertragung bereit.

Diese knappe Beschreibung skizziert den erfolgreichen Ausgang einer Bemühung um eine Nachrichtenübertragung. Es wird vernachlässigt, was alles geschehen muß, wenn sich zwischenzeitlich ein Prozeß für eine Nachrichtenübertragung entschieden hat und damit für keine der außerdem angebotenen Kommunikationsmöglichkeiten mehr zur Verfügung steht. Deshalb muß das Protokoll auch vorsehen, daß ein Prozeß seine Kommunikationsofferten zurückziehen kann.

[2]) Zwar sind auch Signale nichts anderes als Nachrichten. Im folgenden wird jedoch der Deutlichkeit halber der Begriff Signal für solche Nachrichten verwandt, die für die Ausführung des I/O-Guard Protokolls notwendig sind.

Eine Reihe von Signalen unterschiedlichen Typs sind zwischen den Prozessen und dem Vermittler auszutauschen, bis die Nachrichtenübertragung stattfinden kann.

Q_i	$M_{\{i,k\}}$	
J(k)	→	Angebot zur Nachrichtenübertragung.
cncl	→	Rückzug des Angebots einer Kommunikation mit Q_k.
	← rdy_j	für i > k: informiere Q_i.
	← req_j	für i < k: informiere Q_i.
cmt	→	für i > k: Q_i verpflichtet sich.
yes	→	für i < k: Q_i ist bereit.
	← $mtch_j$	für i > k: Q_j erhält die Bestätigung und wird bereit.
	← no	für i > k: Q_i wird entpflichtet.

Es bleibt noch, die Menge J(k) zu spezifizieren. Dazu bezeichnet $I_k \subseteq I$ und $O_k \subseteq O$ für einen Prozeß P_i diejenigen Indizes von Guarded Commands mit einer Eingabe- und einer Ausgabeanweisung an P_k. Hieraus läßt sich genau die Teilmenge von $I_k \cup O_k$ bestimmen, die aktuell kommunikationsfähig ist:

$$J(k) := \{j \in I_k \cup O_k \mid B_j\}$$

Die beiden Zustandsdiagramme (Abb. 3 und Abb. 4) machen deutlich, in welcher Anordnung die Signale zwischen Prozessen und Vermittlern ausgetauscht werden, bis die eigentliche Nachrichtenübertragung stattfinden kann. Die Kanten sind mit den jeweiligen Ein- oder Ausgabeanweisungen markiert.

Für sich betrachtet ist die Signalisierung vernünftig und einleuchtend, wenn da nicht eine wesentliche Bedingung verletzt wäre. Erst bei genauerer Durchsicht des Diagramms wird deutlich, daß die Signalisierung nicht in CSP_{in} abfaßbar ist, ohne daß ein Deadlock möglich

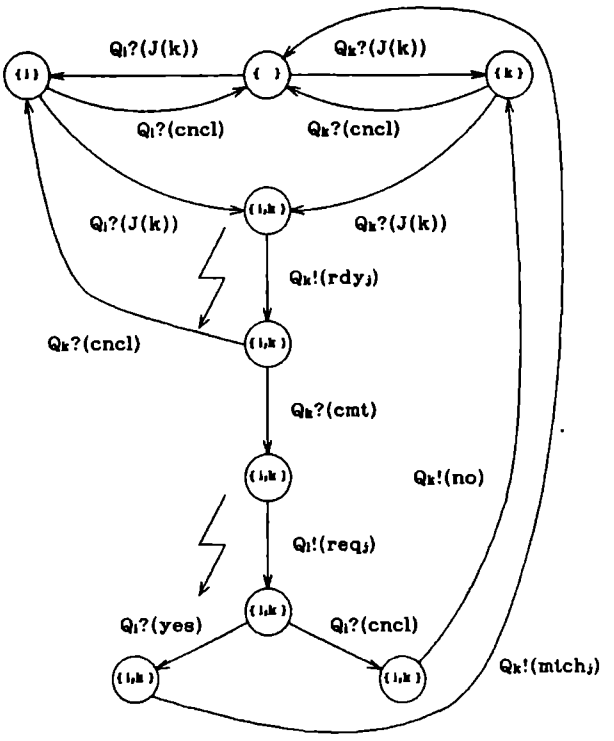


Bild 3. Zustandsdiagramm für den Vermittler $M_{\{i,k\}}$.

wird. Entscheidend sind diejenigen Positionen im Zustandsdiagramm, die durch einen Blitz gekennzeichnet sind. Aus der Sicht des Vermittlers ist ein rdy_j-Signal an Q_k bzw. ein req_j-Signal an Q_i abzusetzen. Da sich aber Q_k bzw. Q_i längst für einen anderen Kommunikationspartner entschieden haben können, sind sie vielleicht gerade dabei, cncl-Signale zu verschicken u.a. auch an $M_{\{i,k\}}$. Damit befinden sich die Prozesse Q_k und $M_{\{i,k\}}$ bzw. Q_i und $M_{\{i,k\}}$ in einem Deadlockzustand.

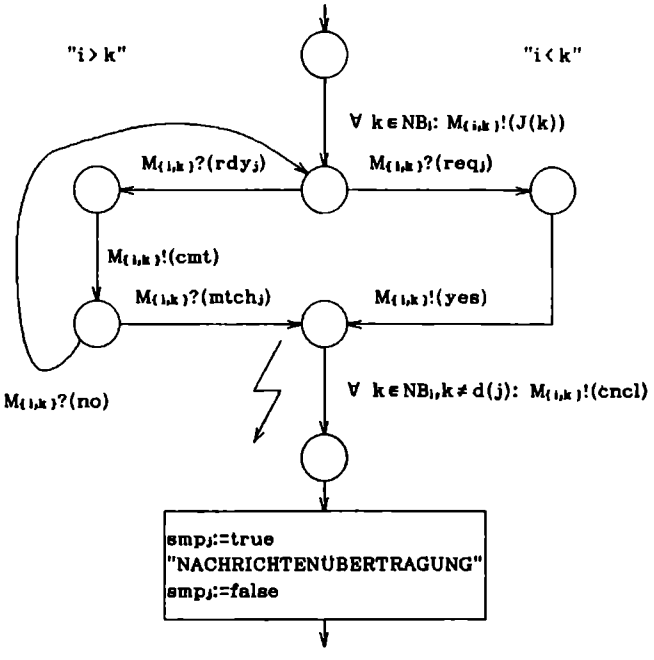


Bild 2. Zustandsdiagramm für den Prozeß Q_i.

5 Programmtransformationen zur Lösung des I/O-Guard Problems

Um die beschriebene Strategie grundsätzlich beibehalten zu können, ist nach den Wurzeln des Übels zu suchen. Dieses ist in der Tatsache begründet, daß der jeweilige Prozeß als auch ein zugehöriger Vermittler Signale senden wollen und währenddessen keine Signale abnehmen können, wie das beispielsweise in CSP_{io} möglich wäre. Das I/O-Guard Protokoll ist jedoch in CSP_{in} abzufassen. Um nun Deadlocks zu verhindern, ist lediglich dafür zu sorgen, daß der Prozeß Q_k bzw. der Prozeß Q_i wieder die Bereitschaft erlangt, rdy_j- bzw. req_j-Nachrichten zu empfangen. Zu diesem Zweck wird das Versenden von cncl-Signalen eigens dafür vorgesehenen Prozessen CM_i, $i \in \{0, \ldots, N-1\}$, zugeordnet. Die nachfolgende Tabelle enthält alle Signale, die neu hinzukommen oder mit anderen Zielen versandt werden.

Q_i CM_i $M_{[i,k]}$

RJCT → Die Menge der Indizes von Guarded Commands, die zu solchen Prozessen Q_k gehören, mit denen keine Kommunikation stattfinden wird.

cncl → Eigentlicher Rückzug des Angebots einer Nachrichtenübertragung mit Q_k.

← fnsh Alle cncl-Signale sind versandt.

Die den Vermittlern angebotenen Kommunikationsmöglichkeiten seien mit der Menge COM bezeichnet.

$$COM := \bigcup_{k \in NB_i} J(k)$$

Diese sind vollständig zurückzuziehen, wenn sich der Prozeß Q_i für die Ausführung des j-ten Guarded Command, $j \in B$, entschließt:

$$RJCT := COM$$

Wird statt dessen mittels Kommunikationsprotokoll ein Guarded Command j, $j \in I \cup O$, ausgewählt, so steht eine Nachrichtenübertragung mit Prozeß $Q_{d(j)}$ an, nachdem alle übrigen Offerten zurückgezogen sind:

$$RJCT := COM - J(k)$$

Durch die Hinzunahme der Prozesse CM_i wird der Nachrichtengraph noch größer:

$$G_Q := (V_P \cup V_E \cup V_{CM}, E_P \cup E_E \cup E_{CM})$$

mit

$$V_{CM} := \{cm_i | i \in V_P\}$$

und

$$E_{CM} := \{(i, cm_i) \cup (cm_i, i) | i \in V_P\} \cup \{(cm_i, e) | (i, e) \in E_E\}$$

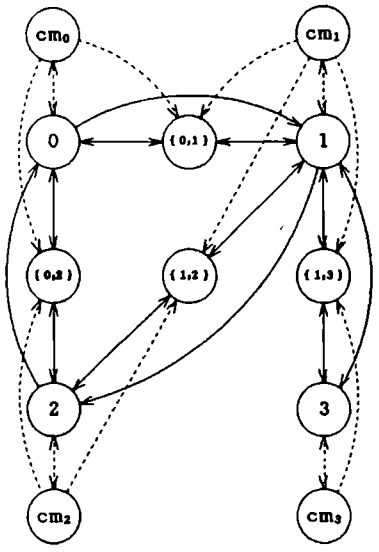


Bild 4. Die neu hinzukommenden Prozesse CM_i, $i \in \{0, \ldots, N-1\}$, sind zur Verdeutlichung durch gestrichelte Linien in den bereits existierenden Graphen eingebunden.

Für jede CSP_{io}-Programmschablone P wird das I/O-Guard Problem mit Hilfe der Transformation COM_PROT gelöst. Für

$$P :: [\ \|_{i \in \{0, \ldots, N-1\}}\ P_i]$$

ist die Programmschablone Q durch COMPROT(P) festgelegt. Dabei setzt sich Q aus den Schablonen der Prozesse Q_i, die die Berechnung der ursprünglichen Prozesse P_i verkörpern, den Vermittlern $M_{[i,k]}$ und den Prozessen CM_i für die cncl-Nachrichten zusammen:

$$Q :: [\ \|_{i \in \{0, \ldots, N-1\}}\ Q_i \ \|_{(i,k) \in E_P \vee (k,i) \in E_P}\ M_{[i,k]} \ \|_{i \in \{0, \ldots, N-1\}}\ CM_i]$$

In drei getrennten Transformationenregeln lassen sich aus P_i, bzw. den Nachbarprozessen P_i und P_k alle Prozesse von Q gewinnen:

$Q_i = COM\_PROT_1(P_i)$ $\quad i \in \{0, \ldots, N-1\}$

$M_{[i,k]} = COM\_PROT_2(P_i, P_k)$ $\quad i, k \in \{0, \ldots, N-1\}, i \in NB_k, k \in NB_i$

$CM_i = COM\_PROT_3(P_i)$ $\quad i \in \{0, \ldots, N-1\}$

$COM\_PROT_1(P_i) =$

step := 0; COM = { }; ∀ j ∈ I ∪ O: smp_j := false;
*[□ step = 0 ∨ step = 1; j ∈ COM; B_j → COM := COM ∪ J(d(j));
j ∈ I ∪ O
$M_{[i,d(j)]}$!(J(d(j))); step := 1
□ step = 0; B_j → C_j
j ∈ B
□ step = 1; COM = {j ∈ I ∪ O | B_j} → step := 2
□ step = 2; B_j → CM_i!(COM); step := 5 + j
j ∈ B
□ step = 2; $M_{[i,d(j)]}$?(rdy_j) → $M_{[i,d(j)]}$!(cmt); step := 3
j ∈ I/O>
□ step = 2; $M_{[i,d(j)]}$?(req_j) → $M_{[i,d(j)]}$!(yes); smp_j := true;
j ∈ I/O<
CM_i(COM − J(d(j))); step := 4
□ step = 3; $M_{[i,d(j)]}$?($mtch_j$) → smp_j := true;
j ∈ I/O>
CM_i(COM − J(d(j))); step := 4
□ step = 3; $M_{[i,d(j)]}$?(no) → step := 2
j ∈ I/O>
□ step > = 4; $M_{[i,d(j)]}$?(rdy_j) → SKIP
j ∈ I/O>
□ step > = 4; $M_{[i,d(j)]}$?(req_j) → SKIP
j ∈ I/O<
□ step = 4; smp_j; CM_i?(fnsh) → I_j; C_j; smp_j := false;
j ∈ I
COM := { }; step := 0
□ step = 4; smp_j; CM_i?(fnsh) → O_j; C_j; smp_j := false;
j ∈ O
COM := { }; step := 0
□ step = 5 + j; CM_i?(fnsh) → C_j; COM := { }; step := 0
j ∈ B
]

Für die Programmtransformation $COM\_PROT_1(P_i)$ ist zwischen denjenigen Guarded Commands mit Kommunikationsoperationen zu Prozessen mit höherem Index und denjenigen mit niedrigerem Index zu unterscheiden. Diese Indexmengen sind direkt ableitbar:

$$I/O_< := \{j \in I \cup O | i < d(j)\}$$
$$I/O_> := \{j \in I \cup O | i > d(j)\}$$

Für die beiden Prozesse P_i und P_k, die in COM_PROT$_2$ eingehen, soll gelten: i < k.

```
COM_PROT2(Pi, Pk) =
  offri := false; offrk := false; step := 0;
  *[ □  step = 0; ¬offri; Qi?(COMi) → offri := true
   i ∈ {i,k}
       □      step = 0; offri; offrk; x ∈ COMi; j ∈ COMk →
   (x,y) ∈ CORR{i,k}
              ji := x; jk := y; step := 1
     □   step = 1; j = jk → Qk!(rdyj); step := 2
   j ∈ Ii ∪ Oi
   □ step = 2; Qk?(cmt) → step := 3
       □    step = 3; j = ji → Qi!(reqj); step := 4
   j ∈ Ik ∪ Ok
       □    step = 1; j = jk; Qi?(yes) → Qk!(mtchj);
   j ∈ Ii ∪ Oi
              offri := false; offrk := false;
              COMi := { }; COMk := { }; step = 0
     □  step = 0; CMi?(cncl) → COMi := { }; offri := false
   i ∈ {i,k}
   □ step = 2; CMk?(cncl) → COMk := { }; offrk := false; step := 0
   □ step = 4; CMi?(cncl) → Qk!(no); COMi := { }; offri := false;
              step := 0
  ]

COM_PROT3(Pi) =
  cncl_rec := false;
  *[Qi?(COM) → cncl_rec := true
      □   cncl_rec; j ∈ COM → M{i,d(j)}!(cncl);
   j ∈ I ∪ O
              COM := COM − J(d(j))
   □ cncl_rec; COM = { } → Qi!(fnsh); cncl_rec := false
  ]
```

6 Nachweis der Korrektheitseigenschaften

Eine notwendige Konsistenzeigenschaft, die von jedem Kommunikationsprotokoll zu fordern ist, besteht darin, daß jedes zu verschickende Signal nach endlich vielen Berechnungsschritten vom Empfängerprozeß aufgenommen wird. Als Grundlage der Beweisführung dienen die Zustandsdiagramme, sofern wir nachweisen können, daß sich die darin enthaltenen Deadlocks verhindern lassen. Zu diesem Zweck sind in Q die Prozesse CM_i vorhanden, die für das Zurückziehen der Kommunikationsangebote eingesetzt werden. Mit ihrer Hilfe kehrt jeder der Prozesse in einen empfangsbereiten Zustand zurück:

- Q_i sendet RJCT an den empfangsbereiten Prozeß CM_i. Anschließend wird Q_i empfangsbereit für fnsh- und für rdy$_i$- bzw. req$_i$-Signale.
- $M_{\{i,k\}}$ muß warten, bis Q_i empfangsbereit wird, um rdy$_j$ an Q_i, i > d(j), bzw. req$_j$ an Q_i, i < d(j), zu senden. Dann ist $M_{\{i,k\}}$ empfangsbereit für ein cncl- oder ein rdy$_j$- bzw. req$_j$-Signal.
- CM_i empfängt RJCT und muß begrenzt warten, bis die jeweiligen Vermittler $M_{\{i,k\}}$ für cncl-Signale empfangsbereit sind. Danach wird fnsh an den empfangsbereiten Prozeß Q_i gesandt und CM_i geht in den empfangsbereiten Initialzustand über.

Die bereits erwähnte Deadlockgefahr ist als durch die Hinzunahme der Prozesse CM_i beseitigt. Es bleibt zu zeigen, daß bezogen auf die Gesamtheit der Prozesse keine Deadlocks in Form zyklischer Wartebeziehungen entstehen können. In diesem Sinn kritisch ist die Verpflichtung eines Prozesses Q_k, auf die Antwort zu einem cmt-Signal zu warten. Diese kann erst erfolgen, nachdem der beauftragte Vermittler $M_{\{i,k\}}$ ein req$_j$-Signal an Q_i absetzen konnte. Damit ergibt sich die Wartebeziehung, wie sie in Abb. 6 dargestellt ist.

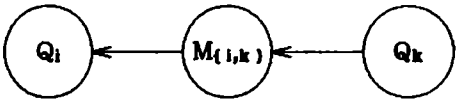


Bild 5. Die gerichteten Kanten verkörpern die Beziehung „wartet auf".

Eine analoge Wartebeziehung, wie sie für Prozeß Q_k entwickelt wurde, läßt sich von Q_i aus weiterentwickeln. Dennoch kann niemals eine geschlossene Kette von Wartebeziehungen entstehen, da folgende Bedingung erfüllt ist:

$$i < k \quad \text{mit } i \in \{0, \ldots, N-2\} \text{ und } k \in \{1, \ldots, N-1\}$$

Es gibt somit immer einen Prozeß Q_{imin} mit kleinstem Index, der über den Vermittler wieder eine Antwort zurückliefert und den zugehörigen Prozeß Q_{kmin} entpflichtet. Damit wird Q_{kmin} in der noch verbleibenden Kette zu Q_{imin}, die durch Wiederholung dieses Vorgangs schließlich zerbricht.

Im Gegenzug gehen keine Deadlocks verloren, die bereits in P vorhanden sind. Dazu nehmen wir an, von P_i aus gäbes es eine Wartebeziehung, die sich nie wieder löst. Dann gilt für Q_i, daß alle der zugehörigen Vermittler kein Paar $(x, y) \in CORR_{\{i,k\}}$ finden, damit die rdy$_j$-, req$_j$-Signalfolge ablaufen kann.

Neben der Deadlockfreiheit bleibt zu zeigen, daß Q das I/O-Guard Protokoll beinhaltet. Für einen einzelnen Prozeß Q_i setzt das voraus, daß die Invariante

$$I_{lokal} \equiv \neg smp_j \vee (B_j \wedge \bigwedge_{l \in I \cup O, l \neq j} \neg smp_l) \qquad j \in I \cup O$$

bei jeder Auswertung der Guards der großen Wiederholungsanweisung erfüllt ist. Die Gültigkeit von I_{lokal} wird algorithmisch erreicht, indem in die Menge COM nur Guarded Commands aufgenommen werden, deren boole'scher Anteil true ist (step = 0 und step = 1). Aus COM wählt dann der Vermittler ein einziges Guarded Command j, j ∈ I ∪ O, zur Nachrichtenübertragung aus. Dazu empfängt Q_i (step = 2) entweder die Signalfolge rdy$_j$ und mtch$_j$, für i > d(j), oder das Signal req$_j$, für i < d(j). Im Anschluß daran ist smp$_j$ für die Dauer der Nachrichtenübertragung erfüllt.

Während I_{lokal} nur notwendig ist, bedarf es einer globalen Invariante, aus der hervorgeht, daß eine Nachrichtenübertragung stattfinden wird, sobald sich einer der Partner dazu entschlossen hat. Sei deshalb $i < k$ und die Variable smp_x für Q_i bereits erfüllt. Dann soll für Q_i und Q_k unmittelbar vor der nächsten Nachrichtenübertragung gelten:

$$I_{global} \equiv \exists\, y \text{ mit } (x,y) \in CORR_{[i,k]}: (smp_x \wedge smp_y)$$

Eine analoge Invariante soll für den Fall $i > k$ erfüllt sein.

Algorithmisch wird diese Invariante dadurch gesichert, daß nur Paare $(x,y) \in CORR_{[i,k]}$ ausgewählt werden. Der Vermittler wird nun eine Nachrichtenübertragung zwischen diesem Paar von Guarded Commands herstellen, wenn sich noch keiner der zugehörigen Prozesse anderweitig entschlossen hat. Im Detail bedeutet das für Q_i und Q_k, $i < k$:

- Q_i setzt aufgrund des req_x-Signals smp_x auf true. Davor hat sich bereits Q_k beim Empfang von rdy_y verpflichtet und wird beim Empfang des $mtch_y$-Signals smp_y auf true setzen.
- Q_k erhält dann und nur dann das $mtch_y$-Signal, um smp_y zu setzen, wenn Q_i bereits smp_x auf true gesetzt hat.

Damit werden beide Prozesse nach Versendung der RJCT-Signale zu der eigentlichen Nachrichtenübertragung gelangen, bei der I_{global} erfüllt ist.

Unerwähnt geblieben ist bisher eine Eigenschaft, die das vorgestellte Verfahren vor anderen (z. B. [Ber 80], [BucSil 83], [Nat 86] und [Bor 86]) auszeichnet. Gemeint ist, daß bei der Auswertung der Guards der großen Wiederholungsanweisung nicht allein I/O-Guards erfüllt sind, sondern auch rein boole'sche Guards:

$$\{j \in B \mid B_j\} \neq \emptyset$$

In Q_i wird dieser Fall algorithmisch so repräsentiert:

- step = 0: Aus der Menge $\{j \in I \cup O \cup B | B_j\}$ wird ein Guarded Command nichtdeterministisch zur Ausführung bestimmt.
- step = 2: Die Vermittler haben bereits Kommunikationsangebote erhalten, d. h.: $COM = \{j \in I \cup O | B_j\} \neq \emptyset$. Nun wird nichtdeterministisch aus den Guarded Commands $j \in COM$, die vom Vermittler bereits ein rdy_j- bzw. req_j-Signal erhalten haben und denjenigen aus $\{j \in B | B_j\}$ eines ausgewählt. Trifft es dabei eine rein boole'sche Guard, so kann die zugehörige Anweisungsfolge C_j erst ausgeführt werden, nachdem alle Kommunikationswünsche COM zurückgezogen worden sind.

Entscheidend für die Korrektheit dieser Konstruktion ist, daß bei jeder fehlgeschlagenen Signalisierung wieder in den Zustand, in dem step = 2 gilt, zurückgekehrt wird. Damit bleibt, wie es in P_i vorgesehen ist, in Q_i auch immer die Möglichkeit offen, eine Anweisungsfolge C_j auszuführen, deren Guard rein boole'sch ist.

7 Schlußbetrachtung

Die vorgestellte Lösung besitzt Charakteristiken, die sie von den beiden bekannten, in Programmform entworfenen I/O-Guard Protokollen deutlich unterscheidet. Der Gegensatz zu der Lösung von Zöbel [Zöb 87] besteht in der strikten Entkoppelung der Prozesse, zwischen denen Nachrichten übertragen werden können. Diese Entkoppelung hat die Auswirkung, daß der Beginn der Signalisierung nicht mehr von den potentiellen Kommunikationspartnern verzögert werden kann. Die Vermittlerprozesse sind zu diesem Zweck immer bereit, Angebote zur Kommunikation entgegenzunehmen und zu behandeln. Diese Strategie ermöglicht, daß die jeweils erste sich bietende Kommunikationsmöglichkeit dann auch zu einer Nachrichtenübertragung führt. Freilich wird bei unserem Ansatz diese Entkopplung durch eine hohe Anzahl von Prozessen bezahlt. Wenn die Prozesse P_i, $i \in \{0, \ldots, N-1\}$, einen Ring bilden, dann erzeugt die Transformation 3N Prozesse, bei einem vollständigen Nachrichtengraphen sind es insgesamt sogar $2N + N(N-1)/2$.

Ein anderes, phasenweise entsprechendes Lösungsverfahren stammt von Bornat [Bor 86] und ist in der Programmiersprache Occam abgefaßt. Wie dieses benötigt auch unser Lösungsverfahren aus der Sicht des Vermittlers fünf Signale, bis sich zwei Prozesse auf eine Nachrichtenübertragung verständigt haben. Darüber hinaus bietet unser Verfahren folgende Vorteile:

(a) Es ist eine Lösung für CSP-Programme, die in einer Untermenge dieser Sprache verfaßt ist.

(b) Die Lösung berücksichtigt alternative Anweisungen sowohl mit Eingabe- als auch mit Ausgabe-Guards sowie, und das geht über den Ansatz von Bornat hinaus, auch rein boole'sche Guards.

(c) Die Lösung ist als Programmtransformation spezifiziert, wodurch jedes CSP_{io}-Programm durch eine Termersetzung in ein äquivalentes CSP_{in}-Programm überführt wird.

Gerade der zuletzt angesprochene Punkt ist wegweisend für die Programmierung verteilter Systeme. Denn gesucht ist einerseits ein hochsprachliches Spezifikationskonzept, das auf implizite Eigenschaften, wie z. B. die gemischten Kommunikationsguards aufbauen kann. Zum anderen lassen die implementierungstechnischen Möglichkeiten nur wenige Grundoperationen zu. Diese Lücke kann durch generative Manipulationen an Programmen geschlossen werden. Die angegebene Programmtransformation zum I/O-Guard Problem stellt in diesem Zusammenhang einen wesentlichen Zwischenschritt dar.

Literatur

[AptFra 84] *Apt, K. R., Francez, N.:* Modeling the Distributed Termination Convention of CSP/ACM-TOPLAS, Vol. 6, No. 3, July 1984, 370–379.

[BalZöb 87] *Balf, U., Zöbel, D.:* Normalform-Transformationen für CSP-Programme/eingereicht bei der Zeitschrift: Informatik – In Forschung und Entwicklung.

[Ber 80] Output Guards and Nondeterminism in „Communicating Sequential Processes"/ACM-TOPLAS, Vol. 2, No. 2, April 1980, 234–238.

[Bor 86] *Bornat, R.:* A Protocol for Generalized Occam/Software – Practice and Experience, Vol 16 (9), September 1986, 783–799.

[Bou 86] *Bougé, L.:* On the Existence of Symmetric Algorithms to Find Leaders in Networks of Communicating Sequential Processes/LIPT Report 86.18, Université Paris 7, January 1986.

[BucSil 83] *Buckley, G. N., Silberschatz, A.:* An Effective Implementation for the Generalized Input-Output Construct of CSP/ACM-TOPLAS, Vol 5, No. 2, April 1983, 223–235.

[Dij 75] *Dijkstra, E. W.:* Guarded Commands, Nondetermancy, and Formal Derivation of Programs/CACM, Vol. 18, No. 8, August 1975, 453–457.

[Hoa 78] *Hoare, C. A. R.:* Communicating Sequential Processes/CACM, Vol. 21, No. 8, August 1978, 666–677.

[Nat 86] *Natarajan, N.:* A Distributed Synchronization Scheme for Communicating Processes/The Computer Journal, Vol. 29, No. 2, 1986, 109–117.

[Sne 81] *Van de Snepscheut, J. L. A.:* Synchronous Communication between Asynchronous Components/Information Processing Letters 13, No. 3, Dec. 1981, 127–130.

[Zöb 86] *Zöbel, D.:* Programmtransformationen zur Ende-Erkennung bei verteilten Berechnungen/Informationstechnik it, 27. Jahrgang, Heft 4, August 1986.

[Zöb 87] *Zöbel, D.:* Transformations for Communication Fairness in CSP/Information Processing Letters, Vol. 25, No. 3, May 1987, 195–198.

Dr. rer.-nat. *Dieter Zöbel* (31). Studium der Informatik, Nebenfach Mathematik an der Universität Kaiserslautern, 1980 Diplom mit einem Thema aus dem Bereich Compilerbau. Seit 1980 wissenschaftlicher Mitarbeiter an der EWH Rheinland-Pfalz, Abt. Koblenz, im Studiengang Informatik. 1984 Promotion mit einem Thema aus dem Bereich Betriebssysteme. Zur Zeit tätig als Hochschulassistent bei derselben Institution. Schwerpunkte in Forschung und Lehre: Synchronisation, Rechnernetze, Echtzeitsysteme, verteilte Berechnungen.

Anschrift des Verfassers: Erziehungswissenschaftliche Hochschule Rhld.-Pfalz, Abt. Koblenz, Rheinau 3–4, D-5400 Koblenz.