Zeitschrift: Bulletin des Schweizerischen Elektrotechnischen Vereins, des Verbandes Schweizerischer Elektrizitätsunternehmen = Bulletin de l'Association suisse des électriciens, de l'Association des entreprises électriques suisses

Herausgeber: Schweizerischer Elektrotechnischer Verein ; Verband Schweizerischer Elektrizitätsunternehmen

Band: 72 (1981)

Heft: 23

Artikel: Mikrocomputer-Sprachen (PASCAL, CHILL, ADA, PORTAL)

Autor: Zwittlinger, H.

DOI: https://doi.org/10.5169/seals-905176

Nutzungsbedingungen

Die ETH-Bibliothek ist die Anbieterin der digitalisierten Zeitschriften auf E-Periodica. Sie besitzt keine Urheberrechte an den Zeitschriften und ist nicht verantwortlich für deren Inhalte. Die Rechte liegen in der Regel bei den Herausgebern beziehungsweise den externen Rechteinhabern. Das Veröffentlichen von Bildern in Print- und Online-Publikationen sowie auf Social Media-Kanälen oder Webseiten ist nur mit vorheriger Genehmigung der Rechteinhaber erlaubt. Mehr erfahren

Conditions d'utilisation

L'ETH Library est le fournisseur des revues numérisées. Elle ne détient aucun droit d'auteur sur les revues et n'est pas responsable de leur contenu. En règle générale, les droits sont détenus par les éditeurs ou les détenteurs de droits externes. La reproduction d'images dans des publications imprimées ou en ligne ainsi que sur des canaux de médias sociaux ou des sites web n'est autorisée qu'avec l'accord préalable des détenteurs des droits. En savoir plus

Terms of use

The ETH Library is the provider of the digitised journals. It does not own any copyrights to the journals and is not responsible for their content. The rights usually lie with the publishers or the external rights holders. Publishing images in print and online publications, as well as on social media channels or websites, is only permitted with the prior consent of the rights holders. Find out more

Download PDF: 09.07.2025

ETH-Bibliothek Zürich, E-Periodica, https://www.e-periodica.ch

Mikrocomputer-Sprachen (PASCAL, CHILL, ADA, PORTAL)

Von H. Zwittlinger

519.682.7:681.322–181.48;

1. Mose 11,7: «Wohlauf, lasset uns herniederfahren und ihre Sprache daselbst verwirren, auf dass keiner des anderen Sprache verstehe.»

1. Einleitung

Nach der Überlieferung scheiterte der Turmbau zu Babel an einer Sprachverwirrung; bei modernen babylonischen Türmen geschieht dies ebenfalls. Aber zu Moses Zeiten gab es nur verschiedene Sprachen; in der Informatik gibt es dazu noch Sprachhierarchien:

Generatorsprachen (COLUMBUS u.a.),
Interpretersprachen (BASIC, APL u.a.),
Compilersprachen (FORTRAN, COBOL, PASCAL, CHILL, ADA, PORTAL u.a.), auch High Level Languages (HLL) genannt,
Makro-Sprachen,
Assemblersprachen,
Maschinensprachen.

Maschinensprachen liegen auf dem niedrigsten Sprachniveau, Generatorsprachen auf dem höchsten.

Anliegen dieser Ausführungen ist es, das «Sprachniveau» zu erhöhen, d.h. von der noch weitgehend verwendeten Assemblersprache auf eine HLL zu wechseln. Dabei bestehen ausgezeichnete Gründe für HLL: Beim Einsatz von HLL wird eine signifikante Erhöhung der Arbeitsproduktivität bis 500% erzielt [1]. Diesem Argument werden sich wenige entziehen können. Daher wird auch eine allgemeine Umstellung von Assembler- auf HLL etwa gemäss Figur 1 vorausgesagt.

Bei der Umstellung auf HLL bestehen prinzipiell drei Möglichkeiten:

– *Erweiterungen bestehender HLL:* PROCESS-BASIC, PROCESS-FORTRAN usw. Dabei ist von Vorteil, dass früher entwickelte Software meist aufwärtskompatibel ist und nicht weggeworfen werden muss; man erspart sich einen grösseren Umschulungsaufwand und geht wenig Risiko ein, bei der Sprachauswahl gänzlich fehlgeplant zu haben. Als Nachteile sind zu vermerken: Die Spracherweiterungen sind nicht normiert, daher geht die Portabilität verloren, d.h., die babylonische Sprachverwirrung breitet sich *innerhalb* einer Sprache aus; die Sprachen stammen meist «aus der Steinzeit der Programmierung» und entsprechen modernen Anforderungen von Modularität, strukturierter Programmierung usw. nicht.

– *Neuere Sprachentwicklungen ohne Real-Time-Sprachelemente:* CORAL, ALGOL, PASCAL usw. Diese Sprachen entsprechen mehr oder weniger den modernen Ansprüchen, erzwingen Modularität, strukturiertes Programmieren; sie sind blockorientiert, einfach erlernbar usw. Da sie aber keine Real-Time-Spracheigenschaften haben, sind sie für Prozeßsteuerungen nicht zu gebrauchen; Zusätze ebenso wie Erweiterungen traditioneller Hochsprachen machen sie nicht mehr portabel.

– *Neuere Sprachentwicklungen mit Real-Time-Sprachelementen:* PEARL, RTL2, MODULA, CONCURRENT PASCAL, ADA, CHILL, PORTAL usw. Sie sind portabel, entsprechen meist modernen Vorstellungen des Software-Engineerings, machen die Assemblersprache vollständig überflüssig, normieren die Betriebssystem-Unterstützung bereits in der Sprachdefinition, sind komfortabel für den Anwender usw. Dagegen sind sie nicht angenehm für den Lieferanten, da die Forderungen nur mit grossem Aufwand zu realisieren sind. Sie bedingen oft ein viel zu grosses Betriebssystem und ein zu aufwendiges Entwicklungssystem für einfache Mikroprozessoranwendungen.

PASCAL gehört zur zweiten Kategorie, also ohne Real-Time-Befehle, ADA, CHILL und PORTAL zur dritten Kategorie dieser Einteilung. Diese haben alle Vorteile von PASCAL: einfach erlernbar, blockorientiert, unterstützen Grundablaufstrukturen («GOTO-loses Programmieren»), Strukturierung von Daten, Datenschutzkonzepte. Darüber hinaus weisen sie Real-Time-Eigenschaften auf.

2. Rückblick auf die Entwicklung (Fig. 2)

PASCAL wurde anfangs der 70er Jahre von *Niklaus Wirth* (ETHZ) kreiert, der die Sprache nach dem französischen Mathematiker Blaise Pascal benannte. Das Wirthsche PASCAL, das in dessen Buch [2] beschrieben ist, wird als die Standardversion bezeichnet. PASCAL ist gegenwärtig schon auf über 50 Rechnern implementiert worden und setzt sich immer mehr als dominierende Programmiersprache durch, besonders an den Schulen, da es viele moderne Konzepte der Programmiertechnik unterstützt. Man kann PASCAL als einen Standard für moderne Programmiersprachen ansehen. Der internationale Erfolg von PASCAL beruht zum grossen Teil auf dem in sich geschlossenen Sprachentwurf. Obwohl in letzter Zeit die Vor- und Nachteile dieser Sprache an Tagungen diskutiert wurden [3], obwohl sie im praktischen Einsatz ohne Erweiterungen nicht auskommt und nun schon gut 10 Jahre alt ist, ist sie ge-

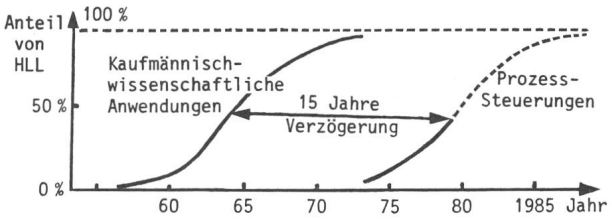

Fig. 1 Ungefähre Entwicklung des Anteils HLL an den Mikrocomputersprachen [2]

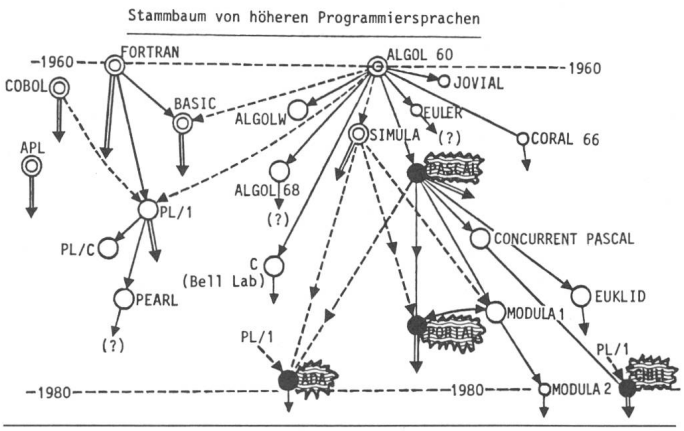

Fig. 2 Stammbaum der höheren Programmiersprachen

genwärtig sehr gefragt, speziell im Einsatz bei Mikroprozessoren.

PORTAL [4] ist auf PASCAL aufgebaut. Die erste Sprachdefinition und die ersten Implementierungen wurden im Zentrallabor von Landis & Gyr in Zusammenarbeit mit der ETHZ durchgeführt. 1977 erfolgte eine Ausweitung: PORTAL wurde ein vom Bund finanziertes Projekt 956A in Zusammenarbeit mit der EPFL. Die Sprachdefinition ist ähnlich dem PASCAL, nur wurden Prozesse, Prozesskommunikation mittels Monitor, hardwarenahe Schnittstellen, ganz allgemein Funktionen für Prozeßsteuerung geschaffen. Compiler gibt es für PDP-11, 8080, Nova 3.

CHILL steht für *C*CITT *H*igh *L*evel *L*anguage. Diese Sprache wurde im November 1980 vom CCITT[1]) zur internationalen Norm im Fernmeldewesen erhoben [5; 6]. CHILL ist eine höhere Programmiersprache, wurde speziell für Telefon- und Telex-Vermittlungssysteme entwickelt, eignet sich aber für Prozeßsteuerung ganz allgemein. Man sieht es dem Sprachkonzept an, dass viele Ideen aus den verschiedensten Sprachen, Betriebssystemen und Erfordernissen der Nachrichtentechnik darin ihren Niederschlag fanden. Ein CCITT-«Team of Specialists» entwickelte CHILL in den letzten 10 Jahren. Für die Lieferanten der internationalen Postorganisationen wird diese Normung Konsequenzen haben. Compiler wurden von allen wichtigen PTT-Zulieferanten bereits entwickelt: auf SIEMENS-, ITT-, Philips-, PDP-11-, IBM 370-, NORD-100-(RUNIT-)Rechnern. Die PTT von Holland, Japan und Italien haben ebenfalls Compiler installiert, deren Zahl sich wegen der Normung laufend weiter vergrössern wird.

ADA wurde vom US Department of Defense (DOD) spezifiziert. PASCAL-Elemente, Modularität, Prozesskonzepte sind in der Sprachdefinition enthalten. Im November 1980 erfolgte die vorläufig endgültige Definition der Sprache [7]. Gegenwärtig gibt es noch keinen Compiler für diese Sprache, dafür aber ein hervorragendes Marketing.

3. Real-Time-Eigenschaften der HLL

PASCAL ist also eine moderne Sprachentwicklung ohne Real-Time-Eigenschaften, während ADA, CHILL und PORTAL Sprachentwicklungen sind, die zwar von PASCAL abstammen, zusätzlich aber noch Assemblereigenschaften und Real-Time-Eigenschaften, d.h. Betriebssystem-Funktionen im Sprachkonzept integriert haben. Diese beiden Eigenschaften sollen im folgenden genauer diskutiert werden. Dazu werden ein paar Definitionen typischer Real-Time-Begriffe benötigt [8]:

Betriebsmittel (resource): Dies ist eine Einrichtung, innerhalb oder ausserhalb des Rechners, die von mehreren Benützern des Rechners abwechselnd verwendet wird. Ein Betriebsmittel ist von sich aus passiv, solange es nicht benützt wird, d.h. erzeugt dann keine Interrupts.

Benutzer (user): Diese Einrichtung, innerhalb (z.B. Uhr) oder ausserhalb des Rechners, erzeugt aus eigener Aktivität Ereignisse bzw. Werte.

Prozess: Für den Begriff Prozess gibt es leider zwei Definitionen: Aus der Sicht der zu automatisierenden Anlage ist es ein zeitbehafteter Vorgang, der aufgrund eigener Initiative oder durch Eintreffen von Werten bzw. Ereignissen Ausgangswerte bzw. Ereignisse erzeugt (deterministischer Automat); aus der Sicht des Betriebssystems ist es der Ablauf eines Programms im Rechner unter Benutzung von Daten, wobei die Umgebung unter vollständiger Kontrolle des Programms liegt.

Ein *Prozessor* ist eine Hardware-Einrichtung, die einen oder mehrere Prozesse ausführt.

Synchronisation bedeutet zeitliche Verkoppelung von Prozessen mit dem Ziel, die ablaufenden Vorgänge in eine vorgegebene Reihenfolge zu bringen (z.B. «Rendez-vous» in ADA).

Kommunikation ist Synchronisation, bei der gleichzeitig zwischen einem sendenden und einem empfangenden Prozess Daten ausgetauscht werden.

Mit Hilfe dieser Begriffsdefinitionen können die typischen Real-Time-Eigenschaften einer Sprache definiert werden:

Eine HLL hat Real-Time-Eigenschaften, wenn sie

- *Funktionen für die Prozesskommunikation implementiert hat,*
- *Input/Output-Funktionen sowie hardwarenahe Schnittstellen für «non-standard»-Interrupts enthält,*
- *Funktionen für Zeitbehandlung (Uhr) beinhaltet.*

Abgesehen von Standard-Input/Output-Funktionen enthält PASCAL keine dieser Anforderungen. In ADA, CHILL und PORTAL sind sie dagegen mehr oder weniger komfortabel, ausführlich und bequem implementiert. Im folgenden werden die Funktionen für die Prozesskommunikation diskutiert und deren Implementierungsgrad in den Sprachen ADA, CHILL und PORTAL angegeben.

4. Prozessdefinition, Start/Stop von Prozessen, Inkarnationen

Warum soll der bisherige Begriff Programm durch den neuen Begriff Prozess ersetzt werden?

Ein Programm ist ein Stück Software, das, gleichgültig wann es läuft, für gleiche Anfangswerte gleiche Resultate liefert. Gehaltsabrechnungen, Buchhaltungen sind Beispiele für Programme. Man sagt, ein Programm ist zeitinvariant und deterministisch (vorherbestimmt).

Bei Prozessen ist das anders: Es ist nicht gleichgültig, wann man einen Prozess startet. Auch wenn die Anfangswerte gleich sind, erhält man zu verschiedenen Zeiten verschiedene Resultate. Ein Prozess ist nicht zeitinvariant und deterministisch. Die Ursache dafür liegt in den parallellaufenden Prozessen, die miteinander kommunizieren, d.h. sich gegenseitig blokkieren und deblockieren, dabei Daten austauschen, so dass die Ergebnisse sehr verschieden ausfallen, je nach den Zuständen, in denen sich die Nachbarprozesse gerade befinden. Die Parallelarbeit ist somit die Ursache, dass Zeitinvarianz und Determiniertheit nicht mehr gültig sind.

Gründe, warum man Software in Prozesse kleiden und parallel laufen lassen soll, sind:

- Hardware wird mehr und mehr verteilt (distributed processing, Rechnernetze usw.), daher ist es natürlich, dass die steuernde Software ebenfalls verteilt wird,
- Funktionen in zentraler ebenso wie in verteilter Hardware laufen parallel; somit auch die steuernde Software,
- parallele Verarbeitung erhöht die Geschwindigkeit,
- parallele Systeme erhöhen die Zuverlässigkeit, erlauben moderne Erholungsstrategien (fail-soft, graceful degradation usw.).

Ein Prozess wird, ähnlich einer Subroutine, zuerst definiert und von aussen angerufen (Fig. 3). Der grosse Unterschied liegt darin, dass Prozesse im allgemeinen die Kontrolle an den

[1]) CCITT = Comité consultatif international télégraphique et téléphonique.

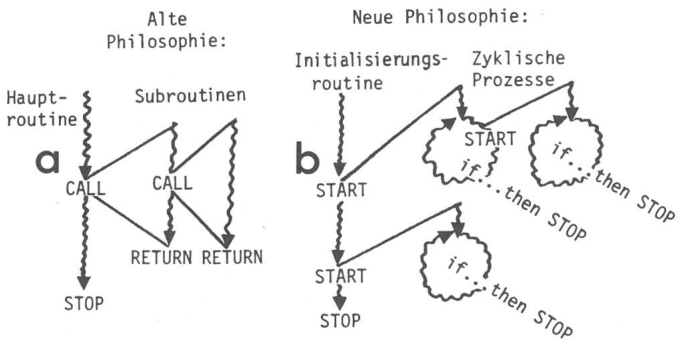

Fig. 3 Vom Programm zum Prozess
a Alte Philosophie b Neue Philosophie

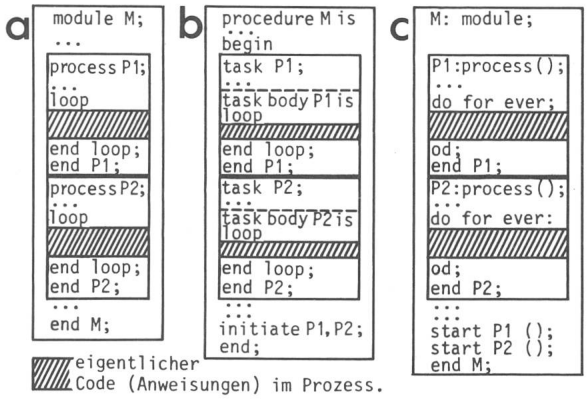

Fig. 4 Definition, Start/Stop, «ewige Loop» von Prozessen in PORTAL (a), ADA (b) und CHILL (c)

Anrufer nicht mehr zurückgeben. Sie kreisen häufig in einer «do forever»-Schlaufe und können höchstens gestoppt werden. So werden Prozesse, eingebettet in Initialisierungsroutinen, in den HLL nach Figur 4 definiert. PASCAL, als reine Batchsprache, kennt keine Prozesse. Das Prozesskonzept von PORTAL, ADA und CHILL ist so ähnlich, dass man sich die Unterschiede kaum merken kann, speziell im Formalismus des Sprachkonzepts. Nach einiger Übung erkennt man jedoch Wesensunterschiede:

PORTAL kennt keinen expliziten Prozeßstart. Ein Prozess startet automatisch, wenn man ihm einen Prozessor zuteilt.

ADA hat einen expliziten Prozeßstart, «initiate», ausserhalb der Prozessdefinition, kann also in einer Initialisierungsroutine (hier procedure *M*) und in Nachbarprozessen gezielt Prozesse starten lassen. Prozesse heissen in ADA «task» und werden in zwei Teile geteilt: Die eigentliche Task-Spezifikation mit Datentyp-Deklarierung, Kommunikationsparametern usw. und den task body mit Variablen-Deklarierung und Anweisungen, z. B. der «ewigen loop»-Anweisung («do forever»-Schlaufe).

CHILL hat ebenfalls einen expliziten Prozeßstart, die «start»-Anweisung, die ausserhalb des Prozesses in einem Modul oder Nachbarprozess codiert werden darf. Aber zusätzlich zu ADA hat CHILL noch Möglichkeiten wie in der Subroutinentechnik, beim Start Parameter mitzugeben: Aktualparameter werden in die Formalparameter der Prozessdefinition geschrieben.

Tabelle I

generic task SHIP is Spezifikation der Betriebsmittel für ein Schiff end SHIP;	task SHIP (1...1000) is Spezifikation der Betriebsmittel für ein Schiff end SHIP;
Nun definiert man die Prozesse für die einzelnen Schiffe: task QUEEN–ELIZABETH–2 is new SHIP; task FRANCE is new SHIP; task WASHINGTON is new SHIP;	Die Anzahl der Inkarnationen ist nun ebenfalls dem Compiler bekannt, im Beispiel nämlich 1000.
Der Start der Prozesse erfolgt durch: initiate QUEEN–ELIZABETH–2; initiate FRANCE, WASHINGTON;	Der Start der Prozesse erfolgt durch: initiate SHIP(1); initiate SHIP(5), SHIP(63);

Prozesse können also in CHILL durch neuerlichen Start mit anderen Aktualparametern neu «inkarniert» werden. Das heisst, dass in einem einmalig geschriebenen Code mehrere Prozess-Inkarnationen kreisen. Diese Inkarnationen (im CHILL-Jargon auch «Instances» genannt) benötigen pro Start einen neuen Satz lokaler Daten, die vom unterstützenden Betriebssystem zur Verfügung gestellt werden müssen, und zwar zur Laufzeit des Systems!

Anders in ADA: Dort sind Inkarnationen während der Laufzeit nicht vorgesehen, wohl aber zur Compilierzeit durch die Anweisung «generic» oder durch Bildung von Task-«Familien» (Tabelle I).

Der *task body* ist natürlich in beiden Fällen nur einmal codiert. Aber der Wesensunterschied zwischen ADA und CHILL ist klar. Bei CHILL sind Re-starts von Prozessen während der Laufzeit vorgesehen, ebenso Stops von Prozessen. Zur Compilierzeit sind keinerlei Angaben darüber zu machen, wieviele Inkarnationen von Prozessen vorgesehen sind. In ADA sind diese Angaben jedoch notwendig. Die Gründe dafür sind offensichtlich: In der Nachrichtentechnik kleidet man Hardwareteile gleichen Typs (z. B. Durchschalteeinheiten, Wahleinheiten usw. einer Telefonzentrale) in einen Prozess. Die Zahl der Re-starts des Prozesses richtet sich direkt nach der Anzahl von Hardwareteilen gleichen Typs, die von vornherein nicht bekannt ist. Daher gibt es während der Laufzeit eines Systems eine ständig variierende Zahl von Inkarnationen eines Prozesses, synchron der variierenden Zahl der Hardwareteile des gleichen Bautyps.

Dagegen wird man in ADA im allgemeinen wissen, wie viele Hardware gleichen Typs (im Beispiel oben SHIP) man ungefähr hat, und kann zur Compilierzeit die vorgesehenen Reservierungen für Daten (pro Inkarnation ein lokaler Datensatz) machen lassen.

In PORTAL sind keine Inkarnationen vorgesehen, ja nicht einmal explizite Prozeßstarts. Damit gibt es auch keinerlei Initialisierungsroutine: Der Prozess läuft bei der Zuteilung eines Prozessors automatisch.

5. Prozessdaten, Prozesskommunikation

Bei einem Prozess können zwei Arten von Daten unterschieden werden: lokale Daten, die nur ihm gehören, genauer, die sogar nur jeder spezifischen Inkarnation gehören, anderseits globale Daten, die mehreren Inkarnationen oder Prozessen gehören.

Tabelle II

PORTAL	ADA	CHILL
process P; var LOKALE–VAR ... ... end P;	task body P is LOKALE–VAR ... ... end P;	process P (LfP)*; dcl LOKALE–VAR ... ... end P;

Lokale Daten: In PORTAL, ADA und CHILL sind sie innerhalb der Prozess- bzw. Task-Deklaration definiert und somit nur innerhalb der Prozesse sichtbar und manipulierbar, wo sie definiert sind (Tabelle II). Im Namenskonflikt-Fall, d.h., wenn lokale Variable in verschiedenen Prozessen mit gleichen Namen definiert werden, gibt es keine Probleme: Die Variablen sind verschieden und eindeutig in ihrem Gültigkeitsbereich festgelegt. CHILL hat zusätzlich noch Parametertransfer-Möglichkeiten: Formale Parameter in der Prozessdefinition werden durch aktuale Parameter beim Start ersetzt. Das gibt die Möglichkeit, *innerhalb des Prozesses* die Parameter-Variable abzufragen und dadurch festzustellen, in welcher «Inkarnation man gerade ist».

Globale Variable dienen zur Kommunikation von Prozessen untereinander. Sie dürfen nicht einfach zugegriffen werden, da es bei parallelem Zugriff zu Konflikten kommen kann: Lesende und schreibende Prozesse können sich gegenseitig störend beeinflussen. Es muss daher Schutzmechanismen für globale Daten geben, sog. «locks». Grundsätzlich gibt es zwei Arten von Lock-Philosophien:

– *Codelocking,* entwickelt von *Dijkstra* [9] durch seine Semaphorentheorie (Fig. 5). Wenn im Code eines Prozesses auf globale Daten zugegriffen wird, wird ein Flag (Semaphor) gesetzt, sodass im kritischen Pfad Exklusivität garantiert ist.

– *Datalocking,* entwickelt von *Hoare* [10] durch seine Monitortheorie (Fig. 6). Wenn im Code eines Prozesses auf globale Daten zugegriffen wird, dann muss eine Zugriffssubroutine aufgerufen werden, die mit den globalen Daten in einem speziellen Softwarebereich abgelegt ist (dem «Monitor» bei *Hoare*). Das Laufzeitsystem sorgt dafür, dass die Subroutinen im Monitor sequentiell durchlaufen werden, so dass immer nur ein Prozess exklusiv auf globale Daten zugreift.

Ein gestarteter Prozess (also im Zustand «running») kann sowohl beim Testen des Dijkstraschen Semaphors P(S) als auch beim Aufruf (CALL) von Zugriffssubroutinen im Hoareschen Monitor angehalten werden (also im Zustand «blocked» sein), weil ein paralleler Prozess auf die globalen Daten zugreift. Später, wenn kein paralleler Prozess mit höherer Priorität mehr auf die globalen Daten zugreift, wird der blockierte Prozess wieder frei und kann weiterlaufend auf die globalen Daten zugreifen.

Prozesse haben somit Zustände (Status, state): Sie sind *nicht aktiv* (noch «schlafend», dormant), werden durch den Start *aktiv* («laufend», running), und können durch die Semaphorenoperation P(S) (Test & Set Flag) bzw. durch das CALL einer Zugriffssubroutine *blockiert* werden. Das typische Zustandsübergangsdiagramm eines Prozesses bzw. einer Prozessinkarnation ist in Figur 7 dargestellt. Ein Nachbarprozess gibt die Blockade frei durch die Semaphorenoperation V(S) (Clear Flag) bzw. durch RETURN von Zugriffssubroutine.

Bezüglich der erwähnten Eigenschaften der Prozesskommunikation lässt sich in PORTAL, ADA und CHILL folgendes feststellen:

In *PORTAL* ist ausschliesslich das Monitorkonzept von *Hoare* vorgesehen. Der Monitor ist ein spezieller Modul, für welchen das Laufzeitsystem gewährleistet, dass er jeweils nur von einem einzigen Prozess besucht werden kann. Das bedeutet, dass für eine Sammlung von Subroutinen, die im gleichen Monitor deklariert werden, garantiert wird, dass immer nur ein Prozess in einer der Subroutinen aktiv sein kann. Die Verschachtelungen von Monitoren, d.h., die Deklaration eines Monitors innerhalb eines anderen Monitors ist nicht gestattet. Neuere Publikationen [11] zeigen, dass dies auch gar nicht erwünscht ist. Demgegenüber ist es erlaubt, innerhalb eines Monitors Subroutinen aus einem andern Monitor ausführen zu lassen, d.h., der Prozess-«Dämon» springt vom ersten zu einem weiteren Monitor (dynamische Schachtelung). In diesem Fall wird der Monitor, von welchem die fremde Monitor-Zugriffssubroutine aufgerufen wird, ebenfalls als «belegt» gekennzeichnet.

Formal besteht zwischen einem Modul mit Subroutinen- und Prozessdeklarationen fast kein Unterschied; einzig das Verbot der statischen Schachtelung führt zu einem formalen Unterschied.

Ein Monitor hat grundsätzlich zwei Zustände: belegt (active) und frei (inactive). Beim Aufruf einer Monitorsubroutine ist zu unterscheiden, ob der Monitor frei oder belegt ist. Im ersten Fall kann der rufende Prozess sofort in den Monitor eintreten und die Zugriffssubroutine benutzen. Im zweiten Fall wird der Prozess blockiert («ready entry» gesetzt). Er muss auf den Zutritt warten. Es ist implementationsabhängig, welcher der so wartenden Prozesse als erster zum Monitor Zutritt erhält.

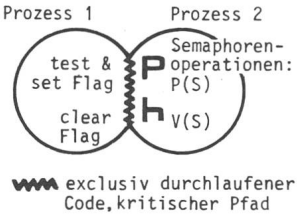

Fig. 5 Codelocking mittels Semaphoren [10]

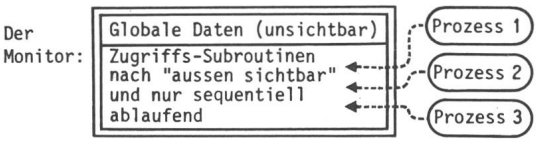

Fig. 6 Datalocking mittels Subroutinen [11]

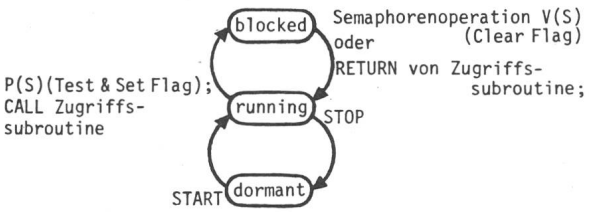

Fig. 7 Zustandsübergangsdiagramm eines Prozesses

Zwischen Aufruf und Verlassen des Monitors kann der Prozess durch ein «*wait*» blockiert werden und so den Monitor wieder freigeben. Ein anderer Prozess kann den so blockierten Prozess durch ein «*send*» wieder wecken. Aber wohlgemerkt, «wait» und «send» (vom Systemtyp «*signal*») dürfen nur in Monitor-Zugriffssubroutinen verwendet werden und nicht im Prozesscode direkt! «Wait» und «send» bilden «extraterritoriale» Punkte im Monitor, indem der eine solche Zugriffssubroutine aufrufende Prozess den Monitor zwangsweise verlassen muss.

CHILL enthält ebenfalls das Hoaresche Monitorkonzept, aber zum Unterschied von PORTAL ist es eine unter mehreren Möglichkeiten, Prozesse kommunizieren zu lassen. Statt Monitor wird in CHILL «region» gesagt, statt «wait» heisst es «delay», für «send» wird «continue» verwendet. Delay und continue wirken auf eine Variable vom Datentyp «event». Abgesehen von neuen Namen sind alle Ideen des Hoareschen Monitorkonzepts übernommen.

CHILL hat noch ein luxuriöseres Monitorkonzept in seiner Sprache implementiert: Für Prozesse sichtbare globale Daten können als Warteschlange (FIFO) deklariert werden. Prozesse können mit «send» lokale Werte in die Warteschlange einreihen bzw. mit «receive» Werte holen. Die Warteschlange erhält einfach den Datentyp «buffer», wobei die Zahl der Einträge definiert werden muss. Blockierungen gibt es beim Senden in die Warteschlange, wenn diese voll ist (Overflow) oder beim Empfangen, wenn sie leer ist (Underflow). Ein so konzipierter Monitor wirkt wie ein Briefkasten (Fig. 8).

ADA kennt das Monitorkonzept von *Hoare* nicht in seinem Sprachkonzept. Keine der hier diskutierten Sprachen hat das Semaphorenkonzept von *Dijkstra* in ihrem Sprachschatz. Aber der Anwender von CHILL und PORTAL kann z. B. mit relativ geringem Aufwand Semaphore in einem Monitor selbst definieren und das Flagtesten, -setzen und -löschen über die Zugriffssubroutinen des Monitors selbst vornehmen. Ebenso ist es relativ einfach, einen Monitor als Warteschlange zu bauen.

ADA hat als einziges Kommunikationsmittel zwischen Prozessen (im ADA-Jargon Tasks) das Rendez-vous-Konzept: Bisher waren alle globalen Daten immer vorhanden, zeitlich wie platzmässig. Bei Dijkstra und Hoare spricht man daher von globalen Daten permanenter Natur. Nun kann man sich auch vorstellen, dass globale Daten von *transienter Natur* sind, dann spart man sich alle Philosophien des Code- und/oder Datalockings. Nur während des Rendez-vous-Vorgangs zwischen Sender und Empfänger sind globale Daten vorhanden. Genauer: Lokale Daten kommen in einen Sendepuffer, dann erfolgt das Rendez-vous mit dem Empfänger, die Empfangs-Task schliesslich findet die Nachricht im Empfangspuffer und kann die Werte in ihren lokalen Datenbereich umschaufeln.

Dieses einfache Nachrichtenübermitteln ist in ADA noch etwas ausgebaut worden. Ein Rendez-vous ist eine asymmetrische Verbindung aufeinander wartender Sende- und Empfangs-Tasks. Die Sende-Task gibt einen Wunsch aus, einen «Eintritt» in die Empfangs-Task zu machen (a request to an «entry», entry call). Die Empfangs-Task erlaubt dies, wenn sie bereit ist, den Wunsch zu akzeptieren (accept the request) (Fig. 9). Der Code zwischen «accept do ... end accept» hat grosse Ähnlichkeiten mit einer Subroutinendefinition. Der entry call dazu hat die Form eines Subroutinenaufrufs. Die Formalparameter, je nachdem ob sie Eingangs- oder Ausgangsparameter sind, erlauben Nachrichtenübermittlung in beiden Richtungen. Eine «select»-Anweisung ermöglicht noch die Unterscheidung mehrerer entry calls. Zusätzlich ist wahlweise eine Zeitüberwachung vorgesehen, aber nur auf der Empfangs-Task-Seite.

CHILL weist diese Rendez-vous-Technik ebenfalls auf, sogar noch subtiler: In CHILL wird diese Technik das Signalkonzept genannt und ist dafür vorgesehen, von einer beliebigen Inkarnation eines Prozesses zu einer beliebigen Inkarnation eines anderen Prozesses einen Datenaustausch vornehmen zu lassen. Zu diesem Zweck wird nicht nur der Name des Zielprozesses angegeben, sondern auch Zusatzinformation für die Inkarnation (Fig. 10). Ein Puffer «MESSAGE» wird definiert, der eine ganze Zahl, einen Charakter (Buchstabe, Ziffer, Sonderzeichen) und eine boolsche Variable (true oder false) enthält. Die Zahl im Beispiel von Figur 10 (nämlich 5) könnte bedeuten, dass man die 5. Inkarnation des Empfangsprozesses, der im «receive case» wartet, herauskippen und aktivieren will. «Receive case» ermöglicht natürlich auch ein Warten auf andere Signale, die nicht vom Typ MESSAGE sind. Mittels dieser Rendez-vous-Technik lassen sich recht einfach *Dijkstras* Semaphoren bzw. Codelocking implementieren [12].

PORTAL kennt keine globalen Daten transienter Natur.

Zusammenfassend kann ganz allgemein gesagt werden: Prozesskommunikation über permanente, globale Daten, die

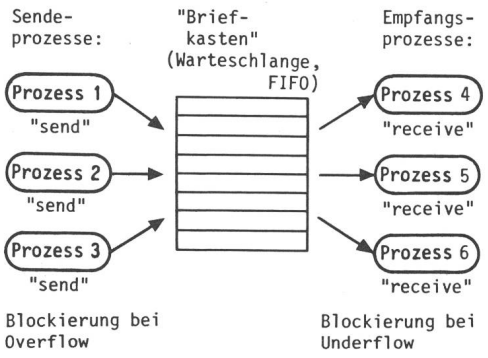

Fig. 8 Kommunikation von Prozessen mittels Warteschlangen (FIFO)

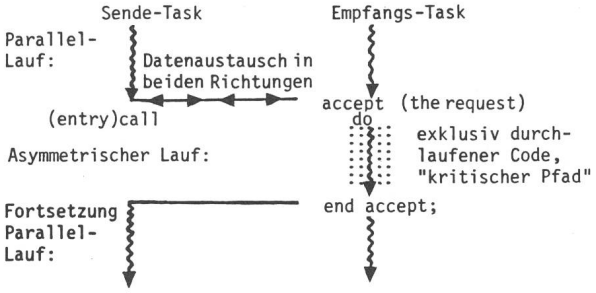

Fig. 9 Rendez-vous in ADA

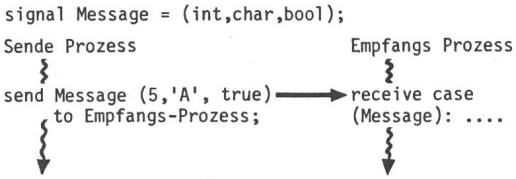

Fig. 10 Signalkonzept in CHILL

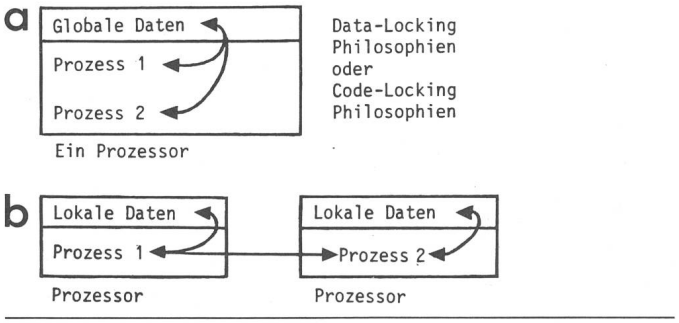

Fig. 11 Prozesskommunikation im Ein- und Mehrprozessor-System

durch Code- oder Datalocking geschützt sind, ist besonders im Einprozessor-System angebracht (Fig. 11a). Prozesskommunikation über transiente, globale Daten ist geeignet für Mehr-Prozessor-Systeme (Fig. 11b).

6. Input/Output-Funktionen sowie hardwarenahe Schnittstellen für «Non-Standard»-Interrupts

Die bisherigen Ausführungen befassen sich gar nicht mit der Hardware, speziell mit Standard- und Nichtstandard-Peripherie. Es wurde lediglich mit abstrakten Konzepten wie Prozessen, Monitoren, Semaphoren usw. gearbeitet.

Für das Erfassen von Interrupts oder das Lesen von Signalen mit absoluten (Hardware) Adressen durch einen Polling-Prozess ist weder PORTAL noch ADA noch CHILL vorbereitet, da man die Ansicht vertritt, solche maschinenabhängigen Funktionen seien als Codefunktionen und -prozeduren zu implementieren. Beispiele:

In CHILL wird empfohlen, jedoch nicht normiert, für Input/Output folgende Funktionen zu gebrauchen:
inint(); inbool(); inchar();
für: «lese einen integer, boolean, character Wert», oder
outint(); outbool(); outchar();
für: «schreibe einen integer, boolean, character Wert», vermutlich auf der Konsole.

Entsprechend sind Input/Output-Funktionen für Nichtstandard-Peripherie zu bilden. Diese Input/Output-Funktionen werden Prozesse starten, die die Peripherie-Geräte bedienen. Werden mehrere Peripherie-Geräte gleichen Typs von einem Prozess bedient, werden Inkarnationen (in CHILL) bzw. generic tasks oder Task-Familien (in ADA) installiert werden müssen.

Der Datenverkehr mit peripheren Geräten geschieht über Geräteregister (I/O-Fenster, Kommunikationsregister usw.). Diese werden als Variable deklariert und behandelt. Prozesse, die periphere Geräte bedienen, werden in «wait»-Anweisungen (PORTAL) auf ein externes Signal warten. Klassische Gerätetreiber oder Händler werden also als Prozesse implementiert, die von der Hardware her synchronisiert werden. Sind die Ereignisse dieser sehr maschinenabhängigen Funktionen aber einmal erfasst, so können sie einfach an den Anwenderprozess weitergeleitet werden.

Ähnliches geschieht mit Zeitfunktionen. Nur ADA hat eine Zeitfunktion definiert im Sprachkonzept: Accept (a request) beim Rendez-vous kann durch ein Zeitglied überwacht werden (delay time). Ansonsten werden sehr anwendungsspezifisch Funktionen für die Uhr zu definieren sein.

In allen Fällen jedoch wird in PORTAL, ADA und CHILL der Gebrauch von Assemblersprache und Kenntnisse von speziellen Betriebssystemfunktionen, die der Computer-Hersteller definiert, überflüssig sein, wenn die Sprachen für ein Hardwarekonzept implementiert sind. Es genügt, das Sprachkonzept und eine Funktionsbibliothek zu kennen, mit der der Anwender sehr benutzerfreundlich arbeiten kann. Nun, das ist hoffentlich nicht mehr lange «Zukunftsmusik» für jedermann.

Literatur

[1] *F.P. Brooks:* The mythical man-month. Reading – Mass. a.o., Addison Wesley, 1975, p. 94.
[2] *K. Jensen* and *N. Wirth:* Pascal; user manual and report. New York a.o., Springer study edition, 1978, ed. 2.
[3] *H.W. Wippermann:* Pascal; Einsatz in der Ausbildung; Implementierung der Sprache; Verwendung auf Kleinrechnern. 3. Workshop und Treffen der German Chapter of the ACM, Kaiserslautern, 14. und 15. Oktober 1977. München und Wien, C. Hanser Verlag, 1978. Applied computer science Band 11.
[4] *H.H. Nägeli:* Programmieren mit Portal. Zürich, Institut für Informatik an der ETH, 1979.
[5] The brown document. Comité Consultatif International Télégraphique et Téléphonique (CCITT), February 1980, paper SG XI/Control number 379.
[6] Introduction to Chill. Comité Consultatif International Télégraphique et Téléphonique (CCITT), May 1980, paper SG XI/3–2.
[7] Institut National de Recherche en Informatique et en Automatique (INRIA): Formal definition of the Ada programming language. Domaine de Voluceau, Rocquencourt, 78153 Le Chesnay, France, November 1980.
[8] *D. Profos:* Programmiersprachen für die Nachrichtentechnik, Nachrichtentechnisches Kolloquium der Universität Bern 1980/81. Bern, Institut für angewandte Mathematik und angewandte Physik der Universität, 1981.
[9] *E.W. Dijkstra:* Co-operating sequential processes. Out of 'Programming languages', lectures given in Villard-de-Lans, edited by F. Genuys. London & New York, Academic Press, 1968, p. 43...112.
[10] *C.A.R. Hoare:* Monitors: an operating system structuring concept. Communications of the Association for Computing Machinery 17(1974)10, p. 549 to 557.
[11] *F. Pieper:* Concurrent Pascal – eine Kritik. Elektronische Rechenanlagen mit Computerpraxis 23(1981)3, S. 115...121.
[12] *P. Wegner:* Programming with Ada; an introduction by means of graduated examples. Englewood Cliffs – N.J., Prentice-Hall, 1980, p. 176.

Adresse des Autors

Dr. *H. Zwittlinger,* Ingenieurschule Bern HTL und Software-Schule Schweiz, Morgartenstrasse 2c, 3014 Bern.