PEARL Rundschau

Inhalt

Juni 1981 Band 2 Nr. 2

PEARL Rundschau

Juni 1981
Band 2
Nr. 2

Der PEARL-Verein e.V. (PEARL-Association) hat das Ziel, die Verbreitung der Realzeitprogrammiersprache PEARL (Process and Experiment Automation Realtime Language) und ihre Anwendung sowie die Einheitlichkeit von PEARL-Programmiersystemen zu fördern.
Sitz des Vereins ist Düsseldorf. Seine Geschäftsstelle befindet sich im VDI-Haus, Graf-Recke-Straße 84, 4000 Düsseldorf 1.

Vorstand des PEARL-Vereins:

Prof. Dr.-Ing. R. Lauber Institut für Regelungstechnik und Prozeßautomatisierung Seidenstraße 36 7000 Stuttgart 1	Vorsitz, zuständig für Technik und Normung
Dipl.-Ing. G. Müller Brown Boverie und Cie. AG Leiter der Vertriebsabteilung Netzführungssysteme, SI/NV 1 Fred-Joachim-Schoeps-Str. 55 6800 Mannheim	stellv. Vorsitz zuständig für Organisation und Finanzen
Dr.-Ing. P. Elzer DORNIER System GmbH Abt. EEA Postfach 1360 7990 Friedrichshafen	zuständig für Öffentlichkeitsarbeit und Marketing

Herstellung durch die VDI-Verlag GmbH, Düsseldorf

Die PEARL-Rundschau ist das Mitteilungsblatt des PEARL-Vereins e.V. Neben Vereinsangelegenheiten werden für alle PEARL-Interessenten Informationen über Erfahrungen, Anwendungsmöglichkeiten und Produkte gegeben.
Preis des Einzelheftes für Nichtmitglieder: DM 15,—, für Studenten DM 5,—. Jahresabonnement DM 75,—.
Zur Veröffentlichung werden sowohl fachliche Abhandlungen über Realzeit-Anwendungen, als auch Kurznachrichten zu Themen des Prozessrechnereinsatzes und der Verwendung höherer Sprachen angenommen, soweit sie für PEARL-Interessenten von Bedeutung sein können.

Es obliegt dem Autor, die Rechte zur Veröffentlichung seines Beitrags in der PEARL-Rundschau sicherzustellen. Der PEARL-Verein erhebt keine Einwände gegen die Kopie einzelner Beiträge bei Angabe der Quelle.

Beiträge können jederzeit, auch unaufgefordert, an ein Mitglied des Redaktionskollegiums geschickt werden. Autoren werden gebeten, bei der Schriftleitung ein Info-Blatt zur Manuskriptgestaltung anzufordern.

Vorwort der Schriftleitung

Zum äußeren Erscheinungsbild der PEARL-Rundschau gibt es diesmal nichts Neues zu sagen. Wir hoffen nur, daß die Anzahl der Tipp-,Druck- und sonstigen Fehlerteufel allmählich gegen Null konvergiert.

Zum Inhalt: Zunächst bringen wir noch drei Beiträge zum Thema PEARL-Implementierungen als Nachträge zum letzten Heft.

Als neues Thema bringen wir in diesem Heft Vergleiche von PEARL mit anderen Programmiersprachen, um damit einem häufig geäußerten Wunsch von Mitgliedern des PEARL-Vereins nach "Argumentationshilfen" nachzukommen. PEARL existiert ja nicht im luftleeren Raum und hat auch keine Monopolstellung (wie manche Programmiersprachen in anderen Ländern). Also müssen bei vielen Einsätzen von PEARL - vor allem in neuen Anwendungsgebieten - die Einsatzwirksamkeit durchdacht und die Vorteile gegenüber anderen Programmiersprachen neu diskutiert werden. Deshalb werden Sprachvergleiche benötigt. Daß manche Fachkollegen, wie z.B. auch der Unterzeichnende, solchen Vergleichen skeptisch gegenüberstehen, tut ihrer allgemeinen Nützlichkeit wenig Abbruch. Die Skepsis kommt vielleicht auch daher, daß es schon zu viele davon gibt.

Die Diskussion um den Wert einer Programmiersprache findet auf sehr verschiedenen Ebenen und in sehr unterschiedlichen technischen Detaillierungsgraden statt. Dem entspricht auch die Zusammenstellung dieses Heftes, in dem eine Reihe von Vergleichen sehr unterschiedlichen Charakters für die Leser der PEARL-Rundschau zusammengestellt wurde. Da Selbstkritik etwas sehr Nützliches ist und zumindest zur Verbesserung der eigenen Argumente, wenn nicht gar der Produkte beiträgt, wurde auch ein Artikel aufgenommen, bei dem PEARL nicht so besonders gut abschneidet.

Es wäre interessant, zu erfahren - am besten durch Leserbriefe was unsere Leser davon halten.

Was die nächsten Nummern angeht, so hoffen wir, damit wieder zwei alte Wünsche befriedigen zu können:
Erstens nach einer allgemeinen verständlichen Einführung in die Normungstechnik von PEARL. Herr Prof. Dr. Rzehak, der Vorsitzende des DIN UA 5.8, des für PEARL zuständige Normungsgremiums, hat dankenswerterweise die Aufgabe übernommen, zu diesem Thema ein Heft als "guest-editor" zusammenzustellen.

Zweitens nach einer Ausgabe für englischsprachige Interessenten an PEARL. Deshalb soll in nächster Zeit eine Zusammenfassung von Übersetzungen ausgewählter Artikel aus der "Rundschau" als Sonderheft herauskommen.

Für die Schriftleitung

Mit freundlichen Grüßen

P.Elzer

PDP-11 PEARL, die PEARL Implementierung von DIGITAL EQUIPMENT

E. Mezera, T. Siebold

1. ZUSAMMENFASSUNG

Der vorliegende Aufsatz gibt einen Überblick über PDP-11 PEARL. PDP-11 PEARL ist ein Standard-Software-Produkt von Digital Equipment, verfügbar für die Rechner der PDP-11 Familie. Weltweit wurden bis heute ca. 150 000 Rechner der PDP-11 Familie installiert, davon ca. 9000 in der Bundesrepublik Deutschland.

PDP-11 PEARL liegt im Sprachumfang nahe an Full-PEARL. Der Compiler ist in das Standard-Betriebssystem RSX-11M eingebettet und wird wie jeder andere Compiler unter diesem System bedient. RSX-11M ist ein Mehrbenutzersystem mit Echtzeiteigenschaften. Das PEARL-Laufzeitsystem stützt sich darauf ab.

Weitere Informationen sind aus der angegebenen Dokumentation zu entnehmen bzw. können von den Autoren erhalten werden.

2. SPRACHUMFANG

Um dem Benutzer ein leistungsfähiges Werkzeug zur Programmerstellung zur Verfügung zu stellen, war es von Anfang an ein Entwicklungsziel für PDP-11 PEARL, eine möglichst reichhaltige Implementierung nahe an Full-PEARL zu realisieren. Dabei wurde beachtet, daß nur ein Subset von Full-PEARL in Frage kommt. Ebenso mußten bestehende Normentwürfe (DIN 66253, Teil 2, Full-PEARL /2/) und Vornormen (DIN 66253, Teil 1, Basic-PEARL /1/), die sich z.T. während der Compilerentwicklung noch selbst in Entwicklung befanden, beachtet werden.

Das Ergebnis ist der implementierte Sprachumfang PDP-11 PEARL, ausführlich beschrieben im PDP-11 PEARL Language Reference Manual /3/.

Gegenüber Full-PEARL sind folgende Einschränkungen vorhanden:

- keine Interruptkanäle in vom Benutzer selbst formulierten Interfaces
- Funktion ORIGIN nicht implementiert
- gewisse Einschränkungen bei Arrays und Structures
- bei Zeichen- und Bitkettenausschnitten nur Konstanten als Indizes.

Folgende wichtige Sprachelemente gehören unter vielen anderen zum Umfang von PDP-11 PEARL:

- Benutzerdefinierte Interfaces
- Benutzerdefinierte Operatoren
- Synchronisationsobjekte vom Typ BOLT und SEMA
- Dynamische Änderung von Taskprioritäten
- Listen von Semaphoren in Semaphoranweisungen
- Referenzobjekte
- Interruptanweisung TRIGGER
- Signalanweisung INDUCE
- Einplanungslisten bei allen Taskanweisungen.

Für weitere realisierte Sprachmittel sei auf die Sprachbeschreibung /3/ verwiesen.

3. COMPILER

Der Compiler ist eingebettet in das Standard-Betriebssystem RSX-11M von Digital Equipment. Er verarbeitet Quellenfiles, die mit dem Standard-editierprogramm (EDI, EDT) des Systems erzeugt werden. Die Ausgabe des Compilers sind Objektmodule, die mit dem Standard-binderprogramm des Systems zu ablauffähigen Programmen zusammengebunden werden. Dabei werden auch die benötigten Laufzeitroutinen angeschlossen. Der vom Compiler erzeugte Code, sowie auch die Laufzeitroutinen sind "REENTRANT". In PEARL-Programme können Unterroutinen, die in anderen Sprachen entwickelt wurden (z.B. Assembler, FORTRAN) über die Standardschnittstelle eingebunden werden.

Der Compiler erzeugt während des Übersetzungsvorganges ein Übersetzungsprotokoll (Programmlisting), das aus folgenden Komponenten besteht:

- Auflistung des Quellprogrammes mit Zeilennummer
- Durchnummerierung der Blöcke
- Kennzeichnung der Blockschachtelungstiefe
- Zuordnung zwischen Quellzeilen und Instruktionsadresse
- Absätze im Listing bei Modul-, Task- und Prozedurbeginn
- Fehlermeldungen kennzeichnen den Fehlerort in der Quellzeile
- Angaben über Speicherbedarf für Code und Daten.

Die Fehlerprüfung des Compilers führt abgesehen von schwerwiegenden Fällen nicht zum Abbruch des Übersetzungsvorganges.

4. PEARL-LAUFZEITSYSTEM

Das Laufzeitsystem übernimmt während des Programmlaufs die Steuerung und Überwachung der PEARL-Tasks, Verwaltung der Einplanungen, Verwaltung von Semaphoren und Bolts, Kontrolle über Interrupts und Signale und die Überwachung der Transferoperationen. Es stützt sich dabei auf vorhandene System-Service-Routinen des Betriebssystems RSX-11M und ergänzt diese hinsichtlich eines optimalen Ablaufs von PEARL-Programmen. Ebenso enthält das Laufzeitsystem die Standardprozeduren sowie die Interfaces zu den Peripheriegeräten und zur Dateiverwaltung von RSX-11M.

Diese Funktionen werden in einer modular aufgebauten Laufzeitbibliothek realisiert. Beim Binden des ablauffähigen PEARL-Programmes werden nur die unmittelbar benötigten Routinen automatisch aus der Bibliothek herangezogen. Der Speicherplatzbedarf ist also abhängig von den benutzten Sprachelementen.

5. HARDWARE- UND SOFTWAREVORAUSETZUNGEN

Sowohl zum Ablauf des PEARL-Übersetzungssystems als auch zum Ablauf von PEARL-Programmen wird eine RSX-11M Systemkonfiguration benötigt. RSX-11M Systemkonfigurationen können mit Zentraleinheiten der Modelle PDP-11/23, PDP-11/34, PDP-11/44, PDP-11/60 und PDP-11/70 realisiert werden. Die Zentraleinheiten müssen mit der Floating-Point Processor Option ausgestattet sein. Der Mindestspeicherausbau beträgt 128k byte. Als Hintergrundspeicher ist ein breites Spektrum von Plattenspeichergeräten möglich:
RK05, RL01, RL02, RK07, RM02, RM03, RM05, RP04 und andere.
Diese Plattengeräte unterscheiden sich in Aufbau, Speicherkapazität und Zugriffsgeschwindigkeit.
Um das PEARL-System auf einem RSX-11M System installieren zu können, muß eines der folgenden Peripheriegeräte angeschlossen sein:
entweder 9-Spur Magnetband oder RK05 Plattengerät oder RL01 Plattengerät.

Das RSX-11M Betriebssystem ist ein Echtzeitbetriebssystem mit Programmentwicklungsmöglichkeiten, dynamischer Hauptspeicherverwaltung und Mehrbenutzer-

betrieb. Dieses leistungsfähige und flexible Betriebssystem wird hauptsächlich dann verwendet, wenn mehrere Benutzer gleichzeitig unabhängig voneinander Echtzeit- und Auswertungsprogramme erstellen, testen und ausführen wollen. Neben in PEARL realisierten Programmen können unter diesen Betriebssystemen ebenso Programme, die in anderen Sprachen, wie Assembler oder FORTRAN entwickelt wurden, zeitlich parallel zum Ablauf gebracht werden.

6. DOKUMENTATION

Folgende Dokumentation ist erhältlich:

PDP-11 PEARL Language Reference Manual /3/ als Beschreibung des implementierten Sprachumfanges.

PDP-11 PEARL User's Guide /4/ enthält die Beschreibung der Zusammenarbeit des PEARL-Systems mit dem Betriebssystem RSX-11M; insbesonders Systemteil, Standardsignale, Ein-/Ausgabemöglichkeiten, Besonderheiten.

7. SCHULUNG

PEARL-Kurse werden regelmäßig im Schulungszentrum der Digital Equipment GmbH., München durchgeführt. Auf Anfrage und bei hinreichend hoher Teilnehmerzahl führt die Schulungsabteilung auch Kurse vor Ort am Einsatzort des Kunden durch.

8. WEITERE PEARL-ENTWICKLUNGEN BEI DIGITAL EQUIPMENT

Zur Zeit wird bei Digital Equipment gerade ein PEARL-Compiler und Laufzeitsystem entwickelt und getestet, das auf den 32-Bit Rechnern der Serie VAX-11/780 und VAX-11/750 ablaufen wird. Die offizielle Produktankündigung wird in Kürze erfolgen, mit der Freigabe des Produktes ist im Herbst 1981 zu rechnen.

Literatur:

/1/ Vornorm DIN 66253 Teil 1
Informationsverarbeitung
Programmiersprache PEARL
Basic PEARL
Beuth Verlag, Berlin, Köln 1981

/2/ Norm-Entwurf DIN 66253 Teil 2
Informationsverarbeitung
Programmiersprache PEARL
Full-PEARL
Beuth Verlag, Berlin, Köln, 1980

/3/ PDP-11 PEARL
Language Reference Manual
Order no.: AA-J750A-TC, Sept.1980
Digital Equipment Corporation,
Maynard, Massachusetts

/4/ PDP-11 PEARL
User's Guide
Order no.: AA-J571A-TC, Sept.1980
Digital Equipment Corporation,
Maynard, Massachusetts

Anschrift der Autoren:

Ernst Mezera, Thomas Siebold
beide

DIGITAL EQUIPMENT GMBH.
Hauptverwaltung
Freischützstraße 91
8000 München 81

mbp PEARL Programmiersystem für SIEMENS-Prozeßrechner der Serie 300

K.-W. Müller

1. Allgemeines

Ziel der mbp-PEARL-Implementierung, die anläßlich der Hannover Messe 1981 erstmals öffentlich vorgestellt wurde, war ein möglichst großer Sprachumfang, der entweder abgestuft mit verschiedenen Testinformationen und dynamischen Kontrollen oder im Hinblick auf Speicher- und Rechenzeiteffizienz übersetzt werden kann.

Ausgangspunkt war eine inzwischen mehrmals überarbeitete portable Compiler- und Laufzeitsystemimplementierung, wie sie auch für Siemens 404/3 verwendet wurde /1, 2/. Die auf Übertragbarkeit gerichtete Konzeption einerseits und die Ähnlichkeit der Maschinencodes andererseits erleichterten eine Anpassung an die verschiedenen 300-er Systeme.

2. Sprachumfang

Für eine anwendungsfreundliche und dennoch effiziente Programmierung reicht der durch die DIN-Norm 66253 vorgegebene Sprachumfang von Basic PEARL nicht aus.
Der Sprachumfang von mbp-PEARL ist daher um wesentliche Elemente von Full PEARL erweitert worden wie

* Referenzen (REFERENCE, CONT)
* Synchronisation über BOLT-Variable
* benutzerdefinierte Datentypen (TYPE)
* benutzerdefinierte OPERATOREn mit Angabe von PRECEDENCE
* Strukturen (STRUCTURE)
  - wobei Subkomponenten Felder oder Strukturen sein dürfen
  - Subkomponenten verschiedener Strukturen gleiche Namen haben können
  - Strukturen auch Resultat von Funktionen sein dürfen
* Felder mit
  - mehr als drei Dimensionen
  - statische Initialisierung mit INIT
  - untere Dimensionsangabe auch negativ
* Felder auch von REF, SEMA, BOLT, TYPE und DATION-Objekten
* Mehrfachzuweisungen
* variable Sub-Strings von Typ BIT und CHARACTER
* Prozeduren auf TASK-Ebene
* Listen von SEMA und BOLT
* SEMA, BOLT, REF und neue Datentypen (TYPE) als Prozedurparameter
* REF, STRUCTURE und neue Datentypen (TYPE) als RETURN-Attribut
* UPB, LWB auch monadisch
* bedingte Zuweisung
* ACTIVATE, CONTINUE mit Prioritätsangabe
* SUSPEND auch auf fremde Task
* TRIGGER
* OPEN, CLOSE mit CAN und PRM

Nähere Einzelheiten sind dem mbp-PEARL-Sprachreport /4/ zu entnehmen, der neben einer informellen Beschreibung anhand von Syntaxdiagrammen (Wirth) und vielen Beispielen auch eine vollständige Syntaxbeschreibung in Backus-Naur-Form enthält.

3. Hardware Voraussetzungen

Der Compiler und die erzeugten PEARL-Programme sind ablauffähig auf den Rechnertypen

330 mit SIM30R
R10
R20
R30

unter den Betriebssystemen

ORG 330 K
ORG 300 P
ORG 300 PV

Für den Compiler ist ein Laufbereich von 32K Worten und ein Plattenspeicher erforderlich.
Die Länge des Laufbereichs für PEARL-Programme beträgt, abhängig von den benötigten Laufzeitfunktionen, im allgemeinen mehr als 32K.

4. Compiler

Der Compiler entspricht in seiner Bedienung dem anderer Compiler auf Siemens-300-Rechnern, ist also auch monitorfähig.

Während der Übersetzung entstehen wahlweise folgende Listen

- Quellprogramm-Liste mit zusätzlichen Angaben über Anweisungsnummer und Schachtelungstiefe
- Cross-Referenz-Liste
- Fehlerliste mit Fehlerklassifizierung
- Zwischensprachenausgabe für Demonstrations- und Testzwecke

Durch spezielle Compiler-Bedienung ist es möglich, entweder optimalen Laufzeitcode oder Code zur Anwendung der Testhilfen zu erzeugen. Der Compiler erzeugt Assemblercode (ASS 300), der entweder in einer Quellbibliothek abgelegt oder ausgedruckt werden kann.
Dadurch sind auch alle für den Assemblerprogrammierer vorhandenen Hilfsmittel verwendbar, beispielsweise Übersetzung für eine andere Anlage und anschließende Übertragung erzeugter Bibliothekselemente, Binde- und Lademechanismen. Das PEARL-Programmiersystem paßt sich so in bereits vorhandene Systemteile ein. Der auf Effizienz bedachte Programmierer kann im generierten Assemblercode die Realisierung verschiedener PEARL-Sprachelemente studieren und wird feststellen, daß fast optimaler und sehr effizienter Code erzeugt wird.

Es ist jedoch nicht erforderlich, daß man beim Testen und Verfolgen von Programmierfehlern diesen Assemblercode jemals zur Kenntnis nimmt.

Zusätzlich enthält der Compiler noch einen Preprocessor, der folgende Aufgaben erfüllen kann

- Ausgabesteuerung (Seitenvorschub, teilweise Protokoll-Unterdrückung)
- bedingte Übersetzung
- Einkopieren von Quellteilen (auch aus einer anderen Bibliothek)
- Testunterstützung

5. PEARL-Betriebssystem

Die Implementierung der Realzeitfunktionen ist weitgehend unabhängig von dem zugrundeliegenden (Gast-) Betriebssystem. Das wird erreicht durch Verwendung einer portablen Betriebssystem-Schnittstelle /3/ .

Die Anpassung an das jeweilige Betriebssystem muß dann nur noch an fünf genau definierten Punkten hergestellt werden:

- Reaktion auf Programmunterbrechungen
- Wechsel des Prozessors von einem Rechenprozeß zum anderen
- Ablesen der Uhr und Behandlung des Taktgebers
- Ein/Ausgabe von und zu den verschiedenen Peripheriegeräten
- Datenhaltung

Eine wesentliche Eigenschaft dieser Betriebssystem-Schnittstelle ist die Tatsache, daß alle PEARL-TASKs in einem einzigen Programm vom PEARL-System so im Multiplex betrieben werden, daß alle Spracheigenschaften des PEARL-TASKing genau erfüllt sind und die Zahl möglicher TASKs nicht durch noch freie Programmnummern im (Gast-) Betriebssystem beschränkt ist.

Das PEARL-System hat damit die parallel arbeitenden TASKs besser unter Kontrolle, als wenn sie unmittelbar unter Regie des Betriebssystems laufen würden. Besonders wichtig ist dies in irregulären Fällen wie TASK-Abbruch mit Betriebsmittelfreigabe oder bei Fehlerrückverfolgung.

6. Testunterstützung

Durch Optionen bei der Übersetzung kann die Codegenerierung so beeinflußt werden, daß ein Test auf Quellsprach-Ebene möglich wird. Die entsprechenden Testanweisungen können über Preprocessor-Befehle in der Quelle aktiviert werden. Folgende Testmöglichkeiten sind gegeben

- Backtrace im Fall von Laufzeitfehlern
- Protokollierung der Aufrufe an das PEARL-Betriebssystem
- Ausgabe der Taskzustände
- Auftragsgesteuerter Zeilentrace (s.u.)
- vollständiger Zeilentrace

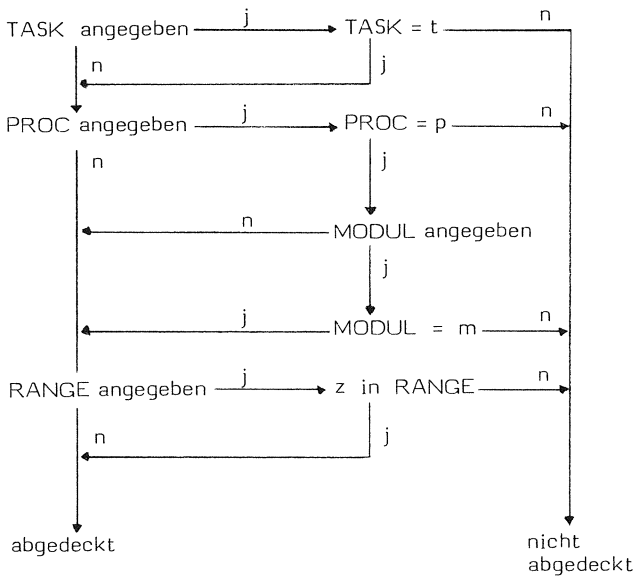
Die Ausgabe kann wahlweise auf einem Drucker oder Sichtgerät erscheinen.

Im Fall eines Auftragstrace werden nur diejenigen PEARL-Zeilen protokolliert, die durch wenigstens einen Auftrag abgedeckt sind.

Wann die aktuelle Zeile durch einen Auftrag abgedeckt wird, zeigt das nachfolgende Diagramm. Es bedeutet

t	aktueller Taskname, leer im Initvorlauf
p	aktuelle Procedure, leer wenn keine
m	Modul der Procedure p enthält
z	aktuelle Zeilennummer

TASK	Taskname	aus Auftrag
PROC	Procedurename	
MODUL	Modulname	
RANGE	Zeilenbereich	



Eine Tracezeile enthält folgende Information

- Name der Task, unter deren Kontrolle die Zeile durchlaufen wurde
- Zeilennummer
- Name der Prozedur
- Name des Moduls

Die Testschnittstelle ist dem Benutzer zugänglich. Er kann selbst weitere Testfunktionen implementieren.

Literaturhinweise

/1/ Allgemeine Informationen zur mbp-PEARL-Implementierung
PEARL-Rundschau, Band 2, Nr. 1, S. 22-23 (1981)

/2/ Anlage zum Projektbericht
Forschungsvorhaben P4.2/25: M-DVF/1
EBOSIPES

/3/ Echtzeit-Mehrprogramm-Betriebssystem
mbp-interne Veröffentlichung, Oktober 1980

/4/ mbp-PEARL
1 st edition, March 1981

Anschrift:

mbp Mathematischer Beratungs- und
Programmierungsdienst GmbH
Geschäftsstelle Lüneburg
Erbstorfer Landstraße 21
2120 Lüneburg

Ein portables PEARL-Kompiliersystem für den Z80-Mikrorechner

P. Holleczek

Einleitung

Das hier beschriebene PEARL-Compiliersystem wurde innerhalb eines Forschungsprojekts des Regionalen Rechenzentrums Erlangen und des Physikalischen Institut der Universität Erlangen-Nürnberg entwickelt, das eine möglichst leichte Portierbarkeit des Gesamtsystems zum Ziel hat. Es stellt eine fast vollständige Überarbeitung früherer Compiliersysteme des Physikalischen Instituts für die Rechner Siemens 306 [1] und 310 [2] dar. Die Anwendungsschwerpunkte des Compiliersystems sollen in der Labormeßtechnik innerhalb der Universität liegen. Den Gegebenheiten der Universität folgend und angesichts der eingeschränkten Leistungsfähigkeit eines 8 bit-Rechners ist es als Cross-Compiliersystem realisiert, bei dem die Übersetzungen auf dem zentralen Großrechner (hier: CYBER 173) durchgeführt werden und das Zielsystem (hier: Z80) über Rechnerkopplung mit den übersetzten Programmen versorgt wird.

1. Sprachumfang

Der Sprachvorrat [3] ist durch das verwendete, von der Firma ESG, München, entwickelte Compileroberteil vorgegeben. Er stellt eine Untermenge von PEARL Stand '77 dar und entspricht, von einigen syntaktischen Unterschieden abgesehen, BASIC-PEARL. Für die Anwendungen in der Labormeßtechik ist wesentlich, daß er über BASIC-PEARL hinausgehend auch die graphische Ein-/Ausgabe enthält. Im Interesse einer vernünftigen Speicherplatzausnutzung des 8 bit-Rechners werden natürlich immer nur die Teile des Laufzeitsystems zur Verfügung gestellt, die vom jeweiligen Anwenderprogramm angefordert werden.

2. Funktionsweise des Compilersystems und Beschreibung der Komponenten

Der Übersetzungsvorgang besteht aus einer Abfolge des Laufes von Compileroberteil, Codegenerator und Assembler.

Das Compileroberteil übersetzt das Programm aus der Quellsprache in die rechnerunabhängige Zwischensprache CIMIC, einer abstrakten Assemblersprache. Dieser Teil des Compiliersystems ist in ca. 10 Läufe aufgeteilt, was einer Segmentierung für "kleine" Übersetzungsrechner stark entgegenkommt. Die erzeugte CIMIC-Sprache ist streng formatiert, enthält noch die verschiedenen Datentypen und Betriebssystem-Aufrufe, sowie bereits aufgelöste Kontrollstrukturen und Ausdrücke.

Aus CIMIC erzeugt der Codegenerator Assembler-Code. Die Umsetzung erfolgt in zwei Stufen. Die erste Stufe setzt den mnemonischen CIMIC-Code in einen komprimierten Zahlen-Code um, die zweite Stufe erzeugt hieraus Assembler-Code. Die Erzeugung von Assembler-Code hat den Vorteil daß ein Programmtest noch mit symbolischem Code möglich ist, solange noch kein quellsprachebezogenes Testsystem zur Verfügung steht.

Das Laufzeitsystem auf dem Zielrechner ist unterteilt in die Schichten Laufzeitfunktionen, Betriebssystem und Treiber.

Die Laufzeitfunktionen dienen zur Abarbeitung der vom Codegenerator nicht aufgelösten Befehlsfolgen und rufen ggf. das Betriebssystem auf. Sie sind reentrantfähig und erlauben reentrantfähige Anwender-Prozeduren. Laufzeitfunktionen gibt es für die EA-Formate, die Behandlung der PEARL-Datentypen, Filehandling und für die Versorgung des Arithmetik-Prozessors.

Das PEARL-Betriebssystem dient zur Realisierung von Task-Steuerung und Synchronisierung, zur Abarbeitung von Einplanungen und zur Geräteverwaltung. Das Betriebssystem belegt 4 k Byte Speicher.

Treiber gibt es zur Bedienung (Datentransfer, Interruptverarbeitung) der anschließbaren Geräte, wie Display, Drucker, Floppies und die (ECB-) Prozeßperipherie. Ein typischer Treiber für ein Standard-Gerät belegt ca. 200 Byte Speicher. Der Floppy-Treiber enthält das System-Programm BDOS (Speicherbedarf: ca. 5 kB) des Betriebssystems CPM.

Der Ablauf des Übersetzungsvorgangs und der Transfer der übersetzten Programme vom Großrechner zum Zielrechner kann vom Zielrechner aus gesteuert werden. Dies wird dadurch ermöglicht, daß sich der Zielrechner wie ein Terminal des Großrechners verhält und der Anwender alle Großrechner-Dialog-Kommandos am Zielrechner absetzen kann.

3. Übersetzungs- und Zielrechner

Der Übersetzunsrechner ist zur Zeit die CYBER 173 des Regionalen Rechenzentrums Erlangen. Die am Übersetzungsrechner laufenden Programme (Compileroberteil und Codegenerator) sind maschinenunabhängig in FORTRAN IV geschrieben und enthalten eine einheitliche Prozedur-Schnittstelle für die meist nicht maschinenunabhängigen Zugriffe auf den Hintergrundspreicher. Das Compileroberteil läuft außerdem auf einer BS 1000-Anlage. Geplant ist die Übernahme auf eine Siemens R30. Eine Vorversion dieses Compileroberteils lief auf verschiedenen Kleinrechnern in 20 kW Laufbereich.

Der Zielrechner ist ein Z80 A-System vom Typ KBS 10 mit 64 kB Speicher, Arithmetikprozessor AM9511, Floppy-Doppellaufwerk und ECB-Interface für Prozeß-Anschlüsse. Der Übergang zu einem anderen Zielsystem bedingt eine Änderung des Codegenerators und Änderungen des Laufzeitsystems. Im einzelnen bedeutet dies: Der Großteil der Laufzeitfunktionen, nämlich für die EA-Formate, liegt in PEARL selbst vor, muß also nur neu übersetzt werden. Der Rest muß neu erstellt werden. Das Betriebssystem liegt zum Großteil maschinenunabhängig in einem abstrakten Assembler vor und wurde in einer Testversion schon früher auf einem Z80 eingesetzt [4]. Der hierfür nötige Umsetzer ist wiederum in FORTRAN IV geschrieben. Eine Portierung des Betriebssystems erfordert die Änderung eines Passes des Umsetzers und eine Neuerstellung der maschinenabhänigen Routinen (z.B. Registerverwaltung, Geräteschnittstelle). Die Gerätetreiber sind neu zu erstellen. Der gesamte Umstellungsaufwand für eine neue Zielmaschine liegt zwischen 1 und 2 Mannjahren.

Die einzelnen Portierungsverfahren und ihre Anwendungen in früheren Implementationen sind ausführlich in [5] dargestellt.

4. Entwicklungsstand

Die Arbeit am Compiliersystem wurde im Juli 1980 begonnen. Die Komponenten sind fertig entwickelt. Ein Kernsystem (ohne File-Handling) befindet sich im Integrationstest. Geplant sind eine Übernahme des Übersetzungssystems auf eine R30 und die Erstellung eines quellsprachebezogenen Testsystems.

[1] P. Elzer, P. Holleczek: ASME-PEARL-Compiler wird der Öffentlichkeit vorgestellt; Regelungstechnik 1975, S. 433-436

[2] A. Fleischmann et al.: PEARL-Cross-Compiliersystem für den Prozeßrechner Siemens 310; Regelungstechnische Praxis 1979, S. 372

[3] PEARL-Subset für Avionic Applications, Language Description, Elektronik-System-Gesellschaft mbH, München, März 1977

[4] R. Rössler: PEARL-Betriebssystem für den Z80, PDV-Entwicklungsnotiz E 119, Kernforschungszentrum Karlsruhe mbH, 1978

[5] Portable Software, Bericht des German Chapter of the ACM 4, Teubner Verlag, Stuttgart, 1980 (herausgegeben von H.J. Schneider)

Dr. Peter Holleczek
Regionales Rechenzentrum Erlangen
Martensstr. 1
8520 Erlangen

Kurzüberblick über Entwicklung und Eigenschaften von PEARL im Vergleich mit anderen Programmiersprachen

P. Elzer

1. Einleitung

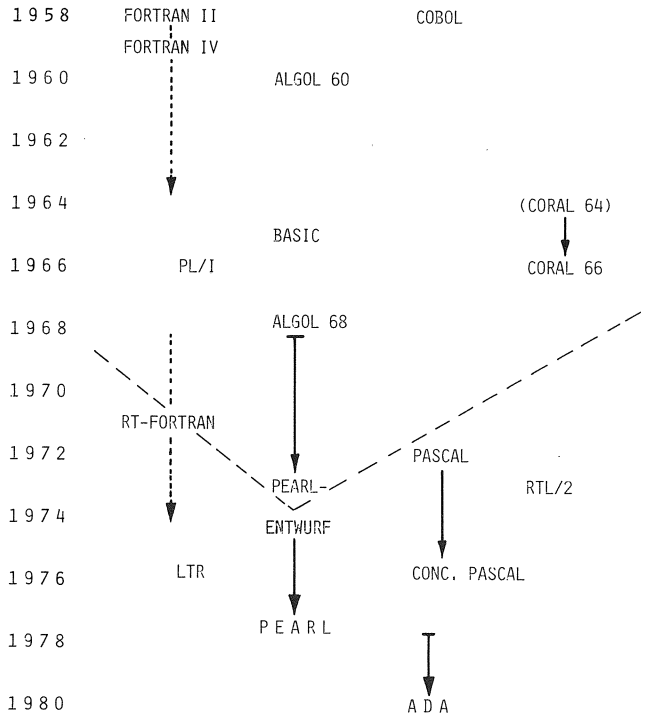
Auf der Jahrestagung des PEARL-Vereins im Dezember 1980 in Düsseldorf wurden auch zwei Vorträge gehalten über die Eigenschaften von PEARL im Vergleich mit den Programmiersprachen: Concurrent PASCAL, Ada, Realtime FORTRAN, CORAL 66, BASIC, RTL/2 und LTR. Diese Vorträge wurden im vollen Wortlaut in der PEARL-Rundschau Nr. 4, 1980 veröffentlicht [1,2], zusammen mit einer Darstellung der Entwicklungsgeschichte von PEARL durch den Verfasser [3]. Während seines Vortrages verwendete der Verfasser jedoch eine etwas andere Darstellungsweise und vor allem zwei Übersichtsbilder, nach denen er in der Zwischenzeit eine große Anzahl von Anfragen erhielt. Er möchte deshalb die Gelegenheit dieses Heftes nutzen, seine improvisierten Ausführungen von damals zu rekonstruieren und zu Papier zu bringen.

2. Die Entwicklung von PEARL

Diese wurde in [3] schon recht ausführlich dargestellt, wobei vor allem auch die Einflüsse anderer Programmiersprachen und parallel ablaufende Sprachentwicklungen berücksichtigt wurden. In Fig. 1 wird nur noch einmal versucht, einen gerafften Überblick in grafischer Form zu geben, da ja bekanntlich "ein Bild mehr sagt als tausend Worte". Ganz verkürzt könnte man sagen, daß alle Programmiersprachen oberhalb der gestrichelt dargestellten "Entwicklungsbugwelle" irgendeinen Einfluß auf die Entwicklung und die Eigenschaften von PEARL hatten, die anderen kaum.

3. Die Problemüberdeckung von PEARL

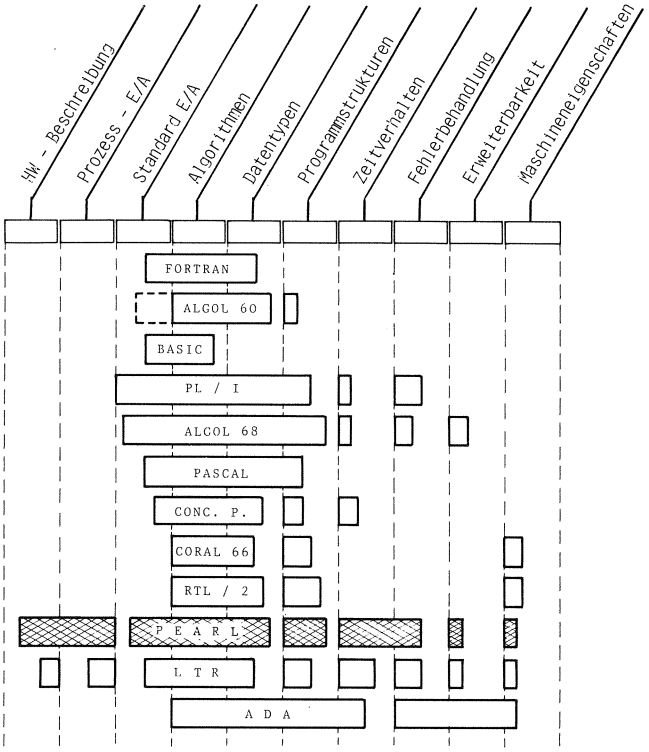
Dieser Ausdruck ist erläuterungsbedürftig. Normalerweise spricht man ja von "Elementen" oder "Eigenschaften" einer Programmiersprache. Diese Begriffe erscheinen aber für einen extrem kurzen Vergleich nicht recht brauchbar. Was den Anwender ja letztendlich interessiert, ist: "Welche Probleme kann ich mit der in Frage stehenden Programmiersprache denn nun eigentlich lösen?"



Figur 1: Die Entwicklung von PEARL im Umfeld anderer Programmiersprachen

Auf diese Frage geben aber Überlegungen bezüglich "Sprachelementen" und "-eigenschaften" immer nur indirekte Antworten. Es ist ja doch so, daß ein bestimmtes Problem jeweils durch verschiedene Sprachelemente gelöst (oder beschrieben) werden kann oder daß die "Eigenschaften" einer Sprache über ihre Problemlösungskapazität nicht viel aussagen, sondern mehr für eine Klassifizierung gemäß anderer wissenschaftlicher Kriterien geeignet sind.

Es wurde deshalb einmal versucht, die Anforderungen der Anwendungsklasse "Realzeitprogrammierung" in Teilgebiete aufzugliedern. Diese gruppieren sich um den ursprünglichen und eigentlich immer noch zentralen Zweck der Formulierung von Algorithmen. Die Beschreibung der Datentypen und die Behandlung von Standard-Ein-/Ausgabe (Text) kamen relativ frühzeitig dazu. Der nächste große Schritt war dann der Einbau von Sprachelementen zur Struk-



Figur 2: Die Problemüberdeckung von PEARL im Vergleich mit anderen Programmiersprachen

turierung von Programmen, zur Beeinflussung ihres Zeitverhaltens und zur Behandlung eventuell auftretender Fehler. Bei Sprachen für Realzeitanwendungen, wie PEARL oder LTR, war es dann nötig, die Beeinflußbarkeit des Zeitverhaltens stark zu verbessern und weitere Teilgebiete, wie Prozeß-Ein-/Ausgabe, Beschreibung der prozeßorientierten Hardware und Maschineneigenschaften zu überdecken.

Auch eine gewisse Erweiterbarkeit der Sprache erschien in Anbetracht der großen Problemvielfalt auf dem Realzeitsektor angebracht. Dafür konnte auf anderen Teilgebieten auf letzten Komfort verzichtet werden.

Dies soll die Darstellung in Fig. 2 veranschaulichen. Die Spalten stellen die einzelnen Teilgebiete dar. Ihre Breite wurde gleichmäßig gewählt, um keine Wertung zu implizieren. Der Grad der Überdeckung der Teilgebiete durch die verschiedenen Sprachen wird jeweils durch die Länge der einzelnen Balken symbolisiert. So liefert PEARL z.B. eine vollständige Beeinflußbarkeit des Zeitverhaltens von Programmen, aber nur die nötigsten Elemente zur Erweiterbarkeit. Bei Ada z.B. ist dies gerade umgekehrt, usf.

4. Schlußbemerkung

Der Verfasser hofft, mit dieser Kurzdarstellung einem oft geäußerten Bedürfnis entsprochen und den Lesern der PEARL-Rundschau einige neue Argumentationshilfen geliefert zu haben.

[13] Frevert,L.: "Vergleich von PEARL mit Concurrent PASCAL und Ada" PEARL-Rundschau Bd.1,4 S.61 1980

[2] Rzehak,H.: " Vergleich mit Programmiersprachen,die vor bzw. parallel zu PEARL entwickelt wurden" PEARL-Rundschau Bd.1, 4 S.65 1980

[3] Elzer, P.: " Rückblick auf die Entwicklung von PEARL" PEARL-Rundschau Bd. 1,4 S.58 1980

Nachdruck aus der Zeitschrift rt-Regelungstechnik, 25. Jahrgang 1977, Heft 11, Seite 337-368

Der folgende Sprachvergleich bezieht sich auf Sprachen, die 1975 verfügbar waren. Aus heutiger Sicht müßte der Vergleich von Realzeitsprachen ADA und Modula berücksichtigen, ebenso die bei der Normung von PEARL vorgenommenen Präzisierungen der Sprache und ihrer Beschreibung.

PEARL im Vergleich mit anderen Echtzeitsprachen*

A. Schwald, R. Baumann

Die Echtzeitprogrammiersprache PEARL *wird mit anderen Programmiersprachen zur Steuerung technischer Prozesse* (FORTRAN-*Erweiterungen*, Coral66, RTL/2, PROCOL *und* HAL/S) *verglichen. Dabei werden zunächst zwei verschiedene Ansätze zur Einführung der für die Echtzeitprogrammierung spezifischen Sprachelemente und Anforderungen an den algorithmischen Teil einer Echtzeitsprache diskutiert.*

Die Erfüllung dieser Anforderungen, die Sprachelemente zur Steuerung von parallelen Abläufen und die Kommunikation mit der Prozeßumgebung werden für die Vergleichssprachen einander gegenübergestellt.

The real time programming language PEARL *is compared with other programming languages for the control of technical processes* (FORTRAN *extensions*, Coral 66, RTL/2, PROCOL *and* HAL/S). *First, two different statements for the introduction of the language elements, which are specific for real time programming, and of the requirements from the algorithmic part of a real time language, are discussed.*

The fulfillment of these requirements, the language elements for the control of parallel processes, and the communication with the process environment are compared for the concerning languages.

1. Einleitung

Die zunehmende Verwendung von Rechnern zur Steuerung von industriellen Prozessen und die Entwicklungen im Bereich der Hardware- und Software-Konstruktion haben seit einigen Jahren die Definition und Implementierung von geeigneten Programmiersprachen für diesen Anwendungsbereich, die sogenannten Echtzeitsprachen, immer mehr in den Mittelpunkt des Interesses gerückt.

Der Trend, dem diese Entwicklungen unterliegen, soll durch einige Stichworte charakterisiert werden.

- Es werden Software-Systeme immer größerer Komplexität gebaut. Die Forderungen an diese Systeme (Verläßlichkeit, Reaktionsgeschwindigkeit u. a.) steigen mit dem Umfang ihres Einsatzes.
- In der Diskussion um Programmiersprachen werden immer stärker Gesichtspunkte wie Sicherheit, Einfachheit oder Modularität betont. Die „großen" Sprachen wie PL/I oder ALGOL 68 haben – zumindest bisher – nicht die erhoffte Verbreitung gefunden.
- Die starke Miniaturisierung und Verbilligung von kleinen Prozessoren und von Speichern verursachte einen starken Trend zur Dezentralisierung. Nicht mehr ein großer Allzweckrechner ist der alleinige Träger der Datenverarbeitungsleistungen, der Prozeßrechner „vor Ort" übernimmt immer mehr Aufgaben der Prozeßsteuerung und der Datenverarbeitung.

Für die folgenden Ausführungen, die eine Einordnung von PEARL in die Gruppe der Echtzeitsprachen bezwecken, wurden als Vergleichssprachen herangezogen:

FORTRAN-Erweiterungen [4; 5; 6],
CORAL 66 und RTL/2 [9; 10],
PROCOL [7; 8],
HAL/S [11].

Diese Auswahl kann natürlich als Einschränkung der Themenstellung angesehen werden. Es wäre interessant, auch andere Sprachen zu berücksichtigen z. B. SIMULA 67, LTR (das PROCOL ablöst) oder MODULA. Eine Auswahl ist jedoch notwendig, um nicht in der Fülle der Einzelfakten zu ersticken. Die wichtigsten Gründe für die Auswahl der genannten Sprachen sind:

- Diese Vergleichssprachen sind geeignet, um einige wichtige Unterschiede deutlich zu machen.
- Sie sind in verschiedenen Ländern (USA, GB, F, D) entwickelt worden, stellen also auch in etwa die national verschiedenen Ansätze dar.

Bezüglich der FORTRAN-Erweiterungen wurden vor allem die Vorschläge der ISA und der Entwurf für eine VDI/VDE-Richtlinie herangezogen [4; 5; 6], für die anderen Sprachen die neuesten verfügbaren Spezifikationen.

In den folgenden Ausführungen wird einerseits von technischen Prozessen, andererseits von Programmen und ihrer Ausführung die Rede sein. Um die verschiedenen Gesichtspunkte auseinanderhalten zu können, soll folgende Terminologie verwendet werden:

Eine *PA* (parallele Aktivität, *parallel activity*) ist ein Programmablauf, der möglicherweise parallel zu anderen Programmabläufen ausgeführt wird.

Eine *Task* ist das einer *PA* zugeordnete Programmstück.

Ein *technischer Prozeß* ist ein physikalischer Ablauf, der durch eine oder mehrere *PA*'s gesteuert oder geregelt wird.

* Überarbeitete Fassung eines Beitrages zum PEARL-Aussprachetag am 9. März 1977 in Augsburg.

2. Zwei verschiedene Ansätze

Die genannten Sprachen wählen zwei verschiedene Ansätze, um dem Benutzer die notwendigen Ausdrucksmöglichkeiten zu bieten.

A

Im Falle von PEARL und PROCOL wird eine Sprache neu definiert, einschließlich der für die Echtzeitprogrammierung erforderlichen Sprachelemente, die bei FORTRAN und anderen Sprachen zur Beschreibung sequentieller Abläufe fehlen.

B

Eine ganz andere Vorgangsweise liegt der Definition von CORAL 66 und RTL/2 zugrunde. Diese Sprachen enthalten nur die Elemente einer Implementierungssprache, dazu einen Erweiterungsmechanismus (z. B. die Möglichkeit, Assemblersequenzen in den Programmtext einzuschieben), um alle notwendigen Leistungen der Hardware und des Betriebssystems, die einer Implementierung zugrunde liegen, direkt nutzen zu können. Beide Sprachen enthalten weder *Task*-Anweisungen noch EA-Anweisungen. Die Steuerung von *PA*'s muß durch Betriebssystemdienste realisiert werden.

Im Falle von Prozeß-FORTRAN dient die genormte Sprache als Grundsprache, die Erweiterungen werden als Aufrufe von speziellen Prozeduren definiert. Die Wirkung dieser Prozeduren und ihre Parameterversorgung werden soweit wie möglich unabhängig von einer speziellen Implementierung beschrieben.

Auf den ersten Blick mag das für CORAL 66 und RTL/2 gewählte Vorgehen als Notlösung erscheinen, die das Ziel einer problemorientierten, maschinenunabhängigen Sprache in wesentlichen Teilen verfehlt. Bei genauer Betrachtung kann dieser Ansatz trotzdem einige gute Argumente für sich beanspruchen. Die folgende Gegenüberstellung vergleicht die beiden Ansätze im Hinblick auf allgemeine und einige besonders für die Echtzeitprogrammierung wichtige Kriterien.

Für A

+ Bessere Portabilität von Programmen; System- und Konfigurationsänderungen sind für den Benutzer unsichtbar oder wirken sich nur in einer Schnittstellenbeschreibung (Systemteil) aus.
+ Effizienz der Programmierung; problemorientierte Schreibweise ist möglich (z. B. *schedules* für die Aktivierung von *PA*'s).
+ Die Sprache fordert implizit eine bestimmte Systemschnittstelle. Bei Verfügbarkeit eines entsprechenden Betriebssystems ist die Implementierung der Sprachelemente zur Steuerung von *PA*'s ohne nennenswerte Effizienzverluste möglich.
+ Explizitheit bezüglich des Gesamtablaufs und der Konfiguration eines Prozeßsteuerungssystems.

Gegen A

– Aktivitäten mit harten quantitativen Randbedingungen (z. B. Speicherkapazität, enge Zeitbedingungen) lassen sich nach dem derzeitigen Stand nicht beherrschen.
– Die zeitgerechte Ausführung einer *PA* kann nicht garantiert werden. Diese hängt nicht nur von der *PA* und ihren EA-Aktivitäten ab, sondern vom Zustand des Gesamtsystems.
– Wesentliche Teile der Betriebssystemfunktionen (z. B. Synchronisationselemente) müssen in der Programmiersprache noch einmal definiert werden.

Für B

+ Einfachheit der Sprache, geringer Implementierungsaufwand (auch auf Kleinrechnern), leichter zu erlernen und zu beherrschen.
+ Effizienz der Objektprogramme (einfacher Übersetzer mit weitergehenden Optimierungsmöglichkeiten; die Effizienz der Objektprogramme kann durch geschickte Programmierung verbessert werden; Mehraufwand durch Abbildung von Sprachelementen auf Folgen von Systemaufrufen entfällt).
+ Flexibilität, neue Systemdienste können ohne Sprachänderungen verwendet werden.
+ Explizitheit bezüglich der einzelnen Operationen, insbesondere ist genaue Kontrolle zeitkritischer Abläufe möglich.

Gegen B

– Bessere Kenntnisse des Betriebssystems sind erforderlich, da seine Dienste direkt verwendet werden müssen.
– Maschinenabhängigkeit bzw. Betriebssystemabhängigkeit; code-Anweisungen verursachen Schnittstellen- und Sicherheitsprobleme.
– Problemorientierte Modellierung der Betriebssystemschnittstelle (z. B. Zusammenfassung mehrerer Einzelleistungen in ein Sprachelement) ist nicht möglich.
– Über die Synchronisation von *PA*'s und das Echtzeitverhalten sind keine direkten Aussagen (in der höheren Sprache) möglich.

Aus dieser Gegenüberstellung sind die wichtigsten Problembereiche bei der Definition, Implementierung und Anwendung von Echtzeitsprachen zu ersehen.

● Der Anwendungsbereich der Grundsprache

Dieser muß entsprechend groß sein, wenn mehr als nur ein ganz spezieller Benutzerkreis angestrebt wird. Es sind ausreichende Ausdrucksmittel für numerische und für nichtnumerische Programme vorzusehen.

● Das Zusammenspiel zwischen dem Betriebssystem und der Sprache

Besonders die Sprachelemente zur Steuerung von *PA*'s enthalten weitgehende und direkte Anforderungen an das einer Implementierung zugrunde liegende Betriebssystem.

● Festlegung von Schnittstellen

Detaillierte Beschreibung der Schnittstelle zur Prozeßperipherie ist erforderlich.

● Quantitative Randbedingungen

Diese betreffen vor allem die Beherrschung von zeitkritischen Aktivitäten und die Speicherkapazität von Kleinrechnern.

3. Anforderungen an den algorithmischen Teil

Der algorithmische Teil einer Sprache zur Prozeßsteuerung ist gewöhnlich einer der bekannten höheren Sprache, z. B. ALGOL oder PL/I, nachempfunden. Spezielle Anforderungen ergeben sich in zwei Bereichen:

α) Aus den Echtzeitanforderungen ergibt sich die Notwendigkeit paralleler Aktivitäten. Die globalen Strukturierungsmittel müssen (abweichend von der ALGOL-Blockstruktur) modularen Programmaufbau, die Deklaration von *Tasks* und die Auszeichnung gewisser Datenbereiche als Kommunikationsbereiche zwischen *PA*'s ermöglichen.

β) Wegen der direkten Kopplung mit der Prozeßperipherie werden viele Einzelheiten der Hardware für den Programmierer sichtbar. Zur Anpassung an vorgegebene Schnittstellen muß die vom Übersetzer gewählte Ablage von Daten beeinflußbar sein. Insbesondere soll die Beschreibung von Datenstrukturen, deren Komponenten Teilworte belegen, und der Durchgriff auf einzelne Bitfolgen ermöglicht werden.

a) PEARL

α) Ein PEARL-Programm besteht aus einer Menge von Moduln mit ALGOL-ähnlicher Blockstruktur. *Tasks* können auf Modulebene, jedoch auch geschachtelt in inneren Blöcken, vereinbart werden.

Als Kommunikationsbereiche zwischen *PA*'s stehen zur Verfügung:

- Dateien, auf die mehrere *PA*'s zugreifen.
- Globale bzw. externe Variable eines Moduls.
- Für modulinterne *Tasks* die in der Blockstruktur globalen Variablen.

β) Die Datentypen von PEARL sind aus ALGOL 68 übernommen (trotz der PL/I-orientierten Schreibweise von Deklarationen).

Die einfachen Datentypen sind fixed, float (beide mit Genauigkeitsangabe), char(n), bit(n), dazu die Zeigertypen, z. B. ref char(1). Der algorithmische Teil verzichtet auf Direktiven zur Steuerung der Internablage von Daten, sieht jedoch Konvertierung zwischen bit(n) und fixed vor. Eine bei der EA verwendete Geräteschnittstelle kann durch ein Format bzw. durch eine Konnektorbeschreibung im Systemteil spezifiziert werden.

Zur Festlegung von Aktivierungsplänen *(schedules)* und zur Synchronisation zwischen *PA*'s gibt es weitere Datentypen:

sema bolt	zur Synchronisation,
duration	Zeitspanne,
clock	Uhrzeit,
interrupt	Ereignis zur Steuerung von *PA*'s.

Der Datentyp signal beschreibt Ereignisse, die den sequentiellen Ablauf innerhalb einer *PA* beeinflussen. Die induce-Anweisung dient zur Anzeige eines Signals. Sie wirkt ähnlich wie die signal-Anweisung von PL/1.

b) HAL/S

α) Ein HAL/S-Programmkomplex wird aus drei Arten von Moduln aufgebaut:

- Programmoduln, die unabhängig voneinander ausführbar sind.
- Externe Prozeduren und Funktionen.
- Sogenannte *compool*-Moduln; sie enthalten Daten, die im ganzen Programmkomplex verwendbar sind.

In einem Programmodul können mehrere *Tasks* (nicht verschachtelt) erklärt werden.

β) Die einfachen Datentypen von HAL/S sind:

- scalar (sonst real genannt), integer, boolean, character.
- Datenstrukturen werden ähnlich wie in PL/I festgelegt; vector und matrix gelten als spezielle arithmetische Typen.
- Daten vom Typ event werden zur Steuerung von *PA*'s verwendet. Dem PEARL-Datentyp signal und der induce-Anweisung entsprechen PL/I-ähnliche Sprachelemente zur Reaktion auf Fehlersituationen innerhalb einer *PA*.

c) PROCOL

α) Ein PROCOL-Programm setzt sich zusammen aus

- Kommunikationsbereichen, welche die Beschreibung von Variablen, von Formaten und von Schnittstellen zur Prozeßperipherie enthalten.
- Externen Unterprogrammen, die von verschiedenen *PA*'s aufgerufen werden können.
- *Tasks*.

PROCOL ist stark FORTRAN-orientiert. In einer *Task* gibt es keine untergeordneten *Tasks* und keine Blockstruktur.

β) Die einfachen Datentypen von PROCOL sind:

- binary (ein Wort wird als Bitfolge aufgefaßt), integer, shorteger (ganzzahlig, maximal elf Stellen) und real.
- Strukturen sind vom Typ binary, ihre Komponenten, die Teile eines Wortes belegen, werden je nach Verwendung als binary oder integer interpretiert.

d) RTL/2

α) Ein RTL/2-Programm wird unterteilt in eine Menge von Moduln, die sich aus Prozedur-, Daten- und Keller-„Bausteinen" zusammensetzen. Einer *PA* wird bei ihrer Erzeugung ein Keller als Arbeitsspeicher und eine Prozedur als Code zugeordnet. Prozeduren und Daten können von mehreren *PA*'s verwendet werden. RTL/2 definiert die Programmstruktur im Hinblick auf ihre Zuordnung zu *PA*'s, ohne selbst *Task*-Anweisungen zu enthalten. Als speziellen Typ von externen Prozeduren gibt es Systemdienste (für die der Übersetzer speziellen Objektcode erzeugen kann).

β) Die einfachen Datentypen sind integer, real, fraction (Wertbereich $[-1, +1]$) und byte (Wertbereich $[0,255]$), dazu die ref-Typen ähnlich ALGOL 68.

Logische Operationen gibt es auf integer und byte (sie wirken bitweise), für integer gibt es auch *shift*-Operatio-

nen. Zum Unterschied von PROCOL und CORAL 66 ist RTL/2 „byteorientiert", d. h. auf einem Rechner mit Byte-Adressierung leichter und effizienter zu implementieren als auf einem Rechner mit Wort-Adressierung.

e) CORAL 66

α) Ein CORAL 66-Programm kann in einen durch common eingeleiteten Kommunikationsbereich und eine Anzahl von Segmenten zerlegt werden. Die Sprachdefinition macht keine Aussage über mögliche Zuordnungen zwischen *PA*'s und Programmteilen.

β) CORAL 66 ist sehr stark an ALGOL 60 orientiert. Seine einfachen Datentypen sind integer, floating, fixed (integer mit Skalenfaktor) und unsigned (positiv ganzzahlig). Gepackte Daten können in tables erklärt werden. Es gibt Operationen zum Teilwortzugriff und zur Adreßrechnung, außerdem logische Operationen auf integer-Größen.

Wie in PROCOL und RTL/2 gibt es in CORAL 66 eine code-Anweisung, die den Einschub von Assemblersequenzen in den Programmtext zuläßt.

f) FORTRAN

α) Die Grundsprache FORTRAN ist bezüglich Programmstruktur und Datentypen vorgegeben. Die in Normungsgremien diskutierten Vorschläge halten sich strikt an den definierten Sprachumfang und verwenden Prozeduraufrufe, um weitergehende Leistungen verfügbar zu machen.

In Prozeß-FORTRAN wird eine *PA* auf einem Programm gestartet. Über Kommunikationsbereiche (ausgenommen Dateien, die von mehreren *PA*'s bearbeitet werden) gibt es in Prozeß-FORTRAN keine Aussagen.

β) Es gibt eine Reihe von Prozeduren, die logische und *shift*-Operationen auf integer-Größen (die als Bitfolgen aufgefaßt werden) durchführen, außerdem die Möglichkeit, einzelne Bits zu setzen oder abzufragen.

Das Vermeiden echter Spracherweiterungen, besonders die Einschränkung auf die in FORTRAN definierten Datentypen, bedingt in manchen Fällen eine mühsame Schreibweise, hat aber den wichtigen Vorteil, daß ein vorhandener FORTRAN-Übersetzer sehr leicht für die Zusätze von Prozeß-FORTRAN adaptiert werden kann.

4. Sprachelemente zur Steuerung paralleler Aktivitäten

Dieser Bereich stellt die wichtigsten und interessantesten Anforderungen an eine Echtzeitsprache, da sich hier drei Problemkreise überschneiden, nämlich

- die zeitabhängige Ausführung von *PA*'s,
- die Ausführung von *PA's* als Reaktion auf bestimmte Ereignisse, insbesondere im Zusammenhang mit EA-Vorgängen,
- die Synchronisation konkurrierender *PA*'s beim Zugriff auf Betriebsmittel.

Eine ausgewogene Lösung muß das möglichst reibungslose Zusammenspiel dieser drei Bereiche sichern.

CORAL 66 und RTL/2 scheiden aus den folgenden Betrachtungen aus, da ihre Definition keine Sprachelemente zur Steuerung von *PA*'s enthält.

a) FORTRAN-*Zusätze*

ISA-S 61.1 [4] enthält die folgenden Prozeduren zur Steuerung von *PA*'s:

START zur Aktivierung einer *PA* (sofort oder mit einer bestimmten zeitlichen Verzögerung),

TRNON zur Aktivierung einer *PA* zu einem bestimmten Zeitpunkt (Uhrzeit),

WAIT zur Verzögerung der aktuellen *PA* um eine bestimmte Zeitspanne.

Diese Prozeduren und die Prozeduren für Prozeß-EA haben Ausgabeparameter, die das Auftreten von Fehlern bzw. den Zustand einer Auftragsbearbeitung anzeigen und so in gewissem Umfang auch ereignisabhängige Programmierung ermöglichen.

Diese Grundfunktionen bieten zwar die Möglichkeit zu zeitabhängiger Steuerung von *PA*'s, lassen jedoch viele Wünsche (z. B. Prioritätensteuerung, zyklische Aktivierung, Reaktion auf Signale, Synchronisierung von Datenzugriffen) offen. Weitergehende Vorschläge [5; 6] befassen sich mit der Bearbeitung von Dateien in wahlfreiem Zugriff und mit einem ausreichenden Satz von Prozeduren zur Verwaltung von *PA*'s, einschließlich der Reaktion auf Ereignisse, der Synchronisation und der Kommunikation zwischen *PA*'s. In diesem Zusammenhang ist insbesondere Prozeß-FORTRAN 75 [6] zu erwähnen, das von einem Arbeitskreis der VDI/VDE-Gesellschaft für Meß- und Regelungstechnik erarbeitet wurde. Dieser Entwurf zeichnet sich gegenüber [5] vor allem durch weitergehende Sicherheitsmaßnahmen aus.

Der Verzicht auf echte Spracherweiterungen und das Fehlen entsprechender Datentypen zur Steuerung von *PA*'s legen den Erfindern und Benutzern der FORTRAN-Zusätze manche Einschränkung auf, machen sich jedoch im geringen Implementierungs- und Lernaufwand bezahlt.

b) PROCOL

Tasks werden mit einer bestimmten Priorität erklärt. Sie können die Attribute resident und wiederaufnehmbar (nach einer Verdrängung durch eine höherpriore *PA*) haben. Auf einer *Task* darf es zu einem bestimmten Zeitpunkt nicht mehr als eine *PA* geben.

Die Aktivierung einer *Task* kann durch eine activate-Anweisung sofort erfolgen, aber auch nach Eintreten eines Ereignisses, beim Erreichen eines bestimmten Zeitpunktes oder zyklisch in vorgegebenen Zeitintervallen.

Weitere *Task*-Anweisungen sind:

cancel zur Beendigung einer anderen *PA*,

exit zur Beendigung der aktuellen *PA* und

wait zur Verzögerung der aktuellen *PA* bis zum Erreichen eines timer-Wertes oder zum Eintreffen eines Unterbrechungssignals.

Systembezogene Größen zur Steuerung von *PA*'s, die in einer *Task* verwendet werden können, sind timer, interrupt und semaphore. Diese werden durch Kommandos (siehe 5.) eingeführt.

Eine timer-Größe dient zur zeitabhängigen Prozeßsteuerung. Durch eine start-Anweisung wird sie initialisiert; mit der per

Kommando festgelegten Zykluszeit wird weitergezählt. Eine stop-Anweisung bewirkt das Anhalten einer timer-Größe und die Beendigung aller mit ihr verknüpften Aktivitäten. Zum Unterschied von PEARL oder FORTRAN zeigt eine timer-Größe immer nur eine Relativzeit bezogen auf den Start an, nicht eine Tageszeit. Um eine timer-Größe gruppieren sich Aktivitäten, die zeitlich aufeinander abgestimmt werden müssen.

Einer interrupt-Größe kann eine *Task* zugeordnet werden, die beim Eintreffen des Unterbrechungssignals aktiviert wird. Ein Unterbrechungssignal kann durch die Hardware oder durch eine signal-Anweisung ausgelöst werden. Zum Schutz gegen Unterbrechungen dienen die Anweisungen mask (für Unterbrechungssignale) und disable (Schutz gegen Unterbrechung durch höherpriore *PA*'s). Ihre Wirkung wird durch unmask bzw. enable wieder aufgehoben.

Zur Synchronisation dienen die Anweisungen request und release für Semaphore. Zur Reservierung von Peripheriegeräten für eine *PA* gibt es die Anweisung attach, die Freigabe erfolgt mittels detach.

Die Speicherverwaltung für PROCOL-Programme besteht aus dem Zuladen bzw. Verdrängen nicht residenter *Tasks*. Die Unterbrechung einer *PA* bewirkt ihre Beendigung, falls nicht vom Benutzer die entsprechende *Task* als wiederaufnehmbar erklärt wird. Für unterbrochene, wiederaufnehmbare *PA*'s gibt es einen Pufferbereich für genau eine *PA*, d. h. bei der Rettung einer unterbrochenen *PA* werden eventuell vorher im Puffer abgelegte Informationen zerstört.

Diese sehr simple Behandlung entspricht der Vorstellung, daß es nicht mehr als eine Vordergrund- und eine Hintergrundaktivität gibt. Sie vereinfacht die Implementierung und ermöglicht genaue Planung des Speicherbedarfs, ist aber doch zu stark vereinfachend und nicht ganz konsistent mit der Vielfalt der anderen Speicherelemente. Durch die Anweisungen mask und disable kann eine *PA* zwar vor Unterbrechung geschützt werden, dies geht jedoch zu Lasten der Reaktionszeit von *PA*'s mit höherer Priorität.

c) HAL/S

Eine *PA* kann auf einem Programmodul oder auf einer *Task* aktiviert werden. Sie kann von ihrem Initiator (eine andere *PA*) abhängig oder unabhängig sein. Eine abhängige *PA* kann nur existieren, wenn ihr Initiator existiert, eine unabhängige *PA* ist nicht an die Existenz anderer *PA*'s gebunden. Eine *PA* auf einer *Task* ist immer von der *PA* auf dem Programmodul, der die *Task*-Deklaration enthält, abhängig. Auf einer *Task* kann es zu einem bestimmten Zeitpunkt nicht zwei *PA*'s geben.

Die zentrale Anweisung zur Steuerung von *PA*'s ist die schedule-Anweisung. Die dadurch erzeugte neue *PA* wird in die *PA*-Warteschlange eingetragen und ihr Aktivierungsplan festgelegt.

Folgende Angaben über die neue *PA* sind vorgesehen:

- Name der *Task* bzw. des Programms, auf dem die neue *PA* erzeugt wird.
- Aktivierungsbedingung (sofort; Eintreten eines Ereignisses; Erreichen eines Zeitpunktes; Ablauf eines Zeitintervalls).
- Priorität.
- Abhängigkeit vom Initiator.
- Zeitintervall für zyklische Aktivierung.
- Abbruchbedingung (Ereignis, Zeitpunkt) für den Aktivierungszyklus.

Absolute und relative Zeitangaben erfolgen in Einheiten der Hardware-Uhr als integer-Größen.

Weitere Anweisungen zur Steuerung von *PA*'s sind:

terminate zur sofortigen Beendigung einer *PA* mit allen abhängigen *PA*'s.

cancel zur Entfernung einer *PA* aus der *PA*-Warteschlange. Eine *PA* im Wartezustand bzw. noch ausstehende Aktivierungen einer zyklischen *PA* werden storniert.

wait zur Überführung der aktuellen *PA* in den Wartezustand. Als Reaktivierungsbedingung kann ein Ereignis, eine Zeitangabe oder die Beendigung der abhängigen *PA*'s angegeben werden.

update zur Änderung der in der schedule-Anweisung

priority festgelegten Priorität.

Es gibt zwei Arten von Ereignissen, die bei der Steuerung von *PA*'s eine Rolle spielen: die Ereignisanzeige (event) und die Veränderung eines Zustands (latched event). Das Eintreffen eines Ereignisses bewirkt die Neuauswertung aller mit dem Ereignis verknüpften *schedules*.

Die signal-Anweisung dient zur Anzeige eines Ereignisses. Mit set- und reset-Anweisungen wird der Wert eines Zustands auf true bzw. false gesetzt.

Zur Synchronisation von Datenzugriffen durch verschiedene *PA*'s gibt es das lock-Attribut für Variable und die den kritischen Regionen entsprechenden *update-blocks*. Eine Variable mit dem Attribut lock (i), $i \geqq 1$, darf nur in *update-blocks* verwendet werden. Wollen zwei *PA*'s simultan Variable ändern, die zur selben *lock*-Gruppe lock (i) gehören, so wird eine *PA* in den Wartezustand versetzt, bis die andere den *update-block* mit der Änderung einer lock (i)-Variablen verlassen hat. Falls mehrere *PA*'s nur lesend auf Variable derselben *lock*-Gruppe zugreifen wollen, kann eine Implementierung simultanen Zugriff ermöglichen.

HAL/S verwendet keine expliziten Semaphore, sondern führt die Semaphor-Operation implizit aus (als Prolog- bzw. Epilog-Aktion eines *update-blocks*). Diese Lösung ist nicht nur komfortabler, sondern auch sicherer als die explizite Verwendung von Semaphoren.

Insgesamt bietet HAL/S einen hinreichend flexiblen Satz von Anweisungen zur Steuerung von *PA*'s, in dem durch einige Einschränkungen (keine beliebige Schachtelung von *Tasks*, nicht mehrere *PA*'s gleichzeitig auf einer *Task*) die Komplexität des semantischen Modells und damit auch der Implementierungsaufwand und der Zeitaufwand für die Verwaltung von *PA*'s in Grenzen gehalten wird.

d) PEARL

Eine *Task* wird mit einer bestimmten Priorität deklariert und kann das Attribut **resident** haben. Eine *Task* kann Deklarationen von *Subtasks* (nach den Regeln der ALGOL-Blockstruktur) enthalten. Auf einer *Task* kann es zu einem bestimmten Zeitpunkt mehrere *PA*'s geben, jedoch kann davon höchstens eine aktiv sein.

Bei jeder *Task*-Anweisung (z. B. **activate**, **wait**) kann ein *schedule* für die Ausführung angegeben werden. Der *schedule* beschreibt die Bedingung für die Ausführung der Anweisung (Eintreffen eines Unterbrechungssignals, Zeitbedingung) und legt die Zeitintervalle für zyklische Ausführungen fest. Zeitangaben erfolgen in Einheiten von Sekunden, als Datentypen werden **clock** (Tageszeit) und **duration** (Zeitspanne) verwendet.

Die folgenden *Task*-Anweisungen ändern entweder den Zustand einer *PA* oder die einer *Task* zugeordneten *schedules*:

activate bewirkt die Aktivierung einer *PA* bzw. die Einplanung einer *PA* aufgrund des in der Anweisung vorgesehenen *schedules*. Für die Aktivierung kann eine von der *Task*-Deklaration abweichende (dynamische) Priorität festgelegt werden. Eine **activate**-Anweisung kann eine **sema**-Variable benützen. In diesem Fall impliziert die Ausführung der Anweisung eine **request**-Operation auf der **sema**-Variablen.

terminate beendigt die Ausführung einer *PA*.

prevent löscht alle für eine *Task* eingerichteten *schedules* (es gibt eventuell verschiedene *schedules* für verschiedene Anweisungen).

suspend blockiert die weitere Ausführung einer *PA*.

continue hebt die Wirkung einer vorangegangenen **suspend**-Anweisung auf.

resume bewirkt die Blockierung einer *PA* und gibt (mit einem *schedule*) die Bedingung für ihre Fortsetzung an.

Die Anweisungen **suspend**, **continue** und **resume** wirken auf alle *Subtasks* der in der Anweisung angegebenen *Tasks*, wenn dies nicht durch eine **except**-Option verhindert wird.

Der Datentyp **interrupt** ist für Ereignisvariable vorgesehen, die zur Steuerung von *PA*'s dienen. Ihre Zuordnung zu Hardware-Signalen wird im Systemteil beschrieben (siehe 5.). Vom Programm her kann ein Unterbrechungssignal durch eine **trigger**-Anweisung ausgelöst werden. Durch die **disable**-Anweisung wird eine Unterbrechungssperre gesetzt, durch die **enable**-Anweisung wieder aufgehoben.

Zu Synchronisationszwecken gibt es Variable vom Typ **sema** und **bolt**. Auf **sema**-Variablen sind die Semaphor-Operationen **request** und **release** vorgesehen. Auf **bolt**-Variablen gibt es vier Operationen:

reserve zur Reservierung eines Betriebsmittels für exklusive Benutzung durch eine *PA*.

free zur Freigabe eines vorher mit **reserve** belegten Betriebsmittels.

enter zur Belegung eines Betriebsmittels gemeinsam mit anderen *PA*'s.

leave zur Freigabe eines vorher mit **enter** belegten Betriebsmittels.

Für eine **bolt**-Variable kann die Maximalzahl gemeinsamer Belegungen angegeben werden. Mit Hilfe von **bolt**-Variablen läßt sich z. B. die Synchronisierung von lesenden und schreibenden Zugriffen auf eine Datei in bequemer und übersichtlicher Weise formulieren.

Es ist zu bemerken, daß mit der Blockstruktur implizite Synchronisationsoperationen verbunden sind. Das Verlassen des Blocks einer *Task*-Deklaration, auf der es noch *PA*'s gibt, wird bis zur Beendigung dieser *PA*'s verzögert.

PEARL bietet von den hier behandelten Sprachen vom Sprachumfang her die meisten Hilfsmittel zur Steuerung von *PA*'s, abgesehen vom Konzept der kritischen Regionen. Dementsprechend ist das semantische Modell zur Beschreibung von *PA*'s wesentlich komplizierter als in anderen Sprachen. Daraus ergibt sich beträchtlicher Aufwand für die Implementierung, insbesondere durch hohe Anforderungen an das Betriebssystem.

Vor allem das *Subtask*-Konzept von PEARL bringt mehr Aufwand und Schwierigkeiten als Vorteile. Auch die Tatsache, daß auf einer *Task* mehrere *PA*'s gleichzeitig aktiv sein können, und die Möglichkeit, für jede *Task*-Anweisung einen eigenen *schedule* festzulegen, erhöhen die Komplexität und erschweren den sicheren Gebrauch dieser Sprachelemente.

5. Einbettung in die Prozeß-Umgebung

Ein Prozeßsteuerungsprogramm ist keine isolierte Aktivität, sondern in mannigfaltiger Weise mit den zu steuernden physikalischen Abläufen verbunden. Die Festlegung dieser Verbindungen erfolgt aufgrund von Betriebssystemkonventionen, von Hardware-Eigenschaften oder von prozeßbezogenen Gesichtspunkten (z. B. Zeitbedingungen). Sie sind im Programm zu berücksichtigen, und ein Übersetzer sollte in der Lage sein, entsprechende Prüfungen vorzunehmen.

a) FORTRAN

Der Umstand, daß die Grundsprache festliegt, wirkt sich für die Beschreibung von Prozeßschnittstellen ungünstig aus.

Die Information zur Auswahl von Geräten und die Konvertierungsvorschriften für die Prozeß-EA werden in **integer**-Größen (in implementierungsabhängiger Weise) abgelegt. Ähnliches gilt für die Benennung von *Tasks* und für Zeitangaben.

Der Datentyp **integer** wird zum Vielzweckdatentyp. Er steht für die PEARL-Datentypen **task**, **clock**, **duration**, **char**(n), **bit**(n) und **integer** und wird außerdem zum Anzeigen von Ereignissen verwendet.

Ein FORTRAN-Übersetzer kann natürlich die jeweils problemgerechte Verwendung von **integer**-Variablen nicht abprüfen, ebensowenig die Konventionen für die Einbettung in die System- und Prozeßumgebung.

b) PROCOL

Der Kommunikationsbereich eines PROCOL-Programms enthält die Deklaration von Variablen und die Beschreibung von Peripheriegeräten mit ihren EA-Schnittstellen.

Peripheriegeräte eines bestimmten Typs werden in PROCOL durch reservierte Namen bezeichnet, z. B. *anin*, *digout* für Analogeingabe bzw. Digitalausgabe. Verschiedene Geräte gleichen Typs werden durch Indizes unterschieden, z. B. *anin*

(2) und *anin* (5). Die Zuordnung zwischen einem bestimmten Gerät und seinem „Hardware-Namen" erfolgt bei der Systemgenerierung. Eine EA-Deklaration im Kommunikationsteil definiert einen Namen für ein Gerät und das Format, das bei Ein- und Ausgabe mit dem Gerät einzuhalten ist.
Charakteristisch für PROCOL sind die Systemanweisungen *(directives moniteur)*, die nicht Bestandteile einer Prozedur oder *Task* sind, sondern sich direkt (im Dialogbetrieb) an das PROCOL unterstützende System wenden, und mit denen man die systembezogenen Größen eines Programms fixiert. Systemanweisungen ermöglichen Vorgaben für die Speicherplanung und Systemgrößen, z. B. Dimensionierung von Pufferbereichen und die Festlegung von *Task*-Eigenschaften (z. B. resident oder nicht). Sie ermöglichen außerdem die Einführung systembezogener Größen des Typs timer (benannt TIM1, TIM2, ...), interrupt (IT1, IT2, ...) und semaphor (SEM1, SEM2, ...). Für timer-Größen wird dabei der Fortschaltzyklus in Sekunden oder Minuten festgelegt, für interrupt-Größen ihre Zuordnung zu einer Menge von Hardware-Niveaus bzw. -Unterniveaus.

Auf diese Weise läßt sich ein fertiges Programm vor seinem Start den jeweiligen Gegebenheiten des Systems und der Prozeßumgebung anpassen.

c) HAL/S

HAL/S enthält allgemeine EA-Anweisungen zum sequentiellen und wahlfreien Zugriff auf Dateien. Sprachelemente zur Beschreibung der Prozeßumgebung sind nicht vorgesehen.

Als system-language-features werden u. a. Zeigervariable *(name-facility)* und spezielle Makros eingeführt. In der Sprache wird dabei lediglich der Aufrufmechanismus festgelegt. Die Definition der Makros bleibt der Implementierung bzw. dem Anwender überlassen.

d) PEARL

Die Verbindungen eines PEARL-Programms mit der System- und Hardware-Umgebung werden im Systemteil beschrieben. Dieser sieht nicht nur eine Schnittstellenbeschreibung vor, sondern auch eine Konfigurationsbeschreibung. Eine systembezogene Größe eines PEARL-Programms wird im Problemteil deklariert. Im Systemteil wird ihr Name einer bestimmten Hardware-Stelle zugeordnet (z. B. Zuordnung zwischen einem Unterbrechungssignal und einer interrupt-Deklaration). Der Systemteil beschreibt nicht nur die Schnittstelle zwischen dem PEARL-Programm und dem Betriebssystem, sondern vielmehr ein Modell der Gesamtkonfiguration: EA-Geräte, Richtung des Informationsflusses, mögliche Übertragungseinheiten und Schnittstellen zwischen Geräten. Durch das Konzept der Datenstationen wird diese Modellierung des Datenflusses und die Beschreibung der Schnittstellen bzw. der Datenkonvertierung zwischen einzelnen Stationen auch innerhalb von PEARL-Programmen ermöglicht.

Wichtig ist, daß der Systemteil mehr enthält als nur eine Schnittstellenbeschreibung. Dadurch wird nicht nur der Dokumentationswert eines Programms erhöht, sondern auch seine Sicherheit, da dem Übersetzer weitergehende Prüfungsmöglichkeiten zur Verfügung stehen.

Die in PROCOL definierten Kommandos zur Herstellung der Systemverbindungen sind in PEARL durch die Beschreibung eines Konfigurationsmodells ersetzt. Dies bedeutet zwar einen gewissen Verlust an Flexibilität, erweitert jedoch den Bereich der vom PEARL-Benutzer beschreibbaren und für die Prozeßsteuerung relevanten Sachverhalte, kann also als ein wichtiger Schritt in die Richtung problemorientierter Programmierung (ohne Spezialisierung auf einen kleinen Anwendungsbereich) verstanden werden.

6. Abschließende Bemerkungen

Einige für die Echtzeitprogrammierung relevanten Fragen wurden ausgewählt, und PEARL wurde bezüglich dieser Fragen mit anderen Sprachen verglichen, soweit dies mit Hilfe der verfügbaren Unterlagen (Sprachbeschreibungen u. ä.) möglich war. Ein detaillierter Vergleich, der Erfahrungsberichte von Implementierern und Anwendern einschließen müßte, konnte mit den vorhandenen Hilfsmitteln nicht durchgeführt werden.

Für zeitkritische Anwendungen, insbesondere für den Fall extrem kurzer Reaktionszeiten, scheint die Kombination von algorithmischer Sprache mit einem Satz von *Task*-Anweisungen und erweiterten EA-Möglichkeiten nicht auszureichen, da einerseits das Streben nach Maschinenunabhängigkeit einen Verzicht auf die genaue Kontrolle des zeitlichen Ablaufs bedeutet, andererseits für *PA*'s zwar ein Aktivierungstermin und eine Priorität angegeben werden können, in keiner der behandelten Sprachen jedoch Terminangaben für die Beendigung einzelner Aktivitäten möglich sind. Diese Probleme bilden neben den früher genannten (siehe 2.) wohl die Hauptursachen dafür, daß *Wirth* in [13] die Echtzeitprogrammierung als "stronghold of assembly programming" bezeichnet. Es ist zu hoffen, daß PEARL dazu beiträgt, diese Festung zu schleifen.

Ein anderer Problemkreis, der nur angedeutet werden kann, betrifft den Weg einer Programmiersprache. Er umfaßt – stark vereinfacht – folgende Phasen:

1. Sprachentwurf und eine Beschreibung, die als Implementierungsgrundlage geeignet ist.
2. Implementierung und erste Phase der Verwendung.
3. Revision der Sprache aufgrund der Erfahrung bei der Implementierung und Anwendung.
4. Stabilisierung und Normung.
5. Verfügbarkeit für einen großen Benutzerkreis (mehrere Implementierungen, verbindliche Sprachbeschreibung und Benutzeranleitungen sind verfügbar).
6. Eventuelle Ablösung durch ein besseres Nachfolgeprodukt.

Bei FORTRAN können [16] etwa folgende Jahreszahlen für die einzelnen Phasen angesetzt werden:

1. 1953–1956.
2. 1954–1958.
3. 1957–1962.
4. 1960–1966.
   (1962–1964: Erarbeitung des ANSI-Standards,
   1966: Verabschiedung durch ANSI,
   1972: Verabschiedung durch DIN)
5. ab 1962 (1961 Fertigstellung der UNIVAC-Implementierung).
6. ab 1966 (PL/I und andere Sprachen).

Die Ablösung von FORTRAN durch andere Sprachen ist bekanntlich nur sehr unvollständig erfolgt. Wie bei anderen Sprachen überlappen sich die einzelnen Phasen sehr stark und es zeigt sich deutlich, daß viele Erfahrungen seitens der Benutzer und Implementierer in die endgültige Version einer Sprache einfließen müssen.

Analogieschlüsse sind gefährlich, deshalb sollen hier keine zeitlichen Zuordnungen für PEARL versucht werden. Es steht jedoch außer Zweifel, daß auch PEARL ähnliche Entwicklungsstufen zu durchlaufen haben wird.

Literatur

[1] PEARL, A proposal for a process- and experiment automation realtime language. KFK-PDV-1, Ges. f. Kernforschung mbH, Karlsruhe 1973.

[2] PEARL Reference Manual. (in Vorbereitung).

[3] PEARL Subset for Avionic Applications Language Description. ESG Elektronik-System-Gesellschaft mbH, München 1976.

[4] ISA – S61.1 Standard. Industrial Computer System FORTRAN. Procedures for Executive Functions, Process Input/Output, and Bit Manipulation. Instrument Society of America, Pittsburgh/USA 1976.

[5] ISA – S61.2 Draft Standard. Industrial Computer System FORTRAN. Procedures for File Access and the Control of File Contention. Instrument Society of America, Pittsburgh/USA 1976.

[6] *Heller, G.* et al: Prozeß-FORTRAN 75, eine Erweiterung von FORTRAN für Prozeßrechner-Anwendungen. Vorschlag des Arbeitskreises „Prozeß-FORTRAN" der VDI/VDE-Ges. f. Meß- und Regelungstechnik.

[7] Spécifications du système PROCOL. STERIA, Le Chesnay, France 1970.

[8] Systeme PROCOL T2000. Notice Technique, STERIA, Le Chesnay, France.

[9] Official Definition of CORAL 66. Ministry of Defense, London 1970.

[10] RTL/2 Language Specifications. ICI Central Research Laboratory, Reading/England.

[11] HAL/S Language Specifications. Intermetrics, Cambridge Mass./USA.

[12] LTR Programmers Manual. Ministère de Defense, Paris 1976.

[13] *Wirth, N.:* MODULA: A language for modular multiprogramming. ETH Zürich 1976.

[14] *Dahl, O. J.; Myrhaug, B.; Nygaard, V.:* SIMULA 67, Common Base Language. Norwegian Computing Center, Oslo/Norwegen 1967.

[15] *Roessler, R.; Schenk, K.:* LTPL-European Group Language Comparison. Physics Institute of the University of Erlangen, Nürnberg 1975.

[16] *Sammet, J. E.:* Programming Language: History and Fundamentals. Prentice-Hall, Englewood Cliffs/USA 1969.

[17] *Martin, T.:* Zur Situation der PEARL-Entwicklung in der BDR. Regelungstechn. Praxis 18 (1976) S. 188–190.

[18] *Martin, T.:* Höhere Programmiersprachen für Prozeßrechner. Regelungstechn. Praxis 18 (1976) S. 251.

[19] *Martin, T.:* PEARL offiziell erschienen. Regelungstechn. Praxis 19 (1977) S. 176–177.

[20] *Martin, T.:* Die Förderung von PEARL im Projekt „Projektlenkung mit Datenverarbeitungsanlagen" des 2. und 3. DV-Programms der Bundesregierung. Regelungstechn. 25 (1977) Heft 10, S. 307–313.

Vergleich von PEARL und Concurrent PASCAL

L. Frevert, R. Rössler

Der folgende Sprachvergleich soll in erster Linie Aussagen über den Gebrauchswert der beiden Sprachen machen. Um ihn möglichst kurz zu halten, ist er in Form eines Streitgesprächs zwischen je einem PEARL-Enthusiasten (EA) und einem PASCAL-Begeisterten (ASS) abgefaßt.

EA: Ich weiß nicht, weshalb wir unsere Sprachen vergleichen sollen. PEARL und CONCURRENT PASCAL wurden doch mit völlig unterschiedlichen Zielsetzungen entwickelt: PEARL für den Aufbau großer Prozeßprogramme, CONCURRENT PASCAL für die Programmierung von Betriebsystemen. Dies wird doch z.B. besonders deutlich darin, daß PEARL ein relativ komfortables Betriebssystem voraussetzt, wogegen PASCAL nur sehr bescheidene Systemfunktionen - beispielsweise für die Ein/Ausgabe - zur Verfügung stellt.

ASS: Sie haben völlig recht - obwohl natürlich bestimmte zusätzliche Mechanismen in PASCAL ohne Schwierigkeiten als Prozedurpaket bereitgestellt werden könnten. Ich denke da vor allem an die Prozeß-E/A...

EA:...abgesehen davon, daß dies zu einer ähnlichen unglücklichen Lösung wie bei Prozeß-FORTRAN führen würde, wäre CONCURRENT PASCAL dann trotzdem noch ungeeignet zur Programmierung großer Systeme. Sie sehen das schon daran, daß Sie keine Programmteile einzeln als schwarze Kästen entwickeln und übersetzen können, wie das mit PEARL - Moduln möglich ist!

ASS: Sie wissen, daß es bestimmte PASCAL-Implementationen gibt, die es gestatten, Programmteile einzeln zu übersetzen und hinterher zu binden...

EA: ... jetzt müßten wir eigentlich über Normung sprechen; ich finde es sehr gut, daß bei PEARL das Nachdenken über die Normung anfing, bevor überhaupt die erste Implementation begonnen wurde...

ASS: ... gut; beschränken wir uns auf den CONRURRENT PASCAL Report von Brinch Hansen. Für das von Ihnen genannte Prinzip, ein Programm in einzelne Teile zu gliedern, die klar definierte Schnittstellen nach außen haben und auf deren Inneres sonst nicht zugegriffen werden kann, haben wir die "Systemtypen" PROCESS, MONITOR und CLASS...

EA:...trotzdem kann aus CONCURRENT PASCAL nie eine Prozeßprogrammiersprache werden. Sie können zwar Nebenläufigkeiten programmieren, aber CONCURRENT PASCAL ist keine Echtzeitsprache: Sie haben nichts, womit Sie auf Interrupts oder Uhrzeiten Bezug nehmen können...

ASS:...dafür bräuchte man nur ein paar Standard-Prozeduren...

EA:...ihre Datentypen ITEGER, REAL, CHAR und BOOLEAN entsprechen in PEARL FIXED ,FLOAT, CHAR (1) und BIT (1), aber Sie haben keine Bit- und Zeichenketten...

ASS:...wir arbeiten mit Zeichenfeldern, und was Sie mit Bitketten machen, können wir mit Operationen auf Mengen (SET) programmieren, die durch Bitketten realisiert werden...

EA:...die Sie aber nicht verschieben können. Aber gut - reden wir von Gemeinsamkeiten: Unsere Prozedur-Konzepte sind praktisch gleich (im Gegensatz zu sequentiellem PASCAL ist Rekursion in CONCURRENT PASCAL ja verboten); bei Feldern und Verbunden bestehen keine prinzipiellen Unterschiede, abgesehen davon, daß PEARL dynamische Felder kennt und die Feldbearbeitung wesentlich komfortabler ist ...

ASS:...dafür bietet PASCAL eleganteren Zugriff auf Strukturkomponenten. In BASIC PEARL gibt es außerdem keine Strukturen von Strukturen...

EA:...was ich auch ärgerlich finde. Aber reden wir doch nicht von diesem etwas knapp geratenen Subset, sondern von FULL PEARL...

ASS:...das von niemandem implementiert worden ist...

EA:...doch, es gibt eine Reihe von Implementationen, die fast den Sprachumfang umfassen...

ASS:...es gibt in PEARL keine varianten Teile in Strukturen...

EA:...doch, über den ONEOF-Konstrukt. Aber um weiter aufzuzählen: IF- und CASE-Anweisung sind praktisch gleich...

ASS:...die CASE-Anweisung ist in PASCAL praktischer, weil ich die Alternativen nicht abzuzählen brauche und weil die Werte beliebig gestreut sein können...

EA:...dafür haben Sie keinen OUT-Zweig...

ASS:...doch, in vielen Implementationen...

EA:...Stichwort Normung! - Wo wir in PEARL mit einer universellen Schleifenkonstruktion auskommen, haben Sie in CONCURRENT PASCAL deren vier; ulkigerweise kann man eine Schleife aber mittendrin nicht ohne GOTO verlassen...

ASS:...in PEARL doch auch nicht...

EA:...trotzdem ist PEARL in punkto Strukturierung bei Schleifen und Verzweigungen überlegen!

ASS: Wieso?

EA: Bei uns endet jede Schleife und Verzweigung explizit mit END; bzw. FIN; dadurch kann ich bei einer Schachtelung die Programmstruktur viel besser erkennen.

ASS: Das sind doch Nebensächlichkeiten!

EA:...die das Programmschreiben unnötig erschweren, genauso wie das Nachdenken, ob nach einer Anweisung ein Semikolon stehen muß oder verboten ist.

ASS: Ach, Sie meinen das Semikolon als Trennzeichen zwischen Anweisungen, und das nach THEN, ELSE und DO immer nur eine Anweisung (oder eine BEGIN...END-Verbundanweisung) stehen darf.

EA: Genau! Ich verstehe nicht, weshalb man diesen ALGOL-Fehler wiederholen mußte. Wenn wir schon von solchen Ärgerlichkeiten sprechen: Bei PASCAL darf ich eine Routine erst aufrufen, dachdem ich sie weiter oben geschrieben habe...

ASS:...dieses Prinzip gilt für alle deklarierbaren Objekte und vereinfacht außerdem den Kompilierer...

EA:...für wen werden Sprachen gemacht, für die vielen Anwender oder die paar Kompilierbauer?

ASS: Leichte Implementierbarkeit kommt dem Anwender dadurch zugute, daß er die Sprache überall verwenden kann; glauben Sie, daß es mehr als eine PEARL-Implementation gäbe, wenn PEARL nicht so gefördert worden wäre? Aber warum ärgert Sie, wenn Sie eine Prozedur erst definieren müssen und dann erst benutzen dürfen?

EA: Weil ich bei top-down-Programmentwicklung zuerst das Programmende schreiben muß, und weil ich bei PASCAL-Programmen erst alle möglichen unwichtigen Details lesen muß, bis ich am Ende die wichtigen Stellen finde. PASCAL ist eine Sprache für bottom-up-Entwickler!

ASS: Streiten wir uns doch nicht über Geschmacksachen! Sie werden aber doch zugeben, daß das Typ-Konzept in PASCAL allgemeiner und schöner ist als in PEARL. Wir können neue Typen durch Aufzählen der möglichen Werte kreieren -TYPE FARBE= (ROT,GELB,GRUEN,BLAU) - und dann z.B. Felder damit indizieren, was die Programme sehr viel lesbarer macht...

EA:...den Typ formaler Parameter dürfen Sie nur als Typnamen angeben - das führt zu unnötigen Querverweisen. Im Übrigen benennt ein guter PEARL-Programmierer seine Index-Konstanten auch...

ASS:...kann aber dabei Fehler machen!

EA:Sie können in PASCAL aber keine neuen Operatoren für Ihre neuen Typen vereinbaren, wie wir in PEARL, und als Ergebnis von Funktionsprozeduren sind bei Ihnen nur einfache Typen zugelassen. Prozeduren dürfen in CONCURRENT PASCAL nicht innerhalb anderer Prozeduren und nicht global vereinbart werden...

ASS:...sondern aus Sicherheitsgründen nur in den Systemtypen MONITOR, PROCESS und CLASS, die eine Ausweitung des Typbegriffes auf ausführbare Programmstücke darstellen.

EA: Prozeduren dürfen bei Ihnen nicht reentrant sein...

ASS:...sie brauchen nicht, weil die Sprachstruktur verbietet, daß zwei Prozesse dieselbe Prozedur gleichzeitig benutzen.Alle die vielen Möglichkeiten, die Sie in PEARL haben, führen doch nur dazu, daß die Wechselwirkungen zwischen den konkurrierenden Prozessen unüberschaubar und fehleranfällig werden. Ich gebe zu, daß Sie gewisse Sicherheitsvorkehrungen, treffen können, indem Sie z.B. den Zugriff auf eine globale Variable auf Lesen beschränken können...

EA:...diese vielen Möglichkeiten spiegeln die Vielfalt der Anforderungen wider, die an eine Prozeß-Programmiersprache gestellt werden. Wir haben jedoch in PEARL alle Hilfsmittel, um Nebenläufigkeiten sicher programmieren zu können.

CONCURRENT PASCAL hingegen legt dem Programmierer derart viele Restriktionen auf,daß er manche Probleme gar nicht lösen kann!

ASS: Geben Sie mir bitte Beispiele!

EA: Gern. Ich habe hier einen PEARL-Modul mitgebracht, in dem Daten verwaltet werden, die von mehreren Tasks gleichzeitig benutzt werden. Tasks, die nur lesen, dürfen alle gleichzeitig lesen - Tasks, die Daten verändern, haben exklusiven Zugriff. Die Daten selbst sind lokal zum Modul. Die Leser und Schreiber können nur über Aufrufe globaler Prozeduren an die Daten ran.

Alle Koordination erfolgt im Modul. Diese Problemstellung taucht übrigens auch bei der Implementation von Betriebssystemen auf!

ASS: Sie haben also für die Verwaltung von Daten, die mehreren Tasks gemeinsam sind, einen "MONITOR" geschrieben, wie er in CONCURRENT PASCAL vorgesehen ist!

EA: Richtig! Wenn ich CONCURRENT PASCAL richtig verstehe, können Prozesse niemals direkt miteinander in Wechselwirkung treten, sondern immer nur über MONITORe, die so strukturiert sind wie mein PEARL-Modul. Meine globalen Prozeduren wären bei Ihnen "ENTRY-Routinen"...

ASS:...ich sagte ja schon, daß wir statt Ihrer Moduln unter anderem MONITORe haben...

EA:...der Witz ist nur, daß sich immer nur ein Prozeß im MONITOR aufhalten darf. Wenn Sie aus meinem Modul also einen MONITOR machen, kann sich immer nur ein Leser in ihm aufhalten und die Daten lesen; das Leser-Problem ist für Sie unlösbar!

ASS: Richtig! Jeder Leser muß sich seine Daten in einen eigenen Puffer kopieren und dann den MONITOR schnellstens verlassen, damit der nächste Leser oder Schreiber drankommen kann.

EA: Sie können also auch keinen Wechselpuffer programmieren?

ASS: Doch - allerdings nur mit drei MONITORen. Dafür kann ich aber Nebenläufigkeiten viel sicherer programmieren als Sie in PEARL. Wir realisieren kritische Abschnitte, indem wir sie in einen MONITOR verlegen. Allein dadurch ist schon garantiert, daß es nicht zu Kollisionen kommen kann.

EA: Können Prozesse auch angehalten werden?

ASS: Ein Prozeß kann sich in einem MONITOR selbst anhalten, indem er sich in eine QUEUE begibt und dadurch den MONITOR verläßt. Ein anderer Prozeß kann dann später beim Verlassen desselben MONITORs den angehaltenen Prozeß zur Fortsetzung bringen.

EA: Und wenn mehrere Prozesse warten müssen?

ASS: Dann brauche ich einen ARRAY .. OF QUEUE, da immer nur ein Prozeß in eine QUEUE paßt.

EA: Die Bezeichnung "QUEUE" ist etwas irreführend. Ich würde Ihren ARRAY .. OF QUEUE Wartenschlange nennen. Wie wird letztere verwaltet?

ASS: Die Verwaltung muß ich mir selber schreiben.

EA: Sie müssen sich Ihre Wartenschlangenverwaltung also selbst schreiben. Das habe ich in PEARL nicht nötig, weil die in der Semaphor-Verwaltung enthalten ist.

ASS: Dafür kann ich mir aber meine Strategie wählen, z.B. FIFO oder nach Priorität.

EA: Sie haben aber doch gar kein Schlüsselwort für Priorität!

ASS:...brauche ich auch nicht. Ich kann meine Prozesse parametrisieren, und einer der Parameter könnte als Priorität gedeutet werden.

EA: Ist das nicht sehr umständlich, wenn Sie überall erst eine Wartenschlangenverwaltung in die MONITORe einbauen müssen?

ASS: Überhaupt nicht! Ich schreibe die Verwaltung (d.h. alle zur Verwaltung nötigen Routinen und Daten) einmal als Muster und definiere sie als Systemtyp CLASS. Die vielen typmäßig gleichen Wartenschlangenverwaltungen erhalte ich dann durch Vereinbarung von "Variablen" dieses Typs. Das geht genauso, wie Sie sich Datenaggregate mit gleicher Struktur durch eine Typdefinition und Variablenvereinbarungen über diesem Typ schaffen.

EA: Sie schaffen sich also Programmstücke gleicher Art dadurch, daß Sie einen Prozeß (bzw. einen Monitor oder eine Klasse) als Typ definieren und dann (mehrfach) Variable dieses Typs vereinbaren?

ASS: So ist es! Alle Prozesse, Monitore und Klassen werden beim Programmstart automatisch initiiert, und das Programm läuft.

EA: Alle Tasks (wie wir Ihre Prozesse nennen) werden also beim Programmstart aktiviert (in der PEARL-Terminologie). Wie werden Prozesse beendet?

ASS: Ein CONCURRENT PASCAL - Prozeß kann sich nur selbst beenden, möglicherweise infolge einer Nachricht, die ihm ein anderer Prozeß in einem Monitor hinterlassen hat.

EA: Kann er wieder gestartet werden?

ASS: Nein, deshalb wird er normalerweise auch nicht beendet.

EA: Sie können den automatischen Wiederanlauf eines zusammengebrochenen Betriebssystems also nicht programmieren?

ASS: Ein in CONCURRENT PASCAL geschriebenes Betriebssystem bricht nicht zusammen!

EA:Für die Spielysteme aus Lehrbüchern mag dies zutreffen. Sie sagten vorhin, daß Sie anstelle von Moduln die Systemtypen MONITOR, PROCESS und CLASS haben. Wenn ich Sie richtig verstehe, ist ein Monitor ein Modul (um mit meinem Begriff zu sprechen), der die kritischen Abschnitte mehrerer Tasks enthält und entsprechende Prozedureinsprünge zur Verfügung stellt?

ASS: So ist es! Prozesse können Monitor-Routinen aufrufen aber nicht umgekehrt. Globale Daten gibt es überhaupt nicht.

EA: Dieses Prinzip kann ich simulieren, indem ich für jede Task einen eigenen Modul schreibe, zur Ausführung kritischer Abschnitte in einen "Monitor-Modul" springe und globale Daten verbiete...

ASS:...richtig! Eine Klasse wäre dann ein Modul, dessen globale Prozeduren nur von genau einem anderen Modul benutzt werden.

EA: Wenn ich meine Programme so gliedere, sind sie doch genauso sicher wie Ihre!?

ASS: Im Prinzip ja. Nur ist diese Art der Programmierung in PEARL nirgends vorgeschrieben. Mit Ihren Sprachmitteln dürfen Sie beliebig viel Unsinn machen!

EA: Selbstverständlich legen mir größere Freiheiten in der Benutzung von Sprachmitteln auch höhere Verantwortung bezüglich ihrer vernüftigen Handhabung auf. Ich bestreite nicht, daß ich beim Schreiben sicherer PEARL-Programme ähnliche Einschränkungen auf mich nehmen muß, wie sie in CONCURRENT PASCAL vorgesehen sind; ich kann aber genau da von ihnen abweichen, wo sie unnötige Komplikationen verursachen oder ein Problem unlösbar machen.

ASS: Sie werden CONCURRENT PASCAL also gewissermaßen für einen PEARL-Programmierer als Vorbild für gute Strukturierung von Programmen ansehen?

EA: So ist es! Da aber bei der Programmierung industrieller Prozesse häufig Ausnahmesituationen auftreten, die mit diesen "sauberen" Methoden nur schlecht oder gar nicht bewältigt werden können, brauche ich für diese Problemstellung einfach die größere Flexibilität von PEARL.

Schrifttum:

zu PEARL:

Werum, W.; Windauer,H.
PEARL - Process and Experiment Automation Realtime Language, Beschreibung mit Anwendungsbeispielen; Vieweg

FULL PEARL - Language Description
KFK-PDV 130, 1977

Basic PEARL - Sprachbeschreibung
KFK-PDF 121, 1977

zu CONCURRENT PASCAL:

Brinch Hansen, P.:

The Programming Language Concurrent Pascal IEEE Transactions on Software Engineering
VOL.SE-1,No.2, June 1975, pp 199-207

Nachdruck aus dem Do-Post

Höhere Programmiersprachen im Programmentwicklungsprozeß

P. Elzer

Höhere Programmiersprachen sind nicht das einzige Mittel zur Lösung von Softwareproblemen. Da sie aber Auswirkungen auf mehrere Stufen des Softwareentwicklungsprozesses haben und durch ihren "stilbildenden" Charakter das Gesamtsystem stark beeinflussen, sollen sie hier stellvertretend für andere Programmierwerkzeuge betrachtet werden. Da es außerdem sehr viele höhere Programmiersprachen gibt, soll die Auswahl auf die beiden für systemtechnische Entwicklungen bedeutsamsten, nämlich "PEARL" und "ADA" beschränkt werden.

Die derzeit existierenden Softwareprobleme kann man grob einteilen in Massenprobleme, die aus einer übergroßen Nachfrage resultieren, der kein entsprechendes Produktionspotential gegenübersteht, und Qualitätsprobleme, die durch die wachsende Komplexität der Systeme entstehen. Abhilfemaßnahmen haben also zwei Zielen zu dienen:

- Erhöhung der Produktivität des Entwicklers
- Erhöhung der Arbeitsqualität.

Hierbei muß gleichzeitig auf zwei Ebenen vorgegangen werden:

- durch Managementmaßnahmen, wie etwa die geeignete Zergliederung des Gesamtproblems, Teamorganisation oder Entwurfs- und Entwicklungsrichtlinien;
- durch technologische Maßnahmen, die hauptsächlich in der Verfügbarmachung geeigneter Programmierwerkzeuge, einer "Software-" oder "Programmierumgebung" bestehen. Diese Klasse von Maßnahmen ist vergleichbar mit den klassischen Investitionen in Maschinen und Ausrüstung.

Diese Maßnahmen sollten folgende Hauptwirkungen haben:

- Abbau der Kommunikationslücke zwischen Anwendungsexperten und Programmentwicklern;
- Verringerung der für den Entwickler sichtbaren Komplexität des Systems;
- Entlastung von Routineaufgaben und Detailentscheidungen, die nicht zur Problemlösung beitragen, um geistige Kapazität für den eigentlich ingenieurmäßigen kreativen Anteil der Arbeit freizusetzen.

Zu einer solchen Programmierumgebung gehören etwa:

- Entwurfsverfahren
- Höhere Programmiersprachen und ihre Übersetzer
- Test- und Integrationshilfen
- Projektdatenbasis und Dokumentationsverfahren.

ENTSTEHUNG VON PEARL

PEARL (=Process and Experiment Automation Realtime Language) wurde in der Bundesrepublik in Zusammenarbeit von Rechnerherstellern, System- und Softwarehäusern, anwendender Industrie und Forschungsinstituten entwickelt. Sowohl die Entwicklung der Sprache selbst als auch die von Übersetzern dazu wurden vom BMBW (Bundesministerium für Bildung und Wissenschaft) und vom BMFT (Bundesministerium für Forschung und Technologie) gefördert. Der erste vollständige Sprachentwurf erschien 1973. 1978 erschien der Normentwurf DIN 66253, Teil 1, Basic PEARL. Dieser legt den Teil der Sprache PEARL fest,

der praktisch allen zur Zeit verfügbaren Implementationen gemeinsam ist. Der Gesamtumfang von PEARL wurde Anfang 1981 als DIN Normentwurf veröffentlicht.
Im Dezember 1979 wurde eine Organisation zur weiteren Betreuung von PEARL gegründet, der PEARL-Verein e.V. Seine Geschäftsstelle befindet sich im VDI-Haus in Düsseldorf.
PEARL-Programmiersysteme werden in der Bundesrepublik unter anderem von ATM, BBC, DEC, Dornier, Krupp-Atlas, mbp und Siemens angeboten.

EINSATZBEREICH VON PEARL

PEARL ist bewußt als eine Programmiersprache für den anwendenden Ingenieur konzipiert. Sie erlaubt dem Anwendungsfachmann, einen Großteil seiner Probleme selbst für den Computer aufzubereiten. Sie stützt sich auf existierende Betriebssysteme und andere Softwarehilfen und erlaubt so die Benutzung vom Rechnerhersteller standardmäßig gelieferter Programmierumgebungen. Es ist jedoch auch möglich, in sich geschlossene, ganz auf PEARL ausgerichtete Programmierumgebungen zu entwickeln. Dies führt sogar zu besonders guten Einsatzresultaten, wie später am Beispiel des bei Dornier entwickelten PEARL-Programmiersystems, gezeigt werden soll.
PEARL unterstützt vom Ansatz her also vorwiegend den Abbau der Kommunikationslücke zwischen Anwendungsspezialist und Programmentwickler und erlaubt eine Verringerung der Komplexität der Problembeschreibung auf relativ hohem, d.h. anwendungsorientiertem Niveau.

WESENTLICHE ELEMENTE VON PEARL

Die Beschreibung der Hardwarekonfiguration im sogenannten "Systemteil" beruht auf dem Konzept allgemeiner Netzwerke. Eine Programmzeile des Systemteils beschreibt ein Teilstück des Weges, den Information im Gesamtsystem nehmen kann, nämlich jeweils eine direkte Verbindung zwischen einer informationserzeugenden und der zugehörigen informationsverbrauchenden Komponente des Hardwaresystems. Vom Übersetzer werden aus diesen Teilstücken die gesamten Datenwege zusammengesetzt und auf ihre Realisierbarkeit geprüft.

Die Realzeitelemente stellen die herausragende Stärke von PEARL dar.

Sie erlauben:
- die Festlegung parallel ablauffähiger Programmstücke (Tasks),
- die Beeinflussung ihres Ablaufes durch: Start, Beendigung, Anhalten, Wiederanlauf,
- ihre Kopplung an externe Zeitbedingung oder Spontanereignisse (Interrupts),
- die gegenseitige Synchronisation solcher parallel ablaufender Programmstücke.

Die Ein-/Ausgabe von PEARL ist auf die Bedürfnisse der Automatisierungstechnik ausgerichtet. Ein allgemeines Modell erlaubt in Ergänzung zur statischen Beschreibung von Datenwegen im Systemteil ihre funktionelle Beschreibung durch Datenstationen ("Dations") und dazwischengeschaltete Interfaces. Die eigentlichen Ein-/Ausgabeanweisungen von PEARL sind anwendungsorientiert und unterteilen sich in folgende Klassen:
- zeichenorientiert, wie etwa bei FORTRAN
- wertorientiert, d.h. Transfer von Daten ohne Wandlung

Der algorithmische Teil von PEARL entspricht weitgehend den Möglichkeiten klassischer Programmiersprachen. Er enthält Elemente zur Verarbeitung von Daten der Arten (Typen): ganzzahlig, Gleitkomma, Bitketten, Zeichenketten. Diese können wiederum in den Modi "Konstante" oder "Variable" vorkommen und zu "Feldern" und "Strukturen" gruppiert werden. Über die Möglichkeiten der meisten klassischen Programmiersprachen hinaus gehen die Datentypen "Uhrzeit" und "Zeitdauer" und die Möglichkeiten zur Definition eigener Typen durch den Entwickler.
Die modulare Struktur von PEARL-Programmen bietet folgende Vorteile:

- Erleichterung von Test und Integration dadurch, daß nur einzelne Teile des Gesamtprogramms neu übersetzt werden müssen.
- Die Trennung von "Systemteil" und eigentlichem Programm ("Problemteil") erleichtert die Wiederverwendung von Programmen, falls sich nur die Hardwarekonfiguration ändert.

DORNIER-PEARL

Bei Dornier System wurde im Auftrag des BMVg ein PEARL-Programmiersystem entwickelt, das speziell auf die Anforderungen rechnergestützter Systeme im Bereich der Luft- und Raumfahrt und der Wehrtechnik zugeschnitten ist. Es wurde entsprechend folgenden Anforderungen entworfen:

- der erzeugte Code muß möglichst effizient sein,
- der Test integrierter Systeme muß unterstützt und erleichtert werden
- eine Programmbibliothek zur Entwurfsunterstützung muß aufgebaut werden können
- Systeme mit verteilten Rechnern müssen in PEARL programmiert werden können
- das Übersetzungssystem muß weitgehend unabhängig sowohl von der jeweiligen Übersetzungsmaschine als auch vom Zielrechner sein.

Das Dornier-PEARL Übersetzungssystem steht bisher für die Zielmaschinen Dornier-MUDAS 432 und AEG-Telefunken 8020 zur Verfügung. Eine Version für den Mikroprozessor INTEL 8086 ist in Vorbereitung. Es ist ablauffähig auf den Übersetzungsrechnern PDP-11/70 und AEG 8020.

ENTSTEHUNG VON ADA

"ADA" ist eine vom US-Verteidigungsministerium finanzierte Entwicklung. Der Name der Sprache ist nicht wie üblich ein Akronym, sondern der Vorname der "ersten Programmiererin der Welt": Augusta Ada Byron (1816-1862), später Lady Lovelace, die zeitweilig mit Charles Babbage, dem Erfinder der ersten programmgesteuerten Rechenmaschine der Welt, zusammenarbeitete.

Das Sprachentwicklungsprojekt begann 1975 unter dem Namen "DOD 1". In einer Reihe von Dokumenten wurden die technischen Anforderungen an die Sprache niedergelegt. Die Version "Ironman" vom Januar 1977 war die Grundlage für eine Ausschreibung der eigentlichen Sprachentwicklung. Zunächst wurden vier Sprachentwürfe im Wettbewerb entwickelt. Nach eingehenden Bewertungen auf internationaler Basis wurde im Mai 1979 der Entwurf von CII-Honeywell Bull ausgewählt und in den "Sigplan-Notices" der Association of Computing Machinery (ACM) veröffentlicht. Nach eingehenden Tests wurde die Sprache im Juli 1980 festgeschrieben. Testimplementationen sind in Arbeit. Erste Produktionscompiler werden für 1983 bis 1984 erwartet.

EINSATZBEREICH VON ADA

Ada wurde vorwiegend für die Programmierung großer militärischer Systeme entwickelt. Es wurde Wert darauf gelegt, daß möglichst wenig Unterstützungssoftware seitens des Rechnerherstellers zur Verfügung gestellt werden muß und daß möglichst viel davon in Ada selbst geschrieben werden kann. Großer Wert wurde auf Prüfarbeit und Zuverlässigkeit der entstehenden Programme gelegt. Übersetzbarkeit auf dem Zielrechner selbst war kein wesentliches Entwurfskriterium. Es wird vielmehr eine Entwicklungsumgebung angestrebt, in der der im Anwendungssystem selbst enthaltene Zielrechner ("embedded computer") von einem großen Entwicklungsrechner aus programmiert wird.

WESENTLICHES ELEMENT VON ADA

Entsprechend den Anforderungen wurde auch das Schwergewicht bei der Entwicklung auf sicherheitsrelevante Sprachelemente gelegt. Das Konzept der Typbildung und Modularisierung entspricht neuesten Forschungsergebnissen. Im Gegensatz zu den meisten anderen Programmiersprachen umfaßt die Definition eines "Typs" in Ada auch die darauf anwendbaren Operationen und den Wertebereich, der angenommen werden kann. Die Definition eines Moduls erlaubt die Spezifikation der Teile, die von außen beeinflußt werden können und solcher, die geschützt werden sollen. Damit sind weitgehende statische Prüfungen möglich, aber meist auch ein entsprechender Schreibaufwand verbunden.
Der Fehlerbehandlungsmechanismus erlaubt gezielte Reaktionen auf Programmfehlverhalten wie z.B. Multiplikationsüberlauf, Division durch Null, Bereichsüberschreitungen, etc. Die Regeln, nach denen betroffene Programmteile abgeschaltet oder Fehler weitergereicht werden dürfen, sind genau festgelegt.

Der Synchronisationsmechanismus, der die Kommunikation zwischen den natürlich auch in Ada vorhandenen parallel ausführbaren Programmteilen (Tasks) ermöglicht, wurde auch unter dem Sicherheitsaspekt entworfen. Es werden hier im Gegensatz zu den sonst üblichen Modellen nicht Daten zwischen Tasks ausgetauscht, sondern vorher definierte Programmteilstücke gemeinsam ausgeführt.

DIE SOFTWAREUMGEBUNG VON ADA

Bereits zu Beginn der Ada-Entwicklung war klar, daß der Erfolg der Sprache von der Verfügbarkeit einer entsprechenden Softwareumgebung abhängt. Es wurden deshalb schon weitgehend parallel zur Entwicklung der technischen Anforderungen an die Sprache auch solche an die Softwareumgebung erarbeitet. Im Rahmen einer Beobachterfunktion für die Bundesregierung arbeitete der Autor an den ersten Dokumenten dieser Art mit ("Pebbleman" und Pebbleman revised"). In diesen Dokumenten werden sowohl Organisationsvorschläge für die Betreuung der Sprache gemacht als auch Forderungen an die Komponenten der Softwareumgebung aufgestellt. Sie soll u.a. Compiler, Codegeneratoren, intelligenten Editor, Datenbasis, Testhilfen und Managementhilfen umfassen. In der neuesten Version dieses Dokumentes ("Stoneman") werden auch Vorstellungen entwickelt, wie eine solche Softwareumgebung stufenweise, aufbauend auf einer Minimalausstattung, entwickelt werden kann.

Der folgende Beitrag ist die Niederschrift eines Vortrags, auf der Prozeßrechnertagung '81 im März 81 in München.

Comparison of Languages (Coral, PASCAL, PEARL, Ada)

H. Sandmayr

Abstract. The facilities of some languages used for realtime applications are summarized and compared. It is not intended to give a recommendation for the use of one of these languages. Instead a set of different approaches is presented which provides an overview.

SOME REMARKS ON LANGUAGE DEVELOPMENT

Before discussing details of the languages considered in this paper some remarks on language development seem to be appropriate. The development of each of the languages was influenced by the state of the art at the time of their design.

In the development of programming languages three phases can be distinguished [We 76]:

- discovery and description of programming concepts and basic implementation techniques in the 1950´s,
- elaboration and analysis of this concepts, development of models, abstractions, and theories concerning languages in the 1960´s, and
- emphasis on the engineering approach in the software development technology (in the 1970´s).

In the first phase languages were regarded as tools to facilitate the formulation of programs;this phase includes the development of FORTRAN, ALGOL 60, COBOL, and many other languages.

The languages development in the second phase, e.g. PL/I, SIMULA 67, and ALGOL 68 are elaborations or generalizations of earlier languages. PL/I, for example, combines features of FORTRAN, ALGOL 60, and COBOL and attempts to replace different languages by one. ALGOL 68 is a systematic generalization of the features of ALGOL 60.

These attempts to achieve greater power of expression led to excessively elaborated and very complex languages. In the third phase we encounter a return to the essentials, to simple languages which support structured programming, modularity, and verification efforts. Examples of such methodology-oriented languages are PASCAL, and MODULA.

The development of real-time languages is embedded in the above mentioned development. CORAL an PEARL are languages developed in the second phase mentioned above. CORAL (1964, 1966) is an attempt to combine features of ALGOL 60, FORTRAN and macroassembly languages into an efficient language suited for real-time applications on small machines. PEARL (1971) follows the ideas of ALGOL 68 and of PL/I, and adds further multiprogramming facilities. PASCAL (1971) is a product of the third phase whereas ADA (1979) is an attempt to unify, elaborate and generalize the features of languages of the third phase.

DESIGN GOALS

In this section a short summary of the design goals of the different languages is given, Some goals were never stated explicitly but were implied by the time of the design.

CORAL has been designed for the implementation of systems on small, dedicated computers to replace machine code in this type of systems. The specific requirements were:

- compilers must be small enough to run in the production systems or standby system, and
- the language must allow to make full use of individual machine hardware and any other special facilities provided for example by an operating system. At the same time, the implementation must be possible on a wide range of machines.

PASCAL was designed and implemented with the following principal aims [Wi 71]:

- To make a notation available in which the fundamental concepts and stuctures of programming are expressible in a systematic, precise and appropriate way.
- To make a notation available which takes into account the various new insights concerning systematic methods of program development.

- To demonstrate that a language with a rich set of flexible data and program structuring facilities can be implemented by an efficient and moderately sized compiler.

System programming aspects were only considered in so far as necessary for compiler developments.

PEARL has been designed as high level language for industrial process control applications. It should provide multiprogramming facilities tailored to the particular application area, and facilities for a suitable description of the interaction between processes and environment. The syntax of PL/I has been adopted for the algorithmic part of the language.

In contrast to CORAL, machine independence and protability are considered more important than efficiency

ADA has been designed with three overriding concerns [AD 79]:

- a recognition of the importance of program reliability and maintenance,
- a concern for programming as human activity, and efficiency.

The intended application range are "embedded computer systems", i.e. software systems which are embedded into an existing physical environment, comparable to process control applications. The language should be used by application prgrammers.

PROGRAM STRUCTURES AND COMPILATION UNITS

A CORAL program consistes of segments and communicators. Communicators allow communication between segments and allow access to items which exist outside the program.

```
program_name
'COMMON' name (specific. of data items, labels,
              switches, procedures, segments and
              overlays);
'LIBRARY' (specification and eventual renaming of
          library routines used in program);
'EXTERNAL' external_symbol_name (List data items);
'ABSOLUTE' (specification of absolute adresses of
           data items);

segment_name
'BEGIN' segment_declarations;
   statement_sequence
'END';

further_segments
'FINISH'
```

Fig. 1: Structure of a CORAL Program

Independent compilation of segments is possible. The start address of a program can be any segment or label in a segment defined in a COMMON. It must be specified explicitly by means of an 'ENTER' definition.

CORAL has adopted the ALGOL 60 block structure, the scope rules and visibility rules for identifiers, except for macro identifiers whose definitions are valid until they are deleted explicitly.

The structure of a PASCAL program is shown in the following figure.

```
PROGRAM name (file_parameters);
  LABEL label_definitions
  CONST constant_definitions
  TYPE  type_definitions
  VAR   variable_declarations
  Procedure_and_function_declarations
BEGIN statement_sequence
END.
```

Fig. 2: Structure of a PASCAL program

In PASCAL the program is the compilation unit; however many implementations allow independent compilation of procedures.

Blocks are bound to procedures, functions, and programs. There exist no anonymous blocks as in ALGOL 60.

A PEARL program consistes of a set of modules; modules as show in Fig. 3 are compilation units and cannot be nested. A module consists of a system part and/or a problem part. The system part defines the relation of the program to elements of the computer systems and of the technical process. The problem part contains algorithms solving the given problem.

```
MODULE (name);
  SYSTEM; description_of_configuration
  PROBLEM; specification_of_imported_objects
           declaration_of_objects
           declaration_of_tasks/procedures

MODEND;
```

Fig. 3: Structure of a PEARL Module

The scope of objects is a module or a block. The scope of objects declared on the module level can be extended to other modules by declaring such objects as global and specifying them in other modules as imported objects.

Modules cannot be nested in contrast to procedures and tasks.

An ADA program can be composed of program units:subprograms and modules. Modules are either tasks, task types, or packages. A package is a set of logically related types, objects, and operations. Units can be nested, i.e. a task can contain subtasks and packages, and a package can contain local tasks as well as packages.

```
 PACKAGE
{        }   module_name  IS
 TASK
  declaration of objects and operations
  visible to environment
PRIVATE
  declaration of structural details
  of exported objects
END module_name;

 PACKAGE
{        }   BODY module_name  IS
 TASK
  declaration of types, objects, and operations
BEGIN statement_sequence
EXCEPTION list_of_exception_handlers
END module_name;
```

Fig. 4: Structure of an ADA Module

In general, a unit consists of two parts, the specification and the body. The entities declared in the specification part are visible outside the unit and can be used by outer units.

Structural details of some declared types or objects may be irrelevant to their use outside a module. Declaring them in the private section prevents other units to make use of this information. Thus, the scope of an entity declared in the declaration part of a program unit is the range from the entity´s declaration to the end of the scope of the program unit containing the declaration.

The scope of an entity declared in the body of a unit or in a block is the respective program unit, or more precisly, the range between an entity´s declaration and the end of the unit containing the declaration.

There exists no explicit feature or restriction for the import of entities which are defined in an outer program unit. Every object whose name is visible at the point of the unit´s declaration is implicitly imported and can be used within the unit, unless the name is hidden by a local redefinition. However, in contrast to the usual scope rules redefinition of an identifier in an inner block does not necessarily hide its definition in the outer block. An identifier denoting more than one entity is said to be overloaded. When using such an overloaded identifier the context must allow to determine which definition is to be used.

Units of compilation are module declaration, module bodies, subprogram declarations, and subprogram bodies. PRAGMAs allow to control the compilation process, e.g. specification of configuration, or optimization criteria. By means of a context specification the set of units visible by the compilation unit can be specified.

Subprograms

Subprograms (procedures and functions) can be declared in all the languages. PEARL and ADA allow to specify whether a subprogram should be expanded inline at each call or whether the usual subroutine mechanism is to be used. In Ada inline subprograms can be used to included assembly code in a program.

Subprograms can have parameters in all the languages. In CORAL, PEARL, and PASCAL, variables can be passed either by value or by reference. In ADA, three parameter modes are provided:

IN or constant,
OUT or result, and
IN OUT or update parameters.

For IN parameters default values can be defined.

Different objects are accepted as parameters:

CORAL: values (represented by expressions)
variables (arrays and tables by reference only)
procedures

PEARL: every object except tasks and modules

PASCAL: values, variables, and subprograms

ADA: values and variables only,
(however variables can be of task types)

Actual parameters are associated to the formal ones by their positional order. In addition, ADA also permits an association by name, i.e. formal and actual parameters are explicitly associated in the actual parameter list.

TYPES AND STRUCTURES

The following tables show a summary of the types and structures provided by the compared languages.

Table 1: CORAL 66 and PASCAL

	CORAL 66	PASCAL
basic types	INTEGER FLOATING FIXED (+ scale)	INTEGER REAL BOOLEAN CHAR enumeration types subrange types pointer
structures	ARRAY (static) TABLE	ARRAY (static) RECORD (variants) SET FILE
remarks	structures cannot be nested one or two dimensional arrays only	structures can be nested
allocation of variables	in common or stack at address determined by compiler or specified in program overlays possible	in heap or stack at address determined by runtime system or compiler resp.
access to variables	by name or absolute address aliasing possible (parameter,overlay, anonymous reference)	by name or reference if dynamically allocated aliasing possible (parameter)

Table 2: PEARL and ADA

	PEARL	ADA
basic types	FIXED FLOAT BIT CHAR REF CLOCK, DURATION SEMA, BOLT	INTEGER (RANGE) FLOAT (DIGITS) fixed point (DELTA) BOOLEAN CHARACTER enumeration derived types subtypes ACCESS DURATION
structures	array (dynamic) STRUCT bit chain DATION	ARRAY (dynamic) RECORD (variants) STRING
remarks	structures can be nested	structures can be nested types can be parameterized
representation specification	no. of bits for numerical values	range, absolute and relative accuracy for numerical values repr. of enumeration types record types
allocation of variables	at addr determined by compiler, RESIDENT attribute indicates fast access	at addr determined by compiler or runtime system for dynamically allocated objects; explicit address and spec possible
access to variables	by name or reference aliasing possible (parameters and references)	by name or reference if dynamically allocated no aliasing (except for dynam. alloc. variables)

The type concept provided by CORAL is rather poor. There are only numeric types and two structures. Arrays are restricted to vectors and matrices of numerical values. Tables, the equivalent to a vector of records, require references to the internal representation of data for the definition of fields.

In contrast to the remaining languages new types cannot be named except by the general macro facilitiy provided in the language.

PEARL adheres to the type concept of PL/I and adds some simple types for multiprogramming purposes and time specifications. Particular features are provided to define the interface to computer and process peripherals (DATION).

PASCAL and ADA have a strong type concept; the type of any object is determinable during translation and therefore the set of applicable operations is known. New types can be defined and named. Type equivalence is related to name equivalence, a solution which is not totally realized in PASCAL.

PASCAL's subrange types are elaborated to subtypes in ADA. The derived type in ADA even allows to distinguish types with formally identical set of values and operations (but eventually different representation).

The PASCAL set structure is not available in ADA. Files are provided in ADA in a predefined package.

References to variables exist in all languages in some form. CORAL and PEARL allow references to any variables with the inherent problem of references to objects in a no longer existing block. In PASCAL and ADA there exist only references to dynamically allocated and (explicitly) deallocated objects.

STATEMENTS

Overview

The following tables list the statements provided in the different languages. The most detailed version is shown for statements allowing several variants.

Table 3: List of CORAL 66 and PASCAL statements

CORAL 66	PASCAL
location := expression GOTO identifier procedure call ANSWER expression	variable := expression GOTO number procedure call
BEGIN declarations statements END	BEGIN statements END
IF condition THEN simple_statement ELSE statement	IF boolean_expression THEN statement ELSE statement
	CASE expression OF constant: statement; ... END
FOR variable := expr STEP expr UNTIL expr DO statement	FOR variable := expr {TO / DOWNTO} expr DO statement
FOR variable := expr WHILE condition DO statement	WHILE boolean_expr DO statement
	REPEAT statements UNTIL boolean_expr
CODE BEGIN assembler_statements END	WITH record_variable DO statement
i/o not defined	predefined i/o procedures

Neither CORAL nor PASCAL provide facilities for multiprogramming. However, tasks can be represented by programs and the procedure call mechanism can be used to access operating system functions, especially functions allowing interprocess (interprogram)communication.

Table 4: List of PEARL and ADA Statements

PEARL	ADA
variable := expression; GOTO identifier;	variable := expression; GOTO identifier; EXIT loop_identifier WHEN condition;
CALL identifier; RETURN(expression); INDUCE signal identifier	proc_call RETURN expression; RAISE exception;
BEGIN declarations statements END;	DECLARE declarations BEGIN statements EXCEPTION exc_handler END;
ON signal_id: statement;	
IF condition THEN statements ELSE statements FIN;	IF boolean_expression THEN statements ELSIF boolean_expression THEN statements ELSE statements END IF;
CASE expression ALT statements ... OUT statements FIN;	CASE expression OF WHEN choice => statements ... END CASE;
FOR variable FROM expr BY expr TO expr WHILE condition REPEAT declaration_list statements END;	{FOR variable IN range / WHILE boolean_expr} LOOP statements EXIT WHEN boolean_expr; statements END LOOP;
OPEN, CLOSE, PUT, GET, TAKE, SEND + formatting facilities, READ, WRITE	predefined packages defining i/o types and operations
statements for multiprogramming see next section	

The statements which control the sequential flow of instructions in the different languages provide almost identical possibilities and differ only respect to their syntax. This difference can however influence the style of a program; note for example the difference between the overloaded loop statement in PEARL and the set of simple loop statements in PASCAL, or the difference between the not very readable CASE statement in PEARL and its counterpart in ADA.

Input/Output-Facilities

CORAL follows ALGOL 60 and gives no definition of input-output facilities. This allows an implementor to use directly the mechanisms provided by an underlying operating system. This solution can be very efficient but does certainly not enhance portability of a program.

PASCAL bases its i/o on the file structure and a set of predefined procedures. The procedures for text i/o are treated by the compiler in a special way. They accept an optional file parameter, a varying number of parameters of different types, and a special field width seperator. The file structure with the basic procedures PUT and GET requires in general a simple runtime interface to the underlying system. Initialisation of this interface is assumed to be implicit. Experience shows that many PASCAL implementations provide further procedures allowing access to special file system facilities, e.g. random access. This is the main source of difficulties when moving a PASCAL program from one installation to another.

Low level- or process-i/o is not defined in the language. It can only be provided by language extensions or the use of operating system procedures.

PEARL provides the most comprehensive (and complex) set of i/o facilities. The basic elements are data stations (DATION). They are either system defined (e.g. terminals, disc, or a sensor) or user defined.

I/O operations read or write data structures from or to such data stations. There are facilities for formatted i/o (PUT/GET + format specifications), for i/o in internal representation (READ, WRITE), and process i/o in form of bit sequences (TAKE, SEND).

A complex set of attributes allows specification of all kinds of data station characteristics but requires a sophisticated runtime system for the support of the different i/o operations.

A totally different approach is taken by ADA. No attempt is made to define special features covering the large range of input-output applications. The language facilities are designed in a way which allows the development of input-output packages without the definition of special features.

Three standard packages are predefined in the language:

INPUT_OUTPUT for general user level input-output operations,

TEXT_IO for text input-output, and

LOW_LEVEL_IO for operations dealing low level input-output.

This solution has the advantage that not every user and every translator must handle the additional complexity; however, a solution realized within the language can be realized in a much more flexible way than by using standard language features, e.g. lists of a varying number of output elements could be supported.

Exception Handling

In PEARL and ADA exceptions can be treated explicitly; however, different solutions are provided.

In PEARL, exceptions are considered to be infrequent events but not necessarily errors. Thus an exception can provoke execution of some actions and then control may return to the point where the exception interrupted the normal execution of a task. It is assumed that the exeption handler has performed some repair actions and normal execution can be resumed. (However, the exception handler can decide to branch to an other point of the program).

Exceptions are related to signals and occurrence of an exception activates an exception handler if present. Exception handlers are statements of the form

```
ON signal_id : statement
```

The scope of an exception handler is the task, procedure, repetition or block containing its declaration. Its scope includes all nested units which do not provide a handler for the particular exception.

Thus, these handlers behave like subroutines which can be anonymously activaded at any point of excecution. This undetermined behaviour poses almost unsolvable problems for the verification of program units containing exception handlers.

In ADA , exceptions are restriced to events which can be considered as erros or at least termination conditions. Therefore exception handlers can be declared at the end of a subprogram body, module body, or block, e.g.

```
BEGIN  statements
EXCEPTION
  WHEN exception_id =   statements
  ...
  WHEN OTHERS =   statements
END;
```

Exceptions can be raised implicitly or explicitly (by means of the RAISE statement). When an exception is raised, normal program execution is suspended and the appropriate local handler is activated and replaces execution of the remainder of the current unit. If no local handler is provided execution of the current unit is terminated and the exception is reraised in the outer unit (for a subprogram the outer unit is the unit containing its call). An exception is propagated in this way until a handler is encountered or the body of a task is reached and the task is terminated.

MULTIPROGRAMMING FACILITIES

Multiprogramming facilities are only provided in PEARL and Ada. Both language allow the declaration of tasks; in PEARL they have a structure similar to that of a subprogram, in ADA that of a module. Table 5 lists the operations available to control execution and synchroniuation of tasks.

Table 5: Multiprogrammirg Facilities

PEARL	ADA
extended_time_specification ACTIVATE task; TERMINATE task; SUSPEND task; time_spec CONTINUE task; time_spec RESUME task; PREVENT task;	tasks are activated implicitly upon task creation ABORT task; RAISE task.FAILURE; DELAY expression;
operations on semaphores: REQUEST, RELEASE operations on bolt varibles: RESERVE, FREE, ENTER, LEAVE operations on interrupts: DISABLE, ENABLE, TRIGGER WHEN interrupt_identifier task_control_statement operations on signals: ON signal: statement; INDUCE signal;	rendezvous: ACCEPT entry_descr DO statements END; SELECT WHEN boolean_expr => select_alternative OR select_alternative ELSE statements END SELECT; entry_call_statement SELECT entry_call ELSE statement(s) END SELECT; SELECT entry_call statement(s) ELSE delay_statement statement(s) END SELECT;

In both languages tasks can be created, and deleted, activated, suspended, and aborted. Whereas ADA only provides a minimal set of basic operations, PEARL follows a more application oriented approach. Some of its operations can be combined with elaborated timing specifications, e.g.

```
AT 16:00:30 RESUME task;

WHEN interrupt_id AFTER 10 SEC EVERY 20 MIN
UNTIL 15:20:00 ACTIVATE task_id;
```

This powerful mechanism requires substantial runtime support and it may be difficult to map it on an underlying operating system. It is even doubtful whether features are to be included in a language or whether they belong to the problem set and should be realized by simpler tools provided in the language.

Synchronization Concepts

Symchronization and mutal exclusion must be performed in PEARL with semaphores and bolt variables. Bolt variables are extended semaphores and are in one of the three states "free", "blocked", and "occupied". Table 6 shows the effect of the bolt operations.

Table 6: Bolt operations

```
RESERVE: {free}     -> blocked}
FREE:    {blocked}  -> free}

ENTER:   {free, occupied}  -> {occupied}
LEAVE:   {occupied} -> {free  (iff #LEAVEs=#ENTERs),
                        occupied}
```

Thus, bolt operations provide the mechanism to achieve exclusive access or simultaneous access to shared objects.

Semaphores and bolt variables are simple and easy to unterstand; however their use tends to be unstructured and prone to error: the respective operations must occur in pairs but no automatic checks for correct use are possible.

The rendezvous concept in ADA tries to circumvent these difficulties. A rendezvous is an (asymmetric) interaction between two tasks. One task issues a request to an "entry" in the second task. The second task performs the interaction when it is ready to accept the request. Entries are declared in a form similar to a subprogram declaration and requests have the form of a subprogram call.

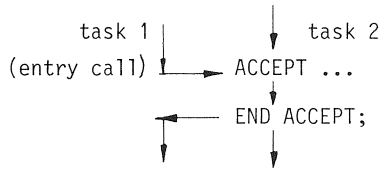


Fig. 5: Rendezvous

The accept statement specifies the actions to be performed during a rendezvous, i.e. when the corresponding entry is called (by task 2).The task arriving statement is reached (by task 2). The task arriving first has to wait for the other.

A select statement combines several accept and delay statements, thus making selective wait and timeout conditions possible. Two other forms of the select statement allow the caller of an entry to issue a conditional entry call, i.e. the entry call is only issued if the redezvous is immediately possible; the timed entry call allows the secification of a maximal delay for the acceptance of the entry call.

The rendezvous concept is an attempt to unify process communication and mutual conclusion. It allows synchronous process communication via the parameter list of an entry. This synchronous communication technique allows any other synchronization or communication concept to be modelled; however, in most cases auxiliary processes are required.

FINAL REMARKS

The following summarizing remarks on each of the languages do not consider the availability of compilers, although availability and quality of a compilers can be the determining factor when a language is to be chosen.

CORAL certainly fulfills the design criteria stated above. It is a simple language, easy to implement and allows efficient access to hardware and operating system facilities. However, the definition leaves many details to a particular implementaion, e.g. I/O. In addition assembly code insertions and usage of anonymous references, i.e. absolute addresses, reduce the portability (and probably also the maintainability) of programs.

PASCAL provides a set of simple control structures and a large variety of data types. The concept of strong typing (although not totally waterproof) allows the detection of many errors at compile time. However, PASCAL does not provide modules, multiprogramming facilities, and support for seperate compilation. There are many languages extending PASCAL in this respect which maintain its original

simplicity. Two examples are MODULA [Wi 78] and PORTAL [Na 79].

PEARL provides a large set of facilities for real-time programming. However, the language is very complex and baroque. Furthermore, its design is not very consistent, e.g. interrupts exist besides the very elaborated timing specification possibilities, and a primitive case statement together with powerful input-output statements. The input-output system and the multi-tasking model require an elaborated run time support. In many cases it is very difficult to map PEARL features on an underlyng operating system in an efficient manner.

ADA also provides a large set of facilities. In comparison with PEARL, the elements are kept on a lower level. For example, timing specifications for process scheduling are not provided but can be realized with the given features. ADA has a consistent typing concept which is stronger than that of PASCAL. Since every single feature is elaborated in a detailed way (e.g. type, subtype, and derived type are distinguished)the whole language becomes rather complex. Since many restrictions ore rules which are only understandable when the underlying concepts are known, the language is difficult to instruct.

REFERENCES

[AD79] --: Preliminary ADA Reference Manual, acm, Sigplan Notices, 14,6 1979

--: Rationale for the Design of the ADA Programming Language, acm, Sigplan Notices, 14,6 1979

[AD80] --: Reference Manual for the Ada Programming Language United States Department of Defence, July 1980

[ES79] --: ESL, Report on Main Task I and II. Siemens, Munich and CII Honeywell Bull, Paris Oct. 1979

[Na79] Naegeli H.H.: Programming in PORTAL: Publication of Landis & Gyr, Zug, Switzerland

[PE77] --: Basic PEARL Language Description. Gesellschaft fuer Kernforschung mbH, Karlsruhe PDV-Report KfK-PDV 120, 1977

--: Full PEARL Language Description. Gesellschaft fuer Kernforschung mbH, Karlsruhe PDV-Report KfK-PDV 130

[We76] Wegner P.: Programming Languages - The First 25 Years. IEEE Transactions on Computers, Vol. C-25, 12, 1976

[Wi71]Wirth N.: The Design of a PASCAL Compiler. Software-Practice and Experience, Vol. 1,1971

[Wi77] Wirth N.: Modula: A Language for Modular Multiprogramming. Software, Vol. 7, 3-35, 1977

[Wi78] Wirth N.: MODULA-2. Report of the Institut fuer Informatik, ETH Zurich, No. 27, Dec. 1978

[Wo70] Woodward P.M.: Official Definition of CORAL 66. Her Majesty's Stationery Office, 1970

[WW78] Werum, W. and Windauer H.: PEARL, Process and Experiment Automation Realtime Language. Vieweg Braunschweig 1978

Sprachen für die Echtzeit-Programmierung; Stellungnahme zum Vortrag von Dr. H. Sandmayr auf der Prozeßrechnertagung (März 1981)

H. Mittendorf

In dem auch als Aufsatz erschienenen Beitrag von Sandmayr werden die vier verglichenen Sprachen in 2 Klassen unterteilt: die erste enthält zur Realisierung der Echtzeitaufgaben nur Aufrufe an Standard- oder Betriebssystem-Programme, die zweite realisiert spezielle Sprachmittel; diese zweite Klasse enthält z.Z. nur die Sprachen Ada und PEARL. Für die Zukunft werden wohl nur diese beiden Sprachen für den Anwender interessant sein. Diese Stellungnahme wird sich daher auch nur mit dem Vergleich dieser Sprachen beschäftigen.

1. Anwendungsbereich und Festlegung der Sprachen

PEARL ist für Anwender der Echtzeit-Programmierung entworfen; die Designer haben für PEARL nie den Anspruch angemeldet, die Sprache sei für System-Implementierungsaufgaben im engeren Sinne besonders geeignet; der Grund ist einfach zu erklären: für System-Implementierungszwecke sind normalerweise andere Anforderungen (größere Maschinennähe, Mittel für Systemaufrufe) als für Anwendungszwecke (vorgefertigte Anweisungen für spezielle Benutzerwünsche, z.B. Tasking) erforderlich. Auf der anderen Seite gibt es erheblich mehr Anwender als "System-Implementierer", die überdies aufgrund ihrer in der Regel stärker spezialisierten Ausbildung (hauptsächlich als Informatiker) auch mit "niederen" Sprachen sicher arbeiten können. Für die Anwender ist jedoch die Brauchbarkeit der Sprache (Sandmayr: "...large set of facilities for real-time programming") von entscheidender Bedeutung. Über Ada sagt Sandmayr: "ADA also provides a large set of facilities. In comparison with PEARL, the elements are kept on a lower level" und weiter "... Since many restrictions or rules which are only understandable when the underlying concepts are known, the language is difficult to instruct." Dann darf wohl der Schluß erlaubt sein, daß Ada mindestens schwerer zu lernen und damit "anwender-unfreundlicher" ist als PEARL.

Bliebe Ada beschränkt auf "Systemimplementierungs-Aufgaben", so wäre hiergegen nichts einzuwenden: Für die "bequemen" Anwender wäre PEARL die geeignete Sprache. - Es wird jedoch anders kommen: Schon, um mehr "Kunden" zu haben, wird Ada versuchen, auch in Kreise der Anwender einzudringen. Sollte dies eintreten, so muß sich auch Ada all den Fragen stellen, die seinerzeit auch potentielle PEARL-Anwender gestellt haben. Und davon werde ich im folgenden noch ausführlicher zu sprechen haben.

2. Ein-/Ausgaben in Ada

Ada sieht bekanntlich für Zwecke der Ein-/Ausgabe nur Pakete und einen "niederen E/A-Level" vor. Cum grano salis könnte man das mit dem "E/A-Angebot" von ALGOL 60 vor zwei Jahrzehnten vergleichen. Die Folgen sind wohlbekannt: die wesentlich "schlechtere" Sprache FORTRAN mit für die damalige Zeit hervorragenden E/A-Sprachelementen lief in kurzer Zeit ALGOL 60 den Rang ab. Für die "E/A-Philosophie" von Ada läßt sich die Problematik in drei Fragen zusammenfassen:

a) Wer schreibt die "E/A-Pakete" (d.h.: wer bezahlt sie, wenn sie für ein aktuelles Problem zur Verfügung stehen müssen)?

b) Wer sorgt dafür, daß vorhandene E/A-Pakete im Rahmen eines einzuhaltenden Standards (so ist doch wohl angestrebt?) nachgeführt, gepflegt, gewartet werden?

c) Wer garantiert dafür, daß bei diesen (wohl in Schichten-Struktur und ohne Unterstützung durch ein Laufzeitsystem implementierten) Paketen eine vom Anwender verlangte Laufzeiteffizienz erreicht wird.

3. Systemumgebung für Ada

In PEARL wird bekanntlich die Systemumgebung im "Systemteil" beschrieben. Vorhandene PEARL-Implementierungen sind also nicht zuletzt nach der Reichhaltigkeit der zur Verfügung gestellten Systemgeräte (und dazugehöriger Laufzeitbesonderheiten = SIGNALs") zu bewerten. In Ada gibt es entsprechende Beschreibungsmittel überhaupt nicht; zusätzlich sind "EXCEPTIONs" in Ada schwächer ausgebildet als SIGNALs in PEARL, da mit EXCEPTIONs nicht notwendigerweise auf Geräteeigenschaften einer vorliegenden Geräteausstattung reagiert wird. Damit ist "in praxi" die Formulierung "sicherer" Programme sehr erschwert. Man wird hierfür in der Regel komplizierte "Assembler-Konstruktionen" wählen müssen.

Für das Fehlen eines SYSTEMteils gibt es in Ada überhaupt keinen Ersatz.

4. Tasking

In Ada gibt es Tasks, aber ein nur schwach ausgebildetes Tasking. Überdies sind viele Begriffe und Konstruktionen unklar (Task als RECORD-Element, automatischer Start bereits bei der Generierung). Dies bedeutet nicht nur erhebliche Anforderungen an das Kompiliersystem selbst, sondern auch hohen Lernaufwand für den Anwender.

> Es ist daher zu erwarten, daß "höhere" Tasking-Konstrukte entstehen. Wer stellt aktuell höhere Tasking-Funktionen zur Verfügung, bzw. pflegt und wartet solche "Tasking"-Pakete? Wer wacht über die m. E. erforderliche Einheitlichkeit?

5. Synchronisationen

Dem altbewährten Semaphor-Ansatz in PEARL (in Full PEARL noch bereichert um ein BOLT-Konzept), den jeder Informatiker in der "Schule" anzuwenden lernt, steht das m. E. schwer durchschaubare "Rendezvous-Konzept" von Ada gegenüber. In PEARL lassen sich die beiden Grundaufgaben der Synchronisation ("Start über Koordinierungszähler" und "schreibender Zugriff zu geschützten Bereichen") in sehr einfacher und zuverlässiger Weise über Semaphore lösen. Die wechselweise Benutzung von REQUEST und RELEASE hat dabei im ersten Falle in zwei verschiedenen Tasks, im zweiten Falle

in der benutzenden Task zu erfolgen. Bei disziplinierter Programmierung treten hierbei kaum Fehler auf. Abgesehen von der Tatsache, daß das Rendezvous-Konzept kaum zuverlässige Aussagen über das Zeitverhalten gestattet, sind bei der praktischen Verwendung häufig Hilfskonstruktionen erforderlich (Sandmayr: "This synchronous communication technique allows any other synchronisation or communication concept to be modelled; however, in most cases auxiliary processes are required"). Insgesamt: für die bei weitem am häufigsten vorkommenden "einfachen" Synchronisationsprobleme bietet Ada weniger übersichtliche Sprachmittel als PEARL; für die komplizierteren Probleme sind (wie auch in PEARL) Hilfskonstruktionen, deren Brauchbarkeit in praxi unbewiesen ist, erforderlich. Ich kann mir keinen Anwender vorstellen, der darüber glücklich wäre.

6. Datentypen

Von Sandmayr werden die Datentypen in PEARL und Ada als ungefähr gleichwertig angesehen. Die Aussage, das Typ-Konzept von PEARL sei von PL/1 übernommen, ist falsch; es basiert tatsächlich auf ALGOL 68. Möglicherweise wird bei dieser Aussage PEARL mit einem "Vorgänger" (PAS 1 von BBC) verwechselt.

Daß in PEARL die 4 Grunddatentypen BIT, FLOAT, FIXED und CHAR auch mit Längenangaben (z.B. CHAR (4)) deklariert werden können, bedeutet nicht, daß die Länge eines Ergebnistyps auf die komplizierte PL/I-Weise aus den längen der Operandentypen berechnet würde. In PEARL sind vielmehr Längenangaben als eine Verfeinerung der üblichen (FORTRAN, ALGOL 68) Längenstufungen zu verstehen, wobei der Typ eines Verknüpfungsergebnisses im allgemeinen der Länge des längeren Operandentyps entspricht.

Bei BIT und CHAR gibt es in PEARL (sogar schon in Basis PEARL) Selektions-Operatoren, mit denen z.B. bestimmte Bit-Bereiche aus einem "Festwort" ausgesondert werden können. Genau diese Möglichkeit ist aber für den Prozeß-Anwender, der immer mit sehr großen und daher nur gedrängt abspeicherbaren Datenmengen zu tun hat, von ausschlaggebender Bedeutung. Die in Ada aus PASCAL übernommene "Array-Lösung" (man führe z.B. anstelle einer BIT (5)-Größe ein BOOLEAN-Array aus 5 Komponenten ein) ist praxisfremd.

7. Stand der Ada-Norm

Bisher ist kein Normvorschlag bekannt geworden, der dem Präzisionsgrad der PEARL-Norm (netzattributierte Grammatiken, Petrinetze und verbale Beschreibung) in DIN 66 253 (Teil 2) entspricht. Hier sei der Hinweis erlaubt, daß eine zukünftige deutsche Ada-Norm nicht nur die Präzision (mindestens!) der PEARL-Norm, sondern darüberhinaus auch vorführbare Implementierungen des gesamten Ada-Umfangs fordern wird (genau so, wie es bei PEARL auch gehandhabt wurde).

8. Stand der Implementierungen von Ada

Vor irgendeiner realistischen Beurteilung von Ada müssen (wenigstens halbwegs) vorführbare Implementationen von Ada inclusive ausreichender E/A-Packages verfügbar sein. Für Ada als "Anwendersprache" sollte gefordert werden, daß ausreichend viele, aus der Praxis entnommene Echtzeit-Probleme in Ada übersetzt und zum Ablauf gebracht sein müssen. Zur Zeit kann davon noch keine Rede sein.

Dr. Mittendorf, Horst
Fa. Siemens AG
E STE 33
Östl. Rheinbrückenstr.50
7500 Karlsruhe 21
Tel.: 0721-595-2284

Kurzmitteilungen

P. Puhr-Westerheide GRS Garching

PEARL-Kurs am IFE in Halden/Norwegen in der Zeit vom 31.3.1981 bis zum 7.4.1981

Am Institut für Energiteknikk (IFE) in Halden wurde ein PEARL-Kurs für eine Gruppe von sieben Personen durchgeführt. Der Kurs wurde vom IFE und der Gesellschaft für Reaktorsicherheit (GRS) mbH in Garching organisiert. Die Kurssprache und alle Unterlagen waren englisch; Unterstützung wurde vom PEARL-Verein, der Universität Stuttgart (Inst.f. Regelungstechnik und Prozeßautomatisierung) und der Universität (Fachber. Informatik-2) geleistet. Die Kursdauer betrug fünf Tage; zwei Tage Vorspann wurden zur System-Einarbeitung und zur Erstellung von Beispiel-Programmen benötigt. Die Zielgruppe des Kurses bestand aus sieben Personen, die über eine langjährige Erfahrung in der Erstellung von großen Prozeß-Softwarepaketen verfügt. Diese Gruppe hat bisher in FORTRAN/NORSK DATA Prozeß-Fortran programmiert, wird aber in Zukunft Teile der Software des Reaktor-Störungsanalysesystems, das in Zusammenarbeit zwischen dem IFE und der GRS entwickelt wurde, in PEARL codieren (s.a PEARL-Rundschau, Heft 1, Band 2, Apr. '81, p. 35-36).Die Installation der PEARL-Compilers auf NORD-Computer wurde am IFE durchgeführt; nach anfänglichen Problemen ist ein arbeitsfähiges PEARL-System entstanden. Der vorgetragene Stoff gliederte sich in die Punkte:

1 System-Teil/Input-Output
2 PEARL-Algorithmik
3 Real-time features
4 PEARL-Analysator

wobei der erste Punkt - gemäß den in PEARL zur Verfügung stehenden Sprachmitteln, die über die anderer Sprachen hinausgehen - am meisten diskutiert wurden. Der letzte Punkt beschäftigte sich mit den Möglichkeiten zur Verifikation von PEARL-Programmen mit Hilfe des PEARL-Analysators, da beim Einsatz von PEARL-Programmen für Probleme, die erhöhte Sicherheitsvorkehrungen benötigen, eine Überprüfung der Programmprodukte erforderlich ist. In Zusammenarbeit mit der Gruppe wurden kleinere PEARL-Module erstellt und getestet; auch hierbei wurde den verschiedenen I/O-Möglichkeiten und Dations erhöhte Aufmerksamkeit geschenkt, wobei auch einige kleinere Fehler des PEARL-Systems entdeckt wurden. Der Kurs lief zur vollsten Zufriedenheit aller Beteiligten ab. Er hat gezeigt, daß PEARL auch im Ausland auf Interesse stößt. Die Veranstalter sind gerne bereit, bei künftigen ähnlichen Vorhaben mit ihren Unterlagen und Erfahrungen beizutragen. An der Erstellung der Unterlagen waren beteiligt:

- A.Ghassemi (Universität Stuttgart)
- E.Nickl (GRS)
- K.Pelz (Universität Erlangen)
- P.Puhr-Westerheide (GRS)

Vorgetragen wurde von Herrn K. Pelz und Herrn P. Puhr-Westerheide (Leitung).

PEARL-Kurse

13.-24.7.1981 PEARL

Der Kurs bietet eine Einführung in PEARL für Siemens Rechner. Der Kurs wird bei Bedarf auch in Englisch angeboten.

Veranstalter: Siemens AG
Schule für Prozeß-rechnertechnik
Herr Heim
Östl.Rheinbrückenstr.50
7500 Karlsruhe 21
Tel. 0721/5954125

Ort: Karlsruhe

27.-31.7.1981 und 21.-25.9.1981 Systematisches Programmieren mit PEARL

Einführungskurs für Programmierer mit praktischen Übungen am Rechner (die Kurse sind bereits vom IRT München ausgebucht).

Leitung: Prof.Dr. Pflügel, Furtwangen

Veranstalter: PEARL-Verein e.V. in Zusammenarbeit mit DORNIER-System GmbH.

Ort: München

21.-25.9.1981 Systematisches Programmieren mit PEARL

Der Kurs gibt an Hand praxisnaher Beispiele eine Einführung in das Programmieren mit PEARL. Für praktische Übungen werden verschiedene PEARL-Systeme zur Verfügung gestellt.

Leitung: Dr. K. Rebensburg Berlin

Veranstalter: PEARL-Verein e.V. in Zusammenarbeit mit dem VDI-Bildungswerk

Ort: Stuttgart

Veranstaltungen und Termine

11.-22-8-1981	AUTOMATICA '81 Oslo, Norwegen The Norwegian Fair Organization, P.O. Box 130, N-Oslo 2
24.-28.8.1981	8 th IFAC World Congress Kyoto, Japan
2.-3.9.1981	Informationstagung "Messen,Prüfen und Steuern in Walzwerken" Luxemburg Kommission der EG, Rue de la Loi 200, B-1049 Brüssel
15.-17.9.1981	Kongress "Kraftwerke 1981" Den Haag, Holland VGB, Klinkestr.27-31 D-4300 Essen
1.-2.10.1981	"PEARL in Ausbildung, Lehre und Schulung" Erlangen PEARL-Verein, Düsseldorf
6.-7.10.1981	BIAS '81 - 17 th International Conference on Control of Industrial Processes Mailand, Italien FAST, Piazza Rodolfo Morandi 2, I-2012 Milano
7.-8.10.1981	Konferenz "Leittechnik in Wärmekraftwerken" Essen
6.-8.10.1981	3 rd Data Processing Conference Zürich UNIPEDE, 39, avenue de Friedland F-75008 Paris
19.-23.10.1981	GI-Jahrestagung München darin: Fachgespräch "Systematischer Entwurf von PDV-Systemen"
19.-23.10.198	SYSTEMS '81 Internationaler Kongress und internationale Fachmesse München
16.-17.11.1981	Automatisierte Materialfluß-Systeme München VDI-Gesellschaft Materialfluß und Förertechnik
7.-8.12.1981	PEARL-Tagung '81 "Die Zukunft von PEARL" Düsseldorf

Neue Mitglieder

63. Ing.(grad.) Werner Kerschke
Erlenstraße 25
2000 Hamburg 54

64. Ing.(grad.) Kurt Wößner
Unter den Linden 35
2093 Stelle

65. Ute Wieland
Friedr. Krupp GmbH
KRUPP FORSCHUNGSINSTITUT
Münchenerstr. 100
4300 Essen

66. Ing.(grad.) Thomas Meyer
Regenbogenstr. 13
8500 Nürnberg 50

67. Prof. Horst Meintzen
G.-Zembrodstr. 23
7753 Allensbach

68. Gerhard Göttle
Martinusstr. 17
7901 Dornstadt 2

69. Prof. Dr. Wilhelm Pieper
Fachhochschule Gießen-Friedberg
Wiesenstr. 14
6300 Gießen

70. Prof. Dr. Helmut Rzehak
Hochschule der Bundeswehr München
Fachbereich Informatik
Werner-Heisenberg-Weg 39
8014 Neubiberg