PEARL, eine prozeß- und experimentorientierte Programmiersprache

B. Eichenauer, V. Haase, P. Holleczek, K. Kreuter, G. Müller

Sonderdruck aus »Angewandte Informatik«, Jahrgang 15, Heft 9/1973, Seite 363–372
Verlag Friedr. Vieweg + Sohn, Braunschweig, Burgplatz 1

Gesellschaft für Kernforschung m. b. H.
Projekt Prozeßlenkung mit DV-Anlagen
7500 Karlsruhe, Postfach 3640

PEARL, eine prozeß- und experimentorientierte Programmiersprache

B. Eichenauer, ESG München V. Haase, UNICOMP Blankenloch P. Holleczek, Universität Erlangen
K. Kreuter, ESG München G. Müller, BBC Mannheim

Zusammenfassung: *Es wird eine Programmiersprache der mittleren Ebene beschrieben, die es erlaubt, Struktur, Algorithmen, Zeitverhalten und Ein-/Ausgabe von Echtzeitprogrammen zu formulieren. Sie wurde von einer Gruppe deutscher Industriefirmen und Forschungsinstitute entwickelt und ist für den Automatisierungsingenieur oder Experimentator mit Programmiererfahrung bestimmt.*

Summary: *A middle-level programming-language is described which allows to formulate the structure, the algorithms, time behaviour and I/O of real-time programs. It was developed by a group of German industrial firms and research institutes and is intended for the process control engineer or experimenter with some programming experience.*

1. Einleitung

Der kostenwirksame Einsatz von digitalen Rechensystemen bei der Lösung von kommerziellen und technisch-wissenschaftlichen Aufgaben bereitet heute auch dem Anwender, dem die Funktionsweise von Digitalrechnern nur wenig vertraut ist, bei Verwendung höherer Programmiersprachen wie ALGOL, FORTRAN, PL/1 keine Schwierigkeiten mehr. Dagegen muß der Anwender, der Digitalrechner bei der Lösung von Prozeßautomationsaufgaben oder in der Experimentsteuerung einsetzen will, umfangreiche Erfahrung im Umgang mit den jeweils zum Einsatz gelangenden Rechnertypen besitzen und sehr viel Zeit für die Erstellung von Automationsprogrammen aufwenden.

In den vergangenen Jahren sind zahlreiche Ansätze gemacht worden, um den Einsatz von Digitalrechnern bei der Lösung von Realzeitaufgaben zu vereinfachen. Dabei lassen sich im wesentlichen drei Wege erkennen, auf denen eine Vereinfachung der Realzeitprogrammierung gesucht wird.

Der erste Weg, der schon ab Beginn der sechziger Jahre verfolgt wird, besteht in der Bereitstellung sog. Programmpakete (z. B. [1], [2], [3]) und von problemorientierten Programmiersprachen (z. B. [4], [5], [6], [7]) für Teilgebiete des Anwendungsfeldes.

Die Erfahrung im Umgang mit Paketen und mit problemorientierten Programmiersprachen zeigt jedoch, daß die Unterschiede in den Aufgabenstellungen eines Anwendungsgebiets normalerweise so groß sind, daß nur ein kleiner Teil des Gebiets, für das ein Paket oder eine Sprache entwickelt wurde, überdeckt werden kann. Zudem müssen Pakete und problemorientierte Programmiersprachen wegen der rasch fortschreitenden Technologie häufig an den aktuellen Stand der Technik angepaßt werden. Für das Gebiet des automatischen Check-out gibt es beispielsweise mindestens 30 problemorientierte Programmiersprachen und Dialekte dieser Sprachen. Trotzdem lassen sich relativ einfache Aufgabenstellungen angeben, die in keiner dieser Sprache vollständig beschrieben werden können.

Der zweite Weg zur Überwindung der Schwierigkeiten in der Prozeßprogrammierung besteht in der Entwicklung von Programmiersprachen mittleren Sprachniveaus wie CORAL 66 ([8]), die lediglich in effizienten Code übersetzbare Sprachmittel zur Beschreibung algorithmischer Zusammenhänge bieten. Die Realzeitprogrammierung wird erledigt, indem Betriebssystemfunktionen des jeweiligen Zielrechners aufgerufen werden.

Die Bereitstellung algorithmischer Sprachelemente, die eine gegenüber der Programmierung in Assembler schnellere Programmerstellung ermöglicht, bietet ansonsten jedoch kaum Vorteile. Der Anwender muß die umgebende Herstellersoftware (z. B. Aufrufe für die Parallelprogrammierung, Treiber für die Prozeß- und Standardperipherie) beinahe genauso gut kennen wie bei Verwendung von Assemblersprachen. Der Dokumentationswert und die Portabilität von Programmen ist wegen der Abhängigkeit von dem jeweiligen Zielrechner eingeschränkt.

Der dritte Weg, der bei der Entwicklung INDAC 8 ([9]), PAS 1 ([10]) und von PEARL ([11]) beschritten wurde, besteht darin, die zur Formulierung algorithmischer Zusammenhänge erforderlichen Sprachmittel mit einer hinreichenden Menge grundlegender Sprachelemente für die Realzeitprogrammierung zu verschmelzen. Ebenso wie die algorithmischen Eigenschaften von Programmiersprachen wie ALGOL oder FORTRAN zur Formulierung beliebiger numerischer Algorithmen herangezogen werden können, erlauben die in PEARL aufgenommenen Ausdrucksmittel zur Beschreibung von Realzeitvorgängen eine problemgerechte Aufteilung von Anwendungsproblemen in schwach gekoppelte Prozesse ([12]) und die Beschreibung der Wechselwirkung von Prozessen mit ihrer Umwelt.

Dadurch werden die mit höheren Programmiersprachen erreichbaren Vorteile auch für die Prozeßprogrammierung nutzbar. Mit Hilfe einer einmal zu erlernenden Menge von

Ausdrucksmitteln können Automationsprogramme in weitgehend maschinenunabhängiger Weise formuliert werden. Programmpakete können vom Anwender selbst in dokumentationsfreundlicher Weise geschaffen und an den aktuellen Stand der Technik angepaßt werden.

2. Struktur eines PEARL-Programmsystems

Um Automationsprogramme problemgerecht in Teilprogramme zerlegen zu können (program management) und um den systematischen Aufbau von Prozeßprogrammen zu ermöglichen, kann in PEARL ein Objektprogramm aus einer Menge unabhängig compilierbarer Einheiten, sog. PEARL-Programmoduln, aufgebaut werden. Die zur Lösung eines Automationsproblems erforderlichen Programmoduln werden nach ihrer Übersetzung in einem Bindelauf mit den Funktionen des Betriebssystems und mit den jeweils erforderlichen Bibliotheksfunktionen verknüpft und ergeben so die lauffähige Version eines Automationsprogrammes, ein sog. PEARL-Programmsystem.

2.1. Programmoduln

Damit Automationsalgorithmen unabhängig von der Konfiguration des jeweils eingesetzten Prozeßrechensystems beschrieben werden können, wird in PEARL zwischen Systembeschreibung und Problembeschreibung unterschieden.

Ein PEARL-Programmodul enthält entweder einen Systemteil oder einen Problemteil oder beides.

Im Systemteil wird der Aufbau von Prozeßrechensystemen, d. h. die Kopplung der jeweils eingesetzten Standard-Geräte (z. B. Rechner, Digitalausgabe, Schreibmaschine usw.) beschrieben. Dabei werden für alle Endstellen, die bei der Problembeschreibung in einem Problemteil angesprochen werden sollen, Namen eingeführt. Diese Namen fungieren als globale Bezeichner innerhalb der Moduln, aus denen ein PEARL-Programmsystem aufgebaut wird.

Unter Verwendung der im Systemteil eingeführten Endstellen-Bezeichner sind im Problemteil Automationsalgorithmen anlagenunabhängig darstellbar. Programmgrößen, die auf Modulebene vereinbart werden, sind innerhalb aller Blöcke und Prozeduren des Problemteils, in dem sie vereinbart wurden, bekannt, sofern der Name der Programmgröße innerhalb eines Blocks oder einer Prozedur nicht erneut vereinbart wird.

2.2. Globale Größen

Über globale Größen wird die Verbindung zwischen verschiedenen Programmodulen hergestellt. Beispielsweise wird durch

```
DCL DRUCK INT (5) GLOBAL;
```

eine globale Integer-Variable vereinbart.

Zur Identifikation globaler Größen verschiedener Moduln dienen außer Bezeichnern für diese Größe sog. Bibliotheksnamen. Um sich auf globale Größen beziehen zu können, die in anderen Moduln deklariert werden, müssen die Attribute dieser Größen in den Moduln, in denen sie angesprochen werden sollen, spezifiziert werden.

Beispiel:

```
MODULE LIBRARY LIB1;
PROBLEM;
DCL  K  INT (5) GLOBAL;
       :
MODEND;
MODULE LIBRARY LIB2;
PROBLEM;
DCL  K  INT (5) GLOBAL LIB1;
       :
MODEND;
```

In dem Modul mit Bibliotheksnamen LIB1 wird die globale Größe K deklariert. Diese soll im Modul mit Bibliotheksnamen LIB2 angesprochen werden. Dazu ist in LIB2 der Name und Typ der globalen Größe sowie der Bibliotheksname des Moduls zu spezifizieren, in dem die Größe deklariert wurde.

3. Beschreibung von Realzeitvorgängen

Programmsysteme zur Steuerung industrieller Prozesse oder zur Laborautomation bestehen normalerweise aus einer Anzahl von Programmteilen zur Steuerung von Teilprozessen, die zwar gelegentlich miteinander koordiniert werden müssen, im übrigen aber autonom sind. Für eine Sprache zur Prozeßautomatisierung sind daher Elemente unerläßlich, mit denen die Überwachung und Steuerung derartiger Vorgänge in asynchron zueinander ablaufende Aufgaben („Tasks") eingeteilt und der Ablauf dieser Aufgaben zur rechten Zeit angestoßen, unterbrochen, fortgesetzt und beendet werden kann. PEARL enthält daher eine Anzahl von Sprachelementen, um solche parallelen Aufgaben abwickeln sowie sie in Abhängigkeit von spontanen oder auch zeitlich fixierten Ereignissen vorplanen zu können. Die Synchronisation unabhängiger Abläufe und die Vergabe von Betriebsmitteln über Prioritäten sind in PEARL ebenfalls möglich.

3.1. Parallele Vorgänge

Der Grundbegriff paralleler Vorgänge in PEARL ist die Task. Eine Task bezeichnet den (dynamischen) Ablauf eines Programmteils unter Kontrolle des Betriebssystems. Sie ist weitgehend von anderen Vorgängen im System unabhängig und kann als ein virtueller Prozessor betrachtet werden. Vom Programmierer kann die Ausführung beliebiger Programmstücke als Task spezifiziert werden, insbesondere sind im allgemeinen Antworten auf externe Ereignisse sowie asynchron ablaufende E/A-Vorgänge als Tasks auszuführen. Organisation paralleler Vorgänge heißt in PEARL: Handhabung von Tasks.

3.2. Arbeiten mit Tasks

Tasks werden mit ihrem Namen angesprochen; dieser ist wie alle anderen Sprachelemente in PEARL zu deklarieren. „Zum Leben erweckt" wird eine Task jedoch erst durch das Statement ACTIVATE. Zu jeder aktivierten Task gehört ein Stück Code, das „Segment"; es kann schon bei der Deklaration oder auch erst bei der Aktivierung spezifiziert werden.

```
Beispiel:   TASK T;
            DECLARE P PROC = BEGIN;
                                :
                                :
                             END;
            :
            :
            ACTIVATE T : CALL P;
            :
            :
```

Während die Prozedur P ein statisches Gebilde ist, eine Codefolge, die auch vor und nach ihrer Ausführung existiert, durchläuft die Task T verschiedene Zustände: „nicht existent", „deklariert" und „aktiviert". Wann sie wirklich aktiv ist (einen Prozessor des Systems belegt), hängt zusätzlich noch von der Verfügbarkeit der benötigten Betriebsmittel ab.

Alle Task-Operationen können vorgeplant werden – in PEARL wird das als „Scheduling" bezeichnet. Eine Schedule gibt an, zu welchen Zeitpunkten, in welchen Zeitabständen oder beim Eintritt welcher Ereignisse (Interrupts) die spezifizierte Taskoperation eingeleitet werden soll.

```
Beispiel:   AT 8 : 0 : 0 ALL 2 HRS UNTIL 20 : 0 : 0
                         ACTIVATE T : CALL P;
```

Die Task T soll zyklisch alle 2 Stunden zwischen 8 und 20 Uhr gestartet werden.

```
ON INTERRUPT1 AFTER 3 MIN SUSPEND T;
```

Die Task T soll 3 Minuten nach dem Ereignis INTERRUPT1 unterbrochen (suspendiert) werden.

Neben den Grundfunktionen des Task-Managements; Deklaration, Einplanung und Aktivierung, sind ferner Anweisungen zur Unterbrechung, zur Fortsetzung und zum Abbruch von Tasks und zur Streichung von Schedules vorgesehen.

Beispiele:

```
SUSPEND T;      Unterbrechung auf unbestimmte Zeit,
CONTINUE T;     Fortsetzung einer suspendierten Task,
DELAY T
DURING 1 SEC;   Unterbrechung für eine Zeitspanne,
TERMINATE T;    Abbruch der Task,
PREVENT T;      Streichung aller Einplanungen von T.
```

In bezug auf die Verwaltung des Speicherplatzes ist die Taskorganisation in die Blockstruktur des Programmsystems eingebettet. Als „Subtasks" einer Task werden Abläufe bezeichnet, deren Codesegment in einem Block liegt, der zum Code jener Task gehört. Auf diese Weise wird es möglich, Variable, die von mehreren Tasks benötigt werden, nur einmal anzulegen (kein Kopieren notwendig). Die Speichervergabe wird durch die zur Compilezeit festgelegte Blockstruktur geregelt, und zeitaufwendige Speicherverwaltungsalgorithmen können zur Laufzeit entfallen.

3.3. Reaktion auf sporadisch auftretende Ereignisse

Typisch für Realzeitsysteme ist es, daß der Anwendungsprogrammierer Reaktionen auf Ereignisse aus der „Außenwelt" des Systems spezifizieren muß. Solche „Interrupts" treten asynchron zur Arbeit des Systems auf und sollen im allgemeinen kurzfristig beantwortet werden.

In PEARL werden Interruptantwortroutinen als Tasks deklariert und in Schedules zur Aktivierung bei Auftreten des Interrupts eingeplant.

```
Beispiel: TASK FAILSAFE : BEGIN;
                             :
                             :
                          END;
          DECLARE INT1 INTERRUPT;
          :
          ON INT1 ACTIVATE FAILSAFE;
```

Bei Auftreten des Interrupts 1 wird das zur Task „Failsafe" gehörende Segment ausgeführt.

Interrupts können auch simuliert werden, was etwa beim Programmtest oder bei der Funktionsprüfung von Systemen erforderlich sein kann.

```
Beispiel: TRIGGER INT1;
```

Um bestimmte Interrupts zeitweise unwirksam zu machen bzw. um diese Sperre wieder aufzuheben, bedient man sich der Operationen:

```
DISABLE INT1;    /* SPERREN */
ENABLE INT1;     /* FREIGEBEN */
```

Der Interrupt wird „möglichst weit außen" abgeklemmt, so daß das System von dem betreffenden Ereignis unberührt bleibt.

In PEARL ist es ferner möglich, auf Ereignisse, die in Zusammenhang mit der Ausführung bestimmter Anweisungen auftreten (z. B. arithmetischer Überlauf), durch irreguläre Fortsetzung der betroffenen Task an einer spezifischen Stelle zu reagieren.

```
Beispiel: RESPONSE SGL:  /* REAKTION */
```

verknüpft die Reaktionsroutine mit einem solchen (in PEARL als Signal bezeichneten) Ereignis.

3.4. Beziehung zwischen Tasks

Obwohl verschiedene Tasks asynchron zueinander ablaufen, bestehen doch häufig zahlreiche Beziehungen zwischen ihnen. Dazu gehört besonders die Kommunikation von für mehrere Tasks wichtigen Daten. Da in PEARL alle Tasks und die bei Ihrer Bearbeitung auszuführenden Anweisungen in die Blockstruktur eingebettet sind, kann das mittels globaler Größen, d. h. Größen, die innerhalb der Anweisungsfolgen aller beteiligten Tasks bekannt sind, geschehen. Damit nun nicht eine Task nach dem Durchlaufen der Anweisungen eines Blocks den Speicherplatz der darin deklarierten Größen freigibt, die gegebenenfalls noch von anderen Tasks benutzt werden sollen, gehört in PEARL zur vollständigen Bearbeitung eines Blocks auch die vollständige Abwicklung aller Tasks, deren Anweisungsfolgen in dem betrachteten Block enthalten sind (Subtasks). Praktisch führt das dazu, daß eine Task nach Durchlaufen der Anweisungen eines Blocks auf die Beendigung aller Subtasks wartet.

Neben dieser Synchronisation aufgrund der Blockstruktur ist die Verwendung von Semaphorvariablen die wichtigste Synchronisationsmethode.

Semaphore und die auf sie wirkenden Operationen sind im Dijkstraschen Sinn definiert [12]. Semaphore enthalten eine positive ganze Zahl, die anfangs gleich Null ist. Die Operation

REQUEST SEMA1;

erniedrigt den Wert der Zahl um 1, vorausgesetzt, das Ergebnis wird nicht negativ; wäre letzteres der Fall, wird die Ausführung der Anweisung verschoben, bis sie durchgeführt werden kann. Das wird der Fall, wenn von einer anderen Task eine Operation

RELEASE SEMA1;

ausgeführt wird, die den Wert von SEMA1 um 1 erhöht. Eine auf diesen Semaphor „wartende" Task kann nun weiterlaufen. Auf diese Weise können „Ereignisse" innerhalb des Programmsystems weitergemeldet werden.

Es ist auch möglich, den Ablauf (die „Lebensdauer") einer Task insgesamt mit einem Semaphor zu verknüpfen, um den Ablauf anderer Tasks darauf abstimmen zu können. Das geschieht durch die Spezifikation

ACTIVATE T2 USING SEMA2;

die eine REQUEST-Operation bei der Aktivierung, eine RELEASE-Operation am Ende der Task bewirkt.

Zur Vereinfachung der Verwaltung von Daten, die zu einer Zeit von mehreren Tasks gelesen, aber nur von einer geändert werden dürfen, dienen komplexere Synchronisationselemente, die BOLT-Variablen und auf sie wirkende Operationen. RESERVE und FREE regeln den exklusiven (z. B. Update-) Zugriff; ENTER und LEAVE regeln den simultanen (z. B. LESE-) Zugriff, wenn sie auf BOLT-Variable angewandt werden. Eine obere Grenze der Simultanität kann bei der Deklaration festgelegt werden.

3.5. Vergabe von Betriebsmitteln

Das Betriebssystem vergibt Betriebsmittel im Konfliktfall gemäß den Prioritäten der anfordernden Tasks. In PEARL werden Prioritäten (dynamisch) im Task-Aktivierungs- oder Task-Fortsetzungs-Statement spezifiziert.

Beispiel: ACTIVATE T1 PRIORITY 3;

Bewerben sich mehrere Tasks um ein Betriebsmittel (beispielsweise um einen Prozessor, um Speicherraum oder um Datenkanäle), so wird das Betriebsmittel der Task mit der höchsten Priorität (kleinste Zahl) zugewiesen.

4. Algorithmischer Sprachteil

Zur Formulierung der Aufgaben, die die einzelnen Tasks eines PEARL-Programmsystems zu lösen haben, dient der algorithmische Sprachteil. Einen Eindruck von der Art, wie Deklarationen, Anweisungen und Prozeduren zu schreiben sind, sollen die folgenden Abschnitte vermitteln.

4.1. Deklaration von Bezeichnern

Die wichtigsten Objekte in einer höheren Programmiersprache sind die frei wählbaren Bezeichner, die zur Identifikation der Variablen, Konstanten, Routinen usw. in einem Programm dienen. Um solche Bezeichner einzuführen, ist ein declare-sentence nötig, der mit einem der Schlüsselwörter DECLARE oder DCL beginnt und eine Liste von identity-declarations enthält. So wird durch

DCL X REAL

ein Bezeichner X eingeführt, der im Programm für einen reellen Zahlenwert stehen soll. X ist in Folge dieser Deklaration nicht irgendein Bezeichner, sondern ein reference-one-identifier, der benutzt wird zur Bezeichnung eines Speicherplatzes, in dem eine reelle Zahl – aber nichts anderes – aufbewahrt werden kann. Das Schlüsselwort REAL ist eines der verschiedenen value-attributes.

Natürlich können mehrere Bezeichner, die mit demselben Attribut verknüpft sein sollen, zusammen deklariert werden. Mit

DCL (I, J, K) INT

erfolgt die Vereinbarung dreier Variabler, die ganzzahlige Werte repräsentieren können. Auf Wunsch kann ein Objekt, für das in einer Deklaration ein Bezeichner eingeführt wird, gleichzeitig initialisiert werden. So würde durch

DCL (I, J, K) INT: = (3, 5, 7)

das gleiche erreicht wie durch die oben angegebene Deklaration, gefolgt von 3 Zuweisungen

I: = 3; J: = 5; K: = 7;

Die erwähnten Attribute REAL und INT sind zusammen mit BIN (für binary) die Charakteristika der Grunddatentypen in PEARL. Da in einer gegebenen Implementation eine im Speicher des Rechners abgelegte reelle Größe immer nur eine anlagenbedingte Näherung für eine reelle

Zahl sein kann, gibt PEARL dem Programmierer die Kontrolle über die Genauigkeit der Näherung explizit in die Hand. So wird durch die Deklaration

DCL Y REAL (10);

verlangt, daß der Speicherplatz, der mit Y bezeichnet wird, eine Mantisse mit der Genauigkeit von wenigstens 10 Dezimalstellen aufzunehmen in der Lage ist. Die Deklaration

DCL BITS BIN (16);

stellt sicher, daß mit dem Bezeichner BITS ein BIT-String angesprochen werden kann, der mindestens 16 BIT-Stellen umfaßt.

Eng verwandt mit den Grundtypen ist eine Zeichenkette. Zusammen mit dem Schlüsselwort STRING ist hier die Angabe einer Höchstzahl von Zeichen erforderlich, die in der Kette enthalten sein sollen. Daneben gibt es eine Reihe von Datentypen, die vor allem die Real-time-Sprachelemente von PEARL erfordern. Alle diese Datentypen sind einfacher Art, ihre Attribute zusammengefaßt unter dem Begriff simple-attributes.

Referenzen oder Zeigervariable

Eine der bedeutenden Eigenschaften von PEARL ist die Möglichkeit, nicht nur alle Arten von BIT-Mustern anzusprechen, die im Speicher eines bestimmten Computers als interne Darstellung von Daten vorkommen können, sondern darüber hinaus auch die Verweise auf die Speicherplätze – im Grunde die Speicheradressen also – manipulieren zu können. Diese Speicheradressen oder Referenzen können als eine besondere Art von Datentypen aufgefaßt werden. In der gleichen Weise, wie das Bit-Muster, das die interne Darstellung einer ganzen Zahl ist, über einen Bezeichner im Programm angesprochen werden kann, kann auch ein Bit-Muster, das eine Referenz, eine Speicheradresse, darstellt, über einen Bezeichner im Programm angesprochen werden. Ein solcher Bezeichner wird durch eine Deklaration der Art

DCL XX REF REAL;

eingeführt. Der Bezeichner XX, ein reference-two-identifier, dient zur Ansprache eines Speicherplatzes, der eine Referenz auf einen anderen Speicherplatz aufnehmen kann, jedoch nur eine solche, die auf einen Speicherplatz verweist, der für die Aufnahme von Daten des Typs REAL vorgesehen ist. Dadurch bleibt es möglich, alle Typprüfungen schon zur Übersetzungszeit durchzuführen, was erheblich zur Effizienz der Programme beiträgt. Nach einer Zuweisung im Ablauf des Programms wie

XX: = X;

ist der „Wert", der durch XX bezeichnet wird, die Referenz, die intern für den Bezeichner X eingeführt wurde. XX ist also die Adresse, die zu dem Speicherplatz führt, der Gleitpunktzahlen aufnehmen kann, die mit X im Programm angesprochen werden können. XX wird daher auch als „Zeiger" betrachtet, während X gemeinhin als „Variable" angesprochen wird.

Programmkonstante

Wenn es häufig nötig ist, innerhalb eines Programmes mehrfach die gleichen Konstanten zu verwenden, bietet PEARL die Möglichkeit, „Abkürzungen" für Datenkonstante einzuführen. Nach einer Deklaration

DCL PI VAL REAL (16) = 3.14159 26535;

kann der Bezeichner PI überall dort im Programm geschrieben werden, wo sonst die Zahlkonstante stehen müßte, und zwar mit genau der gleichen Wirkung. Zur Verdeutlichung sei darauf hingewiesen, daß an Bezeichner, die „Abkürzungen" sind (also keine „Variablen", auch keine „Zeiger") nicht zugewiesen werden kann.

4.2. Erweiterbarkeit der Sprache

Die Programmiersprache PEARL ist eine erweiterbare Sprache. Der Programmierer ist nicht beschränkt auf die ursprünglich in der Sprache definierten und durch eine spezielle Implementation weiter eingeschränkten Datentypen wie Ganzzahlen, Bit-Strings usw. Er kann vielmehr nach Belieben neue, problemspezifische Datentypen in sein Programm einführen. Diese neuen Datentypen können im weiteren Verlauf dann in der gleichen Weise als Attribute mit frei gewählten Bezeichnern verküpft werden wie die standardmäßig schon vorhandenen.

Neue Datentypen

Im allgemeinen sind neue Datentypen Komplexe, die aus standardmäßig vorhandenen Datentypen aufgebaut sind. Das structure-attribute von PEARL kann dazu benutzt werden, einen Bezeichner für einen Datensatz einzuführen, der aus Elementen mit möglicherweise ganz unterschiedlichen simple-attributes aufgebaut ist. Ein sehr einfaches Beispiel dafür ist die Deklaration

DCL RS STRUCT (VALUE REAL, NAME STRING (12));

Der Bezeichner RS ist der Name für ein Objekt, das zwei Elemente umfaßt, eine Gleitpunktzahl und einen Zeichen-String mit maximal 12 Zeichen. Um das Element, das eine Gleitpunktzahl repräsentiert, ansprechen zu können, wird die selection

RS. VALUE

benutzt, also die Kombination aus dem durch die Deklaration eingeführten Bezeichner (hier RS) und dem in der Strukturbeschreibung angegebenen Selektor (hier VALUE). Eine solche selection hat eine gewisse Ähnlichkeit mit einem indizierten Namen wie etwa B (3) oder C (I + K), mit dem ein Element eines Feldes angesprochen werden kann. Der wesentliche Unterschied gegenüber einem indizierten Namen ist der, daß eine selection auch bei einer noch so kompliziert aufgebauten Struktur eine genaue Speicherplatzbestimmung (Adreßberechnung) zur Übersetzungszeit gestattet, wogegen etwa C (I + K) stets eine Auswertung zur Laufzeit des Objektprogrammes erfordert. Um das immer zu gewährleisten, kann eine Struktur nur aus solchen Objekten aufgebaut werden, deren Speicherplatzbedarf zur Übersetzungszeit bestimmbar ist. Der Datensatz, der mit RS aufgrund der vorange-

gangenen Deklaration angesprochen wird, kann als ein Exemplar eines neuen Datentyps angesehen werden. Damit weitere Exemplare eines solchen neuen Datentyps einfach und in der gleichen Weise wie REAL- und BIN-Größen deklariert werden können, kann durch eine Typ-Deklaration ein Schlüsselwort für den neuen Datentyp festgelegt werden. So wird beispielsweise durch

```
TYPE COMPL = STRUCT (RE REAL, IM REAL);
```

ein neuer Datentyp eingeführt, dessen Schlüsselwort COMPL (vom Programmierer frei gewählt) in der Folge gleichberechtigt wie alle anderen z. B. REAL und INT verwendbar ist. Mit einer Deklaration

```
DCL (V, W) COMPL
```

würden zwei Bezeichner für zwei komplexe Variable eingeführt.

Neue Operatoren

Die Möglichkeit, neue Datentypen einzuführen, indem über eine TYPE-Deklaration ein neues attribute geschaffen wird, wird ergänzt durch die Möglichkeit, neue Operatoren zu deklarieren. Erst damit ist es letztlich gerechtfertigt, von einer erweiterbaren Sprache zu sprechen. Auch hier soll ein Beispiel einen Eindruck von diesem Hilfsmittel geben. Um einen Operator zu vereinbaren, der es gestattet, eine Formel der Art

```
W: = W + V;
```

zu schreiben, obwohl PEARL standardmäßig keine komplexe Arithmetik vorsieht, ist lediglich die Operatorvereinbarung

```
OPTR +    (COMPL, COMPL) COMPL =
          ((A, B) COMPL)
RETURN ((A.RE + B.RE), (A.IM + B.IM))
```

notwendig. Sie unterscheidet sich äußerlich kaum von einer Prozedurvereinbarung, wie der nächste Absatz zeigen wird. Nach dem Zeichen „=" steht eine routine, der Rumpf einer Prozedur. Der Hauptunterschied zwischen Prozedur- und Operatorvereinbarung liegt in der Verwendbarkeit des durch sie eingeführten Bezeichners, des Prozedurnamens im einen Fall, des Operatorzeichens (im Beispiel +) bzw. -bezeichners im anderen Fall. Ein Prozedurname unterliegt der allgemeinen Regel, daß ein Bezeichner eindeutig sein muß, daß er nur einmal in einer Deklaration vorkommen darf. Operatorzeichen und Bezeichner dürfen beliebig oft wieder verwendet werden, denn es soll damit gerade erreicht werden, daß einmal mit J + *l* eine Addition von Ganzzahlen, eventuell sofort danach mit V + W aber eine Addition zweier komplexer Zahlen erfolgt.

4.3. Prozeduren

PEARL ist eine prozedurorientierte höhere Sprache. Die Art und Weise, in PEARL eine Prozedur zu vereinbaren, sie aufzurufen und die genaue Wirkungsweise der Parameterübernahme zu kennen, ist deshalb von zentraler Bedeutung. Einen kleinen Eindruck von dieser komplexen Programmierhilfe soll in diesem nur der Einführung dienenden Abschnitt anhand eines sehr einfachen und im Grunde „sinnlosen" Beispieles gegeben werden.

Ein Prozedur UP mit zwei Parametern und der Wirkung, den ersten Parameter um den zweiten zu erhöhen, wird in PEARL wie folgt vereinbart:

```
DCL UP PROC = (A REAL, B VAL REAL)
        BEGIN A : = A + B; END
```

Die Wirkung eine Prozeduraufrufes

```
UP (X, 2)
```

kann so beschrieben werden: Der Aufruf selbst wird ersetzt durch die Anweisungsfolge

```
BEGIN
      DCL A REAL       = X,
          B VAL REAL = 2
      BEGIN A : = A + B; END;
END
```

Diese wird so ausgeführt, als stünde sie am Ort des Prozeduraufrufes. So verweist der Bezeichner A zu derselben Größe, demselben Speicherplatz, wie der Bezeichner X; der Bezeichner B ist eine „Abkürzung" für die Zahlkonstante 2. Die Formel A := A + B hat deshalb exakt die gleiche Wirkung wie die Formel X := X + 2. Um eine so einfache Ersetzungsregel gewährleisten zu können, verlangt PEARL, daß jeder Prozedurparameter spezifiziert sein muß, das heißt, zu jedem Bezeichner für einen formalen Parameter gehört das attribute des Wertes, der beim Aufruf aktuell eingesetzt werden soll. Als weitere angenehme Folge davon kann schon zur Übersetzungszeit eine Verträglichkeitsprüfung zwischen Prozedurvereinbarung und Prozeduraufruf erfolgen, was zu erheblich effektiverem Laufzeitverhalten beiträgt.

4.4. Besondere Spracheigenschaften

Um Listenverarbeitung treiben zu können, kann es notwendig sein, einzelne strukturierte Größen miteinander zu verketten. Notwendig dazu ist die Möglichkeit, in eine strukturierte Größe als ein Element einen „Zeiger" aufnehmen zu können, der auf eine gleichartig strukturierte Größe verweisen kann. Die Deklaration

```
TYPE LINK =
STRUCT (CONTENTS STRUCT (anything),
            POINTER REF LINK);
```

führt einen neuen Datentyp LINK ein, der im weiteren Verlauf als attribute verwendet werden kann. Dieser neue Datentyp kommt jedoch auch in seiner Deklaration selbst schon vor, und zwar zusammen mit dem Schlüsselwort REF als attribute für die mit POINTER ansprechbare Größe der Struktur. Durch die nachfolgende Deklaration

```
DCL LIST (1 : 100) LINK;
```

wird ein Feld von hundert Elementen des Typs LINK eingerichtet. Über eine Schleife wie

```
FOR I TO 99
DO LIST (I) .POINTER := LIST (I + 1); DONE;
LIST (100)  .POINTER := LIST (1);
```

werden die einzelnen Elemente ringförmig verkettet.

Wiederholungsschleife

Die angegebene Form FOR I TO 99 zur Steuerung einer Schleife ist nur eine der möglichen Formen, die PEARL vorsieht. Der vollständige Schleifenkopf kann mit

```
FOR I FROM J BY K TO L WHILE P
```

jede nur denkbare Steuerung beschreiben. In der Mehrzahl der Anwendungsfälle ist jedoch der Steuermechanismus viel einfacher. Da jede Gruppe von Schlüsselwort und Ausdruck (z. B. BY K) fehlen kann, wenn eine standardmäßige Normalform (z. B. BY 1) ausreicht, ist der Compiler in der Lage, stets die optimale Codefolge abzusetzen. Wird nur eine DO-Gruppe programmiert, ohne Steuerkopf, so wird diese Befehlsfolge eben immer wieder durchlaufen, bis aus ihr herausgesprungen wird.

Verteilersprünge

Ein weiteres Hilfsmittel geübter Programmierer, der Verteilersprung, findet sich ebenfalls in PEARL wieder. So wird mit

```
CASE I
IN
      Anweisungsfolge 1
,
      Anweisungsfolge 2
,
:       .
:
,
      Anweisungsfolge n
OUT
      Alternative
ESAC;
```

verlangt, daß in Abhängigkeit des ganzzahligen Wertes von I die erste, zweite bzw. n-te Anweisungsfolge ausgeführt oder aber die Alternative durchlaufen wird.

5. Systemteil

5.1. Aufgaben des Systemteils

Im Systemteil beschreibt der Programmierer auf dem Niveau einer höheren Programmiersprache sowohl

- die gesamte Hardware-Konfiguration des jeweils einzusetzenden Prozeßrechensystems als auch
- die symbolischen Namen für jene Elemente, die im Problemteil des Anwenderprogramms angesprochen werden sollen.

Diese Fähigkeiten von PEARL bringen entscheidende Vorteile mit sich:

- Der Systemplan der jeweiligen Konfiguration wird im Programm selbst leicht lesbar dokumentiert.
- Mit Hilfe des Systemteils ist es möglich, ein für den jeweiligen Anwendungsfall zugeschnittenes und somit optimales Betriebssystem automatisch zu generieren.
- Die vollständige Beschreibung der Verknüpfungen zwischen Hard- und Software im Systemteil macht eine für jeden Rechner spezielle "job control language" für diesen Zweck überflüssig.
- In den Ein-/Ausgabe-Anweisungen des Problemteils muß die oft sehr umfangreiche Beschreibung des Weges zwischen Datenquelle und Datensenke nicht mehr vorgenommen werden, da man lediglich die im Systemteil definierten symbolischen Namen anzugeben hat.

5.2. Syntax und Semantik des Systemteils

Im Systemteil werden die Verbindungen zwischen jeweils zwei Geräten mit der jeweils zugelassenen Richtung des Datenflusses beschrieben. Z. B. wird durch

```
CPU ←→ CHN;
```

ausgesagt, daß ein Gerät, dessen Gerätebezeichnung „CPU" lautet, mit einem Gerät der Bezeichnung „CHN" derart verbunden ist, daß in beiden Richtungen ein Datenfluß möglich ist. Die Gerätebezeichnungen, die vom Implementierer festgelegt wurden, sind dem Compiler des PEARL-Quellenprogramms bekannt und werden von ihm mit den jeweils zutreffenden Programmen der Grundsoftware in Verbindung gebracht.

Erscheinen in einem System dieselben Geräte mehrmals, so müssen sie, um identifiziert werden zu können, vom Programmierer numeriert werden:

```
MPX (1) ← DIGIN;
MPX (2) → DIGOUT;
```

Diese Gerätebezeichnungen existieren in der angegebenen Bedeutung jedoch nur im Systemteil. Sollen im Problemteil Geräte angesprochen werden, so müssen diese mit Namen versehen werden:

```
MULT: MPX (2) ←→ CHN;
```

In diesem Beispiel ist das Gerät mit der Geräte-Bezeichnung MPX, welches mit der Unterscheidungs-Nummer 2 versehen wurde, im Problemteil mit dem vom Programmierer gewählten Namen „MULT" ansprechbar.

Die Beschreibung der Konfiguration an der Schnittstelle zwischen den peripheren Geräten des Prozeßrechners und dem Prozeß selbst stellt einen Sonderfall dar. Einerseits muß man die an die Prozeßperipherie angeschlossenen Prozeßendstellen (z. B. Meßfühler, Stellglieder) mit frei wählbaren Namen versehen können, andererseits ist es nicht sinnvoll, über den Compiler Grundsoftware-Funktionen für die Prozeßendstellen zur Verfügung zu stellen, da auf das spezielle Verhalten der im allgemeinen großen

Anzahl verschiedener Meßfühler, Stellglieder usw. durch das jeweilige Anwenderprogramm im Problemteil selbst eingegangen werden muß. Z. B. sind bei:

```
TEMP: → ANINPUT;
LAMP: ←DIGOUT;
```

„TEMP" und „LAMP" die vom Programmierer gewählten und im Problemteil anzusprechenden Namen von Prozeßendstellen, „ANINPUT" und „DIGOUT" die Bezeichnungen der mit den Prozeßendstellen verbundenen prozeßperipheren Geräte.

Besitzt ein Gerät mehrere Anschlüsse, so müssen diese besonders gekennzeichnet werden. Beispielsweise werden in:

```
MPX (2)∗ 0 ← DIGIN (1);
        ∗ 1 ← SENSE : ANIN;
```

die Anschlüsse 0 und 1 des Gerätes MPX (2) angesprochen. (Bei direkter Aufeinanderfolge desselben Geräts braucht dessen Geräte-Bezeichnung nur beim erstenmal geschrieben zu werden; in den nachfolgenden Anweisungen wird dann nur noch die Anschlußnummer angegeben.)
Bei digitalen Ein-/Ausgabe-Werken sind noch zusätzliche Anschluß-Informationen erforderlich. So sind z. B. bei:

```
COUNTER: → DIGIN ∗ 1 ∗ 3, 4;
```

außer der Kanalnummer „1" noch anzugeben die Nummer der ersten belegten Bitstelle („3") sowie die Anzahl der belegten Bitstellen („4"). Es ist dabei in PEARL erlaubt, daß die Anzahl der zu belegenden Bitstellen größer ist als die noch freien des gerade betrachteten Kanals. In diesem Fall werden vom Compiler die restlichen Bitstellen dem benachbarten Kanal zugeordnet.

Bei der Beschreibung umfangreicher Prozeßrechensysteme ist es zweckmäßig, daß mehrere Geräte desselben Typs zu einem eindimensionalen Feld zusammengefaßt werden können. Die Definition eines solchen „Geräte-Feldes" im Systemteil von PEARL geschieht einerseits durch die Angabe des Feldnamens und der (festen) Komponentenanzahl und andererseits durch die Beschreibung sämtlicher Anschlüsse, die durch die Verwendung der einzelnen Feldkomponenten angesprochen werden können.

Beispiel:

```
TEMP (1 : 24) : → ANIN (1) ∗ 7 + ANIN (1) ∗ 10
                + ANIN (3) ∗ (10 : 15) + ANIN (4) ∗ 17
                + ANIN (5) ∗ (23 : 37);
```

In dieser Anweisung wird ein Feld „TEMP" mit 24 Komponenten definiert. Der 1. Komponente wird der Anschluß „7" des mit der Kennziffer „(1)" versehenen Gerätes der Bezeichnung „ANIN" zugeordnet, der 2. Komponente der Anschluß „10" dieses Gerätes, den Komponenten 3 bis 8 die Anschlüsse „10" bis „15" des mit der Kennziffer „(3)" versehenen Gerätes „ANIN" usw. Die Anzahl der definierten Komponenten muß mit der Anzahl der zugewisenene Anschlüsse übereinstimmen.

6. Die Ein-/Ausgabe in PEARL

Die Ein-/Ausgabe in PEARL umfaßt

a) den Datenaustausch mit und zwischen peripheren Geräten und Files – ausgeführt durch „communication-statements". Geräte („devices") werden im Systemteil („system-division") eingeführt. Sie müssen nicht unbedingt physikalisch existieren, wie z. B. ein Schnelldrucker oder ein Analog-Digital-Konverter, sondern können auch logischer Natur sein, wie z. B. Spuren eines Plattenstapels;
b) die Organisation von Datenmengen – ausgeführt durch „file-handling-statements".

Der folgende Text beschäftigt sich hauptsächlich mit den unter a) aufgeführten „communication-statements", d. h. mit den E/A-Anweisungen im engeren Sinn.
Diese E/A-Anweisungen werden nach folgendem Schema eingeteilt:

		nicht-organisierte E/A auf Geräte wirkend	organisierte E/A auf Files wirkend
nicht-formatierte E/A		MOVE Anweisung	TRANSFER Anweisung
formatierte E/A	zeichenweise E/A	READ/WRITE Anweisung	READ/WRITE Anweisung
	graphische E/A	SEE/DRAW Anweisung	SEE/DRAW Anweisung

Im einzelnen enthält eine E/A-Anweisung Angaben über

- die Art des Datenaustausches (formatiert, nicht formatiert, organisiert, nicht organisiert, zeichenweise, graphisch) – angezeigt durch die Schlüsselwörter MOVE, TRANSFER, READ, WRITE, SEE, DRAW;
- „Quelle" und „Senke" des Datenflusses (Arbeitsspeicherzellen, Geräte, File);
- Richtung des Datenaustausches – angezeigt durch die Schlüsselwörter FROM, TO;
- eine eventuelle Formatierung – eingeleitet durch das Schlüsselwort FORMAT.

Außerdem sind noch folgende Besonderheiten bei E/A-Anweisungen in PEARL zu beachten:

- (Quelle und Senke, speziell aber) Geräte und Files werden durch selbstgewählte Namen angesprochen.

  Beispiel:

  ```
  DCL LINEPRINTER, QDEVICE, ADCREGISTER
  DEVICE;
  ```

  Wie man Namen an Geräte verteilt, wird aus der Beschreibung des Systemteils ersichtlich.
- Wird in einer formatierten E/A-Anweisung kein Zielgerät genannt, so wird ein (vom Programmierer) vorher deklariertes Gerät standardmäßig eingesetzt.

6.1. Nicht-formatierte E/A-Anweisungen

Bei nicht-formatierten E/A-Anweisungen wird keine Umwandlung der übertragenen Daten in eine (für den Menschen

verständliche) zeichenweise oder graphische Darstellung vorgenommen. Sie dient demnach vorzugsweise zur systeminternen Kommunikation.

Die MOVE-Anweisung

Aufgrund ihrer Eigenschaften kann die MOVE-Anweisung zum Betrieb von nicht standardisierten Geräten und zur Abwicklung der (geeichten) Prozeß-Ein-/Ausgabe verwendet werden.

Die MOVE-Anweisung hat folgenden Aufbau:

MOVE Quelle TO Senke;

oder

MOVE Quelle TO Senke GAUGE Eichprozeduraufruf;

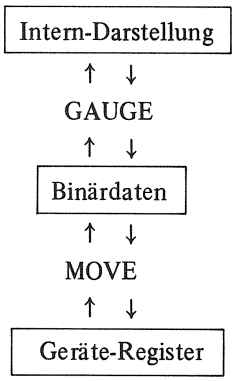
Als „Quelle" und „Senke" kann eine Arbeitsspeicherzelle oder ein Gerät angegeben werden.

Durch die MOVE-Anweisung werden also binäre Daten zwischen Arbeitsspeicherzellen und Geräten oder zwischen verschiedenen Geräten ausgetauscht.

Beispiel:

```
/* NORMALE MOVE-ANWEISUNG */
DCL ADCREGISTER DEVICE;
DCL ADCSTATUS BIN;
MOVE ADCREGISTER TO ADCSTATUS;
```

Die MOVE-Anweisung kann zusätzlich den Aufruf einer Eichprozedur (eingeleitet durch das Schlüsselwort GAUGE) enthalten, die eine Umrechnung der übertragenen Daten vor einer Ausgabe oder nach einer Eingabe erlaubt.



Beispiel:

```
/* DEKLARATIONEN */
DCL (MEASURE, BO) REAL;
DCL QDEVICE DEVICE;
DCL QUAD PROC (REAL, REAL) BIN = (AO REAL,
X REAL)
BEGIN
DCL HELP INT;
HELP := SQROOT (X-AO);
RETURN BITS (HELP);
END; ;
/* MOVE-ANWEISUNG MIT AUFRUF EINER
   GAUGE-PROZEDUR */
MOVE MEASURE TO QDEVICE GAUGE QUAD, BO;
```

Die TRANSFER-Anweisung überträgt wie die MOVE-Anweisung Daten ohne Umwandlung und Formatierung.

„Quelle" oder „Senke" in dieser Anweisung ist immer ein File.

Die TRANSFER-Anweisung ist hier nur der Vollständigkeit halber aufgeführt. Sie gehört ihrer Wirkungsweise nach zu dem hier nicht näher behandelten "file-handling".

6.2. Formatierte E/A-Anweisungen

Die Umwandlung zwischen interner und externer Informationsdarstellung wird durch Formate kontrolliert.

Zu diesem Zweck gibt es in PEARL den Datentyp FORMAT.

Beispiel: DCL LAYOUT FORMAT;

Mögliche Operationen auf den Typ FORMAT sind Zuweisung zur FORMAT-Variablen (FORMAT-Konstante werden in die Zeichen „F'" und „'" eingeschlossen) und Verkettung.

Beispiel:

```
LAYOUT := F'E (2, 5, 3)'//F'I (7)';
```

Formatierte E/A-Anweisungen sind folgendermaßen aufgebaut:

```
READ  }
      } Senke FROM Quelle FORMAT Format;
SEE   }
```

Bei der Eingabe ist als „Quelle" ein Gerät oder ein File zugelassen. Die „Senke" (interne Endstelle) kann eine Liste von Variablen sein.

```
WRITE }
      } Quelle TO Senke FORMAT Format;
DRAW  }
```

Bei der Ausgabe ist als „Quelle" (interne Endstelle) eine Liste von Variablen zugelassen, als „Senke" ein Gerät oder ein File.

Als „Format" ist eine Liste von Daten vom Typ FORMAT erlaubt. Bei der Abarbeitung von E/A-Anweisungen wird dann jedem Element der internen Endstelle ein Element der Format-Liste zugeordnet.

Die formatierten E/A-Anweisungen sind aufgeteilt in die zeichenweise E/A und die graphische E/A, die sich im wesentlichen durch die zugelassenen FORMAT-Konstanten unterscheiden.

Die READ/WRITE-Anweisung (zeichenweise E/A).

Die FORMAT-Konstanten für zeichenweise E/A in PEARL sind den aus FORTRAN und PL/1 bekannten Formaten ähnlich.

Zusätzlich gibt es Formate, um Daten vom Typ CLOCK und DUR darstellen zu können.

Besonders zu erwähnen ist eine FORMAT-Konstante (und zwar F"), mit deren Hilfe alle Daten der Typen REAL, INT, BINAL, CLOCK, DUR, STRING ausgegeben werden können.

Beispiel:
```
DCL TEXT STRING (4);
TEXT := „ABCD“;
WRITE TEXT FORMAT F″;
```

Die SEE/DRAW-Anweisung (graphische E/A)

Bei der graphischen Ausgabe sind als „Quelle“ bzw. „Senke“ Daten vom Typ INT und REAL zugelassen. Ihre graphische Bedeutung erhalten sie erst durch die ihnen zugeordneten (graphischen) Formatelemente.

Um z. B. einen Punkt darstellen zu können, braucht man Information über seine Koordinaten, seine Helligkeit (und evtl. Farbe). Folglich gibt es FORMAT-Konstante, mit deren Hilfe Elemente der „Quelle“ als Koordinaten-, Helligkeits- oder Farbwert eines Punktes interpretiert werden können.

Durch die Anweisungsfolge

```
DCL DISPLAY DEVICE;
:
/* INITIALISIERUNGEN WURDEN WEGGELASSEN */
:
DRAW (O, O) TO DISPLAY FORMAT F 'XA, YA';
```

kann z. B. ein Punkt auf einem Display im Ursprung des Koordinatensystems dargestellt werden.

Außerdem gibt es Formate zum

- Vergrößern, Verschieben und Löschen eines Bildes,
- automatischen Inkrementieren einer Koordinate,
- Interpolieren von Punktfolgen,
- Drehen von Darstellungen.

Der Vorteil dieser Art von graphischen Ausgabe ist die Möglichkeit, einmal aufgebaute fest Datenlisten durch Anpassung der Formatlisten in der gewünschten Weise interpretieren zu können.

7. Schluß

PEARL wurde von einem Arbeitskreis entwickelt, an dem Mitarbeiter aus deutschen Hochschulinstituten und Industriefirmen teilnahmen. Die Arbeit wurde vom „Bundesministerium für Bildung und Wissenschaft“ und vom Projekt „Prozeßlenkung mit DV-Anlagen“ bei der GFK Karlsruhe gefördert.

Wir möchten diese Gelegenheit benutzen, um allen Mitarbeitern und Institutionen zu danken, die den Entwurf PEARL unterstützt haben. Insbesondere danken wir H. P. ELZER für seinen unermüdlichen Einsatz bei der Koordinierung und Durchführung der PEARL-Definition.

Literaturangaben

[1] 1 800 process supervisory program (PROSPRO/1 800). IBM no. H 20-0473-1 (1968).

[2] *Donald G. Bates:* PROSPRO/1 800. IEEE transactions on industrial electronics and control instrumentation, Vol. IECI-15, no. 2, December 1968, p. 70–75.

[3] BICEPS summary manual/BICEPS supervisory control. GE Proc. Comp. Dept., A GET-3539 (1969).

[4] ATLAS Abbreviated Test Language for Avionics Systems. ARINC Specification 416-1, Aeronautical Radio Inc., June 1969.

[5] *Gene S. Metsker:* Checkout test language: An interpretive language designed for aerospace checkout tasks. Fall Joint Comp. Conf., (1968), p. 1329–1336.

[6] Bendix OPTOL Programming System. The Bendix Corporation, Teterboro, New Jersey, March 1968.

[7] PLACE The compiler for the programming language for automatic checkout equipment. Battelle Memorial Institute, Columbus Laboratories, Technical Report AFAPl-TR-68-27, May 1968.

[8] Official definition of CORAL 66. Inter-Establishment Committee for Computer Applications, February 1970.
A. A. Callaway, D. C. Ae., D. Eng., A. F. R. Ae. S.: A guide to CORAL programming. Royal Aircraft Establishment, Technical Report 70102, June 1970.

[9] INDAC8. Digital equipment corporation, Maynard Mass.

[10] PAS1 Prozeß-Automationssprache 1. BBC, Mannheim.

[11] *K. H. Timmesfeld* et al.: Pearl a proposal for a process and experiment automation real-time language. Gesellschaft für Kernforschung mbH, Karlsruhe, PDV-Bericht KFK-PDV1, April 1973.

[12] *E. W. Dijkstra:* Co-operating sequential processes. published in “programming languages”, edited by F. Genuys, Academic Press, London 1968.