atp-Softwarepraxis

Die Programmiersprache Ada

R. G. Herrtwich und A. Knoll, Berlin

In einer Reihe ähnlich aufgebauter Beiträge werden innerhalb der atp-Softwarepraxis regelmäßig Programmiersprachen für Prozeßautomatisierungsaufgaben beschrieben. Dabei geht es nicht darum, eine Sprache im Detail zu beschreiben, sondern darum, sie anhand von Beispielen im Überblick darzustellen und ihre allgemeinen Leistungsmerkmale herauszuarbeiten. Im Rahmen dieser Reihe befaßt sich der vorliegende Artikel mit der Sprache Ada.

1. Einleitung

Ada entstand aufgrund einer Initiative des amerikanischen Verteidigungsministeriums, das sich als weltweit größter Softwareabnehmer angesichts der von ihm für die Softwarewartung aufgewendeten Unsummen im Jahre 1975 entschloß, eine einheitliche Programmiersprache für Anwendungen im Verteidigungsbereich zu schaffen. Derartige Anwendungen erfordern vorwiegend die Entwicklung sogenannter *eingebetteter Systeme*, bei denen ein Rechensystem mit Hardware und Software Bestandteil eines größeren technischen Systems ist. Solche Systeme finden sich nicht nur im militärischen Bereich, sondern auch in zivilen Anwendungen der Prozeßautomatisierung. Ada kann demzufolge in diesem Bereich eingesetzt werden.

Die Anforderungsdefinition, mit der das Ada-Projekt gestartet wurde [1], schrieb neben einer Vielzahl auf den Anwendungsbereich ausgerichteter Detailpunkte drei zentrale Forderungen für den Sprachentwurf fest: Die zu entwickelnde Sprache sollte

- die Programmzuverlässigkeit und -wartbarkeit gewährleisten,
- den Programmiervorgang als menschliche Aktivität unterstützen und
- hinsichtlich des Laufzeitverhaltens der in ihr geschriebenen Programme effizient sein.

Da vor allem die ersten beiden Ziele nicht allein durch eine Sprache erfüllt werden können, sondern instrumentelle Unterstützung für den Prozeß der Programmentwicklung erfordern, wurde neben der eigentlichen Sprachentwicklung frühzeitig Wert auf die Konzeption einer begleitenden Softwareentwicklungsumgebung unter dem Namen *Ada Programming Support Environment (APSE)* gelegt [2].

Aus einer Reihe um die Erfüllung der Anforderungsdefinition konkurrierender Sprachentwürfe wurde 1979 jener Entwurf zur Verfeinerung ausgewählt, der bei CII-Honeywell Bull unter der Leitung von *J. Ichbiah* entwickelt worden war. Im Jahr 1983 wurde diese Weiterentwicklung nach einer Phase intensiver Begutachtung durch etwa 500 Fachleute und Organisationen abgeschlossen und ein die Sprache Ada beschreibender Standard durch das amerikanische Standardisierungsgremium *ANSI* verabschiedet [3]. Erste Ada-Systeme erschienen kurze Zeit später auf dem Markt.

2. Allgemeine Kriterien

Orientierung der Sprache

Ada entstammt der ALGOL-Sprachfamilie und ist ein unmittelbarer Abkömmling der Sprache Pascal. Es handelt sich bei ihr folglich um eine universelle, blockorientierte, typgebundene, imperative, höhere Programmiersprache. Ada übernimmt einige bewährte Pascal-Konstrukte, etwa bei Kontrollstrukturen und Deklarationsmechanismen. In anderen Punkten erweitert Ada die in Pascal angebotenen Sprachmittel, z. B. bei der Vereinbarung von Datentypen. In vielen Punkten sieht Ada Konstrukte vor, die in Pascal nicht zu finden sind, so etwa bei der Modularisierung, bei der Beschreibung nebenläufiger Algorithmen und bei der Behandlung von Ausnahmesituationen. Dies führt dazu, daß der Umfang der Sprache (und demzufolge auch der Umfang von Ada-Programmiersystemen) gegenüber dem von Pascal um ein Vielfaches größer ist.

Definition der Sprache

Der Ada-Standard bildet die verbindliche Definition der Sprache. Er legt formal die kontextfreie Syntax der Sprache fest und gibt ausführliche, englische Erklärungen der einzelnen Sprachelemente. Auf der Basis des Standards wurden auch formale Semantikdefinitionen der Sprache erarbeitet [4; 5]. Verbindlich bei Konflikten hinsichtlich der Auslegung der Sprachbeschreibung – angesichts ihres Umfangs nicht zu vermeiden – sind jedoch allein Auskünfte des hierzu vom amerikanischen Verteidigungsministerium eingesetzten *Ada Joint Program Office (AJPO)*, das als genereller Ansprechpartner für Fragen im Zusammenhang mit Ada anzusehen ist.

Woher rührt die Komplexität der Sprachdefinition? Eigentlich hat man sich bemüht, durch die konsequente Verwendung einiger weniger Grundprinzipien den Sprach-

(Fortsetzung auf Seite 273)

(Fortsetzung von Seite 272)

umfang klein und allgemein zu halten. Diese Allgemeinheit stand jedoch im Widerspruch zur effizienten Implementierbarkeit der Sprache, so daß man durch eine Vielzahl von Regeln die beliebige Kombinierbarkeit von Sprachkonstrukten eingeschränkt hat. So liegt denn auch die Kompliziertheit der Sprache nicht an den ihr innewohnenden Prinzipien, sondern an den Randbedingungen, denen diese Prinzipien unterworfen sind.

Zertifizierung

Ada ist ein eingetragenes Warenzeichen des amerikanischen Verteidigungsministeriums, dessen Verwendung durch das *AJPO* überwacht wird. Nur ein System, das vom *AJPO* durch erfolgreiche Übersetzung und Ausführung von etwa 3000 Testprogrammen validiert wurde, darf als Ada-System bezeichnet werden. Eine einmal erhaltene Validierung gilt nur für ein Jahr und muß dann erneuert werden.

Unterstützung

Neben der institutionalisierten Unterstützung der Sprachpflege durch das *AJPO* gibt es mittlerweile eine Reihe von Anwendergruppen, die Gelegenheit zum Austausch über die Sprache bieten. Die international wichtigste Interessenvertretung dürfte dabei die *ACM Special Interest Group on Ada (Sigada)* sein, die eine zweimonatliche Publikation herausgibt, in der auch regelmäßig Überblicke über Ada-Produkte veröffentlicht werden [6]. Sie veranstaltet alljährlich Konferenzen zum Thema Ada.

Auch in Europa und Deutschland fanden bereits wiederholt Konferenzen über Ada statt, und namhafte Firmen haben sich frühzeitig an der Entwicklung von Ada-Systemen beteiligt. Demzufolge ist auch das Angebot an Schulungskursen über Ada groß.

Erlernbarkeit

Aus den bereits zuvor erläuterten Gründen ist die Sprache in ihrem Gesamtumfang recht schwer erlernbar. Es ist hilfreich, vor dem Erlernen von Ada bereits Erfahrungen in der Programmierung mit einer ALGOL-artigen Sprache gesammelt und an der Programmierung eines größeren Programmsystems mitgearbeitet zu haben. Dann nämlich lassen sich die über die Ideen der klassischen strukturierten Programmierung hinausgehenden Sprachmittel von Ada in ihrer Zielsetzung und Anwendbarkeit besser verstehen. Es gibt eine Reihe von Lehrbüchern, die speziell auf diesen Zugang zu Ada als „Zweitsprache" ausgerichtet sind [7 bis 11].

Der Ada-Standard selbst ist, wie bei Sprachdefinitionen üblich, als Lehrbuch für die Ada-Programmierung ungeeignet, auch wenn er für die Arbeit mit Ada-Systemen unentbehrlich ist. Letzteres gilt auch für Implementierungshandbücher von Herstellern, aus denen die Besonderheiten des jeweiligen Ada-Systems hervorgehen.

Lesbarkeit

Ada bleibt dem Pascal-Prinzip treu, Kontrollstrukturen und Datenvereinbarungen mit verhältnismäßig wenigen, prägnanten Schlüsselworten nach weitgehend dem gleichen Aufbauschema zu notieren. Verwendet der Programmierer darüber hinaus aussagekräftige Bezeichner für die von ihm eingeführten Programmobjekte, so entstehen lesbare Programme, die sich innerhalb der hierbei möglichen Grenzen selbst kommentieren. Zu dieser Lesbarkeit tragen auch die Modularisierungsmechanismen der Sprache bei, die es dem Betrachter gestatten, jeweils kleine Programmteile isoliert zu begutachten.

Programmierdisziplin

Ada unterstützt die schrittweise und unabhängige Entwicklung von Programmteilen durch getrennte Angabe der Schnittstelle einer Programmeinheit und deren Implementierung sowie durch die getrennte Übersetzbarkeit der erstellten Programmeinheiten. Dies ermöglicht nicht nur die Arbeitsteilung der an einem größeren Programmsystem arbeitenden Programmierer, sondern erlaubt auch, Teile von Ada-Programmen, die bereits in anderen Systemen eingesetzt wurden, auf elegante Weise wiederzuverwenden. Die getrennte Übersetzbarkeit reduziert zudem die Zeit und Kosten der Erzeugung eines lauffähigen Systems, da bei Vornehmen einer Änderung nur jene Teile neu übersetzt werden müssen, die von der Änderung betroffen sind.

Sicherheit und Zuverlässigkeit

Grundlage der sicheren Programmierung mit Ada ist die strenge Typisierung aller verwendeten Datenobjekte und deren blockgebundene Gültigkeit. Auch die Modularisierung und das mit ihr verbundene Geheimnisprinzip tragen zur Sicherheit bei.

Durch in der Sprache vorgesehene Mechanismen zur Ausnahmebehandlung läßt sich auf Fehler oder unerwartete Situationen beim Programmablauf reagieren, ohne daß der Programmlauf – wie sonst üblich – abgebrochen wird. Durch diese Mechanismen unterstützt Ada prinzipiell die Konstruktion zuverlässiger Systeme. Auch andere Sprachmittel, etwa die Möglichkeit der Zeitüberwachung von Befehlen, tragen hierzu bei.

Unter den vielen Einzelanforderungen, denen die Ada-Entwicklung gerecht werden mußte, waren auch einige, deren Erfüllung den Zielen von Sicherheit und Zuverlässigkeit entgegensteht. So erlaubt Ada z. B. den unsynchronisierten Zugriff mehrerer Prozesse auf gemeinsame Variable, gestattet es, Speicher ungeprüft freizugeben, und ermöglicht eine ungeprüfte Typkonvertierung. Man sollte von diesen Möglichkeiten der Sprache keinen Gebrauch machen.

Übertragbarkeit

Die Standardisierung der Sprache, verbunden mit der Validierung und Zertifizierung von Ada-Systemen, gewähr-

leistet eine vergleichsweise hohe Übertragbarkeit von Programmen auf verschiedene Rechner.

Ihre Grenzen findet diese Übertragbarkeit naturgemäß an jenen Stellen, wo mit den hierfür vorgesehenen Ada-Sprachmitteln spezielle Hardwareabhängigkeiten ausgedrückt werden, wie sie etwa bei der Systemprogrammierung nötig sind. Sie sind zwar durch die entsprechenden Konstrukte für Portierungen identifizierbar, jedoch nicht zentral an einer Stelle im Programm zusammengefaßt, so daß ihr Auffinden schwierig sein kann.

Projektabwicklung

Über die Sprache hinaus wird durch die Programmierumgebung *APSE* ein Instrumentarium zur strukturierten Abwicklung von Programmierprojekten mit Ada geschaffen. Hierzu gehört z. B. eine Datenbank, in der die Versionen einzelner Programmeinheiten verwaltet werden, die im Laufe des Projekts entstanden sind. Der Funktionsumfang der Programmierumgebung über einen gewissen Minimalrahmen hinaus ist bewußt offen gelassen worden. Die Einbettung von Spezifikations- und Verifikationshilfsmitteln – so diese einmal verfügbar sind – ist beabsichtigt.

Verfügbarkeit und Bewährtheit in der Praxis

Mittlerweile sind validierte Ada-Systeme für Rechner aller Leistungsklassen, beginnend beim PC/AT, verfügbar. Die Sprache wird naturgemäß im Verteidigungsbereich der NATO und in der damit verbundenen Industrie eingesetzt, gewinnt aber auch im zivilen Bereich, wie etwa beim Industrieanlagenbau, an Bedeutung.

Der Makel des Rüstungshintergrundes der Ada-Entwicklung führt gerade im europäischen Raum zu einem reservierten Einsatz der Sprache im zivilen Bereich. Auch Bedenken aufgrund der angesichts des Sprachumfangs von Ada recht hohen Wahrscheinlichkeit von Fehlern in Ada-Programmiersystemen hemmen die Verbreitung der Sprache [12]. Gerade deshalb (und aus eher pragmatischen Gründen) gehen auch die Hochschulen nur zögernd zur Verwendung der Sprache über.

Effizienz

Compiler und Laufzeitsysteme bestehender Ada-Systeme sind groß. Ihr Umfang z. B. auf dem PC/AT liegt bei 40000 bis 400000 (Ada-)Programmzeilen. Entsprechend lang sind auch die Übersetzungszeiten; ein „Turbo-Ada" ist bis auf weiteres nicht in Sicht. Allerdings handelt es sich bei den verfügbaren Übersetzern um moderne Produkte mit guten Code-Optimierern, so daß die erzeugten Programme bezüglich ihres Laufzeitverhaltens den Vergleich mit denen aus anderen, vermeintlich effizienteren Sprachen erzeugten, nicht zu scheuen brauchen. Auch die Größe des erzeugten Codes ist nicht notwendigerweise größer als bei anderen Sprachen.

3. Technische Kriterien

In der folgenden Vorstellung der Sprachmittel von Ada soll nur auf jene Mechanismen genauer eingegangen werden, die Erweiterungen gegenüber den entsprechenden Pascal-Konstrukten darstellen.

Datenstrukturen

Jede in Ada zur Aufnahme einer Datenstruktur verwendete Variable muß vor ihrem Gebrauch deklariert und typisiert werden. Dabei kann ihr bei Bedarf gleich ein Initialwert zugewiesen werden.

```
ZAEHLER : INTEGER := 0;
ZEICHEN1, ZEICHEN2 : CHARACTER;
```

Abgesehen von den aus Pascal bekannten Mengentypen stellt Ada alle aus imperativen Programmiersprachen geläufigen Typen zur Verfügung.

Typen mit diskretem Wertebereich

- Basistypen:

  BOOLEAN, INTEGER, CHARACTER

- Aufzählungstypen:

```
type AMPEL is (ROT, GELB, GRUEN);
type TAG is
    (MONTAG, DIENSTAG, MITTWOCH,
    DONNERSTAG, FREITAG,
    SAMSTAG, SONNTAG);
type LAMPEN is
    (GLUEHBIRNE, LEUCHTROEHRE);
```

- Untertypen:

```
subtype WERKTAG is
    TAG range MONTAG .. SAMSTAG;
subtype POSITIV is
    INTEGER range 1 .. INTEGER'LAST;
```

- Abgeleitete Typen:

```
type PROZESSOR_ZAHL is new POSITIV;
type PLATTEN_ZAHL is new POSITIV;
type WOCHENENDE is
    new TAG range SAMSTAG .. SONNTAG;
```

  Abgeleitete Typen haben den gleichen Wertebereich wie die zugehörigen Stammtypen, und es sind die gleichen Operationen auf ihnen definiert. Abgeleiteter Typ und Stammtyp sind jedoch nicht zuweisungskompatibel, so daß z. B. Variablen der obigen Typen PROZESSOR_ZAHL und PLATTEN_ZAHL einander nicht irrtümlich zugewiesen werden können.

Typen ohne diskreten Wertebereich

- Gleitkommatypen:

```
type REELL is digits 8;
type UNGENAU is digits 4;
type WAHRSCHEINLICHKEIT is
    digits 16 range 0.0 .. 1.0;
```

  Die Genauigkeit von Gleitkommazahlen ergibt sich aus den für ihre Darstellung insgesamt vorgesehenen Stellen. Sie ist folglich relativ zur Größe der Zahl.

- Festkommatypen:

```
type GEHALT is
   delta 0.01 range 0.0 .. 1000000.0;
```

Die Genauigkeit von Festkommazahlen ist absolut und resultiert aus der mit **delta** angegebenen Auflösung des Wertebereichs.

Zugriffstypen

```
type GEHALTSEINTRAG is access GEHALT;
```

Zugriffstypen erlauben die dynamische Erzeugung neuer Datenobjekte zur Programmlaufzeit. Mit Hilfe der Anweisung

```
G := new GEHALTSEINTRAG
```

erzeugt man zur Laufzeit ein neues Datenobjekt vom Typ GEHALT und besitzt dann in G einen Namen, mit dem man es ansprechen kann.

Zusammengesetzte Typen

- Statische Feldtypen:

```
type VEKTOR is array (1 .. 3) of REELL;
type MATRIX is array (1 .. 3, 1 .. 3) of REELL;
type DYN_VEKTOR is array (1 .. N) of REELL;
type DYN_MATRIX is
  array (1 .. N, 1 .. M) of REELL;
```

Die Größe dynamischer Felder wird zur Laufzeit bestimmt. Der Wert der dynamischen Feldgrenzen (hier N und M) muß erst bei der Erzeugung von Objekten dieses Typs feststehen.

- Feldtypen mit unspezifizierten Grenzen:

```
type STRING is
   array (POSITIV range < >) of CHARACTER;
type GEN_MATRIX is
  array (POSITIV range < >, POSITIV range < >)
   of REELL;
```

Felder mit unspezifizierten Grenzen dienen zur Formulierung von Algorithmen, für die keine Kenntnis der Feldgröße erforderlich ist. Sie repräsentieren jedes Feld des angegebenen Elementtyps, dessen Indizes einen Untertyp des angegebenen Indextyps bilden. Solche Typen sind besonders geeignet für Parameter von Unterprogrammen.

- Verbundtypen:

```
type DATUM is
  record
    TAG   : INTEGER range 1 .. 31;
    MONAT : INTEGER range 1 .. 12;
    JAHR  : INTEGER range 1900 .. 2100;
  end record;
```

- Parametrisierte Verbundtypen:

```
type PUFFER (LAENGE : INTEGER) is
  record
    POSITION : INTEGER;
    SPEICHER : STRING (1 .. LAENGE);
  end record;
```

- Variante Verbundtypen:

```
type LAMPENDATEN (LAMPE : LAMPEN) is
  record
    WATT_ZAHL : INTEGER;
    case LAMPE is
      when GLUEHBIRNE =>
        FASSUNG : INTEGER;
      when LEUCHTROEHRE =>
        LAENGE : REELL;
    end case;
  end record;
```

Attribute

Für alle Typen sind sogenannte Attribute definiert, die spezifische Eigenschaften von Objekten des jeweiligen Typs bezeichnen. So liefert z. B. die Anwendung des Attributs FIRST das erste, die von LAST das letzte Element eines diskreten Typs. SIZE ermittelt die Größe des vom Objekt belegten Speicherplatzes und ADDRESS liefert seine Speicherplatzadresse. Für Felder liefert RANGE den Indexbereich.

```
INTEGER'LAST
LAMPENDATEN'ADDRESS
VEKTOR'RANGE
```

Konstanten

Typen können außer zur Deklaration von Variablen auch für die Definition von Konstanten verwendet werden, wobei auch Verbund- und Feldkonstanten vereinbart werden können.

```
GRENZE : constant INTEGER := 100;
MELDUNG : constant STRING := "HALLO!";
GEBURTSTAG : constant DATUM := (19, 3, 1961);
```

Kontrollstrukturen

Ada verfügt über alle typischen Kontrollstrukturen ALGOL-artiger Programmiersprachen.

- Zuweisungen:

```
ZAEHLER := ZAEHLER + 1;
```

- Blöcke:

Blockanweisungen erlauben eine nicht an sonstige Programmstrukturen gebundene Kontrolle des Gültigkeitsbereichs von Variablen, indem sie lokale Variable mit den auf ihnen arbeitenden Anweisungen zusammenfassen.

```
TAUSCHE:
declare
  HILF : INTEGER;
begin
  HILF := A; A := B; B := HILF;
end TAUSCHE;
```

- Alternativen:

```
if ZAHL > 0 then
  SIGNUM := 1
elsif ZAHL < 0 then
  SIGNUM := -1
else
  SIGNUM := 0
end if;
```

Mit Hilfe des **elsif**-Konstrukts wird die aus Pascal bekannte Kaskadierung von **if**-Anweisungen vermieden.

- Fallunterscheidungen:

```
case HEUTE is
  when MONTAG =>
    ERRECHNE_INITIALSTAND;
  when FREITAG =>
    ERRECHNE_ENDSTAND;
  when DIENSTAG .. DONNERSTAG =>
    ERRECHNE_TAGESSTAND;
  when others =>
    ÜBERNEHME_TAGESSTAND;
end case;
```

- Schleifen:

Alle klassischen Schleifenformen finden sich in Ada wieder. Schleifen können mit einer **exit**-Anweisung an einer beliebigen Stelle verlassen werden.

```
for I in 1 .. 10 loop
  ZAHL := ZAHL + I;
end loop;

I := 10;
while I > 0 loop
  ZAHL := ZAHL + I;
  I := I - 1;
end loop;

I := 10;
loop
  ZAHL := ZAHL + I;
  exit when I = 1;
  I := I - 1;
end loop;
```

- Sprünge:

```
goto << ENDE >>;
```

Ada schränkt die Möglichkeit von Sprüngen stark ein: Sprünge sind nur innerhalb einer begrenzten Umgebung möglich und können nur aus Anweisungen heraus, nie in sie hinein führen. Selbst auf diese eingeschränkten Sprünge sollte man zugunsten einer übersichtlicheren Programmstruktur verzichten.

- Verlassen von Unterprogrammen:

```
return;
```

Kommentare in den mit diesen Anweisungen notierten Algorithmen und im Rest eines Ada-Programms stehen immer am Ende einer Zeile und werden mit dem Symbol „--“ eingeleitet.

Unterprogramme

Unterprogramme lassen sich als *Prozeduren* (ohne Ergebniswert) und *Funktionen* (mit Ergebniswert) vereinbaren. Prozeduren tragen gemäß der Festlegung im Prozedurkopf Wertparameter (**in** oder keine Angabe), Ergebnisparameter (**out**) oder Durchgangsparameter (**in out**).

```
procedure INKREMENT
  (ZAHL : in out POSITIV; DIFF : POSITIV := 1) is
begin
  ZAHL := ZAHL + DIFF;
end INKREMENT;
```

Der Aufruf einer Prozedur geschieht in der üblichen Weise durch Angabe ihres Namens und einer Liste aktueller Parameter. Der Typ eines jeden aktuellen Parameters muß mit dem Typ des entsprechenden formalen Parameters im Prozedurkopf übereinstimmen. Die Zuordnung aktueller Parameter zu formalen kann entweder anhand ihrer Stellung, oder aber durch eine explizite Angabe der Parameternamen erfolgen. Ferner ist es möglich, beim Aufruf Parameter wegzulassen, falls in deren Deklaration Standardwerte angegeben wurden (wie oben für DIFF der Standardwert 1).

```
INKREMENT (A, 5);
INKREMENT (DIFF => 5, ZAHL => A);
INKREMENT (A);
```

Funktionen liefern Werte eines beliebigen Typs, insbesondere auch Werte zusammengesetzter Typen. Sie können nur Wertparameter tragen, um zu verhindern, daß sie andere Außenwirkungen zeigen, als das Liefern ihres Ergebnisses (Seiteneffekte).

```
function ADD_VEKTOR (VEK1, VEK2: VEKTOR)
  return VEKTOR is
  HILF : VEKTOR;
begin
  for I in VEKTOR'RANGE loop
    HILF (I) := VEK1 (I) + VEK2 (I);
  end loop;
  return HILF;
end ADD_VEKTOR;
```

Aufruf und Ergebniszuweisung einer Funktion haben die Form

```
V3 := ADD_VEKTOR (V1, V2);
```

Statt mit einem Namen können Funktionen auch durch ein Operatorsymbol bezeichnet werden. Ihr Aufruf läßt sich dann in gewohnter Infix-Notation notieren. Wäre die obige Funktion als

```
function "+" (VEK1, VEK2 : VEKTOR)
  return VEKTOR;
```

vereinbart worden, ließe sie sich aufrufen durch

```
V3 := V1 + V2;
```

Vor allem durch die Definition neuer Operatoren entstehen verschiedene Unterprogramme, die zwar gleich bezeichnet werden, jedoch unterschiedliche Parameterlisten besitzen. Ada erlaubt dieses sogenannte *Überladen* von Unterprogrammnamen und stellt bei Namenskonflikten zur Übersetzungszeit anhand des Typs der aktuellen Parameterwerte fest, welches Unterprogramm aufgerufen wurde.

```
procedure SCHREIBE (X : INTEGER;
  STELLEN : INTEGER);
procedure SCHREIBE (X : STRING);
procedure SCHREIBE (X : FLOAT;
  VORKOMMA, NACHKOMMA : INTEGER);
```

Pakete

Pakete sind das wichtigste Modularisierungshilfsmittel in Ada. Sie stellen eine Sammlung von Daten oder auch Unterprogrammen über eine klar definierte Schnittstelle zur Verfügung und verbergen alle Details der Realisierung ihrer Dienste nach außen.

Die Festlegung eines Pakets besteht aus zwei verschiedenen Programmteilen, die auch getrennt übersetzbar sind:

- In einer *Paketspezifikation* wird die Schnittstelle eines Pakets festgelegt.
- In einem *Paketrumpf* werden die Dienstleistungen eines Pakets implementiert.

Mit Paketen lassen sich in einem Ada-Programm folgende Arten von Modulen bilden:

- Ein *Datenmodul* sammelt häufig benötigte Informationen in Form von Konstanten oder Variablen.
- Ein *Funktionsmodul* faßt verschiedene Unterprogramme zusammen und stellt sie in Art einer Funktionsbibliothek zur Verfügung.
- Ein *abstraktes Datenstrukturmodul* verfügt sowohl über Daten als auch über Funktionen, weist also gegenüber dem Funktionsmodul zusätzlich einen modulinternen Speicher auf. Der direkte Zugriff auf diesen Speicher kann einem Benutzer verwehrt werden. Er kann dann nur über die ihm vom Modul zur Verfügung gestellten Prozeduren zugreifen.
- Ein *abstraktes Datentypmodul* exportiert Datentypen und Operationen, durch die auf Objekte der Datentypen zugegriffen werden kann. Die interne Struktur der Objekte kann vor dem Benutzer verborgen werden, er kann sie dann nur mit den Operationen des Moduls verändern.

Zur Realisierung eines Datenmoduls reicht allein eine Paketspezifikation aus:

```
package KESSEL is
  GRENZDRUCK : constant FLOAT := 55.6;
  MOMENTANDRUCK : FLOAT;
end KESSEL;
```

Für die anderen oben angeführten Modultypen ist zusätzlich ein Paketrumpf erforderlich, in dem die Unterprogramme des Pakets festgelegt werden. Als Beispiel eines abstrakten Datenstrukturmoduls sei die Steuerung eines Mischventils definiert, das zwei unterschiedlich temperierte Medien zusammenführt.

```
package MISCH_VENTIL is
  procedure KALT_ZUFLUSS
    (ABSOLUT : INTEGER);
  procedure WARM_ZUFLUSS
    (ABSOLUT : INTEGER);
end MISCH_VENTIL;
```

```
package body MISCH_VENTIL is
  KALT, WARM : INTEGER;
  procedure KALT_ZUFLUSS
    (ABSOLUT : INTEGER) is
    DIFFERENZ : INTEGER;
  begin
    DIFFERENZ := ABSOLUT-KALT;
    ...--Öffne bzw. schließe Ventil um Wert
       --von DIFFERENZ
    KALT := ABSOLUT;
  end KALT_ZUFLUSS;
  procedure WARM_ZUFLUSS
    (ABSOLUT : INTEGER) is
    ...
  end WARM_ZUFLUSS;
end MISCH_VENTIL;
```

Einem Benutzer des Pakets MISCH_VENTIL stehen nur die Prozeduren zur Einstellung der beiden Ventilwege zur Verfügung. Die Variablen KALT und WARM, die den jeweils aktuellen Wert der Ventilstellung speichern, bleiben ihm verborgen.

Zur Illustration der Realisierung eines abstrakten Datentypmoduls mit Ada sei nun vorausgesetzt, daß mehr als ein Ventil gesteuert werden soll. Jedes Ventil sei durch eine Reihe von Daten gekennzeichnet, unter anderem durch seine Steuerleitung und die Art des Ventils. Der Benutzer muß diese Daten zwar angeben, um ein bestimmtes Ventil anzusprechen, doch soll ihm der interne Aufbau der Ventilbeschreibung verborgen bleiben. Er könnte sonst unter Umgehung der Moduldienste die Ventildaten manipulieren und das korrekte Funktionieren der Steuerung gefährden.

Für den Fall, daß zwar der Typ einer Datenstruktur bekannt gemacht werden muß, aber deren interner Aufbau verborgen bleiben soll, sieht Ada sogenannte *private Typen* vor. Benutzer privater Typen können Variablen dieses Typs deklarieren und solche Variablen einander zuweisen. Zugriff auf die Variable haben sie aber nur über ihnen zur Verfügung gestellte Zugriffsoperationen.

```
package MISCH_VENTILE is
  type VENTIL is private;
  function INIT_VENTIL (VENTILTYP,
    STEUERLEITUNG : INTEGER)
    return VENTIL;
  procedure KALT_ZUFLUSS
    (V : in out VENTIL;
    ABSOLUT : INTEGER);
  procedure
WARM_ZUFLUSS
    (V : in out VENTIL; ABSOLUT : INTEGER);
  private
    type VENTIL is
      record
        VENTILTYP : INTEGER;
        STEUERLEITUNG : INTEGER;
        VENTILSTELLUNG : INTEGER;
      end record;
end MISCH_VENTILE;
```

Ein Benutzer kann nun z. B. eine Variable VENT des privaten Typs VENTIL deklarieren, initialisieren und verwenden.

```
with MISCH_VENTILE; use MISCH_VENTILE;
VENT : VENTIL := INIT_VENTIL (42, 4711);
KALT_ZUFLUSS (VENT, 50);
WARM_ZUFLUSS (VENT, 100);
```

Um die von einem Paket angebotenen Dienste in Anspruch zu nehmen, kann man sie entweder ausführlich in der Form

```
MISCH_VENTILE.INIT_VENTIL
```

benennen oder ihre Namen zuvor mit Hilfe einer **use**-Klausel wie oben direkt in den augenblicklichen Gültigkeitsbereich übernehmen. Die vor der **use**-Klausel notierte **with**-Klausel gibt dabei an, daß das folgende Programmteil direkten Gebrauch vom Paket MISCH_VENTILE macht und demzufolge dieses Paket zur Übersetzung des Programmteils benötigt wird.

Generische Programmeinheiten

Generische Programmeinheiten erlauben die Formulierung von Algorithmen unabhängig von der Struktur der mit ihnen manipulierten Datenobjekte. Sie geben die Schablone eines Verfahrens vor, aus der konkrete Pakete oder Prozeduren für diese Daten erzeugt werden, sobald die Struktur der zu bearbeitenden Daten feststeht. Mit ihnen läßt sich z. B. eine gemeinsame Schablone für das Sortieren von Zahlen und Zeichenketten angeben.

Auch die Festlegung einer generischen Programmeinheit besteht aus einer Spezifikation und einem Rumpf. Dies zeigt das folgende Beispiel eines Vertauschungsalgorithmus, der ohne Kenntnis der ihm zugrundeliegenden Datentypen formuliert werden kann.

```
generic
   type ELEM is private;
   procedure TAUSCHE (in out A, B: ELEM);
procedure TAUSCHE (in out A, B : ELEM) is
   HILF : ELEM;
begin
   HILF := A; A := B; B := HILF;
end;
```

Von der generischen Prozedur TAUSCHE können nun mehrere Exemplare angefertigt werden, die jeweils auf einen speziellen Datentyp anwendbar sind.

```
procedure TAUSCHE_INT is
   new TAUSCHE (INTEGER);
procedure TAUSCHE_FLOAT is
   new TAUSCHE (FLOAT);
```

Die Erzeugung solcher Exemplare geschieht in Ada immer zur Übersetzungszeit, nie zur Laufzeit. Nur so kann eine vollständige Typenprüfung zur Übersetzungszeit garantiert werden.

Prozesse

Prozesse stellen Programmeinheiten dar, die nach ihrer Aktivierung nebenläufig zu anderen Algorithmen innerhalb des Programmsystems bearbeitet werden können. Ihre Vereinbarung besteht wie bei Paketen aus der Festlegung ihrer Schnittstelle in Form einer *Prozeßspezifikation* und der Angabe der vom Prozeß abzuarbeitenden Anweisungen in Form eines *Prozeßrumpfes*.

Zur Veranschaulichung des Prozeßkonzepts in Ada betrachten wir die Steuerung eines Mischventils, dessen Eingänge starken Druckschwankungen unterworfen sind. Die Aufgabe der Steuerung bestehe darin, diese Schwankungen kontinuierlich etwa alle 0,1 Sekunden auszugleichen. Die Spezifikation eines solchen Prozesses lautet dann:

```
task VENTILREGELUNG is
   entry SOLL_KALT (SOLL : INTEGER);
   entry SOLL_WARM (SOLL : INTEGER);
end VENTILREGELUNG;
```

SOLL_KALT und SOLL_WARM stellen nach außen sichtbare *Eingangspunkte* des Prozesses dar. Aus der Sicht anderer Prozesse, die z. B. mit dem Prozeß VENTILREGELUNG in Verbindung treten wollen, um einen neuen Sollwert einzustellen, hat die Kontaktaufnahme die Form eines Prozeduraufrufs.

```
VENTILREGELUNG.SOLL_KALT (42)
```

Derartige Prozeduren können alle Arten von Parametern tragen. So gerät der aufrufende Prozeß in die Rolle eines Dienstleistungsnehmers, der Wertparameter bereitstellt, der andere Prozeß in die des Diensterbringers, der Ergebnisparameter liefert.

Im Prozeßrumpf des Diensterbringers wird die Verarbeitung einer solchen Prozedur bestimmt.

```
with MISCH_VENTIL; use MISCHVENTIL;
task body VENTILREGELUNG is
   KALT, WARM : INTEGER;
begin
   loop
      select
         accept SOLL_KALT (SOLL : INTEGER)
         do
            KALT := SOLL;
         end SOLL_KALT;
      or
         accept SOLL_WARM (SOLL : INTEGER)
         do
            WARM := SOLL;
         end SOLL_WARM;
      or
         delay 0.1;
      end select;
      ... --Berechnung der Regelgröße
      KALT_ZUFLUSS (...);
      WARM_ZUFLUSS (...);
   end loop;
end VENTILREGELUNG;
```

Beim Aufruf des Eingangspunktes kommt die entsprechende **accept**-Anweisung des Partnerprozesses (hier

VENTILREGELUNG) zur Ausführung. Dies geschieht allerdings erst dann, wenn dieser an der **accept**-Anweisung angekommen ist. Dieses Zusammentreffen zweier Prozesse wird treffend als *Rendezvous* bezeichnet.

Im Vorfeld eines Rendezvous können zwei Fälle eintreten: Der Dienstleistungsnehmer ruft den entsprechenden Eingangspunkt auf, bevor der Diensterbringer dazu bereit ist, den Aufruf zu akzeptieren, oder aber der Erbringer erreicht die **accept**-Anweisung, bevor ein Aufruf vorliegt. In beiden Fällen wird der schnellere Prozeß so lange blockiert, bis der langsamere bereit ist. Während die **accept**-Anweisung vom Erbringer ausgeführt wird, bleibt der Aufrufer blockiert. Am Ende der **accept**-Anweisung können über Ergebnisparameter Daten an den Aufrufer zurückgereicht werden, danach arbeiten beide Prozesse unabhängig voneinander weiter.

Wenn ein Prozeß wie im obigen Beispiel mehrere Dienste zur Verfügung stellt, so können sie mit Hilfe einer **select**-Anweisung als gleichberechtigt erklärt werden. Trifft der Erbringer auf eine **select**-Anweisung und liegen mehrere verschiedene Aufrufe vor, so entscheidet er nichtdeterministisch über seinen Rendezvouspartner. Er kann dabei die Annahme eines Rendezvous zusätzlich auch von Vorbedingungen abhängig machen.

Wenn verhindert werden soll, daß ein Diensterbringer endlos auf Aufrufe wartet, während er statt dessen andere sinnvolle Aktivitäten ausführen könnte, können **accept**-Anweisungen auch zeitlich überwacht werden. Im obigen Beispiel wartet der Prozeß maximal 0,1 Sekunden auf einen Rendezvouspartner, bevor er die Programmausführung fortsetzt. So ist gewährleistet, daß er beim Ausbleiben neuer Sollwert-Forderungen spätestens nach 0,1 Sekunden das Ventil neu regeln kann. Ähnliche Zeitüberwachungen sind auch dem Dienstleistungsnehmer möglich. Sie beziehen sich jedoch immer nur auf den Zeitpunkt des Zustandekommens eines Rendezvous, nie auf dessen Abschluß.

Die Ada-Sprachbeschreibung macht keine Annahme darüber, ob die Prozesse auf einem Prozessor zeitlich verschachtelt oder auf verschiedenen Prozessen gleichzeitig ablaufen. Den einzelnen Prozessen sind zwar Prioritäten zugeordnet, der Algorithmus zur Zuteilung von Prozessoren kann jedoch nicht verändert werden.

Analog zur Definition von Datentypen können in Ada auch Prozeßtypen definiert werden, mit denen beispielsweise die Deklaration ganzer Felder algorithmisch gleichartiger Prozesse möglich ist.

Echtzeitabhängigkeit

Sprachmittel zur zeitlichen Einplanung von Prozessen (wie etwa in PEARL) gibt es in Ada nicht; die einzige Einflußmöglichkeit auf Prozesse von außen besteht darin, diese abzubrechen. Jeder Rechenprozeß kann sich selbst jedoch mit der bereits vorgestellten **delay**-Anweisung für eine gewisse Zeit blockieren. Auf diese Weise lassen sich die vom technischen Prozeß verlangten zeitlichen Abläufe erreichen.

In einem standardmäßig vorgegebenen Paket CALENDAR werden Datenstrukturen und Funktionen zur Behandlung von Zeitpunkten (Typ TIME) und Zeitabschnitten (Typ DURATION) bereitgestellt.

Ausnahmebehandlung

Unter Ausnahmebedingungen versteht man seltene Ereignisse, deren Auftreten im Programm dessen weitere normale Abarbeitung unmöglich oder unsinnig macht. Beispiele solcher Ausnahmen sind die Division durch Null oder der Zugriff auf ein nicht definiertes Feldelement. Ada sieht durch Mechanismen zur Ausnahmebehandlung vor, einen eingetretenen Fehler zu behandeln, um den Programmablauf geordnet weiterführen zu können.

Algorithmen zur Ausnahmebehandlung werden am Ende eines Blocks oder einer Programmeinheit notiert. Auf die vom System vorgesehene Ausnahmebedingung NUMERIC_ERROR für arithmetische Fehler oder Überläufe kann z. B. durch Ausgabe einer Fehlermeldung wie folgt reagiert werden:

```
procedure ADD_MUL (ZAHL : in out INTEGER) is
begin
   ZAHL := ZAHL + 1;
   ZAHL := ZAHL * 2;
exception
   when NUMERIC_ERROR =>
      PUT ("Zahlenbereich überschritten.");
      ZAHL := INTEGER'LAST;
end INKREMENT;
```

Tritt eine Ausnahme auf, so wird sofort die zugehörige Ausnahmebehandlungsroutine ausgeführt und der Block danach verlassen. Die Programmbearbeitung wird in dem Block fortgesetzt, in dem der von der Ausnahme betroffene Block aktiviert wurde. Ist in einem Block für eine aufgetretene Ausnahme keine Behandlungsroutine vorgesehen, wird die Bearbeitung ebenfalls abgebrochen und die Ausnahme in den umgebenden Block weitergereicht. Diese *Ausnahmefortpflanzung* setzt sich so lange fort, bis entweder eine passende Routine gefunden wird oder die Ebene des umgebenden Prozesses erreicht ist. Findet sich hier auch keine Ausnahmebehandlungsroutine, wird der Prozeß abgebrochen. Auf diese Art und Weise ist es möglich, sehr detailliert festzulegen, auf welcher Stufe welche Ausnahmen (mit unterschiedlicher Wirkung für das Gesamtprogramm) behandelt werden sollen.

Neben Ausnahmebedingungen, die vom Ada-System vorgegeben sind, kann ein Programmierer eigene Ausnahmebedingungen benennen. Die Vereinbarung benutzereigener Ausnahmen erlaubt die saubere Behandlung von Fehlerfällen mit den gleichen Sprachmitteln, wie sie zur Behandlung von Systemfehlern verwendet werden. Selbstdefinierte Ausnahmebedingungen sind jedoch insbesondere dann sinnvoll, wenn in einer Programmeinheit Fehler auftreten, die nicht direkt dort behoben werden können.

Während systemeigene Ausnahmebedingungen automatisch angezeigt werden, müssen benutzereigene Ausnahmen mit einer **raise**-Anweisung explizit ausgelöst werden.

Von der zuvor vereinbarten Mischventil-Steuerung kann so z. B. eine Blockierung des Ventils gemeldet werden, wenn die nach dem Einstellen des Ventils gelesene Ventilstellung nicht dem zuvor eingestellten Wert entspricht.

```
package MISCH_VENTIL is
   VENTIL_BLOCKIERT : exception;
   procedure KALT_ZUFLUSS
      (ABSOLUT : INTEGER);
   procedure WARM_ZUFLUSS
      (ABSOLUT : INTEGER);
end MISCH_VENTIL;

package body MISCH_VENTIL is
   procedure KALT_ZUFLUSS
      (ABSOLUT : INTEGER) is
   begin
      ... --Ventil stellen
      if LIES_STELLUNG /= ABSOLUT then
         raise VENTIL_BLOCKIERT;
      end if;
   end KALT_ZUFLUSS;
   ...
end MISCH_VENTIL;
```

In der Programmeinheit, in der die Prozeduren von MISCH_VENTIL benutzt werden, oder in einer sie umgebenden kann nun eine Ausnahmebehandlung für VENTIL_BLOCKIERT definiert werden, in der die für diesen Fall geeigneten Maßnahmen getroffen werden. Wird nirgendwo eine Behandlung definiert, so wird im Fehlerfall das Auftreten der Ausnahme vom Laufzeitsystem gemeldet und der Prozeß terminiert.

Ein-/Ausgabe-Verkehr und Peripherieansteuerung auf Maschinenebene

Für Ein- und Ausgabe-Operationen sind im Ada-Standard die generischen Pakete SEQUENTIAL_IO, DIRECT_IO und TEXT_IO spezifiziert. Sie stellen alle üblichen Prozeduren zur Bedienung von Geräten mit stromorientierter Ein-/Ausgabe und mit wahlfreiem Zugriff zur Verfügung. In einem separaten Paket IO_EXCEPTIONS sind alle Ausnahmen zusammengefaßt, die bei der Ein- und Ausgabe auftreten können. Da die Ein-/Ausgabe-Prozeduren nicht Bestandteil der Sprache selbst sind, kann sie der Benutzer leicht ändern, um beispielsweise spezielle Ein-/Ausgabe-Formate zu verarbeiten. Dabei können mit Hilfe des Pakets LOW_LEVEL_IO Geräteeinheiten auch direkt auf Registerebene angesprochen werden.

Mit Hilfe sogenannter *Repräsentationsklauseln* können Variablen auf bestimmte Speicheradressen gelegt und die Darstellungen ihrer Werte auf Speicherebene bestimmt werden. Gerade hierfür ist es hilfreich, daß Ada die Angabe von Zahlenwerten zu einer beliebigen Basis erlaubt, also insbesondere die Angabe von Oktal- und Hexadezimalzahlen.

```
type CODE is (MOVE, PUSH, POP);
for CODE use (MOVE => 16#7F#,
   PUSH => 16#8A#, POP => 16#8B#);
for CONTROLLER_STATUS use at 8#177650#;
for POSITIV use 2*BYTE;
```

Die Behandlung von Unterbrechungen durch Ein-/Ausgabe-Geräte kann direkt durch Ada-Prozesse vorgenommen werden. Dazu verbindet man einen Eingangspunkt des Prozesses mit dem entsprechenden Unterbrechungsvektor der Hardware.

```
task UNTERBRECHUNGSBEHANDLUNG is
   entry ZEICHEN_ENTNEHMEN;
   for ZEICHEN_ENTNEHMEN use at 16#80#;
end UNTERBRECHUNGSBEHANDLUNG;
```

Derartige Sprachelemente sind nicht für alle Rechnerarchitekturen sinnvoll und deshalb nicht in den zur Validierung notwendigen Testprogrammen des *AJPO* enthalten. Bedauerlicherweise führt dies dazu, daß sie oft auch auf Maschinen, für die sie eigentlich gedacht waren, nicht verfügbar sind.

Mit Hilfe der Dienste des Pakets MACHINE_CODE ist das direkte Einfügen von Objektcode in eine Anweisungsfolge möglich.

Werte, die von der Implementierung des Ada-Systems oder der Maschine, auf der es eingesetzt wird, abhängen, sind im Paket SYSTEM zusammengefaßt. Darin findet sich beispielsweise die Spezifikation der Wortbreite, der Speichergröße sowie der Wertebereiche von INTEGER-Zahlen und Prioritäten. Programmeinheiten, die von diesem Paket keinen Gebrauch machen, sind frei von Maschinenabhängigkeiten und sollten daher nach Neuübersetzung auf jedem Rechner laufen.

4. Schlußbemerkung

Vergleicht man die in Ada für die Lösung von Prozeßautomatisierungsaufgaben bereitgestellten Sprachelemente mit den in [13] für diesen Bereich als sinnvoll erachteten Mechanismen, kommt man zu dem Schluß, daß Ada den für dieses Anwendungsgebiet geltenden Anforderungen im großen und ganzen genügt. Die Sprache stellt zudem ein aus heutiger Sicht der Softwaretechnik unabdingbares Gerüst zur Strukturierung von Programm und Programmierung bereit, über das die wenigsten anderen gängigen Sprachen verfügen.

Dennoch sei nicht verhohlen, daß auch in Ada einige Sprachelemente nicht enthalten sind, die man sich als Programmierer für den Bereich der Prozeßautomatisierung wünschen würde. So gibt es keine Möglichkeiten zur Beschreibung von Konfigurationen eines verteilten Ada-Systems oder zur Festlegung harter Zeitschranken für Rechenprozesse. Eine Berücksichtigung dieser Punkte hätte allerdings eine noch weitere Erhöhung des ohnehin schon großen Umfangs der Sprache und ihrer Systeme zur Folge.

Schrifttum

[1] *United States of America, Department of Defense:* Steelman Requirements for High Order Programming Languages. 1978.

[2] *United States of America, Department of Defense:* Requirements for the Ada Programming Support Environment (Stoneman). 1980.

[3] *United States of America, Department of Defense:* Ada Programming Language. Military Standard ANSI/MIL-STD-1815A, American National Standards Institute, Washington (D.C.), 1983.

[4] *Bjørner, D.*, und *Oest, O.N.* (Ed.): Towards a Formal Description of Ada. Lecture Notes in Computer Science 98. Springer, Berlin-Heidelberg–New York, 1980.

[5] *Uhl, J.*, et al.: An Attribute Grammar for the Semantic Analysis of Ada. Lecture Notes in Computer Science 139. Springer, Berlin–Heidelberg–New York, 1982.

[6] *Sigada:* Ada Letters. ACM Special Interest Group on Ada, erscheint alle zwei Monate.

[7] *Barnes, J.G.P.:* Programmieren in Ada. Carl Hanser, München–Wien, 1983.

[8] *Goos, G., Persch, G.*, und *Uhl, J.:* Programmiermethodik mit Ada. Springer Compass. Springer, Berlin–Heidelberg–New York, 1987.

[9] *Habermann, A.N.*, und *Perry, D.E.:* Ada for Experienced Programmers. Addison-Wesley, Reading (MA.), 1983.

[10] *Ledgard, H.:* Ada – An Introduction. Springer, Berlin–Heidelberg–New York, 1983.

[11] *Nagl, M.:* Einführung in die Programmiersprache Ada. Vieweg, Braunschweig–Wiesbaden, 1982.

[12] *Hoare, C.A.R.:* The Emperor's New Clothes. Communications of the ACM, 24/2, 1981.

[13] *Herrtwich, R.G.:* Sprachmittel zur Programmierung verteilter Echtzeitsysteme. Automatisierungstechnische Praxis 30 (1988), H. 8, S. 388–396.

Dr.-Ing. R.G. Herrtwich und Dr.-Ing. A. Knoll, Technische Universität Berlin, Fachbereich Informatik, Fachgebiet Prozeßdatenverarbeitung und Robotik, Franklinstraße 28/29, D-1000 Berlin 10.

Die PEARL-Spalte

PEARL-Echtzeitsystem für den PC

Im Fachgebiet Prozeßautomatisierung der Universität-GH-Paderborn wird seit einigen Jahren die Implementierung von PEARL auf Rechnern mit Prozessoren der Familie INTEL 8086 betrieben, wobei sich in letzter Zeit die Aktivitäten ganz auf die PCs vom Typ XT oder AT (auch mit 80386) konzentrieren.

Der Ausgangspunkt für diese Arbeiten war eine PEARL-Implementierung, die *W. Gerth* auf einem älteren Prozeßrechner entwickelte und die er in dem schon weit verbreiteten RTOS-UH für die Prozessorfamilie MOTOROLA 68000 fortführte. Auch wenn die Benutzeroberfläche unseres Systems sich schon in vielen Punkten von RTOS-UH unterscheidet, sind maßgebliche Eigenschaften erhalten geblieben:

1. Das PEARL-System bildet eine in sich geschlossene Einheit, das heißt, es enthält alles, was man zum Entwickeln, Übersetzen, Austesten und Laufenlassen eines PEARL-Programmes benötigt. Die Hilfe eines fremden Betriebssystems wird nicht in Anspruch genommen.
2. Der Compiler ist in einer speziellen virtuellen Sprache geschrieben und erzeugt in einem Durchlauf den teilweise virtuellen Objectcode.
3. Der Sprachumfang des Compilers entspricht im wesentlichen Basic PEARL nach DIN 66253, wobei die bei RTOS-UH vorgenommenen Erweiterungen bei uns in gewissen Abständen nachvollzogen werden.

Die INTEL-Prozessoren verfügen über vergleichsweise wenige Register und wenig komfortable Maschineninstruktionen. Deshalb besteht der bei uns erzeugte Objektcode nur aus virtuellen Befehlen. Die so definierte virtuelle Sprache stellt gewissermaßen eine leistungsfähige Maschinensprache dar, die so angelegt ist, daß man ein PEARL-Programm möglichst einfach in diese Sprache übersetzen kann. Der virtuelle Code wird zum Laufzeitpunkt interpretiert. Zur Erhöhung der Leistungsfähigkeit des Systems wurde von Anfang an der Coprozessor 80X87 verwendet.

Das PEARL-System wird von DOS aus geladen und gestartet. Da das PEARL-System den gesamten RAM-Speicher benutzt, ist eine Rückkehr nach DOS nur durch Booten möglich. Ein Datenaustausch zwischen den beiden Betriebssystemwelten ist allerdings möglich, da das Ansprechen von Disk und Floppy in einem DOS-kompatiblen Format erfolgt, so daß auf der Disk die DOS-Partitionen auch durch das PEARL-System genutzt werden können und so ein Datenaustausch über Files möglich ist.

Ein gravierendes Problem bei der Implementierung von PEARL auf PCs stellt die große Vielfalt verfügbarer kompatibler Hardware dar. Das Problem ist deshalb so schlimm, weil unter Kompatibilität im PC-Bereich meist verstanden wird, daß das in ROMs abgelegte BIOS die gleichen Aufrufe kennt und ausführen kann. Nun ist dieses BIOS aber für Echtzeitanwendungen ungeeignet, so daß wir genauso wie andere Entwickler von Echtzeitbetriebssystemen das BIOS nicht benutzen und statt dessen direkt auf die Ports z.B. auf einer Videokarte zugreifen. Deshalb müßte eine tiefergehende Kompatibilität gegeben sein, die aber gerade im Videobereich durch unterschiedliche Standards wie Hercules, EGA, VGA usw. keineswegs vorhanden ist. Ähnliche Probleme gibt es im Bereich der Disk-Controller, was von uns durch ständige Anpassung an neue Hardware berücksichtigt werden muß.

Um die Anpassung an andere Peripheriegeräte, insbesondere an Prozeßperipheriegeräte wie analoge und digitale Ein- und Ausgabe zu erleichtern, wird von DOS aus nur ein Basisteil mit dem Betriebssystemkern einschließlich der Treiber für Bildschirm, Tastatur, Disk und Floppy geladen. Dieser Basisteil lädt dann weitere Betriebssystemteile nach, indem er ein vom Anwender modifizierbares Kommando-File (vergleichbar dem AUTOEXEC.BAT bei DOS) abarbeitet. Auf diese Weise können weitere Treiber nachgeladen werden, so daß z.B. an jede beliebige parallele und serielle Schnittstelle Geräte der Standardperipherie wie Drucker oder Plotter angeschlossen werden können. Für einige gängige Peripheriegeräte liegen die nachladbaren Module fertig vor. Das System ist aber so angelegt, daß der Anwender auch selbst in Assembler geschriebene Module entwickeln kann.

Das PEARL-System wurde bisher an einige Hochschulen und Industriebetriebe zur Erprobung gegeben. Eine kommerzielle Verbreitung des Systems ist uns nicht möglich; zu diesem Zweck suchen wir einen geeigneten Distributor.

B. Reißenweber

Dr.-Ing. B. Reißenweber, Universität-GH-Paderborn, Fachgebiet Prozeßautomatisierung, Pohlweg 47, D-4790 Paderborn.