PEARL Rundschau

Inhalt

Juli 1982 Band 3 Nr. 2

PEARL Rundschau

Juli 1982
Band 3
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 in 7000 Stuttgart 1, Seidenstraße 36

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 das Kopieren 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

Software-Verifikation war einige Zeit lang ein Schlagwort, das bei vielen, insbesondere bei Leuten, die für Qualitätssicherung verantwortlich sind, die vage Hoffnung weckte, man könne durch ein - wie immer geartetes - Hilfsmittel, eine Art 'black box', automatisch die Richtigkeit eines fertigen Programmes überprüfen. Besonders allerhand Veröffentlichungen über 'Programmbeweise' nährten solche Hoffnungen. Als man jedoch bemerkte, welch großen Aufwand besonders die Verfahren der letztgenannten Klasse zu ihrem Einsatz erforderten und wie schwierig ihre Handhabung war, wurde es eine Zeitlang wieder sehr still um all diese Techniken.

Man begann, mehr Wert auf die 'richtige Entwicklung' mit Hilfe von Software-Entwicklungswerkzeugen zu legen, d.h. man wollte Fehler gar nicht erst entstehen lassen, anstatt sie hinterher 'herauszuprüfen'. Techniken dieser Art wurde in der PEARL-Rundschau ja schon breiter Raum gewidmet.
Inzwischen hat sich aber eine etwas ausgewogenere Betrachtungsweise durchgesetzt. Um möglichst gute und damit auch zuverlässige Programme zu erhalten, ist eben eine Kombination aus mehreren Methoden notwendig: Zunächst muß ein Programm mit Hilfe von Spezifikations- und Entwicklungswerkzeugen so sauber und handwerklich einwandfrei erstellt werden, wie dies mit vertretbarem Aufwand möglich ist. Danach müssen mit Hilfe von Test- und Verifikationshilfsmitteln die trotz alledem unvermeidlichen Fehler weitgehend aufgespürt und auch die Funktionen des Programms gegen seine Spezifikationen abgeprüft werden. Natürlich wird der Aufwand, den man sowohl auf der Entwicklungs- als auch auf der Testseite zu treiben gewillt ist, von der Kritikalität des erzeugten Programms abhängen, d.h. davon, wie schwerwiegend die schädlichen Folgen sind, die man sich von einem eintretenden Fehler erwartet.
Eines darf man jedoch von keinem Hilfsmittel für Test und Verifikation erhoffen: eine Aussage darüber, ob das Programm 'richtig' im Sinne einer vernünftigen Problemlösung ist. Alles, was auch die besten und umfangreichsten Verifikationshilfsmittel je werden leisten können, ist, daß sie Hilfestellung bieten beim Vergleich des fertigen Programms mit der Spezifikation. Für deren Richtigkeit ist aber der Entwickler immer selbst verantwortlich!

Ist man sich über diese prinzipielle Einschränkung im Klaren, so können richtig eingesetzte Test- und Verifikationshilfsmittel eine große Hilfe bei der Arbeit sein, und auch gegenüber unsystematischem Testen eine Menge Geld bei der Programmentwicklung einsparen helfen. So betrachtet, sind auch Hilfsmittel schon sehr wertvoll, die auf völlig pragmatischer Basis arbeiten, d.h. eigentlich nichts weiter tun, als die Struktur eines vorliegenden Programms offenzulegen und transparent zu machen, so daß sie überprüfbar wird.

Interessant ist, daß es bisher nur für ganz wenige höhere Programmiersprachen Verifikationshilfsmittel gibt. PEARL gehört glücklicherweise zu diesen 'privilegierten' Sprachen und in Heft 6, Band 2, der PEARL-Rundschau wurde auch ein solcher Analysator vorgestellt.

Der Schwerpunkt dieses Heftes liegt daher mehr auf den prinzipiellen Gesichtspunkten von Softwarezuverlässigkeit, Validation und Verifikation. Besonderen Dank schulden wir Herrn Dr. Ehrenberger von der Gesellschaft für Reaktorsicherheit in Garching, der sich der Mühe unterzogen hat, als Gastredakteur die Beiträge dieses Heftes zu beschaffen und zusammenzustellen. In seinem Einführungsartikel zeigt er auch eine Reihe interessanter Gesichtspunkte auf, dahingehend, auf welche Weise man während des Programmentwicklungsprozesses Fehler machen und vermeiden kann.

Der Artikel von Dr. Puhr-Westerheide geht dann mehr auf die für den praktischen Einsatz von solchen Verfahren möglicherweise relevanten Gesichtspunkten und die von verschiedenen Hilfsmitteln unterstützten Verfahren ein.

Krzykacz geht auf einen speziellen Aspekt, nämlich das Testen, ein und stellt statistische Betrachtungen in den Vordergrund, auf Grund derer man Konfidenzaussagen über das Testergebnis machen kann.
Die Arbeit von Lucas bezieht sich auf einen speziellen, aber wichtigen Gesichtspunkt beim Montieren von Programmen aus Moduln - die Konsistenzprüfung. Man erkennt hieran auch die Überlappung der Gebiete der Verifikation und der Software-Entwicklung im engeren Sinne.

Die beiden letzten Artikel dieses Heftes gehören dann wieder mehr in die weitere Problematik um PEARL. Der Beitrag von Eichenauer ist der nachgeholte Abdruck eines Vortrags auf der PEARL-Tagung im Dezember 1981 und beschäftigt sich sehr eingehend mit den Unterschieden zwischen den Tasking-Mechanismen von PEARL und Ada.
Schließlich beschreibt das Papier von Dahll die Implementation von PEARL auf dem 'NORD'-Computer in Halden und eine interessante Anwendung.

Wir hoffen, daß wir mit diesem Heft einige Anregungen bezüglich der Brauchbarkeit der Verifikation für die Programmierpraxis geben konnten und bedanken uns nochmals ganz herzlich bei Herrn Dr. Ehrenberger für seinen Einsatz und die Mühe, die er sich mit diesem Heft gemacht hat.

Für die Schriftleitung
Mit freundlichen Grüßen

P. Elzer

Softwarezuverlässigkeit und Programmiersprache

W. Ehrenberger, Garching

Kurzfassung

Im Rahmen der Diskussion um die Vorzüge und Nachteile einzelner Rechnersprachen, der ja auch PEARL unterworfen ist, tritt mitunter die Frage auf, ob man eine bestimmte Sprache auch für sicherheitsrelevante Einsätze vorsehen solle oder könne. Der vorliegende Beitrag stellt allgemein zusammen, welche Wünsche an Programmiersprachen und zugehörige Hilfsmittel zu richten sind, um sie für Sicherheitsanwendungen besonders geeignet zu machen. Er stützt sich besonders auf die Arbeiten des TC7 des European Workshop on Industrial Computer Systems und der WG A3 des IEC SC45a. Forderungen an Sprache und Hilfsmittel betreffen insbesondere: Problemnähe, Förderung irrtumsvermeidender Konstruktionen, Fehlererkennung und -behandlung während des Laufs, weitgehendes Abprüfen möglicher Fehler während des Übersetzens und Bindens sowie Zuverlässigkeit der Hilfsmittel selbst.

Abstract

During the discussion on benefits and drawbacks of computer languages, which also affects PEARL, sometimes the question arises whether or not a particular language should be used for safety related purposes. This contribution compiles the demands on programming languages and related tools that qualify a language for safety applications. The contribution is based primarily on work done at TC7 of the European Workshop on Industrial Computer Systems and at WG A3 of IEC SC 45a. Demands on language and tools comprise in particular: orientation to problem, support of error limiting constructions, failure detection and handling online, extensive test of possible errors during compilation and linking time and reliability of the tools themselves.

1. Das Problem des Programmversagens

Mit dem Fortschritt auf dem Gebiet der Rechnerentwicklung im engeren Sinn sind auch Fortschritte auf dem Gebiet der Rechnereinsätze verbunden. Unter anderem dringen Rechner auch in sicherheitsrelevante Gebiete immer weiter vor, z.B. bei

Kernkraftwerken	in Begrenzungs- und Schutzsysteme
Eisenbahnanlagen	in Stellwerke und in die Geschwindigkeitssteuerung von Zügen
Chemische Fabriken	in die Regelung der Prozesse
Flugzeugen	in die Flugregelung
Medizinischen Anwendungen	in Infusionspumpen
PKWs	in Bremssysteme
der Haustechnik	in Aufzugsteuerungen und Brennersteuerungen.

Selbstverständlich werden hierbei beträchtliche Anforderungen an die Sicherheit und die Verfügbarkeit der Rechner und ihrer Programme gestellt. Die zunächst intuitiv erhobene Forderung nach vollkommener Freiheit von Versagen läßt man in der Regel nach kurzem Besinnen fallen; denn sie ist bei Systemen von nur einiger Komplexität praktisch nicht zu erfüllen.

So bleiben zumeist folgende Forderungen an die Programmzuverlässigkeit und ihren Nachweis bestehen:

a) Die Software darf nicht das unzuverlässigste Glied in der Kette der zum ordnungsgemäßen Funktionieren benötigten Einrichtungen und Handlungen sein.

b) Die durch Programmversagen verursachten Schäden müssen (weit) unter dem Wert bleiben, der zu erwarten gewesen wäre, wenn man

auf einen Rechnereinsatz verzichtet hätte.

c) Die zum Erreichen und zum Nachweis der geforderten Zuverlässigkeit entstehenden Kosten müssen (weit) unter dem aus dem Rechnereinsatz zu erwartenden Gewinn bleiben.

Der Forderung a) nachzukommen, erfordert eine Untersuchung der sonst mit dem Technischen Prozeß befaßten Geräte und Handlungen. Soweit Geräte betroffen sind, kann man sich auf eigene oder fremde, etwa in der Literatur niedergelegte, Betriebserfahrungen stützen. Das gleiche gilt auch bezüglich menschlicher Handlungen. In letzter Zeit in den USA durchgeführte Untersuchungen über menschliches Fehlverhalten [1] lassen vermuten, daß die menschliche Versagenswahrscheinlichkeit nur selten unter 10^{-4} pro Fall liegt. In vielen Fällen wird daher zu fordern sein, daß die Versagenswahrscheinlichkeit pro Anforderung an ein Programm diesen Wert nicht überschreiten darf.

Forderung b) verlangt eine Risikobetrachtung, basierend auf den Schadenshöhen, die mit Programmversagen verbunden sein können und den Häufigkeiten mit denen Anforderungen an Programme kommen, sowie den anzunehmenden Versagenswahrscheinlichkeiten. Üblicherweise werden, falls auch Menschenleben betroffen sein können, zwei Rechnungen aufgemacht, eine für Sachschäden und eine für Personenschäden.

In Anlehnung an [2] definieren wir das Risiko als:

$$R = \Sigma X_i \, h_i \, p_i \qquad (1)$$

Die Summe ist über alle Ereignisarten i zu nehmen, die während im Laufe des Programm-Lebens auftreten können, X steht für die Schadenshöhe, h für die Häufigkeit des Eintritts schadensauslösender Ereignisse und p ist die Wahrscheinlichkeit, daß ein solches Ereignis von der Software nicht abgefangen wird. Die einzelnen p_i müssen für die verschiedenen Programmfunktionen also unter bestimmten Grenzen bleiben. Von einer dieser Grenzen war bei der Betrachtung der menschlichen Versagenswahrscheinlichkeit schon die Rede. Im übrigen bestätigt (1), was man intuitiv weiß, nämlich daß es sich besonders lohnt, häufig angesprochene Funktionen, die hohe Versagenskosten nach sich ziehen können, sehr zuverlässig zu machen. Große Anstrengungen im Hinblick auf Funktionen mit kleinen h und X dagegen lohnen sich weniger.

Die Forderung c) nun gibt uns auf, den Aufwand für die Klein-Haltung der p_i oder für den Nachweis ihres Klein-Seins in wirtschaftlichen Grenzen zu halten. Hierzu dienen die Betrachtungen dieses Beitrags.

Obgleich wir von sicherheitsbezogenen Rechnereinsätzen ausgehen, gilt ein Großteil des Weiteren auch für "normale" industrielle Anwendungen von Programmen. Denn finanzielle Verluste, Sachschäden im weiteren Sinn, treten auch dann auf, wenn ein Programm in Zusammenarbeit mit irgend einem technischen Prozeß nicht ordnungsgemäß funktioniert.

2. Das Problem der Programmierfehler

Es liegt nahe, das Programmier-Fehler-Machen als eine allgemein menschliche Eigenschaft zu betrachten und zu versuchen, auf die Zahl der zu erwartenden Programmierfehler aus allgemeinen Eigenschaften zu schließen. Ein solcher Versuch ist von Halstead in [3] unternommen worden. Das Ergebnis seiner umfangreicheren Betrachtung lautet für $\hat{B}$, die Anzahl der zu erwartenden Fehler in einem Programm:

$$\hat{B} = \frac{(N_1 + N_2) \, ld \, (\eta_1 + \eta_2)}{3000} \qquad (2)$$

Hierbei bedeuten:

η_1 Anzahl der im Programm verwendeten Operatoren, jeder einmal gezählt
$\{\eta_1\}$ = {=,+,-,/, ..., GOTO A, GOTO B, ..., IF, BEGIN ... END, ...}

N_1 Gesamtzahl aller vorkommender Operatoren, jeder sooft gezählt, wie er auftritt

η_2 Anzahl der im Programm verwendeten Operanden, jeder nur einmal gezählt
$\{\eta_2\}$ = {Variable 1, Variable 2, ... Feld 1, Feld 2, ...}

N_2 Gesamtzahl aller vorkommenden Operanden, jeder sooft gezählt, wie er auftritt.

Die Zahl 3000 errechnet sich aus einer Betrachtung der Eigenschaften unseres Kurzzeitgedächtnisses und dem Niveau der Englischen Sprache.

Einen intuitiven Zugang zur Sinnhaftigkeit von (2)

kann man auf folgende Weise gewinnen: $\eta = \eta_1 + \eta_2$ ist der Umfang des Vokabulars, das beim Programmieren zur Verfügung steht. Während des Programmierens ist aus diesem Vokabular eine Auswahl zu treffen; und zwar ist in jedem einzelnen Programmierschritt auszuwählen, welche Operanden und Operatoren gerade zur Problemlösung benötigt werden. Die Anzahl der binären Auswahlschritte beträgt für jeden Auswahlvorgang ld η. $N = N_1 + N_2$ gibt die Zahl der notwendigen Auswahlvorgänge an. Eine nähere Betrachtung unter Einschluß des Nenners zeigt übrigens, daß die Basis des gewählten Logarithmus das Ergebnis nicht beeinflußt.

Es liegt auf der Hand, daß menschliches Tun, wie Programmieren durch eine so einfache Beziehung wie (2) kaum sehr zutreffend oder gar erschöpfend beschrieben werden kann; dies dürfte auch dann zutreffen, wenn es nur auf einen einzigen Parameter - seine Fehlerhaftigkeit - hin betrachtet wird. Demgemäß kommt auch eine Anwendung der Halstead'schen Theorie in [4] zu durchaus gemischten Ergebnissen: Die Theorie traf für 2 von 3 Versuchen einigermaßen zu, zeitigte aber beim dritten völlig abwegige Resultate.

Dennoch wollen wir auf (2) aufbauen und fragen, was wir zu tun haben, um Programmierfehler zu vermeiden. Die Antwort lautet klar: Man halte N_1 und N_2 klein! Eine Vergrößerung von η_1 oder η_2 schadet dagegen nur wenig. Mit anderen Worten heißt dies: Je besser die zu verwendende Sprache an das zu lösende Problem angepaßt ist, desto weniger Fehler sind zu erwarten. Eine Vergrößerung des Vokabulars der Sprache stört nur geringfügig, dagegen ist eine Verlängerung des Programms recht fehlerträchtig. Sieht man sich (2) im Hinblick auf den Gegensatz von Höherer Sprache und maschinennaher Sprache an, so wird man sofort die Höhere Sprache vorziehen, denn bei der maschinennahen Sprache haben alle 4 in (2) auf der rechten Seite stehenden Größen höhere Werte, als etwa in PEARL, falls man ein Problem von nur einigem Umfang zugrunde legt.

Wie schon angedeutet, dürfte das Fehler-Machen beim Programmieren nicht so einfach zu beschreiben sein, wie in (2). Tatsächlich ist je auch schon des längeren bekannt, daß noch eine Reihe weiterer Parameter entscheidend zu Buche schlagen, etwa

- wie das Programm entwickelt wird,
- ob Zwischentests gemacht werden,
- wie das Programmierteam organisiert ist,
- ob Restriktionen in der Verwendung der Sprachmittel beachtet werden,
- welche Hilfsmittel zur Verfügung stehen,
- usw.

Falls eine dem zu lösenden Problem sehr gut angepaßte Sprache zur Verfügung steht, verlieren natürlich manche der obigen Punkte an Bedeutung, doch dürfen sie im allgemeinen Fall nicht unbeachtet bleiben. Eine Zusammenstellung einschlägiger Aspekte befindet sich in [5] und [6].

3. Wünsche an die Programmiersprache

Nachdem wir im vorhergehenden Kapitel zu dem Ergebnis gekommen sind, daß Höhere Sprachen für Sicherheitsanwendungen und mithin für industrielle Anwendungen von Rechnern ganz allgemein vorzuziehen seien, ist weiter zu fragen, welche zusätzlichen Randbedingungen für eine Prozeßrechner- oder Sicherheitssprache zu beachten wären.

Zunächst haben wir den Gegensatz von Problemnähe und Verbreitung der Sprache zu beachten. Es liegt auf der Hand, daß man eine selten benutzte und darum kaum bekannte Sprache nur zögernd für einen verantwortungsvollen Einsatz heranziehen wird. Außer den im nächsten Kapitel zu diskutierenden Unzulänglichkeiten, die man bei ihrem Übersetzer und sonstigen Hilfsmitteln vermuten wird, spielt ihr Unbekannt-Sein bei Entwickler und Gutachter eine abmahnende Rolle. Andererseits aber muß eine Sprache in weiten Kreisen umso unbekannter sein, je problemnäher sie ist und lassen sich die gewünschten Funktionen nur in problemorientierten Sprachen knapp ausdrücken. Zwischen den beiden Gesichtspunkten wird ein Kompromiß gefunden werden müssen. Angesichts der zunehmenden Verbreitung von Rechnern und den damit verbundenen Kenntnissen, sowie moderner Techniken im Compilerbau, dürfte aber die Zukunft ganz klar bei den problemnahen Lösungen liegen; sie gestatten es, in (2) mit minimalen N_i und η_i auszukommen.

Bezüglich der einzelnen zu fordernden Sprachеigenschaften folgen wir den Darlegungen in [5] und [6].

- Syntax und Semantik der Sprache sollen vollständig und unzweideutig definiert sein.

Dies ist vor allem zur Transportabilität der Programme wichtig und für Nachweise ihrer Eigenschaften auf dem Quelltextniveau. Sonst soll eine

Sicherheitssprache natürlich ganz allgemein irrtumsanfällige Konstruktionen vermeiden helfen und leicht überblickbare Konstruktionen fördern. Im einzelnen:

Modularisierung

- Die Sprache soll eine Modularisierung unterstützen.
- Einsprünge in Module sollen nicht erlaubt sein, Aussprünge aus Modulen sollen nur an deren Ende führen und nicht an beliebige Programmstellen, vor allem nicht an Stellen, die in der Aufschreibung weiter vorne liegen. Ausnahme: Fehlerausgang.
- Module sollen in ihrer Länge auf 50 bis 100 ausführbare Anweisungen beschränkt sein.
- Ihre Parameterzahl soll auf etwa 5 beschränkt sein.

Die Gründe für diese Forderungen liegen in dem Bestreben nach Übersichtlichkeit im zu erstellenden Quellcode. Nach Möglichkeit sollen Module für sich allein vorab verifiziert werden. Ihr Ablauf soll nicht durch Sprünge hinein oder hinaus unübersichtlich gemacht werden können; ihre Länge soll so sein, daß gemäß (2) noch damit gerechnet werden darf, daß sie von vorne herein richtig sind; mit nur 5 Parametern sollen sie arbeiten, weil dies die Zahl von Einzelfakten ist, die unser Kurzzeitgedächtnis noch gut verarbeiten kann [3,7].

Prozeduren

Außer dem soeben Dargestellten gelten noch die Forderungen:

- Funktionen dürfen die Parameter, mit denen sie gerufen werden, nicht verändern.
- Bei Aufrufen soll es keine Mischung von Namens- und Werteparametern geben.
- Jede Prozedur soll überprüfen, ob die aufrufende Stelle zum Aufruf berechtigt war.
- Jede Prozedur soll feststellen, ob die Parameter, mit denen sie aufgerufen wird, zulässig sind.
- In den Aufrufen von Prozeduren sollen keine anderen Prozeduren oder Funktionen als Parameter auftreten dürfen.

Fragen der Zeit und Reihenfolge

- Zugriffe zu gemeinsamen Betriebsmitteln sollen synchronisiert werden.
- Während der Ausführung einer Anweisung sollen ihre Parameter generell nicht von außen veränderbar sein.
- Interrupts sollen während des Laufs vorzugebender sicherheitsrelevanter Programmteile verboten werden können.
- Die Zeit der höchsten Interruptverbotsdauer soll vorgebbar sein.
- Während des Programmablaufs soll die verbrauchte Rechenzeit leicht zu überwachen sein.

Diese Forderungen sollen es erleichtern, so zu programmieren, daß definierte Zugriffsverhältnisse zu gemeinsamen Resourcen vorliegen und daß der Zeitablauf eines Programmes stets "bewußt" bleibt. Beides erleichtert die Verifikation.

Sprünge, Schleifen und Verzweigungen

- Unbedingte Sprünge nach rückwärts sollen nicht zulässig sein.
- Es soll keine LABEL VARIABLEN geben; bei Alternativen (CASE) soll die Liste der Sprungziele von Anfang an vollständig sein.
- Schleifen sollen stets eine bekannte höchstmögliche Durchlaufzahl haben.

Im übrigen gilt das zu Sprüngen in und aus Modulen Ausgeführte.

Daten

- Es sollen keine impliziten Typkonversionen stattfinden.
- Art, Bereich und Genauigkeit jeder Veränderlichen soll konstant sein.
- Veränderliche und Felder sollen ausdrücklich vereinbart werden müssen, einschließlich ihrer Typen.
- Variablennamen sollten beliebig lang sein dürfen
- Namen sollen unterscheiden helfen, ob der Parameter
  - einen Eingangs-, Ausgangs- oder transienten Wert repräsentiert,
  - welcher Art er ist, etwa Feld, Veränderliche, Konstante
  - wo er gilt
  - ob er abgeleitet ist, eine Länge, ein Zähler oder dergleichen,
  - welche Bedeutung er ganz allgemein hat.
- Veränderliche Systemparameter sollen hervorgehoben werden.
- Zwischen lokalen und globalen Veränderlichen ist

zu unterscheiden.
- Für einen Datentyp soll es nur eine Adressierungstechnik geben.
- Feldlängen sollen nach Möglichkeit Konstante sein.
- Bei Aufrufen von Feldkomponenten sollen stets alle Felddimensionen vorkommen.
- Während des Programmlaufs soll automatisch überprüft werden, ob die Feldgrenzen eingehalten werden.
- Es soll keine Art von EQUIVALENCE geben.
- Es soll keine einfachen COMMON's geben.
- Konstante und Veränderliche sollen voneinander getrennt abgespeichert werden.
- Nicht berechtigtes Beschreiben soll automatisch registriert und verhindert werden.

Diese Forderungen erklären sich fast alle selbst aus den übergeordneten Verlangen nach leicht verständlichen und nachvollziehbaren Programmen. Das getrennte Abspeichern von Variablen und Konstanten soll eine Überprüfung der Konstanten auf Unveränderheit erleichtern.

Aufschreibung

- Leichte Lesbarkeit des Programms soll den Vorzug vor leichter Aufschreibbarkeit haben.
- Eine Art der Aufschreibung soll stets nur eine einzige Bedeutung haben und umgekehrt.
- Es soll eine standardisierte Folge oder feste Plätze im Programm geben für
  - nichtausführbaren Code/ausführbaren Code
  - Deklarationen
  - Initialisierungen
  - Formate
  - Kommentare

Fehlerbehandlung während des Programmlaufs

- Die Sprache sollte eine Fehlerbehandlung während des Programmlaufs vorsehen. Auf Fehlerbehandlungen sollten automatisch führen:
  - Überschreitung von Feldgrenzen
  - Überschreitung von Wertebereichen
  - Zugriff zu nicht vorbesetzten Veränderlichen
  - Abschneiden signifikanter Stellen von Zahlenwerten
  - Übergabe von Parametern falschen Typs.
- Auch die übrigen im früheren Text verlangten Prüfungen sollen bei negativem Ausgang auf Fehlerbehandlungen führen.
- Die Sprache sollte "Assertions" vorsehen und zulassen; auch das Nicht-Erfüllen einer Assertion soll auf eine Fehlerhandlung führen.
- Die Fehlerbehandlungen selbst sollen vom Anwenderprogramm durchgeführt werden.

Die aufgestellten Forderungen entspringen im wesentlichen dem Wunsch nach Einbringung von Redundanz in das Programm in Form von Plausibilitätsprüfungen während des Laufs. Obgleich die unter dem ersten Spiegelstrich aufgelisteten Punkte im allgemeinen keine sinnvolle Programmfortsetzung mehr zulassen, sollte doch dem Anwender bei Entscheidung über das weitere Verfahren überlassen bleiben, da nur er über eine schrittweise Verringerung der Programmfunktionen entscheiden kann.

Es ist klar, daß die obigen Forderungen nicht für alle Prozeßrechneranwendungen sinnvoll sind. Teilweise sind sie auch nur mit großen Aufwand in einer Sprache zu verwirklichen, teilweise wird man sich damit zufrieden geben können, sie in Programmierrichtlinien niederzulegen.

4. Wünsche an Hilfsmittel

In ähnlicher Weise, wie man vom Standpunkt der Programmverifikation aus Wünsche an die zu verwendende Sprache äußern kann, lassen sich auch Wünsche an die Hilfsmittel, die mit dieser Sprache zusammen angeboten werden, formulieren. Nachdem in dieser Nummer noch ein Beitrag über Testhilfsmittel folgt, können wir uns hier beschränken auf die Betrachtung von

Übersetzern
Bindern
Ladern
Bibliotheksprogrammen
Laufzeitsystemen und
Betriebssystemen.

Die allgemein zu erhebenden Forderungen lauten selbstverständlich

- Die Hilfsmittel sollen in ihren Funktionen gut und vollständig beschrieben sein.
- Ihre Zuverlässigkeit soll anhand einer ausreichenden Betriebserfahrung nachgewiesen sein.
- Die o.g. Forderungen an die Sprache oder den abzusetzenden Code sind bestmöglich zu unterstützen.

Forderungen, die sich insbesondere an den zu ver-

wendenden Übersetzer richten, lauten:

- Während des Übersetzungsvorgangs oder danach sollen auf Wunsch Zwischenergebnisse ausgegeben werden, von denen Einzelheiten über den erzeugten Maschinencode entnommen werden können. Diese Zwischenergebnisse sollen gut lesbar sein.

Es mag sinnvoll sein, in manchen Einzelfällen das Compilationsergebnis selbst der Verifikation zugrunde zu legen. Weiter gilt:

- Während der Übersetzungszeit sollen möglichst viele Prüfungen des Quelltexts durchgeführt werden.
- Von den insgesamt vorzunehmenden Prüfungen soll ein möglichst großer Teil während des Übersetzungsvorgangs durchgeführt werden und nur das dort unbedingt Notwendige zur Laufzeit des Programms. Dies betrifft z.B. Parameter-Typ-Prüfungen.
- Falls Prüfungen während der Übersetzung und des Bindevorgangs auf Fehler stoßen, sollen diese Fehler nur gemeldet werden; Korrekturversuche sollen unterbleiben.
- Falls unklar bleibt, ob irgendwelche Regeln verletzt wurden, soll eine Warnung gegeben werden.
- Während der Übersetzungszeit soll insbesondere überprüft werden,
  - der Wertebereich Veränderlicher
  - ihr Typ
  - ihre Genauigkeit
  - die Zulässigkeit von Zuweisungen und
  - alle syntaktischen Konstrukte.

Über einen Binder, der der aufgeführten Forderung nach weitestgehender Prüfung der zusammenzusetzenden Module während des Bindevorgangs nahekommt, wird an anderer Stelle dieses Hefts berichtet.

Der zweite der mit Spiegelstrich versehenen Punkte wirft die Frage auf, wann ein Hilfsmittel, etwa ein Übersetzer, als hinreichend ausgetestet angenommen werden dürfe. Bild 1 zeigt, wie die Zusammhänge zwischen der Zahl der Einzelfunktionen eines solchen Programms, der Zahl der mit ihnen durchgeführten Probeläufe und der Wahrscheinlichkeit eines vollständigen Tests liegen. Hat ein Übersetzer z.B. 5000 Eigenschaften und werden 100000 mal Eigenschaften von ihm angesprochen, so beträgt die Wahrscheinlichkeit, daß eine seiner 5000 Eigenschaften überhaupt nie angesprochen wurde 10^{-5} . Hierbei ist angenommen, daß bei jeder Benutzung alle Eigenschaften mit der gleichen Wahrscheinlichkeit zum Zuge kommen. Da diese

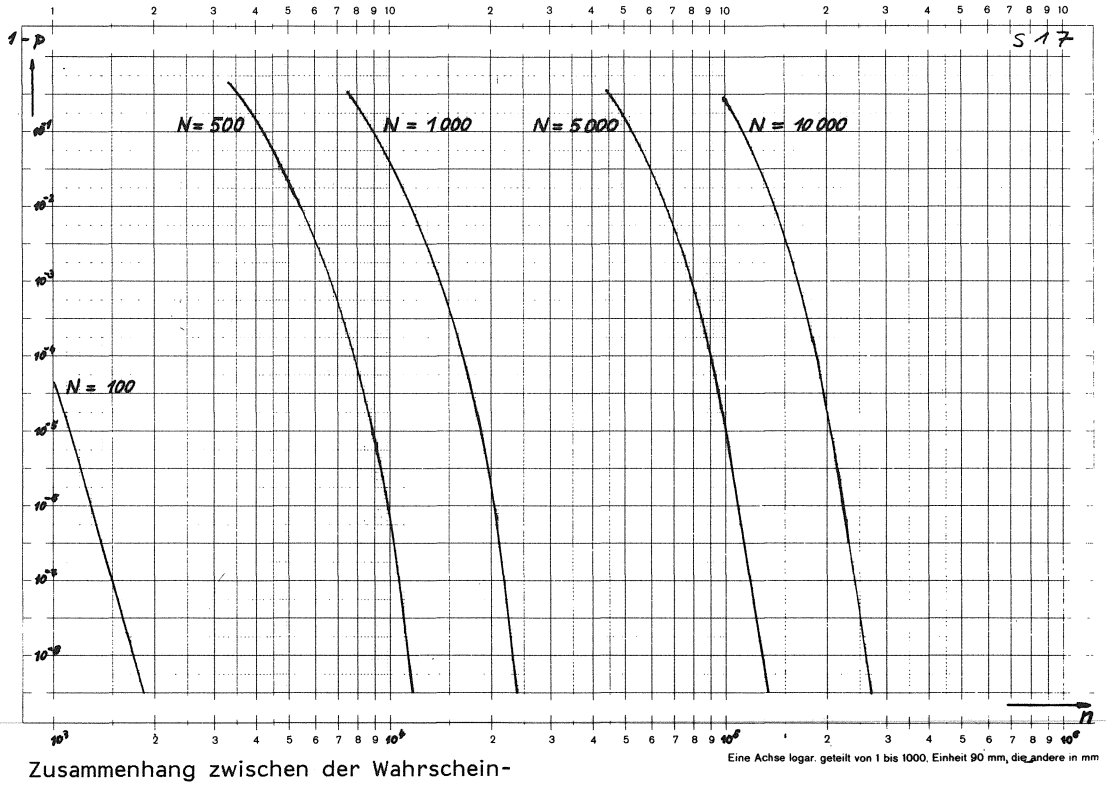


Bild 1: Zusammenhang zwischen der Wahrscheinlichkeit p, daß jede von N Einzelfunktionen eines Programms mindestens ein Mal getestet wird und der Anzahl n der Testläufe; Annahme: Jede der N Einzelfunktionen wird mit gleicher Wahrscheinlichkeit angesprochen. Aus [9].

Vorausetzung bei tatsächlichen Einsätzen nicht erfüllt werden kann, wäre als die Anzahl des Ansprechens von Eigenschaften die Zahl zugrundezulegen, die auch für die am seltensten benutzten Eigenschaften angenommen werden darf. Bei einem Übersetzer könnte man z.B. davon ausgehen, daß seine selten benutzten Teile im Mittel allenfalls ein Mal pro Lauf zum Zuge kommen und daß von diesen Teilen nur eines pro Lauf benutzt wird. Im obigen Beispiel würde das heißen, daß, um eine Unwahrscheinlichkeit des Getestet-Seins von nur $1 - 10^{-5}$ nachzuweisen, 100000 Übersetzungen durchgeführt werden müssen. Ist der Übersetzer etwa an 100 Anlagen installiert, wird er pro Arbeitstag im Mittel je 2 mal unabhängig von anderen Benutzungen zum Laufen gebracht, so kann nach 500 Arbeitstagen davon ausgegangen werden, daß er im o.g. Sinn getestet worden ist.

Für sicherheitsrelevante Einsätze folgt aus dieser Betrachtung natürlich, daß man bei neu auf dem Markt befindlichen Produkten vor allem von deren allgemein benützten Eigenschaften Gebrauch machen soll und nicht von ihren Spezialitäten.

5. Zusammenfassung

Es sind eine Reihe von Forderungen an Sprache und Übersetzer dargestellt worden, die beide für Sicherheitsanwendungen besonders geeignet erscheinen lassen. Die Sprache soll vor allem Programmierfehler-begrenzende Konstruktionen anbieten und Redundanz zum Auffinden von Programmversagen während des Laufs zur Verfügung stellen. Der Übersetzer und die sonstigen Hilfsmittel zur Programmerstellung sollen während ihres Einsatzes Hinweise auf mögliche Programmierfehler geben und in sich gut ausgetestet sein.

6. Schrifttum

[1] S w a i n, A.D.:
Handout Material for the Seminar on Effects of Human Performance on Nuclear Power Plant Operations
Garching, October 1980

[2] Gesellschaft für Reaktorsicherheit: Deutsche Risikostudie Kernkraftwerke, Hauptband (1980), Verlag TÜV Rheinland

[3] H a l s t e a d, M.H.: Elements of Software Science, North Holland, New York (1978)

[4] M e a l s, R.R and G u s t a f s o n, D.A.: An Experiment in the Implementation and Application of Halstead's and Mc Cabe's Measures of Complexity, Software Engineering Standards Application Workshop (August 1981), San Francisco, IEEE Cat. No. 81CH1633-7

[5] European Workshop on Industrial Computer Systems, TC7 Systems Reliability, Safety and Security: Development of Safety Related Software (October 1981), position paper

[6] International Electrotechnical Commission SC 45 A/WG-A3 (WG Experts) 5, Draft: Software for Computers in the Safety System of Nuclear Power Stations (August 1981)

[7] B e r n h o l t z, A.: Verbal presentation at IFIP Congress 1971

[8] B a s t l, W.; B e r a h a, D.; E h r e n b e r g e r, W.; F e l k e l, L.; H ö l d, A.: Developments in the Field of Man/Machine Communication, IAEA-CN-39/37, "Current Nuclear Power Plant Safety Issues" Vol. III

[9] E h r e n b e r g e r, W.; P l ö g e r t, K.: Einsatz statistischer Methoden zur Gewinnung von Zuverlässigkeitskenngrößen von Prozeßrechner-Programmen, Bericht KfK-PDV 151 (Juni 1978).

Anschrift des Verfassers:

Gesellschaft für Reaktorsicherheit
Forschungsgelände

8046 Garching

Aspekte von Software-Validationsverfahren und ihrer Automatisierbarkeit

P. Puhr-Westerheide, Garching

Zusammenfassung

Der vorliegende Aufsatz befaßt sich mit einem Überblick über einige Verfahren und Werkzeuge, die zur Validation von Software eingesetzt werden. Anhand einer Veröffentlichung des NBS/U.S. Department of Commerce wird zunächst ein Überblick über gängige Validationsverfahren gegeben, die in 96 Validationswerkzeugen praktisch realisiert wurden. Die Kurzbeschreibungen dieser Werkzeuge sind in einer Datenbank des NBS/U.S. Department of Commerce abgelegt und in [1] wiedergegeben. Die verwendeten Verfahren sind von unterschiedlicher Komplexität; ihre Bandbreite reicht von einfachen Datenhaltungsverfahren zur Dokumentation bis zum anspruchsvollen Programmbeweis.

Der letzte Teil des Aufsatzes befaßt sich mit Problemen, die die Automatisierung dieser Verfahren aufwirft, und schildert einige Gesichtspunkte zu ihrer Bewertung.

Abstract

The presented paper deals with an overview on some methods and tools which are in use for the validation of software. By means of a publication of the NBS/U.S. Department of Commerce a survey on current valiation methods is given which are implemented in 96 validation tools. Th abstracts of these tools are stored in a database of the NBS/U.S. Department of Commerce, and are listed in [1]. The utilized methods are spread over a wide range of complexity, starting at simple database systems for documentation purposes up to pretentious program proofs. The latter part of the paper deals with problems arising with the automation of these methods, and outlines some viewpoints on their qualification.

Einleitung

Die wachsende Komplexität und der steigende Umfang von Software, die sich rasch vergrößernde Anzahl von Software-Projekten und die anwachsende Zahl von Menschen, die an solchen Projekten mitarbeiten, hat in der jüngeren Vergangenheit die Entwicklung von Verfahren zur Bewältigung von Software-Problemen gefördert. Die Zeiten, in denen hochqualifizierte Fachleute im Alleingang Codes geschrieben haben und danach Codefehler beseitigt haben, scheinen dem Ende zuzugehen. Anstelle dieser Vorgehensweise treten zunehmend Verfahren, die eine systematische Behandlung von Entwurf, Codierung, Validierung und Dokumentation von Software bewirken. Wegen der Vielschichtigkeit von Software-Problemen scheint es jedoch keine einfache und universelle Vorgehensweise zu geben, mit der alle anstehenden Probleme lösbar wären.

Deshalb ist die Anzahl unterschiedlicher Verfahren groß, entsprechend den verschiedenen Blickwinkeln, unter denen Software-Probleme betrachtet werden können.

Der Anteil der ursprünglich vorhandenen Anstrengungen zur Software-Validation hat sich zu Gunsten von Entwurfsmethoden verringert, gemäß der Erkenntnis, daß ungemachte Fehler nicht beseitigt werden müssen. Auf die Validation kann jedoch schon deshalb nicht verzichtet werden, weil nur mit ihr gewisse Entwurfsfehler entdeckt werden können. Darüberhinaus zwingt der Einsatz von Software an Orten, an denen sich eine Gefährdung von Menschen oder Sachwerten aus ihrem Fehlverhalten ergeben kann, zu ihrer sorgfältigen Validation.

Es hat sich gezeigt, daß gewisse Maßnahmen zur Fehlererkennung, -beseitigung und Validation stets vorgenommen werden müssen. Jedes moderne Übersetzungssystem liefert beispielsweise eine Anzahl von Kreuz-Referenz-Listen, die Aufschluß über die Verwendung von Variablen, externals, o.ä. geben.

Die existierenden Validations-Werkzeuge sind Programme, die automatisch gewisse Untersuchungen an Software vornehmen, deren Durchführung per Hand zu zeitraubend und fehleranfällig wäre. Darüberhinaus gibt es noch eine Reihe von Validationsverfahren, die bisher nicht automatisiert werden konnten.

In der Praxis verwendete Verfahren

Das NBS (National Bureau of Standards/U.S. Department of Commerce) hat Ende 1980 eine Sammlung von Kurzbeschreibungen von 257 Software-Werkzeugen herausgegeben [1]. Der Bericht beinhaltet den Abzug einer Datenbank, in dem die Werkzeug-Kurzbeschreibungen abgelegt sind, sowie einige Querverweis-Listen die die Klassifizierung der Werkzeuge, verwendete Verfahren, Literaturverweise, die verwendete Sprache, die Zielsprache der zu untersuchenden Software, u.a. enthält. Die Verfasser des NBS-Berichts betonen ausdrücklich, keine Wertung der Werkzeuge vorzunehmen, und rufen dazu auf, weitere Werkzeuge bekannt zu machen, behalten sich aber das Recht der Ablehnung vor, ohne auf die von ihnen benutzten Bewertungsmaßstäbe einzugehen.

In diesem Bericht werden die Werkzeuge wie folgt klassifiziert:

Gruppe	Klasse	Anzahl der Werkzeuge
1	Software-Unterstützungssysteme/ Programmier-Umfeld (Software Support System/Programming environment)	2
2	Software-Management und Wartung (Software Management, Control, and Maintenance)	110
3	Anforderungs/Entwurfsspezifikation, Analyse und Programm-Erzeugung (Requirements/Design-Specification, Analysis and Program Generation)	42
4	Software-Modellierung und Simulation (Software Modeling and Simulation)	7
5	Quellprogramm-Testen und Analyse (Source Program Testing and Analysis)	96

Tabelle 1: Klassifizierung der Werkzeuge

Der Großteil der Werkzeuge gehört zur Klasse "Software-Management, Kontrolle und Wartung" (110 Werkzeuge), die die Unterstützung von Software-Aktivitäten über den ganzen Lebenszyklus betrifft. Die verwendeten Verfahren sind meist wenig speziell und betreffen häufig die Dokumentationsverwaltung.

Die zweitstärkste Gruppe (96 Werkzeuge) betrifft die Klasse "Quellprogramm-Testen und Analyse", von der hier die Rede sein soll und die hier kurz als "Programm-Validation" bezeichnet werden soll.

Um eine grobe Übersicht der verwendeten Verfahren in dieser Klasse zu erhalten, wurden die Schlüsselwörter der Verfahren hier nach der Anzahl der Werkzeuge aufgelistet, in denen sie verwendet werden. Dabei können mehrere Verfahren zu einem Werkzeug gehören und sich auch inhaltlich überschneiden, wie z.B. die Verfahren "Pfadausdruck-Erzeugung" und "Pfadbedingungs-Auflösung". Da die verwendeten Begriffe von den Entwicklern angegeben wurden und für sie noch keine verbindliche Nomenklatur besteht, kann die Auflistung nur einen tendenziellen Überblick geben.

Verfahren, die nur ein- oder zweimal genannt wurden, wurden aus Platzgründen in der Tabelle nicht aufgelistet. Die Kürzel "S" und "D" in der letzten Spalte geben an, ob es sich um ein dynamisches Verfahren (d.h. mit Ausführung des zu testenden Programms) oder ein statisches Verfahren (ohne Ausführung) handelt.

Die aufgelisteten Verfahren sind untereinander keinesfalls bezüglich ihrer Aussagekraft gleichwertig oder von ähnlicher Komplexität; dennoch können einige Schlüsse gezogen werden. Zunächst ist auffällig, daß kein Werkzeug zur Unterstützung statistischer Tests existiert. Weiterhin hat nur ein Werkzeug (in obiger Tabelle nicht aufgelistet) als Verfahren den Programm-Beweis angegeben, das als schwierigstes Verfahren bekannt ist. Nur fünf Werkzeuge prüfen die Einhaltung von Programmier-Standards, und mit Ausnahme der Verfahren Zeit-Analysen (Lfd.Nr.14) und Laufzeit-Analysen (Lfd. Nr.4) werden keine Verfahren zur Validation von speziellen Echtzeit-Problemen angegeben (z.B. Aufzeigen verflochtener Zugriffe auf geschützte Resourcen).

Tabelle der verwendeten Verfahren

Lfd. Nr.	Schlüsselwort des Verfahrens	Anzahl der Werkzeuge, die das Verfahren enthalten	
1	Kreuz-Referenz-Listen	33	S
2	Dokumentationserzeugung	25	S/D
3	Test-Effizienz-Messung	23	D
4	Überdeckungs-Analyse	23	S
5	Code-Inspektion (code auditing)	20	S
6	Flußdiagramm-Erzeugung	20	S
7	Global Data Management	17	
8	Report-Erzeugung	15	D
9	Zuverlässigkeitsmaße	13	S/D
10	Datenfluß-Analyse	12	S/D
11	Programmfluß-Verfolgung (Trace)	9	D
12	Testdaten-Management	9	S/D
13	Konsistenz-Prüfungen	8	S
14	Zeit-Analysen	8	D
15	Versions-Überprüfung	7	
16	Testfall-Erzeugung	7	S
17	Profil-Erzeugung	7	D
18	Änderungs-Validation	7	
19	Ausführungs-Überwachung (execution monitoring)	7	D
20	Interface-Analyse	7	S
21	Pfadausdruck-Erzeugung	5	S
22	Programmstruktur-Untersuchung	5	S
23	Durchsetzung von Standards	5	S
24	Symbolische Exekution (symbolic evaluation)	5	S
25	Komplexitätsmaße	4	S
26	Laufzeit-Analysen	4	D
27	Zusicherungs-Überprüfung (assertion checking)	3	S
28	Fehler-Analyse	3	S/D
29	Pfadbedingungs-Auflösung	3	S

Tabelle 2: Verwendete Verfahren nach ihrer Häufigkeit

Beurteilung der Automatisierbarkeit von Validationsverfahren

Einige Validationsaufgaben können nur mit Werkzeuguntersützung sinnvoll gelöst werden. Eine Programm-Instrumentierung, deren Ergebnisse während eines Testlaufs zur Fortsetzung des Tests benötigt werden, ist nur dann sinnvoll, wenn ein Dialogsystem existiert, mit dem der Fortlauf des Tests beeinflußt werden kann. Solche Dialog-Test- und Bediensysteme gehören deshalb auch zu den ältesten Werkzeug-Entwicklungen. Auch die Instrumentierung selbst oder gewisse Dokumentationen zur Laufzeit des Testlings (Lfd.Nr.10,11,19,26) (Trace-Verfahren oder post-mortem-dumps) werden seit langem mit automatischen Werkzeugen durchgeführt.

Werkzeuge eignen sich sehr gut zu allgemeinen Dokumentationsaufgaben, etwa der Protokollierung von Testläufen (Lfd.Nr.8,17) mit den zugehörigen Eingangsdaten oder zur Generierung von Kreuz-Referenz-Listen (Lfd.Nr.1) und anderen lexikalischen Untersuchungen. Hierbei liegen die Schwierigkeiten eher in einer sinnvollen Ordnung und Begrenzung der erzeugten Datenmengen. Sie können aber auch weitaus schwierigere Aufgaben bewältigen, wie bei dynamischen Testhilfen die Überwachung des Testvorgangs zur Laufzeit mit Einbau von Haltepunkten, Steuerung der Uhr, die Kontrolle von Prozessen und Zugriffe auf Objekte im Dialog (Lfd.Nr.11,19, 26).

Statische Analysewerkzeuge enthalten häufig komplexere Verfahren zur Strukturaufklärung (Lfd.Nr. 6,12) und zur Testdatenerzeugung (Lfd.Nr.16).

Die Resultate der Strukturaufklärung müssen angemessen protokolliert werden. Dies geschieht häufig unter Verwendung eines Graphik-Software-Pakets.

Bei der Validation von Software ist die Gewinnung von Testdaten von großer Bedeutung. Günstig ist es, wenn bereits in der Spezifikations- und Codierungsphase Testfälle angegeben werden. Häufig steht man jedoch vor der Aufgabe, relevante Testdaten aus dem zu prüfenden Programm abzuleiten oder die Vollständigkeit vorhandener Testdaten zu prüfen. Da ein erschöpfendes Testen meist nicht möglich ist, müssen sinnvolle Test-Strategien ermittelt und befolgt werden. Dabei zeichnet sich ab, daß bestimmte Überdeckungs-Strategien (Lfd.Nr.4) allgemein als sinnvoll anerkannt werden [2]. Sie haben zum Ziel, alle Programmteile mindestens einmal auszuführen (C1-Überdeckung [2]) oder darüberhinaus alle Kombinationen von Ein- und Ausgängen von Einzelverzweigungen mindestens einmal zu durchlaufen.

Die Überdeckungsermittlung baut auf dem Kontrollflußgraphen des Testlings auf und liefert eine Menge von Pfaden, die getestet werden sollen. Für diese Pfade müssen dann die Testdaten gefunden werden, die die Pfad-Durchläufe bewirken. Das Verfahren der Pfad-Testdaten-Erzeugung (Lfd.Nr.21, 29) ähnelt dem der symbolischen Exekution (Lfd.Nr. 24) und ist in der Praxis nur schwer zu realisieren. Das Ergebnis des Verfahrens ist ein dem Pfad zugeordneter mathematischer Ausdruck ("Pfad-Ausdruck") aus dem direkt die Eingangsdaten abgelesen werden können, die zur Ausführung des Pfades dienen. Da dieses Verfahren z.Zt. nicht für beliebige Pfade praktisch mit vertretbarem Aufwand realisierbar ist, erfordert die Handhabung eines solchen Werkzeugs eine detaillierte Kenntnis seiner Wirkungsweise. Diese Problematik liegt auf der Hand, wenn man bedenkt, daß die Ermittlung aller vollständigen Pfadausdrücke eines beliebigen Programms es ermöglichen würden, das Programm so umzuschreiben, daß es außer einer Anfangs-Verzweigung keine weiteren inneren Verzweigungen enthält!

Werkzeuge können jedoch durch eine sinnvolle Darstellung der Quellprogramm-Informationen zur Lösung des Problems der Testdaten-Ermittlung beitragen.

Zwei weitere Fragen der Validation wurden bisher noch nicht vollständig beantwortet.

- Läßt sich ein quantitatives Maß für die Validation oder den Grad der Austestung angeben?

und

- läßt sich ein quantitatives Maß für die Komplexität eines Testlings angeben?

Es gibt mehrere Ansätze zur Klärung dieser Fragen; Einer wird an anderer Stelle dieses Heftes besprochen. Eine Bewertung der Validation wird von vielen Werkzeugen durchgeführt (Lfd.Nr.3), auch die Bestimmung von Komplexitätsmaßen (Lfd.Nr.25) wird von einigen Werkzeugen vorgenommen.

Zur Lösung der anfallenden Probleme bei der Software-Entwicklung und -Pflege reicht die Schaffung einer attraktiven Sprache und eines guten Übersetzungssystems allein nicht aus. Software-Entwicklung und -Pflege sollen nach Möglichkeit in ein Umfeld eingebettet sein, das in allen Phasen des Software-Lebenszyklus Hilfsmittel zur Bewältigung der Probleme anbietet. Im Zusammenhang mit den jüngsten Sprachentwicklungen wird dementsprechend die Schaffung eines Umfelds von Software-Werkzeugen angeregt, z.T. sogar gefordert. Daher wurden für die Sprache PEARL mehrere solcher Werkzeuge geschaffen, wie etwa Entwurfs-Hilfsmittel und Test-und Bediensysteme. In einer der letzten Ausgaben der PEARL-Rundschau wurde bereits ein bei der Gesellschaft für Reaktorsicherheit (GRS) entwickeltes Validationswerkzeug vorgestellt, der PEARL-Analysator [3]. An dieser Stelle sei kurz die Einordnung dieses statischen Analysewerkzeugs in die Reihe der Validationswerkzeuge dargestellt (Abb. 1-3). Die im Analysator verwendeten Verfahren betreffen die Lfd.Nr.1,2,6,10,13,16,21,22, und 29 aus Tabelle 2.

Eine detaillierte Beschreibung des PEARL-Analysators mit dem Titel "Der PEARL-Analysator, ein Software-Werkzeug zur Analyse von PEARL-Programmen" [4] kann auf Anfrage über die GRS bezogen werden.

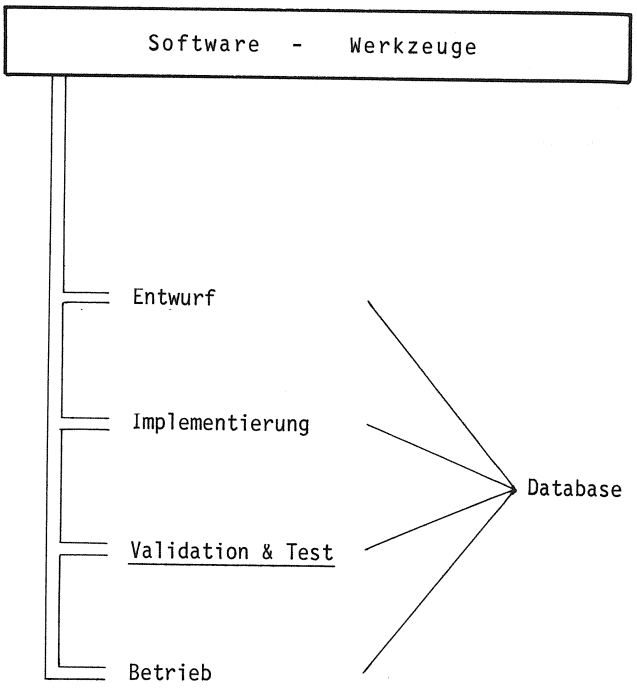


Abb. 1: Übersicht über Software-Werkzeuge. Unterstrichen: Anwendungsbereich des PEARL-Analysators

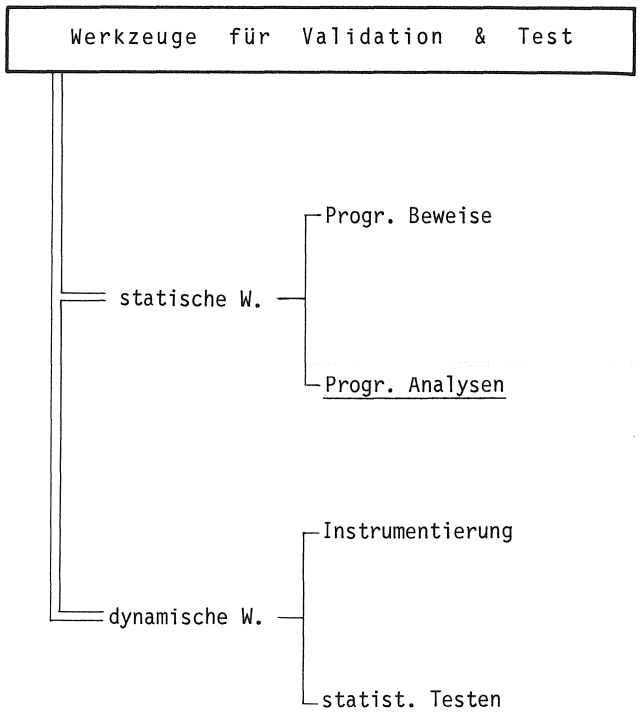


Abb. 2: Übersicht über Validations- und Testwerkzeuge.
Unterstrichen: Anwendungsbereich des PEARL-Analysators

Zusammenfassung

Validationswerkzeuge sind unterstützende Hilfsprogramme zur Analyse und zum Test von Programmen zum Zweck der Qualitätssicherung. Sie basieren auf verschiedenen Verfahren, angefangen bei einfachen Verfahren zur Verwaltung der Dokumentation in einem DV-System bis zur Implementierung von Programmbeweisverfahren. Ein Teil der Verfahren analysiert die zu validierenden Programme

- ohne sie auszuführen (statische Verfahren),

ein anderer basiert auf der

- Ausführung der zu validierenden Programme (dynamische Verfahren).

Werkzeuge und Verfahren werden eingesetzt, um

- die "Handarbeit" zu reduzieren
- die Validation zu formalisieren
- die Validation managebar zu gestalten
- die Validation besser dokumentieren zu können.

Die verwendeten Verfahren sind sehr vielfältig und betreffen meist nur bestimmte Gesichtspunkte der Validation. Ein allgemein anerkanntes, praktisch realisierbares und durchgängiges Verfahren zur Validation ist z.Zt. werder existent noch absehbar, die Entwicklung der Verfahren ist jedoch noch nicht abgeschlossen. Einige einfache Verfahren scheinen sich als Standard-Verfahren herauszukristallisieren, wie etwa das dynamische Testen nach bestimmten Kontrollflußgraphüberdeckungen. Die Dokumentation von Analysen und Tests ist bereits durchgängig in den Werkzeugen vorhanden.

Der Einsatz von Werkzeugen zur Validation von Software wird in Zukunft nötiger als je zuvor, nicht zuletzt aufgrund der steigenden zu bewältigenden Datenmengen.

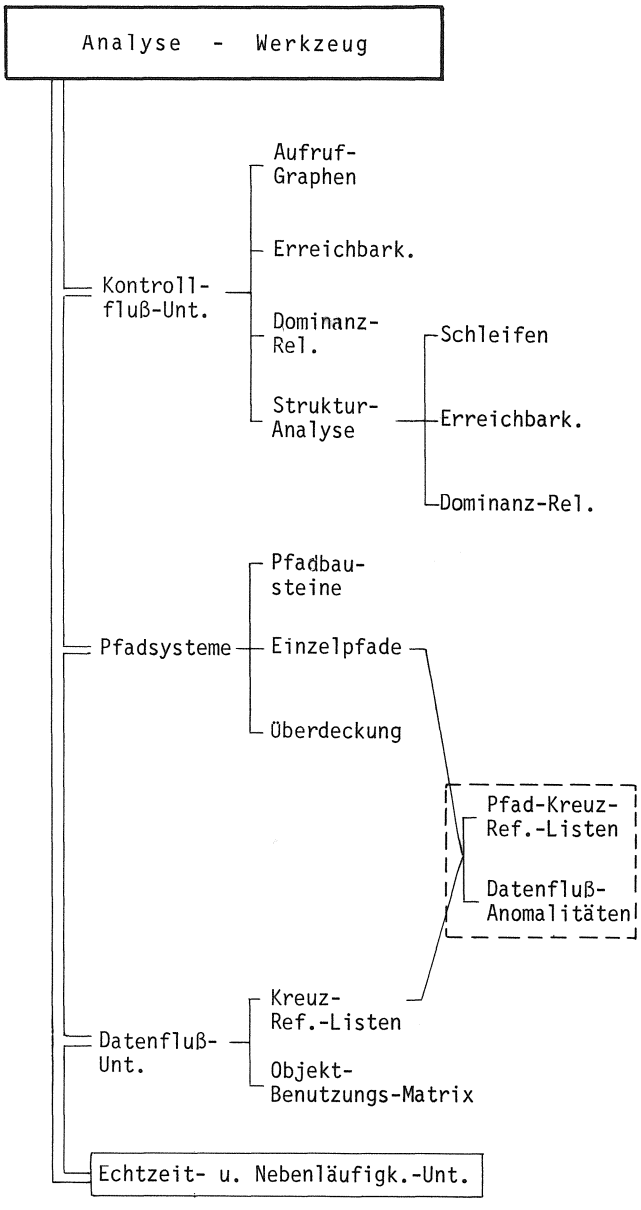


Abb. 3: Von Analyse-Werkzeugen abgedeckte Untersuchungen
□ : im PEARL-Analysator nicht realisiert
⬚ : z.Zt. in Arbeit befindlich

LITERATUR

[1] Houghton, R.C., Oakley, K.A. (Editors)
"NBS Software Tools Database"
U.S. Department of Commerce, Whashington
D.C. 20234, NBSIR 80-2159, October 1980

[2] Miller, E., Howden, W.E.
"Tutorial: Software Testing & Validation Techniques"
IEEE Computer Society New York, 1978
IEEE Catalog No.: EHO 138-8

[3] Puhr-Westerheide, P.
"Die statische Analyse von PEARL-Moduln mit dem PEARL-Analysator"
PEARL-Rundschau, Heft 6, Band 2, Dezember 1981

[4] Puhr-Westerheide, P., Märtz, J.
"Der PEARL-Analysator, ein Software-Werkzeug zur Analyse von PEARL-Programmen"
GRS-Bericht, GRS-A-669, Garching/München, Januar 1982

Dr. Puhr-Westerheide, Peter
Gesellschaft für Reaktorsicherheit (GRS) mbH
Forschungsgelände

8046 Garching bei München

Ein risiko-orientiertes statistisches Software-Testverfahren

B. Krzykacz, Garching

Zusammenfassung

Der vorliegende Beitrag befaßt sich mit einem statistischen Software-Testverfahren unter dem Gesichtspunkt des Risikos. Risiko ist definiert als der Erwartungswert des durch Softwareversagen bedingten Verlustes im künftigen betrieblichen Einsatz. Das Testverfahren liefert statistische Konfidenzaussagen über dieses Risiko; es ermittelt

a) eine obere Konfidenzgrenze für das Risiko zum Konfidenzniveau von (z.B.)95%

b) die Anzahl der durchzuführenden Testläufe, die notwendig sind, um eine vorgegebene obere Konfidenzgrenze für das Risiko einzuhalten.

Abstract

This paper introduces a statistical software test procedure with respect to risk. Risk is defind as the expected loss due to software failure in real operation. The main results of the procedure are some confidence statements about the risk; it evaluates

a) an upper (e.g. 95%) confidence bound for the risk

b) the smallest number of test cases required in order to verify some specified figure r to be the upper condidence limit for the risk.

1. Einführung

Die meisten statistischen Software-Testverfahren befassen sich mit der Berechnung oder Abschätzung bestimmter Software-Zuverlässigkeitskenngrößen wie MTTF (=mittlere Zeitdauer bis zum Versagen), MTBF (=mittlere Zeitdauer zwischen Versagensfällen), Versagensrate, Versagenswahrscheinlichkeit pro Anforderung, Anzahl der Fehler im Programm, usw. (vgl. [1],[2] und Schrifttum darin).

Diese Verfahren berücksichtigen in ihrer ursprünglichen Form weder die Möglichkeit unterschiedlicher Bewertung von Programmfehlern noch die Möglichkeit unterschiedlicher Folgen und Auswirkungen von Programmversagen im betrieblichen Einsatz.

Der vorliegende Beitrag befaßt sich daher mit einem statistischen Software-Testverfahren unter dem Gesichtspunkt des Risikos. Risiko ist hier definiert als der Erwartungswert des durch Softwareversagen bedingten Verlustes. Es setzt sich zusammen aus den Wahrscheinlichkeiten für Programmversagen und den Verlusten (gemessen in geeigneten Einheiten) die durch Programmversagen verursacht werden.

Der Test gehört zu den sog. geschichteten Testverfahren, bei welchen der Eingabedatenraum in Teilmengen (=Schichten) zerlegt wird und die Eingabedaten für die durchzuführenden Testläufe aus diesen Schichten in geeigneter Weise zufällig ausgewählt werden.

In diesem Beitrag erfolgt die Zerlegung des Eingabedatenraums in Schichten nach dem sog. Verlustprofil und die Zufallsauswahl der Testläufe nach dem sog. geschichteten Auswahlverfahren [3,S.318-334].

Das Ziel unserer Betrachtungen ist die Ermittlung
a) einer oberen Konfidenzgrenze für das Risiko zum Konfidenzniveau von (z.B.)95%.
b) der Anzahl der durchzuführenden Testläufe, die notwendig sind, um eine vorgegebene obere Konfidenzgrenze für das Risiko einzuhalten.

2. Grundlegende Bezeichnungen, Definitionen und Annahmen

Der Eingabedatenraum für das zu testende Programm wird in folgender Weise in Schichten zerlegt:
Es seien $L_1, L_2, \ldots, L_K$ alle möglichen Verluste (gemessen in geeigneten Einheiten, z.B. DM) die infolge eines Programmversagens im Betrieb entstehen können. Die Schicht h ist definiert als die Menge

aller Eingabedaten die im Falle vom Programmversagen zum Verlust L_h führen, h=1,...,K.

Auf diese Art wird der gesamte Eingabedatenraum vollständig in dispunkte Teilmengen (=Schichten) zerlegt.
K = Anzahl der Schichten.
Sei π_h die Wahrscheinlichkeit für die Schicht h, d.h. die Wahrscheinlichkeit dafür, daß im betrieblichen Einsatz Eingabedaten aus dieser Schicht ausgewählt werden.

Annahme: Die Zahlen $L_1,\ldots,L_K$ und $\pi_1,\ldots,\pi_K$ seien bekannt.

In analoger Weise zur Definition des Anforderungsprofils, bezeichnet man $(L_1,\ldots,L_K)$, $(\pi_1,\ldots,\pi_K)$ als Risikoprofil des zu testenden Programms.
Sei ρ_h definiert durch

$$\rho_h = \frac{L_h \pi_h}{\sum_{i=1}^{K} L_i \pi_i}$$

ρ_h ist bekannt, da L_h und π_h als bekannt vorausgesetzt wurden. ρ_h heißt "Bedeutung der Schicht h", da es interpretiert werden kann als Anteil der Schicht h an den zu erwartenden Verlusten im Falle von Programmversagen.

Sei p_h die Wahrscheinlichkeit für Programmversagen in Schicht h, d.h. p_h = Wahrscheinlichkeit aus der Schicht h Eingabedaten gemäß dem Anforderungsprofil auszuwählen, die zum Versagen führen.

p_h ist konstant und wird nicht als bekannt angenommen.
Sei L der Verlust, der durch Programmversagen verursacht wird. L ist eine diskrete Zufallsgröße und kann nur die Werte $0, L_1, L_2, \ldots, L_K$ annehmen.

Das Risiko r des zu testenden Programms ist definirt als Erwartungswert EL von L, d.h.

$$r = EL$$

Es ist leicht zu zeigen, daß r dargestellt werden kann durch

$$r = EL = \sum_{h=1}^{K} L_h \pi_h p_h$$

Da die Wahrscheinlichkeiten $p_1,\ldots p_K$ unbekannt sind, muß das Risiko r als eine feste aber unbekannte Konstante betrachtet werden.

3. Das Testverfahren

Sei H eine über der Menge {1,...,K} definierte diskrete Zufallsgröße mit $p(H=h)=\rho_h$ (h=1,...K);d.h. H ist die Zufallsgröße, die die Schicht h mit der Wahrscheinlichkeit ρ_h auswählt.

Der Test wird in zwei Schritten durchgeführt:

1) Eine Schicht wird gemäß der Verteilung von H zufällig ausgewählt (Zufallsauswahl nach der Bedeutung der Schichten)
2) Ein Testfall aus der zuvor ausgewählten Schicht wird ausgeführt. Die Inputdaten für diesen Testfall werden gemäß der Verteilung im Betrieb aus dieser Schicht entnommen.

Die beiden Schritte werden n mal unabhängig voneinander wiederholt.

Als Ergebnis des Testverfahrens werden die beiden Zahlen
n: Anzahl sämtlicher durchgeführter Programmläufe
k: Anzahl derjenigen Programmläufe, die zum Versagen geführt haben
bestimmt.

4. Konfidenzaussagen für das Risiko

Nach der Durchführung dieses Testverfahrens lassen sich einige nützliche Konfidenzaussagen für das Risiko r formulieren.

1) Die obere Konfidenzgrenze $\hat{r}_{0.95}$ für das Risiko r zum Konfidenzniveau von (z.B. 95%) lautet:

$$\hat{r}_{0.95} = \left(\sum_{h=1}^{K} L_h \pi_h\right) \frac{(k+1)\, F_{0.95}(2(k+1), 2(n-k))}{(n-k) + (k+1)\, F_{0.95}(2(k+1), 2(n-k))}$$

wobei
n = Anzahl der durchgeführten Testläufe
k = Anzahl der Testläufe die zum Versagen geführt haben
$F_{0.95}(\ldots,\ldots)$ = 95%-Quantil der F-Verteilung mit (2(k+1), 2(n-k)) Freiheitsgraden.

2) Im wichtigen Spezialfall k=0, d.h. wenn kein einziger der Testläufe zum Programmversagen geführt hat, lautet die obere 95%-Konfidenzgrenze für das Risiko

$$\hat{r}_{0.95} = \left(\sum_{h=1}^{K} L_h \pi_h\right)\left(1 - \sqrt[n]{0.05}\right)$$

$$\approx \left(\sum_{h=1}^{K} L_h \pi_h \right) \frac{3}{n}$$

für große n, z.B. n≧100.

3) Wenn a priori anzunehmen ist, daß während des gesamten Tests kein Versagen beobachtet werden wird, d.h. k=0 sein wird, kann man die erforderliche Testanzahl n so angeben, daß eine vorgegebene 95%-Konfidenzgrenze $\tilde{r}$ eingehalten wird.

$$n = \frac{\ln 0.05}{\ln\left(1 - \frac{\tilde{r}}{\sum_{h=1}^{K} L_h \pi_h}\right)}$$

$$\approx \frac{3}{\tilde{r}} \sum_{h=1}^{K} L_h \pi_h \qquad \text{falls} \quad \tilde{r} \ll \sum_{h=1}^{K} L_h \pi_h$$

5. Beweis der Konfidenzaussagen

ad 1):
Sei H_1, H_2,... eine Folge von unabhängigen und wie H verteilten Zufallsgrößen. H_i bezeichnet die Nummer der Schicht, die für den i-ten Testlauf zufällig ausgewählt wird; $p(H_i=h)=\rho_h$. Für eine vorgegebene Schicht h sei $X^{(h)}$ die Zufallsgröße, die das Ergebnis eines Tests mit Inputdaten aus dieser Schicht bezeichnet, d.h.

$$X^{(h)} = \begin{cases} 1 & \text{falls Programmversagen eintrat} \\ 0 & \text{sonst} \end{cases}$$

Da die Eingabedaten gemäß dem Anforderungsprofil ausgespielt werden, gilt:

$$p(X^{(h)} = 1) = p_h$$

Sei Y_i die Zufallsgröße, die das Testergebnis des i-ten Testfalls bezeichnet, d.h.

$$Y_i = \begin{cases} 1 & \text{falls im Testfall Nr i Versagen eintritt} \\ 0 & \text{sonst} \end{cases}$$

Da für den Testfall Nr i die Schicht von der Zufallsgröße H_i festgelegt wird, gilt die Darstellung:

$$Y_i = X^{(H_i)} \qquad i = 1,\ldots,n$$

Die Wahrscheinlichkeit für Programmversagen beim i-ten Test lautet:

$$p(Y_i = 1) = p(X^{(H_i)} = 1)$$

$$= \sum_{h=1}^{K} p\left(X^{(H_i)} = 1 \,/\, H_i = h\right) p(H_i = h)$$

$$= \sum_{h=1}^{K} p(X^{(h)} = 1)\, p(H = h)$$

$$= \sum_{h=1}^{K} p_h \rho_h$$

Die Anzahl Y aller fehlerhaften Programmläufe ist eine Zufallsgröße und wird dargestellt durch

$$Y = \sum_{i=1}^{n} Y_i$$

Da $Y_1,\ldots,Y_n$ stochastisch unabhängig sind, hat Y eine Binomialverteilung mit den Parametern n und

$$p = \sum_{h=1}^{K} p_h \rho_h \qquad \text{[3,S.167], d.h.}$$

$$Y \sim Bin\left(n;\ p = \sum_{h=1}^{K} p_h \rho_h\right)$$

Nach der Durchführung des Tests und der Beobachtung von Y (z.B. Y=k die Anzahl der Versagensfälle), kann man die 95%-Konfidenzgrenze $p_{0.95}$ für p nach der Formel

$$\hat{p}_{0.95} = \frac{(k+1)\, F_{0.95}(2(k+1), 2(n-k))}{(n-k) + (k+1)\, F_{0.95}(2(k+1), 2(n-k))}$$

angeben [3,S.224].
Da p durch r wie folgt ausgedrückt werden kann

$$p = \sum_{h=1}^{K} \rho_h p_h = \sum_{h=1}^{K} \frac{L_h \pi_h}{\sum_{i=1}^{K} L_i \pi_i} p_h =$$

$$= \frac{1}{\sum_{i=1}^{K} L_i \pi_i} \cdot r$$

erhält man die gewünschte Konfidenzgrenze für r:

$$\hat{r}_{0.95} = \left(\sum_{h=1}^{K} L_h \pi_h\right) \hat{p}_{0.95}$$

$$= \left(\sum_{h=1}^{K} L_h \pi_h\right) \frac{(k+1)\, F_{0.95}(\ldots,\ldots)}{(n-k) + (k+1)\, F_{0.95}(\ldots,\ldots)} \quad .$$

ad 2):
Falls k=0, so gilt: $\hat{p}_{0.95} = \dfrac{F_{0.95}(2,2n)}{n + F_{0.95}(2,2n)}$

Nach [3,S.225] gilt außerdem:

$$X \sim F(2,2n) \;\Rightarrow\; Z := \frac{X}{n+X} \sim \text{Beta}(1,n) \text{, d.h.:}$$

hat die Zufallsgröße X eine F-Verteilung mit (2,2 n) Freiheitsgraden, so hat $Z = \frac{X}{n+X}$ eine Beta-Verteilung mit den Parametern (1,n).

Da Z eine monoton wachsende Funktion in X ist (über dem Intervall $(0,\infty)$), berechnet sich das 95%-Quantil $z_{0.95}$ von Z aus $z_{0.95} = \frac{x_{0.95}}{n+x_{0.95}}$, wobei $x_{0.95}$ das 95%-Quantil von X ist.

Da die Verteilungsfunktion von Z durch

$$F_Z(z) = 1-(1-z)^n$$

gegeben ist, errechnet sich das 95%-Quantl $z_{0.95}$ von Z durch Auflösung der Gleichung

$$1-(1-z_{0.95})^n = 0.95$$

Daraus folgt:

$$z_{0.95} = 1 - \sqrt[n]{0.05}$$

Damit ist

$$\hat{r}_{0.95} = \left(\sum_{h=1}^{K} L_h \pi_h\right) \hat{p}_{0.95}$$

$$= \left(\sum_{h=1}^{K} L_h \pi_h\right) z_{0.95}$$

$$= \left(\sum_{h=1}^{K} L_h \pi_h\right) \left(1 - \sqrt[n]{0.05}\right)$$

Da $e^{-3} \approx 0.05$:

$$\sqrt[n]{0.05} = \sqrt[n]{e^{-3}} = e^{-\frac{3}{n}} \approx 1 - \frac{3}{n} \quad \text{für große } n$$

$$\Rightarrow\; 1 - \sqrt[n]{0.05} \approx \frac{3}{n} \quad .$$

ad 3):
Die gesuchte Anzahl n der benötigten Programmläufe bekommt man durch Auflösung der Gleichung

$$\tilde{r} = \left(\sum_{h=1}^{K} L_h \pi_h\right) \left(1 - \sqrt[n]{0.05}\right)$$

nach n und Anwendung der Näherungsformel

$\ln(1-x) \approx x$ für kleine Werte von x, sowie über das Ergebnis von 2).

6. Abschließende Anmerkungen

1. Setzt man $L_1 = L_2 = \ldots = L_K = 1$, so erhält man das geschichtete Testverfahren nach der Versagenswahrscheinlichkeit (partition testing).
2. Setzt man K=1 und $L_1 = 1$ so erhält man das gewöhnliche Testverfahren (black box testing) nach der Versagenswahrscheinlichkeit.
3. Die Anzahl N_h der Testfälle für jede Schicht h=1,...,K ist eine Zufallsgröße. Offensichtlich ist die gemeinsame Verteilung von $N_1,\ldots,N_K$ eine Polynomialverteilung mit den Parametern $n, \rho_1,\ldots,\rho_K$.
   Die zu erwartende Anzahl $E(N_h)$ der Testfälle für die Schicht h lautet deshalb

$$E(N_h) = \rho_h n = \frac{L_h \pi_h}{\sum_{i=1}^{K} L_i \pi_i}\, n$$

4. Die hier formulierten Konfidenzaussagen über das Software-Risiko machen nur von den Ergebnissen der statistischen Testfälle Gebrauch. Eventuell zuvor durchgeführte deterministische Testfälle oder Verifikationsversuche müssen unberücksichtigt bleiben. Es erscheint daher sinnvoll, Erfahrungen und Ergebnisse aus früheren Testphasen zu quantifizieren und in die statistischen Aussagen einzubeziehen. Für das vorliegende Testverfahren kann dies geschehen durch Anwendung des Bayes'schen Satzes mit einer zuvor geeignet gewählten a priori Verteilung vom Beta-Typ [4,S.175].

Schrifttum

[1] G.J. Schick, R.W. Wolverton: "An Analysis of Competing Software Reliability Models", IEEE Trans. Software Eng., Vol. SE-4, March 1978

[2] G. Rzewski: "Recent Advances in Software Reliability Methods" Third National Reliability Conference 1981, Birmingham 1981

[3] Heinhold, Gaede: "Ingenieur-Statistik" Oldenbourg-Verlag 1968

[4] Kurt Stange: "Bayes-Verfahren", Springer Verlag 1977

Anschrift des Verfassers:

Dipl.-Math. Bernard Krzykacz
Gesellschaft für Reaktorsicherheit (GRS) mbH
Forschungsinstitute

8046 Garching bei München

Tel.: (089) 32004-196

Konsistenzprüfung beim Integrieren von Basic PEARL Moduln

K. Lucas, München

Zusammenfassung

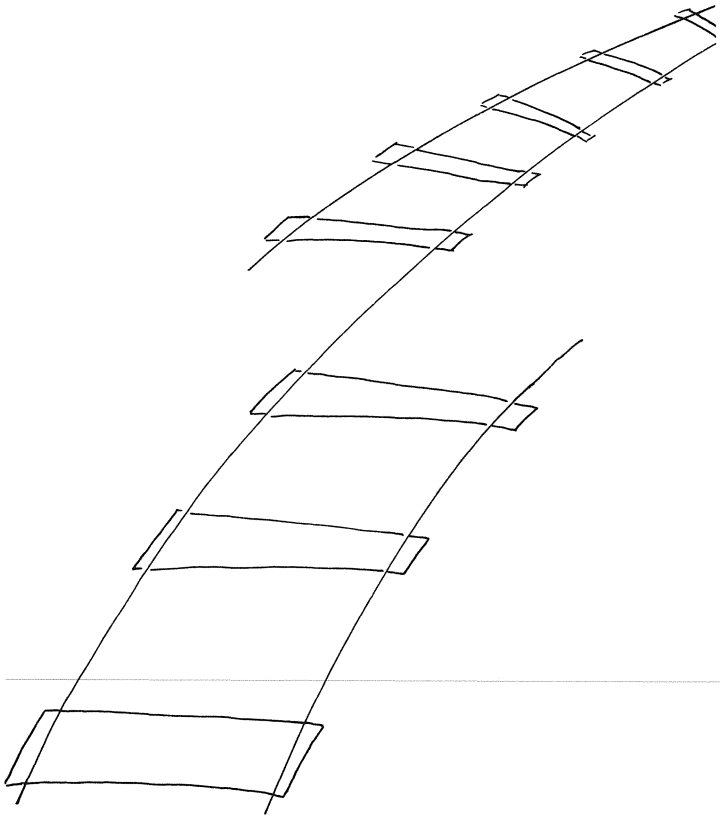
Es wird dargestellt, welchen Stellenwert die Integration in einem Übersetzungssystem für (Basic) PEARL einnimmt. Beim Zusammenstellen zu einem Programm müssen alle Module einer Prüfung unterzogen werden, um die konsistente Verwendung ihrer gegenseitigen Schnittstellen zu garantieren. Diesbezüglich unterschiedliche Vorgehensweisen in PEARL und Ada werden einander gegenübergestellt. Zuletzt wird die Implementierung eines portablen Konsistenzprüfprogrammes vorgestellt.

Summary

The importance of the integration phase in a Basic PEARL compilation system is described. During the linking phase of a program, the interfaces of all modules should be checked to guarantee consistent use of the global definitions.
The approaches taken in PEARL and Ada are compared. Finally, the implementation of a portable consistency checking program is presented.

Einleitung

Was nützt es dem PEARL-Anwender, wenn er glaubt, alle seine Module richtig programmiert zu haben und er erst zur Laufzeit feststellt, daß sie nicht aufeinander abgestimmt sind?



Die Konsistenzprüfung ist ein Beitrag zum Thema "PEARL in der Praxis". Sie widmet sich einer Phase in der Projektabwicklung, der i.a. wenig Beachtung geschenkt wird: der Integration. Die Erfahrung zeigt jedoch, daß die Integration einzelner Module zu einem funktionsfähigen Modulverband - dem Programm - häufig genug ein aufwendiges Unterfangen darstellt, vor allem dann, wenn das Zusammenspiel der Module von Phänomenen der oben dargestellten Art begleitet ist.

Behandelt werden in diesem Beitrag zur Konsistenzprüfung die folgenden Themen:

- Zuerst wird die Frage beantwortet, weshalb es in PEARL [1] überhaupt einer Konsistenzprüfung bedarf.
- Dann werden die verschiedenen Vorgehensweisen bei der Programmerstellung in Ada [2] und PEARL einander gegenübergestellt.
- Zuletzt wird die GPP-Implementierung eines portablen Konsistenzprüfprogrammes vorgestellt.

Warum bedarf es einer Konsistenzprüfung?

Eine Antwort auf diese Frage gibt folgendes Beispiel:

```
DCL
  F FIXED
    GLOBAL

F : = F + F
```

```
SPC
  F FLOAT
      GLOBAL

F : = F + F
```

Es handelt sich hierbei um zwei PEARL-Module, deren Inhalt auf das hier wesentliche reduziert wurde. Jeder Modul stellt für sich betrachtet bezüglich der Zugriffsfunktion F eine korrekte Übersetzungseinheit dar. Somit ist auch in jedem Modul eine Addition mittels F erlaubt.
Erst die Integration in einen Modulverband läßt erkennen, daß die gegenseitigen Schnittstellen nicht aufeinander abgestimmt sind. Dabei werden diese unterschiedlichen Zugriffsfunktionen wegen des gleichnamigen Bezeichners F einander zugeordnet. In bezug auf die Verträglichkeit gilt dann, daß die SPC-Definition sich der DCL-Definition unterordnen muß. Vor allem darf sie keine weitergehende Semantik zulassen, als in der DCL-Definition vorgesehen ist. In diesem Beispiel liegt jedoch nicht einmal Übereinstimmung in den Typen vor. Findet bei der Integration keine Konsistenzprüfung statt, dann geht die Zuverlässigkeit, die in jedem Modul für sich vorliegt, im Modulverband verloren.

Zur Untermauerung sollen nun typische Fehler aus der Praxis zitiert werden. Auch diese sind auf das wesentliche reduziert.

1. Gleicher Speicher:

```
DCL   A    FIXED   (7)
SPC   A    BIT     (7)
```

Die Binder eines Zielsystems identifizieren i.a. die globalen Objekte anhand der gleichen Adresse, oder besser anhand des gleichen Namens. Damit haben hier beide Module Zugriff auf ein- und denselben Speicher, z.B. ein 16 Bit-Wort. Während des Programmablaufs wird nun mittels der SPC-Zugriffsfunktion die linke Hälfte des Wortes erfaßt, mittels der DCL-Zugriffsfunktion dagegen die rechte Hälfte.

2. Vertauschung von Parametern

```
P:  PROC (X BIT, Y FIXED)
SPC P   ENTRY  (FIXED, BIT)
```

Dieser Fall tritt dann auf, wenn bereits entwickelte Module in einem weiteren Projekt wiederum Verwendung finden. Hier ist die Reihenfolge der Parameter nicht adaptiert worden.

3. Falscher SYSTEM-Teil

```
E:  CONNECTION-TO-INTERRUPT
SPC E   SIGNAL
```

Hier hat der Verantwortliche für den SYSTEM-Teil eine Hardwaregegebenheit dargestellt, die mit der Spezifikation im PROBLEM-Teil nicht abgestimmt wurde.

Ohne Konsistenzprüfung ist das Aufdecken solcher Fehler nur zur Ausführungszeit, bestenfalls zur Testzeit möglich. In jedem Fall ist ihre Lokalisierung äußerst aufwendig und kann in der Regel nur von denen durchgeführt werden, die mit der Implementierung vertraut sind. Fehler bei modulübergreifenden Schnittstellen äußern sich nicht zu Beginn des Programmablaufs, sondern erst dann, wenn sich gegenseitig ausschließende Operationen auf ein Objekt stattgefunden haben. Dann wiederum ist man auf der Suche nach Anweisungen, deren vermeintlich falsche Implementierung dieses Objekt überschrieben haben könnten.

Worauf ist eine derartige Fehlersituation zurückzuführen?

Die Ursache dafür liegt im Konflikt der beiden Ziele

- Typkonzept
- Modulare Programmerstellung

Der Compiler übernimmt die Aufgabe eine Übersetzungseinheit hinsichtlich der in der Sprachdefinition vorgegebenen Kriterien zu analysieren und sie schließlich, sofern sie diesen Kriterien genügt, aus der Quellsprache in eine Zielsprache zu transformieren. In PEARL -wie in allen aktuellen Programmiersprachen mit ausgeprägtem Typkonzept - liegt der Schwerpunkt der statischen Programmanalyse auf der Überprüfung der Typdefinitionen und deren korrekter Verwendung. Dadurch wird die Sicherheit und Zuverlässigkeit der Programme erhöht.
Andererseits ist für die Entwicklung von Software das Zusammensetzen eines Programmes aus einzelnen Programmteilen, den Moduln, von großer Wichtigkeit. In modernen Programmiersprachen handelt es sich nicht nur darum, Unterprogramme bereitzustellen; vielmehr ist auch zu berücksichtigen, daß die Unterprogramme ersetzenden Module in

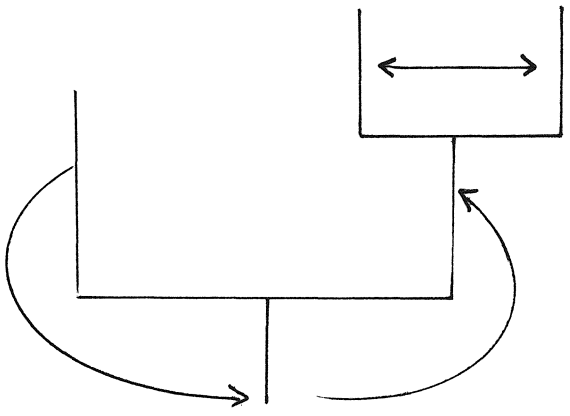
vielfacher komplexer Weise zu höheren, problemangepaßten Einheiten zusammengesetzt werden können. Ihre Wiederverwendung ist schon aus ökonomischen Gründen geboten.

Die Lösung des aufgezeigten Konfliktes ist über eine Konsistenzprüfung erreichbar. Schließlich soll die Sicherheit und Zuverlässigkeit eines Programmes nicht darunter leiden, daß es aus mehreren Moduln gebildet wurde. Diese Prüfung muß Teil einer statischen Programmanalyse sein. Ein Übersetzungssystem muß die korrekte Verwendung der Typen gemäß ihrer Definitionen auch im Modulverband garantieren.

Welche Aufgaben übernimmt die Konsistenzprüfung?

Da ist vor allem die Typprüfung. Die Vollständigkeit und Verträglichkeit aller globalen Bezüge ist zu überprüfen. Dadurch wird die Aussage des Compilers über die Zuverlässigkeit eines Moduls auf den gesamten Modulverband ausgedehnt. Alle Module müssen bei ihrer Integration dieser Prüfung unterzogen werden, um die konsistente Verwendung ihrer gegenseitigen Schnittstellen zu garantieren.

Als weiteres folgen Aufgaben, die nicht in der Programmiersprache selbst begründet sind, sondern die sich daraus ergeben, daß zum Ablauf nicht genügend Platz vorhanden ist und deshalb eine Überlagerungsstruktur erzeugt werden muß.



Globale Bezüge in einem segmentierten Programm

Ein Segmentbaum nimmt Einfluß auf die Sichtbarkeit der globalen Bezüge. Die Konsistenzprüfung muß solche Einflüsse auf die Zuordnung berücksichtigen.

1. Abwärtsreferenzen, die für Daten und Prozeduren gleichermaßen unproblematisch sind.

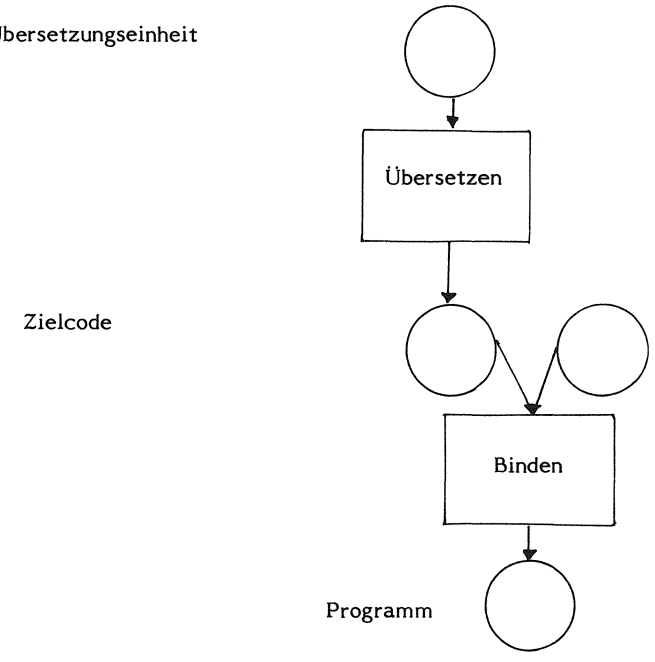
2. Aufwärtsreferenzen, die für Daten unzweckmäßig sind; für Prozeduren dagegen den Grund zum Segmentwechsel darstellen.

3. Seitenreferenzen, die in jedem Fall unzulässig sind.

Eine weitere Aufgabe stellt das Thema des sog. Vorbindens dar. Das Programm wird schrittweise aus Moduln zusammengestellt. Dabei werden nicht unbedingt alle globalen Bezüge zugeordnet. Solche im Prinzip nicht ablaufberechtigten Einheiten werden bspw. in Modulbibliotheken aufbewahrt. Die Eigenständigkeit solcher Einheiten kann es auch erforderlich machen, daß bei der weiteren Integration nicht mehr alle globalen Bezüge zur Verfügung stehen sollen; dann muß die Gültigkeit solcher Definitionen abgebaut werden. Man denke dabei an das Zusammenstellen eines mathematischen Paketes aus vielen Moduln mit vielen gegenseitigen Schnittstellen, das hinterher für seine Anwendung nurmehr wenige Schnittstellen aufweisen soll.

Programmerstellung in PEARL und Ada

In der Sprachdefinition von PEARL sind hinsichtlich der Programmerstellung keine speziellen Restriktionen enthalten. Demnach ist hier auch die klassische Vorgehensweise (Übersetzen-Binden-Laden) angebracht



Die einzelnen Arbeitsphasen sind speziellen Aufgaben zugeordnet. Beim Übersetzen werden die Module aus einer Sprache in eine andere transformiert. Auf diese Weise werden Anwendersprachen, eventuell über mehrere Zwischensprachen, auf den ausführbaren Zielcode überführt. Beim Binden hingegen werden Module aus nur einer Sprachebene zu einem Modulverband zusammengesetzt. Dabei werden zwar Referenzen gelöst und Adressen zugeordnet, eine Sprachtransformation jedoch findet nicht statt.

Als Gegensatz dazu soll jetzt die Vorgehensweise in Ada - stellvertretend für alle artverwandten Sprachen - skizziert werden.

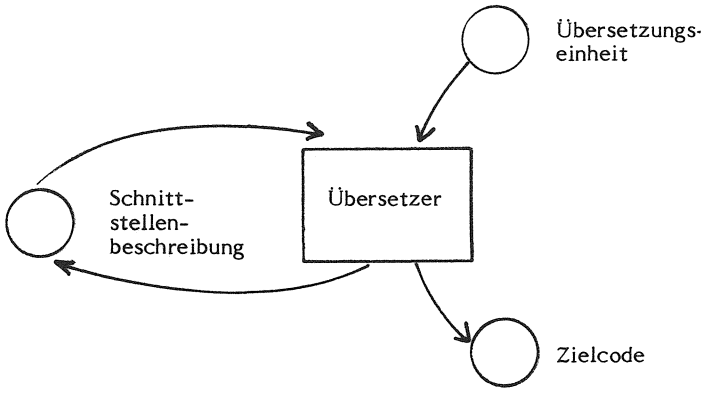
Das eingangs formulierte Beispiel für globale Schnittstellen präsentiert sich in Ada folgendermaßen:

```
PACKAGE L
F: INTEGER

F: = F + F
```

```
PACKAGE R
WITH   L

F: = F + F
```

Es handelt sich hier um zwei Übersetzungseinheiten, deren Inhalt auf das hier Wesentliche reduziert wurde. In L wird das Objekt F definiert; eine Addition ist zulässig. In R dagegen wird F nicht definiert. Um eine Addition durchzuführen, muß die Definition von F sichtbar gemacht werden. Dies geschieht durch die Anweisung WITH L. Durch sie werden alle Definitionen aus L importiert, und damit auch die von F. Grundsätzlich wird der Bezug auf globale Schnittstellen nicht explizit zum Ausdruck gebracht, sondern implizit anhand der Namen von Übersetzungseinheiten. Auf diese Weise besteht zwischen den Übersetzungseinheiten eine baumartige Ordnung.

Um konsistente Schnittstellen innerhalb eines Programmes zu garantieren, sieht die Sprachdefinition ein sog. libraryfile vor. Es bildet die Datenbasis der Schnittstellenbeschreibungen aller Übersetzungseinheiten eines Programmes.



Für jedes zu erstellende Programm wird ein solches libraryfile angelegt. Die Buchführung in dieser Datenbasis wird vom Compiler übernommen. Bei jeder Übersetzung müssen sowohl die globalen Definitionen aus vorangegangenen Übersetzungen entsprechend den Sichtbarkeitsregeln herangezogen, als auch Neuzugänge integriert werden.

Folgende Gegenüberstellung verdeutlicht die unterschiedlichen Vorgehensweisen bei der Programmentwicklung.

Modulentwicklung

In PEARL ist die Entwicklung eines Moduls unabhängig von anderen Moduln. Deshalb kann er auch für sich alleine übersetzt werden und in einer speziell für ihn zugeschnittenen Umgebung getestet werden.

In Ada ist eine Übersetzungseinheit fast immer von anderen Übersetzungseinheiten abhängig. Ihre Übersetzung ist erst dann möglich, wenn das library file alle Vorgänger enthält; dabei wird sie dann selbst integriert.

Modulhaltung

Übersetzte PEARL-Module lassen sich in Bibliotheken halten. Dort können sie auch mit Moduln aus anderen Programmiersprachen zusammentreffen.

Übersetzte Ada-Einheiten lassen sich nur zentral im library file des jeweiligen Programms halten. Eine Änderung im bereits erstellten library file zieht eine erneute Übersetzung und Integration aller abhängigen Einheiten nach sich.

Integration

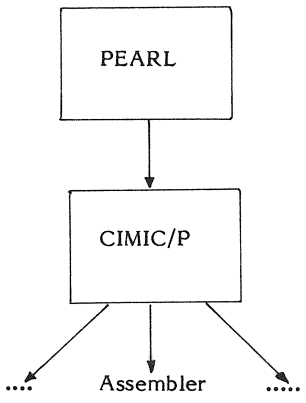
Bei der klassischen Vorgehensweise ist das Zusammenstellen eines Programms aus Moduln - die Integration - eine spezielle Arbeitsphase. In PEARL bedarf es dabei der Überprüfung der gegenseitigen Schnittstellen. Wird aus Platzgründen das Programm segmentiert, dann muß dabei zusätzlich die Sichtbarkeit der globalen Definitionen berücksichtigt werden.

In Ada entfällt diese Arbeitsphase völlig; die Integration ist Bestandteil der Übersetzung. Eine Überlagerungsstruktur läßt sich nicht explizit beschreiben. Implementationsabhängig kann der Aufbau eines Segmentbaumes mittels entsprechender PRAGMA's gesteuert werden. Allerdings muß dann das gesamte Programm so oft übersetzt werden, bis eine günstige Platzaufteilung gefunden ist.

Implementierung eines portablen Konsistenz-Prüfprogrammes

Beim Konzipieren eines Konsistenzprüfprogrammes für Basic PEARL Module erhebt sich die Frage nach der Sprachebene, auf der diese Überprüfung stattfinden soll.

Die GPP-Implementierung führt diese Prüfung auf der Zwischensprache CIMIC/P [3] durch.



Bei der Übersetzung eines PEARL-Moduls überprüft das front-end des Compilers die sprachspezifischen semantischen Zusammenhänge und transformiert ihn in CIMIC/P. Diese Zwischensprache ist einer abstrakten, PEARL ausführenden Maschine zugeordnet. Der Übergang auf reale Maschinen wird jeweils mit einem speziellen back-end vorgenommen.

Die Überprüfung der globalen Schnittstellen auf der Quellsprachebene durchzuführen ist nicht zweckmäßig. Bedeutet es doch eine Aussage zu treffen, die erst nach erfolgreicher semantischer Überprüfung des Moduls selbst sinnvoll ist. Andererseits ist auf der Ebene der Zielsprache wegen des bei der Übersetzung erfolgten Informationsverlustes über Typdefinitionen das Integrationsproblem nicht vollständig lösbar. In CIMIC/P dagegen liegen die nötigen Informationen über die Modulschnittstellen noch vor. Sie sind ihrer Bedeutung entsprechend in verschiedenen Listen aufbereitet. Somit kann eine schnelle Überprüfung der Konsistenz eines Modulverbandes auf Zielrechner-unabhängige Weise erfolgen.

Die Leistungen der portablen GPP-Implementierung eines Konsistenzprüfprogrammes für Basic PEARL beinhalten

Typprüfung

Sämtliche in DIN 66253 für Basic PEARL festgehaltenen Konsistenzregeln zwischen globalen Definitionen werden präzise geprüft.

Überlagerungsstruktur

Der Einfluß der Segmentierung auf die Sichtbarkeit von Definitionen wird berücksichtigt.

Vorbinden

Sowohl stufenweises Aufbauen eines Programmes als auch eventuelles Zudecken globaler Definitionen für weitere Integrationsschritte wird unterstützt.

Crossreference

Eine Übersicht über Deklaration und Spezifikation aller globalen Bezüge im Modulverband wird erstellt.

Das Konsistenzprüfprogramm ist Teil des GPP-Übersetzungssystems für Basic PEARL, dessen Verfügbarkeit aus der folgenden Tabelle hervorgeht.

Übersetzungs-Rechner / Ziel-Rechner	PDP-11/ /03 /23	INTEL 8086	MICRO NOVA
DATA-GENERAL NOVA			X
INTERDATA 7/32		X	
PDP-11/23 /34	X	X	
SIEMENS 330	X	X	X
SIEMENS 7.531	X	X	X

Stand: 01-Dez.81

Schrifttum

[1] DIN 66253: Programmiersprache PEARL;
Teil 1 - Basic PEARL (Vornorm), Teil 2 - Full PEARL (Normentwurf). Beuth-Verlag, Berlin/Köln, 1980.

[2] United States Department of Defense:
Reference Manual for the Ada Programming Language; proposed standard document; july 1980

[3] B.F. Eichenauer:
Spezifikation der Zwischensprache CIMIC/C;
PDV-Entwicklungsnotizen PDV-E88; GfK Karlsruhe 1976
B.F. Eichenauer, R. Henn, K. Lucas, A. Zeh:
Spezifikation von CIMIC/P; Technische Notiz GPP/5/77; GPP, München 1977.

Sprachkonzepte für die Parallelprogrammierung in Ada und PEARL

B. P. Eichenauer, München, vorgetragen auf der PEARL-Tagung 1981

Zusammenfassung

Die Sprachkonzepte für die Parallelprogrammierung in Ada und PEARL werden miteinander verglichen, um das Einsatzgebiet der beiden Programmiersprachen abzustecken. Aufgrund der weitgesteckten Ziele von Ada werden mit Ada mehr die Bedürfnisse des Systemprogrammierers abgedeckt. PEARL bietet dem Automatisierungsingenieur Ausdrucksmittel für die problemangemessene Darstellung der Parallelverarbeitung in Prozeßautomatisierungsprogrammen.

Abstract

The language concepts for parallel programming in Ada and PEARL are compared to find the applications area for both programming languages. As Ada has longrange goals Ada better satisfies the necessities of systems programming. PEARL offers language elements for the elegant formulation of parallel processing in process automation programs.

In jeder technischen Disziplin gibt es Methoden und Verfahren, die zur Lösung von bestimmten Aufgabenstellungen besonders geeignet sind. Wenn man den Aussagen zahlreicher Kollegen Glauben schenken dürfte, so würde eine technische Disziplin hiervon allerdings eine Ausnahme machen, nämlich die Programmierung von Digitalrechnern. Nach PL1 und ALGOL 68 kommt nun Ada. Alle diese Programmiersprachen erheben den Anspruch auf universelle Einsetzbarkeit entweder im kommerziell /technisch wissenschaftlichen Bereich oder im Bereich der Realzeit-Anwendungen. Als Beispiel seien hier die Ausführungen von I.C. Pyle / 1 / über das Einsatzspektrum von Ada zitiert:

"Ada is for programming embedded computer systems... ."

Embedded computer systems range from intelligent terminals and smart instrumentation to air traffic control or factory automation, via laboratory data monitoring, numerically controlled machine tools, navigation and guidance systems, stored program controlled telephone exchanges, batch and continuous production control, environmental monitoring, and future domestic products containing microcomputers. The computer involved may be large or small, single or a collection of many processors, or part of a computer network."

Außer diesem Anspruch auf Universalität findet man in Sprachvergleichen zwischen Ada und PEARL u.a. folgende Ansichten über PEARL (Sandmayr / 2 /):

"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"

oder

"Furthermore, its design is not very consistent, e.g. interrupts exist besides the very elaborated timing specification possibilities, ..."

Angesichts solcher Aussagen muß man sich die Fragen stellen,

- ob und in welchen Teilbereichen PEARL weiterhin Vorteile gegenüber universellen Realzeitprogrammiersprachen wie z.B. Ada bietet und
- ob denn die Ziele, die mit der Entwicklung von PEARL verfolgt wurden, allgemein richtig verstanden worden sind?

Zur Beantwortung dieser Fragen werden im folgenden die Voraussetzungen und Konzepte geschildert, von denen bei der Entwicklung der Sprachelemente für die Parallelprogrammierung in den beiden Realzeitprogrammiersprachen Ada und PEARL ausgegangen wurde. Es kommt uns dabei nicht auf eine lehrbuchhafte Schilderung aller gebotenen Möglichkeiten für die Parallelprogrammierung an. Wir wollen vielmehr plausibel machen, daß PEARL nach wie vor für den Anwenderkreis, für den die Sprache eigentlich geschaffen wurde, nämlich für Automatisierungsingenieure und Experimentatoren, attraktiv ist.

Die Abstraktionsebenen von PEARL und Ada

PEARL / 3 , 4 / ist eine Programmiersprache, die speziell für die Automatisierung von technischen Prozessen und von wissenschaftlichen Experimenten entworfen wurde. Die Abstraktionsebene bzw. das Vokabular von PEARL wurde so gewählt, daß die ingenieurmäßige Darstellung der Ergebnisse des Systementwurfs und der Systemdefinition von Prozeßautomatisierungssystemen möglichst unmittelbar in ein Automatisierungsprogramm umgesetzt werden kann. PEARL soll es auch dem nicht ständig programmierenden Automatisierungsingenieur ermöglichen, Automatisierungsprogramme zu entwerfen oder doch wenigstens ohne großen Aufwand nachzuvollziehen.

Ada / 5 / wurde wie PEARL für die Programmierung von Prozeßrechensystemen (sog. embedded computer systems) entworfen, soll aber das gesamte Spektrum aller denkbaren Anwendungen, d.h. von der konventionellen Prozeßrechner-Programmierung bis hin zur Entwicklung von Spezialprogrammen mit minimaler Betriebssystem-Unterstützung abdecken. Um dieses Ziel zu erreichen, wurden in Ada zwar gewaltige Sprachmittel für die Vereinbarung von Daten, für den Datenschutz und für die Formulierung von Rechenalgorithmen bereitgestellt; bei den Sprachelementen für die Parallelprogrammierung und die Ein/Ausgabe hat man sich jedoch auf das Allernotwendigste zu beschränken versucht (im Fall der Ein/Ausgabe gibt es nur einen Vorschlag für ein Unterprogrammpaket). Ada geht davon aus, daß die Programme für die zeitliche und logische Synchronisation des Programm- und Prozeßgeschehens und für die Prozeß-Ein/Ausgabe von Programmierern im Rahmen des Automatisierungsprogramms geschaffen werden. Der Automatisierungsingenieur kann Ada nur benutzen, wenn er fundierte Kenntnisse in der Systemprogrammierung besitzt.

Vereinbarung von Tasks

Prozeßautomatisierungsprogramme unterscheiden sich von kommerziellen Programmen im wesentlichen dadurch, daß sie synchron zueinander und mit Vorgängen außerhalb des Prozeßrechensystems ablaufen müssen. Während in kommerziellen Programmen normalerweise nur der durchzuführende Rechenalgorithmus beschrieben werden muß und es unerheblich ist, wann das Programm abläuft, sind in einem Prozeßautomatisierungsprogramm im Rahmen der sog. Automatisierungsfunktion / 6 / zu jedem Rechenalgorithmus auch die Bedingungen für seinen Ablauf und die Anforderungen an die Ein/Ausgabe anzugeben.

In der Regel werden mit einem Prozeßrechensystem mehrere parallel zueinander ablaufende Vorgänge in einem technischen Prozeß (sog. Teilprozesse) bearbeitet. Deshalb sind innerhalb eines Automatisierungsprogramms auch Vorkehrungen für die parallele bzw. quasiparallele Durchführung der zugeordneten Automatisierungsfunktionen zu treffen. Den verschiedenen parallel zueinander ablaufenden sequentiellen Teilprozessen in einem technischen Prozeß werden dabei im Automatisierungsprogramm sog. Programme oder Tasks zugeordnet, welche die Automatisierungsfunktionen für die Teilprozesse realisieren.

Betrachtet man die gängigen Automatisierungsvorhaben, so zeigt sich, daß sich die durchzuführenden Aufgaben mit einer festen Anzahl von Tasks realisieren lassen, die den Aufgaben statisch zugeordnet werden. Ein derartiger Entwurf eines Automatisierungsprogramms ist wünschenswert, weil dadurch u.a. die Betriebsmittelplanung und insbesondere auch der Sicherheitsnachweis erheblich erleichtert wird. Auch die Betriebssysteme der gängigen Prozeßrechner setzen voraus, daß eine feste Anzahl von Programmen bzw. Tasks in unmittelbar startbarer Form vorliegen.

In PEARL wurde deshalb nur die statische Definition von Tasks vorgesehen, denen bei ihrer Vereinbarung eine Priorität zugeordnet werden kann. PEARL-Tasks liegen beim Start eines PEARL-Programmsystems in startbarer Form vor und können bei den meisten Prozeßrechner-Betriebssystemen unter weitgehender Verwendung der vorhandenen Programmorganisation implementiert werden.

Die ursprünglich in PEARL vorgesehenen dynamisch entstehenden und vergehenden Tasks (sog. Subtasks) wurden 1980 / 7 / wieder aus dem PEARL-Sprachentwurf entfernt, weil ihre Implementierung bei den heutigen Prozeßrechnern erhebliche Probleme bereitet und darüber hinaus einen enormen Laufzeit- und VerwaltungsOverhead erfordert (u.a. Warten am Blockende auf die Beendigung aller Subtasks; Zugriff auf gemeinsame Datenbereiche; mehrfache Inkarnation einer Subtask, usw.).

Während bei der Konzipierung des PEARL-Taskings anwendungsorientierte Gesichtspunkte den Umfang der Sprachmittel bestimmen, soll das Ada-Tasking universell einsetzbar sein. Es soll z.B. gleichermaßen für die Entwicklung von Prozeßautomatisierungsprogrammen wie für die Entwicklung von Betriebssystemen geeignet sein.

Um dieses Einsatzspektrum zu befriedigen, wird in Ada von einem sehr allgemeinen dynamischen Taskmodell ausgegangen, das bezüglich der Vereinbarung von Tasks große Ähnlichkeiten mit dem ehemaligen PEARL-Subtaskkonzept auf-

weist. Ada- Tasks werden im Vereinbarungsteil einer Ada-Programmeinheit (Paket, Unterprogramm, Block oder Task) vereinbart; die Vereinbarung von Tasktypen und von sogenannten Taskfamilien (Felder von Tasks) ist möglich.

Hinsichtlich der Implementierung des Ada-Taskings gelten die oben bei der Erwähnung des PEARL-Subtaskkonzepts angeführten Bemerkungen. Insbesondere dürften unter Verwendung der derzeit im Einsatz befindlichen Prozeßrechner-Betriebssysteme dynamische Tasks und die damit verbundene komplexe Speicherorganisation kaum effizient realisierbar sein.

Die Autoren von Ada waren sich dieser Tatsache wohl bewußt. Sie gehen davon aus, daß für Ada ein relativ kleiner maschinenorientiert zu schreibender Kern erstellt wird, auf dem das ggf. anwendungsspezifisch in Ada zu schreibende Betriebssystem aufsitzen soll. Es bleibt abzuwarten, ob sich für heutige Prozeßrechnerstrukturen die Parallelläufigkeiten in Betriebssystemen hinreichend effizient mittels des Ada-Taskingkonzepts darstellen lassen.

Sowohl in PEARL als auch in Ada werden die relativen Prioritäten von Tasks untereinander als wichtige Voraussetzung für den ordnungsgemäßen Ablauf eines Automatisierungsprogramms angesehen. Da sich die Wichtigkeit einer Task während des Ablaufs eines PEARL-Programmsystems in Abhängigkeit von äußeren Ereignissen ändern kann, ermöglicht PEARL im Gegensatz zu Ada auch Prioritätsänderungen während des Ablaufs einer Task. Für PEARL ist die Prioritätsangabe i.A. eine Anweisung an die Laufzeitorganisation zur prioritätsgerechten Bereitstellung von Betriebsmitteln, während Ada die Prioritätsangabe als Organisationsanweisung an den Compiler ansieht.

Explizite Steuerung von Tasks

Für die Ablaufsteuerung von Tasks gemäß den Angaben in einer Automatisierungsfunktion sind in PEARL Steueranweisungen vorgesehen, mit denen Tasks gestartet (ACTIVATE), zurückgestellt (SUSPEND), fortgesetzt (CONTINUE) und beendet (TERMINATE) werden können. Beim Aktivieren und Fortsetzen einer Task kann die Priorität einer Task geändert werden.

In Ada sind keine Sprachelemente für die explizite Steuerung von Task vorgesehen. Alle Tasks, die in einem Vereinbarungsteil einer Ada-Programmeinheit vereinbart sind, werden kreiert und gestartet, sobald das Programm in die Programmeinheit eintritt. Unterbrechungen des Ablaufs einer Task werden nur implizite über Synchronisationsanweisungen herbeigeführt.

Synchronisation

Bei der Synchronisation von Tasks untereinander und mit zugeordneten Teilprozessen in einem technischen Prozeß unterscheiden wir zwischen logischer und zeitlicher Synchronisierung.

Logische Synchronisierung

Die logische Synchronisierung von Tasks untereinander erfolgt in PEARL über sog. Synchronisationsvariable. Vorgesehen sind sog. BOLT-Variable zur Organisation des synchronisierten Zugriffs auf Betriebsmittel (z.B. Speicherplatz) und SEMAphores zur Organisation des synchronisierten Ablaufs von Tasks.

In Ada wird die Synchronisierung von Tasks über das sog. Rendevouz-Konzept herbeigeführt.

Ada geht davon aus, daß die Synchronisierung von Tasks normalerweise erforderlich ist, weil eine Task eine Service-Leistung von einer anderen Task benötigt. Die Task, welche die Service-Leistung anfordert, begibt sich, falls die Leistung noch nicht erbracht worden ist, in den Wartezustand. Falls die Leistung schon vor ihrer Anforderung erbracht wurde, wartet die Task, welche die Service-Leistung erbringt, auf die Task, welche die Service-Leistung benötigt. Zu dem Zeitpunkt, zu dem die Leistung sowohl gefordert wird als auch erbracht ist, kann zwischen den beiden Tasks Information ausgetauscht werden; danach laufen beide Tasks wieder unabhängig voneinander weiter.

In dem folgenden PEARL-Beispiel werden zwei Tasks P und Q synchronisiert, wobei Q am Synchronisationspunkt Information an P übergibt:

```
DCL  S  SEMA ;
DCL  MESSAGE  FIXED ;
       .
       .
       .

P :  TASK ;
       .
       .
       .

     REQUEST S;
       .
       .

     (Verwendung von MESSAGE)
       .
       .

     END;
```

```
Q : TASK
      .
      .
      .

    MESSAGE : = .....
    RELEASE S;
      .
      .
      .
    END;
```

Dasselbe Beispiel läßt sich in Ada z.B. gemäß:

```
task P is
    entry A;
end P;
task body P is
        .
        .
        .
    accept A (MESSAGE : in INFO) do
        .
        .
        .
    end A;
        .
        .
        .
end P;
task Q;
task body Q is
  MESS : INFO;
        .
        .
        .
    P.A (MESS);
        .
        .
        .
end Q;
```

darstellen. Wie daraus ersichtlich ist, läßt sich in Ada die Synchronisation von Tasks unter Vermeidung von Programmgrößen aus der Umgebung der Tasks formulieren. Insbesondere wird der bei vielen Aufgabenstellungen notwendige Informationsaustausch zwischen Tasks (z.B. Auftraggeberidentifikation) in adäquater Weise ermöglicht.

Für die Erstellung zuverlässiger Programme ist auch selektive Anforderung oder Bereitschaft zu einem Rendezvous wichtig. Hierzu zwei Beispiele:

Soll in dem vorangegangenen Beispiel dafür gesorgt werden, daß spätestens nach einer Sekunde eine Maßnahme getroffen wird, falls P nicht zu einem Rendezvous bereit ist, so kann dies in Ada gemäß:

```
select P.A (MESS)
or delay 1.0;
      .
      .
      .
    (Maßnahme)
end select;
```

erfolgen.
Auch die das Rendezvous anfordernde Task kann Ersatzmaßnahmen angeben, falls das Rendezvous nicht innerhalb einer bestimmten Zeitspanne zustandekommt:

```
select accept A (MESSAGE : in INFO) do
      .
      .
      .
    end A;
or delay 1.0;
else begin
      .
      .
      .
    end;
end select;
```

Das in Ada vorgesehene Rendezvous-Konzept ermöglicht i.A. eine sehr elegante und durchsichtige Formulierung der Synchronisierung des Ablaufs von Tasks. Da es der einzige Mechanismus zur Synchronisierung in Ada ist, muß es aber auch zur Organisation des synchronisierten Zugriffs auf gemeinsame Daten mehrerer Tasks eingesetzt werden und kann hier zu umständlichen und aufwendigen Konstruktionen führen.

Als einfaches Beispiel sei hier die gemeinsame aber sich gegenseitig ausschließende Benutzung eines Datenfiles FILE betrachtet. In PEARL kann dieses Problem relativ einfach und durchsichtig durch Zuordnung einer Semaphore-Variablen erledigt werden:

```
DCL S SEMA PRESET 1
      .
      .
      .
P : TASK;
    REQUEST S;
      .
      .
      .
  (Verwendung von FILE)
      .
      .
    RELEASE S;
    END;
```

```
Q : TASK;
    REQUEST S;
      .
      .
      .
  (Verwendung von FILE)
      .
      .
      .
    RELEASE S;
    END;
      .
      .
      .
```

Da in Ada nur die Ablaufsynchronisierung über das Rendezvous-Konzept möglich ist, muß das Datenfile über eine eigens zu diesem Zweck einzuführende Task verwaltet werden:

```
task ORG is
    entry GET;
    entry DONE;
end ORG;
task body ORG is
    FILE : DATA;
      begin loop
        accept GET (F : out DATA) do
          F : = FILE;
        end GET;
        accept DONE (F : in DATA) do
          FILE : = F;
        end DONE;
        end loop;
        end;
end ORG;
task P;
task body P is
    FILE : DATA;
    begin
          ORG.GET (FILE);
            .
            .
            .
            .
          ORG.DONE (FILE);
    end P;
```

Wie auch bei dem früheren Beispiel ist auch hier ersichtlich, daß die vielleicht größere Zuverlässigkeit bei der Programmierung durch erhebliche Speicher- und Laufzeit-Overhead erkauft wird.

Die logische Synchronisation von Tasks mit parallelen Abläufen im technischen Prozeß erfolgt normalerweise über Interrupts. Im Systemteil eines PEARL-Moduls kann das Prozeßrechensystem und insbesondere auch das Interruptwerk des eingesetzten Prozeßrechners beschrieben werden. Dabei können an die verschiedenen im Benutzerhandbuch ausgewiesenen Interruptklemmen Namen vergeben werden.

Beispiel: INTR : KLEMME (7)⟶;

Der Interrupteingang, der im Benutzerhandbuch mit KLEMME (7) bezeichnet wird, erhält den Namen INTR.

Im Problemteil eines PEARL-Moduls wird nun zunächst die Verwendung der Unterbrechungen, die über INTR erzeugt werden, festgelegt. Dabei wird zwischen Unterbrechungen, die das PEARL-Programmsystem insgesamt (sog. INTERRUPTs) und Unterbrechungen, welche von einer der gerade ablaufenden Tasks erzeugt wurden und nur diese betreffen (sog. SIGNALS) unterschieden.

Nachdem nun INTR zu Beginn des PEARL-Problemteils gemäß:

```
SPECIFY INTR INTERRUPT;
```

als Unterbrechung, auf welche das gesamte Programmsystem zu reagieren hat, ausgewiesen wurde, kann die Synchronisierung zwischen Task und Teilprozess formuliert werden. Dazu kann einer der früher erwähnten Anweisungen zur expliziten Steuerung von Tasks eine WHEN-Bedingung (WHEN-Schedule) vorangestellt werden, die angibt, wann die Steueranweisung ausgeführt werden soll:

```
WHEN INTR ACTIVATE T;
```

Außerdem kann mit einer RESUME-Anweisung der Ablauf einer Task unterbrochen werden, bis bestimmte Interrupts eintreffen:

```
WHEN INTR RESUME;
```

Die logische Synchronisation zwischen Tasks und Abläufen außerhalb des Rechensystems wird in Ada auf das früher skizzierte Rendezvous-Konzept zurückgeführt, in dem einem Eingang in die Task eine Interruptklemme zugeordnet wird:

```
      .
      .
      .
INTR : integer : = 8#706#;
```

```
.
.
.
task INTR_HANDLER is
  entry HANDL;
  for HANDL use at INTR;
end INTR_HANDLER;
```

Im Unterschied zu PEARL, wo die Zuordnung zwischen Interrupt und Task beim Starten einer Task erfolgt, wird in Ada die Verbindung zwischen Interrupt und Task bei der Taskvereinbarung vorgenommen. Ein Austausch des Interruptantwortprogramms zur Laufzeit ist deshalb nicht möglich.

Zeitliche Synchronisierung

Ebenso wichtig wie die logische Synchronisation ist für das Prozeßrechnen die Ausführung von Tasks unter vorgegebenen Zeitbedingungen. Um die Häufigkeit, mit der Tasks ausgeführt werden sollen bzw. die Ablaufgeschwindigkeit von Tasks an die Geschwindigkeit der Abläufe außerhalb des Rechensystems anpassen zu können, müssen die bei der Systemanalyse ermittelten Daten über das zeitliche Prozeßverhalten - und ggf. auch das zeitliche Verhalten der Automatisierungsmittel selbst -bei der Erstellung eines Automatisierungsprogramms berücksichtigt werden.

Da, wie eingangs erwähnt, die ingenieurmäßige Beschreibung einer Automatisierungsaufgabe möglichst unverändert in ein PEARL-Programm umgesetzt werden soll, wurden in PEARL neue Datentypen und zugeordnete Rechenoperatoren vorgesehen, die eine ingenieurmäßige Behandlung von relativen und absoluten Zeitangaben (DURATION und CLOCK) ermöglichen.

Unter Verwendung von Zeitangaben können die Bedingungen für den zeitbedingten Start einer Task oder zur zeitweisen Unterbrechung einer Task problemorientiert formuliert werden:

```
ALL 5 SEC ACTIVATE MESSWERT_ERFASSUNG;
AT 12:0:0 ACTIVATE ZAEHLER_LESEN;
AFTER 10 MIN RESUME;
```

Die Möglichkeiten zur Behandlung von Interrupts und zur Angabe von Zeitbedingungen stellt keineswegs, wie u.a. Sandmayr / 2 / meint, eine Inkonsistenz in PEARL dar. Der PEARL-Anwender muß die Möglichkeit haben, auf Interrupts aus dem technischen Prozeß spezifisch zu reagieren; er soll aber möglichst weitgehend von der Entwicklung von Systemprogrammen, die nichts mit seinem Automatisierungsproblem zu tun haben (z.B. Ableitung der Uhrzeit aus Zeitunterbrechungen, E/A-Treiber, usw.) befreit werden.

In Ada ist für die zeitliche Synchronisation von Task mit Vorgängen außerhalb des Rechensystems nur eine Anweisung zur zeitweisen Unterbrechung des Taskablaufs vorgesehen:

```
DELAY zeitangabe;
```

die der PEARL-Anweisung:

```
AFTER durationangabe RESUME;
```

entspricht. Alle übrigen zeitsteuernden Bedingungen, wie z.B. das ALL- oder AT-Schedule, müssen in Ada im Rahmen des Anwenderprogramms ausgehend von Unterbrechungssignalen eines Zeitgebers realisiert werden.

Fazit

Wie die Gegenüberstellung der Konzepte für die Parallel-Programmierung in Ada und PEARL zeigt, bieten beide Sprachen ein hinreichendes Vokabular zur Erstellung von Prozeßautomatisierungsprogrammen.

PEARL ist mehr auf die Bedürfnisse des automatisierenden Ingenieurs und auf die Leistungsfähigkeit heutiger Prozeßrechensysteme zugeschnitten und nimmt dafür ein reduziertes Einsatzspektrum in Kauf.

Die Parallelprogrammierung in Ada orientiert sich an den Bedürfnissen des System-Programmierers um möglichst alle denkbaren Einsatzfälle abzudecken, macht aber die Entwicklung von speziellen Betriebssystemen und von Anwenderpaketen erforderlich, in denen die regelmäßig für die Automatisierung technischer Prozesse nötigen Dienste angeboten werden. Da solche Dienste nicht durch die Sprache vorgeschrieben werden, ist bei Ada die gleiche Situation wie bei Realzeit-FORTRAN zu erwarten. Automatisierungsprogramme werden unter Voraussetzung unterschiedlicher Laufzeitsysteme erstellt und sind damit nur bedingt austauschbar.

Schrifttum

/ 1 / Pyle, J.C:
The Ada Programming Language;
A. Guide for Programmers;
Prentice Hall International; 1981

/ 2 / H. Sandmayr:
Sprachen für die Echtzeitprogrammierung;
Fachtagung Prozessrechner 1981 München
abgedruckt in: PEARL Rundschau, II, Nr. 2,
(Juni 1981)

/ 3 / DIN 66253:
Teil 1 - Basic PEARL (Vornorm)

/ 4 / Teil 2 - Full PEARL (Normentwurf)
Beuth-Verlag Berlin/Köln; 1980

/ 5 / United States Department of Defense:
Reference Manual for the Ada
Programming Language;
proposed standard document; July 1980

/ 6 / R. Lauber:
Prozeßautomatisierung I
Aufbau und Programmierung von
Prozeßrechensystemen;
Springer-Verlag Berlin/Heidelberg/New York; 1976.

/ 7 / PEARL Arbeitskreis:
Bericht zur 38. PAK-Sitzung
am 14.7.1980 (PAK 4-80)

Introducing PEARL at the OECD Halden Project

G. Dahll and **C. V. Sundling,** OECD Halden Reactor Project

ABSTRACT

The PEARL language has been implemented on the NORD computer at the OECD Halden Reactor Project. To obtain experience with PEARL, as well as to evaluate its quality, a disturbance analysis system, previously written in FORTRAN and assembly, was recorded in PEARL. This article describes the implementation as well as the recoding, and emphasizes the experiences and views of the persons working on the project, both on PEARL as a language and the specific implementation at the Halden Project.

1. INTRODUCTION

The OECD Halden Reactor Projects is an international project doing experimental operation on the Halden Boiling Water Reactor and associated research programmes. One of the main ares of interest is the use of computers in control and supervision of the process.
At the Halden Project (HP) FORTRAN and assemply language have been the main programming languages for batch and real-time applications. As a part of the HP-programme, a study of other and modern high level languages was started in 1977. The proporties of such a language should fulfill the following main requirements:

- Portable, machine independent
- Structured programming
- Real time oriented

The FORTRAN ANSII standard has been redefined (1977), especially concerning character variables and IF-THEN-ELSE structures, and is to some degree portable between different computer systems. FORTRAN, however, has no defined standard functions for real time operations, like starting programmes, aborting programmes, setting programmes on intervals, operating on synchronizers, etc.

For these functions each computer manufacturer supply their own subroutines with interface to the operating system.

PEARL was considered to be more portable than FORTRAN and is originally designed for real time applications. Especially I/O and tasking functions are taken well care of in PEARL, since they are defined as standard features of the language. FORTRAN also has standard I/O statements, but experience show that they are implemented with slight differences from one computer to another. Some systems have extensions which make it inconvenient to move the source code to another computer and compile it without modifications.

During 1977 a group of German nuclear utilities recommended the introduction of PEARL at the Halden-Project. Previously, a programme system for disturbance analysis (The STAR system (1) was developed as a joint project between Gesellschaft für ReaktorSicherheit (GRS) in Garching and HP. This programme system was written partly in FORTRAN and partly in assembly, and on a contract from GRS, HP should translate this programme system into PEARL and implement it on a NORD computer.
On background of the German interest, the manufacturer of the NORD computers, Norsk Data A/S, was interested in implementing PEARL as a language in their computers. As the STAR system is an interactive system, it uses a set of standard subroutines, KEYCON, which handle dialogues between one, or more, operators and the computer. KEYCON was coded in assembly, but interest was expressed in translating it into PEARL, and HP was given a contract by PDV on performing such a translation.

This article will contain a description, as well as views and experiences, on the implementation of PEARL and the coding of STAR and KEYCON.

2. IMPLEMENTATION OF PEARL IN A NORD COMPUTER

It was essential to use the existing Norsk Data software including the operating system for NORD computers, SINTRAN III, as far as possible. System software like the file system and the loaders which are parts of SINTRAN-III, should preferably not be modified. The reentrant FORTRAN library to be used has most of the arithmetic routines required for the runtime library.

The following major programme modules concerning the implementation included:

- Compiler
- Runtime library
- SINTRAN-III monitor
- SINTRAN-III real-time loader
- I/O package, formatted and binary

The implementation work was performed partly by Entwicklungsbüro Wulf Werum (EWW), Lüneburg, and partly by HP. We will in the following shortly describe some aspects of the implementation of each module.

- The Compiler

The compiler, which was made by EWW, was transferred to the N-10 by means of a SIEMENS 300 computer. It consists of two parts, the uper part, the compiler itself, and the code generator. Both are coded in the GBL language. The upper part translates PEARL source code to the intermediate language IL1 and the code generator from IL1 to object code. The upper is of a quite general nature, whereas the code generator had to be changed to adapt the object code for the specific computer where it shall run.

In the first version of the compiler the object code was made in the assembly code (MAC) used in the NORD computers. This code should again be assembled by the MAC assembler into a binary relocable form (BRF) before it could be loaded for execution. It turned out, however, that this compilation method was very time consuming, and it also created some other problems which were difficult to overcome. The next version of the compiler was therefore translating the IL1 code directly into a BRF code. This reduced the compilation time substantially, and reduced the number of passes to 8.

Since the compiler is written in GBL and translated by the code generator itself, which does not generate optimal code, the throughput is low. The philosophy of the compiler constuction is that it shall be easy to adapt it to different computer systems. A lot of I/O to disk also slows down the compilation time and the throughput. Compared to the FORTRAN compiler it is rather slow, but more checking is done since the PEARL language is more complex. Some test programmes have been made to test loops, calculations, character handling etc. for PEARL versus FORTRAN. Generally the execution speed is somewhat slower (ca. 1,5 times) for PEARL except for character handling.

- The Run-Time Library

PEARL requires reentrant run-time library rountines. It was decided to use the library for reentrant FORTRAN made by Norsk Data since most of the arithmetic functions existed in this library. An interface routine (interpretative) similar to the one used in FORTRAN was designed for calls to PEARL procedures. This interface routine makes it possible to call subroutines written in FORTRAN and MAC as well.

- The SINTRAN-III Monitor

To satisfy the PEARL tasking control functions, changes in the existing SINTRAN-III monitor were necessary. The changes were made by HP, and Norsk Data assisted when problems ariesed. To avoid patching of an existing SINTRAN-III at Halden, at complete SINTRAN-III generating system was retrieved from Norsk Data. This system is used to generate and update a special PEARL SINTRAN operating system, and it will also perform a correctness test of the operating system after a modification.

The following parts of SINTRAN had to be changed:

RTENTRY, which is testing programme status and scheduling programmes for execution etc. PEARL requires more status and programme information than SINTRAN in its standard version. To handle this extra information a RTENTRY programme particularly designed for PEARL that also was able to perform the standard funktions, became necessary.

The RT-description, which is a table in SINTRAN resident area containing status and registers for each RT-programme in the system. This table has a fixed length for each programme and is made when SINTRAN is generated for each specific CPU. PEARL need more information in this table, but the length of the table could not be increased since this would influence other users that are not running PEARL. This causes a lot of problems, but they were overcome.

- The Real-Time Loader

It was decided that the loading of a PEARL task should be similar to loading a FORTRAN reentrant programme. The stack for each task is defined by the real-time loader, and the FORTRAN library is loaded automatically. A separate PEARL real-time library must be loaded separtely.

- The I/O Package

As mentioned before the PEARL I/O package made by Werum was coded in PEARL. The package consists of about 100 procedures and occupies some 11,3 Kwords of memory, including 20 COMMON blocks of 1 Kwords. Because of the amount of calls between the different procedures and calls to library functions, all done via the interface routine, the execution speed was far from optimal. Also the object code produced by the compiler does not use the NORD instruction set

one would do in assemply language. The source of the I/O package became available to HP and a recoding to assembly started. After having recoded some of the routines, it was experienced that it would speed up the I/O so much that a complete recording was justifield. The size of the object code would be drastically reduced as well. The object code is now 4.1 K-words and is not using COMMON. The number of defined entry points are reduced from 100 to 10 and make the symbol table in the loaders smaller.

3. A SUMMARY OF THE STATUS OF PEARL ON NORD COMPUTERS

The PEARL implemented on NORD computers is a subset of full PEARL. In addition to Basic PEARL some features of full PEARL are available, making efficient programming easier and solutions to complex problems more elegant. Other features of full PEARL that require additional memory and execution overhead are not implemented. The extensions and restrictions are listed in the PEARL Language Reference Manual (2).

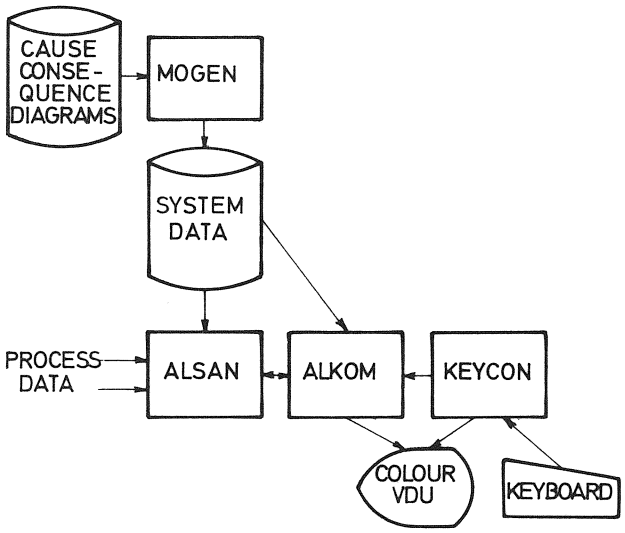
The following points describe some of the features of PEARL on NORD computers:

- PEARL can run on any N-10/N-100 CPU with 48 bits floating point format under SINTRAN-III
- All standard functions of SINTRAN-III are available
- A task with its data is limited to 64 Kwords. Several tasks can be loaded on the same segment, or several segments linked to a common segment containing data and/or procedures.
- The comiler runs as reentrant subsystems and can be used by several users simultaneously.
- The compiler generates BRF code accepted by the RT-LOADER
- Separate compiling of modules is possible, no parameter checking is done at load-time.
- Compiler options:
  - Generate BRF or MAC code
  - Enable/disable runtime index checking
  - Cross-reference tables
  - Conditional compiling
- Tasks can trigger software interrupts and wait for software interrupts. (16 channels with 16 lines)
- PROCEDURES are reentrant and PEARL tasks can call subroutines written in FORTRAN and MAC
- Tasks coded in other languages than PEARL can be scheduled by means of the standard functions from a PEARL task
- PEARL tasks can run as real time or batch/background programmes under SINTRAN-III

4. RECODING OF STAR IN PEARL

THE STAR SYSTEM

STAR is an acronym for "Störungs Analyse Rechner" and is a computer based system developed as a joint project between GRS and HP (1).Its task is to analyse disturbances in a process and present to the operator information of causes and possible consequences to the disturbance. It was developed for nuclear power plants, but is of general nature and can be applied to any process plant. A pilot implementation of the system is made in the nuclear power plant Grafenrheinfeld.



The structure of the STAR system is shown in Figure 1. A prerequisite for the on-line analysis is an apriori analysis of the process. This analysis is made in the term of cause-consequence diagrams (3). A cause-consequence diagram is a directed graph, where the nodes are of various types (e.g.prime causes, events,logical gates etc.)The nodes may also have other attributes (e.g. observability, text etc.). All this information of the nodes are written in a "user friendly" language, with syntax defined in BNF, and translated by the MOGEN (MOdel GENerator) programme into a set of tables (System Data). These tables contain information on the conditions for when an analysis shall be made and how it shall be performed, and is structured to make the analysis time efficient. The MOGEN programme is a batch programme written in PL1 and runs on an Amdahl computer. The System Data are transferred via a floppy disk to a NORD computer used for the real-time analysis.

The real-time part of the STAR system is divided into three modules. The ASLAN module is the one which performs the analysis. Binary and digitized analog process data are read concurrently from the plant and analysed according to the rules given in the System Data tables. The result of the analysis is transferred in a coded form to the ALKOLM module. The task of the ALKOM module is to decode and display the result of the analysis on a colour VDU. The STAR system has an interactive mode of operation, in two ways. Firstly, the results are not presented automatically to the operator, but only upon this request. Secondly, the system can ask the operator for information which it cannot get directly from process instruments, and the operator can answer these questions via a keyboard. The operators' interaction via a keyboard is handled by the KEYCON module. This module is not specific for STAR, but can be used by any interactive programme system.

The Coding of STAR in PEARL

As the MOGEN module is a batch programme written in PL1 and run on a general purpose computer, and as PEARL is a language especially suited for real-time programmes, only the modules ALSAN, ALKOM and KEYCON were recoded in PEARL. As one of the objectives of this project was to spread the knowledge about PEARL and to evaluate its qualities, the three modules were coded by three different persons. (These are not necessarily modules in the PEARL sense. The latter will hereafter be written with capital first letter as is done with other PEARL specific words). The mode of proceeding for the work should follow the general rules:

- Each person should make his module as independet of the others as possible
- The interfaces (control and data transfer) should be clearly specified in agreement between the programmers of the modules concerned
- The programmers should be guided, althoug not necessarly follow, the recommendations for design and coding made by EWICS TC7 (4)
- Each programme part (task, subroutine or function) should be documented according to the recommendation given in part 3, section 3.3.3 of the EWICS TC7 guidelines for documentation (5)
- The functioning of the recoded STAR should be the same as for the one already existing (i.e. the same input should give the same output), but otherwise the programmers were free to programme the way they wished. They should, however, avoid to translate the existing programmes statement by statement, but rather try to utilize the characteristics of PEARL.

One of the main reasons for making these rules was to promote a utilization of the possibilities in PEARL to make a transparent and structured programme. This should lead to a more reliable programme and a programme which is more easy to validate.

We will in the following give some examples on how the coding was realized, considering these rules.

The ALSAN and ALKOM programmes are running asynchronous. ALSAN consists of two tasks, one is activated only once and reads systems data from file and does some initialization. The other task, which is performing the analysis, is acivated by a Schedule Statement at regular intervals. The interface to ALKOM is made in two ways. The transfer of results to ALKOM is made via a ring buffer controlled by two Sema Variables. ALKOM consists of two tasks. One of these handles the output from ALSAN. This task is set in an eternal loop and runs as long as there are messages in the buffer. The other task activated from KEYCON. KEYCON is running in an eternal loop waiting for input from the keyboard. This input is stored in a buffer, and by the input of certain control characters ALKOM is Activated and uses the data in the input buffer. If the buffer contains the answer to a question, this will be put in an answer buffer which will be used by ALSAN. There is, however, no transfer of control from ALKOM to ALSAN.

In this way the interfaces between the modules were agreed upon between the programmers, and the programming of the modules was then performed independently. This mode of progress turned out to work without problems. In this way we also got an experience in utilizing the ability of PEARL to handle concurrent processing.

Even if the three modules were made independently, and even if the guidelines of EWICS TC7 (4) were not followed literally, it turned out that the programmes followed the general ideas of these guidelines. This includes hierarchic programme structure; structured programming; hierarchic stucturing of data using Structures; mnemonic names of data; single entry - single exit subroutines, with well defined functions etc. There were parts of the guidelines of (4) which certainly could not be followed. This was first of all the recommendations on the compiler and operative system, which was completely new, respectively substantially modified. Also, the guidelines on the design

could neither be followed. But as this programme should not be implemented in its present form in a safety related function, but rather be used as an example for studying the qualities of PEARL, the coding part of the programming was specially emphasized. We think that the coding was made in a style suitable for verification. There is one recommendation of (4) which was purposely not followed. Since subroutines were used to break up the programme modules in an hierarchic structure, the recommendation that subroutines should communicate exlusively via their parameters to their environment would be very clumsy to follow and produce time consuming code. The programmers of ALKOM and KEYCON utilized the hierarchic scope structure of PEARL, where variables declared at a higher level are also known at lower levels. All the subroutines in these modules did, however, have an introduction,in accordance with (5), where all variables used in the subroutine are mentioned. The programmer of ALSAN did not like the possibility of using this hierarchic scope structure (which is also used in Pascal and other languages). He therefore solved the problem by utilizing PEARL's options to compile each subroutine separtely. All variables which are common to more than one subroutine are therefore Declared in the main Module and Specified in all other Modules.

The Programmers'Views on PEARL

Below is a list of some general views the programmers had on the PEARL language itself, and on the specific implementation at HP. The programmers had a mixed experience in FORTRAN, Algol, Pascal and assembly. This has influenced the views of the programmers, but we will not here discriminate between the views of the different programmers.

- PEARL has clear advantages compared to FORTRAN (and assembly) as a language for writing well structured programme which is easy to understand and analyse for verification. However, we find Pascal even better in this respect, especially in the data structuring possibilities.
- It is a clear advantage that PEARL, in contrast to FORTRAN, has a well defined syntax. It is, however, written in a form which is difficult to read. We think it will be helpful if the syntax could also be written in syntax diagrams.
- PEARL is specially designed for real-time programming, and these aspects are very well taken care of. This enhances the ease of programming as well as the understandability of the real time aspects. One exception is the declaration of DATION's which we found somewhat difficult to understand and use. This is partly due to the fact this concept covers so many possibilities, but also due to the freedom the programmer has in writing a DATION declaration, e.g. by allowing default values and the possibility to interchange the sequence of the terms in the declarations (this may be implementation dependent).
- We find the way PEARL allows the writing of separate Modules which can be compiled and tested independently very good. This eases the division of the programing to different programmers. The interfaces between the modules are also easy to define.
- Even if it is the intention that PEARL programmes should be portable, there are various items which are not well defined in the PEARL language, and which are therefore implementation dependent. This has negative effect on the portability. In our case, we had special problems with the conversion between Characters, Fixed Variables and Bit Strings, which were solved in a peculiar way.
- The PEARL compiler implemented for this project was a prototype one used for the first time. It is therefore understandable that it is by no means perfect. It contains erros, is slow and uses much memory space. It generates too much code and the code is not as time efficient as it could be. The error messages could also be improved. However, one aim of this project has been to evaluate the compiler to provide input for the development of an efficient and reliable PEARL compiler for the NORD computers.

CONCLUSIONS

Althoug we think that many of the characteristics of the PEARL language could have been improved, we generally find that it could be a very useful language for real time purposes. This would, however, require improvements of the compiler, concerning reliability as well as efficiency and user friendliness. We also think that those parts of PEARL which are implementation dependent should be well defined to enhance the portability.

REFERENCES

1. W.E. Büttner, L Felkel, R. Grumbach,B. Thomassen, F. Øwre: "Functions and Design Characteristics of the STAR Disturbance Analysis System". Paper presented at the IAEA Specialists' Meeting on NPPCI, December 5-7, 1979, Munich, Germany.
2. Werum: PEARL Language Refernce Manual, Reg. FB 141/8008.

3. D. S. Nielsen: "The Cause-Consequence Diagram Method as a Basis for Quantitative Accident Analysis". Danish Atomic Energy Report, Risö-M-1374, May 1971.

4. EWICS TC7 Working Paper 235: "Recommentation for the Design and Coding of Safety Related Software", edited by W. Ehrenberger.

5. EWICS TC7 Working Paper 243: "Guidelines for the Documentation of Safety-Related Computer Systems. Part 3.System Description", edited by H. Ryland.

Kurzmitteilungen

PEARL-Kurse im IRT

Für die Mitarbeiter der öffentlich-rechtlichen Rundfunkanstalten (ARD und ZDF) wurden im Jahr 1981 im IRT vier Einführungskurse in PEARL durchgeführt.

Zu den insgesamt 70 Teilnehmern gehörten auch neun Mitarbeiter des IRT und zwei Mitarbeiter des österreichischen Rundfunks (ORF).

Die Dozenten sowie die schriftlichen Unterlagen für die jeweils einwöchigen Kurse stellte der PEARL-Verein.

Der Übungsbetrieb, der etwa die Hälfte des Kursumfangs ausmachte, wurde am Rechner des IRT, einer HP 3000 durchgeführt. Dafür stand für je 2 bis 3 Teilnehmer ein Terminal im Seminarraum zur Verfügung. Der Rechner wurde während dieser Zeit im üblichen Umfang von den Benutzern des IRT in Anspruch genommen.

Die Kursteilnehmer, sowohl Programmierer, wie auch Angehörige der Planungs- und Betriebsabteilungen, waren nach dem Kurs in der Lage, PEARL-Programme selbständig zu schreiben.

EINFÜHRUNG VON PEARL IN BRASILIEN AUF BASIS DER COMPILER-TECHNOLOGIE VON WERUM.

In Anwesenheit des deutschen Wissenschaftsattachés unterzeichnete die brasilianische Staatsfirma DIGIBRAS am 14.05.1982 in Brasilia den mit dem deutschen Entwicklungsbüro Wulf Werum, Lüneburg, abgeschlossenen Vertrag zur Einführung von PEARL in Brasilien auf Basis der Compiler-Technologie von WERUM.

Ziel des Vertrages ist

- die Einführung der in der Bundesrepublik entwickelten Echtzeit-Programmiersprache PEARL in Brasilien sowie
- die Übertragung der Compiler-Technologie von WERUM auf DIGIBRAS.

Zur Erreichung dieses Ziels werden zunächst sechs Mitarbeiter brasilianischer Software-Häuser und Universitäten in Lüneburg in die Handhabung, vor allem aber in die Techniken des portablen PEARL-Programmiersystems von WERUM eingeführt. Anschließend werden sie gemeinsam mit WERUM und weiteren brasilianischen Ingenieuren dieses portable PEARL-Programmiersystem in Brasilien auf drei verschiedenen brasilianischen Rechnern implementieren. Parallel dazu wird PEARL in die Lehre an brasilianischen Universitäten aufgenommen werden. Nach erfolgtem Technologie-Transfer kann DIGIBRAS die erworbenen Compiler-Techniken, -Werkzeuge und -Bausteine außerhalb Europas frei nutzen.

Das Projekt hat eine Laufzeit von 18 Monaten und erfordert auf brasilianischer und deutscher Seite Aufwendungen in Höhe von mehr als drei Millionen DM. Der BMFT unterstützt das Projekt mit ca. 25% Zuschuß im Rahmen des Abkommens über Zusammenarbeit in der wissenschaftlichen Forschung und technologischen Entwicklung in Brasilien. Auf deutscher Seite wird das Projekt durch das Internationale Büro der GMD koordiniert.

Im Anschluß an die Vertragsunterzeichnung würdigte DIGIBRAS-Präsident Cotrim vor der Presse die hiermit eingeleitete Zusammenarbeit als einen entscheidenen Schritt zur Entwicklung und Einführung von Echtzeit-Programmiersprachen und damit zur automatisierten Prozeßsteuerung in Brasilien. Als Anwendungsbereiche nannte Cotrim die Sektoren Transport, Energieerzeugung, Bergbau, Stahlindustrie und Erdölchemie.

Literatur zu PEARL

Neuerscheinung: "SYSTEMATISCHES PROGRAMMIEREN MIT PEARL"

In der Reihe Studien-Texte Informatik der Akademischen Verlagsgesellschaft Wiesbaden erscheint im Herbst dieses Jahres ein neues Buch über PEARL: "Systematisches Programmieren mit PEARL". Die Autoren sind H. Brinkkötter, K. Nagel, H. Nebel und K. Rebensberg, Mitarbeiter der Prozessrechnerverbund-Zentrale der T.U. Berlin.

Das Buch, das auf den von den selben Autoren erstellten Kursunterlagen des PEARL-Vereins aufgebaut ist, wendet sich an Ingenieure, Programmierer, Organisatoren und Studenten, die die problemorientierte und portable Echtzeitprogrammiersprache PEARL kennenlernen und anwenden wollen. Anhand ausgewählter Beispiele aus dem Bereich der Prozeßdatenverarbeitung wird der Prozeß der Programmentwicklung, die dabei verwendeten Methoden und Techniken sowie wichtige Elemente der Programmiersprache PEARL vermittelt.

Es werden folgende allgemeine Methoden und Problemstellungen vertieft:

- Vollständige Fallunterscheidung
- Interaktion mit technischen Prozessen
- Realisierung zeitabhängiger Vorgänge
- Behandlung von Meßwerten hoher Datenrate
- Organisation komplexer Datenbestände
- Konstruktion abstrakter Datentypen
- Realisierung paralleler Vorgänge
- Zeitlistengesteuerte Prozeßkontrolle
- Wechselseitiger Zugriff auf einen Datenpuffer
- Konfliktsteuerung
- Nebenläufiger Zugriff auf einen Datenbestand.

Das Buch schließt mit einer anwenderorientierten Beschreibung der Sprache PEARL.

Es werden die folgenden Methoden und Techniken und ihre Einordnung in den Problemlösungsprozess behandelt:

- Genaue Beschreibung des Problems
- Gliederung in (nebenläufige) lösbare Teilalgorithmen, statische Zerlegung des Problems in einzelne Moduln
- Konstruktion der Lösungsalgorithmen mit der Methode der schrittweisen Verfeinerung, Konstruktion problemgerechter Datenstrukturen, Plausibilitätsbetrachtungen
- Umsetzung in ablauffähige PEARL-Programme.

INHALTSVERZEICHNIS

4.3 Nebenläufiger Zugriff auf einen Datenbestand
- Verwaltung von Platzreservierungslisten

Weitere Informationen erhältlich über:
Akademische Verlagsgesellschaft
Postfach 1107
6200 Wiesbaden

WERUM/WINDAUER: INTRODUCTION TO PEARL

Von dem im Vieweg-Verlag erschienenen Buch von Wulf Werum und Hans Windauer:"PEARL-Process and Experiment Automation Realtime Language", ist Anfang des Jahres eine aktualisierte englische Übersetzung unter dem Titel "Introduction to PEARL" erschienen. In dem Buch wird der Subset von Full-PEARL beschrieben, der dem von der Firma Werum implementierten Sprachumfang entspricht.

Herausgeber: Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig ISBN 3-528-03690-0

K.W. PLESSMANN:

GRUNDSATZUNTERSUCNUNG ZUR NUTZUNG DER PROGRAMMIERSPRACHE PEARL FÜR MIKRORECHNER

In der Reihe der PDV Entwicklungsnotizen wurden jetzt vom Kernforschungszentrum Karlsruhe die Ergebnisse einer 1980 von Prof. K.W. Plessmann an der RWTH Aachen durchgeführten Untersuchung zum Einsatz von PEARL auf Mikrorechnern veröffentlicht.

In dem Bericht werden die Gründe für die Verwendung von PEARL, vor allem beim Einsatz mit Mikrorechnern dargelegt und ein "Pflichtenheft" für ein, auf einem Mikrorechner lauffähigem System erstellt. Ferner enthielt der Bericht eine Gegenüberstellung mit anderen modernen Sprachen.

Im Anhang wird ein Vorschlag für einen implementierbaren Sprachumfang auf einen Mikrorechner-Entwicklungssystem gemacht.

Erhältlich unter dem Titel:
PDV-Entwicklungsnotiz PDV-E 152 bei
Kernforschungszentrum Karlsruhe
Projekt PDV
Postfach 3640
7500 Karlsruhe 1

Veranstaltungen und Termine

14.7.1982	PEARL-Informationsveranstaltung bei der Bundesanstalt für Flugsicherung und dem Deutschen Wetterdienst. Frankfurt
6.-10.9.1982	PEARL-Kurs/VDI-Bildungswerk und PEARL-Verein Stuttgart
8.-10.9.1982	EUROGRAPHICS Manchester
14.-17.9.1982	Kongress "Kraftwerke 1982" Mannheim (VGB)
21.9.1982	Vorstellung der PEARL-Programmbibliothek des Instituts für Verfahrenstechnik und Dampfkesselwesen (Prof. Welfonder) Stuttgart-Vaihingen
21.-22.9.1982	6. Sitzung des PEARL-Anwender-Ausschusses Stuttgart-Vaihingen
6.10.1982	Workshop "PEARL in der Wehrtechnik" Koblenz
5.-7.10.1982	12. GI-Jahrestagung Kaiserslautern
5.-8.10.1982	SOCOCO '82 3rd IFAC/IFIP Symposium on Software for Computer Control Madrid
11.-14.10.1982	SAFECOMP '82 Workshop on Safety of Computer Control Lafayette, USA
19.-20.10.1982	Aussprachetag "Prozeßrechner '82" Lahnstein (GMR)
3.-5.11.1982	PEARL-Time-Data '82 Versailles
8.-12.11.1982	PEARL-Kurs VDI-Bildungswerk und PEARL-Verein München (IRT)
2.12.1982	Mitgliederversammlung des PEARL-Vereins Düsseldorf
2.-3.12.1982	PEARL-Tagung '82 "PEARL in der Praxis" Düsseldorf
9.-11.12.1982	IFAC Symposium on Components and Instruments for Distributed Control Systems Paris