PEARL in Vorlesungen und Praktika an der FH Bielefeld

Prof. Dr. L. Frevert, Bad Salzuflen

Zusammenfassung

Für die Einführung in Basis PEARL wurde ein Skript mit 51 Beispiel-Programmen verfaßt. PEARL wurde in einer Betriebssystem-Vorlesung für die Simulation von Strategien und Koordinationsmethoden verwendet, z.B. für die Simulation von Monitoren durch PEARL Module. Im Praktikum Prozeßdatenverarbeitung entwickeln die Studenten Module eines ziemlich umfangreichen Programmes. Die Modul-Schnittstellen wurden spezifiziert mit Hilfe einer neuartigen Methode zur Top-down-Spezifikation, -Entwicklung und -Dokumentation von Programmen.

Schlüsselworte: Basis PEARL, Betriebssystem, Monitor, Entwurfs-Methode, Dokumentation

Abstract

For teaching of Basis PEARL a script containing 51 example programs was written. PEARL has been used in a lecture about operating systems for simulation of monitors by PEARL moduls. Within a practical course in process control, modules of a rather large program are to be developed by students. The module interface have been specified using a novel top down method for program specification, development and documentation.

Key words: Basic PEARL, operating systems, monitors, program design, documentation

Die Fachbereiche Elektrotechnik und Maschinenbau der FH Bielefeld betreiben gemeinsam ein Prozeßrechenzentrum, das mit einem EPR 1300-System (Hauptrechner und Satellitenrechner) der Firma Krupp-Atlas-Elektronik ausgerüstet ist. Die Verfügbarkeit von PEARL hatte bei der Auswahl des Systems eine nicht unwesentliche Rolle gespielt. Es handelt sich um Basis-PEARL, das inzwischen um einige Elemente aus full PEARL erweitert worden ist.

Das größte Hindernis für die Verwendung von PEARL in der Vorlesung und im Praktikum "Prozeßdatenverarbeitung" bildete zunächst das Nichtvorhandensein eines preisgünstigen Lehrbuches über Basis-PEARL. Ich habe deshalb als erstes ein Skript verfaßt, das im Offset-Druck vervielfältigt wurde und den Studenten zum Selbstkostenpreis von DM 5.- zur Verfügung gestellt wurde. Trotz einer Gesamtauflage von 250 Exemplaren ist es inzwischen vergriffen.

Bei der Abfassung des Skriptes wurden die einzelnen Sprachelemente von PEARL grundsätzlich anhand kompletter Beispiel-Programme erläutert - insgesamt 51. Sie sind in einem Begleitheft zusammengefaßt, sodaß beschreibender Text und Programm gleichzeitig aufgeschlagen werden können. Die Programme sind so geschrieben, daß durch möglichst geringfügige Änderungen das nächste aus dem vorhergehenden abgeleitet werden kann. Das erste Beispiel liest z.B. eine Zeichenkette von der Tastatur ein und gibt sie auf dem Bildschirm aus, das zweite liest zwei Ketten ein, kateniert sie (Ausdruck), weist das Ergebnis einer Variablen zu und gibt diese aus, das dritte vergleicht die Ketten (bedingte Anweisung), das vierte Beispiel weist das Ergebnis des Vergleichs einer Bit-Variablen zu (Einführung des Datentyps BIT), usw.. Die Studenten können deshalb die Programme mit relativ wenig Aufwand selbst ausprobieren.

Da ihnen dabei in der Regel die Prozeßperipherie des Rechners nicht zur Verfügung steht, wird ihnen Parallelarbeit, Einplanungen, Synchronisation usw. nicht mit Beispielen aus der Prozeßdatenverarbeitung erläutert, sondern durch Tasks, die Personen repräsentieren, die z.B. ein Badezimmer zusammen oder nacheinander benutzen wollen.

Das Verfahren, Systeme, die Nebenläufigkeiten enthalten, mit Hilfe von PEARL-Programmen zu simulieren, läßt sich übrigens ausgezeichnet verwenden, um die Auswirkungen verschiedener Betriebssystem-Strategien zu demonstrieren. In fast jedem Lehrbuch über Betriebssysteme wird das Problem der 5 Philosophen behandelt, denen nur 5 Gabeln zur Verfügung stehen, die aber zum Essen je zwei Gabeln benötigen. Wenn sie alle gleichzeitig eine Gabel nehmen, müssen sie verhungern

- klassisches Beispiel für eine Verklemmung. Eine mögliche Abhilfe scheint darin zu bestehen, daß ein Philosoph, der essen will, entweder zwei Gabeln gleichzeitig oder garkeine nimmt. Natürlich läßt sich mit Hilfe theoretischer Überlegungen zeigen, daß es auch bei dieser Strategie vorkommen kann, daß ein Philosoph nie gleichzeitig in den Besitz beider Gabeln kommen kann; diese Überlegungen lassen sich jedoch sehr schön konkretisieren, wenn man das Verhalten der Philosophen durch ein PEARL-Programm simuliert.

Um beim Nutzen von PEARL für Betriebssystem-Vorlesungen zu bleiben: In der Literatur und in Diskussionen wird gegenüber dem Gebrauch von Semaphoren immer wieder der Einwand gemacht, daß er Programme anfällig für Verklemmungen macht, deren Möglichkeit nur schwer erkannt werden kann, und stattdessen der Gebrauch von Monitoren (wie in CONCURRENT PASCAL) oder anderer Verfahren, z.B. von Rendezvous (Ada) empfohlen.

Ich halte diese pauschale Kritik für ebensowenig gerechtfertigt, wie ein pauschales Verbot von GOTO oder von Pointern - beide Sprachelemente entsprechen sich in etwa in ihren Auswirkungen auf die Integrität von Programmen. Diese Kritiker übersehen nämlich, daß das Modul-Konzept von PEARL dazu verwendet werden kann, die Zuverlässigkeit von Programmen zu erhöhen. Man braucht sich nur an die Regel zu halten, die Koordination oder Synchronisation von Tasks in eigens geschriebene Moduln zu verlegen. Ich möchte dies an einem Beispiel verdeutlichen: beim Leser-Schreiber-Problem dürfen beliebig viele Tasks gleichzeitig aus einem Datenbereich lesen, während immer nur eine Task die Daten schreibend verändern darf, und zwar nur dann, wenn nicht gleichzeitig gelesen wird. In full PEARL könnte diese Aufgabe mit Hilfe einer Bolt-Variablen gelöst werden, die allen Tasks bekannt ist. Es läge dann in der Verantwortung der Programmierer, vor und nach jedem Lesen eine ENTER- bzw. LEAVE-Anweisung zu programmieren, und vor und nach jedem Schreiben eine RESERVE- bzw. FREE-Anweisung. Offensichtlich ist bei diesem Ansatz schwer sicherzustellen, daß diese Programmiervorschriften immer befolgt werden. Besser ist es, sowohl die Daten als auch die Bolt-Variable in einem eigenen Modul zu "verstecken" und nur über globale, reentrante Prozeduren LESEN und SCHREIBEN auf die Daten zuzugreifen. Am Anfang und Ende der Prozedur LESEN müßte ENTER DATENBOLT, bzw. LEAVE DATENBOLT stehen, und am Anfang und Ende der Prozedur SCHREIBEN RESERVE DATENBOLT, bzw. FREE DATENBOLT. Da die Programmierer der zugreifenden Tasks jetzt nur noch über die Aufrufe CALL LESEN (.....) und CALL SCHREIBEN (.....) an die Daten herankommen, ist eine völlig sichere Programmierung gewährleistet.

Bei Verwendung von Basis-PEARL läßt sich das Problem mit Hilfe zweier Semaphoren KOORDINATOR und EXKLUSIVZUGRIFF und einer Variablen ZAHLDERLESER lösen, die mit 1,1 bzw. 0 initialisiert werden müssen.

Statt ENTER muß stehen

```
            REQUEST KOORDINATOR;
              ZAHLDERLESER:= ZAHLDERLESER + 1;
              IF ZAHLDERLESER == 1 THEN
                 REQUEST EXKLUSIVZUGRIFF;
              FIN;
            RELEASE KOORDINATOR;
statt LEAVE:
            REQUEST KOORDINATOR;
              ZAHLDERLESER:= ZAHLDERLESER - 1;
              IF ZAHLDERLESER == 0 THEN
                 RELEASE EXKLUSIVZUGRIFF;
              FIN;
            RELEASE KOORDINATOR;
statt RESERVE:
            REQUEST EXKLUSIVZUGRIFF;
statt FREE:
            RELEASE EXKLUSIVZUGRIFF;
```

Das Beispiel zeigt, daß es möglich ist,
1) PEARL-Mudule zu schreiben, die den MONITORen von Concurrent PASCAL entsprechend - der Witz dabei ist, daß das Leser-Schreiber-Problem in Concurrent PASCAL nicht lösbar ist, weil sich grundsätzlich nur eine Task in einem PASCAL-Monitor aufhalten darf;
2) andere Koordinations-Methoden mit Semaphoren und Merkvariablen zu simulieren - es ist eine hübsche Übungsaufgabe, dafür zu sorgen, daß kein neuer Leser zugreifen darf, solange ein Schreiber schreiben möchte.

Eine andere hübsche Übungsaufgabe besteht darin, beliebig viele Tasks über Botschaften zu koordinieren, unter Verwendung von 3 Semaphoren.

Ich muß zugeben, daß ich bisher noch nicht sehr systematisch untersucht habe, inwieweit sich in der Betriebssystem-Literatur angegebene Koordinations-Verfahren mit PEARL simulieren und untersuchen lassen - mir scheint das jedoch ein lohnendes Feld zu sein.

Eines der Probleme beim Unterricht in Datenverarbeitung ist, die Studenten vom Nutzen der Methoden

zu überzeugen, die bei der Erstellung großer Programme unumgänglich sind. Die herkömmlichen Übungsaufgaben müssen notwendigerweise im Umfang beschränkt sein und lassen sich notfalls auch durch unsystematisches Vorgehen lösen. Ich bin deshalb dazu übergegangen, in den Praktika Aufgaben zu stellen, die Teilaspekte eines größeren Problemes lösen und aufeinander aufbauen, sodaß am Ende des Praktikums ein relativ umfangreiches Programm entstanden ist. Das erfordert eine sehr sorgfältige Aufgliederung des Gesamtproblems in Einzelteile, die vom Dozenten vorgenommen werden muß, weil den Studenten die dazu nötige Erfahrung fehlt. Wir besitzen in unserem Praktikum eine Modelleisenbahn, die vom Rechner gesteuert werden kann. Dabei wird der jeweilige Ort der Lokomotiven aufgrund von Interrupts ermittelt, die durch Gabellichtschranken erzeugt werden.

Die Gesamtaufgabe besteht darin, einen sicheren Betrieb der Anlage zu gewährleisten, den jeweiligen Anlagenzustand und die Zugorte auf einem alphanumerischen Sichtgerät darzustellen, Ereignisse, wie das Stellen von Weichen und Signalen zu protokollieren, usw.. Die Fahrwege der Züge und Aufenthaltsdauern vor Signalen sollen ohne Rücksicht auf das Vorhandensein anderer Züge festgelegt werden können. Darüber hinaus sollen alle Programme weitgehend unabhängig von der Topologie der speziellen Anlage geschrieben werden; die Topologie wird in einer Datenbank beschrieben, auf die die einzelnen Programmteile über Zugriffsprozeduren zugreifen.

Die einzelnen Teilaufgaben beginnen stets: "Für die rechnergesteuerte Modelleisenbahn ist ein PEARL-Modul zu entwickeln und zu testen, der folgende Prozeduren enthält: ....."

Das gesamte Programm besteht aus 10 Moduln, von denen die Studenten-Gruppen etwa 5 entwickeln; die übrigen sind von mir programmiert worden und werden im begleitenden Seminar besprochen, wie die Prinzipien, nach denen die Aufteilung vorgenommen wurde. Vorsichtshalber habe ich auch die anderen Moduln in Reserve, um auch den Studentengruppen ein abschließendes Erfolgserlebnis beim Integrationstest vermitteln zu können, bei denen sich der eine oder andere Modul als noch fehlerhaft erweist.
Um die Schnittstellen zwischen den Moduln festlegen zu können, wurden die wichtigsten Moduln mit einem Top-down-Verfahren in Pseudocode programmiert. Das Verfahren ist so angelegt, daß bei weitergehender Verfeinerung die Ebene von PEARL-Anweisungen erreicht wird; ein eigens geschriebener Preprozessor wandelt die so erhaltenen Programm-Entwürfe in kompilierbare Programme um. Ich möchte das Verfahren an einem Beispiel erläutern:

Einer der Moduln enthält die Tasks und Unterprogramme, die dafür sorgen, daß einerseits die Weichen und Signale für den Fahrweg eines Zuges richtig gestellt werden, andererseits die Züge aber auch nicht zusammenstoßen. Der Modul hat folgenden Aufbau:

```
/* ZENTRALE SICHERHEITSSTEUERUNG
*/ MODULE BAHNU; /*
    #1    SCHNITTSTELLEN ZU ANDEREN MODULN
          (SPEZIFIKATIONEN)
    #2    VARIABLE
          (DEKLARATIONEN)
    #3    PROZEDUREN
    #4    TASKS
*/ MODEND; /*
```

Da das Ziel der Programmierung darin besteht, die Schnittstellen zu anderen Moduln zu gewinnen, können diese noch nicht spezifiziert werden; desgleichen werden sich die benötigten Variablen und Unterprogramme erst aus dem Entwurf der Tasks ergeben. Die weitere Entwicklung beginnt deshalb mit der Verfeinerung von #4, für die die obige Zeile die Überschrift darstellt. Um Überschriften und zugehörige Verfeinerungen leicht unterscheiden zu können, werden bei ersteren die zugehörigen Kennziffern eingerückt, bei letzteren ganz an den Rand geschrieben. Die Verfeinerung von #4 lautet:

```
#4
    #41  FUER JEDE GABEL-LICHTSCHRANKE EXISTIERT
         EINE TASK, DIE IN EINER PROZEDUR BAHNUI
         EINGEPLANT WIRD, SODASS SIE DURCH DEN
         GABEL-LICHTSCHRANKEN-INTERRUPT AKTIVIERT
         WIRD.
    #42  FUER JEDEN FAHRWEG EXISTIEREN ZWEI TASKS;
         DIE EINE SORGT DAFUER, DASS DIE WEGELE-
         MENTE FUER DIE FORTSETZUNG DES FAHRWEGES
         EXKLUSIV RESERVIERT, DIE WEICHEN UND SIG-
         NALE RICHTIG GESTELLT WERDEN. DIE ANDERE
         GIBT BEREITS DURCHFAHRENE WEGELEMENTE FREI.
         BEIDE TASKS WERDEN BEI BEDARF DURCH DIE
         GABEL-LICHTSCHRANKENTASKS AKTIVIERT.
```

Hier soll uns zunächst die Verfeinerung von #4 interessieren:

```
#41
    #411 TASKS
    #412 ALLEN TASKS GEMEINSAME BEARBEITUNGSPROZE-
         DUR
```

Die Verfeinerung von ╫ 411 besteht aus PEARL-Code:

```
╫411     */ SENSORTASK1: TASK;
              CALL MELDUNGPRUEFEN(1);
            END;
            :
            SENSORTASK11: TASK;
              CALL MELDUNGPRUEFEN(11);
            END /*
```

(unsere Anlage hat insgesamt 11 Gabel-Lichtschranken)

Die Verfeinerung ╫412 muß weiter verfeinert werden:

```
╫412
        */ MELDUNGPRUEFEN: PROC(SENSORNR FIXED)
                                        REENT; /*
  ╫4121 STELLE FEST, OB AM BETREFFENDEN SENSOR
        DAS DURCHFAHREN EINES ZUGES ERWARTET
        WURDE
  ╫4122 IF MELDUNG ERWARTET
  ╫4123 THEN HANDLE ENTSPRECHEND
  ╫4124 ELSE MELDE FEHLER; FIN
        */ END; /*
```

Die Verfeinerung von ╫4121 besteht aus dem Aufruf einer Zugriffsprozedur zur zentralen Datenbank, in der Topologie und Zustand der Anlage notiert sind; aus der Pseudocode-Programmierung hier ergibt sich, daß in dieser Datenbank notiert werden muß, ob ein Zug an einer Gabel-Lichtschranke erwartet wird.

```
╫4121    */ FAHRWEGNR:=HOLFAHRWEGNR(SENSORNR); /*
╫4122    */ IF FAHRWEGNR/= 0/*
```

FAHRWEGNR ist eine lokale Variable in MEDLUNG-PRUEFEN. Der zugehörige Pseudocode läßt sich leicht an entsprechender Stelle einfügen.

```
  ╫4120 LOKALE VARIABLE
```

Der Programmentwurf läßt sich fortsetzen mit

```
╫4123    */ THEN /*
  ╫41231 VERFOLGE FAHRWEG BIS ZUM NAECHSTEN
         SENSOR ODER SIGNAL UND TRAGE DORT DIE
         FAHRWEGNR EIN
  ╫41232 IF SIGNAL
  ╫41233 THEN AKTIVIERE FORTSETZUNGSTASK; FIN
  ╫41234 AKTIVIERE FREIGABETASK
```

usw..

Wie man sieht, braucht in einem so gewonnenen vollständigen Programmentwurf nur jeweils die zugehörige Verfeinerung hinter einem Pseudocode eingeordnet zu werden, um ein PEARL-Programm zu erhalten, in dem die Pseudocodestücke als Kommentare stehen. Genau diese Umordnung der Zeilen-Reihenfolge bewerkstelligt der oben erwähnte Preprozessor, der außerdem den Entwurf auf Vollständigkeit und richtigen Gebrauch der Kommentar-Symbole untersucht. Ich habe schon erwähnt, daß die Studenten im Praktikum die Moduln des Modellbahn-Programmes einzeln entwickeln und testen, bevor sie zum Gesamtprogramm integriert werden. Dazu muß jeder Modul mit Programmteilen versehen werden, die nur zum Modultest benötigt werden - z.B. zur Simulation der Schnittstellen zu anderen Moduln. Diese Programmteile lassen sich bei Programmierung mit dem eben beschriebenen Verfahren praktisch vollautomatisch aus den fertigen Moduln entfernen: der Preprozessor übergeht auf Wunsch alle Verfeinerungen, die mit ╫ % beginnen, und ordnet sie nicht in das kompilierbare Programm ein. Diese Eigenschaft des Preprozessors läßt sich übrigens auch dafür ausnützen, top-down gegliederte Spezifikationen mit in das Entwurfsdokument zu integrieren. Auf diese Weise sind Spezifikationen, Programmentwurf und Testschnittstellen in einem einzigen Dokument enthalten. Umgekehrt werden Programmteile, die für den Test überflüssig, für das Gesamtprogramm jedoch notwendig sind (z.B. Spezifikationen von Schnittstellen zu anderen Moduln) mit ╫§ gekennzeichnet.

Seit einiger Zeit habe ich das Verfahren auch für die Studenten freigegeben. Die Erfolge sind ermutigend; die Akzeptanz ist groß, weil der zusätzliche Lern- und Schreibaufwand gering ist und die Vorteile - schnellere Programmentwicklung und übersichtlichere Dokumentation - sofort ersichtlich sind.

Anschrift des Autors

Prof.Dr.L.Frevert
Ostersiek 29, 4902 Bad Salzuflen
Telefon: 05222/10126