Lösung von Echtzeitproblemen mit PEARL90-Objekten

von L. Frevert

Einleitung und Aufgabenstellung

Anfang der siebziger Jahre gab es den Trend, anwendungsorientierte Sprachen zu entwickeln, deren Sprachelemente alle denkbaren Anwendungen innerhalb eines technischen Gebietes abdecken sollten. Erfolgreiche Programmiersprachen werden jedoch mehrere Jahrzehnten benutzt, und über derart lange Zeiten läßt sich kaum voraussagen, in welche Richtung sich ein Anwendungsgebiet entwickeln wird. Auch für PEARL sind immer wieder Spracherweiterungen vorgeschlagen worden, die seine Nutzung für Systeme mit hohen Sicherheitsanforderungen erleichtern sollten, an die damals kaum jemand dachte.

Neue objektorientierte Sprachen wie Java folgen einem anderen Paradigma: sie stellen eine Basis zur Verfügung, mit der sich Objektklassen erstellen lassen. Spezielle Anforderungen werden nicht durch Sprachelemente erfüllt, sondern durch Klassenbibliotheken.

Auch in PEARL90 kann objektorientiert programmiert werden. Es ist deshalb möglich, für ungewöhnliche Anforderungen entsprechende Objekte bzw. Klassen zu schaffen. Zum Beispiel kann das Leser-Schreiber-Problem auch mit Semaphoren und Merkvariablen gelöst werden; statt des Sprachelementes BOLT mit den zugehörigen Echtzeitanweisungen ENTER, LEAVE, RESERVE und FREE könnte man eine Klasse `Bolt` mit den Methoden `Enter`, `Leave` usw. verwenden.

Als weiteres Beispiel sei hier folgende Aufgabe genannt: Bei einer digitalen Regelung muß eine harte Echtzeitbedingung eingehalten werden. Aus Prozeßdaten, die mit festen zeitlichen Abständen aus einem technischen Prozeß eingelesen werden, sollen Ausgabedaten berechnet werden, die unbedingt innerhalb eines Zeitlimits an den Prozeß ausgegeben werden müssen. Die Berechnung ist relativ langwierig, bei kurzzeitiger starker Belastung des Rechners könnte das Zeitlimit überschritten werden. Deshalb soll seine Einhaltung überwacht werden; bei drohender Überschreitung sollen Ergebnisse eines ungenaueren, weniger zeitaufwendigen Verfahrens ausgegeben werden. Falls im Extremfall die Rechnerleistung auch dazu nicht ausreicht, soll eine Ausnahme (Exception) signalisiert werden, auf die dann mit einer Ausnahmebehandlung reagiert werden kann.

Bei Betrachtung der Regelung als zyklisch ausgeführte Sequenz Einlesen, Rechnen, Ausgeben scheint das Problem mit den jetzigen Sprachmitteln von PEARL nicht lösbar zu sein. Bei objektorientierter Programmierung läßt sich jedoch relativ leicht eine Lösung finden: es brauchen nur zwei Objekte `regelung` und `waechter` geeignet zusammenzuarbeiten.

Klassen, Objekte und Methoden

Bei Anwendung des Paradigmas "Objektorientierte Programmierung" bestehen auch PEARL-Programme aus einem Netz von Objekten. Ein Objekt umfaßt Daten und Methoden, die auf diese Daten zugreifen. Außerdem kann es auch Tasks, Semaphore, Bolts usw. enthalten, wie das Programmbeispiel zeigen wird. Die Vernetzung entsteht dadurch, daß jedes Objekt Referenzen auf diejenigen Objekte enthält, mit denen es in direkter Beziehung steht.

Jedes Objekt gehört einer Klasse an, gleichartige Objekte der selben Klasse. Als Begriffe der Sprache gibt es in PEARL zwar keine Klassen, Objekte und Methoden. Klassen können jedoch durch Vereinbarungen von Verbund-Datentypen nachgebildet werden, Objekte als Deklarationen über diesen Typen. Einer Methode eines Objektes entspricht eine Prozedur, deren erster Parameter das jeweilige Objekt ist; auf diese Weise können alle Objekte einer Klasse für die selbe Methode die selbe PEARL-Prozedur benutzen.

Das erwähnte Objekt `regelung` gehöre der Klasse `REGELUNGEN` an; jedes Objekt dieser Klasse enthält unter anderem einen Verbund-Datentyp `DATEN` als Datenbasis, eine Task, die zyklisch gestartet wird, die zugehörige Zykluszeit sowie das Zeitlimit für die Durchführung der Regelung. Die entsprechende Typvereinbarung (Beispiel 1) enthält als Komponenten außer der Datenbasis Referenzen auf diejenigen Bestandteile eines Objektes, die nicht direkt in der Typvereinbarung stehen dürfen, z. B. die Task und die Methoden der Klasse. Außerdem gibt es noch eine Referenz auf ein Objekt der Klasse `UEBERWACHER`, mit dem die Regelung zusammenarbeiten soll.

```
  TYPE REGELUNGEN STRUCT[
     datenbasis DATEN,
     zugriffskoord INV REF BOLT,
     (zykluszeit,
      zeitlimit) DURATION,
     zyklustask INV REF TASK,
     (start,
     regeln,
     genau,
     kurz) INV REF PROC(INV REF REGELUNGEN),
     zeitwaechter REF UEBERWACHER];
```

Beispiel 1: Typvereinbarung zur Klasse `REGELUNGEN`

```
regelung_task:TASK;
  CALL regelung.regeln(regelung);
END;
DCL regelung_zugriffskoord BOLT;
DCL regelung REGELUNGEN INIT(
     ............,                ! Anfangswerte der Datenbasis
     regelung_zugriffskoord,      ! für Koordination der Datenbasis-Zugriffe
     10 SEC,                      ! Zykluszeit
     5 SEC,                       ! Zeitlimit
     regelung_task,               ! zyklisch gestartete Regeltask
     REGELUNGEN_start,            ! Start-Methode der Klasse REGELUNGEN
     REGELUNGEN_regeln,           ! wird indirekt von der Regeltask aufgerufen
     REGELUNGEN_genau,            ! zeitaufwendige genaue Version der Regelung
     REGELUNGEN_kurz,             ! abgekürzte Version der Regelung
     waechter);                   ! Überwacher für Einhaltung des Zeitlimits
```

Beispiel 2: Deklaration eines Objektes zur Klasse REGELUNGEN

Die Deklaration des Objektes `regelung` besteht aus drei Teilen: der Deklaration der zyklisch zu startenden Task, der Deklaration eines Bolts und der Deklaration der eigentliche Regelung mit einer INIT-Liste. Sie lautet (Beispiel 2):

Diejenigen Elemente der INIT-Liste, die hier ohne Interesse sind, sind durch Punktfolgen angedeutet. Die Bezeichner `regelung_task` und `regelung_zugriffskoord` sind syntaktisch gesehen Konstanten-Bezeichner; deshalb müssen sie vor der INIT-Liste stehen. Das gleiche gilt auch für die Prozeduren-Bezeichner `REGELUNGEN_start` und `REGELUNGEN_regeln`, die im Programm am besten gleich hinter der Typvereinbarung für `REGELUNGEN` stehen.

```
REGELUNGEN_start:PROC (dieseregelung INV REF REGELUNGEN);
   ALL dieseregelung.zykluszeit ACIVATE dieseregelung.task;
END;
```

Beispiel 3: Methode zur Einplanung einer Regelungs-Task

Die Methode `REGELUNGEN_start` ist leicht zu programmieren (Beispiel 3). Schwieriger ist es mit der Methode `REGELUNGEN_regeln`; eigentlich handelt es sich dabei um zwei Methoden `REGELUNGEN_genau` und `REGELUNGEN_kurz`, wobei beide der Zeitüberwachung durch das Objekt `waechter` unterliegen sollen. Dieses Objekt soll `REGELUNGEN_genau` bei Überschreitung des Zeitlimits abbrechen. Da sich Prozeduren in PEARL90 nicht an beliebiger Stelle von außerhalb beenden lassen, wohl aber Tasks, muß `REGELUNGEN_genau` in eine Task eingebettet werden; gleiches gilt für `REGELUNGEN_kurz`. Die Zeitüberwachung selbst erfordert eine dritte Task.

Das Objekt zur Zeitüberwachung, seine Tasks und Methoden

Das Objekt `waechter` muß deshalb drei Tasks enthalten. Ihm müssen das Zeitlimit, die beiden Methoden-Prozeduren sowie deren erste Parameter (hier das Objekt `regelung` als Auftraggeber für die Überwachung) bekannt sein. Sie können ihm beim Aufruf einer entsprechenden Methode vom Auftraggeber als Argumente übergeben werden.

Die beiden Methoden `REGELUNGEN_genau` und `REGELUNGEN_kurz` werden so durch zwei Tasks ausgeführt, haben aber die selbe Datenbasis. Um Inkonsistenzen in dieser Datenbasis zu vermeiden, müssen die beiden Methoden-Prozeduren bei Beginn ihrer Arbeit die benötigten Daten in lokale Variable kopieren und später die errechneten Daten in die Datenbasis zurückschreiben. Diese Lese- und Schreibvorgänge müssen innerhalb des Objektes `regelung` durch entsprechende Anweisungen für einen Bolt `zugriffskoord` koordiniert werden.

Das Objekt `waechter` muß eventuell eine der Tasks oder beide terminieren. Um eine Verklemmung zu vermeiden, darf es das aber nicht, solange die Tasks sich zwischen ENTER und LEAVE oder RESERVE und FREE befinden; deshalb muß vor einer TERMINATE-Anweisung eine RESERVE-Anweisung für den Bolt ausgeführt werden, dessen Name dem `waechter` ebenfalls bei der Beauftragung übergeben werden muß.

Beispiel 4 zeigt die Typvereinbarung für die Klasse `UEBERWACHER`. Der Typ des Auftraggebers ist wegen `STRUCT[]` nicht festgelegt; ein Objekt der Klasse `UEBERWACHER` kann deshalb für beliebige Auftraggeber tätig werden; es ist polymorph.

```
TYPE UEBERWACHER STRUCT[
     status FIXED,
     (langtask,
      kurztask,
      wachetask) INV REF TASK,
      synchronisierung INV REF SEMA,
      (langproc,
       kurzproc) REF PROC(INV REF STRUCT[]),
      auftraggeber REF STRUCT[],
      auftraggeberbolt REF BOLT,
      zeitlimit DUR,
      beauftragen INV REF PROC(
                INV REF UEBERWACHER,
                langproc INV REF PROC(INV REF STRUCT[]),
                kurzproc INV REF PROC(INV REF STRUCT[]),
                auftraggeber INV REF STRUCT[],
                zeitlimit DURATION,
                koordinator REF BOLT),
      (langearbeit,
      kurzearbeit,
      wachprozedur) INV REF PROC(INV REF UEBERWACHER)];
```

Beispiel 4: Typvereinbarung zur Klasse UEBERWACHER

```
waechter_langtask:TASK;
   CALL waechter.langearbeit(waechter);
END;
waechter_kurztask:TASK;
   CALL waechter.kurzearbeit(waechter);
END;
waechter_wachetask:TASK;
   CALL waechter.wachprozedur(waechter);
END;
DCL waechter_synchronisierung SEMA;
DCL waechter UEBERWACHER INIT(
               NICHTFERTIG,
               waechter_langtask,
               waechter_kurztask,
               waechter_wachetask,
               waechter_synchronisierung,
               NIL,NIL,NIL,NIL,0 SEC, ! bekommen Inhalte bei Beauftragung
               UEBERWACHER_beauftragen,
               UEBERWACHER_langearbeit,
               UEBERWACHER_kurzearbeit,
               UEBERWACHER_wachprozedur);
```

Beispiel 5: Vereinbarung eines Objektes zur Klasse UEBERWACHER

Der `waechter` und seine Tasks (Beispiel 5) können dann auch deklariert werden (die Deklarationen müssen im Programm vor denjenigen für `regelung` stehen).

Jetzt fehlen noch die Methoden der Klasse `UEBERWACHER`. Die Methode `beauftragen` (Beispiel 6) wird im Beispielprogramm durch die Methode `regeln` der Regelung aufgerufen (Beispiel 10). Dabei übernimmt ein Überwacher zunächst unter anderem Referenzen auf die beiden Prozeduren, deren Ausführungszeiten er überwachen soll, und auf den Auftraggeber. Dann startet er drei Tasks, die dritte mit Verzögerung, und wartet mittels einer REQUEST-Anweisung. Wenn die zugehörige RELEASE-Anweisung ihn zum Weiterlaufen veranlaßt, induziert er das SIGNAL `zeitueberschreitung`, falls der `status` nicht inzwischen durch eine der Tasks geändert worden ist

```
UEBERWACHER_beauftragen:PROC(
                  ueberwacher INV REF UEBERWACHER,
                 (langproc,
                  kurzproc) INV REF PROC(INV REF STRUCT[]),
                  auftraggeber INV REF STRUCT[],
                  zeitlimit DURATION,
                  auftraggeberbolt INV REF BOLT);
    ueberwacher.langproc:=langproc;
    ueberwacher.kurzproc:=kurzproc;
    ueberwacher.auftraggeber:=CONT auftraggeber;
    ueberwacher.zeitlimit:=zeitlimit;
    ueberwacher.auftraggeberbolt:=auftraggeberbolt;
    ueberwacher.status:=NICHTFERTIG;
    ACTIVATE ueberwacher.kurztask PRIO PRIO-5;
    ACTIVATE ueberwacher.langtask PRIO PRIO;
    AFTER ueberwacher.zeitlimit ACTIVATE ueberwacher.wachetask PRIO PRIO-8;
    REQUEST ueberwacher.synchronisierung;
    IF ueberwacher.status == NICHTFERTIG THEN
       INDUCE zeitueberschreitung;
    FIN;
END;
```

Beispiel 6: Methode zur Beauftragung einer Überwachung

```
UEBERWACHER_langearbeit:PROC(ueberwacher INV REF UEBERWACHER);
  CALL ueberwacher.langproc(ueberwacher.auftraggeber);
  RESERVE ueberwacher.auftraggeberbolt;
    ueberwacher.status:=LANGFERTIG;
    PREVENT ueberwacher.wachetask;    ! damit sie nicht anläuft
    TERMINATE ueberwacher.wachetask;  ! falls sie schon arbeitet
    TERMINATE ueberwacher.kurztask;   ! sie könnte unerwartet noch laufen
    RELEASE ueberwacher.synchronisierung; ! damit Auftraggeber weitermacht
  FREE ueberwacher.auftraggeberbolt;
END;
```

Beispiel 7: Methode zur Ausführung des Auftrages zur ausführlichen Berechnung

Die Task `langtask` erhält die Priorität derjenigen Task, die den Überwacher beauftragt hat. Sie ruft die Prozedur `UEBERWACHER_langearbeit` (Beispiel 7) unter Benutzung einer Referenz auf (s. Beispiel 5). Die Prozedur führt den Auftrag zur ausführlichen Berechnung aus. Dann notiert sie deren Beendigung im `status` und sorgt dafür, daß die beiden anderen Tasks nicht anlaufen bzw. beendet werden; endlich sorgt sie dafür, daß die beauftragende Task, die auf der REQUEST-Anweisung wartet, weiterläuft.

Die Task `kurztask` erhält höhere Priorität, damit sie bevorzugt abgearbeitet wird. Sie ruft per Referenz die Prozedur `UEBERWACHER_kurzearbeit` auf (Beispiel 8). Diese Prozedur führt die ungenaue, aber schnelle Berechnung durch und notiert nur, daß sie fertig ist.

```
UEBERWACHER_kurzearbeit:PROC(ueberwacher INV REF UEBERWACHER);
  CALL ueberwacher.kurzproc(ueberwacher.auftraggeber);
  RESERVE ueberwacher.auftraggeberbolt;
    ueberwacher.status:=KURZFERTIG;
  FREE ueberwacher.auftraggeberbolt;
END;
```

Beispiel 8: Methode zur Ausführung des Auftrages zur schnellen Berechnung

```
UEBERWACHER_wachprozedur:PROC(ueberwacher INV REF UEBERWACHER);
  RESERVE ueberwacher.auftraggeberbolt;
    TERMINATE ueberwacher.kurztask;
    IF NOT ueberwacher.status == LANGFERTIG THEN
      TERMINATE ueberwacher.langtask;
      RELEASE ueberwacher.synchronisierung;
    FIN;
  FREE ueberwacher.auftraggeberbolt;
END;
```

Beispiel 9: Methode zur Zeitüberwachung

Die Task `wachetask` erhält noch höhere Priorität. Sie wird erst bei Ablauf des Zeitlimits gestartet und ruft per Referenz die Prozedur `UEBERWACHER_wachprozedur` auf (Beispiel 9). Da die Prozedur - wenn überhaupt - nach Ende des Zeitlimits ausgeführt wird, ist ihre Hauptaufgabe, die beiden anderen Tasks zu terminieren, falls sie noch laufen (falls nicht, schlagen die TERMINATE-Anweisungen ins Leere). Da die beiden Tasks jedoch möglicherweise gerade den Bolt reserviert haben, darf das Terminieren nur geschehen, wenn der Bolt wieder freigegeben ist. Deshalb enthält die Prozedur ebenfalls eine RESERVE-Anweisung. Falls die `langtask` noch nicht fertig gewesen sein sollte, muß außerdem dafür gesorgt werden, daß die beauftragende Task weiterläuft, die in der Methode `beauftragen` wartet; deshalb die RELEASE-Anweisung.

Der Vollständigkeit halber soll auch die Methode `regeln` der Klasse `REGELUNGEN` skizziert werden (Beispiel 10). Die für die Regelung notwendigen beiden Versionen der Berechnung `genau` und `kurz` sind hier nicht von Interesse.

```
REGELUNGEN_regeln:PROC(dieseregelung INV REF REGELUNGEN);
  ON zeitueberschreitung: /* z. B. Interrupt ausloesen */;
  /* Prozeßdaten einlesen */
  dieseregelung.zeitwaechter.beauftragen
                         (dieseregelung.zeitwaechter,
                         dieseregelung.genau,
                         dieseregelung.kurz,
                         dieseregelung,
                         dieseregelung.zeitlimit,
                         dieseregelung.zugriffskoord);
  /* Berechnete Daten ausgeben */
END;
```

Beispiel 10: Methode für die zyklische Durchführung der Regelung

Abschließende Würdigung

Es hat selbstverständlich auch bei Basis-PEARL schon die Möglichkeit gegeben, die oben gestellte Aufgabe zu lösen. Da man damals aber keine Referenzen auf Tasks, Prozeduren usw. benutzen konnte, wäre die Lösung viel unübersichtlicher geworden. Sie hätte das hier gegebenen Grundprinzip benutzt, und die vier Tasks hätten Körper besessen, die den oben gegebenen Methoden-Prozeduren ähnlich gewesen wären. Die Bezeichner der Tasks, Bolts und Semaphore hätten aber direkt in den Tasks verwendet werden müssen. Deshalb hätten sich die technischen Funktionen Regelung und Überwachung in der Software nicht sauber trennen lassen.

Der Klasse `UEBERWACHER` entspricht die Typvereinbarung (Beispiel 4) und die in den Beispielen 6 bis 9 beschriebenen zugehörigen Prozeduren. Wenn man jetzt noch ein neues Objekt `zweiterwaechter` dieser Klasse schaffen möchte, könnte man die in Beispiel 5 gegebenen Deklarationen für `waechter` kopieren und brauchte dann nur überall `waechter` durch `zweiterwaechter` zu ersetzen. Diese Aufgabe könnte durch einen Makrogenerator wesentlich erleichtert werden. Darüber hinaus könnte man die Begriffe "Klasse" und "Objekt" in PEARL-Programme einführen, ohne die vorhandenen PEARL90-Kompilierer zu ändern, indem man diese zusätzlichen "Sprachelemente" mit Hilfe eines Preprozessors realisiert.

Durch den hier vorgestellten Programmierstil ist es möglich, für die Lösung von Echtzeitproblemen ausgetestete wiederverwendbare Klassen zu schaffen. Dadurch können auch hohe Sicherheitsanforderungen leichter erfüllt werden. Die Verwendung von Referenzen dürfte dabei kein Sicherheitsrisiko bilden, weil die wenigen, die in den Beispielen nicht invariant sind, notfalls ebenfalls invariant gemacht werden können.