Pascal Plus

Another Language for Modular Multiprogramming

J. Welsh and D.Bustard

von

Rudolf Postel
0202907
E533

Pascal Plus
Another Language for Modular Multiprogramming

Kurzfassung

Bei Pascal Plus handelt es sich um eine erweiterte Version von Pascal, die die Qualitäten zweier Programmiersprachen in sich vereint:
Von SIMONE (sehr effektiv für getrennte Fallsimulation) und von „model operating system“ (PPP) (verwendet für parallele Prozesse *monitors* und *condition queues*) miteinander vereint; diese können entweder im Simulations- oder im Echtzeitmodus implementiert werden. Der Simulationsmodus dient dazu Programme, die später in Echtzeit laufen sollen zu testen, und logische Fehler zu erkennen.
Außerdem verfügt Pascal Plus über das *envelope*-Konstrukt, das einen sicheren Ausdruck von Interfaces zwischen Modulen von aufeinanderfolgenden oder parallelen Programmen erlaubt.

Geschichtlicher Einstieg

Pascal Plus wurde von Jim Welsh und D. Bustard entwickelt, und zum ersten Mal auf einem ICL 1900 Series Computer implementiert und wurde jahrelang von der Queen´s University of Belfast verwendet. Später wurde noch eine portable Lösung von Pascal Plus entwickelt.

Syntax und Semantik

Die wesentlichen Konstrukte von Pascal Plus sind:

1) *envelope*
2) *process, monitor, condition*

1) envelope definiert:
- eine Datenstruktur,
- die Operationen, die mit der Datenstruktur verwendet werden können,
- eine Kontrollstruktur – diese Kontrollstruktur kann verwendet werden, um Initialisierung und Finalisierung zu gewährleisten.

```
envelope histogram;
    const N = 10;
    type range = 0...N;
    var count : array [range] of integer;
      i: range;
    procedure *record (value: range);
      begin count [value] := count [value] + 1 end;
    begin
      for i := 0 to N do count [i] : = 0;
      ***;
      writeln (´value occurences´);
      for i : = 0 to N do writeln (i: 4, count [i]: 12)
    end;
```

Dieses Beispiel dient dazu, den Aufbau eines *envelopes* zu illustrieren:
Zuerst wird count initialisiert, danach wird das innere Statement „*******“ ausgeführt, und am Schluss wird das Statement, um count auszugeben ausgeführt.

Das innere Statement führt Block B aus, in der die Instanz H deklariert ist:
instance *H: histogram;* oder *instance H1, H2: histogram;*
H.record(j) oder
with *H* **do begin** *…; record(j); …; record(k);…* **end**
Wenn mehr als eine *envelope*-Instanz in einem Block deklariert sind, ist die Ordnung durch die Ordnung der Deklarationen bestimmt: Eine Instanz umschließt immer die nachkommende.
Wenn aber nur eine *envelope*-Instanz erforderlich ist, wird die Definition und Deklaration kombiniert zu einem Modul (*envelope module*).
Das *envelope*-Modul stellt ein einfaches Mittel dar, um ein Programm in modulare Blöcke aufzuteilen.
Jeder Bezeichner, der in einem *envelope* definiert ist, soll starred (beinhaltet oder erwartet eine Adresse, nicht aber einen Wert) sein, nicht nur seine Prozeduren oder Funktionen. Zugang zu einer starred-Variablen ist nur lesbar, d.h. der Block, der eine *envelope*-Instanz schafft soll nur Zugang haben, aber der starred-Variablen nicht einen Wert zuordnen. Er darf den globalen Variablen aber nicht einen Wert zuordnen. Diese Zugriffsberechtigungen, bzw. –einschränkungen sind gleich bleibend in den meisten modularen Programmen, und werden erreicht mit minimaler Verzerrung der existierenden Bereichsrichtlinien von Pascal.
Das reine Leserecht ist auferlegt auf nicht-lokale Variablen innerhalb von *envelopes*, was dazu führt, dass der Gebrauch von „import“- oder „use“-Deklarationen erheblich reduziert wird, was ja in anderen Sprachen unbedingt benötigt wird.
Die verfahrenähnliche Syntax für eine *envelope*-Definition wurde gewählt, um Parameterisation zu ermöglichen. Jede Definition hat eine formale Parameterliste, die entsprechende Parameter bei der Deklaration von jeder Instanz des *envelope* liefert.
Der Block, der eine Instanz eines *envelopes* deklariert und verwendet, hat -und braucht auch- kein Wissen weder über deren Darstellung, noch über die Initialisierung und Finalisierung, die die Instanz benötigt. Er manipuliert die Instanz im Sinne der abstrakten Eigenschaften, die durch die starred-Bezeichner dargebracht werden.
Das *envelope* könnte man vergleichen mit dem class-Feature von Concurrent Pascal oder mit dem module in Euclid. Das envelope-Modul ist vergleichbar mit dem Modul in Modula. Jedoch sind seine Vorteile gegenüber diesen:

- Die Notationen brauchen ein Minimum von den Notationen und Konzepten von Pascal.
- Die automatische Initialisierung und Finalisierung entlasten den Benutzer-Block von seiner Verantwortung, die andere Einkapselungsmechanismen benötigen.

2) process, monitor, condition

Die Fähigkeiten von parallelem Programmieren in Pascal Plus basieren auf denen beschrieben von Charles Hoare 1974; *process*, *monitor* und *condition*.
Ein Prozess ist definiert als ein Block mit einem Kopf der, wie der *envelope*-Kopf, Parameter enthält.
Instanzen eines Prozess werden genauso deklariert, wie Instanzen eines *envelope*. Wenn nur eine Instanz benötigt wird, werden die Definition und die Deklaration vereint zu einem Prozessmodul.

Betrachten wir zum Beispiel das folgende Programm:

```
program producers and consumers;
    type itemtype = …;
    … module Buffer … {see later} …;
    process producer;
      var i: itemtype;
      begin
          repeat
            produce item i;
            Buffer.put (i)
          until switch off
      end;
    process consumer;
      var i: itemtype;
      begin
          repeat
            Buffer.get (i);
            consume item i
          until switch off
      end;
```

Hier gehen wir von 3 Produzenten und 4 Konsumenten aus. Produzenten und Konsumenten „teilen“ sich einen gemeinsam Puffer. Man kann deshalb eine Instanz producer schaffen, weil man voraussetzt, dass das Handeln jedes Produzenten gleich ist (analoges gilt für Instanz consumers):

```
instance
    P: array [1…3] of producer;
    C: array [1…4] of consumer;
begin
    ***
end.
```

Der Körper des Programms hat exakt dieselbe Gestalt, wie die eines *envelope*, aber hier stellt das innere Statement die Ausführung der Prozesse des Programms dar. Einmal aktiviert, laufen die Prozesse parallel, bis sie beendet werden.
Das Puffer-Modul enthält notwendigerweise die Prozeduren get und put. Es darf immer nur ein Prozess den Puffer verändern, d.h. der Puffer muss durch „gegenseitigen Ausschluss“ geschützt werden Deshalb schafft man einen *monitor*, der garantiert, dass nur ein Prozess zur selben Zeit seinen Code ausführen kann und damit seine Lokaldaten verändert.

```
monitor module Buffer;
    var item: array [0...99] of itemtype;
      putcount, getcount: integer;
    instance notfull, notempty: condition;
    procedure *put (i: itemtype);
      begin
          if putcount-getcount = 100 then notfull.wait;
          item [putcount mod 100] : = i;
```

```
            putcount : = putcount + 1;
            notempty.signal
        end;
    procedure *get (var i: itemtype);
        begin
            if putcount-getcount = 0 then notempty.wait;
            i : = item [getcount  mod 100];
            getcount : = getcount + 1;
            notfull.signal
        end;
begin putcount : = 0; getcount : = 0; *** end;
```

Wie dieses Beispiel zeigt, muss ein Prozess (z.B. ein Produzent, der den Puffer voll auffindet) warten, bis es durch eine Aktion eines anderen Prozesses, wieder möglich ist, fortzufahren (z.B. wenn ein Konsument wieder einen Teil vom Puffer wegnimmt).
Das ist der Grund dafür, warum eine Warteschleife benötigt wird, in der Prozesse warten, bis sie fortgesetzt werden können. Im Beispiel sind 2 solche *condition*-Warteschleifen deklariert.
Die Operationen signal und wait geben dem Prozess bescheid, ob er ausgeführt (bzw. fortgesetzt) werden kann, oder warten muss. Hier gilt, dass derjenige, der zuerst kommt, auch zuerst wieder fortgesetzt wird.
Diese Priorität kann man jedoch mittels pwait selber festlegen, um so mehr Kontrolle über das Programm zu erhalten. Dies geschieht mittels einem Integerwert, den man pwait zuweist. Je geringer der Integerwert, desto höher ist die Priorität des Prozesses, und desto kürzer ist die Wartezeit!
Eine Pascal Plus *condition* besteht aus einer Instanz eines eingebauten *monitors* mit dem folgenden Interface:

```
monitor condition;
    type range = 0…maxint;
    procedure *pwait (p: range);
    procedure *wait;
    procedure *signal;
    function *length: range;
    function *priority: range;
end
```

Eine wait Operation in einer leeren Warteschleife entspricht der delay Operation, die von Concurrent Pascal für Einzelprozess-Warteschleifen bereitgestellt wird; d.h. Pascal Plus Warteschleifen sind nicht weniger effektiv, wie die von Concurrent Pascal, falls sie in der gleichen Weise verwendet werden.
Prozesse und *monitors* sind die grundlegenden Strukturierungswerkzeuge für Programme, die Parallelismus enthalten.
Die zusätzliche Interface-Sicherheit, die *envelope* ermöglicht, kann man gut anhand von *monitors* zeigen (gehen wir von der Festlegung einer Einzelressource zwischen konkurrierenden Prozessen aus): Eine Instanz dieses *monitors* ermöglicht die Festlegung einer Einzelressource, aber das korrekte Verhalten der Benutzerprozesse bleibt in der Hand des Programmierers.
In Pascal Plus kann dieses korrekte Verhalten durch Einbettung der Festlegungsinstanz in ein *envelope* Interface garantiert werden. Diese Verwendung

von *envelope* stellt ein sicheres Interface für jeden Prozess bereit, der eine gemeinsam verwendete Ressource verwendet, was eine Standardprogrammier-technik in Pascal Plus darstellt.

Simulation

Die Simulationsfähigkeiten in die Sprache einzubauen, die durch den Vorgänger SIMONE bereitgestellt wurden, war eine der Designaufgaben von Pascal Plus. Dies wurde auch erreicht durch die Bestimmung eines Standardmonitormodul, welches das folgende Interface besitzt:

```
monitor module simulation;
        type timerange = 0…maxint;
        var *pseudotime: timerange;
        procedure *hold (t: timerange);
begin
        pseudotime : = 0; ***
end
```

Verknüpft mit simulation stellt *monitor* eine geordnete Warteschleife dar, die auch als Zeitwartschleife bezeichnet wird, auf der Prozesse sich selbst für eine Pseudozeit, mit Hilfe von hold, ausschalten. Die Aufwachzeit für einen Prozess ist die Summe der bestimmten Verzögerungszeit T und dem Wert der Pseudozeit, als das der Prozess die Prozedur hold aufgerufen hat.
Die Effektivität dieser Besonderheiten für getrennte Fallsimulation wurde durch Experimente mit SIMONE gezeigt. In Pascal Plus ist es nicht auf die Simulation von abstrakten Modellen der wirklichen Welt begrenzt. Die Bereitstellung von Simulationsfähigkeiten in einer Sprache verwendbar für Echtzeitsystem-programmierung führt dazu, dass man die Programme, die schließlich für eine Echtzeitverwendung eingesetzt werden sollen, testen und bewerten kann.
Damit kann man die logische Korrektheit vom System testen, und sein wahrscheinliches Echtzeitverhalten untersuchen, ohne sofortige Verbindung mit den verwendeten Echtzeitvorrichtungen zu haben. Die Leichtigkeit mit der dies erreicht werden kann, kommt von der Integration von einer Simulationsfähigkeit, mit der Prozess- und Monitorkostrukts für Echtzeitverwendung bereitgestellt werden.

Echtzeit-Features

Nachdem ein Pascal Plus Programm durch Simulation getestet wurde, kann man mit einer Echtzeit-Vorrichtung verbinden, indem man einfach den Simulationscode mit dem Code ersetzt, um die Vorrichtung bzw. das Laufwerk zu betreiben.

Zusammenfassung:

Pascal Plus stellt eine Erweiterung der von Nikolaus Wirth geschaffenen Programmiersprache Pascal dar. Die signifikanteste Änderung die von den Entwicklern vorgenommen wurde war die Einführung des *envelope*-Konstrunkts, welches einen sicheren Ausdruck von Interfaces zwischen Modulen von aufeinanderfolgenden oder parallelen Programmen erlaubt. Das parallele Programmieren wird durch die Konstrukte *monitor*, *process* und auch durch die *condition*-Warteschleifen erreicht. Außerdem kann Pascal Plus sowohl im Echtzeit-, als auch im Simulationsmodus betrieben werden, was dem Programmierer bzw. dem Entwickler erlaubt, das erstellte Programm ausgiebig zu testen, bevor es auf

Echtzeitvorrichtungen betrieben wird. Ein großer Vorteil liegt auch darin, dass es sehr rasch möglich ist, von der Simulation in die Echtzeit zu wechseln.

Quellenverzeichnis:

J. Welsh, D. Bustard „Pascal Plus – Another Language for Modular Multiprogramming“, Software-Practice and Experience, 9, 947-957, 1979

W.H.Kaubisch, R.H.Perrot and C.A.R.Hoare, “Quasiparallel programming”, Software – Practice and Experience, 6, 341-356, 1976