7.4 Synchronisation durch Tupelräume

Die bisherigen Synchronisationsmechanismen waren Client/Server-orientiert, d. h. es wurden typischerweise zwei Partner/Prozesse synchronisiert. Ein alternativer Ansatz ist das Konzept **Linda** (Carriero and Gelernter, 1989): Hier erfolgt die Kommunikation durch Austausch von Tupeln. Das grundlegende Modell ist dabei wie folgt charakterisiert:

- Es existiert ein zentraler Speicher: der **Tupelraum**, der als Multimenge von Tupeln aufgefasst werden kann.
- Es existieren viele unabhängige Prozesse, die Tupel in den Tupelraum einfügen oder auslesen.
- Das Linda-Konzept ist sprachunabhängig: Linda ist ein Kommunikationsmodell, das zu verschiedenen Sprechen hinzugefügt werden kann. So gibt es C-Linda, Fortran-Linda, Scheme-Linda, Java-Linda (Java Spaces / Jini), ...

Linda basiert auf folgenden Grundoperationen:

`out(t):` Füge ein Tupel `t` in den Tupelraum ein, wobei ein Tupel eine geordnete Liste von Elementen ist. Vor dem Einfügen werden die einzelnen Elemente entsprechend ihrer Bedeutung in der jeweiligen Programmiersprache ausgewertet.

Beispiel: Durch Abarbeitung von

```
out("hallo",1+3,6.0/4.0)
```

wird das Tupel `("hallo",4,1.5)` in den Tupelraum eingefügt.

`in(t)`: Entferne Tupel `t` aus dem Tupelraum, falls es vorhanden ist; sonst warte, bis `t` in den Tupelraum eingefügt wird.

`t` kann auch ein Muster sein, wobei dann einige Argumente Variablen sind. In diesem Fall wird ein passendes Tupel gesucht und die Variablen entsprechend den Werten im Tupel instantiiert.

Beispiel in C-Linda: Durch die Abarbeitung von

```
in("hallo",?i,?f)
```

wird das oben eingefügte Tupel aus dem Tupelraum entfernt. Dabei werden die Variablen `i=4` und `f=1.5` gesetzt.

`rd(t)`: wie `in(t)`, aber ohne Entfernung des Tupels.

`eval(t)`: Erzeuge einen neuen Prozess, der dann `t` (nach Auswertung) in den Tupelraum einfügt. Der Unterschied zu `out(t)` ist also, dass das auszuwertende Argument auch eine Funktion sein kann, die als Ergebniswert ein Tupel hat.

`inp(t), rdp(t)` (manchmal vorhanden): wie `in(t)` oder `rd(t)`, aber ohne Blockade, falls das Tupel nicht vorhanden ist (sondern z. B. `false` als Ergebnis).

Beispiel 7.10 (Linda)**.** Wir beschreiben in C-Linda zwei Prozesse, die abwechselnd aktiv werden („Ping-Pong-Spieler“):

```
ping(...)
{ while(...)
  { out("ping");  /* spiele ping */
    in("pong");   /* warte auf pong */
  }
}

pong(...)
{ while(...)
  { in("ping");
    out("pong");
  }
}
```

Das Hauptprogramm besteht dann aus:

```
eval(ping(...)); eval(pong(...));
```

Auch Datenstrukturen sind durch Tupel darstellbar: wir können einen n-elementigen Vektor $V = (e_1, \ldots, e_n)$ als n-Tupel darstellen:

("V",1,e_1)
⋮
("V",n,e_n)

Der Vorteil dieser Darstellung ist, dass ein direkter Zugriff auf einzelne Komponenten möglich ist, z. B. bei der Multiplikation der 3. Komponente mit der Zahl 3:

```
in("V",3,?e);
out("V",3,3*e);
```

Als Anwendung dieser Technik können komplexe Datenstrukturen z.B. auf verschiedene Berechnungseinheiten verteilt und dann parallel bearbeitet werden.

Beispiel 7.11 (Parallele Berechnung)**.** Wir starten eines Rechenprozess für jede Komponente:

```
/* ein Prozess fuer jede Komponente */
for (i=1; i<=n; i++) eval(f(i));
⋮
f(int i)
{ in("V",i,?v);
  out("V",i,compute(v));
}
```

Diese parallele Berechnung ist nützlich, falls `compute` eine aufwändig zu berechnende Funktion ist.

Das Linda-Modell erlaubt die verständliche Formulierung vieler Synchronisationsprobleme, wie folgende Beispiele zeigen.

Beispiel 7.12 (Dining Philosophers)**.** Eine Implementierung in Linda:

```
phil(int i)
{ while(true)
  { think();
    in("stab", i);
    in("stab", (i + 1) % 5);
    eat();
    out("stab", i);
    out("stab" (i + 1) % 5);
  }
}
```

Initialisierung:

```
for(i = 0; i < 5; i++)
{ out("stab", i);
  eval(phil(i));
}
```

Beispiel 7.13. Wir können auch eine Client-Server-Kommunikation in Linda realisieren. Die Idee ist hierbei, dass wir zur Zuordnung der Antworten zu den Anfragen alle Anfragen durchnummerieren (dies entspricht dem Vergeben einer Auftragsnummer und Zuordnung dieser Nummer zu den Ergebnissen):

```
server()
{ int i=1;
  out("serverindex", i);
  while(true)
  { in("request", i, ?req);
    ...
    out("response", i, result);
    i++;
  }
}

client()
{ int i;
  in("serverindex", ?i); /* hole Auftragsnummer */
  out("serverindex", i + 1);
  ...
  out("request", i, req); /* in req steht der eigentliche Auftrag */
  in("response", i, ?result);
  ...
}
```