Fachhochschule
Münster University of
Applied Sciences

Nebenläufige Programmierung

Modellierung paralleler Threads

Prof. Dr. Nikolaus Wulff

Modellierung von Nebenläufigkeit

Es gibt verschiedene mathematische Beschreibungen, die zum Teil noch unter aktiver Entwicklung sind:

- Petri-Netze (1962)
- Communicating Sequential Processes **CSP** (1978).
- Calculus of Communicating Systems **CCS** (1980).
- „Autonome Agenten“ **π-Calculus** (1992).
- Concurrent Kleene Algebra **CKA** (2006).

Als Actor-Model fanden diese Ideen in Programmier-sprachen Eingang: Erlang, Haskel, Scala, ...

Prozess Notationen

Es seinen P und Q im folgendem Anweisungen für Prozesse oder Threads. Sequenzielle und nebenläufige Aufführung werden geschrieben als:

- P ; Q bedeutet nach der Terminierung von P läuft Q.
- P || Q bedeutet P und Q laufen parallel (zeitgleich).
  - Parallel || darf nicht mit der Auswahl | verwechselt werden!
- Bei P || Q entwickelt sich jeder Prozess unabhängig von dem Anderen weiter. Interaktion kann nur über mögliche gemeinsame Ereignisse erfolgen.

Nebenläufige Algebra

Nebenläufige Ausführung || von Anweisungen P,Q,R erfüllt einfache algebraische Regeln:

- Kommutativ: $(P \parallel Q) = (Q \parallel P)$
- Assoziativ: $(P \parallel (Q \parallel R)) = ((P \parallel Q) \parallel R) \equiv (P \parallel Q \parallel R)$
- Transitiv: $(P \parallel Q) \wedge (Q \parallel R) \Rightarrow (P \parallel R)$

Nicht relevant für uns sind die „neutralen Elemente“ für Mathematiker und theoretische Informatiker:

- Null Element: $(P \parallel STOP) = STOP$
- Eins Element: $(P \parallel RUN) = (RUN \parallel P) = P$

Laufzeit und Komplexität

Indexschreibweise: Es seien $\sum_{i=1}^{n} P_{(i)} = (P_1; P_2; \ldots; P_n)$ die sequentielle und $\prod_{i=1}^{n} P_{(i)} = (P_1 \| P_2 \| \ldots \| P_n)$ die nebenläufige Ausführung der $P_{(i)}$.

Einige interessante Fragen stellen sich:

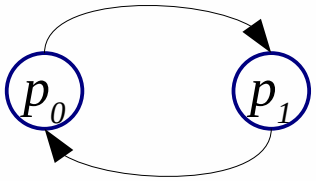
- Wie sehen mögliche (Ereigniss)Spuren bei bei der sequentiellen oder parallelen Ausführung der $P_{(i)}$ aus?
- Wie lautet der Zustandsraum der sequentiellen oder parallelen Ausführung der $P_{(i)}$? Wie groß ist er?

Praxis: Wie groß ist die mögliche Menge der Fehler bei der sequentiellen oder parallelen Ausführung der $P_{(i)}$?

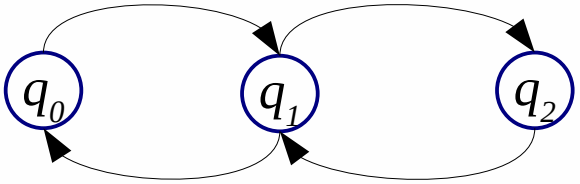
Produktautomat

- Der Zustandsraum eines parallelen Automaten P || Q wächst multiplikativ per Kreuzprodukt:

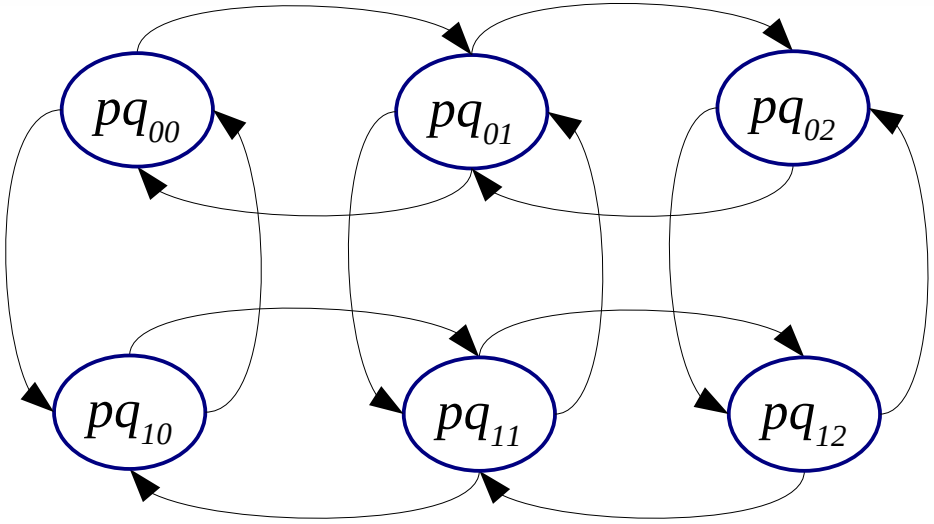
P:



Q:



P||Q:



Offene Fragen

- Sind alle Zustände aus P $\times$ Q erlaubt und erreichbar?
- Gibt es Eingaben $x \in \mathbf{\Sigma_P}$ die auch Transitionen für Q bedingen, d.h. $x \in \mathbf{\Sigma_Q}$?
- Wie soll dann verfahren werden? Z.B. gleichzeitiger Übergang im System P und Q => Synchronisierung
- Was ist wenn P und Q ein unterschiedliches Alphabet verwenden, aber logisch dieselbe Operation gemeint ist?
  - Z.B. Kunde → bestellen und Warenhaus → eingang, sind dann nicht *bestellen* und *eingang* nur unterschiedliche Namen für den selben Vorgang? => Relabeling

Erweitertes Alphabet

- Für R = P || Q gilt $\Sigma_R = \Sigma_P \cup \Sigma_Q$.
- Bei Eingaben aus $\Sigma_R \setminus \Sigma_P$ und aus $\Sigma_R \setminus \Sigma_Q$ können Q bzw. P vollkommen unabhängig reagieren.
- Bei Eingaben aus $\Sigma_P \cap \Sigma_Q$ muss zwischen P und Q eine Synchronisierung vorgenommen werden.
  - Dies setzt voraus, dass das Alphabet entsprechend relabled und Dubletten bereinigt wurden...
  - Der Produktautomat darf nicht in unbestimmte oder verbotene Zustände geraten.
  - Eine genaue Analyse des Zustandraums und aller Übergänge ist erforderlich!

Pseudo-Code

- Die nebenläufige (concurrent) Ausführung $\prod_i P_{(i)}$ der Anweisungen wird geschrieben als:

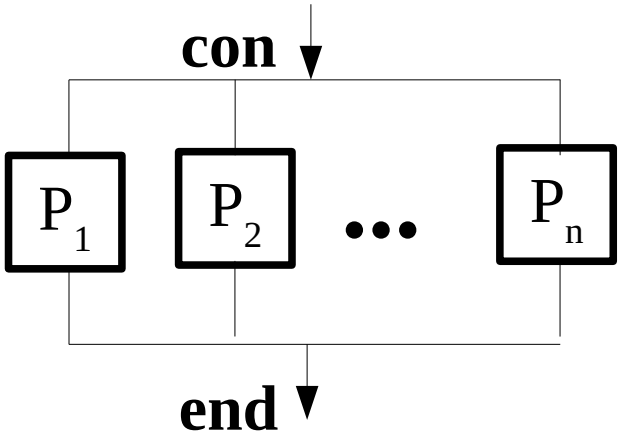
  con $P_1; \ldots ; P_n$ **end**

  bzw.

  con $i = 1$ **to** n **do** $\mathbf{P}_{(i)}$ **end**

- Symbolisch dargestellt durch:

$$(P_1 \parallel P_2 \parallel \ldots \parallel P_n) =$$



Globale und lokale Variablen

- Variablen die innerhalb von **con** deklariert werden sind **lokal** zu dem jeweiligen sequentiellen Programmteil $P_{(i)}$ auch wenn sie denselben Namen haben.
- Variablen die außerhalb von **con** deklariert werden sind für die Anweisungen $P_{(i)}$ **global** sichtbar.
  - Bemerkung: Java Threads verwenden auch für globale Variablen **lokale Kopien**, die nicht immer sofort mit den globalen Variablen synchronisiert werden!
  - Ist sofortige Sichtbarkeit erforderlich so muss das Java Schlüsselwort **volatile** verwendet werden.
  - Dies hat natürlich Performancenachteile...

Inferenz

- Greifen parallele Prozesse auf globale Variablen zu, so kann es zu unerwünschten Seiteneffekten d.h. Inferenzen kommen wie das Programm „sum“ zeigt:

  sum:

  $x = 0; y = 0; z = 0;$

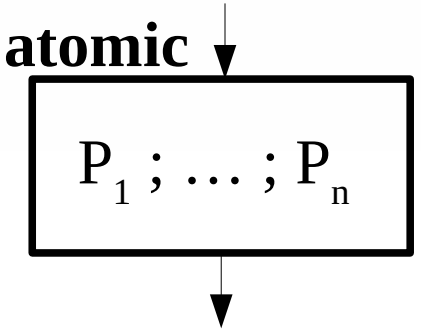
  con $z = x + y; \{x = 1; y = 2;\}$ **end**

- Selbst wenn jede Anweisung $P_{(i)}$ für sich alleine korrekt ist, wird die parallele Ausführung im Fall von Inferenzen u. U. ein falsches Ergebnis liefern!
- Dieses Programm ist nicht deterministisch, da es die möglichen Ergebnisse $z = 0, 1, 2(?)$ oder 3 liefert!

Atomare Aktion

- Sollen Variablen exklusive gelesen/geschrieben werden so müssen die Anweisungen $P_{(i)}$ atomar in einem sogenannten kritischen Abschnitt (critical section) exklusiv ausgeführt werden.

atomic $P_1; \ldots ; P_n$ **end**



Definition:
Ein Programmteil heißt Atomare Aktion wenn kein Zwischenschritt, Zwischenergebnis und kein Zwischenzustand in einem anderen dazu parallelen Programmteil sichtbar ist.

Pragmatik

- Es dürfen keine Annahmen über die zeitliche Ablauffolge paralleler Programmteile gemacht werden. Abhängigkeiten sind explizit abzuwarten!
- Werden in einem Programmteil keine globalen Variablen gelesen oder geschrieben, so ist es atomar.
  - Funktionsaufrufe mit nur lokalen Variablen sind atomar. Daher erlebt auch das λ-Kalkül z. Z. eine Renaissance.
- Die Zuweisung einer Variablen *x* = *expr* ist atomar, wenn *expr* atomar ist.
  - In Java gilt dies nur für int und float Variablen. Bereits bei long oder double sind u.U. zwei seq. Takte notwendig, die zwischendurch unterbrochen werden könnten!

„sum“ Beispiel reloaded

- Als Beispiel werden die Variablen in einem kritischen Abschnitt des Programm „sum“ gesetzt:

$x = 0; y = 0; z = 0;$
con $z = x + y;$ **atomic** $x = 1; y = 2;$ **end end**

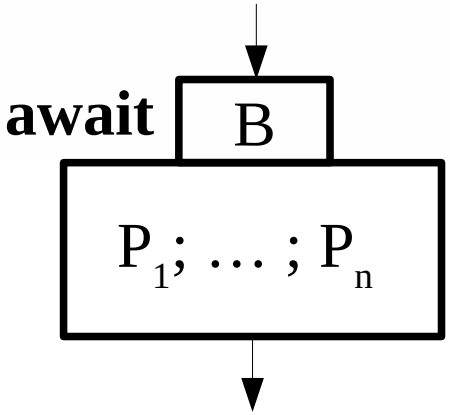
- Jetzt gibt es nur noch die Möglichkeiten $z = 0$ oder 3, je nach dem, ob die Zuweisung vor oder nach der Summation erfolgt.
- Als Bedingung für ein korrektes „sum“ Programm muss verlangt werden, dass die Zuweisung vor der Summation beendet ist.

Explizite Synchronisation

- Um Synchronisation zwischen (parallelen) Programmteilen zu ermöglichen wird eine atomare Bedingung B eingeführt:

  await B [**then** P_1; … ; P_n] **end**

- Die Anweisungen $P_{(i)}$ werden erst dann ausgeführt wenn B wahr ist.



  – Per Definition ist **await** atomar und entspricht:

atomic await B [**then** P_1; … ; P_n] **end end**

await in der Praxis

Das **await** Konstrukt kommt in der Praxis häufiger vor, als auf den ersten Blick vermutet:

- Alle Stream-IO APIs warten blockierend in der read-Operation auf weitere Daten.
  - z.B. System.in.read wartet auf Benutzereingaben.
- Alle ServerSockets warten in der *accept* Methode auf eingehende Anfragen.
- ….
- Java realisiert **await** mit dem **wait** Aufruf und bietet die Möglichkeit künstliche Konstrukte der Informatik wie Mutex, Semaphore, Ring-Buffer etc. zu bauen.

wait - notify

- Die **wait**-Methode der Klasse Object muss in einer *critical section* ausgeführt werden und versetzt den aktuellen Thread in einen „schlafenden Zustand“.
- Ein atomarer Abschnitt wird in Java durch das **synchronized** Schlüsselwort markiert.
- Um den Thread wieder zu reaktivieren muss an dem wartenden Objekt – von einem anderen Thread – die **notify** (oder **notifyAll**) Methode aufgerufen werden.
  - Innerhalb des atomaren Abschnitt können gemeinsame (volatile) Variablen gesetzt werden.
  - Mit diesem simplen Mechanismus kommunizieren Java Threads untereinander.

„sum“ mit await

- Einführung von await in der parallelen Ausführung.

$x = 0; y = 0; z = 0;$
con
await $x>0$ **then** $z = x + y;$ **end**
atomic $x = 1; y = 2;$ **end**
end

- Jetzt gibt es nur noch die Möglichkeiten $z = 3$, da die Summation erst nach der Zuweisung erfolgt.
- Zugegeben ein sinnloses paralleles Programm...

Gefahr: Deadlock

- **await** eliminiert Interferenz bei geteilten Variablen.
- Durch das Einführen von **await** kann es zu einem Stillstand im Programm, einer Verklemmung – dem sogenannten **Deadlock** kommen.
- Es ist bei paralleler Programmierung wichtig darauf zu achten, dass das Programm deadlock frei, lebendig und fair ist, d.h. jeder parallele Programmteil muss eine Chance bekommen seine Aufgabe zu erledigen.
- Da Deadlocks nur schwer durch Tests zu entdecken sind, ist es wichtig eine genaue Analyse durchzuführen und sich anhand der Prozessalgebra von der Fehlerfreiheit des Programms zu überzeugen.