Labelled Transition System Analyzer

Manuel Schulze

mschulze@inf.fu-berlin.de

Inhalt

- Warum modelliert man?
- Überblick über das LTSA Tool
- „Die Philosophen zu Tisch“ (1)
- Modellierung nebenläufiger Programme mit FSP
- „Die Philosophen zu Tisch“ (2)
- Vorteile/Nachteile
- Quellen

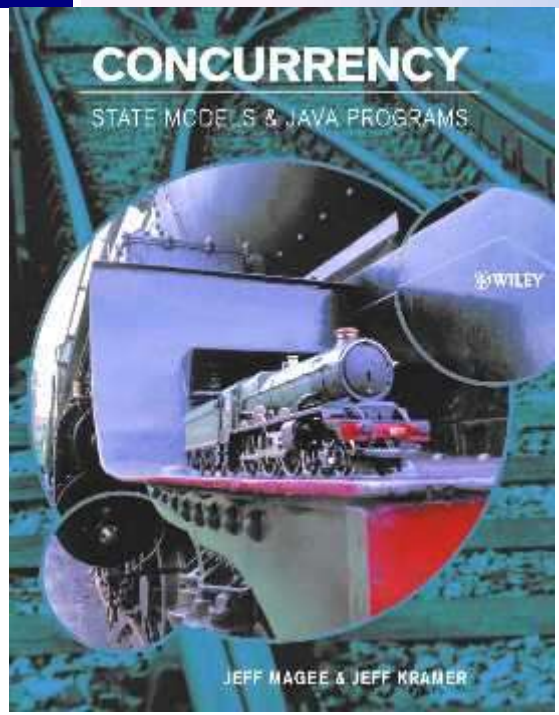
Die Autoren

Jeff Magee

- Beide arbeiten am Imperial College, London
- Professoren für verteilte Softwaretechnik
- Interessen in Softwarearchitekturen, Verteilten Systemen, mobiles Rechnen

Jeff Kramer

„Concurrency – State Models & Java Programs“



- Buch zur gleichnamigen Vorlesung von Jeff Magee
- Beschäftigt sich mit der nichtsequentiellen Programmierung in Java
- Behandelt aber auch das Problem der Modellierung nebenläufiger Systeme

Problem

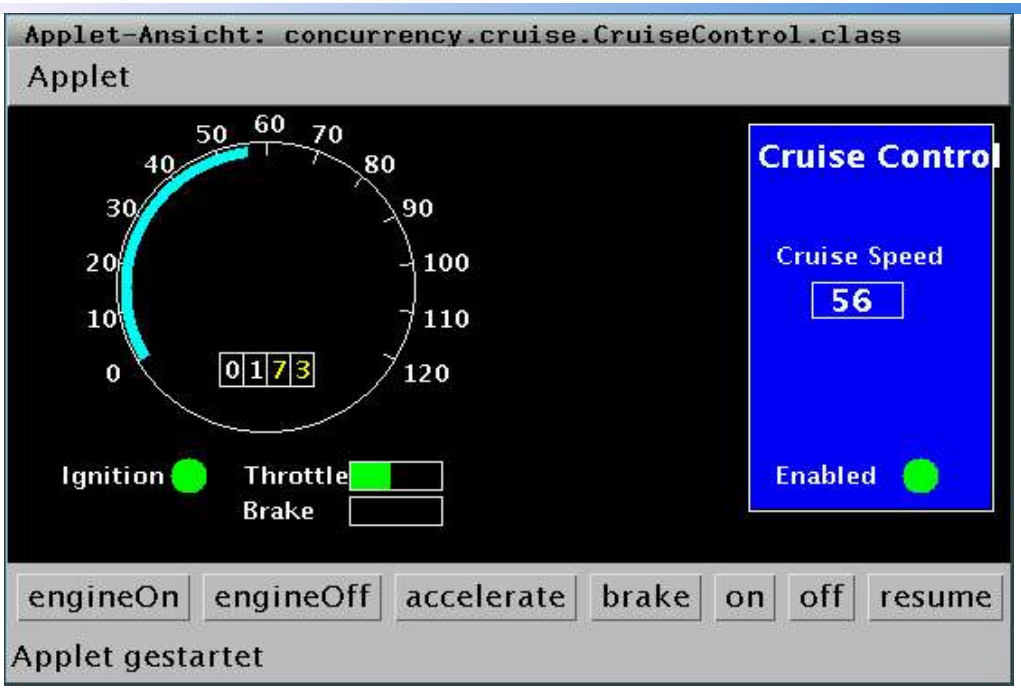
- Zwischen 1985 und 1987 wurden 6 Menschen verletzt oder getötet weil eine Strahlentherapie Maschine (Therac-25) nicht richtig funktionierte

- Schuld waren u.a. sporadisch auftretende Schmutzeffekte im Kontrollprogramm

- Die Ingenieure brauchten lange Zeit diese Fehler zu finden

- Gewünscht eine Möglichkeit Fehler in den Designs verteilter/nebenläufiger Anwendungen/Systeme vor der Benutzung zu entdecken (oder noch besser vor der Implementierung/Realisierung)

Geschwindigkeits Kontroll System (1)



Angenommen man möchte einen Tempomaten programmieren. Wenn die Zündung an ist (**engineOn**) und der Tempomat eingeschaltet (**on**) ist wird die aktuelle Geschwindigkeit aufgenommen und beibehalten. Mit **accelerate**, **brake** oder **off** wird der Tempomat abgeschaltet. Bei **resume** wird die zuvor festgelegte, mit **on** eine neue Geschwindigkeit eingestellt.

Geschwindigkeits Kontroll System (2)

- Implementierung in Java liegt vor => Testen

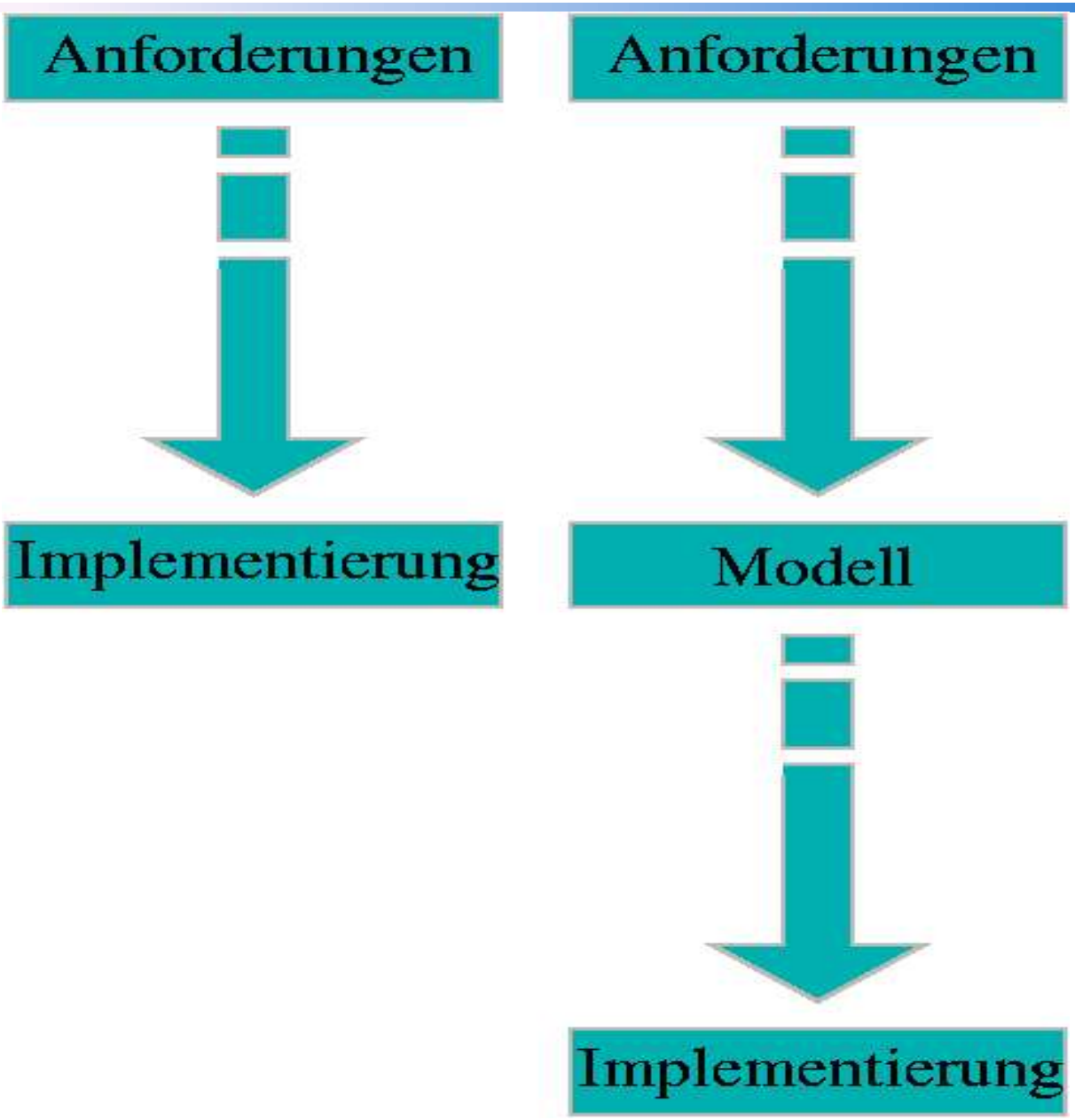
- Verschiedene Testszenarien:
  - Ist das System aktiv nachdem der Motor eingeschaltet und on betätigt wurde?
  - Ist das System ausgeschaltet wenn die Bremse betätigt wurde?
  - Ist das System eingeschaltet wenn auf resume gedrückt wurde?

- Problem: Wie viele Testfälle will man untersuchen? Wieviele gibt es überhaupt?

- Was passiert wenn der Motor abgeschaltet wird und das Kontrollsystem noch aktiviert ist?

Hätte man den Fehler durch Testen gefunden? Im Falle der Therac 25 fand man sie nicht.

Modellierung nebenläufiger Anwendungen/Systeme (1)



Modellierung nebenläufiger Anwendungen/Systeme (2)

- Sequentielles Programm ein Kontrollfluss
- Nebenläufige Programme haben mehrere Kontrollflüsse welche Anweisungen parallel ausführen können
- Daraus ergeben sich Probleme die in sequentiellen Programmen nicht auftauchen (Synchronisation, Verklemmungen etc.)
- Modelle sind vereinfachte Sichten auf die reale Welt
- Erlauben Blick auf den Bereich den man betrachten möchte (z.B. Nebenläufigkeit anstatt Datenrepräsentation etc.)

Modellierung nebenläufiger Anwendungen/Systeme (3)

- Ein realer Prozess:
  - Hat explizite Variablen (vom Programmierer festgelegt)
  - Hat implizite Variablen (Programmzähler, Register etc.)
  - Zustandsänderungen finden durch die Ausführung von Programm Befehlen statt
  - Jeder Befehl besteht aus einer Menge atomarer Aktionen

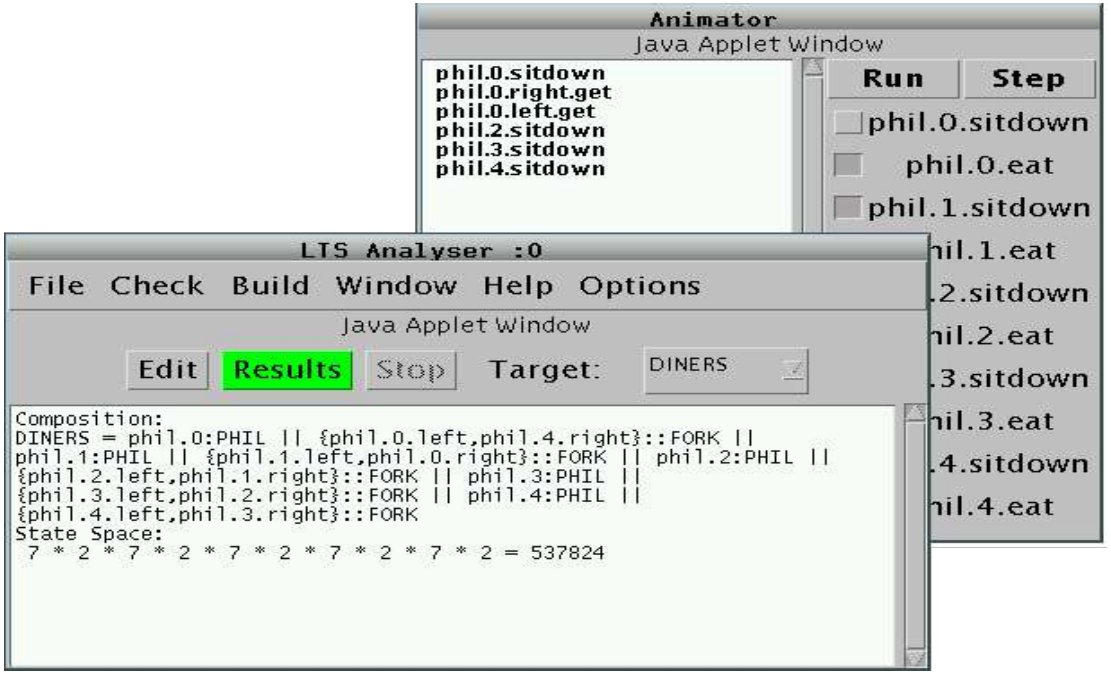
- Ein modellierter Prozess:
  - Hat einen Zustand der von unteilbaren Aktionen verändert wird
  - Eine Aktion überführt den aktuellen Zustand in einen neuen Zustand

Modellierung nebenläufiger Anwendungen/Systeme (4)

- Wie modelliert man die Ausführungszeit eines Prozesses?
  - Abstraktion von der Zeit
- Wie modelliert man Nebenläufigkeit?
  - Relative Ordnung der Operationen unterschiedlicher Prozesse
- Wie sieht das Ergebnis der Modellierung aus?
  - Ein allgemeines Modell unabhängig vom Zeitmanagement der Prozesse
- Konkret: Modelliert wird mit der allgebraischen Beschreibungssprache „finite state process“ (FSP), „labelled transition systems“ (LTS) analysieren und visualisieren solche Modelle

LTSA – Labelled Transition System Analyzer (1)

- Tool geschrieben in Java
- Basiert auf dem Java SceneBeans Framework (Framework zur Erzeugung und Kontrolle animierter Grafiken)
- In zwei Versionen und als Applet oder Standardanwendung erhältlich

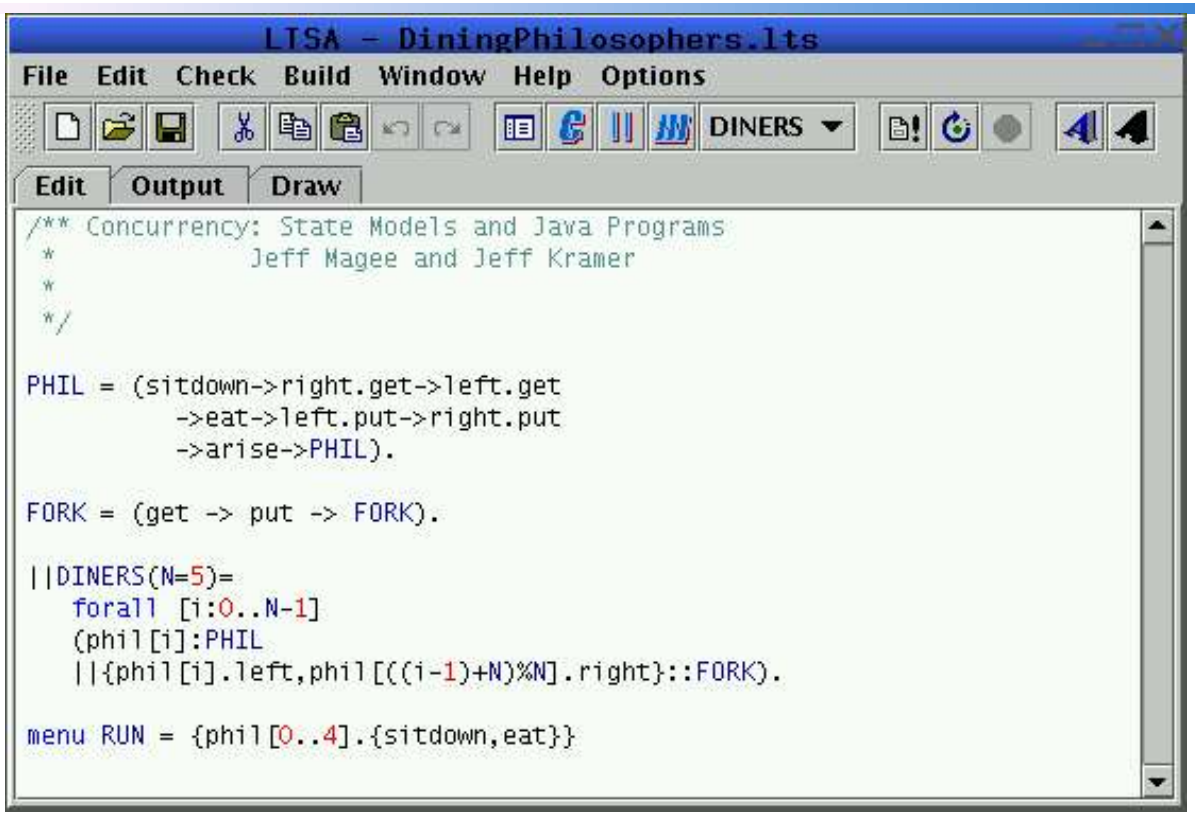


LTSA V1

LTSA – Labelled Transition System Analyzer (2)

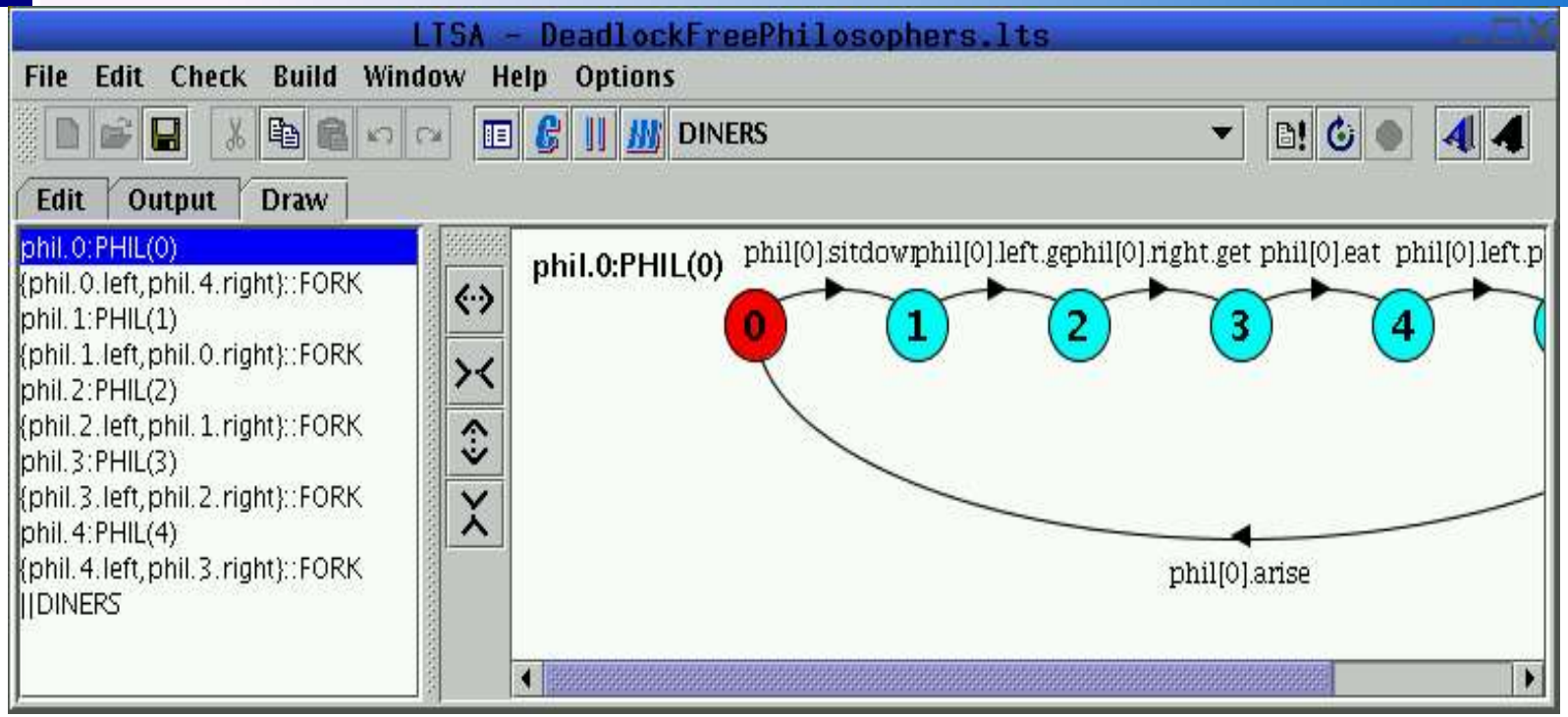
- LTSA V2.2 vom 1.9.2001
- Versteht die FSP (Finite State Process) Sprache und visualisiert solche Programme in Form von Graphen
- Erlaubt interaktive Ausführung der einzelnen Schritte des Programms
- Ist in der Lage FSP Programme zu minimieren (falls möglich)
- Prüft FSP Programme auf Gültigkeit und erkennt mögliche Verklemmungen

LTSA – Labelled Transition System Analyzer (3)



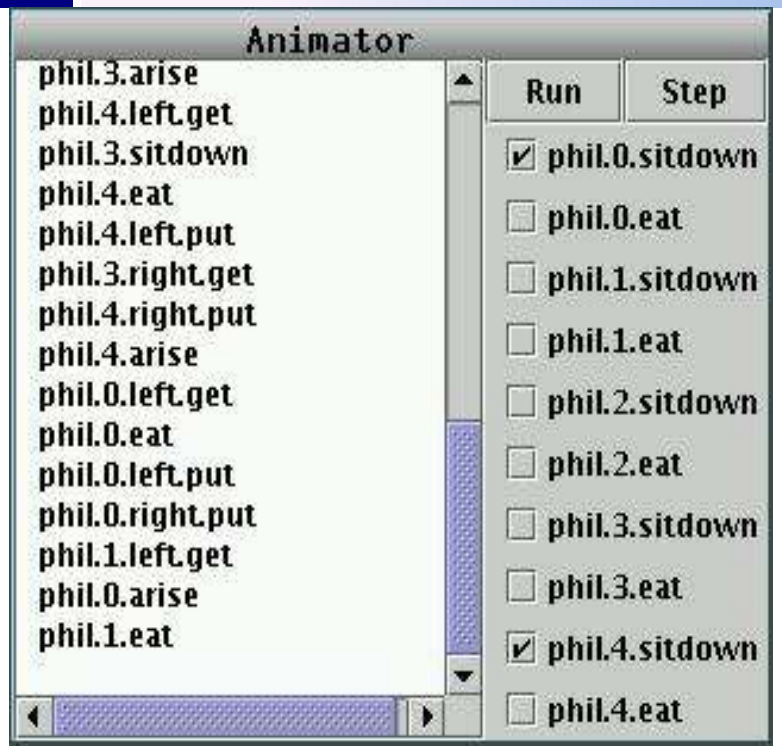
- Hauptfenster des LTSA 2.2
- Auschneiden/Kopieren/Einfügen und Undo Funktionen für den Texteditor
- FSP Editor mit Syntax Highlighting

LTSA – Labelled Transition System Analyzer (4)



- Visualisierung der Prozesse in Form von Graphen
- Aktueller Prozess wird rot dargstellt
- Prozesse in denen nach der Ausführung einer Aktion eine Zustandsänderung auftrat werden in der Liste rot hinterlegt

LTSA – Labelled Transition System Analyzer (5)



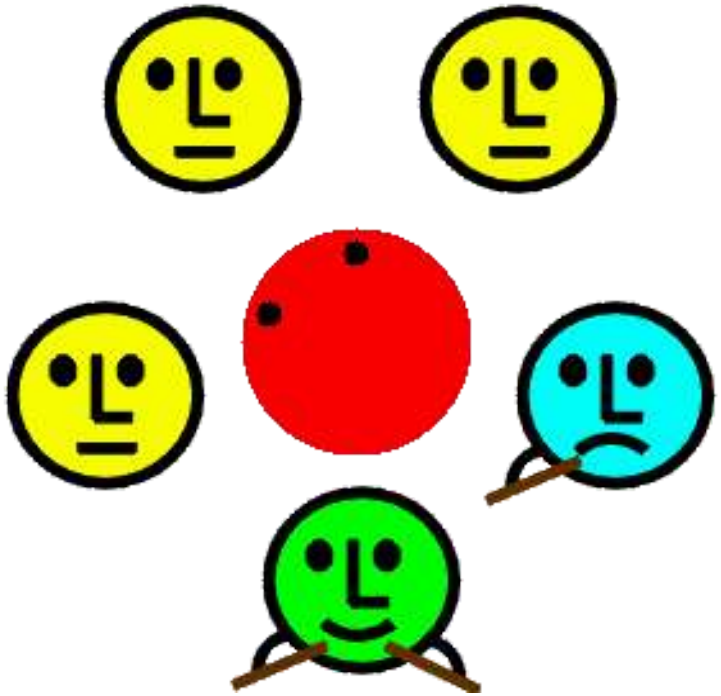
- Im Animator kann das modellierte Programm interaktiv ausgeführt werden
- Die Liste zeigt die ausgeführten Aktionen
- Die Ausführung wird auch grafisch im Graphen dargestellt
- „stumme Aktionen“ können entweder Schrittweise mit **Step** oder alle auf einen Schlag mit **Run** ausgeführt werden
- Im Animator Wiedergabe von Fehlersequenzen möglich (z.B. Schritte bis zu einem Deadlock)

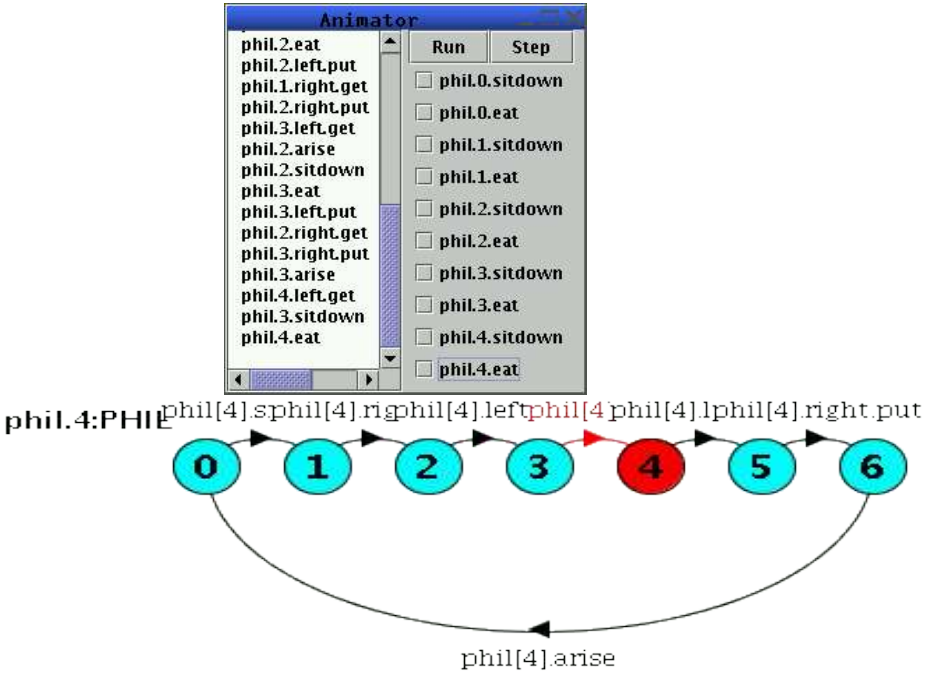
Beispiel – Das „Dining Philosophers“ Problem (Dijkstra 71)

- 5 Philosophen sitzen am Tisch und tun was Philosphen so tun, essen und philosophieren

- Auf dem Tisch liegen 5 Gabeln und in der Mitte steht ein Teller mit Spaghetti

- Jeder Philosoph benötigt 2 Gabeln zum essen





Modellierung mit der „Finite State Process“ Sprache

- Modelliert werden Prozesse (vglb. mit Zuständen in Automaten und Übergänge (Transitionen) (vglb. mit Zustandsübergängen)

- Es werden 3 Prozesstypen unterschieden (lokale, primitive und zusammengesetzte Prozesse)

- 3 Endzustände die ein Prozess erreichen kann (**END**, **STOP**, **ERROR**)

- Modell eines Prozesses kann sich über mehrere Zeilen erstrecken wobei jede Zeile mit einem **,** (Komma) abgeschlossen werden muss

- Am Ende einer Prozessdefinition steht ein **.** (Punkt)

- Zeilenkommentare mit **//**, Blockkomentare zwischen **/**...*/**

FSP – Action Prefix

Sei **x** eine Aktion und **P** ein Prozess, dann beschreibt **(x ->P)** einen Prozess der erst **x** ausführt und sich anschliessend wie **P** verhält.

- Beispiel:

```
SWITCH = OFF,
OFF    = (on -> ON),
ON     = (off-> OFF).
```

- Ein einfacher Lichtschalter (Startprozess OFF)

- Vereinfachung durch Substitution zu:

```
SWITCH = (on -> off -> SWITCH).
```

FSP - Auswahl

Wenn **x** und **y** Aktionen sind dann beschreibt **(x -> P | y -> Q)** einen Prozess der **x** oder **y** ausführt. Nachdem die erste Aktion ausgeführt wurde verhält sich das System wie **P** wenn **x** die erste Aktion war und wie **Q** wenn **y** die erste Aktion war.

- Beispiel:

```
DRINKS = (red ->coffee->DRINKS
         |blue->tea   ->DRINKS
         ).
```

Wer entscheidet welche Alternative gewählt wird?

FSP – nichtdeterministische Auswahl

Der Prozess (**x -> P | x -> Q**) beschreibt einen Prozess der **x** ausführt und sich dann wie **P** oder **Q** verhält.

- Beispiel:

```
COIN = (toss -> HEADS | toss -> TAILS),
HEADS = (heads -> COIN),
TAILS = (tails -> COIN).
```

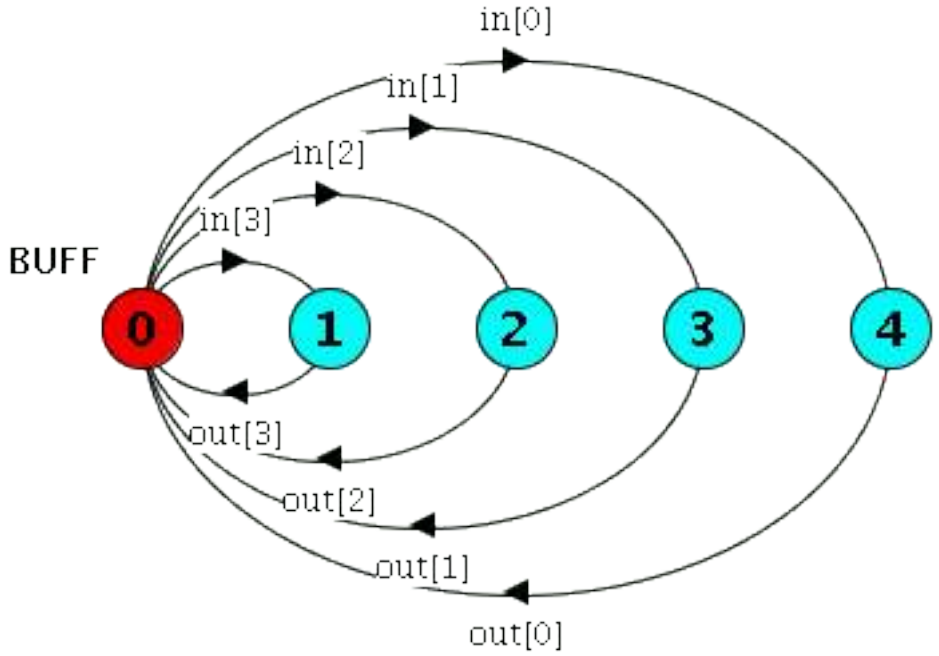
Wer entscheidet welche Alternative gewählt wird?

FSP – Prozesse und Aktionen mit Indexe (1)

- Um Prozesse und Aktionen mehrfach verwenden zu können kann man sie mit Indexe versehen

- Beispiel für Aktionen mit Index:

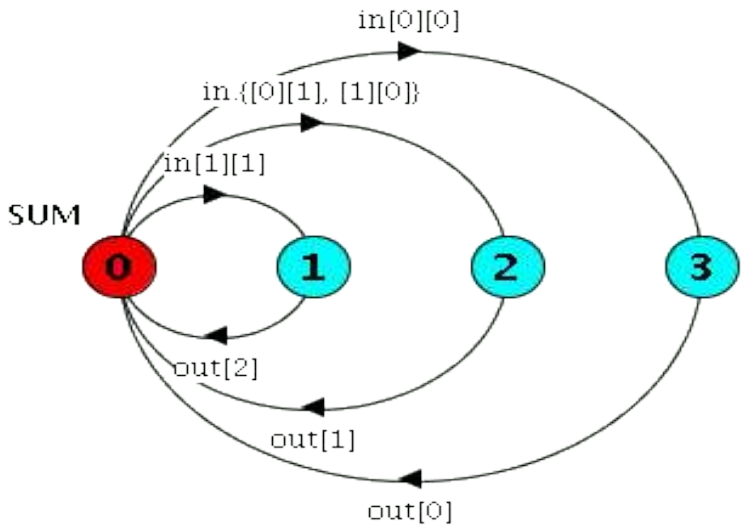
```
BUFF = (in[i:0..3]->out[i]-> BUFF).
```



FSP – Prozesse und Aktionen mit Indexe (2)

- Beispiel für Prozesse mit Index:

```
const N = 1
range T = 0..N
range R = 0..2*N

SUM        = (in[a:T][b:T]->TOTAL[a+b]),
TOTAL[s:R] = (out[s]->SUM).
```



- Das Beispiel zeigt auch die Verwendung von Konstanten und Wertebereiche

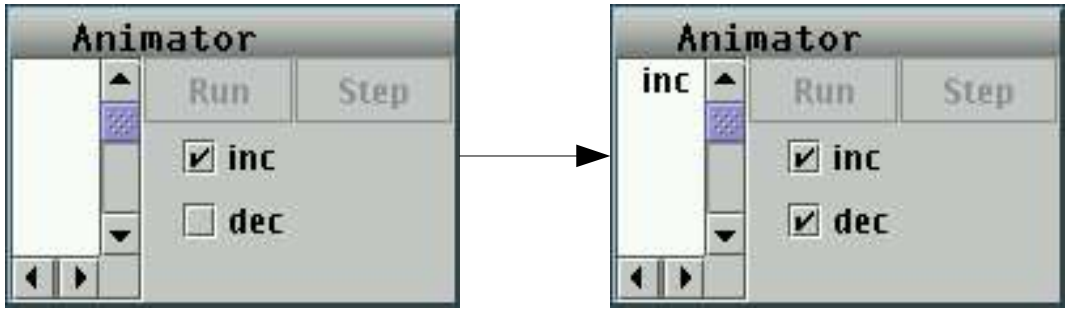
FSP – bedingte Aktionen

Die Auswahl (**when B x -> P | y -> Q**) heisst, wenn **B** wahr ist dann kann eine der Aktionen **x** oder **y** ausgeführt werden. Anderenfalls kann nur **y** ausgeführt werden.

- Beispiel:

```
COUNT (N=3)    = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
                |when(i>0) dec->COUNT[i-1]
                ).
```

- Wie sieht die Ausführung aus?



Warum?

FSP – Prozess Alphabet/implizite Alphabeterweiterung

Das Alphabet eines Prozesses ist die Menge der Aktionen welche der Prozess ausführen kann.

- Beispiel:

```
WRITER = (write[1]->write[3]->WRITER)
       +{write[0..3]}.
```

- Das Alphabet `{write[1], write[3]}` wird um das Alphabet `{write[0], write [2]}` erweitert

FSP – Modellierung von Nebenläufigkeit (1)

Wenn **P** und **Q** Prozesse sind dann beschreibt (**P || Q**) die nebenläufige Ausführung von **P** und **Q**.

- Beispiel:

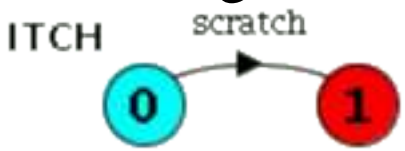
```
ITCH  = (scratch->STOP).
CONVERSE = (think->talk->STOP).

||CONVERSE_ITCH = (ITCH || CONVERSE).
```

- Mögliche Kontrollflüsse:

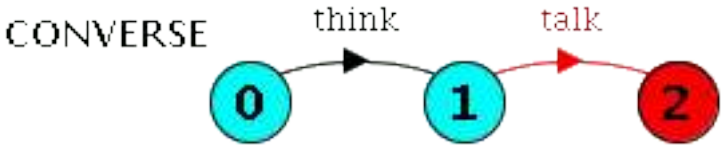
think	talk	scratch
think	scratch	talk
scratch	think	talk

FSP – Modellierung von Nebenläufigkeit (2)

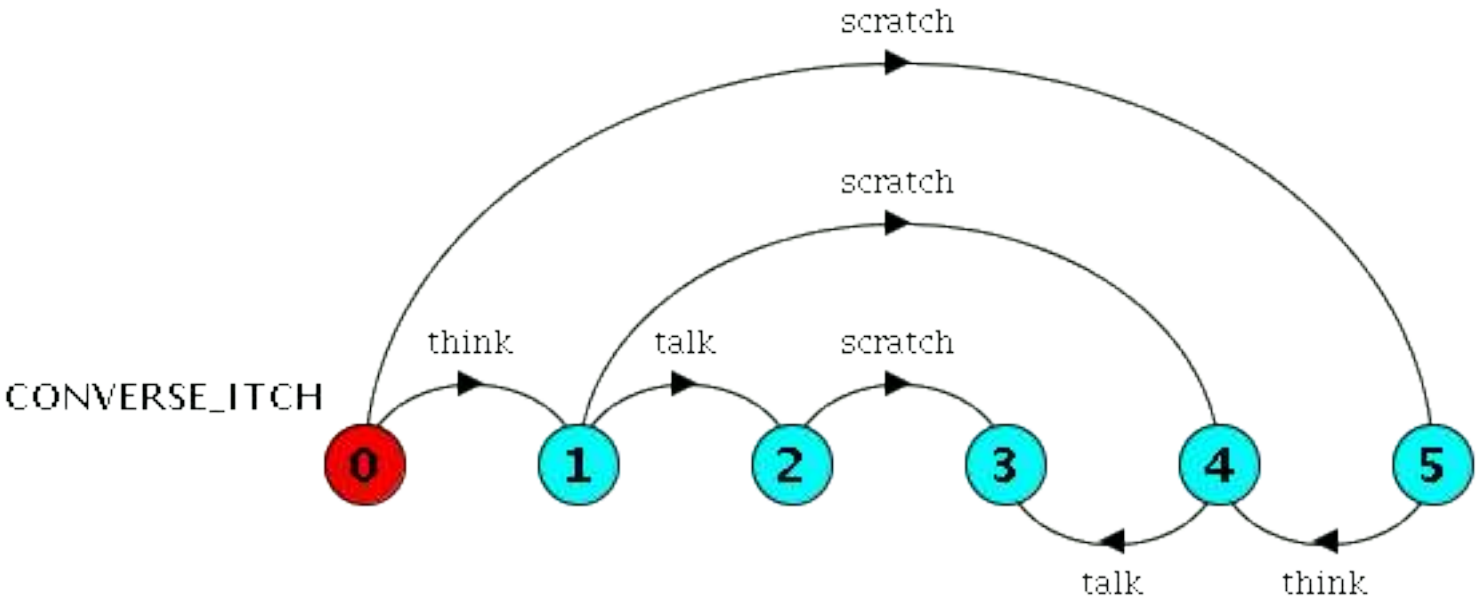
- LTSA kann nebenläufige Prozesse zusammensetzen und so einen Blick auf den gesamten Prozess liefern



2 Zustände



3 Zustände



2 x 3 Zustände

FSP – Modellierung von Nebenläufigkeit (3)

Der || Operator ist:

Kommutativ: **(P || Q) = (Q || P)**

Assoziativ: **(P || (Q || R)) = ((P || Q) || R) = (P || Q || R).**

- **||X = (P || Q)** wird auch als zusammengesetzer Prozess bezeichnet und kann Bestandteil in anderen zusammengesetzten Prozessen sein

- Beispiel:

```
||MAKERS = (MAKE_A || MAKE_B).
||FACTORY = (MAKERS || ASSEMBLE).
```

- Siehe auch die Grafik auf Folie 26, die Prozesse `CONVERSE` und `ITCH` als Kompositum im LTSA

FSP – gemeinsame Aktionen

Wenn ein zusammengesetzter Prozess allgemeine Aktionen hat dann werden diese Aktionen als gemeinsame Aktionen bezeichnet. Mit ihnen wird die Interaktion zwischen Prozessen modelliert. Gemeinsame Aktionen müssen von allen beteiligten Prozessen zur gleichen Zeit ausgeführt werden.

- Beispiel:

```
MAKER = (make->ready->MAKER).
USER  = (ready->use->USER).

||MAKER_USER = (MAKER || USER).
```

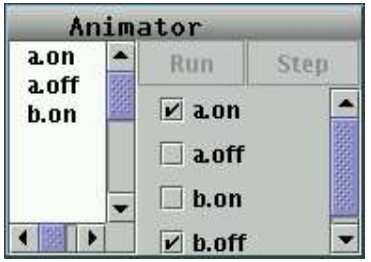
- Der Prozess `USER` kann `use` erst ausführen nachdem `MAKER` `make` ausgeführt hat und die gemeinsame Aktion `ready` ausgeführt wurde.

FSP – Prozess Benennung (1)

a:P ergänzt jede Aktion des Prozesses **P** um den Präfix **a**.

- Beispiel: 2 Exemplare des Prozesses `SWITCH`

```
SWITCH =  (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

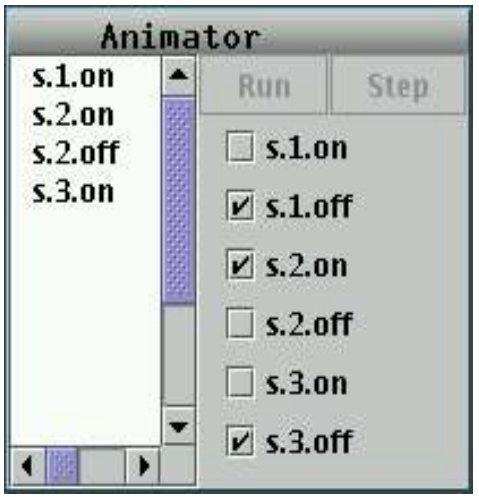


- Die Anweisung:

```
||SWITCHES(N=3) = (forall[i:1..N] s[i]:SWITCH).
```

erzeugt drei Exemplare von `SWITCH`. Alternativ kann man auch schreiben:

```
||SWITCHES1(N=3) = (s[i:1..N]:SWITCH).
```

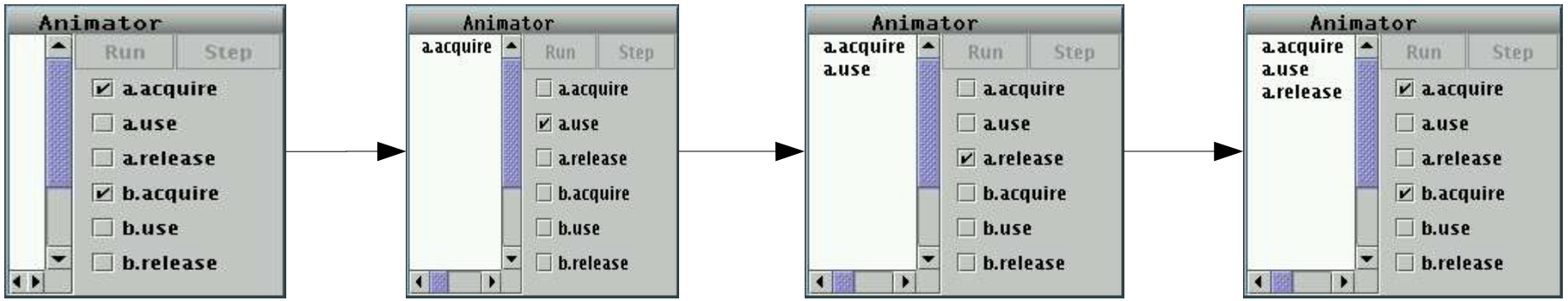


FSP – Prozess Benennung (2)

Die Konstruktion **{a1, ..., ax}::P** ersetzt jede Aktionsbezeichnung **n** im Alphabet von **P** durch die Bezeichner **a1.n,...,ax.n**. Ausserdem wird jede Transition **(n -> X)** von **P** durch **({a1.n,...ax.n} -> X)** ersetzt.

- Beispiel:

```
RESOURCE = (acquire->release->RESOURCE).
USER = (acquire->use->release->USER).

||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).
```



- Warum wird nur eine Ressource angefordert und freigegeben?

FSP – Aktions-Umbenennung

Aktionen können mit dem Konstrukt /{labelneu_1/labelalt_1,...,labelneu_n/labelalt_n} umbenannt werden.

- Mit diesem Konstrukt kann man zusammengesetzte Prozesse für bestimmte Aktionen synchronisieren

- Beispiel:

```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).

||CLIENT_SERVER = (CLIENT || SERVER)
                  /{call/request, reply/wait}.
```

- request wird zu call und wait zu reply, CLIENT und SERVER sind in call/request und wait/reply synchronisiert

FSP – Stumme Aktionen

- 2 Möglichkeiten:

<u>1. Möglichkeit:</u> Der Operator **\{a1,...,ax}**, angehängt an einen Prozess **P**, entfernt die Aktionen **a1,...,an** aus dem Alphabet von **P** und macht sie zu „stummen“ Aktionen. Im Graphen werden diese Aktionen mit tau bezeichnet

<u>2. Möglichkeit:</u> Hängt man an einen Prozess **P** den Operator **@{a1,...,ax}** an, dann werden genau die Aktionen aus dem Alphabet von **P** versteckt die nicht in der Menge **{a1,...,ax}** auftauchen.

- Beispiel:

```
USER = (acquire->use->release->USER)\{use}.
```

⇕

```
USER = (acquire->use->release->USER)@{acquire,release}.
```

Nochmal das „Dining Philosophers“ Problem (1)

- Modellierung eines Philosophen und einer Gabel

```
PHIL = (sitdown->right.get->left.get
          ->eat->left.put->right.put
          ->arise->PHIL).

FORK = (get -> put -> FORK).
```

- Die Anweisungen im Block:

```
||DINERS(N=5)=
   forall [i:0..N-1]
   (phil[i]:PHIL
   ||{phil[i].left,phil[((i-1)+N)%N].right}::FORK).
```

erzeugen 5 Exemplare der Philosophen und Gabel Prozesse die alle nebenläufig sind

- Kommuniziert wird über gemeinsame Aktionen

Nochmal das „Dining Philosophers“ Problem (2)

- Es findet eine Umbenennung der Aktionen in `FORK` statt, durch die Anweisung:

```
{phil[i].left,phil[((i-1)+N)%N].right}::FORK)
```

- Die letzte Anweisung erzeugt einen Eintrag im Menü RUN

- Die Aktionen die auf `sitdown` und `eat` enden werden versteckt

```
menu RUN = {phil[0..4].{sitdown,eat}}
```

Nochmal das „Dining Philosophers“ Problem (3)

- Die Alphabete der Prozesse sehen im Prinzip wie folgt aus:

FORK.0 = ({phil.0.left.get, phil.4.right.get} -> {phil.0.left.put, phil.4.right.put} -> FORK.0)
FORK.1 = ({phil.1.left.get, phil.0.right.get} -> {phil.1.left.put, phil.0.right.put} -> FORK.1)
FORK.2 = ({phil.2.left.get, phil.1.right.get} -> {phil.2.left.put, phil.1.right.put} -> FORK.2)
FORK.3 = ({phil.3.left.get, phil.2.right.get} -> {phil.3.left.put, phil.2.right.put} -> FORK.3)
FORK.4 = ({phil.4.left.get, phil.3.right.get} -> {phil.4.left.put, phil.3.right.put} -> FORK.4)

PHIL.0 = (phil.0.sitdown -> phil.0.right.get -> phil.0.left.get -> phil.0.eat -> phil.0.left.put -> phil.0.right.put -> phil.0.arise -> PHIL)
PHIL.1 = (phil.1.sitdown -> phil.1.right.get -> phil.1.left.get -> phil.1.eat -> phil.1.left.put -> phil.1.right.put -> phil.1.arise -> PHIL)
PHIL.2 = (phil.2.sitdown -> phil.2.right.get -> phil.2.left.get -> phil.2.eat -> phil.2.left.put -> phil.2.right.put -> phil.2.arise -> PHIL)
PHIL.3 = (phil.3.sitdown -> phil.3.right.get -> phil.3.left.get -> phil.3.eat -> phil.3.left.put -> phil.3.right.put -> phil.3.arise -> PHIL)
PHIL.4 = (phil.4.sitdown -> phil.4.right.get -> phil.4.left.get -> phil.4.eat -> phil.4.left.put -> phil.4.right.put -> phil.4.arise -> PHIL)

- Gabel 0 kann entweder vom Philosophen 0 oder 4 genommen werden usw.

Vorteile/Nachteile

Vorteile

- Stabil
- Deadlock Erkennung
- Gute Oberfläche
- Wird vom Buch „Concurrency“ und vielen anderen Professoren benutzt (Siehe VL „Verteilte Algorithmen“ WS03/04)

Nachteile

- FSP schwer zu verstehen und zu schreiben
- Bezug zum realen Code schwer herstellbar (Diskussion)

Quellen (1)

- Jeff Magee, Jeff Kramer: Concurrency (Wiley 1999)
Steht auch in der Bibliothek

- Homepage von Jeff Magee
[http://www.doc.ic.ac.uk/~jnm/]

- Homepage von Jeff Kramer
[http://www.doc.ic.ac.uk/~jk/]

- LTSA V1 Homepage
[http://www.doc.ic.ac.uk/%7Ejnm/book/ltsa/LTSA.html]

- LTSA V2 Homepage
[http://www.doc.ic.ac.uk/~jnm/book/ltsa-v2/]

- Folien zum Buch „Concurrency“
[http://www.doc.ic.ac.uk/~jnm/book/slides.html]

Quellen (2)

- Applets zum Buch „Concurrency“
[http://www.doc.ic.ac.uk/~jnm/book/book_applets/concurrency.html]

- Das SceneBeans Framework
[http://www-dse.doc.ic.ac.uk/Software/SceneBeans/]

- Webseite zur VL Nichtsequentielle Programmierung WS01/02
[http://www.inf.fu-berlin.de/lehre/WS01/19530-V/lectures.html]

- Webseite zur VL Nichtsequentielle Programmierung WS02/03
[http://www.inf.fu-berlin.de/lehre/WS02/nsp/index.html]

Danke