Modelle in Verteilten Berechnungen und Simulationen

Th Letschert

Verteilte Systeme: Modell-Begriff

- **Modell**
  Festlegung von relevanten Eigenschaften der „Welt“ der Programme

- **Sequentielle Systeme**
  - **Modelle äquivalent / unerheblich (Churchsche These)**

- **Verteilte Systeme**
  - **Modelle wichtig**
  - **Beispiel für Modell-Eigenschaften**
    - **synchrone / asynchrone Kommunikation**
    - **FIFO- / nicht-FIFO -Kanal**
    - **Fixe / variable Prozesstopologie**
    - **Fehlerbehafteter / fehlerfreier Nachrichtentransfer**
    - **. . .**

Standard-Modell: Kommunizierende Prozesse

- **Kommunizierende Prozesse**
  System: Menge von strikt sequenziellen Prozessen die sich gegenseitig Nachrichten zusenden.
  Kein Nachrichtenverlust, keine Verfälschung, kein Prozess-Ausfall.
- **Varianzen im Standard-Modell**
  - **Topologie: statisch / dynamisch**
    Ändert sich Menge der Prozesse während einer Berechnung oder ist sie fix
  - **Prozessbekanntschaften: Lebenszyklus**
    Wann wird der Partner einer Kommunikation festgelegt
    - **Statisch: im Quellcode**
    - **In Bindephase vor Programmstart (Port-Konzept)**
    - **Dynamisch zur Laufzeit**
  - **Synchronität der Kommunikation**
  - **Natur der Nachrichten**
    - **Einfache Werte**
    - **Referenzen und Werte**
    - **Alles inklusive Prozesse**

Reaktive vs kontrollorientierte Notation

Prozesse werden in einer reaktiven oder einer kontroll-orientierten Notation spezifiziert

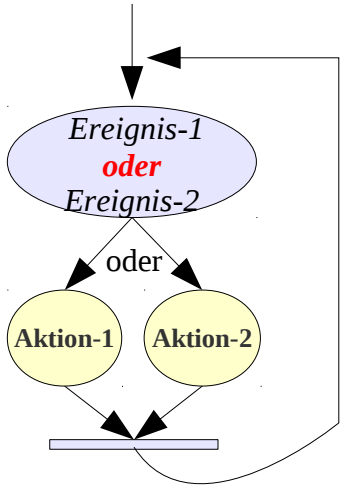
- **Kontrollorientiert**

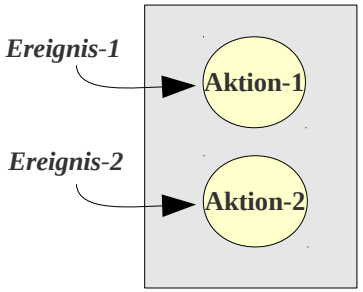
  Ein sequentieller Algorithmus definiert das Verhalten eines Prozesses

- **Ereignisorientiert / Reaktiv**

  Ein Prozess wird über seine Reaktionen auf externe Ereignisse definiert

Theoretisch äquivalent – Praktisch bedeutsame Konsequenzen



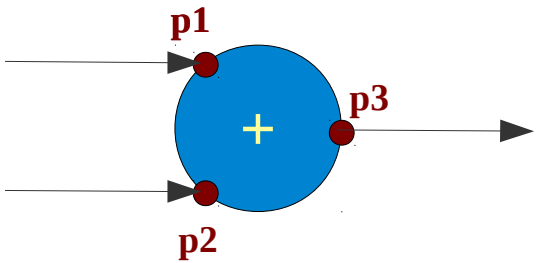


Beispiel Addierer / Kontrollorientierte Notation

```
Prozess Adder {

  PortIn p1: Integer
  PortIn p2: Integer
  PortOut p3: Integer

  while (true) {
    var v1 = p1.read()
    var v2 = p2.read()
    var v3 = v1 + v2
    p3.send(v3)
  }
}
```



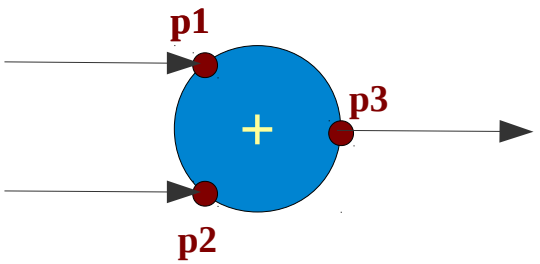
Beispiel Addierer / Ereignisorientierte Notation

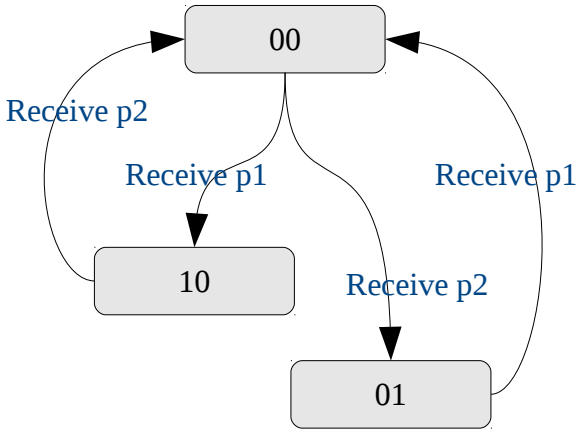
```
Prozess Adder {
  PortIn p1: Integer
  PortIn p2: Integer
  PortOut p3: Integer

  var state = 00
  var v1
  var v2

  Ready(p1) =>
    case (state) {
        00 : v1 = p1.receive(); state = 10
        01 : v1 = p1.receive(); p3.send (v1+v2); state = 00
    }
  }

  Ready(p2) =>
    case (state) {
        00 : v2 = p2.receive(); state = 01
        10 : v2 = p2.receive(); p3.send (v1+v2); state = 00
    }
  }
}
```





Zustandsdiagramm

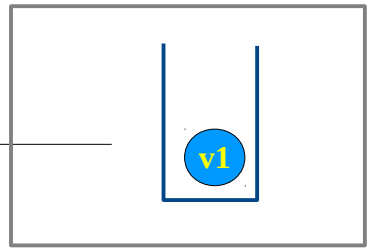
Vorteil der ereignisorientierten Notation

Die Prozessspezifikation ist die Spezifikation eines Prozesses der während seiner Blockade keine Daten im Stack hat!

```
Prozess Adder {

  PortIn p1: Integer
  PortIn p2: Integer
  PortOut p3: Integer

  while (true) {
    var v1 = p1.read()
    var v2 = p2.read()
    var v3 = v1 + v2
    p3.send(v3)
  }
}
```



Zustandsinfo bei kontrollorientierter Notation im Stack.

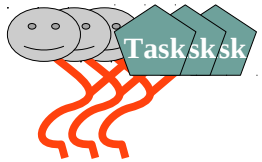
Bei kontroll-orientierter Notation ist der Stack bei einer blockierenden Operation naturgemäß i.d.R. nicht leer.

Prozessdefinition im Standardmodell: Reaktiv oder Kontrollorientiert

Vorteil der ereignisorientierten Notation

Die Prozessspezifikation ist die Spezifikation eines Prozesses

*der während seiner Blockade **keine Daten im Stack** hat!*



Konsequenz: schwere oder leichte Prozesse

- **Kontrollorientiert definierte Prozesse belegen permanent einen zugeordneten Thread.**

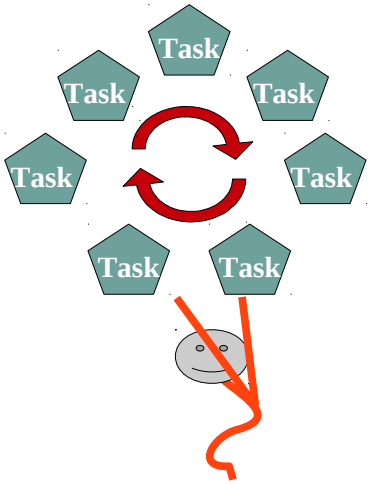
  Prozesse sind schwer

  Relativ wenige Prozesse lasten das System aus.

- **Ereignisorientierte definierte Prozesse können ohne dezidierten Thread implementiert werden.**

  Prozesse sind leicht

  Viele tausend Prozesse sind kein Problem.



In Scala kann zwischen den beiden Notationen unterschieden werden (reaktive und thread-basierte Aktoren) .

Java 7 (Fork/Join-Framework) bietet leichtgewichtige „Tasks“ für spezielle Problemstellungen.

Akka hat reaktive Aktoren.

Multi-Receive: Optimierung der ereignisorientierten Notation

Die ereignisorientierte Notation ist umständlich.

Eine unterstützenden Notation mit „Multi-Receive“ kann ihren Gebrauch vereinfachen!

Nicht mit „Gleichzeitigem Empfang“ verwechseln ! (s.u.)

Beispiel*:*

```
Prozess Adder {
  PortIn  p1: Integer
  PortIn  p2: Integer
  PortOut p3: Integer

  Ready(p1, p2) =>
     p3.send (p1.receive()+p2.receive());
}
```

Bei kontroll-orientierte Notation mit „Multi-Receive“ (p1 und p2 sind bereit)
Kann Stack-frei „leicht“ implementiert werden: Transformation in komplexeren Zustandsautomat (siehe oben).

Visuelle Datenfluss-Systeme

Charakteristika:

- Statische Topologie
- Reine Werte als Nachrichten
- Spezifikation
  - Knotensprache für Prozessdefinitionen (Ports)
  - Netzsprache zur Definition von Prozessverknüpfungen
- Vorteil
  - einfach, ermöglicht graphische Netzsprache
- Nachteil
  - Beschränkte Ausdruckskraft
- Semantik meist nur implizit definiert
- Beispiel: Knime

FBP – Flow Based Programming

Charakteristika:

- Ähnlich den visuellen Datenfluss-Systeme
- Textuelle Notation
- Zweistufige Spezifikation
  - Netzsprache (Interne / externe DSL)
  - Knotensprache (Programmiersprache)
- Beispiele
  - Morris / Steinseifer / Hoffmann / Braden / ….

Datenfluss-Systeme

Charakteristika:

- Prozesse = Funktionen auf Daten-Strömen
- Rekursive Definitionen möglich
- Prozessnetze nicht statisch
- Visuelle Programmierung nicht möglich

Datenfluss-Systeme

Entwurfsentscheidungen:

statisch oder dynamisch

- Statische Netztopologie
- Dynamische Netztopologie

Schwere oder Leicht

- thread-basierte und / oder
- leichte (reaktive) Prozesse

Push oder Pull

- Push-Netze
  - Datenproduzenten treiben die Verarbeitung
  - typischerweise mit Synchronisation über Puffer beschränkter Kapazität
  - übliche Praxis
- Pull-Netze
  - Nachfrage-getriebene Netze
  - nur theoretisch interessant

Kalküle: Auf der Suche nach der Essenz der Berechnungen

NASA E/PO, Sonoma State University, Aurore Simonne...

Kalküle: Auf der Suche nach der Essenz der Berechnungen

Essenz der sequentiellen Programme

λ-Kalkül

- Einfach
- Wohldefinierte Semantik
  die Bedeutung des Ganzen ergibt sich stets aus der Bedeutung seiner Teile
- Vollständig

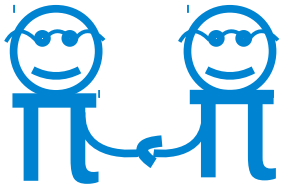
Kalküle: Auf der Suche nach der Essenz der Berechnungen

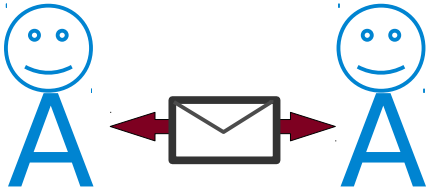
Essenz nebenläufiger / verteilter Programme

- Kein Modell das mit dem λ-Kalkül vergleichbar wäre
  Die Semantik des Ganzen ergibt sich nicht (immer) aus der seiner Teile
  $D[[P_1 \,||\, P_2]] \not\sim F(\, D[[P_1]], D[[P_2]] \,)$

Zwei wichtige Modell-Klassen (neben vielen anderen)

- **Prozess-Kalküle**
  Robin Milner / Tony Hoare u.a.
- **Aktoren**
  Carl Hewitt u.a.





Prozess-Kalküle Beispiel R. Milner CSS

CSS – *Communicating Sequential Systems*

Idee basierend auf Lambda-Kakül

- **λx.M[x] N ~> M[N]**
  λ ist ein „Kanal“ über den N als x in M kommuniziert wird
- **ax.P[x] | âVQ ~> P[V] | Q**
  a ist ein „Kanal“ über den der Wert V von Q an P gesendet wird.

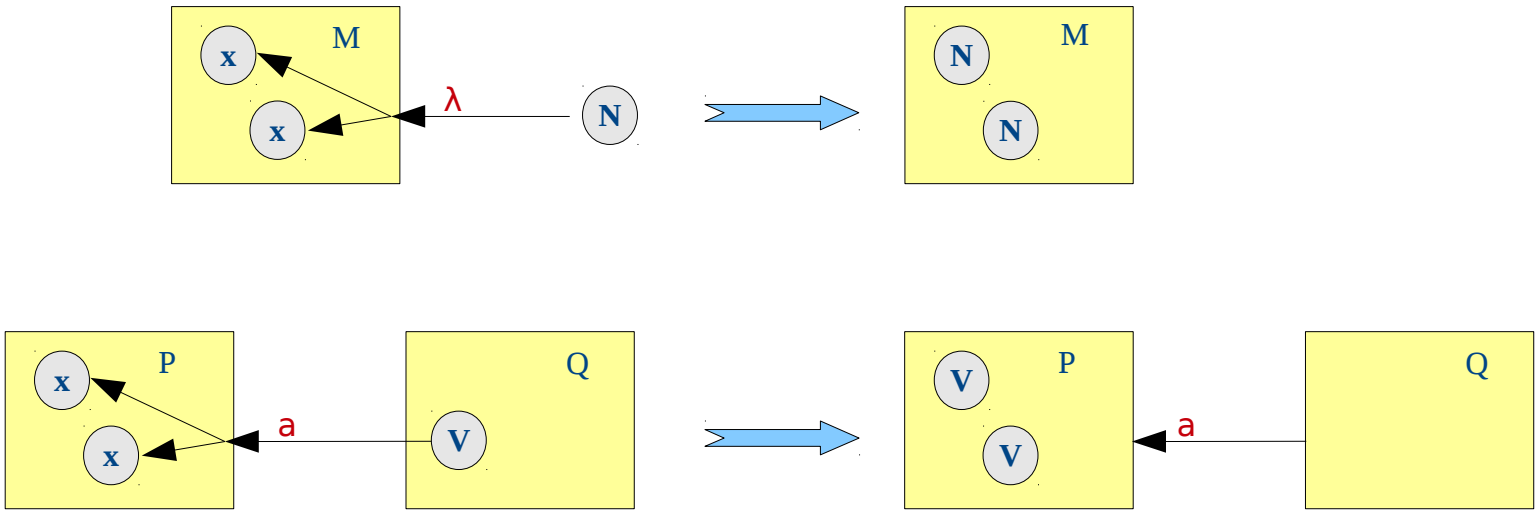


Bild nach: January 1993/Vo1.36, No.1 / COMMUNICATIONS OF THE ACM

Prozess-Kalküle Beispiel Milner

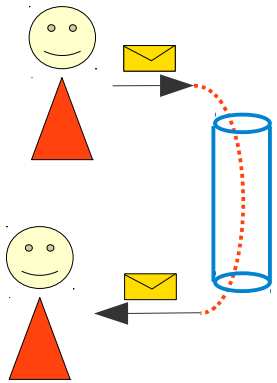
π-Kalkül

Weiterentwicklung von CSS

Keine Unterscheidung Prozess / Wert

Prozesse können erzeugt und kommuniziert werden

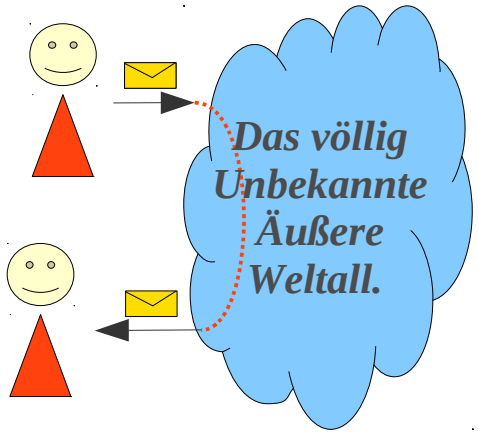
Das Modell besteht nur aus Kanälen und Prozessen



Aktor-Kalkül

Ein Aktor kann

- **Senden:** Nachrichten (die Aktoren sind) zu anderen Aktoren
- **Erzeugen:** Neue Aktiren
- **Ändern:** sein Verhalten, d.h seine Reaktion auf die nächste Nachricht



Zentraler Unterschied zu Prozess-Kalkülen

- Keine Kanäle
- Direkte Kommunikation
- Eine atomare Einheit: Aktor
- Der Nachrichtentransport liegt außerhalb des Modells

  "indeterministic" message delivery: jede Nachricht wird zugestellt, Zeit und Reihenfolge ist völlig unbestimmt

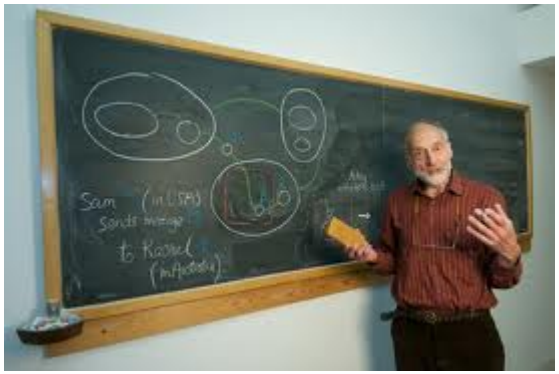
Aktoren in Programmiersprachen haben natürlich einen wohl-definierten Zustellmechanismus!

http://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask

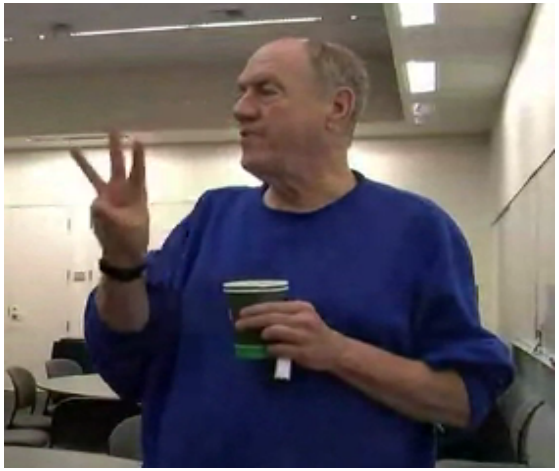
Die Essenz der Berechnungen ist die Essenz der Wirklichkeit

Much of what I have been saying was already well understood in the sixties by Carl-Adam Petri, who pioneered the scientific modeling of discrete concurrent systems. Petri's work has a secure place at the root of concurrency theory.

He declared the aim that his theory of nets should - at its lowest levels - serve impartially as a model of the physical world and as a model of computation. **[R. Milner]**



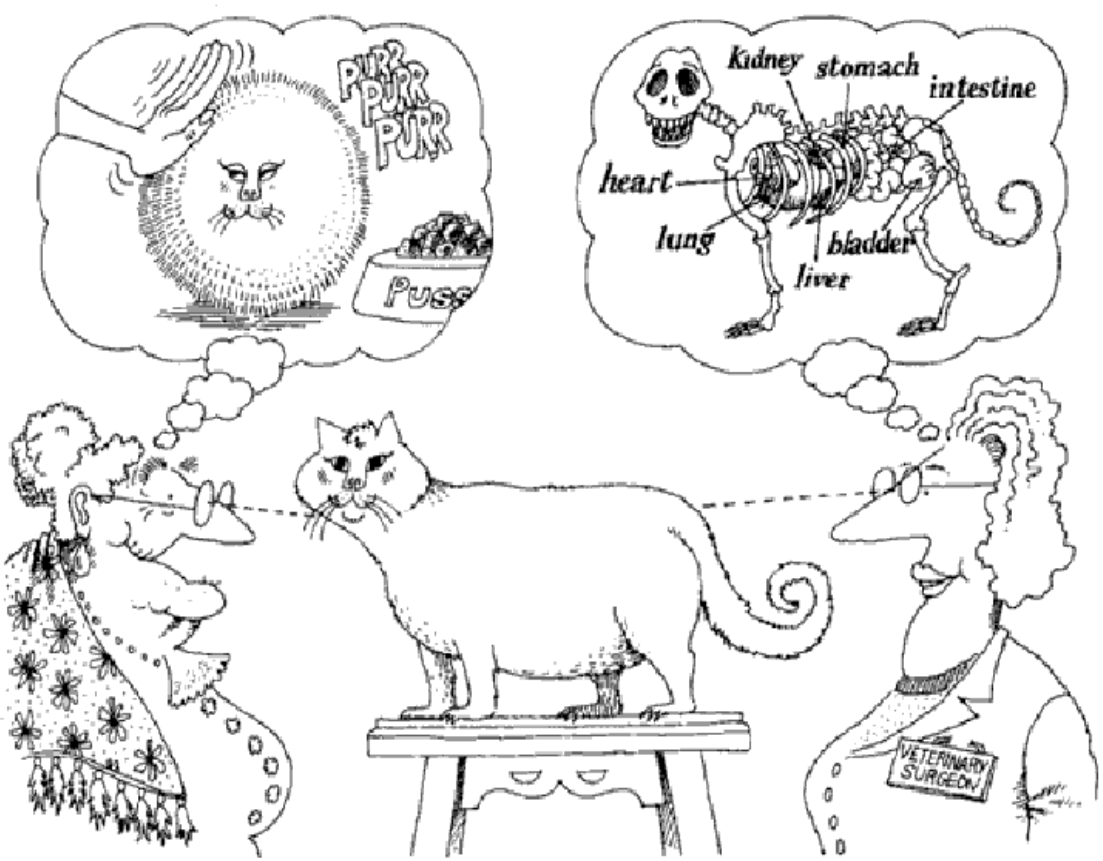
Robin Milner



Carl Hewitt

Für Milner, Hewitt etc. war/ist die Essenz der nebenläufigen Berechnungen gleich der Essenz der physikalischen Realität.

Modellbegriff und Simulation



aus: OBJECT-ORIENTED ANALYSIS AND DESIGN
Grady Booch

Modellbegriffe bei Simulationen

– **Modell** in der Simulation

„Bild /Darstellung“ einer realen Gegebenheit

– **Bezug** zu Berechnungs-Modellen

Berechnungsmodell: die Farbpalette mit der das Bild gemalt werden kann.

D.h. Mit welchen Ausdrucksmitteln kann modelliert werden.

Wirklichkeit

λx. (x = Streichel) ? Schnurr : Fauch

Modell

Erweiterter Lambda-Kalkül

Berechnungsmodell

Berechnungsmodelle für Simulationsmodelle

Eine kaum übersehbare Fülle von Berechnungs-Modellen wie etwa

- Zellular-Automaten
- Differenzialgleichungen
- Petri-Netze
- Prozessalgebren
- Aktorsysteme
- Bayessche Netze
- Statecharts
- . . .

kommt in einer kaum überschaubaren Menge von Systemen zum Einsatz

Beispiel James II

***JA**va-based **M**ultipurpose **E**nvironment for **S**imulation II*
(ehemals Java-based Agent Modeling Environment for Simulation)

Simulationssystem (Framework für die Modellierung, Simulation) der

Arbeitsgruppe *„Modellierung und Simulation“* FB Informatik, Uni Rostock

Leitung Prof. Adelinde M. Uhrmacher

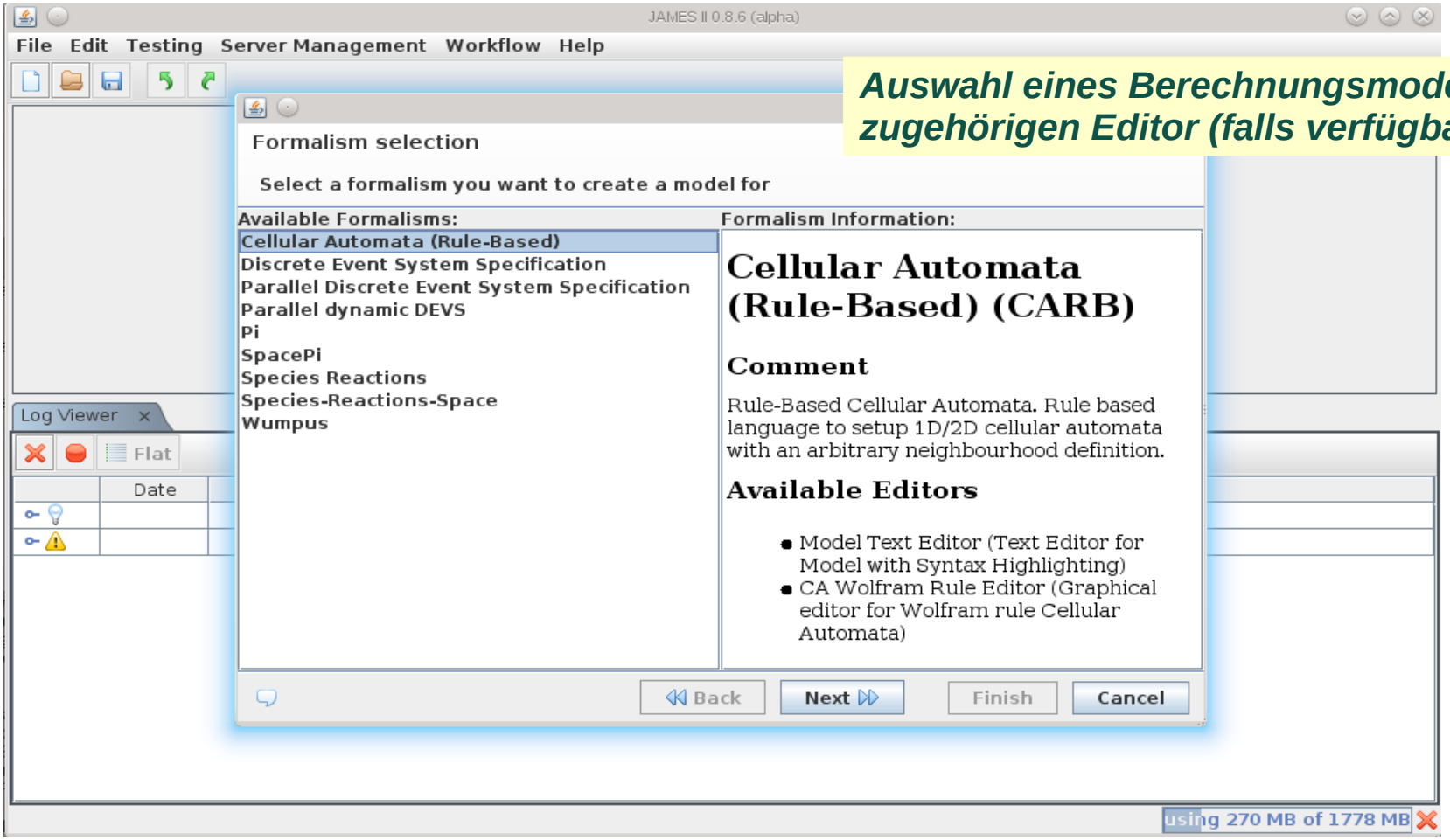
Entwickler : Jan Himmelspach, *et al*

http://wwwmosi.informatik.uni-rostock.de/mosi

Besondere Kennzeichen

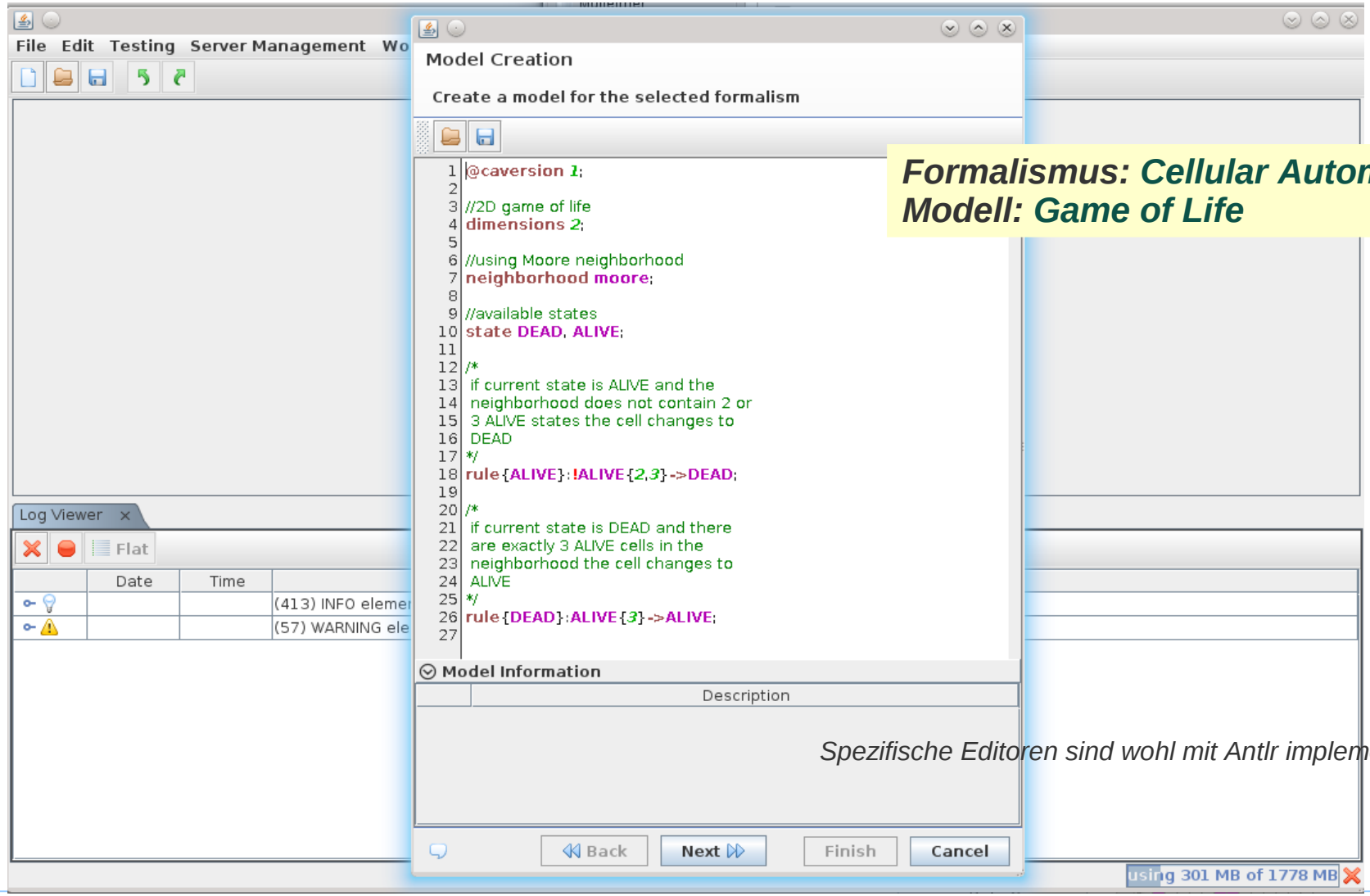
- **Frei verfügbar (mit Quellcode!)**
- **Unterstützt verschiedenen Berechnungsmodelle (Formalismen)**
- **Erweiterbar um neue (eigene) Formalismen**
- **Verschiedene Ausführungsstrategien**

Beispiel James II / Formalismen (Berechnungsmodelle)



Auswahl eines Berechnungsmodells und zugehörigen Editor (falls verfügbar)

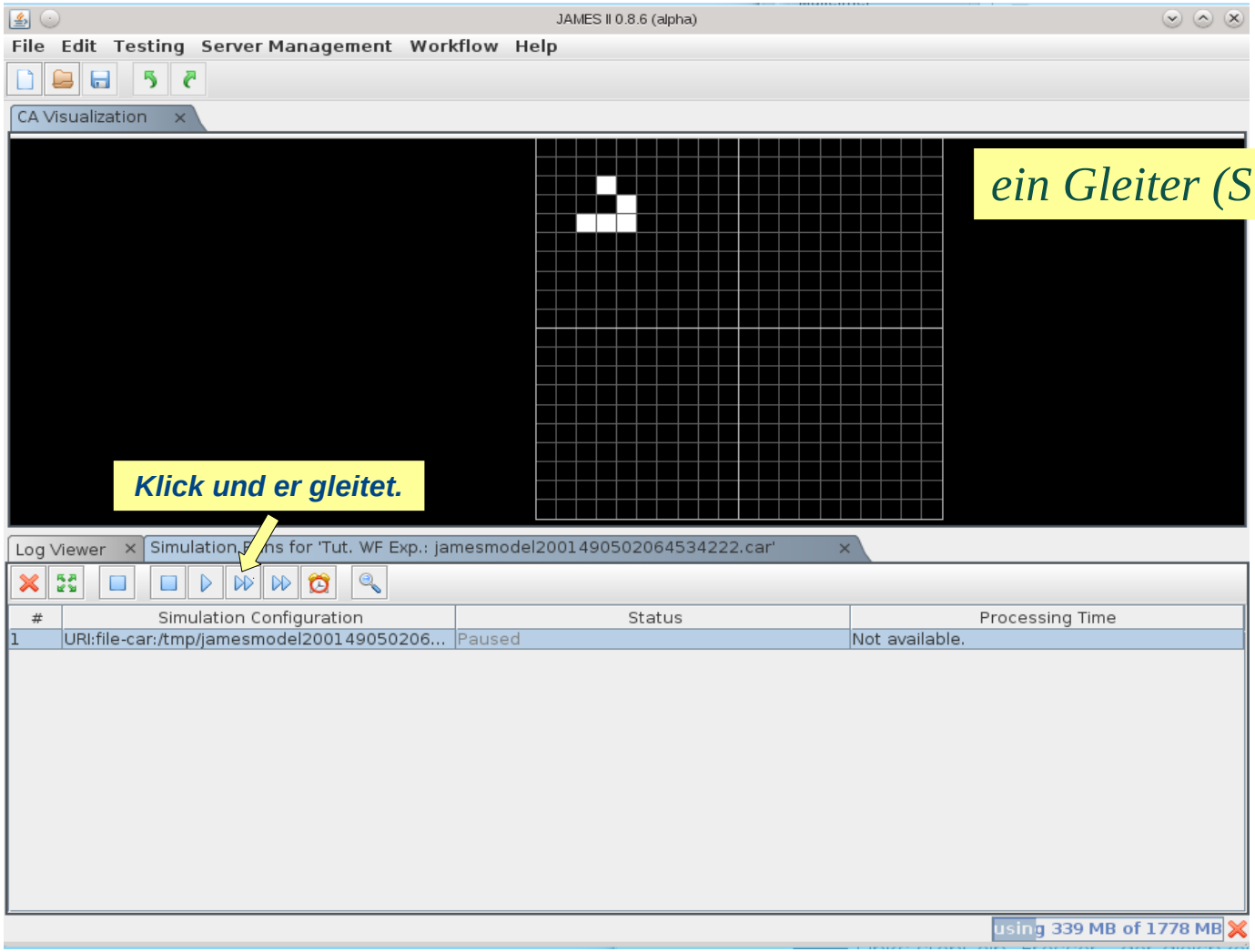
Beispiel James II / Modell



Formalismus: Cellular Automat
Modell: Game of Life

Spezifische Editoren sind wohl mit Antlr implementiert.

Beispiel James II / Experiment (Simulation)



Simulations-Systeme

Systeme zur Formulierung von Modelle und deren Ausführung (Simulation) durch einen Simulator.

Klassifikation der *praktisch relevanten* Systeme

- **(Zeit-) kontinuierliche dynamische Simulation**
  - **(Differential-) Gleichungen mit periodisch veränderten Variablen werden numerisch gelöst**
  - **Typischer Einsatz: Spiele, Ingenieur- und naturwissenschaftliche Anwendungen (Elektrische Schaltkreise)**
  - **Beispiel-System: *Modelica* (https://modelica.org)**
- **(Zeit-) diskrete Ereignis-Simulation**
  - **Modelle werden mit periodisch (u.U. stochastisch) erzeugten Ereignisse konfrontiert und verarbeiten sie (in Submodellen).**
  - **Typischer Einsatz: Ökonomie (wirtschaftliche Prozesse, z.B. Produktion, …)**
  - **Beispiel-System: MATLAB mit Simulink/SimEvents (http://www.mathworks.de/discrete-event-simulation/)**

Nicht systematisch belegte Einschätzung!

Zeitdiskrete Ereignis-Simulation / Berechnungsmodell

DEVS

- *Discrete Event System Specification* DEVS
- Allgemein verbreites und anerkanntes Berechnungsmodell (Formalismus) für zeitdiskrete Ereignis-Simulation
- Autor: Zeigler, Autor diverser Bücher über Simulation
- Charakteristik: Modelle sind atomar oder zusammengesetzt

Atomare Modelle

- Zustände und Zustandsübergänge in Reaktion auf Ereignisse
- Ports (formal Teil des Namens der Ereignisses)
- Zustand + Ereignis + Zeitspanne seit letztem Zustandsübergang
  => neuer Zustand
- Timer-basierte innere Ereignisse

Zusammengesetzte Modelle

- Kombination mehrerer Modelle (atomar oder zusammengesetzt)
- Das Verhalten jedes zusammengesetzten Modells kann als äquivalentes atomares Modell ausgedrückt werden.

Beispiel atomares Modell (DEVS): Job-Verarbeitung

System ist *busy* oder *idle*

In Zustand *busy* werden neue Jobs ignoriert

Im Zustand idle werden Jobs angenommen

und nach der Bearbeitungzeit t_b ausgegeben

Falls kein neuer Job eintrifft verbleibt das System unendlich lange im Zustand *idle*

$S = \{idle, busy\}$

$X = Y = \{Job\}$

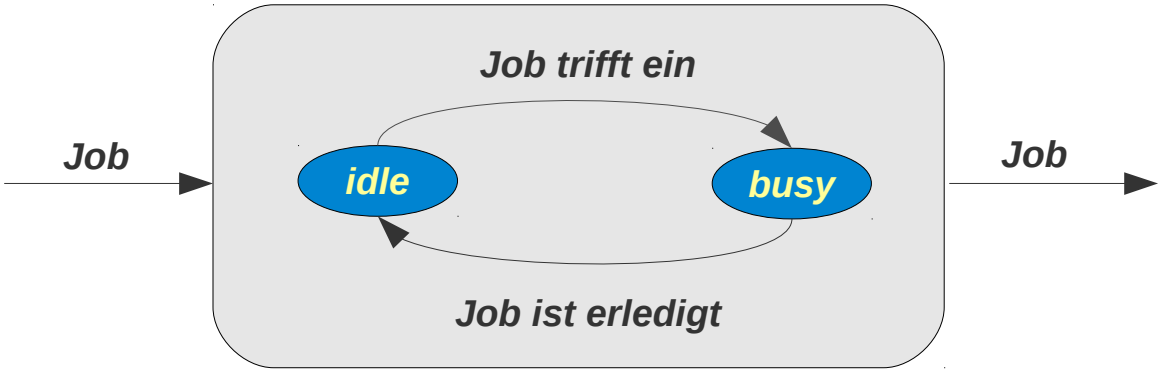
$ta(idle) = \infty$

$ta(busy) = t_b$

$\delta_{int}(busy) = idle$

$\lambda(busy) = Job$

$\delta_{ext}(idle, e, Job) = busy$



DEVS und James II

James II unterstützt DEVS-Modelle

- sie müssen in Java implementiert werden
- das Framework bietet dazu ein Interface das zu implementieren ist

Zeitdiskrete Ereignis-Simulation / Gleichzeitigkeit

PDEVS

– ***Parallel Discrete Event System Specification*** **PDEVS**

– **Erweiterung von DEVS um das Konzept**
gleichzeitig stattfindender Ereignisse

– **Charakteristik**

Modelle sind atomar oder zusammengesetzt

Atomares Modell

- **Wie DEVS, zusätzlich**
- **Gleichzeitig**
  - **an unterschiedlichen oder dem gleichen Port eintreffende externe und**
  - **interne**

Ereignisse können explizit modelliert / behandelt werden.

Zusammengesetzt

- **Wie DEVS**

Exkurs: Gleichzeitigkeit

Gleichzeitigkeit

Formale Berechnungsmodelle für verteilte Systeme haben üblicherweise kein Konzept der Gleichzeitigkeit.

- **Zeit ist kein Bestandteil des Berechnungsmodells**
- **Globale Zeit darf kein Bestandteil des Berechnungsmodells sein**
- **Gleichzeitig: definierbar als Abwesenheit kausaler Abhängigkeiten**
- **Gleichzeitig eintreffende Ereignisse ~ Reihenfolge ist ohne Belang**

Gleichzeitigkeit

Notationen und ihre Interpretation

```
Prozess Adder {
  PortIn  p1: Integer
  PortIn  p2: Integer
  PortOut p3: Integer

  Ready(p1, p2) =>
     p3.send (
        p1.receive()
        +p2.receive());
}
```

Interpretationen:

- **an Port p1 und p2 trifft gleichzeitig ein Wert ein**
  ***Unsinnig** bei undefinierter Gleichzeitigkeit also in jedem Berechnungsmodell.*
  ***Sinnvoll** bei Zeit als Modellbestandteil, z.B. bei Simulationen.*

- **Gleichwertige Zustandsübergänge die jeweils mit einem Wert an p1 bzw. p2 beginnen**
  Sinnvolle abgekürzte Notation für zwei Verhaltensweisen deren Reihenfolge irrelevant ist.

Atomare Modelle

Übersicht formale Definition des Modells

- Zustände S
- Externe Eingaben X
- Ausgaben Y
- $\delta_{int}: S \longrightarrow S$ interne Transition
- $\delta_{ext}: Q \times X^b \longrightarrow S$ externe Transition
- $\delta_{con}: S \times X^b \longrightarrow S$ konfluente Transition (PDEVS)
- $\lambda: S \longrightarrow Y$ Ausgabefunktion
- $ta: S \longrightarrow R_0^+ \cup \{\infty\}$ Zeitfortschrittsfunktion

mit:

- $Q = \{ (s,e) \mid s \in S, 0 \leq e \leq ta(s) \}$ e ~ e*lapsed time*
- X^b : die Menge der Multimengen über X
- R_0^+ : die positiven reellen Zahlen inklusive 0

Gleichzeitig eintreffende Ereignisse sind explizit modellierbar!

Details des atomaren Modells

$\delta_{int}: S \longrightarrow S$ **interne Transition**

Jeder Zustand s hat eine interne Verweilzeit $ta(s) \in R_0^+ \cup \{\infty\}$

wenn diese Zeit ohne externe Ereignisse verstrichen ist, dann wird der Zustandsübergang $\delta_{int}(s)$ ausgeführt

$\delta_{ext}: Q \times X^b \longrightarrow S$ **externe Transition**

Ein externer Zustandsübergang hängt ab von

- den externen Ereignissen $x \in X^b$
- dem aktuellen Zustand s und
- der seit der letzten Transition vergangene Zeit e ($0 \leq e \leq ta(s)$)

$ta: S \longrightarrow R_0^+ \cup \{\infty\}$ **Zeitfortschrittsfunktion**

Jedem Zustand s ist eine Zeit ta(s) zugeordnet nach der nächste interne Übergang erfolgt

$\lambda: S \longrightarrow Y$ **Ausgabefunktion**

bestimmt die Ausgabe $y \in Y$ die beim Verlassen von s durch eine interne Transition erzeugt wird. (Nur interne Transitionen erzeugen Ausgaben)

Details des atomaren Modells

$\delta_{con}: S \times X^b \longrightarrow S$ konfluente Transition

Die konfluente Transition wird ausgeführt, wenn ein interner und ein externe Übergang gleichzeitig ausgeführt werden könnten. Das ist der Fall, wenn e = ta(s)

möglich ist

- $\delta_{ext}((s,ta(s)), x)$
- $\delta_{int}(s)$

Führe aus: $\delta_{conf}(s, x)$

Mögliche Default-Lösung ohne explizite Definition von δ_{con}

$$\delta_{con}((s, x) = \delta_{ext}((\delta_{int}(s),0),x)$$

interne dann externe Transition

die interne Transition $\delta_{int}(s)$ wird ausgeführt und dann wird in neuen Zustand mit der Verweilzeit 0 die externe Eingabe x verarbeitet.

oder

$$\delta_{con}((s, x) = \delta_{int}(\delta_{ext}((s,ta(s)),x))$$

externe dann interne Transition

die externe Transition $\delta_{ext}((s,ta(s)),x)$ wird mit der maximalen Wartezeit ausgeführt und dann wird in neuen Zustand sofort die interne Transition ausgeführt.

Übersicht atomaren Modell

$\delta_{int}: S \longrightarrow S$ interne Transition

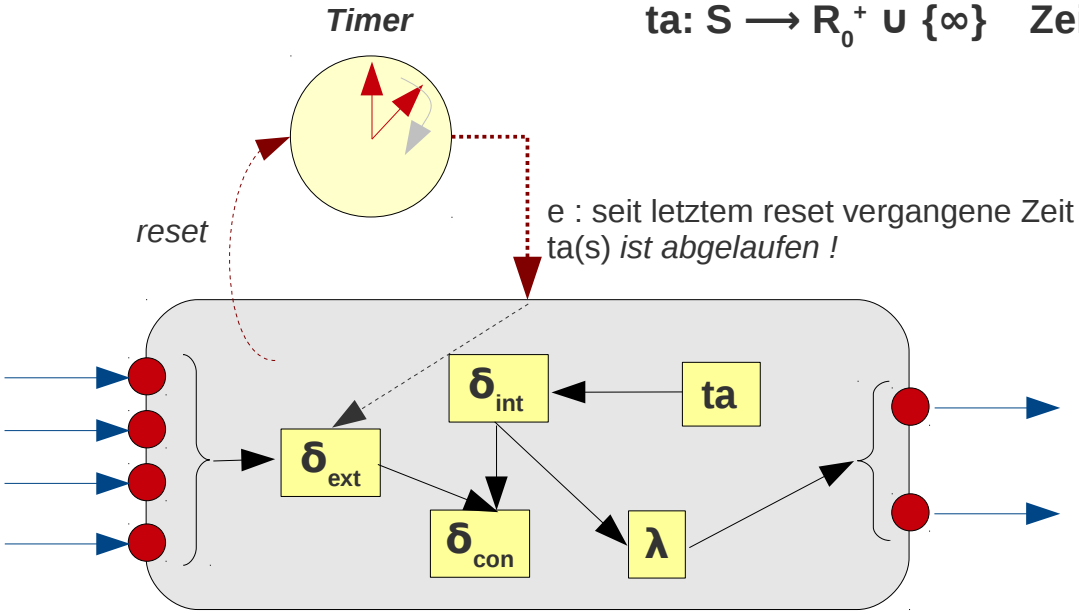
$\delta_{ext}: Q \times X^b \longrightarrow S$ externe Transition

$q \in Q = S \times (R_0^+ \cup \{\infty\})$

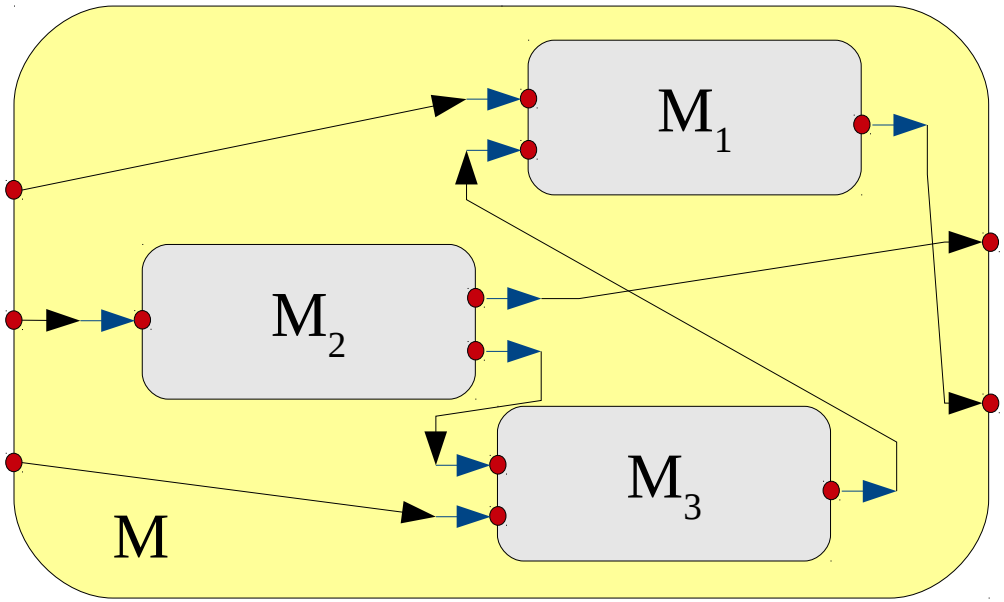
$\delta_{con}: S \times X^b \longrightarrow S$ konfluente Transition

$\lambda: S \longrightarrow Y$ Ausgabefunktion

$ta: S \longrightarrow R_0^+ \cup \{\infty\}$ Zeitfortschrittsfunktion



gekoppelte Modelle



Bei der Kopplung werden Ports miteinander verknüpft bzw. zu Ports des Gesamtmodells erklärt. (Technische Details uninteressant)

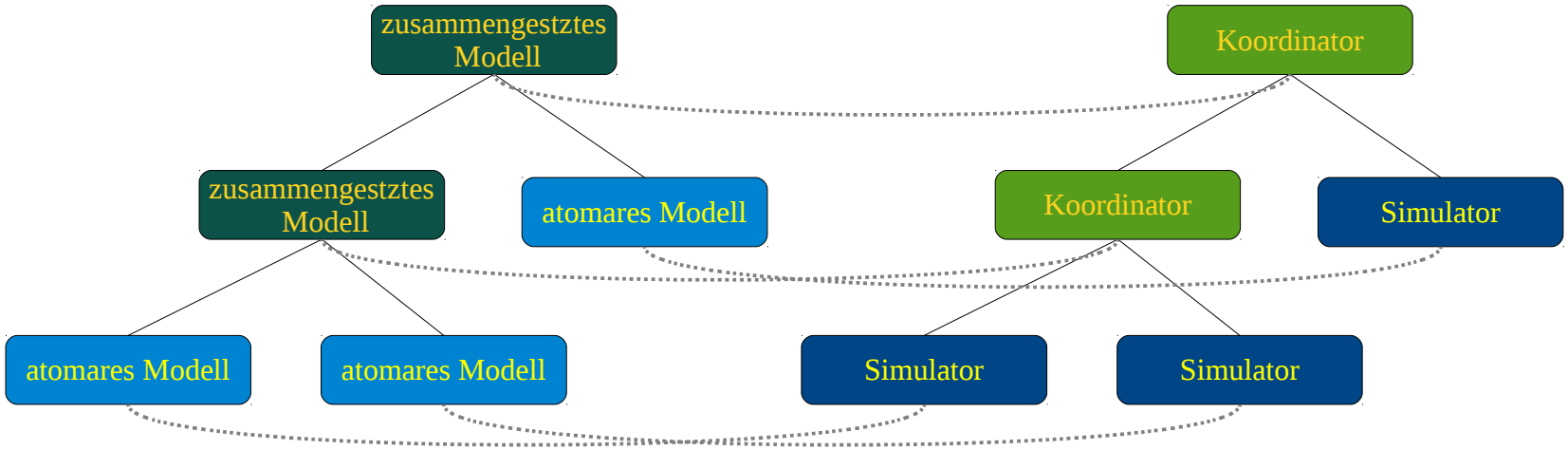
Semantik: Modell ~ abstrakter Simulator

*Die **Semantik** der Modelle wird durch einen abstrakten Simulator definiert*

*Der **abstrakte Simulator** beschreibt das Verhalten korrekter realer Simulatoren*

- *Simulatoren führen atomare Modelle aus*
- *Koordinatoren koordinieren die Aktionen in zusammengesetzten Modellen*

***Simulation**: Ausführung durch einen konkreten Simulator*



Simulation

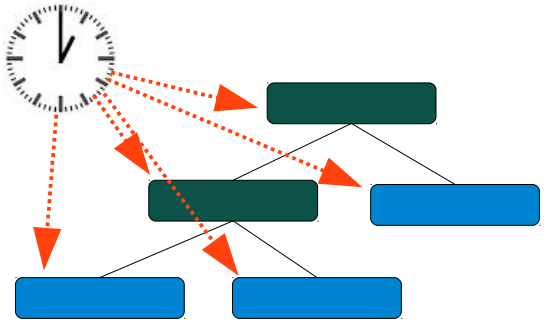
Ausführen eines Modells durch eine Simulator

Prinzipielle Varianten der Ausführung

– **Zeitgesteuert**

Uhr(en) löst / lösen zu jedem Zeitpunkt die diesem zugeordneten Ereignisse aus.

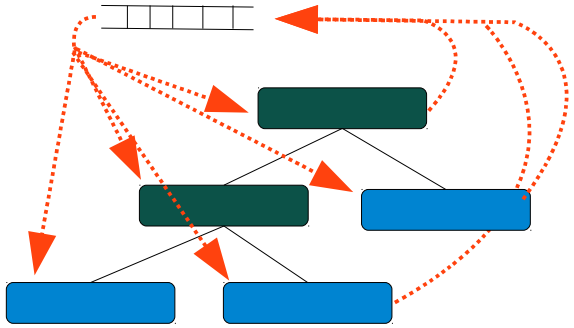
unüblich, aufwändig, speziell bei Modelle mit kontinuierlicher Zeit z.B. (P)DEVS



– **Ereignisgesteuert**

nach Zeiten priorisierte Warteschlange(n) von Ereignissen wird / werden abgearbeitet.

Übliches Verfahren.



Ereignis-gesteuerte Simulation

Monolithische Lösung

Ein Prozess arbeitet eine Warteschlange ab.

Ein verteiltes / paralleles Modell muss nicht naturgemäß verteilt/parallel ausgeführt werden!

Verteilte / Parallele Lösung

Protokoll

zwischen verteilten aktiven Simulatoren mit lokaler Ereignis-Verwaltung

Problem

Lokal trifft zur (Simulations-) Zeit t_x Ereignis x ein und löst eine Transition aus, aber eventuell wird noch zur (Simulations-) Zeit t_y ein Ereignis y eintreffen mit $t_y < t_x$

Local Causality Constaint*: Verarbeite ein Ereignis erst dann, wenn sicher ist dass kein (in der Simulationszeit) früheres Ereignis eintreffen wird.*

Lösungsvarianten

- **Konservativ**

  Ein Ereignis wird erst ausgelöst, wenn sicher ist, dass früheres auftreten kann
- **Optimistisch mit *Rollback***

  Vorweggenommene Ereignisse werden zurück genommen

Local Causality Constaint

Die Verarbeitungsreihenfolge von Ereignissen muss konsistent mit der Simulationszeit sein.

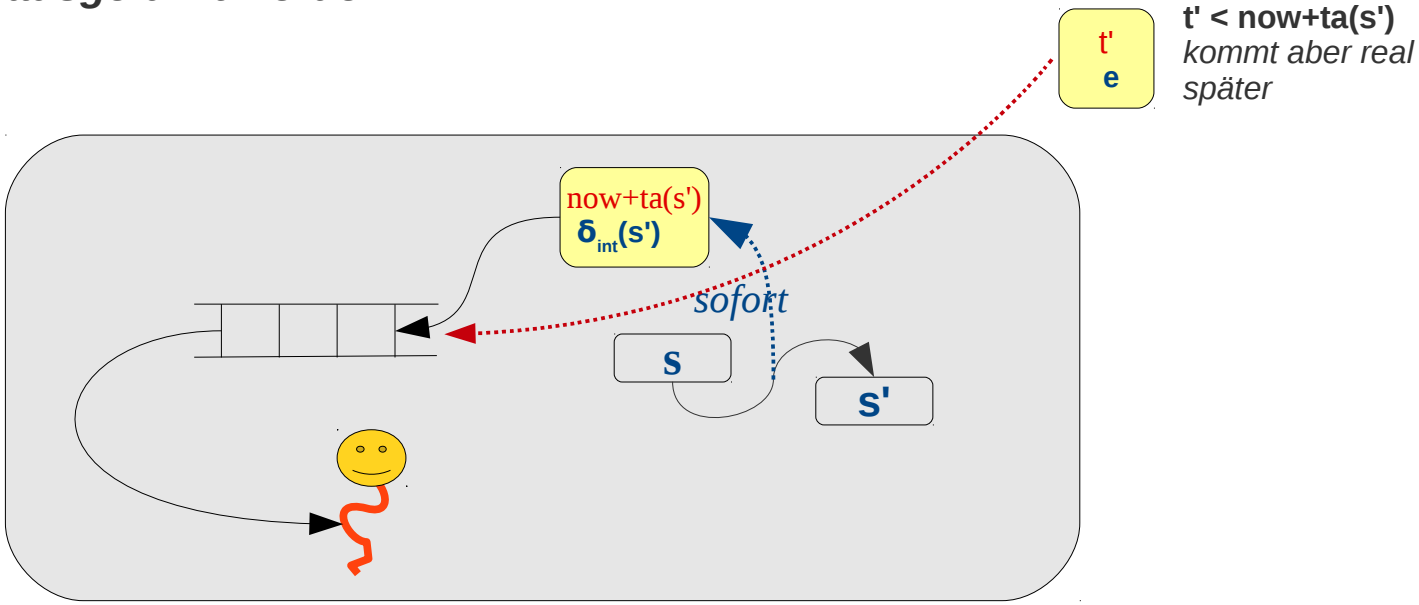
Triviales Beispiel in DEVS:

ta(s) = Timeout für den Zustand s

~> Zustandsübergang s ⟶ s'

definiert „Timeout-Transition“ $\delta_{int}(s')$ zum Zeitpunkt *jetzt*+ta(s')

In einem Ereignis-gesteuerten System kann dieser Übergang sofort ausgeführt werden.



Aktuelle Herausforderungen

Anwendungsgebiet: Technik / Energiemanagement: Hybride Systeme

Kombination von „mathematischen“ Systemen und diskreter Ereignissimulation wichtig. (Wechselstromnetze sind gut durch DGLs beschreibar.)

Gegenseitiges Emulieren scheint möglich zu sein:

- **Modellica (DGL basiert) emuliert DEVS**

  z.B.: Sanz, Urquia, Dormido:
  Parallel DEVS and Process-Oriented Modeling in Modelica

- **DEVS emuliert DGL-Systeme**

  z.B.: Kofman, Lee, Zeigler:

  DEVS Representation of Differential Equation Systems

Aktuelle Herausforderungen

Anwendungsgebiet: Systembiologie

Systembiologie (Computational Biology) ist aktuell das anspruchsvollste Anwendungsgebiet für Modellierung und Simulation

- **Hierarchische / Hybride Modelle**
  (Molekül, Protein, Zelle, Gewebe, Organ, Organismus)
- **Selbst-modifizierende Systeme (Wachstum, Teilung)**
  Verwendung dynamischer Formalismen
  - **DEVS Erweiterungen**
    1. **mlDEVS multilevel DEVS**
    2. **dynDEVS (Struktur-Änderungen während der Simulation)**
    3. **ρ-DEVS (zusätzlich Änderungen der Ports)**
  - **π-Kalkül und Abwandlungen**
  - **Term-Ersetzungssysteme**
  - **Graph-Ersetzungssysteme**
  - **....**

Simulation : Beispiel Systembiologie

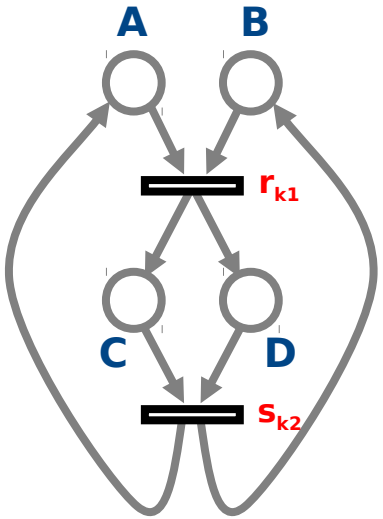
Luca Cardelli:
„Abstract State Machines of System Biology“
http://lucacardelli.name/

Systembiologie

Modellbildung Chemische Reaktion

~ Transition in Petri-Netz

~ Prozess-Kommunikation



Petrinetz

r: A + B $\rightarrow_{k1}$ C + D

s: C + D $\rightarrow_{k2}$ A + B

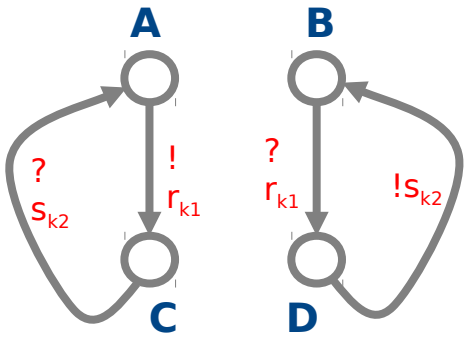
Chemie

A = $!r_{k1}$; C

C = $?s_{k2}$; A

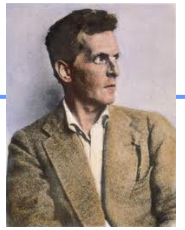
B = $?r_{k1}$; D

D = $!s_{k2}$; B



π-Kalkül

Schluss: Wozu?



„Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt“.

Modelle – Wozu ?

Formale Modelle in der Berechnungstheorie:

- dienen dazu den Diskussionsgegenstand einzugrenzen
- dienen dazu das innere Wesen der Dinge zu erfassen

  was ist das Einfachste das alle realen Probleme in sich birgt

Formale Modelle in der Simulation:

- Simulationsmodelle dienen dazu reale Dinge zu beschreiben
- Simulationsmodelle basieren auf (expliziten oder impliziten) formalen Modellen
- explizite formale Modelle sind sinnvoll
  - um Modelle einfach
  - und mit definierter Semantik

  beschreiben zu können
- Bieten vielfältige theoretische und praktische Fragestellungen mit spannenden Anwendungen

Formale Modelle der Berechnungstheorie und Simulationssysteme

- überlappen sich
- haben aber unterschiedliche Zielsetzungen