Modellierung biologischer Prozesse

Christian Maidorfer

Thomas Zwifl

(Seminar aus Informatik)

Überblick

- Einführung
- Arten von Modellen
- Die stochastische Pi-Maschine

Warum Modelle

- Die Biologie konzentriert sich auf die einzelnen Elemente eines Systems.
- Aber was geschieht fügt man nun diese Elemente zu einem Ganzen zusammen.

Randbedingungen

- Biologische Systeme weisen hohen Komplexitätsgrad auf.
- Modelle von unterschiedlichen Teilen des Systems müssen zusammenarbeiten können
- Aktuelle Forschung erzeugt Unmengen von Daten die analysiert werden müssen durch:
  - Genom Sequenzierung
  - Protein Interaktionskarten
  - DNA Microarrays
  - …

Eine einfache Kohn Karte

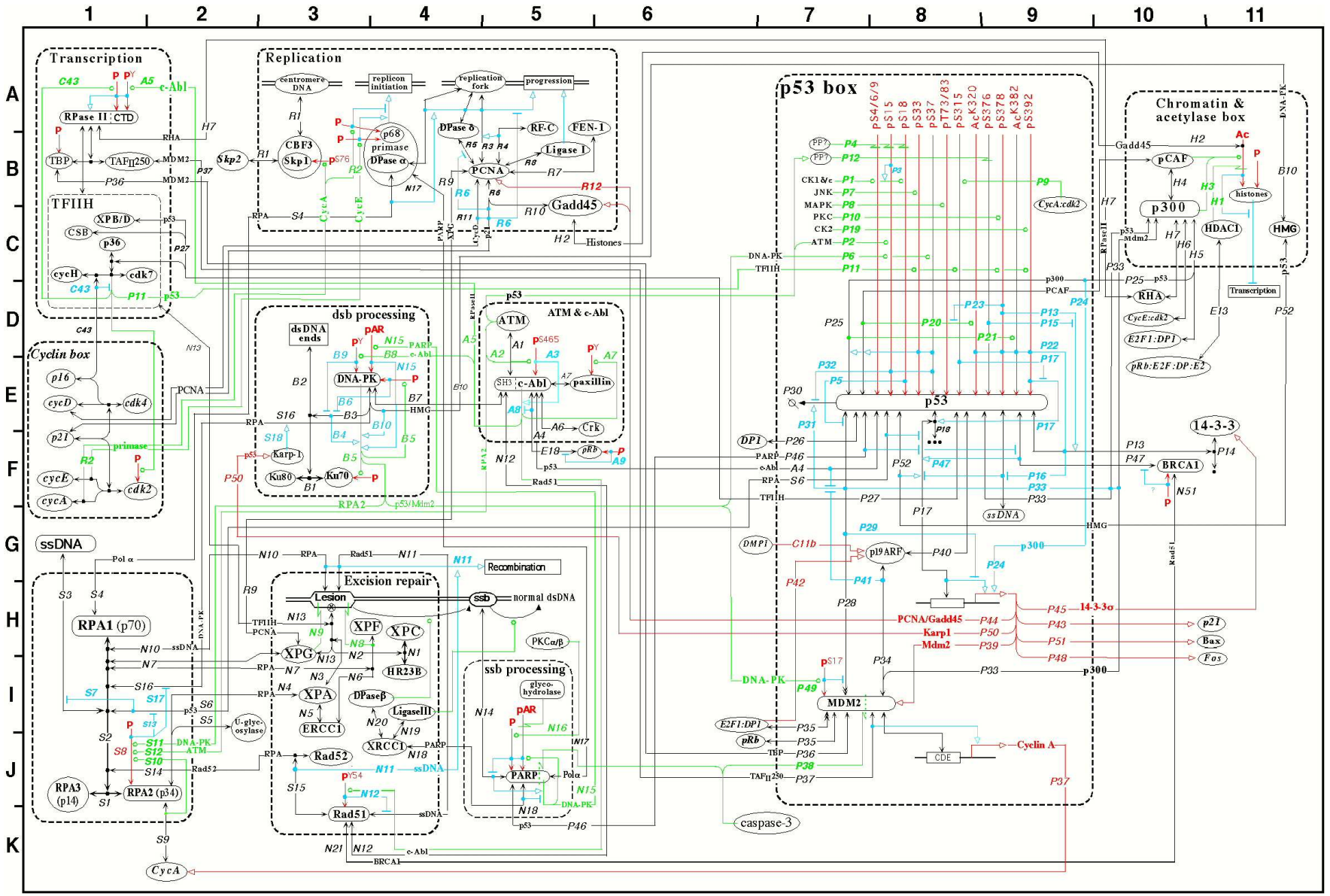


Figure 6B: The p53-Mdm2 and DNA repair regulatory network (version 2p - May 19, 1999)

Arten von Modellen

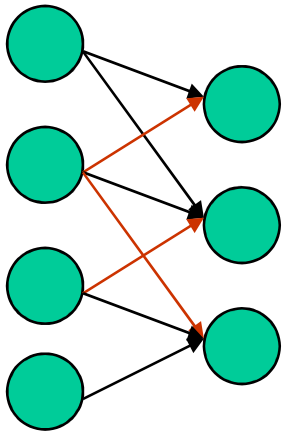
- Quantitative Modelle:
  - Mathematische Modelle
- Qualitative Modelle:
  - Boolsche Netze
  - Petri-Netze
  - Modelle basierend auf Prozessalgebras

Mathematische Modelle

- In manchen Bereichen stark verbreitet (z.B.: Physik)
- Modelle haben quantitativen Charakter, d.h. sie beschreiben quantitative Beziehungen zwischen Systemteilen
- Bestehen meist aus (Systeme von) Differentialgleichungen
- Problem der Lösbarkeit
- Sehr genaue Kenntnis über System nötig

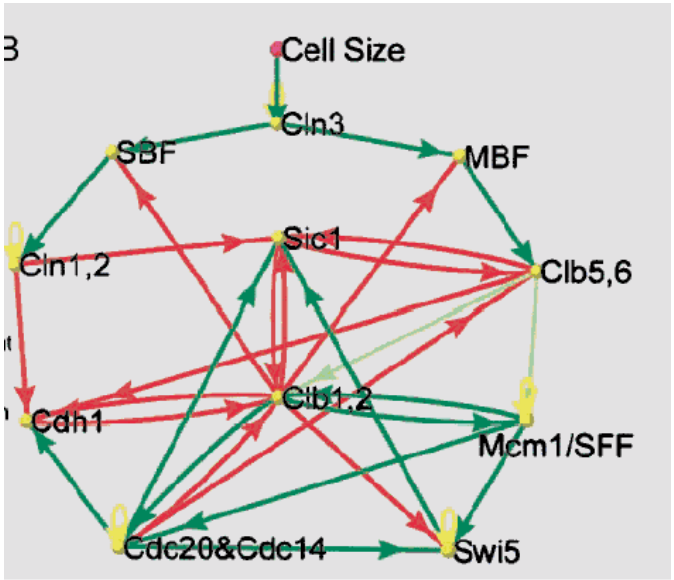
Boolsche Netzwerke (1)

- Ein boolsches Netzwerk ist ein gerichteter Graph
- Menge von boolschen Variablen wird dargestellt
- Variablen werden durch Knoten modelliert
- Abhängigkeiten durch Kanten



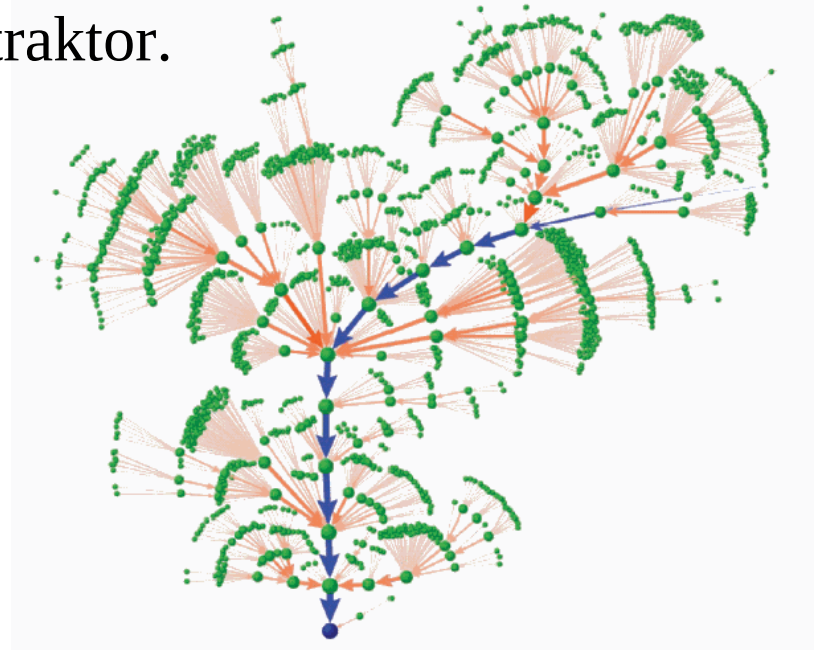
Boolsche Netzwerke (2)

- Boolsche Netzwerke werden z.B. zur Modellierung von Genregulierung verwendet.
- Knoten entspricht einem Gen
- Zustände der Knoten entsprechen Exprimierung oder Inhibition des Gens



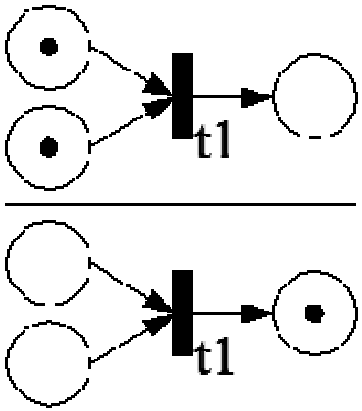
Boolsche Netzwerke (3)

- Da Boolsche Netzwerke Deterministisch sind, gibt es nur eine begrenzte Anzahl an Zuständen. (2^N bei N Knoten)
- Wird ein bereits erreichter Zustand wieder erreicht, entsteht ein Zyklus. Die Menge an Zuständen in solch einem Zyklus nennt man Attraktor.



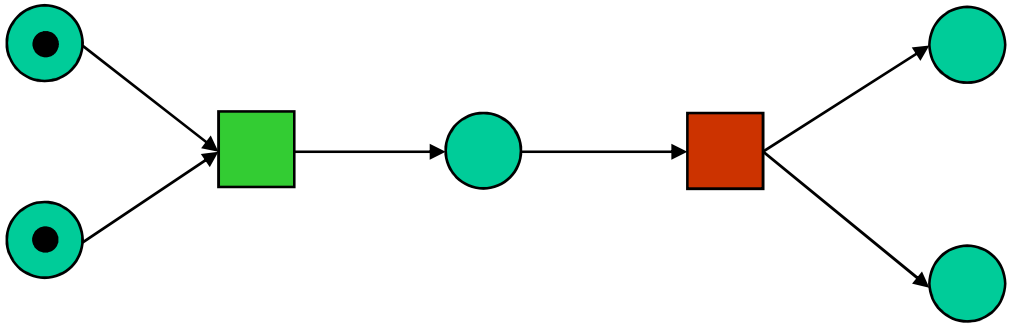
Petri Netze

- Ein Petri Netz besteht aus zwei verschiedenen Typen von Knoten:
  - Stellen (Places)
  - Übergänge (Transitions)
- Kanten gehen entweder von Stellen zu Übergängen oder umgekehrt
- Zustand des Netzes wird durch vorhandene Marken an den Stellen definiert



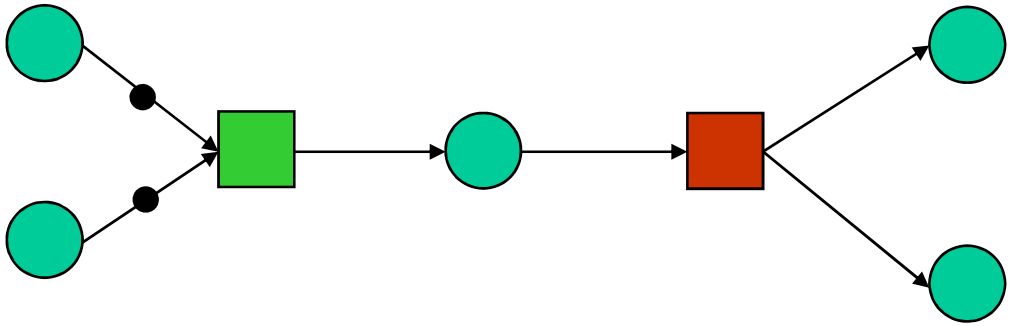
Petri Netze (2)

- Stellen können unterschiedliche Kapazitäten an Marken haben
- Alle Transitionen, dessen eingehende Stellen Marken besitzen, sind aktiv
- Der Status des Netzes ändert sich, wenn Transitionen feuern und Marken entfernt und erzeugt werden.



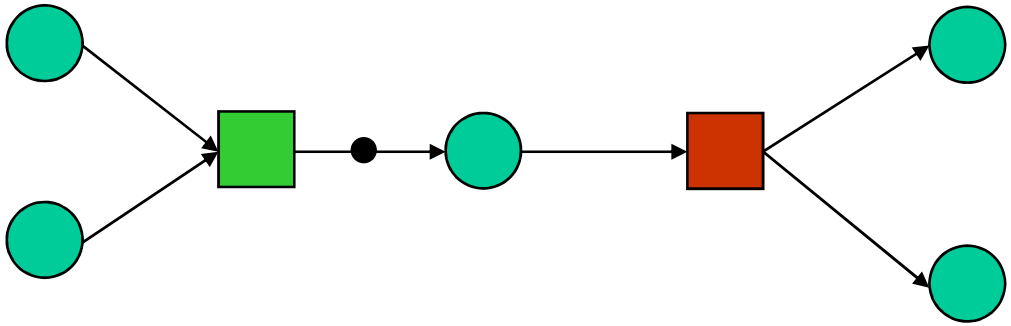
Petri Netze (2)

- Stellen können unterschiedliche Kapazitäten an Marken haben
- Alle Transitionen, dessen eingehende Stellen Marken besitzen, sind aktiv
- Der Status des Netzes ändert sich, wenn Transitionen feuern und Marken entfernt und erzeugt werden.



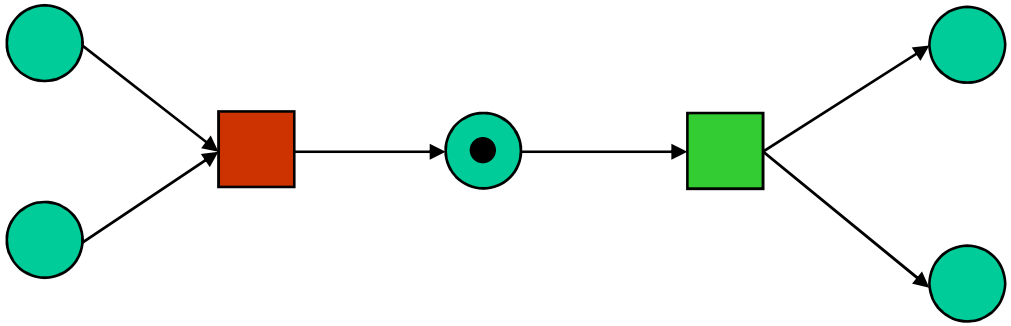
Petri Netze (2)

- Stellen können unterschiedliche Kapazitäten an Marken haben
- Alle Transitionen, dessen eingehende Stellen Marken besitzen, sind aktiv
- Der Status des Netzes ändert sich, wenn Transitionen feuern und Marken entfernt und erzeugt werden.



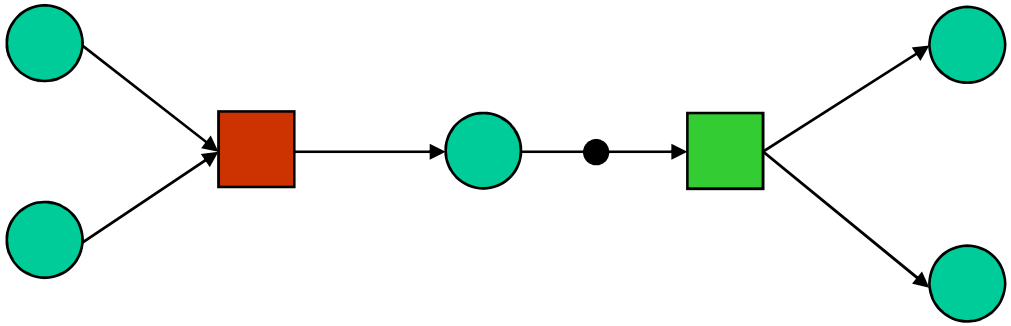
Petri Netze (2)

- Stellen können unterschiedliche Kapazitäten an Marken haben
- Alle Transitionen, dessen eingehende Stellen Marken besitzen, sind aktiv
- Der Status des Netzes ändert sich, wenn Transitionen feuern und Marken entfernt und erzeugt werden.



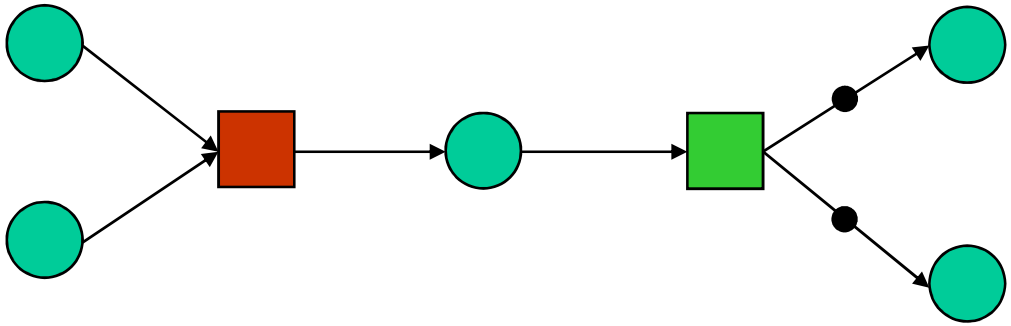
Petri Netze (2)

- Stellen können unterschiedliche Kapazitäten an Marken haben
- Alle Transitionen, dessen eingehende Stellen Marken besitzen, sind aktiv
- Der Status des Netzes ändert sich, wenn Transitionen feuern und Marken entfernt und erzeugt werden.



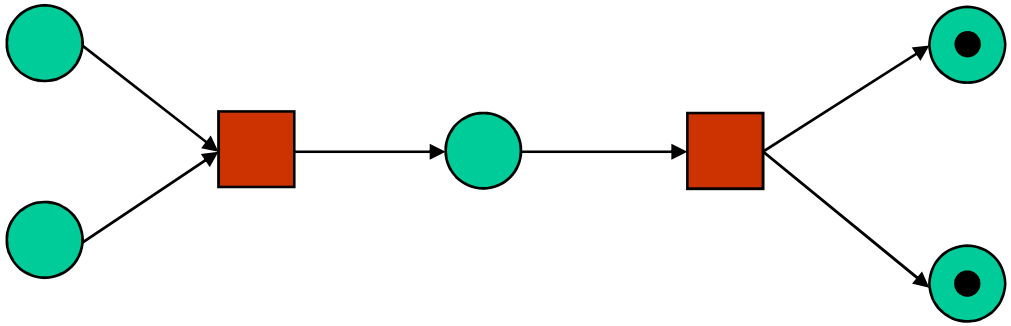
Petri Netze (2)

- Stellen können unterschiedliche Kapazitäten an Marken haben
- Alle Transitionen, dessen eingehende Stellen Marken besitzen, sind aktiv
- Der Status des Netzes ändert sich, wenn Transitionen feuern und Marken entfernt und erzeugt werden.



Petri Netze (2)

- Stellen können unterschiedliche Kapazitäten an Marken haben
- Alle Transitionen, dessen eingehende Stellen Marken besitzen, sind aktiv
- Der Status des Netzes ändert sich, wenn Transitionen feuern und Marken entfernt und erzeugt werden.



Hybride Modelle

- Mischung zwischen mathematischen und computational Models
- Zustände und kontinuierliche Variablen

Prozessalgebras

- Formale Theorien zu Systemen parallel ablaufender Prozesse
- Ursprünglich für technische Systeme
- Vorteile von Prozessalgebras:
  - Komplexität, Nebenläufigkeit, Änderung der Systemtopologie, Interaktion, Zufälligkeit und Nichtdeterminismus

PI-Kalkül Prozessalgebra

Seien P, Q Prozesse und ein Kanal eine Abstraktion einer Kommunikationsverbindung zwischen Prozessen.

Syntax	
0	Prozess der nichts macht
P | Q	Prozess in dem P und Q parallel ablaufen (Parallele Komposition)
P + Q	Prozess verhält sich wahlweise wie P oder Q
!P	unendliche Anzahl von parallel laufenden Prozessen P. (Replikation)
a(x).P	Prozess liest x von Kanal a und verhält sich danach wie P (Input-Prefix)
$\bar{a}$<x>.P	Prozess sendet x über Kanal a und verhält sich danach wie P (Output-Prefix)
τ.P	Prozess geht direkt auf P über ohne Lesen oder Schreiben (Silent-Prefix)
(νa)P	Erzeugt eine neuen lokalen Kanal in P (Restriktion)
A($y_1, ..., y_n$)	Identifier
A($y_1, ..., y_n$) $=^{def}$ P	Definition (für $i \neq j \Rightarrow y_1 \neq y_j$)
if $x = y$ then P	Match
if $x \neq y$ then P	Mismatch

Beispiel

- Nehmen wir an wir wollen einen RPC zwischen Client und Server modellieren. Am Server ist die Funktion **addOne** implementiert:
- *int* function **addOne**(*int* x){return x+1}
- Clientcode: y = **addOne**(1)

Beispiel

- Server:

```
int function addOne(int x){return x+1}
```

in Pi-Kalkül-Notation:

$$!addOne(a, x).\bar{a} < x + 1 >$$

- Client:

```
y = addOne(1)
```

in Pi-Kalkül-Notation:

$$(\nu a)(\overline{addOne} < a, 1 > |a(y))$$

Der gesamte Prozess

$$!addOne(a, x).\bar{a} < x + 1 > |(\nu a)(\overline{addOne} < a, 1 > |a(y))$$

Stochastic Pi-Machine (SPiM)

- Ist ein Simulations-Framework
- Führt Modelle in SPiM-Language aus
- Die Sprache basiert auf Pi-Kalkül
- Syntax an F# angelehnt

SPiM - Definition		
Program ::=	$Directive_1 ... Directive_M$ $Declaration_1 ... Declaration_N$	Directives $M \geq 1$ Declarations $N \geq 1$
Directive ::=	*directive sample* Float *Integer* | *directive graph* | *directive plot* $point_1 ... point_n$	Sample Directive Graph Directive Plot Directive
Point ::=	!Channel {*as* String} | ?Channel {*as* String} | Name $(Value_1, ..., Value_N)$ {*as* String}	Output Point Input Point Process Point, $N \geq 0$
Declaration ::=	*new* Name{@Value}:Type | *type* Name = Type | *val* Pattern = Value | *run* Process | *let* $Definition_1$ *and . . . and* $Definition_N$	Channel Declaration Type Declaration Value Declaration Process Declaration Definitions, $N \geq 1$
Definition ::=	Name $(Pattern_1, ..., Pattern_N)$ = Process	Definition, $N \geq 0$
Process ::=	() | $(Process_1 \| ... \| Process_M)$ | Name $(Value_1, ..., Value_N)${; Process} | ActionProcess | *do* $ActionProcess_1$ *or . . . or* $ActionProcess_M$ | *replicate* ActionProcess | *if* Value *then* Process {*else* Process} | *match* Value *case* $Case_1$ *. . . case* $Case_N$ | Integer *of* Process | $(Declaration_1 ... Declaration_N Process)$	Null Process Parallel, $M \geq 2$ Instantiation, $N \geq 0$ Action Process Choice, $M \geq 2$ Replicated Action Conditional Process Matching, $N \geq 1$ Repetition Nested Declarations, $N \geq 1$
Case ::=	V alue -> Process	Match Case
ActionProcess ::=	Action{; Process}	Action Process
Action ::=	*delay*@Value | !Channel $\{(Value_1, ..., Value_N)\}$ {*Value} | ?Channel $\{(Pattern_1, ..., Pattern_N)\}$ {*Value}	Delay Output, $N \geq 0$ Input, $N \geq 0$

Pattern ::=	_ | Name{:Type} | $(Pattern_1, ..., Pattern_N)$	Wildcard Pattern Name Pattern Patterns, $N \geq 0$
V alue ::=	(Integer | Float | Charakter | String) | (*true* | *false*) | *int_of_float* | *float_of_int* | *sqrt* |Name | *show* Value | -Value | Value (+ | - | * | /) Value | Value (= | <> | < | > | <= | >=) Value | [] | Value::Value | $(Value_1, ..., Value_N)$	int, float, char, string Values Boolean True / False Float to Integer Integer to Float Square Root Variable String Representation Negation Arithmetical Ops Comparison Empty List List Value Values, $N \geq 0$
Type ::=	(*string* / *int* / *float* / *char* / *bool*) | Name | 'Name | *chan*{ $(Type_1, ..., Type_N)$} | *proc* $(Type_1, ..., Type_N)$ | $Type_1 \| ... \| Type_M$ | *list*(Type) | $(Type_1, ..., Type_N)$	string, int, float, char, bool Types Type Variable Polymorphic Type Channel Type, $N \geq 0$ Process Type, $N \geq 0$ Data Type, $M \geq 2$ List Type Types, $N \geq 0$

SPiM-Beispiel

Beispiel: Wir nehmen an wir modellieren Natrium- und Chlor-Moleküle die sich gegenseitig ionisieren bzw. deionisieren. Welches Na-Molekül mit welchem Cl-Molekül reagiert ist zufällig. Na kann Cl mit einer Rate von $100s^{-1}$ ionisieren. Cl^- kann Na^+ mit einer Rate von $10s^{-1}$ deionisieren.
Die Reaktion sieht wie folgt aus:

$$Na + Cl \leftrightarrow Na^+ + Cl^-$$

SPiM-Programm

```
[ 1]          (* Na + Cl <==> Na+ + Cl- *)
[ 2]          directive sample 0.03
[ 3]          directive plot Na(); Na_plus()
[ 4]          directive graph
[ 5]
[ 6]          new ionize@100.0 : chan
[ 7]          new deionize@10.0 : chan
[ 8]
[ 9]          let Na() = !ionize; Na_plus()
[10]          and Na_plus() = ?deionize; Na()
[11]
[12]          let Cl() = ?ionize; Cl_minus()
[13]          and Cl_minus() = !deionize; Cl()
[14]
[15]          run (100 of Na() | 100 of Cl())
```

PIM-SPiM

- Programming Interface for Modelling with the Stochastic Pi-Machine
- Programmierung in englischer Metasprache
- Übersetzung in SPiM-Language

Beispiel