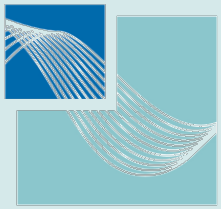


University of Applied Sciences

HOCHSCHULE
EMDEN·LEER

Sichere Automatisierung mit HATA

Formale Spezifikation und Verifikation von Automatisierungssystemen

Gerrit Jan Veltink

Sichere Automatisierung?

Hauptfrage:

- Wie können wir uns **sicher** sein,
  dass wir die richtigen Systeme implementieren?

- insbesondere problematisch im Bereich der parallelen Systeme!

- Automatisierungssysteme sind de facto parallel!
  - wir arbeiten und denken aber meistens nur sequenziell!

HATA - Ziele

- HATA
  - Hierarchical Algebraic Transaction Architecture
    - Backronym
    - auch Drachen (Japanisch) 

- Ziel des HATA-Projektes der HS Emden/Leer ist:
  - die **systematische** und **durchgängige** Benutzung von **formalen Methoden**

    bei der **Spezifikation** und **Verifikation** von Automatisierungssystemen

HATA – Industrie 4.0

- besonderes Augenmerk:
  Fragestellungen im Kontext von Industrie 4.0

- die **Kompositionalität** von existierenden Systemen steht dabei im Mittelpunkt

- besser bekannt unter den beiden Schlagwörtern:
  - *Orchestrierung* (Kontroll-Logik)
  - *Choreographie* (Abläufe)

mögliche Fragestellungen

- *Wie kann man beschreiben,*
  dass Systeme miteinander zusammenarbeiten sollen?

Orchestrierung

- *Welches Verhalten hat*
  der Zusammenschluss dieser Systeme im Anschluss?

Choreographie

- *Wie kann man sicherstellen,*
  dass keine Deadlocks, keine Livelocks (Endlosschleifen) oder andere Probleme im zusammengesetzten System auftreten?

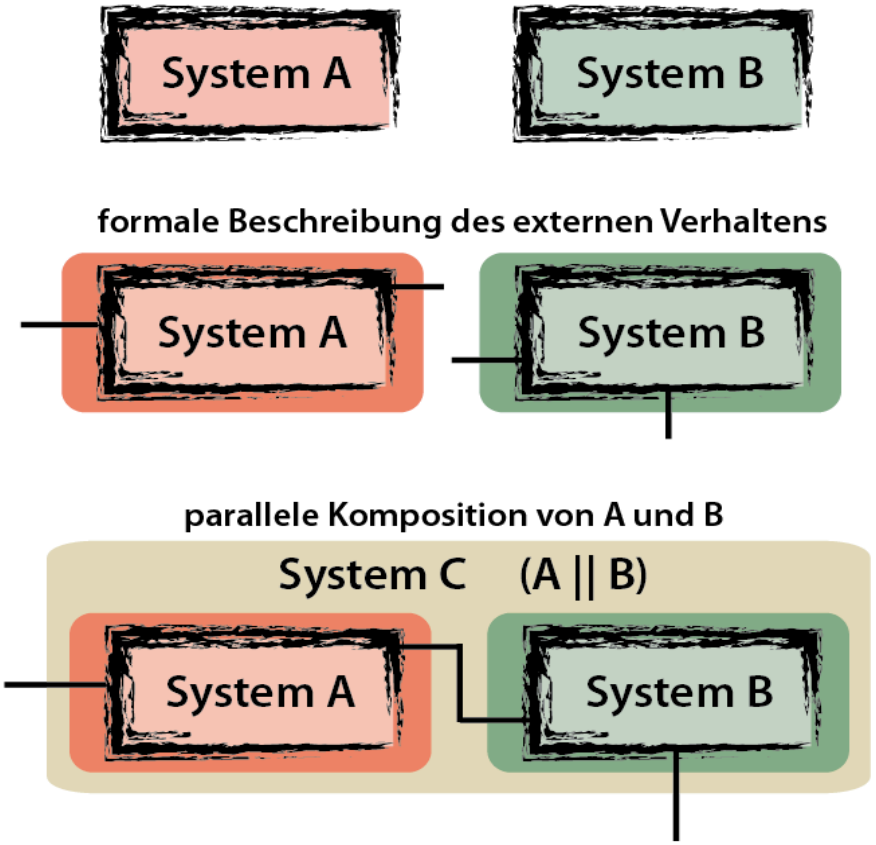
Sicherheit

möglicher Lösungsansatz

- formale Spezifikationen nutzen, um das (externe) Verhalten der (Einzel-)Systeme zu beschreiben

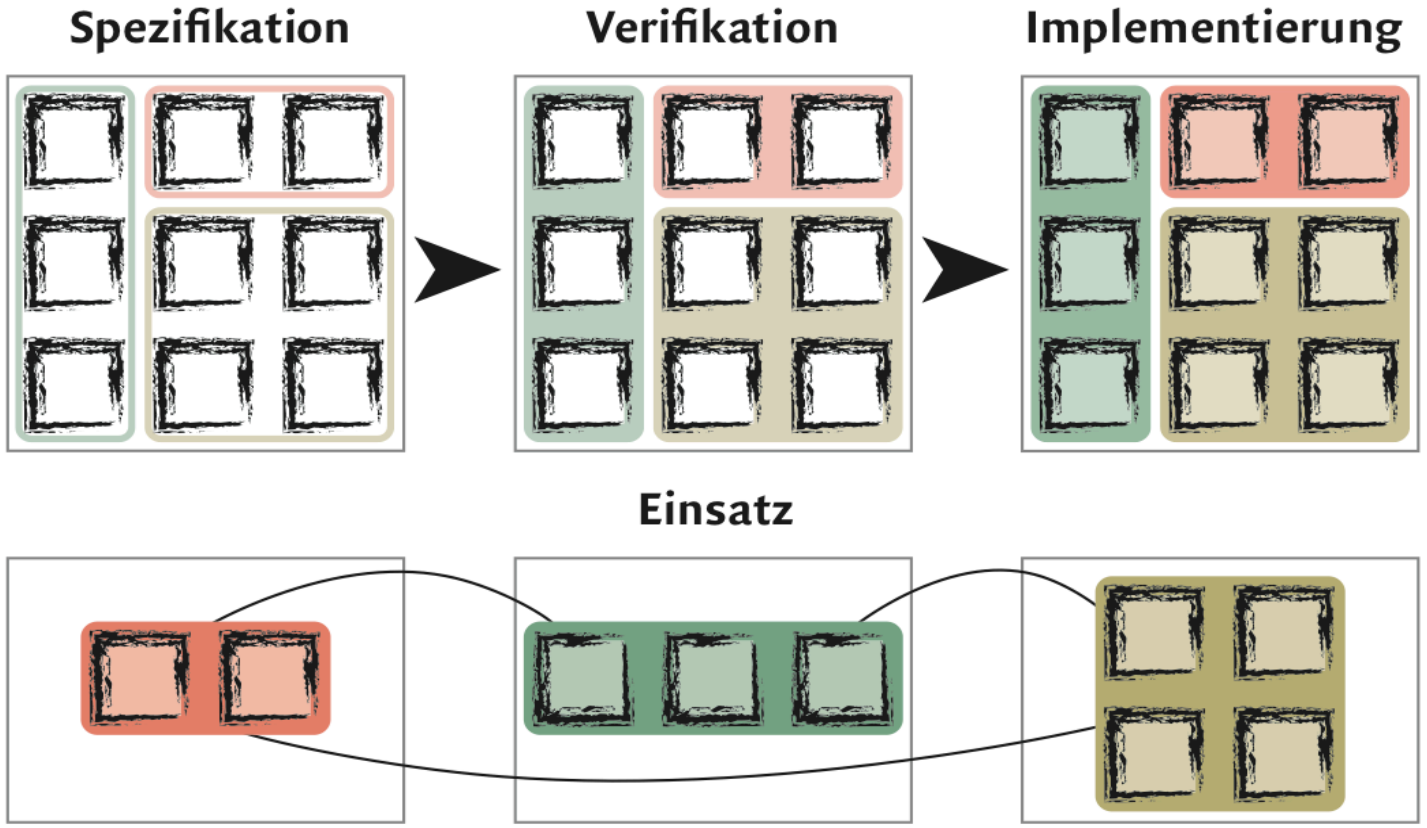
- die Beschreibungen mit den diversen Werkzeugen
  - kombinieren
  - analysieren
  - visualisieren

dies ist nicht IEC 61113 oder IEC 61499, sondern eine abstraktere mathematische Methode!



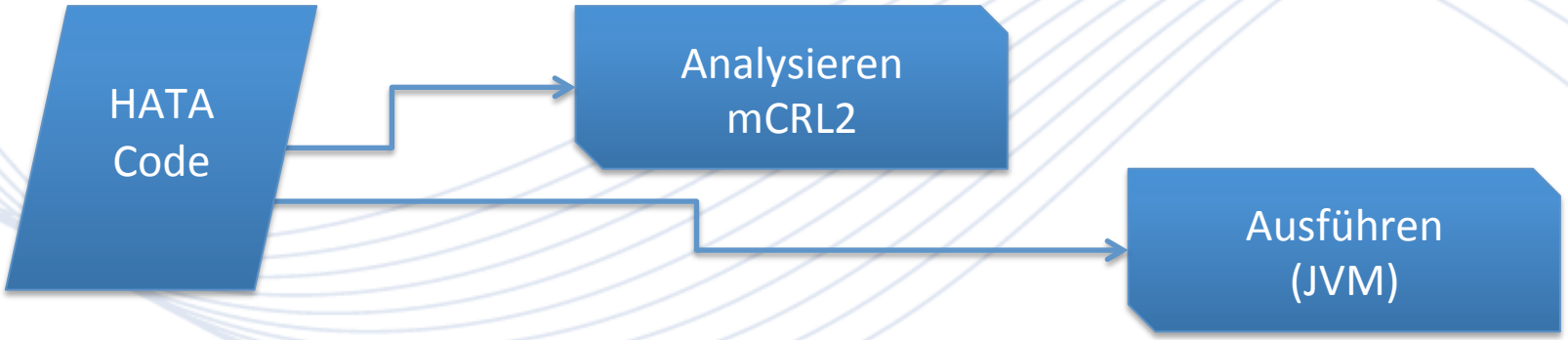
Vision: Ausführbare Spezifikationen

- **Vision**: Spezifikation und Implementierung eines Systems
  - in einem Dokument definieren & mit einer Entwicklungsumgebung bearbeiten



HATA - Lösung

- Programmiersprache entwickeln,
  die sich für C(++)/Java-Programmierer
  oder ST-Entwickler sofort ‘vertraut anfühlt’
  - es dennoch ermöglicht, aus dem Quell-Code
    formale Spezifikationen zu generieren
  - die mit Hilfe von bereits existierenden Werkzeugen weiter
    verarbeitet werden können



formale Zielsprache

- mCRL2 (micro Common Representation Language 2)
  - Zielsprache für formale Spezifikationen
  - wurde als universelle Spezifikationssprache für parallele Systeme und deren Tooling konzipiert
  - Nachfolger von µCRL (**1991**)
- mCRL2 und die zugehörigen Werkzeuge wurden ab **2006** entwickelt an:
  - der Technischen Universität Eindhoven
  - in Zusammenarbeit mit dem CWI (Amsterdam)
  - und der Universität Twente (Enschede)

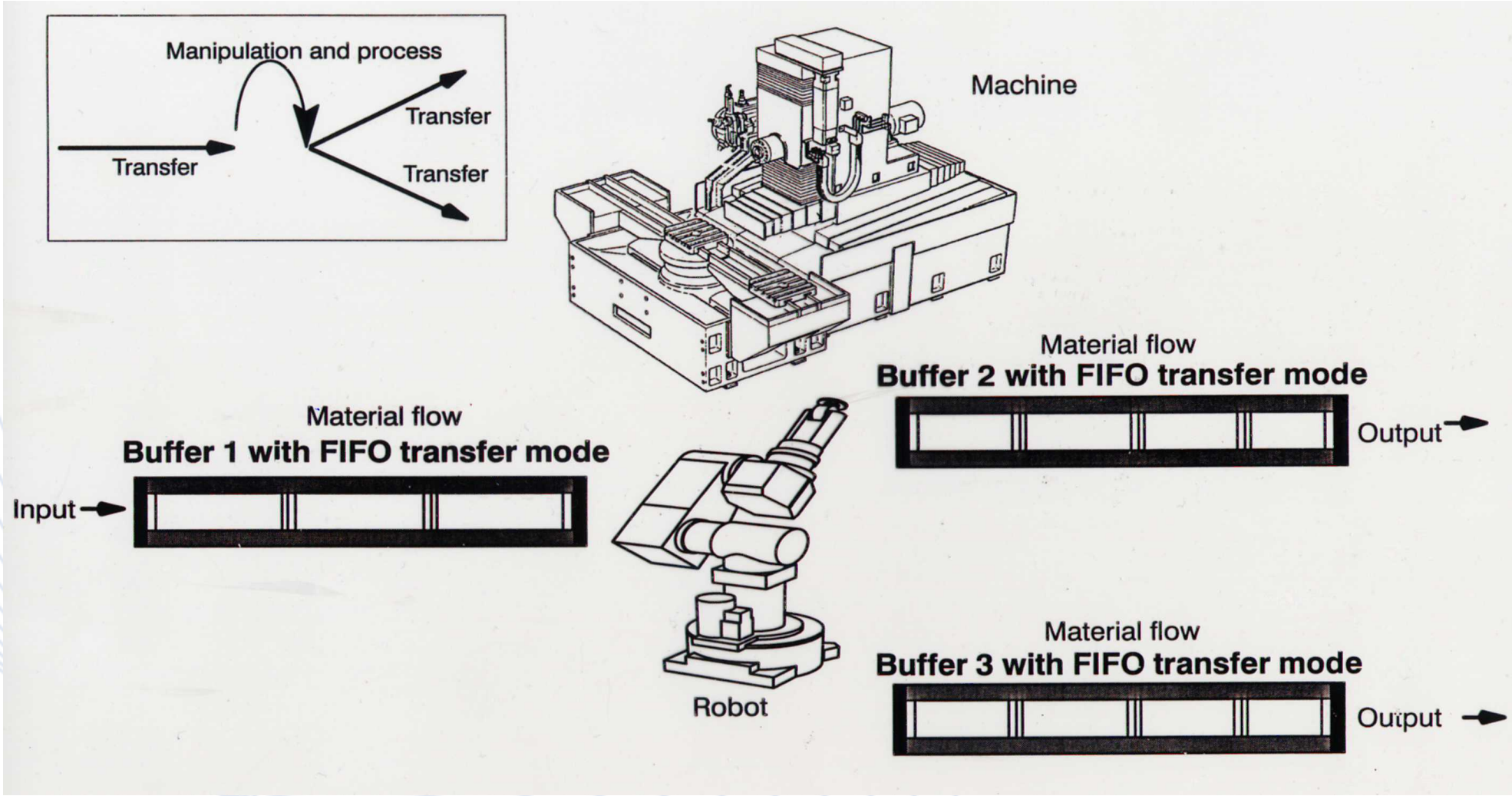
HATA - Stand

- HATA-Sprache im Moment noch in Entwicklung
- als *Proof of Concept* des Verfahrens wird die Benutzung der formalen Komponente gezeigt (in mCRL2)
- ein Beispiel zeigt Schritt für Schritt
  - die Fragestellung der Kompositionalität
  - die interaktiven Simulationsmöglichkeiten
  - die unterschiedlichen Visualisierungsmöglichkeiten

Aufbau des Beispiels

1. eine einfache Problemstellung der Automatisierung
2. Einführung: Fragment der mCRL2-Sprache
3. Spezifikation der Einzelsysteme der Problemstellung
   - Einzelsystem simulieren / analysieren / visualisieren
4. Einzelsysteme zu einem Gesamtsystem verbinden
   - Gesamtsystem simulieren / analysieren / visualisieren

Beispiel



Quelle: Vorlesungsunterlagen, Prof. Colombo

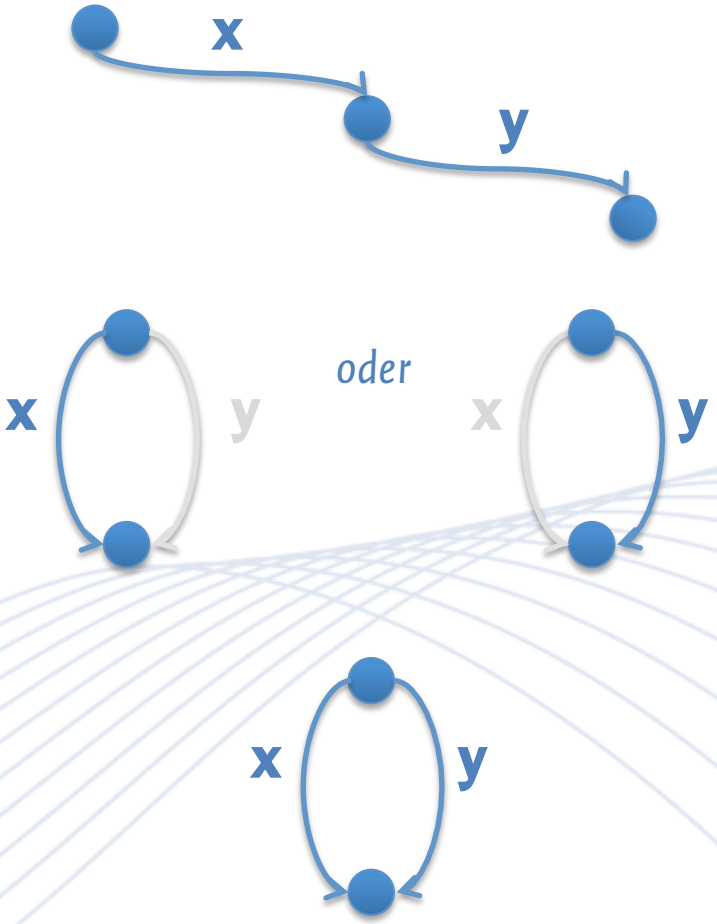
mCRL2 – Aktionen und Prozesse

- Aktionen, beobachtbare, unteilbare Ereignisse
  - z.B.: a, b, c
- Aktionen werden mit Hilfe von Operatoren verbunden zu Prozess-Termen
  - z.B.: $a + c$, $a \,.\, b$, $b \mid\mid c$
- Prozessnamen können benutzt werden als „Abkürzung“ für Prozess-Termen
  - z.B.: $X = (a + b) \mid\mid (c \,.\, d)$

mCRL2 – Basis-Operatoren: . + ||

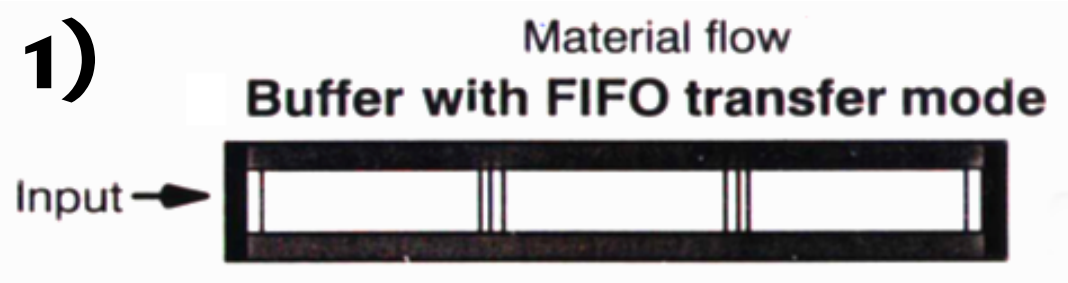
x und y sind Prozess-Terme

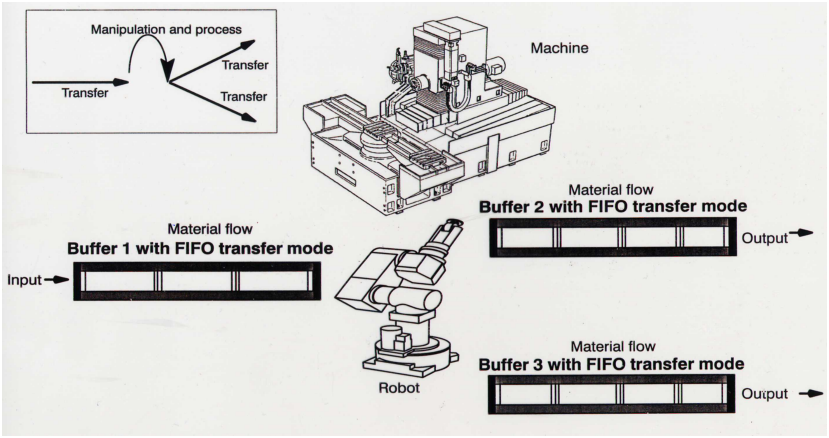
- **Sequenzielle Komposition:**
  - Beispiel: x . y
  - führe **x** aus und dann **y**
- **Alternative Komposition:**
  - Beispiel: x + y
  - führe entweder **x** oder **y** aus
- **Parallele Komposition:**
  - Beispiel: x || y
  - führe **x** und **y** gleichzeitig aus



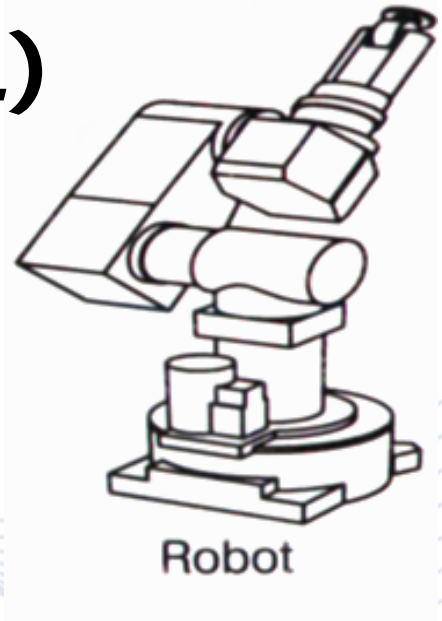
System modellieren • 3 Teilsysteme

1)

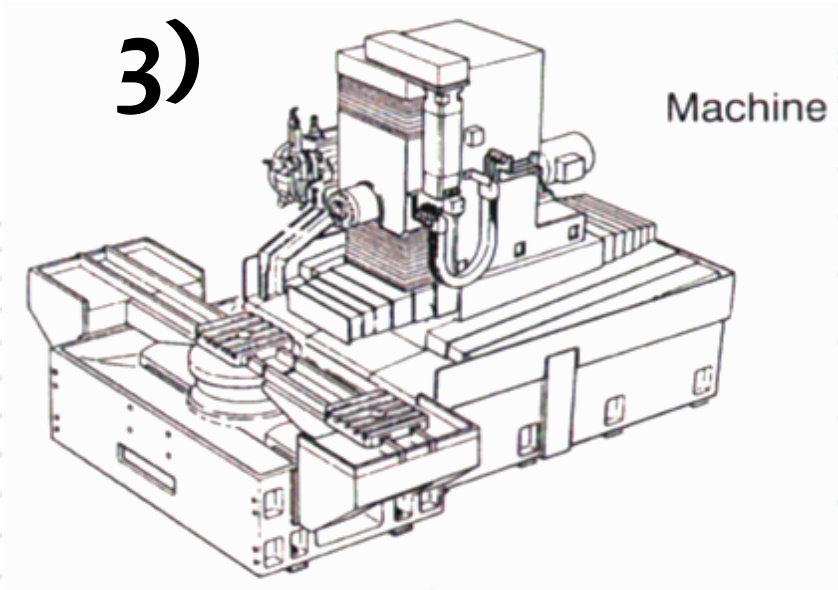




2)



3)



Buffer

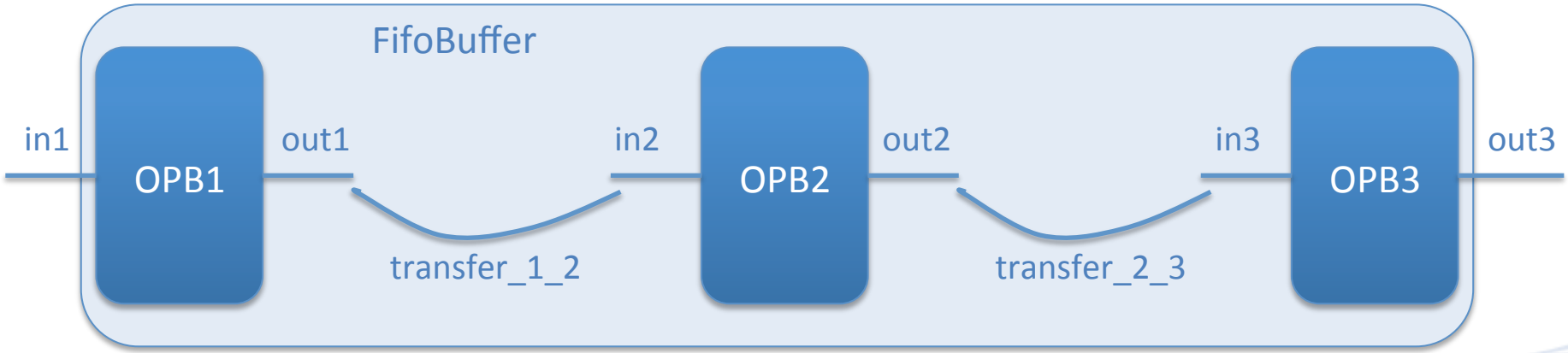
- zuerst ein Puffer mit einem Platz


```
act
productIn;      % put a product into the buffer
productOut;     % get a product out of the buffer

proc            % a one-place Buffer segment
OnePlaceBuffer = productIn . productOut . OnePlaceBuffer;
```

FifoBuffer

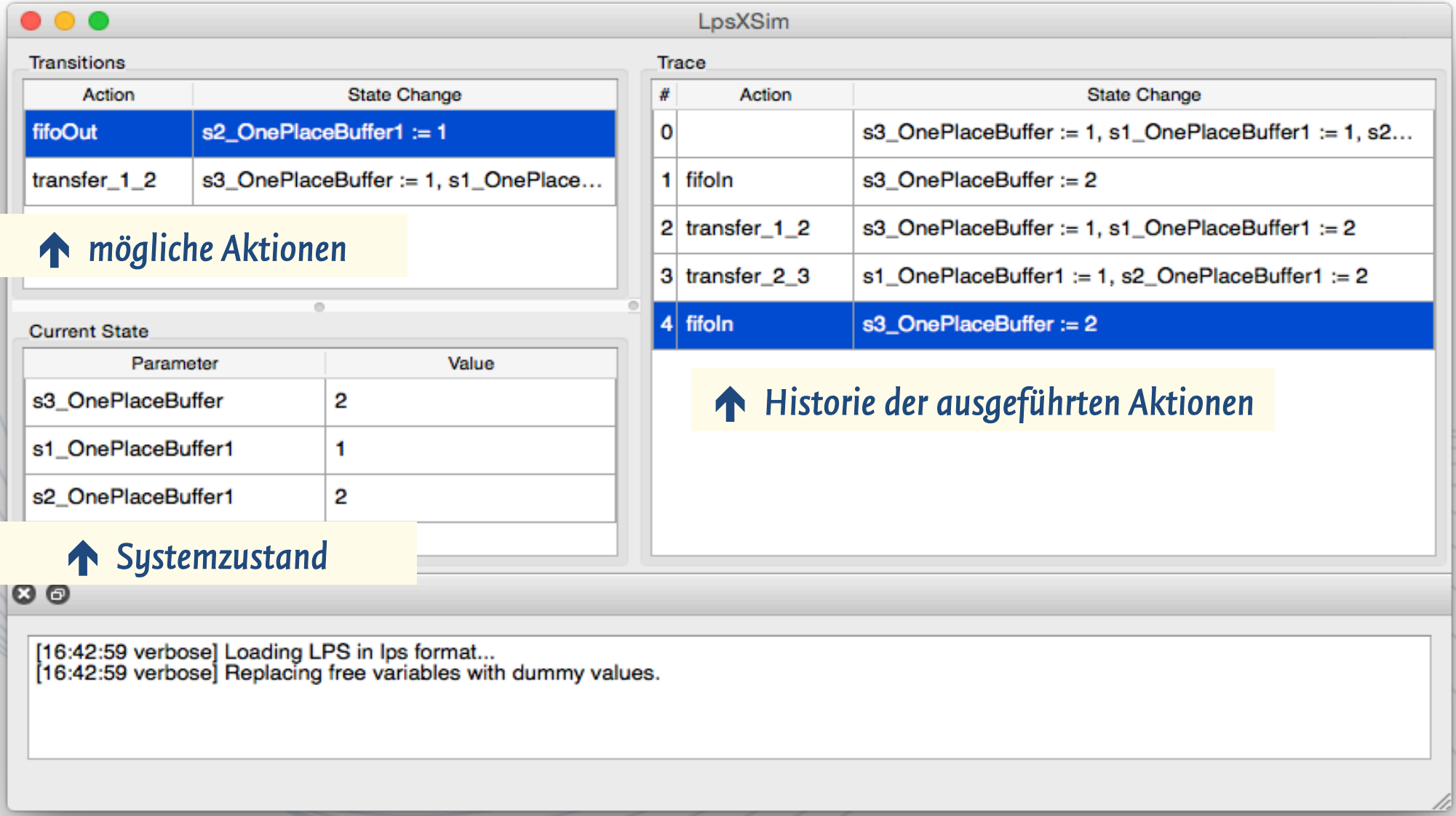
auch ein Beispiel für Kompositionalität!

- 3 Plätze durch Parallelität

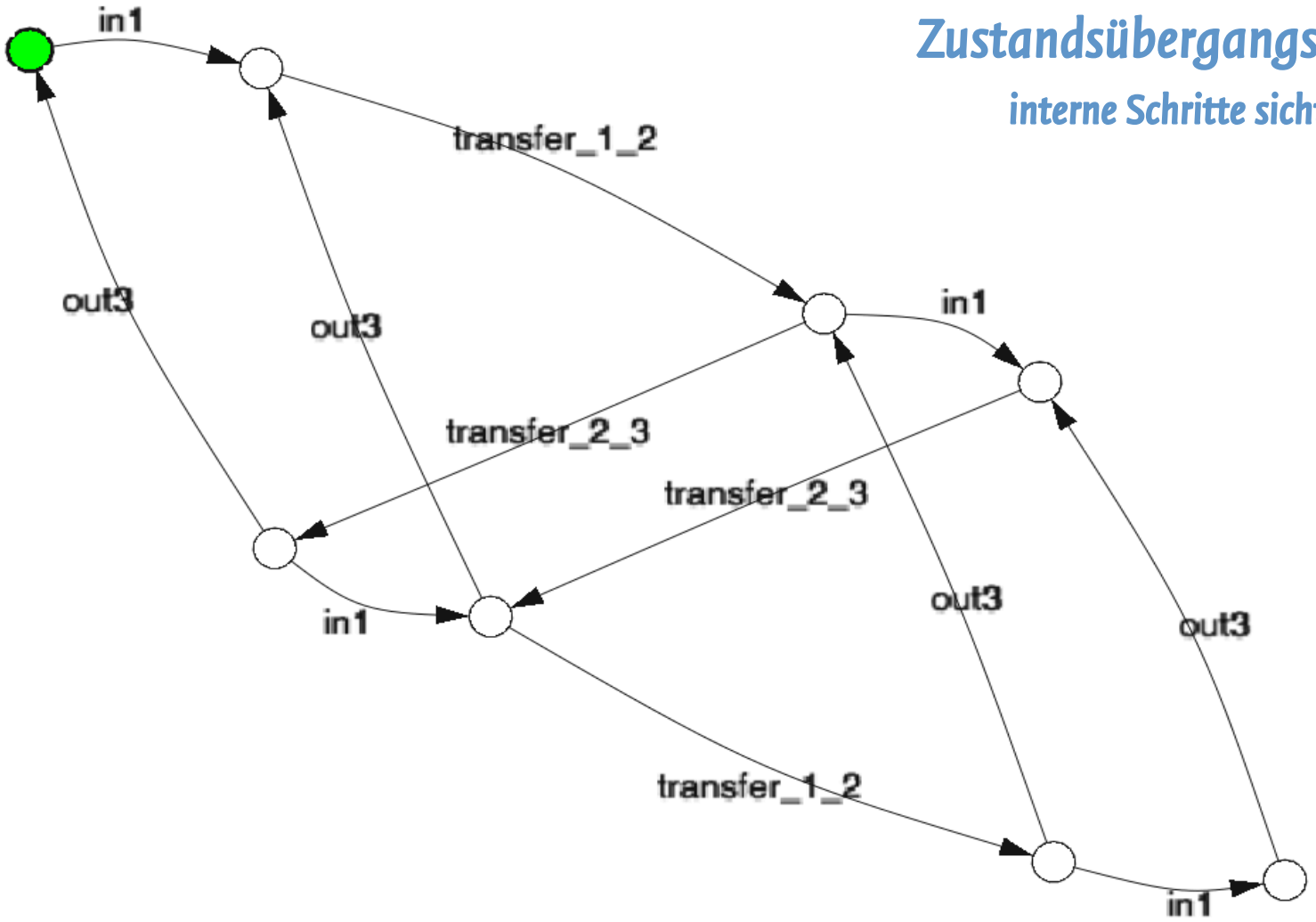



```
act in1, out1, in2, out2, in3, out3, transfer_1_2, transfer_2_3;
proc
OPB1 = rename({productIn -> in1, productOut -> out1}, OnePlaceBuffer); % 1st segment
OPB2 = rename({productIn -> in2, productOut -> out2}, OnePlaceBuffer); % 2nd segment
OPB3 = rename({productIn -> in3, productOut -> out3}, OnePlaceBuffer); % 3rd segment

FifoBuffer =  allow({ in1, out3 },                          % allowed to happen
              hide({ transfer_1_2, transfer_2_3 },          % hide internal processing
              comm({ out1 | in2 -> transfer_1_2,            % communication actions
                     out2 | in3 -> transfer_2_3},
                     OPB1 || OPB2 || OPB3 )));              % 3 Fifo-segments in parallel!
```

FifoBuffer • interaktive Simulation



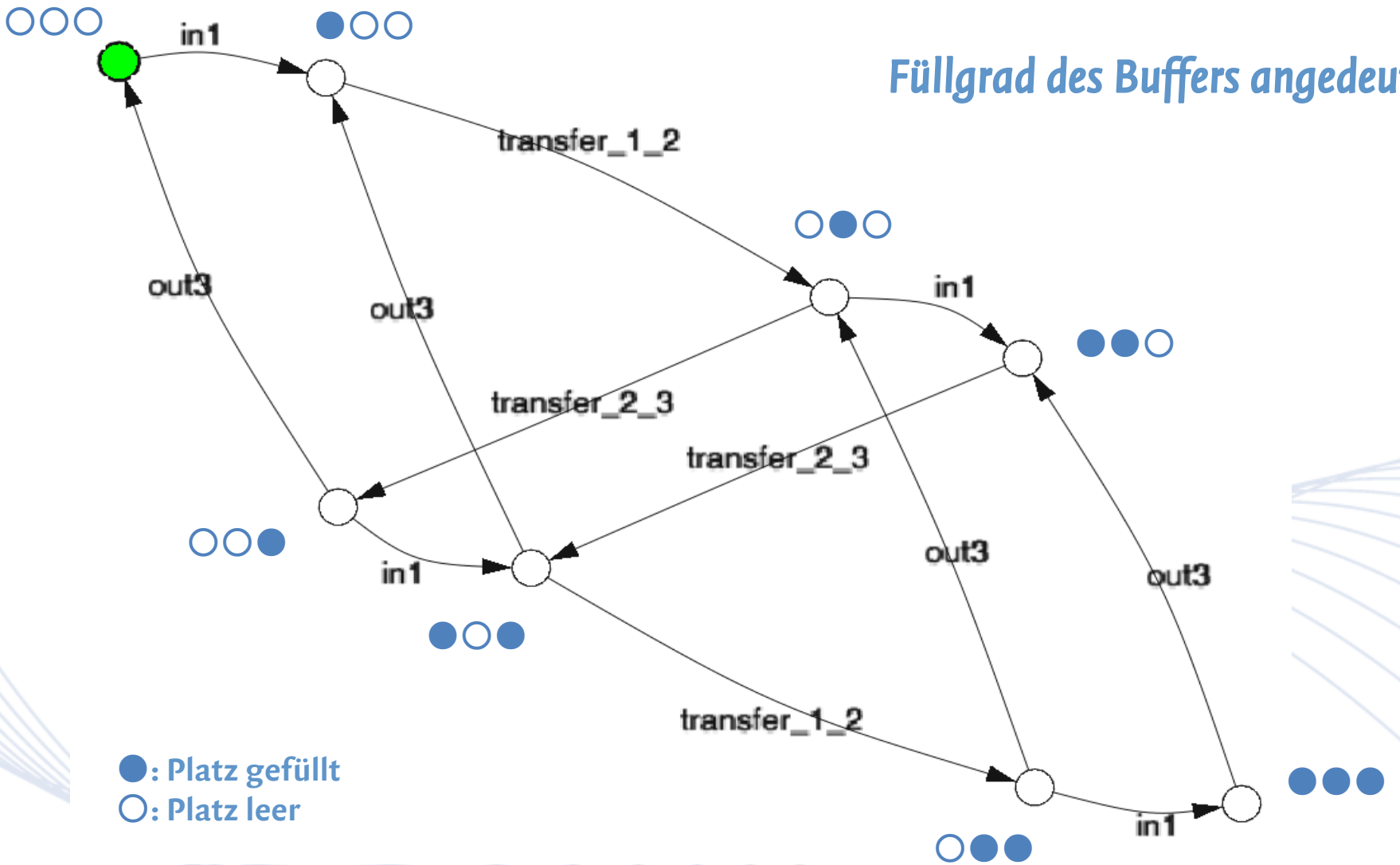
FifoBuffer • Visualisierung



Zustandsübergangsdiagramm

interne Schritte sichtbar gemacht!

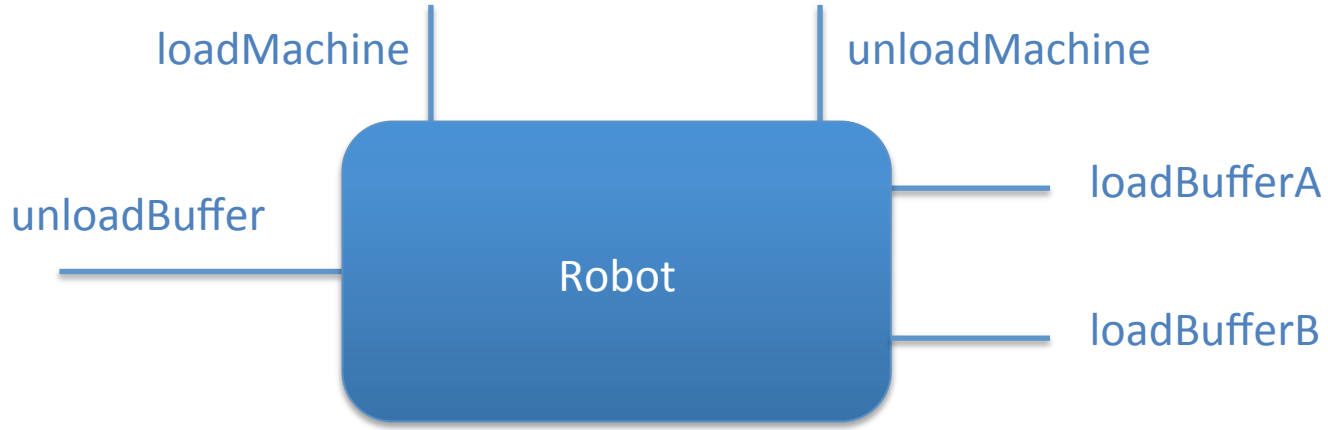
FifoBuffer • Visualisierung mit Annotation



Füllgrad des Buffers angedeutet

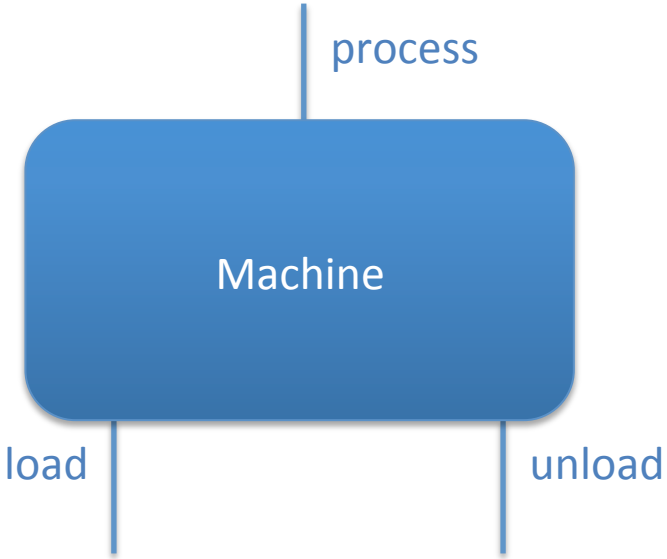
●: Platz gefüllt
○: Platz leer

Robot



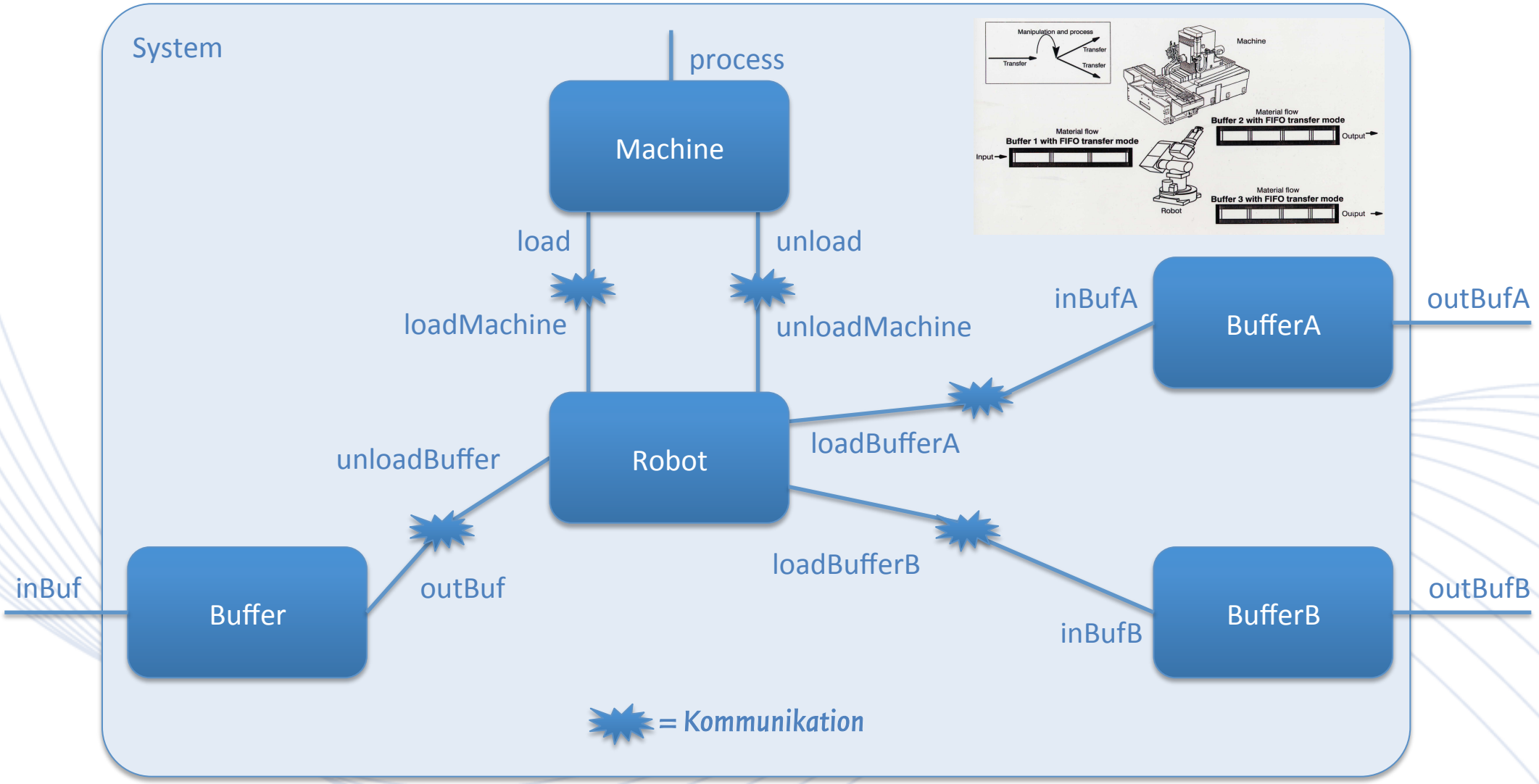

```
act
unloadBuffer, loadMachine, unloadMachine,
loadBufferA, loadBufferB;

proc
Robot = unloadBuffer . loadMachine . unloadMachine
        . (loadBufferA + loadBufferB) . Robot;
```

Machine




```
act
       load, process, unload;

proc
       Machine = load . process . unload . Machine;
```

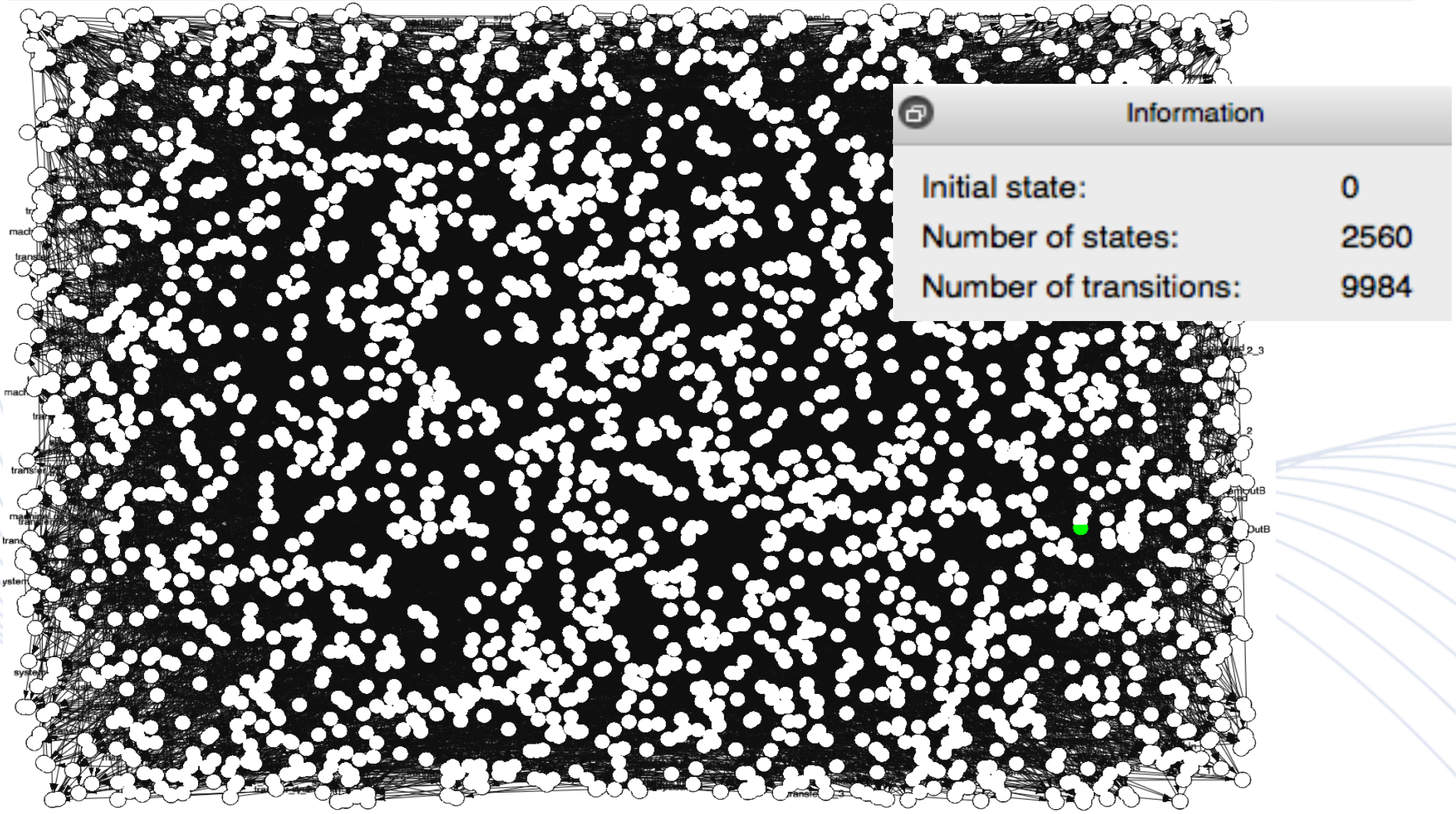
Gesamtsystem • Diagramm



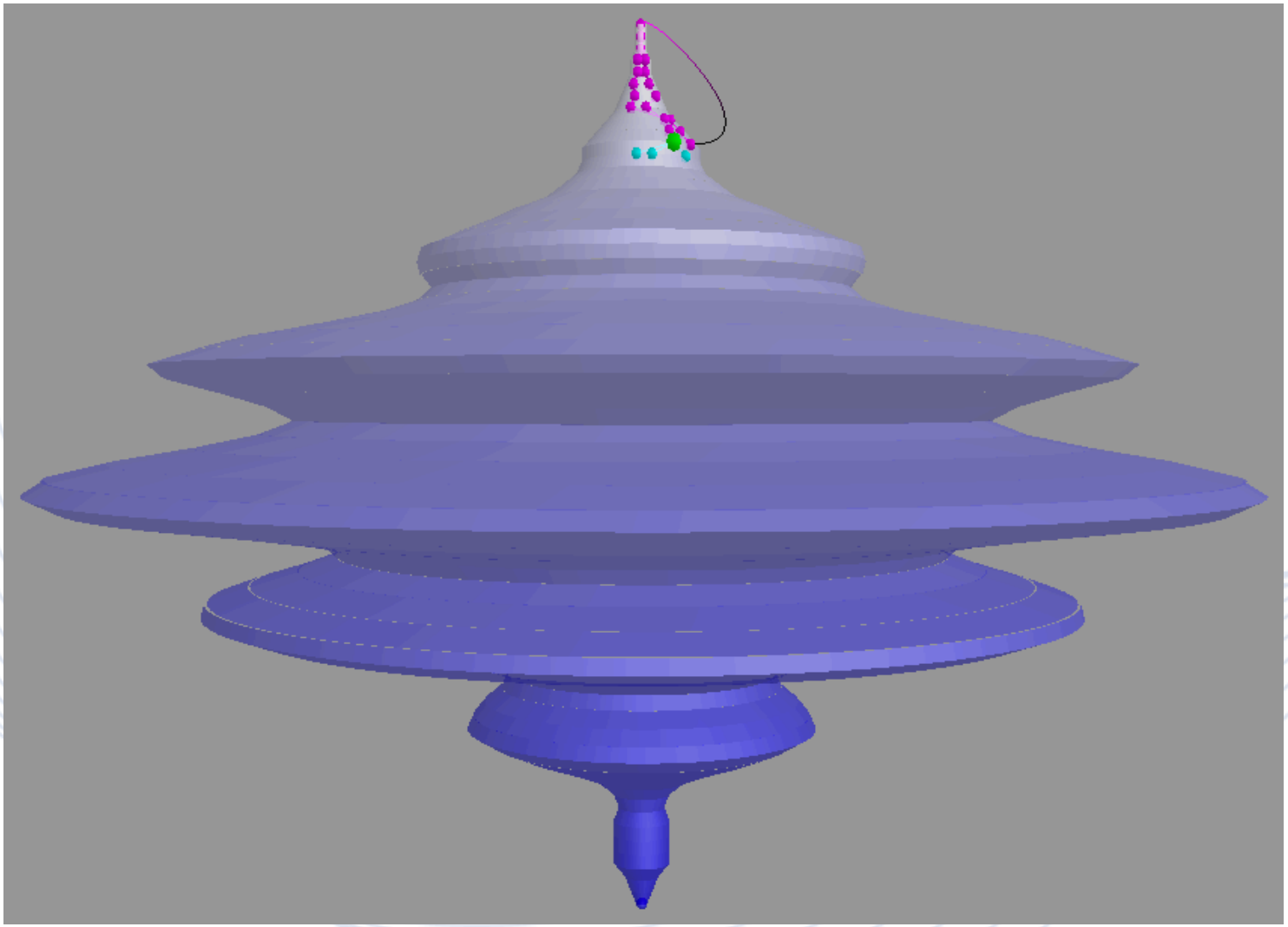
Gesamtsystem • Spezifikation

```
act
        systemIn, outBuf, bufferUnloaded, machineLoaded, machineUnloaded,
        bufferALoaded, bufferBLoaded, inBufA, inBufB, systemOutA, systemOutB;

proc
        Buffer  = rename({fifoIn -> systemIn, fifoOut -> outBuf}, FifoBuffer);
        BufferA = rename({fifoIn -> inBufA, fifoOut -> systemOutA}, FifoBuffer);
        BufferB = rename({fifoIn -> inBufB, fifoOut -> systemOutB}, FifoBuffer);
        System =
                allow({  systemIn, systemOutA, systemOutB,
                         transfer_1_2, transfer_2_3, bufferUnloaded, machineLoaded,
                         process, machineUnloaded, bufferALoaded, bufferBLoaded },

                comm({   outBuf        | unloadBuffer  -> bufferUnloaded,
                         loadMachine   | load      -> machineLoaded,
                         unloadMachine | unload        -> machineUnloaded,
                         loadBufferA   | inBufA        -> bufferALoaded,
                         loadBufferB   | inBufB        -> bufferBLoaded },
                         Buffer || Robot || Machine || BufferA || BufferB

                ));
```

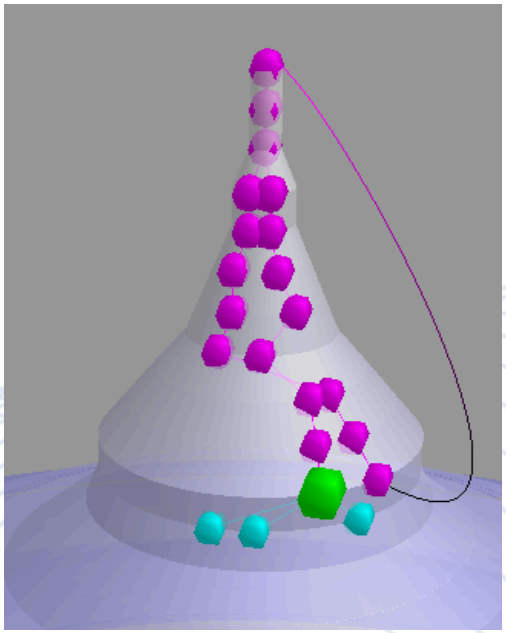
die kombinatorische Explosion!



Cluster-Darstellung notwendig



Einzelzustände werden gruppiert dargestellt!



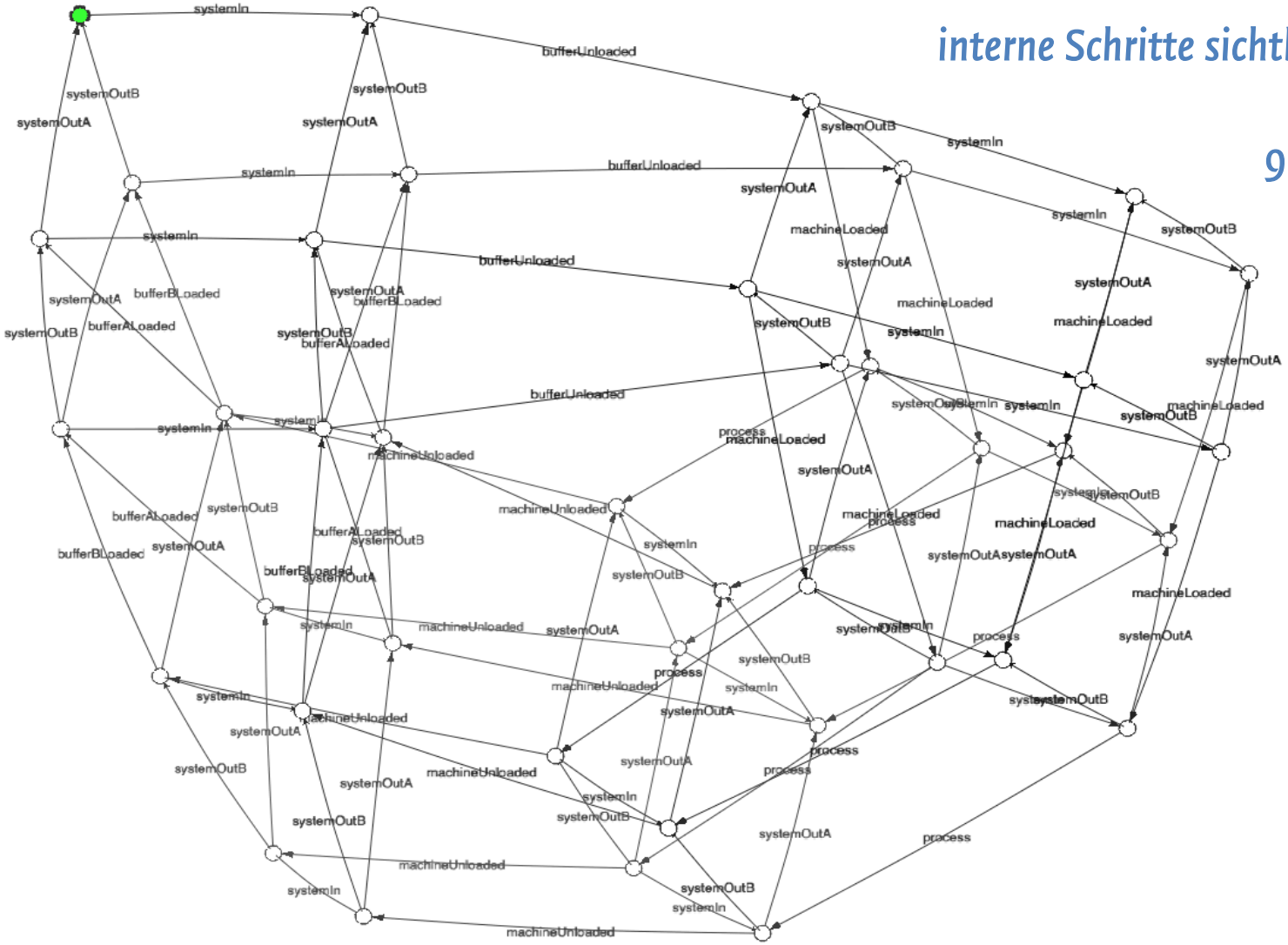
interaktive Simulation in der Visualisierung

Gesamtsystem • Spezifikation • reduziert

jetzt OnePlaceBuffer benutzt!

```
act
        systemIn, outBuf, bufferUnloaded, machineLoaded, machineUnloaded,
        bufferALoaded, bufferBLoaded, inBufA, inBufB, systemOutA, systemOutB;

proc
        Buffer  = rename({productIn -> systemIn, productOut -> outBuf}, OnePlaceBuffer);
        BufferA = rename({productIn -> inBufA, productOut -> systemOutA}, OnePlaceBuffer);
        BufferB = rename({productIn -> inBufB, productOut -> systemOutB}, OnePlaceBuffer);
        System =
                allow({  systemIn, systemOutA, systemOutB,
                         transfer_1_2, transfer_2_3, bufferUnloaded, machineLoaded,
                         process, machineUnloaded, bufferALoaded, bufferBLoaded },

                comm({   outBuf        | unloadBuffer  -> bufferUnloaded,
                         loadMachine   | load      -> machineLoaded,
                         unloadMachine | unload        -> machineUnloaded,
                         loadBufferA   | inBufA        -> bufferALoaded,
                         loadBufferB   | inBufB        -> bufferBLoaded },
                         Buffer || Robot || Machine || BufferA || BufferB

                ));
```

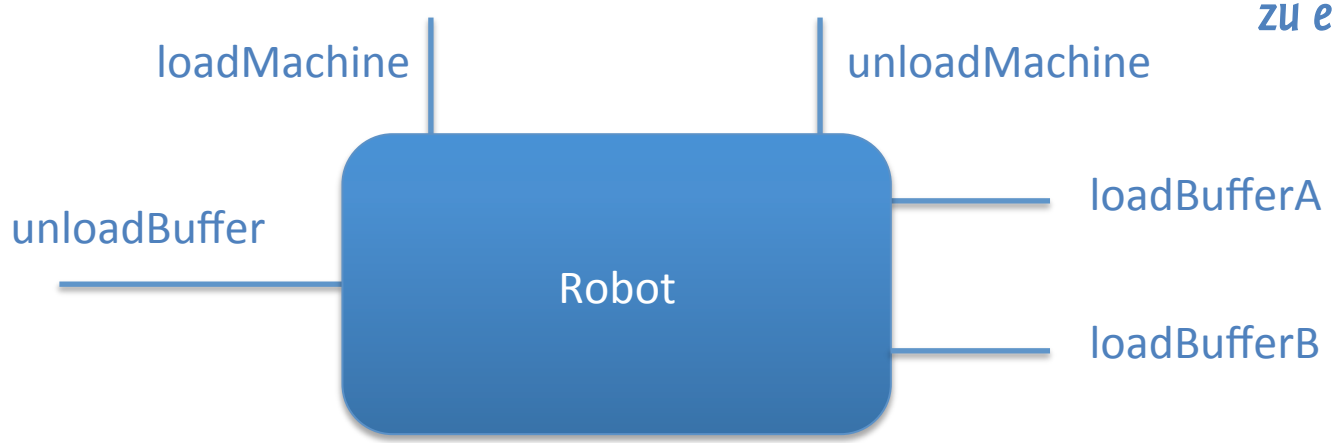
Gesamtsystem · Visualisierung



interne Schritte sichtbar gemacht!
40 Zustände
96 Übergänge

Fehlersuche • Robot mal anders

erst im Gesamtsystem führt diese Variante zu einem Problem!



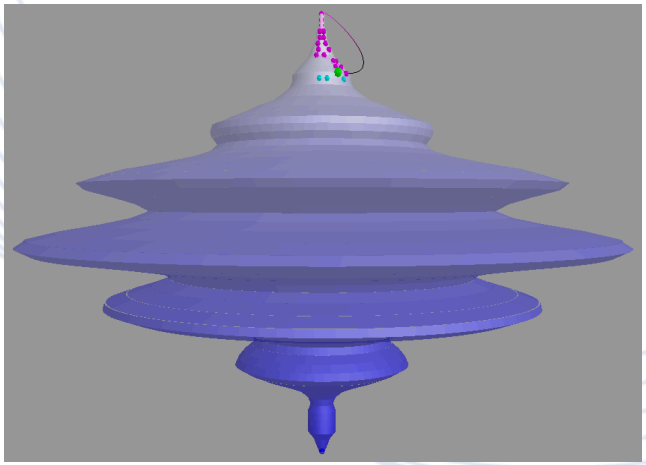

```
act
unloadBuffer, loadMachine, unloadMachine,
loadBufferA, loadBufferB;

proc
Robot = ( unloadBuffer . loadMachine
        + unloadMachine . (loadBufferA + loadBufferB)
        ) . Robot;
```

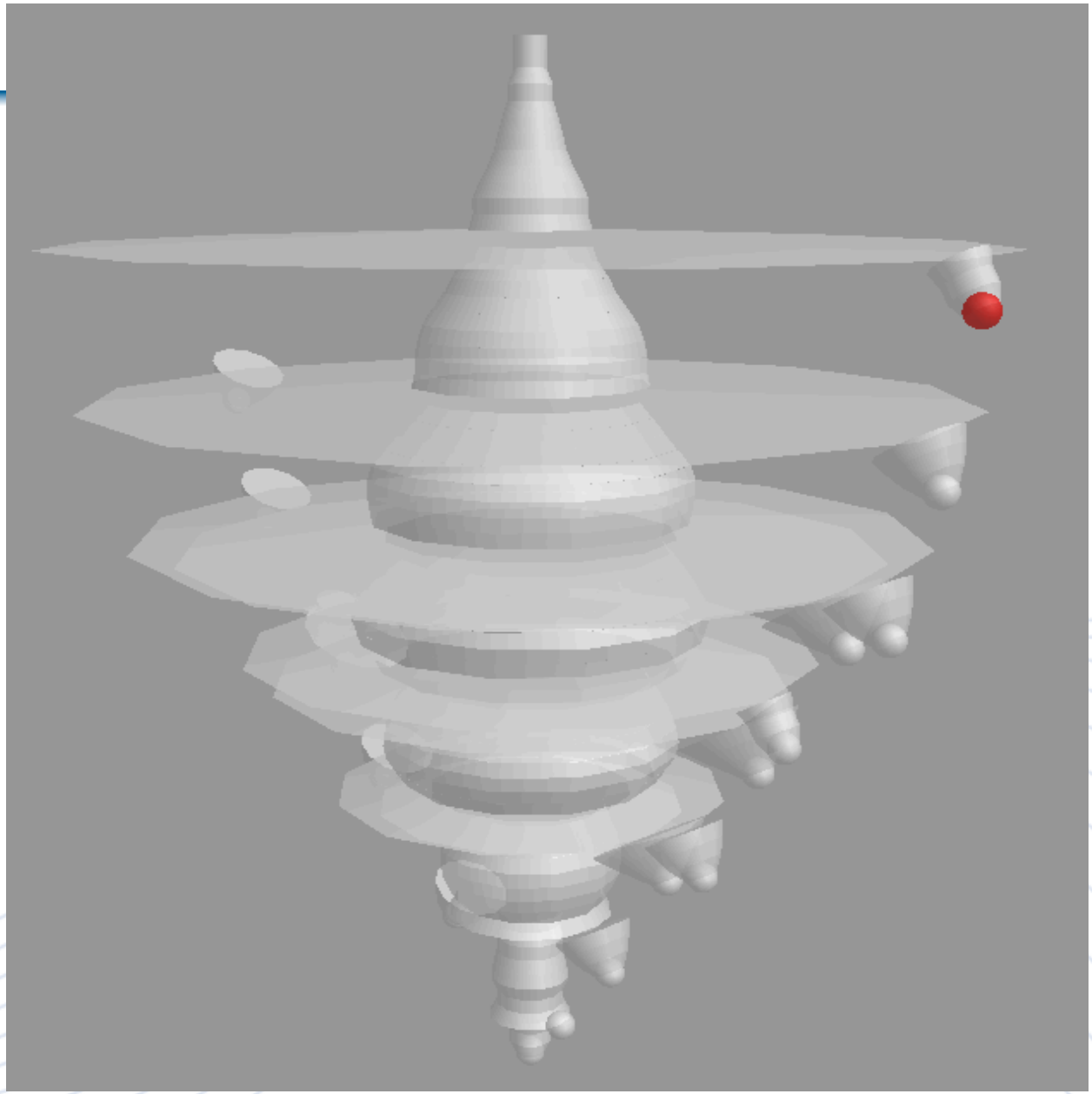
Fehlersuche

Deadlocks werden automatisch angezeigt!

weitere Prüfmethoden stehen zur Verfügung, u.a. Hennessy-Milner-Logik

Originaldarstellung





Demo

wenn gewünscht, anschließend an dieser Session:

Demo am Rechner

bitte melden Sie sich kurz vorne im Raum