Chapter 3
Specification Models

Lothar Thiele
Discrete Event Systems
Winter 2004/2005

Overview

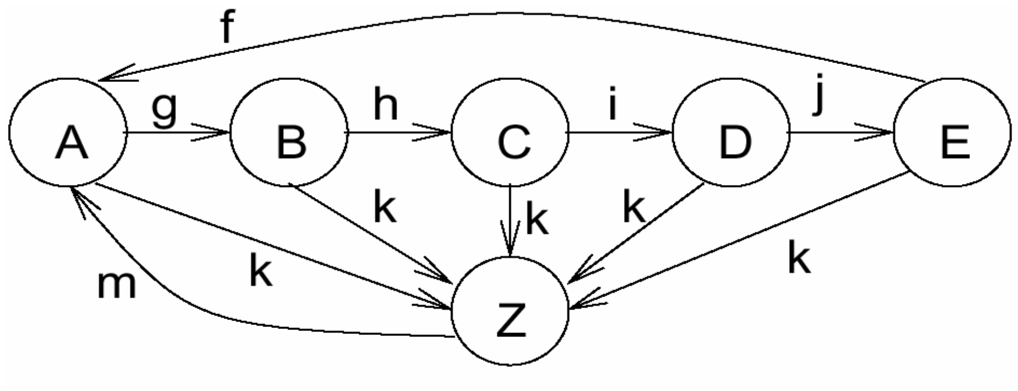
- StateCharts
  - Motivation
  - State hierarchy
  - Representing computations
  - Semantics
  - Tools
- Petri nets
  - Definition
  - Token game
  - Examples
  - Extensions

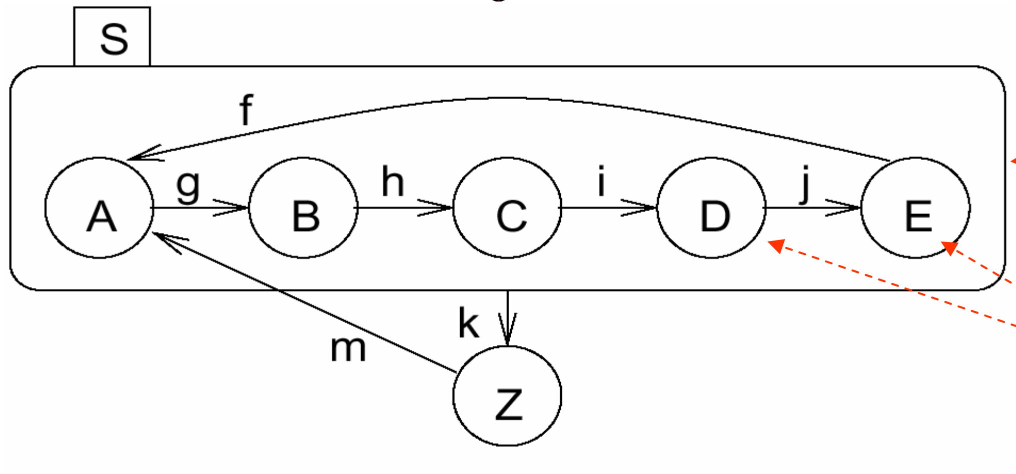
some of the transparencies are based on lectures by Peter Marwedel, Dortmund.

Motivation

- ***Deficits of finite automata for modeling***:
  - only one sequential process, no concurrency
  - no hierarchical structuring capabilities
- ***Extension***:
  - StateCharts-Model von D. Harel [1987].
  - StateCharts introduces hierarchy, concurrency and computation.
  - Model is used in many tools for the specification, analysis and simulation of discrete event systems, e.g. Matlab-Stateflow, UML, Rhapsody, Magnum.
  - Complicated semantics: We will only cover some basic mechanisms.

Introducing hierarchy





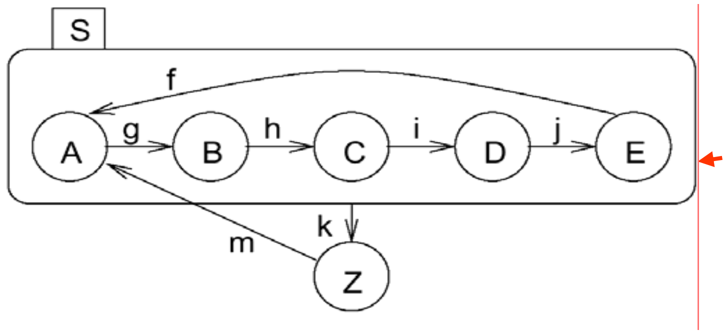
FSM will be **in** exactly one of the substates of S if S is **active**
(either in A or in B or ..)

superstate

substates

Definitions

- Current states of FSMs are also called ***active*** states.
- States which are not composed of other states are called ***basic states***.
- States containing other states are called ***super-states***.
- For each basic state s, the super-states containing s are called ***ancestor states***.
- Super-states S are called ***OR-super-states***, if exactly one of the sub-states of S is active whenever S is active.

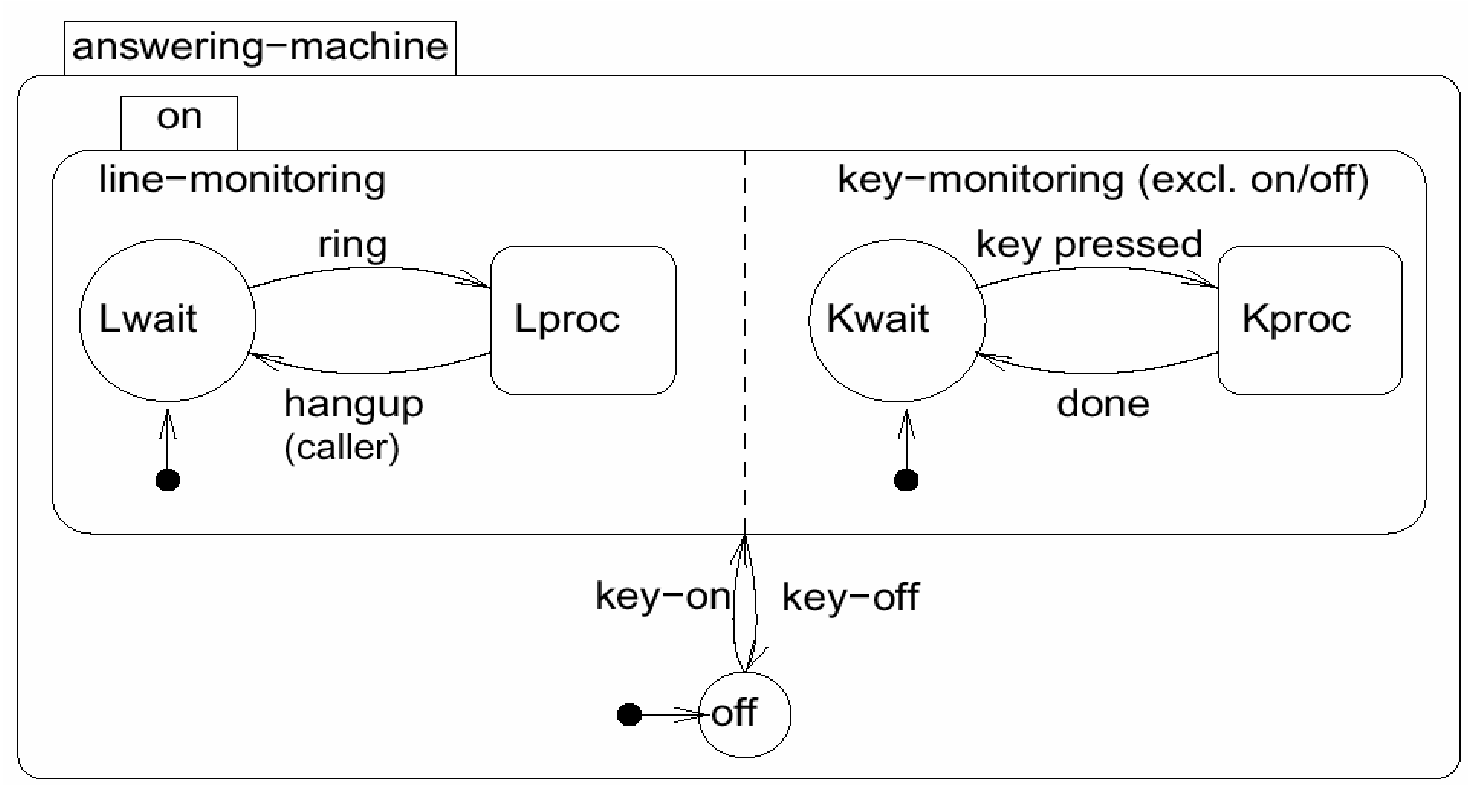


ancestor state of E

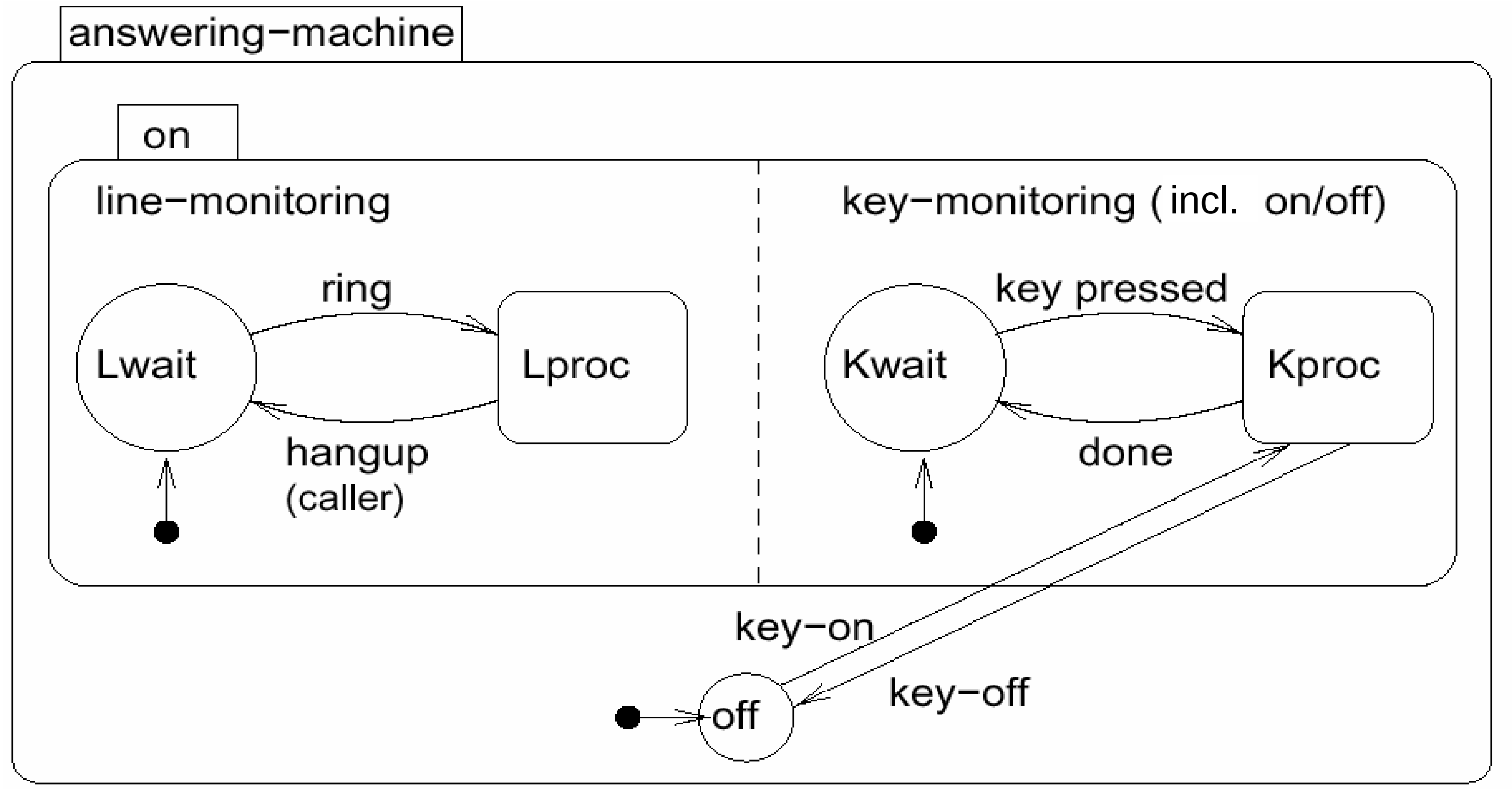
Concurrency

Convenient ways of describing concurrency are required.

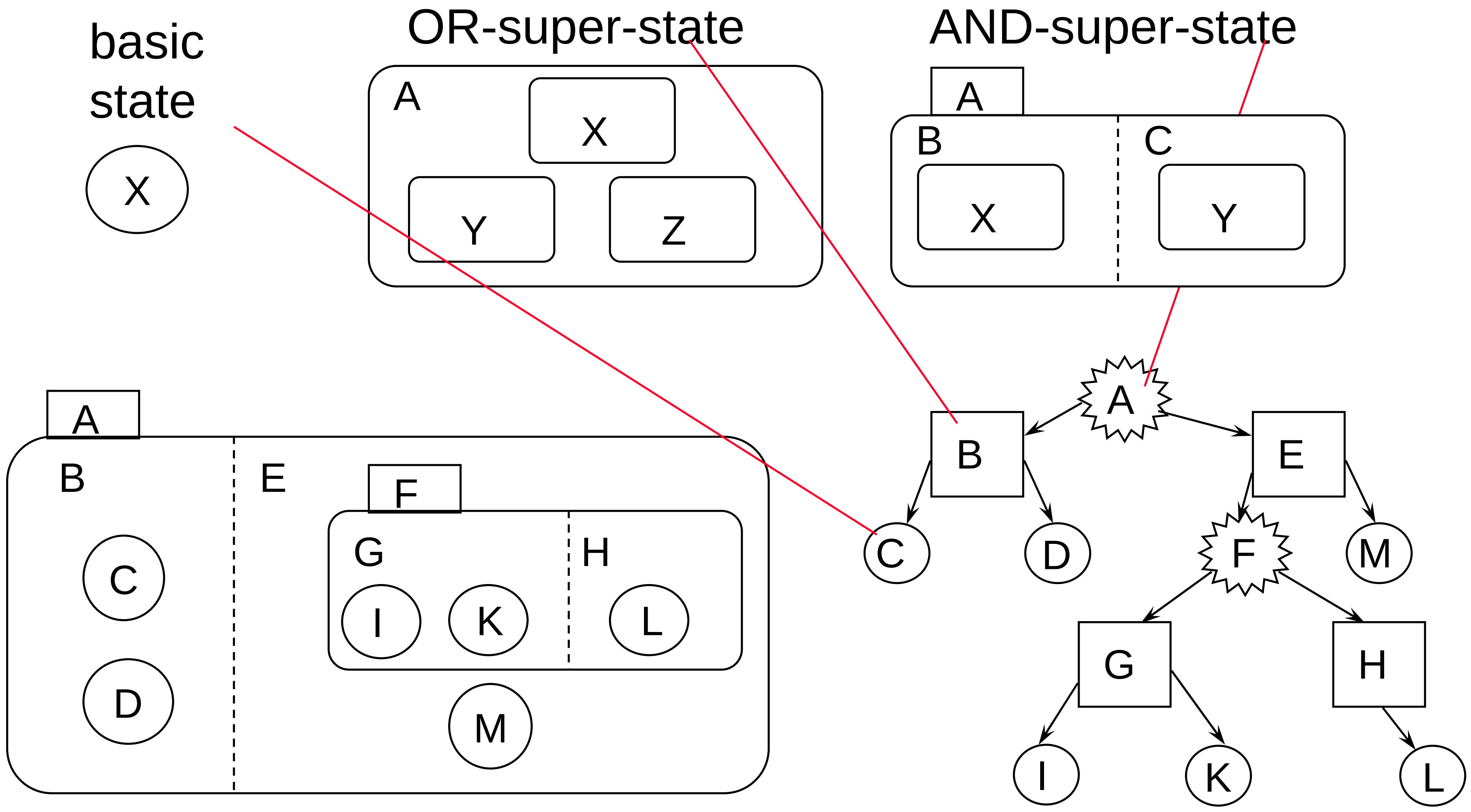
AND-super-states: FSM is in **all** (immediate) sub-states of a super-state.



Entering and leaving AND-super-states

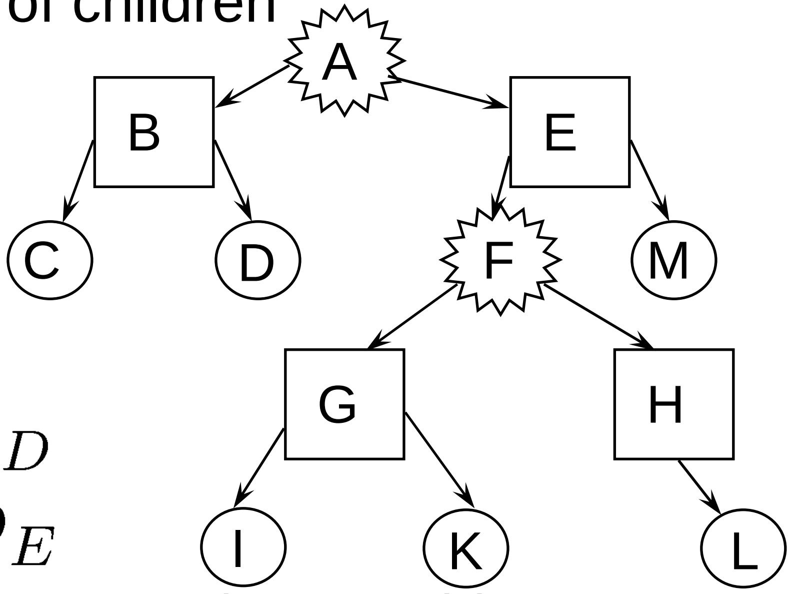


Tree representation of state sets



Computation of state sets

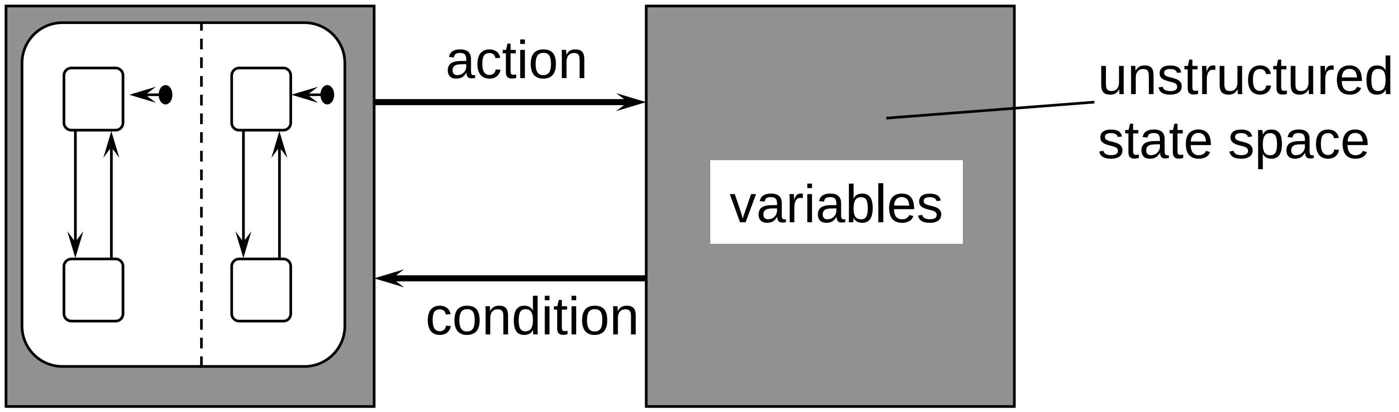
- Computation of state sets by ***traversing the tree*** from leaves to root:
  - basic states: state set = state
  - OR-super-states: state set = Cartesian product of children
  - AND-super-states: state set = union of children



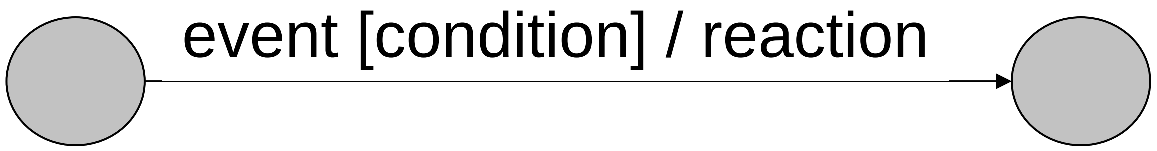
$$Q_H = Q_L,\ Q_G = Q_I \cup Q_K$$
$$Q_F = Q_G \times Q_H,\ Q_B = Q_C \cup Q_D$$
$$Q_E = Q_F \cup Q_M,\ Q_A = Q_B \times Q_E$$
$$Q_A = (Q_C \cup Q_D) \times (Q_M \cup ((Q_I \cup Q_K) \times Q_L))$$

Representation of computations

- Besides states, arbitrary many other variables can be defined. This way, not all states of the system are modeled explicitly.
- These variables can be changed as a result of a state transition ("***action***"). State transitions can be dependent on these variables ("***condition***").



General form of edge labels



***Event*:**

Events exist only until the next evaluation step of the model

Can be either internally or externally generated

***Condition*:**

Refer to values of variables that keep their value until they are reassigned.

***State transition*:**

Transition is enabled if event exists and condition evaluates to true

***Reaction*:**

Can be assignments for variables (“**action**”) and/or creation of events

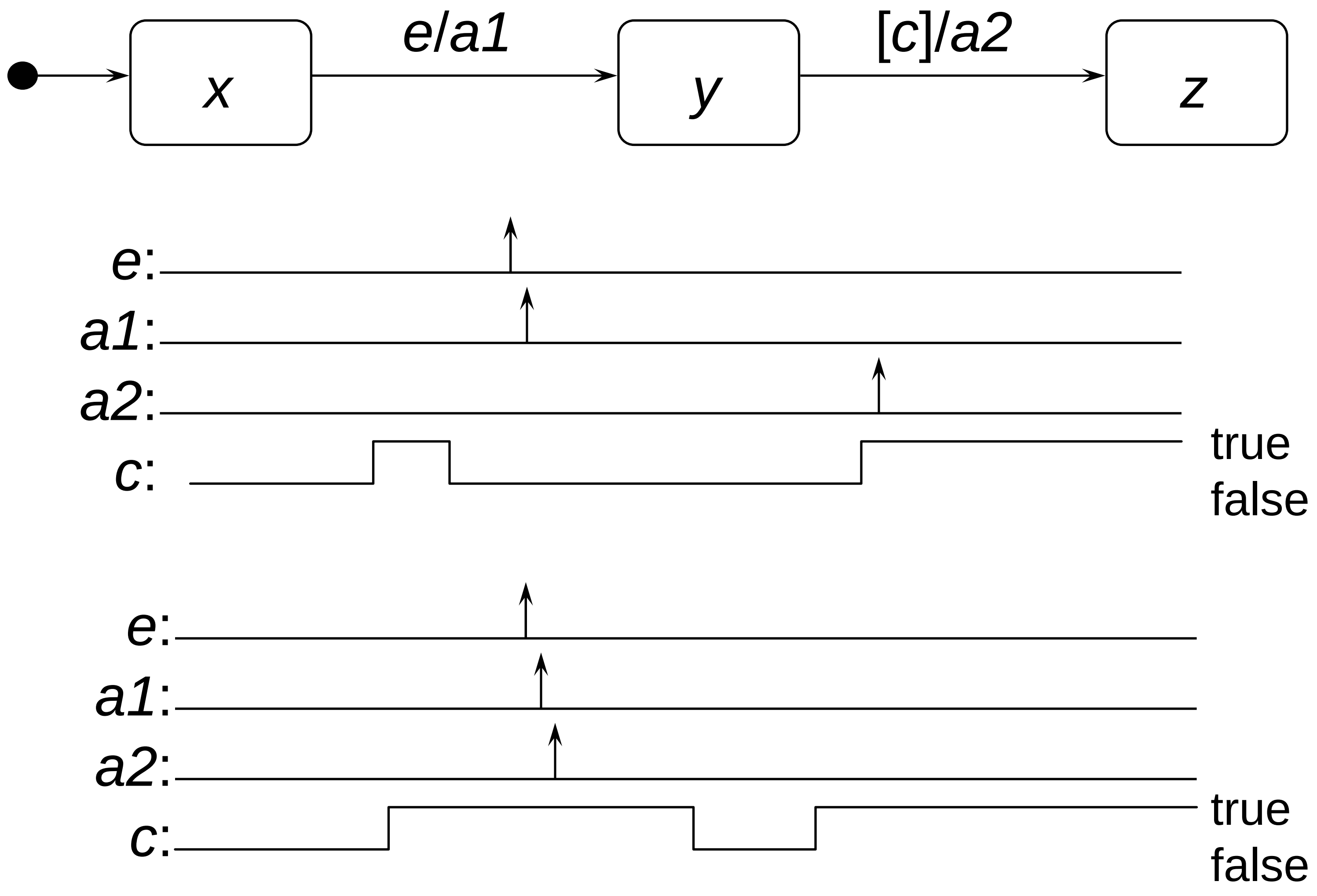
Events and actions

- "***event***" can be composed of several events:
  - **(*e1* and *e2*)** : event that corresponds to the simultaneous occurrence of e1 and e2.
  - **(*e1* or *e2*)** : event that corresponds to the occurrence of either e1 or e2 or both.
  - **(not *e*)** : event that corresponds to the absence of event e.

- „***action***“ can also be composed:
  - **(*a1*; *a2*)** : actions a1 und a2 are executed sequentially.

- All events, states and actions are globally visible.

Example



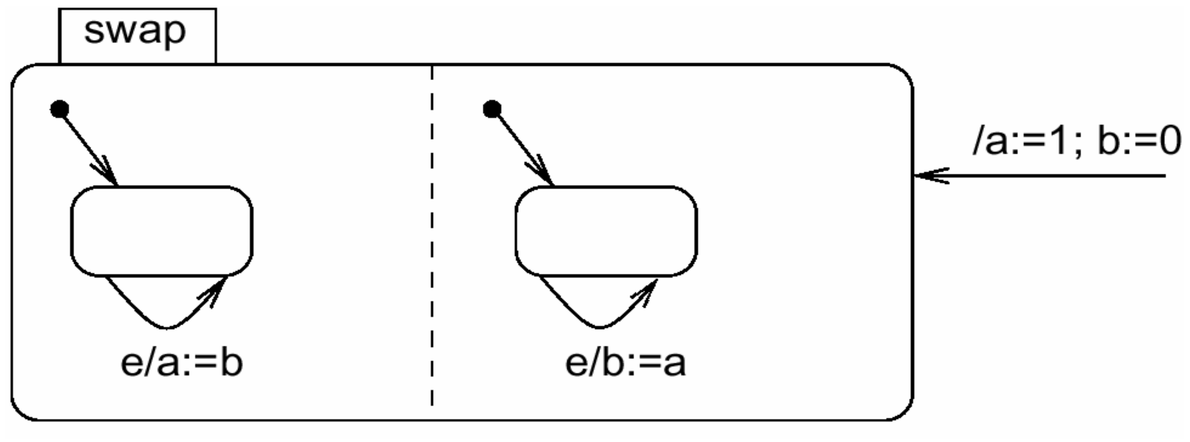
The StateCharts simulation phases

How are edge labels evaluated in one ‘simulation’ step?

Three phases:

1. Effect of changes on events and conditions is evaluated,
2. The set of transitions to be made in the current step and right hand sides of assignments are computed,
3. Transitions become effective, variables obtain new values.

Example



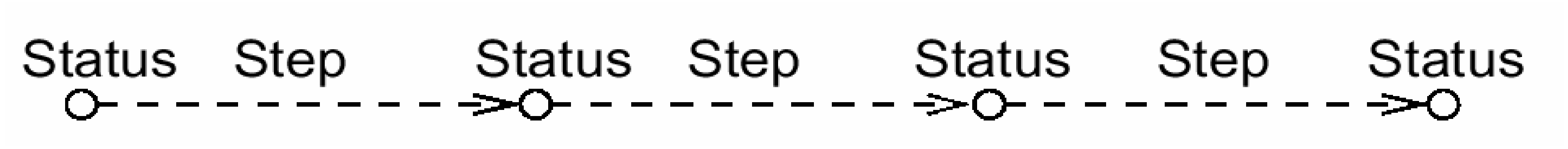
In phase 2, variables a and b are assigned to temporary variables.

In phase 3, these are assigned to a and b.

As a result, variables a and b are swapped.

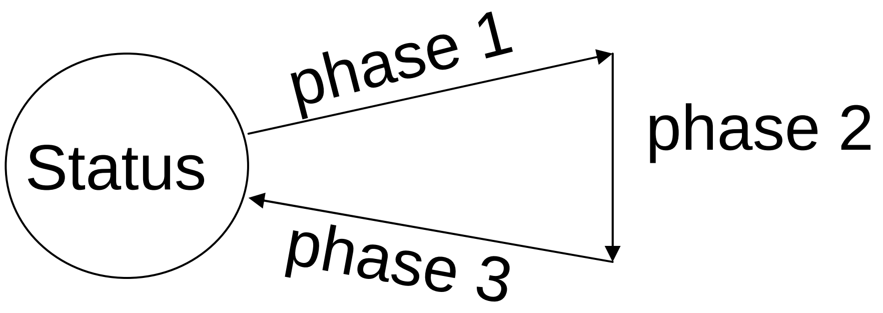
Steps

Execution of a model consists of a sequence of (status, step) pairs.



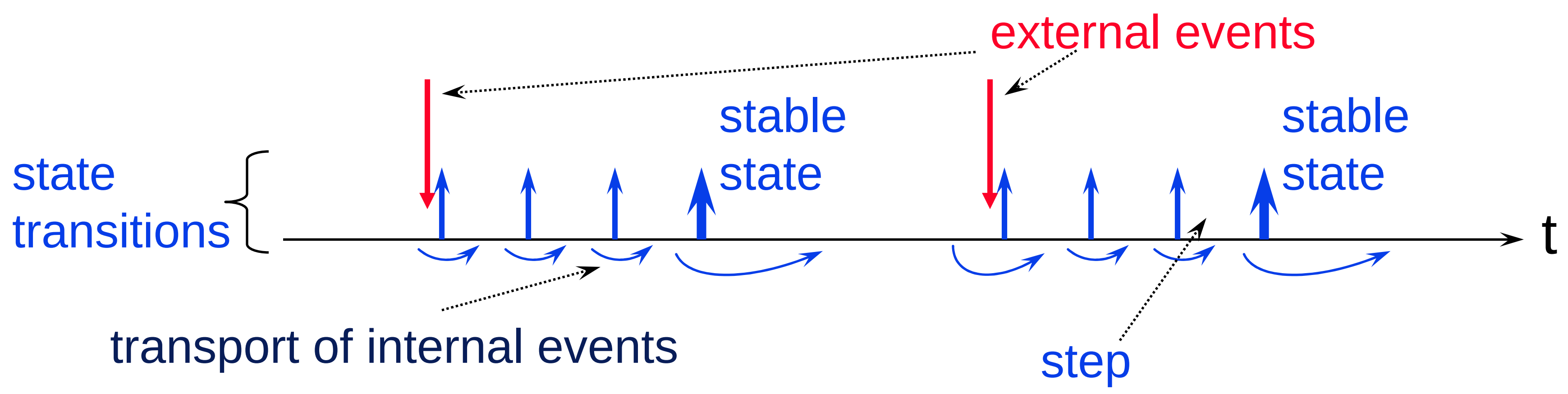
Status= values of all variables + set of events + current time

Step = execution of the three phases

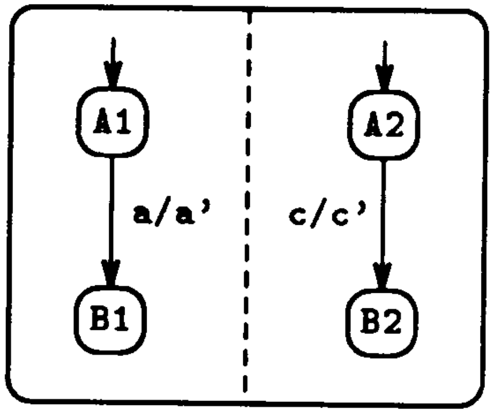


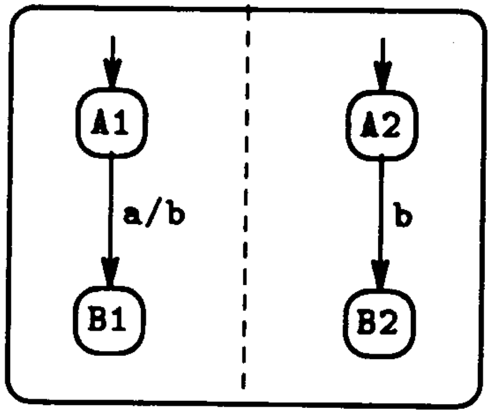
More on semantics of StateCharts

- Unfortunately, there are several time-semantics of StateCharts in use. This is one possibility:
  - A step is executed in arbitrarily small time.
  - Internal (generated) events exist only within the next step.
  - External events can only be detected after a stable state has been reached.



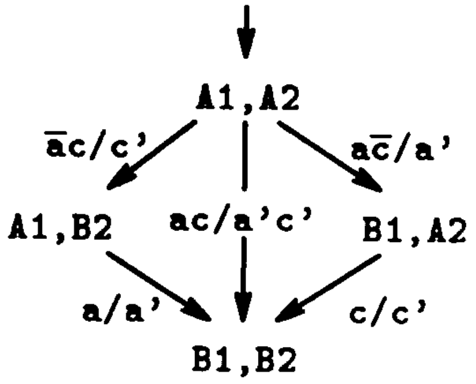
Examples

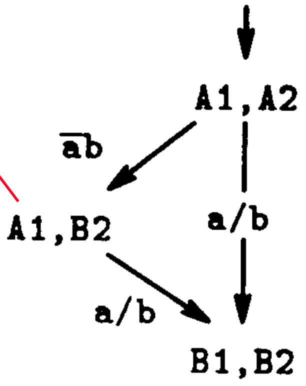




state diagram:

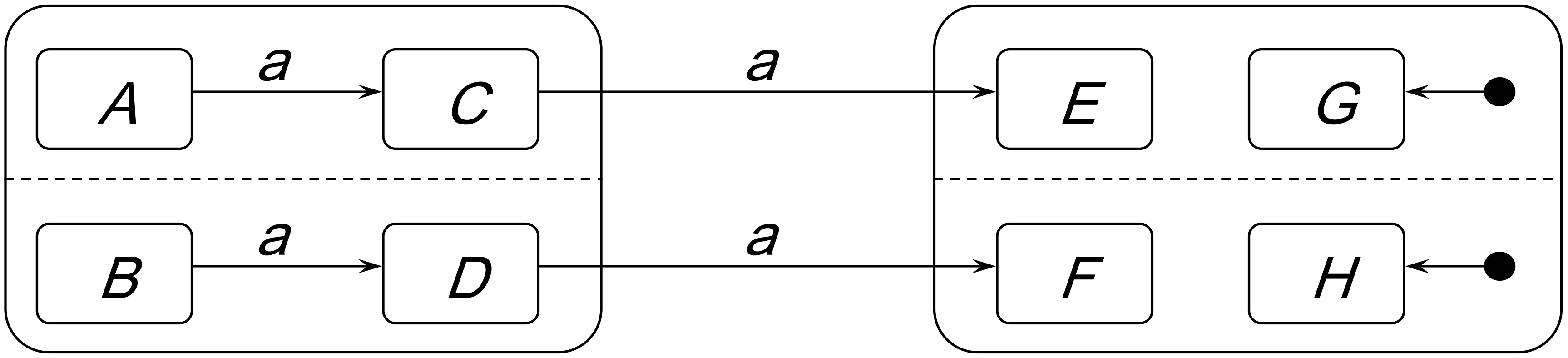
stable state





Example

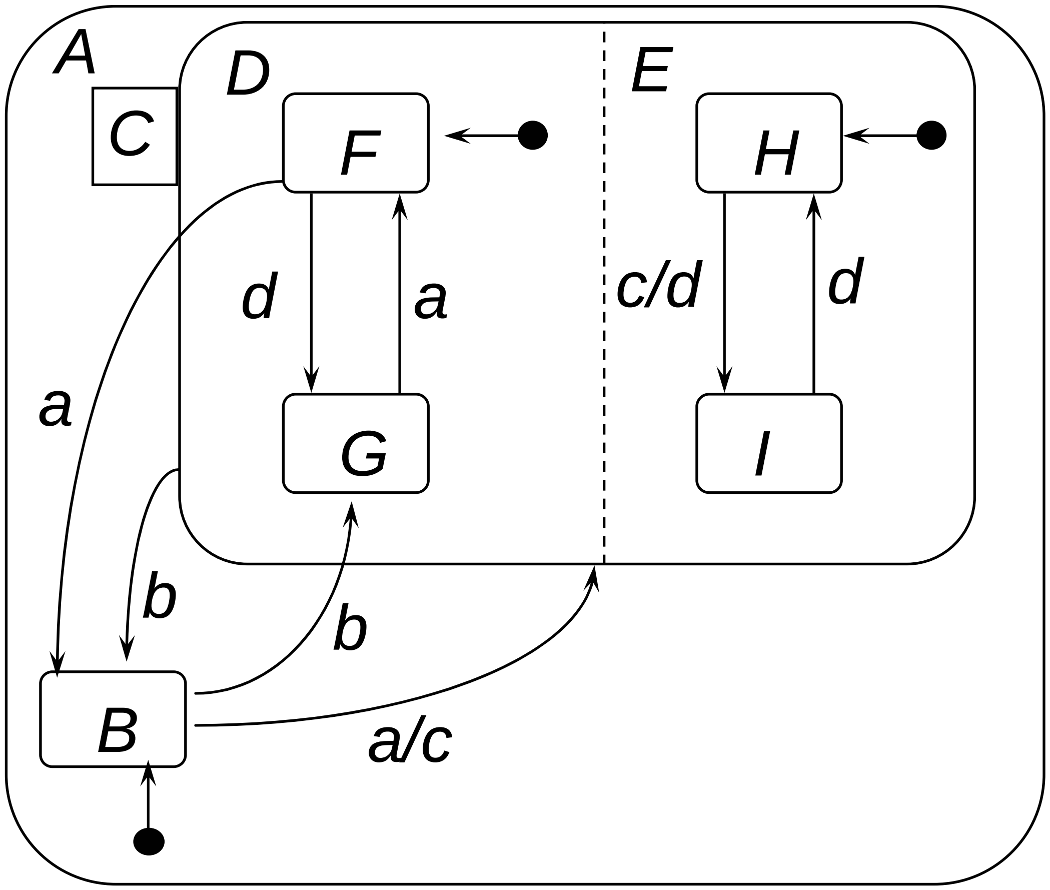
- Non-determinism



state diagram:

$$\begin{tikzcd}
 & & E,H \\
A,B \arrow[r, "a"] & C,D \arrow[ur, "a"] \arrow[dr, "a"'] & \\
 & & F,G
\end{tikzcd}$$

Example



state diagram (only stable states are represented):

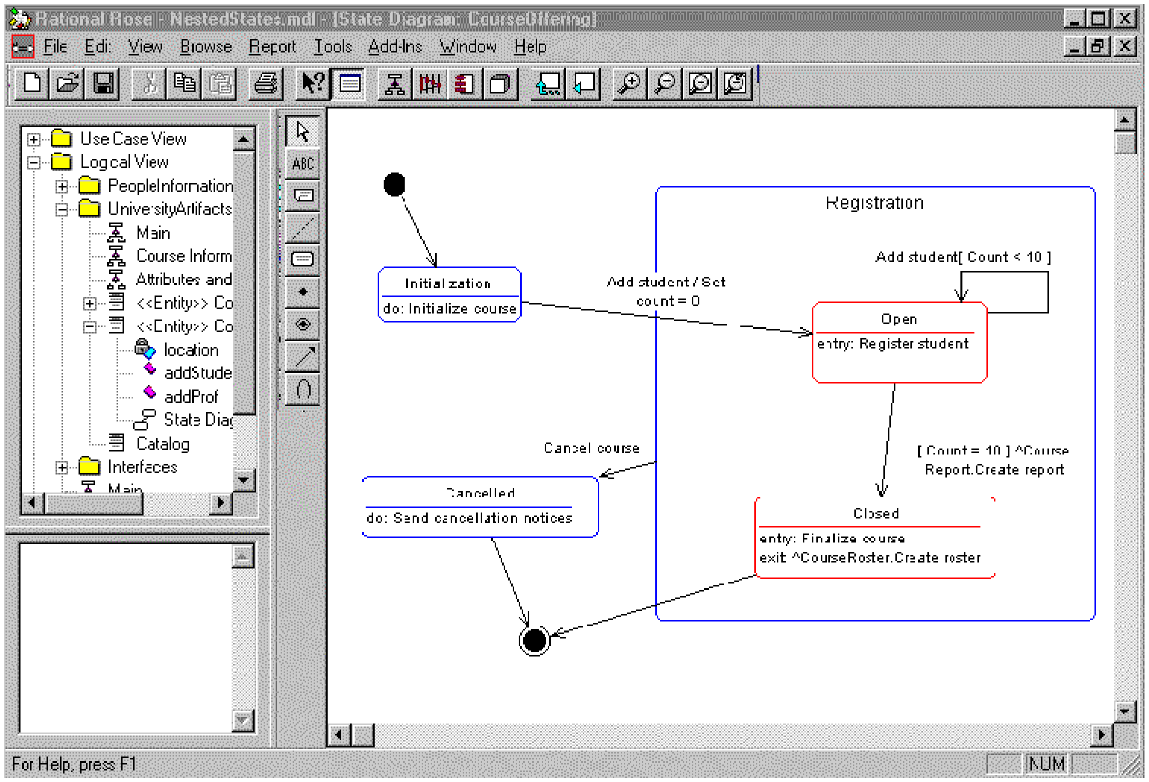
$$\begin{tikzcd}
B \arrow[d, shift right, "a \text{ or } b"'] \\
G,H \arrow[u, shift right, "b"'] \arrow[d, "a \text{ and } \overline{b}"] \\
F,H \arrow[uu, bend right=60, "a \text{ or } b"']
\end{tikzcd}$$

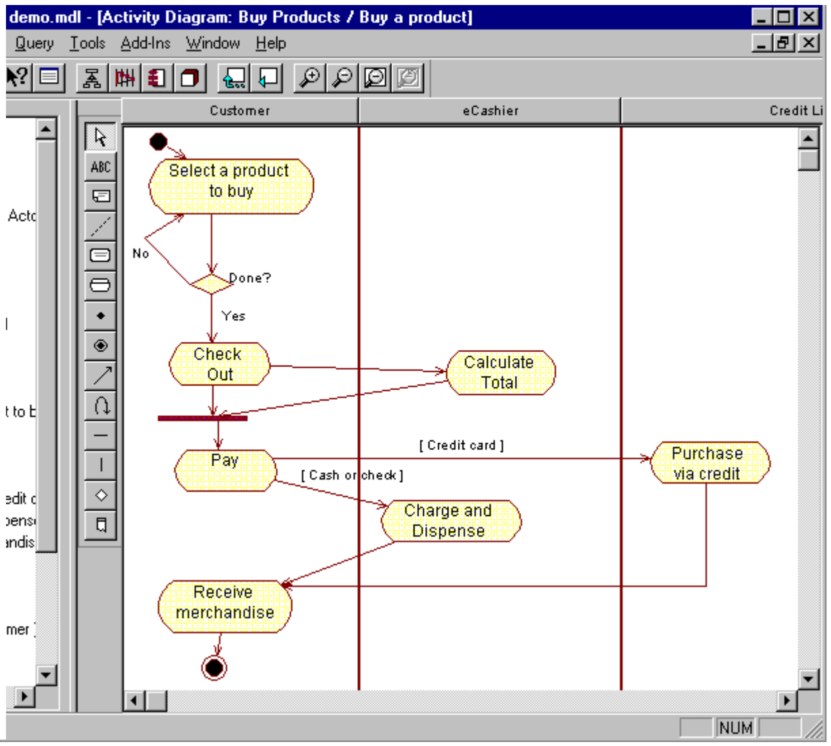
Summary

- ***Advantages*** of hierarchical state machines:
  - Simple transformation into efficient hardware and software implementations.
  - Efficient simulation.
  - Basis for formal verification (usually via symbolic model checking), if in reactions only events are generated.
- ***Disadvantages***:
  - Intricate for large systems, limited re-usability of models.
  - No formal representation of operations on data.
  - Large part of the system state is hidden in variables. This limits possibilities for efficient implementation and formal verification.

Example UML

- ***UML (unified modeling language)*** is used for the specification of large software systems and embedded (real-time) systems. The dynamics of a system are modeled using StateCharts and ActivityCharts (similar to Petri Nets).

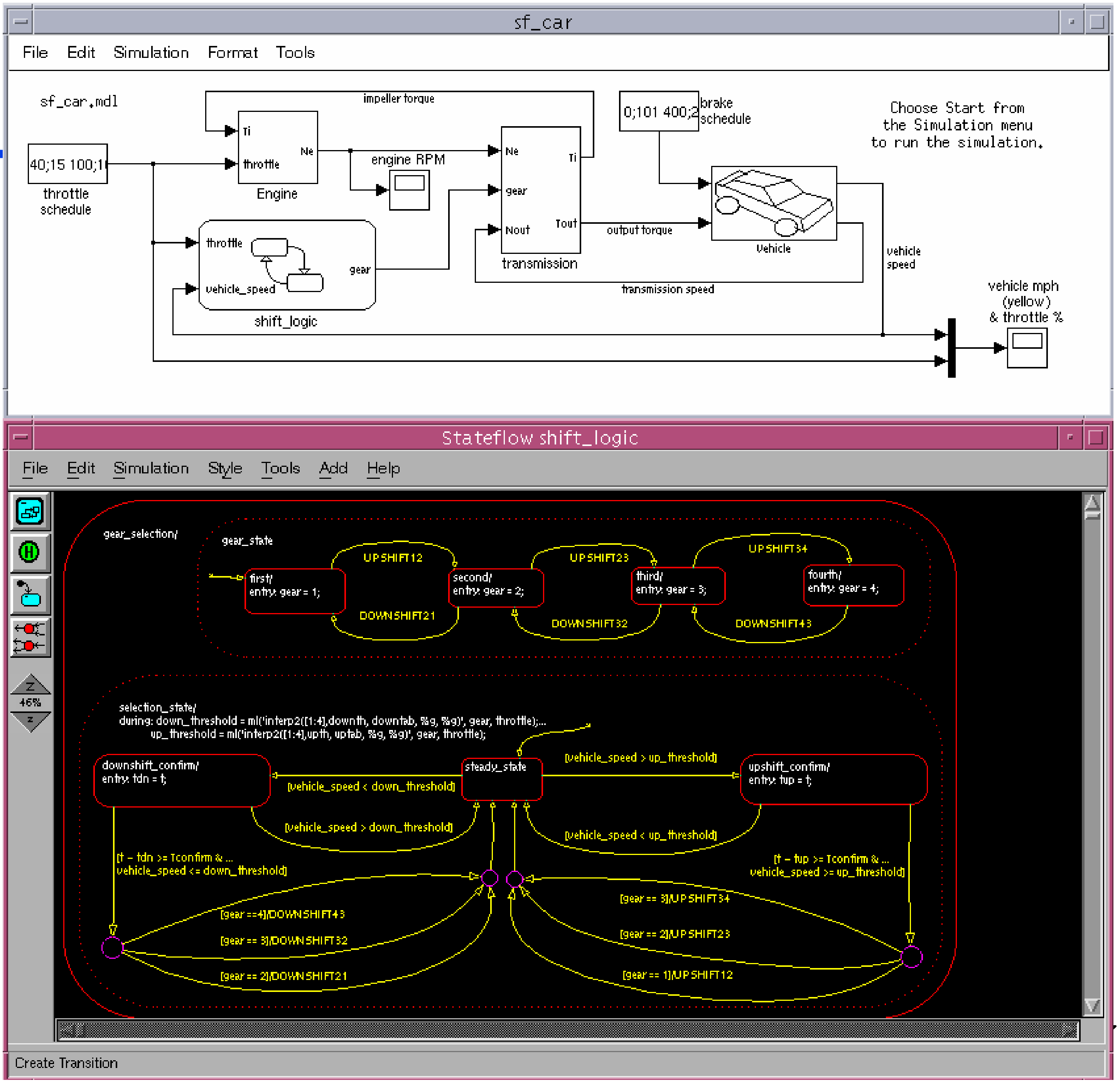




StateFlow

- Part of Matlab-Simulink

- Combines discrete event and continuous models



sf_car
File Edit Simulation Format Tools
sf_car.mdl
throttle schedule
Engine
engine RPM
impeller torque
transmission
output torque
transmission speed
brake schedule
Vehicle
vehicle speed
shift_logic
Choose Start from the Simulation menu to run the simulation.
vehicle mph (yellow) & throttle %
Stateflow shift_logic
File Edit Simulation Style Tools Add Help
gear_selection/
gear_state
first/ entry: gear = 1;
second/ entry: gear = 2;
third/ entry: gear = 3;
fourth/ entry: gear = 4;
UPSHIFT12
UPSHIFT23
UPSHIFT34
DOWNSHIFT21
DOWNSHIFT32
DOWNSHIFT43
selection_state/
during: down_threshold = ml('interp2([1:4],downth, downtab, %g, %g)', gear, throttle);...
up_threshold = ml('interp2([1:4],upth, uptab, %g, %g)', gear, throttle);
downshift_confirm/ entry: tdn = t;
steady_state
upshift_confirm/ entry: tup = t;
[vehicle_speed > up_threshold]
[vehicle_speed < down_threshold]
[vehicle_speed > down_threshold]
[vehicle_speed < up_threshold]
[t - tdn >= Tconfirm & ... vehicle_speed <= down_threshold]
[t - tup >= Tconfirm & ... vehicle_speed >= up_threshold]
[gear ==4]/DOWNSHIFT43
[gear == 3]/DOWNSHIFT32
[gear == 2]/DOWNSHIFT21
[gear == 3]/UPSHIFT34
[gear == 2]/UPSHIFT23
[gear == 1]/UPSHIFT12
Create Transition

Petri nets - Motivation

- In contrary to hierarchical state machines, state transitions in ***Petri nets*** are ***asynchronous***. The ordering of transitions is partly uncoordinated; it is specified by a partial order.
- Therefore, Petri nets can be used to model ***concurrent*** distributed ***systems***.
- There are many models of computation in use that are variants or specializations of Petri nets, e.g.
  - activity charts (UML)
  - data flow graphs and marked graphs
- Finite state machines can be modeled in Petri nets.

Net graph

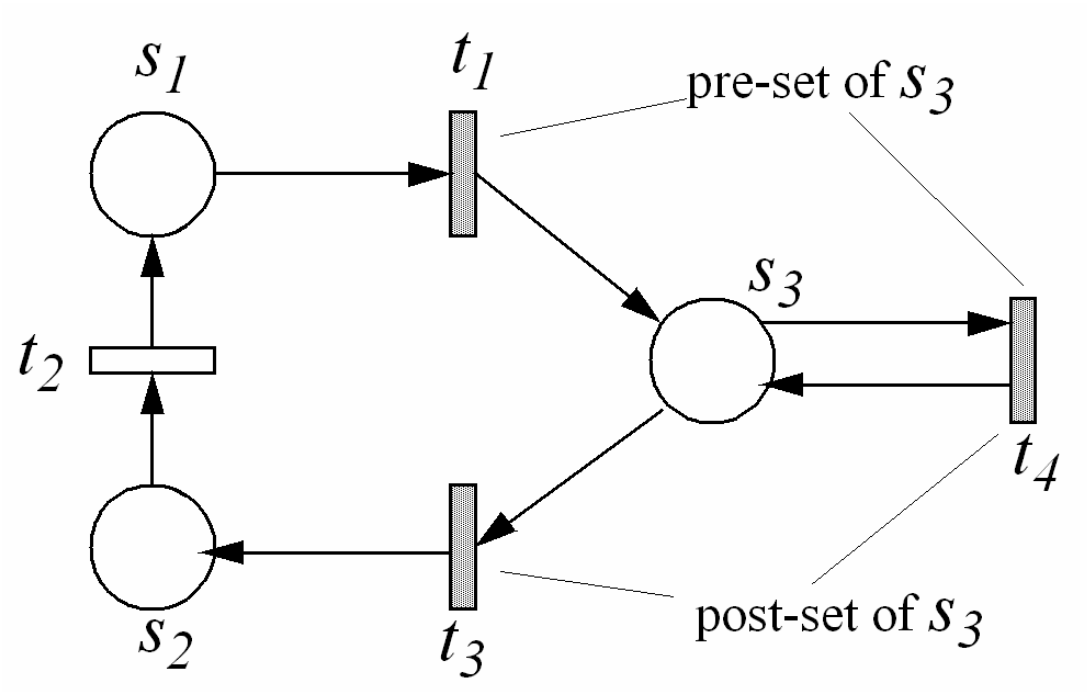
A net graph is a tupel $N = (S, T, F)$ with $S \cap T = \emptyset$. The elements $s \in S$ and $t \in T$ are denoted as places and transitions, respectively, and define the nodes of the net. The relation $F \subseteq (S \times T) \cup (T \times S)$ defines the edges of the net.

The pre-set and post-set of a place or transition x are defined as

$$\bullet x = \{y \in S \cup T \ : \ (y, x) \in F\}$$
$$x \bullet = \{y \in S \cup T \ : \ (x, y) \in F\}$$

Net graph - example

- The net-graph is a bipartite graph.



Petri net - definition

A tupel (S, T, F, M, M_0) denotes a Petri net. Then (S, T, F) is a net-graph, the marking M is a function $M : S \longrightarrow \mathbf{Z}_{\geq 0}$ and M_0 denotes the initial marking.

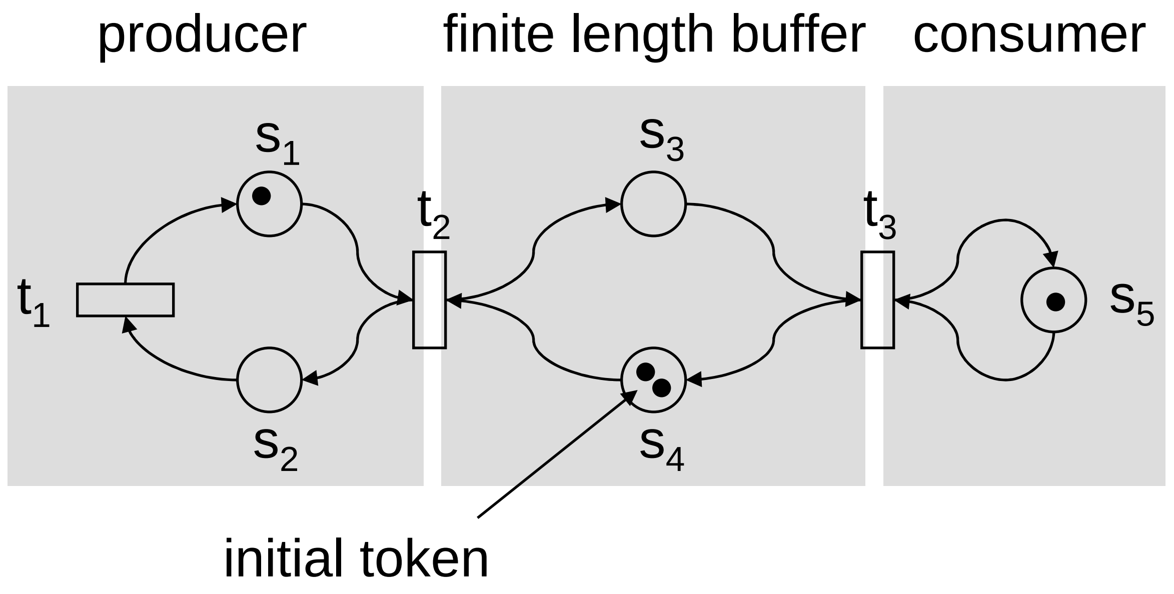
- The state of a Petri net is its marking *M*.
- *M(s)* denotes the marking of a place *s*. Usually, we say that place *S* contains *M(s)* token. In other words, the distribution of tokens on places defines the state of a Petri net.
- The dynamics of a Petri net is defined by a ‘token game’.

Token game of Petri nets

A marking M activates a transition $t \in T$ iff $M(s) \geq 1$ for all $s \in \bullet t$. If a transition t is activated by M, then a state transition to the marking M' happens eventually. The associated state transition function with $M' = f(M,t)$ is

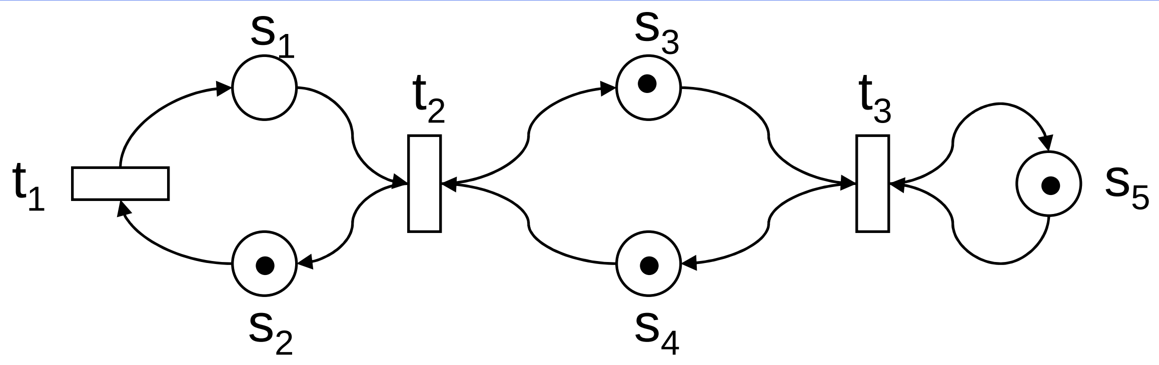
$$M'(s) = \begin{cases} M(s) - 1 & \text{if } s \in \bullet t \setminus t\bullet \\ M(s) + 1 & \text{if } s \in t\bullet \setminus \bullet t \\ M(s) & \text{otherwise} \end{cases}$$

Example



- Initial state represented as state vector: $M_0 = (1,0,0,2,1)$
- Activated transitions: t_2
- After ***firing*** t_2: $M = (0,1,1,1,1)$.

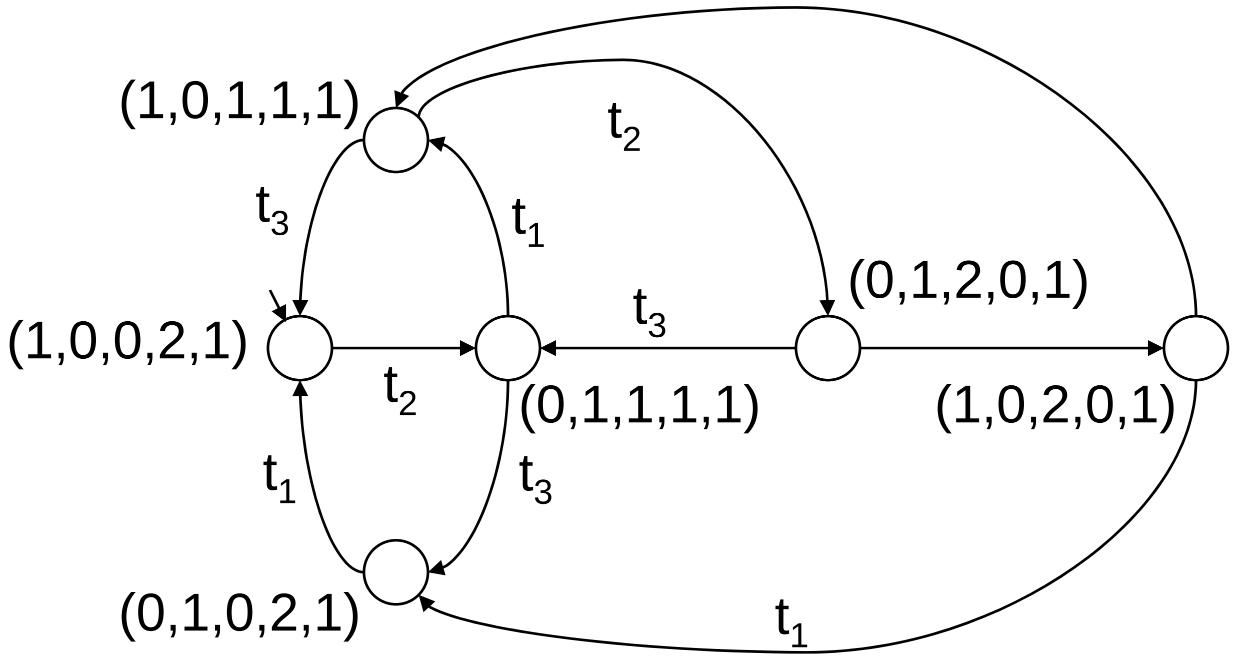
Example continued



- Activated transitions: t_1, t_3 .
- Non-deterministically, one of them is chosen for ***firing***, e.g. t_3. Then we obtain as new state $M = (0,1,0,2,1)$.
- We can see the 'properties' of Petri nets: ***Asynchronous*** firing of activated transitions, possibility to model distributed systems.

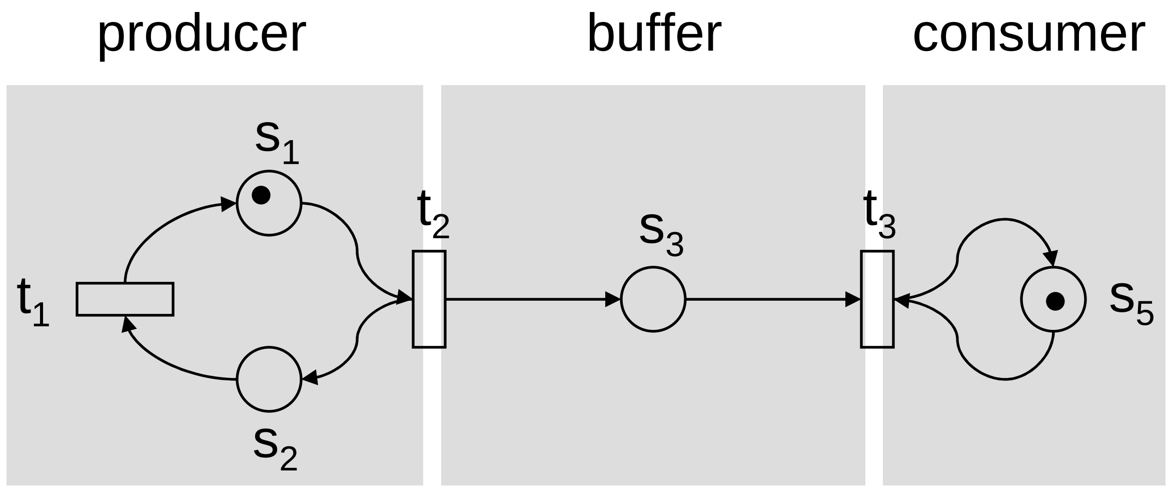
Example continued

- If the number of token in the network is bounded, we can determine a finite state transition graph.

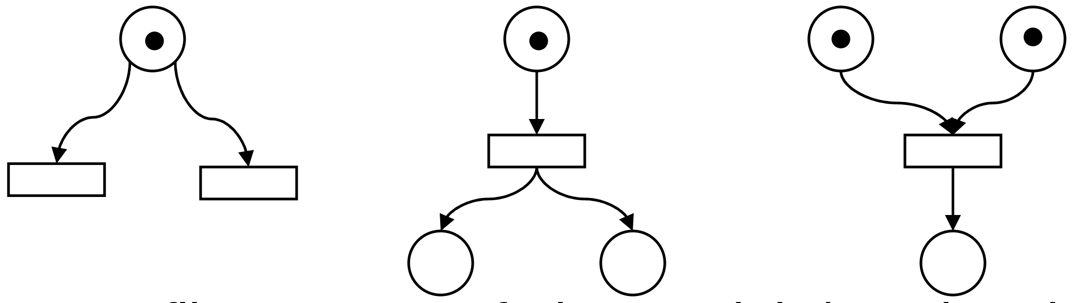


Modeling capabilities

- But we can also systems with unbounded state set!



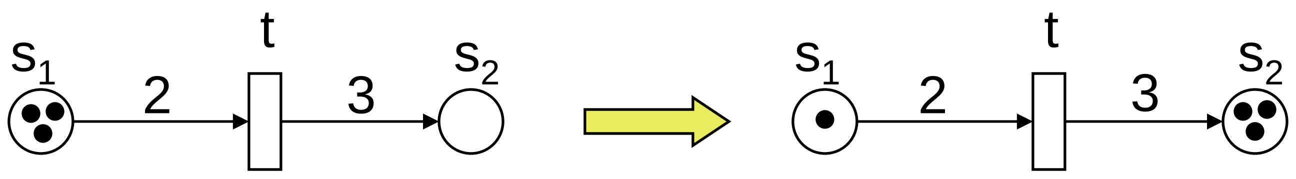
- And we can model basic scenarios such as



conflict fork join/synchronization

Common model extensions

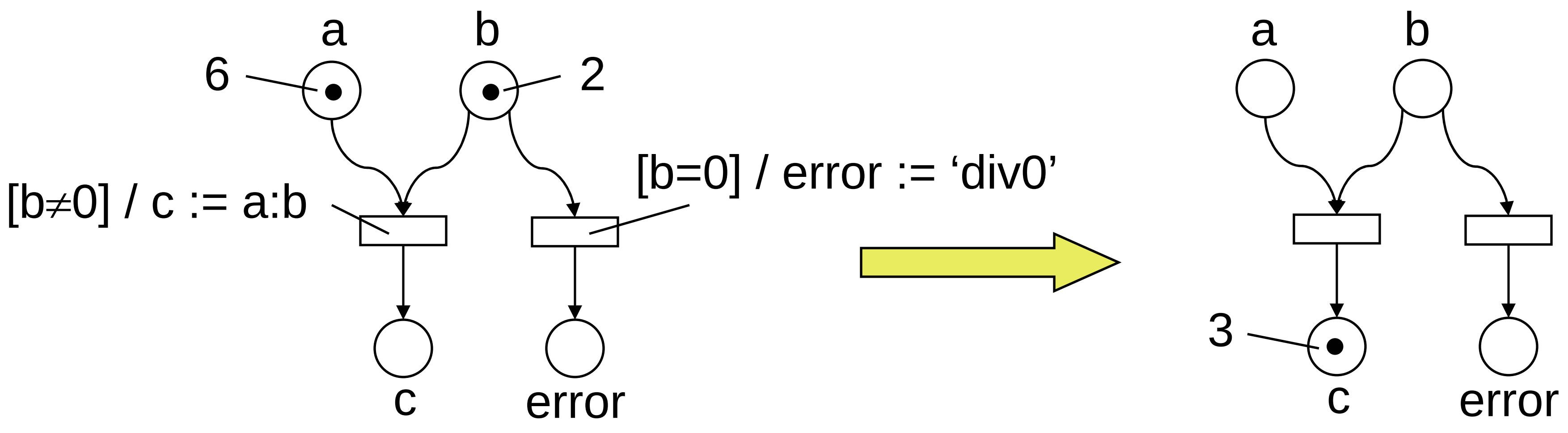
- Associating ***weights*** *W* to edges:
  - Transition *t* is enabled if there are at least $W(s_1,t)$ token in s_1.
  - If transition *t* fires, then $W(t,s_2)$ token are added to place s_2 and $W(s_1,t)$ token are removed from s_1.



- Adding ***time*** to transitions:
  - Specification of discrete event systems with time!
  - One possibility: A transition fires iff it was continuously activated for a certain time period.

Common model extensions

- ***Individual tokens***:
  - Tokens can ‘carry’ data.
  - Transitions operate on data of input tokens and associate data to output token.
  - The activation of a transition can be dependent on data of token in places of its pre-set.



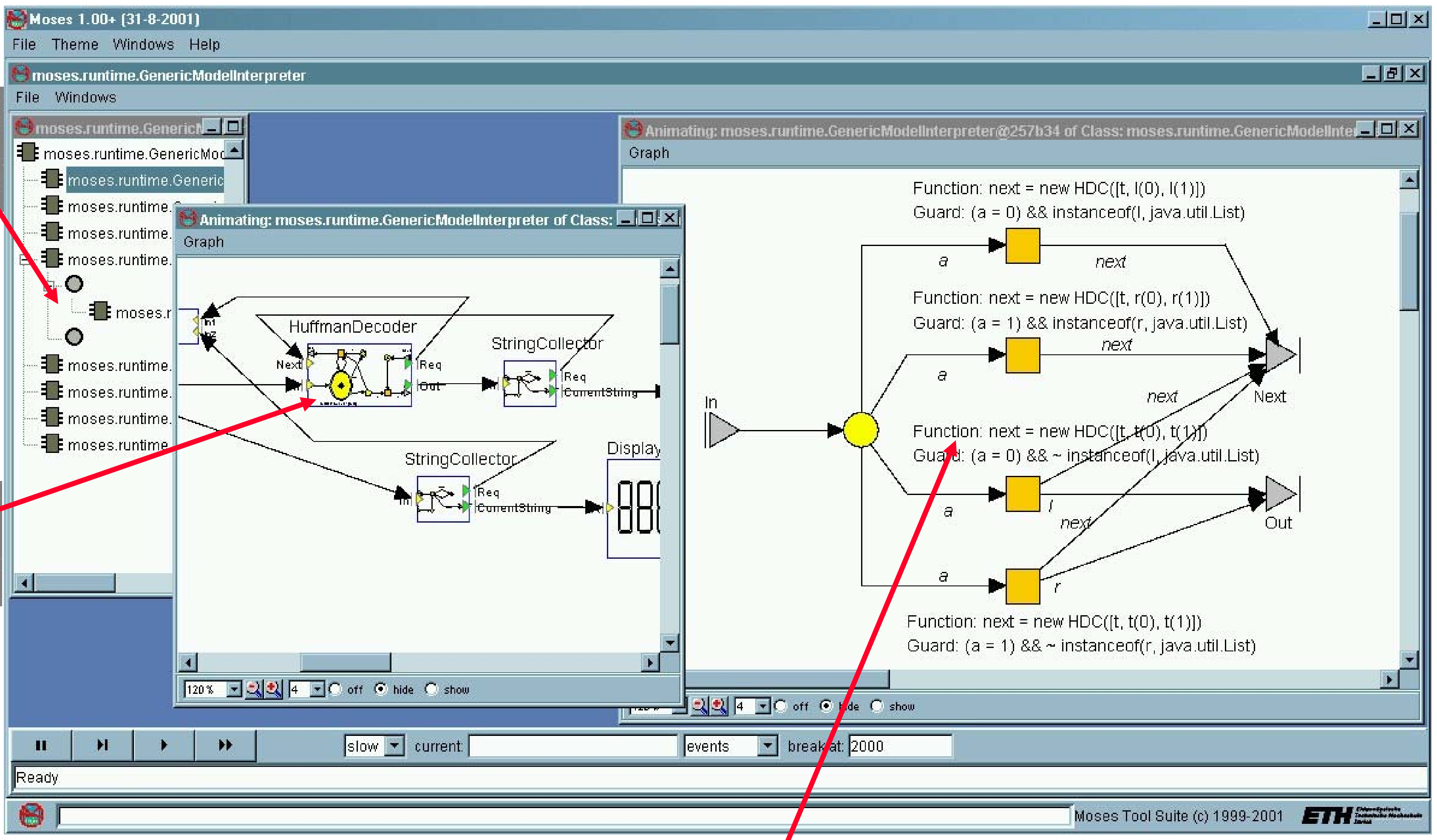
What can we do with Petri nets?

- We can ***model*** (timed, distributed) discrete event systems.
- We can ***simulate*** them using tools, e.g. MOSES.
- We can ***analyze*** their ***timing*** properties. Methods exist, if the delays of token are constant or even determined by stochastic processes.
- We can ***answer questions*** like:
  - What is the maximum number of tokens in a specific place?
  - Is the Petri net bounded (bounded number of tokens under any firing sequence)?
  - Does the Petri net eventually enter a state where no transition is activated (deadlock) ?
  - Several methods are available to answer these questions (not part of this lecture).

Example MOSES

Component hierarchy

Petri net component


Moses 1.00+ (31-8-2001)
File Theme Windows Help
moses.runtime.GenericModelInterpreter
File Windows
moses.runtime.GenericModelInterpreter
Animating: moses.runtime.GenericModelInterpreter of Class:
Animating: moses.runtime.GenericModelInterpreter@257b34 of Class: moses.runtime.GenericModelInte
Graph
HuffmanDecoder
StringCollector
Display
Next
Req
Out
CurrentString
In
a
next
Next
Out
l
r
120 %
off
hide
show
slow
current:
events
break at: 2000
Ready
Moses Tool Suite (c) 1999-2001

```
Function: next = new HDC([t, l(0), l(1)])
Guard: (a = 0) && instanceof(l, java.util.List)
Function: next = new HDC([t, r(0), r(1)])
Guard: (a = 1) && instanceof(r, java.util.List)
Function: next = new HDC([t, t(0), t(1)])
Guard: (a = 0) && ~ instanceof(l, java.util.List)
Function: next = new HDC([t, t(0), t(1)])
Guard: (a = 1) && ~ instanceof(r, java.util.List)
```

Petri net