

An Introduction to Reactive and Real-time System Modelling

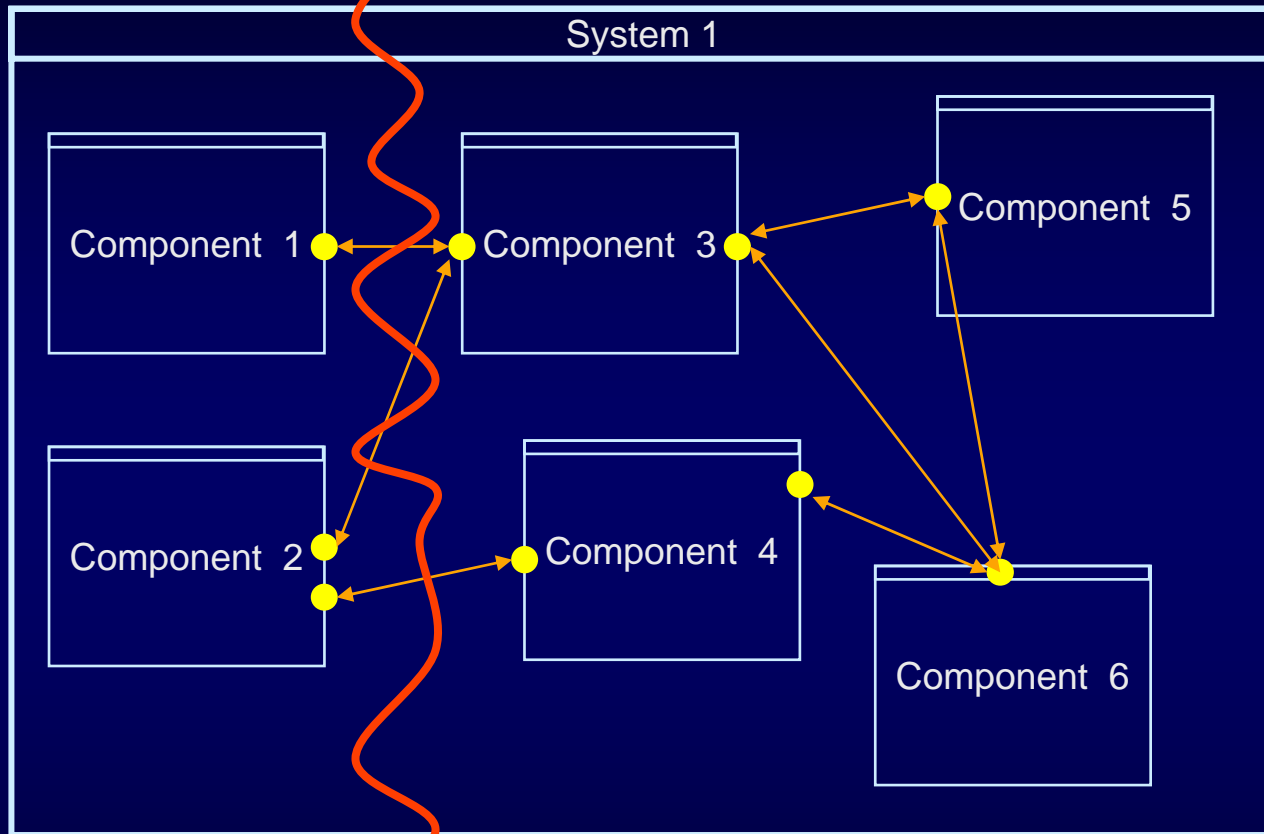
(slides by Brian Nielsen)

Agenda

- Finite state machine (FSM)
- High-level FSM languages
- Modelling **untimed** systems using Uppaal
- Timed automaton (TA)
- Modelling **timed** systems using Uppaal
- Verification using Uppaal

Finite State Machine (FSM)

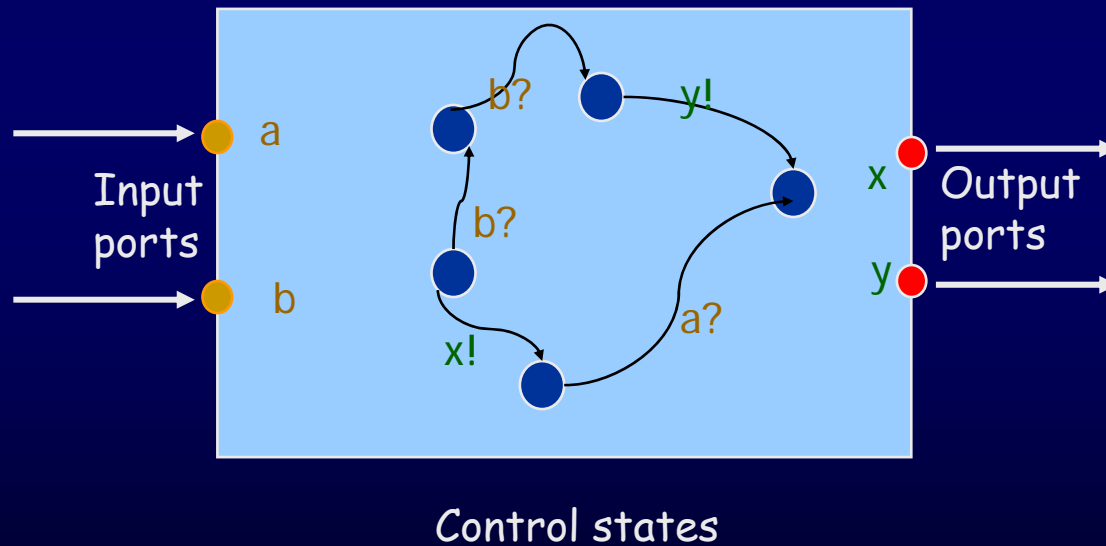
System Structure



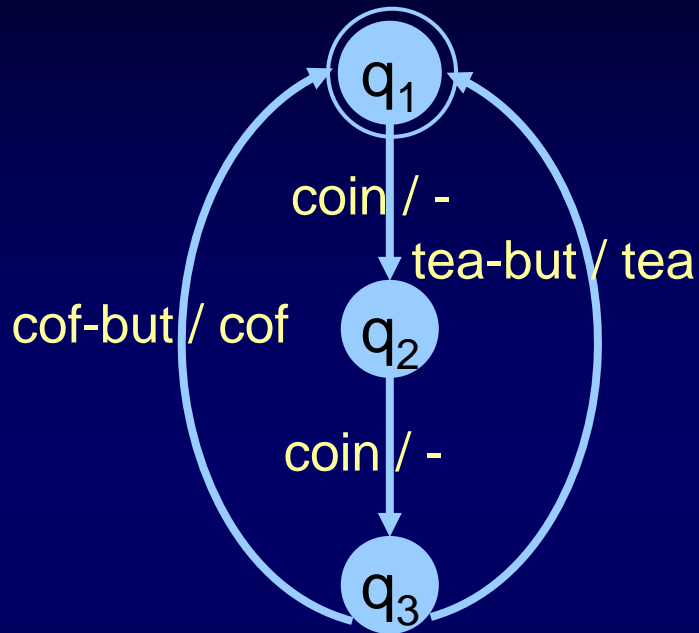
- How do we model components?
- How do components interact?
- How do we specify environment assumptions?
- How do we ensure correct behaviour?

Component Behavior

Unified Model: **State Machine**



Finite State Machine (Mealy machine)



condition		effect	
current state	input	output	next state
q ₁	coin	-	q ₂
q ₂	coin	-	q ₃
q ₃	cof-but	cof	q ₁
q ₃	tea-but	tea	q ₁

Inputs = {cof-but, tea-but, coin}

Outputs = {cof, tea}

States: {q₁, q₂, q₃}

Initial state = q₁

Transitions = {

(q₁, coin, -, q₂),

(q₂, coin, -, q₃),

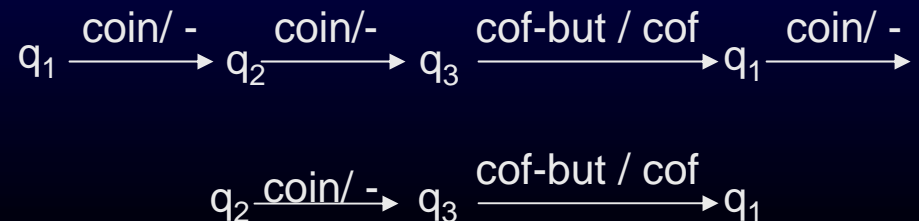
(q₃, cof-but, cof, q₁),

(q₃, tea-but, tea, q₁)

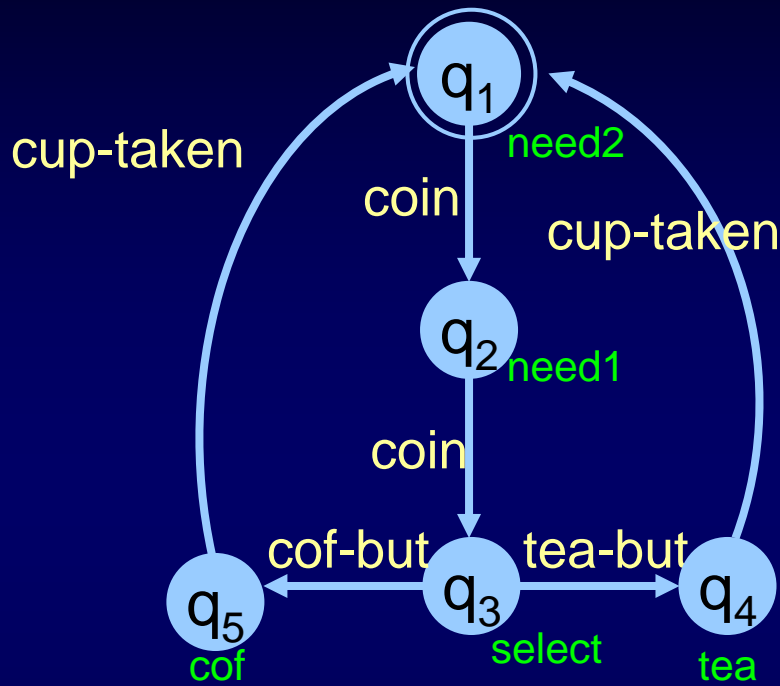
}

In Mealy machine the **output** depends on the **current state** as well as the **input**

Sample run:



Finite State Machine (Moore machine)



condition		effect
current state	input	next state
q ₁	coin	q ₂
q ₂	coin	q ₃
q ₃	cof-but	q ₅
q ₃	tea-but	q ₄
q ₅	cup-taken	q ₁
q ₄	cup-taken	q ₁

condition	effect
current state	activity
q ₁	need2
q ₂	need1
q ₃	select
q ₅	cof
q ₄	tea

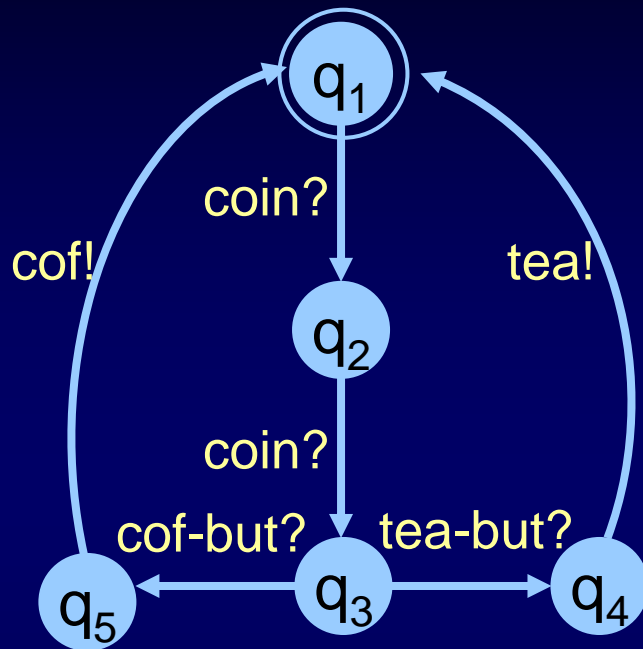
In Moore machine the **output** (or "activity") depends on the **current state** only

Input sequence: coin.coin.cof-but.cup-taken.coin.cof-but

Output sequence: need2.need1.select.cof. need2.need1.select.cof

need2=display shows "insert two coins"

Input-Output FSM (IO-FSM)



condition		effect
current state	action	next state
q ₁	coin?	q ₂
q ₂	coin?	q ₃
q ₃	cof-but?	q ₅
q ₃	tea-but?	q ₄
q ₄	tea!	q ₁
q ₅	cof!	q ₁

Inputs = {cof-but, tea-but, coin}

Outputs = {cof,tea}

States: {q₁,q₂,q₃}

Initial state = q₁

Transitions= {

(q₁, coin, q₂),
 (q₂, coin, q₃),
 (q₃, cof-but, q₅),
 (q₃, tea-but, q₄),
 (q₄, tea, q₁),
 (q₅, cof, q₁)
 }

Sample run:

q₁ $\xrightarrow{\text{coin?}}$ q₂ $\xrightarrow{\text{coin?}}$ q₃ $\xrightarrow{\text{cof-but?}}$ q₅ $\xrightarrow{\text{cof!}}$ q₁

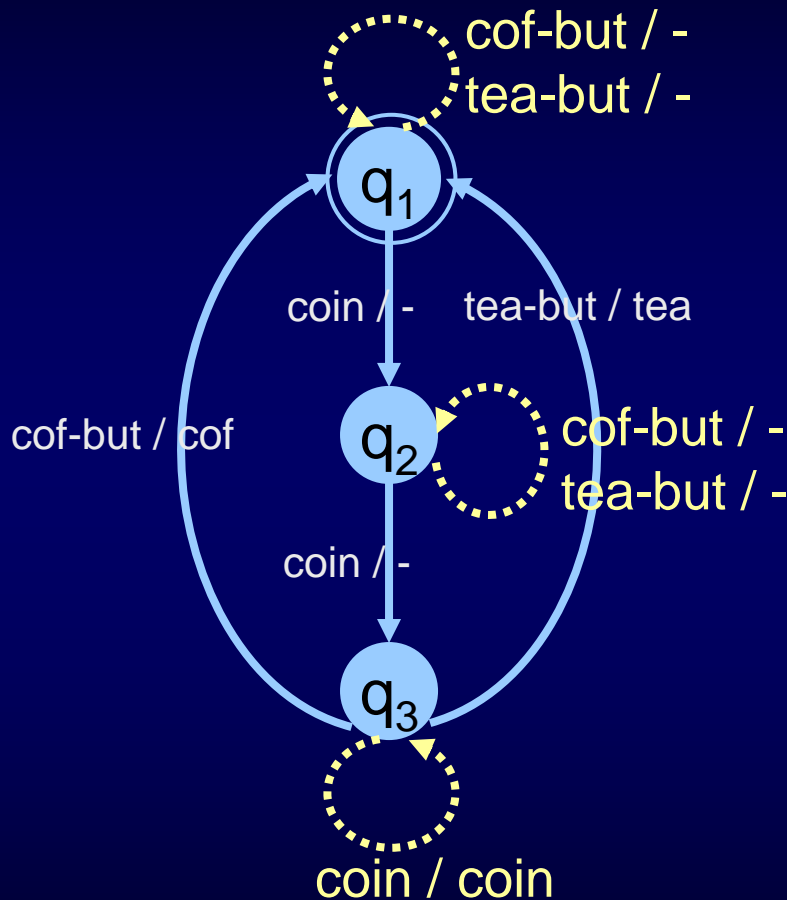
q₁ $\xrightarrow{\text{coin?}}$ q₂ $\xrightarrow{\text{coin?}}$ q₃ $\xrightarrow{\text{cof-but?}}$ q₅ $\xrightarrow{\text{cof!}}$ q₁

action trace: coin?.coin?.cof-but?.cof!.coin?.coin?.cof-but?.cof!

input sequence: coin.coin.cof-but.coin.coin.cof-but

Output sequence: cof.cof

Fully Specified FSM (Mealy)



condition		effect	
current state	input	output	next state
q_1	coin	-	q_2
q_2	coin	-	q_3
q_3	cof-but	cof	q_1
q_3	tea-but	tea	q_1
q_1	cof-but	-	q_1
q_1	tea-but	-	q_1
q_2	cof-but	-	q_2
q_2	tea-but	-	q_2
q_3	coin	coin	q_3

for each state
for each input

...

Mealy FSM as program (1)

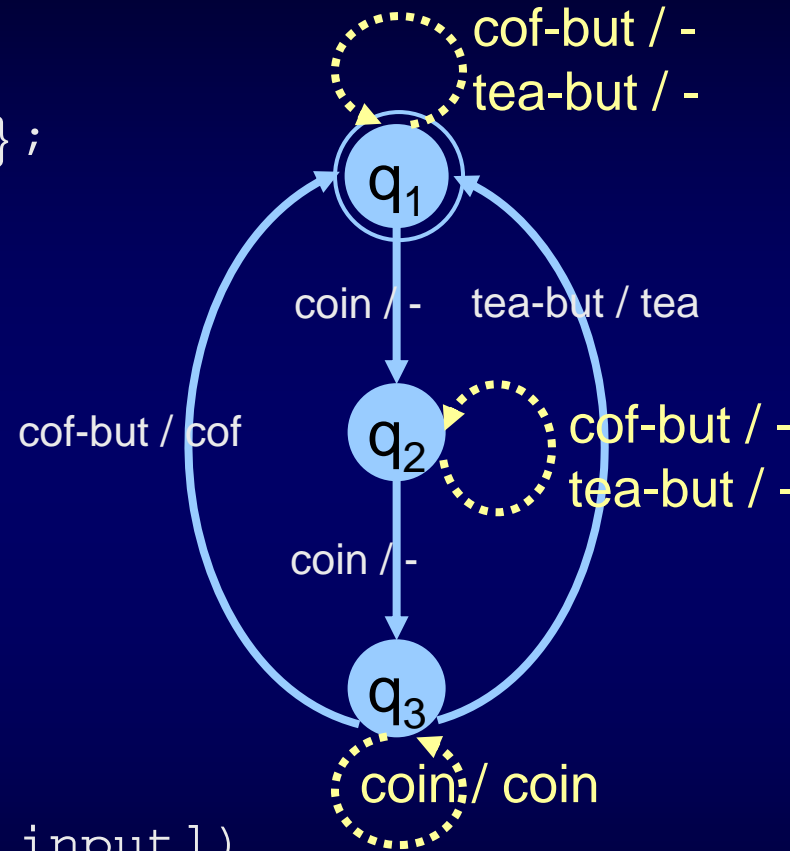
```
enum currentState {q1,q2,q3};
enum input {coin, cof_but,tea_but};
int nextStateTable[3][3] = {
    q2,q1,q1,
    q3,q2,q2,
    q3,q1,q1 };

```

```
int outputTable[3][3] = {
    0,0,0,
    0,0,0,
    coin,cof,tea};

```

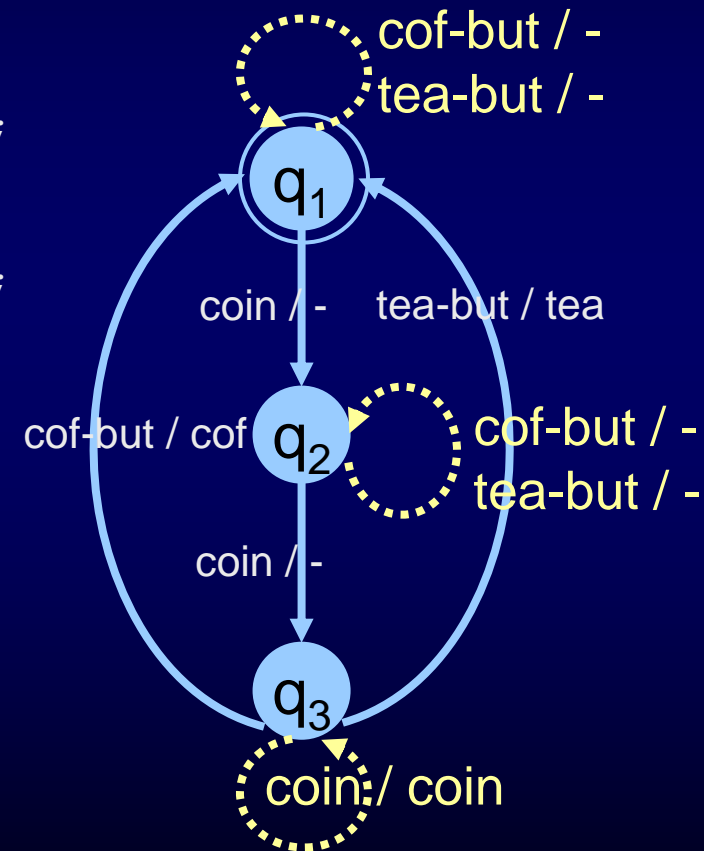
```
While(input=waitForInput()) {
    OUTPUT(outputTable[currentState,input])
    currentState:=nextStateTable[currentState,input];
}
```



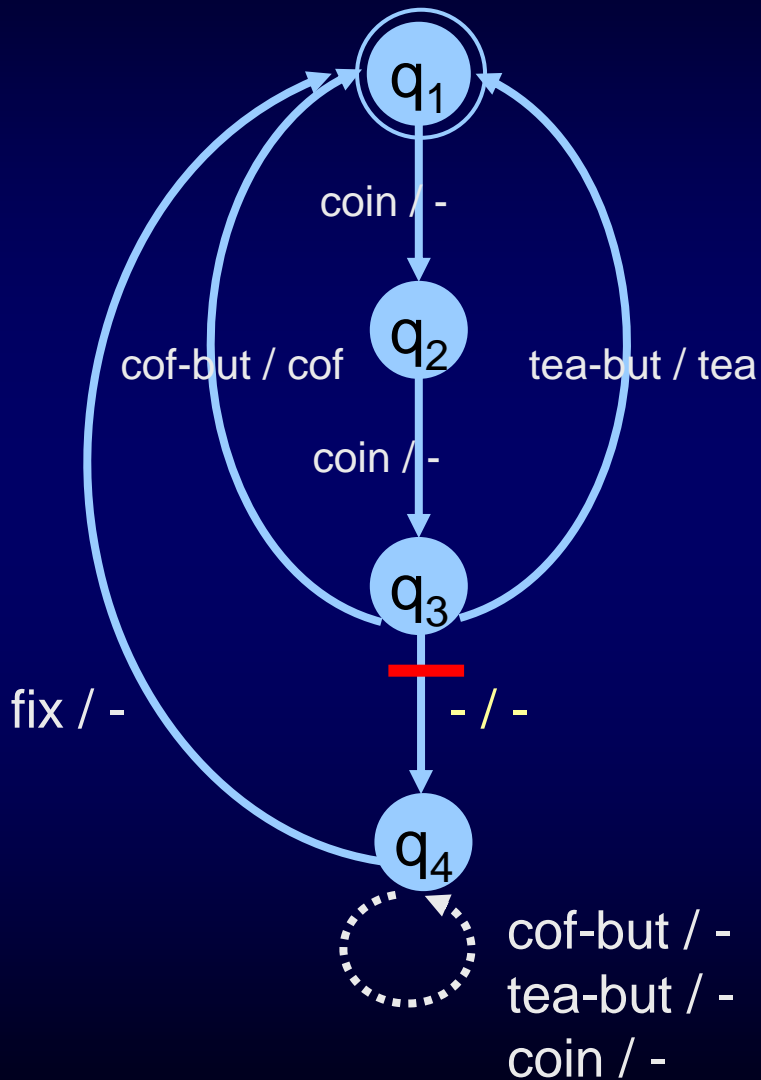
Mealy FSM as program (2)

```
enum currentState {q1,q2,q3};
enum input {coin,cof,tea_but,cof_but};

While(input=waitForInput){
  Switch(currentState){
  case q1: {
    switch (input) {
      case coin: currentState:=q2; break;
      case cof_but:
      case tea_but: break;
      default: ERROR("Unexpected Input");
    }
    break;
  case q3: {
    switch(input) {
      case cof_but: {currentState:=q3;
                     OUTPUT(cof);
                     break;}
    }
  }
  ...
  default: ERROR("unknown currentState");
} // end of switch
}
```



Spontaneous Transitions



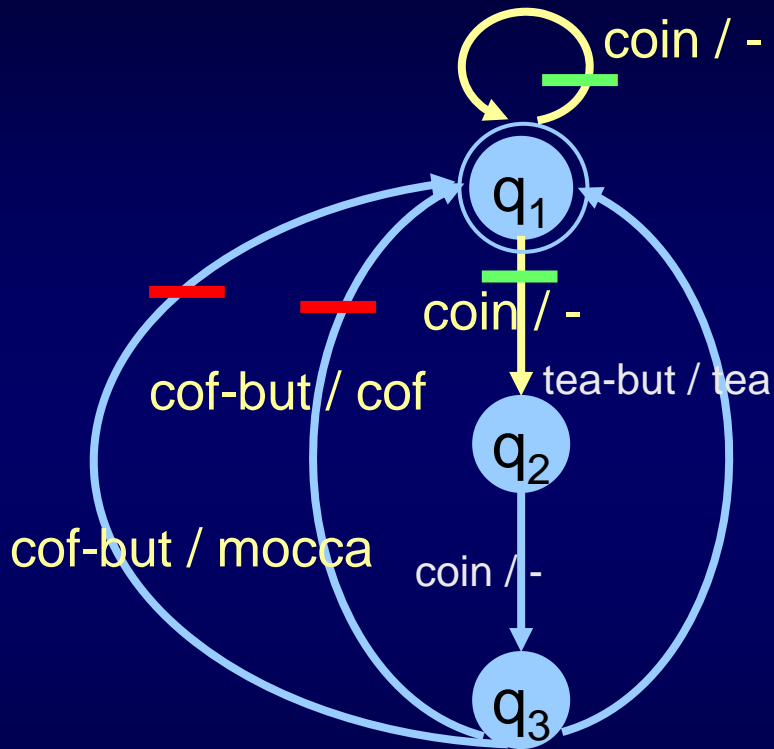
condition		effect	
current state	input	output	next state
q_1	coin	-	q_2
q_2	coin	-	q_3
q_3	cof-but	cof	q_1
q_3	tea-but	tea	q_1
q_3	-	-	q_4
q_4	fix	-	q_1

A spontaneous transition is a transition in response to **no** input at all.

alias: **internal** transition

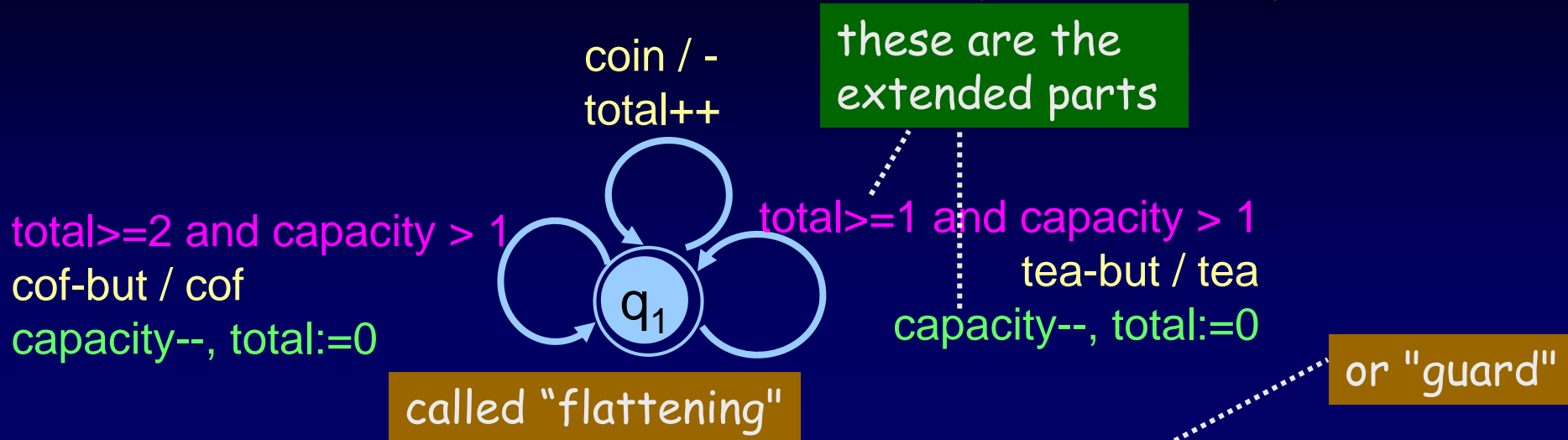
alias: **unobservable** transition

Non-deterministic FSM



condition		effect	
current state	input	output	next state
q ₁	coin	-	q ₂
q ₁	coin	-	q ₁
q ₂	coin	-	q ₃
q ₃	tea-but	tea	q ₁
q ₃	cof-but	cof	q ₁
q ₃	cof-but	mocca	q ₁

Extended FSM (EFSM)

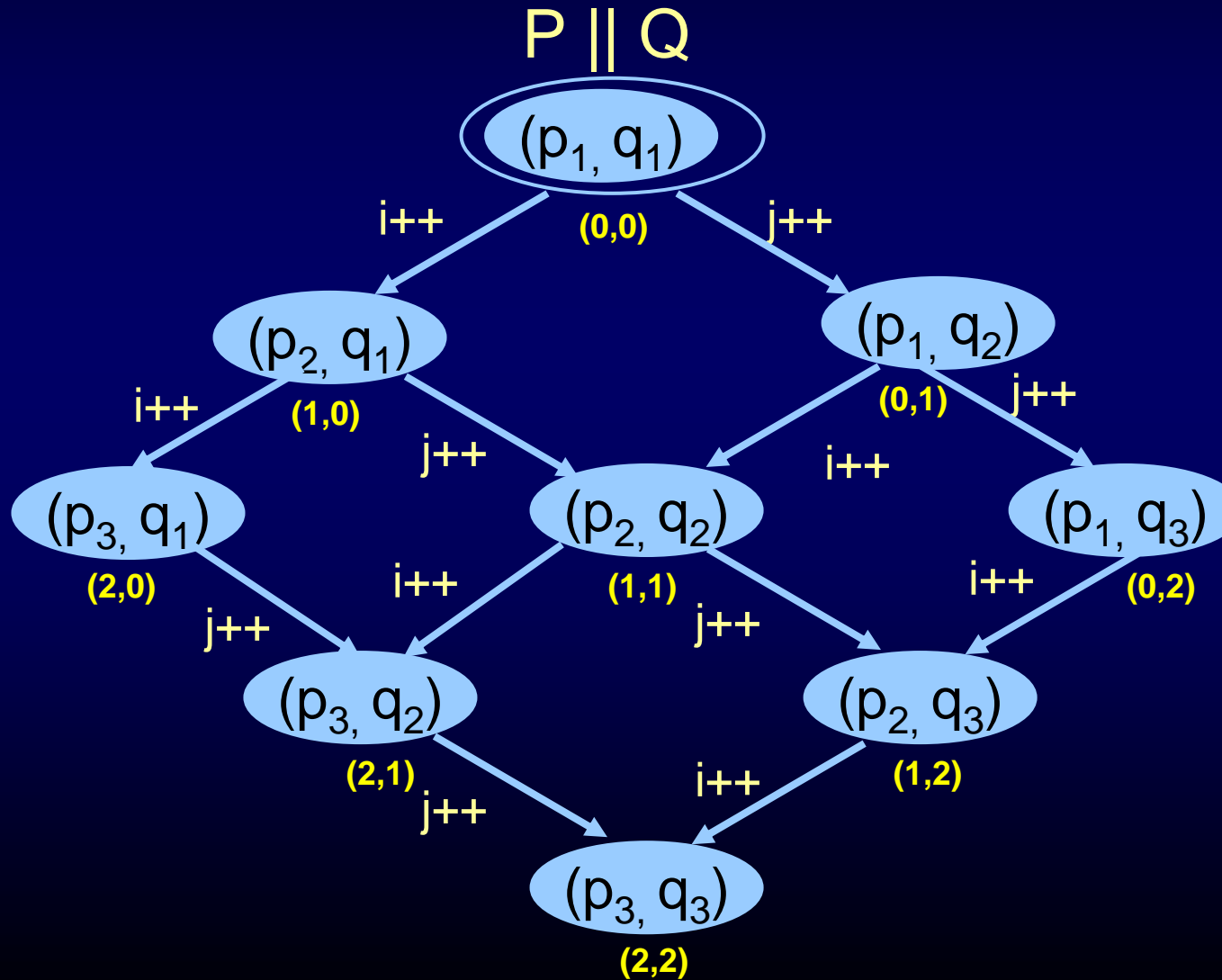
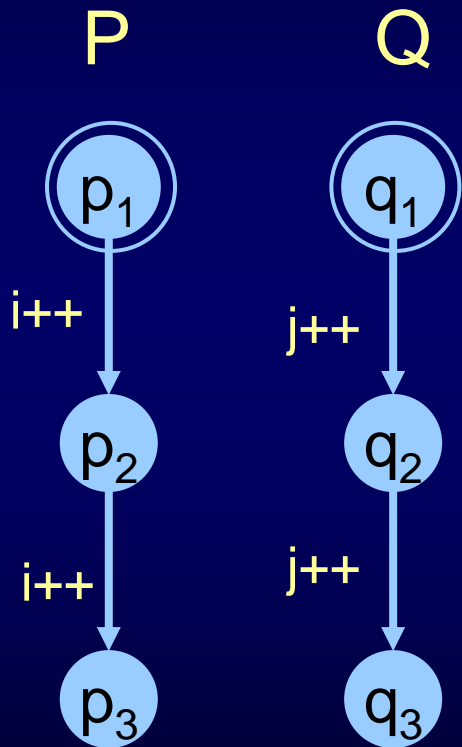


- EFSM = FSMs + variables + **enabling conditions** + **assignments**
- Can model the control aspect as well as the data aspect
- Can **be translated** into FSM if variables have bounded domains
- EFSM state: control location + variables' valuation

(q, total, capacity)

$(q_1, 0, 10) \xrightarrow{\text{coin / -}} (q_1, 1, 10) \xrightarrow{\text{coin / -}} (q_1, 2, 10) \xrightarrow{\text{cof-but / cof}} (q_1, 0, 9)$

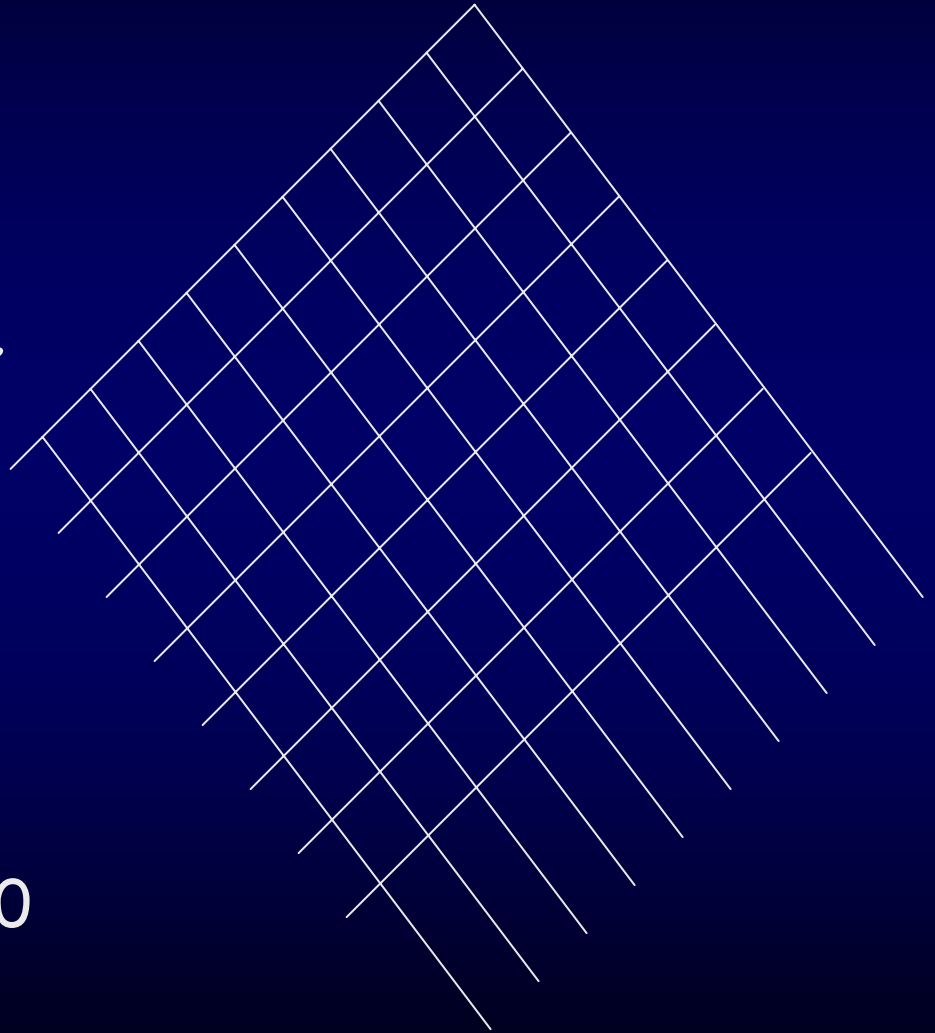
Parallel Composition (independent)



interleaving "execution"

State Explosion Problem

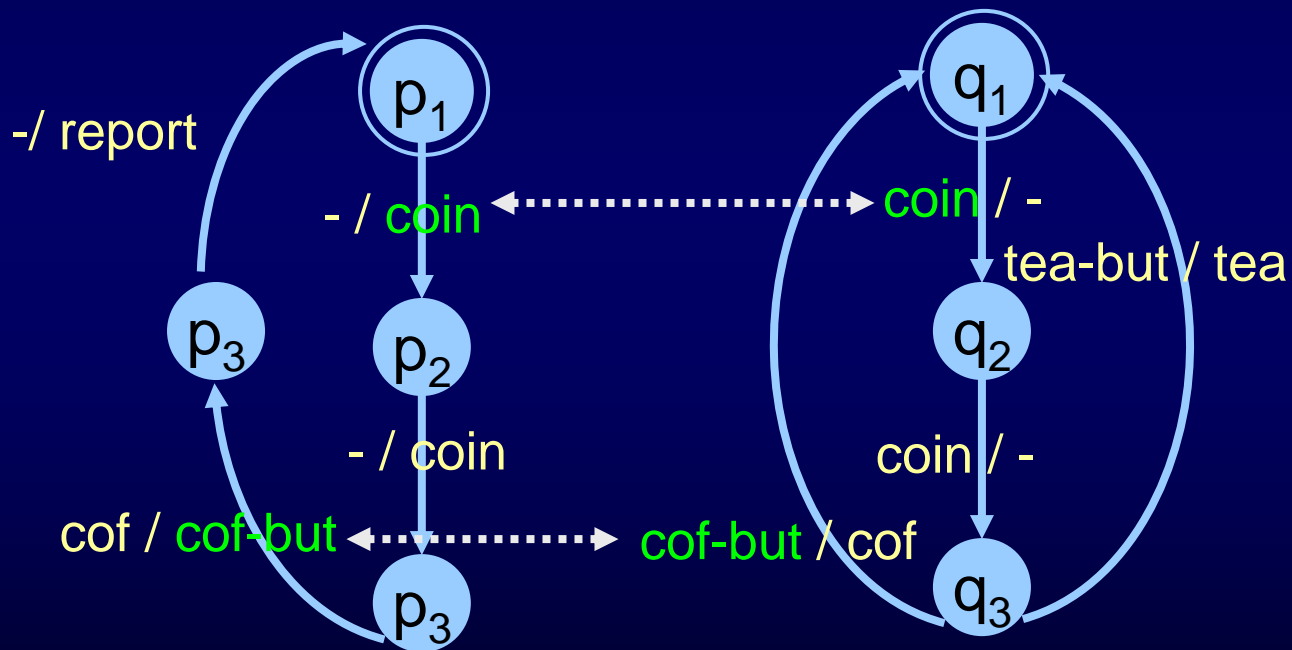
- n parallel FSMs or EFSMs
- Each with k states
- In parallel they have k^n states
- **EXPONENTIAL!**
 - $10^2 = 100$
 - $10^3 = 1000$
 - $10^4 = 10000$
 - $10^{10} = 10000000000$



Synchronous Parallel Composition

Handshake on **complementary** actions

e.g., one "sending" with another "receiving"

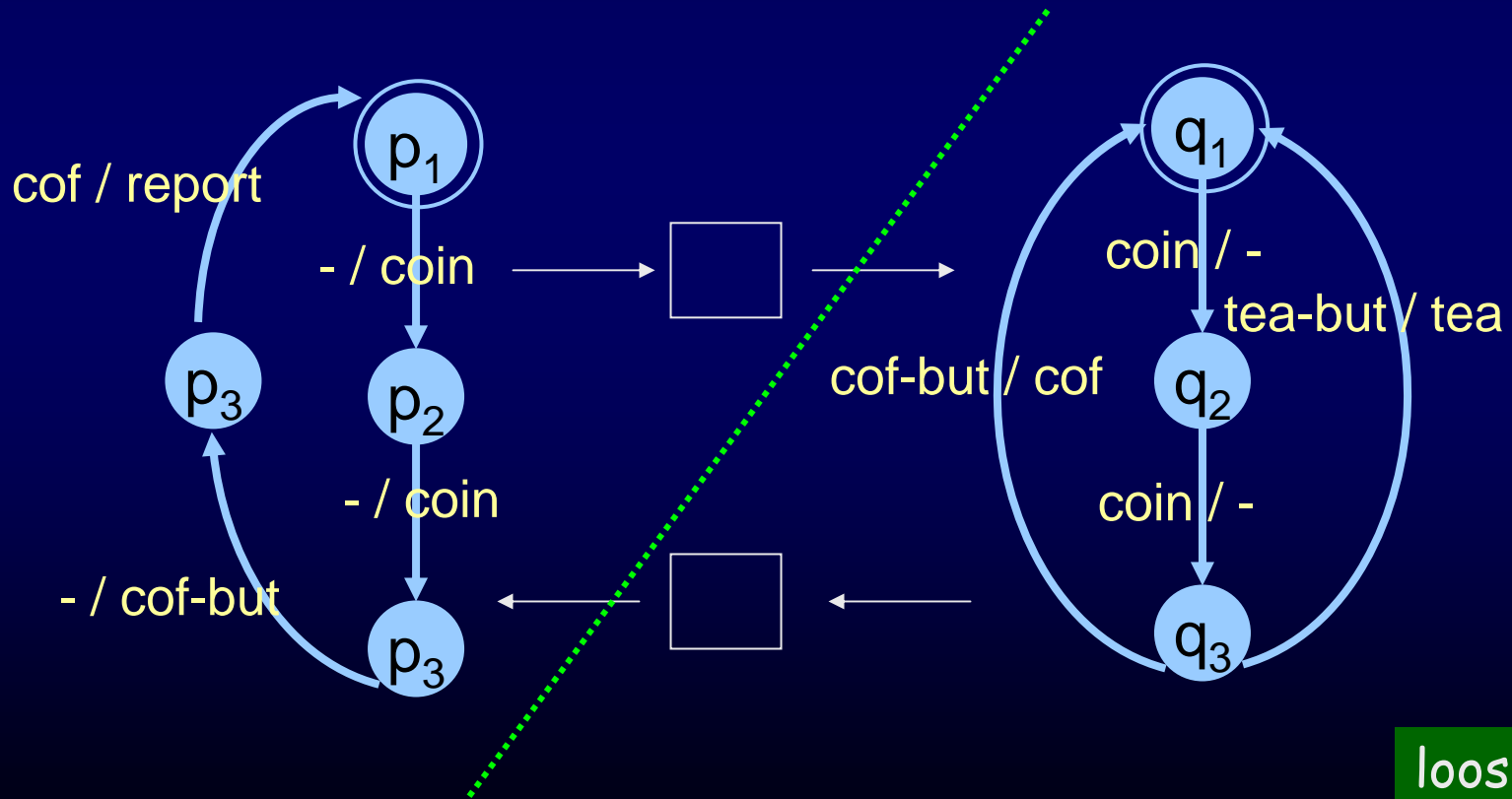


strict!

Asynchronous Parallel Composition

Single output variable per FSM holds last "written" output

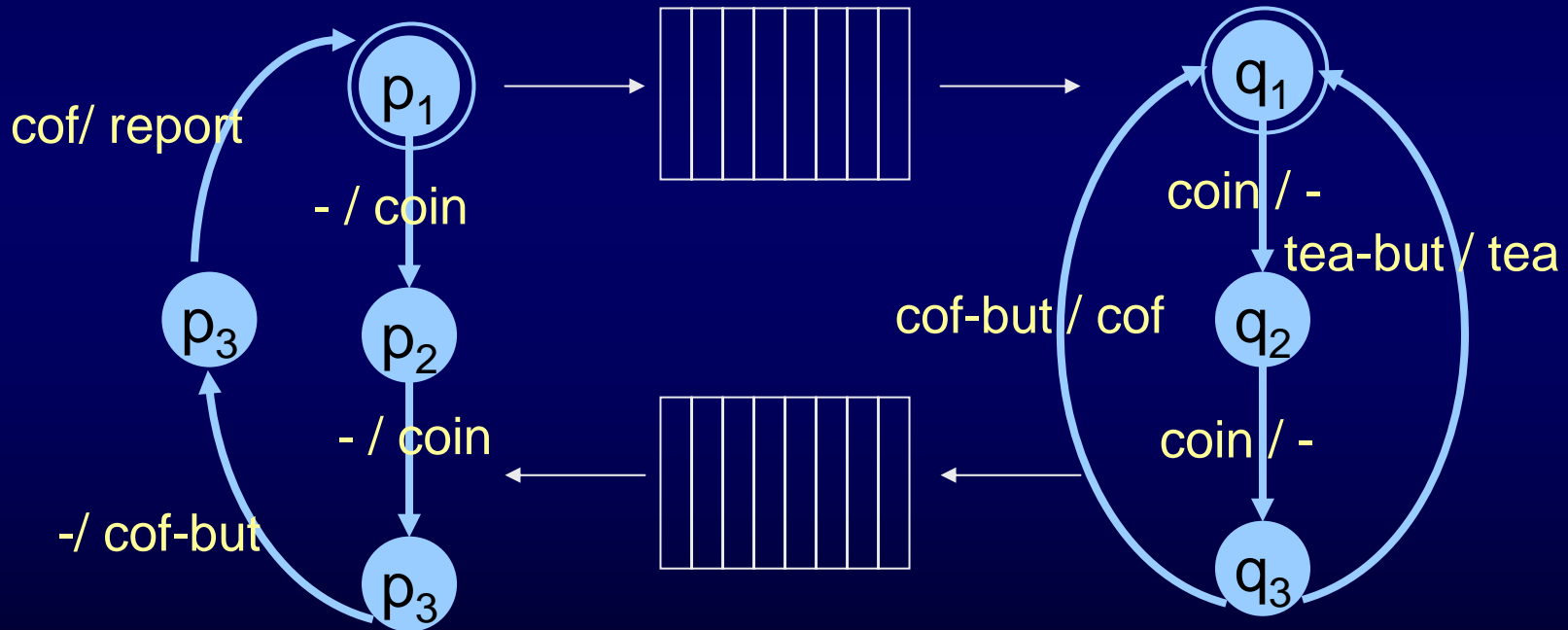
no handshaking any more!



Queued Parallel Composition

Output is queued in (un)bounded queue

The queue may be per process (component), per action, or explicitly defined



System state: a snapshot of all (E)FSMs and queues

looser ..

Blackboard exercise: Bank-box Code



- O
- B
- Y

To open a bank box
the code must contain at least 2 ●

To open a bank box
the code must end with ●●●

To open a bank box
the code must end with ●●●
or with ●●●

To open a bank box
the code must end with a palindrom

!!

- e.g.: ●●●
-
-
-

.....

Palindrome: Word that reads the same forth and back!

Notes

- Palindrome not recognizable by FSM:
infinitely many/long palindromes
- Recognizes bank-box opening sequence:
- If non-deterministic:
→ determinize it → minimize it

Minimized FSM

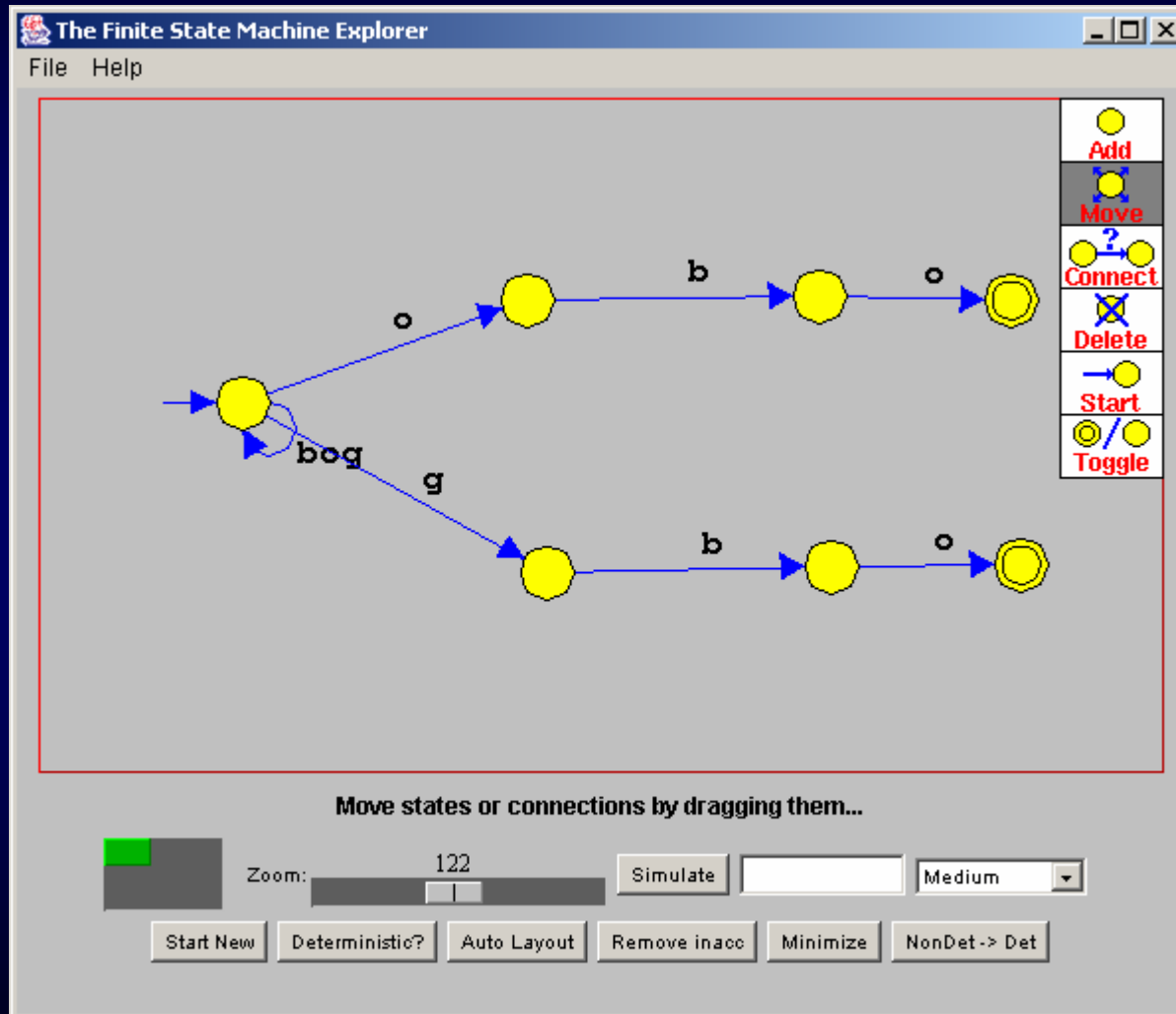
- Two states s and t are (language) **equivalent** iff
 - s and t accepts same language
 - have same traces: $tr(s) = tr(t)$
- Two Machines M_0 and M_1 are **equivalent** iff initial states are equivalent
- A **minimized** (or "reduced") M is one that has no equivalent states
 - for no two states $s, t, s \neq t, s$ equivalent t

Fundamental Results

- Every FSM may be **determinized** accepting the same language (potential explosion in size).
- For each FSM there exists a language-equivalent *minimal deterministic* FSM.
- FSM's are closed under \cap and \cup
- FSM's may be described as regular expressions (and vice versa)

Determinization + Minimization

The Finite State Machine Explorer (<http://www.belgarath.org/java/fsme.html>)



Many other tools for FSM editing, simulation, determinization, minimization, ...
(http://en.wikipedia.org/wiki/List_of_state_machine_CAD_tools)

High-level FSM languages



UML State Machines

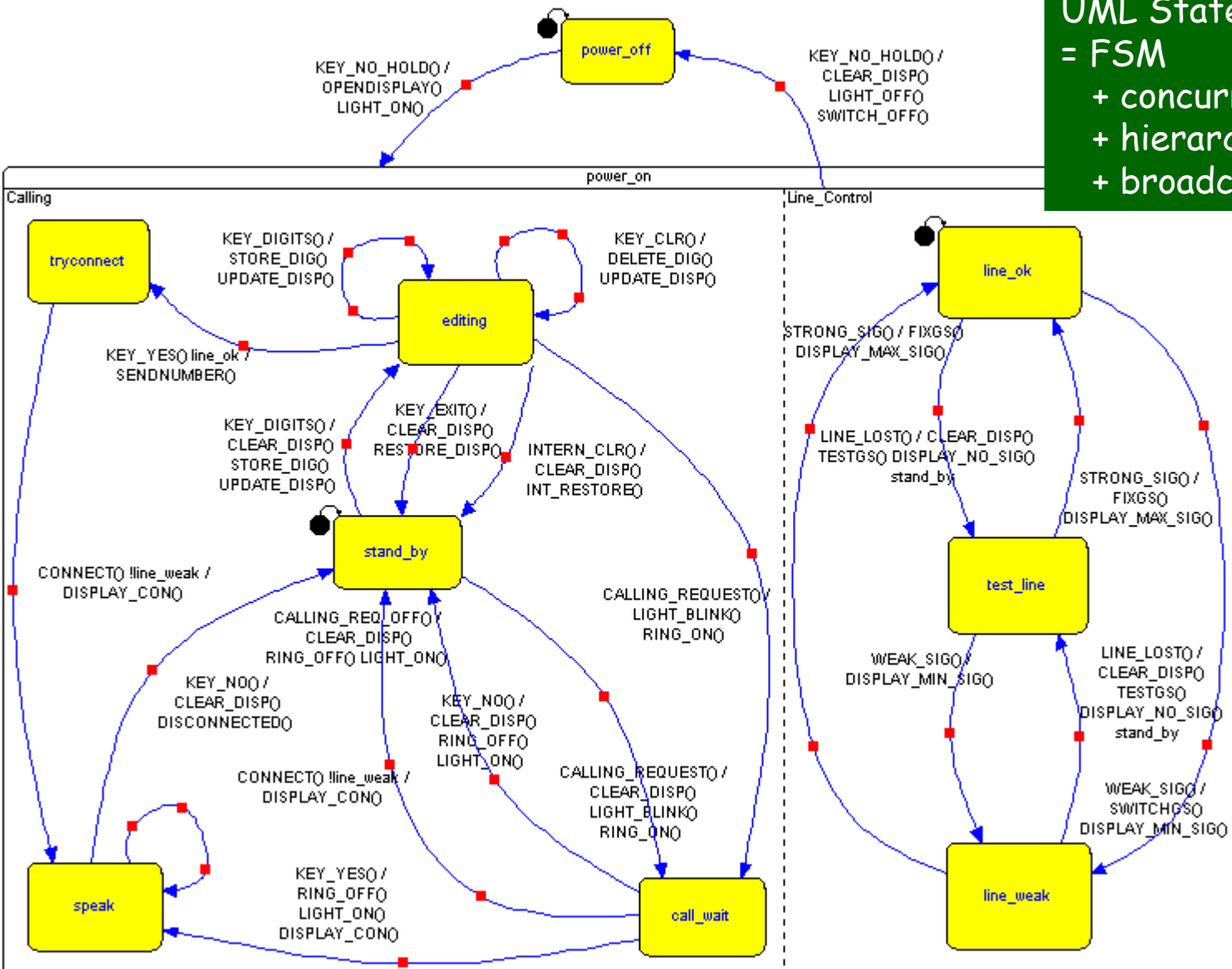
UML State Machine

= FSM

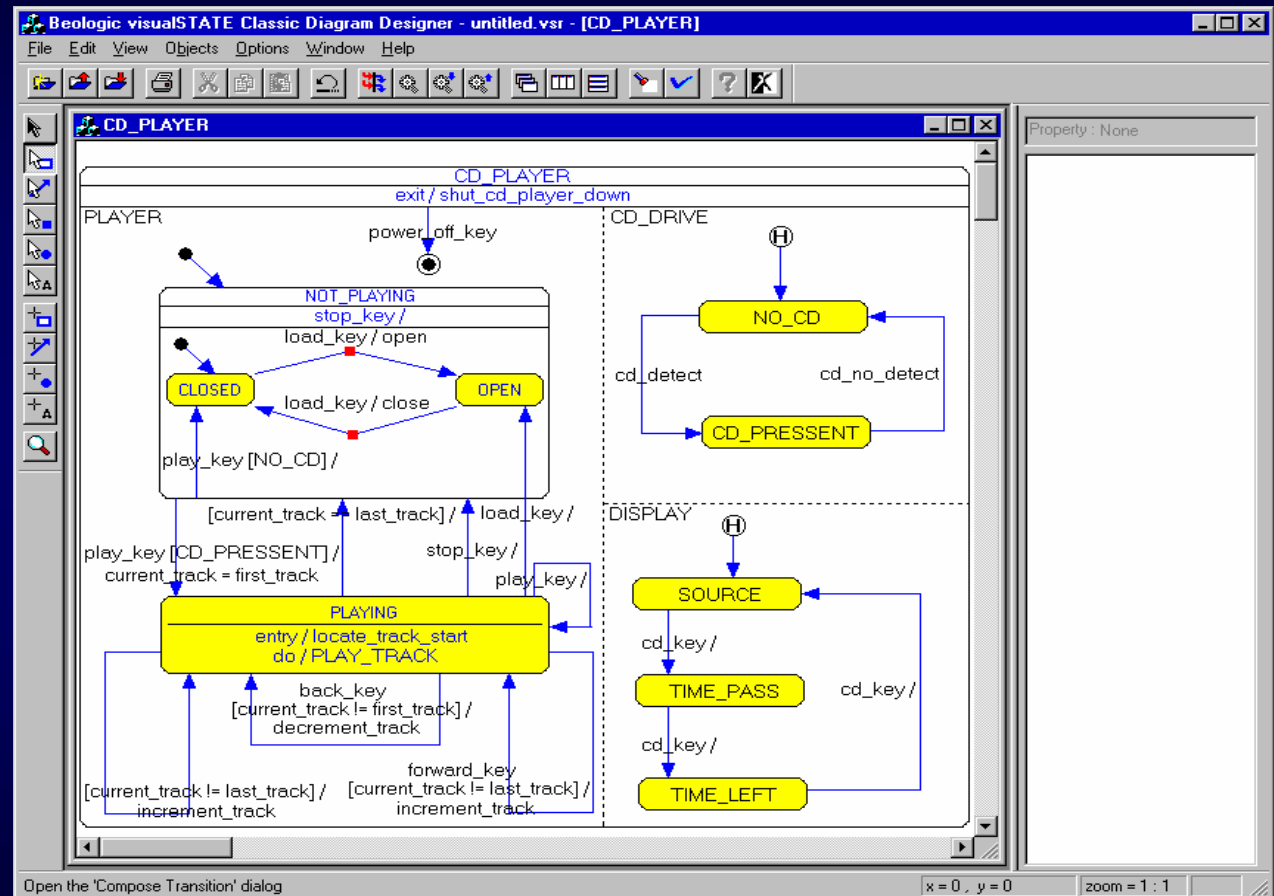
+ concurrency

+ hierarchy

+ broadcast communication



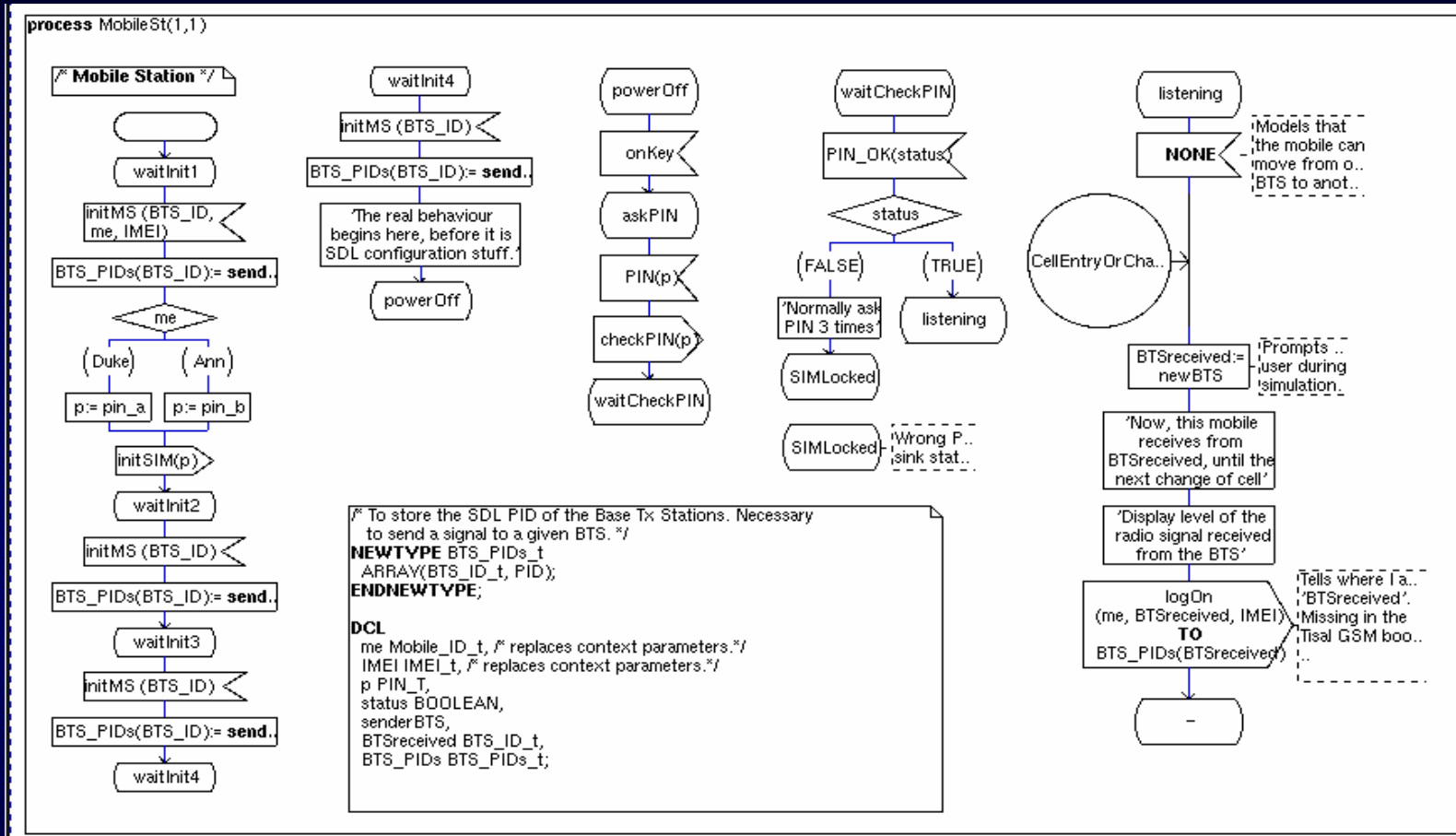
Tool: visualSTATE Designer



- Hierarchical state systems
- Flat state systems
- Multiple and inter-related state machines
- Supports UML notation

SDL

a system is specified as a set of interconnected abstract machines which are **extensions of FSM**



Specification and Description Language (SDL):

- for unambiguous specification and description of the behaviour of reactive and distributed systems
- defined by the ITU-T (Recommendation Z.100.)
- originally focused on telecommunication systems
- current areas of application include process control and real-time applications in general

Esterel

a synchronous programming language for the development of complex reactive systems

Simulation Output

Name	Value	Type
RingBell		
TILT		
GameOver		
Go		
Display	.*	integer
GameNormal.RemainingMe	.*	integer

All Outputs Locals Traps Variables Watch

Simulation Control

Name	Value	Type
Coin		
On_off		
Ready		
Stop		
MS		

All Inputs Sensors

Command Reset Keep Inputs

Current Session 1

Playback Session Reset on Loading

Speed

Dump control

Waveform Start

Configuration file Edit Stop

Coverage Start

Output file Stop

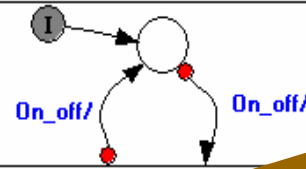
Compact Coverage Files

eflexGameNormal.scg - ReflexGameNormal #0

nat Code Coverage Help

100 Module Abbrev Prior

ReflexGameNormal



MachineON

ingMeasures:integer,
integer
(0, call InitRNDGenerator())

Game Over

sustain GameOver

Coin/

GAME

asures(MEASURE_NUMBER)



PAUSE_LENGTH MS/
Display(?MEAN/MEASURE_NUMBER)

ningMeasures > 0/

the development environment: Esterel Studio

Textual Notations for FSM

In: Promela/SPIN

```
int x;
proctype P(){
  do
    :: x<200 --> x=x+1
  od}

proctype Q(){
  do
    :: x>0 --> x=x-1
  od}

proctype R(){
  do
    :: x==200 --> x=0
  od}

init
{run P(); run Q(); run
R()}
```

In: FSP/LTSA

```
SERVERv2 = (accept.request
            ->service>accept.reply->SERVERv2).
CLIENTv2 = (call.request
            ->call.reply->continue->CLIENTv2).

||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)
                    /{call/accept}.
```

FSP: Finite State Processes

LTSA: Labelled Transition System Analyser

Promela: the input language of tool SPIN

```

SPIN CONTROL 3.1.3 -- 16 March 1998 -- File: p123
File.. Edit.. Run.. Help SPIN DESIGN VERIFICATION Line#: 18 Find:
mtype = { msg0, msg1, ack0, ack1 };
chan sender = [1] of { byte };
chan receiver = [1] of { byte };

proctype Sender()
{
  byte any;
  again:
  do
    :: receiver!msg1;
    if
      :: sender?ack1 -> break
      :: sender?any /* lost */
      :: timeout /* retransmit */
    fi
  od;
  do
    :: receiver!msg0;
    if
      :: sender?ack0 -> break
      :: sender?any /* lost */
      :: timeout /* retransmit */
    fi
  od;
  goto again
}

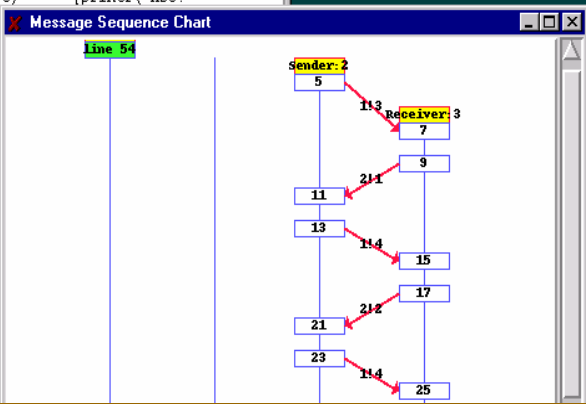
proctype Receiver()
{
  byte any;
  again:
  do
    :: receiver?msg1 -> sender!ack1; break
    :: receiver?msg0 -> sender!ack0
    :: receiver?any /* lost */
  od;
P0:
  do
    :: receiver?msg0 -> sender!ack0; break
  od;
}

```

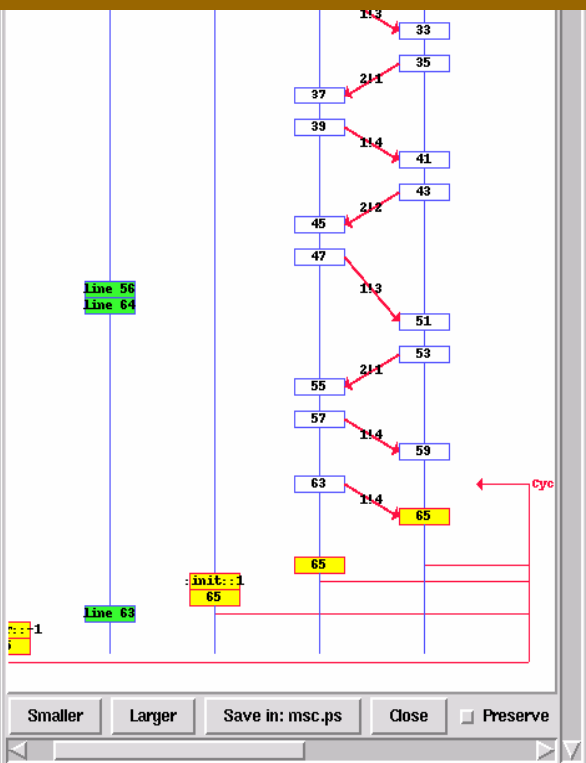
```

) line 41 "pan in" (state 16)
line 23 "pan_in" (state 16)
line 50 "pan_in" (state 4)
ne 63 "never" (state 0) [printf('MSC:
line 63 "pan_in" (sta

```



SPIN, by Gerald Holzmann at AT&T



```

Ghost View
Verification Output
warning: for p.o. reduction to be valid the never claim must be stutter-closed
(never claims generated from LTL formulae are stutter-closed)
pan: acceptance cycle (at depth 59)
pan: wrote pan_in.trail
(Spin Version 3.1.3 -- 16 March 1998)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never-claim +
assertion violations + (if within scope of claim)
acceptance cycles + (fairness disabled)
invalid endstates - (disabled by never-claim)

State-vector 32 byte, depth reached 67, errors: 1
35 states, stored (41 visited)
6 states, matched
47 transitions (= visited+matched)
1 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

2.542 memory usage (Mbyte)

Save in: p123.out Clear Close

```

Ghost View

macdow

Netscape Communicator

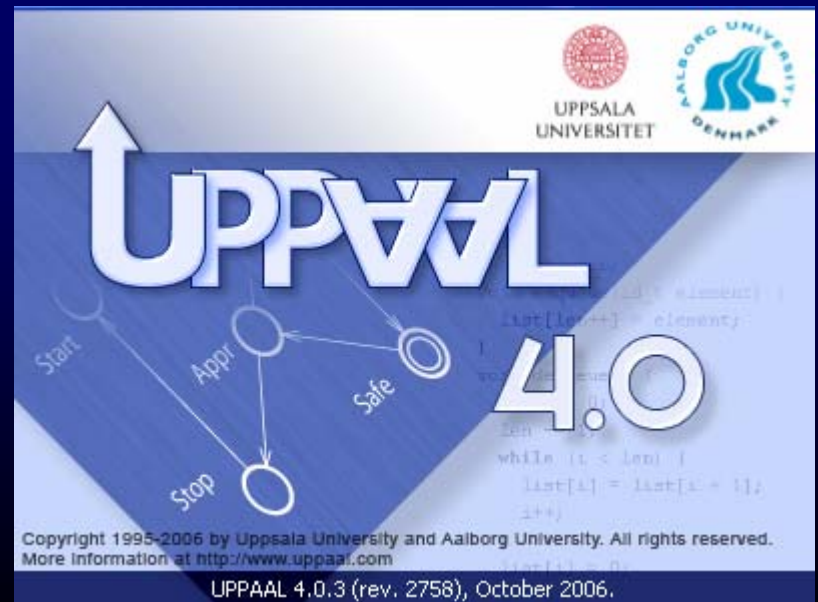
Internet9809...

FskCentOpfl...

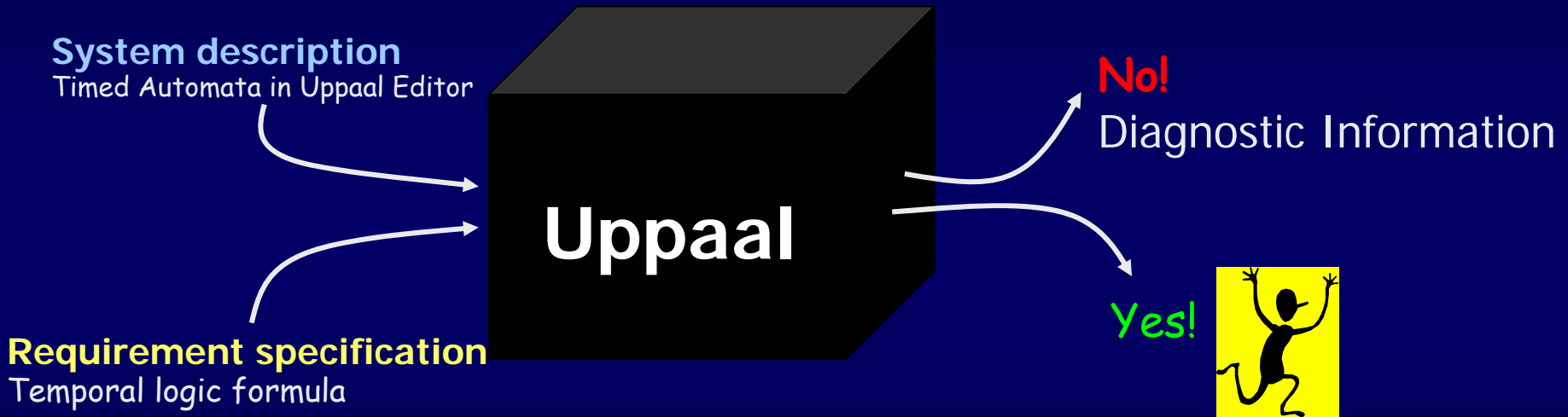
Modelling Untimed Systems using Uppaal

Uppaal

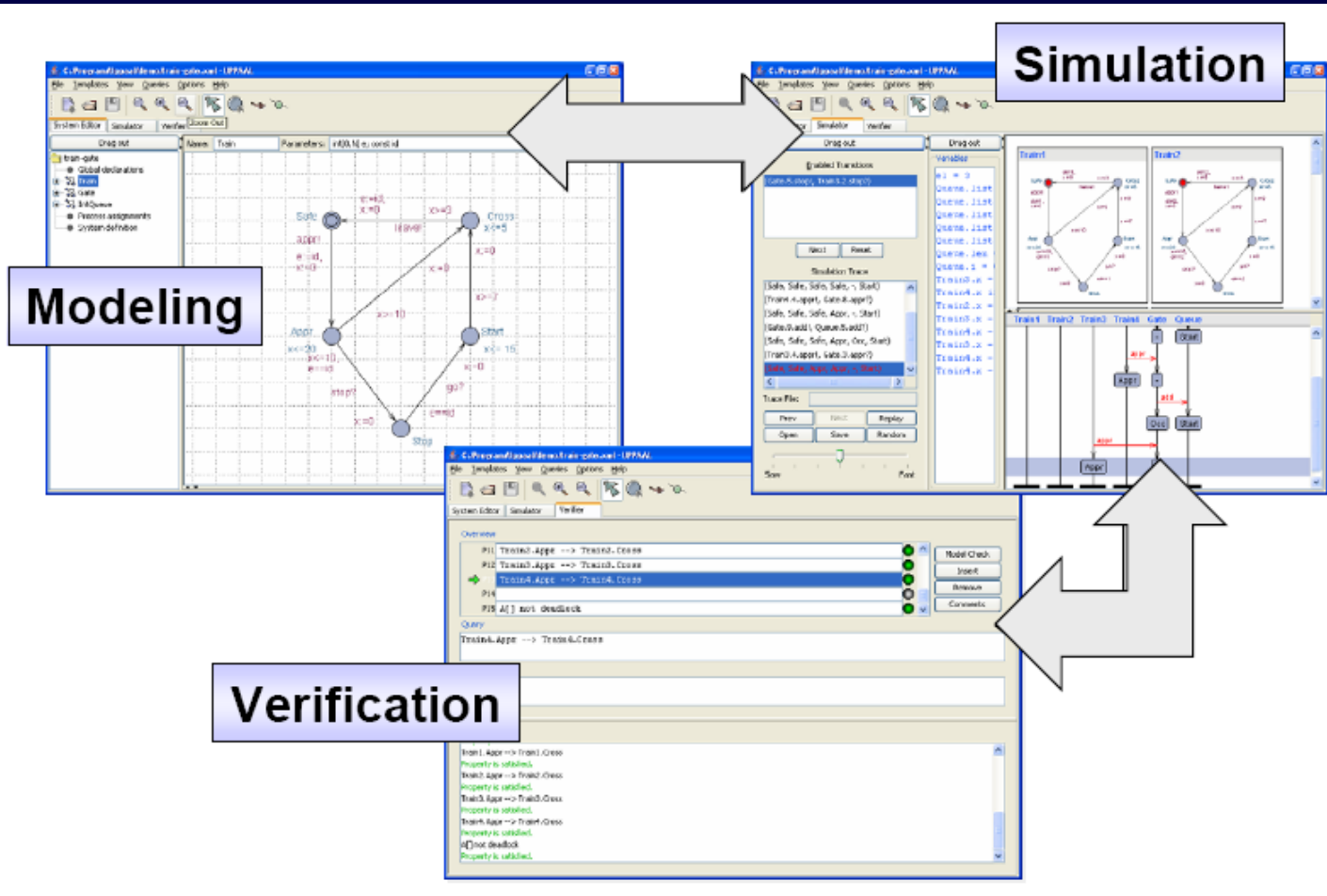
- An integrated tool environment for **modeling**, **simulation** and **verification** of **real-time** systems modelled as networks of timed automata, extended with data types
- However, it is also capable of **untimed** system **modelling**, simulation and verification



Uppaal Verification as a box...



Working Modes of Uppaal



Uppaal Simulator Screenshot

The screenshot displays the Uppaal Simulator interface for a file named `c:\Brian\MMIC\cruise.xml`. The window title is `UPPAAL`. The menu bar includes `File`, `Templates`, `View`, `Queries`, `Options`, and `Help`. The toolbar contains icons for file operations and navigation.

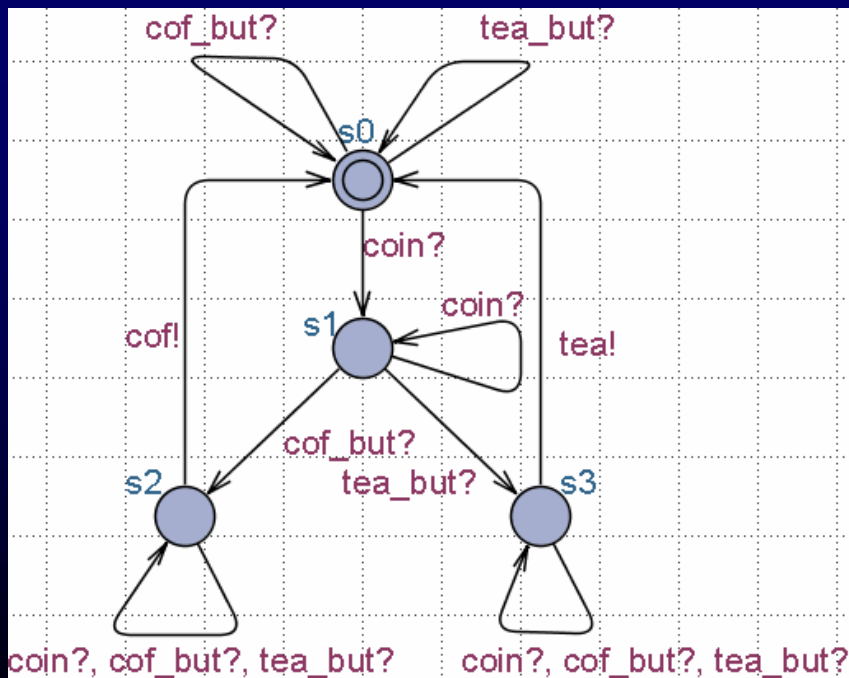
The interface is divided into several panels:

- System Editor:** Contains tabs for `System Editor`, `Simulator`, and `Verifier`.
- Drag out:** A panel for managing transitions, showing `Enabled Transitions` and `Simulation Trace`.
- Variables:** A panel for defining variables, currently showing `eOn = 1`, `brk = 0`, `throttleControl = 0`, and `time = 0`.
- State Machine Diagram:** A state transition diagram for the `cc` component. States include `inactive`, `engine On?`, `clearSpeed!`, `active`, `on?`, `on_requested`, `record Speed!`, and `enable Control!`. Transitions are labeled with events like `engine On?`, `engine Off?`, `disable Control!`, `on?`, `throttle Control = 0`, `accelerator? throttle Control = 0`, `resume?`, and `engine Off? throttle Control = 0`.
- Simulation Trace:** A list of simulation events, including `(-, inactive, disabled, -, -)`, `(u.1.engineOn!, cc.1.engineOn?)`, `(-, -, disabled, -, -)`, `(cc.2.clearSpeed!, sc.7.clearSpeed?)`, `(-, active, disabled, -, -)`, `(u.3.on!, cc.3.on?)`, `(-, on_requested, disabled, -, -)`, `(e.2.speed!, sc.6.speed?)`, and `(-, on_requested, disabled, -, -)`. The last two events are highlighted in blue.
- Trace File:** A field for entering a trace file name.
- Navigation:** Buttons for `Prev`, `Next`, `Replay`, `Open`, `Save`, and `Random`.
- Speed Control:** A slider at the bottom for adjusting simulation speed from `Slow` to `Fast`.
- Component Diagram:** A diagram showing the interaction between components `u`, `cc`, `sc`, `e`, and `ufix`. States like `inactive`, `disabled`, `active`, and `on_requested` are shown with associated signals like `engineOn`, `clearSpeed`, `on`, and `speed`.

FSM in Uppaal

- Basically an Extended FSM (variables, guards, assignments)
- Also may be thought of as an LTS, or IO Automaton
 - actions are either **inputs** or **outputs**
 - internal actions are not explicitly given

LTS can be viewed as a **degradation** of finite state machine (FSM)



Home-Banking?

```
int accountA, accountB; //Shared global variables
```

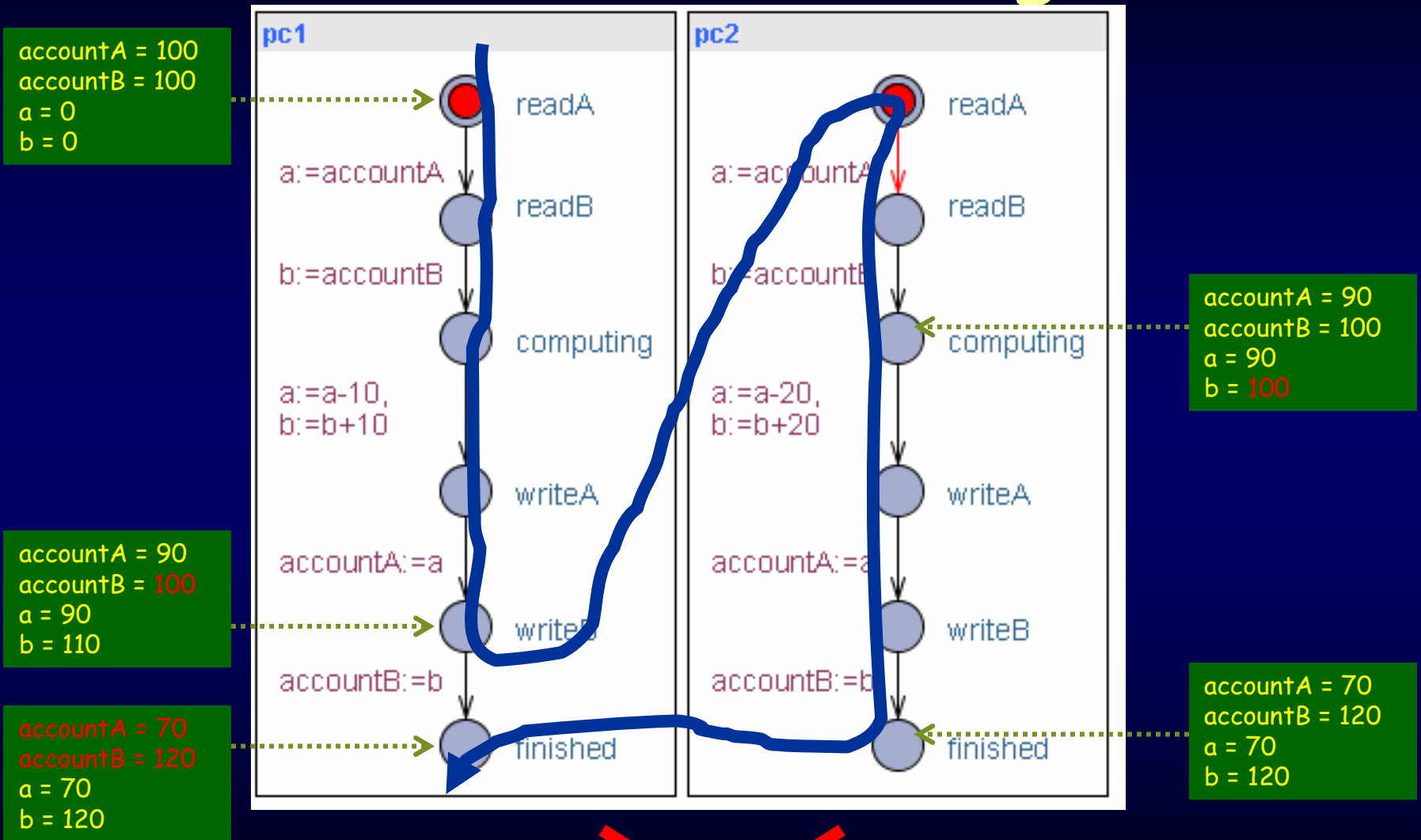
```
//Two concurrent bank costumers
```

```
Thread costumer1 () {  
    int a,b; //local tmp copy  
  
    a=accountA;  
    b=accountB;  
    a=a-10;b=b+10;  
    accountA=a;  
    accountB=b;  
}
```

```
Thread costumer2 () {  
    int a,b;  
  
    a=accountA;  
    b=accountB;  
    a=a-20; b=b+20;  
    accountA=a;  
    accountB=b;  
}
```

- Are the accounts in balance after the transactions?
 - Suppose initially: $\text{accountA} + \text{accountB} = 200$
 - Note that local variables a, b are shared by the two threads

Home-Banking



~~A[] (`pc1.finished` and `pc2.finished`) imply (`accountA+accountB==200`)?~~

Home-Banking: another attempt

```
int accountA, accountB; //Shared global variables
Semaphore A,B;          //Protected by sem A,B
```

```
//Two concurrent bank costumers
```

exclusive access to shared variables via semaphore

```
Thread costumer1 () {
    int a,b; //local tmp copy

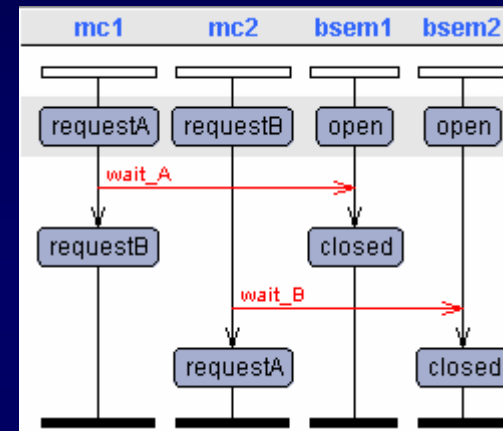
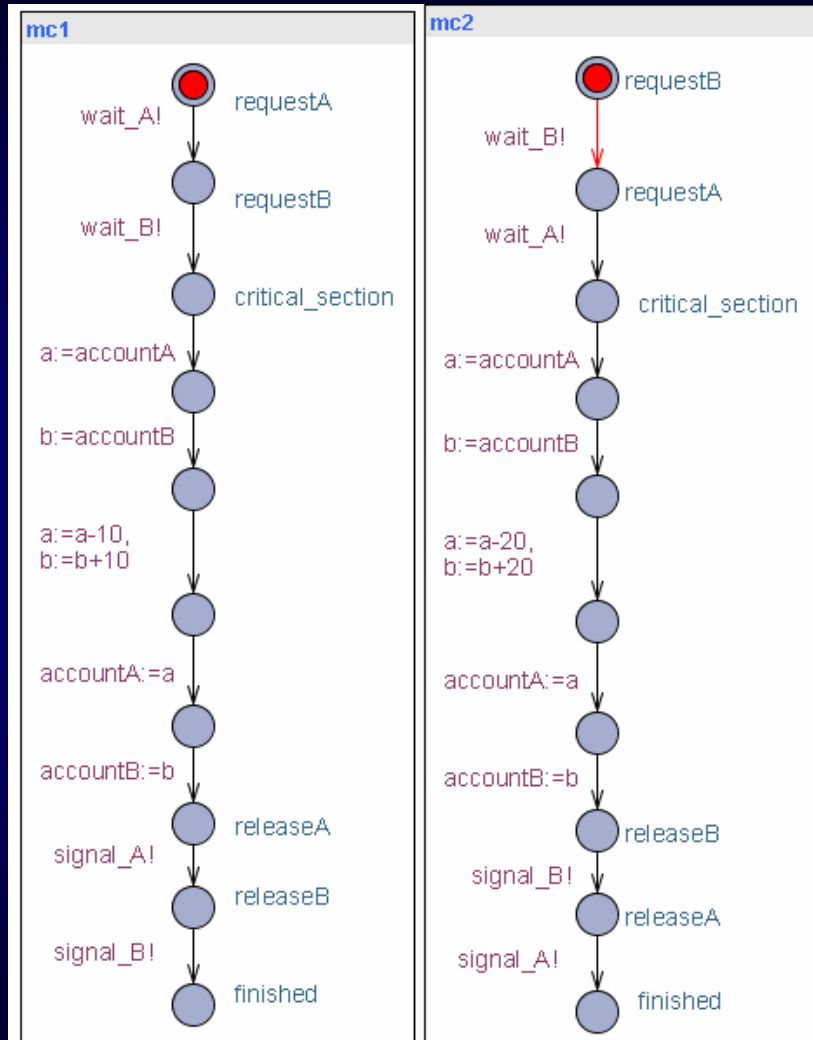
    wait(A);
    wait(B);
    a=accountA;
    b=accountB;
    a=a-10;b=b+10;
    accountA=a;
    accountB=b;
    signal(A);
    signal(B);
}
```

```
Thread costumer2 () {
    int a,b;

    wait(B);
    wait(A);
    a=accountA;
    b=accountB;
    a=a-20; b=b+20;
    accountA=a;
    accountB=b;
    signal(B);
    signal(A);
}
```

- semaphore: a special kind of boolean variables.
- wait(A): if A is true, go to next sentence and set A to false; if A is false, just wait here until A becomes true
- signal(A): set A to true

Home-Banking: another attempt



1. Consistency? (Balance)
2. Race conditions?
3. Deadlock?

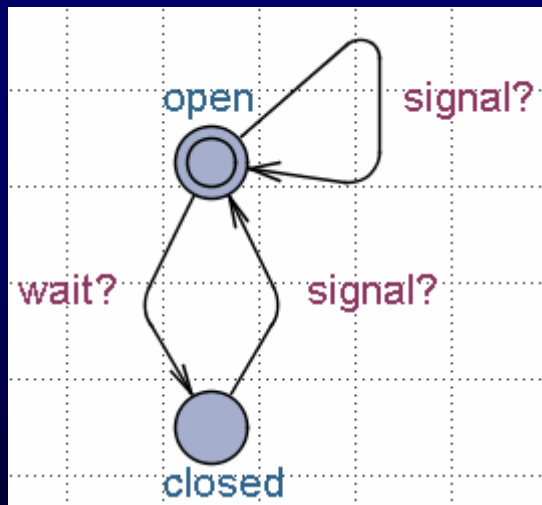
Semaphore Really Works!

1. $A[]$ (mc1.finished and mc2.finished) imply (accountA+accountB==200) ✓
2. $E \langle \rangle$ mc1.critical_section and mc2.critical_section ✓
3. $A[]$ not (mc1.finished and mc2.finished) imply not deadlock ✓

Semaphore FSM Model

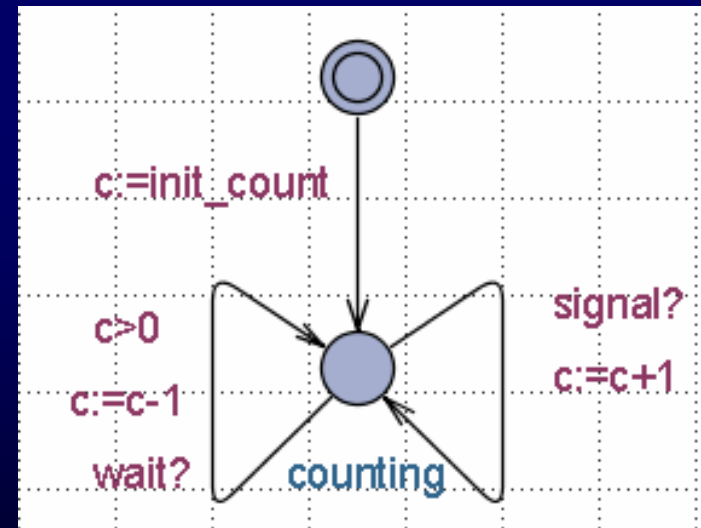
The critical resource can be accessed by only one thread! (exclusive access)

Binary Semaphore



The critical resource can be simultaneously accessed by at most n threads! (restricted shared access)

Counting Semaphore



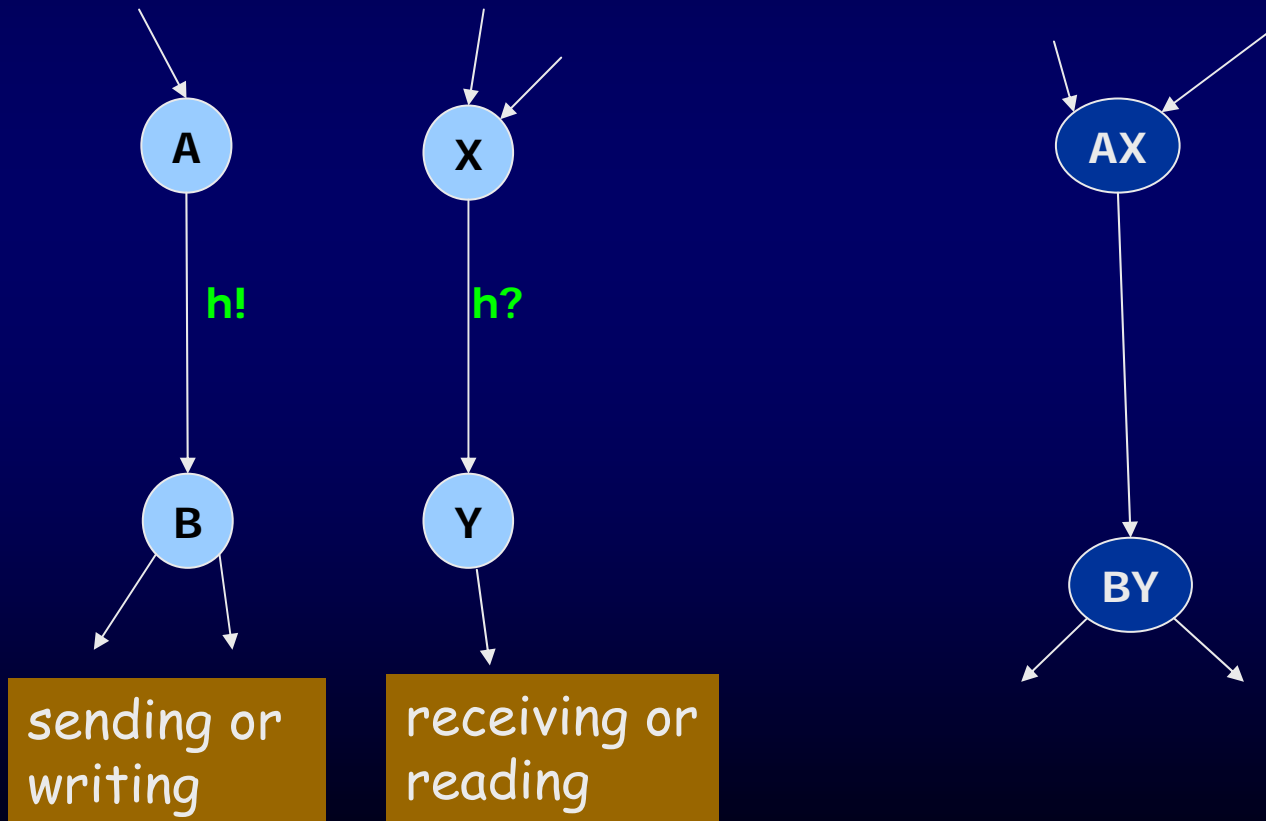
wait: a thread wants to occupy this semaphore
signal: a thread wants to release this semaphore

Composition

IO Automata (2-way synchronization)

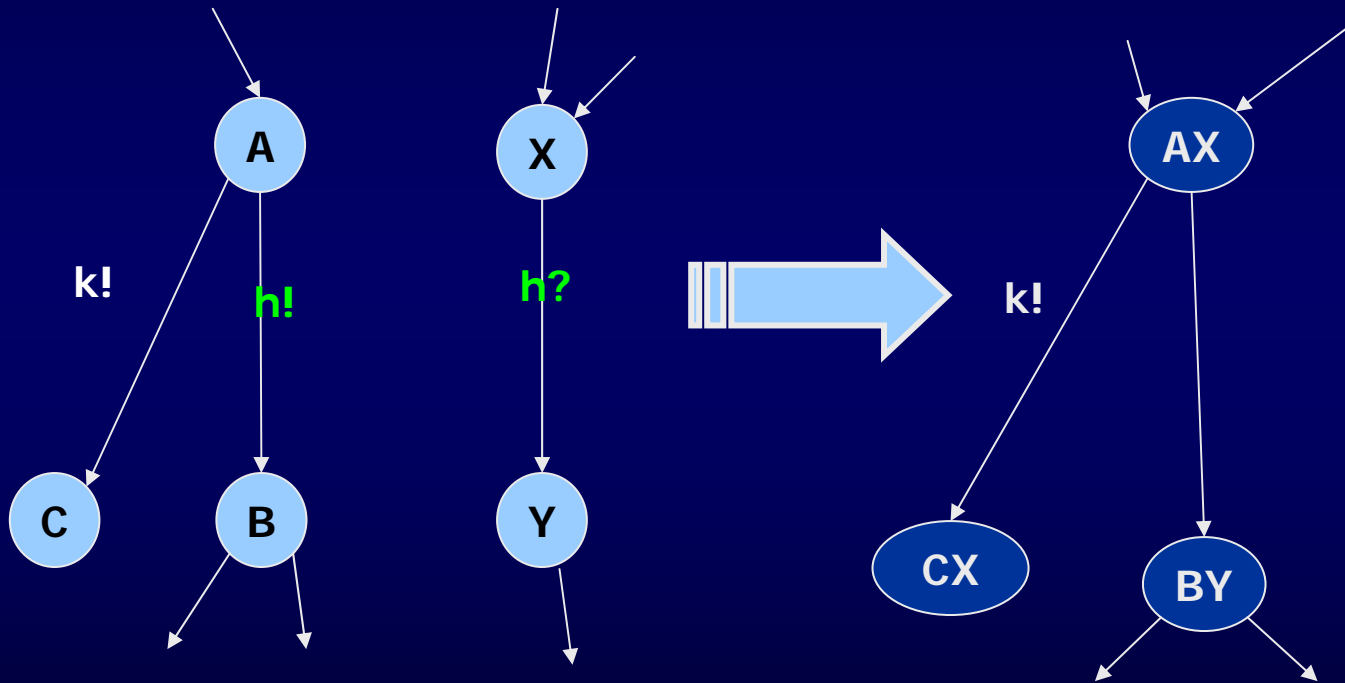
or pairwise synchronization

or "handshaking"



Composition

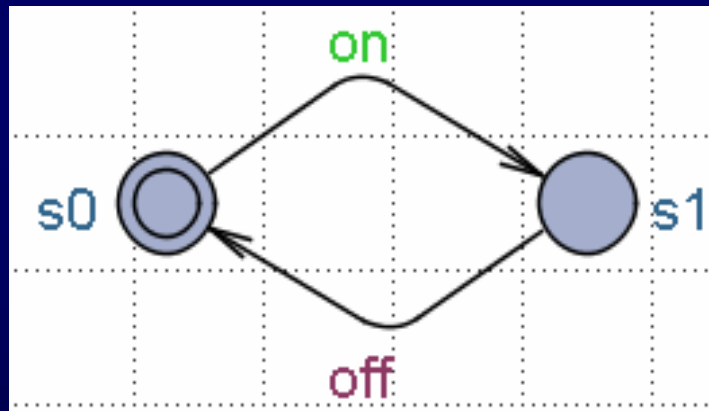
IO Automata



Modelling Processes

- A process is the execution of a sequential program
- modelled as a labelled transition system (LTS)
 - transits from state to state
 - by executing a sequence of *atomic* actions.

a light switch
LTS



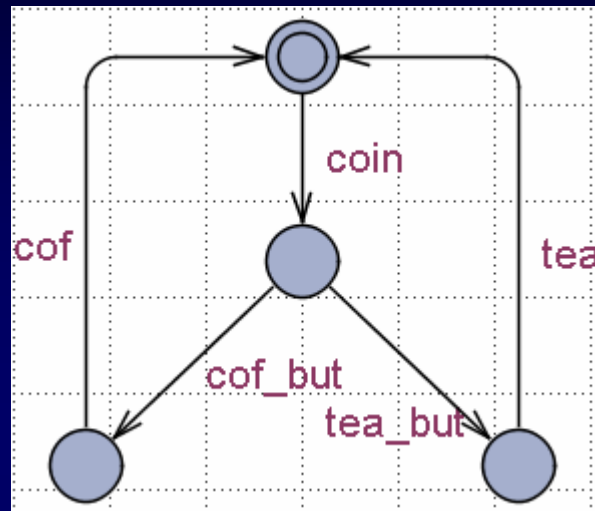
on→off→on→off→on→off→

a sequence of actions

or

a *trace*

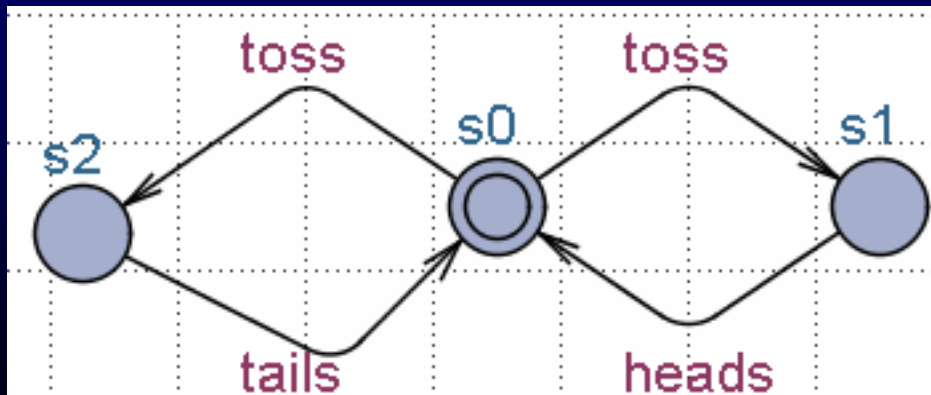
Modelling Choices



- Who or what makes the choice?
- Is there a difference between input and output actions?

Non-deterministic Choice

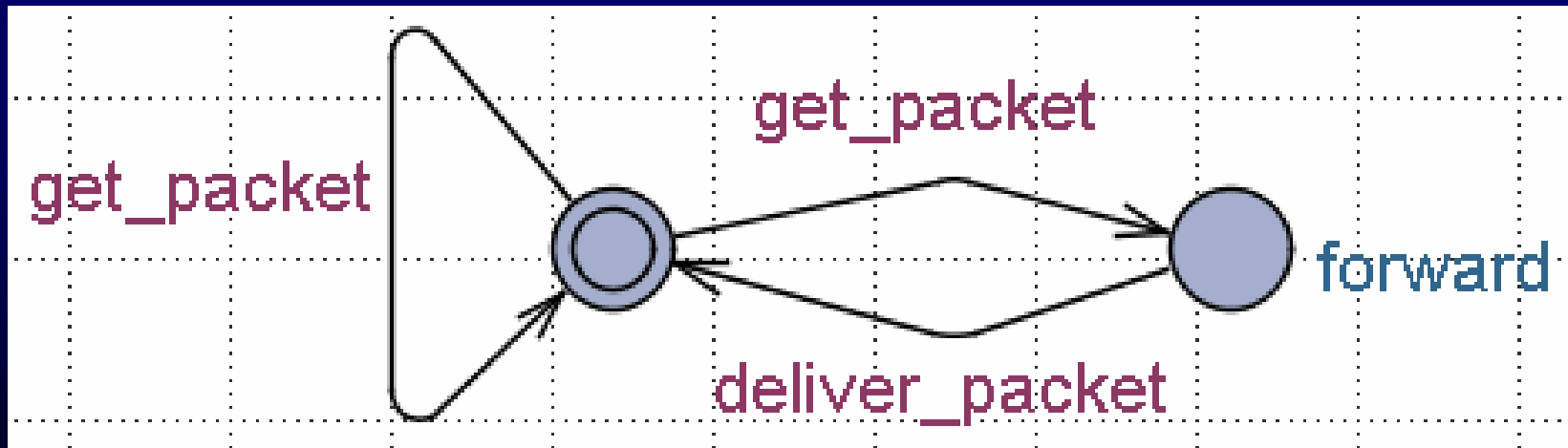
- Tossing a coin
- Possible traces?
 - Both outcomes possible
 - Nothing said about relative frequency
 - If coin is fair, the outcome is 50/50



Non-deterministic Choice modelling failure

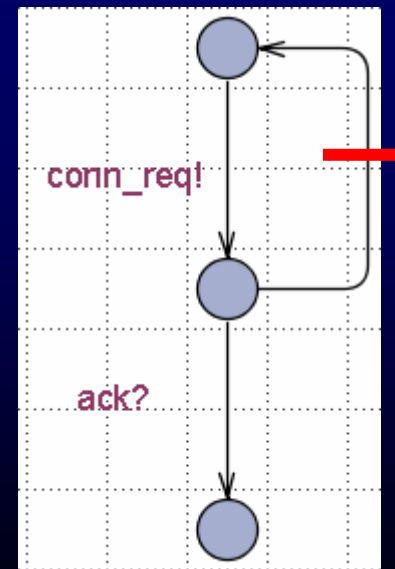
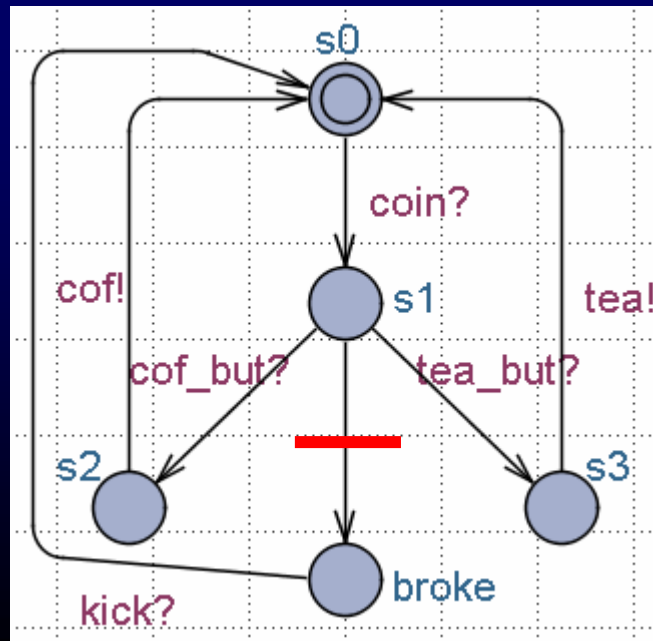
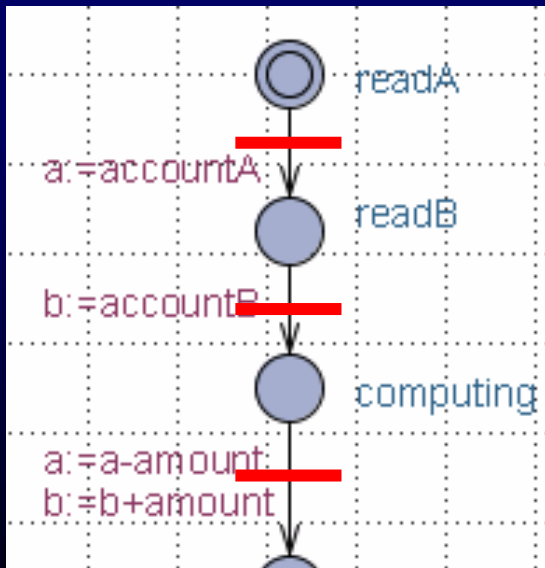
How do we model an **un**reliable communication channel which **accepts** packets, and if a failure occurs **produces no output**, otherwise **delivers** the packet to the receiver?

Use non-determinism...

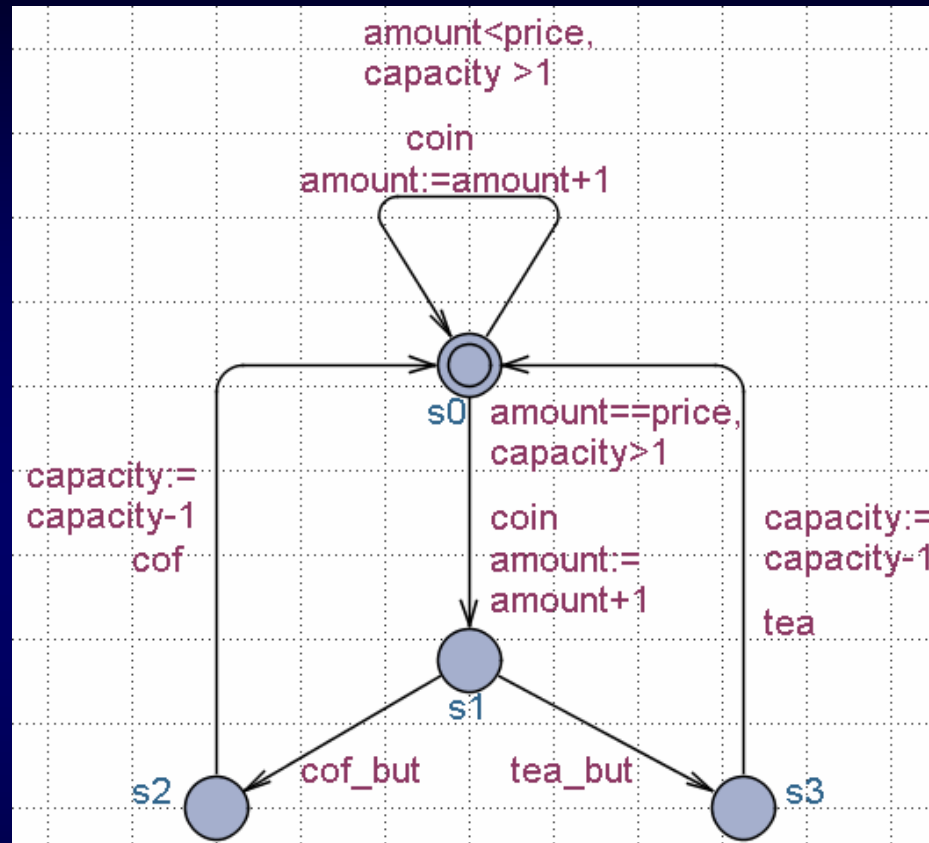


Internal Actions

- Internal actions also called
 - spontaneous actions, or
 - tau-actions
- Internal transitions can be taken on the initiative of **a single machine without** coupling with another one

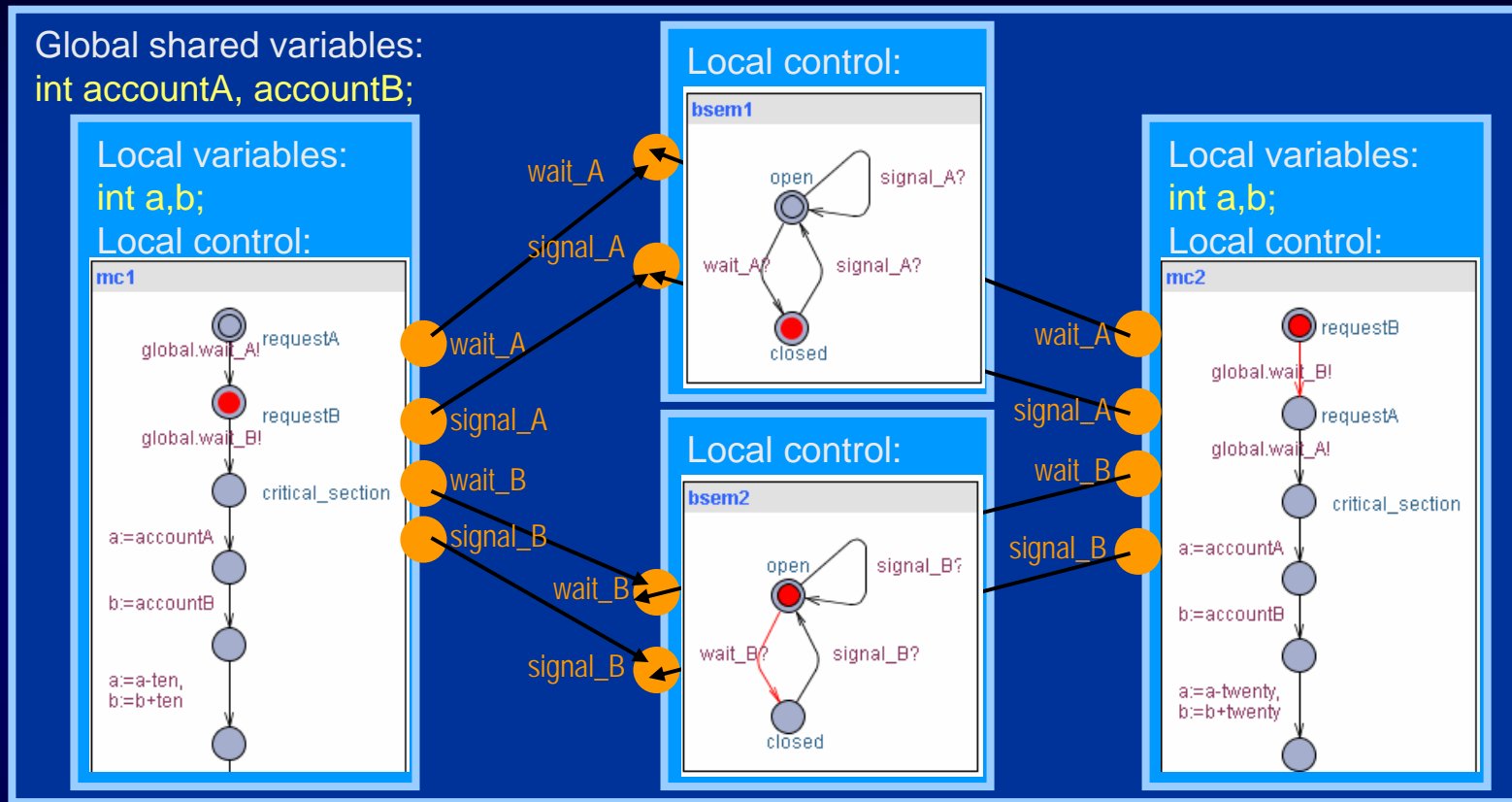


Modelling Extended FSM (EFSM)



- EFSM = FSM + variables + enabling conditions + assignments
- Transition still atomic
- Can be translated into FSM if variables have bounded domains
- State: control location + variables' valuation
- (state, total, capacity), e.g.: (s0, 5, 10)

Uppaal Network of Automata



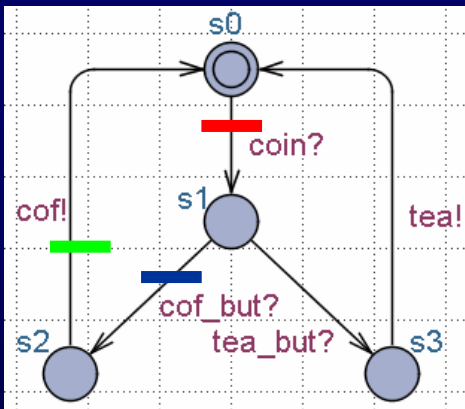
• **system state** = snapshot of (all machines' control locations + local variables + global variables)

e.g.: mc1.control=requestB, mc1.a=0, mc1.b=0,
 mc2.control=requestB, mc2.a=0, mc2.b=0,
 bsem1.control=closed, bsem2.control=open,
 accountA=100, accountB=100

Process Interaction

- “!” denotes output, “?” denotes input
- Handshake communication
- Two-way

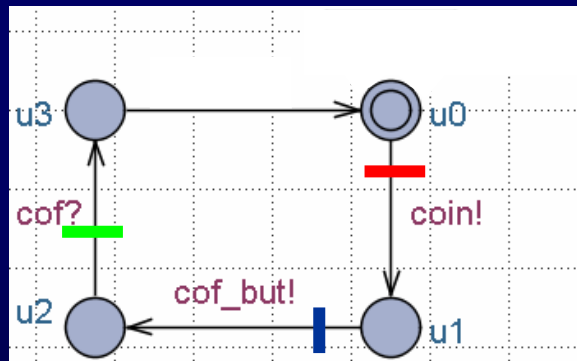
Coffee Machine



4 states



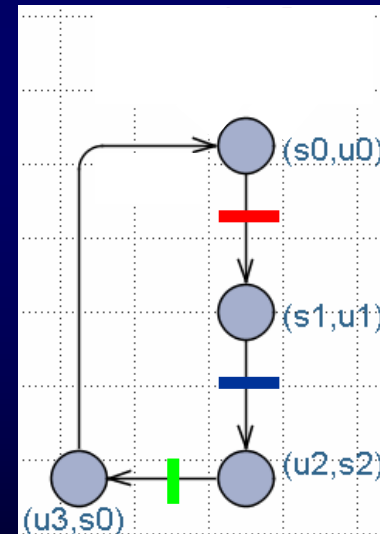
Lecturer



4 states

University =
Coffee Machine || Lecturer

=



synchronization results in internal actions

LTS?
How many states?
Traces?

4 states:

(interactions constrain overall behavior)

Broadcasts

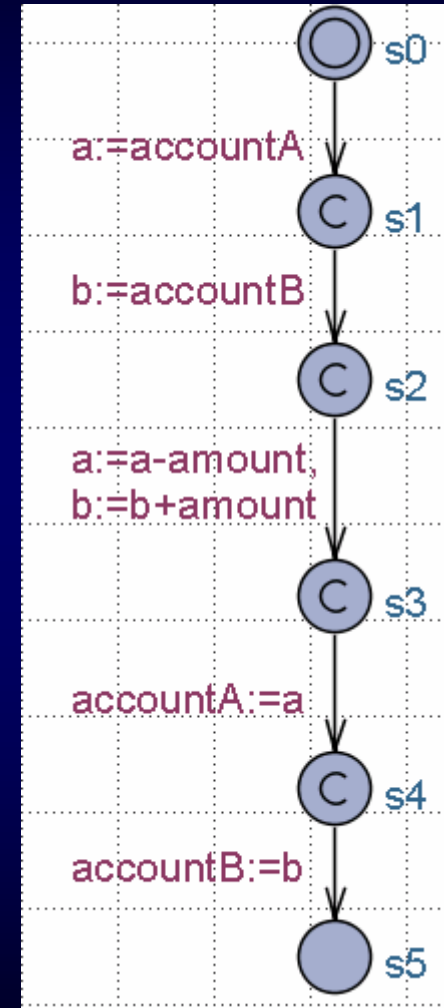
```
chan coin, cof, cofBut;  
broadcast chan join;
```



- the sending party: one automaton outputs **join!**
- the receiving party: several automata accept **join!**,
 - each of them makes a move upon receiving **join!**,
 - ie. every automaton with enabled "**join?**" transition moves in one step
- the number of recipients may be 0 (one "speaker", but no "audience")

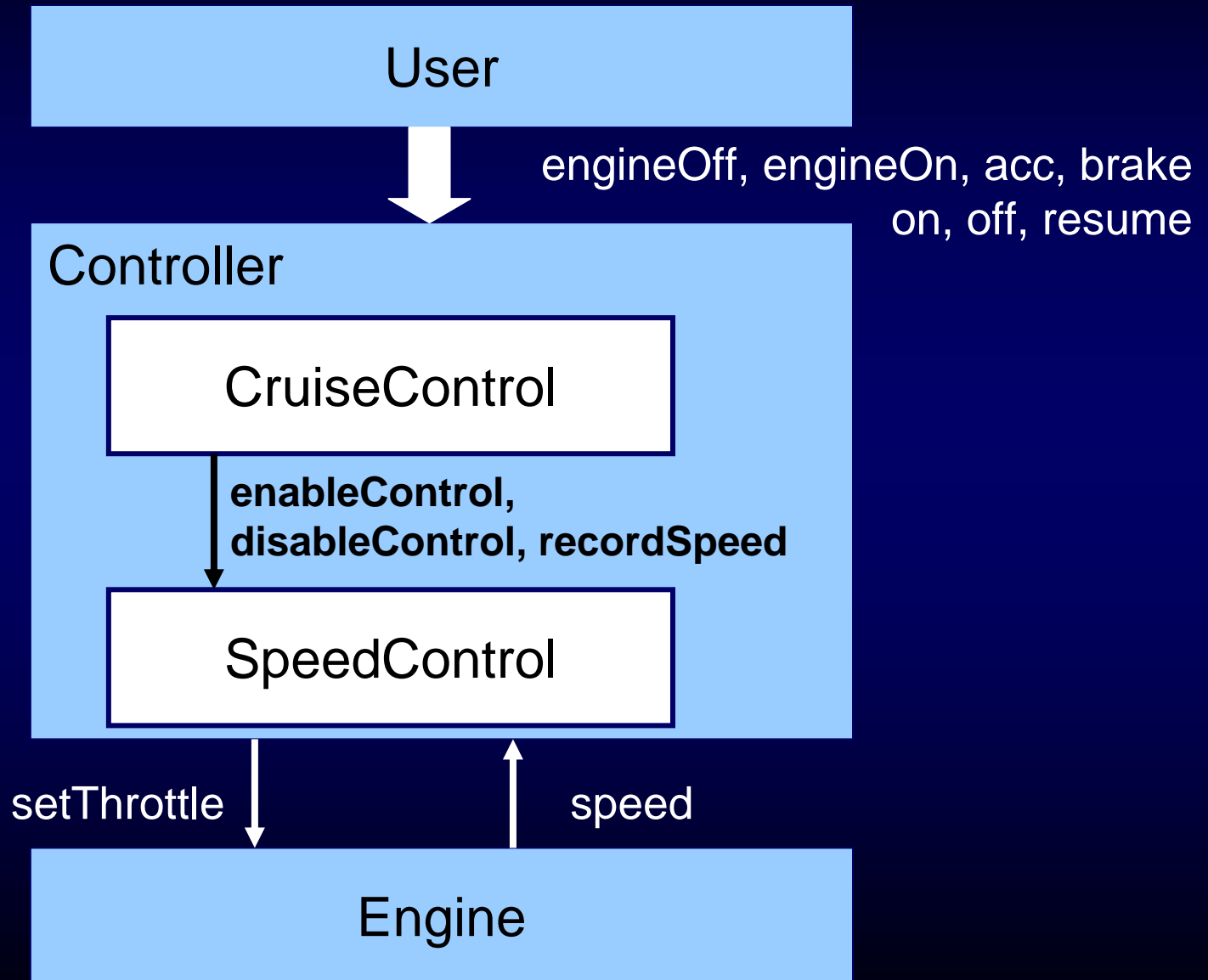
Committed Locations

- Locations marked "C"
 - *No delay* in committed location
 - Next transition must involve one of those automata in *committed locations*
- Handy to model atomic sequences
 - An "input/output"-style transition of **Mealy** machine can be modelled by 2 atomic actions "input?" and "output!", which are connected by a **committed location**
- The use of committed locations significantly reduces the state space of a model, thus allows for more efficient analysis and verification



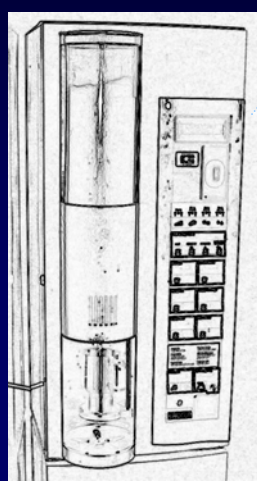
s0 to s5 executed atomically
they will not be interrupted

The Cruise Controller



Timed Automata

Real-time Systems



Plant

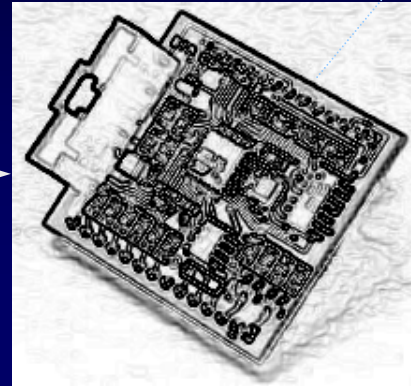
Continuous

The "environment"

sensors

actuators

We are interested in this one!



Controller Program

Discrete

Eg.:

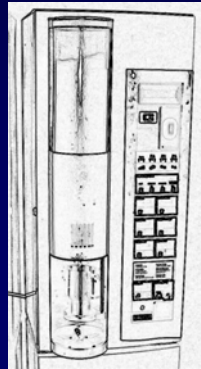
- Realtime Protocols
- Pump Control
- Air Bags
- Robots
- Cruise Control
- ABS
- CD Players
- Production Lines

Real Time System

A system where correctness not only depends on the logical order of events but also on their **timing!!**

Real-time System Modelling

Plant
Continuous



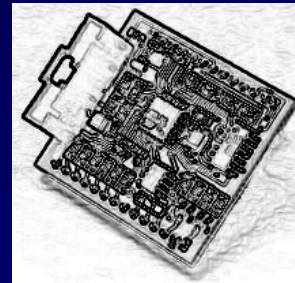
sensors



actuators

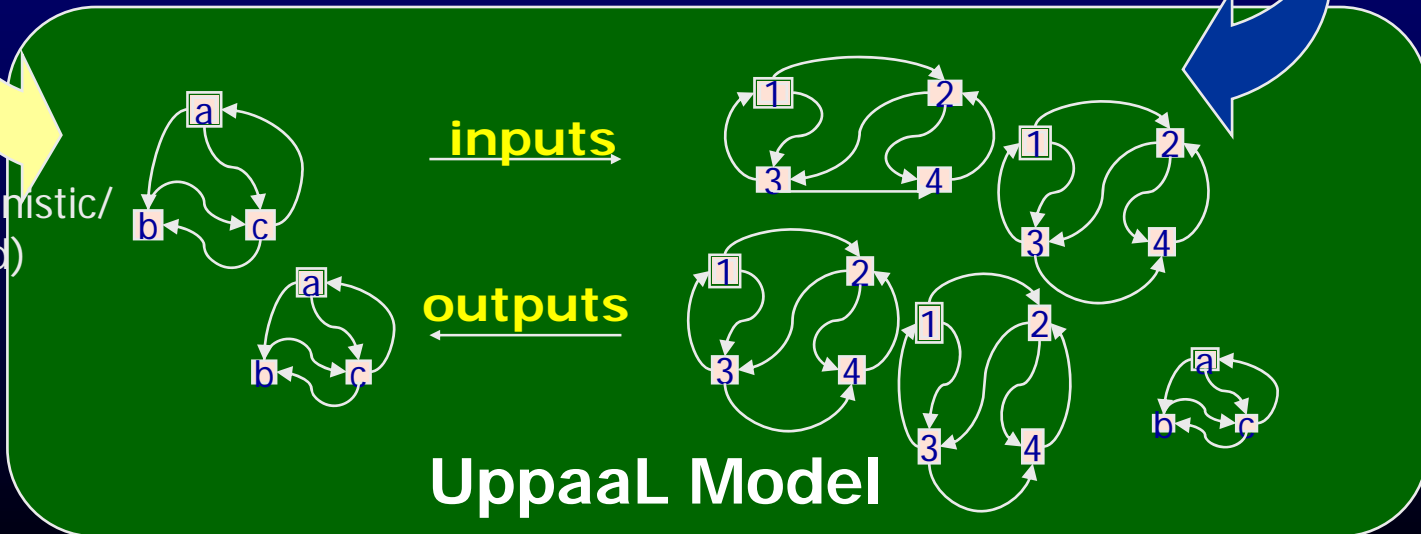


Controller Program
Discrete

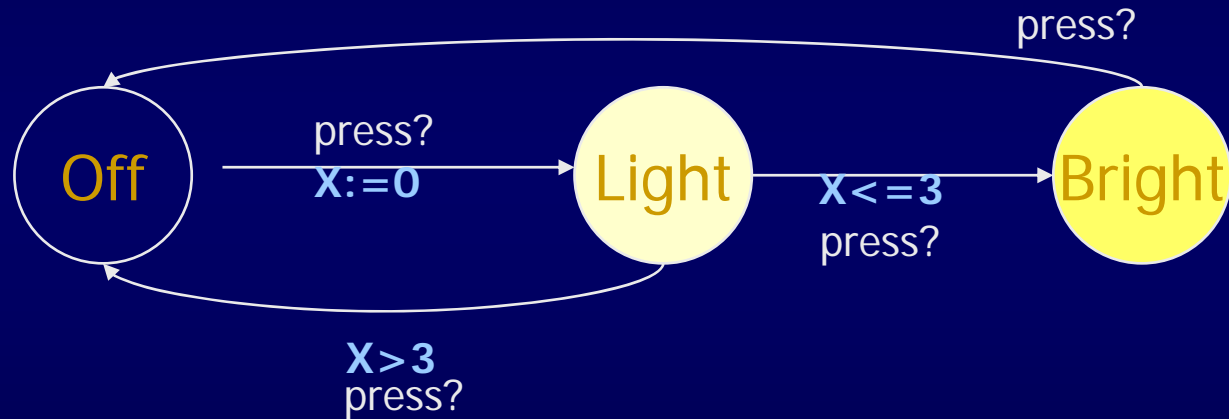


Model of Tasks
(user supplied
/automatic?)

Model of Environment
(non-deterministic/
User-supplied)



An Intelligent Light Control

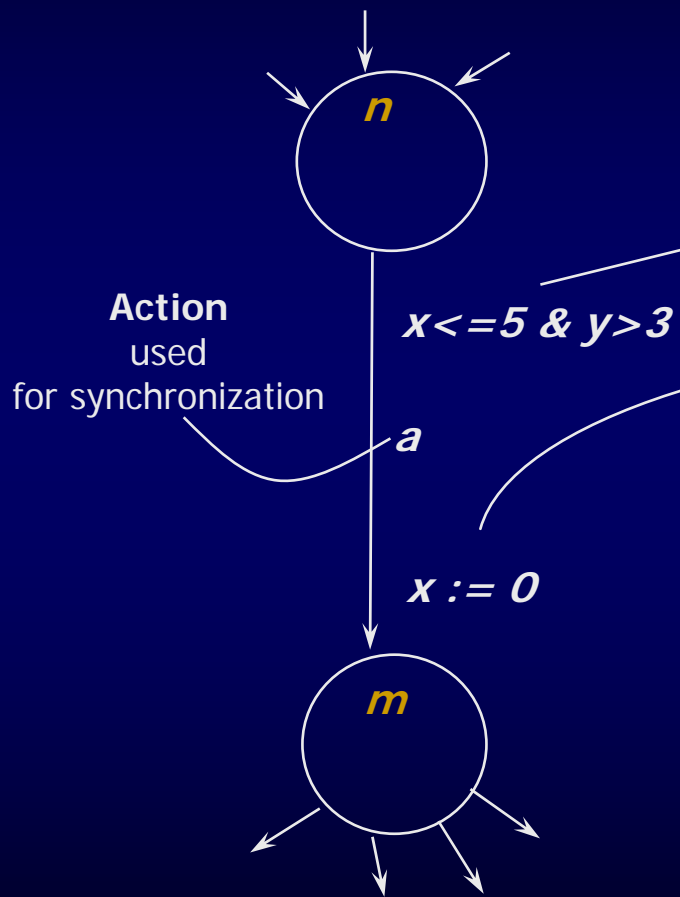


WANT: if "press" is issued twice **quickly** then the **light** will get **brighter**; if "press" is issued twice **slowly** the light is turned **off**.

Solution: Add a real-type variable (a real-valued clock) x

Timed Automata

(Alur & Dill 1990)



Clocks: x, y

Guard

Boolean combination of comp with integer bounds

Reset

Action performed on clocks

State

(*location* , $x=v$, $y=u$) where v, u are in \mathbf{R}

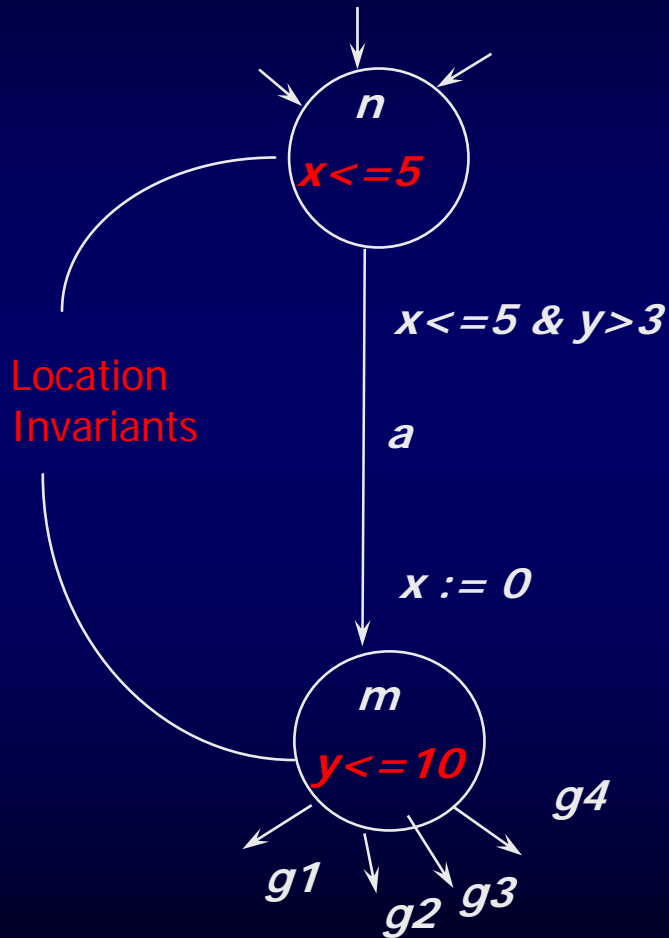
Transitions

(n , $x=2.4$, $y=3.1415$) \xrightarrow{a} (m , $x=0$, $y=3.1415$)

(n , $x=2.4$, $y=3.1415$) $\xrightarrow{e(1.1)}$ (n , $x=3.5$, $y=4.2415$)

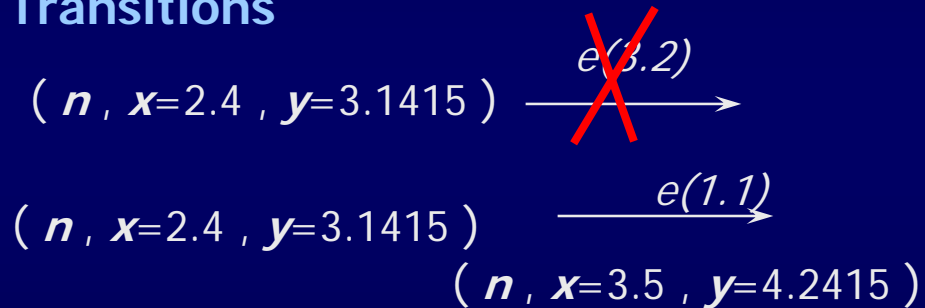
Timed Automata

location invariants



Clocks: x, y

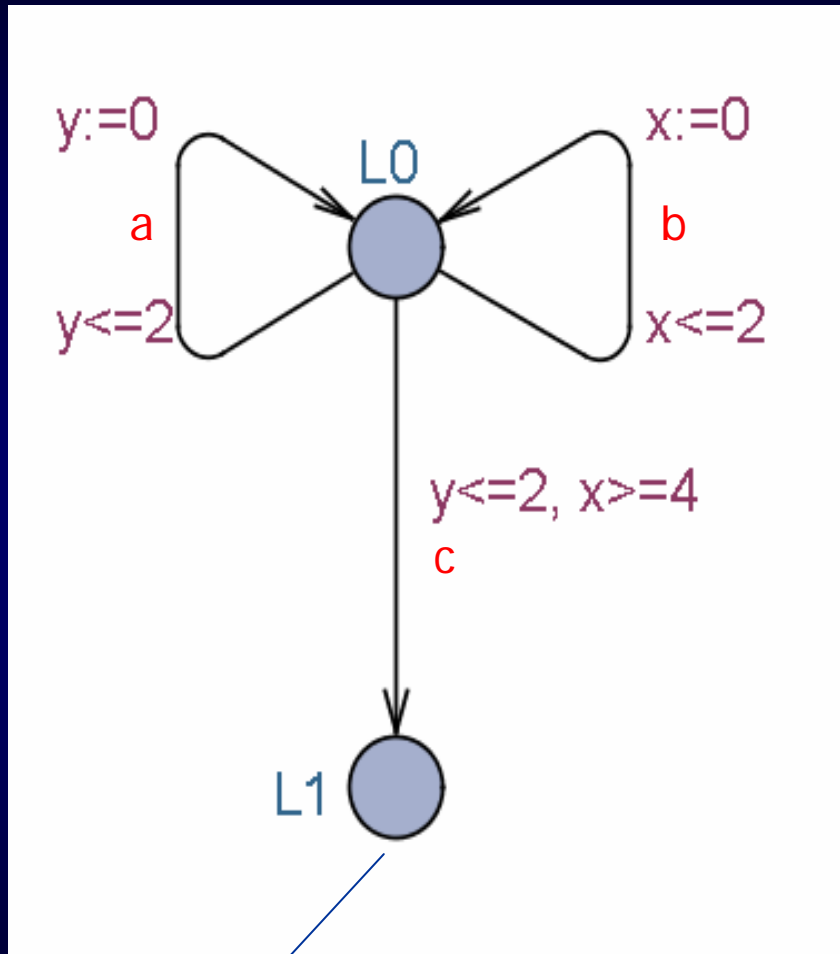
Transitions



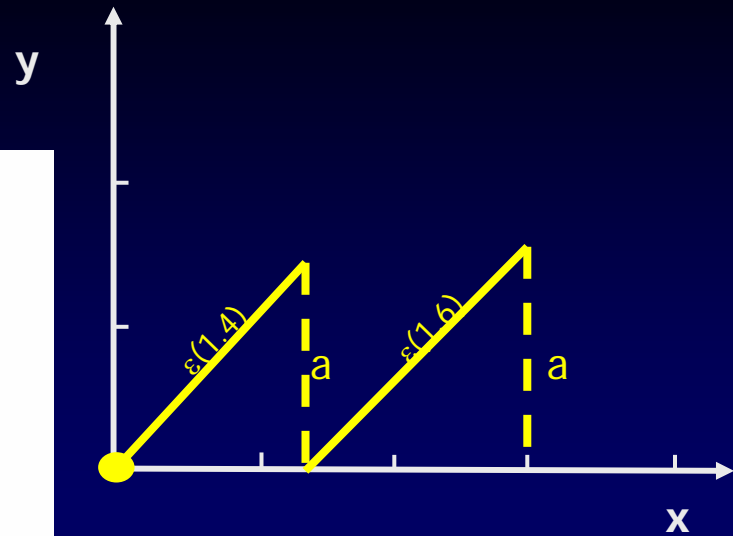
Invariants ensure progress!!

you cannot stay in this location forever;
you must leave before the deadline!

Example



Reachable?



$(L0, x=0, y=0)$

$\rightarrow_{\epsilon(1.4)}$

$(L0, x=1.4, y=1.4)$

\rightarrow_a

$(L0, x=1.4, y=0)$

$\rightarrow_{\epsilon(1.6)}$

$(L0, x=3.0, y=1.6)$

\rightarrow_a

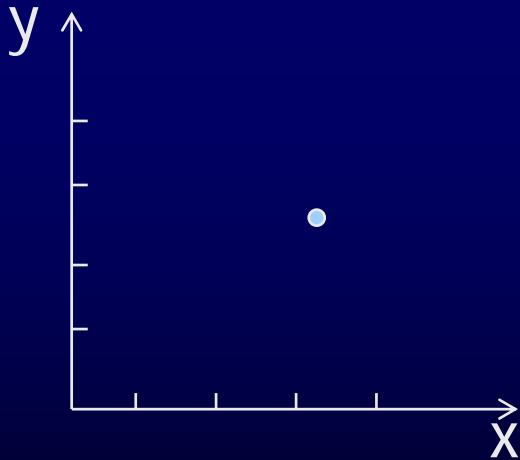
$(L0, x=3.0, y=0)$

Zones

from infinite to finite

a state

$(n, x=3.2, y=2.5)$



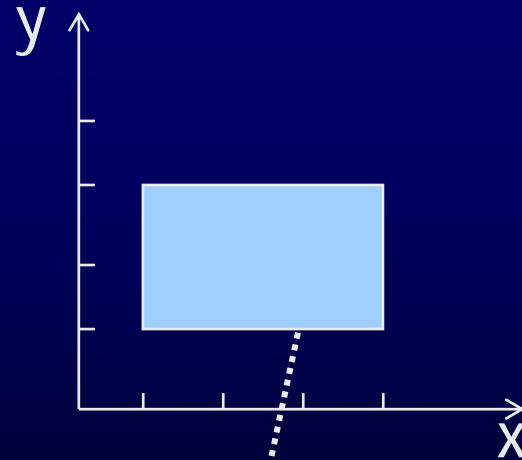
a bunch of concrete states

a symbolic state (set)

$(n, 1 \leq x \leq 4, 1 \leq y \leq 3)$

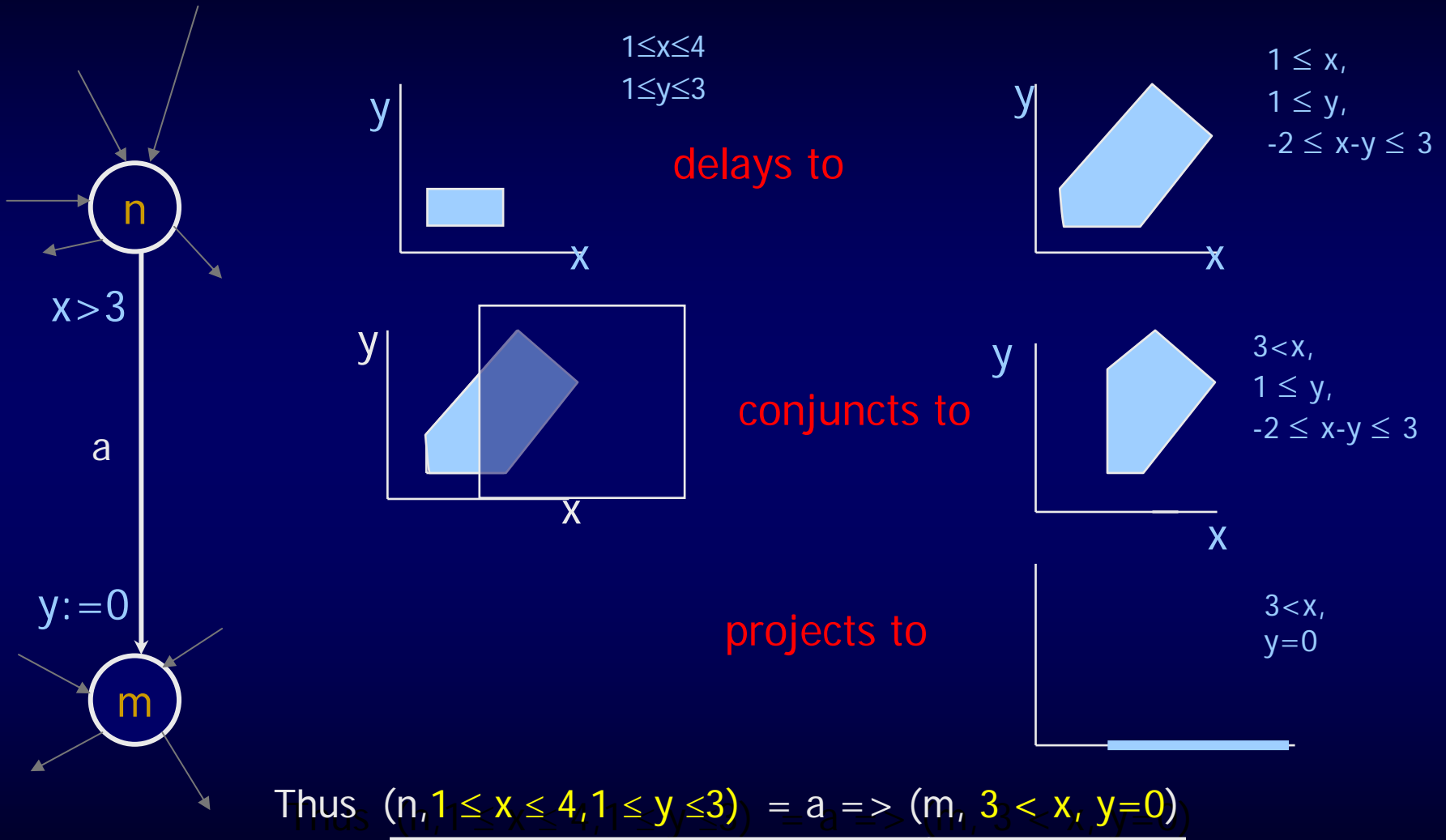
Zone:

conjunction of
 $x-y \leq n, x \leq y \leq n$



this is a time zone

Symbolic Transition



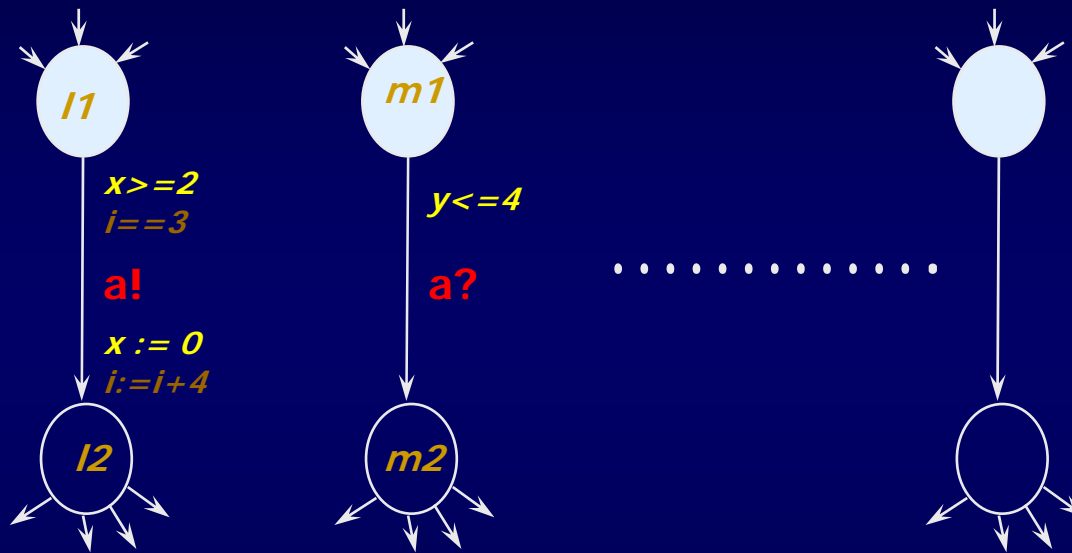
Finite symbolic simulation graph and reachable states can be computed

this is a symbolic transition (a bunch of concrete transitions)

Modelling Timed Systems using Uppaal

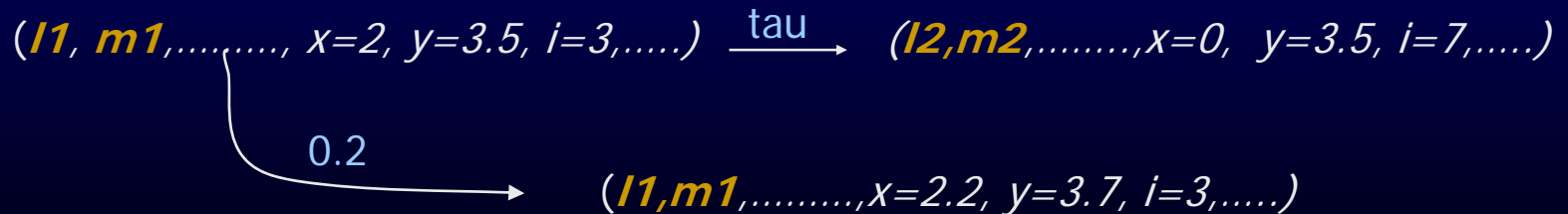
The Uppaal Model

= Networks of Timed Automata + Integer Variables + ...



Two-way synchronization on *complementary* actions.
Closed Systems!

Example transitions



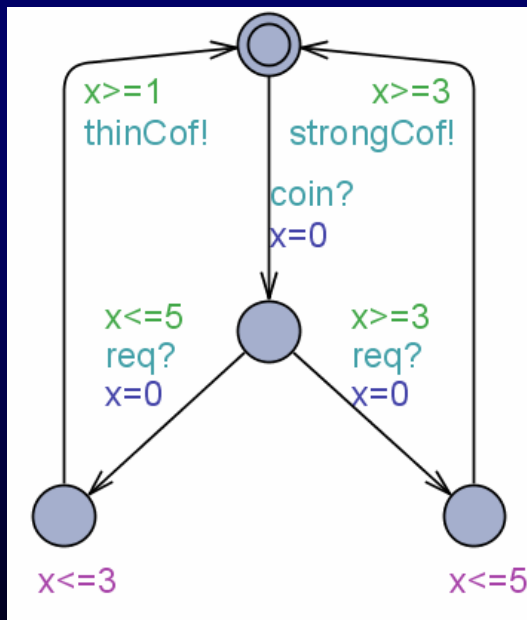
Modelling using Uppaal ...

The image displays the Uppaal software interface, which is used for modeling, simulating, and verifying real-time systems. The interface is divided into several main sections:

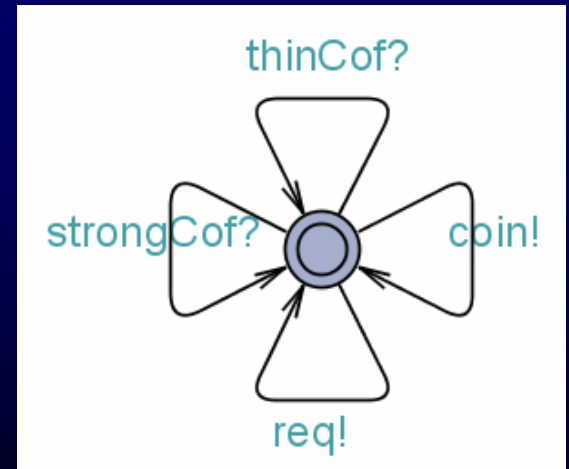
- Modeling:** The top-left window shows a state transition diagram with nodes representing states and edges representing transitions. Each transition is labeled with a guard condition and an action. For example, a transition from 'Safe' to 'Cross' is guarded by $x=0$ and has the action $x:=x+1$. Other states include 'Appr', 'Start', and 'Stop'.
- Simulation:** The top-right window shows a detailed simulation of the model. It includes a 'Simulation Trace' table with columns for state variables (e.g., x , y , z) and a 'Trace Plot' showing the execution of the system over time. The plot shows the sequence of states and transitions, with a red arrow indicating the current state of the simulation.
- Verification:** The bottom window shows the results of model checking. It lists several properties that have been verified, such as 'Train1.Appr \rightarrow Train1.Cross' and 'Train2.Appr \rightarrow Train2.Cross'. The results indicate that all properties are satisfied, except for one which is marked as 'not deadlock'.

Large grey arrows indicate the flow of information between these stages: from Modeling to Simulation, and from Simulation to Verification.

Timed Automaton of Coffee Machine



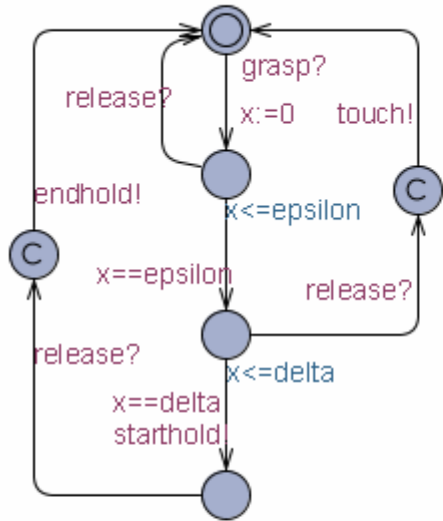
Machine Model



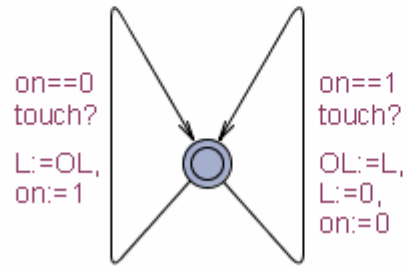
Possible users-model

Touch Sensitive Light Controller

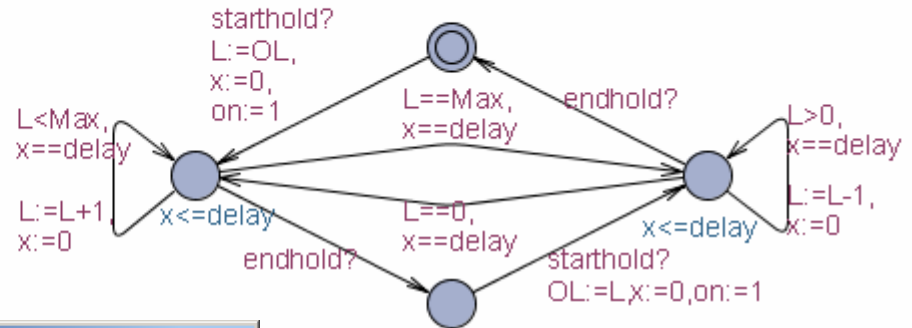
Interface



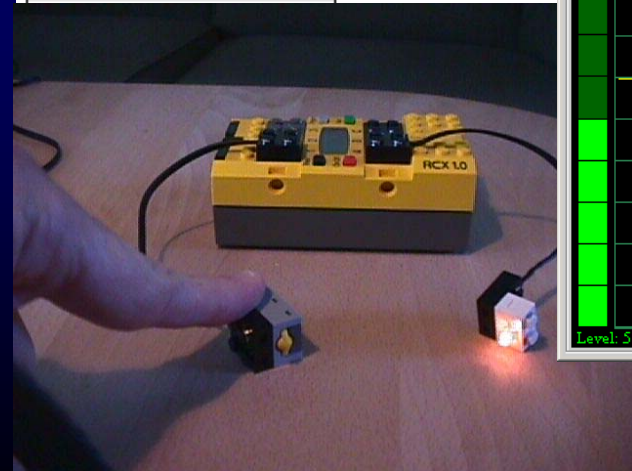
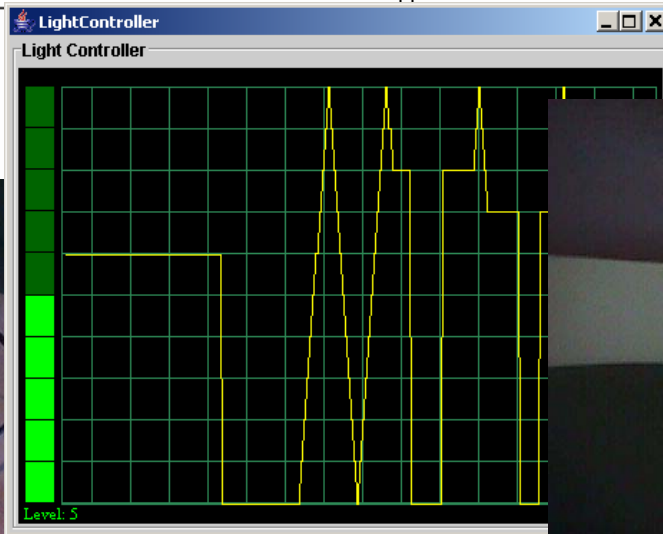
Switch



Dim



User

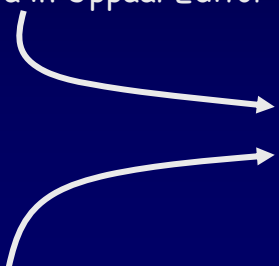


Verification using Uppaal

Uppaal as a box...

System description

Timed Automata in Uppaal Editor



Requirement specification

Temporal logic formula

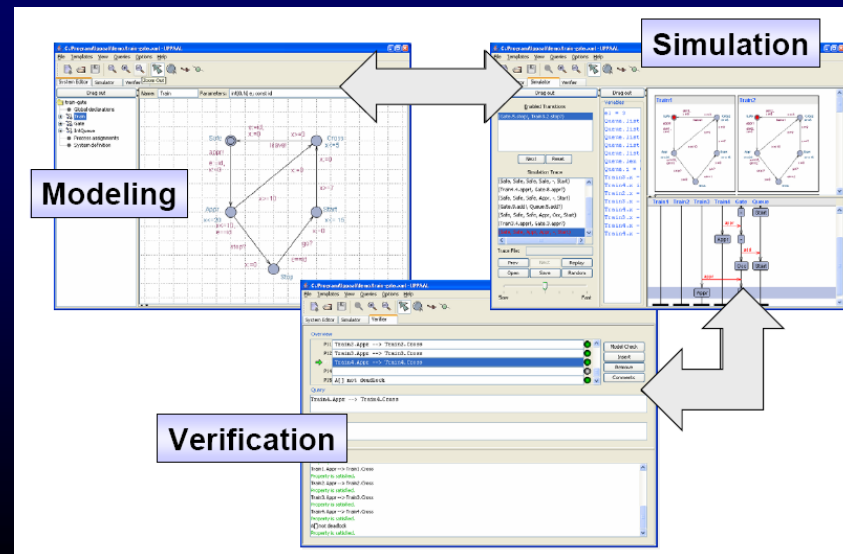


No!

Diagnostic Information



Yes!



What does Verification do

- Compute *all* possible execution sequences
- And consequently to examine *all* states of the system
- *Exhaustive search => proof*
- Check if
 - every state encountered does not have the **undesired** property --> **safety property**
 - some state encountered has the desired property --> **reachability property**

Properties

- **Safety**

- Nothing bad happens during execution
- System never enters a bad state
 - Eg. mutual exclusion on shared resource

- **Liveness**

diffent from reachability property

- Something good eventually happens
- Eventually reaching a desired state
 - Eg. a process' request for a shared resource is eventually granted

UPPAAL Property Specification Language

- $A[] p$
- $A<> p$
- $E<> p$
- $E[] p$
- $P \dashrightarrow q$

process location

data guards

clock guards

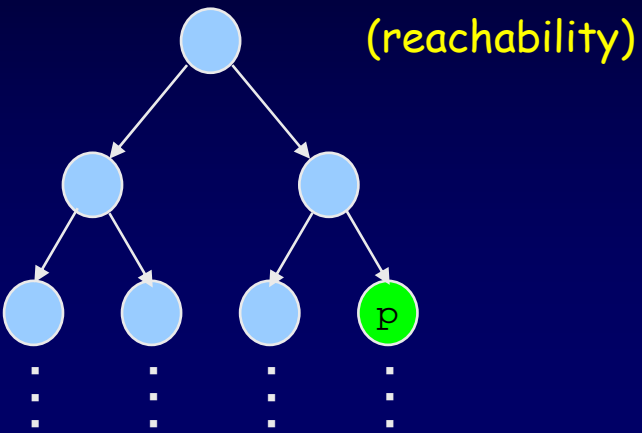
"p leads to p":
 $A[] (p \text{ imply } A<> q)$

$p ::= a.l \mid g_a \mid g_c \mid p \text{ and } p \mid$
 $p \text{ or } p \mid \text{not } p \mid p \text{ imply } p \mid$
 $(p) \mid \text{deadlock (only for } A[], E<>)$

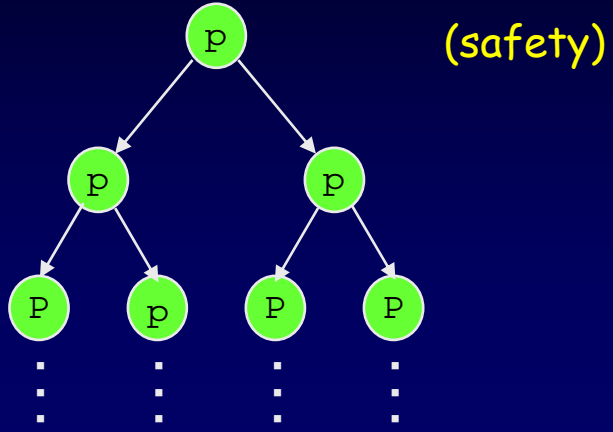
$A[] (\text{mc1.finished and mc2.finished}) \text{ imply } (\text{accountA+accountB==200})$

Uppaal "Computation Tree Logic"

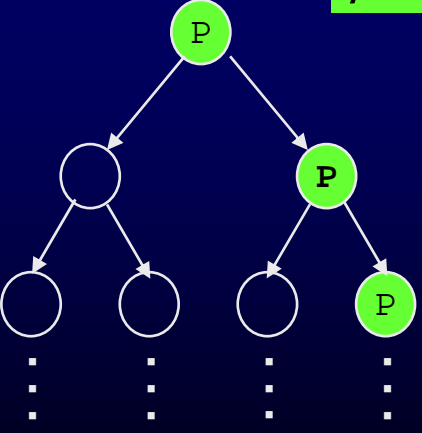
$E \langle \rangle p$ **Possible**



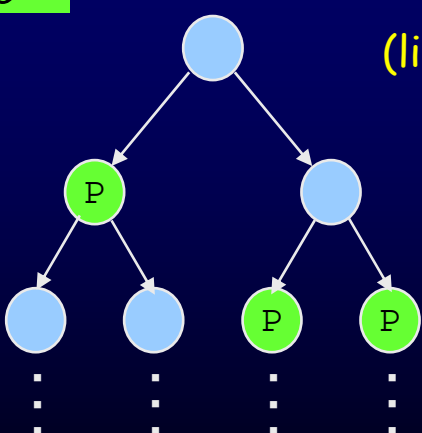
$A [] p$ **always**



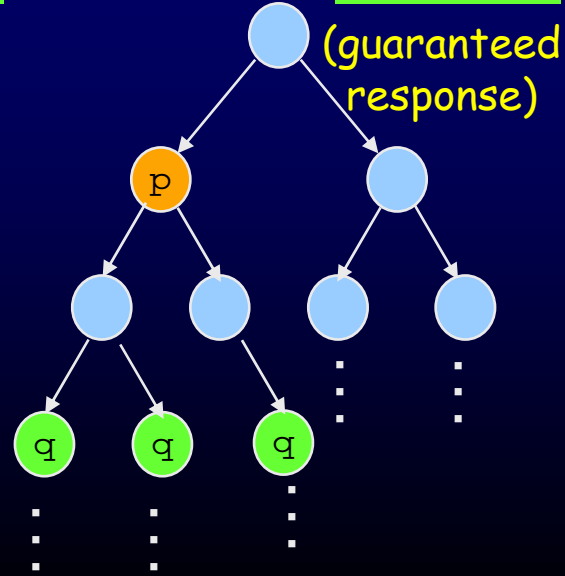
$E [] p$ **potentially always**



$A \langle \rangle p$ **inevitable**

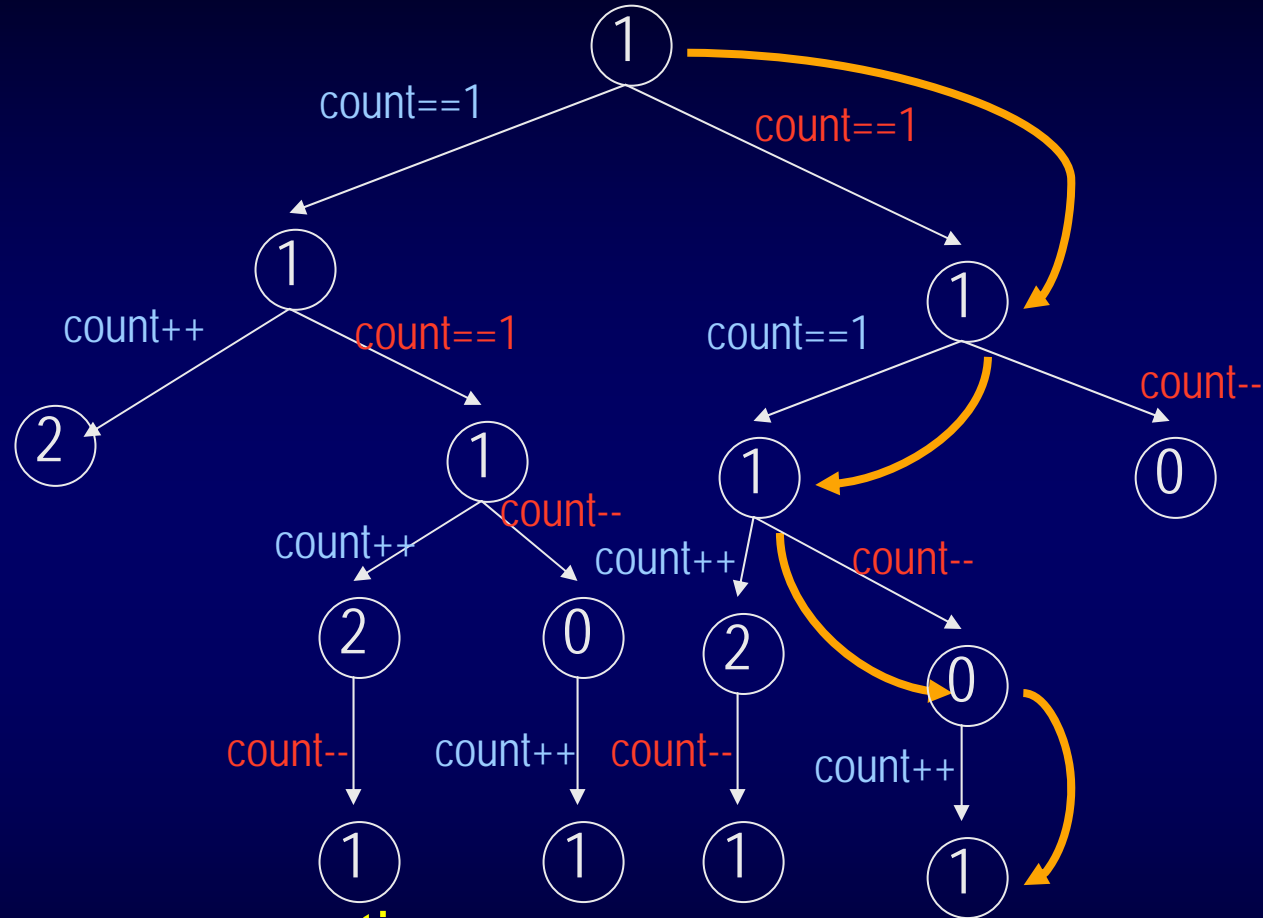
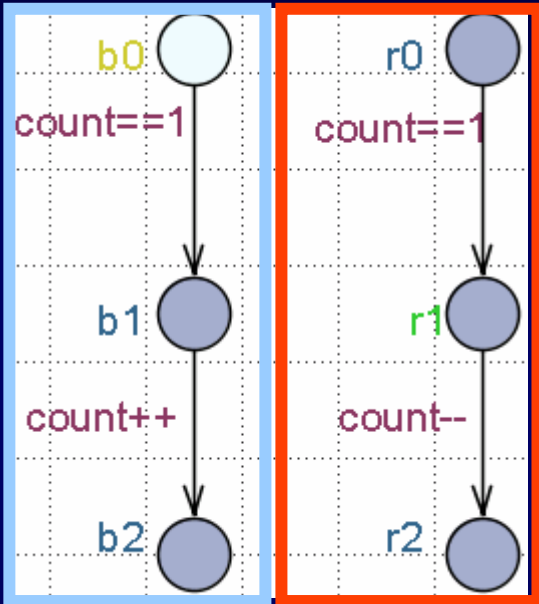


$p \dashrightarrow q$ **leads-to**



State Space Exploration

Int count := 1



- Each trace = a program execution

- Uppaal checks *all* traces

- Is count *possibly* 3 ?

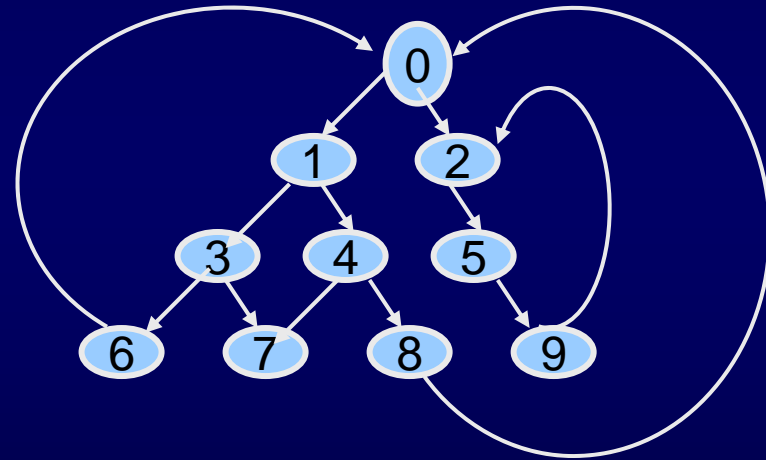
E<> count==3

- Is count *always* 1 ?

A[] count==1

Reachability Analysis

```
Passed:=∅           //already seen states
waiting:={S_0}      //states not examined yet
While(waiting!=∅) {
  waiting:=waiting\{s_i}
  if s_i ∉ Passed
    whenever (s_j → s_j) then
      waiting:=waiting ∪ s_j
}
```



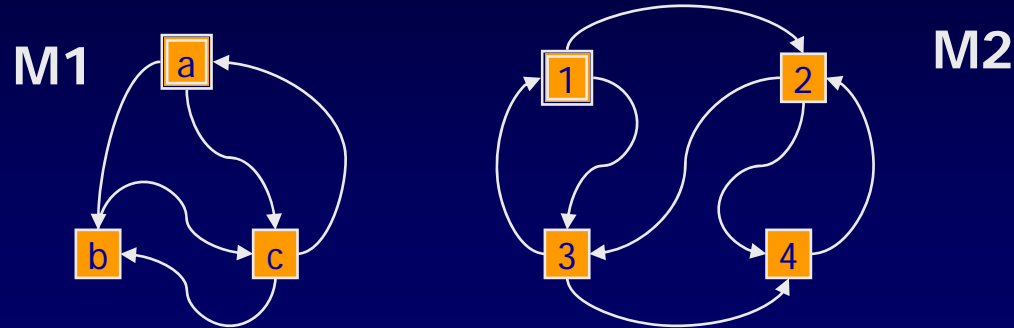
Depth-First: maintain waiting as a **stack**

Order: 0 1 3 6 7 4 8 2 5 9

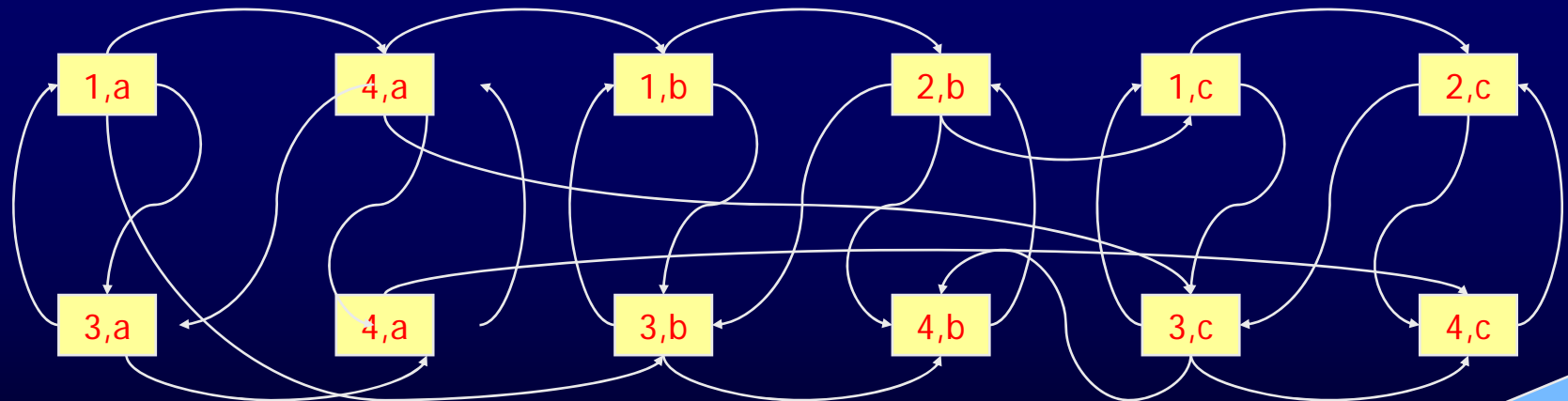
Breadth-First: maintain waiting as a **queue**
(shortest counter example)

Order: 0 1 2 3 4 5 6 7 8 9

'State Explosion' problem



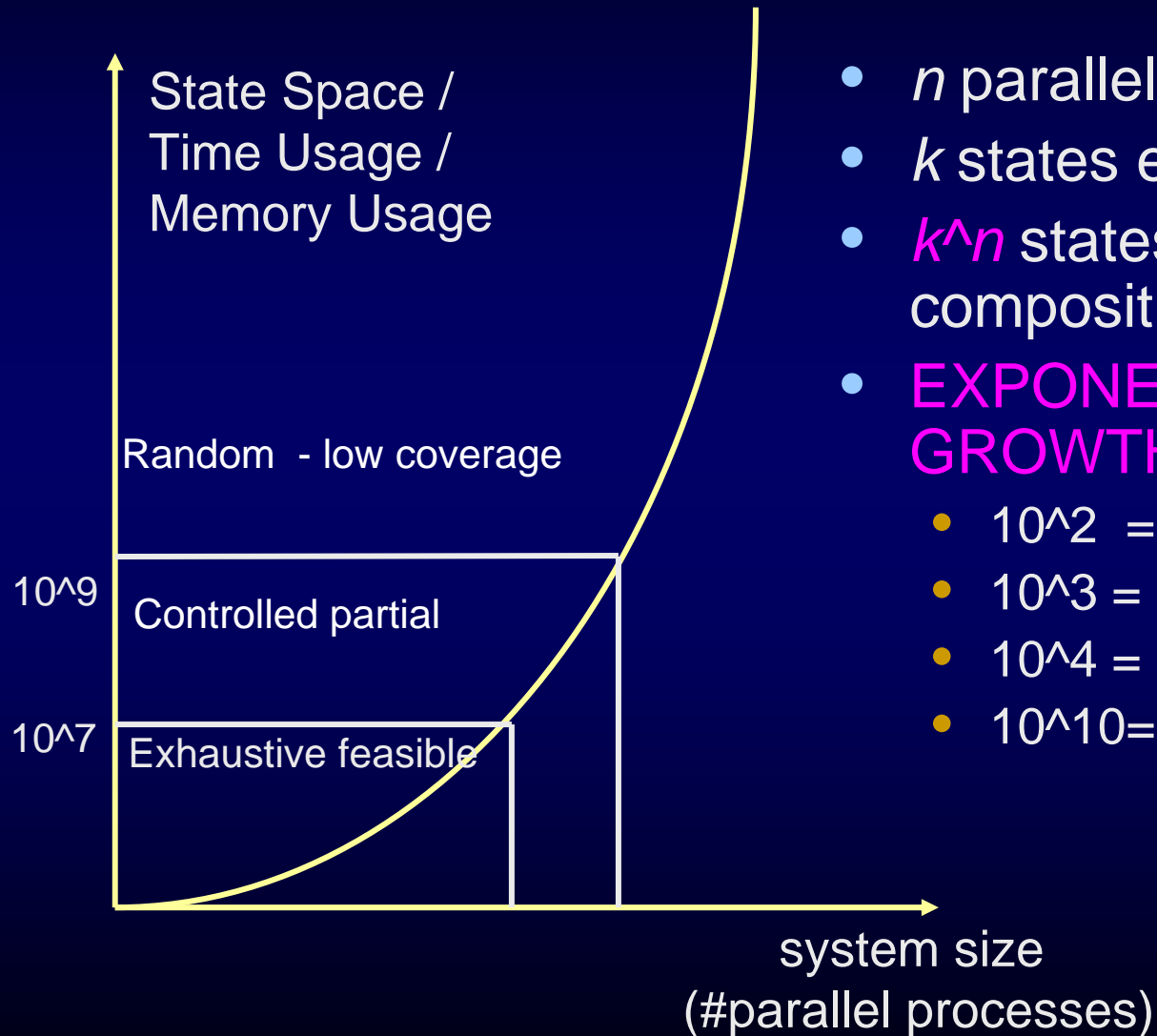
M1 x M2



All combinations = exponential in no. of components

Provably theoretical intractable

Limitations to Reachability Analysis



- n parallel FSMs
- k states each
- k^n states in parallel composition
- **EXPONENTIAL GROWTH**
 - $10^2 = 100$
 - $10^3 = 1000$
 - $10^4 = 10000$
 - $10^{10} = 10000000000$

What Influences System Size?

- Number of parallel processes
- Amount of non-determinism
- Queue sizes
- Range of discrete data values
- Environment assumptions
 - Speed
 - Kinds of messages that can be sent in what states
 - Data values

Counter Measures

- Use abstraction, simplification
 - Only model the aspects relevant for the property in question
- Economize with (loosely synch'ed) parallel processes
- Make precise assumptions and restrictions
- Range of data values
 - Use *bounded* data values: integer (0:4);
 - *Reset variables* to initial value whenever possible
 - Avoid complex data structures
- Partial (controlled) search heuristics
 - Bit-State hashing
 - Limit search depth
 - Restrict scheduling
 - Priority to internal transitions over env input
 - Schedule process in FIFO style rather than ALL interleavings

Does verification guarantee correctness?

- Only models verified, not (physical) implementations
- Made the right model?
- Properties correctly formulated?
- The right properties?
- Enough properties?
- System size too large for exhaustive check
- Modelling effort itself revealing
- Increased confidence earlier
- Cheaper
- Even partial and random search increases confidence

Next lecture
- Model-Based Testing!