

Modeling, Verification, and Testing of Reactive Systems

(Adapted from Brian Nielsen's slides)

Agenda

- Overview
 - Reactive systems
 - Formal models (LTS, FSM, EFSM, Statecharts)
- System modeling
 - Simple FSM modeling
 - FSM modeling and simulation using Uppaal
- System verification
 - Model checking using Uppaal
- Model-based testing

Reactive vs. Transformational Systems

- **reactive systems:**
 - "process control"
 - control-intensive
 - running "forever"
- **transformational systems:**
 - "data processing"
 - computation-intensive
 - to deliver a result within a time frame

computer
+ control
+ communication

Examples

computational science:
huge sparse matrices,
partial differential equations,
...

Reactive

- embedded systems (e.g., in consumer electronics, mobile phones, GPS)
- operating systems
- communication protocols
- web servers
- air traffic control
- computer games
- ...



Transformational

- numerical analysis and statistics software packages such as MatLab, Mathematica, R
- a "filter", e.g., programming language compilers
- ...



Why Software Models

- Jumping from informal project (paper) documentations to code implementation:
 - Not advisable, and probably
 - Not feasible
- The benefits of models:
 - for system development, (model-based development)
 - for system validation,
 - Verification
 - Simulation
 - Testing

A Classification of Software Models

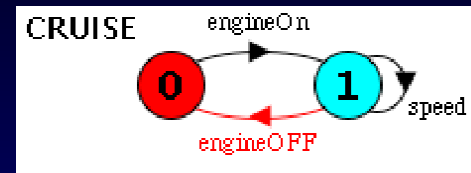
- Informal model
 - Documentation in prose, schematic block diagrams, etc.
- Semi-formal model
 - Unified Modeling Language (UML) diagrams
- • Formal model
 - Well-formed mathematical models, usually with clearly defined syntax and semantics

Formal Models Classified

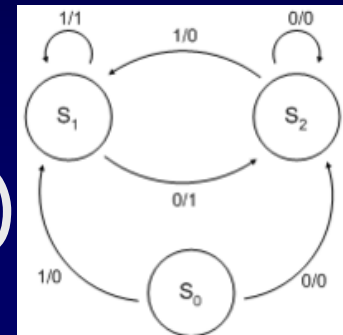
- history-based model (temporal logics)
- state-based models (Z, VDM, B spec.)
- • state transition-based models (LTS, FSM, EFSM, Statecharts)
- scenario-based models (MSC, LSC)
- operational models (Petri nets, process algrbras)

State Transition-Based Models

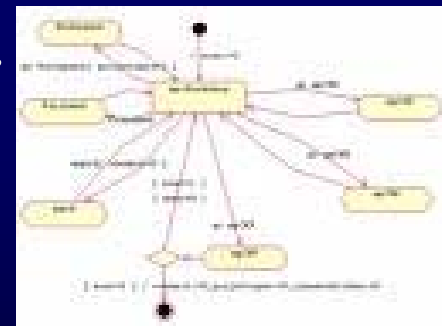
- LTS (Labeled Transition Systems)



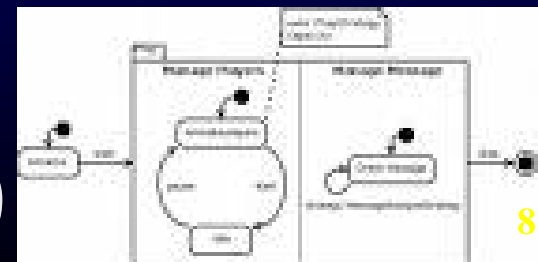
- FSM (Finite State Machines)



- EFSM (FSM + guards + assignments)



- Statecharts (EFSM + concurrency + hierarchy + broadcast communication)



Labeled Transition System (LTS)

Labeled Transition Systems

- Labeled Transition System (LTS)
 - Transition system labeled with (input, output, or internal) actions
 - A very basic model for describing system behavior

Behavior: How the system accepts inputs (external stimuli), changes its internal states, and produce outputs (reactions).

An Example LTS

Labelled Transition System

$\langle S, L, T, s_0 \rangle$

states

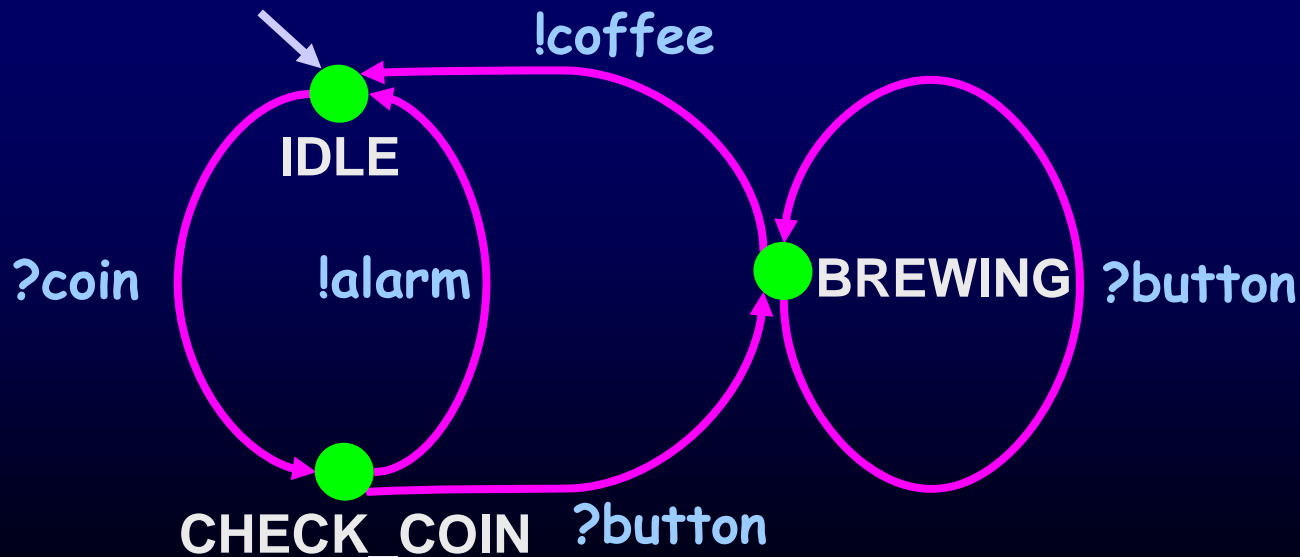
actions

transitions

initial state
 $s_0 \in S$

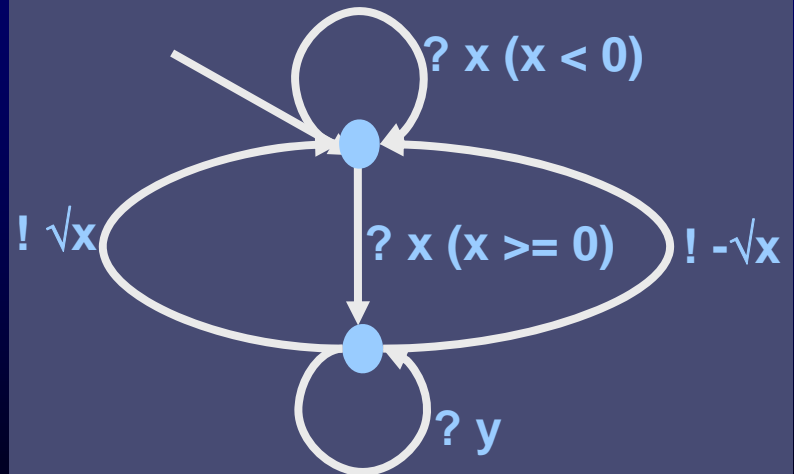
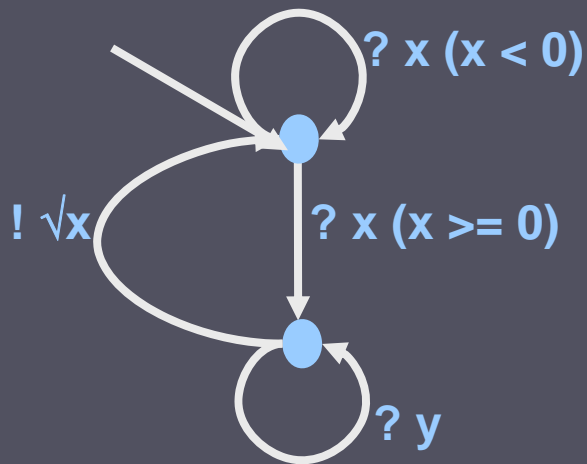
$T \subseteq S \times (L \cup \{\tau\}) \times S$

coffee vending machine



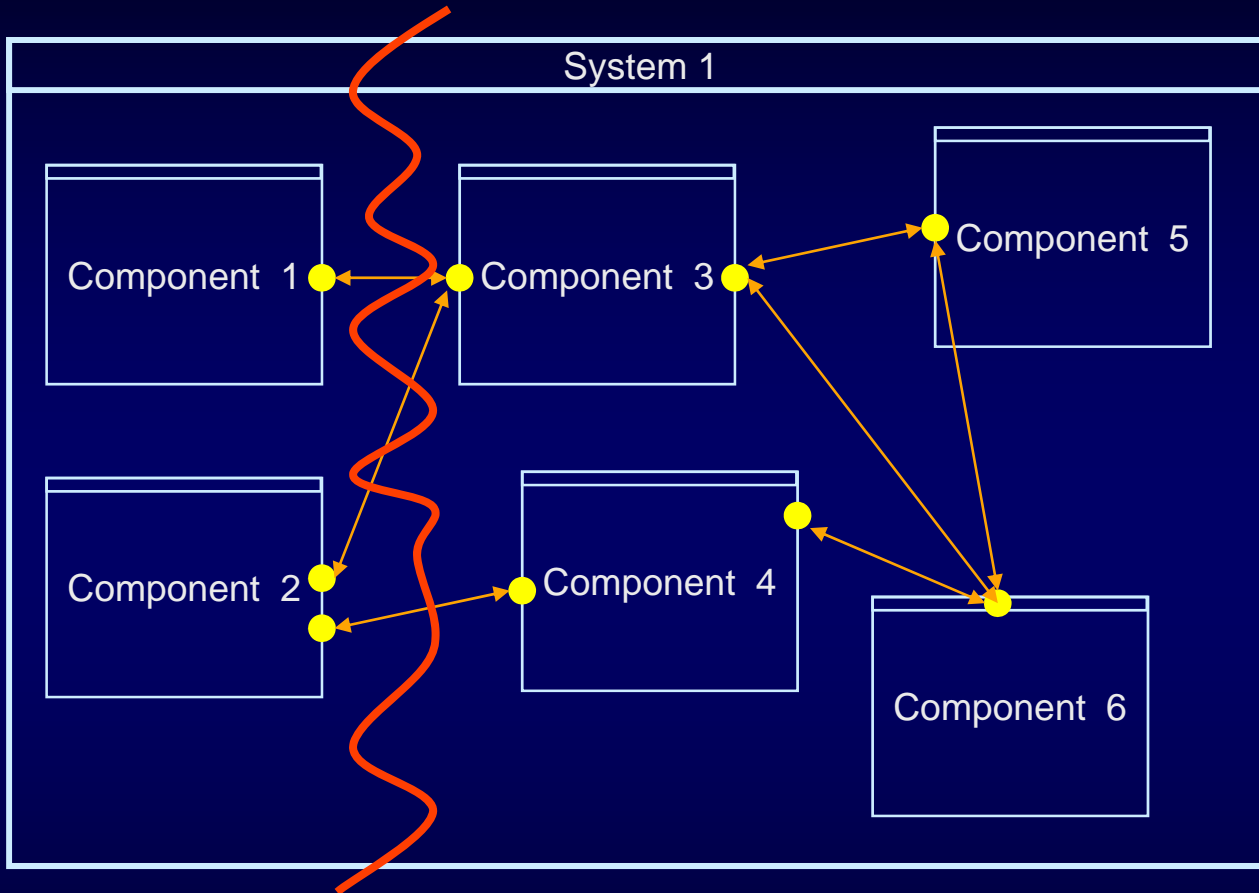
Input-Output LTS (IOLTS)

- Special kind of LTS:
Input-Output Labelled Transition System - IOLTS
 - Output actions(!), and input actions(?)
- IOLTS with variables - **equation solver** for $y^2 = x$:
 - different implementations



Finite State Machine (FSM)

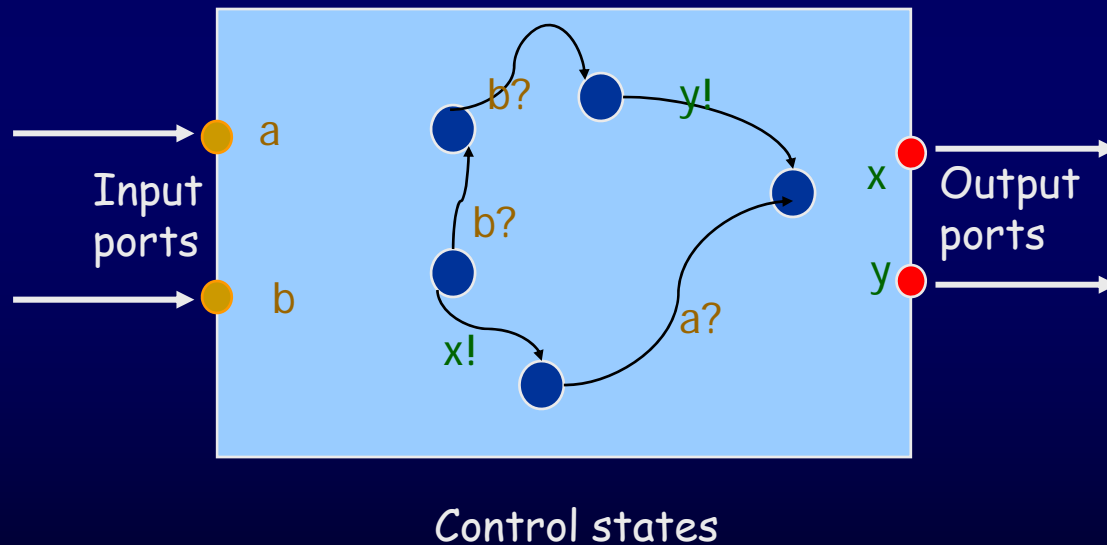
System Structure



- How do we model individual components?
- How do components interact? // by message passing
- How do we specify environment assumptions?
- How do we ensure correct behaviour?

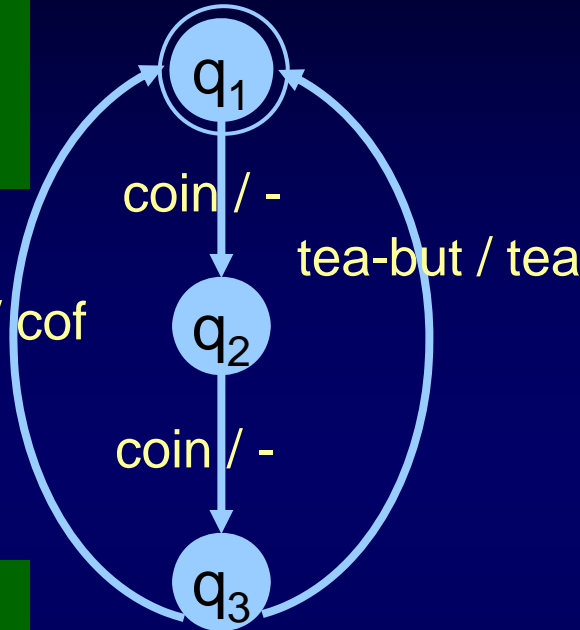
Behavior of a Component

Unified Model: **State Machine**



Finite State Machine (Mealy machine)

coffee vending machine



in tabular form,

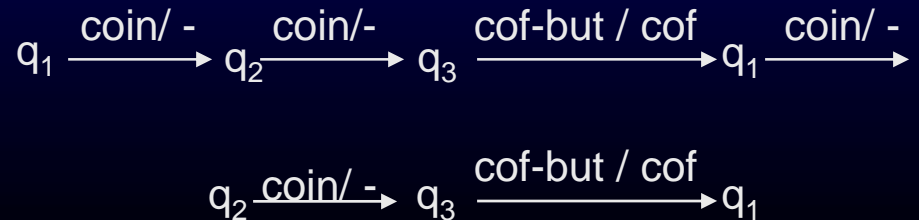
condition		effect	
current state	input	output	next state
q ₁	coin	-	q ₂
q ₂	coin	-	q ₃
q ₃	cof-but	cof	q ₁
q ₃	tea-but	tea	q ₁

formally,

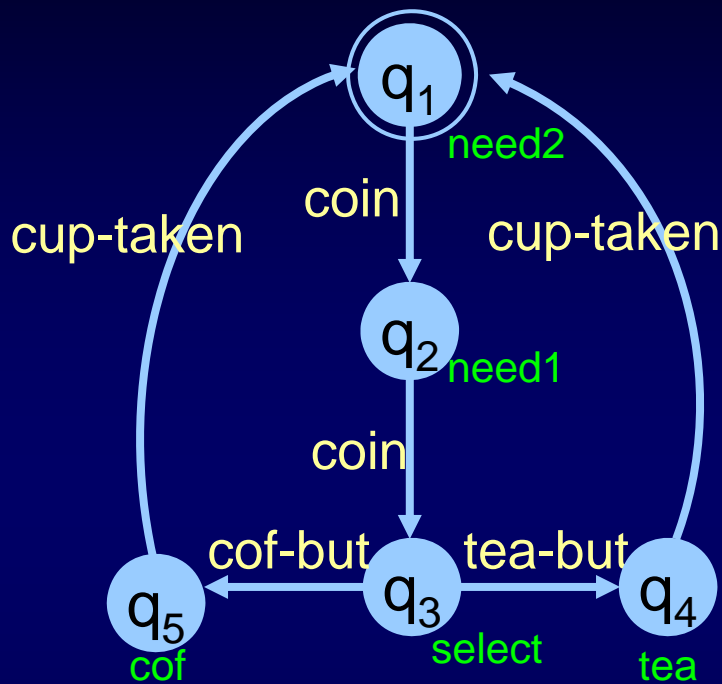
Inputs = {cof-but, tea-but, coin}
 Outputs = {cof, tea}
 States: {q₁, q₂, q₃}
 Initial state = q₁
 Transitions = {
 (q₁, coin, -, q₂),
 (q₂, coin, -, q₃),
 (q₃, cof-but, cof, q₁),
 (q₃, tea-but, tea, q₁)
 }

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

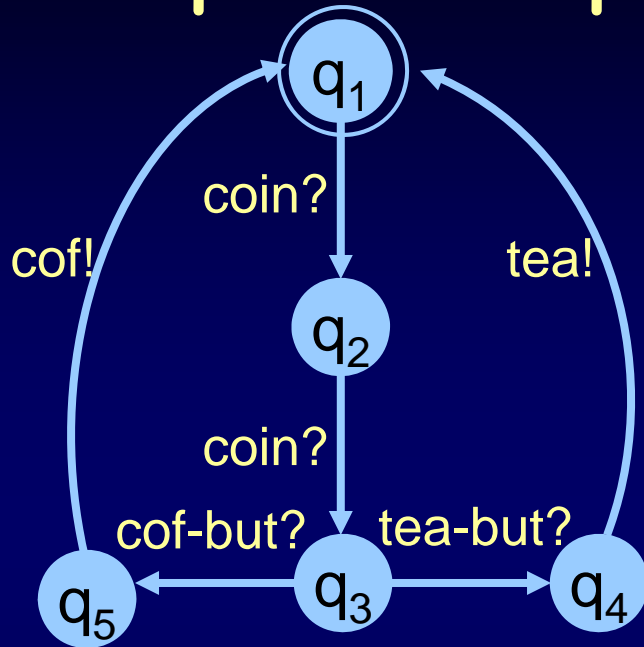
Output sequence: need2.need1.select.cof. need2.need1.select.cof

e.g., need2 = to prompt "please insert two coins"

Comparing FSM and LTS

- LTS is more fundamental, more naive and simpler
 - each transition step is "atomic"
- FSM has always alternation between inputs and outputs
 - though sometimes they may be "-"
- LTS can serve as underlying semantics model for many other formalisms (including FSM)

Input-Output FSM (IO-FSM)

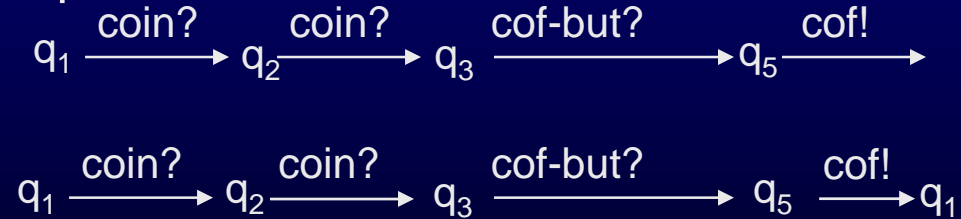


condition		effect
current state	action	next state
q ₁	coin?	q ₂
q ₂	coin?	q ₃
q ₃	cof-but?	q ₅
q ₃	tea-but?	q ₄
q ₄	tea!	q ₁
q ₅	cof!	q ₁

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)
 }

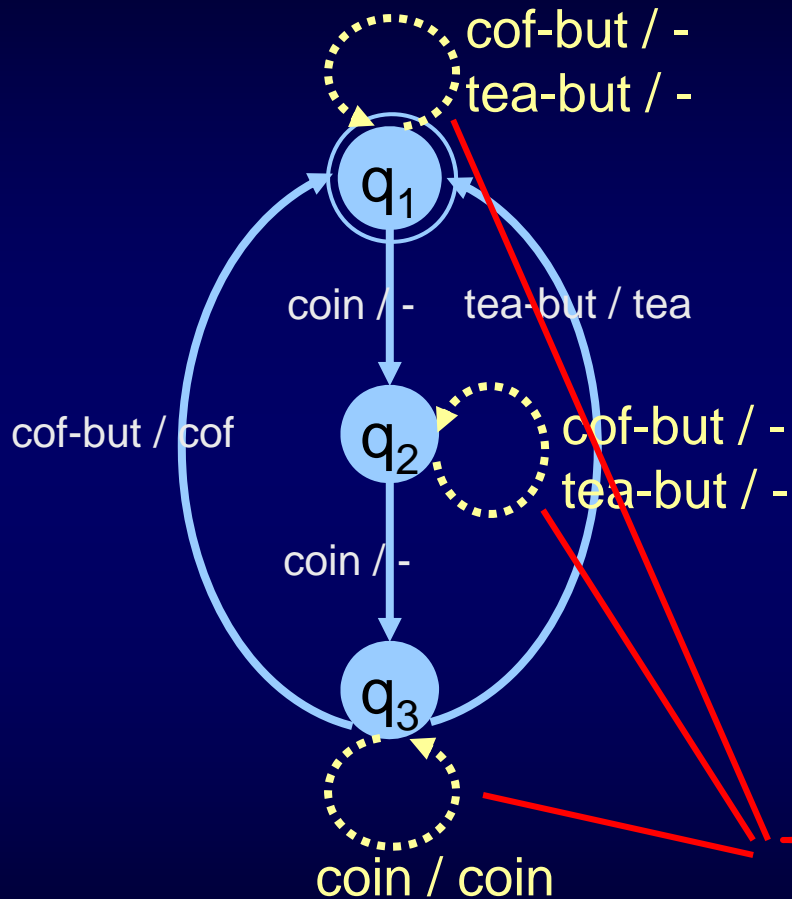
In IO-FSM, "activity" is no longer an effect.

Sample run:



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

...

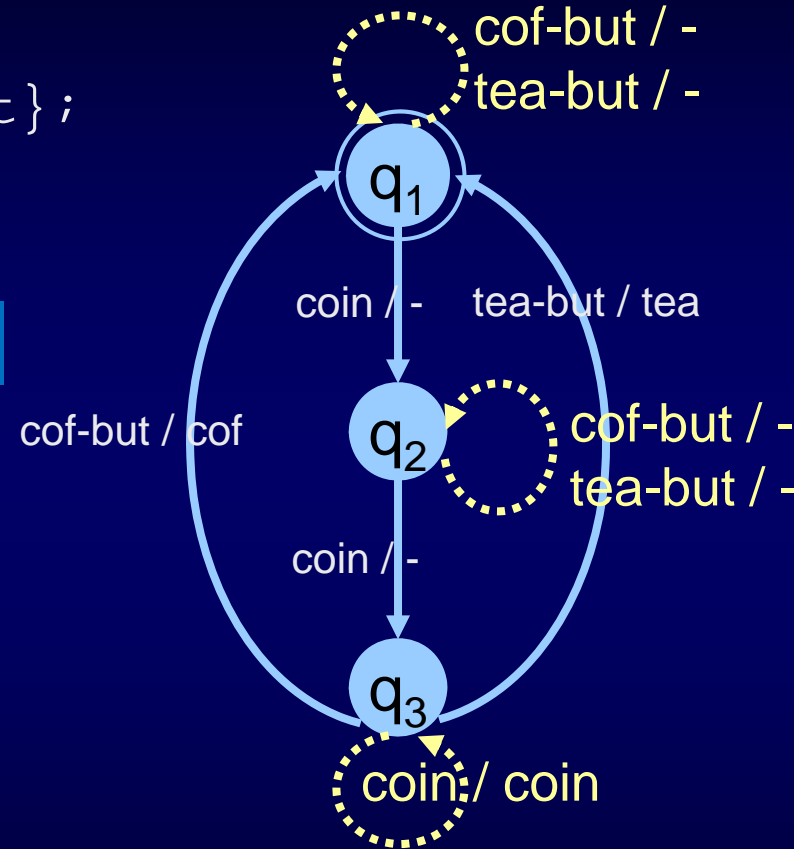
Implementing Mealy FSM (1)

```
// data structures
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 };
q1 -- coin/- --> q2
```

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

```
// skeleton algorithm
```

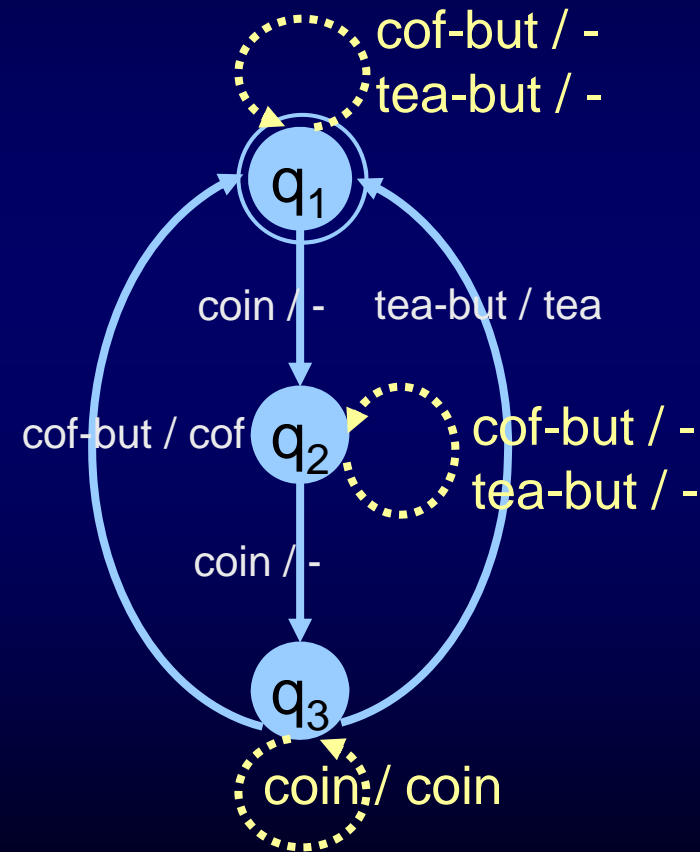
```
While(input = waitForInput()) {
    OUTPUT(outputTable[currentState, input])
    currentState := nextStateTable[currentState, input];
}
```



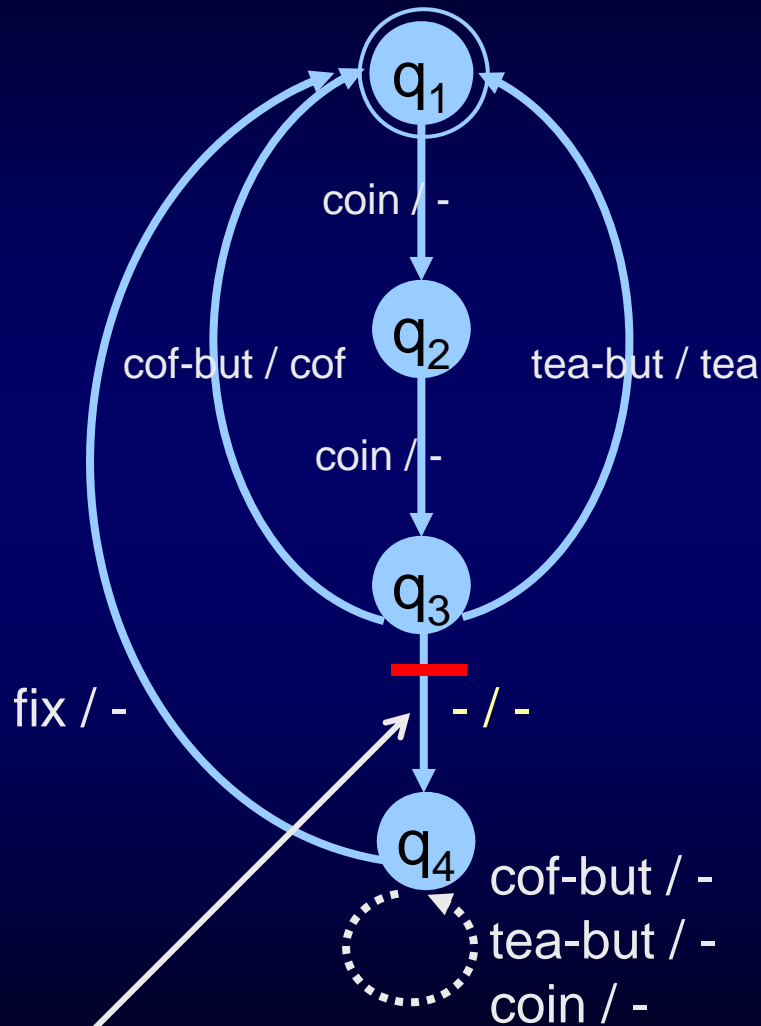
Implementing Mealy FSM (2)

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

// algorithm in more details
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 q2: ...
  case q3: {
    switch(input) {
      case cof_buif: {currentState:=q3;
                      OUTPUT(cof);
                      break;}
      ... }
    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

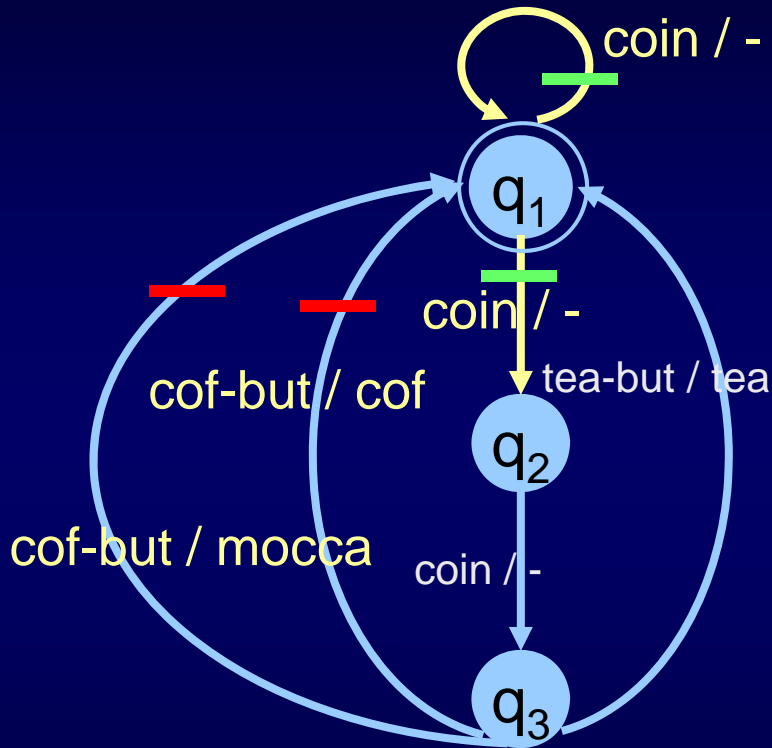
A spontaneous transition is a transition in response to **no** external input at all.

alias: **internal** transition

alias: **unobservable** transition

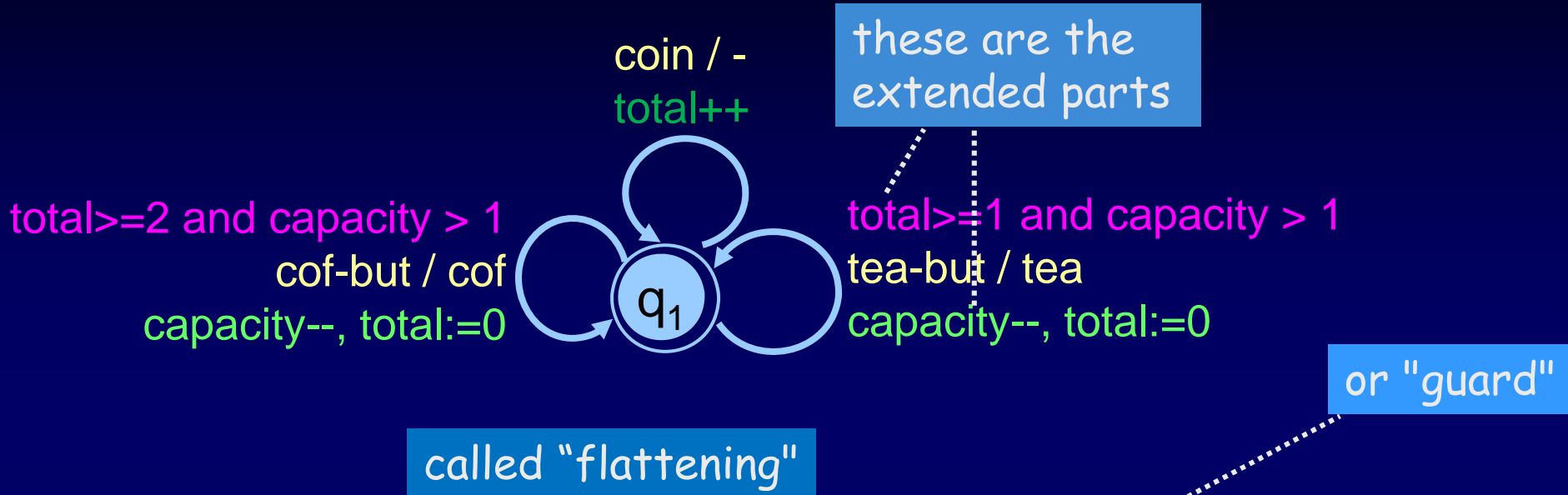
possibly due to internal sub-components interactions

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)

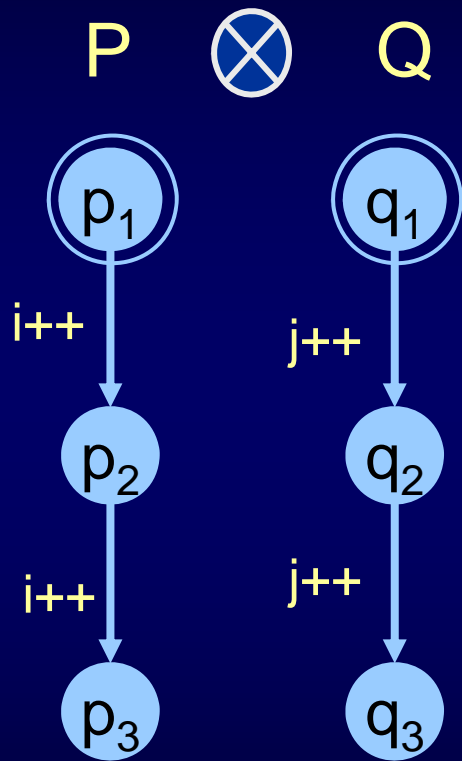


- EFSM = FSMs + variables + **enabling conditions** + **assignments**
- Can model the control aspects as well as the data aspects
- Can **be translated** into FSM if variables have bounded domains
- EFSM state: control location + variables' valuation

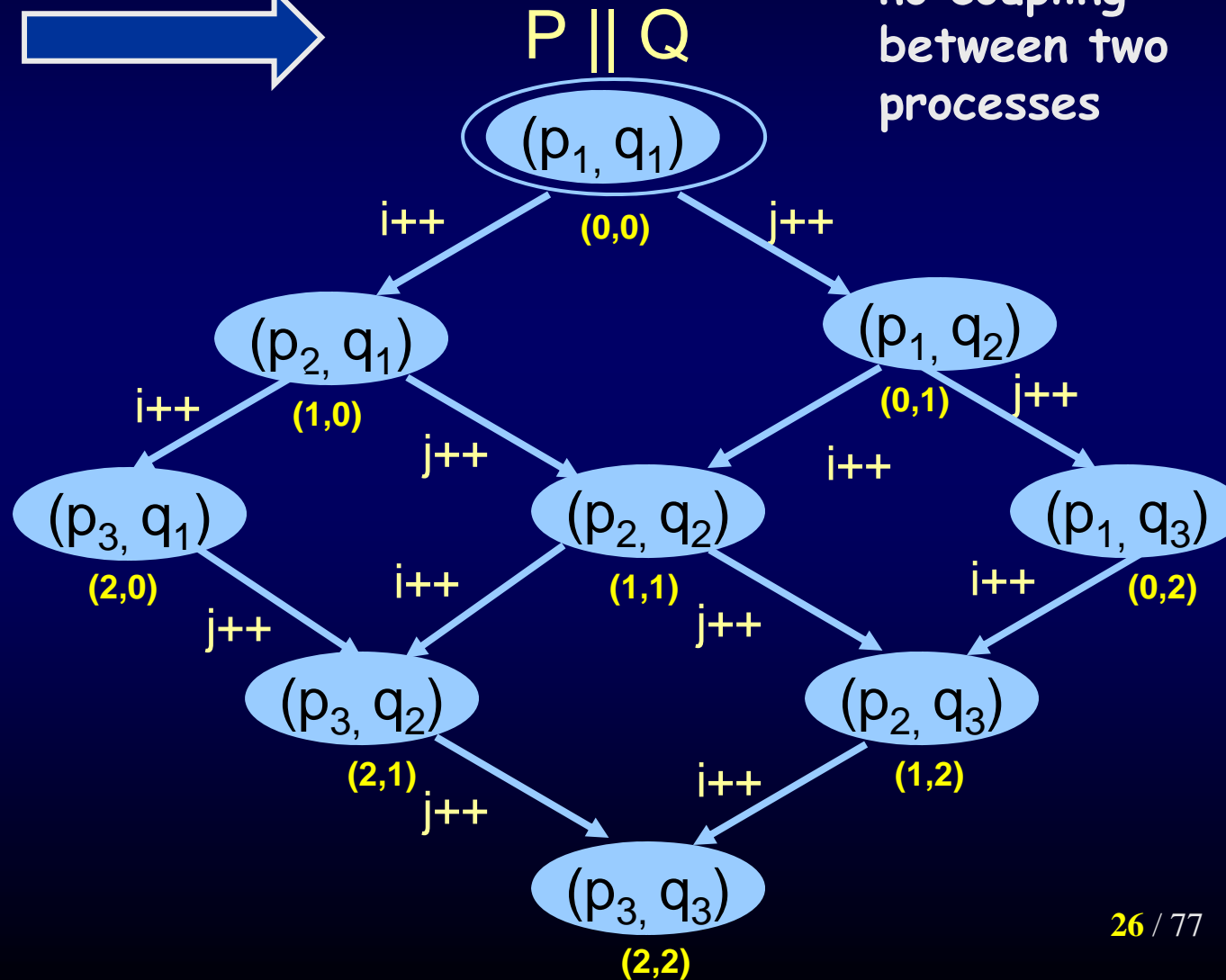
(q, total, capacity)

(q₁, 0, 10) $\xrightarrow{\text{coin} / -}$ (q₁, 1, 10) $\xrightarrow{\text{coin} / -}$ (q₁, 2, 10) $\xrightarrow{\text{cof-but} / \text{cof}}$ (q₁, 0, 9)

Parallel Composition (independently)

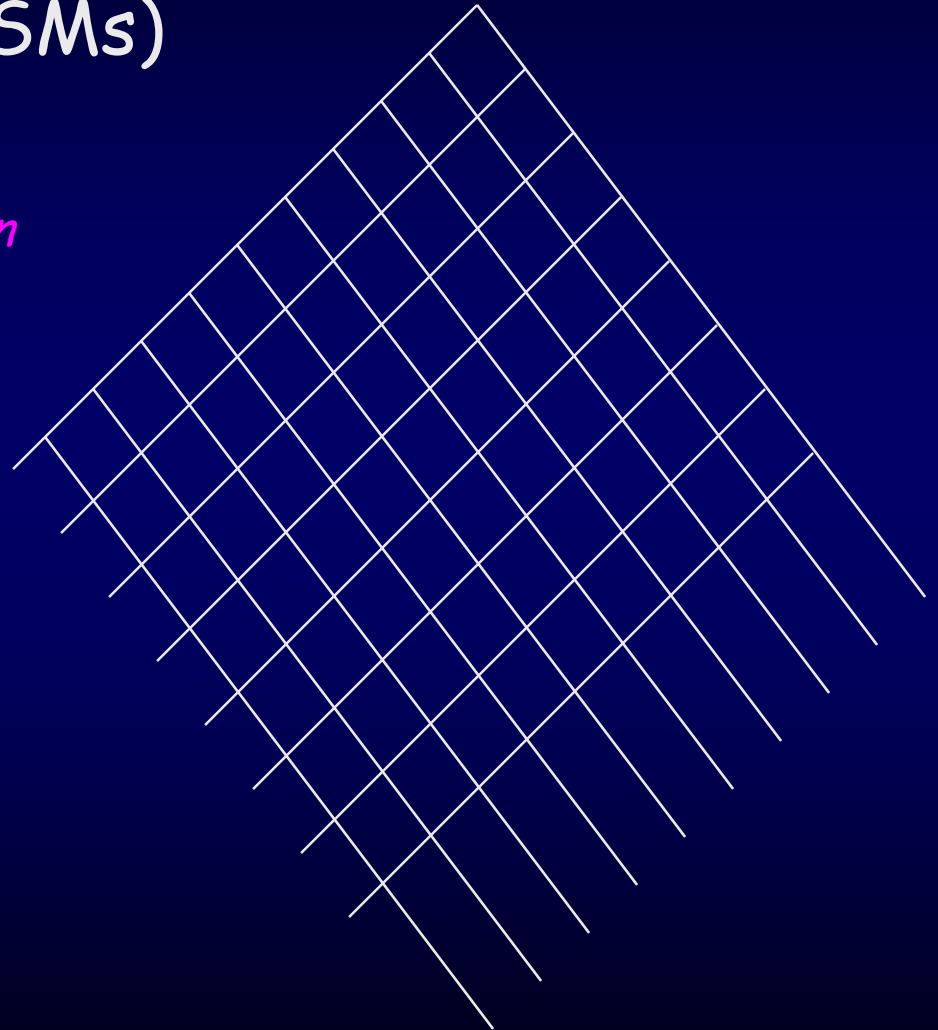


interleaving "execution"



State Space Explosion

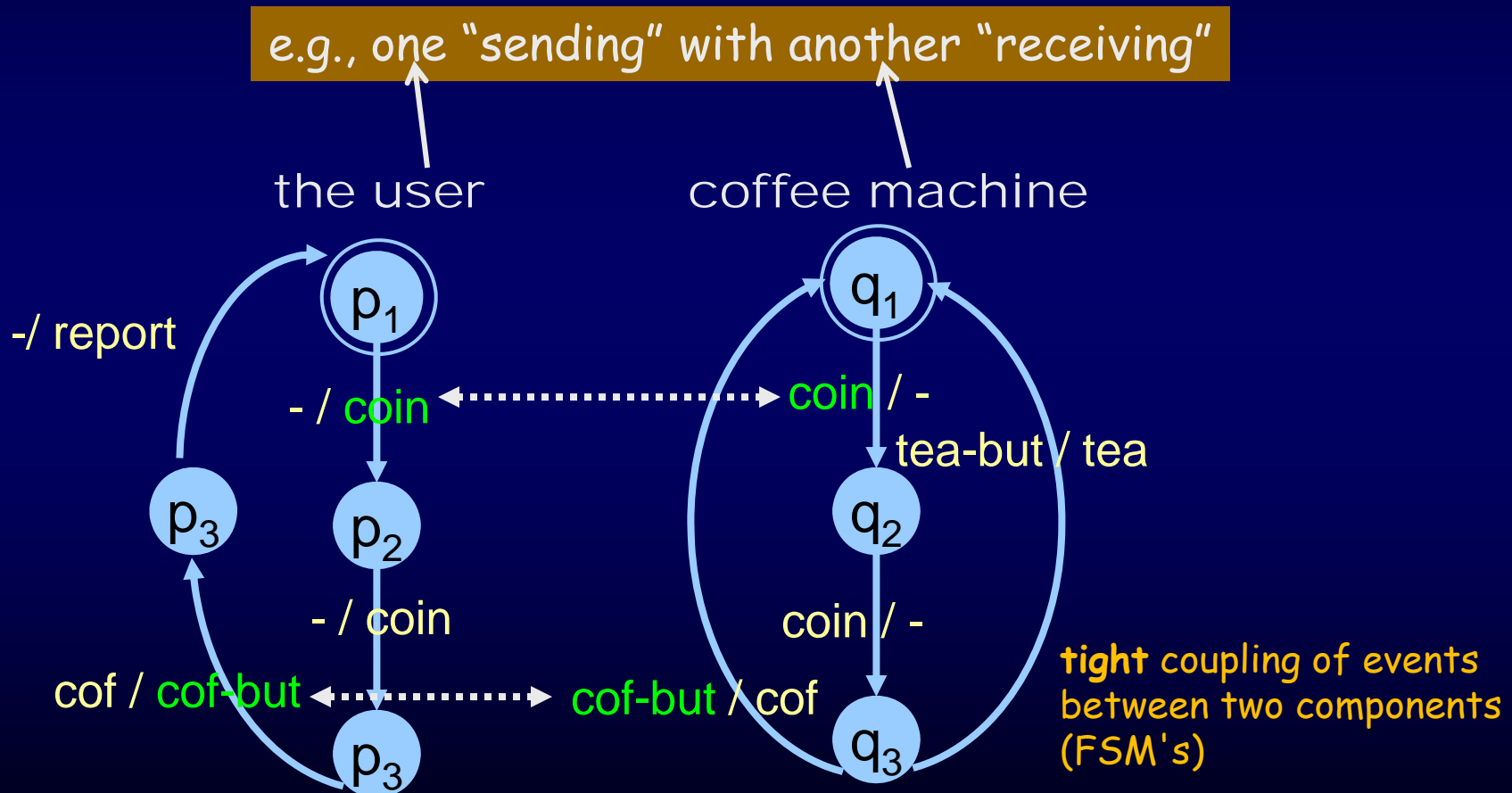
- n parallel FSMs (or EFSMs)
- Each with k states
- In parallel they have k^n states
- **EXPONENTIAL!**
 - $10^2 = 100$
 - $10^3 = 1000$
 - $10^4 = 10000$
 - $10^{10} = 10000000000$



Parallel Composition (Synchronous)

Handshake on **complementary** actions

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

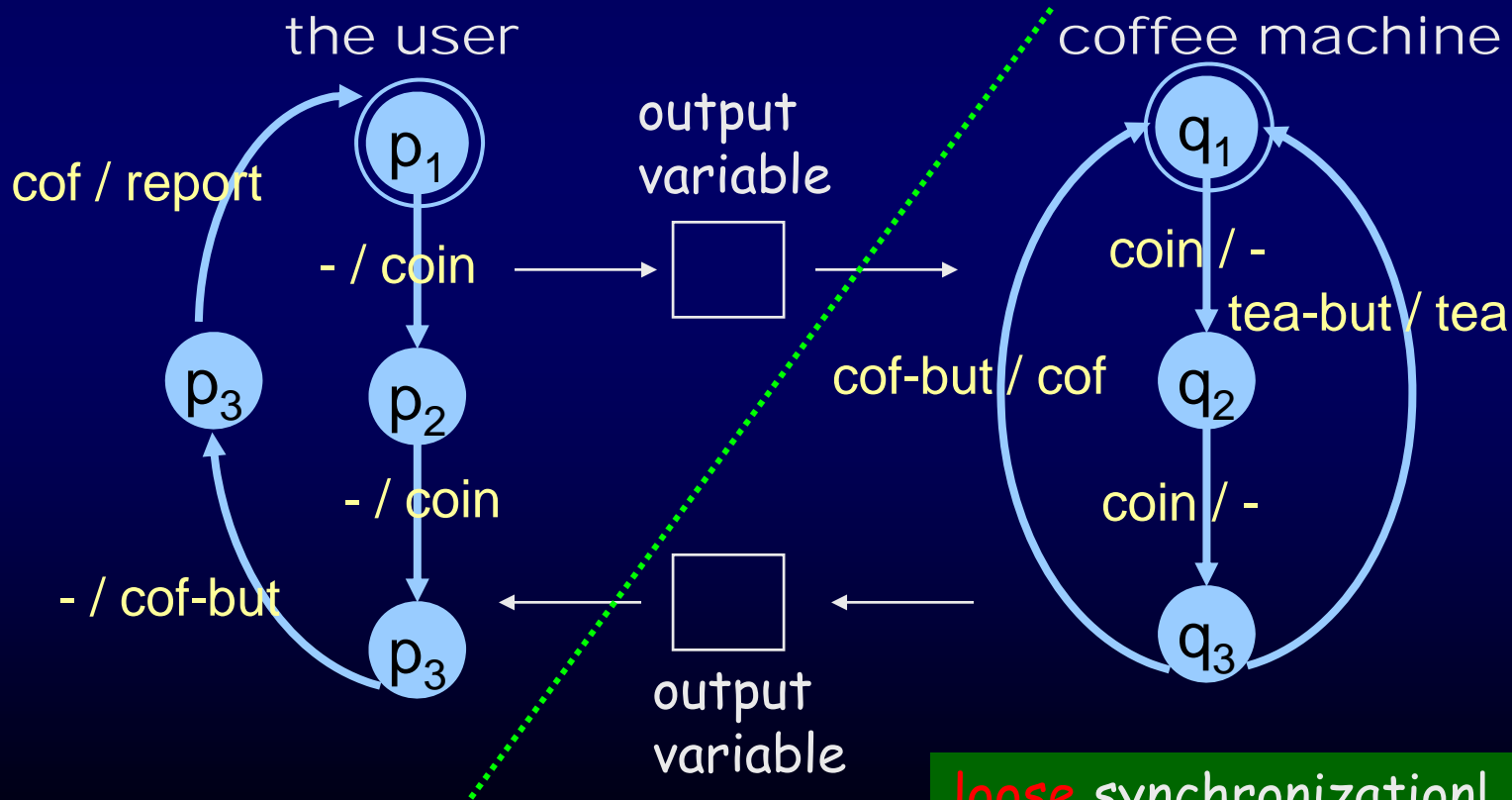


strict synchronization!

Parallel Composition (**A**synchronous)

Single output variable per FSM holds last "written" output

no handshaking any more!

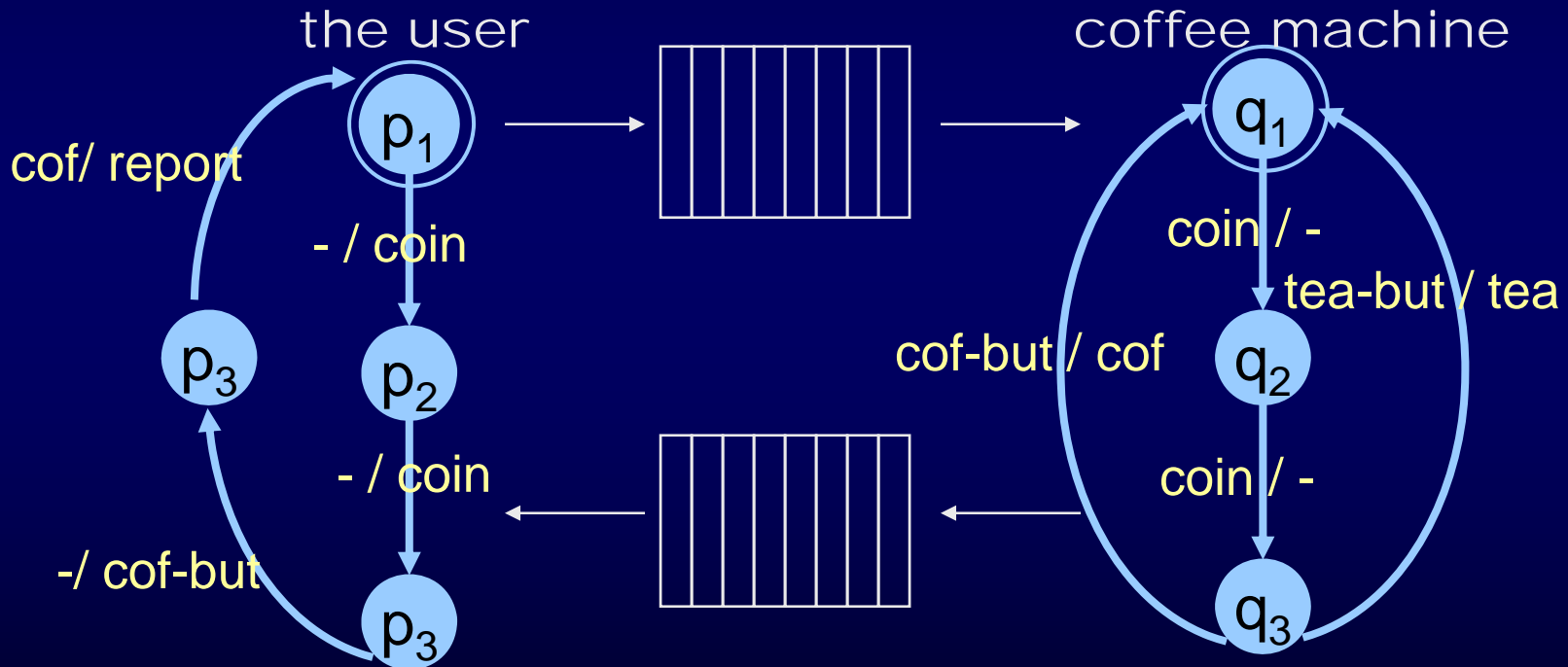


loose synchronization!

Parallel Composition (Queued)

Output is queued in (un)bounded queue

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



A system state: a snapshot of all (E)FSMs' current states and all queues

even looser synchronization!

Refactoring FSM models

- Determinizing FSM model
- Minimizing FSM model
 - For improved readability, maintainability, and implementation efficiency

Determinizing FSM

- How to determinize an FSM?
 - "subset construction" method
- Fundamental result:
 - **Every** FSM may be determinized accepting the same language.

Minimizing FSM

- Two states s and t are (language) **equivalent** iff
 - s and t accepts the same language
 - have the same set of possible traces: $tr(s) = tr(t)$
- Two Machines M_0 and M_1 are **equivalent** iff their initial states are equivalent
- A **minimized** (or "reduced") M is one that has no equivalent states
 - i.e., for all states s, t : $(s \text{ equivalent } t) \implies (s = t)$

Fundamental Results

- For each FSM there exists a language-equivalent *minimal deterministic* FSM.
- FSM's are closed under \cap and \cup
 - Or, their languages are closed under the intersection and union operators.
- FSM's may be described as regular expressions (and vice versa)

High-level State Transition-Based Models



UML State Machines

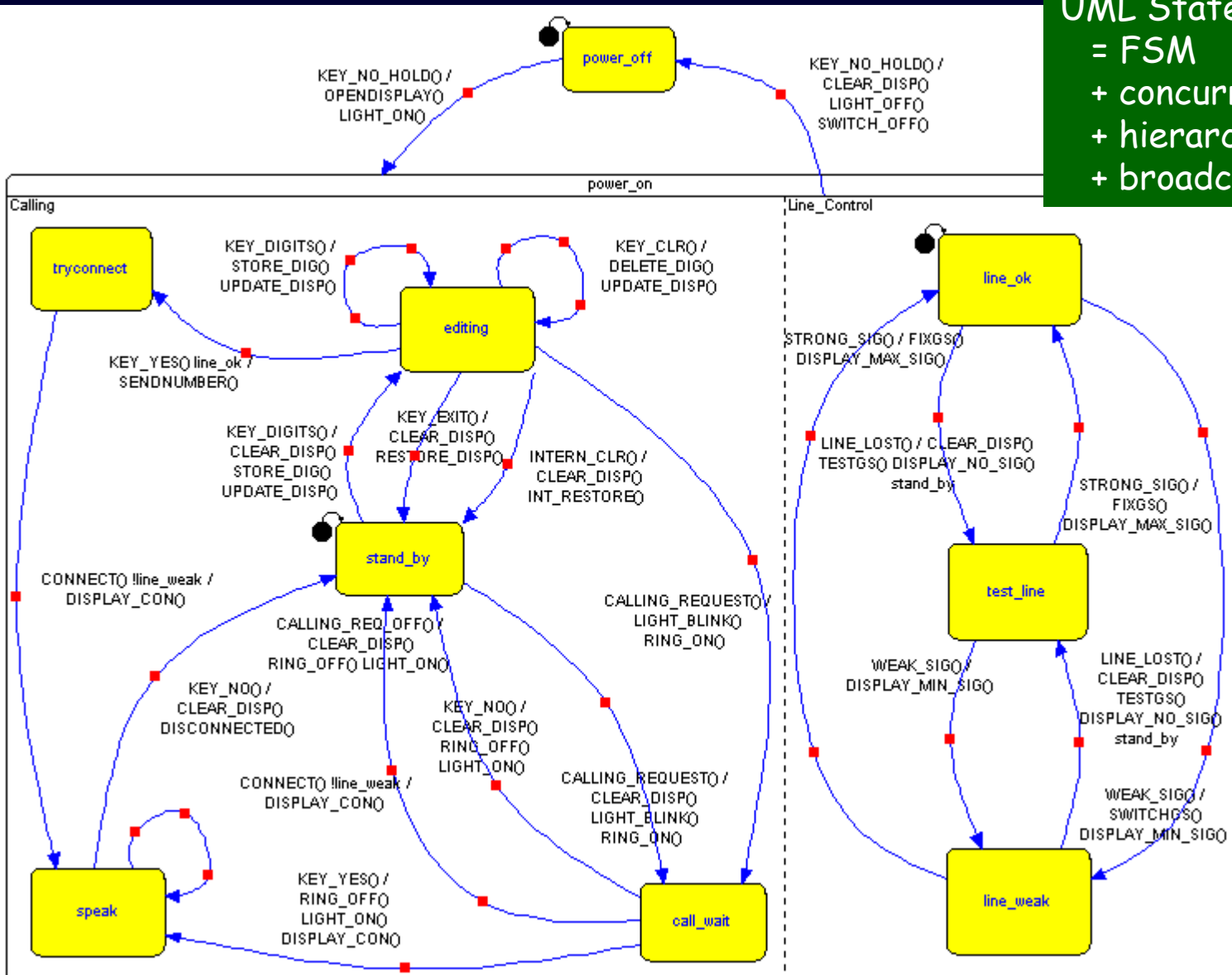
UML State Machine

= FSM

+ concurrency

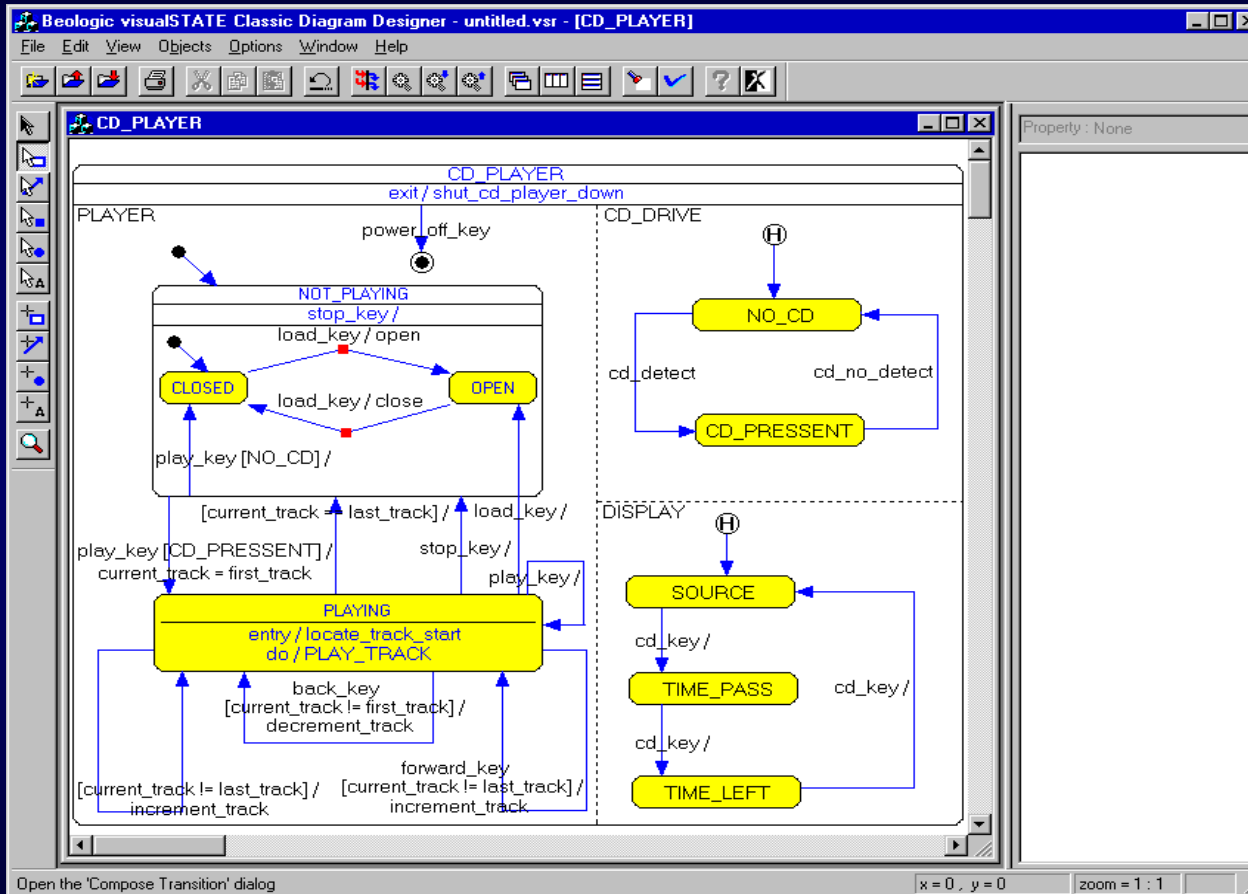
+ hierarchy

+ broadcast communication





Tool: visualSTATE Designer

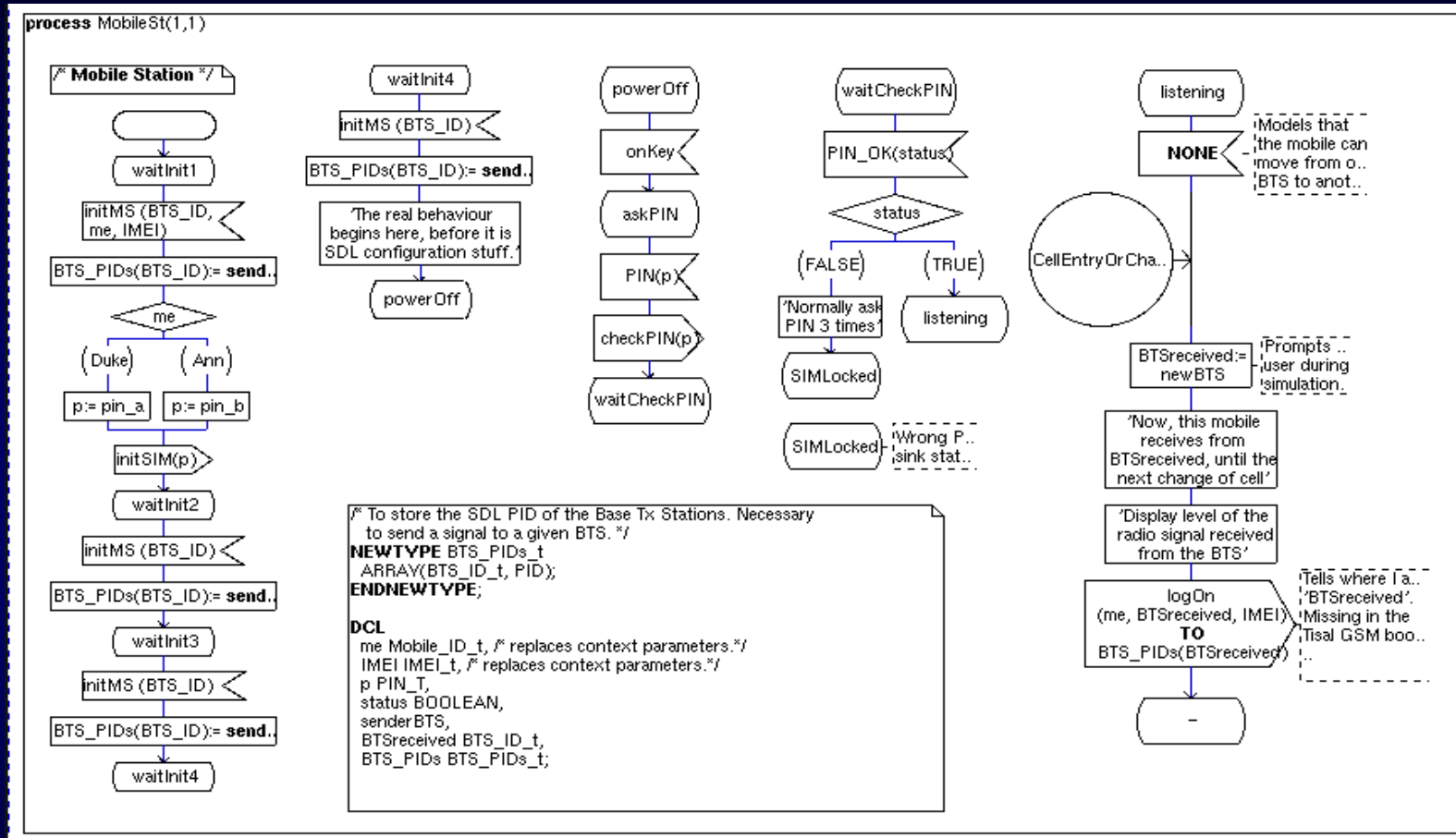


other supporting
tools exist, e.g.
Rational Rose

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

SDL language

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 language

a synchronous programming language for the development of complex reactive systems

The screenshot displays the Esterel Studio development environment. On the left, there are two panels: 'Simulation Output' and 'Simulation Control'. The 'Simulation Output' panel shows a table of variables:

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

The 'Simulation Control' panel shows a table of control variables:

Name	Value	Type
Coin		
MS		

The main workspace shows a state transition diagram for 'ReflexGameNormal'. The diagram includes states like 'MachineON', 'GAME', and 'PAUSE_LENGTH MS/ Display(?MEAN/MEASURE_NUMBER)'. Transitions are labeled with signals such as 'On_off/', 'Coin/', and 'sustain GameOver'. The diagram also shows a signal declaration: 'signal RemainingMeasures:integer, MEAN:integer'.

At the bottom, there are controls for 'Commands' (Tick, Reset, Keep Inputs), 'Playback Session' (Reset on Loading, Speed), and 'Dump control' (Waveform, Output file, Configuration file, Coverage).

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


```

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;
}

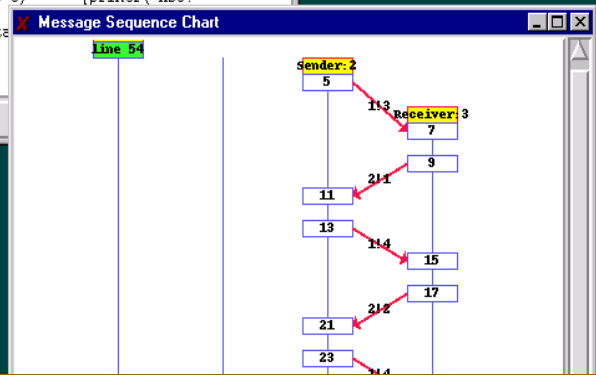
proctype Receiver()
{
  byte any;
again:
  do
    :: receiver?msg1 -> sender!ack1; break
    :: receiver?msg0 -> sender!ack0
    :: receiver?any /* lost */
  od;
P0:
  do
    :: receiver?msg0 -> sender!ack0; break
  od;
}

```

```

) 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

```

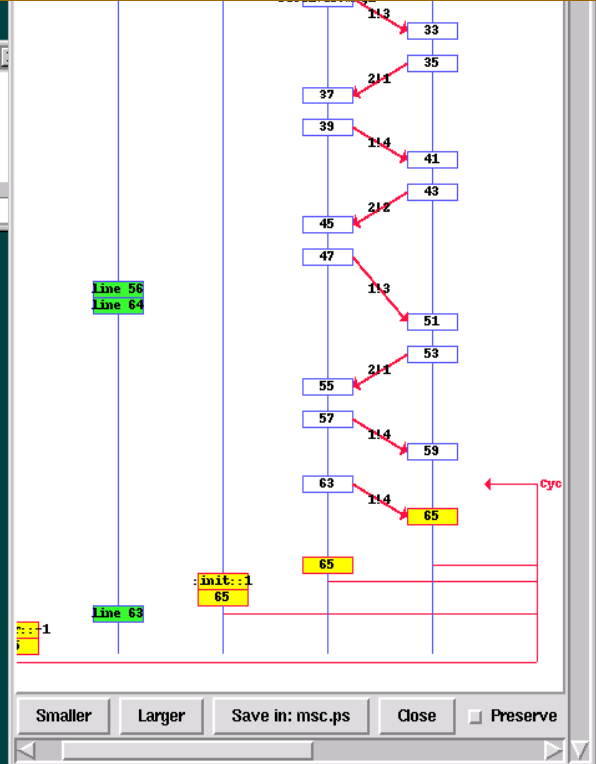


SPIN, by Gerald Holzmann at AT&T

```

<starting simulation>
/pack/PS/Spin.prog/spin-3.13/bin/spin -X -p -v -g -l -s -r -t -j0 pan_in
<at end of trail>

```



```

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)

```

Ghost View

madcow

Netscape Communicator

Internet9809...

FskCentOpti...

Simple FSM modeling and model manipulation

Example: Bank-box Code



- Orange
- Blue
- Yellow

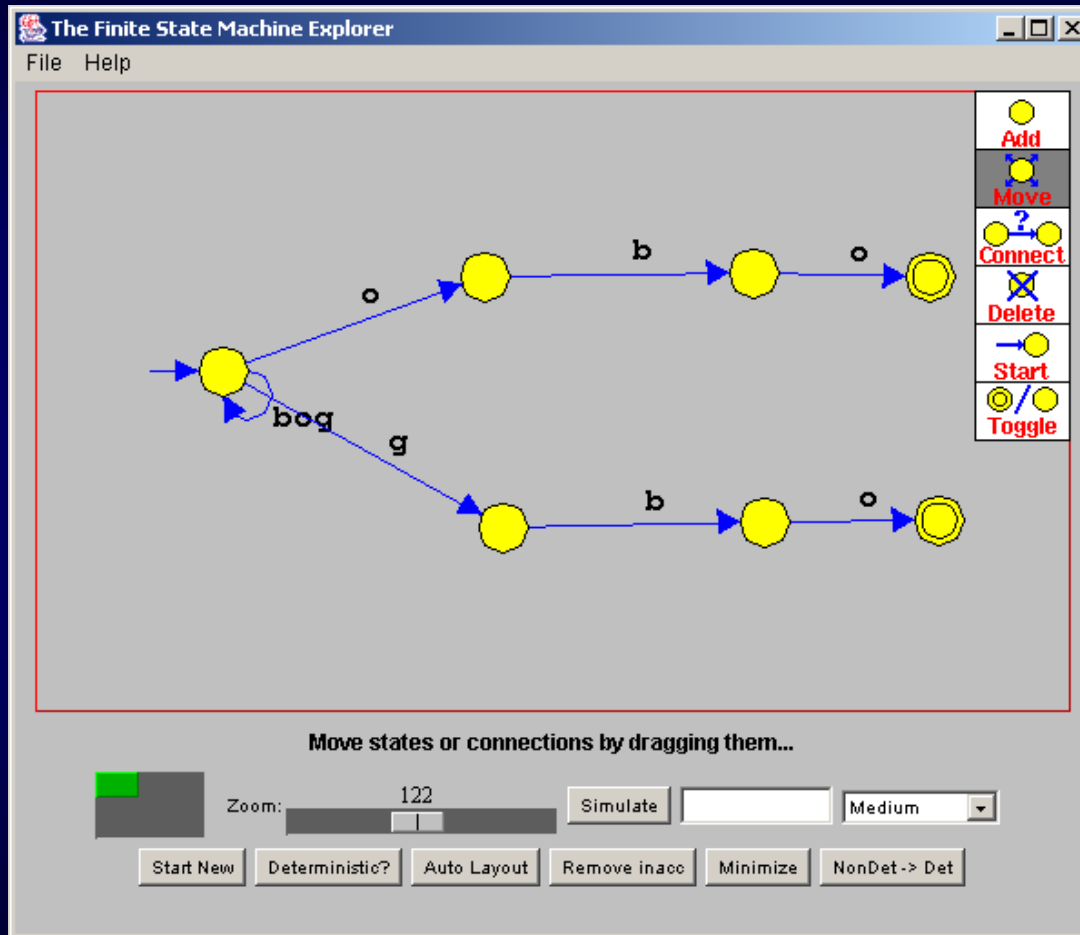
- (1) To open a bank box
the code must contain **at least 2** ●
- (2) To open a bank box
the code must **end with** ●●●
- (3) To open a bank box
the code most end with ●●●
or with ●●●
- (4) To open a bank box
the code must end with a **palindrom**
e.g.: ●●●, or
●●●●, or
●●●●●

.....

Palindrome: A word that reads the same forth and back, e.g., madam, radar, etc.

Tool: The Finite State Machine Explorer

Freely available (<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)

Note:

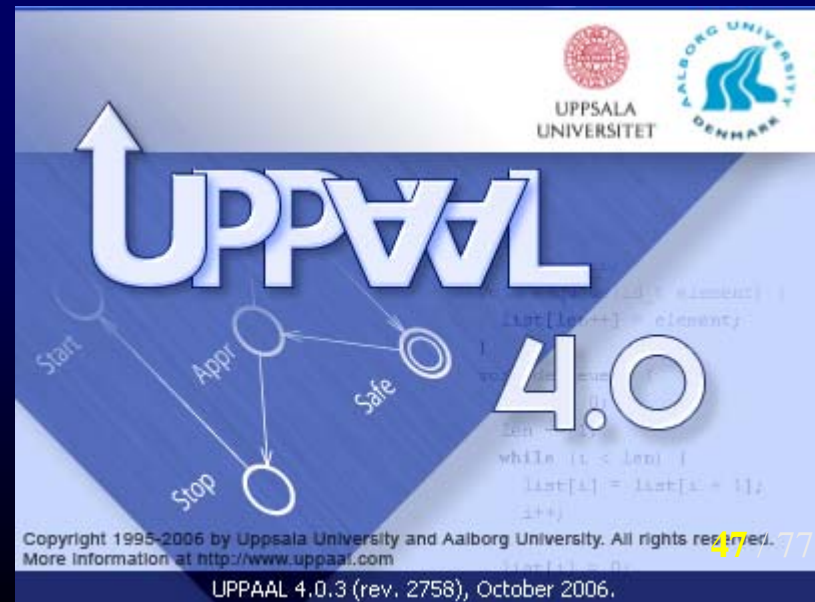
- An arbitrary palindrome is not recognizable by FSM: consider infinitely many/long palindromes
- FSM can recognize a given bank-box opening sequence.
- If non-deterministic:
 - determinize it → minimize it (using the FSME tool)

FSM modeling and simulation using Uppaal

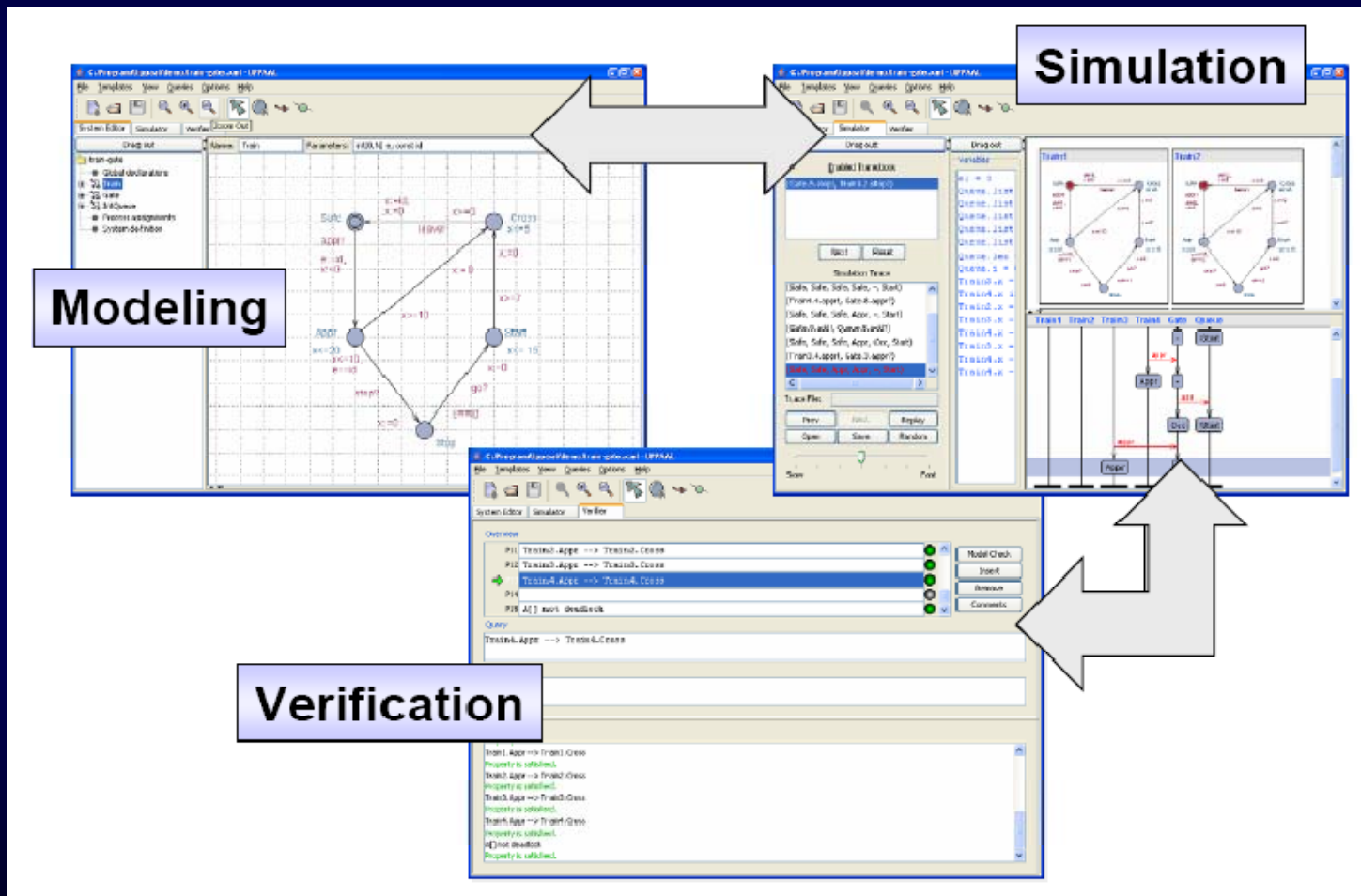
Uppaal

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

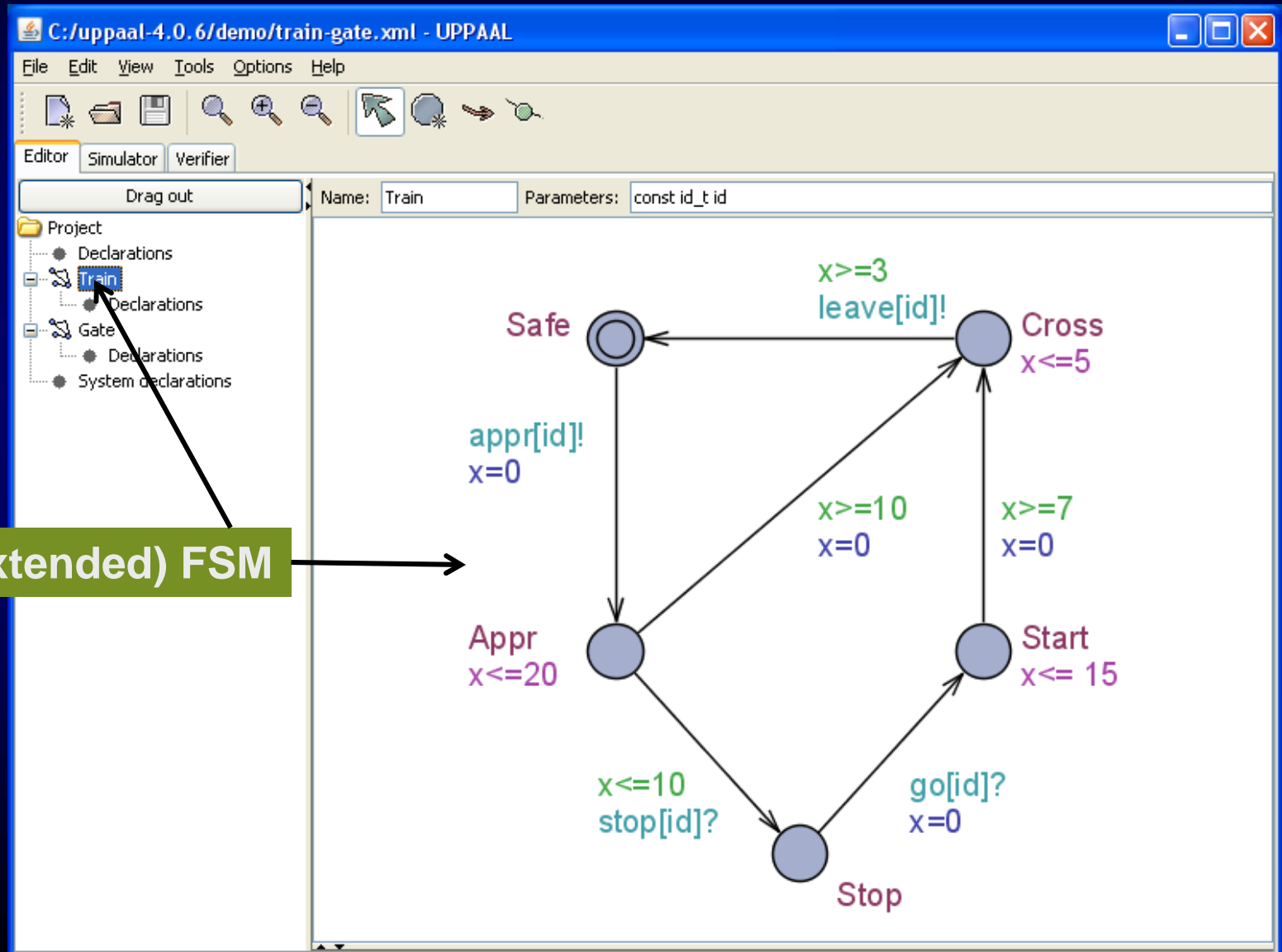
Freely available
(<http://www.uppaal.com>)



Working Modes of Uppaal



Uppaal Editor (Modelling View)

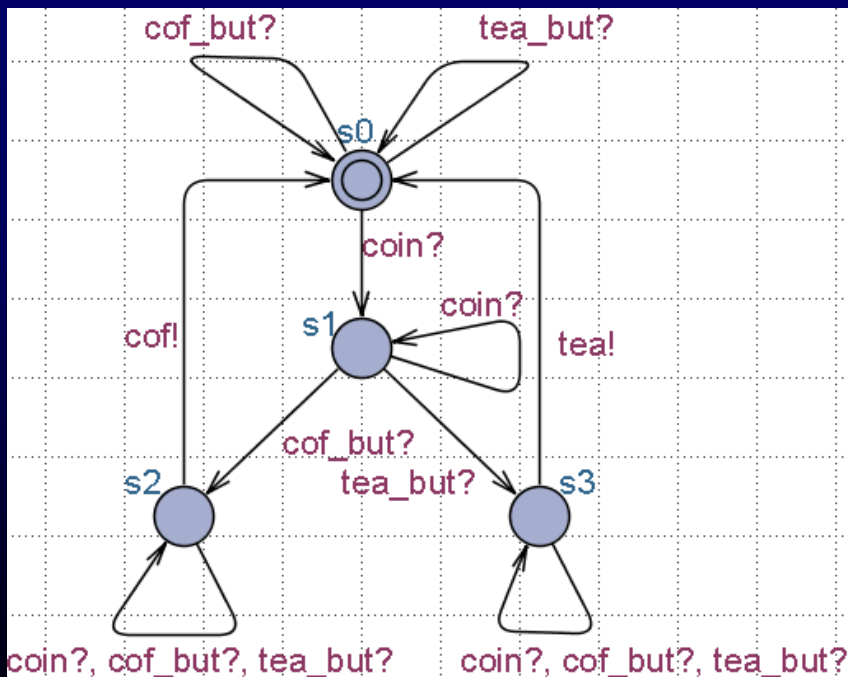


FSM in Uppaal

- 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)

But **not** a real FSM. Because in Uppaal model, each edge is an **atomic** transition, and it does **not** take the form "input/output"



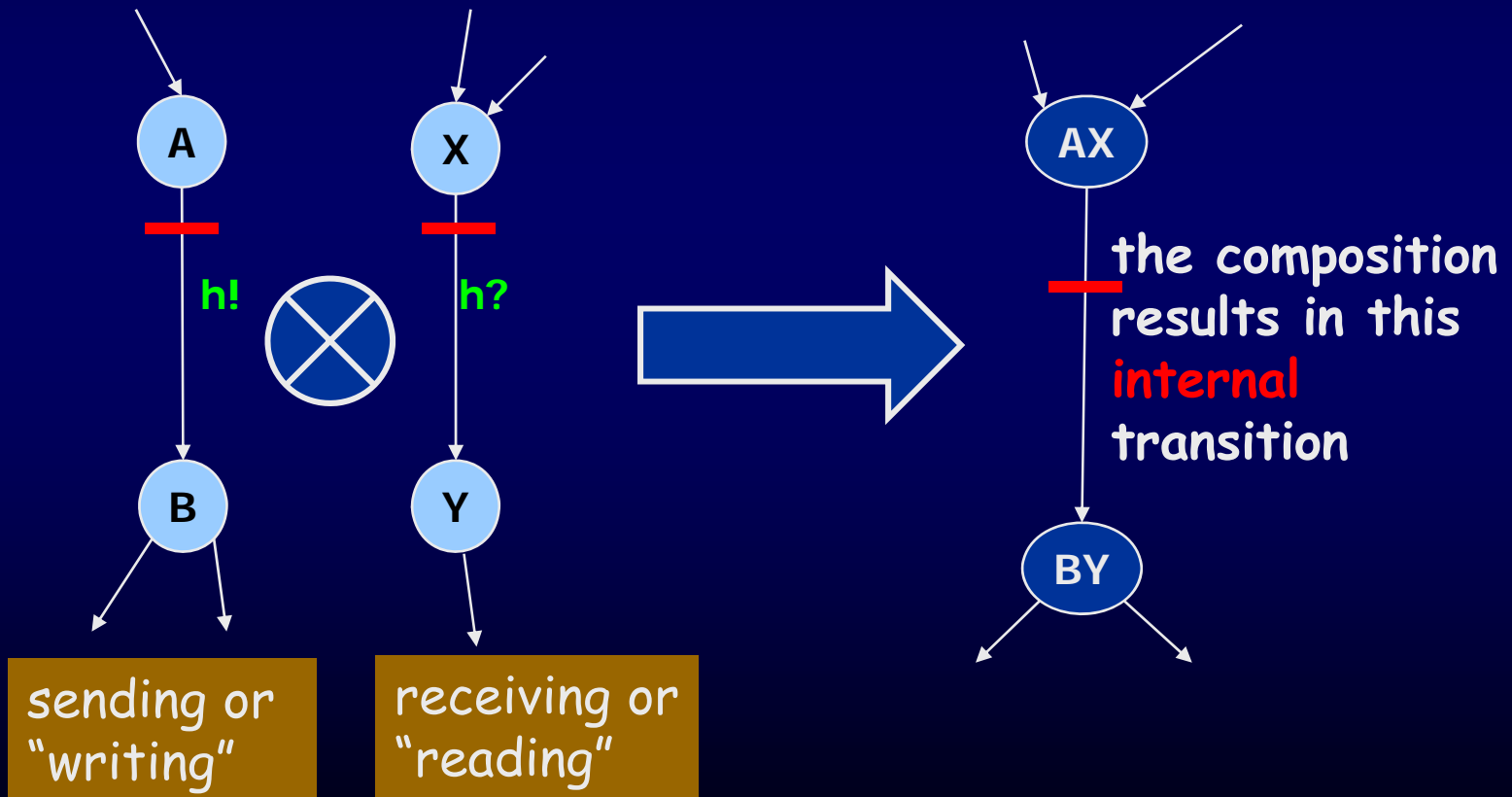
coffee vending machine

Model Composition

IO Automata (2-way synchronization)

or pairwise synchronization

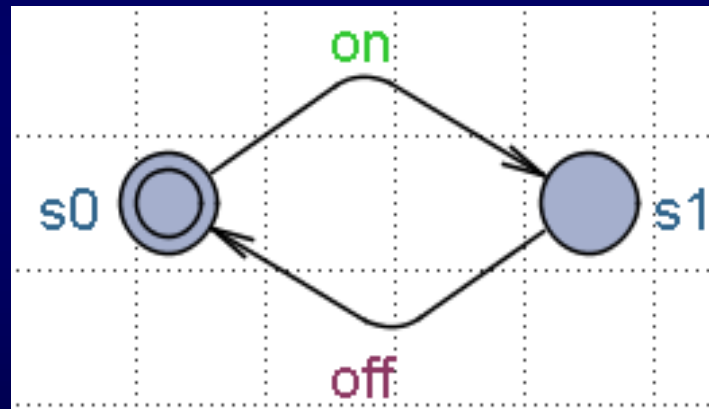
or "handshaking"



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.

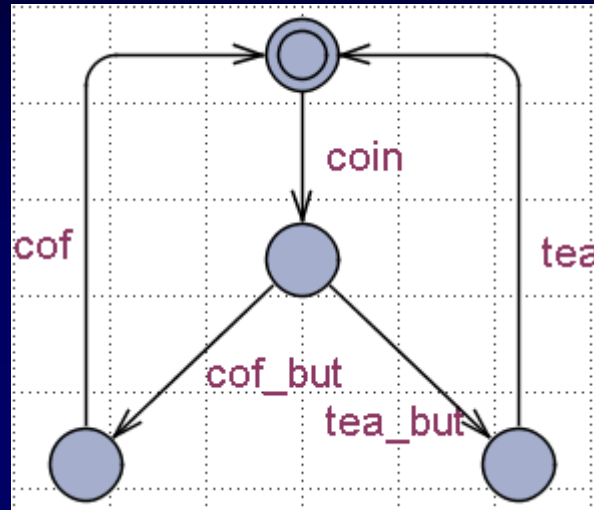
a light switch
LTS



on→off→on→off→on→off→

a sequence of actions
or
a *trace*

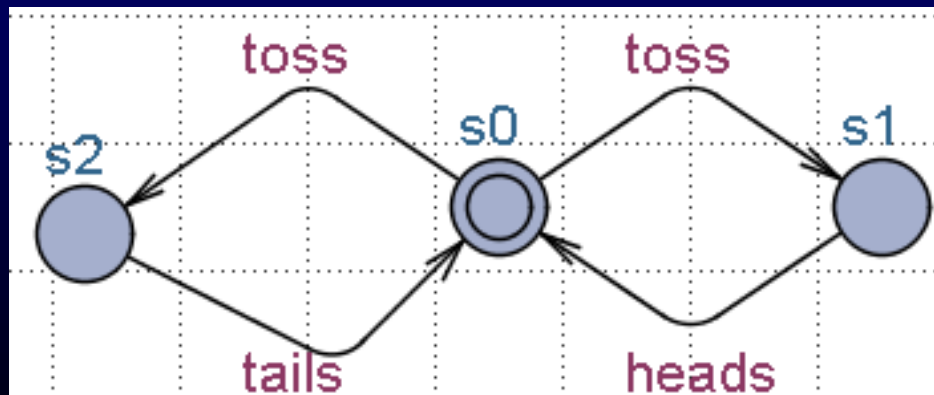
Modelling Choices



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

Non-deterministic Choice: modeling random event

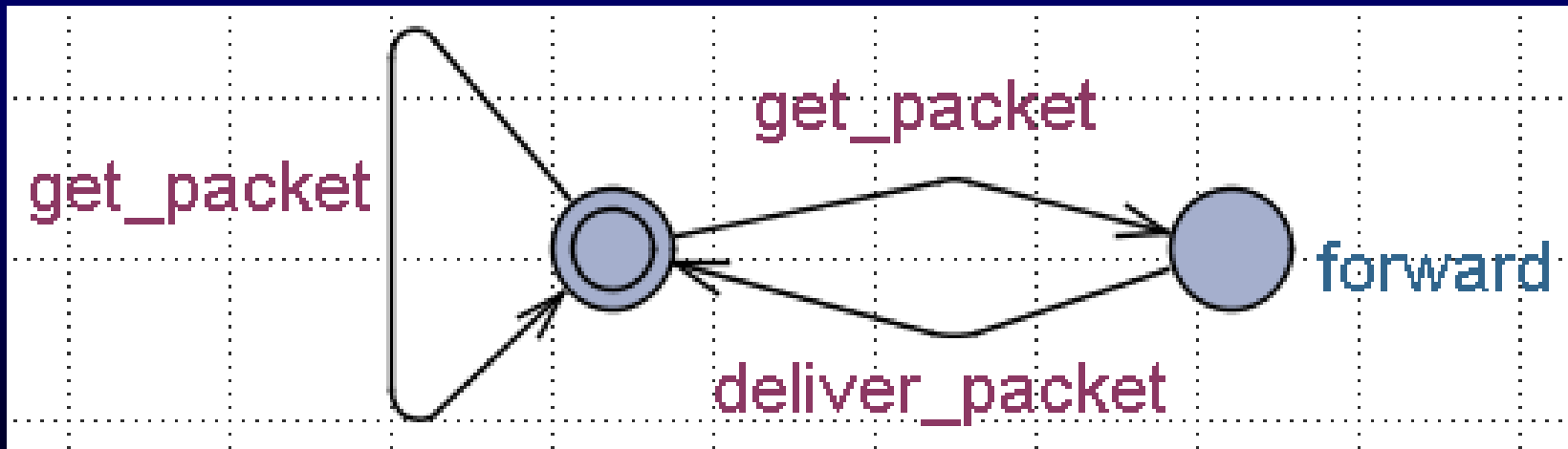
- Tossing a coin
- Possible traces?
 - Both outcomes (head or tail) possible
 - Nothing said about relative frequency
 - If coin is fair, the outcome is 50/50



Non-deterministic Choice: modeling failure

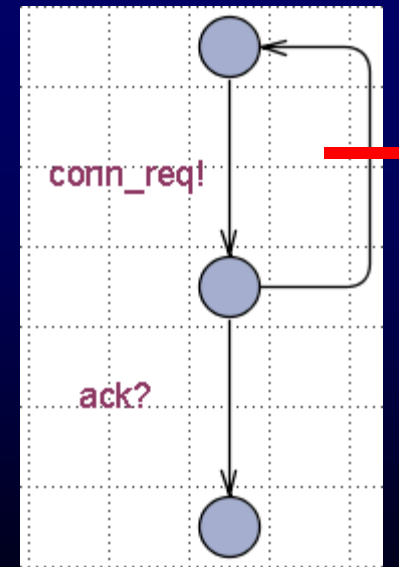
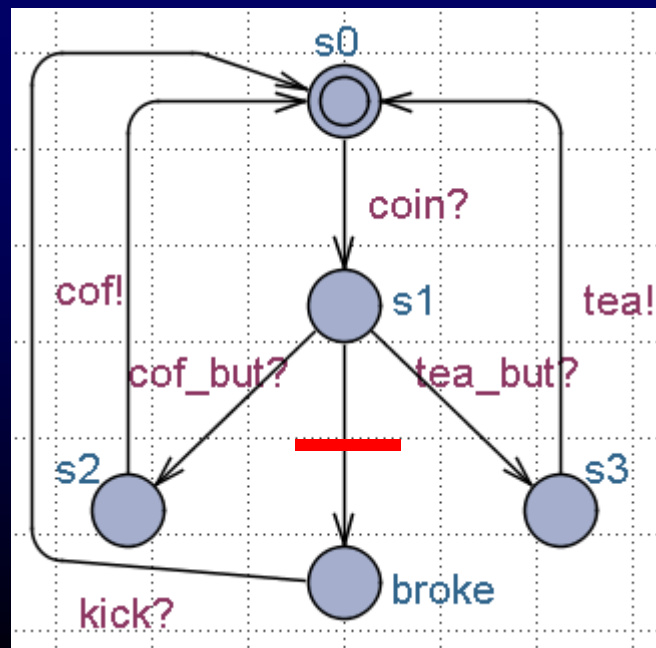
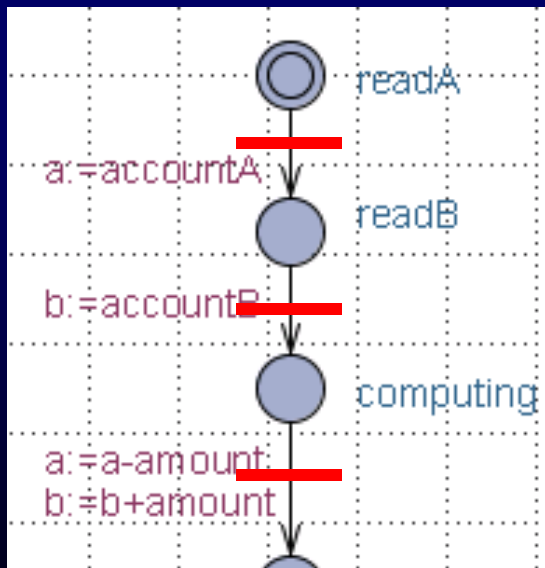
How do we model an **un**reliable communication channel which **accepts** packets, and if a failure occurs **produces no output**, otherwise **delivers** the packet to the receiver?

Use non-determinism...

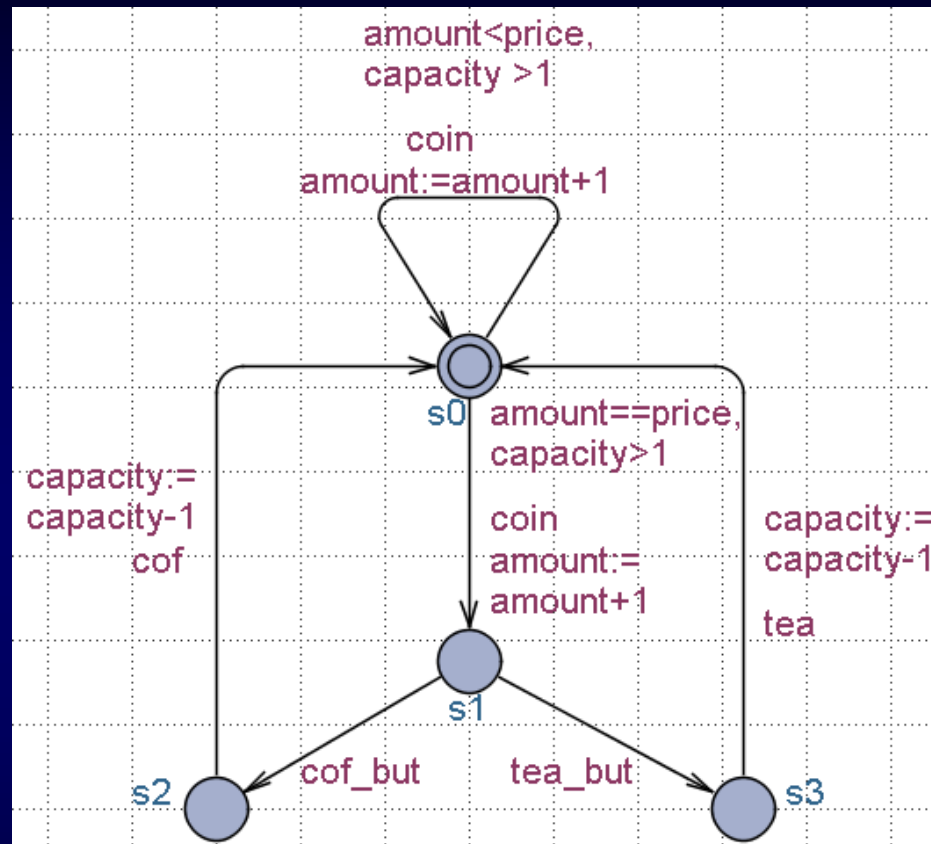


Internal Actions

- Internal actions are 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 (thus not really an EFSM!)
- Can be translated into FSM if variables have bounded domains
- State: control location + variables' valuation
- (state, total, capacity), e.g.: (s0, 5, 10)

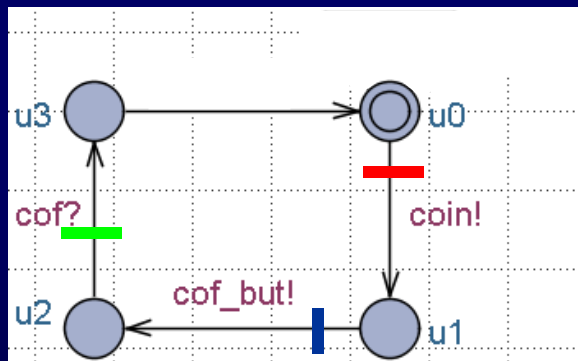
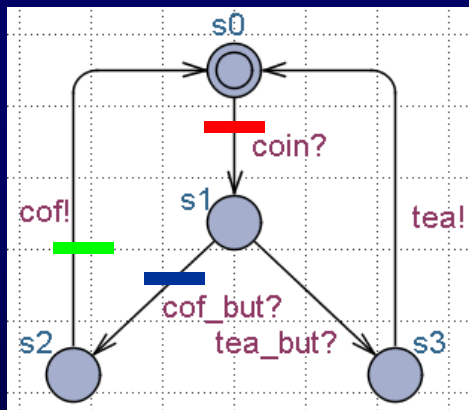
Process Interaction

- "!" denotes output, "?" denotes input
- Handshake communication 
- Two-way

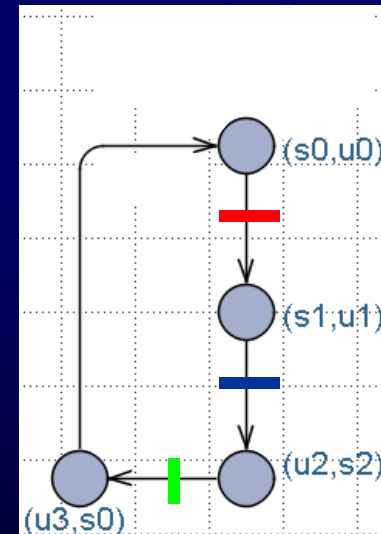
Coffee Machine

Lecturer

University =
Coffee Machine || Lecturer



=



4 states

4 states

synchronization results in internal actions

LTS?
How many states?
Traces?
4 states: 58 / 77

(interactions constrain overall behavior)

Broadcasts

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

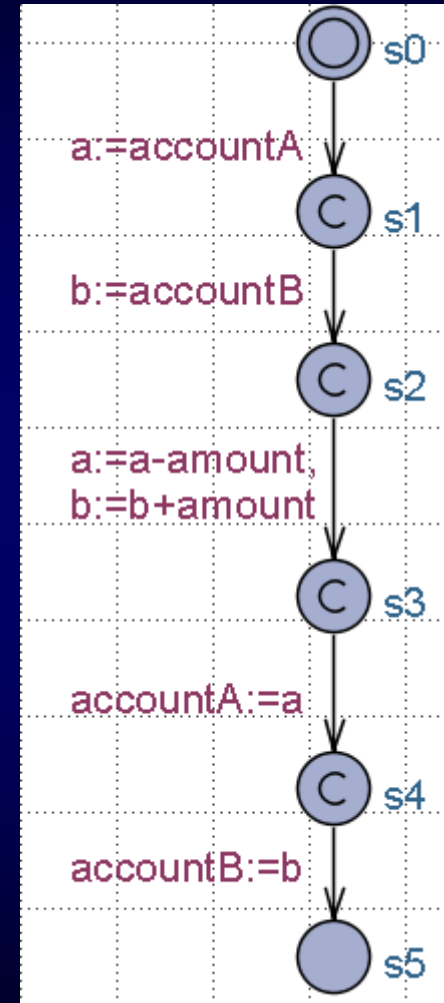


- the sending party: one automaton outputs **join!**
- the receiving party: several automata accept **join!**,
 - each of them makes a move upon receiving **join!**,
 - ie. every automaton with enabled "**join?**" transition moves in one step
- the number of recipients may be **0** (one "speaker", but **zero** "audience")

Committed Locations

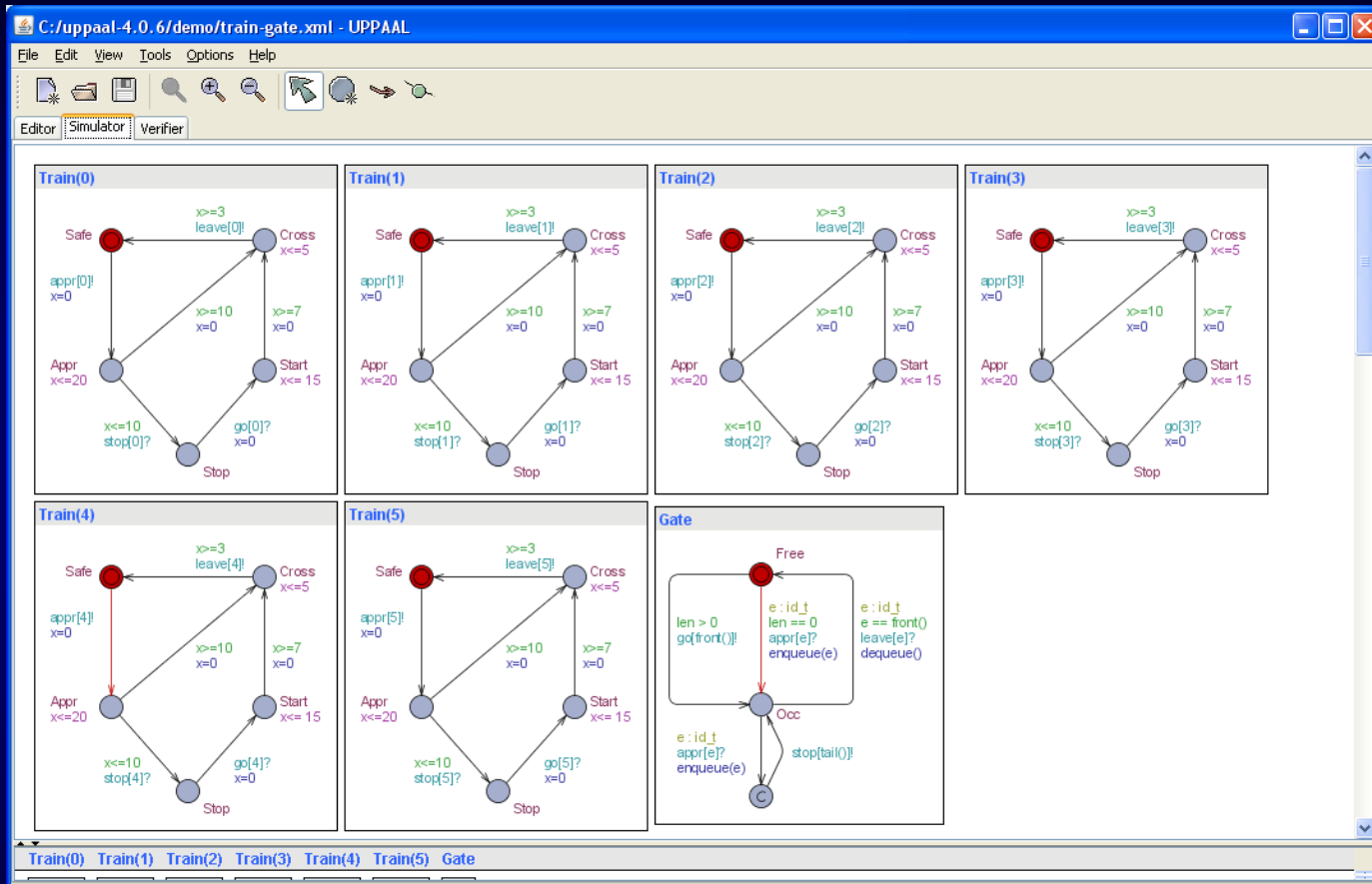
- Locations marked "C"
 - *No delay* in committed location
 - Next transition must involve one of those automata in *committed locations*
- Handy to model atomic sequence of actions
 - An "input/output"-style transition of Mealy machine can be modelled by 2 atomic actions "input?" and "output!", which 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

Committed locations help **regain** the FSM expressiveness of Uppaal models.



s0 to s5 executed atomically
they will not be interrupted

Uppaal Network of Automata



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

They constitute a **closed** system.

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`.
- Enabled Transitions:** A list of transitions that are currently enabled in the simulation. The selected transition is `(e.2.speed!, sc.6.speed?)`. Other transitions include `(cc.4.recordSpeed!, sc.8.recordSpeed?)`.
- Simulation Trace:** A list of events that have occurred during the simulation. The selected event is `(-, on_requested, disabled, -, -)`. Other events include `(u.1.engineOn!, cc.1.engineOn?)`, `(cc.2.clearSpeed!, sc.7.clearSpeed?)`, and `(e.2.speed!, sc.6.speed?)`.
- Variables:** A list of variables and their current values: `eOn = 1`, `brk = 0`, `throttleControl = 0`, and `time = 0`.
- State Machine Diagram:** A state transition diagram for the `cc` component. States include `inactive`, `active`, `on_requested`, and `enable Control!`. Transitions are labeled with events and guards, such as `engine On?`, `clearSpeed!`, `on?`, `throttle Control = 0`, `enable Control!`, `throttle Control = 0`, `resume?`, `accelerator? throttle Control = 0`, `engine Off?`, `engine Off?`, `disable Control!`, and `record Speed!`.
- Sequence Diagram:** A sequence diagram showing the interaction between components `u`, `cc`, `sc`, `e`, and `ufix`. Messages include `engineOn`, `clearSpeed`, `on`, `on_requested`, and `speed`.

At the bottom right, there is a speed control slider ranging from `Slow` to `Fast`. The page number `62 / 77` is visible in the bottom right corner.

Model Simulation in Uppaal

the enabled transitions that the user can choose

the system state variables

graphical display of the current system state

The screenshot displays the Uppaal Model Checker interface for a simulation. The window title is "C:/uppaal-4.0.6/demo/train-gate.xml - UPPAAL". The interface is divided into several panels:

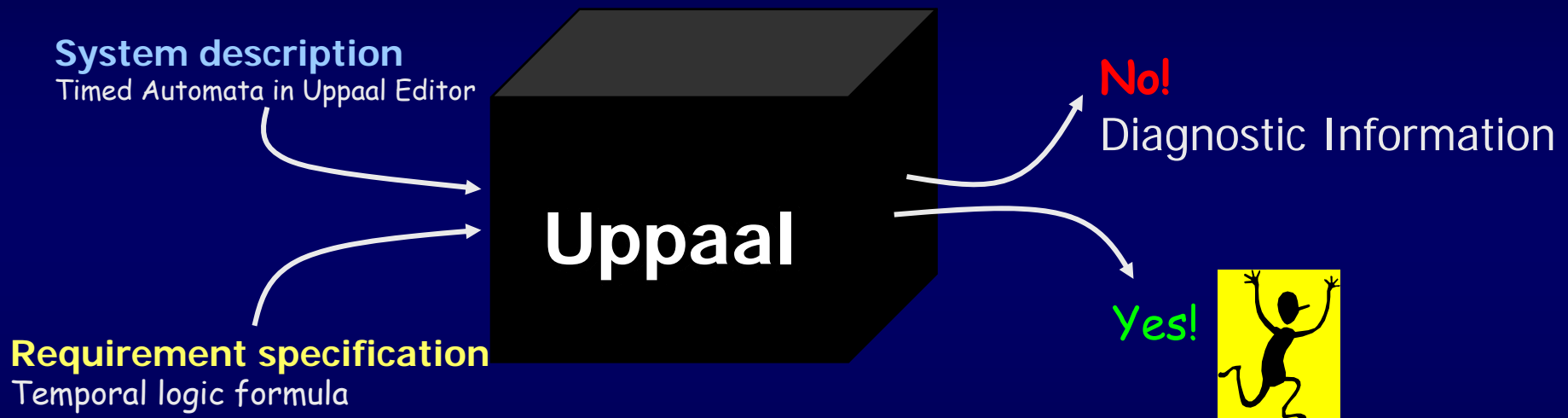
- Enabled Transitions:** A list of transitions available for selection, including "Train(0) appr[1]: Train(1) --> Gate", "Train(2) appr[2]: Train(2) --> Gate", "Train(4) appr[3]: Train(4) --> Gate", and "Train(5) appr[5]: Train(5) --> Gate".
- Simulation Trace:** A log of events such as "(Safe, Safe, Safe, Safe, Safe, Free)", "(Appr, Safe, Safe, Safe, Safe, Occ)", and "(Appr, Safe, Safe, Stop, Safe, Safe, Occ)".
- System State Variables:** A list of variables and their values, including "Gate.list[0] = 0", "Gate.list[1] = 3", "Gate.list[2] = 0", "Gate.list[3] = 0", "Gate.list[4] = 0", "Gate.list[5] = 0", "Gate.len = 2", and "Train(0).x in [0,20]".
- Component State Diagrams:** Six diagrams for Train(0) through Train(5) and the Gate component, each showing a state machine with nodes like "Safe", "Cross", "Start", "Appr", "Stop", and "Occ", and transitions labeled with conditions and actions.
- Message Sequence Chart (MSC):** A diagram at the bottom showing the sequence of messages between components: "Safe", "Appr", "Occ", and "Free". Red arrows indicate the flow of messages, such as "appr[0]" from Train(0) to the Gate, and "appr[5]" from Train(5) to the Gate.

history information as trace

history information on component interaction as documented in Message Sequence Chart

System verification using Uppaal

Uppaal Model Checking as a box



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

path quantifier

state quantifier

• $A[] p$

• $A<> p$

• $E<> p$

• $E[] p$

• $\underline{P \dashrightarrow q}$

process location

data guards

clock guards

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

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

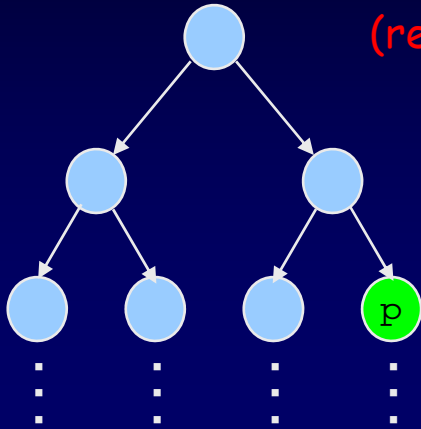
$A[] (\text{mc1.finished and mc2.finished}) \text{ imply } (\text{accountA+accountB}=200)$

Uppaal "Computation Tree Logic"

$E \langle \rangle p$

"possible"

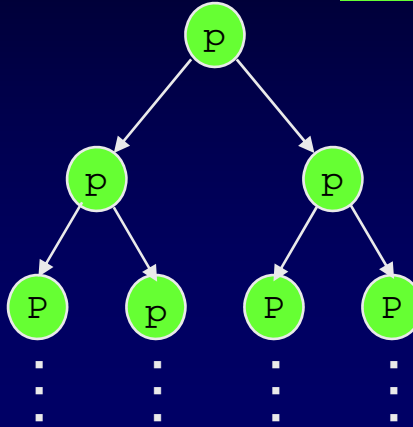
(reachability)



$A [] p$

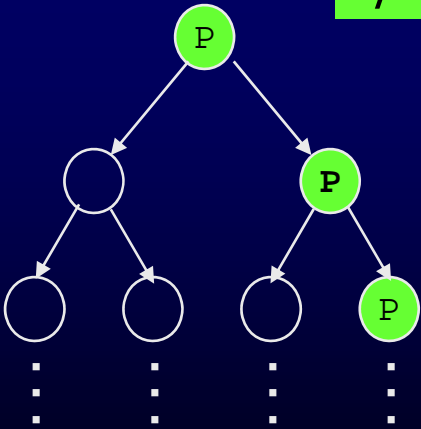
"always"

(safety)



$E [] p$

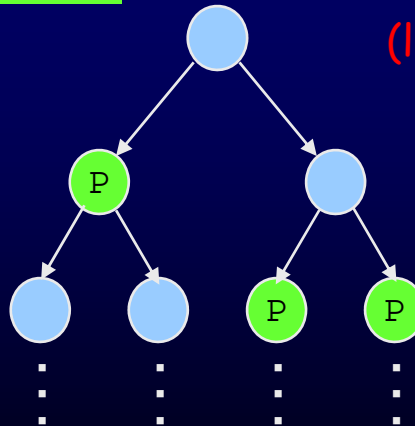
"potentially always"



$A \langle \rangle p$

"inevitable"

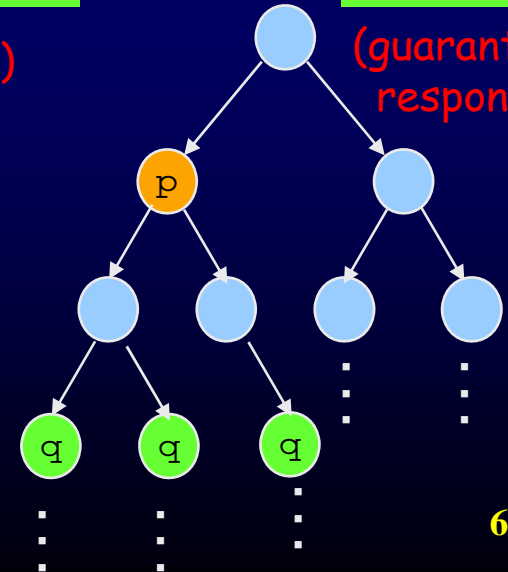
(liveness)



$p \dashrightarrow q$

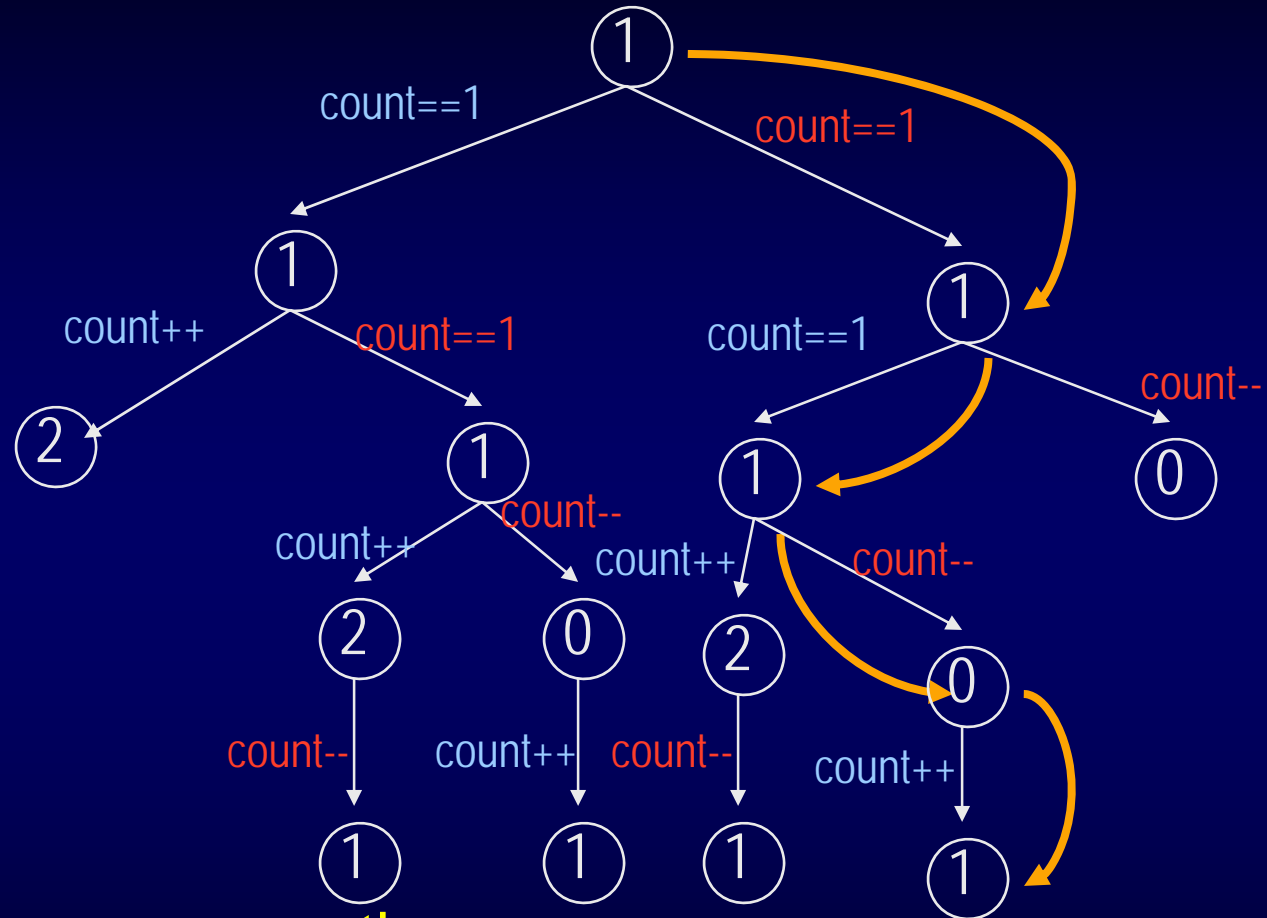
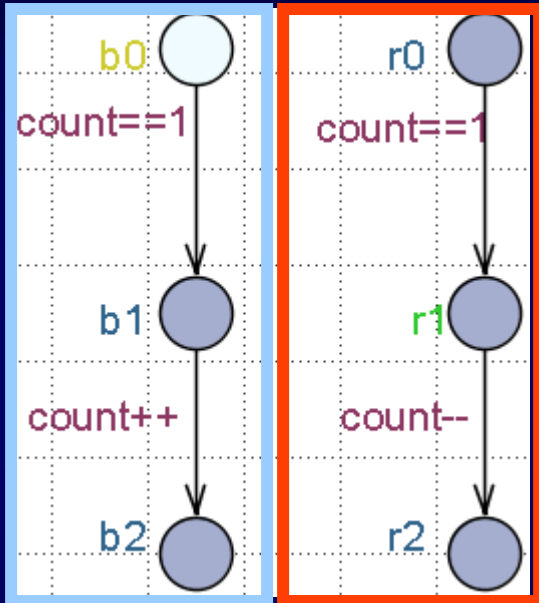
"leads-to"

(guaranteed response)



State Space Exploration

Int count := 1



- Each trace = a program execution

- Uppaal checks *all* traces

- Is count *possibly* 3 ?

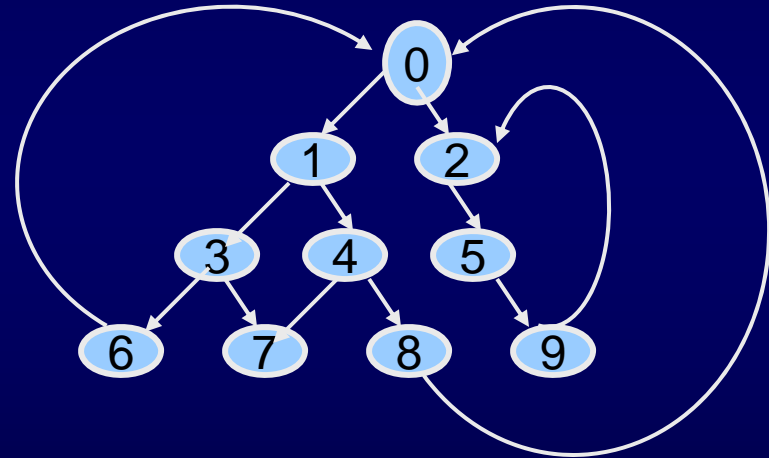
E<> count==3

- 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
}
```



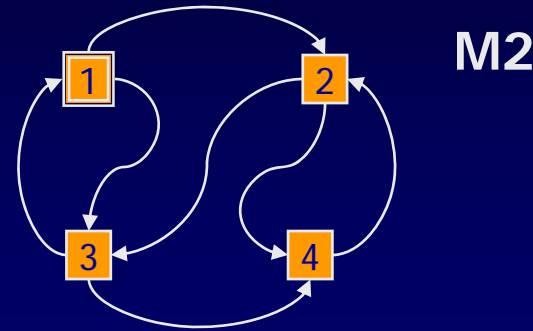
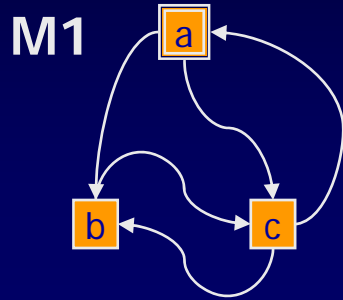
Depth-First: maintain waiting as a **stack**

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

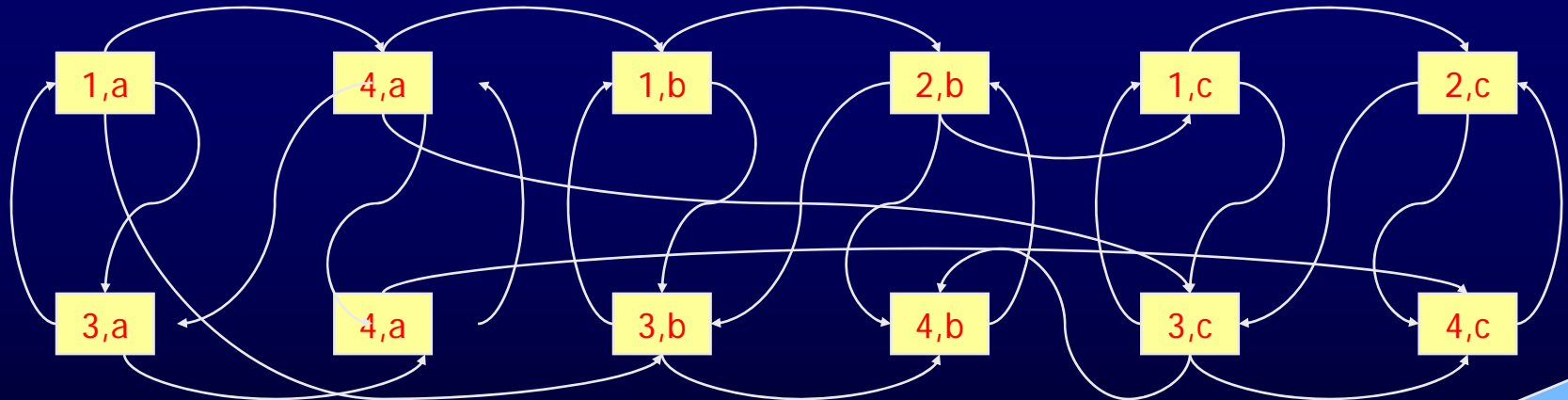
Breadth-First: maintain waiting as a **queue**
(shortest counter example)

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

'State Explosion' problem



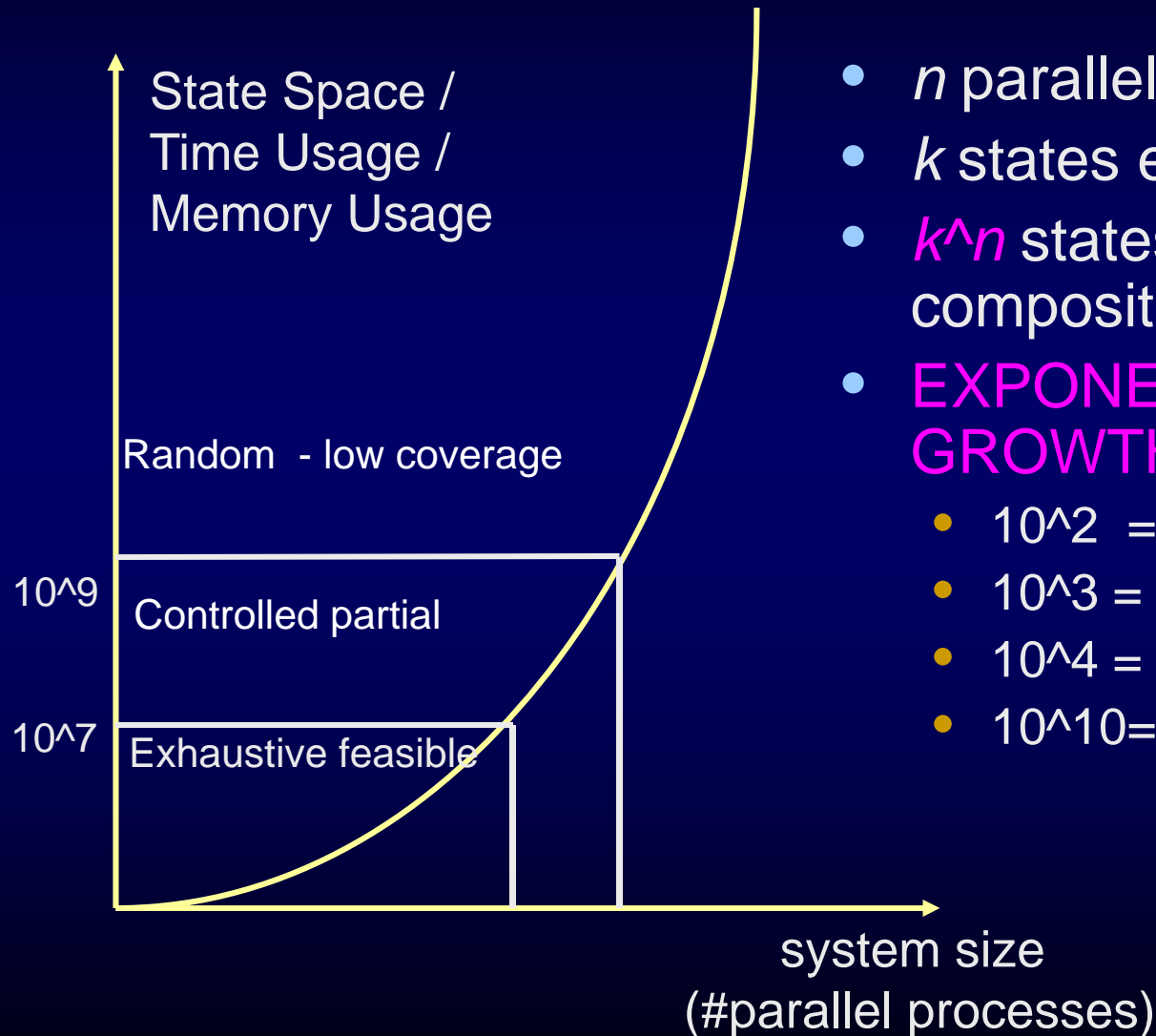
M1 x M2



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

Any other remedy?
- Model-Based Testing!