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,
...

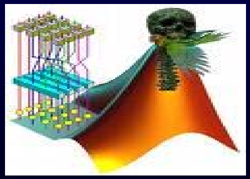
Reactive

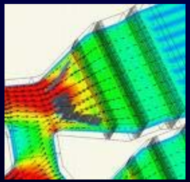
- embedded systems (e.g., in consumer electronics, mobile phones, GPS)
- operating systems
- communication protocols
- web servers
- air traffic control
- computer games
- ...



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

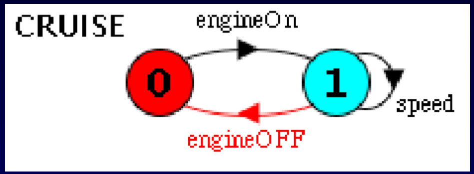
- Informal model
  - Documentation in prose, schematic block diagrams, etc.
- Semi-formal model
  - Unified Modeling Language (UML) diagrams
- Formal model
  - Well-formed mathematical models, usually with clearly defined syntax and semantics

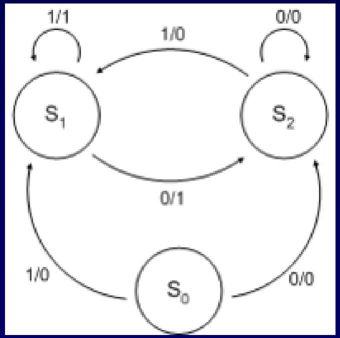
Formal Models Classfied

- 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)

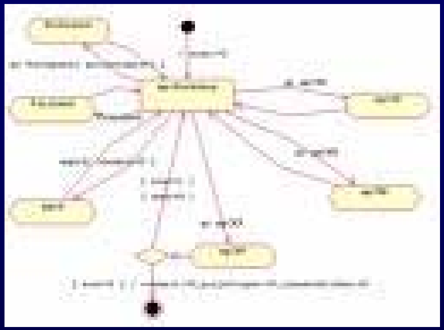
State Transition-Based Models

- LTS (Labeled Transition Systems)

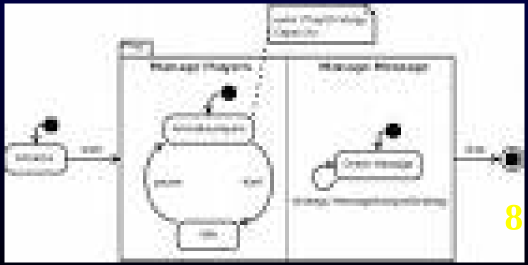




- FSM (Finite State Machines)
- EFSM (FSM + guards + assignments)



- 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

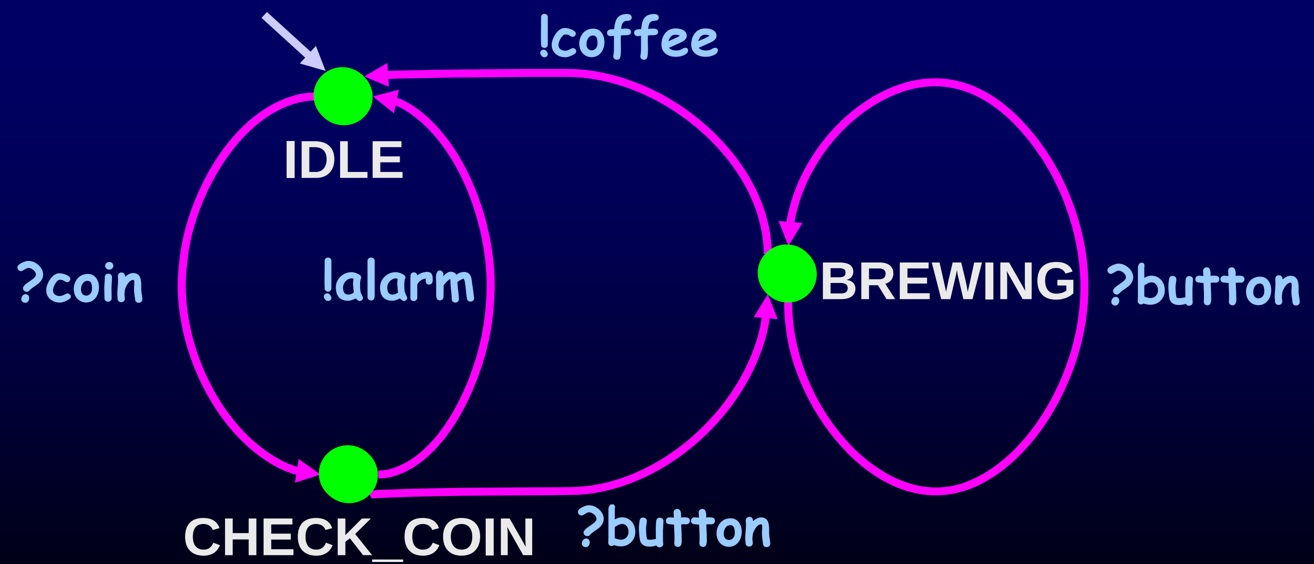
transitions

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

initial state

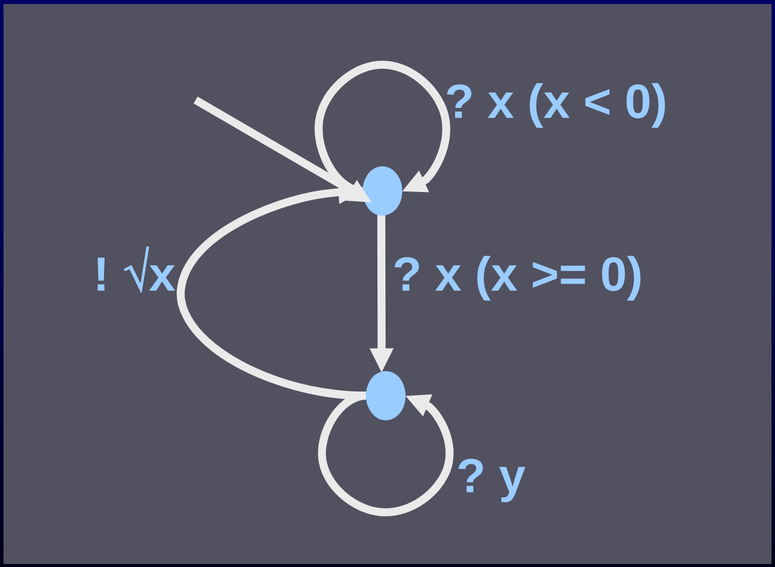
$s_0 \in S$

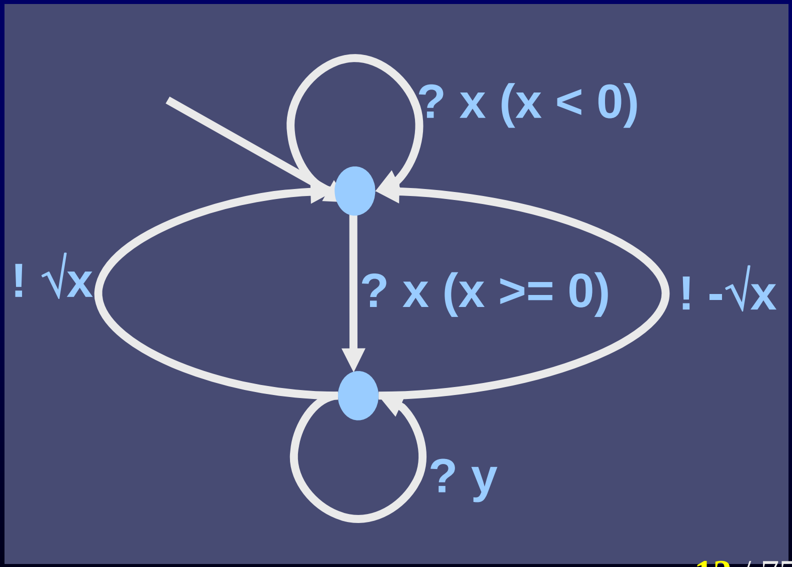
coffee vending machine



Input-Output LTS (IOLTS)

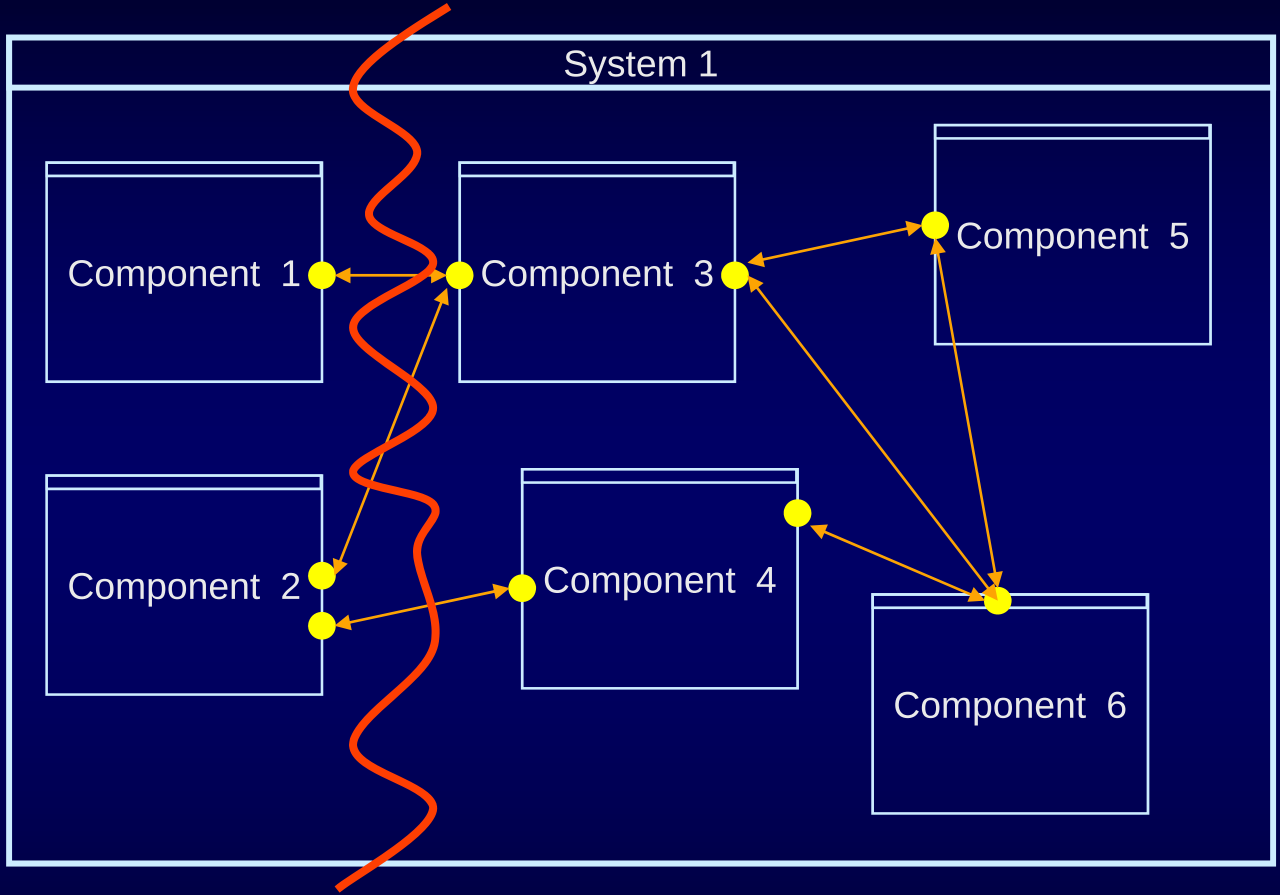
- Special kind of LTS:
  Input-Output Labelled Transition System - IOLTS
  - Output actions(!), and input actions(?)
- IOLTS with variables - equation solver for $y^2 = x$:
  - different implementations





Finite State Machine (FSM)

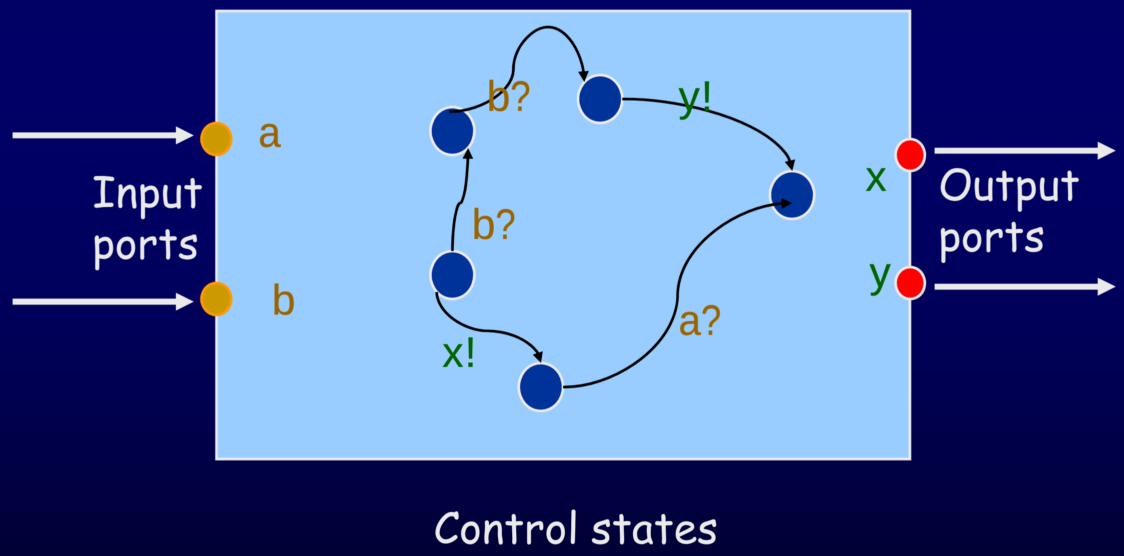
System Structure



- 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?

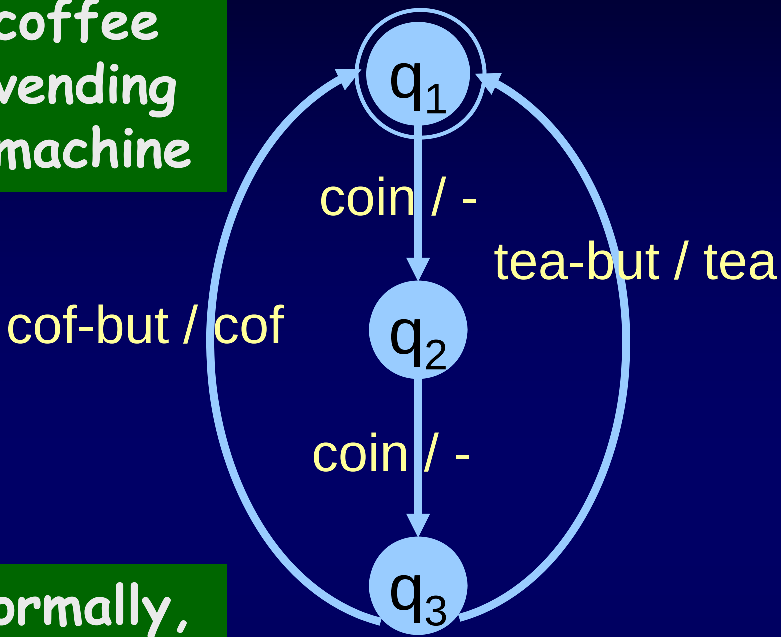
Behavior of a Component

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: $\{q_1,q_2,q_3\}$
Initial state = q_1
Transitions= {
(q_1, coin, -, q_2),
(q_2, coin, -, q_3),
(q_3, cof-but, cof, q_1),
(q_3, tea-but, tea, q_1)
}

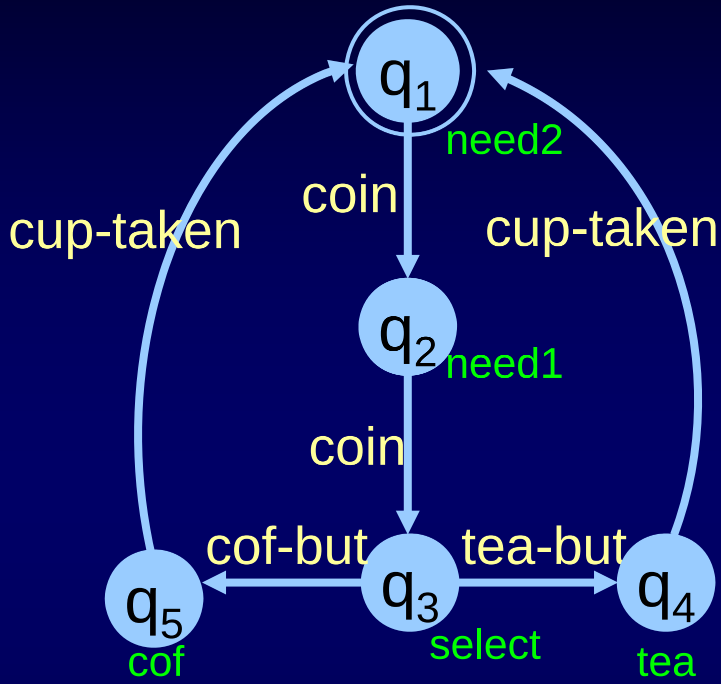
In Mealy machine the **output** depends on the **current state** as well as the **input**

Sample run:

$q_1 \xrightarrow{\text{coin/ -}} q_2 \xrightarrow{\text{coin/-}} q_3 \xrightarrow{\text{cof-but / cof}} q_1 \xrightarrow{\text{coin/ -}}$

$q_2 \xrightarrow{\text{coin/ -}} q_3 \xrightarrow{\text{cof-but / cof}} q_1$

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

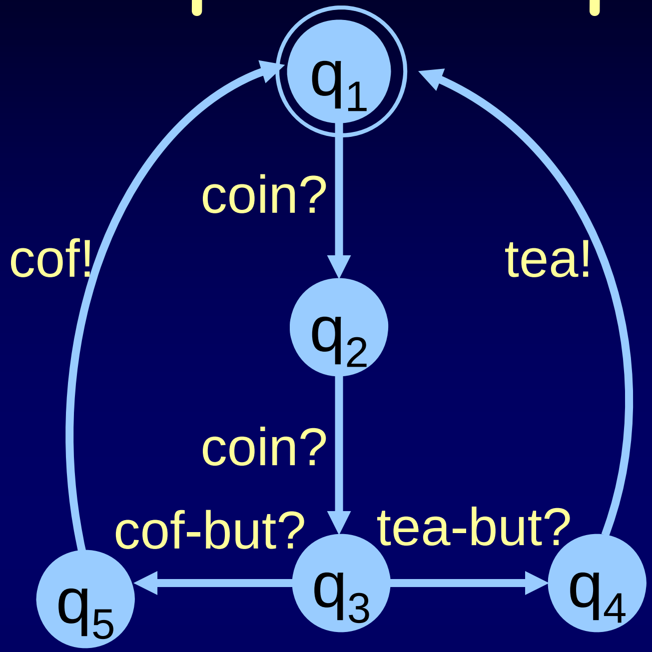
Input sequence: coin.coin.cof-but.cup-taken.coin.cof-but
Output sequence: need2.need1.select.cof. need2.need1.select.cof

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

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

Inputs = {cof-but, tea-but, coin}
Outputs = {cof,tea}
States: $\{q_1,q_2,q_3\}$
Initial state = q_1
Transitions= {
$(q_1, coin, q_2)$,
$(q_2, coin, q_3)$,
$(q_3, cof\text{-}but, q_5)$,
$(q_3, tea\text{-}but, q_4)$,
(q_4, tea, q_1),
(q_5, cof, q_1)
}

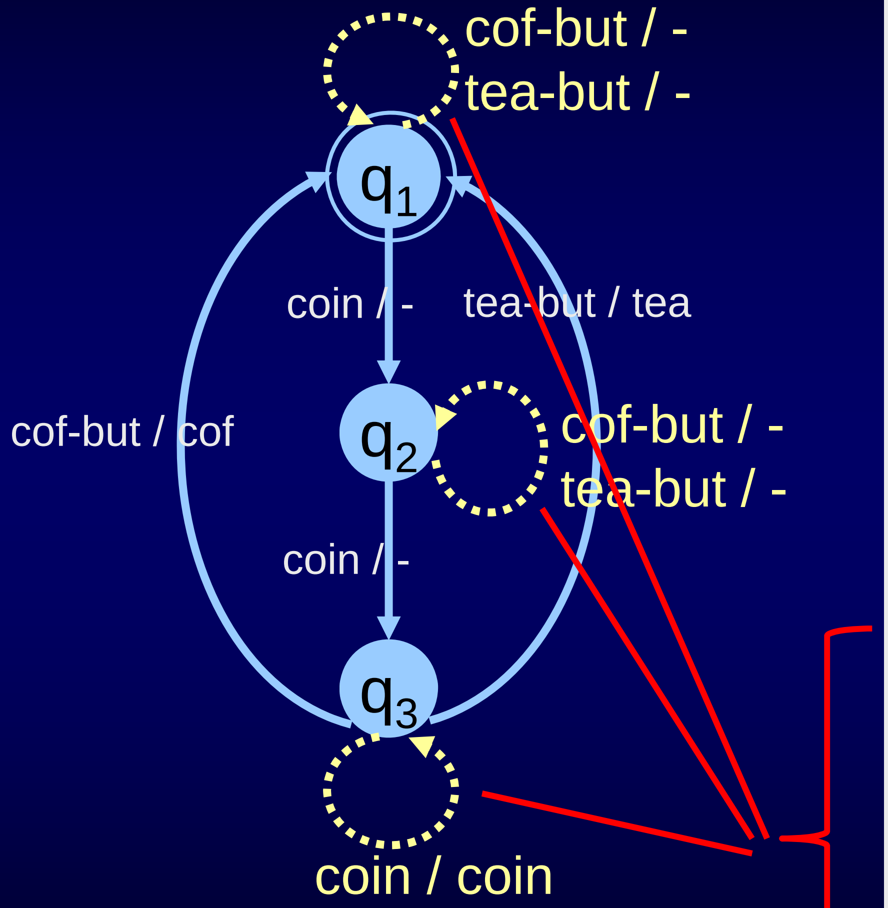
Sample run:

$$q_1 \xrightarrow{\text{coin?}} q_2 \xrightarrow{\text{coin?}} q_3 \xrightarrow{\text{cof-but?}} q_5 \xrightarrow{\text{cof!}}$$

$$q_1 \xrightarrow{\text{coin?}} q_2 \xrightarrow{\text{coin?}} q_3 \xrightarrow{\text{cof-but?}} q_5 \xrightarrow{\text{cof!}} q_1$$

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

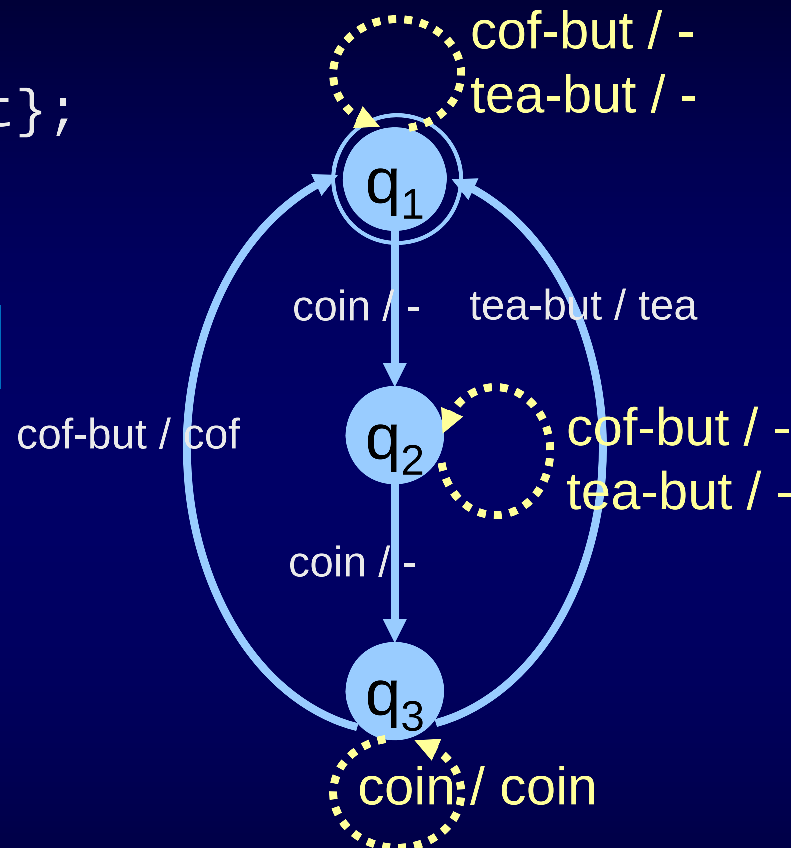
for each state
  for each input
    ...

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 };

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

q1 -- coin/- --> q2

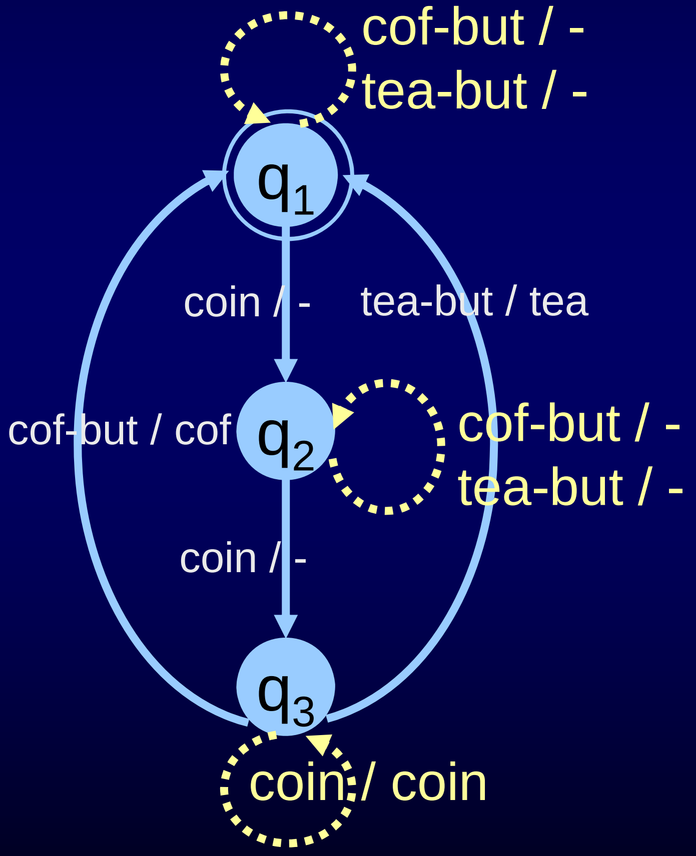



```
// skeleton algorithm
While(input = waitForInput()) {
  OUTPUT(outputTable[currentState,input])
  currentState:=nextStateTable[currentState,input];
}
```

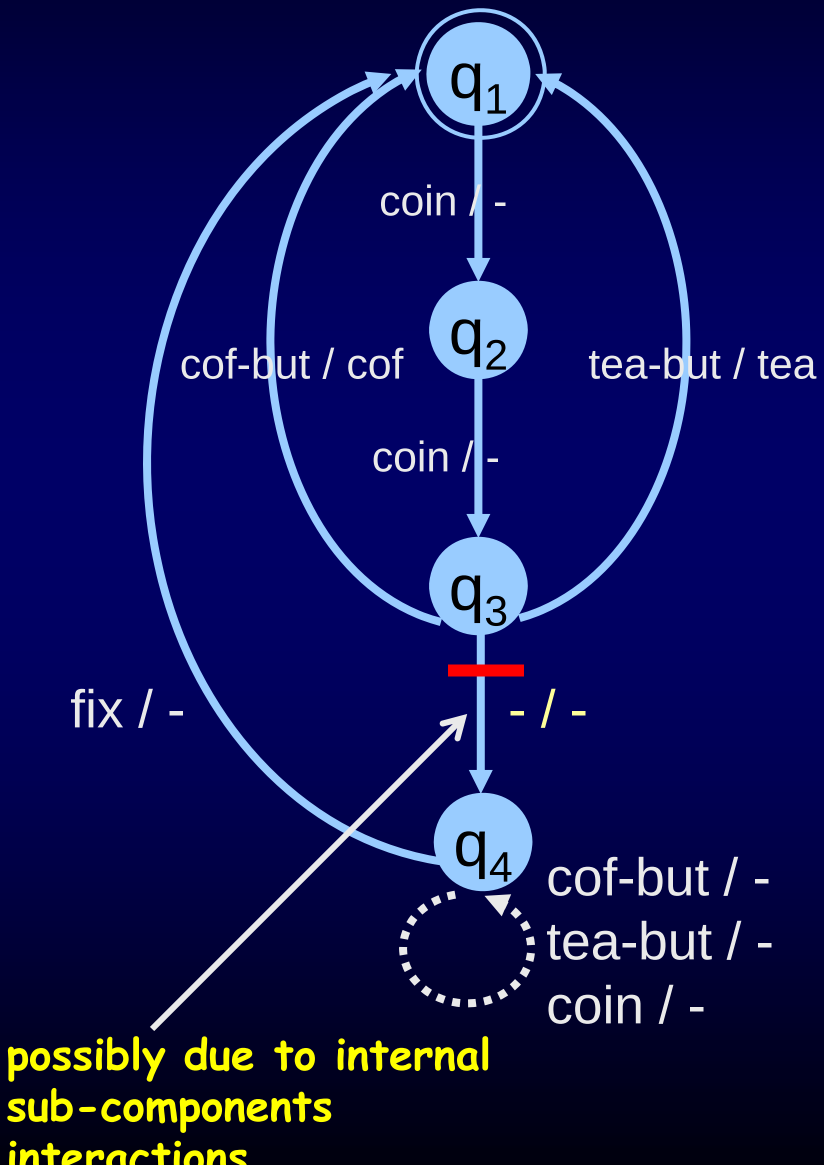
Implementing Mealy FSM (2)

```
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 cuf_but:
        case tea_but: break;
        default: ERROR(”Unexpected Input”);
      }
      break;
  case q2: ...
  case q3: {
      switch(input) {
          case cof_buf: {currentState:=q3;
                         OUTPUT(cof);
                         break;}
          … }
      break;
  default: ERROR(”unknown currentState”);
  } // end of switch
}
```



Spontaneous Transitions



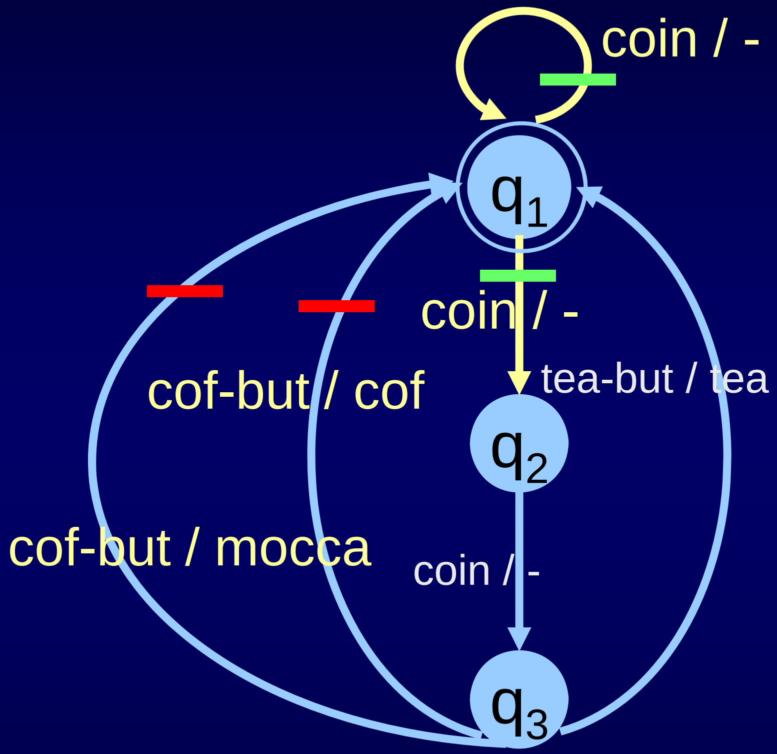
condition		effect	
current state	input	output	next state
q_1	coin	-	q_2
q_2	coin	-	q_3
q_3	cof-but	cof	q_1
q_3	tea-but	tea	q_1
q_3	**-**	**-**	**q_4**
q_4	fix	-	q_1

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

alias: internal transition

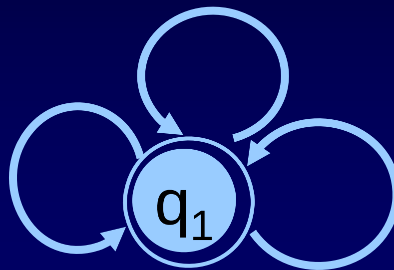
alias: unobservable transition

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)



these are the extended parts

or "guard"

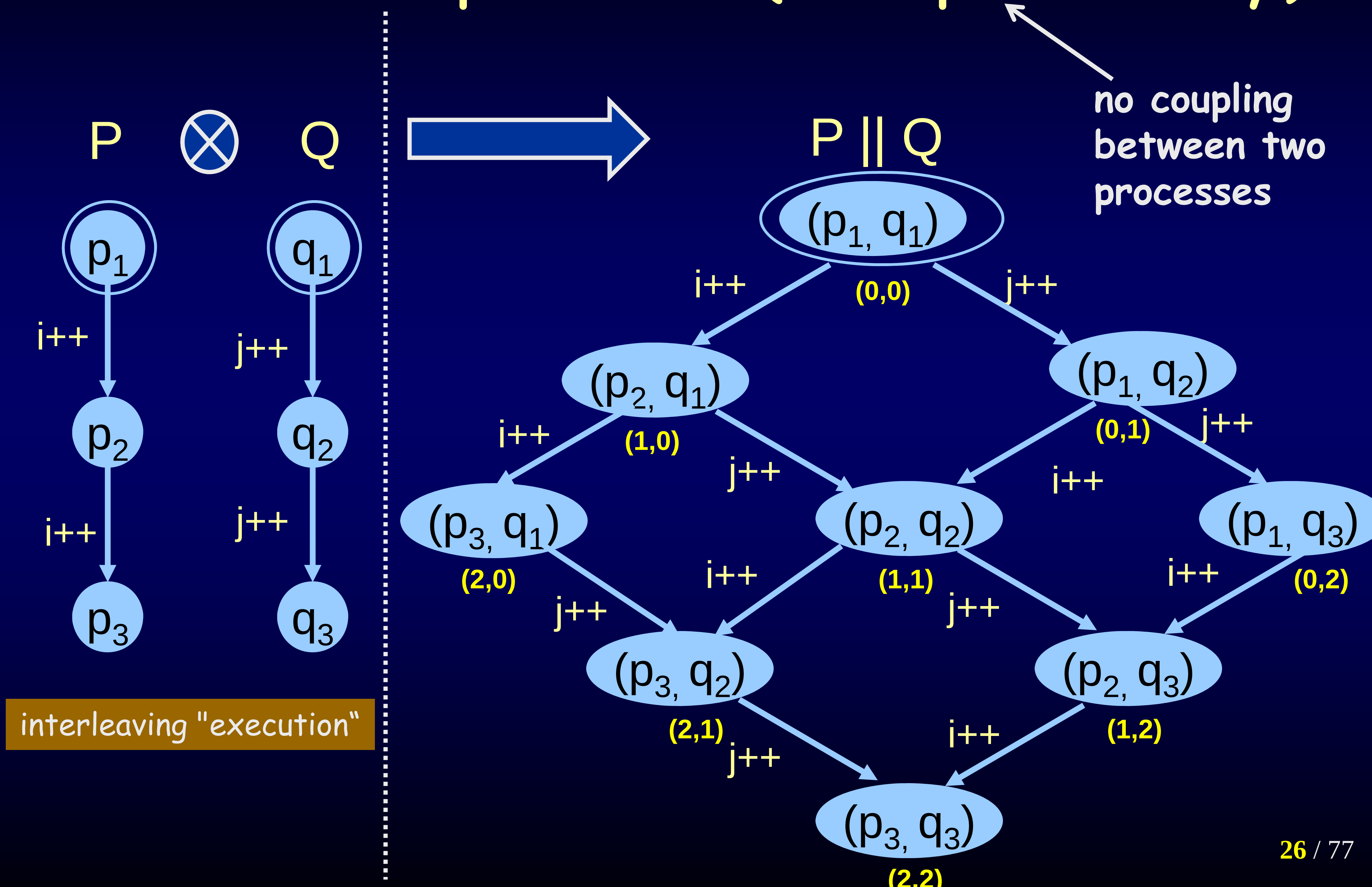
called “flattening"

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

(q, *total*, *capacity*)

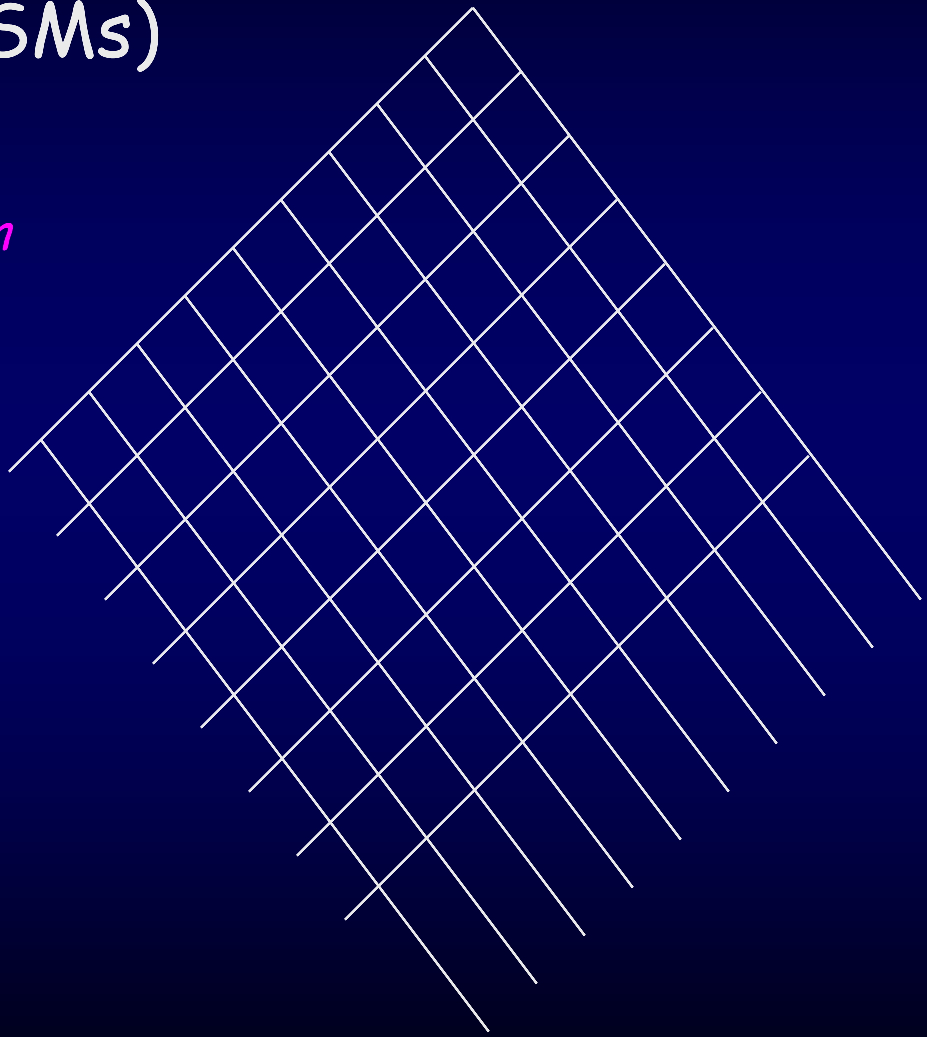
$(q_1,0,10) \xrightarrow{\text{coin / -}} (q_1,1,10) \xrightarrow{\text{coin / -}} (q_1,2,10) \xrightarrow{\text{cof-but / cof}} (q_1,0,9)$

Parallel Composition (independently)



State Space Explosion

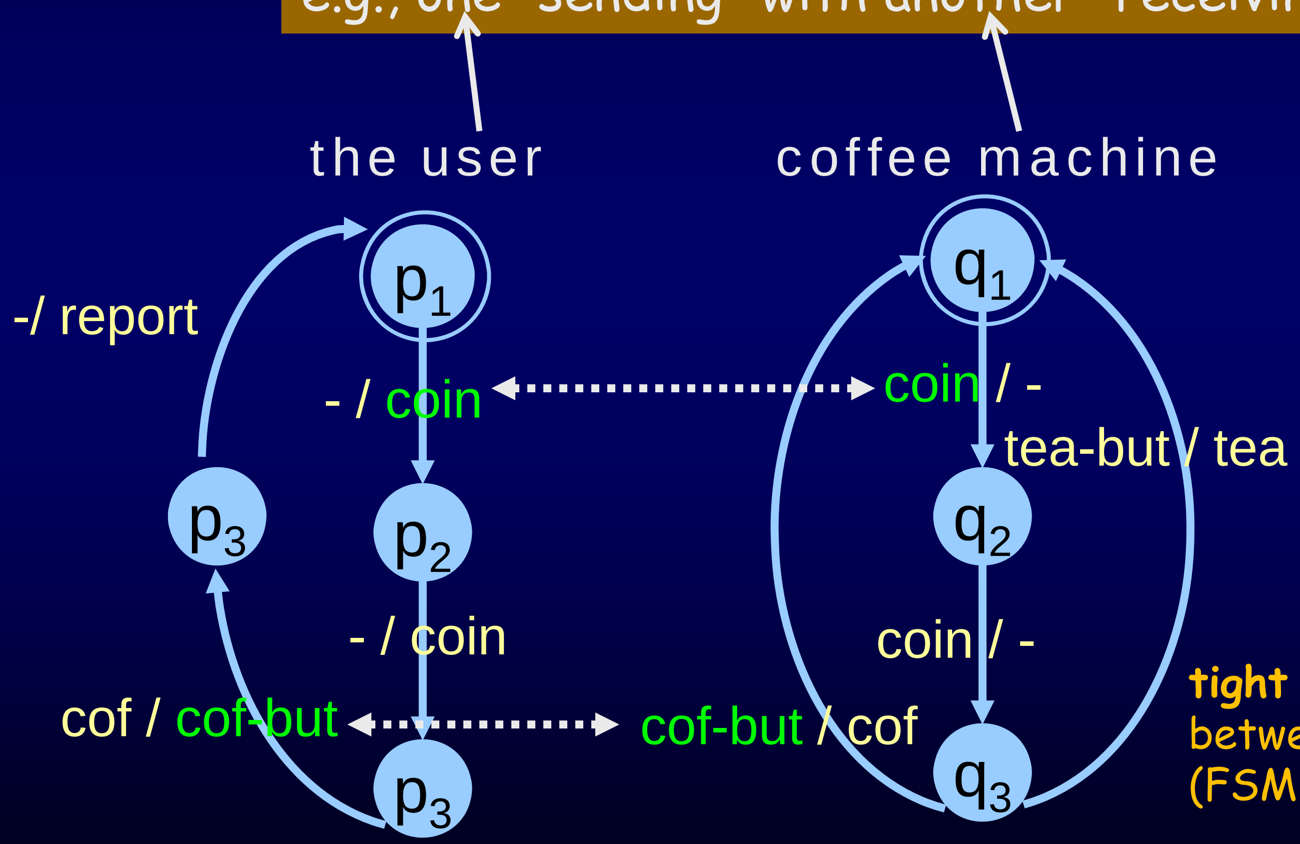
- *n* parallel FSMs (or EFSMs)
- Each with *k* states
- In parallel they have k^n states
- **EXPONENTIAL!**
  - 10^2 =100
  - 10^3 = 1000
  - 10^4 = 10000
  - 10^10 =10000000000



Parallel Composition (Synchronous)

Handshake on **complementary** actions

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



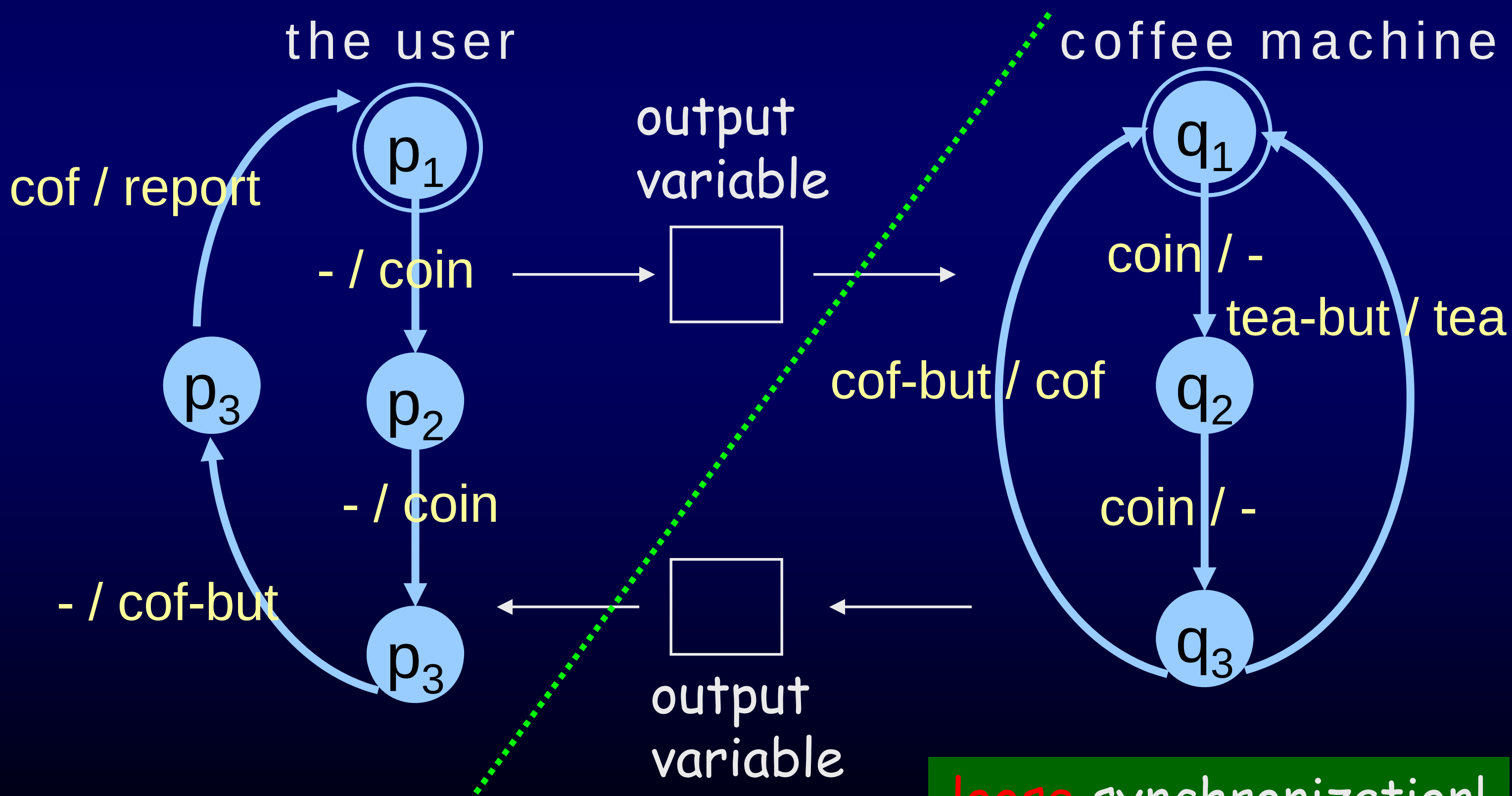
tight coupling of events between two components (FSM's)

strict synchronization!

Parallel Composition (Asynchronous)

Single output variable per FSM holds last “written” output

no handshaking any more!

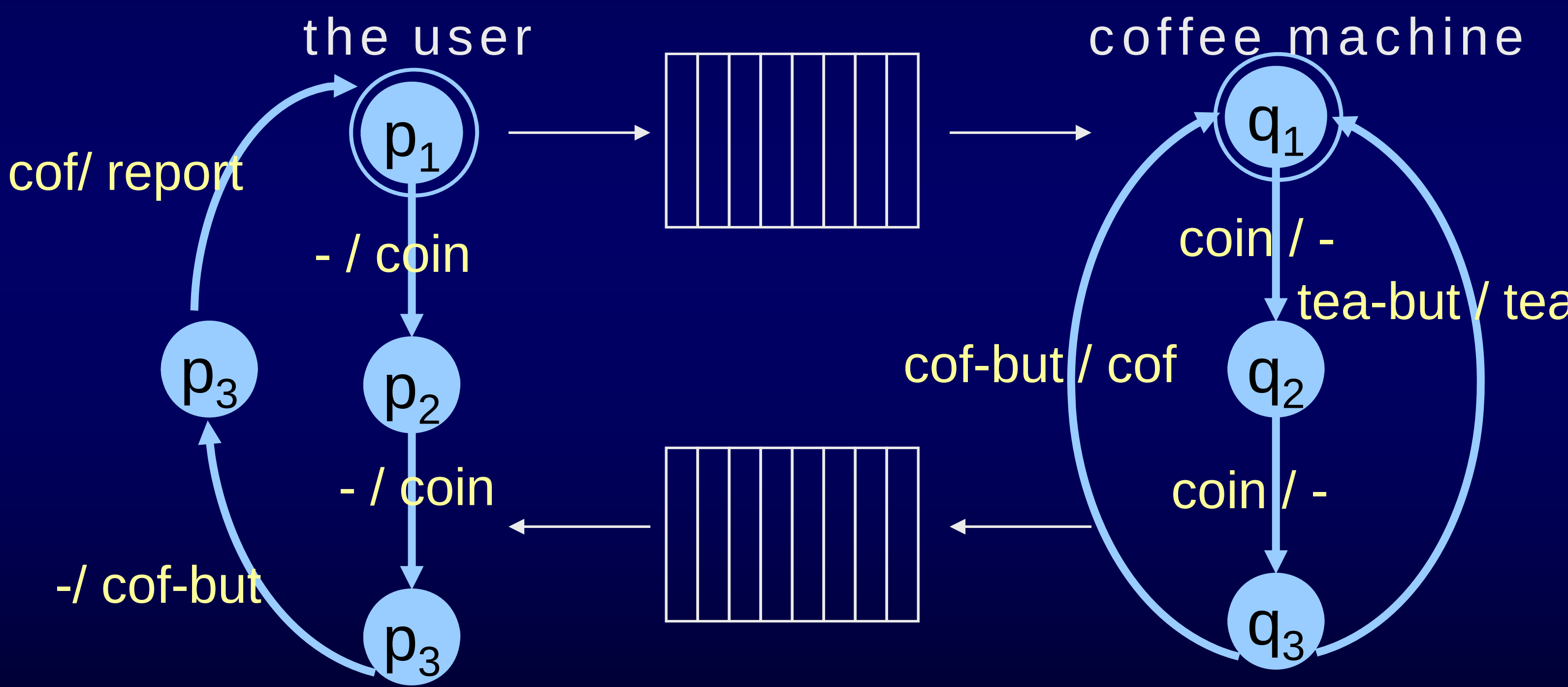


loose synchronization!

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

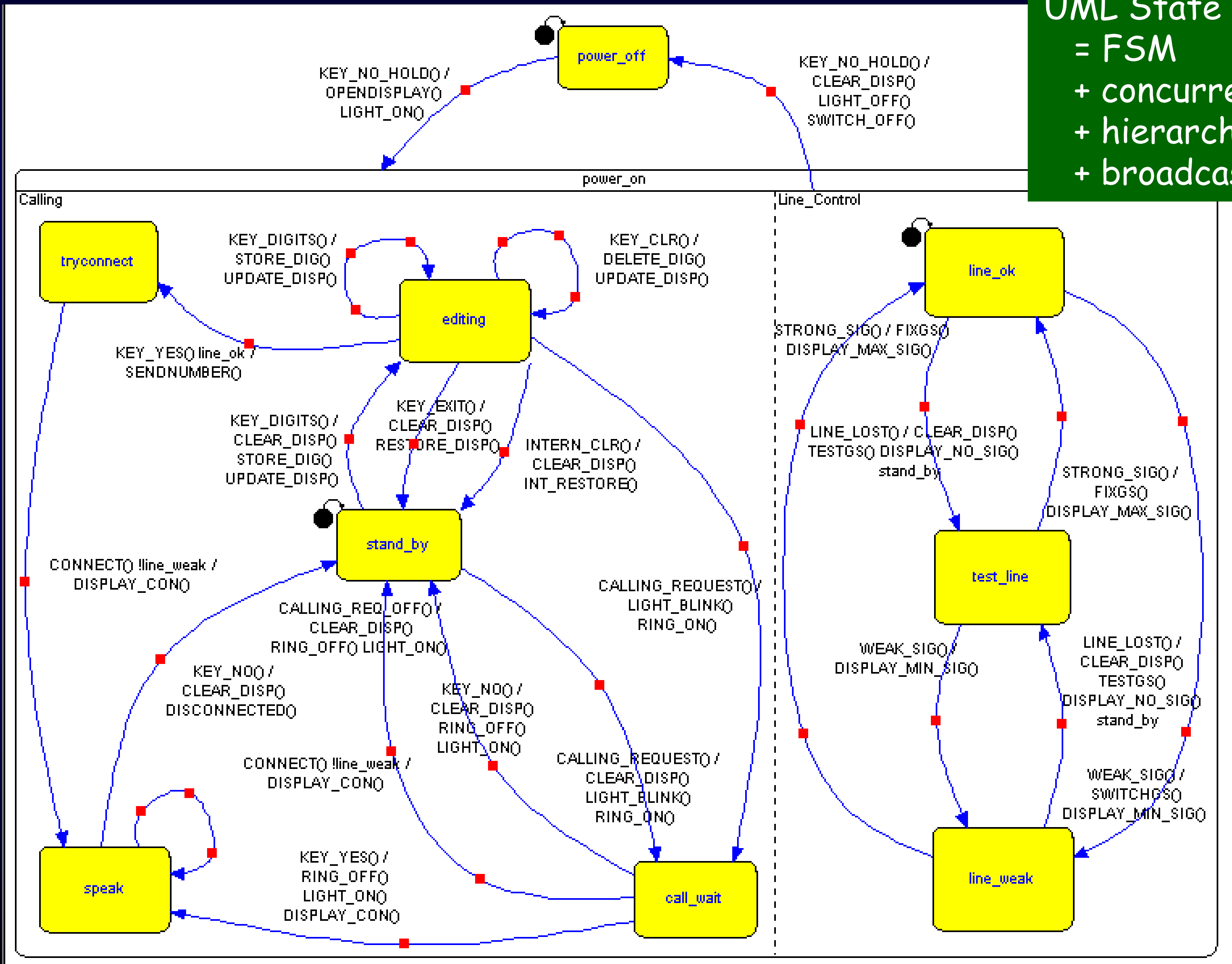
- A minimized (or "reduced") M is one that has no equivalent states
  - i.e., for all states s, t : (s equivalent t) ==> (s = t)

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)

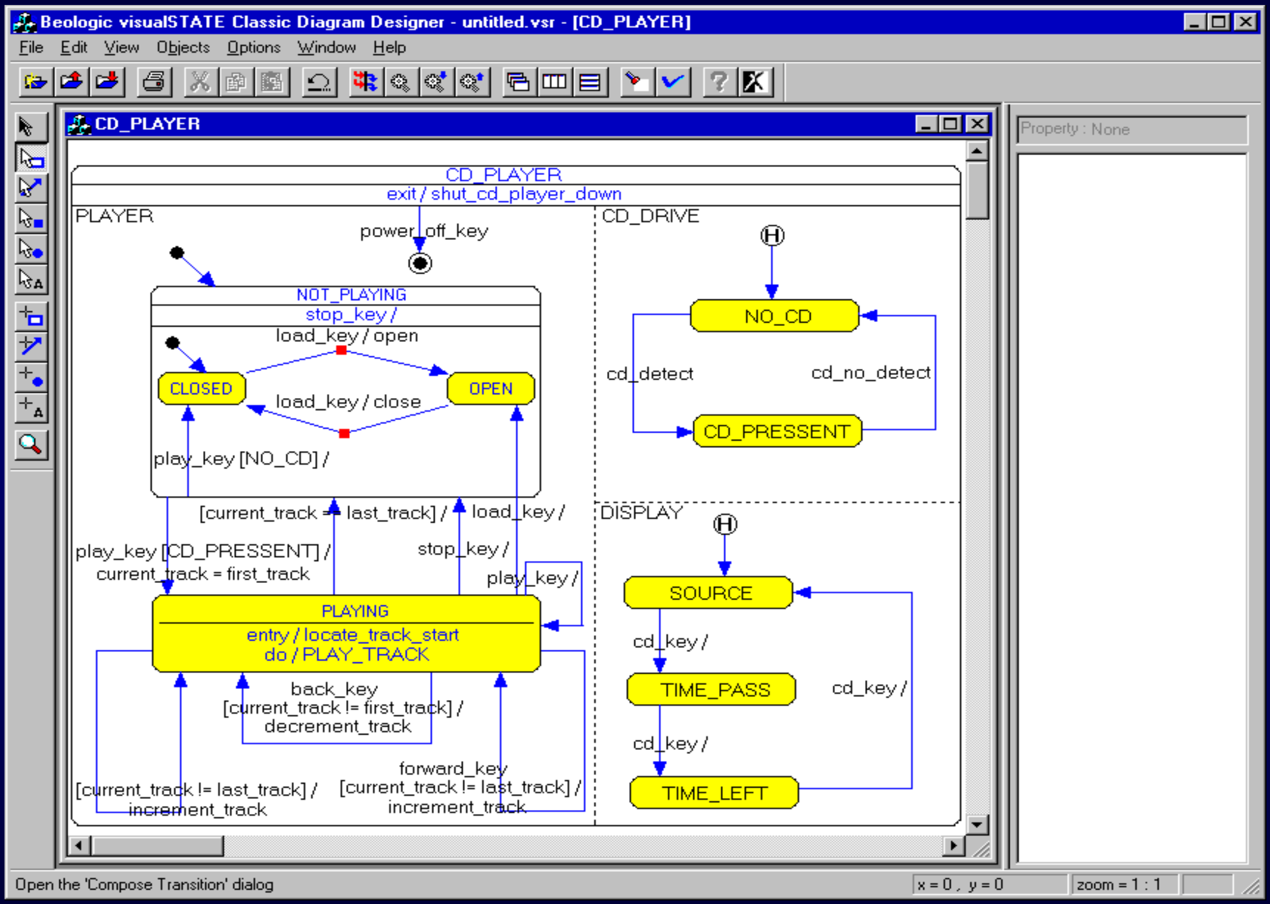
High-level
State Transition-Based Models

UML State Machines



UML State Machine
= FSM
+ concurrency
+ hierarchy
+ broadcast communication

Tool: visualSTATE Designer

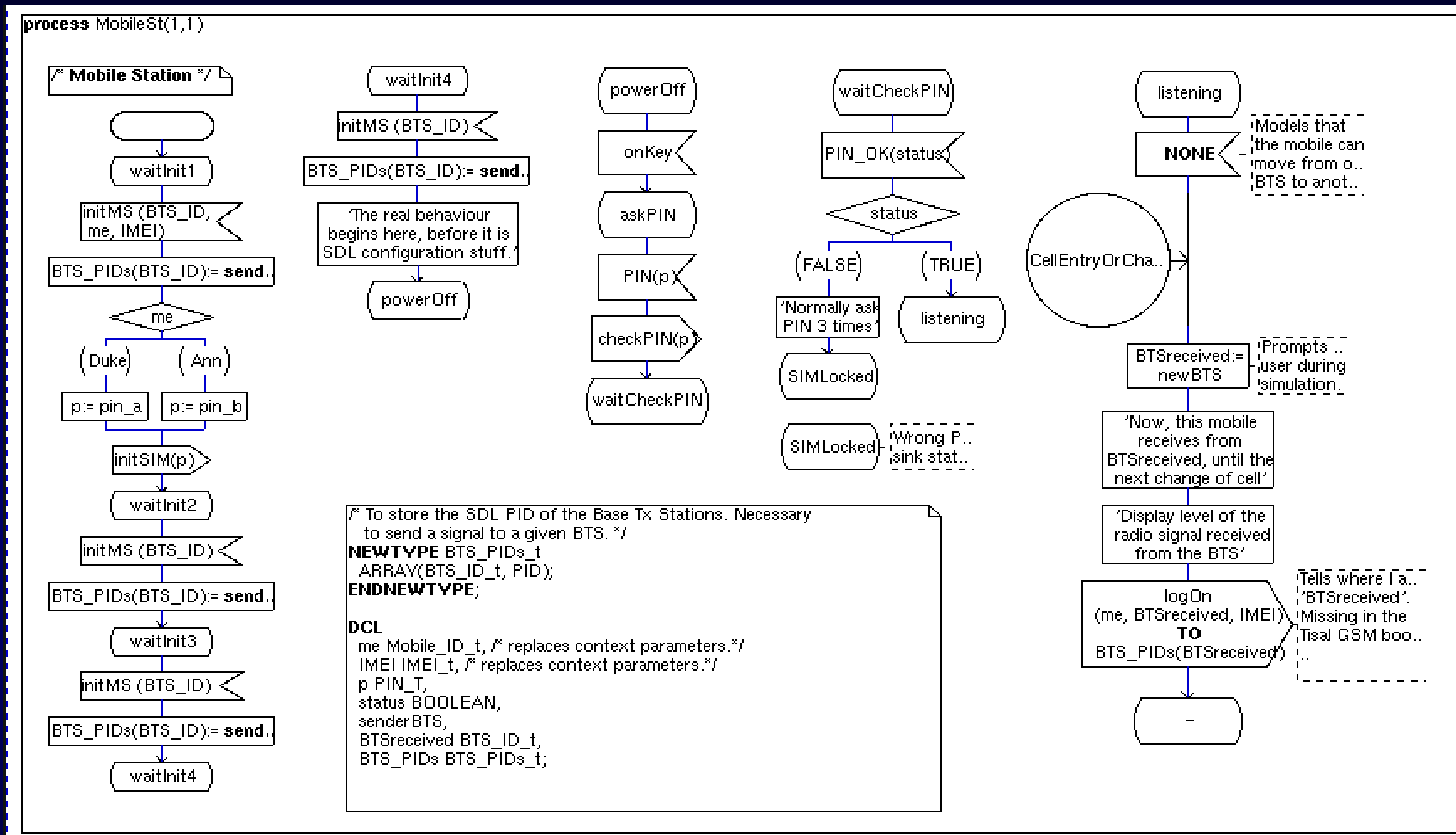


other supporting tools exist, e.g. Rational Rose

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

SDL language

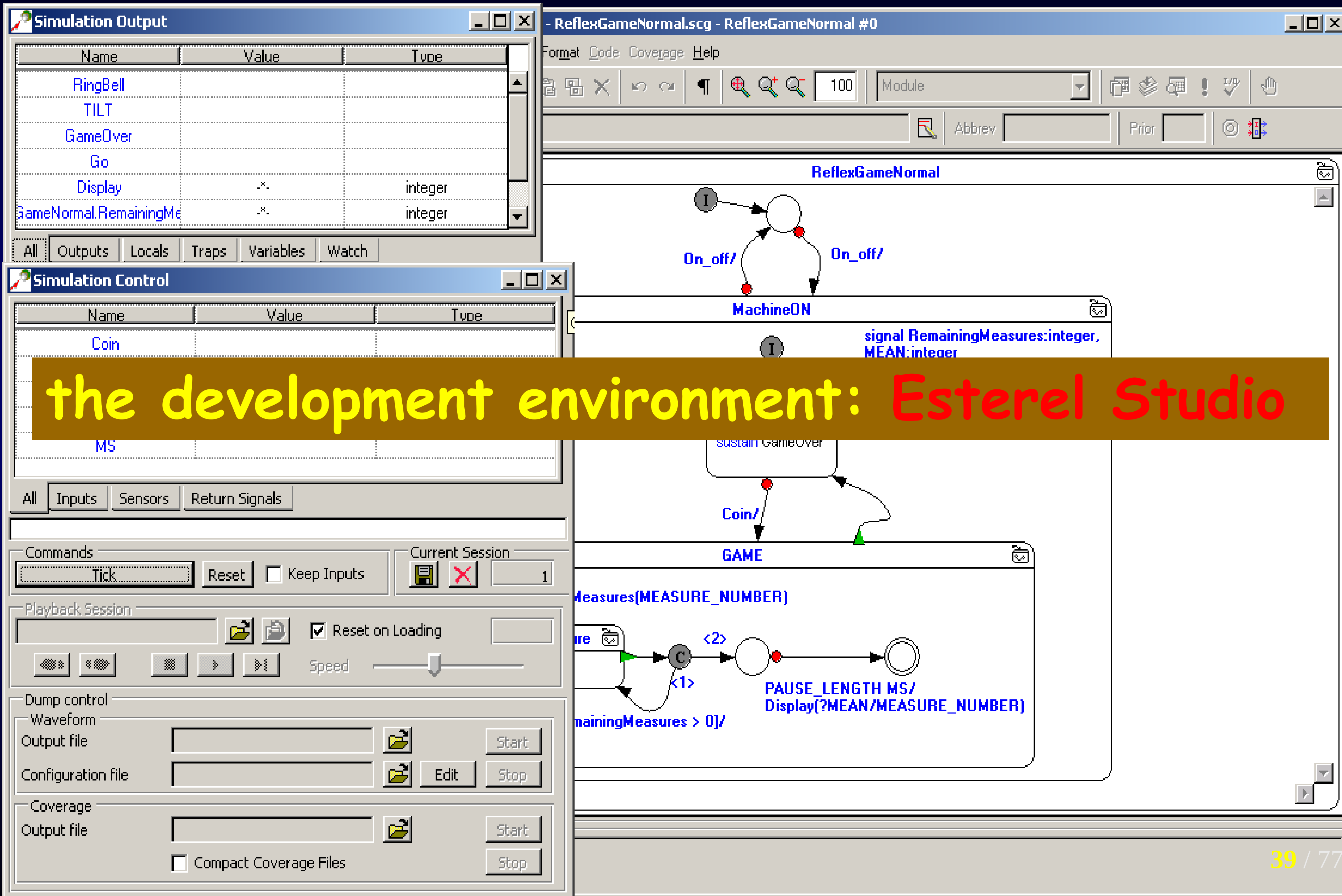
a system is specified as a set of interconnected abstract machines which are **extensions of FSM**



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 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()}
```

(Promela language: the input language of tool SPIN)

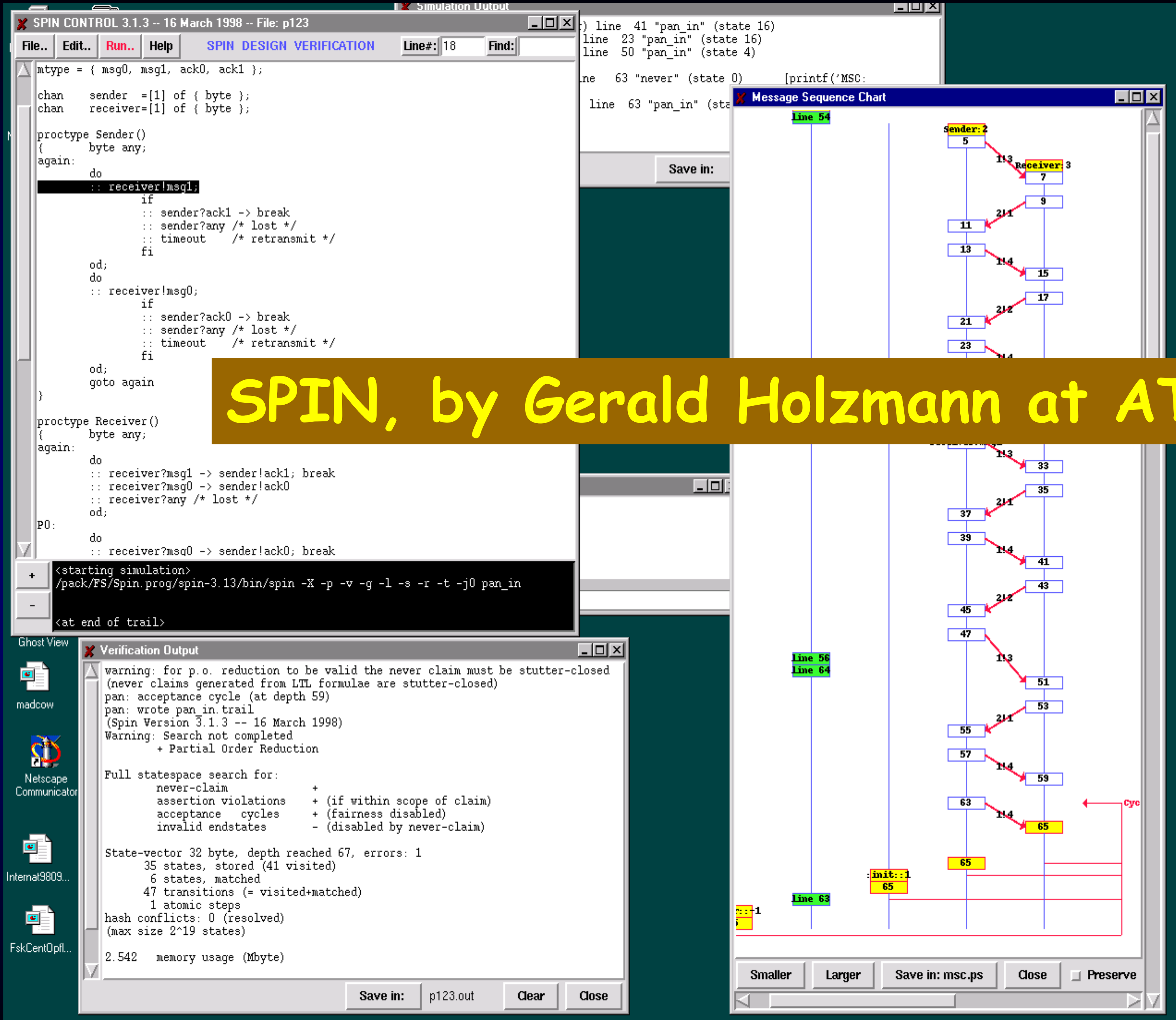
In: FSP/LTSA

```
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, by Gerald Holzmann at AT&T



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

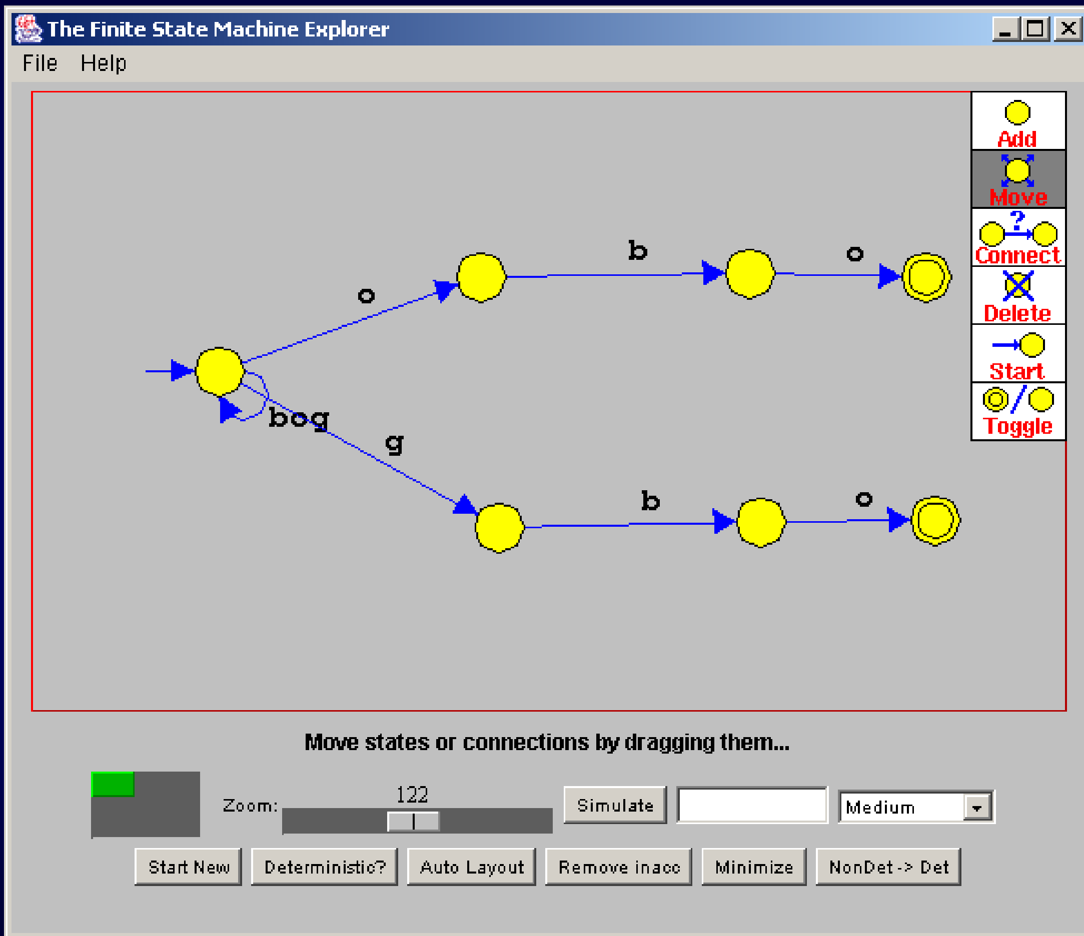
(4) To open a bank box
the code must end with a palindrom
e.g:. , or
, or

....

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)



try it out!

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

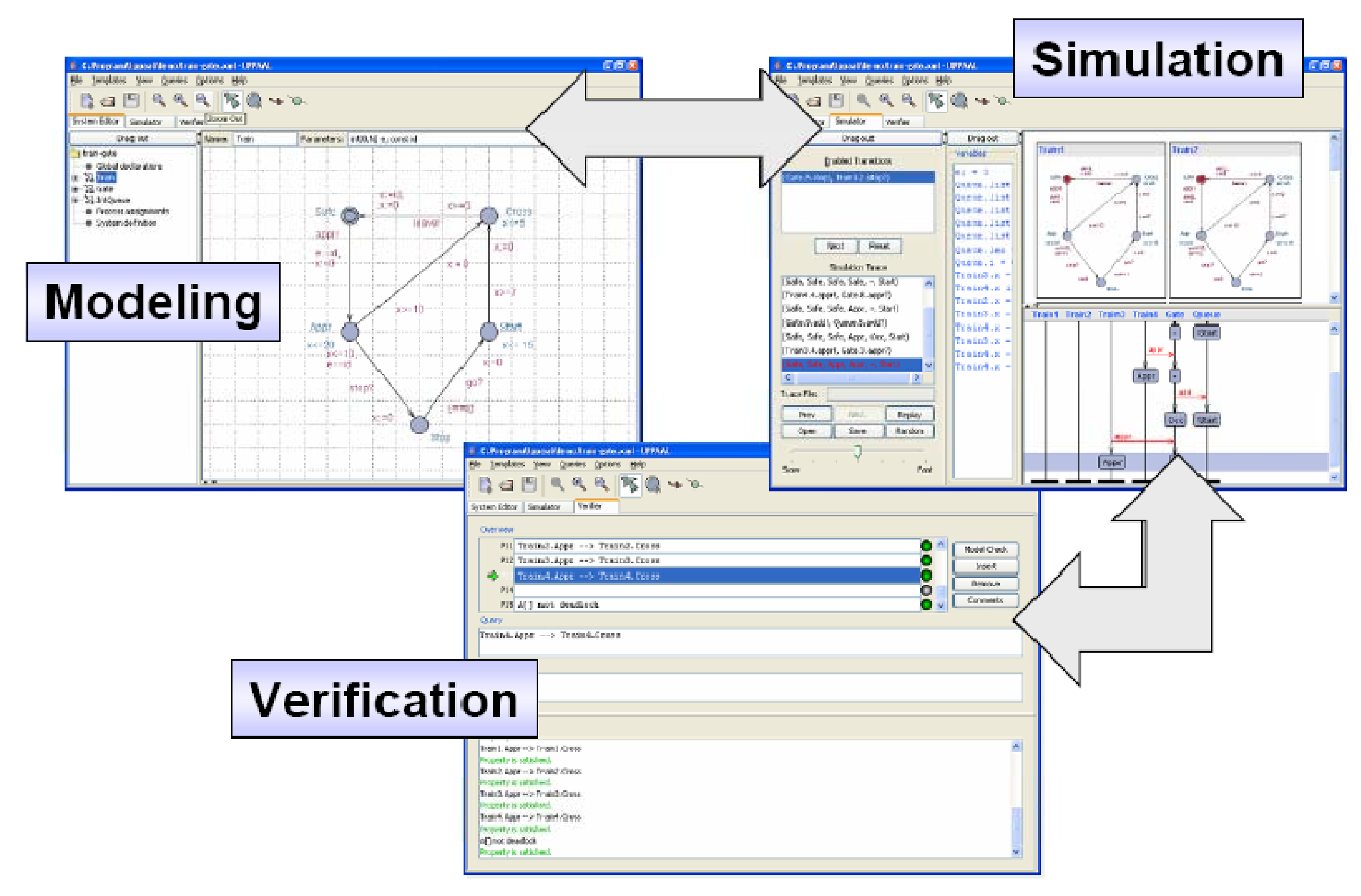
Uppaal

- 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

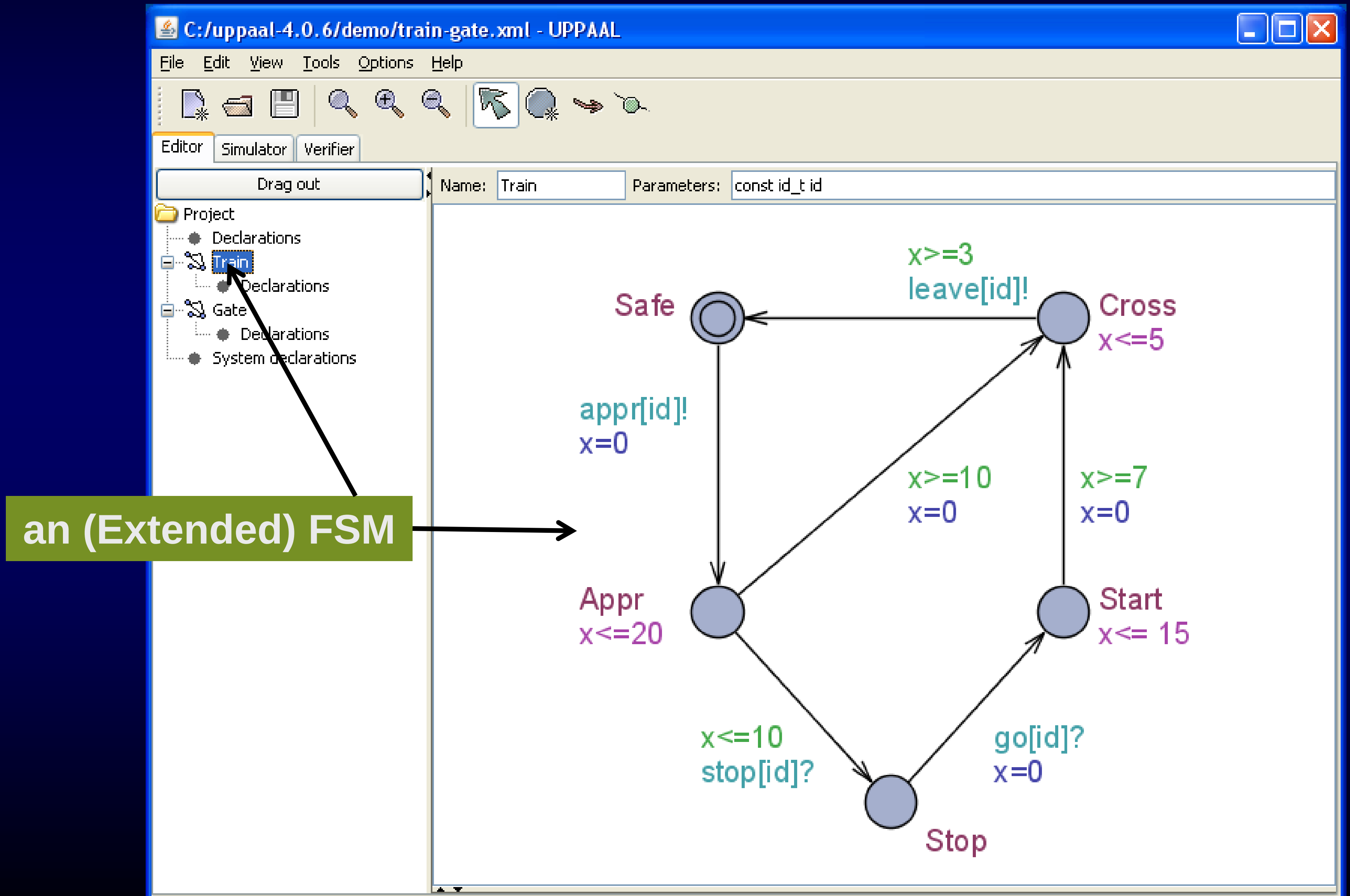
- However, it is also capable of untimed reactive system modelling, simulation and verification

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

Working Modes of Uppaal



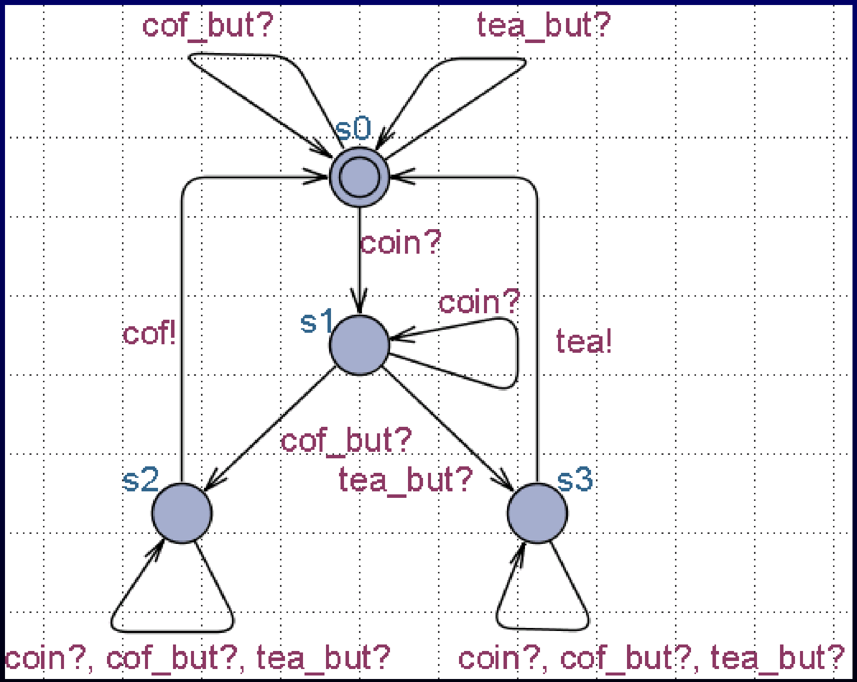
Uppaal Editor (Modelling View)



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"

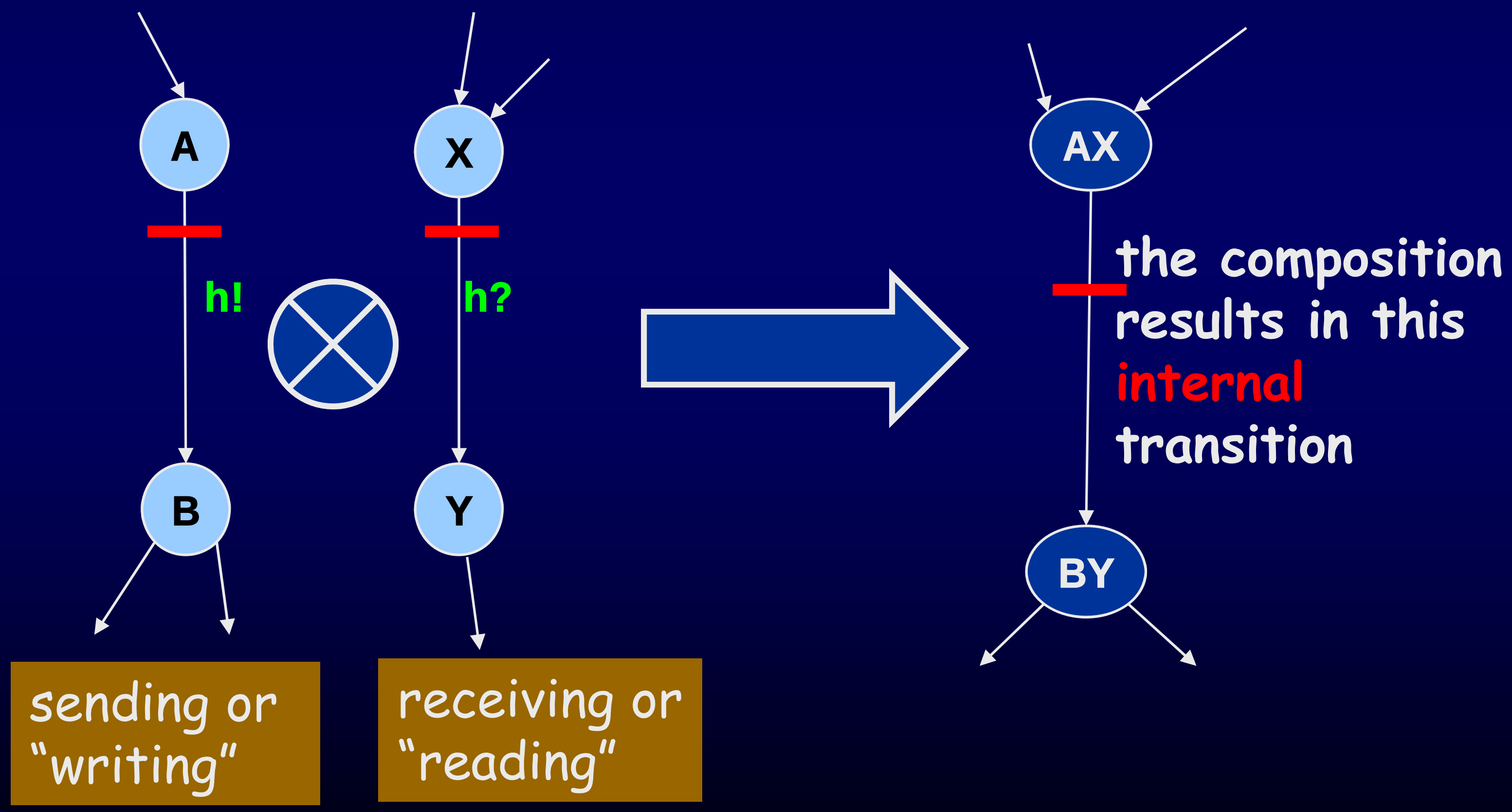
coffee vending machine

Model Composition

IO Automata (2-way synchronization)

or pairwise synchronization

or “handshaking”

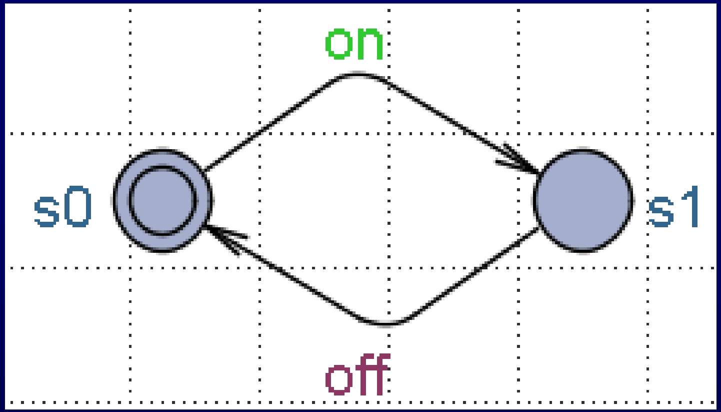


Modelling Processes

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

a light switch

LTS



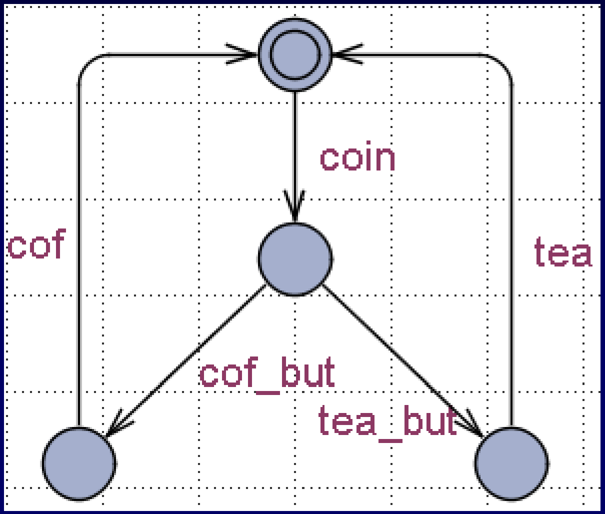
on→off→on→off→on→off→ ……….

a sequence of actions

or

a *trace*

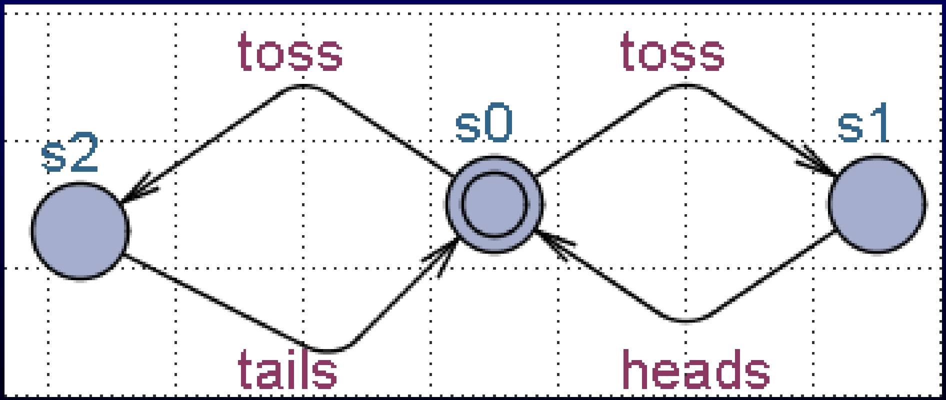
Modelling Choices



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

Non-deterministic Choice: modeling random event

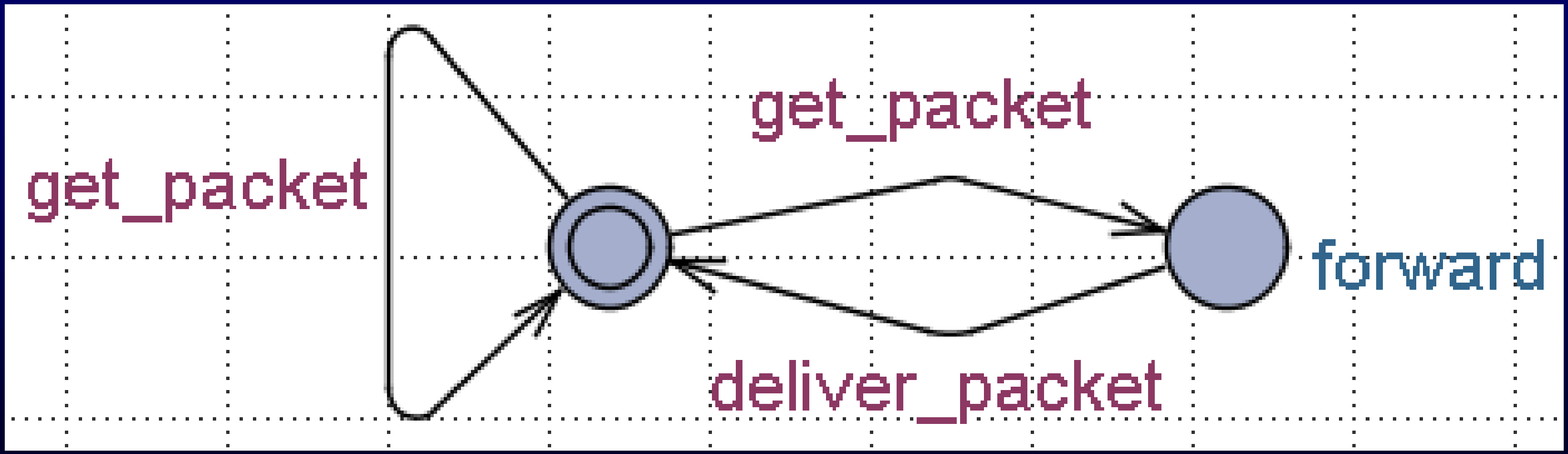
- Tossing a coin
- Possible traces?
  - Both outcomes (head or tail) possible
  - Nothing said about relative frequency
  - If coin is fair, the outcome is 50/50



Non-deterministic Choice: modeling failure

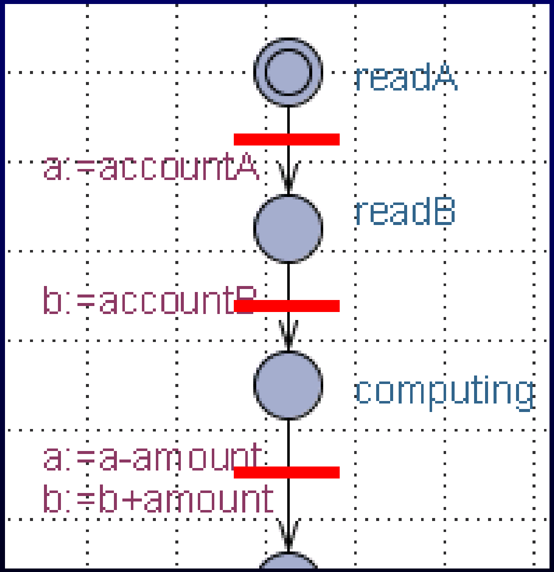
How do we model an unreliable communication channel which accepts **packets**, and if a failure occurs produces no output, otherwise delivers the packet to the receiver?

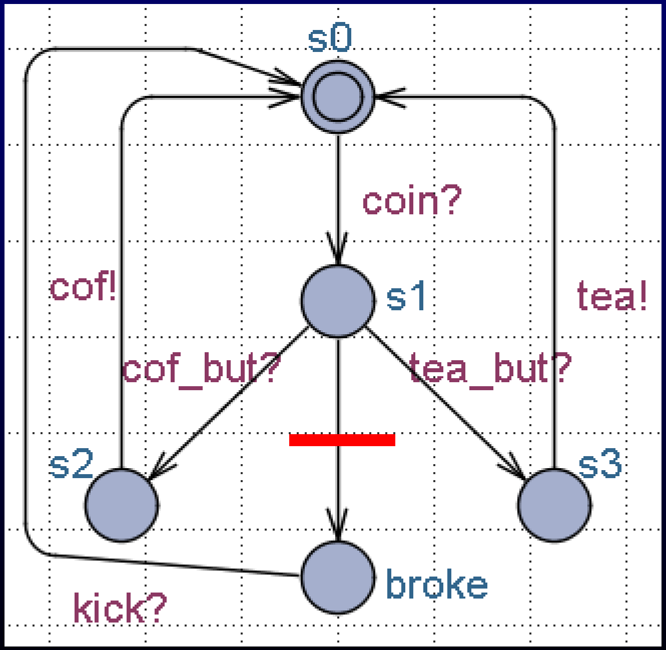
Use non-determinism...

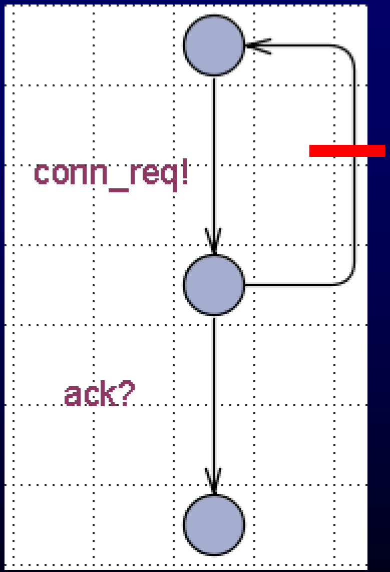


Internal Actions

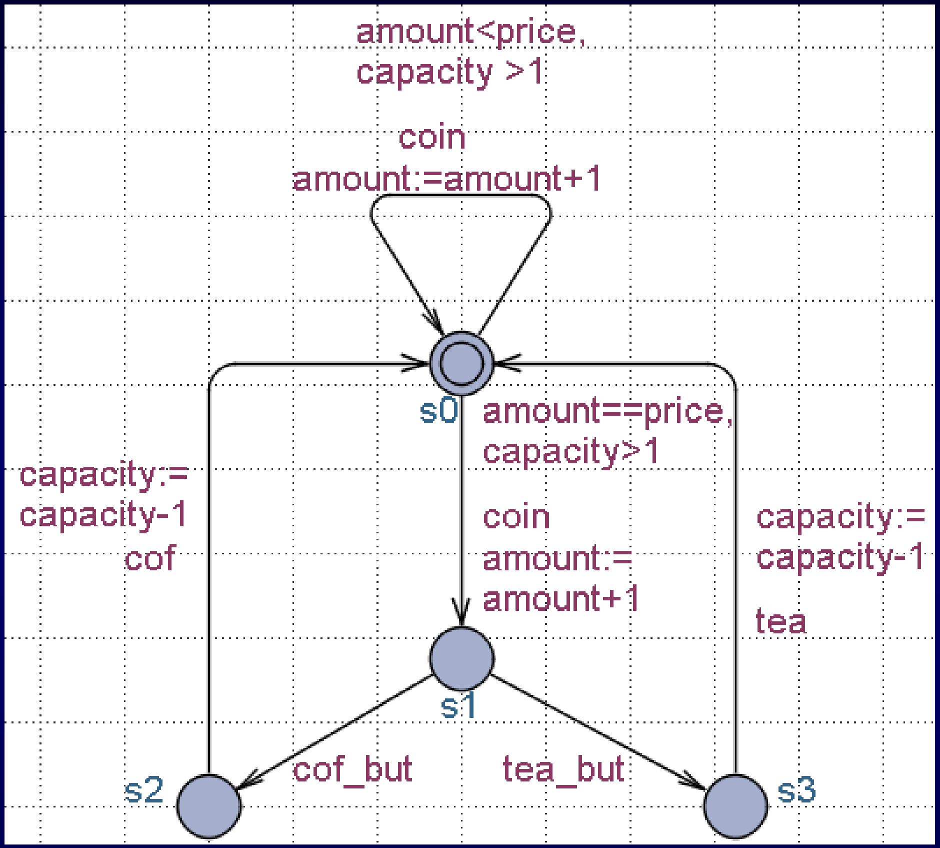
- Internal actions are also called
  - spontaneous actions, or
  - tau-actions
- Internal transitions can be taken on the initiative of a single machine without coupling with another one







Modelling Extended FSM (EFSM)

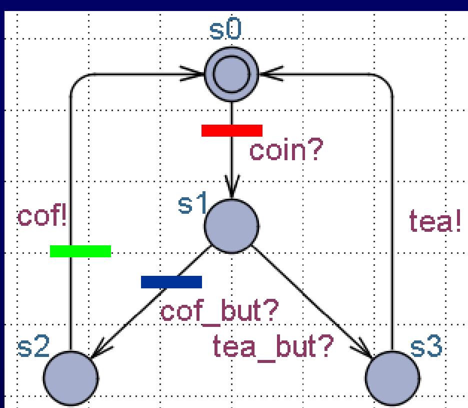


- EFSM = FSM + variables + enabling conditions + assignments
- Transition still atomic (thus not really an EFSM!)
- Can be translated into FSM if variables have bounded domains
- State: control location + variables' valuation
- (state, total, capacity), e.g.: (s0, 5, 10)

Process Interaction

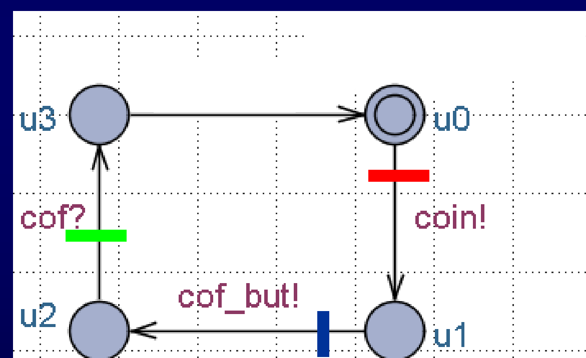
- "!" denotes output, "?" denotes input
- Handshake communication 
- Two-way

Coffee Machine



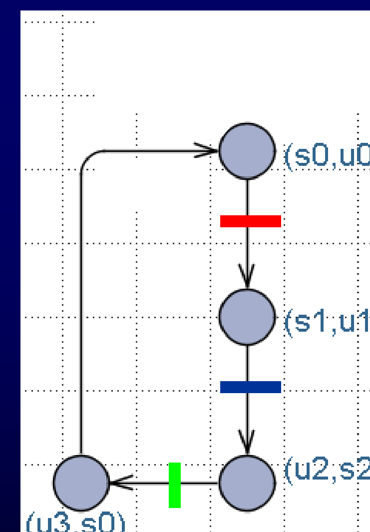
4 states

Lecturer



4 states

University=
Coffee Machine || Lecturer



synchronization results in internal actions

LTS?
How many states?
Traces ?

4 states:

(interactions constrain overall behavior)

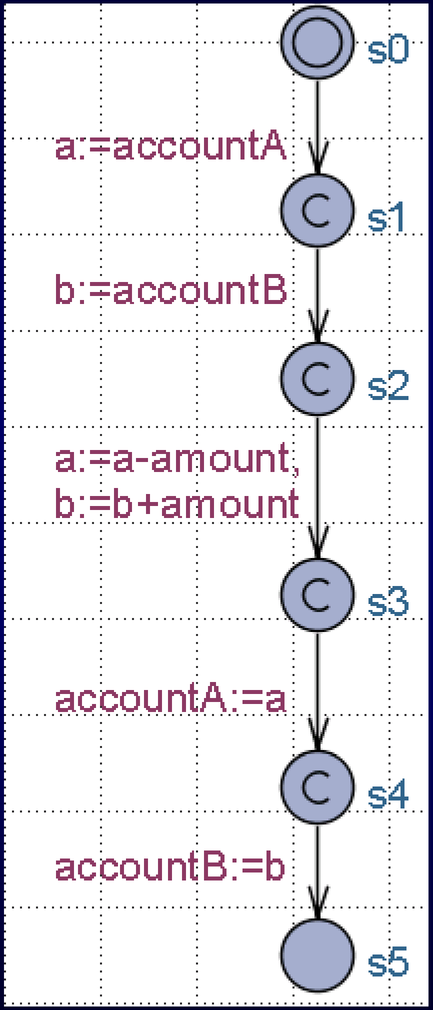
Broadcasts

```
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 zero "audience")

Committed Locations

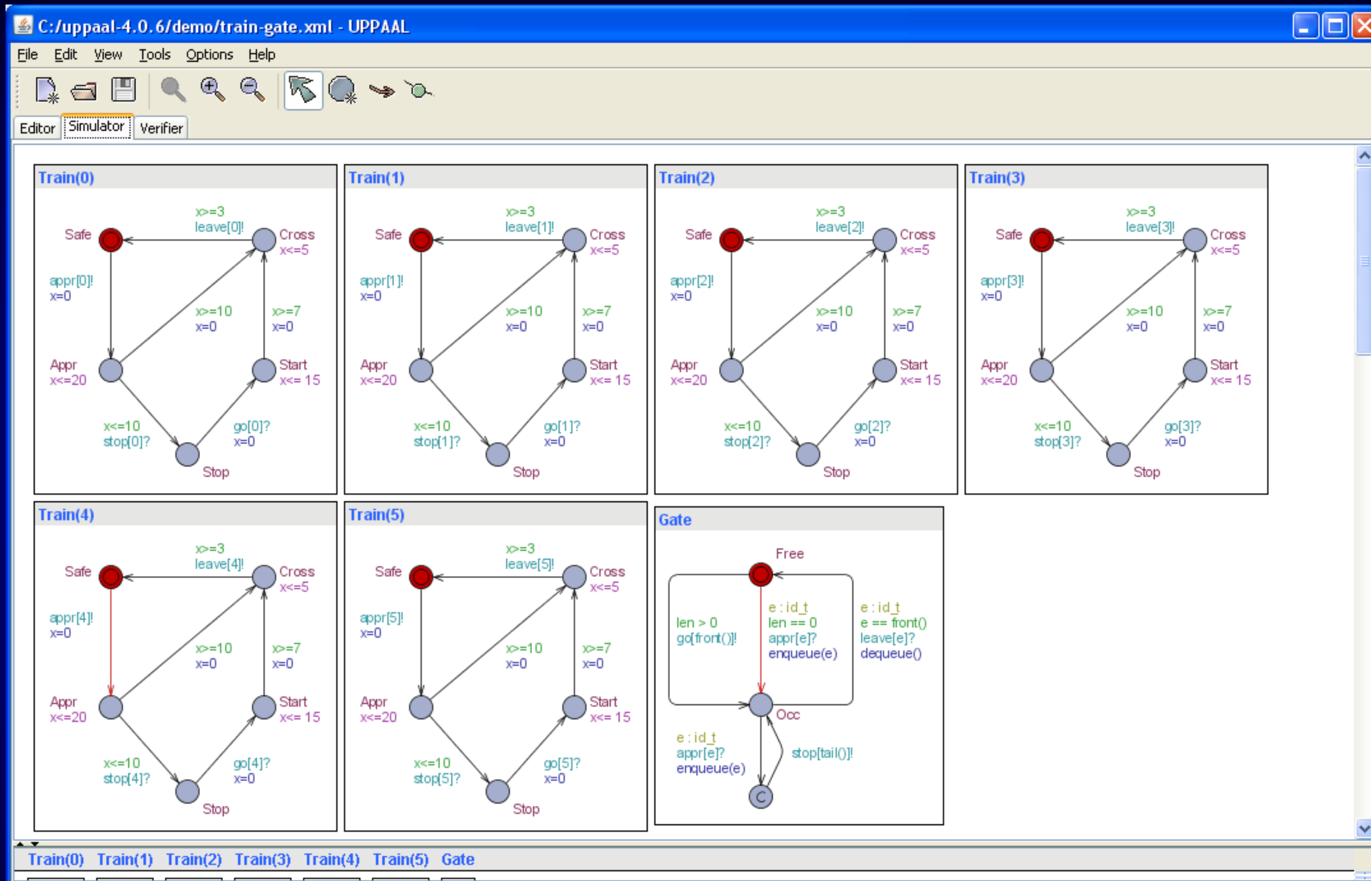
- Locations marked "C"
  - *No delay* in committed location
  - Next transition must involve one of those automata in ***committed locations***
- Handy to model atomic sequence of actions
  - An "input/output"-style transition of Mealy machine can be modelled by 2 atomic actions "input?" and "output!",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

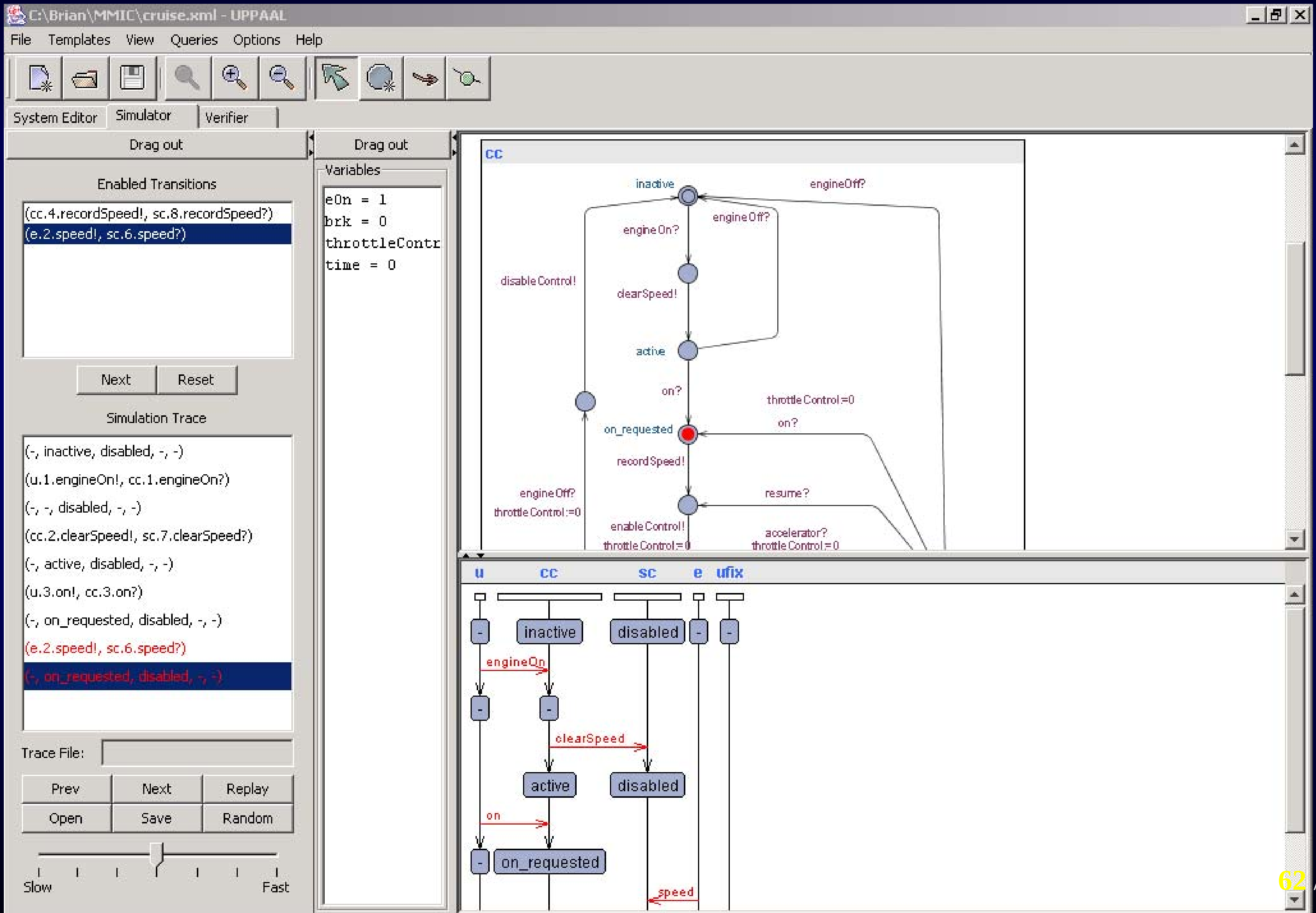
Uppaal Network of Automata



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

They constitute a closed system.

Uppaal Simulator Screenshot

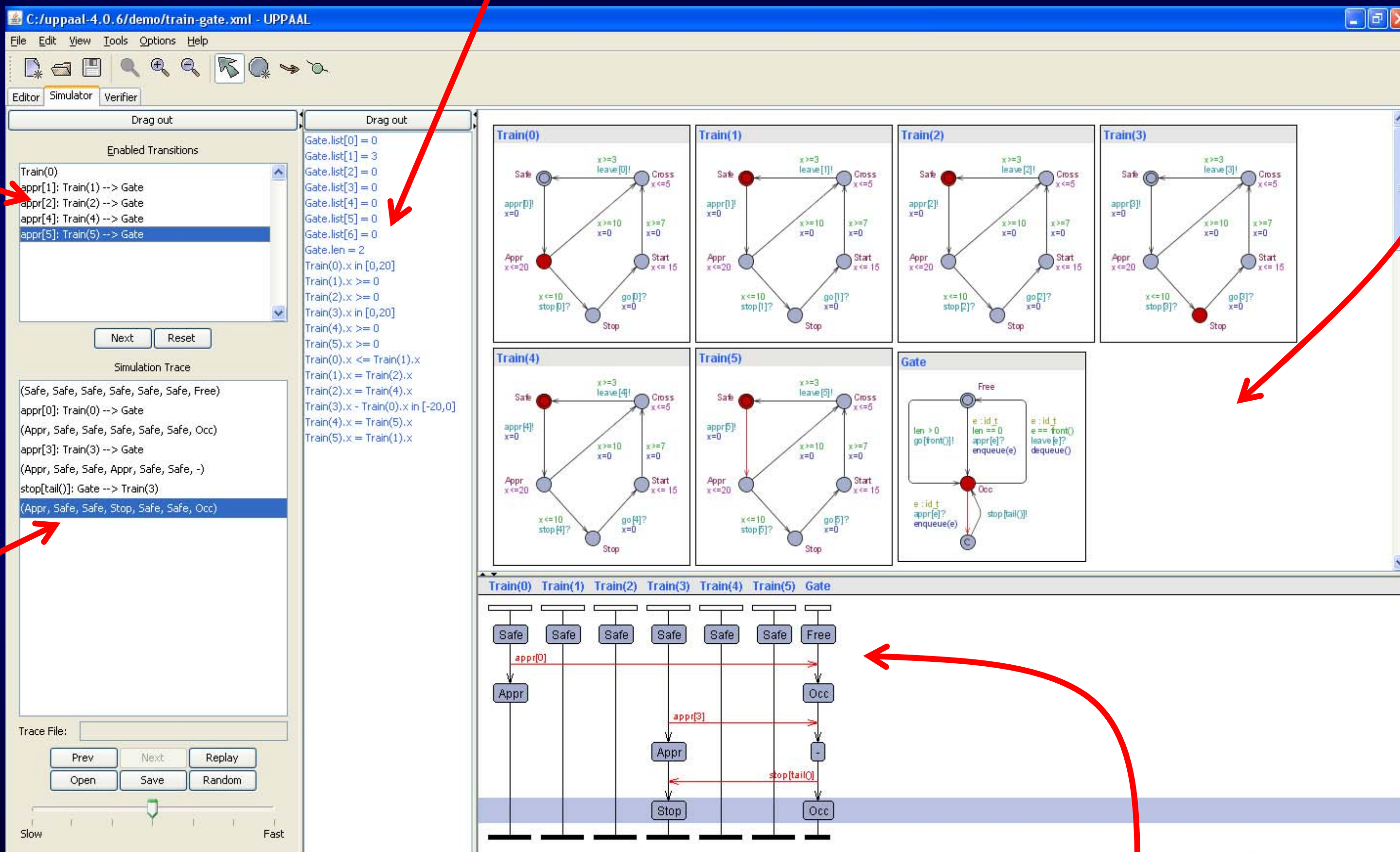


Model Simulation in Uppaal

the enabled transitions that the user can choose

the system state variables

graphical display of the current system state

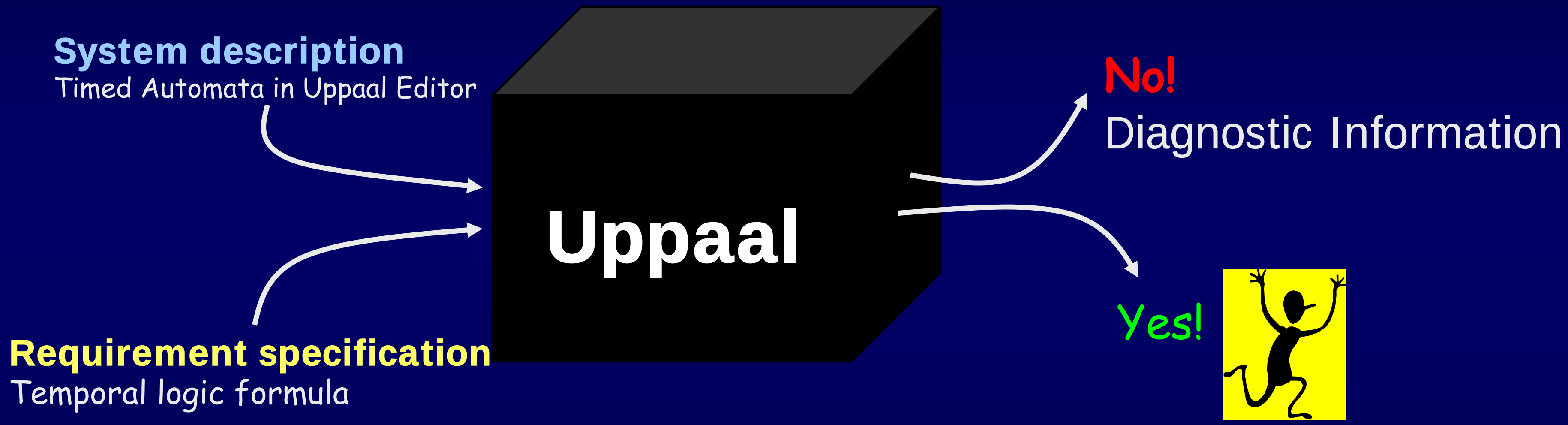


history information as trace

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
- P --> q

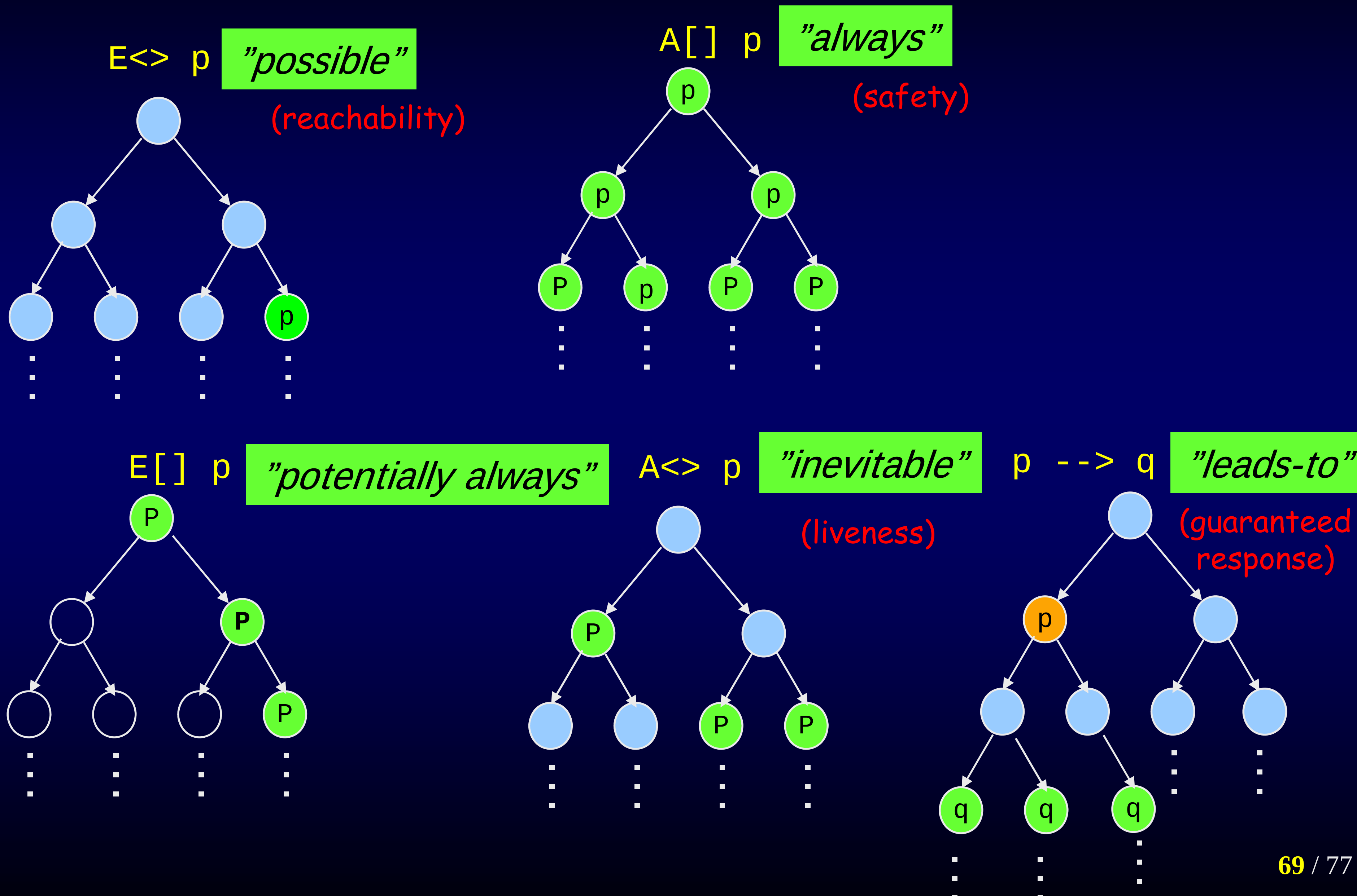
process location | data guards | clock guards

“p leads to p”:
A[ ] (p imply A< > q)

p::= a.l | g_d | g_c | p and p |
p or p | not p | p imply p |
(p) | deadlock (only for A[],E<>)

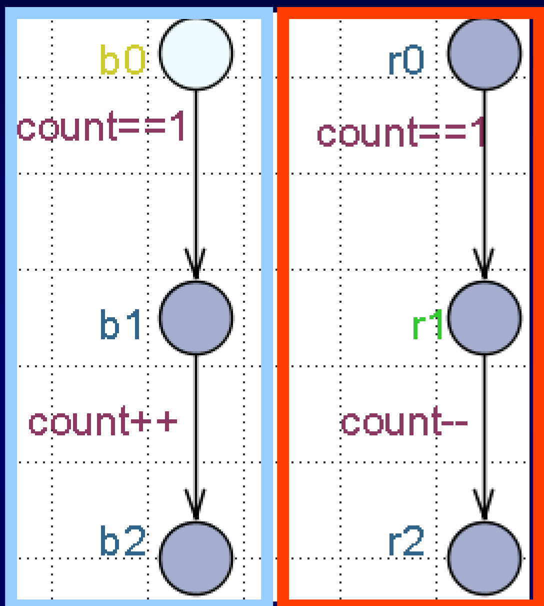
A[] (mc1.finished and mc2.finished) imply (accountA+accountB==200)

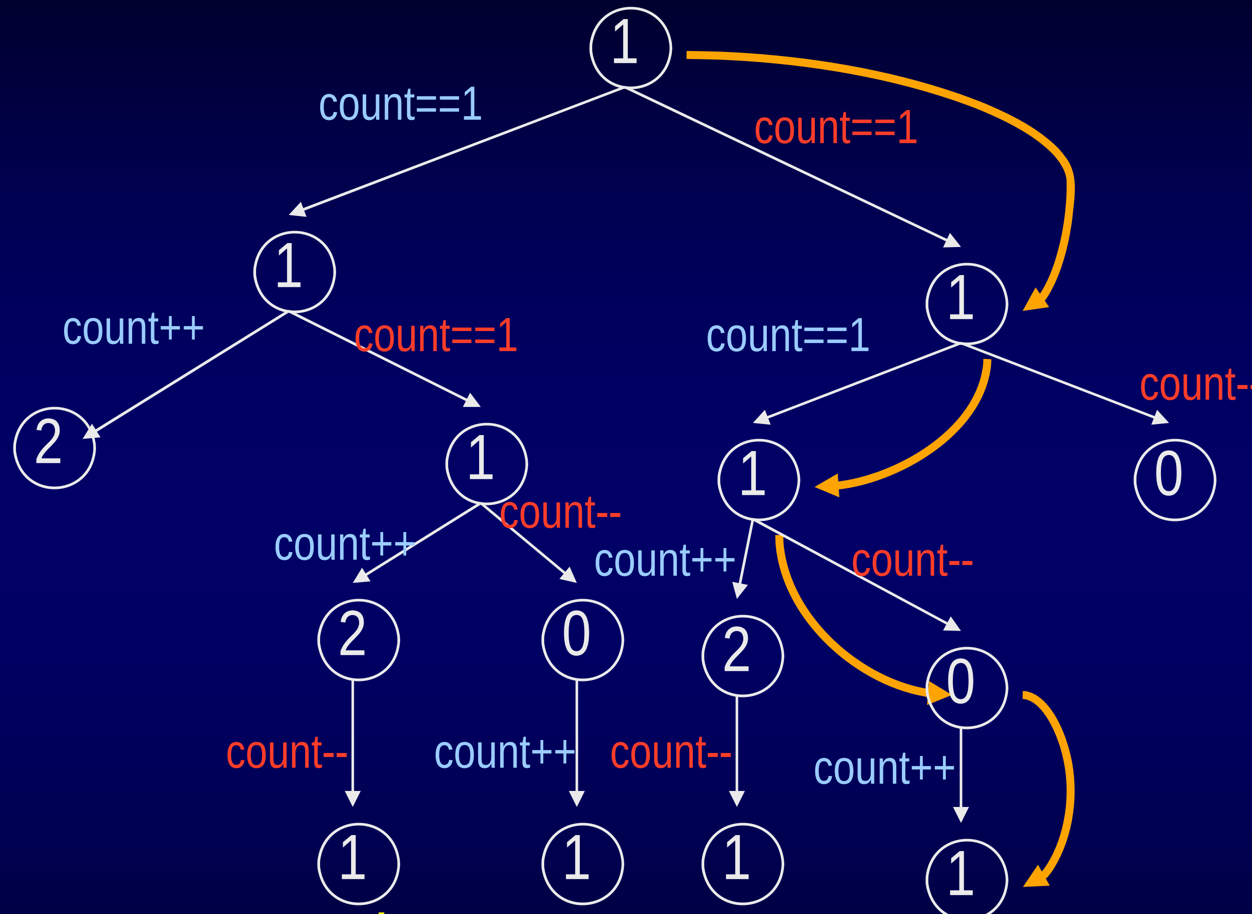
Uppaal "Computation Tree Logic"



State Space Exploration

Int count:=1

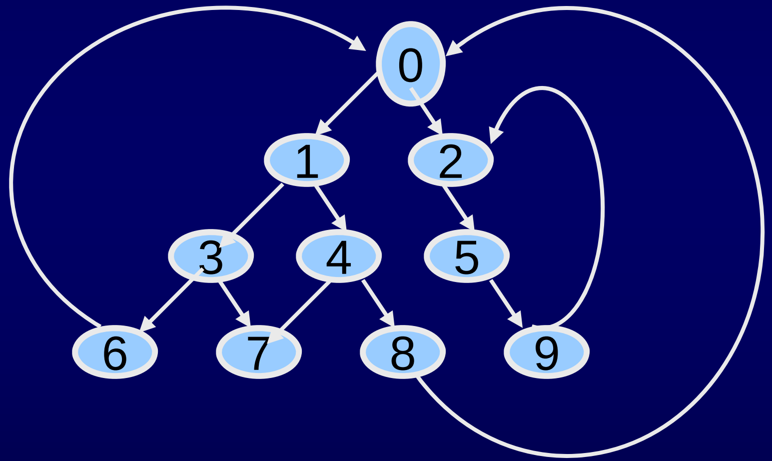




- Each trace = a program execution
- Uppaal checks *all* traces
  - Is count *possibly* 3 ? E<> count==3
  - Is count *always* 1 ? A[] count==1

Reachability Analysis

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



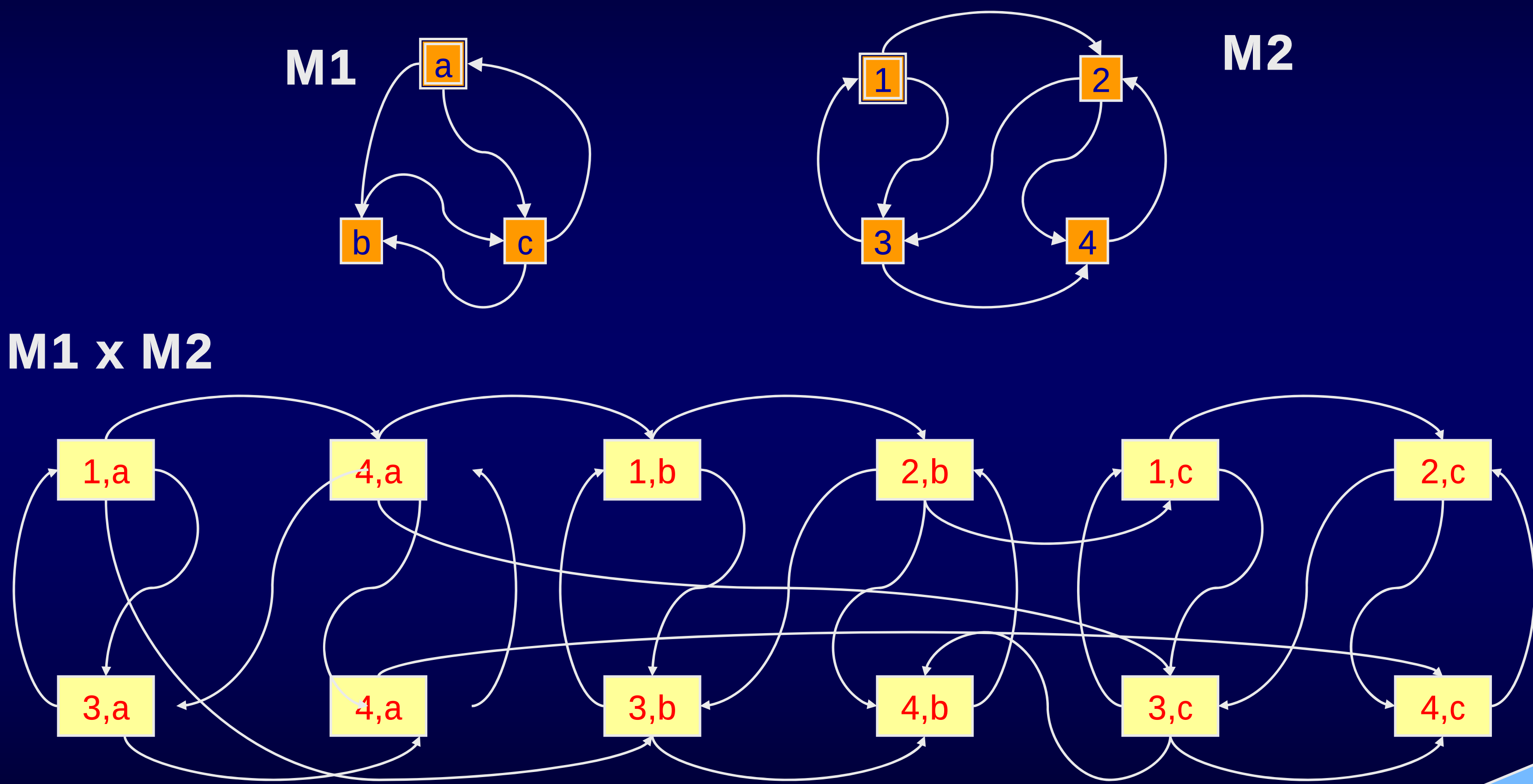
Depth-First: maintain waiting as a stack

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

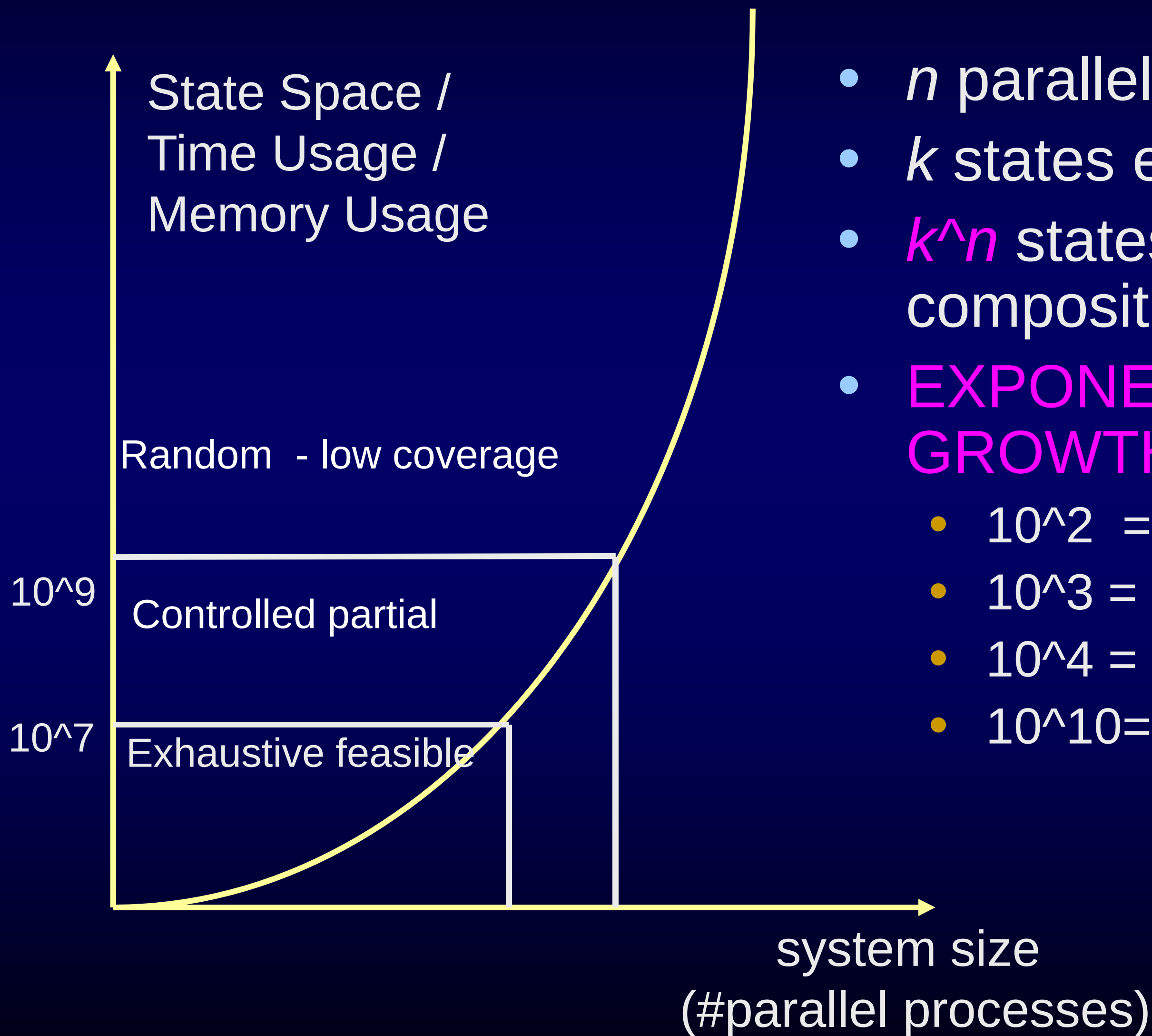
'State Explosion' problem



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 rathar 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!