Example 1: A Communication Protocol

A machine M1 sends a message to another machine M2, through some unreliable network. The communication protocol is in charge of making sure that messages reach their destination: they use acknowledge messages, and may retry sending a message when it is not received.

How to know that a message *is not received?*
Wait a reasonable amount of time (precision not needed), and if the message is not received, act as if it was lost.

Example 2: Image Processing

One may need to produce outputs at a given rate (audio, video...).

Need to count time internally, with some precision.

Example 3: Watchdogs in critical embedded systems

In a car, for instance, there's a HW (or SW) element playing the role of a watchdog.

It's a counter that decreases automatically and may reset or reconfigure the whole system in a degraded mode when it reaches 0.

The embedded SW should rearm the watchdog (re-assign the max value) from time to time. If it does not, it means there's a deadlock or a livelock (e.g., non-ending loop behavior) somewhere.

Same principle as: how to make sure the driver is alive in a train?

Example 4: Physical Timing Constraints

Recall the loop behavior of most reactive programs:

```
// as fast as possible
// not regular in phy time
init
while true loop
      get inputs
      compute
      emit outputs
end loop
```

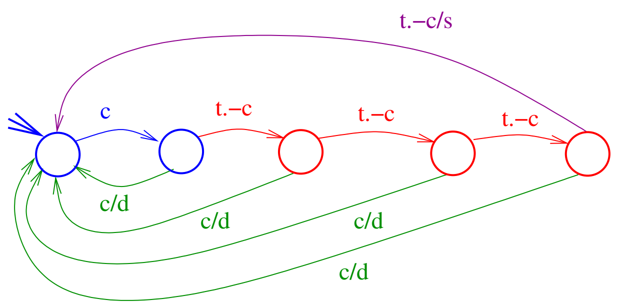
```
// close to regular
// in phy time
init
Each tick of  a timer do
      get inputs
      compute
      emit outputs
end do
```

The System

A mouse with one button. If you click twice "quickly", it means a double click, otherwise it's a simple click.

Signal names: c is the click input, d is the "double" output, s is the "simple" output, t is the additional input that counts time.

As a Mealy machine



In Lustre (not by Systematic Encoding)

```
const D : int = 10 ;

node doubleclick (button : bool) returns
                 (simple : bool ; double : bool) ;
var  foo, yaclick, reset : bool ;
     count : int    ;
let
        (foo, yaclick) = twostates (button, reset) ;
        count = if foo then 0 else pre(count) + 1 ;
        reset = (count = D) or button ;
        simple = (count = D) ;
        double = yaclick and button ;
tel.

node twostates (a, b : bool) returns (un, deux : bool) ;
let     un = true -> pre ((un and not a) or (deux and b)) ;
        deux = false -> pre ((deux and not b) or (un and a)) ;
tel.
```

Compilation into single-loop C code

When the program is compiled into single-loop C code:

- If there is an explicit input for time, connect it to the HW timer
- If there is no explicit input for time (we use the implicit base clock of the synchronous program to mean "time"), we use the programming scheme with `Each tick of a timer do`.

Demo with Luciole and the "real-time clock" option.

The System to be Programmed/Modeled

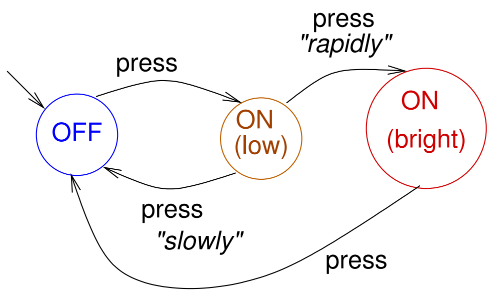
The lamp has three states: off, low, and bright. (The initial state is `off`). If the user presses a button `press`, then the lamp is turned `on`. If the user presses the button again, the lamp is turned `off`. However, if the user is *fast and rapidly presses the button twice*, the lamp is turned `on` and becomes `bright`.

Taken from "A Tutorial on Uppaal 4.0", Updated November 28, 2006.
Gerd Behrmann, Alexandre David, and Kim G. Larsen
Department of Computer Science, Aalborg University, Denmark

Towards a More Precise Specification...

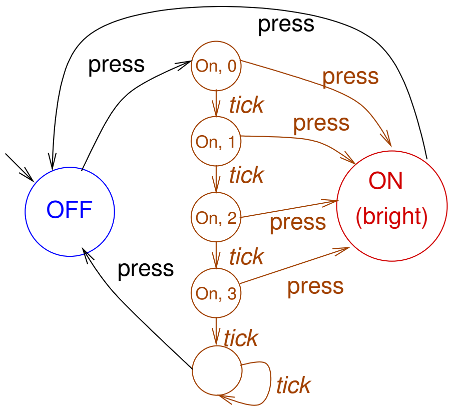
- *(What happens if the user presses the button when the lamp is "bright"?)*
- What is "rapidly"?
- What are the inputs and outputs of the system?
- How to count time?
- How would you implement such a system?

Towards an Automaton



We need to "measure" the time spent in state ON (between 2 "press"). Same idea as for the mouse double click.

A Synchronous Model or Program (Lustre or Explicit Automata)



Time is counted thanks to an additional input `tick`. Here, "rapidly" means: "(strictly) before the fourth tick occurs, after ON is entered".
+ A note on determinism and priorities...

A Synchronous Model or Program Trace Examples + Implementations

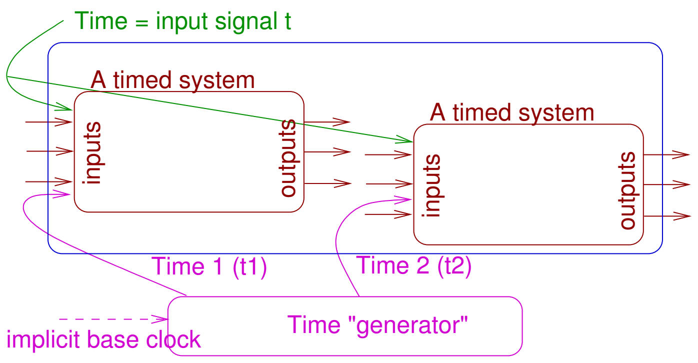
Trace Examples: Just to be convinced that the measure of time is necessarily unprecise.

Implementations:
1) as a synchronous program, connecting `tick` to a regular time basis
Exercice: encode the automaton into Lustre, use Luciole.
2) as a low-level C program with timers.

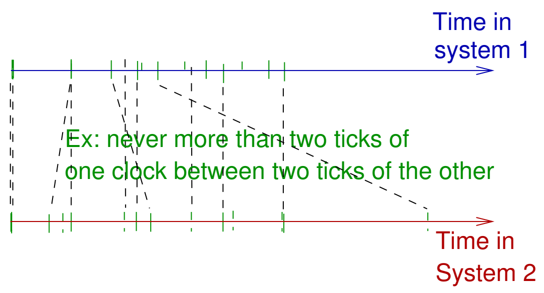
Rmk: The synchronous program is also a formal model of the system.

Composing (Synchronous) Models of Time

Recall time is only an additional input.



Composing (Synchronous) Models of Time


Time in system 1
Ex: never more than two ticks of one clock between two ticks of the other
Time in System 2

Unrelated Time Scales
Related Time Scales
Same time (not necessarily regular in physical time)

Composing (Synchronous) Models of Time

Same time is appropriate if you model:

- Several blocks in a synchronous HW circuit
- SW layers that make use of the same physical timer, and run on the same processor

Composing (Synchronous) Models of Time

Unrelated time scales are appropriate if you model:

- Machines on the internet without a clock synchronization protocol

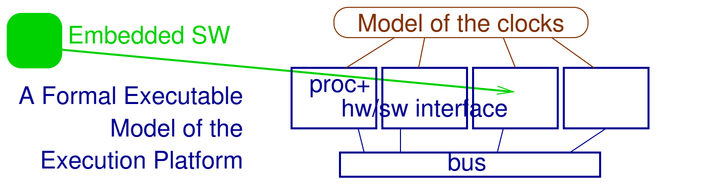
See also "pure" asynchronous models later.

Composing (Synchronous) Models of Time

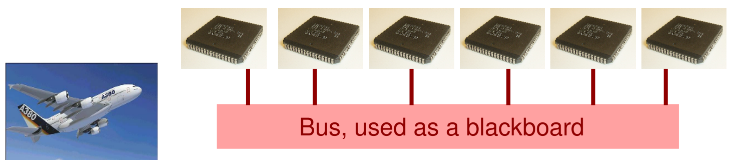
Related time scales are appropriate if you model:

- Asynchronous systems with a clock synchronization mechanism
- Several processors in a plane (no clock synchronization mechanisms, but the effect of clock drift during a flight can be safely assumed to be "bounded"). Example: the *quasi-synchrony* constraint.

Quasi-Synchronous Systems



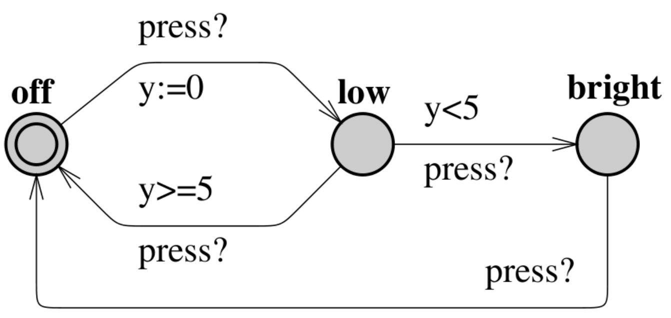
No synchronization between clocks
But: they don't differ too much



The Lamp Example Again

The lamp has three states: off, low, and bright. (The initial state is off). If the user presses a button press, then the lamp is turned on. If the user presses the button again, the lamp is turned off. However, if the user is fast and rapidly presses the button twice, the lamp is turned on and becomes bright.

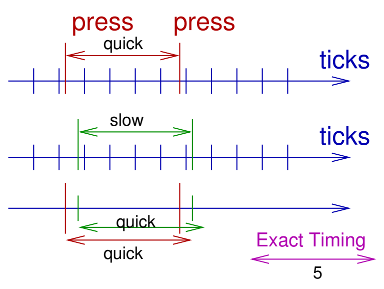
The Lamp Example With Explicit Time Counters



The Lamp Example - Timed Automata

- press? is a kind of input (more details to come...)
- y is called a *clock* in timed automata; it can be reset (y:=0) on a transition, and transitions can have a condition on its value ($y < 5$ or $y >= 5$).
- Time flows implicitly in states, transitions are instantaneous
- Time is continuous (or dense: values in $\mathbb{R}$).

The Lamp Example - Dense Time vs Tick Input



Is “exact timing” implementable?

General Definition of Timed Automata (With Invariants)

$TA = (L, \ell_0 \in L, C, A, E, Inv)$ where:
L is a set of locations (the states of the automaton)
ℓ_0 is the initial location
C is the set of clocks
A is a set of actions (more details later)
$E \subseteq L \times A \times \mathscr{B}(C) \times 2^C \times L$ is the set of edges (transitions)
$Inv : L \longrightarrow \mathscr{B}(C)$ assigns invariants to locations

$\mathscr{B}(C)$ denotes the possible conditions on the clocks of C

General Definition of Timed Automata (With Invariants)

- Conditions on the clocks: conjunctions of simple conditions of the form: $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$, c is a constant in $\mathbb{N}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$.
- Invariants attached to locations: conditions on clocks (same form as above) that should be true whenever the system is in the location. Can force evolutions of the automaton to avoid staying in a location "too long" (i.e., reaching a value of the clocks that no longer satisfies the condition).

Invariants: Examples

Typical case: $(\ell_1, a, x \geq 2, \emptyset, \ell_2)$ and $Inv(\ell_1) = x \leq 3$.
Not the same as: $(\ell_1, a, x \geq 2 \wedge x \leq 3, \emptyset, \ell_2)$

See also section 2.3 of: "A Tutorial on Uppaal 4.0", Updated November 28, 2006.
Gerd Behrmann, Alexandre David, and Kim G. Larsen
Department of Computer Science, Aalborg University, Denmark
`www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf`

Trace Semantics of Timed Automata (With Invariants) - 1

Notations:

- Clock Valuation: $u : C \longrightarrow \mathbb{R}^+$
- Set of all clock valuations: $\mathbb{R}^C$
- Initial value of clocks: $u_o(x) = 0, \quad \forall x \in C$
- $u \in Inv(\ell)$ means: u satisfies the invariant $Inv(\ell)$

Trace Semantics of Timed Automata (With Invariants) - 2

$TA = (L, \ell_0 \in L, C, A, E, Inv)$
The semantics is given by the transition system $(S, S_0, \longrightarrow)$ where S is the set of states (a subset of $L \times \mathbb{R}^C$), $s_0 = (\ell_0, u_0)$ is the initial state, and $\longrightarrow$ is the transition relation, a subset of $S \times (\mathbb{R}^+ \cup A) \times S$ defined by:

- $(\ell, u) \xrightarrow{d} (\ell, u + d)$ with $d \in \mathbb{R}^+$, if $\forall d' \in [0, d], u + d' \in Inv(\ell)$
- $(\ell, u) \xrightarrow{a} (\ell', u')$ if $\exists e = (\ell, a, g, r, \ell') \in E$ such that $u \in g, u' = [r \rightarrow 0]u, u' \in Inv(\ell')$

— $u' = [r \rightarrow 0]u$ is the clock valuation u where all clocks in r are reset, and all the others left unchanged.
— $u + d$ is the function: $\lambda x.u(x) + d$

Trace Semantics of Timed Automata (With Invariants) - 3

$TA = (L, \ell_0 \in L, C, A, E, Inv)$

For a global state made of: a location ℓ, and a valuation of the clocks u, there are two ways to "move":

- Let time d pass: the location does not change, all the clocks are incremented by d. Possible only if all the values between u and $u + d$ satisfy the invariant of the location ℓ.
- Take an instantaneous transition (ℓ, a, g, r, ℓ'): the guard g on clocks should be true for u, and as a result, the new location ℓ' is reached, the clocks in r are reset, and this makes a transition by action a.

Why Put Constraints on the Form of the Conditions?

Because Timed Automata are meant to be the input language of a model-checker, and with the constraints imposed on the clock conditions, the set of all states (locations and clock values) has good properties: it can be represented in a finite way.

Extension of Timed Automata with variables

In Uppaal, one can use additional variables (of almost any type) that can be tested and assigned to on transitions.

Composition of Timed Automata (in Uppaal)

In Uppaal, a network of timed automata are composed by a *rendez-vous mechanism* on action names, and all their clocks evolve according to the same global and continuous time.

Counting time in the various automata is therefore *perfect*. Not always suitable for modeling real systems in which sharing a global time is in most cases impossible.

Rendez-vous vs Synchronous Broadcast

Synchronous Broadcast: output signals in Mealy or Moore machines; the emitter can always proceed; any number of listeners can react to the signal. This is an asymmetrical synchronization principle, compatible with the idea of *inputs* and *outputs*; and time is an input.

Rendez-vous: This is a symmetrical synchronization principle; a process doing *a*! moves together with a process doing *a*?. If one of them is not ready to do its action, the other one cannot move.
a! and *a*? are neither inputs nor outputs, just paired names.

Using Rendez-Vous to Express Input/Output synchronization

On the "producer" side: use *a*!.
On the "consumer" side: make sure that there is a transition labeled by *a*? in each location (even if it's a loop with no effect); this makes sure that the producer is never blocked on *a*!.

(See also the notion of input-enabledness *in the input-output automata by Nancy Lynch).*
More details later, with asynchronous models.

Conclusion on Timed Automata and Uppaal

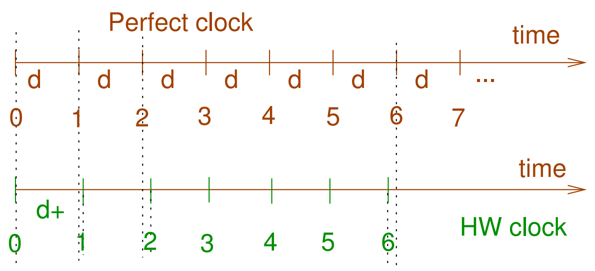
- Timed Automata: A very nice and clever idea: a model with dense time that can be regarded as a finite-state model; tools for model-checking properties.
- Uppaal: A complete tool for the design and analysis of timed automata; a not so nice modeling language, using rendez-vous instead of more natural inputs/outputs, and a perfect global time shared by all parts of a model. To be used carefully to avoid biaises in the models.

Source

Modelling Clock Synchronization in the Chess gMAC WSN Protocol
Mathijs Schuts, Feng Zhu, Faranak Heidarian, Frits Vaandrager
Institute for Computing and Information Sciences; Radboud University Nijmegen
Workshop on Quantitative Formal Methods: Theory and Applications (QFM'09) EPTCS 13, 2009, pp. 41-54

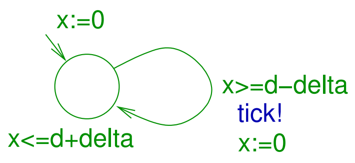
A First Remark on HW Clocks

HW Clocks are never perfect.
A HW clock is a mechanism meant to "tick" at regular instants $n \times d, n \in \mathbb{N}$, in physical time, but the physical time between two successive ticks is in fact in an interval $[d - \delta, d + \delta]$.



Modeling Time and Clocks

When modeling a clock synchronization protocol, it's very important not to use a model in which clocks are intrinsically perfectly synchronized! But the clocks in a network of timed automata all describe the same "perfect" global time... How to overcome the problem?



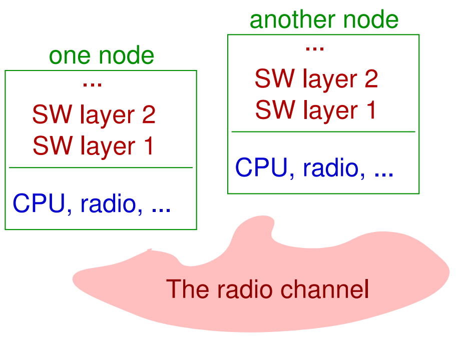
The implicit time of timed automata corresponds to physical time; tick! represents an imperfect HW clock; it is "emitted" with successive intervals $[d - \delta, d + \delta]$.

Time in a Node

One node of the sensor network is modeled by a network of 5 timed automata, synchronized with the rendez-vous mechanism on actions like `start_sending`. Fig 4. resembles a dataflow diagram, but the synchronizations are in fact symmetrical. To mimick the behavior of inputs/outputs, remember that the consumers (on the right side of dataflow arrows) have to be *input-enabled*.

Observe Fig 6. What about sending time, transmission delays, switching time in the radio? Is it legitimate to use the same "time" (represented by ticks) in the 5 automata?

All the Notions of Time Involved...



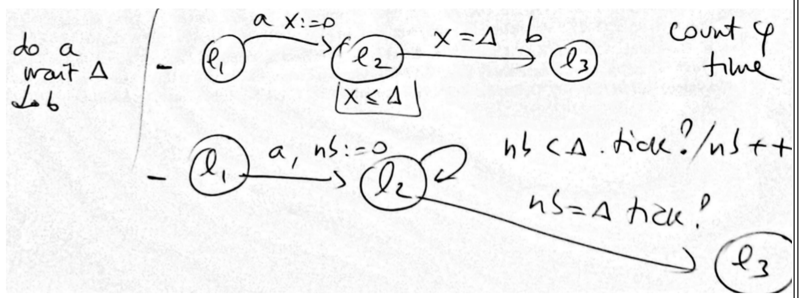
- Modeled Physical time: e.g., transmission delay, switching time in the radio, ...
- Clock of the CPU
- Time as seen by SW layers via timers

Potential Biases in Models with Time

- A model of the SW in which we count on the physical time scale (this would mean that the SW has access to a perfect measure of time)
- A model of the distributed system in which all the clocks tick at the same rythm (this would mean that we consider perfect HW clocks)
- Any model of the HW too "friendly" for the SW (the model of the HW should be *conservative*, i.e., the worst thing that can happen in reality should also happen in the model; it might be the case that the SW does not work with the model of the HW, but would work on some specific real HW, though).

— What about the model described in the paper on gMAC?
— How to model a SW timer?

Potential Biases in Models with Time
Example: SW Counting real physical time

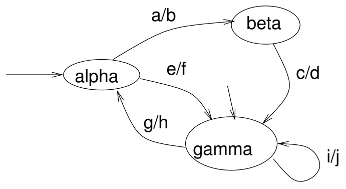


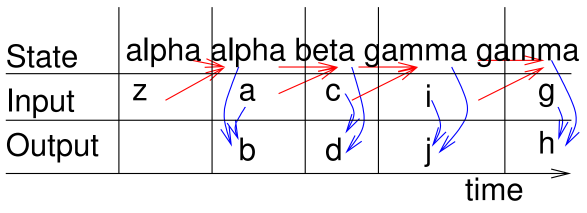
Simple Mealy machines

$M = (Q, Q_0 \subseteq Q, T \subseteq Q \times I \times O \times Q)$ where
- I (resp. O) is the set of inputs (resp. outputs)
- Q is the set of states
- Q_0 is the set of initial states
- T is the set of transitions, labeled by tuples (input,output)

Drawings: $q \overset{\text{input/output}}{\longrightarrow} q'$

Trace Semantics





State	alpha	alpha	beta	gamma	gamma
Input	z	a	c	i	g
Output		b	d	j	h

time

Imperative Program

```
State := alpha  -- take one of the initial states
while (true) loop
  get (I) ; O := null ;
  case State :
     when alpha  =>  if I = a then State := beta ; O := b ;
                     elsif I = e then ...
     when beta   =>  if I = c then State := gamma ; O := d ;
     when gamma  =>  if I = i then O := j
                     elsif I = g then O:=h ; State := alpha
  end case
  put (O)
end loop
```

Boolean Mealy Machines

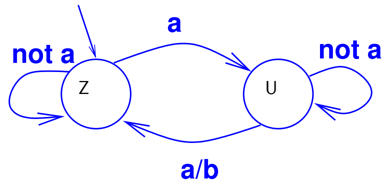
Same definition, but with $I = 2^{\text{BI}}$ and $O = 2^{\text{BO}}$.

BI : input signals, any configuration is possible.
BO : output signals, any configuration is possible.

On the "2 lights" example: a.-b/LightA_on

(read: if a is pressed and not b, then switch lightA on)

Example Mealy machine : one 'b' every two 'a's



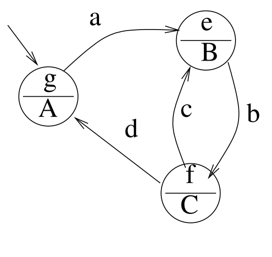
Simple Moore Machines

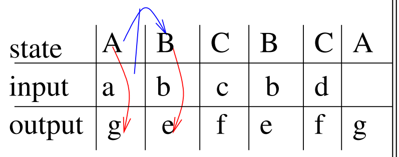
$M = (Q, Q_0 \subseteq Q, T \subseteq Q \times I \times Q, F : Q \longrightarrow O)$ where

- I (resp. O) is the set of inputs (resp. outputs)
- Q is the set of states
- Q_0 is the set of initial states
- T is the set of transitions, labeled by inputs
- $F : Q \longrightarrow O$ associates an output with each state

Drawings: $q(\text{output}) \overset{\text{input}}{\longrightarrow} q'(\text{output})$

Trace Semantics





state	A	B	C	B	C	A
input	a	b	c	b	d	
output	g	e	f	e	f	g

Imperative Program

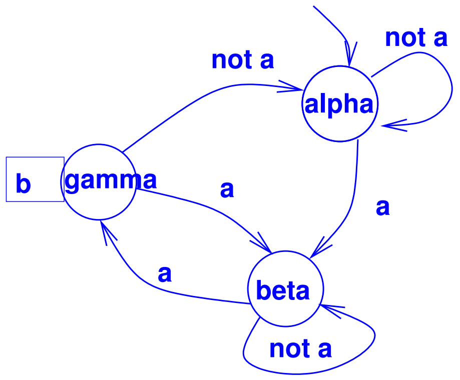
```
State := A  -- take one of the initial states
while (true) loop
    get (I) ; O := null ;
    case State :
        when A : O := g ;
                 if I = a then State:=B ;
        when B : O := e ;
                 if I = b then State:=C;
        when C : O := f ;
                 if I = c then State:=B;
                 elsif I = d then State:=A ;
    end case
    put (O)
end loop
```

Boolean Moore Machines

Same definition, but with $I = 2^{\text{BI}}$ and $O = 2^{\text{BO}}$.

BI : input signals, any configuration is possible.
BO : output signals, any configuration is possible.

Example Moore machine : one 'b' every two 'a's


not a
not a
alpha
b
gamma
a
a
a
beta
not a