Modelling Timed Systems

Natalia Sidorova

What is time?

Webster’s Collegiate Dictionary:

“time is the measured or measurable period during which an action, process, or condition exists or continues."

Folk definition of timed systems

Time-critical systems are systems in which correctness depends:

- not only on the logical result of the computation, but
- also on the time at which the results are produced.

What are timed systems?

Computer systems where the timing of input and output is relevant for its correctness.

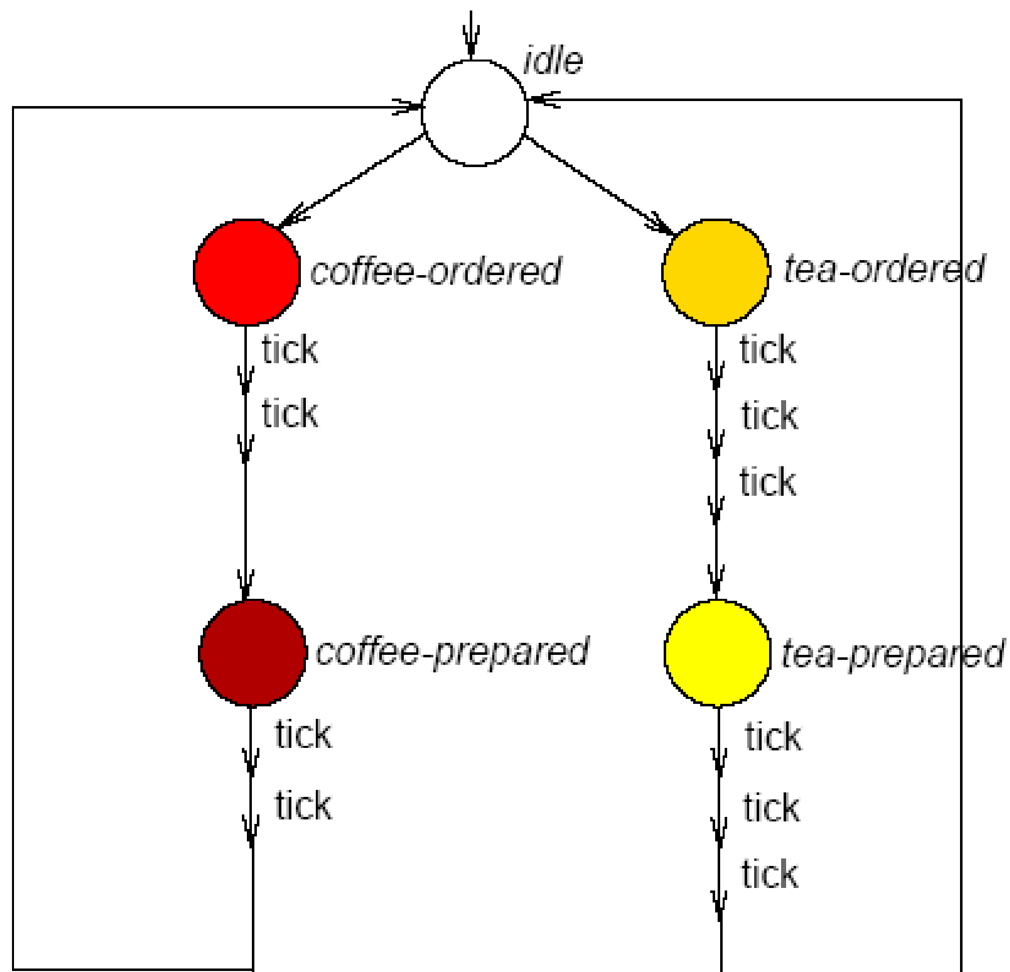
- Hard real time systems: hard deadlines
- Soft real-time systems: average time constraints
- Safety critical systems: missing a deadline might have disastrous effects.
- Time easily enters development process too early as a solution, especially in specification.
  Related question: are special timed formalisms needed, or untimed ones are sufficient?

Discrete time

A discrete time-domain:

- time has a discrete nature, i.e., time is advanced by discrete steps
- time is modelled by non-negative integers
- specific tick events are used to model the advance of one time unit
- events can only happen at integer time values
- delay between any two events is always a multiple of the minimal delay of one time unit

A discrete-time coffee machine



A discrete time-domain

Main advantage is conceptual simplicity:

- just one new event needed to measure time in model (tick-event)
- no need to adapt temporal logic, as next-operator "measures" time
  - $\Box(red \Rightarrow (\bigcirc \bigcirc green))$
    exactly 2 time-units after red, the light is green
  - $\Box(red \Rightarrow (green \wedge \bigcirc green \wedge \bigcirc \bigcirc green))$
    within 2 time-units after red, the light is green

A discrete time-domain

Main application areas:

- systems in which components proceed in a lock-step fashion, i.e., at each tick each component performs a step for example, synchronous hardware circuits
- systems where timers are only used to model timeouts

Limitations of discrete time

- (minimal) delay between any pair of events is some multiple of an a priori fixed minimal delay
- difficult (or impossible) to determine this in practice
- can be inadequate for asynchronous systems such as distributed systems and communication protocols

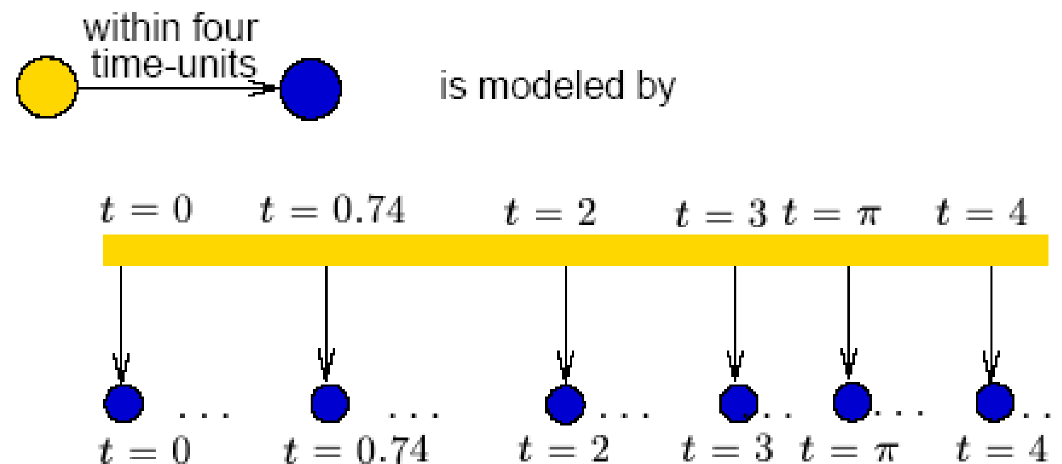
Dense time

A continuous time-domain

- time has a continuous nature, i.e., time advances continuously
- time is modelled by real numbers
- delay between two events can be arbitrarily small
- invariance against changes of time scale
- often, more adequate representation of reality
- often, more suited for asynchronous systems
- but . . . more complicated too!

Dense time

If time is continuous, state changes can happen at any point in time

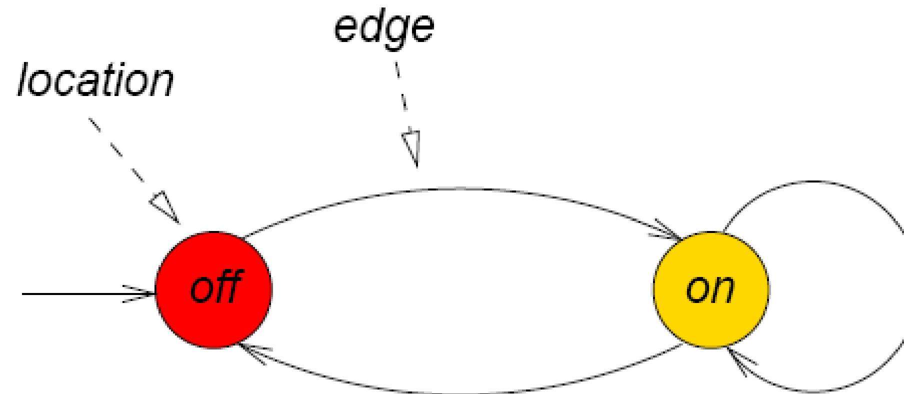


how to check a property like once in a yellow state, eventually the system is in a blue state within π time-units?

How to cope with these problems?

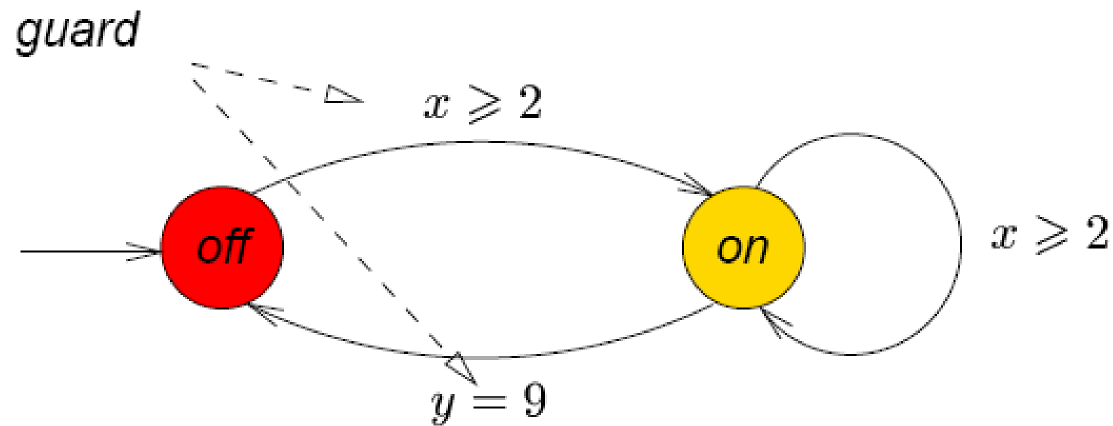
- restrict expressivity of the property language
  (= temporal logic)
- model behaviour of timed systems symbolically
  rather than explicitly (= infinite-state)
  (see Timed Automata [Alur & Dill, 1989])
- realise a discretisation of the infinite (underlying)
  state space on-demand, i.e., depending on the
  property and model under consideration
  (see Region Automata [Alur et al., 1991])

What is a timed automaton?



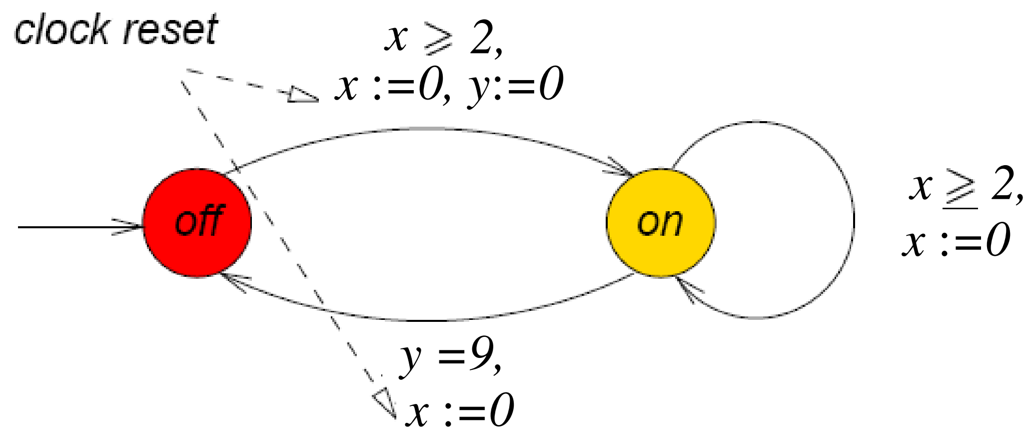
- a finite-state automaton with locations and edges
- a location is labelled with atomic propositions that are valid in that location
- taking an edge is instantaneous, i.e, consumes no time

What is a timed automaton?



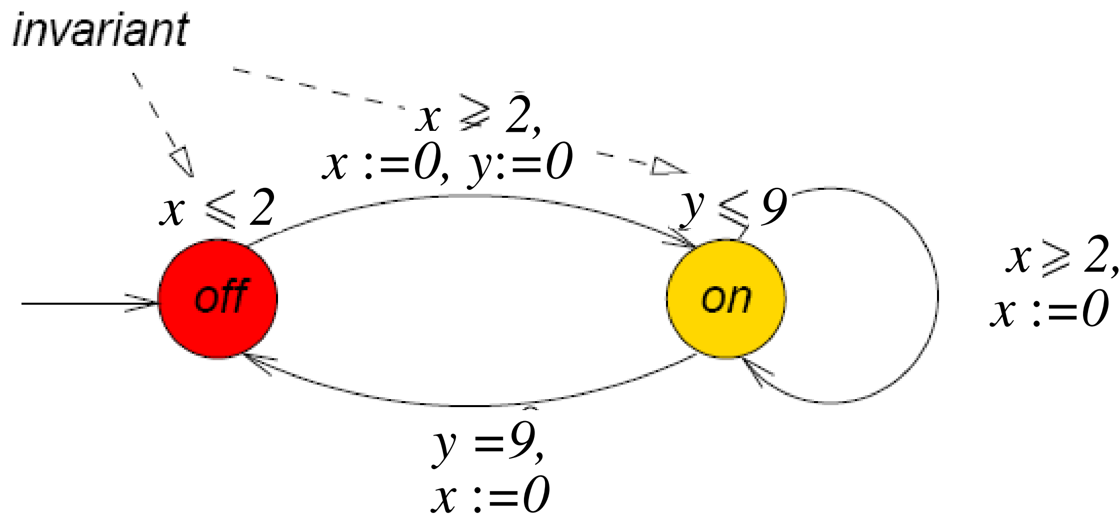
- equipped with real-valued clocks $x, y, z, \ldots$
- clocks advance implicitly, all at the same speed
- logical constraints on clocks can be used as enabling conditions (called guards) of edges

What is a timed automaton?



- clocks can be reset when taking an edge
- assumption: all clocks start when entering the initial location initially

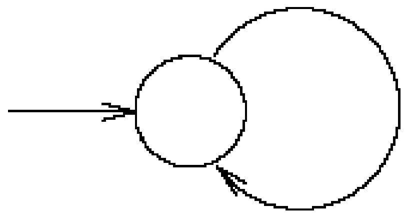
What is a timed automaton?

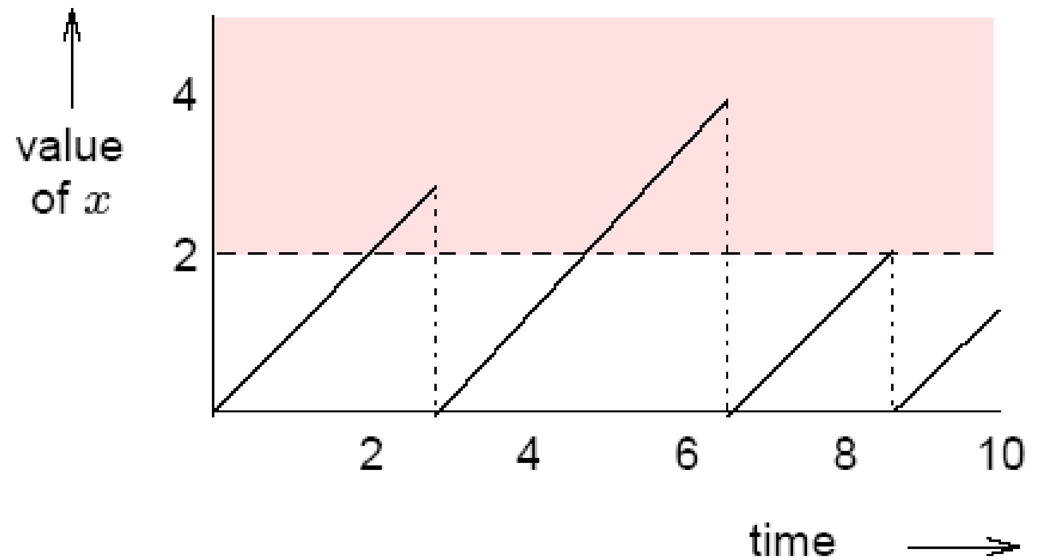


- guards indicate when an edge may be taken, (location) invariants are used to force an edge to be taken
- an invariant is a clock constraint that specifies the amount of time that may be spent in a location

Guards vs. invariants

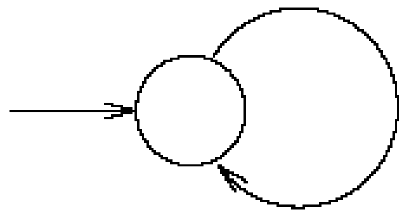
The effect of a lowerbound guard:



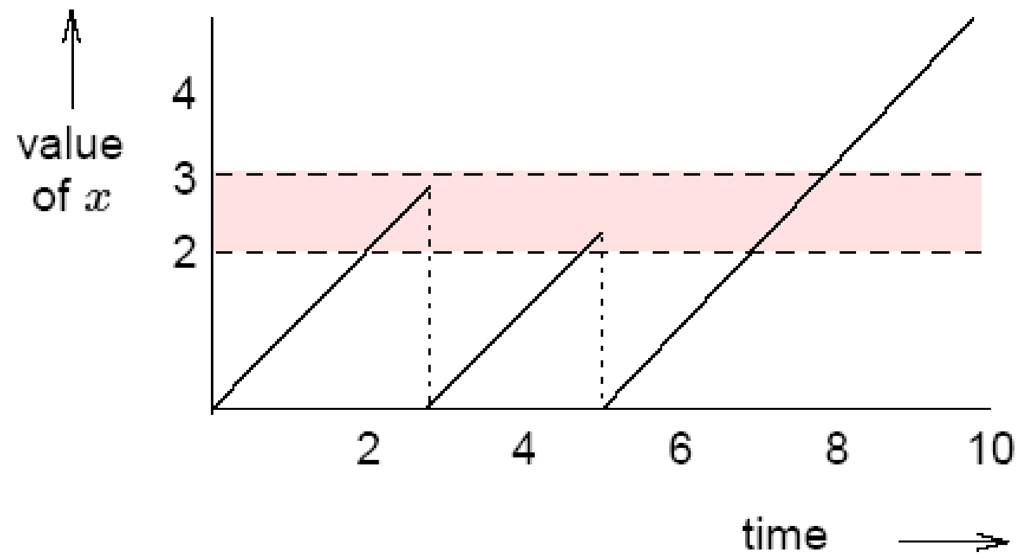


Guards vs. invariants

The effect of a lowerbound and upperbound guard:

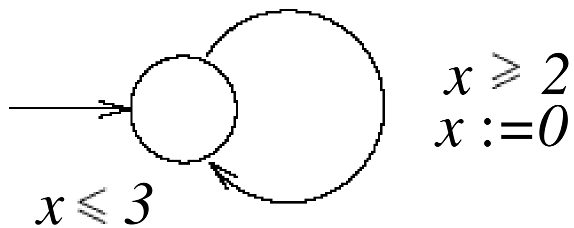


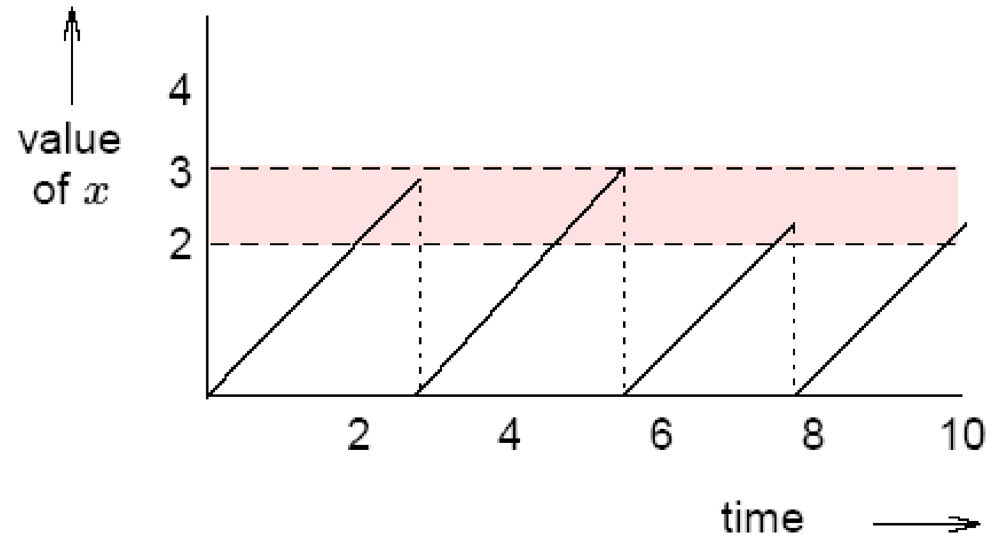
$2 \leqslant x \leqslant 3$
$x := 0$



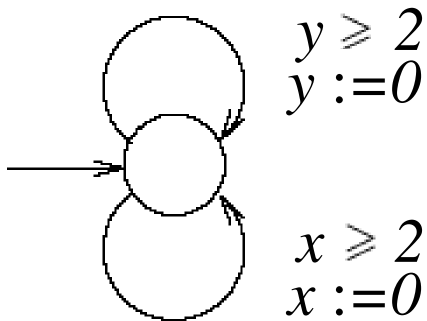
Guards vs. invariants

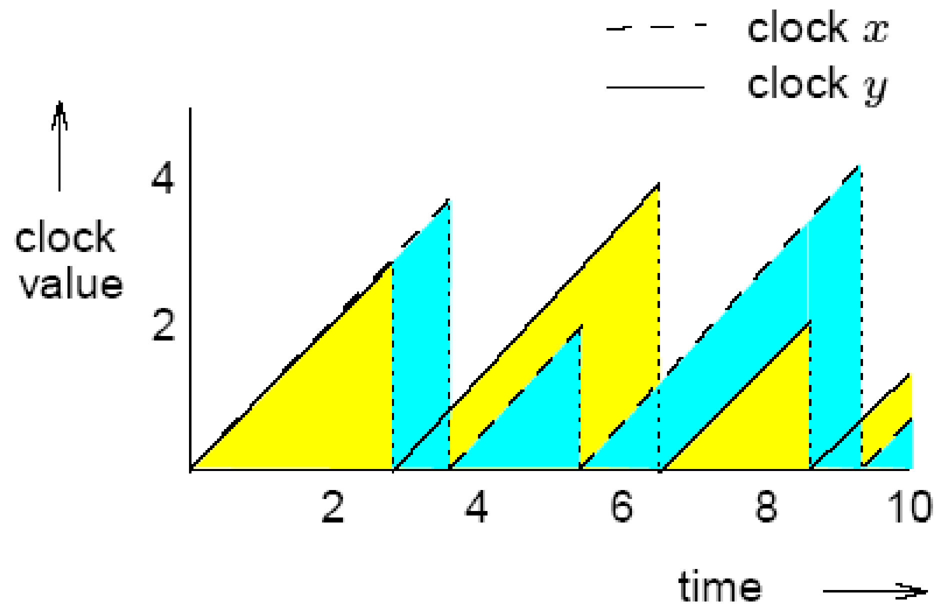
The effect of a guard and an invariant:





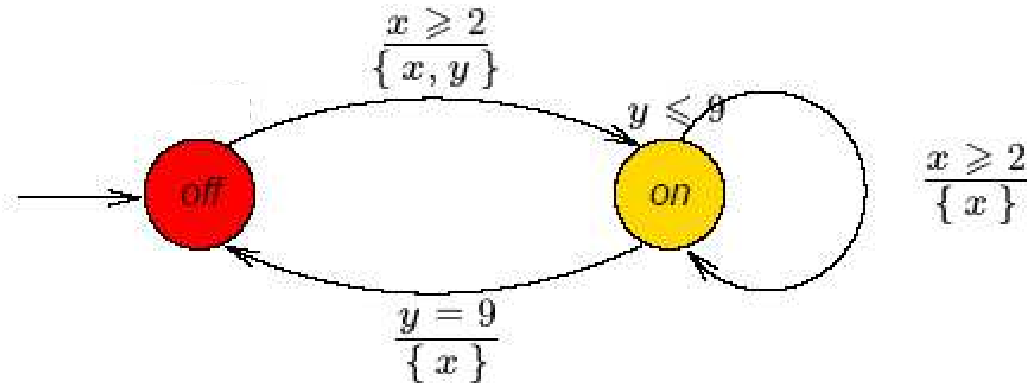
Arbitrary clock differences





In a discrete-time model, the difference between two concurrent clocks is always a multiple of one unit of time. That's not the case with dense time.

Some example paths

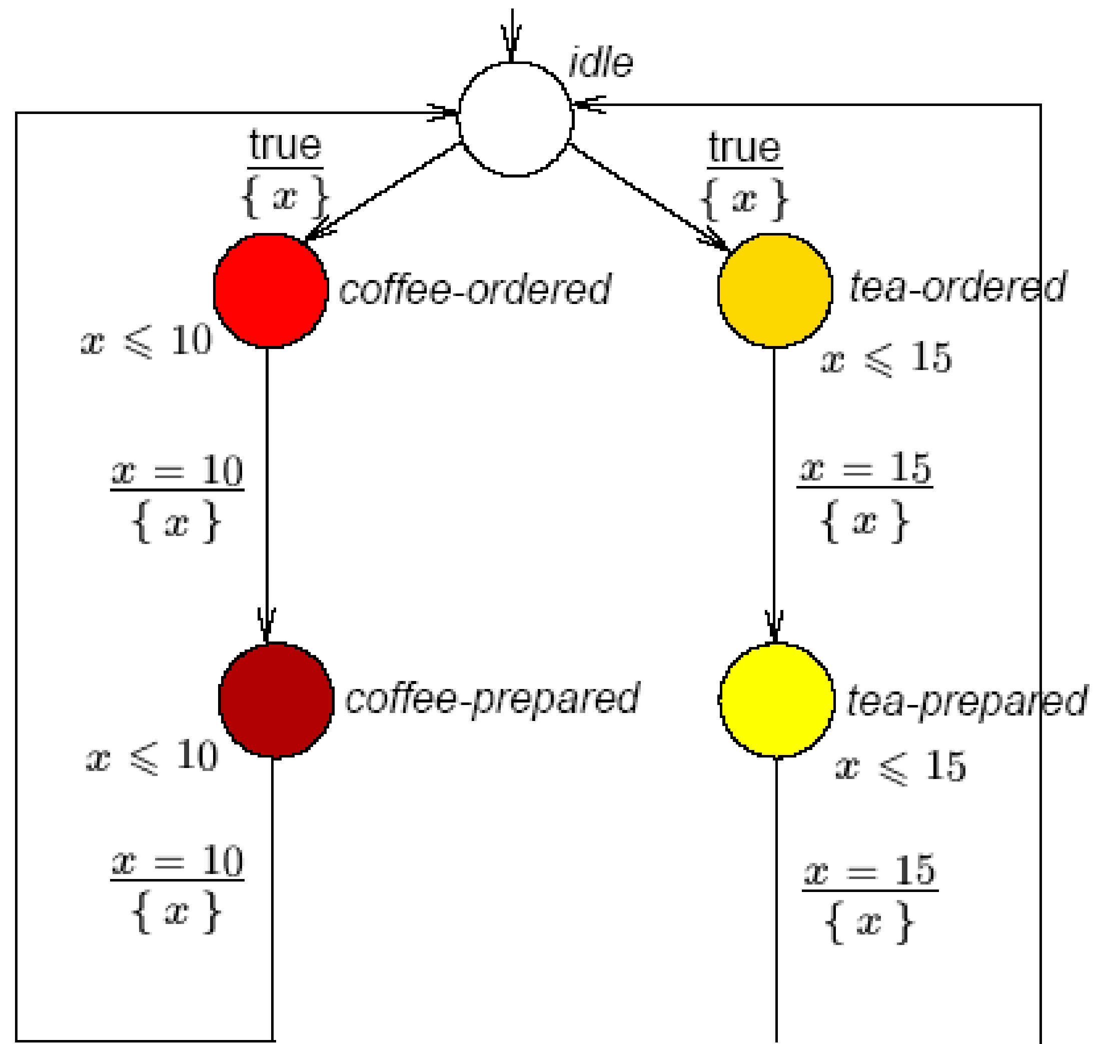


$$(\mathit{off}, v_0) \xrightarrow{3} (\mathit{off}, v_1) \xrightarrow{*} (\mathit{on}, v_2) \xrightarrow{4} (\mathit{on}, v_3) \xrightarrow{*} (\mathit{on}, v_4)$$

$$\xrightarrow{1} (\mathit{on}, v_5) \xrightarrow{2} (\mathit{on}, v_6) \xrightarrow{2} (\mathit{on}, v_7) \xrightarrow{*} (\mathit{off}, v_8) \ldots$$

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
	0	$+3$	$x, y := 0$	$+4$	$x := 0$	$+1$	$+2$	$+2$	$x := 0$
x	0	3	0	4	0	1	3	5	0
y	0	3	0	4	4	5	7	9	9

A dense-time coffee machine



Back to discrete time

A less expensive, discrete time solution is for many systems as good as dense time in the modelling sense, and better than the dense one when the verification is concerned.

It suffices for a large and important class of systems and properties including all systems that can be modelled as timed transition systems and such properties as time-bounded invariance and time-bounded response.

Discrete time automata can be analysed using any representation scheme used for dense time, and *in addition* can benefit from enumerative and symbolic techniques which are not naturally applicable to dense time.

Clocks of timers?

Clocks: growing when time progresses, possibly to ∞.

Timers: decreasing when time progresses, until expiration.

Expiration of a timer is a natural way to model e.g. a trigger for a software event.

If this event is to be handled exactly once,
with taking an event guarded by a timer condition,
the timer which triggered this event should become deactivated.

Time semantics

We consider systems where delays are significantly larger than the duration of normal events within the system.

$\Rightarrow$ we assume system transitions to be instantaneous.

$\Rightarrow$ time progress can never take place if there is still an untimed action enabled

$\Rightarrow$ the time-progress transition has the least priority in the system and may take place only when the system is blocked: there is no any transition enabled except for time progress and communication with the environment.

Introducing discrete time in Spin

is based on the use of timeout — a system defined condition.

Timeout is a predefined global read-only variable, that has the value true in all global system states where no statement is executable in any active process, and false in all other states.

We assume all “normal” actions to be instantaneous. expire(timer_name) is used as a guard for some actions. We define the special process Timers, which decreases all active timers by 1 when there is no any action enabled. It does so until some timer gets expired, which enables some “normal” action.

Introducing discrete time in Spin

```
#define OFF -1 /* inactive timer*/
#define EXP 0 /* expiration point*/

typedef timer
{
short val = OFF;
}

#define set(tmr,value) tmr.val = value;
#define reset(tmr) tmr.val = OFF;

#define expire(tmr) (tmr.val == EXP) /*timeout*/
#define tick(tmr) if :: tmr.val != OFF -> tmr.val=tmr.val -1;\
                     :: tmr.val == OFF-> skip; fi
```

backslash \ at the end of a line in #define ... means continuation of the macro; otherwise the macro is supposed to consist of a single line

Introducing discrete time in Spin

Define timers as global variables:

```
timer mc_timer; /* timer for the message channel */
timer ac_timer; /* timer for the ack channel */
timer s_timer; /* timer for the sender */
```

Specify a process that ticks down all active timers:

```
active proctype Timers()
{
do
   :: timeout -> atomic{tick(mc_timer); tick(ac_timer);
                              tick(s_timer); };
od
}
```

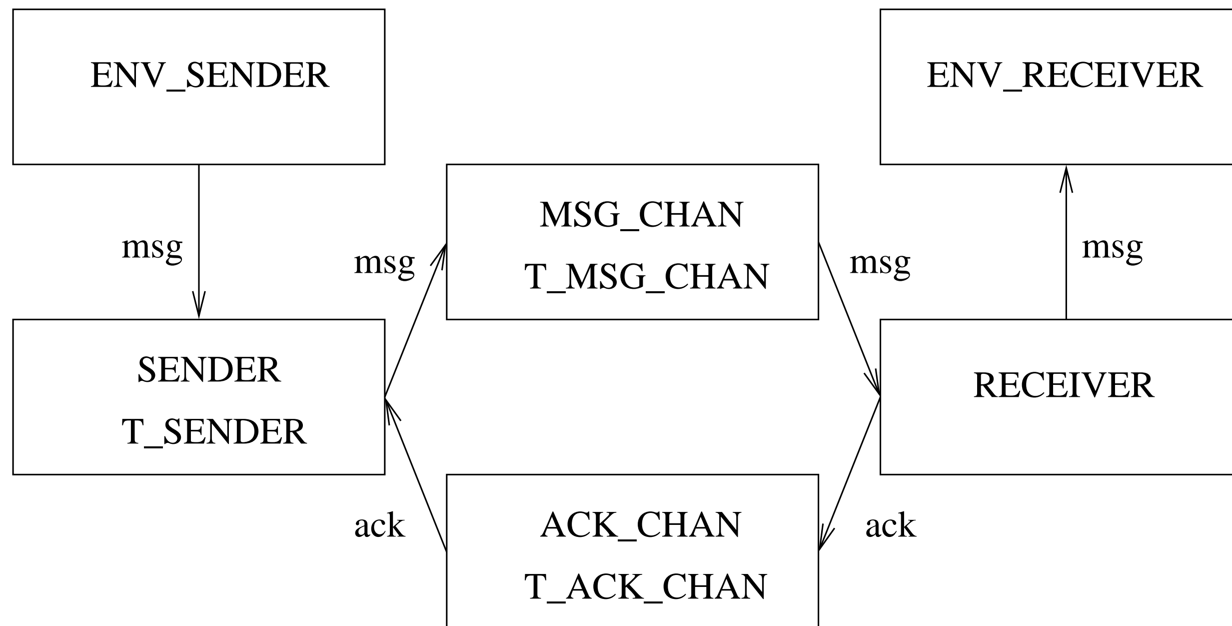
Positive Acknowledgment with Retransmission protocol

Positive Acknowledgment Retransmission protocol: PAR (Tanenbaum: Computer Networks) is used in the Transmission Control Protocol (TCP).

- PAR involves a sender, a receiver, a message channel and an acknowledgment channel.
- The sender receives a frame from the upper layer, sends it to the receiver via the message channel and waits for a positive acknowledgment from the receiver via the acknowledgment channel.
- When the receiver gets the frame, it delivers it to the upper layer, and sends the acknowledgment to the sender.
- After the positive acknowledgment is received, the sender becomes ready to send the next frame.

Positive Acknowledgment with Retransmission protocol

- The channels delay the delivery of messages.
- They can lose or corrupt messages.



Positive Acknowledgment with Retransmission protocol

- The sender handles lost frames by timing out. If the acknowledgement is not received in certain time, the sender re-sends the message. This process is repeated until an acknowledgement is received by the sender.
- Potential problem: the receiver receives both the original frame and the re-transmitted frame.
  Solution: The sender sends a sequence number with every message. Each frame has a unique sequence number. The re-transmitted frame(s) carry the same sequence number as the original one.
- We assume the transmission order is preserved, i.e. the early transmitted frames will arrive earlier than later transmitted ones. Thus only one bit sequence number is needed.

Modelling PAR

Components: Sender, Receiver, Message Channel, Acknowledgement Channel, Environment.

Channels are lossy
$\Rightarrow$ we model them as “channel plus process plus channel” combination, where the process decides whether to deliver message properly or not.

Modelling PAR: fragment

```
/* delays */
#define mc_delay 3
#define ac_delay 4
#define sender_delay 5

/* message type */
mtype={a, b, c, ack}

/* internal channels */
chan S2MC = [2] of {mtype, bit}
chan MC2R = [2] of {mtype, bit}
chan R2AC = [2] of {mtype}
chan AC2S = [2] of {mtype}

/* external channels */
chan IN = [2] of mtype
chan OUT = [2] of mtype
```

Modelling PAR: fragment

```
/* message channel */
active proctype MChannel(){
mtype msg;
bit sn;

get_msg: S2MC?msg,sn -> set(mc_timer, mc_delay); goto msg_wait;

msg_wait:
atomic{
if
:: expire(mc_timer) -> MC2R!msg,sn; goto get_msg;
:: expire(mc_timer) -> goto get_msg;
fi }
}
```

Modelling PAR: fragment

```
/*Environment process supplying messages to the IN channel*/

active proctype Env1(){

A: atomic{IN!a; goto B;}

B: atomic{IN!b; goto C;}

C: atomic{IN!c; goto A; }
}

/*Environment process reading messages from the OUT channel*/

active proctype Env2(){

A: atomic{OUT?a; goto B;}

B: atomic{OUT?b; goto C;}

C: atomic{OUT?c; goto A; }
}
```

Properties of interest (examples)

- No deadlock happens
- No overflow for internal buffers
- Every message sent is eventually delivered (under the fairness assumption on the channel behaviour)
- ...

Homework for this week

- Finish the Promela model of PAR
- Investigate what the setting for the Sender's timer should be, depending on the given channels' delays, to guarantee the proper functioning of the protocol (at least, properties from the previous slide should hold). Give examples of "good" and "bad" settings.
- Which inequation characterises "good settings" from your point of view?
  (check it for some boundary case)

Soldiers problem (1)

Four soldiers who are heavily injured, try to flee to their home land. The enemy is chasing them and in the middle of the night they arrive at a bridge that spans a river which is the border between the two countries at war. The bridge has been damaged and can only carry two soldiers at a time. Furthermore, several land-mines have been placed on the bridge and a torch is needed to sidestep all the mines. The enemy is on their trail, so the soldiers know that they have only 60 minutes to cross the bridge. The soldiers only have a single torch and they are not equally injured.

Soldiers problem (2)

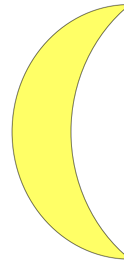
The following table lists the crossing times (one-way!) for each of the soldiers:

soldier S_0 5 minutes
soldier S_1 10 minutes
soldier S_2 20 minutes
soldier S_3 25 minutes

Does a schedule exist which gets all four soldiers to the safe side within 60 minutes?

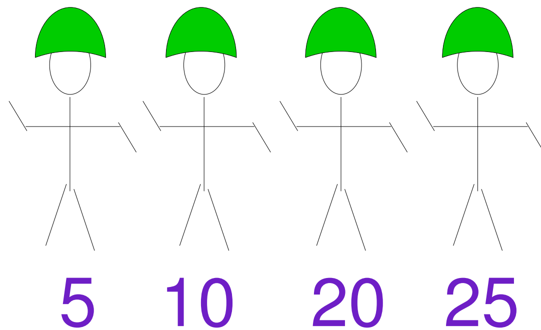
(Ruys & Brinksma 1998)

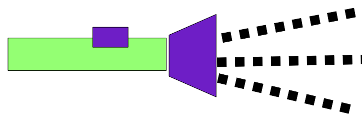
Soldiers problem

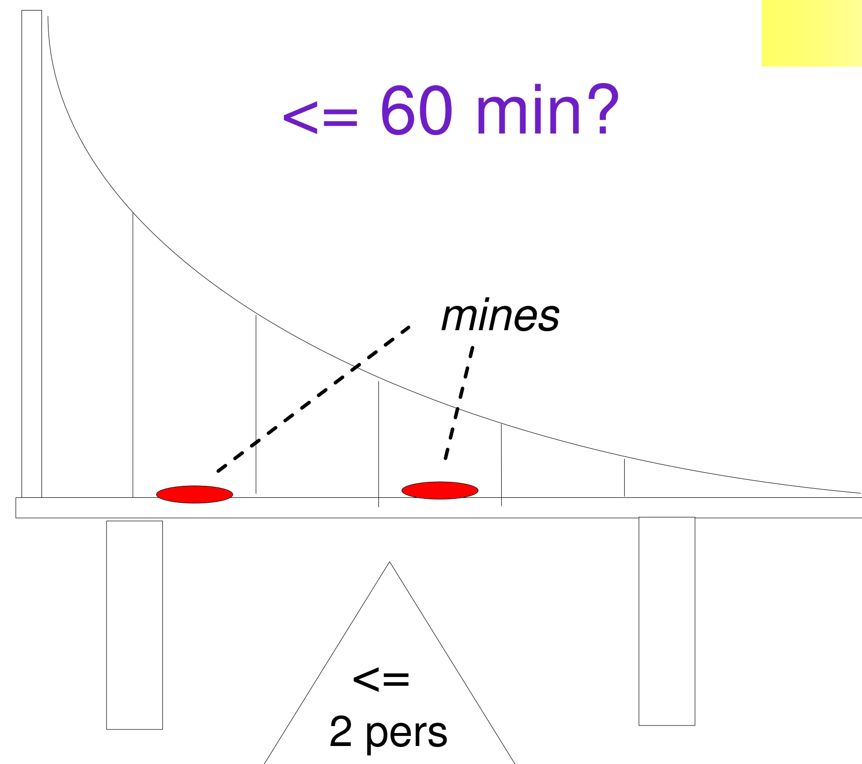


Unsafe Side

Safe Side







A task from Assignment 2

- Build a Promela model describing all possible behaviours of soldiers
- Define a property postulating that there exists a trace such that all four soldiers reach the safe side. (we will check that its negation gets violated when working with LTL!)
- Find a solution ("good trace") to the problem using Spin.

Next lecture:

process equivalences,

Spin tutorial (continued)