Modelling with Timed Automata in UPPAAL

02224 Real-time Systems

February 22 2023

Hans Henrik Løvengreen

DTU Compute

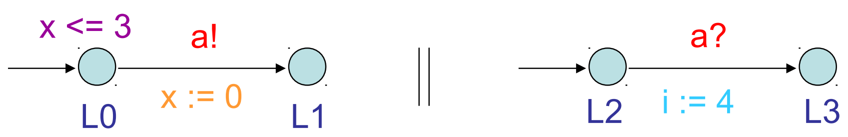
Contents

- Timed Autamata in UPPAAL (recap)

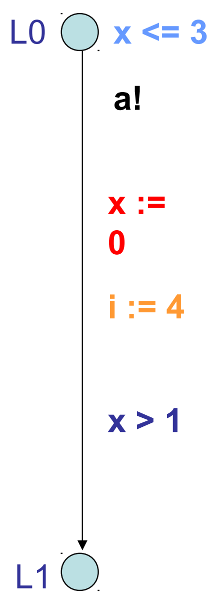
•Advanced modelling in UPPAAL:
  - Urgent and committed locations
  - Urgent and broadcast channels
  - Variable reduction
  - Synchronous value passing
  - Atomicity
  - Urgent edges
  - Timers
  - Bounded liveness checking

Modelling Timed Automata in Uppaal

- A timed automaton in Uppaal is modelled using locations and edges between locations
- A template may be used to declare a generic automaton that can be instantiated with a parameters list
- A system consists of a network of timed automata: A1 || A2 ||…|| An, where Ai is a timed automaton
- Two automata may communicate via synchronization channels or via global variables



Invariants, Assignments, Channels and Guards



Location invariants

$\boxed{inv ::= x < n \mid x \leq n \mid inv, inv}$

clock natural number and

Clock assignment

$\boxed{x := n}$

Channel synchronization

$\boxed{Chan ::= a! \mid a?}$

Data assignment

$\boxed{i := Expr}$

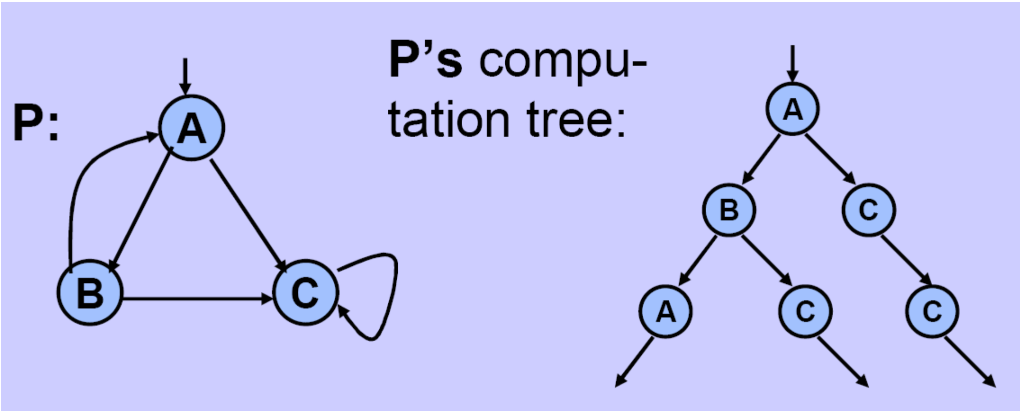
Guards

$$g ::= g_c \mid g_d \mid g, g$$
$$g_c ::= x \prec n \mid x \prec y + n$$
$$g_d ::= Expr\ op\ Expr$$
$$\prec \in \{<, <=, =, >=, >\}$$
$$op \in \{<, <=, =, >=, >, != \}$$

$$Expr ::= i \mid n \mid i[Expr]$$
$$\mid -Expr \mid\mid Expr \otimes Expr$$
$$\mid (g_d\ ?\ Expr : Expr)$$
$$\otimes \in \{+, -, *, /\}$$

Specification Language

- A subset of Timed Computation Tree Logic (TCTL)
- Formulae refer to the computation tree of the system
- Example:

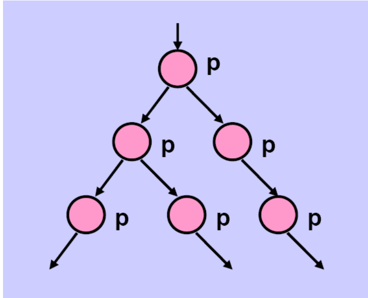


Specification Language

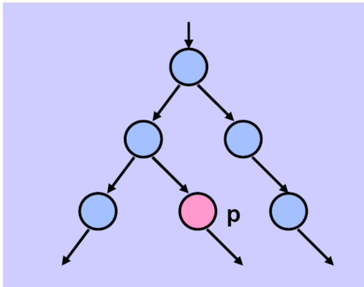
- Reachability properties:
  - Something good will possibly happen
  - “Does **p** hold in some state along some path?”
- Safety properties:
  - Something bad will never happen
  - “Does **p** hold in all states along all paths?”
- Liveness properties:
  - Something good will eventually happen
  - “Does **p** hold in some state along all paths?”

Verification: A[] p and E<> p

- **A[] p** – p is invariant
- p is true in all reachable states



- **E<> p** – p is reachable
- It is possible to reach a state in which p is satisfied
- p is true in (at least) one reachable state

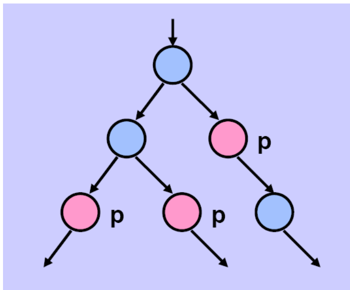


- **A[]** p = ¬ **E<>** ¬p

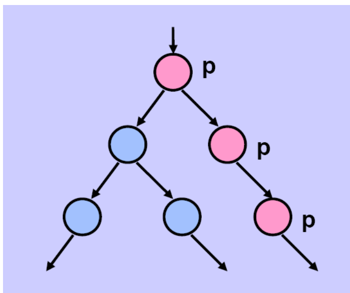
DTU

Verification: A<> p and E[] p

- **A<> p** – p will inevitably become true
- The system is guaranteed to eventually reach a state in which p is true
- p is true in some state of all paths



- **E[] p** – p is potentially always true
- There exists a path in which p is true in all states

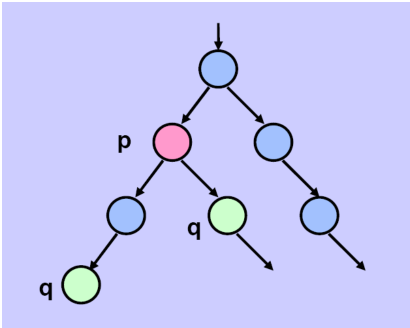


- **A<>** p = ¬ **E[]** ¬p

DTU

Verification: p --> q (p leads to q)

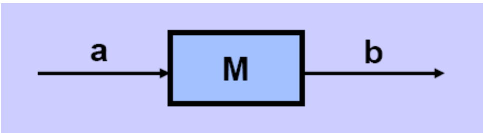
- **p --> q** – if p becomes true then q will inevitably become true
- In all paths, if p becomes true, q will inevitably become true
- **p --> q** = **A[]** (p imply **A<>** q)



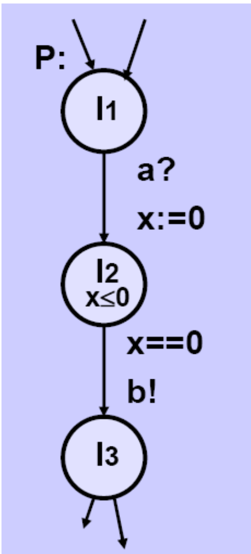
DTU

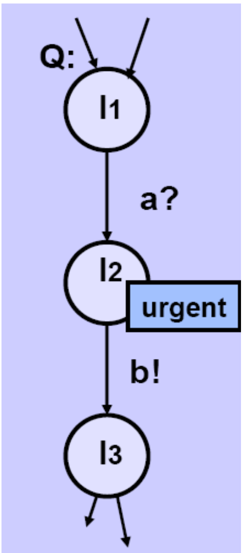
Urgent Locations

- We want to model a simple media M that receives packages on channel a and immediately sends them on channel b:



- P models the media using the clock x
- Q models the media using an **urgent location** and has the same behavior
- Informal Semantics: There will be no delay in an urgent location
- Note: Urgent locations reduce the number of clocks, and thus simplifies the analysis

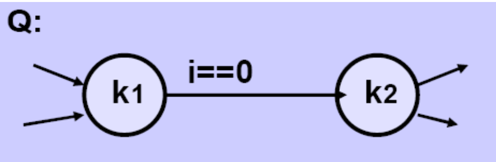




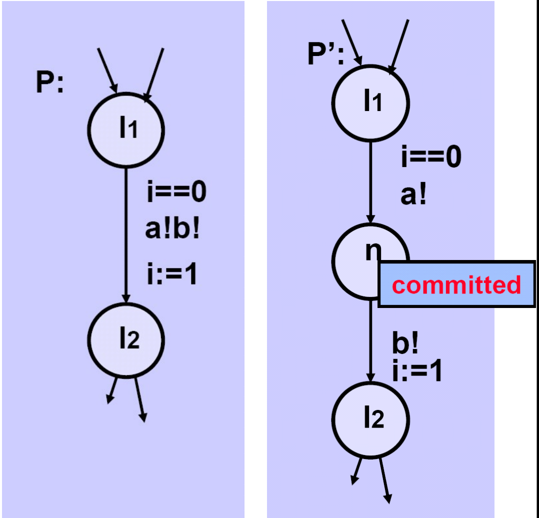
DTU

Committed Locations

- We want to model a process P that simultaneously sends messages a and b to two receiving processes when i=0
- P’ sends messages at the same time, but in location n, another automaton (e.g., Q) may interfere:



- **Solution**: mark n as **committed** instead of urgent
- Informal semantics: There will be no delay in committed location, and the next transition must involve an automaton in a committed location
- Note: committed locations reduce the number of clocks, and simplifies the analysis

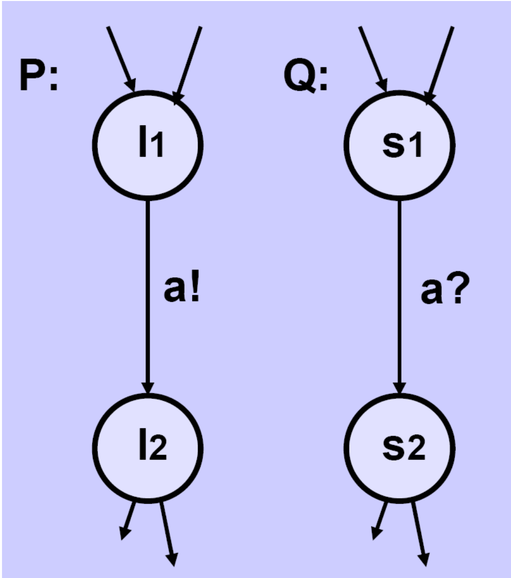


Urgent and Committed Locations

- Difference:
  - In a committed location, an out-going transition must be taken before anything non-committed can happen
  - In an urgent location, anything can happen as long as it takes no time
- A location cannot be declared both urgent and committed
- An committed location is implicitly urgent

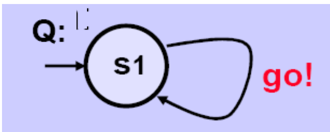
Urgent Channels

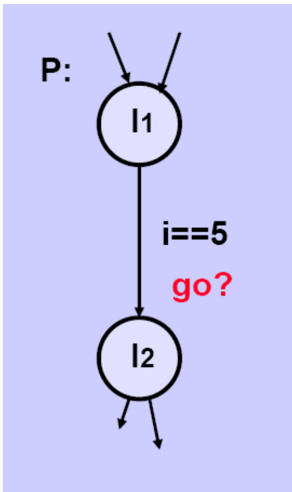
- Suppose the two edges in automata P and Q should be taken as soon as possible
- That is, as soon as both automata are in locations l1 and s1
- How to model with invariants if either one may reach l1 or s1 first?
- **Solution**: Declare the channel as **urgent**
- Informal Semantics:
  - There will be no delay if an edge with a synchronization over an urgent channel can be taken
- Restrictions:
  - No clock guard allowed on edges with urgent actions
  - Invariants and data-variable guards are allowed



Urgent Transitions Hack (old)

- Assume **i** is a data variable
- We want P to take the transition from l1 to l2 as soon as i==5
- **Solution**: We add extra automaton Q, and an **urgent channel go** which forces P to take the edge:



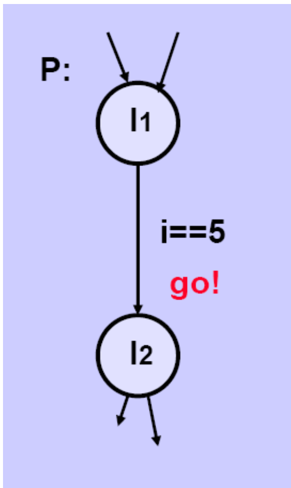


Broadcast Channels

- A channel a can be declared as a broadcast channel to allow the sender to synchronize with more than one receiver
- If a is a broadcast channel, then a set of edges in different processes can synchronize if one is emitting (a!) and the others are receiving (a?)
- A process can always emit (a!) on a broadcast channel without any receivers (a?)

Urgent Transitions Hack - new

- Assume **i** is a data variable
- We want P to take the transition from l1 to l2 as soon as i==5
- **Solution**: We add an **urgent broadcast channel go** which forces P to take the edge:



Modeling Patterns – Overview

- Variable reduction
- Synchronous value passing
- Atomicity
- Urgent edges
- Timers
- Bounded liveness checking

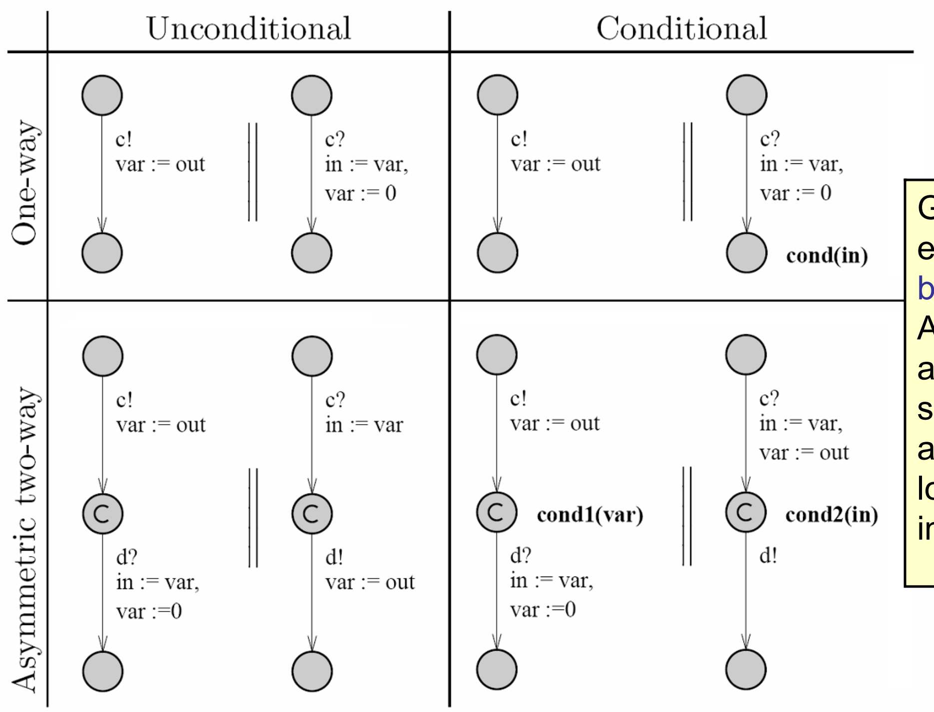
Variable Reduction

- Reduce the size of the state space by explicitly resetting variables when they are not used, thus speeding up the verification
- Idea: Reset a variable v to the initial value on all incoming edges of a location l, if v is inactive in l (i.e., v will be reset on any path from l before used again)
- UPPAAL automatically performs this optimization for all clock variables (this option is called active clock reduction)

Synchronous Value Passing

- Synchronously pass data between processes
- Idea: synchronize over shared binary channel and exchange data via shared variables
- UPPAAL evaluates the assignment of the *sending* synchronization first, so the sender can assign a value to the shared variable which the receiver can then access directly
- Four types: one-way or two-way, unconditional or conditional (i.e., receiver may reject)

Synchronous Value Passing



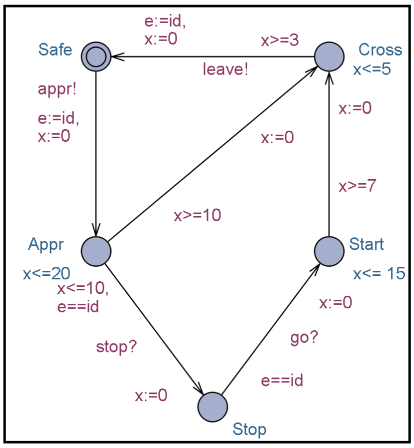
Guard are evaluated before Assignments are executed, so conditions are added as location invariants.

Atomicity

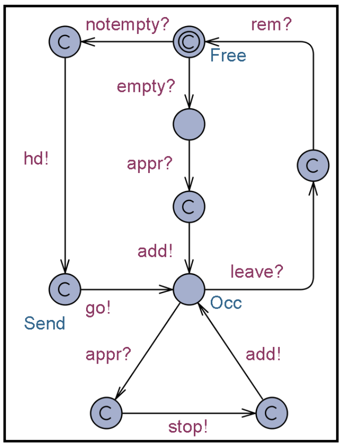
- Reduce the size of the state space by reducing unnecessary interleavings, thus speeding up the verification
- Idea: Use committed locations
- UPPAAL uses an asynchronous execution model, i.e., edges from different automata can interleave, and UPPAAL will explore all possible interleavings
- See for example the IntQueue from the train gate model

Example: Train Gate Model

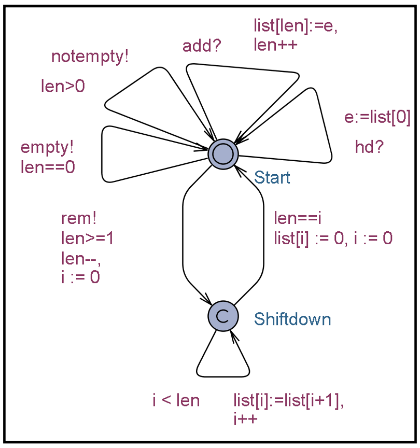
Train(int[0,N] e; const id)



Gate



IntQueue(int[0,N] e)



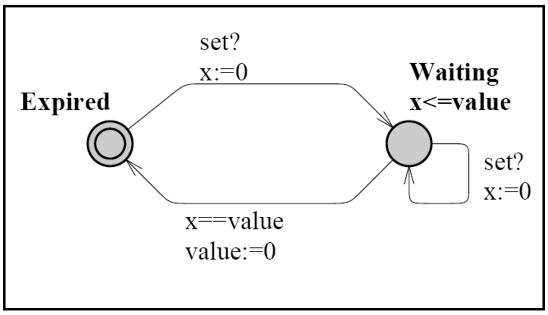
// Global declarations:
const N 5; // # trains + 1
chan appr, stop, go, leave;
chan empty, notempty, hd, add, rem;

// Local declarations
// IntQueue declarations
int[0,N] list[N], len, i;

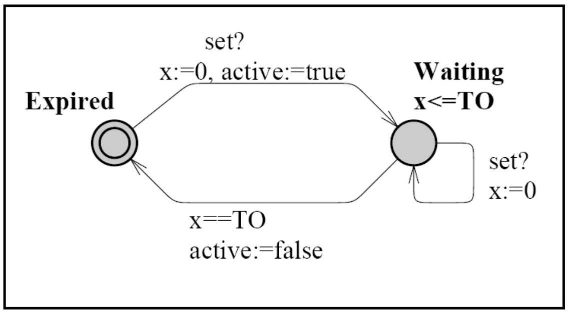
// Train declarations:
clock x;

Timers

- Emulate a timer where, in principle, time decreases until it reaches zero, at which point the timer is said to time-out
- Idea: Add extra automaton with the channel set and the integer value



Timer with variable time-out



Timer with constant time-out

Bounded Liveness Checking

- Check bounded liveness properties:
  - A property is guaranteed to hold within some specified upper time-limit
  - p $-->_{<=t}$ q
  - In TCTL corresponds to **AG** (p ⇒ $\mathbf{AF}_{\leq t}$ q)
- Idea 1 (reduce to unbounded leads-to):
  - Add a clock z
  - Whenever p starts to hold, reset z
  - Check that p --> (q and z ≤ t)
- Idea 2 (reduce to safety property):
  - Add a clock z
  - Add a Boolean variable b
  - Whenever p starts to hold, set b to true and reset z
  - Whenever q starts to hold, set b to false
  - Check that A[] (b imply z ≤ t)