UPPAAL Tutorial

Modeling Patterns

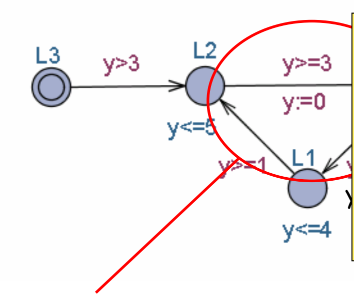
Alexandre David
Paul Pettersson
RTSS'05

In This Session

- Some patterns to use well UPPAAL
  - Clearer models
  - More efficient models
  - Avoid pitfalls
  - Common tricks

Accelerating Cycles *Window*

- Problem: fragmentation of symbolic states.
- Solution: accelerate cycles.



Definition

The **WINDOW** of a cycle
$C = (e_1, e_2, \ldots, e_k)$ is [a,b] iff

1. Every execution of C has accumulated delay between a and b.
2. For any delay d between a and b there exists an execution of C with accumulated delay d.

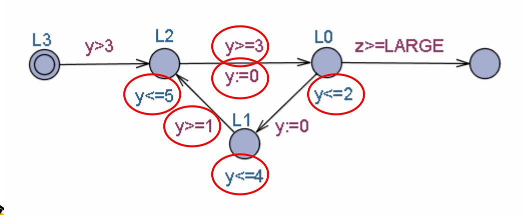
The cycle L0,L1,L2 has the window [3,7] for y

Acceleratable Cycles

Definition

Let $C = (e_1, e_2, \ldots, e_k)$ be a cycle and let y be a clock, then (C,y) is an **acceleratable cycle** if:

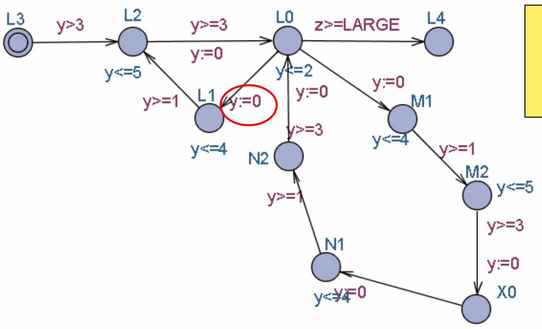
1. Every invariant of C is of the form y<=n (or true)
2. Every guard of C is of the form y>=m (or true)
3. y is reset on all ingoing edges to src(e_1)



Theorem

Every acceleratable cycle has a window.

Accelerated Cycles



Acceleration of cycle
=
"Unfolding of cycle"

Efficiency

If y is reset on the first edge in the accelerated cycle C, then one execution of the appended cycle suffices!!

Theorem

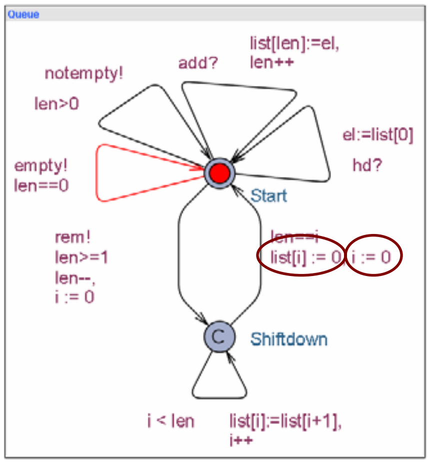
$$3a \leq 2b \Rightarrow (M \models \phi \Leftrightarrow Acc(M,A) \models \phi)$$

(w.r.t. a cycle A)

3*3 ≤ 2*7

Variable Reduction

- **Intent:** reduce state-space by resetting unused variables to a known value (0).
  - Even if a variable is meaningless in some states, its value is still part of the state.
  - Mostly applicable to local variables.



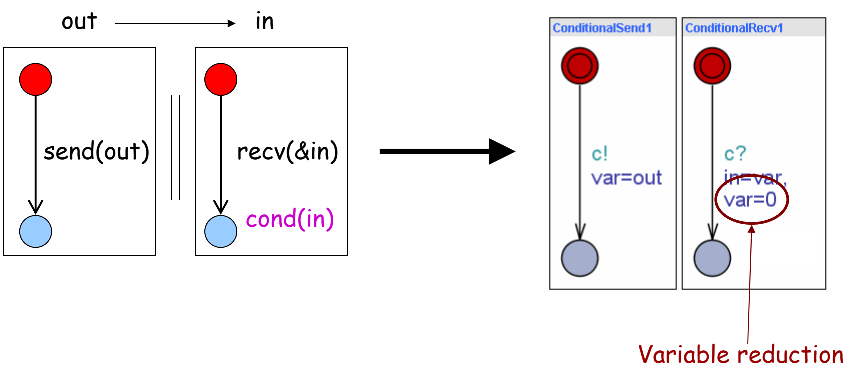
Atomicity

- **Intent:** to reduce the state-space by avoiding interleavings.
  - May be needed from pure modeling point of view too.
  - Useful in synchronization patterns.
  - Use **committed** locations.
  - For sequences of actions, better use C-like code.

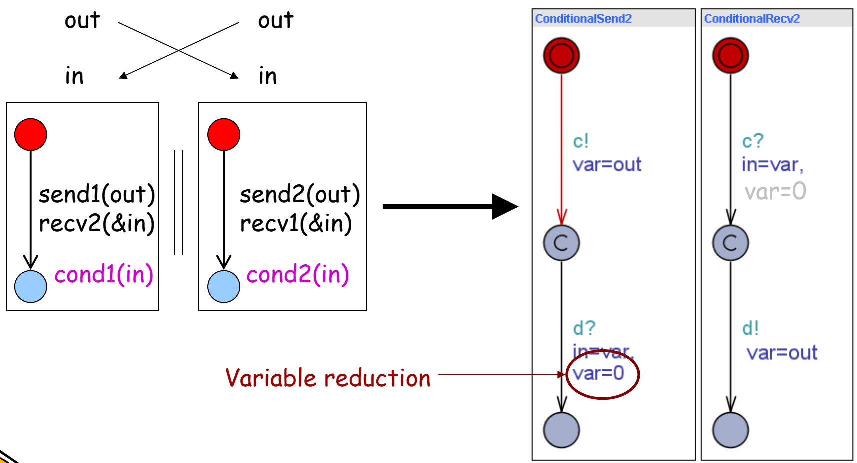
Synchronous Value Passing

- **Intent:** send data synchronously between processes.
  - Typically between local variables.
  - Feature used: UPPAAL evaluate expressions at the sender first.
  - Different variants depending on
    - conditional/unconditional value passing,
    - one/two way value passing.
- Asymmetric

Synchronous Value Passing – One-way

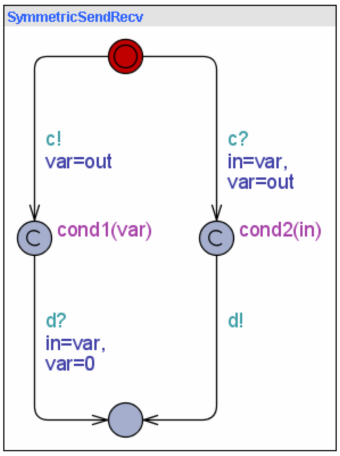


Synchronous Value Passing – Two-way



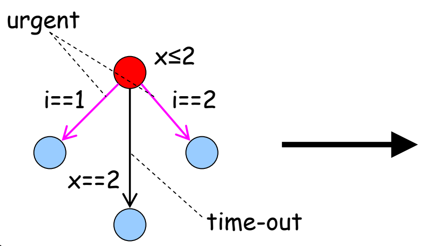
Synchronous Value Passing – Two-way Symmetric Encoding

2x

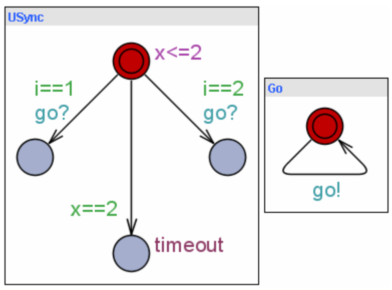


Urgent Edges

- **Intent:** take an edge as soon as it is enabled (without delay).
  - Condition on the edge, not the location.
  - Solution limit: no clock constraint (yet).



urgent chan go;



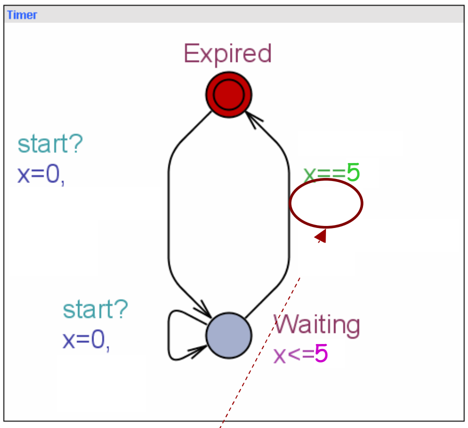
Timers

- **Intent:** code a classical timer that emits a time-out event.
  - In principle time (in timers) decreases but in UPPAAL it only increases.
  - More natural for some models.
  - Operations:
    - start(value)
    - expired?
    - time-out event

Timers

Basic timer:

- (re-)start
  `start!`
- expired?
  `active` (bool)
  `active go?`
  (bool+urgent chan)
- time-out event
  `timeout?`

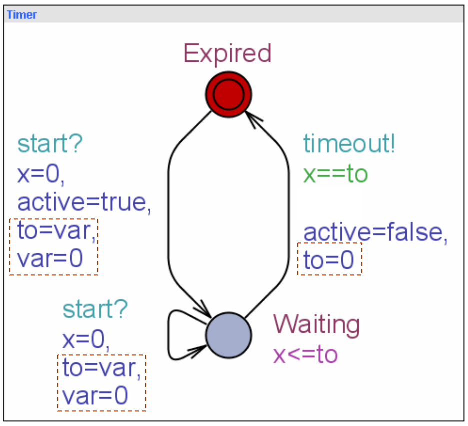


Not needed for clocks:
Active clock reduction takes care of this.

Timers

Parametric timer:

- (re-)start(value)
  `start! var=value`
- expired?
  `active` (bool)
  `active go?`
  (bool+urgent chan)
- time-out event
  `timeout?`



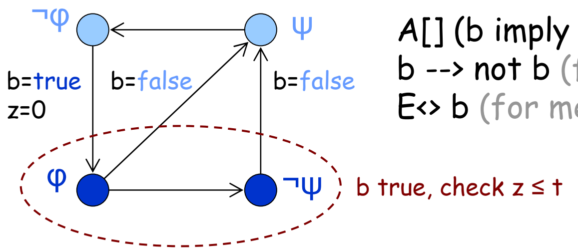
Declare 'to' with a tight range.

Bounded Liveness

- **Intent:** Check for properties that are guaranteed to hold eventually within some upper (time) bound.
  - Provide additional information (with a valid bound).
  - More efficient verification.
  - φ leadsto$_{\leq t}$ ψ reduced to $A\Box(b \Rightarrow z \leq t)$ with bool b set to *true* and clock z reset when φ starts to hold. When ψ starts to hold, set b to *false*.

Bounded Liveness

- The truth value of b indicates whether or not ψ should hold in the future.



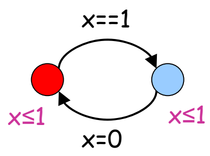
A[] (b imply z≤t)
b --> not b (for non zenoness)
E<> b (for meaningful check)

Zenoness

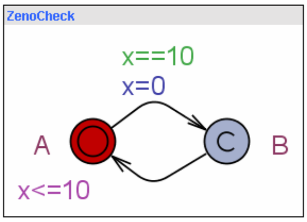
- Problem: UPPAAL does not check for zenoness directly.
  - A model has "zeno" behavior if it can take an infinite amount of actions in finite time.
  - That is usually not a desirable behavior in practice.
  - Zeno models may wrongly conclude that some properties hold though they logically should not.
  - Rarely taken into account.
- Solution: Add an observer automata and check for non-zenoness, i.e., that time will always pass.

Zenoness

OK



Detect by
•adding the observer:



Constant (10) can be anything (>0), but choose it well w.r.t. your model for efficiency. Clocks 'x' are local.

•and check the property
`ZenoCheck.A --> ZenoCheck.B`

Some Pitfalls

- Unbounded integers
  - Model uses the full range.
- Unsynchronized processes
  - Combinatorial explosion.
- Unused active variables specially in arrays

Tricks

- How to copy a template?
  - Rename, save, rename to original, import.
- State predicates (bool) evaluate to 0 or 1 and can be used as integers:
  - Mutual exclusion:
    `A[]P1.cs+P2.cs+P3.cs<=1`