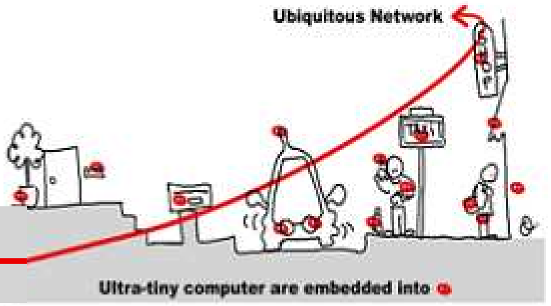
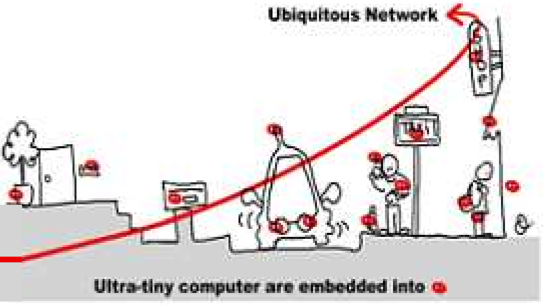


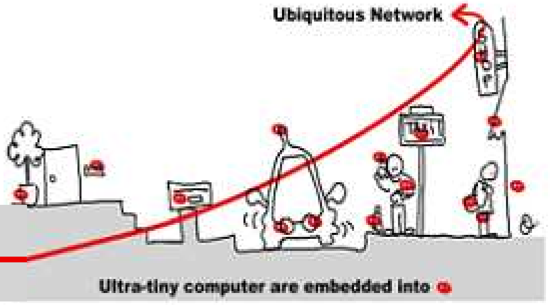
Verification
Introduction to WComp Validation

WComp Verification



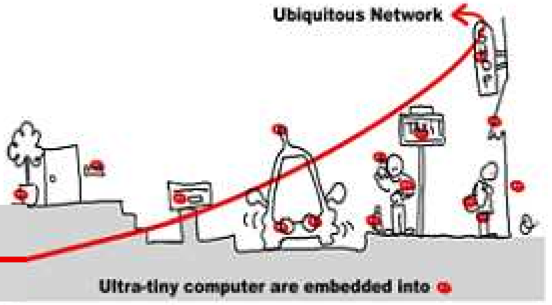
- WComp may be used to design critical applications
- Ensure a safe usage of WComp wrt component behavior
- Apply techniques used to develop critical software

Outline



1. Critical system validation
2. Model-checking Techniques
    1. Model specification as synchronous models
        - Introduction to synchronous modeling
        - Introduction to Lustre synchronous language
    2. Express and prove properties
3. Application to WComp

Critical Software

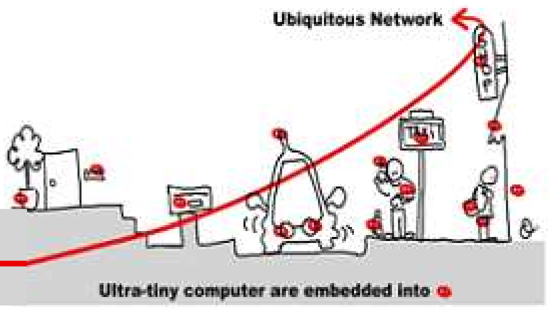


A critical software is a software whose failing has serious consequences:

- Nuclear technology
- Transportation
  - Automotive
  - Train
  - Aircraft construction

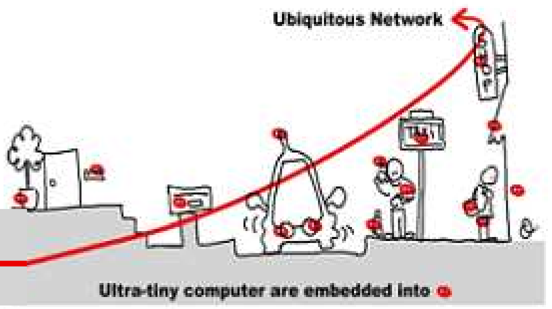
…

Exemple: The Patriot Missile Failure



- On February 25, 1991, during the Golf War, an american patriot missile battery in Dharam, Saudi Arabia, failed to track and intercept an incoming Iracq scud missile. The scud struck american army baracks, killing 28 soldiers and injuring around 100 others people.

Exemple: The Patriot Missile Failure



- A report on the general accounting office, entitled Patriot Missile Defense: software problem led to system failure at Dharam reported on the cause of the failure. It turns out that the cause was an inaccurate calculation of the time since boot due to computer arithmetic errors.

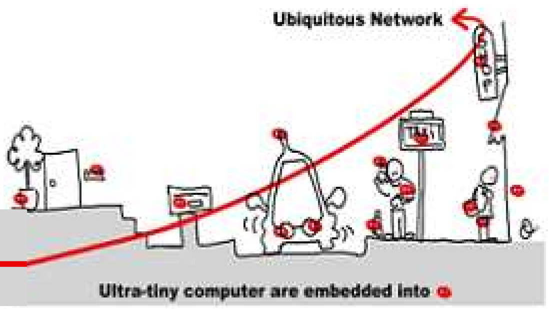
Software Classification

Depending of the level of risk of the system, different kinds of verification are required

Example of the aeronautics norm DO178B:

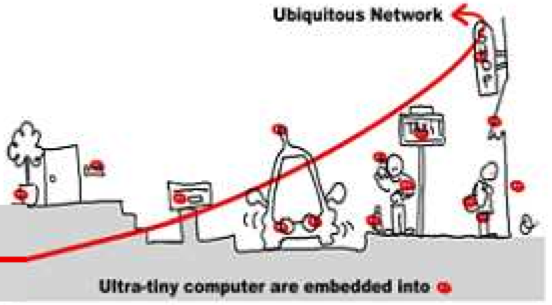
A	Catastrophic (human life loss)
B	Dangerous (serious injuries, loss of goods)
C	Major (failure or loss of the system)
D	Minor (without consequence on the system)
E	Without effect

Software Classification



Minor	🟩	🟩	🟩 acceptable situation	🟩
Major	🟥	🟩	🟩	🟩
Dangerous	🟥 Unacceptable situation	🟥	🟥	🟩
catastrophic	🟥 10^{-3} / hour	🟥 10^{-6} / hour	🟥 10^{-9}/hour	🟩 10^{-12}/hour
probabilities	probable	rare	very rare	very improbable

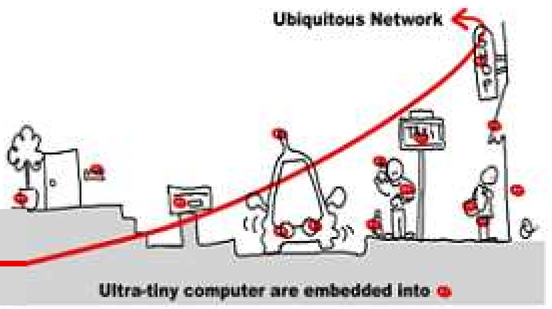
Critical Software



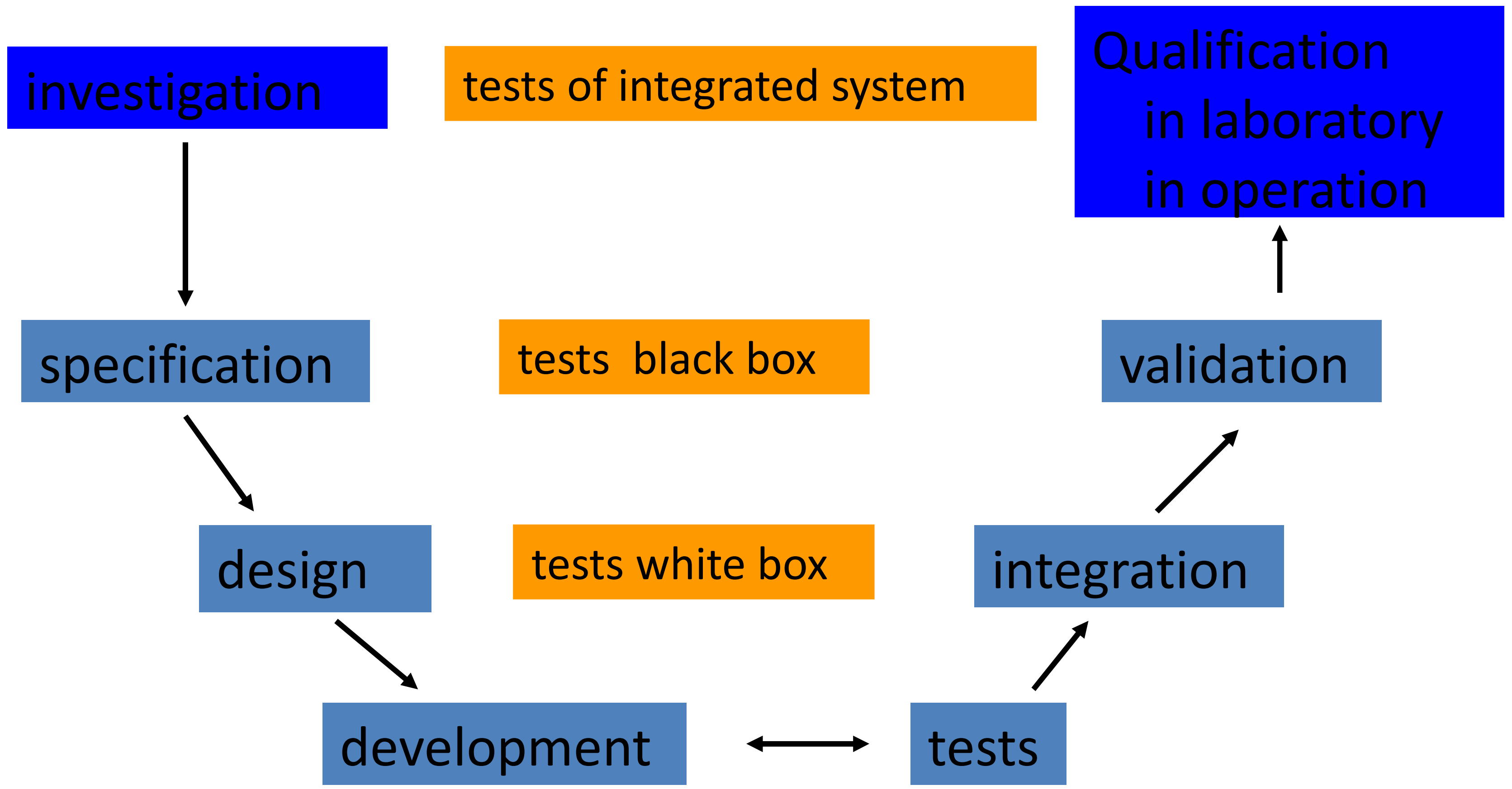
In addition , other consequences are relevant to determine the critical aspect of a software:

- Financial aspect
  - Loosing of equipment, bug correction
  - Equipment callback (automotive)
- Bad advertising
  - Intel famous bug

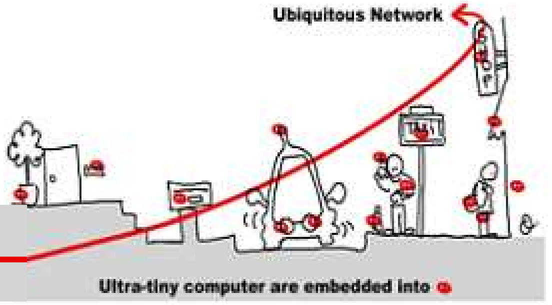
How Develop critical software ?



Classical Development U Cycle

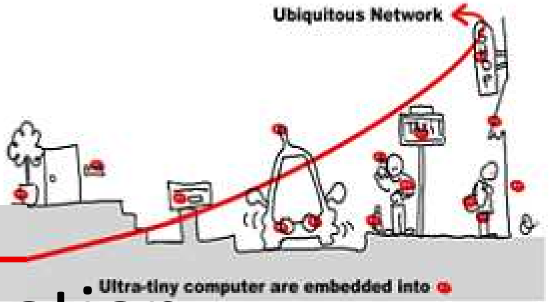


How Develop Critical Software ?



- Cost of critical software development:
  - Specification : 10%
  - Design: 10%
  - Development: 25%
  - Integration tests: 5%
  - Validation: 50%
- Fact:
  – Earlier an error is detected, less expensive its correction is.

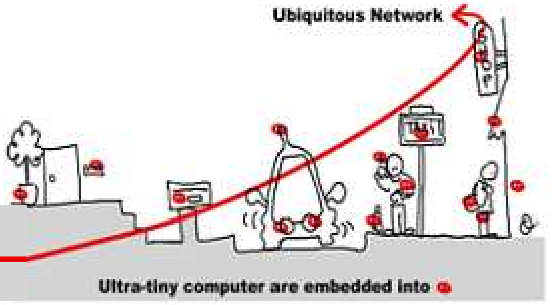
How Develop Critical Software ?

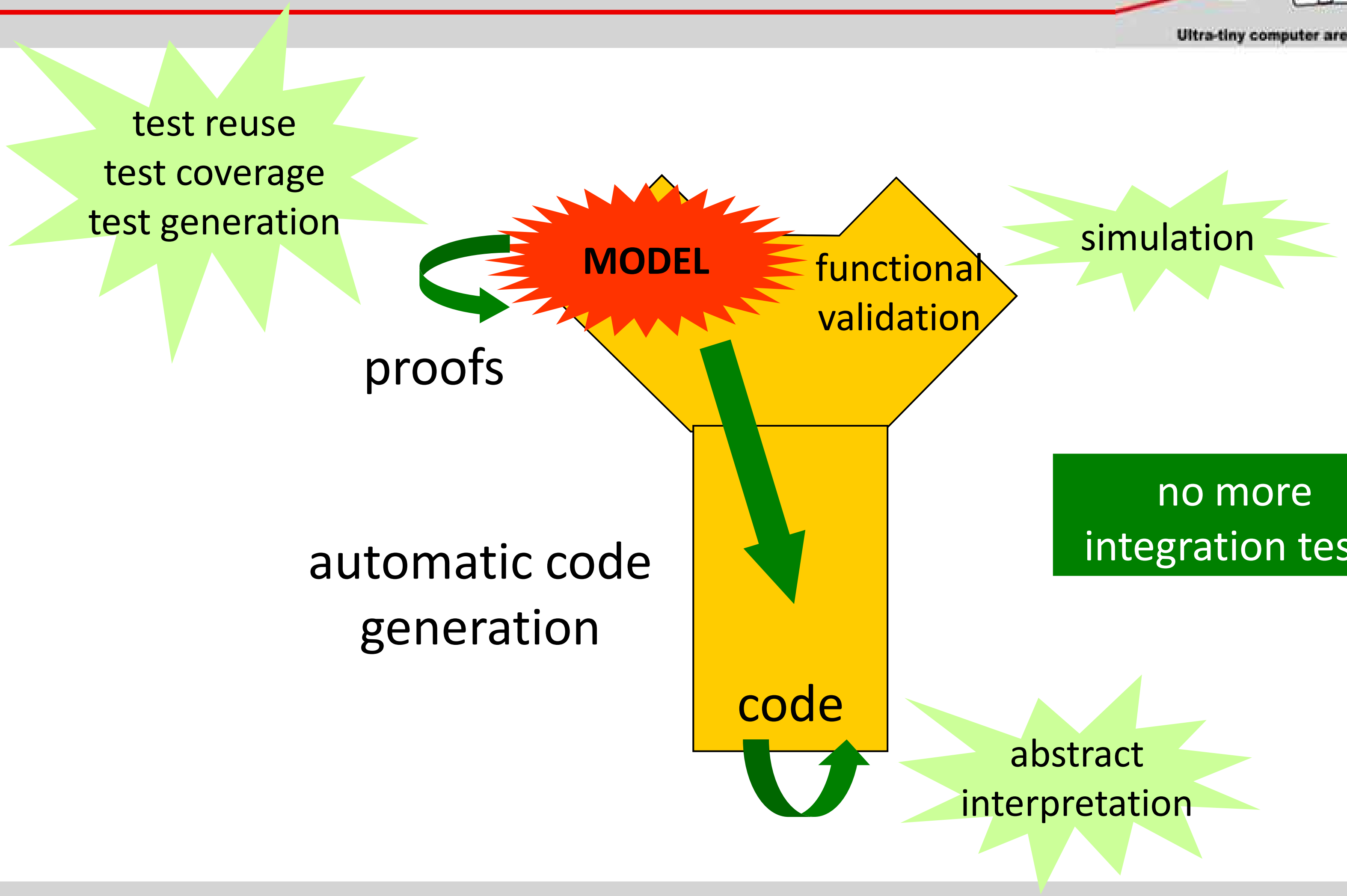


- Goals of critical software specification:
  - Define application needs
    - ⇒ specific domain engineers
  - Allowing application development
    - Coherency
    - Completeness
  - Allowing application functional validation
    - Express properties to be validated

⇒ Formal models usage

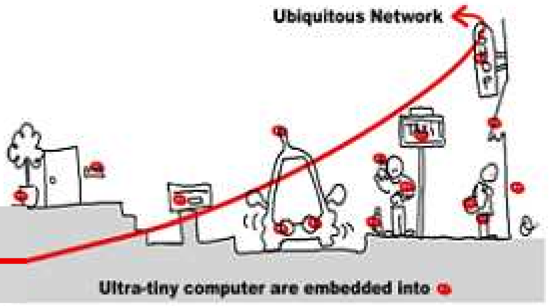
How Develop Critical Software

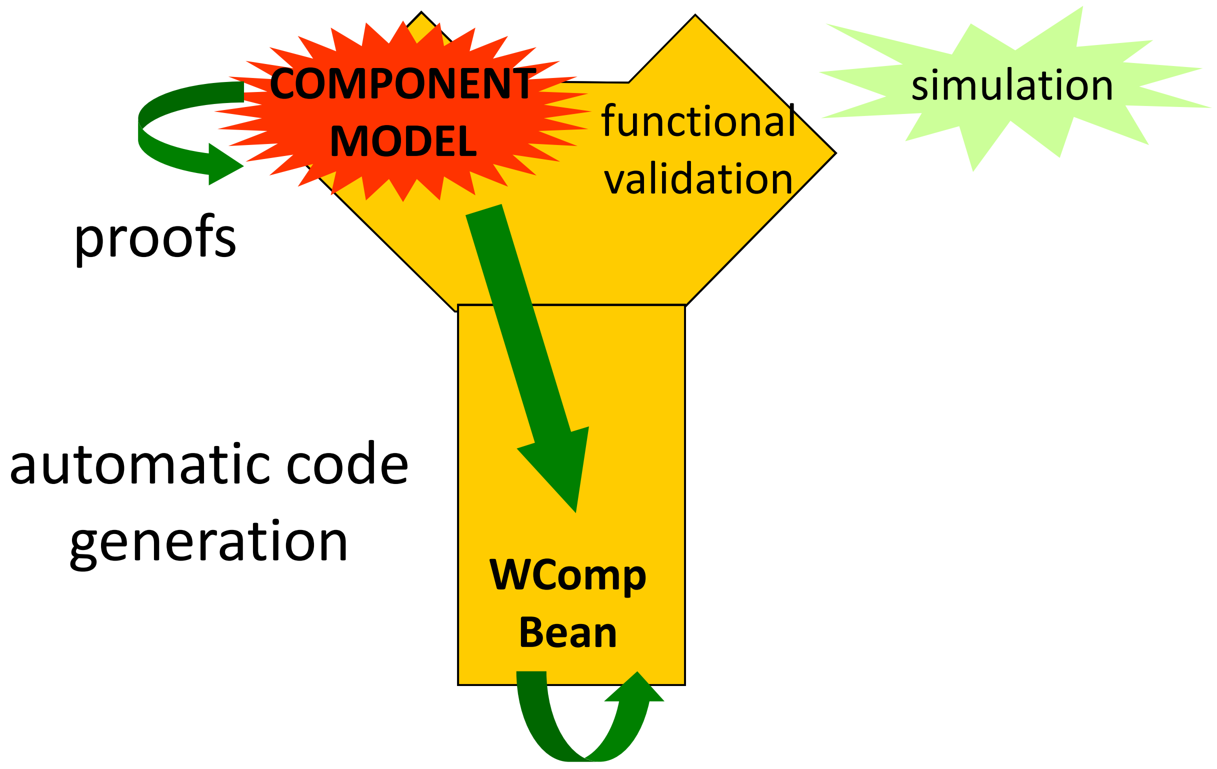


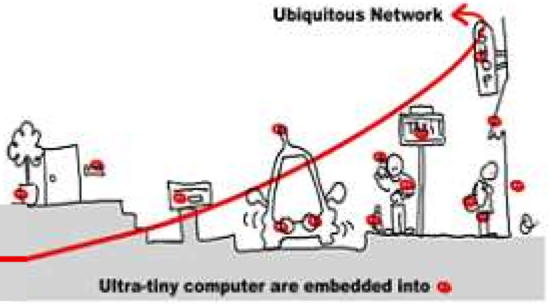


no more integration tests

Application to Wcomp

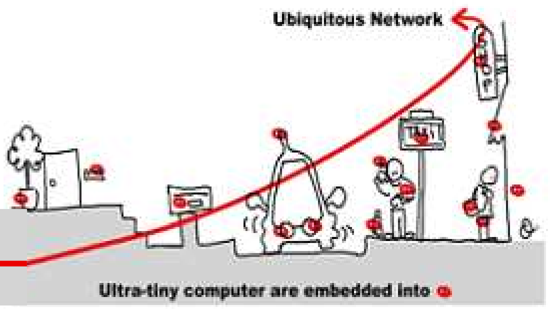






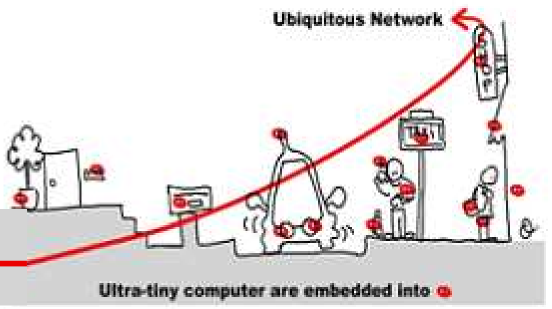
Verification
Critical Software Validation

Critical Software Validation



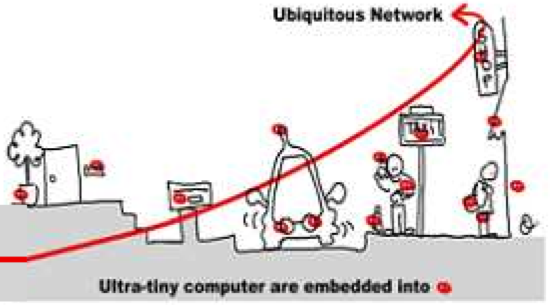
- What is a correct software?
  - No execution errors, time constraints respected, compliance of results.
- Solutions:
  - At model level :
    - Simulation
    - Formal proofs
  - At implementation level:
    - Test
    - Abstract interpretation

Validation Methods



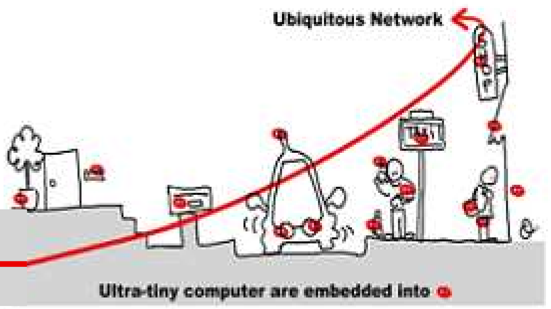
- Testing
  - Run the program on set of inputs and check the results
- Static Analysis
  - Examine the source code to increase confidence that it works as intended
- Formal Verification
  - Argue formally that the application always works as intended

Formal verification



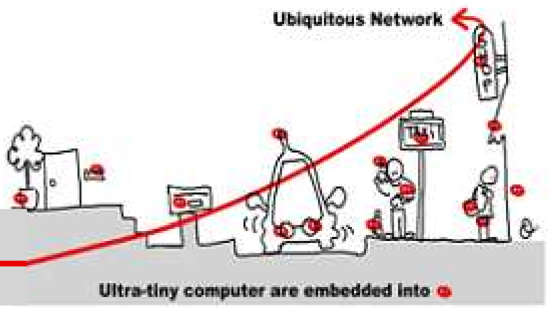
- What about functional validation ?
  - Does the program compute the expected outputs?
  - Respect of time constraints (temporal properties)
  - Intuitive partition of temporal properties:
    - Safety properties: something bad never happens
    - Liveness properties: something good eventually happens

Safety and Liveness Properties



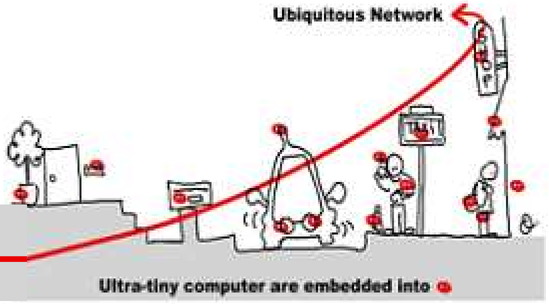
- Example: the beacon counter in a train:
  - Count the difference between beacons and seconds
  - Decide when the train is ontime, late, early

Safety and Liveness Properties



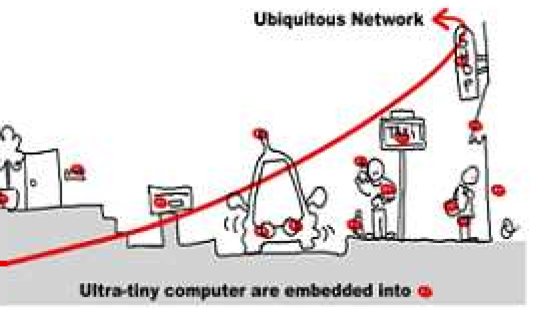
- Some properties:
  1. It is impossible to be late and early;
  2. It is impossible to directly pass from late to early;
  3. It is impossible to remain late only one instant;
  4. If the train stops, it will eventually get late
- Properties 1, 2, 3 : safety
- Property 4 : liveness

Safety and Liveness Properties



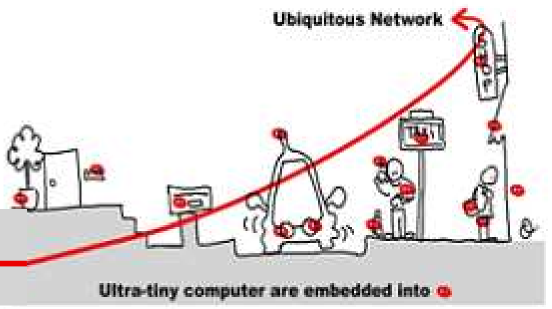
- Some properties:
  1. It is impossible to be late and early;
  2. It is impossible to directly pass from late to early;
  3. It is impossible to remain late only one instant;
  4. If the train stops, it will eventually get late
- Properties 1, 2, 3 : safety
- Property 4 : liveness (refer to unbound future)

Safety and Liveness Properties Checking

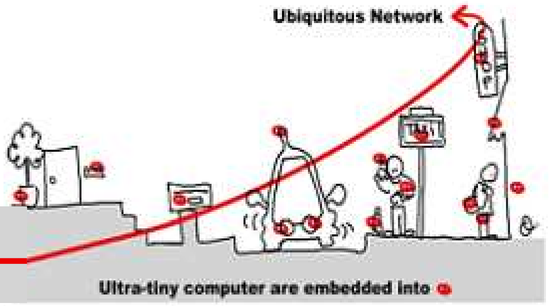


- Use of model checking techniques
- Model checking goal: prove safety and liveness properties of a system in analyzing a model of the system.
- Model checking techniques require:
  - model of the system
  - express properties
  - algorithm to check properties on the model ($\Rightarrow$ decidability)

Model Checking Techniques

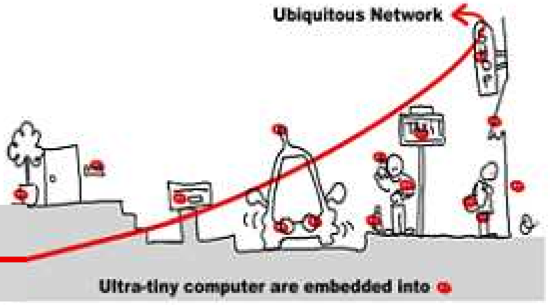


- Model = automata which is the set of program behaviors
- Properties expression = temporal logic:
  - LTL : liveness properties
  - CTL: safety properties
- Algorithm =
  - LTL : algorithm exponential wrt the formula size and linear wrt automata size.
  - CTL: algorithm linear wrt formula size and wrt automata size



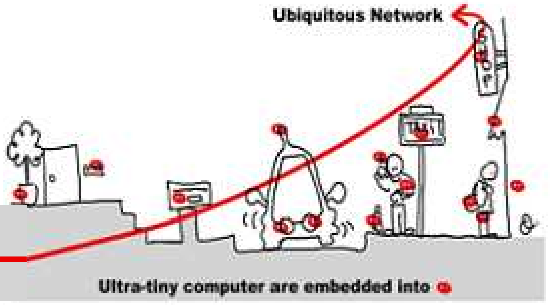
Model Checking
Model Specification

Model Checking Technique



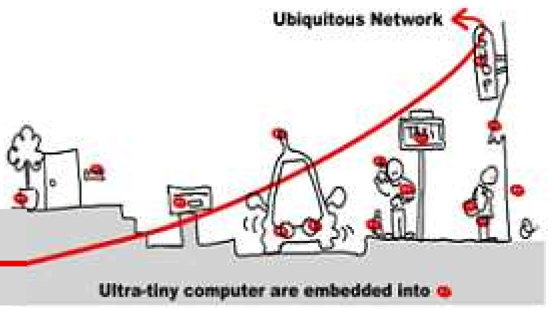
- Model = automata which is the set of program behaviors
- Properties expression = temporal logic:
  - LTL : liveness properties
  - CTL: safety properties
- Algorithm =
  - LTL : algorithm exponential wrt the formula size and linear wrt automata size.
  - CTL: algorithm linear wrt formula size and wrt automata size

Component Models



- WComp Components represent software specification
- To achieve component behavior verification we need to build its model well suited to software validation
- Component behavior specification with a Synchronous language
- Specification = model

Determinism & Reactivity



- Synchronous languages are deterministic and reactive
- Determinism:

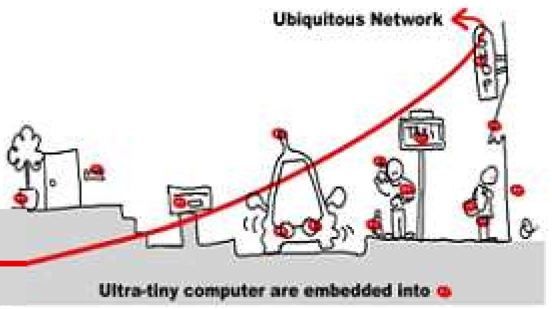
  The same input sequence always yields

  The same output sequence
- Reactivity:

  The program must react[(1)] to any stimulus

  Implies absence of deadlock

(1) Does not necessary generate outputs, the reaction may change internal state only.



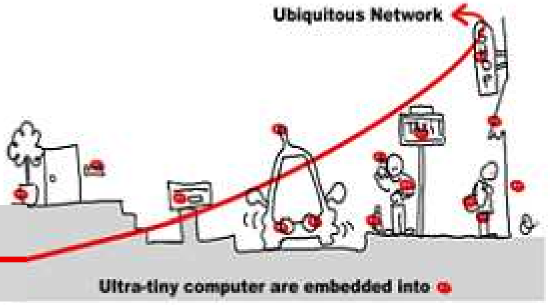
Synchronous Hypothesis

- Actually, a synchronous model works on a logical time.
- The time is
  - Discrete
  - Total ordering of instants.

$\left.\right\}$ Use N as time base

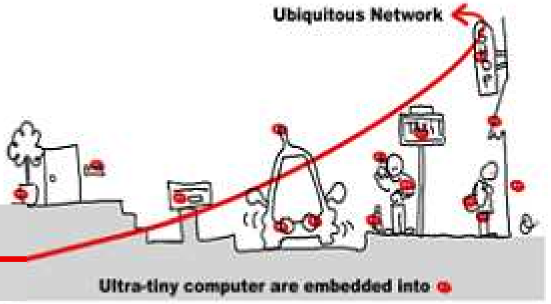
- A reaction executes in one instant.
- Actions that compose the reaction may be partially ordered.

Synchronous Hypothesis



- Communications between actors are also supposed to be instantaneous.
- All parts of a synchronous model receive exactly the same information (instantaneous broadcast).
- Outcome: Outputs are simultaneous with Inputs (they are said to be synchronous)
- Thanks to these strong hypotheses, program execution is fully deterministic.

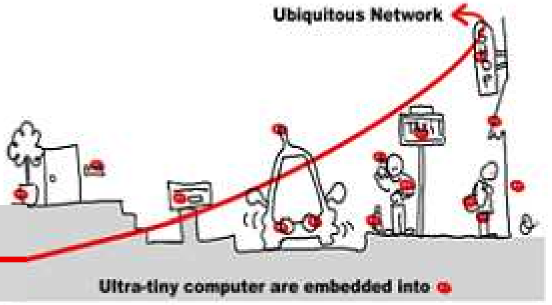
Reactive ?



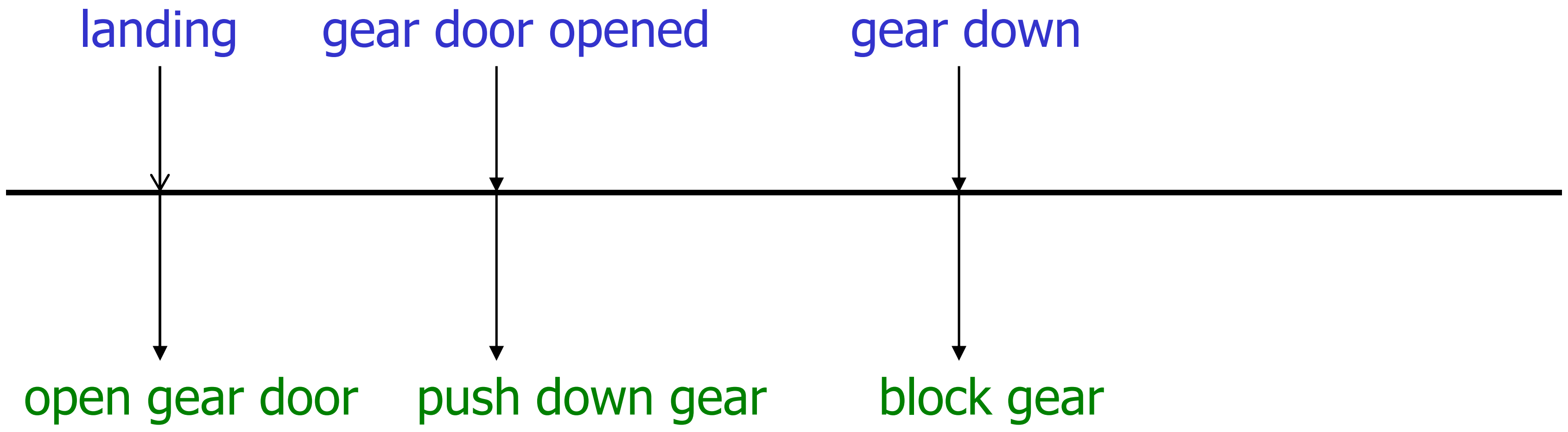
- Different ways to "react" to the environment:
  - Event driven system:
    - Receive events
    - Answer by sending events
  - Data flow system:
    - Receive data continuously
    - Answer by treating data continuously also

Some systems have components of both kinds

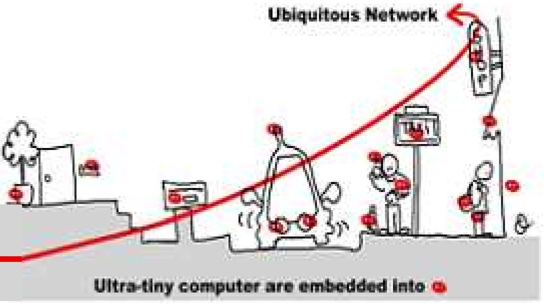
Event Driven Reactive System



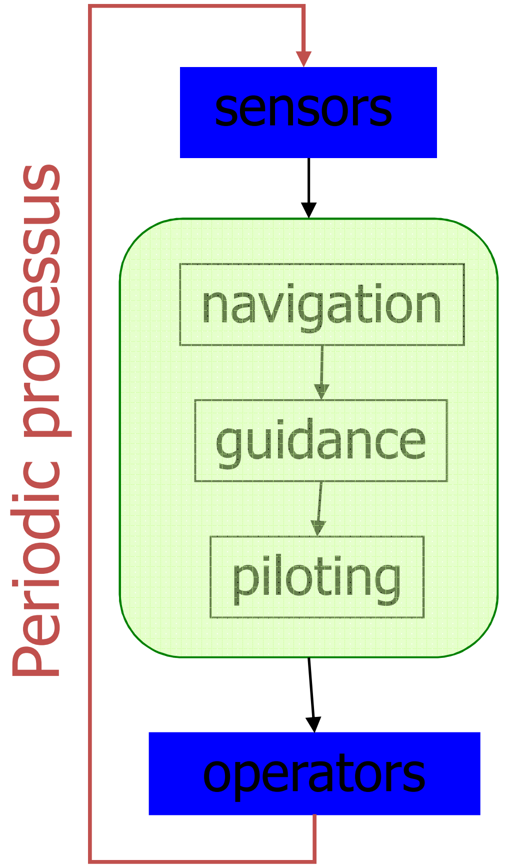
Langing gear management



Data Flow Reactive System (Example)

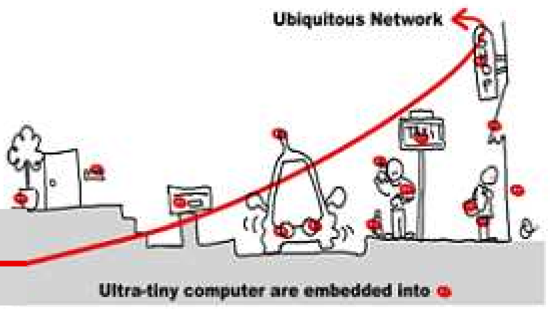


Control/Command vehicle



- get measures
- where am I ?
- where go I ?
- command computation
- command to operators

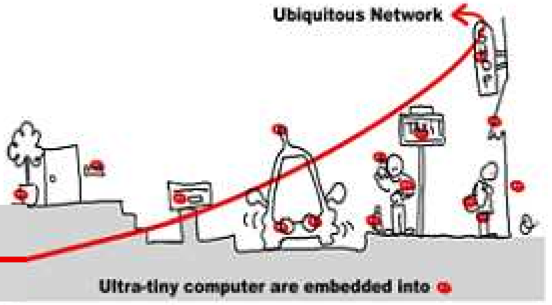
LUSTRE



LUSTRE is a data flow synchronous language:

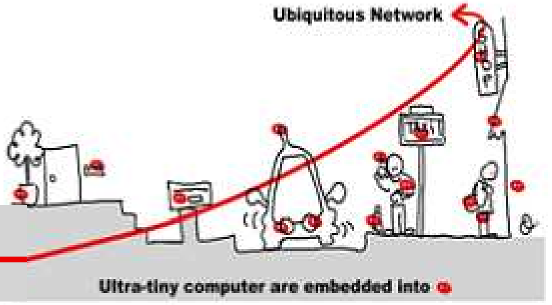
- It is a very simple language (4 primitive operators to express reactions)
- Relies on models familiar to engineers
    - Equation systems
    - Data flow network
- Lends itself to formal verification (it is a kind of logical language)

Operator Networks

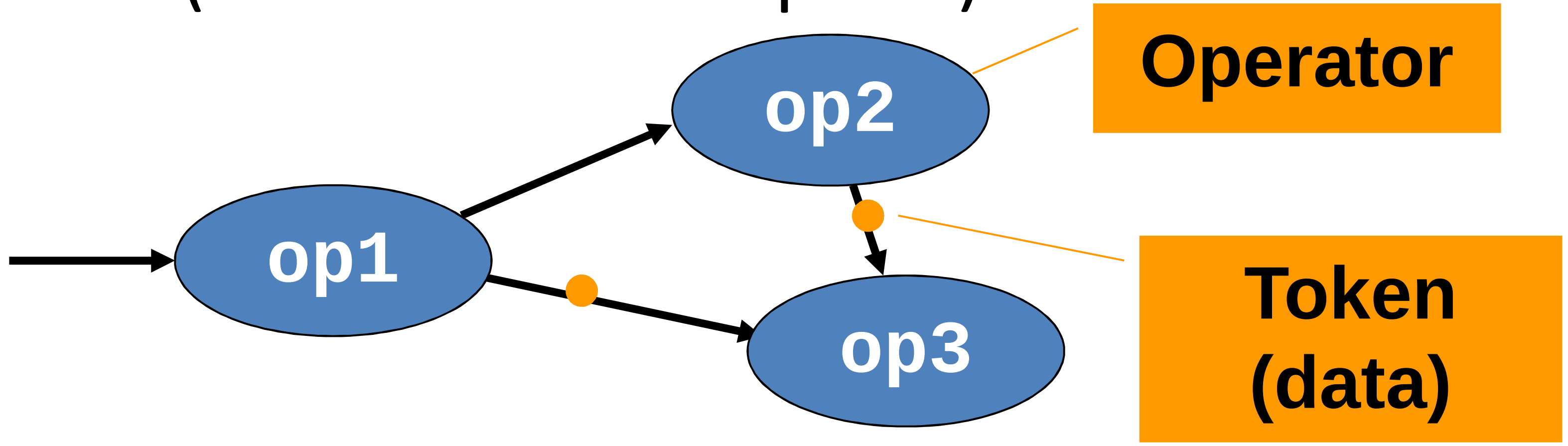


- Very simple (mathematical) semantics
- LUSTRE programs can be interpreted as networks of operators.
- Data « flow » to operators where they are consumed. Then, the operators generate new data. (Data Flow description).

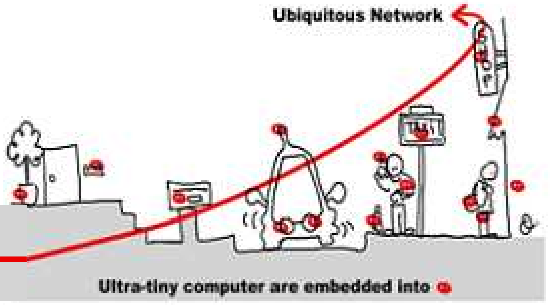
Operator Networks



- LUSTRE programs can be interpreted as networks of operators.
- Data « flow » to operators where they are consumed. Then, the operators generate new data. (Data Flow description).



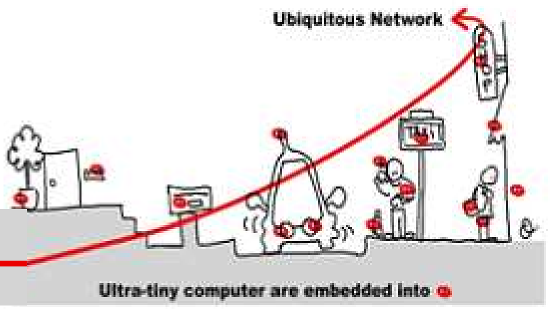
Flows, Clocks



- A flow is a pair made of
  - A possibly infinite sequence of values of a given type
  - A clock representing a sequence of instants

$$\mathbf{X:T} \qquad (x_1, x_2, \ldots, x_n, \ldots)$$

Language (1)



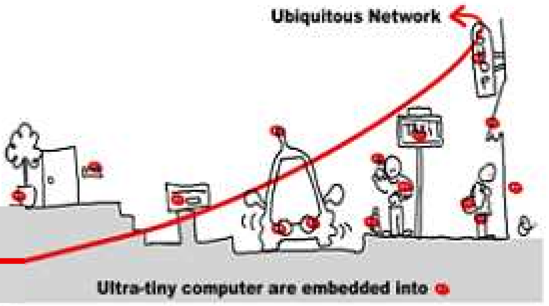
Variable : (= flow) :

- typed
- If not an input variable, defined by 1 and only 1 equation

Equation : **X = E** means $\forall k, x_k = e_k$

Assertion : Boolean expression that should be always **`true`** at each instant of its clock.

Language (2)



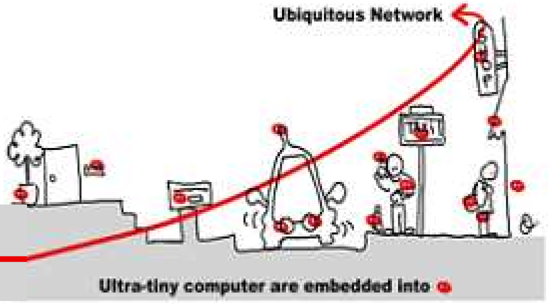
Substitution principle:

if **X = E** then **E** can be substituted for **X** anywhere in the program and conversely

Definition principle:

A variable is fully defined by its declaration and the equation in which it appears as a left-hand side term

Expressions



Constants

`0`, **`1`**, …, **`true`**, **`false`**, …, **`1.52`**, …

real

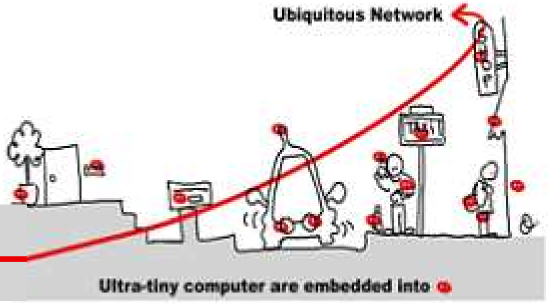
int

bool

+
Imported types and operators

$$c : \alpha \Leftrightarrow \forall k \in \mathbb{N}, c_k = c$$

« Combinational » Lustre



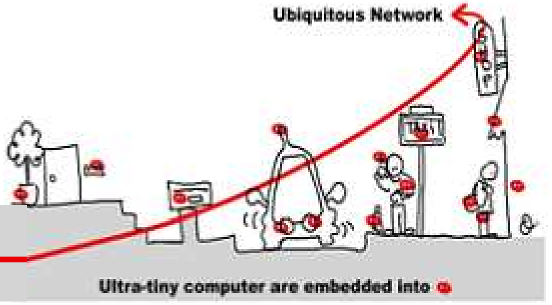
Data operators

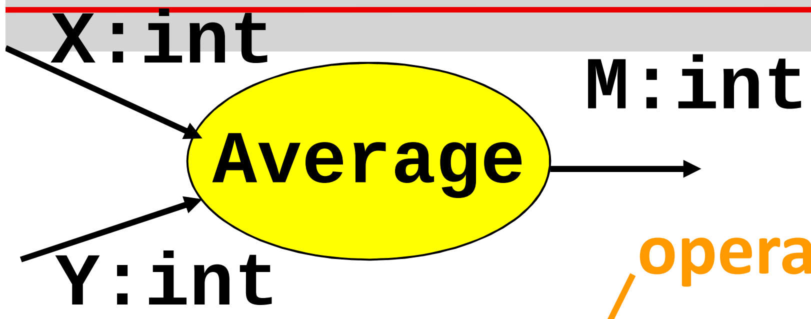
Arithmetical: **+**, **-**, *****, **/**, **div**, **mod**
Logical: **and**, **or**, **not**, **xor**, **=>**
Conditional: **if … then … else** …
Casts: **int**, **real**

« Point-wise » operators

$$X \; op \; Y \Leftrightarrow \forall k, (X \; op \; Y)_k = X_k \; op \; Y_k$$

« Combinational » Example





operator

Input flows

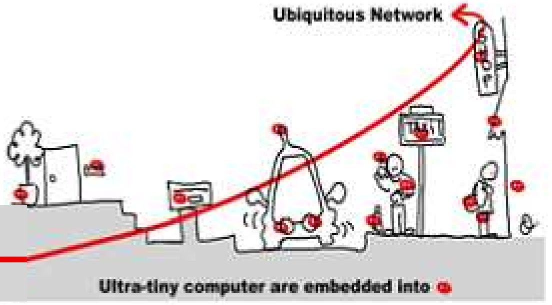
Result

Definition

```
node Average (X,Y:int)
  returns (M:int);
let
  M = (X + Y) / 2;
tel
```

$$\forall k \in \mathbb{N}, M_k = (X_k + Y_k)/2_k$$

« Combinational » Example

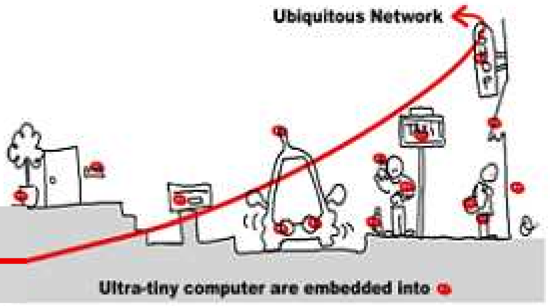


- if operator

```
 node Max (a,b : real) returns (m: real)
let
    m = if (a >= b) then a else b;
tel
```

functional «if then else »; it is not a statement

« Combinational » Example

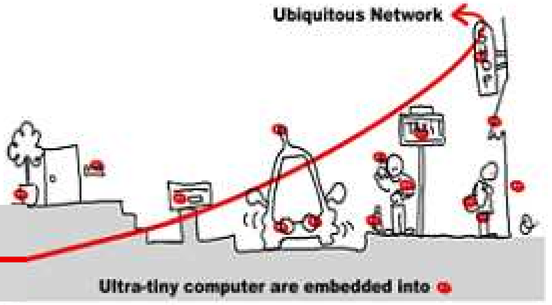


- if operator

```
 node Max (a,b : real) returns (m: real)
let
    m = if (a >= b) then a else b;
tel
```

~~let~~
~~if (a >= b) then m = a ;~~
~~else m = b;~~
~~tel~~

Memorizing



Take the past into account!

pre (previous):

$$X = (x_1, x_2, \cdots, x_n, \cdots) \ : \ pre(X) = \left(\text{nil}, x_1, \cdots, x_{n-1}, \cdots\right)$$

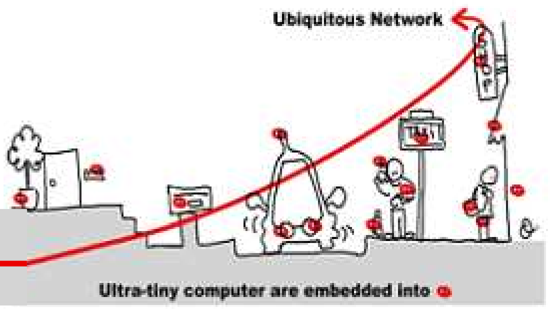
Undefined value denoting uninitialized memory: nil

-> (initialize): sometimes call “followed by”

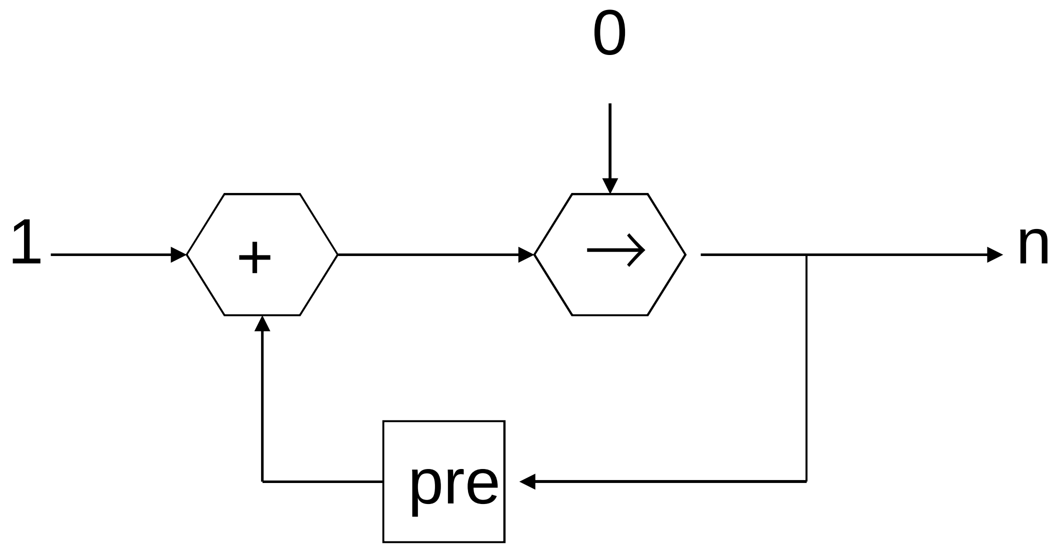
$$X = (x_1, x_2, \cdots, x_n, \ldots) \ , \ Y = (y_1, y_2, \cdots, y_n, \ldots) \ :$$

$$(X -> Y) = (x_1, y_2, \cdots, y_n, \ldots)$$

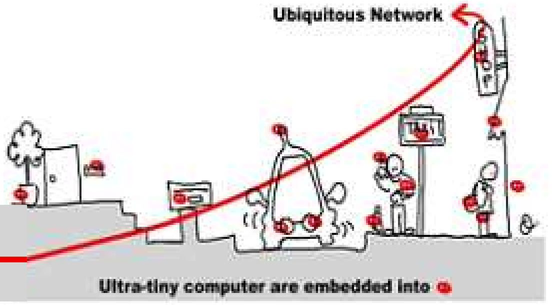
« Sequential » Examples



$$n \;=\; 0 \rightarrow \mathrm{pre}(n) + 1$$

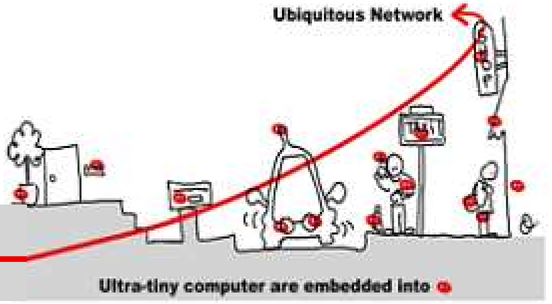


Sequential » Examples



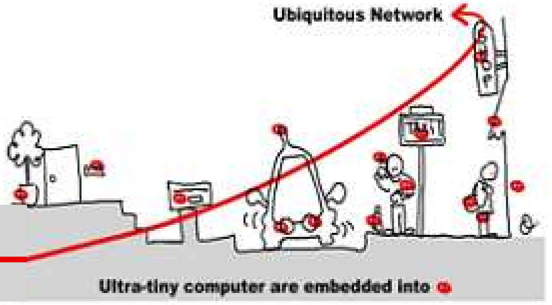
```
node MinMax (X:int) returns (min,max:int);
let
   min = X -> if (X < pre min) then X else pre min;
   max = X -> if (X > pre max) then X else pre max;
tel
```

« Review » Example



```
node CT (init:int) returns (c:int);
 let c = init -> pre c + 2; tel

 node DoubleCall (even:bool) returns (n:int);
 let
        n = if even then CT(0) else
                             CT(1);
 tel

 Doublecall(ff ff tt tt ff ff tt tt ff) = ?
```

Recursive definitions



Temporal recursion

Usual. Use **pre** and **->**

e.g.: nat = 1 **->** **pre** nat + 1

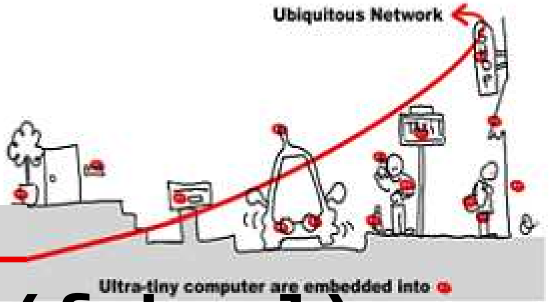
Instantaneous recursion

e.g.: X = 1.0 / (2.0 – X)

Forbidden in Lustre, even if a solution exists!

Be carefull with cross-recursion.

Edges



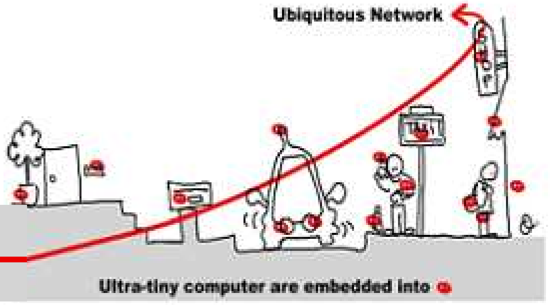
```
node Edge (b:bool) returns (f:bool);
-- detection of a rising edge
let
   f = false -> (b and not pre(b));
tel;
```

initial

Undefined at
the first instant

```
Falling_Edge = Edge(not c);
```

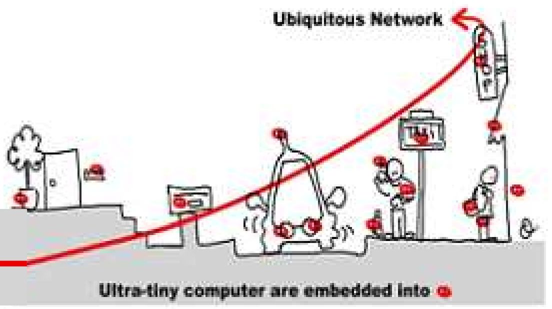
Bistable



- Node Switch (on,off:bool) returns (s:bool); such that:
  - S raises (false to true) if on, and falls (true to false) if off
  - must work even off and on are the same

```
node Switch (on,off:bool) returns (s:bool)
let
    s = if (false→pre s) then not off else on;
tel
```

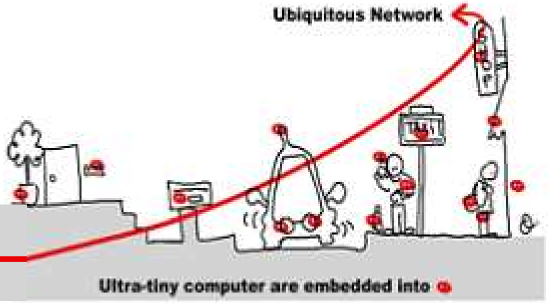
Count



- A node Count (reset, x: bool) returns (c:int) such that:
  - c is reset to 0 if reset, otherwise it is incremented if x

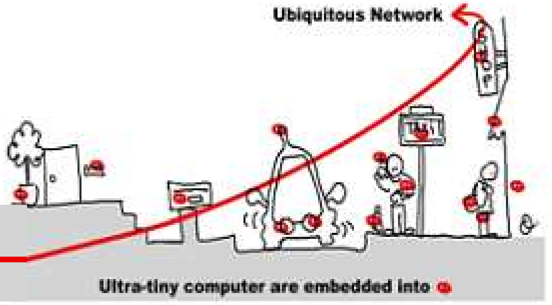
```
node Count (reset, x: bool) returns (c:int)
let
  c = if reset then 0
        else if x then (0 -> pre c) + 1
        else (0 -> pre c)
tel
```

Osc and Osc2



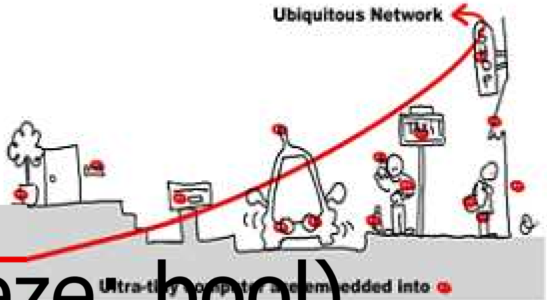
```
node osc (reset: bool) returns (b:int)
let
  b = true -> not pre(b);
tel

node osc2 (reset: bool) returns (b:int)
let
  b = true -> (c and not pre(b)) or
                   (not c and pre(b));
  c  = osc(reset);
tel
```

A Stopwatch



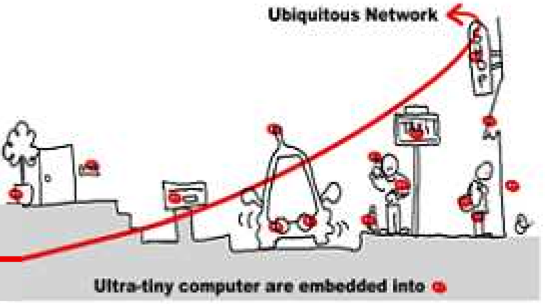
- 1 integer output : time
- 3 input buttons: on_off, reset, freeze
  - on_off starts and stops the watch
  - reset resets the stopwatch (if not running)
  - freeze freezes the displayed time (if running)
- Local variables
  - running, freezed : bool (Switch instances)
  - cpt : int (Count instance)

A stopwatch



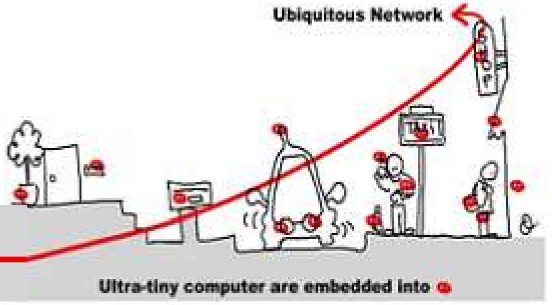
```
node Stopwatch (on_off, reset, freeze: bool)
                               returns (time:int)
 var running, freezed: bool; cpt:int
let
  running = Switch(on_off, on_off);
  freezed = Switch(freeze and running,
                                freeze or on_off);
  cpt = Count (reset and not running, running);
  time = if freezed then (0 -> pre time) else cpt;
tel
```

Modulo Counter



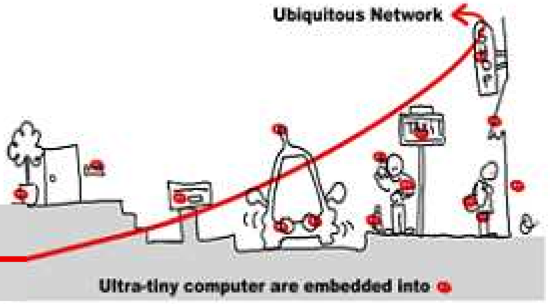
```
node MCounter (incr:bool; modulo : int)
                        returns (cpt:int);
   var count : int;
  let
    count = 0 -> if incr pre (cpt) + 1)
                        else pre (cpt);
    cpt =  count mod modulo;
  tel
```

Modulo Counter with Clock



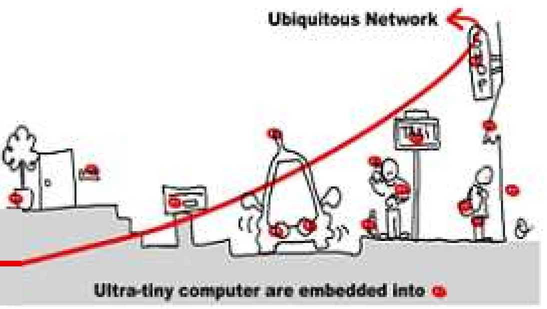
```
node MCounterClock (incr:bool; modulo : int)
                       returns (cpt:int;
                                modulo_clock: bool);
  var count : int;
 let
   count = 0 -> if incr pre (cpt) + 1)
                      else pre (cpt);
   cpt =  count mod modulo;
   modulo_clock = count <> cpt;
 tel
```

Timer



```
node Timer (dummy:bool)
                returns (hour, minute, second:int);
var hour_clock, minute_clock, day_clock : bool;
let
   (second, minute_clock) = MCounterClock(true, 60);
   (minute, hour_clock) =
                          MCounterClock(minute_clock,60);
   (hour, dummy_clock) =
                          MCounterClock(hour_clock, 24);
tel
```

Numerical Examples

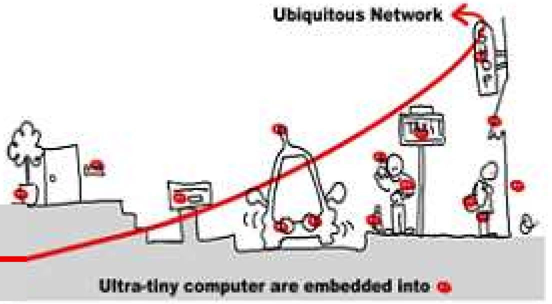


- Integrator node:
  - *f* : real function and Y its integrated value using the trapezoid method:
  - F, STEP : 2 real such that:

$$F_n = f(x_n) \text{ and } x_{n+1} = x_n + STEP_{n+1}$$

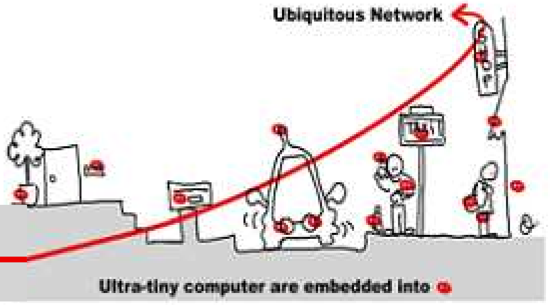
$$Y_{n+1} = Y_n + (F_n + F_{n+1}) * STEP_{n+1}/2$$

Numerical Examples



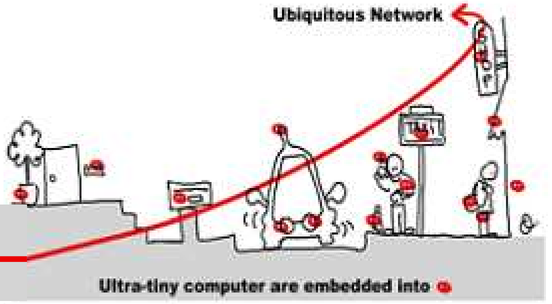
```
node integrator (F, STEP, init : real)
            returns (Y : real);
let
  Y =  init ->pre(Y) + ((F + pre(F))*STEP)/2.0
tel
```

Numerical Examples



```
node sincos (omega : real)
        returns (sin, cos : real);
let
  ~~sin = omega * integrator(cos, 0.1, 0.0);~~
  ~~cos = 1 – omega * integrator(sin, 0.1, 0.0);~~
tel
```

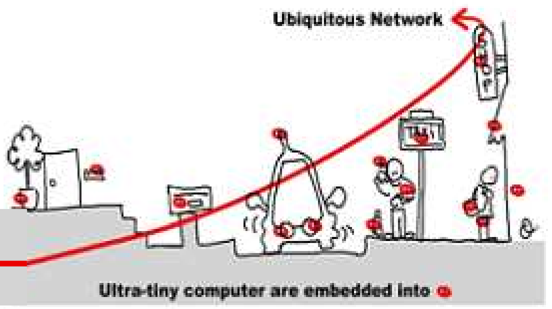
Numerical Examples



```
node sincos (omega : real)
      returns (sin, cos : real);
let
  sin = omega * integrator(cos, 0.1, 0.0);
  cos = 1 – omega * integrator(    , 0.1, 0.0);
tel
```

(0.0 ->pre(sin))

Lustre Program Compilation

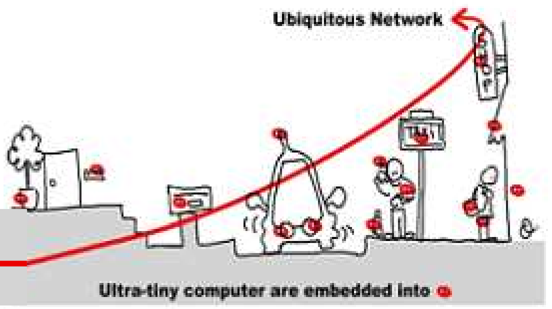


- Static verifications are performed:
  - local and output variables have one equation definition;
  - non recursive node call;
  - absence of uninitialized expression;
  - no cyclic definition (each cyclic definition $\Rightarrow$ pre operator usage);

```
x =  if c then y else z;
y = if c then z else x;
```

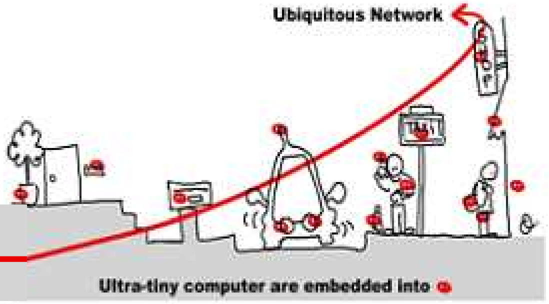
structural deadloack (not real)

Lustre Program Compilation

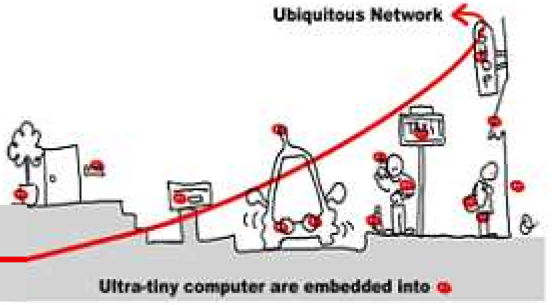


```
node WD (set, reset, deadline:bool)
   returns (alarm:bool);
var is_set:bool;
let
  alarm = is_set and deadline;
  is_set = false -> if set then true
                         else if reset then false else pre(is_set);
  assert not(set and reset);
tel.
```

Lustre Program Compilation



- automaton like code
  - choose state variables among:
    - boolean expressions resulting from pre operator;
    - variables (like _init) associated with some clock whose value is true at first instant

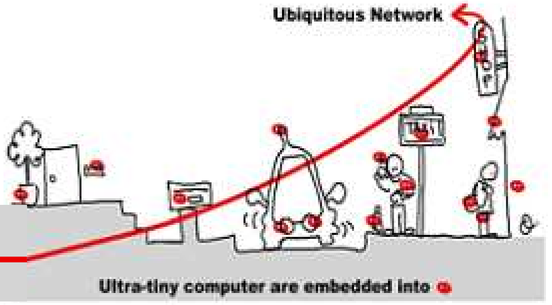


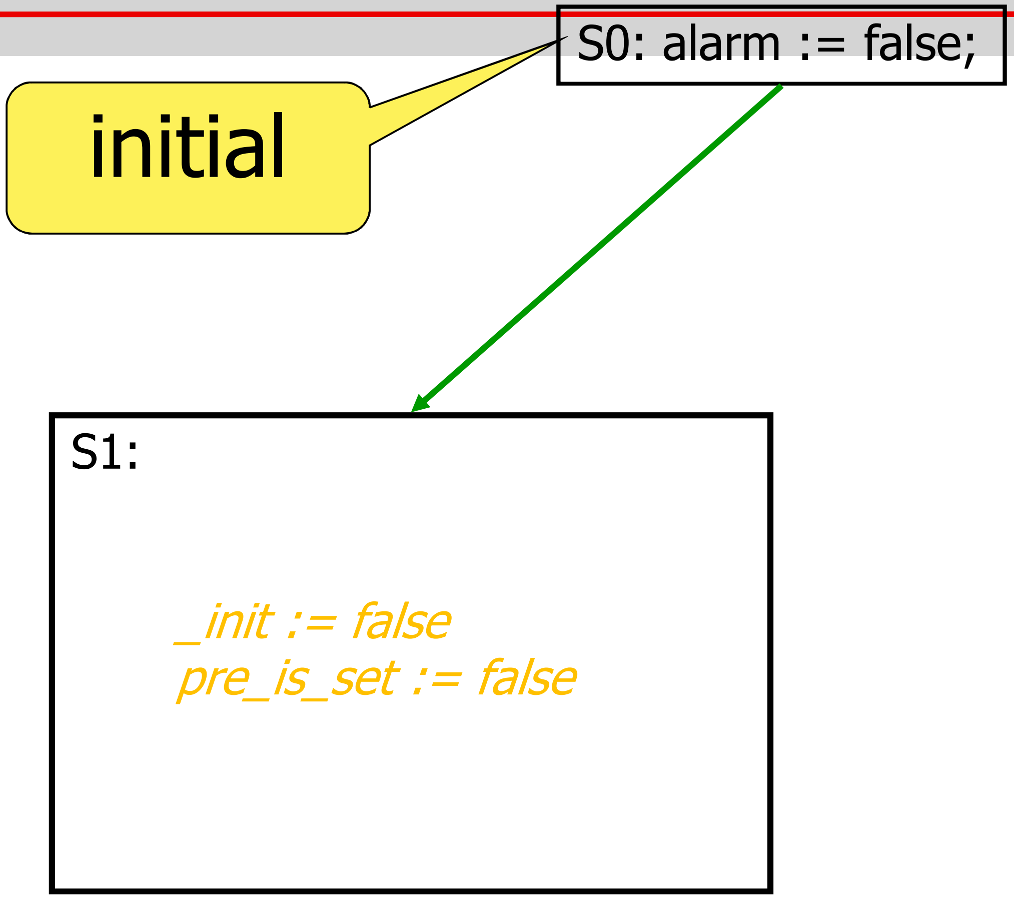
Lustre Program Compilation

For WD, we consider 2 state variables:
_init (true, false, false, ….) and pre_is_set

3 states:
S0: _init = true and pre_is_set = nil
S1: _init = false and pre_is_set = false
S2: _init = false and pre_is_set = true

Lustre Program Compilation

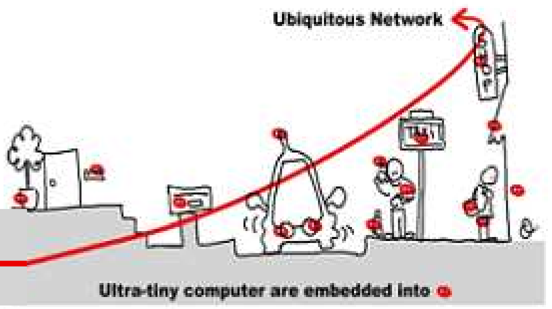




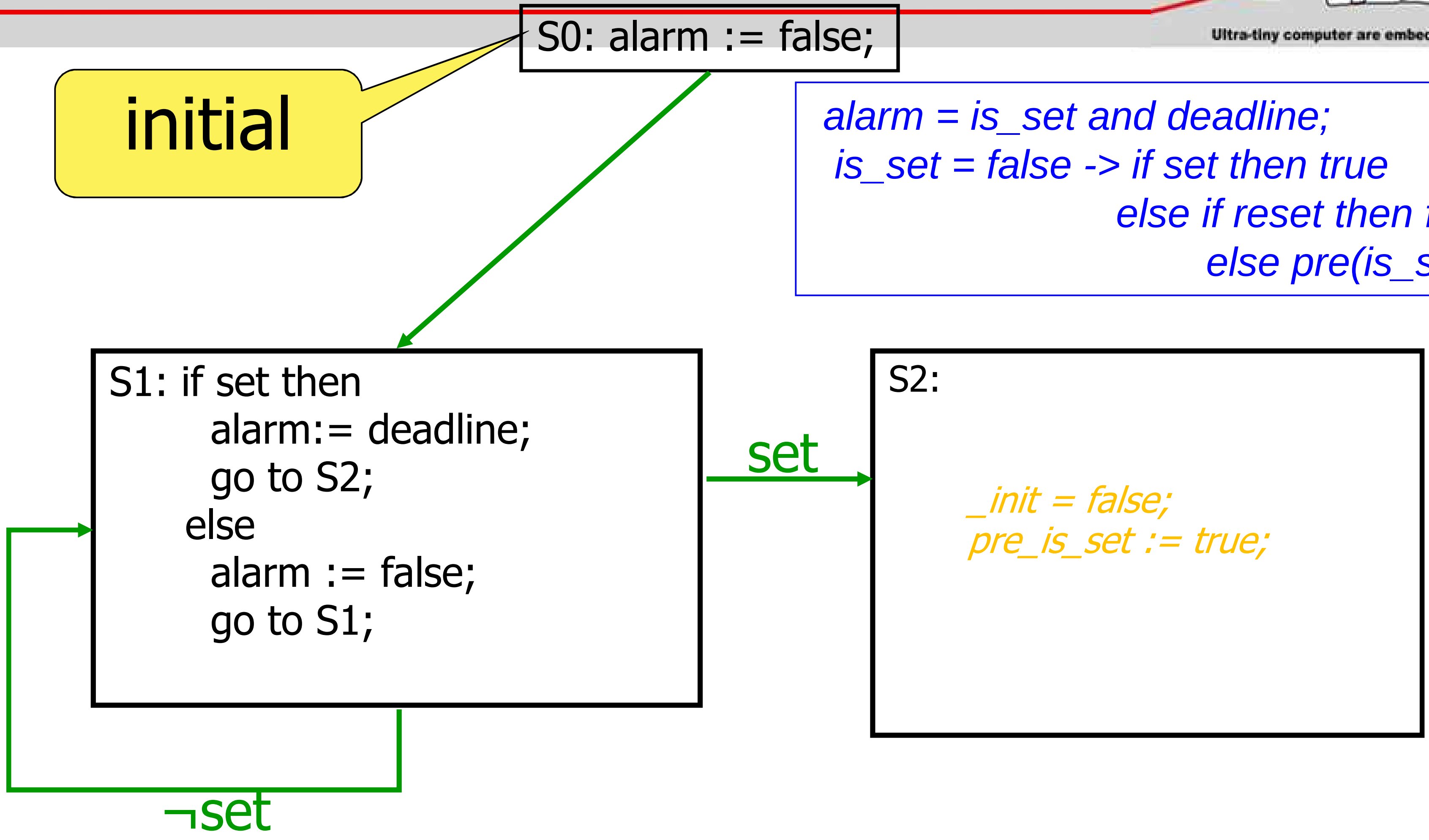
```
S0: alarm := false;
```

```
S1:
    _init := false
    pre_is_set := false
```

Lustre Program Compilation



```
alarm = is_set and deadline;
 is_set = false -> if set then true
                   else if reset then false
                      else pre(is_set);
```

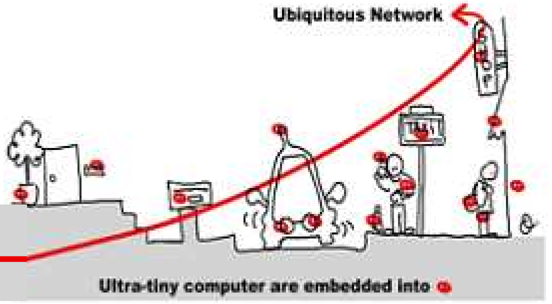


```
S0: alarm := false;
```

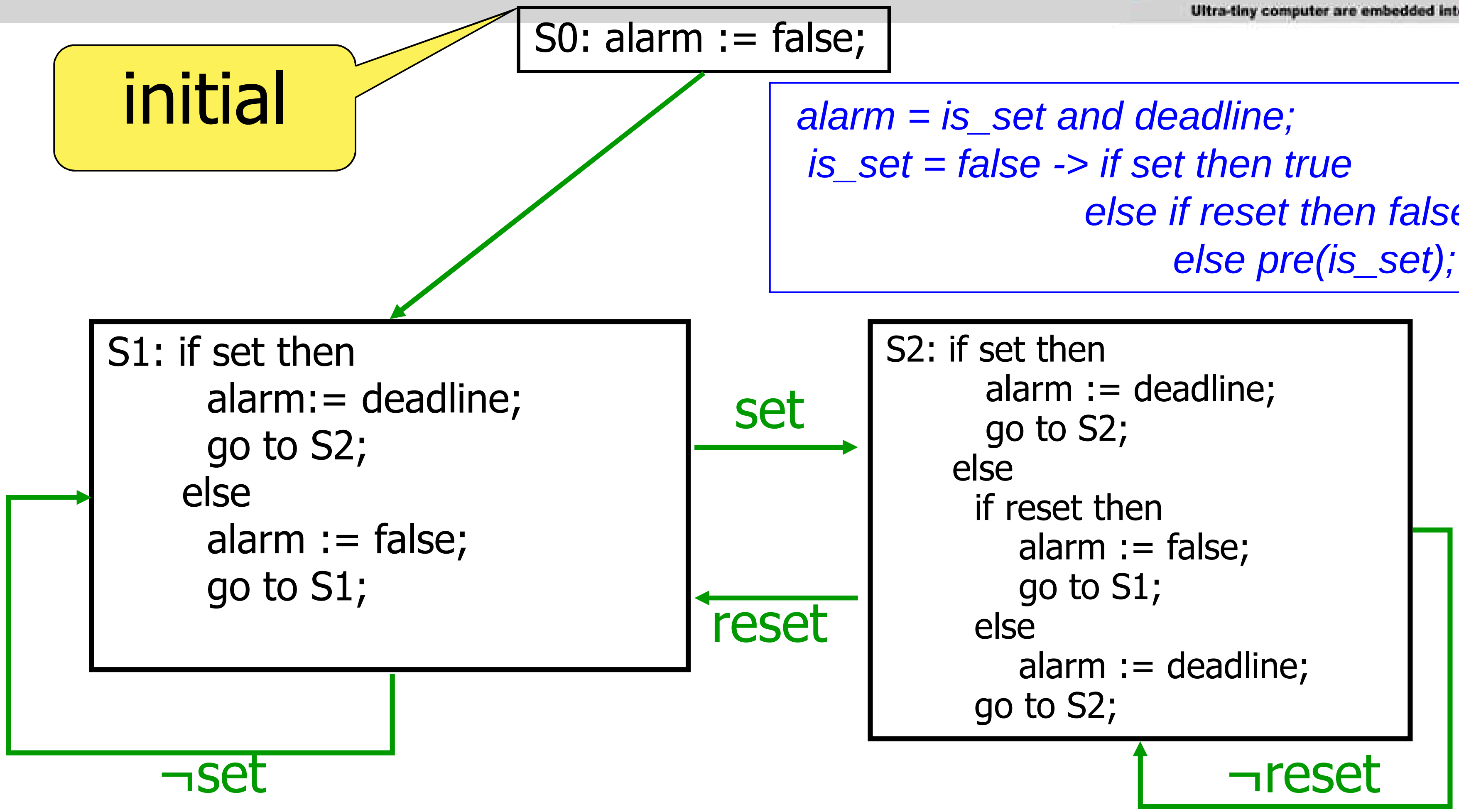
```
S1: if set then
        alarm:= deadline;
        go to S2;
      else
        alarm := false;
        go to S1;
```

```
S2:
        _init = false;
        pre_is_set := true;
```

Lustre Program Compilation



```
alarm = is_set and deadline;
 is_set = false -> if set then true
                        else if reset then false
                             else pre(is_set);
```

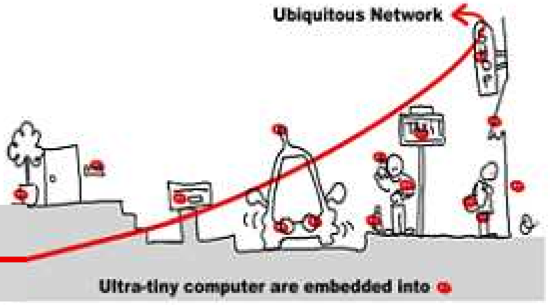


```
S0: alarm := false;
```

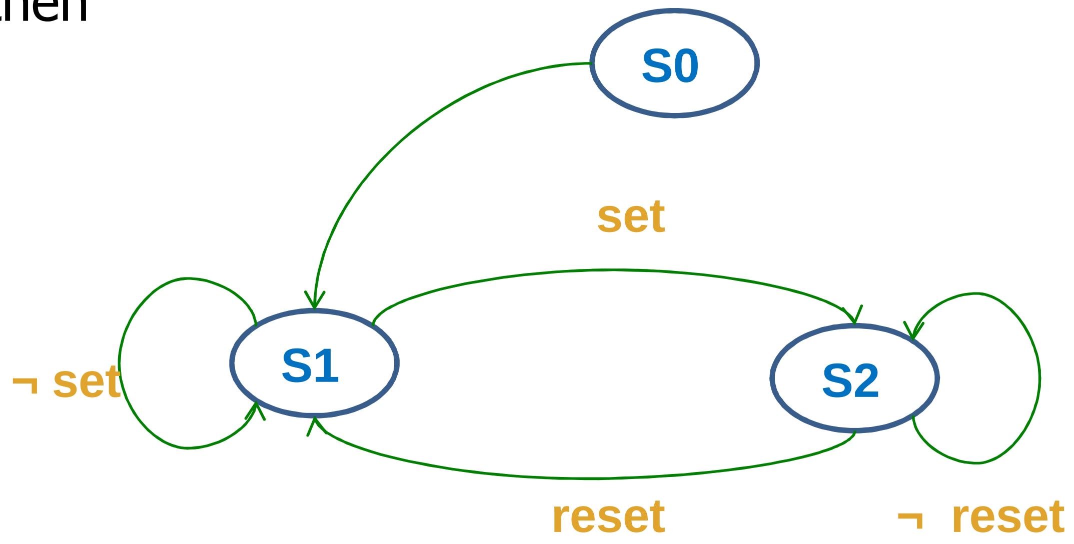
```
S1: if set then
        alarm:= deadline;
        go to S2;
      else
        alarm := false;
        go to S1;
```

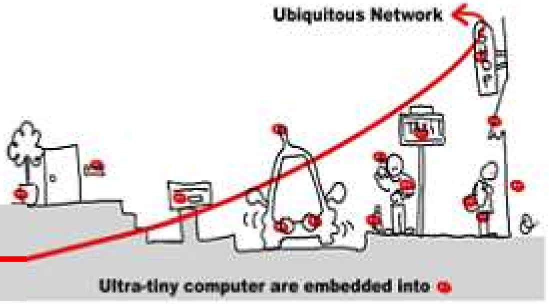
```
S2: if set then
        alarm := deadline;
        go to S2;
      else
        if reset then
            alarm := false;
            go to S1;
        else
            alarm := deadline;
        go to S2;
```

Lustre Program = Model



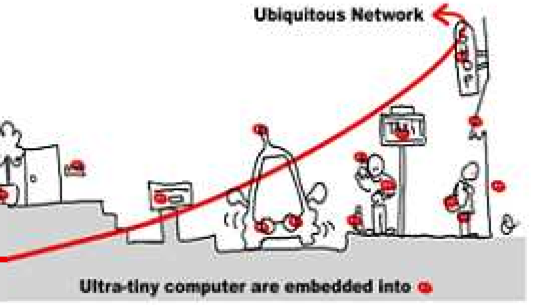
```
node WD (set, reset, deadline:bool)
   returns (alarm:bool);
var is_set:bool;
let
  alarm = is_set and deadline;
  is_set = false -> if set then true
                         else if reset then
false else pre(is_set);
  assert not(set and reset);
tel.
```


$$\begin{tikzcd}
 & & S0 \arrow[dll, bend right] \\
S1 \arrow[loop left, "\neg\ set"] \arrow[rrr, bend left, "set"] & & & S2 \arrow[lll, bend left, "reset"] \arrow[loop right, "\neg\ reset"]
\end{tikzcd}$$



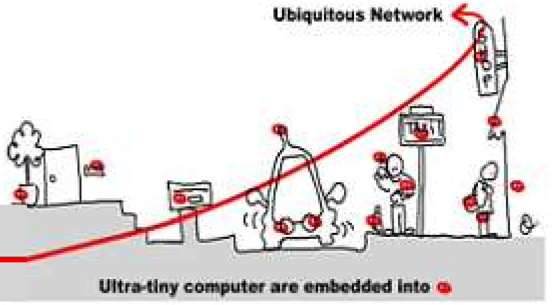
Model Checking Technique

Model Checking Technique



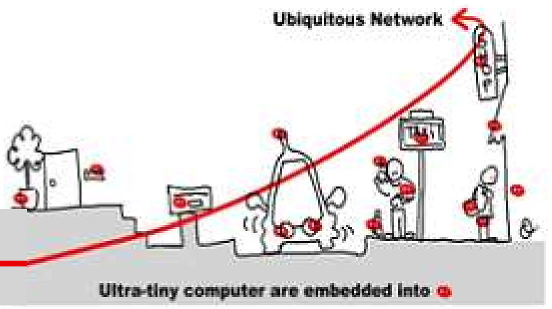
- Model = automata which is the set of program behaviors
- Properties expression = temporal logic:
  - LTL : liveness properties
  - CTL: safety properties
- Algorithm =
  - LTL : algorithm exponential wrt the formula size and linear wrt automata size.
  - CTL: algorithm linear wrt formula size and wrt automata size

Properties Checking



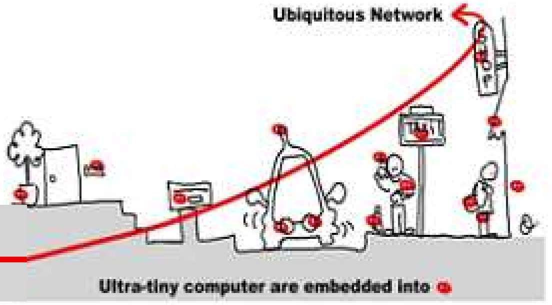
- Liveness Property Φ :
  - $\Phi \Rightarrow$ automata $B(\Phi)$
  - $\mathbb{L}(B(\Phi)) = \varnothing$ décidable
  - $\Phi \models \mathcal{M}$: $\mathbb{L}(\mathcal{M} \otimes B(\sim\Phi)) = \varnothing$

Safety Properties



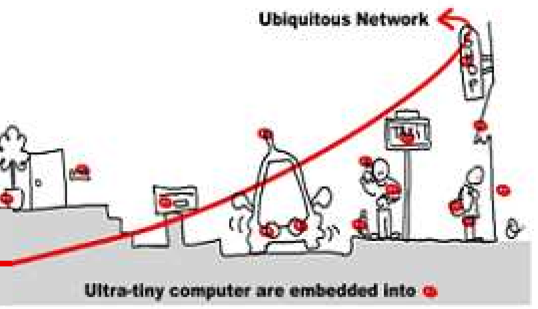
- CTL formula characterization:
  - Atomic formulas
  - Usual logic operators: not, and, or ($\Rightarrow$)
  - Specific temporal operators:
    - EX $\varnothing$, EF $\varnothing$, EG $\varnothing$
    - AX $\varnothing$, AF $\varnothing$, AG $\varnothing$
    - EU($\varnothing_1$,$\varnothing_2$), AU($\varnothing_1$,$\varnothing_2$)

Safety Properties Verification

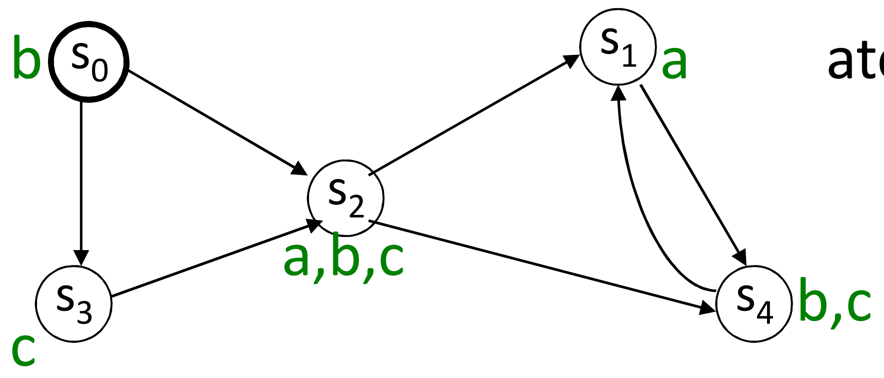


- We call Sat(∅) the set of states where ∅ is true.
- $\mathcal{M}$ |= ∅ iff $s_{init} \in$ Sat(∅).
- Algorithm:
  - Sat(Φ) = { s | Φ |= s}
  - Sat(not Φ) = S\Sat(Φ)
  - Sat(Φ1 or Φ2) = Sat(Φ1) U Sat(Φ2)
  - Sat (EX Φ) = {s | ∃ t ∈ Sat(Φ) , s → t} (Pre Sat(Φ))
  - Sat (EG Φ) = *gfp* (Γ(x) = Sat(Φ) ∩ Pre(x))
  - Sat (E(Φ1 U Φ2)) = *lfp* (Γ(x) = Sat(Φ2) U (Sat(Φ1) ∩ Pre(x))

Example



$$\begin{tikzcd}
s_0 \arrow[d] \arrow[r] & s_2 \arrow[r] \arrow[rr] & s_1 \arrow[r] & s_4 \arrow[l, bend left] \\
s_3 \arrow[ru] & & &
\end{tikzcd}$$



atomic formulas: a, b, c

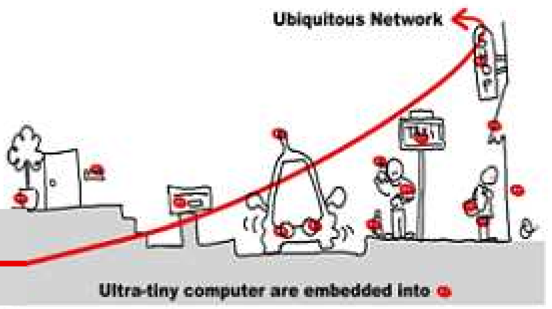
EG (a or b)

gfp (Γ(x) = Sat(a or b) $\cap$ Pre(x))

$\Gamma(\{s_0, s_1, s_2, s_3, s_4\})$ = Sat (a or b) $\cap$ Pre($\{s_0, s_1, s_2, s_3, s_4\}$)

$\Gamma(\{s_0, s_1, s_2, s_3, s_4\}) = \{s_0, s_1, s_2, s_4\} \cap \{s_0, s_1, s_2, s_3, s_4\}$

$\Gamma(\{s_0, s_1, s_2, s_3, s_4\}) = \{s_0, s_1, s_2, s_4\}$



Example

$$\begin{tikzcd}
b\ s_0 \arrow[dr] \arrow[dd] & & s_1\ a \arrow[dd, bend left] \\
& s_2\ a,b,c \arrow[ur] \arrow[dr] & \\
c\ s_3 \arrow[ur] & & s_4\ b,c \arrow[uu, bend left]
\end{tikzcd}$$

atomic formulas: a, b, c

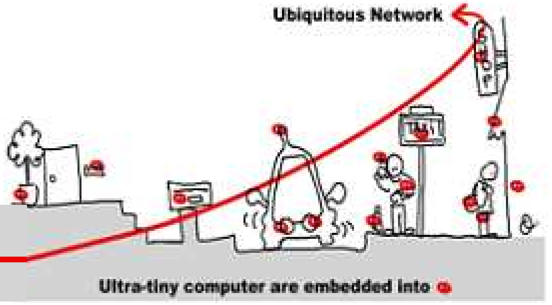
EG (a or b) $\Gamma(\{s_0, s_1, s_2, s_3, s_4\}) = \{s_0, s_1, s_2, s_4\}$

$\Gamma(\{s_0, s_1, s_2, s_4\})$ = Sat (a or b) $\cap$ Pre($\{s_0, s_1, s_2, s_4\}$)

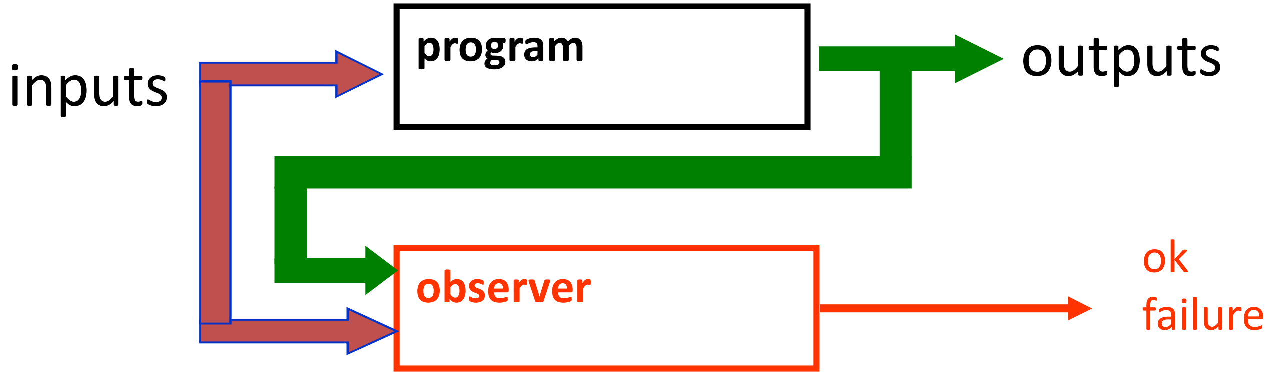
$\Gamma(\{s_0, s_1, s_2, s_4\}) = \{s_0, s_1, s_2, s_4\}$

$S_0 \models$ EG(a or b)

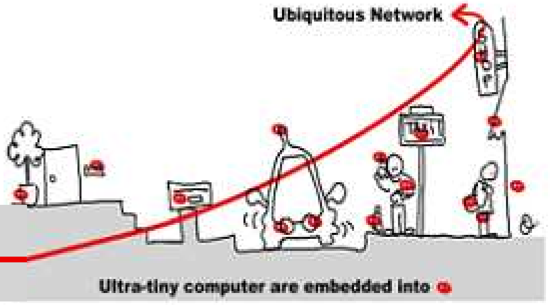
Model Checking with Observers



- Express safety properties as observers.
- An observer is a program which observes the program and outputs ok when the property holds and failure when its fails

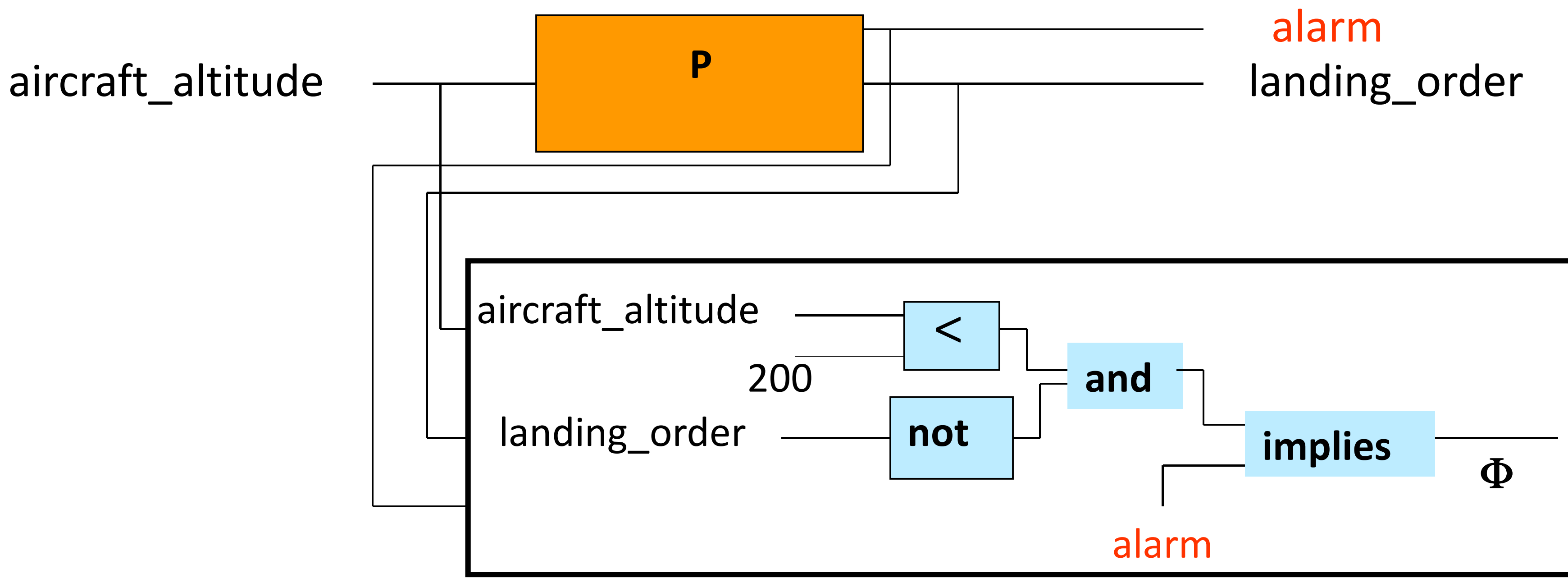


Model Checking with observers (2)

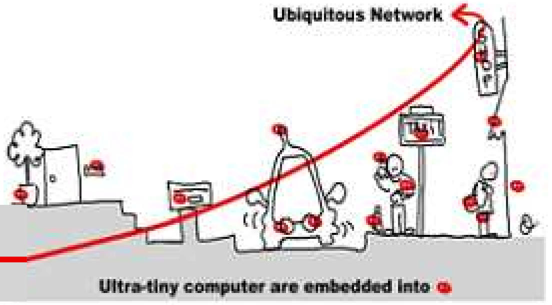


Observers in Scade

P: aircraft autopilot and security system

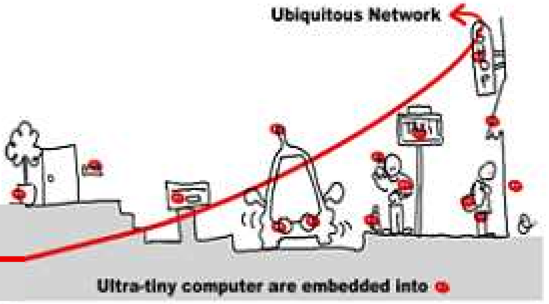


Edge Satefy Property



```
node Edge (b: bool) returns (f : bool);
let
  f = b and not pre (b);
tel
node Edge_verif (b: bool) returns (prop: bool);
 var res : bool;
 let
    res = Edge(b);
    prop = true -> res and not pre(res);
tel
```

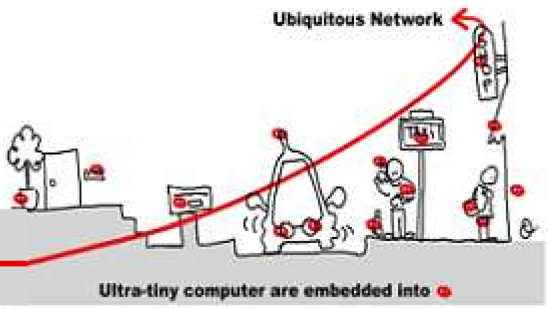
Train Safety Properties



- Example: the beacon counter in a train:
  - Count the difference between beacons and seconds
  - Decide when the train is ontime, late, early

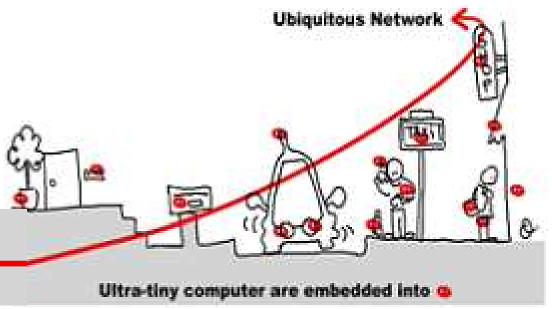
```
node train (sec, bea : bool) returns (ontime, early, late: bool)
  let
      diff = (0 ->pre diff) + (if bea then 1 else 0) + (if sec then -1 else 0);
      early = (true -> pre ontime) and (diff > 3) or
                 (false -> pre early) and (diff > 1);
      late  = (true -> pre ontime) and (diff < -3)  or
                (false -> pre late) and (diff < -1);
      ontime = not (early or late);
  tel
```

Train Safety Properties



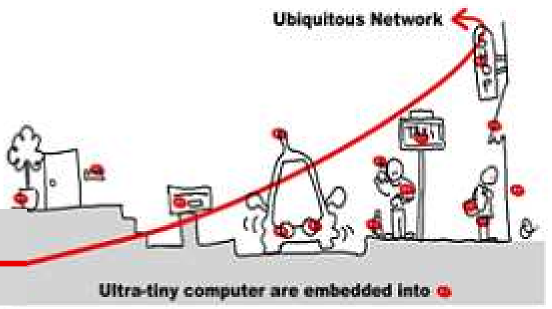
- It is impossible to be late and early;
  - ok = not (late and early)
- It is impossible to directly pass from late to early;
  - ok = true -> (not early and pre late);
- It is impossible to remain late only one instant;
  - Plate = false -> pre late;
    PPlate = false -> pre Plate;
    ok = not (not late and Plate and not PPlate);

Properties Validation

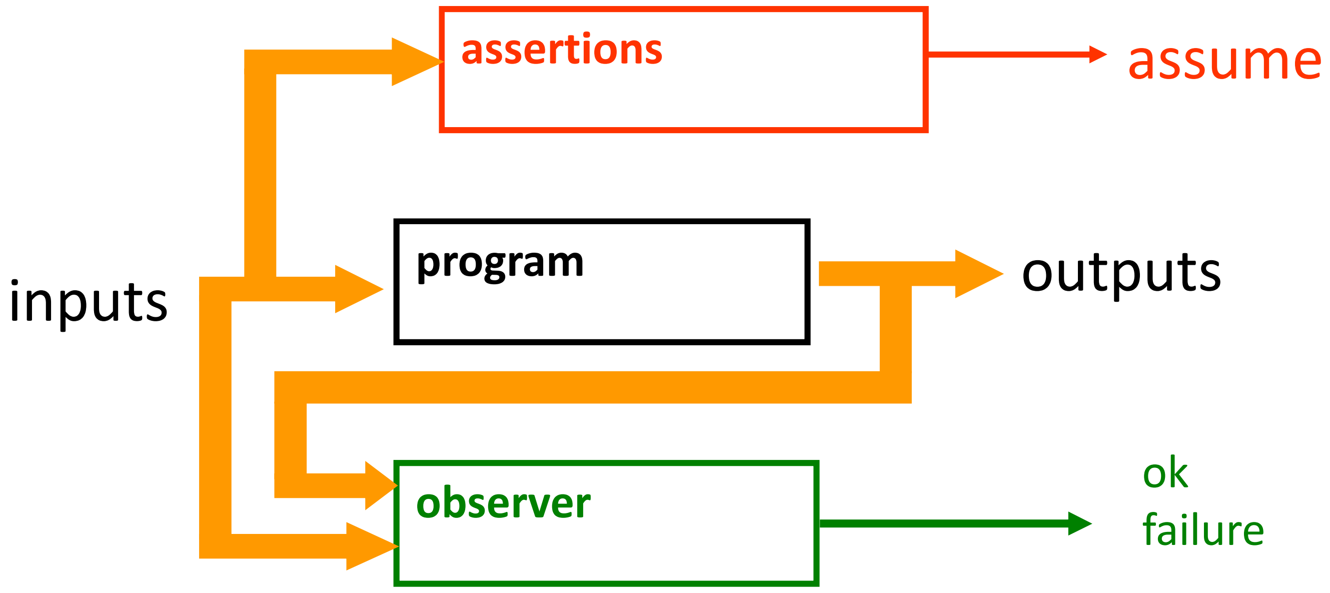


- Taking into account the environment
  - without any assumption on the environment, proving properties is difficult
  - but the environment is indeterminist
    - Human presence no predictable
    - Fault occurrence
    - …
  - Solution: use assertion to make hypothesis on the environment and make it determinist

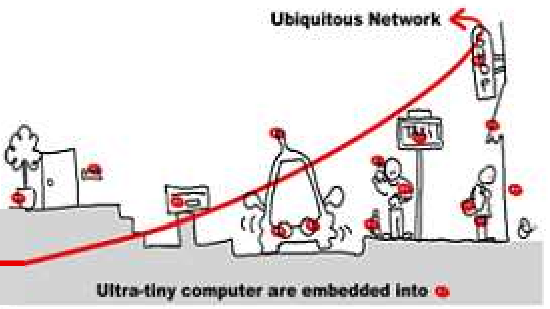
Properties Validation (2)



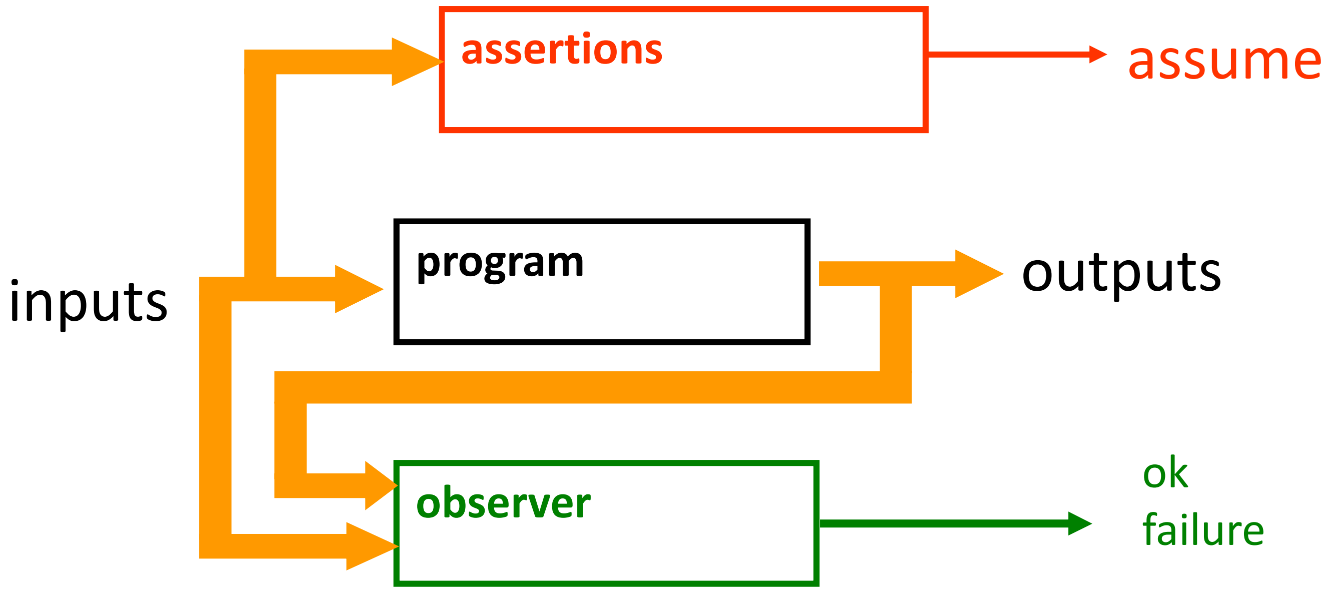
- Express safety properties as observers.
- Express constraints about the environment as assertions.



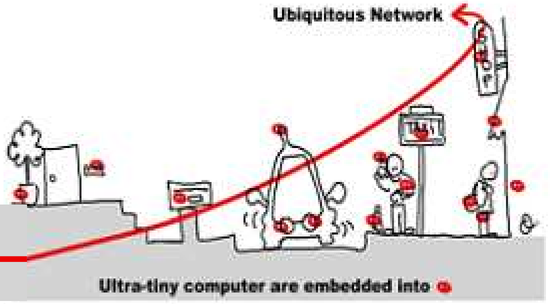
Properties Validation (3)



- if assume remains true, then ok also remains true (or failure false).

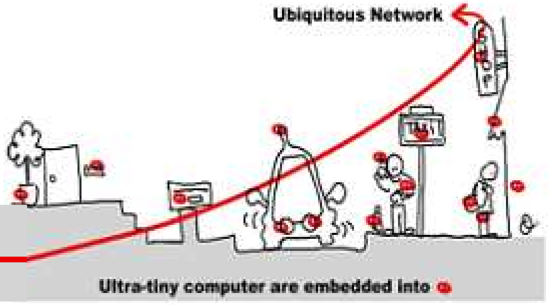


Train Assumptions

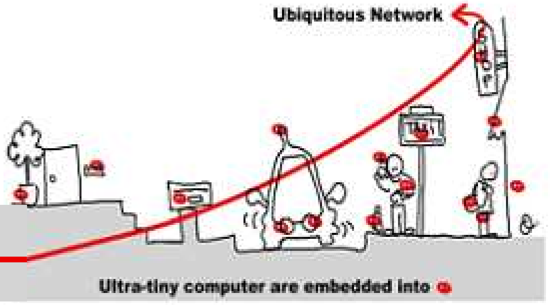


- property = assumption + observer: *" if the train keeps the right speed, it remains on time"*
- observer = ok = ontime
- assumption:
  - naïve: assume = (bea = sec);

Train Assumptions

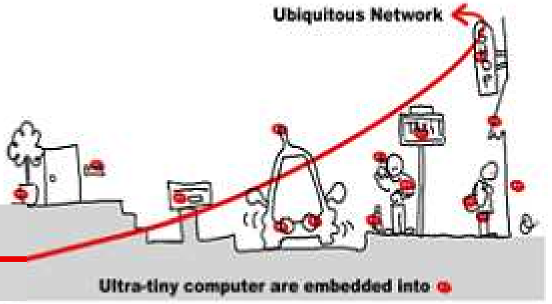


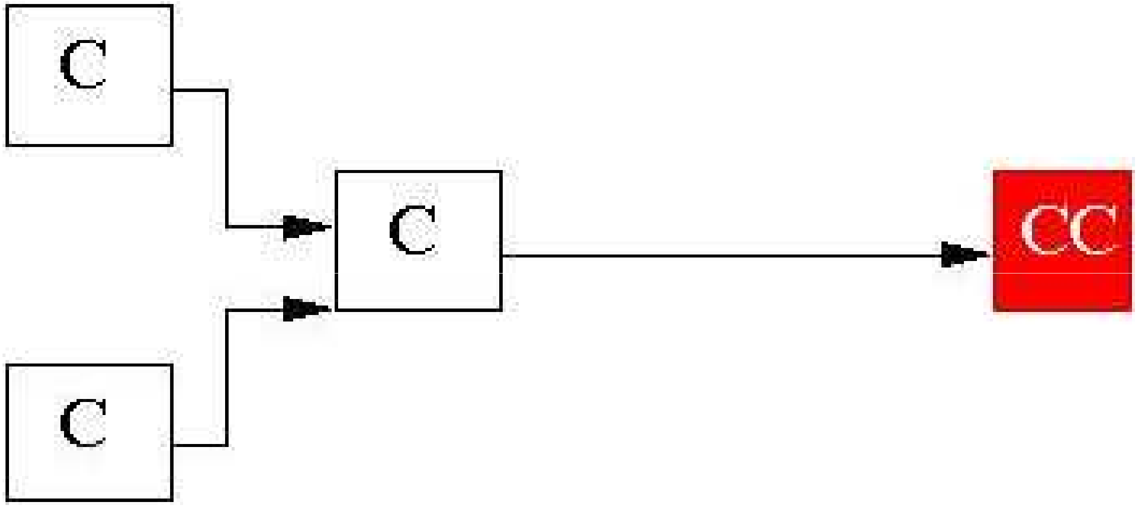
- property = assumption + observer: *" if the train keeps the right speed, it remains on time"*
- observer = ok = ontime
- assumption:
  - more precise : bea and sec alternate:
    - SF = Switch (sec and not bea, bea and not sec);
    - BF = Switch (bea and not sec, sec and not bea);
      assume = (SF => not sec) and (BF => not bea);



WComp Component Validation

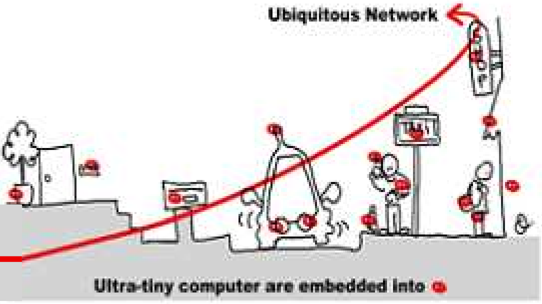
Component Validation

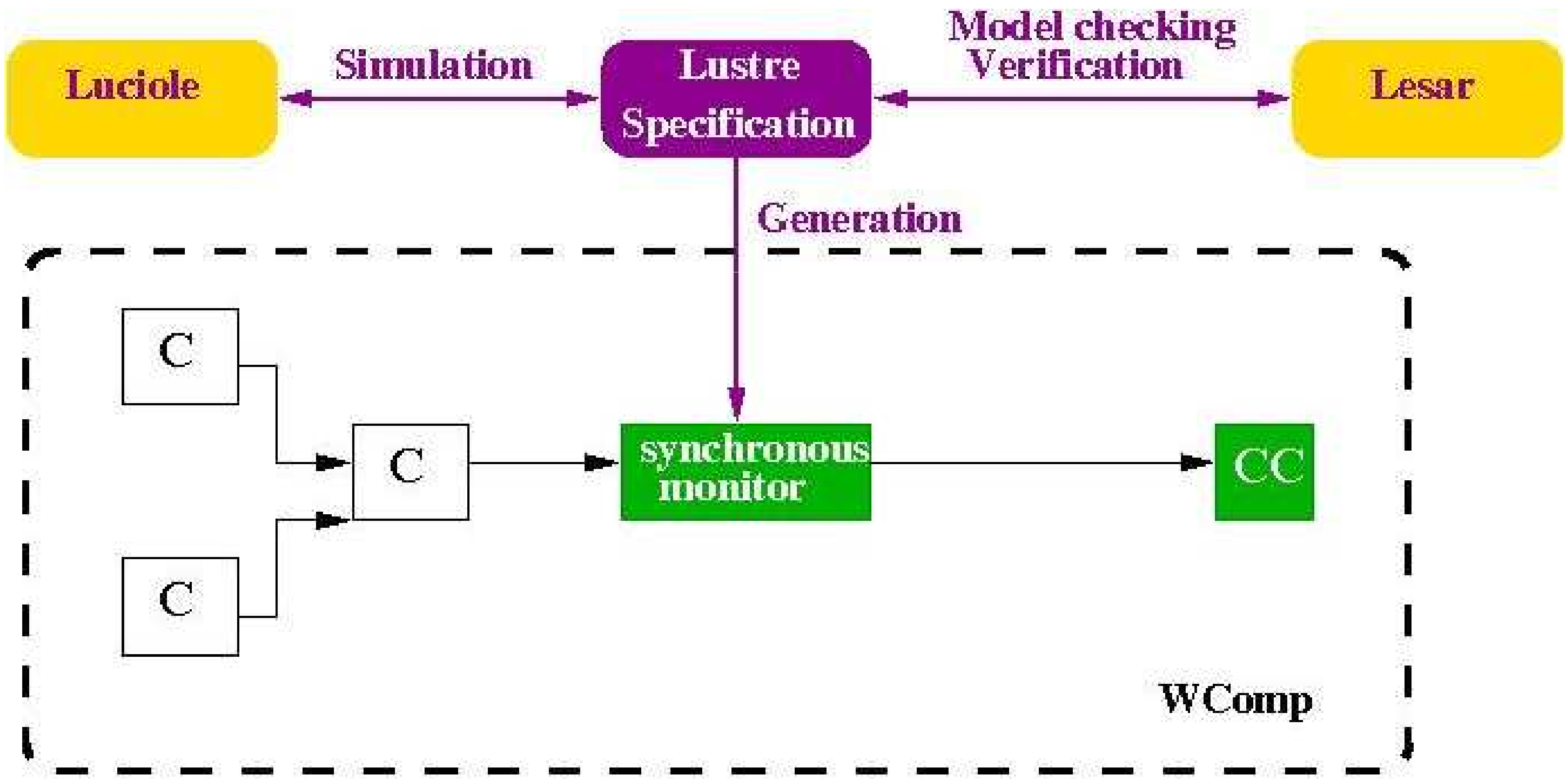




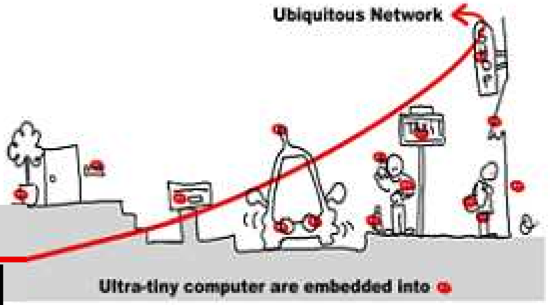
WComp critical component usage validation

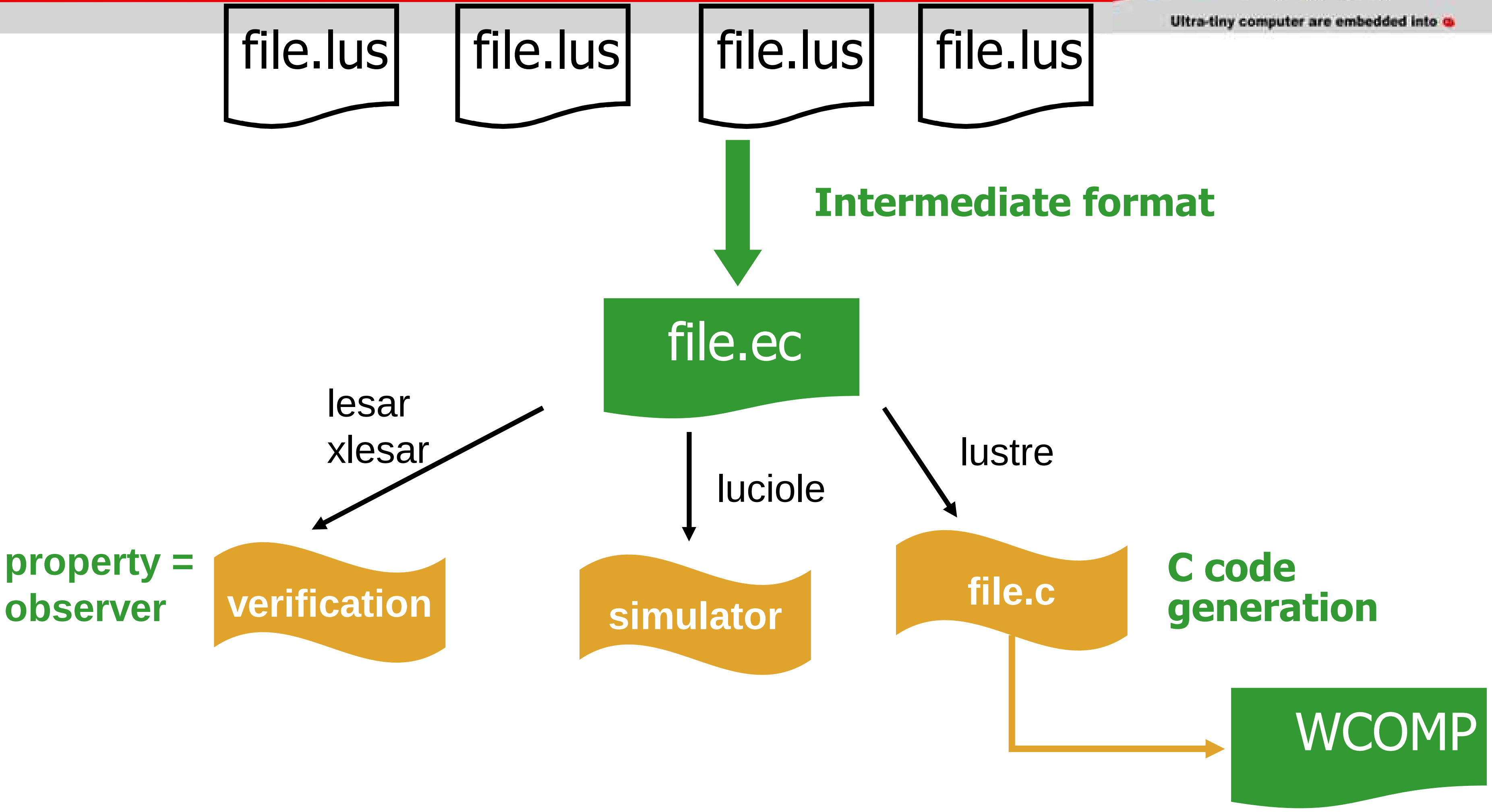
Component Validation



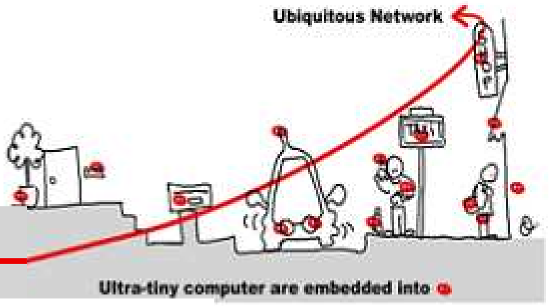


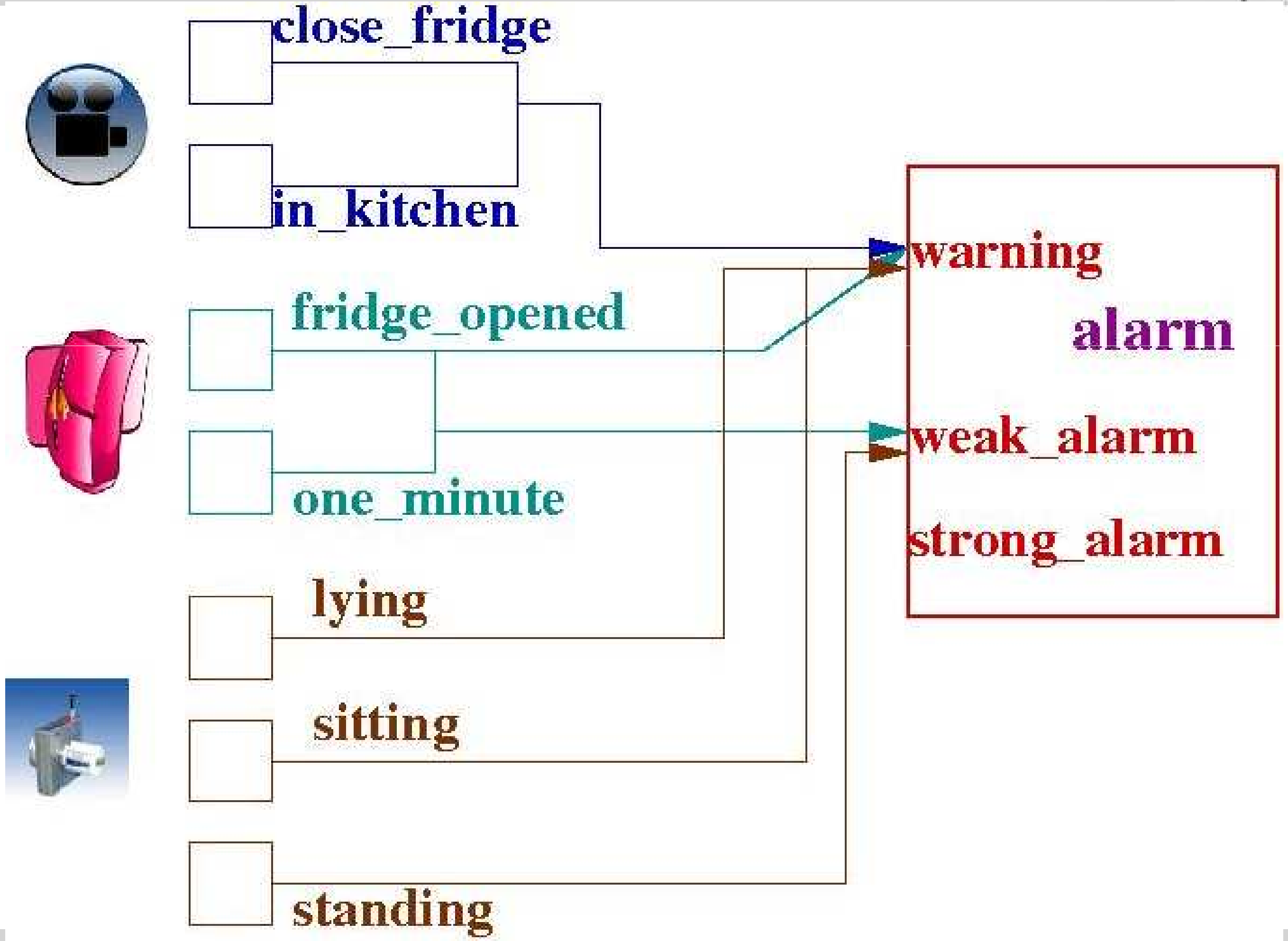
Lustre to WCOMP



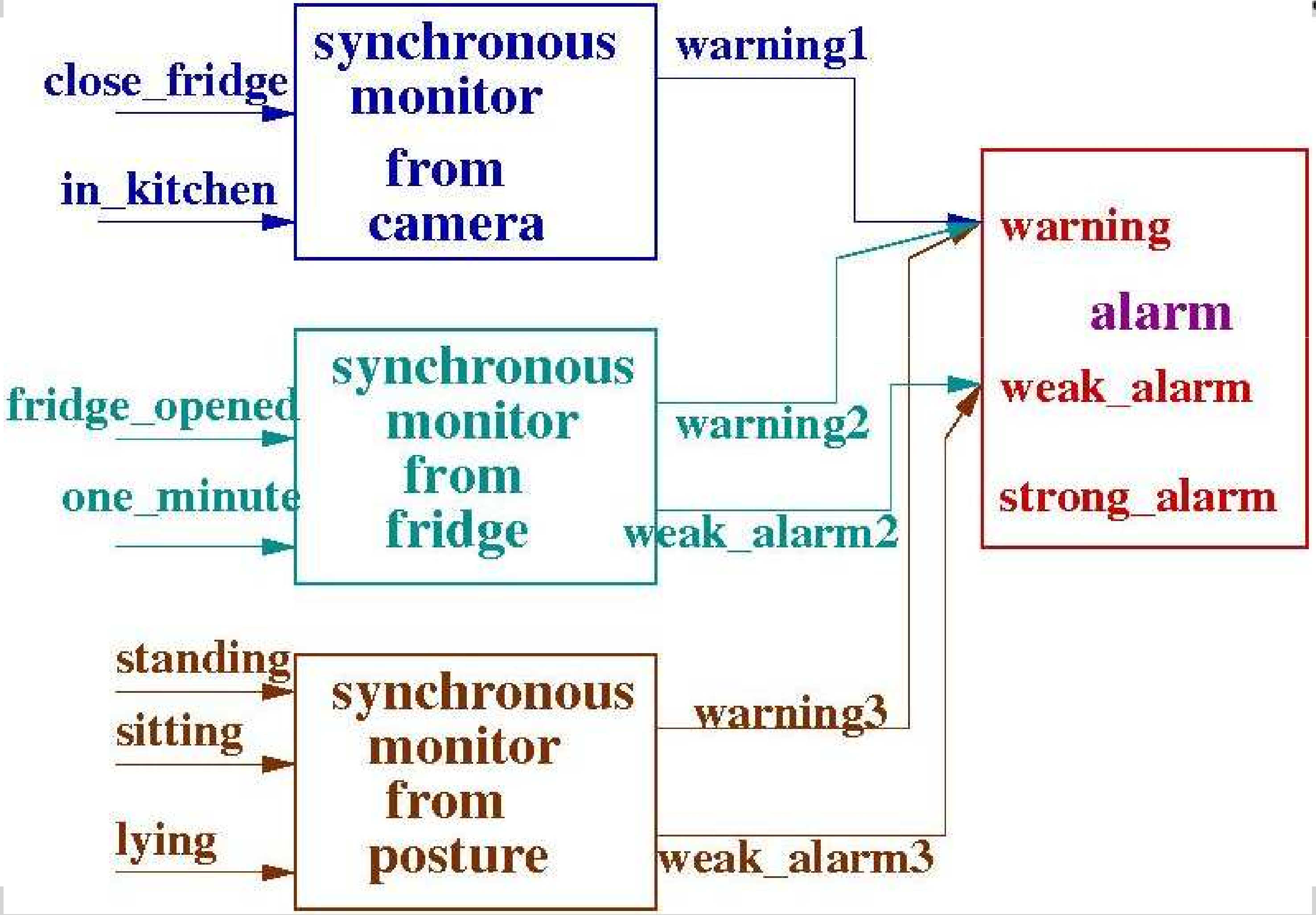


Example: monitoring eldery people at home

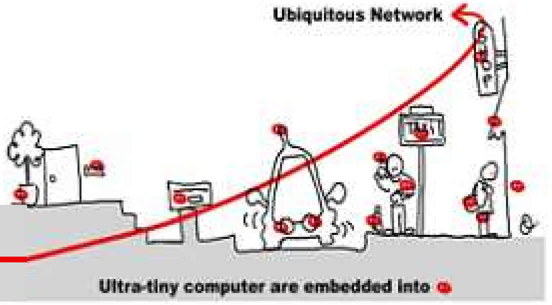


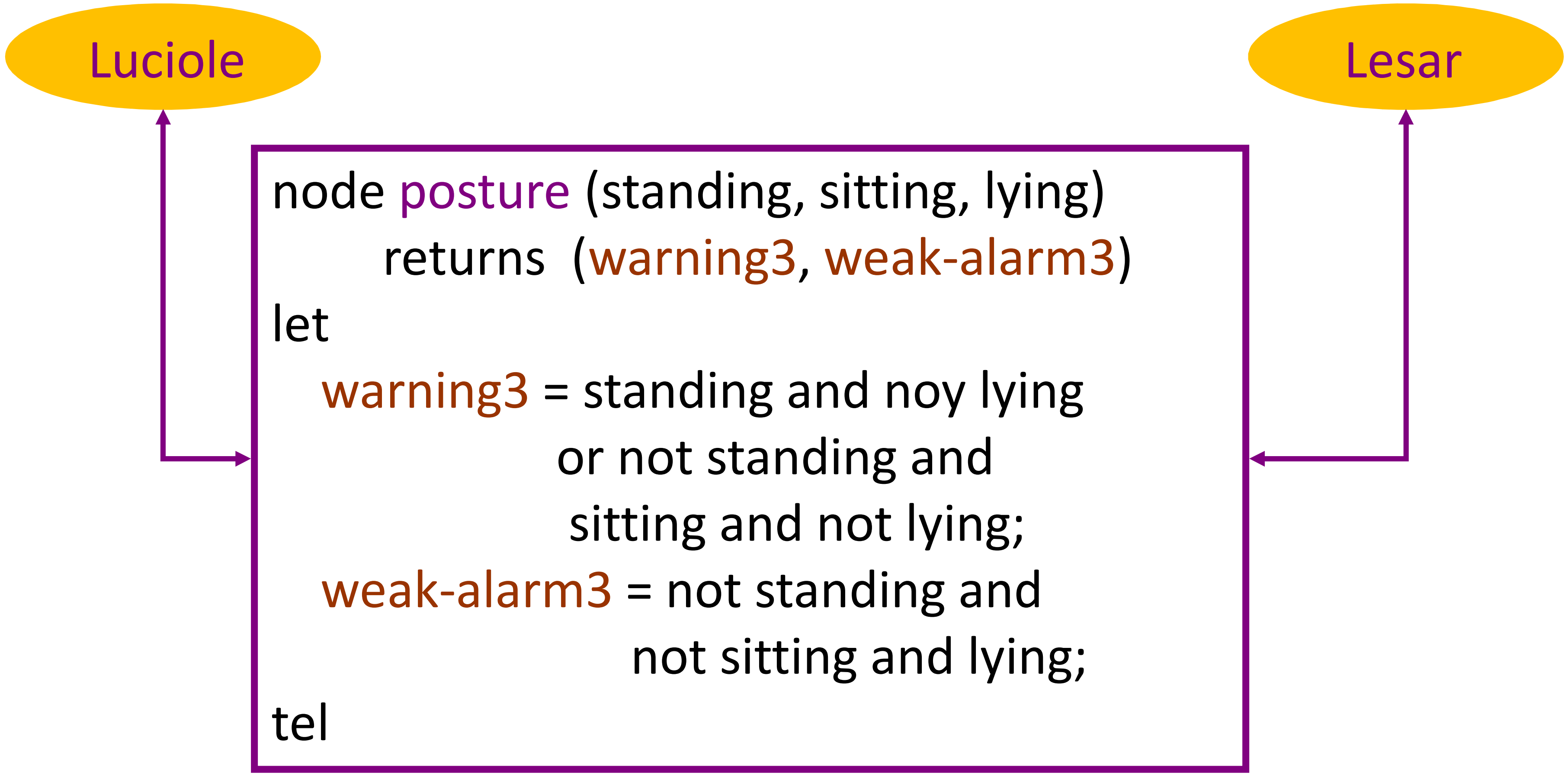


Example: Synchronous Monitors



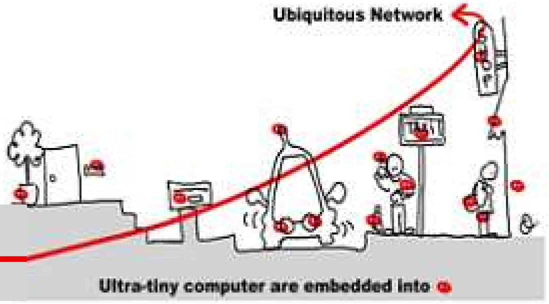
Example: Posture Monitor

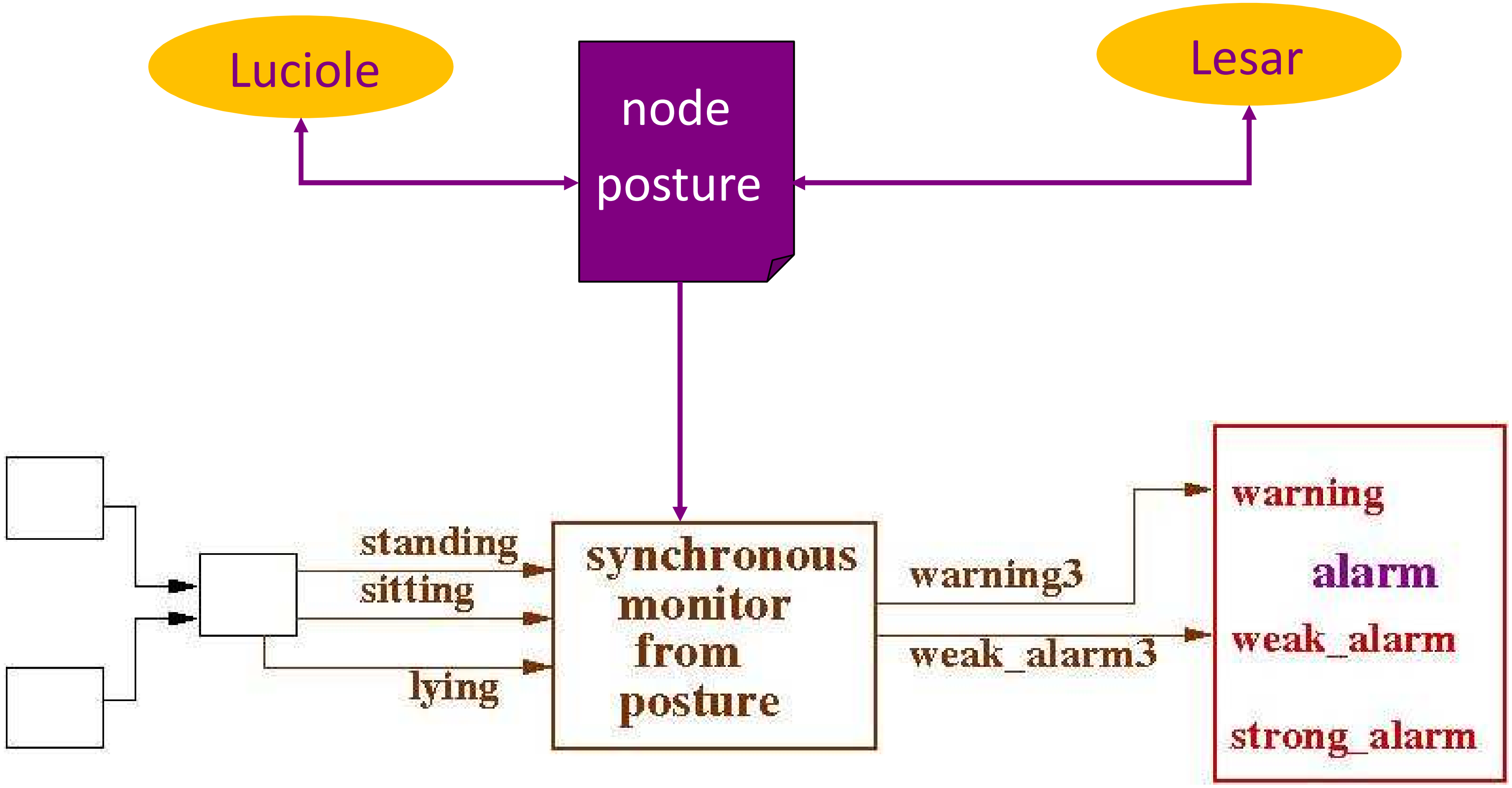




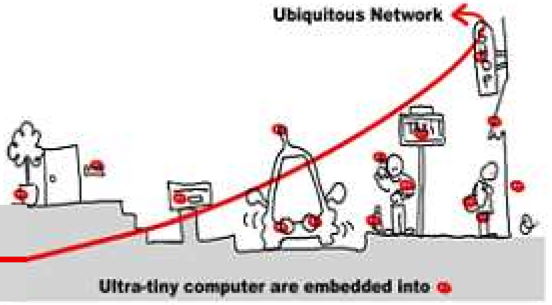
```
node posture (standing, sitting, lying)
       returns  (warning3, weak-alarm3)
let
   warning3 = standing and noy lying
                  or not standing and
                   sitting and not lying;
   weak-alarm3 = not standing and
                      not sitting and lying;
tel
```

Example: Posture Monitor



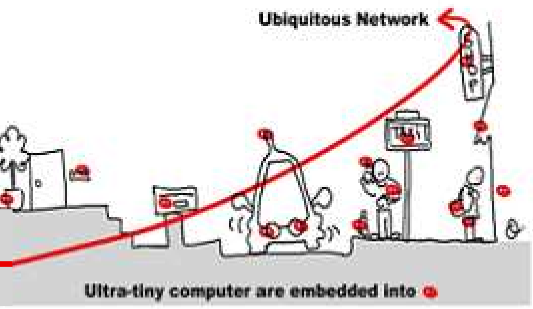


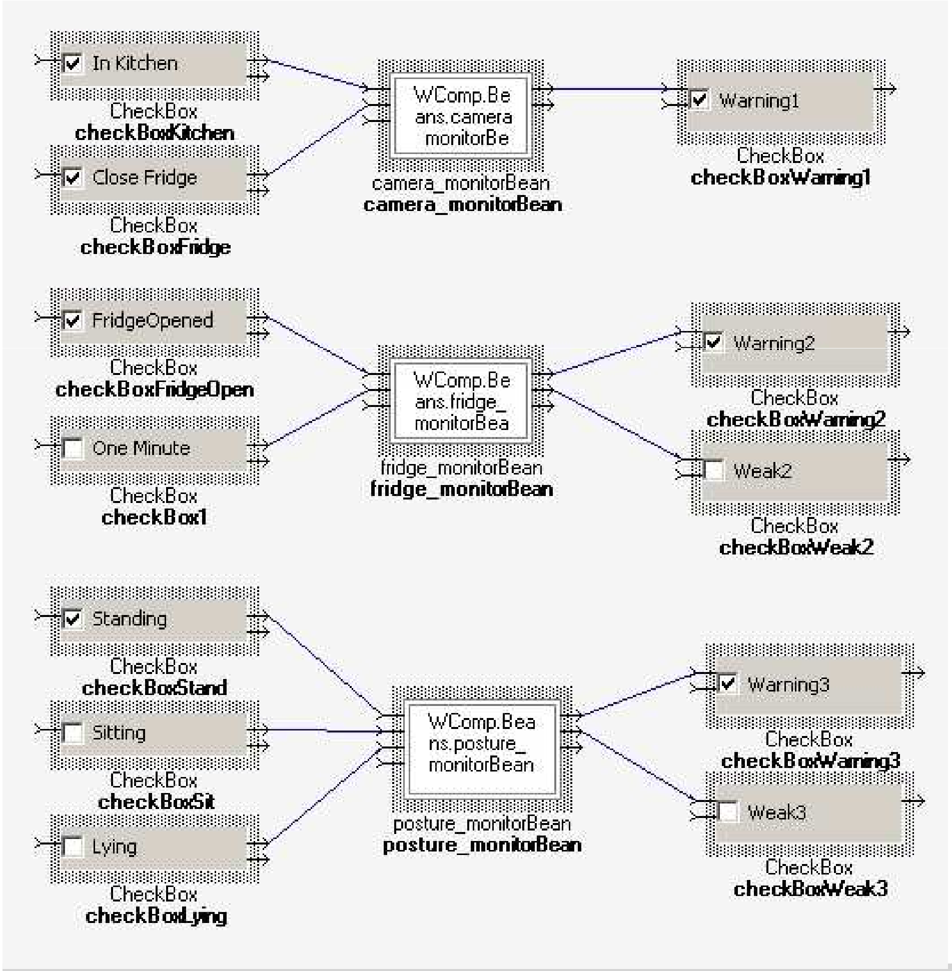
Example: camera and fridge



```
node camera  (in_kitchen, close_fridge: bool)
        returns (warning1: bool);
let
 warning1= in_kitchen and close_fridge
tel
 node fridge (fridge_opened, one_minute: bool)
        returns (warning2, weak_alarm2: bool);
 let
  warning2= fridge_opened and not one_minute;
  weak_alarm2= fridge_opened and one_minute;
 tel
```

Example: WComp Assembly

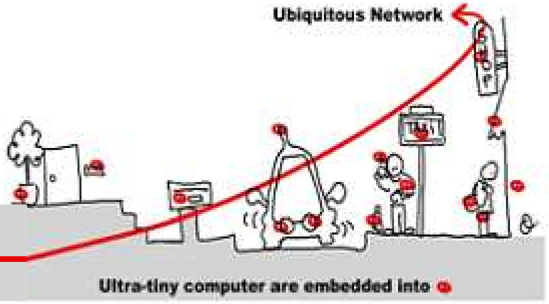




Need for synchronous monitor composition:

1. Parallel composition is obvious in Lustre (||)
2. Combination function (ζ) to specify how outputs are combined.

Example: Monitor Composition

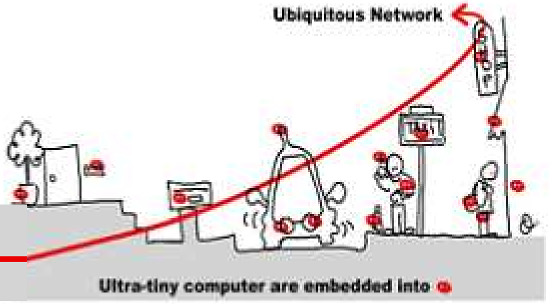


```
node comp (close_fridge, fridge_opened, one_minute, standing, sitting,
                 lying,    in_kitchen : bool)
        returns (warning, weak_alarm, strong_alarm : bool)
  var warning1, warning2, warning3, weak_alarm2, weak_alarm3 : bool;
let
  warning1 = camera (in_kitchen, close_fridge);
  (warning2, weak_alarm2) =
                        fridge (fridge_opened, one_minute);
 (warning3, weak_alarm3) =
                         position (standing, sitting, lying);

 warning = warning1 and warning2 and warning3 and not weak_alarm2
                and not weak_alarm3;
 weak_alarm = weak_alarm2 xor weak_alarm3;
 strong_alarm = weak_alarm2 and weak_alarm3;

tel
```

camera ||
fridge ||
posture

ζ

Example: Composition Verification

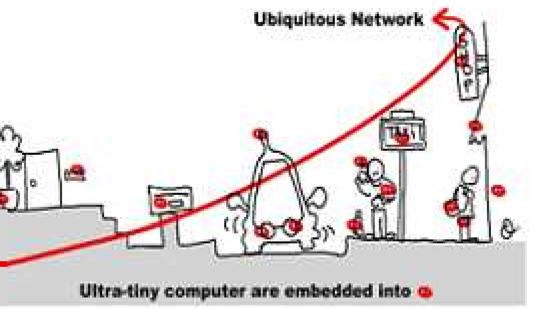


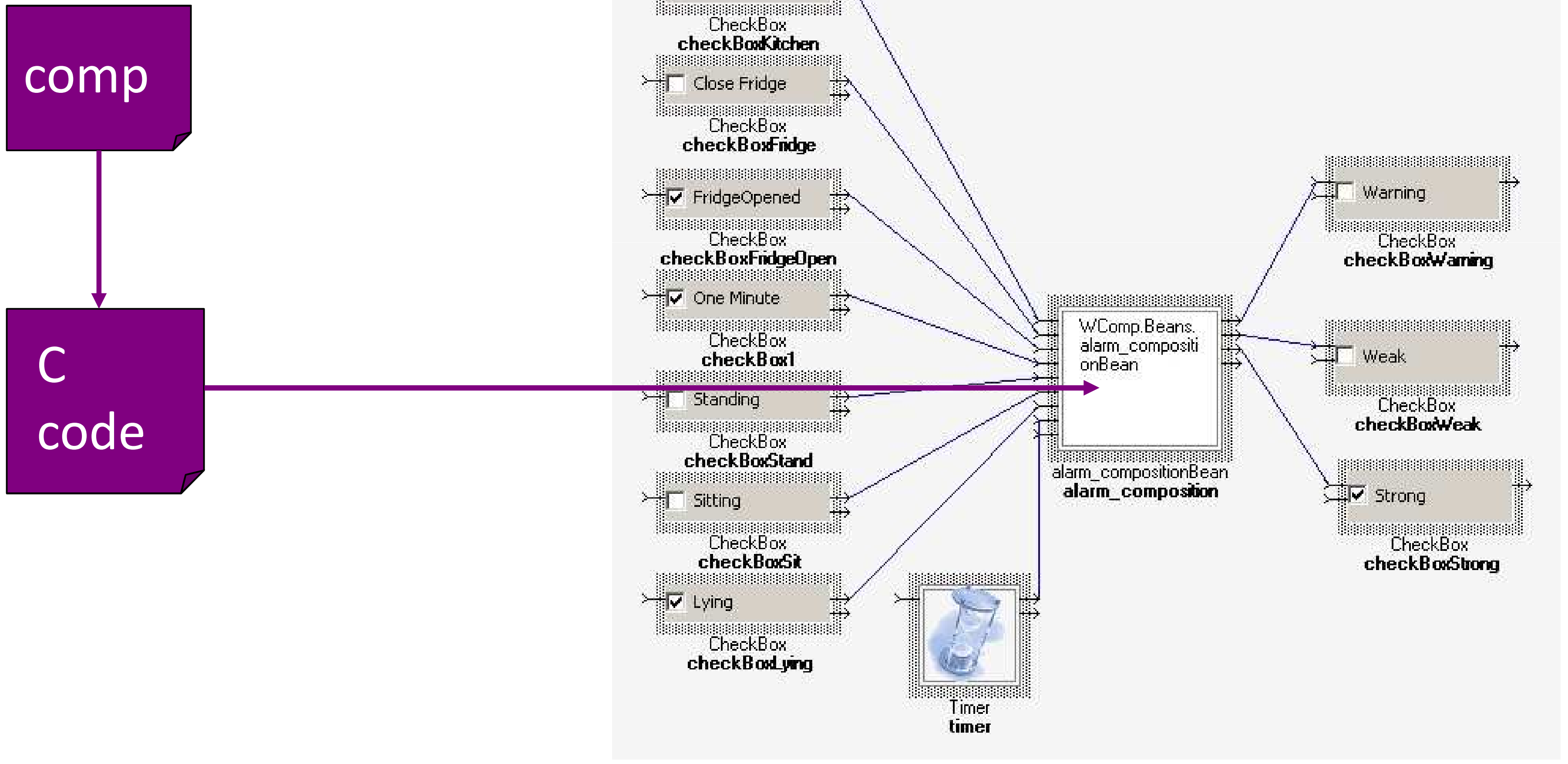
```
node verif (close_fridge, fridge_opened, one_minute,
                standing, sitting, lying, in_kitchen : bool)
        returns (prop: bool)
  var warning, weak_alarm, strong_alarm : bool;
let
  (warning, weak_alarm, strong_alarm) =
        comp(close_fridge, fridge_opened, one_minute, standing,
    sitting, lying, in_kitchen);
  assert (not ((standing and lying) or (standing and sitting) or
    (lying and sitting))
  prop = if (fridge_opened and one_minute and lying)
             then strong_alarm else true;
tel
```

Assertion on environment

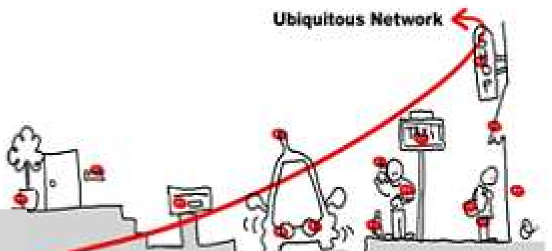
Property verified with Lesar (prop always true)

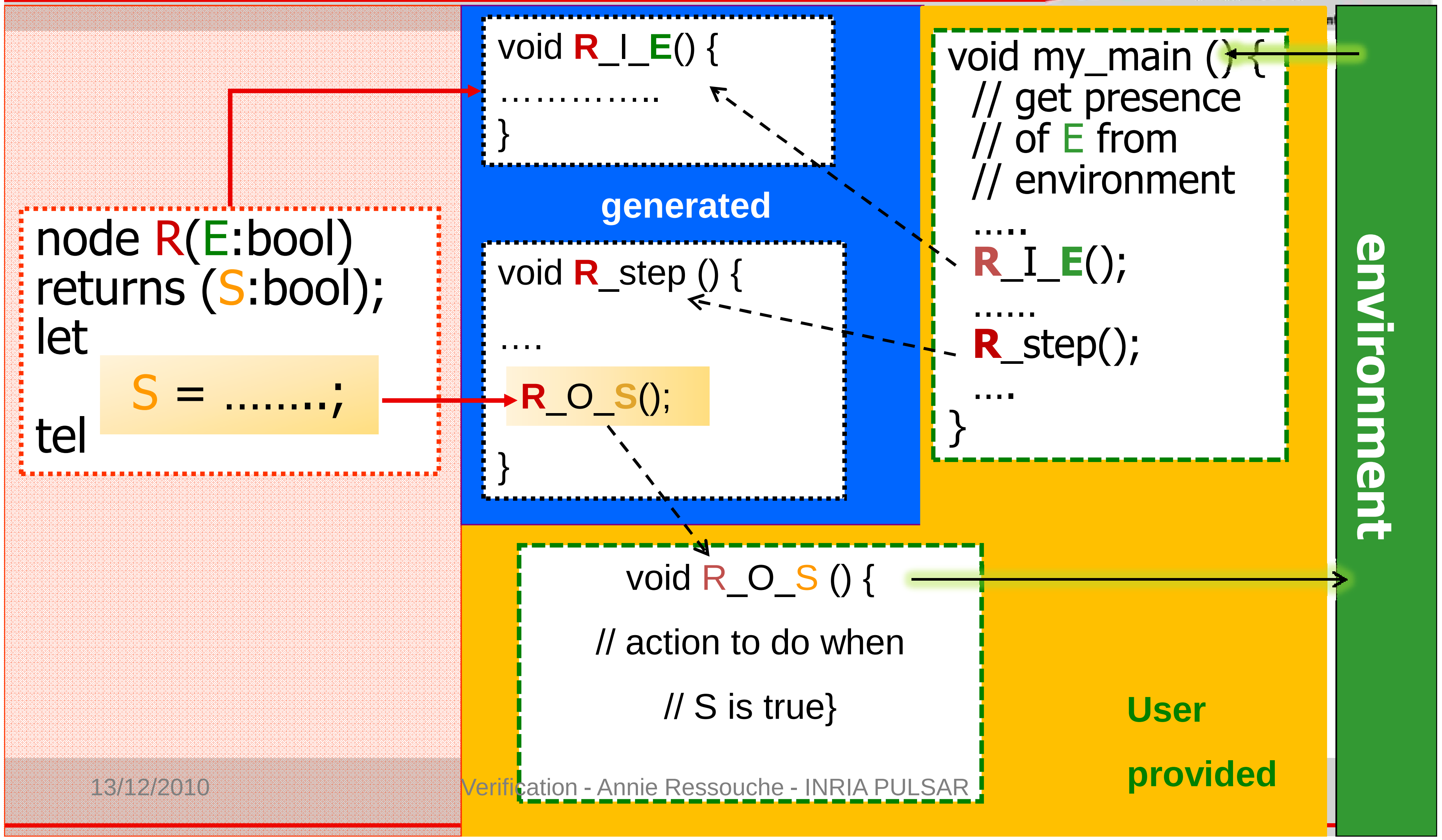
Example: WComp assembly





Lustre API





```
node R(E:bool)
returns (S:bool);
let
   S = .......;
tel
```

```
void R_I_E() {
.............
}
```

```
void R_step () {
....
R_O_S();
}
```

```
void my_main () {
  // get presence
  // of E from
  // environment
  .....
  R_I_E();
  ......
  R_step();
  ....
}
```

```
void R_O_S () {
// action to do when
// S is true}
```