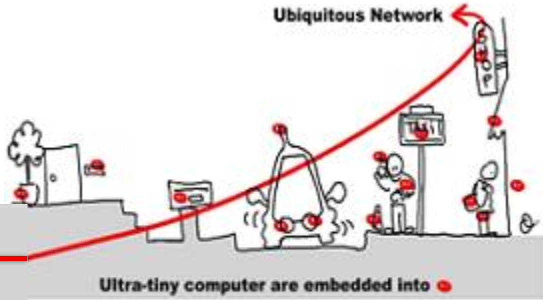
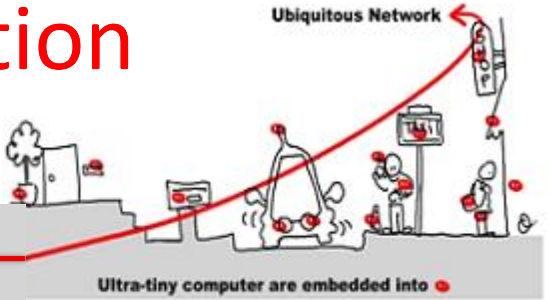


UbiComp Middleware and Verification

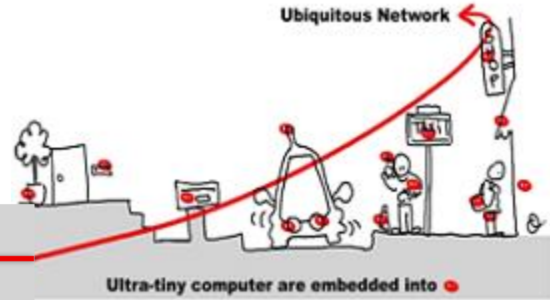
Annie Ressouche
Inria-sam (stars)
annie.ressouche@inria.fr

Ubiquitous Middleware Application Validation



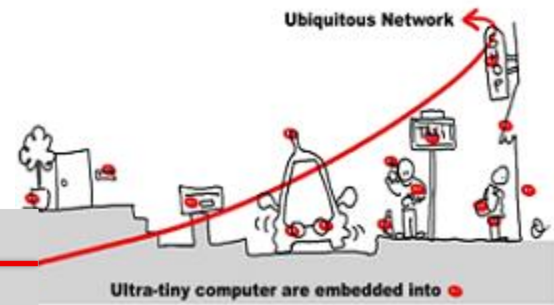
- Ubiquitous and adaptive middleware may be used to design critical applications
- Ensure a safe usage of these middleware wrt component behavior
- Apply general techniques used to develop critical software

Outline



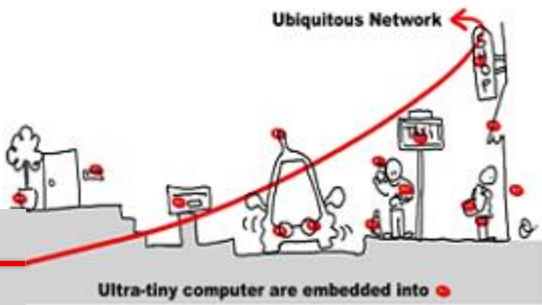
1. Critical system validation
2. Model-checking solution
   1. Model specification
   2. Model-checking techniques
3. Application to component based adaptive middleware
   1. Middleware critical component as synchronous models to allow validation
   2. The Scade and CLEM solutions

Outline



1. Critical system validation
2. Model-checking solution
    1. Model specification
    2. Model-checking techniques
3. Application to component based adaptive middleware
    1. Middleware critical component as synchronous models to allow validation
    2. The Scade and CLEM solution

Critical Software

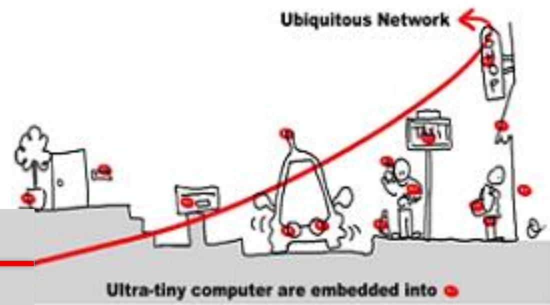


A critical software is a software whose failing has serious consequences:

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

…

Critical Software

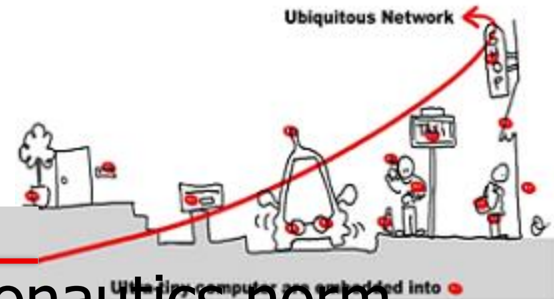


- In addition, other consequences are relevant to determine the critical aspect of software:
  - Financial aspect
    - Loosing equipment, bug correction
    - Equipment callback (automotive)
  - Bad advertising

Example: Ariane5 launcher

- 9 Jul 1996 Ariane5 launcher explodes
- Same software as Ariane4
- Causes:
  - Variable to carry horizontal acceleration encoded with 8 bits (ok for Ariane4, not sufficient for Ariane5)
  - Result: variable overflow
  - The rocket had an incorrect trajectory and engineers blow it up
- Cost: > 1 million euros (2 satellites lost)

Software Classification

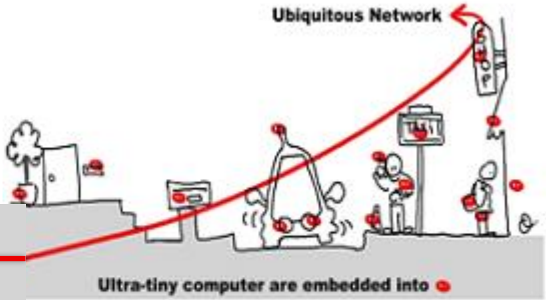


Example of the aeronautics norm DO178B:

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

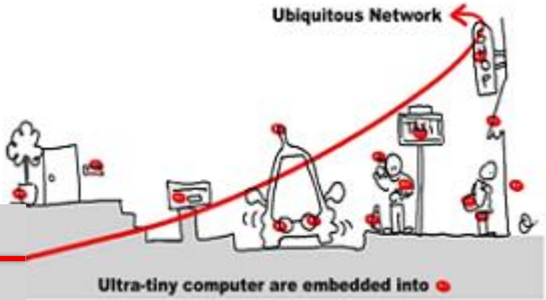
Depending of the level of risk of the system, different kinds of verification are required

Software Classification

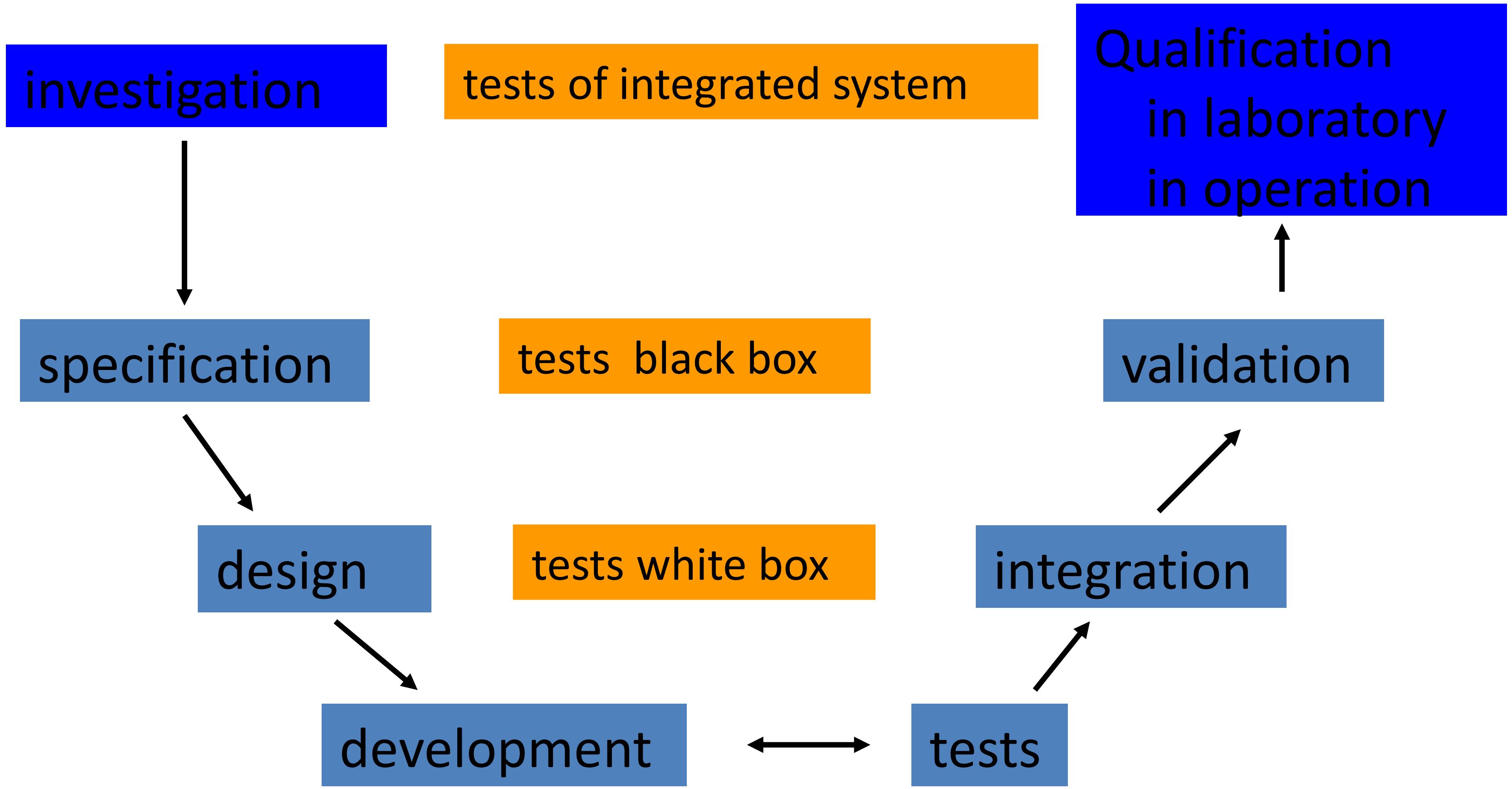


Minor			acceptable situation	
Major				
Dangerous	Unacceptable situation			
catastrophic	10^{-3} / hour	10^{-6} / hour	10^{-9}/hour	10^{-12}/hour
probabilities	probable	rare	very rare	very improbable

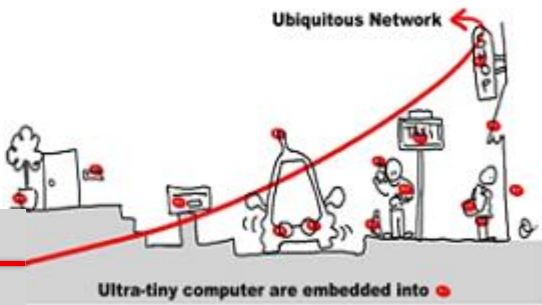
How Develop critical software ?



Classical Development U Cycle

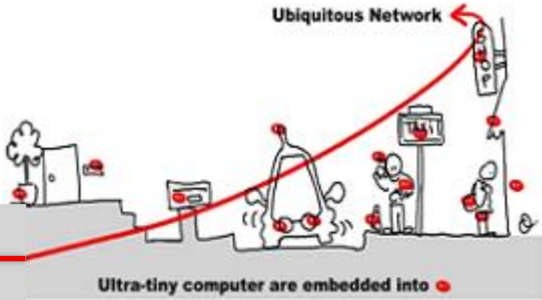


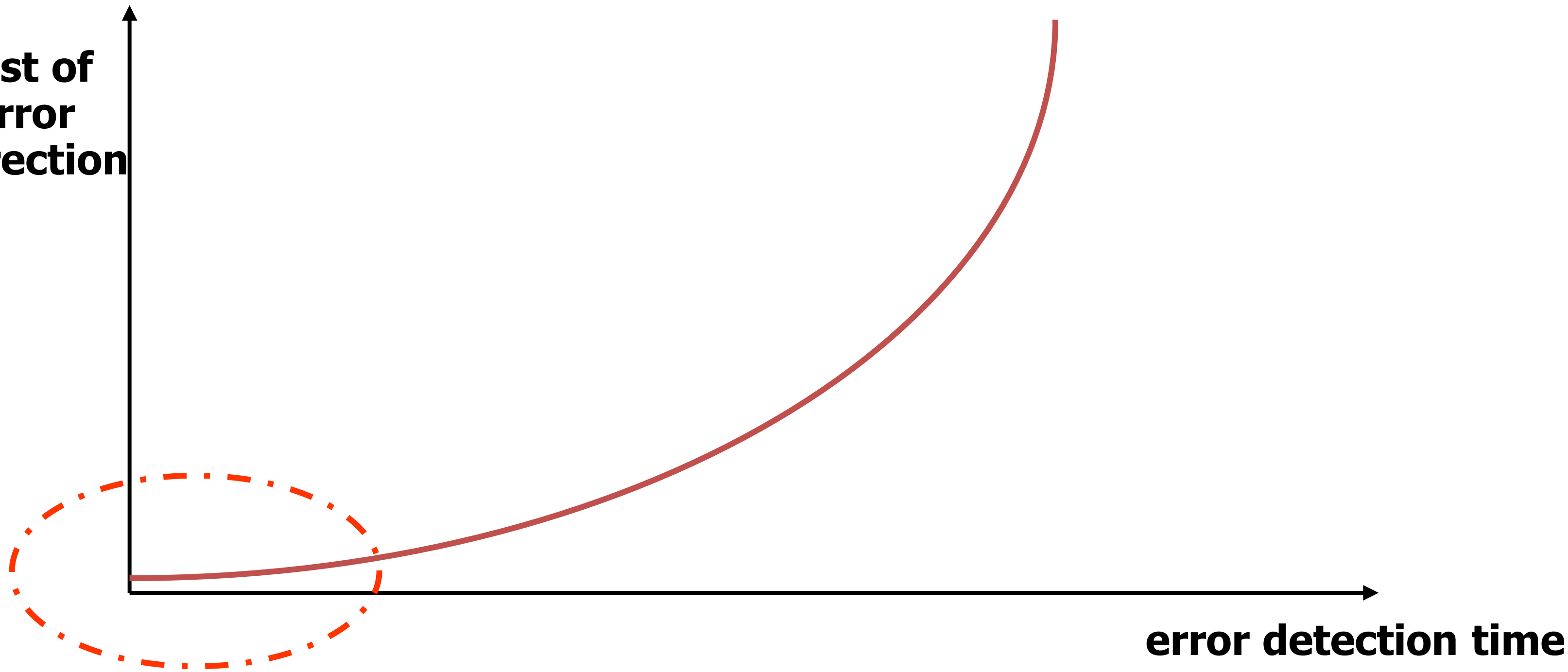
How Develop Critical Software ?



- 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.

Cost of Error Correction

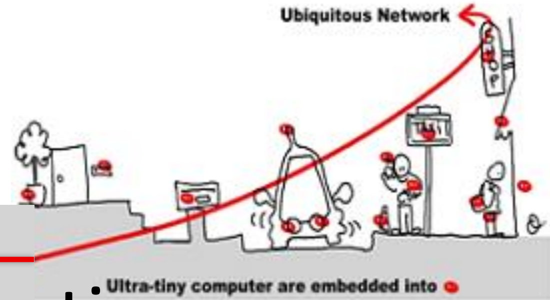




Put the effort on the upstream phase

↳ **development based on models**

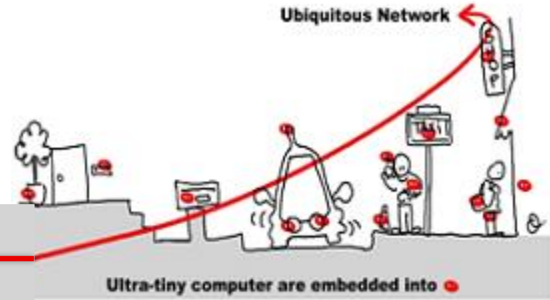
How Develop Critical Software ?



- 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

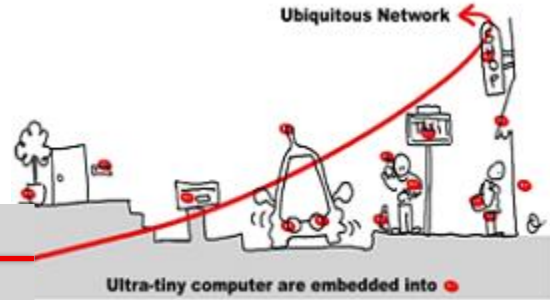
⇒ Formal model usage

Critical Software Specification

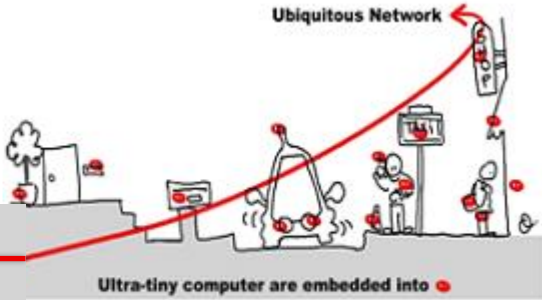


- First Goal: must yield a formal description of the application needs:
  - Standard to allowing communication between computer science engineers and non computer science ones
  - General enough to allow different kinds of application:
    - Synchronous (and/or)
    - Asynchronous (and/or)
    - Algorithmic

Critical Software Specification



- Second Goal: allowing errors detection carried out upstream:
  - Validation of the specification:
    - Coherency
    - Completeness
    - Proofs
  - Test
    - Quick prototype development
    - Specification simulation



Example of non completeness

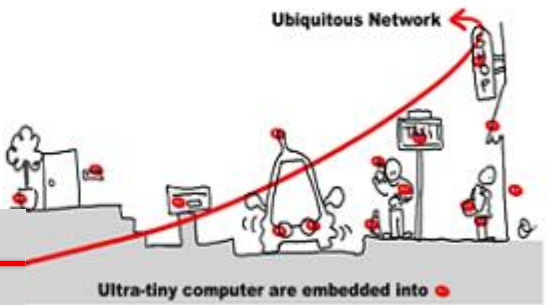
From Ariane 5:

helium tank low ⇩ **action**

Simultaneous events ? ⇩ **unspecified action**

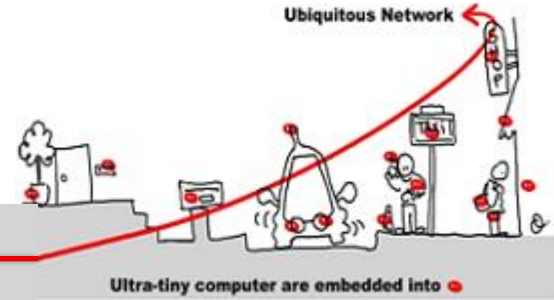
hydrogen tank low ⇩ **action**

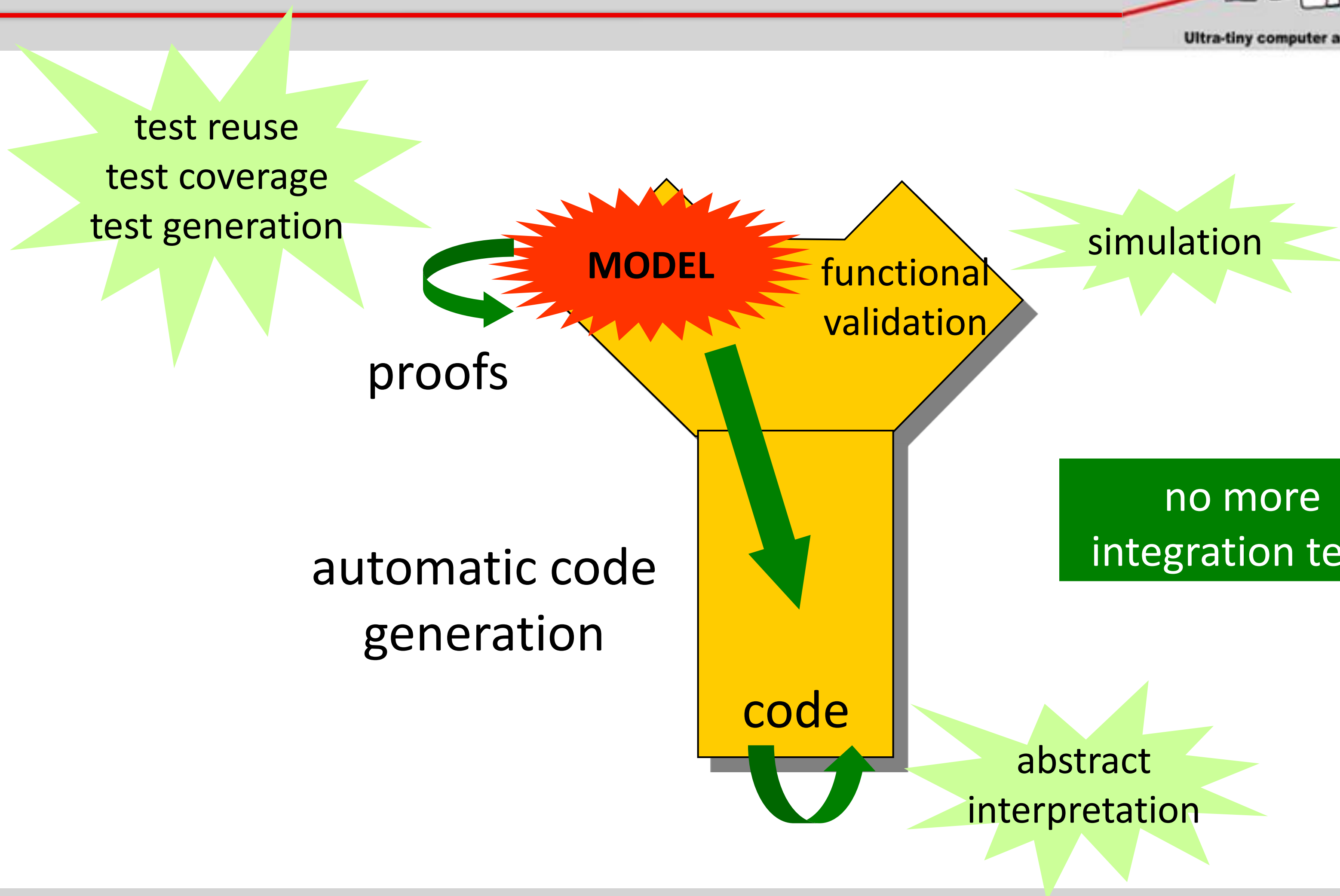
Critical Software Specification



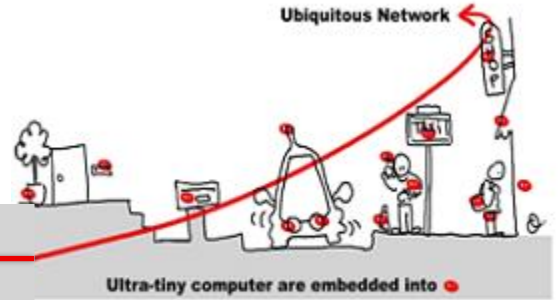
- Third goal: make easier the transition from specification to design (refinement)
  - – Reuse of specification simulation tests
  - – Formalization of design
  - – Code generation
    - Sequential/distributed
    - Toward a target language
    - Embedded/qualified code

How Develop Critical Software

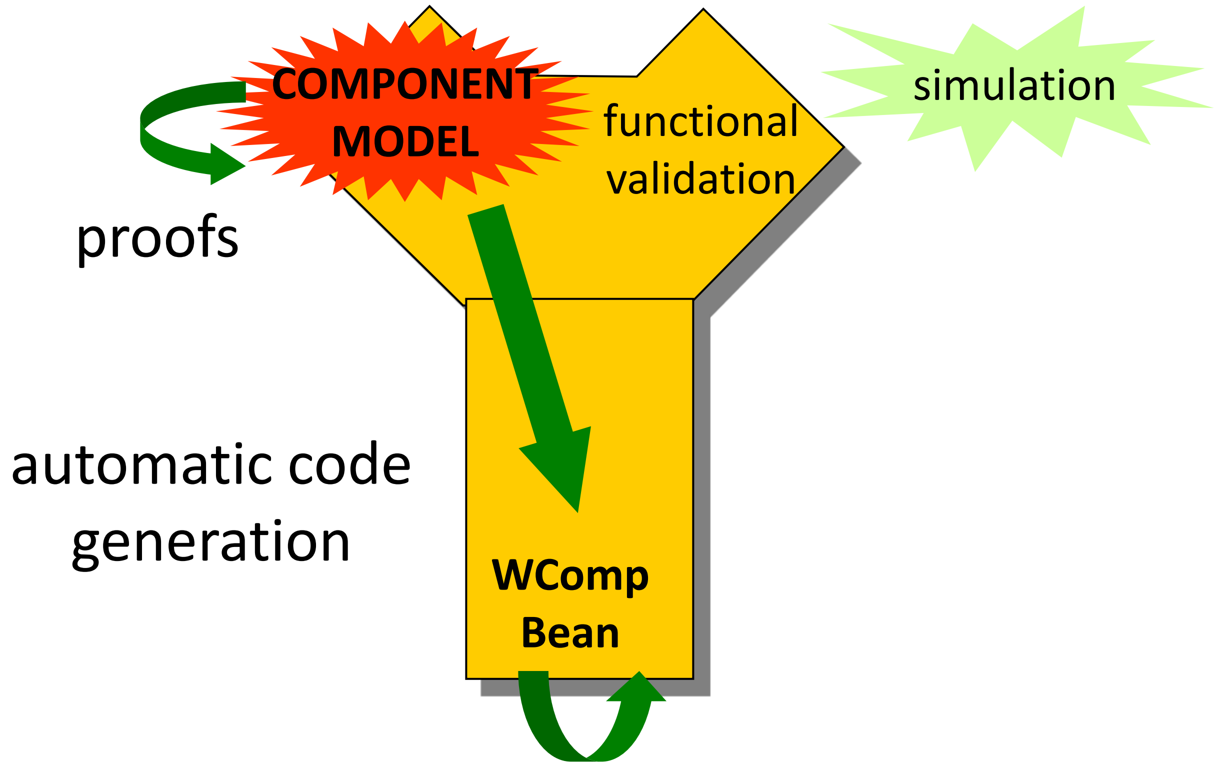




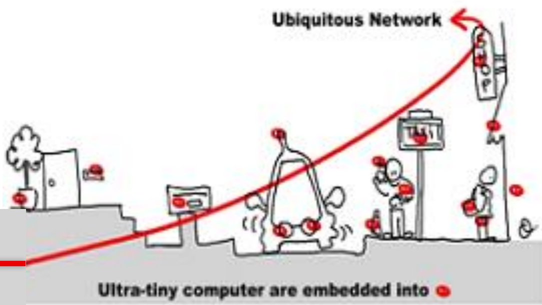
Application to Middleware



In WComp

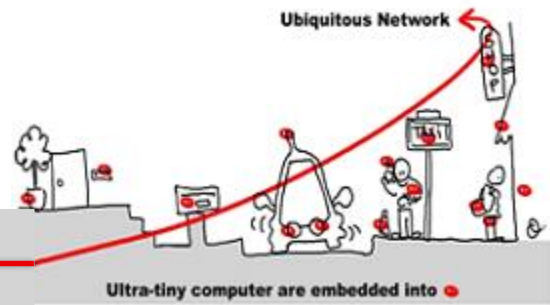


Critical Software Validation



- 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

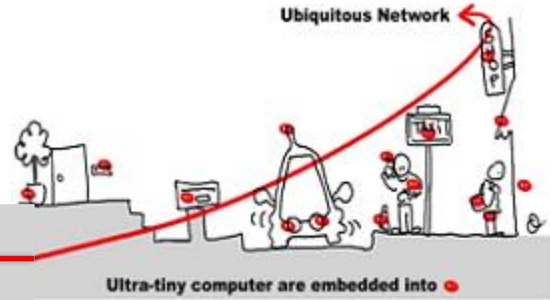


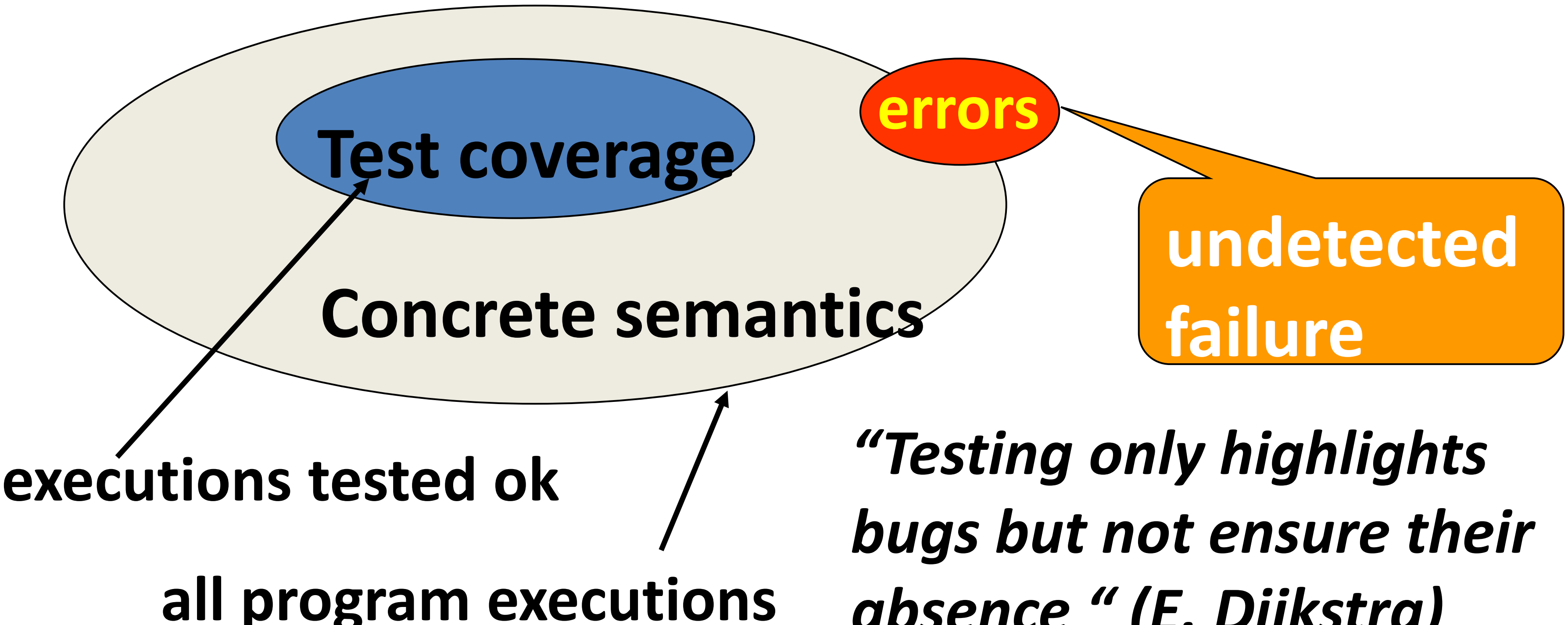
- Testing
  - Run the program on set of inputs and check the results
- Static Analysis
  - Examine the source code to increase confidence that it works as intended
- Formal Verification
  - Argue formally that the application always works as intended

Testing

- Dynamic verification process applied at implementation level.
- Feed the system (or one if its components) with a set of input data values:
  - Input data set not too large to avoid huge time testing procedure.
  - Maximal coverage of different cases required.

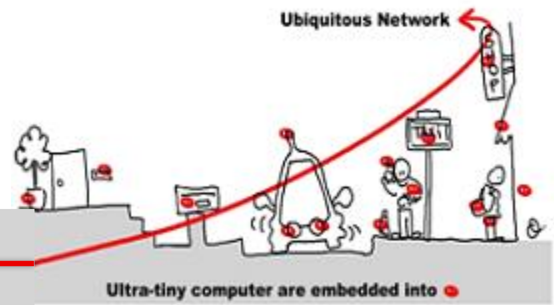
Program Testing





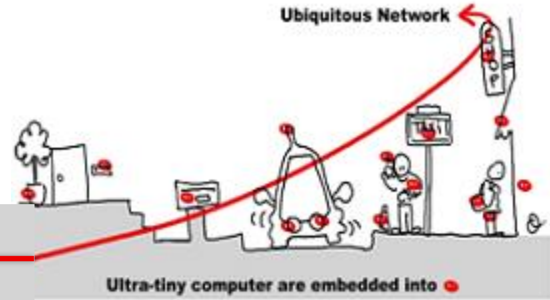
"Testing only highlights bugs but not ensure their absence " (E. Dijkstra)

Static Analysis



- The aim of static analysis is to search for errors without running the program.
- *Abstract interpretation* = replace data of the program by an abstraction in order to be able to compute program properties.
- Abstraction must ensure :
    - $\mathcal{A}$(P) “correct” $\Rightarrow$ P correct
    - But $\mathcal{A}$(P) “incorrect” $\Rightarrow$?

Static Analysis: example



abstraction: integer by intervals

1: x:= 1;
2: while (x < 1000) {
3: x := x+1;
4: }

$x1 = [1,1]$

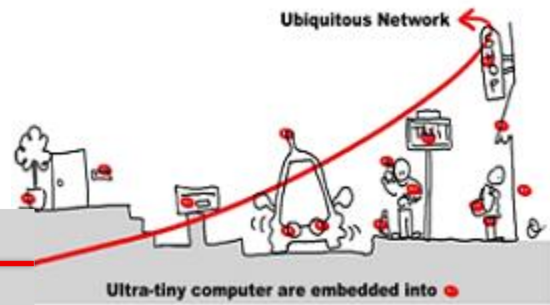
$x2 = x1 \cup x3 \cap [-\infty, 999]$

$x3 = x2 \oplus [1,1]$

$x4 = x1 \cup x3 \cap [1000, \infty]$

Abstract interpretation theory $\Rightarrow$ values are fix point equation solutions.

Formal Verification

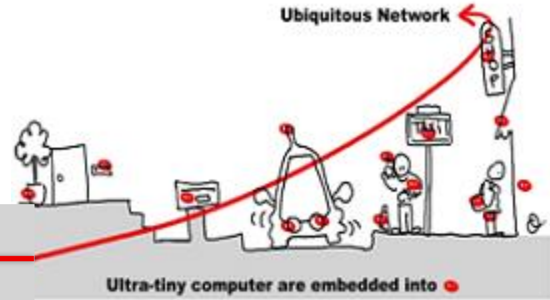


- What about functional validation ?
  - Does the program compute the expected outputs?
  - Respect of time constraints (temporal properties)
  - Intuitive partition of temporal properties:
    - Safety properties: something bad never happens
    - Liveness properties: something good eventually happens

Safety and Liveness Properties

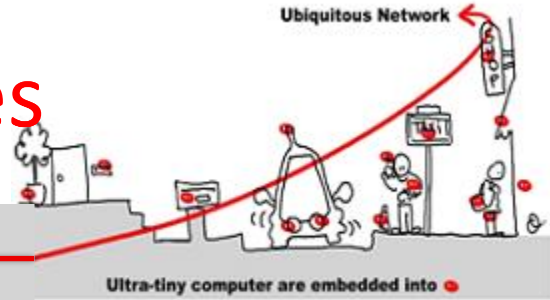
- Example: train timetable
  - Count the difference between marks and seconds
  - Decide when the train is ontime, late, early
    - **ontime** : difference = 0
    - **late** : difference > 3 and it was ontime before or difference > 1 and it was already late before
    - **early** : difference < -3 and it was ontime before or difference < -1 and it was early before

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



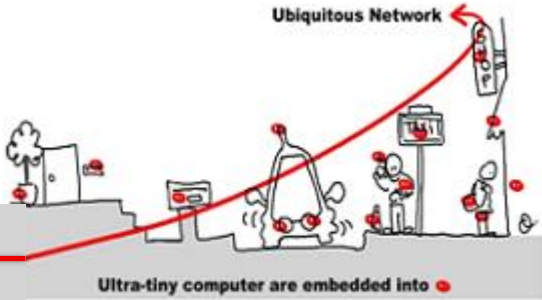
Some properties:

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

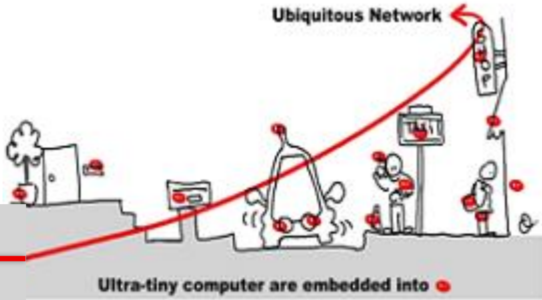
Property 4 : liveness (refer to unbound future)

Outline



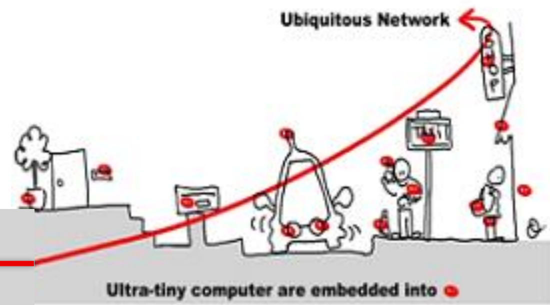
1. Critical system validation
2. Model-checking solution
   1. Model specification
   2. Model-checking techniques
3. Application to component based adaptive middleware
   1. Middleware critical component as synchronous models to allow validation
   2. The Scade solution

Safety and Liveness Properties Checking



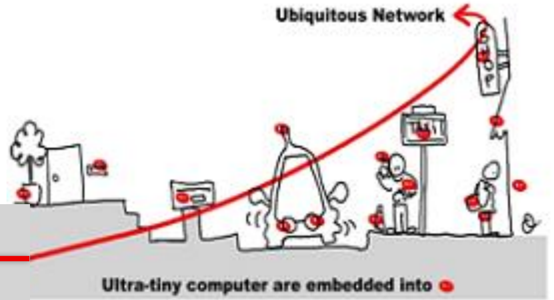
- Use of model checking technique
- Model checking goal: prove safety and liveness properties of a system in analyzing a model of the system.
- Model checking techniques require:
  - model of the system
  - express properties
  - algorithm to check properties againts the model ($\Rightarrow$ decidability)

Model Checking Techniques



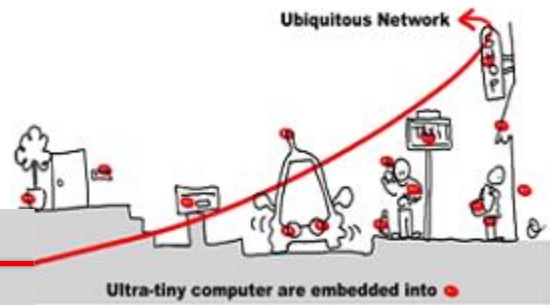
- Model = automata which is the set of program behaviors
- Properties expression = temporal logic:
  - LTL : liveness properties
  - CTL: safety properties
- Algorithm =
  - LTL : algorithm exponential wrt the formula size and linear wrt automata size.
  - CTL: algorithm linear wrt formula size and wrt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

Model Checking Model



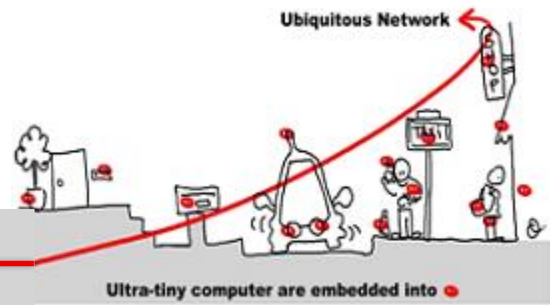
- Model = finite state machine (automata) which is the set of program behaviors
- Kripke structure:
  - non deterministic automata
  - Oriented graph
  - Nodes are program states
  - To each state , a set of atomic (basic) properties is associated

Model Checking Model



- Model = finite state machine (automata) which is the set of program behaviors
- Kripke structure over $\mathbb{AP}$ (set of atomic propositions)
  - A finite set of states ($\mathbb{S}$)
  - A set of initial states $\mathbb{I} \subseteq \mathbb{S}$
  - A transition relation $\mathbb{R} \subseteq \mathbb{S} \times \mathbb{S} \mid \forall s \in \mathbb{S}, \exists s' \in \mathbb{S}$ and $(s,s') \in \mathbb{R}$
  - A labeling function L: $\mathbb{S} \rightarrow \mathbb{AP}$
- How specify such a model ?

Model Specification



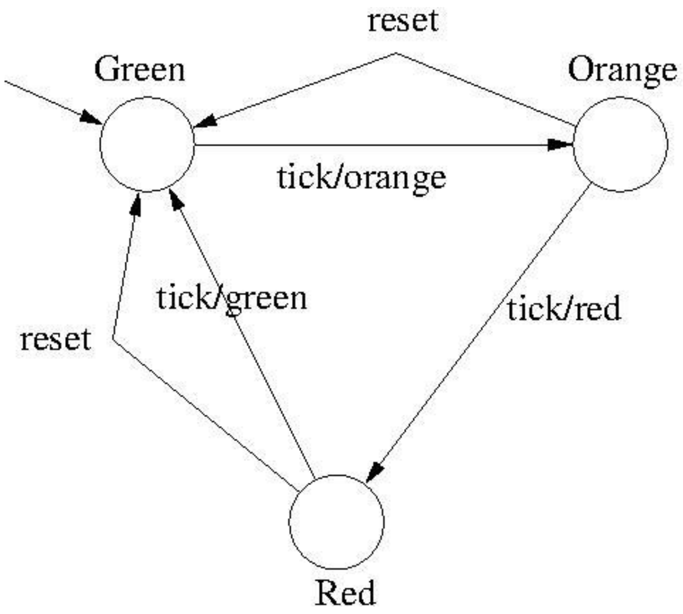
- Model = Mealy automata which is the set of program behaviors (deterministic)
- A Mealy automata is composed of:
  1. A finite set of states ($\mathbb{Q}$)
  2. A finite alphabet of triggers ($\mathbb{T}$)
  3. A finite alphabet of actions ($\mathbb{A}$)
  4. An initial state ($q^{init} \in \mathbb{Q}$)
  5. A transition function **δ:** $\mathbb{Q} \times \mathbb{T} \rightarrow \mathbb{Q}$
  6. An output function **λ :** $\mathbb{Q} \times \mathbb{T} \rightarrow 2^{\mathbb{A}}$

Notation: a transition is denoted $q_1 \xrightarrow{t/a} q_2$

Model Specification

- Model = Mealy automata which is the set of program behaviors

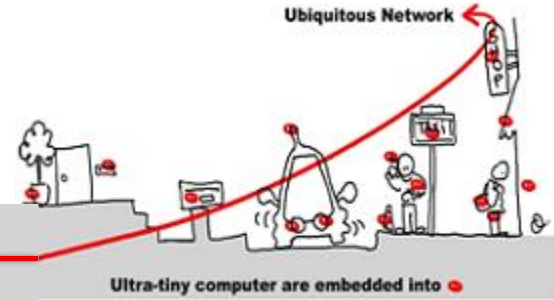
Example: **Traffic Light**



trigger: tick, reset

action:green,orange,red

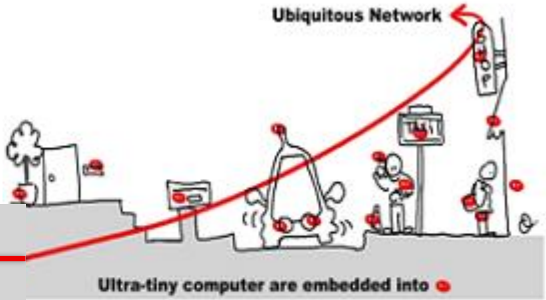
Model Specification



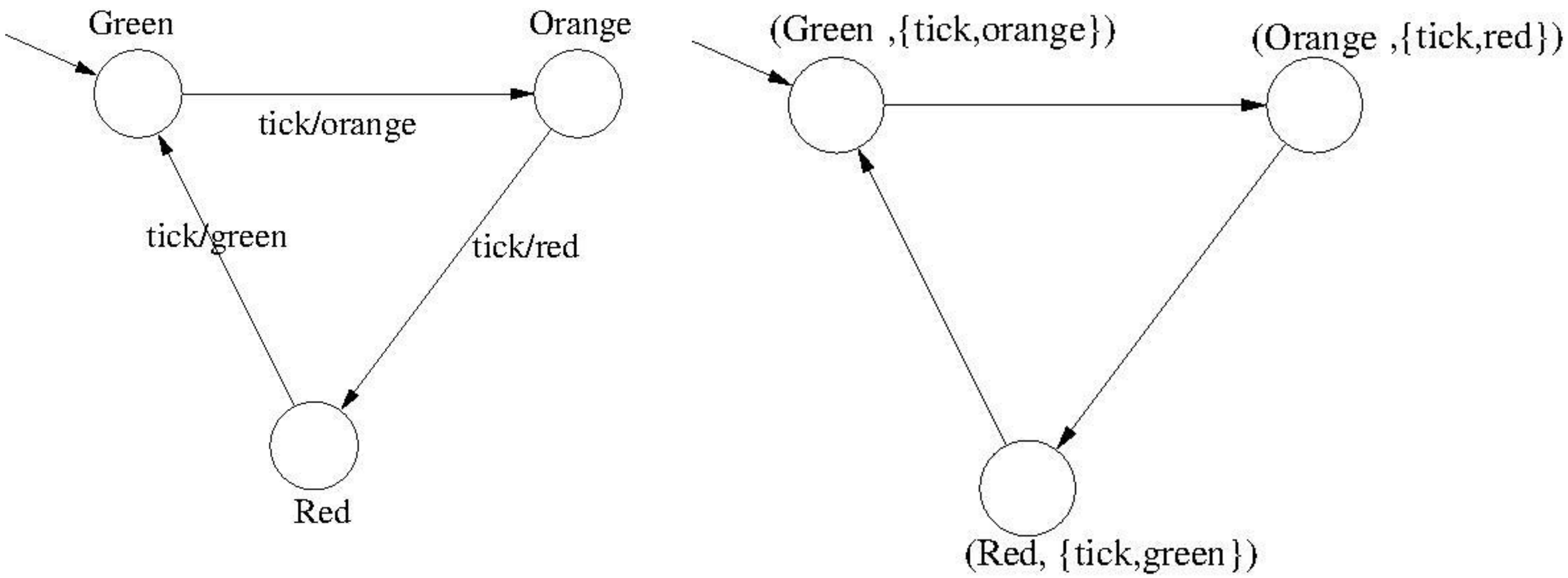
Mealy automata = Kripke structure

- $\mathbb{AP} = \mathbb{T} \cup \mathbb{A}$
- $\mathbb{S} \subseteq \mathbb{Q} \times 2^{\mathbb{AP}}$; $\{(q, v) \mid \exists\, q \xrightarrow{t/a} q'$ and $v = \{t\} \cup a$ or $v = \varnothing \}$
- $I = \{q^{init}\} \times 2^{\mathbb{AP}} \cap \mathbb{S}$
- $\mathbb{R} = \{(q,v), (q',v') \mid \exists\, q \xrightarrow{t/a} q'$ and $v = \{t\} \cup a$ and $(q',v') \in \mathbb{S}$
- $L(q,v) = v$

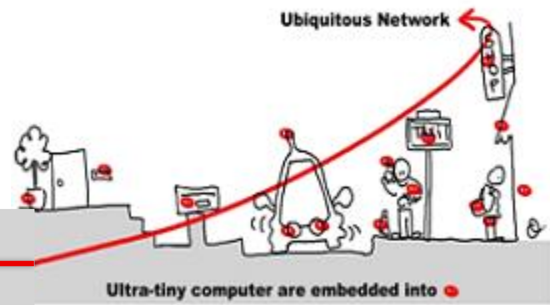
Model Specification



Mealy automata = Kripke structure

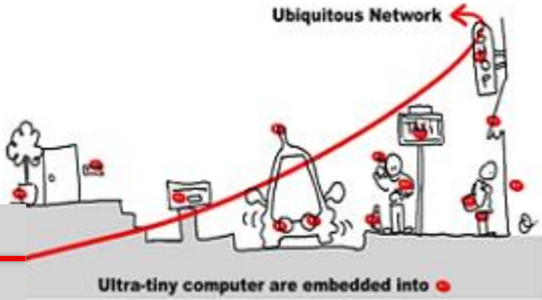


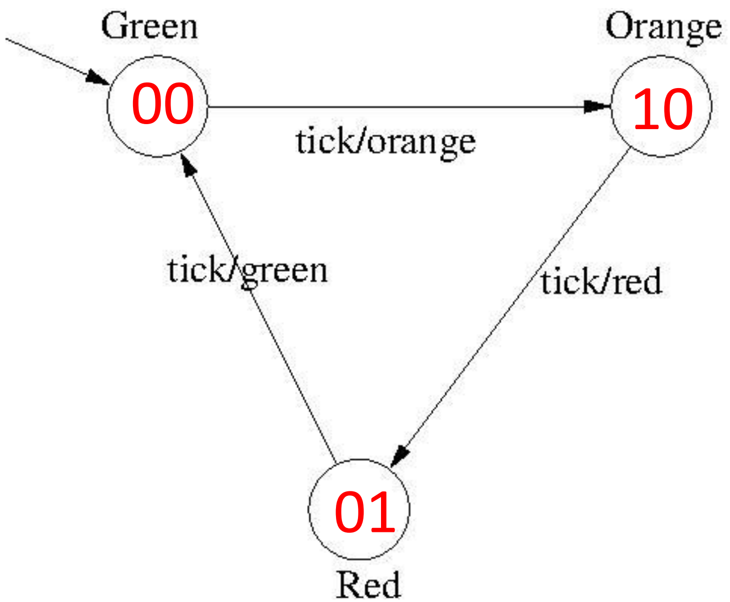
Implicit vs Explicit Mealy Machine



- Mealy automata is an explicit Mealy Machine
- Implicit representation as Boolean equation system with registers.
- $M = \langle Q, q^{init}, T, A, \delta, \lambda \rangle$ $\xi(M) = \langle T \cup A, R, D \rangle$:
  - R: Boolean registers
  - D : definitions or equations of the form x=e
    - $X \in A \cup R^{+}$ and e Boolean expr built from $T \cup R$
    - States are encoded as register combination: $\{q_1,q_2,q_3\}$ is encoded with 2 registers r_1, r_2 and a possible encoding is : 00, 01,10
    - For each state, δ and λ encoded with truth tables

Implicit vs Explicit Mealy Machine





Registers: X0, X1

Initial values: X0 = 0 and X1 = 0

X0next = not X0 and not X1;

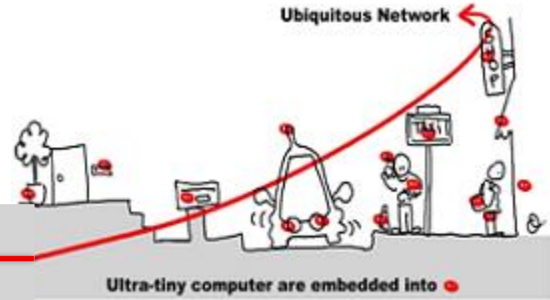
X1next = X0;

orange = not X0 and not X1;

green = not X0 and X1;

red = X0 and not X1;

Model Checking

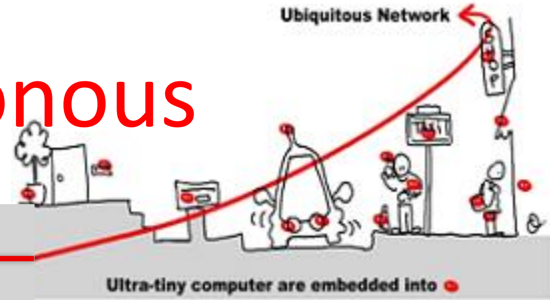


How design Mealy automata ?

Use synchronous languages to specify critical systems.

Synchronous programs = Mealy automata

Model Specification with Synchronous Languages

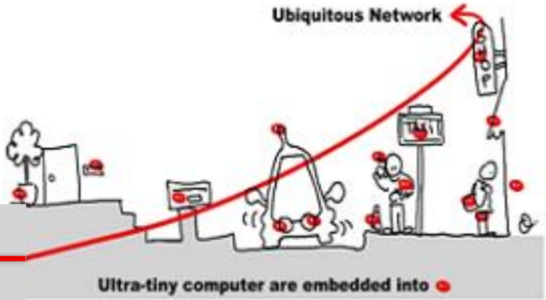


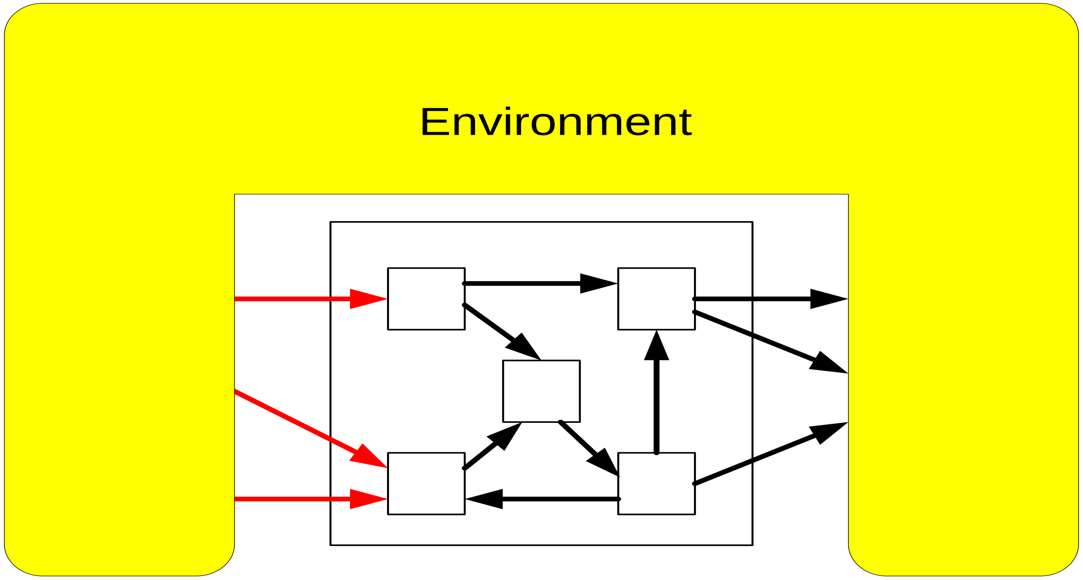
1. Synchronous languages have a **simple formal model** (a finite state machine) making formal reasoning tractable.
2. Synchronous languages support **concurrency** and offer an implicit or explicit means to express parallelism.
3. Synchronous languages are devoted to design **reactive systems**.

Determinism & Reactivity

- Synchronous languages are deterministic and reactive
- Determinism:
  - The same input sequence always yields the same output sequence
- Reactivity:
  - The program must react(*) to any stimulus
  - Implies absence of deadlock
    - (*) *Does not necessary generate outputs, the reaction may change internal state only.*

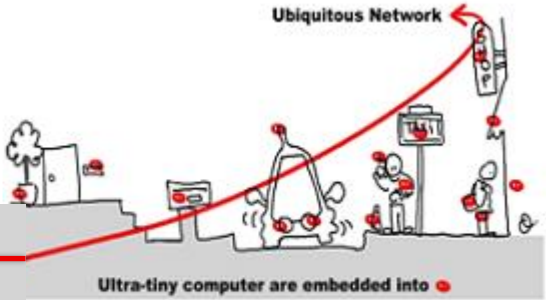
Synchronous Reactive Programs (1)

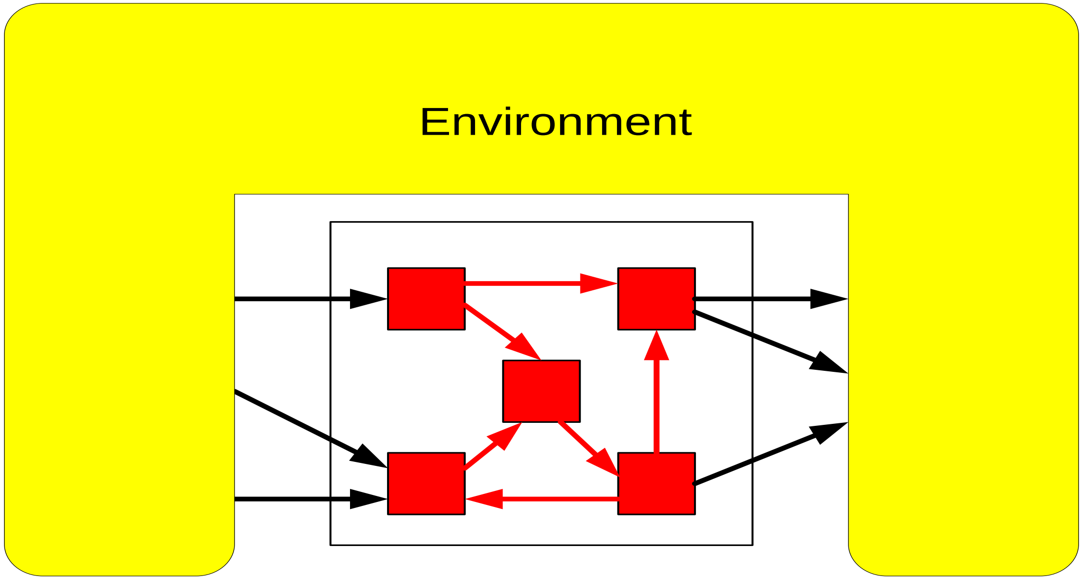




Read

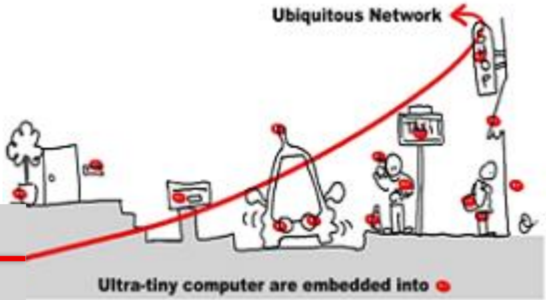
Synchronous Reactive Programs (1)

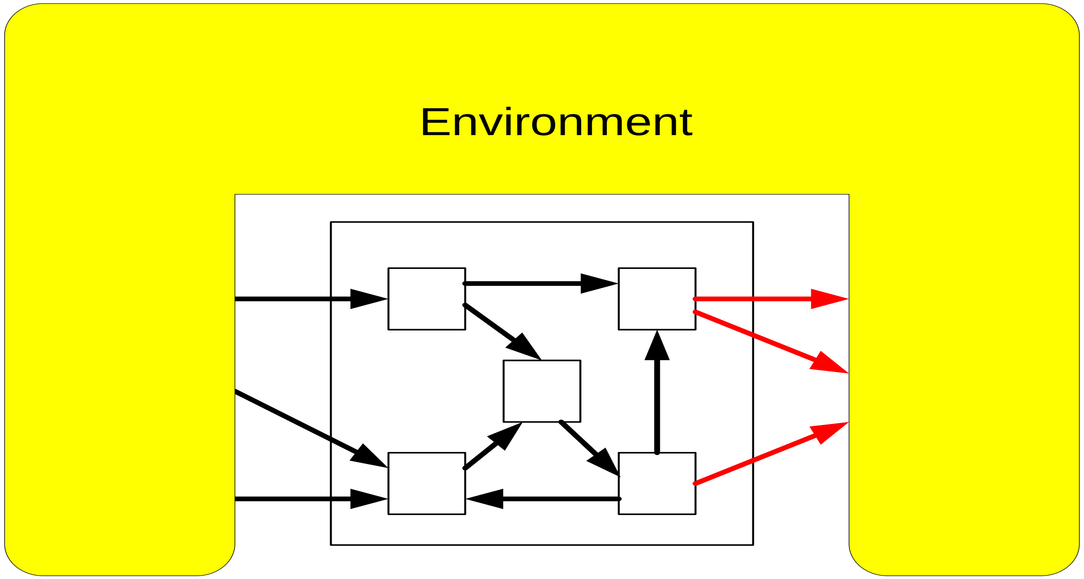




Computations

Synchronous Reactive Programs (1)

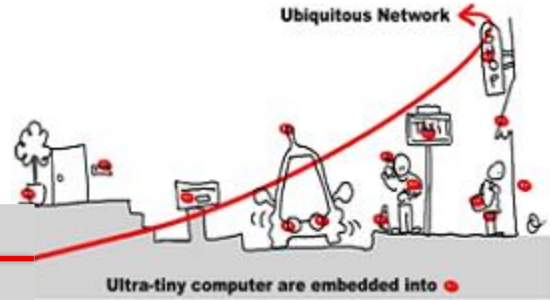




Write

Atomic execution: read, compute, write

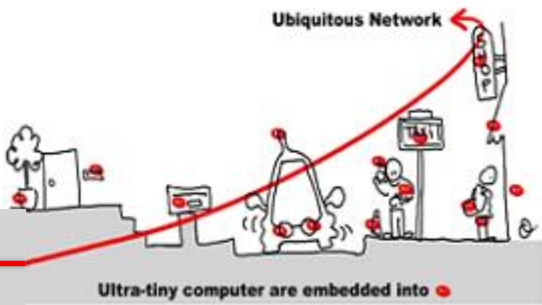
Synchronous Hypothesis



- Synchronous languages work on a logical time.
- The time is
  - Discrete
  - Total ordering of instants.

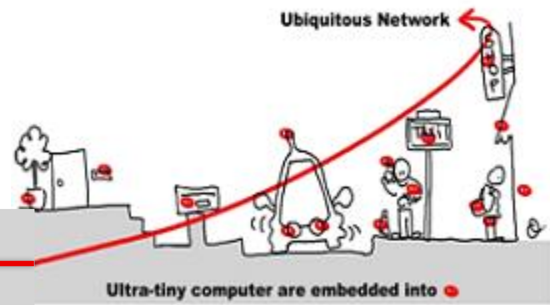
  Use N as time base
- A reaction executes in one instant.
- Actions that compose the reaction may be partially ordered.

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.

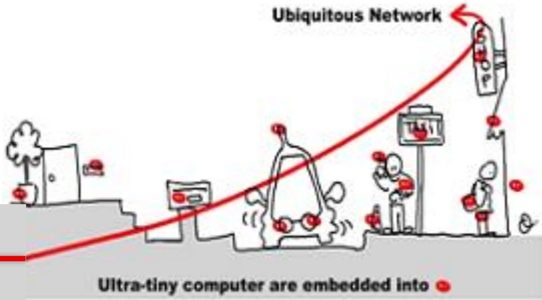
Reactive ?



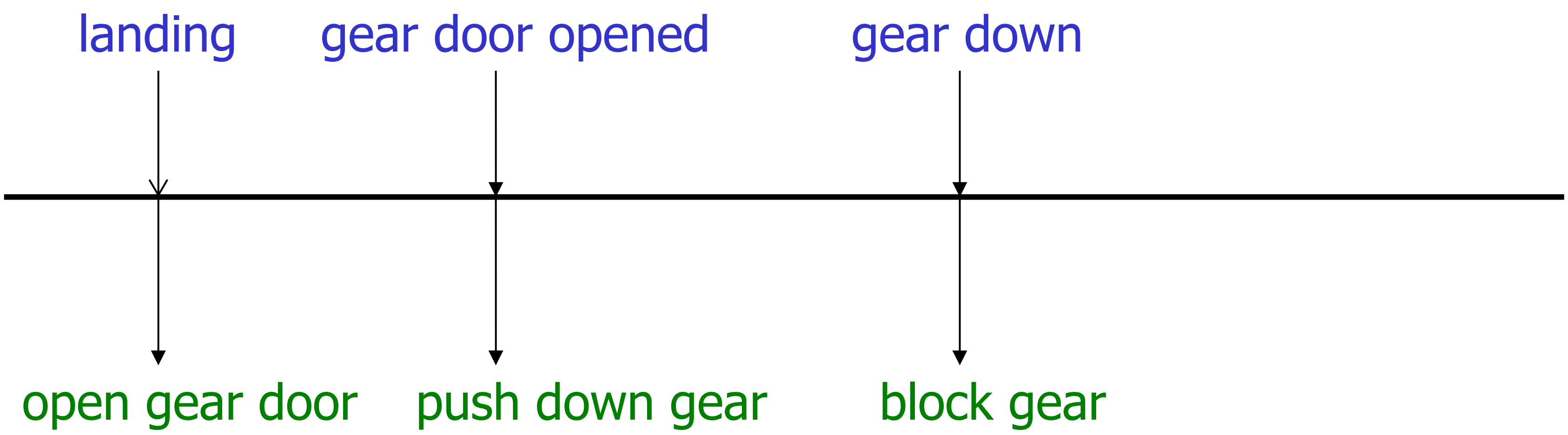
- Different ways to "react" to the environment:
  - Event driven system:
    - Receive events
    - Answer by sending events
  - Data flow system:
    - Receive data continuously
    - Answer by treating data continuously also

Some systems have components of both kinds

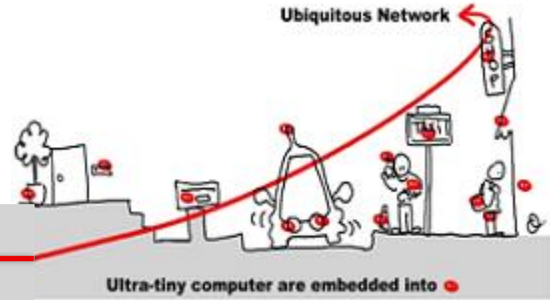
Event Driven Reactive System



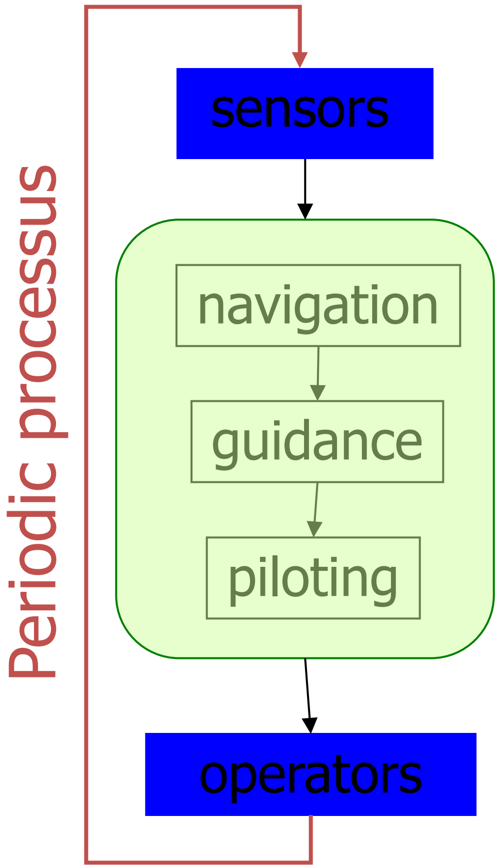
Langing gear management



Data Flow Reactive System (Example)

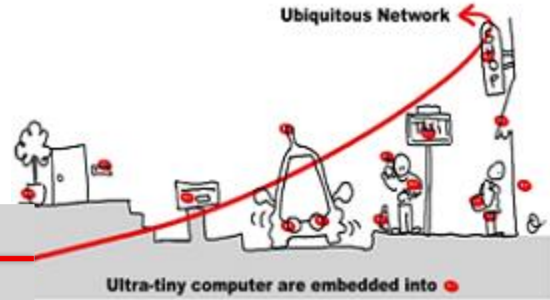


Control/Command vehicle



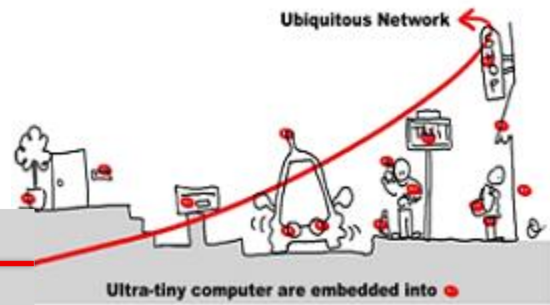
- get measures
- where am I ?
- where go I ?
- command computation
- command to operators

Imperative and Declarative languages



- Different ways to express synchronous programs:
  1. Imperative languages rely on implicitly or explicitly **finite state machines**, well suited to design event driven reactive system
  2. Declarative languages rely on operator networks computing **data flows**, well suited to design data flow reactive system

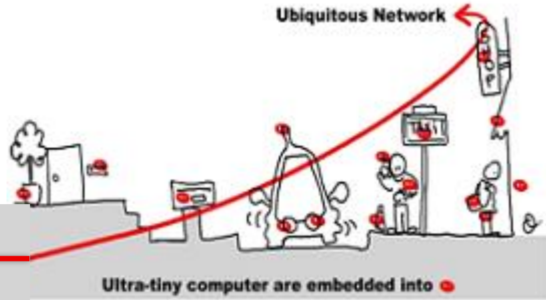
Imperative Language



Event driven applications can be designed with an imperative language (as **Esterel**)

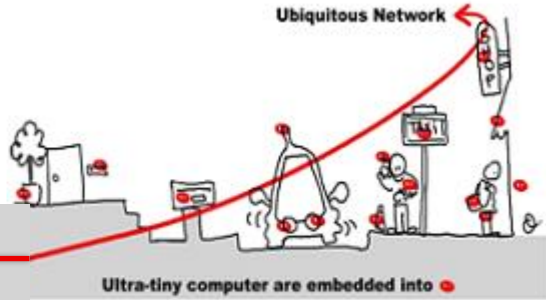
1. Listen input and output events
2. Specific operators to deal with the logical time (await)
3. Test of presence or absence of signals (present)
4. Synchronous parallelism (||)
5. Emit to change the environment (emit S)
6. Usual operators (loop, abort when)

Esterel program example




```
module RUNNER:
Constant  NumberOfLaps : integer;
input Morning, Second, Meter, Step, Lap;
output Walk, Jump, Run;

      Program body   (next slide)

end module
```

Esterel program example

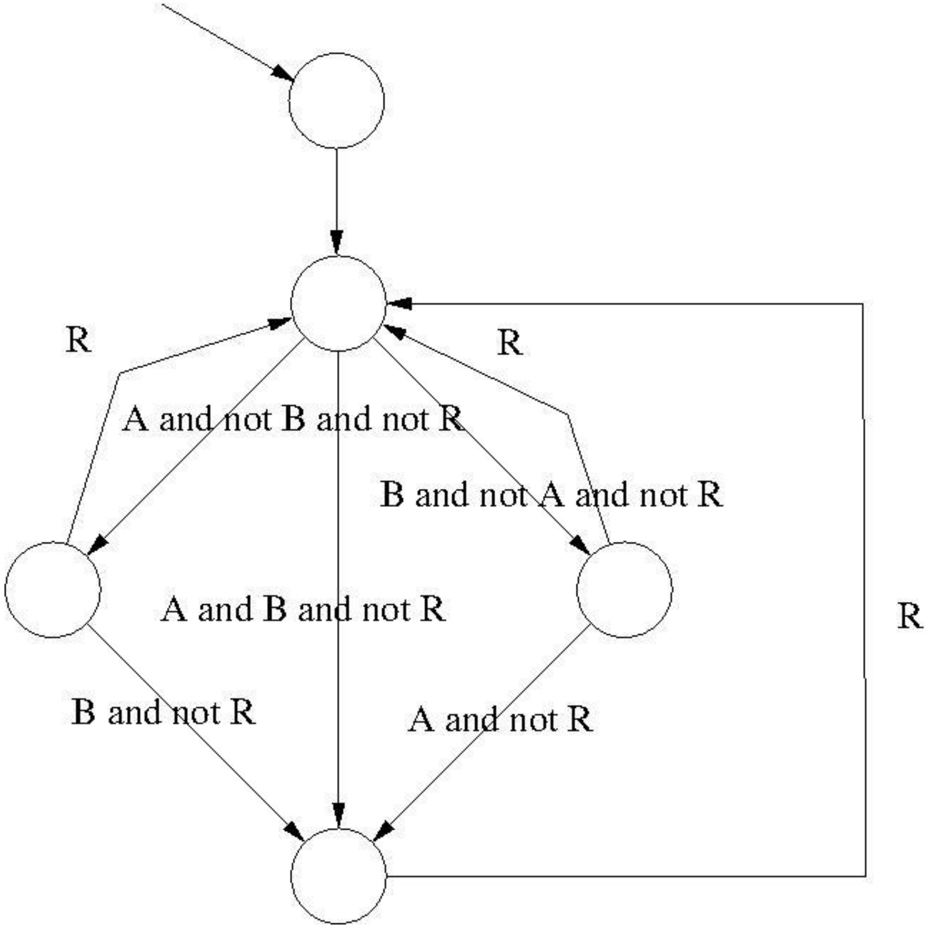



```
every Morning do
  repeat NumberOfLaps times
    abort
      abort  sustain Walk when 100 Meter;
      abort
         every Step do emit Jump end every
      when 15 Second;
      sustain Run
   when Lap
  end repeat
end every
```

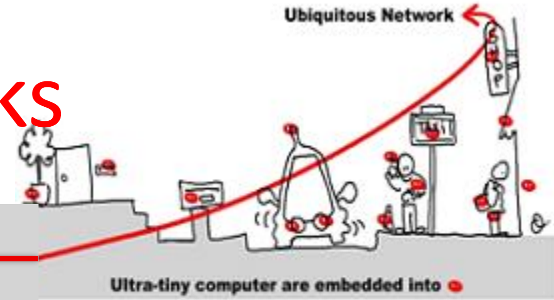
sequence

Esterel program = Mealy Machine

```
module ABRO:
  input A, B, R;
  output O;
  loop
    [ await A || await B ];
    emit O;
  each R
end module
```

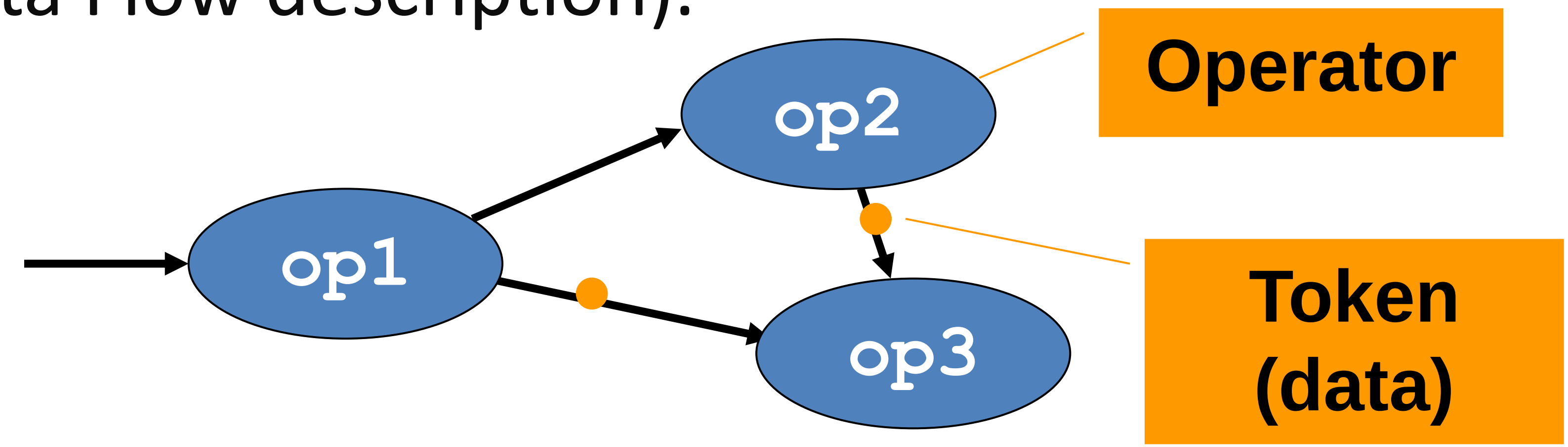


Data flow = Operator Networks

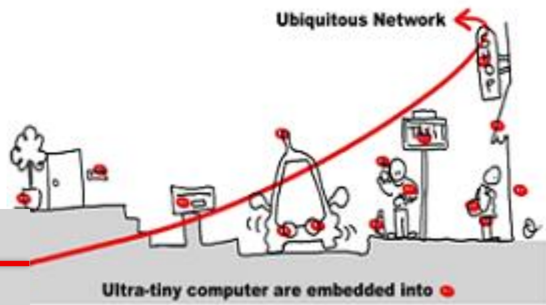


Data flow programs can be interpreted as networks of operators.

Data « flow » to operators where they are consumed. Then, the operators generate new data. (Data Flow description).



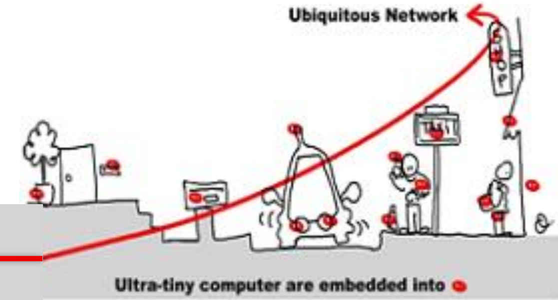
Flows, Clocks

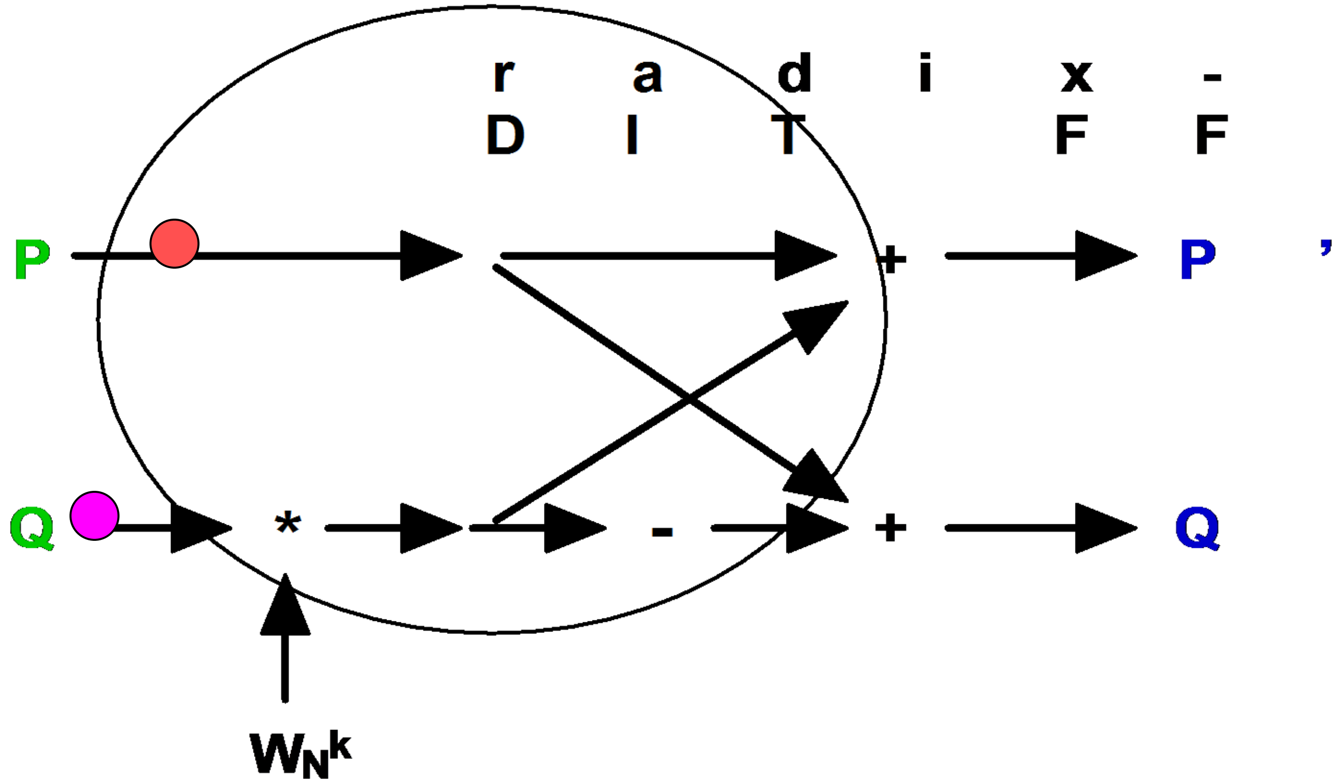


- A flow is a pair made of
  - A possibly infinite sequence of values of a given type
  - A clock representing a sequence of instants

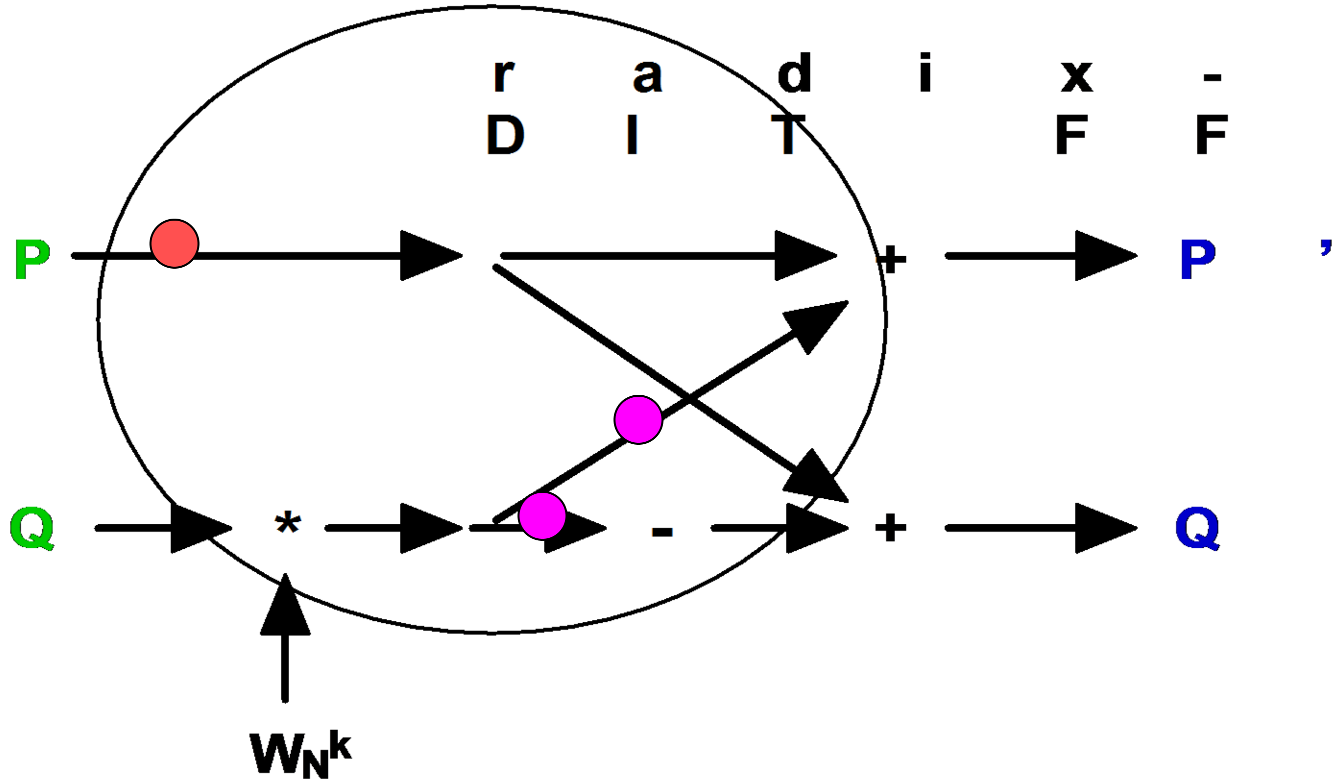
$$\mathbf{X{:}T} \qquad (x_1, x_2, \ldots, x_n, \ldots)$$

An example of Data Flow

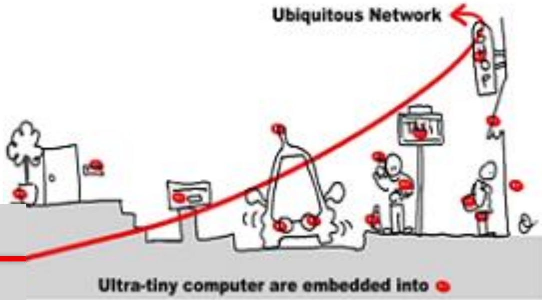




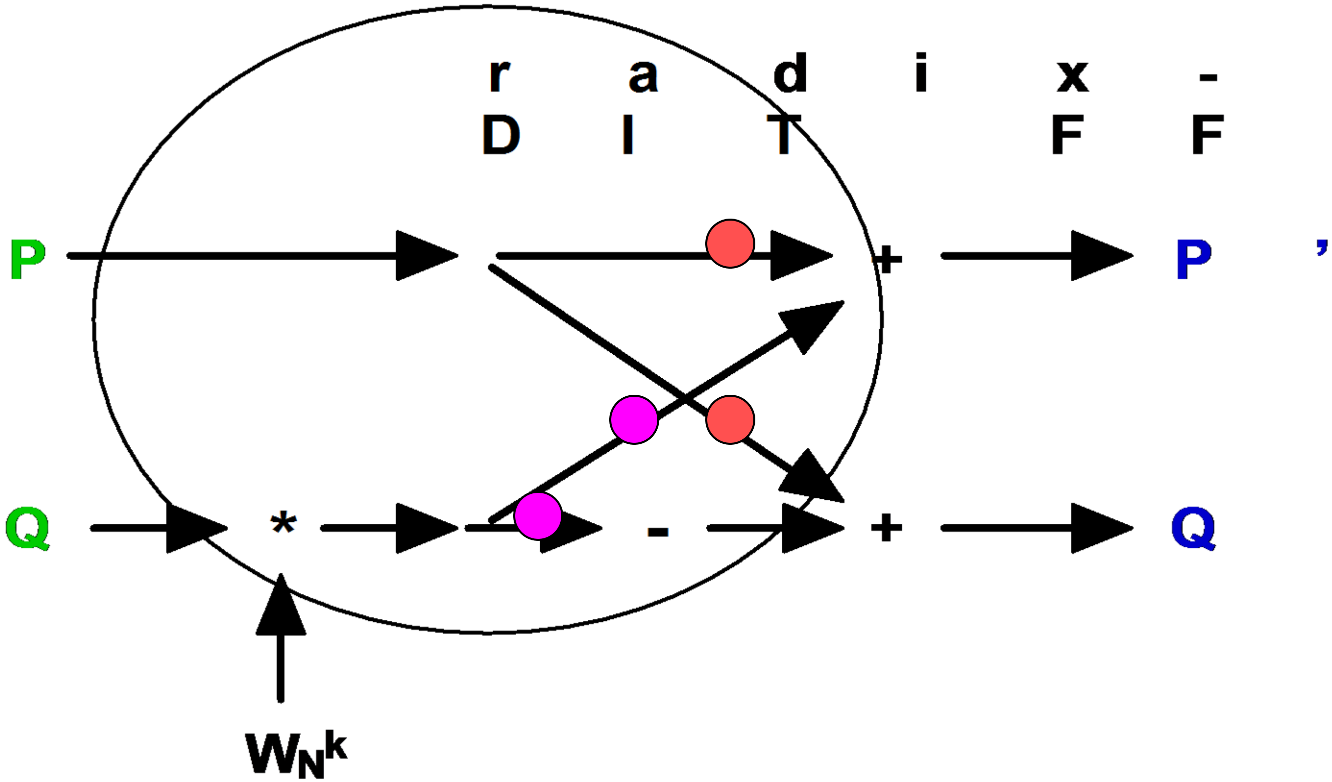
Data Flow



Data Flow

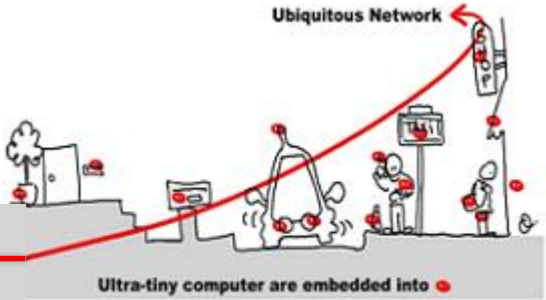


Ubiquitous Network
Ultra-tiny computer are embedded into

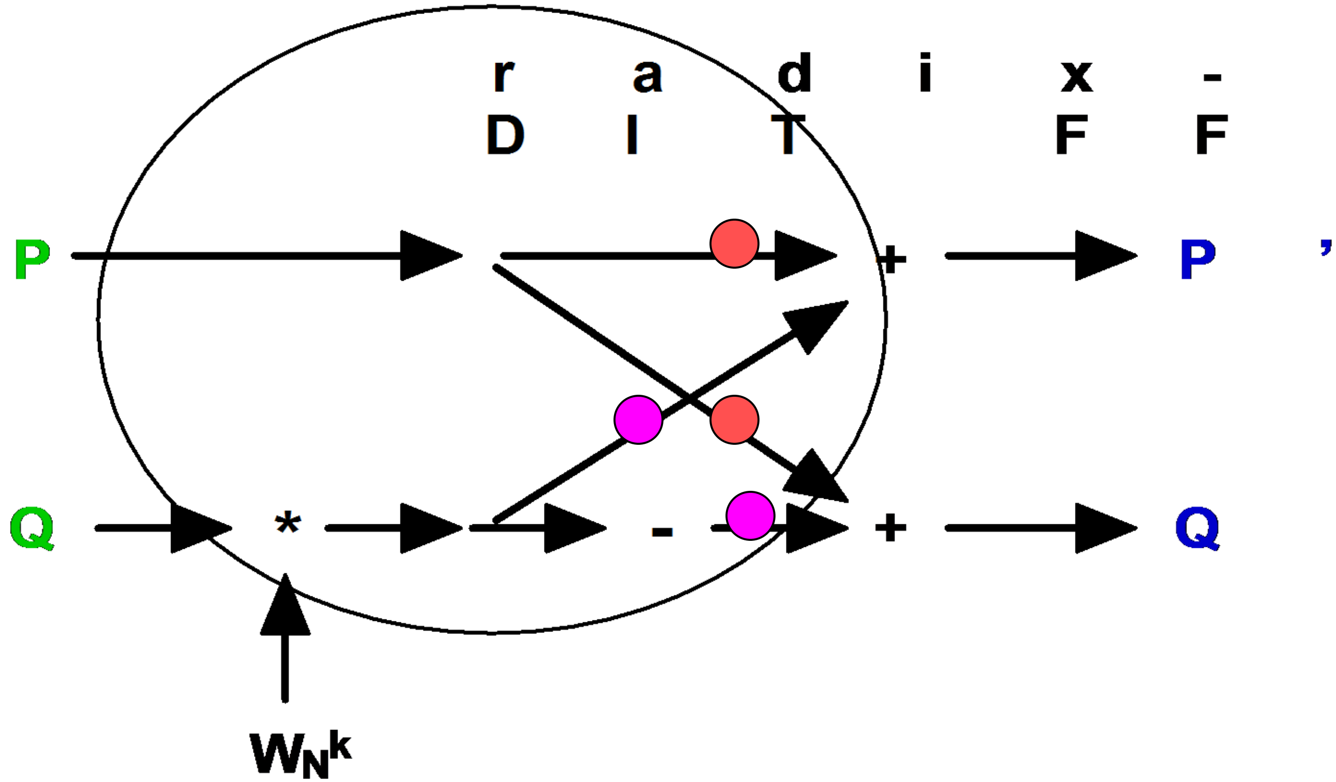


r a d i x -
D I T F F
P
Q
*
-
+
+
P ’
Q
W_N^k

Data Flow

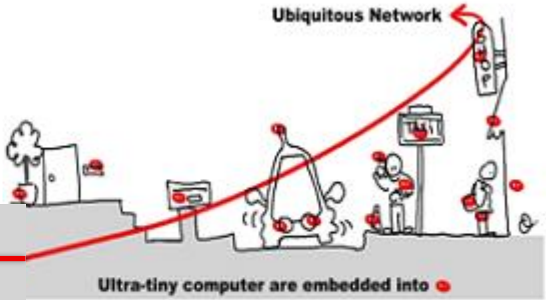


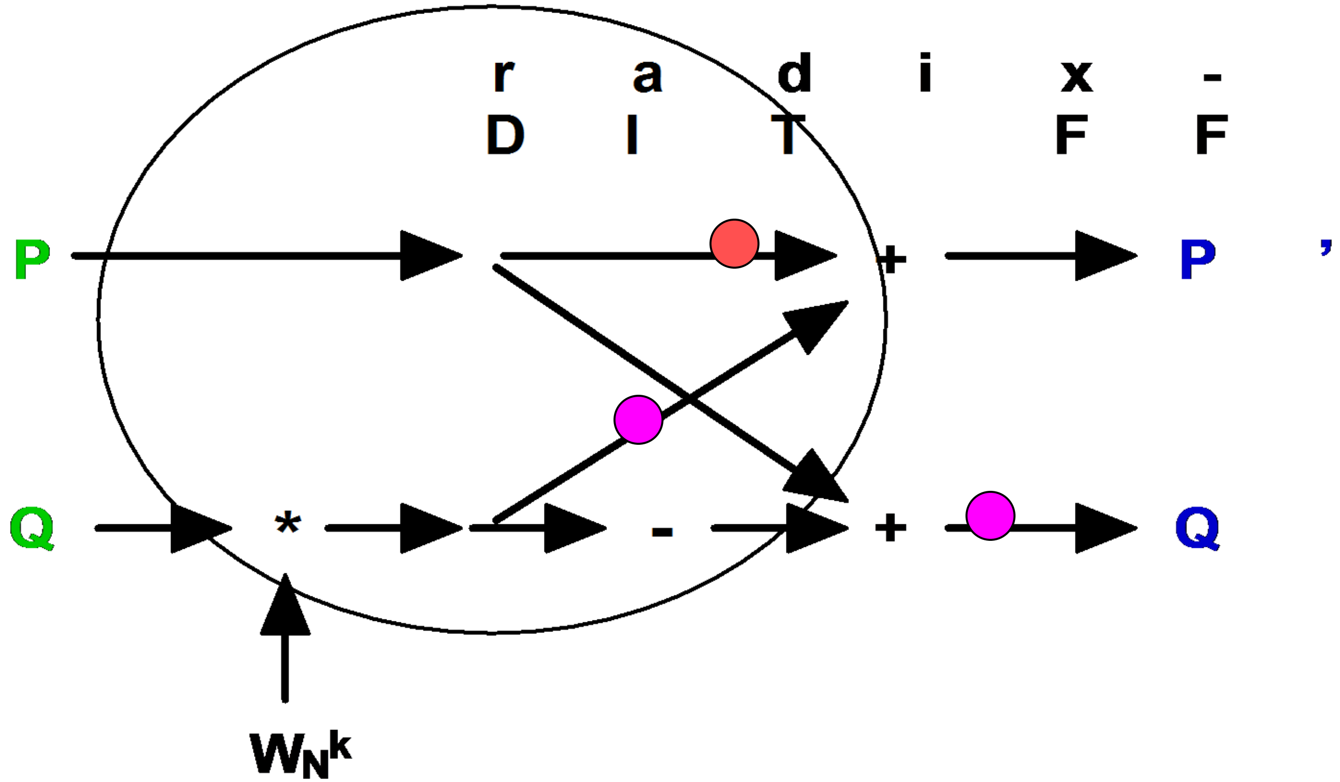
Ubiquitous Network
Ultra-tiny computer are embedded into



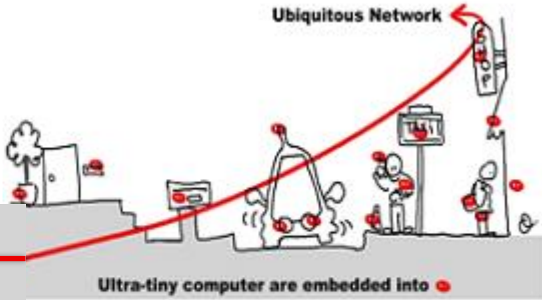
r a d i x -
D I T F F
P
Q
*
W_N^k
-
+
+
P '
Q

Data Flow

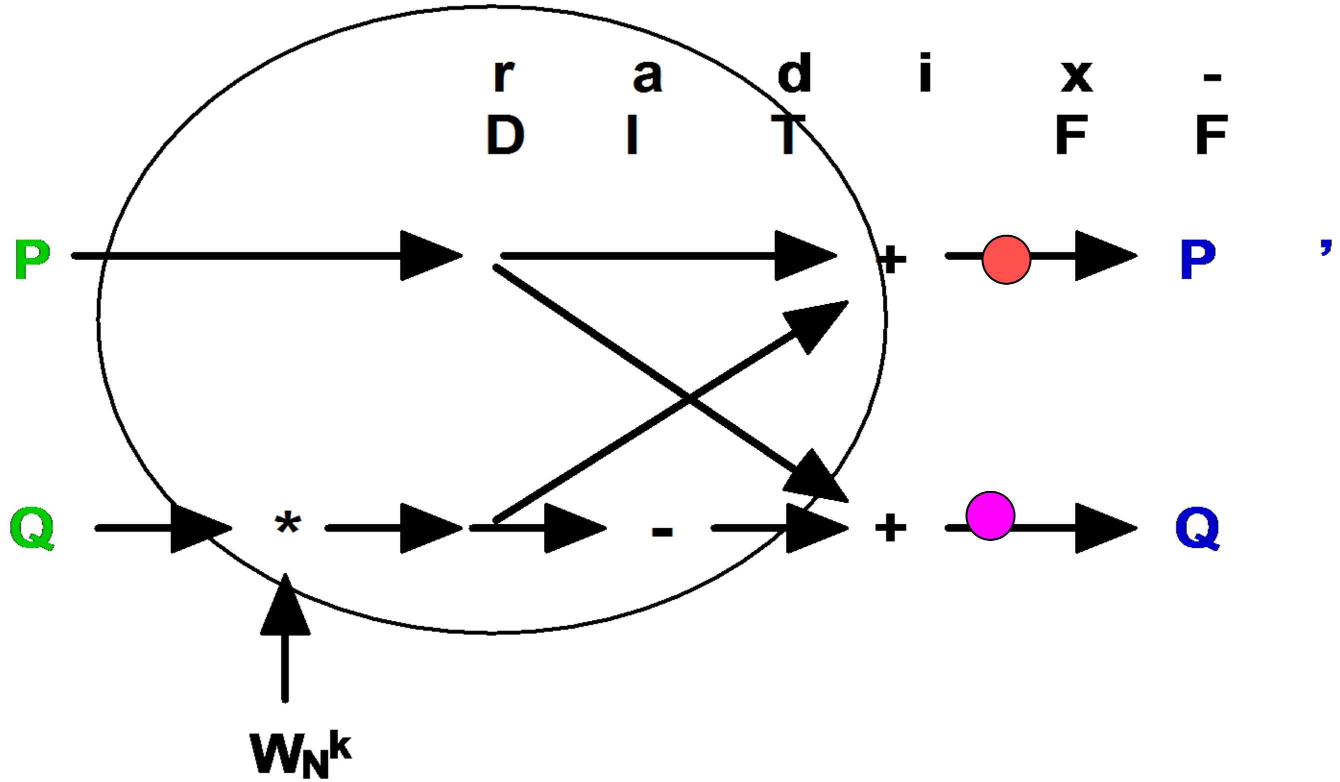

Ubiquitous Network
Ultra-tiny computer are embedded into


r a d i x -
D I T F F
P
Q
*
-
+
+
P ’
Q
WNk

Data Flow

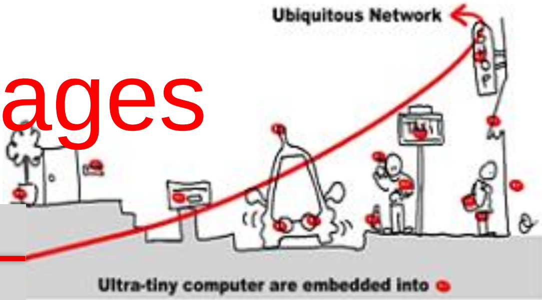


Ubiquitous Network
Ultra-tiny computer are embedded into



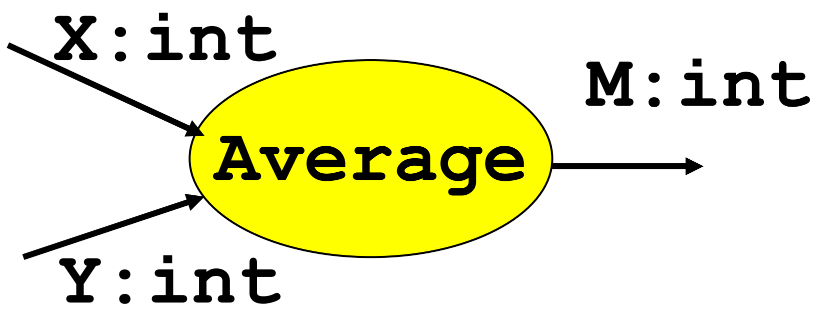
r a d i x -
D I T F F
P
Q
*
-
+
+
P '
Q
W_N^k

Data Flow Synchronous Languages



1. Data flow programs compute output flows from input flows using:
   1. Variables (= flows)
   2. Equation: x = E means $\forall k \;\; x_k = E_k$
   3. Assertion: Boolean expression that should be always true.
2. Data flow programs define new data flow operators.

Data Flow Synchronous Languages



operator Average (X,Y:int) returns (M:int)

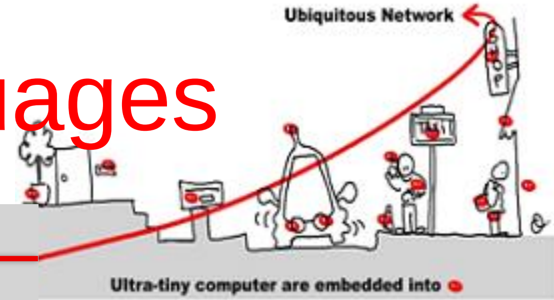
$$M = (X + Y)/2$$

$X = (X_1, X_2, \ldots., X_n, \ldots\ldots.)$

$Y = (Y_1, Y_2, \ldots.., Y_n, \ldots\ldots..)$

$M = ((X_1+Y_1)/2, (X_2+Y_2)/2, \ldots\ldots, (X_n+Y_n)/2, \ldots.)$

Data Flow Synchronous Languages



Memorizing to take the past into account:

1. pre (**previous**):

   $X = (x_1, x_2, \ldots, x_n, \ldots\ldots)$:

   pre(X) = $(nil, x_1, x_2, \ldots, x_n, \ldots\ldots)$

   nil undefined value denoting uninitialized memory

2. → (**initialize**):

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

   $X \rightarrow Y = (x_1, y_2, \ldots, y_n, \ldots\ldots)$

Sequential examples

n= 0 → pre(n) + 1

operator MinMax (x:int) returns (min,max:int):

min = x→ if (x < pre(min) then x else pre(min)

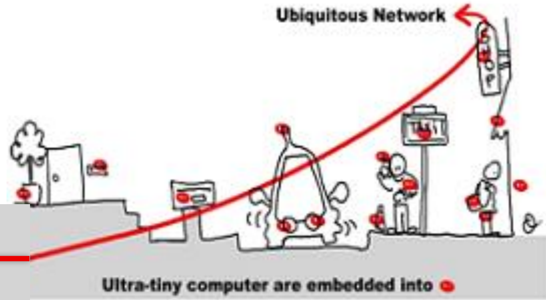
max = x → if (x > pre(max) then x else pre(max)

x= (3, 4, 5, 2, 7, ….)

min = (3, 3, 3, 2, 2,…)

max = (3, 4, 5, 5, 7,…)

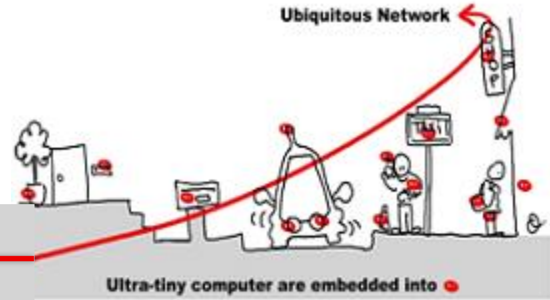
Sequential examples




```
operator CT (init:int) returns (c:int):
        c = init → pre(c) + 2
```

```
operator  DoubleCall (even:bool) returns (n:int)
    n= if (even) then CT(0) else CT(1)
DoubleCall (ff,ff,tt,tt,ff,ff,tt,tt,ff) = ?
```

Sequential examples



operator CT (init:int) returns (c:int):

c = init → pre(c) + 2

CT(0) = (0,2,4,6,8,10,12,14,16,18,….)

CT(1) = (1,3,5,7,9,11,13,15,17,19,….)

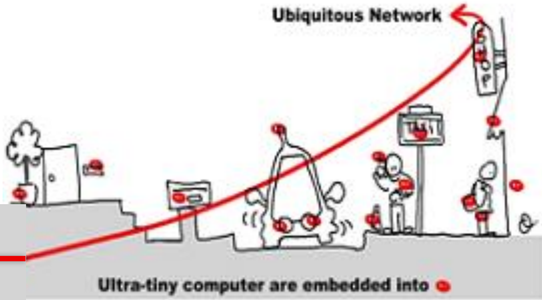
operator DoubleCall (even:bool) returns (n:int)

n= if (even) then CT(0) else CT(1)

DoubleCall (ff,ff,tt,tt,ff,ff,tt,tt,ff) = ?

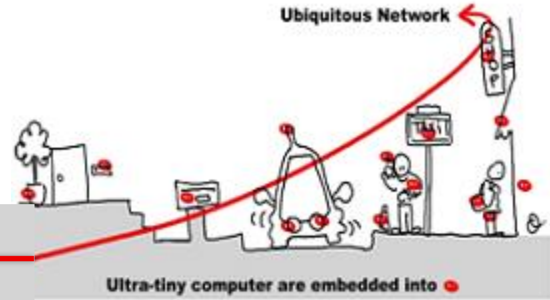
(1,3,4,6,9,11,12,14,17)

Modulo Counter



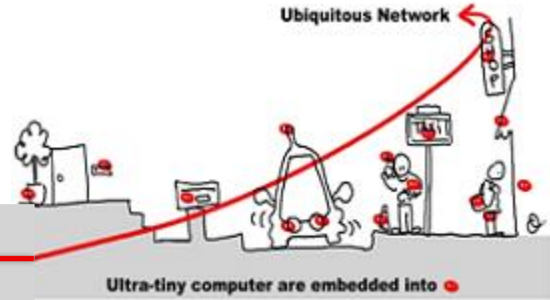

```
operator MCounter (incr:bool; modulo : int)
                                returns (cpt:int);
   var count : int;

     count = 0 -> if incr pre (cpt) + 1
                          else pre (cpt);
     cpt =  count mod modulo;
```

Modulo Counter Clock




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

Modulo Counter Clock



MCounterClock(true,3):

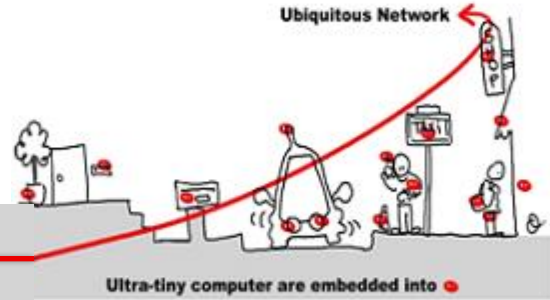
count: 0 1 2 3 1 2 3……

cpt = 0 1 2 0 1 2 0…….

modulo_clock = ff ff ff tt ff ff tt ….

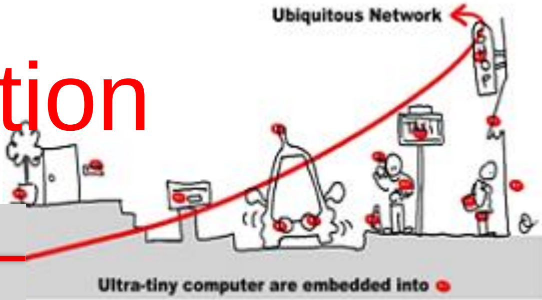
```
var count : int;
    count = 0 -> if incr pre (cpt) + 1
                 else pre (cpt);
    cpt =  count mod modulo;
    modulo_clock = count != cpt;
```

Timer




```
operator Timer returns (hour, minute, second:int);
var hour_clock, minute_clock, day_clock : bool;

(second, minute_clock) = MCounterClock(true, 60);
(minute, hour_clock) = MCounterClock(minute_clock,60);
(hour, dummy_clock) = MCounterClock(hour_clock, 24);
```

Data Flow Programs Compilation

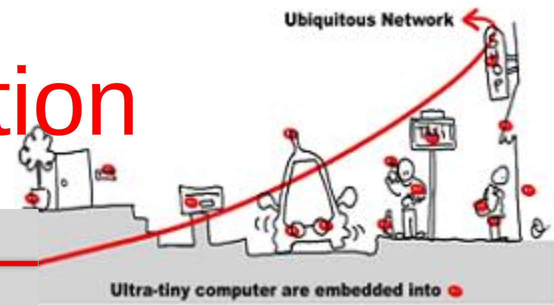


Data flow programs are compiled into automata

Data Flow Program Compilation

```
operator WD (set, reset, deadline:bool)
                        returns (alarm:bool);
var is_set:bool;
  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.
```

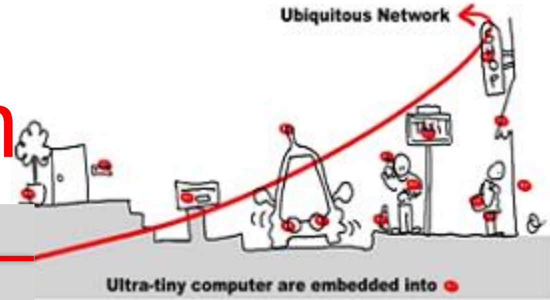
Data Flow Program Compilation



First, the program is translated into pseudo code:

```
if _init then // first instant (or reaction)
  is_set := false; alarm := false;
   _init := false;
else   // following reactions
  if set then is_set := true
  else
    if reset then is_set := false;
    endif
  endif
  alarm := is_set and deadline;
endif
```

Data Flow Program Compilation



Choose state variables : _init and variables which have pre.

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

Data Flow Program Compilation

initial

S0: alarm := false;

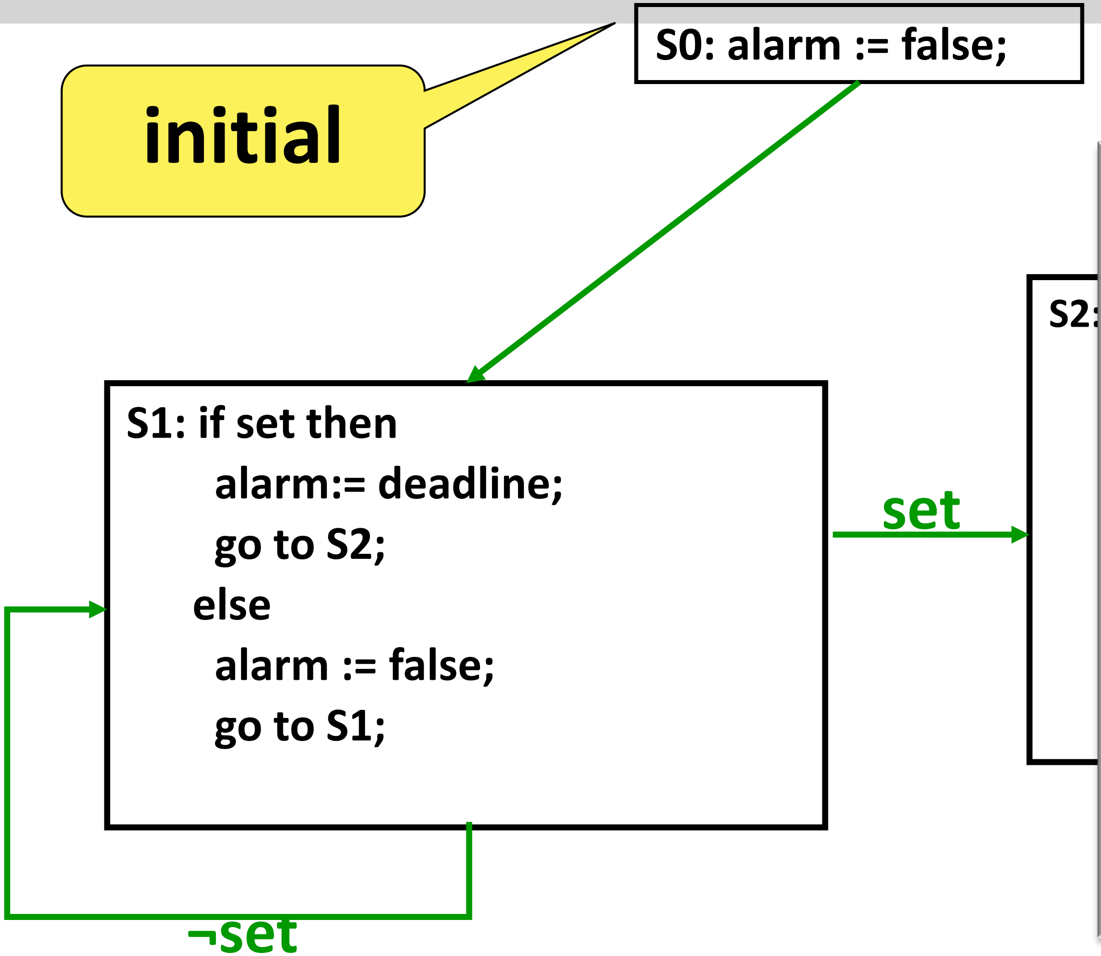
S1:

_init := false

pre(is_set) := false

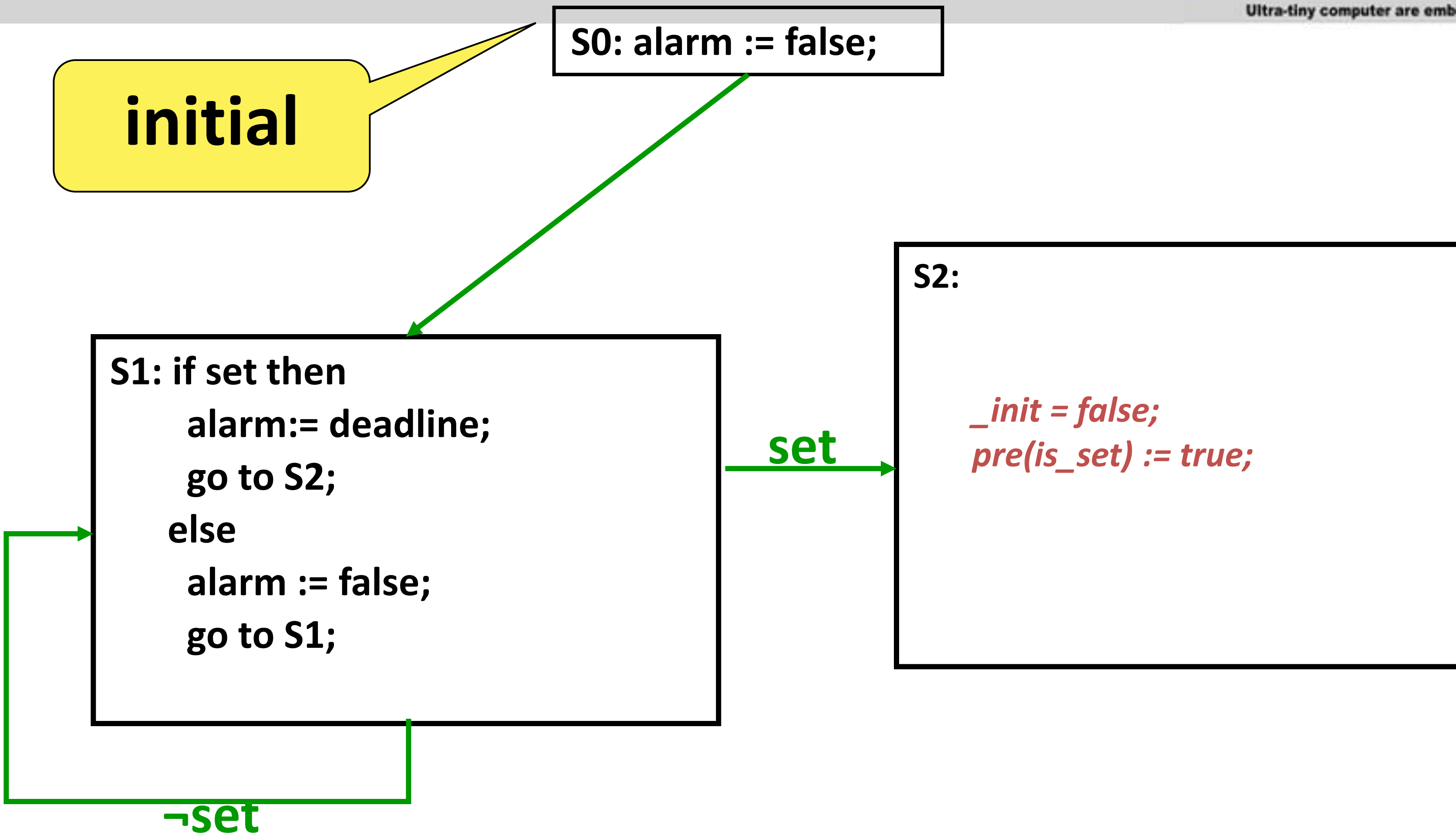
```
if _init then // first instant (or reaction)
  is_set := false; alarm := false;
   _init := false;
else   // following reactions
  if set then is_set := true
  else
    if reset then is_set := false;
    endif
  endif
  alarm := is_set and deadline;
endif
```

Lustre Program Compilation




```
if _init then // first instant (or reaction)
  is_set := false; alarm := false;
   _init := false;
else   // following reactions
  if set then is_set := true
  else
    if reset then is_set := false;
    endif
  endif
  alarm := is_set and deadline;
endif
```

Lustre Program Compilation



Lustre Program Compilation

initial

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

```
if _init then // first instant (or reaction)
  is_set := false; alarm := false;
   _init := false;
else   // following reactions
  if set then is_set := true
  else
    if reset then is_set := false;
    endif
  endif
  alarm := is_set and deadline;
endif
```

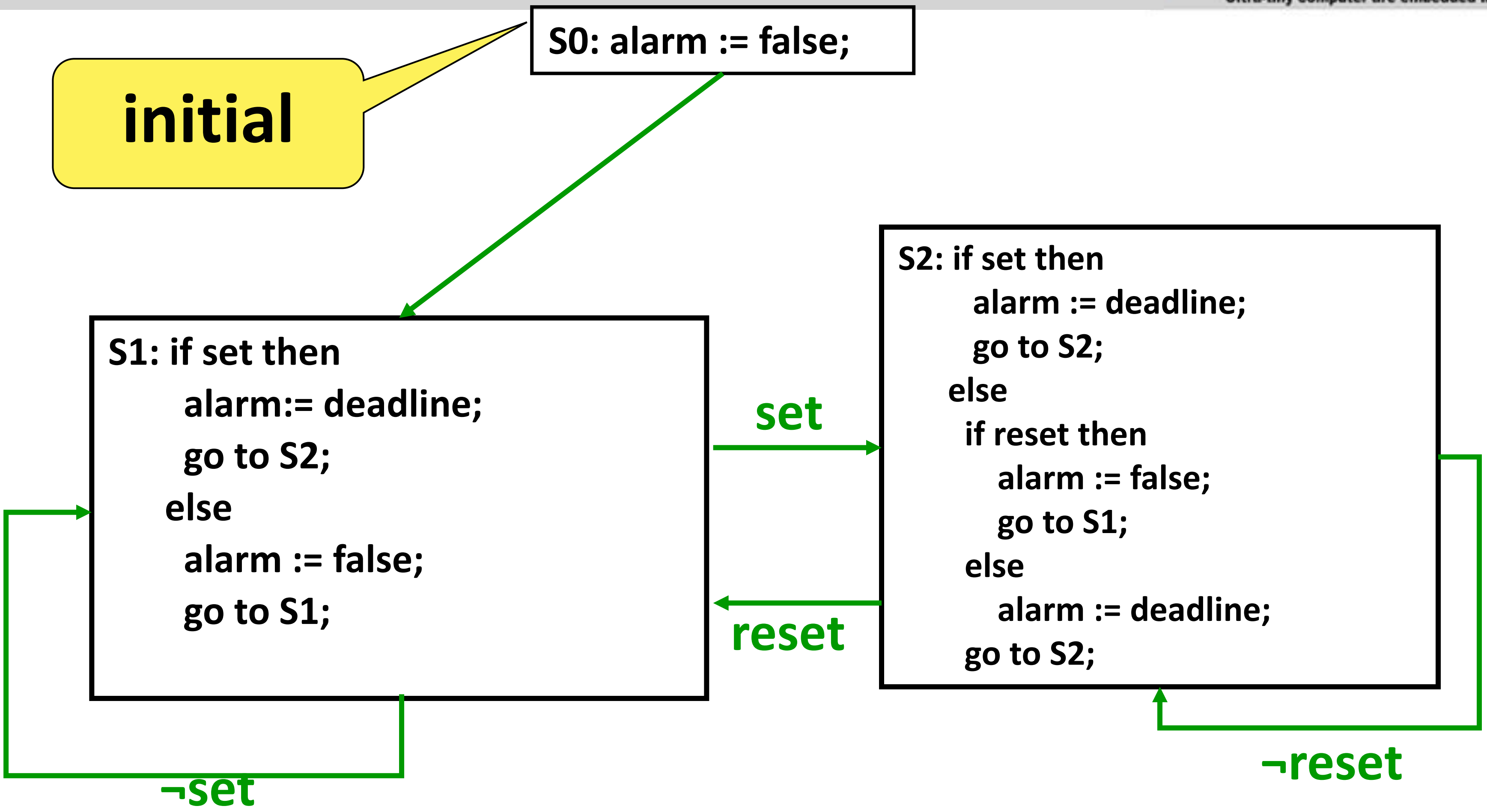
set

reset

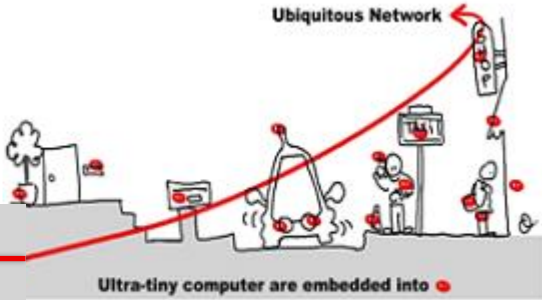
```
S2: if set then
        alarm := deadline;
        go to S2;
     else
      if reset then
         alarm := false;
         go to S1;
      else
         alarm := deadline;
      go to S2;
```

¬reset

Lustre Program Compilation

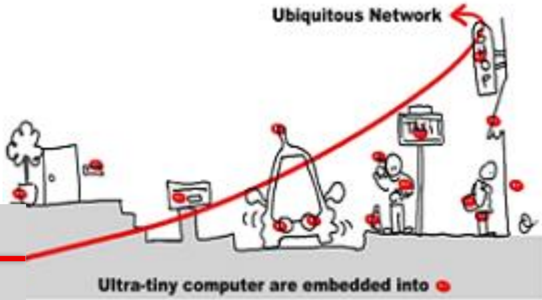


Model Checking Technique



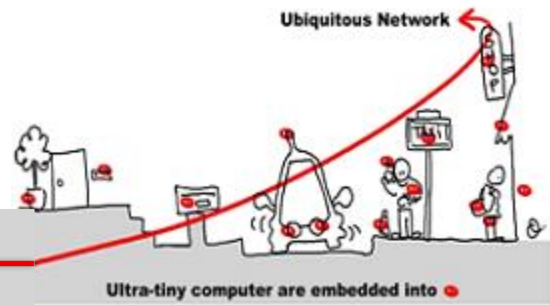
- Model = automata which is the set of program behaviors
- Properties expression = temporal logic:
  - LTL : liveness properties
  - CTL: safety properties
- Algorithm =
  - LTL : algorithm exponential wrt the formula size and linear wrt automata size.
  - CTL: algorithm linear wrt formula size and wrt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

Properties Checking



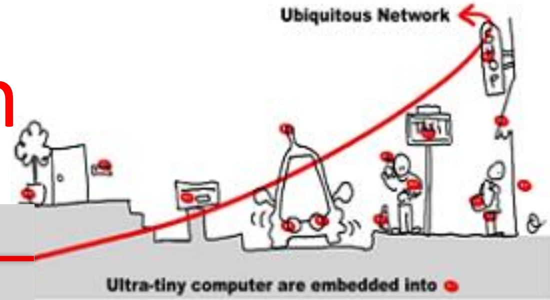
- Liveness Property Φ :
  - $\Phi \Rightarrow$ automata $B(\Phi)$
  - $\mathbb{L}(B(\Phi)) = \varnothing$ decidable
  - $\Phi \models \mathcal{M}$: $\mathbb{L}(\mathcal{M} \otimes B(\sim\Phi)) = \varnothing$

Safety Properties



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

Safety Properties Verification



We call Sat(∅) the set of states where ∅ is true.

$\mathcal{M} \models \varnothing$ iff $s_{init} \in Sat(\varnothing)$.

Algorithm:

$Sat(\Phi) = \{ s \mid \Phi \models s \}$

$Sat(\text{not } \Phi) = S \setminus Sat(\Phi)$

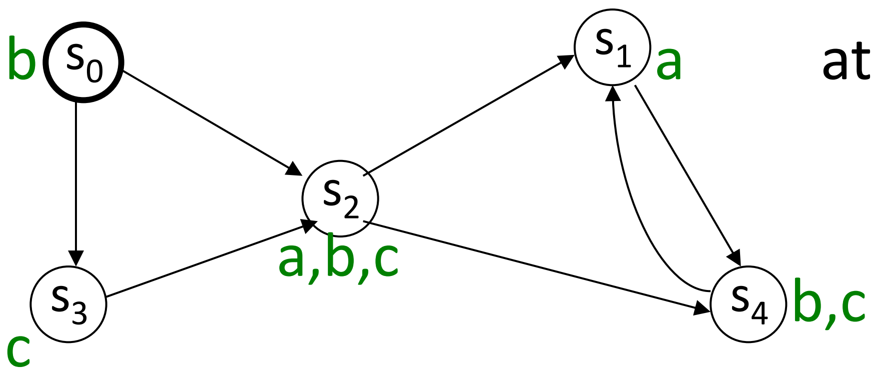
$Sat(\Phi 1 \text{ or } \Phi 2) = Sat(\Phi 1) \cup Sat(\Phi 2)$

$Sat(EX\ \Phi) = \{ s \mid \exists\, t \in Sat(\Phi),\ s \rightarrow t \}$ (Pre Sat(Φ))

$Sat(EG\ \Phi) = \mathit{gfp}\ (\Gamma(x) = Sat(\Phi) \cap Pre(x))$

$Sat(E(\Phi 1\ U\ \Phi 2)) = \mathit{lfp}\ (\Gamma(x) = Sat(\Phi 2) \cup (Sat(\Phi 1) \cap Pre(x))$

Example



atomic formulas: a, b, c

EG (a or b)

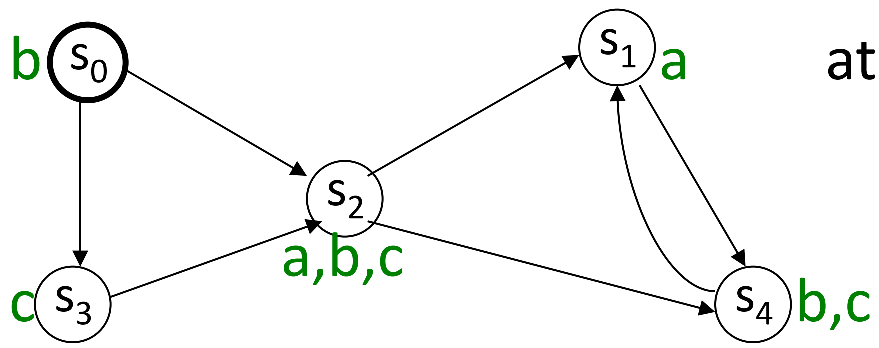
gfp (Γ(x) = Sat(a or b) $\cap$ Pre(x))

$\Gamma(\{s_0, s_1, s_2, s_{3,} s_4\})$ = 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



atomic formulas: a, b, c

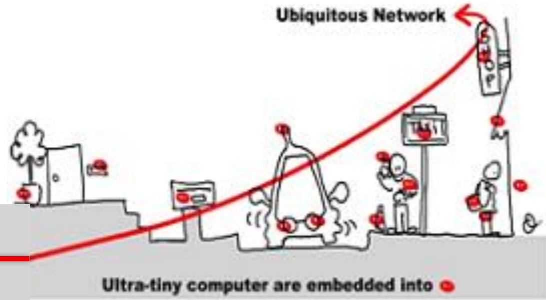
EG (a or b) $\Gamma(\{s_0, s_1, s_2, s_3, s_4\}) = \{s_0, s_1, s_2, s_4\}$

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

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

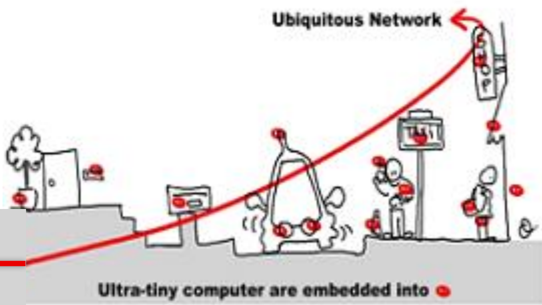
$S_0 \models EG(\text{ a or b})$

Model Checking Implementation



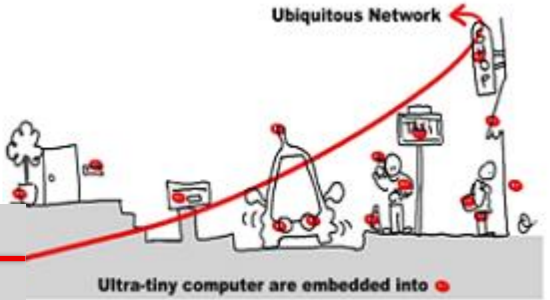
- Problem: the size of automata
- Solution: symbolic model checking
- Usage of BDD (Binary Decision Diagram) to encode both automata and formula.
- Each Boolean function has a unique representation
- Shannon decomposition:
  - $f(x_0,x_1,\ldots,x_n) = f(1, x_1,\ldots, x_n) \vee f(0, x_1,\ldots,x_n)$

Model Checking Implementation

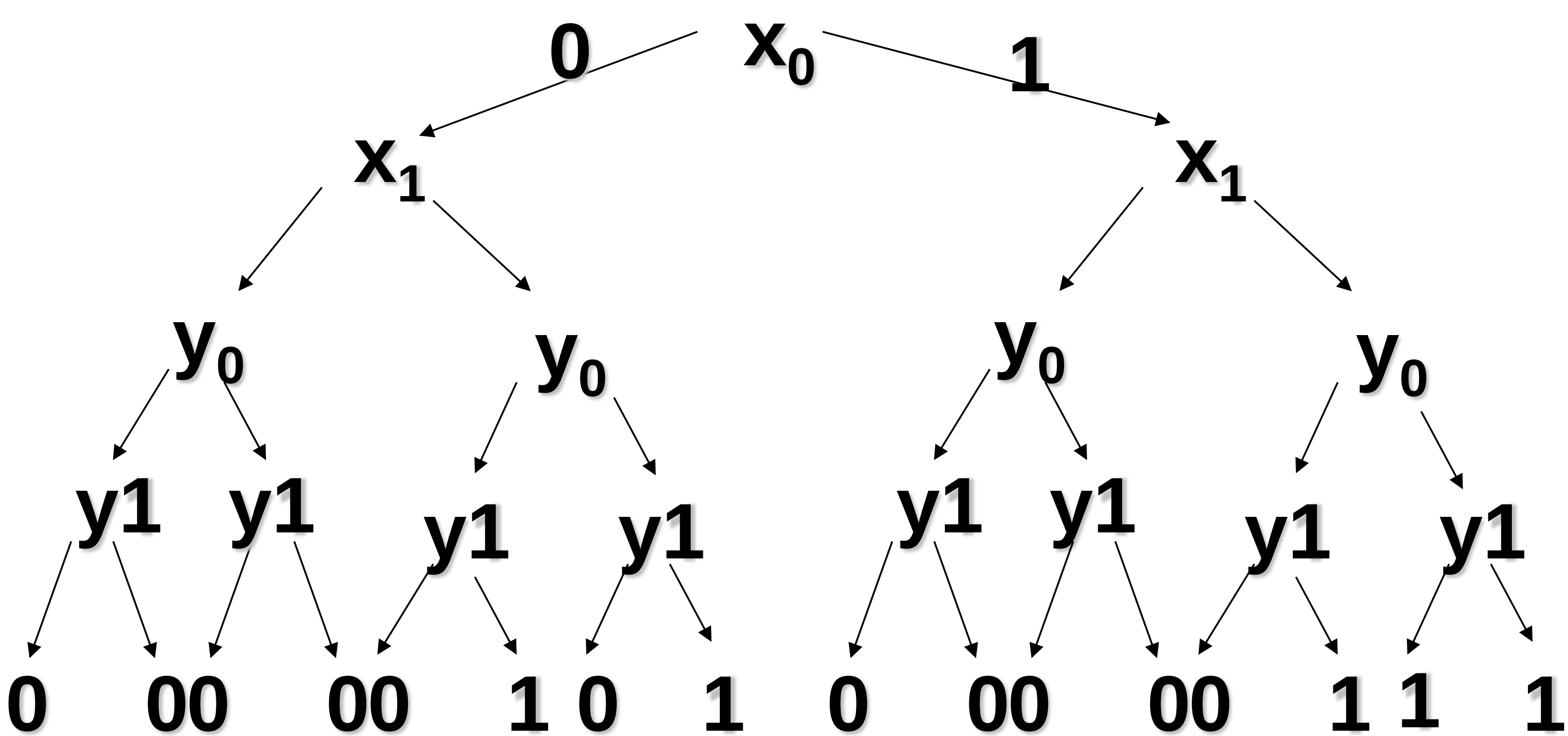


- When applying recursively Shannon decomposition on all variables, we obtain a tree where leaves are either 1 or 0.
- BDD are:
  - A concise representation of the Shannon tree
  - no useless node (if x then g else g ⇔ g)
  - Share common sub graphs

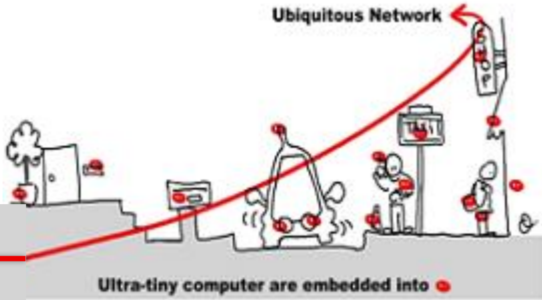
Model Checking Implementation (2)



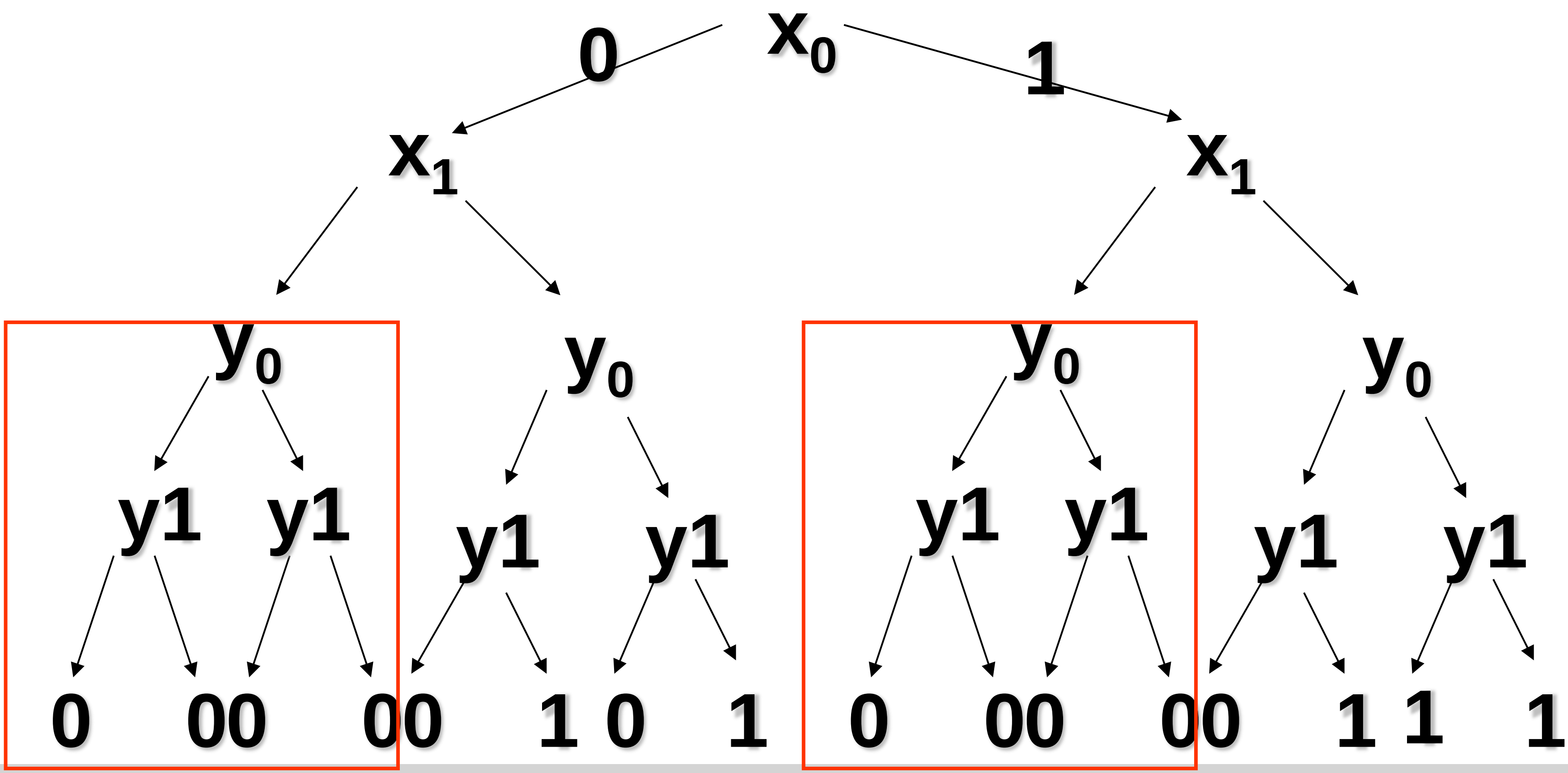
$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$



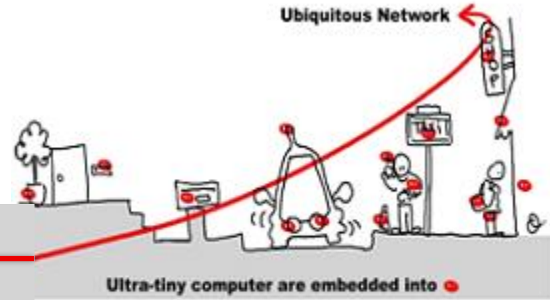
Model Checking Implementation (2)



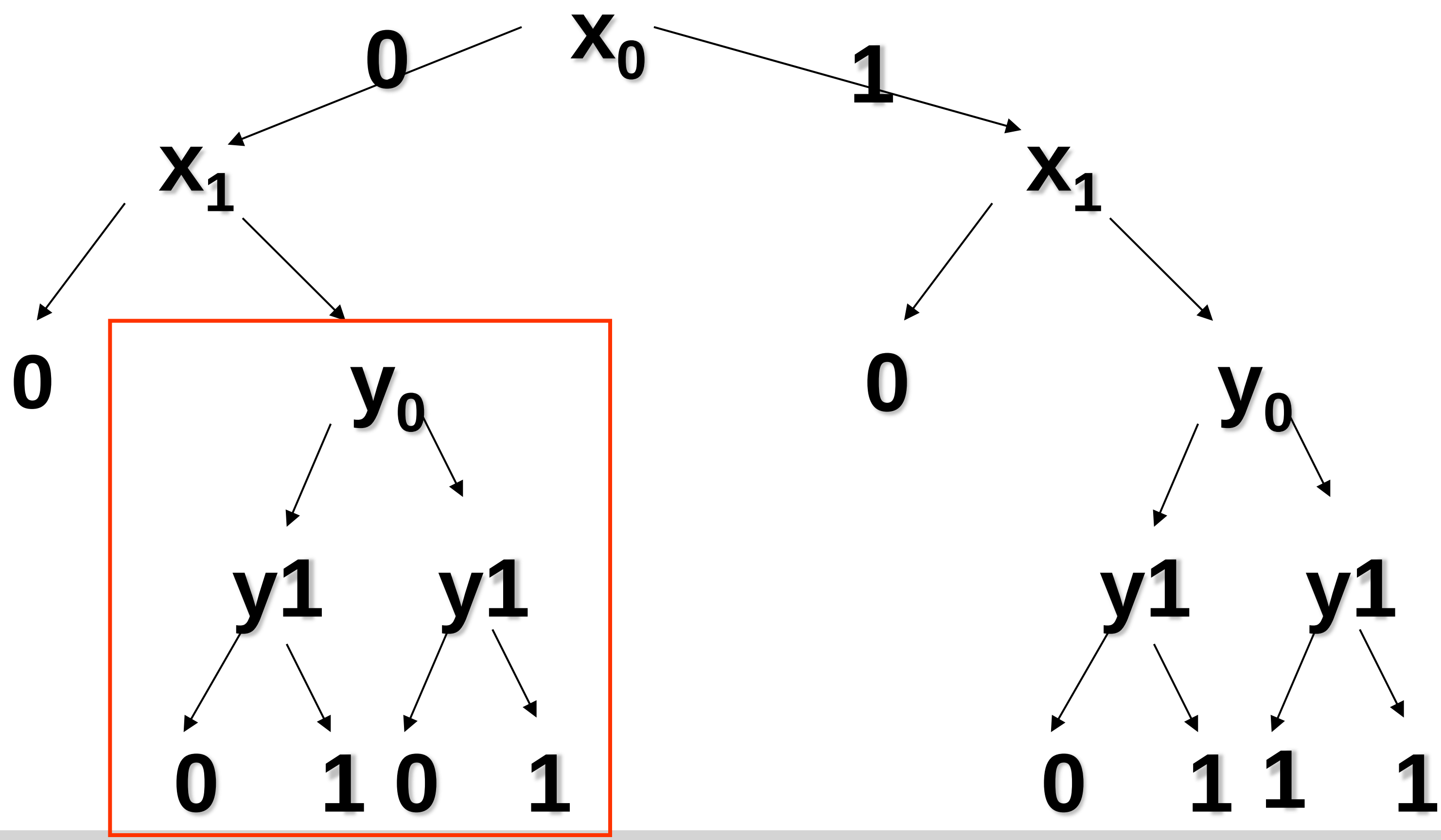
$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$



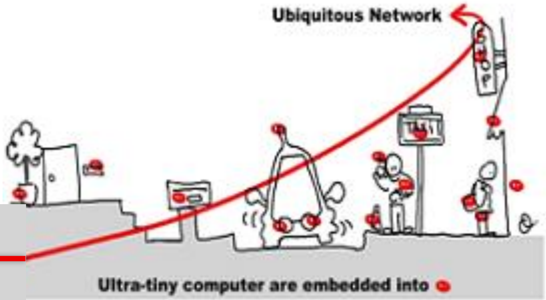
Model Checking Implementation (2)



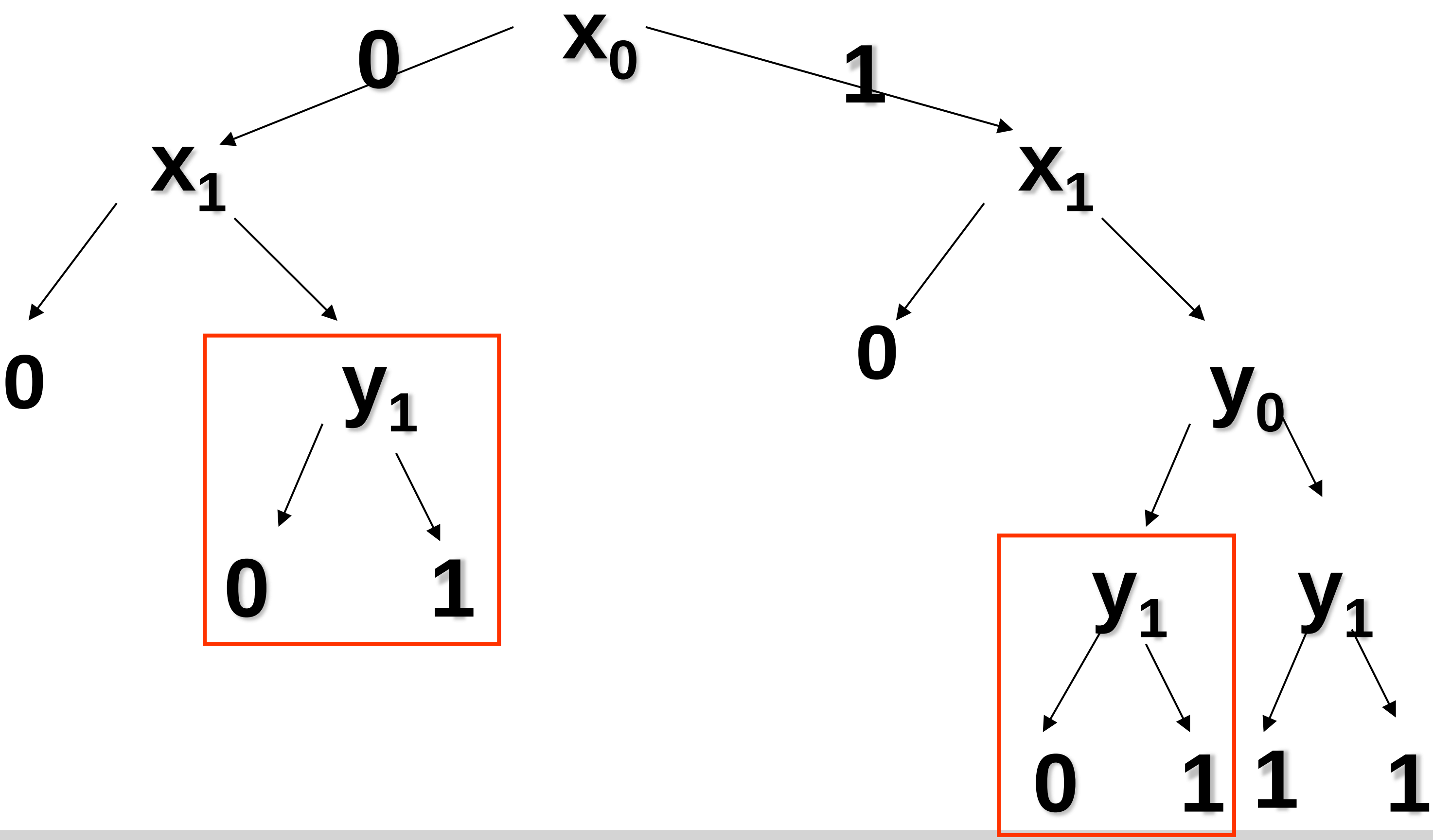
$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$



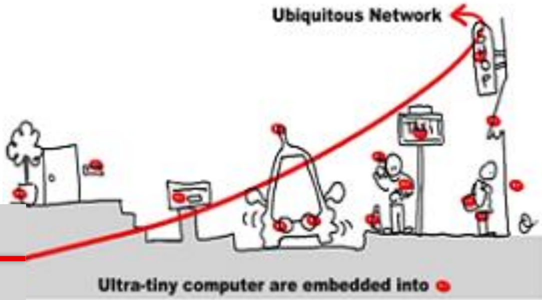
Model Checking Implementation (2)



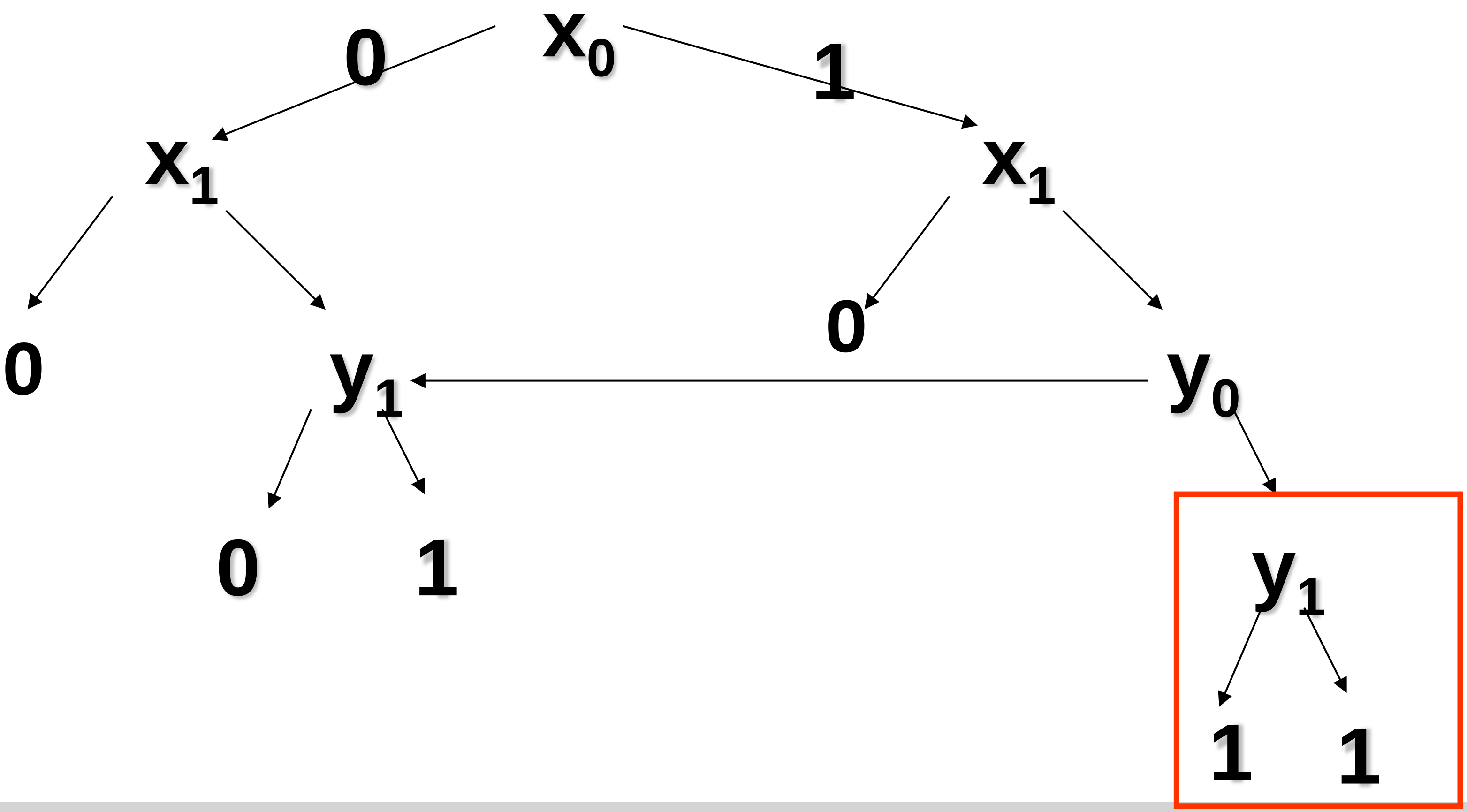
$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$



Model Checking Implementation (2)

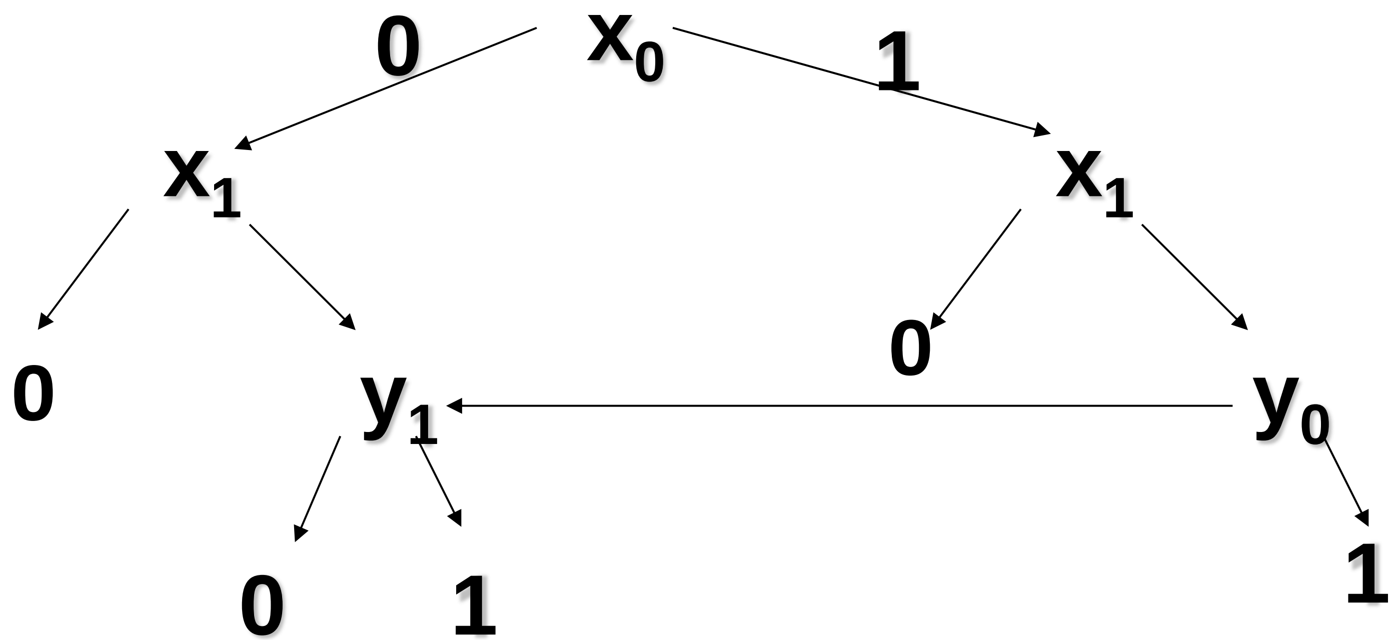


$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$

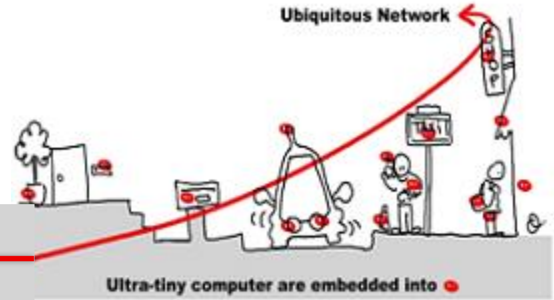


Model Checking Implementation (2)

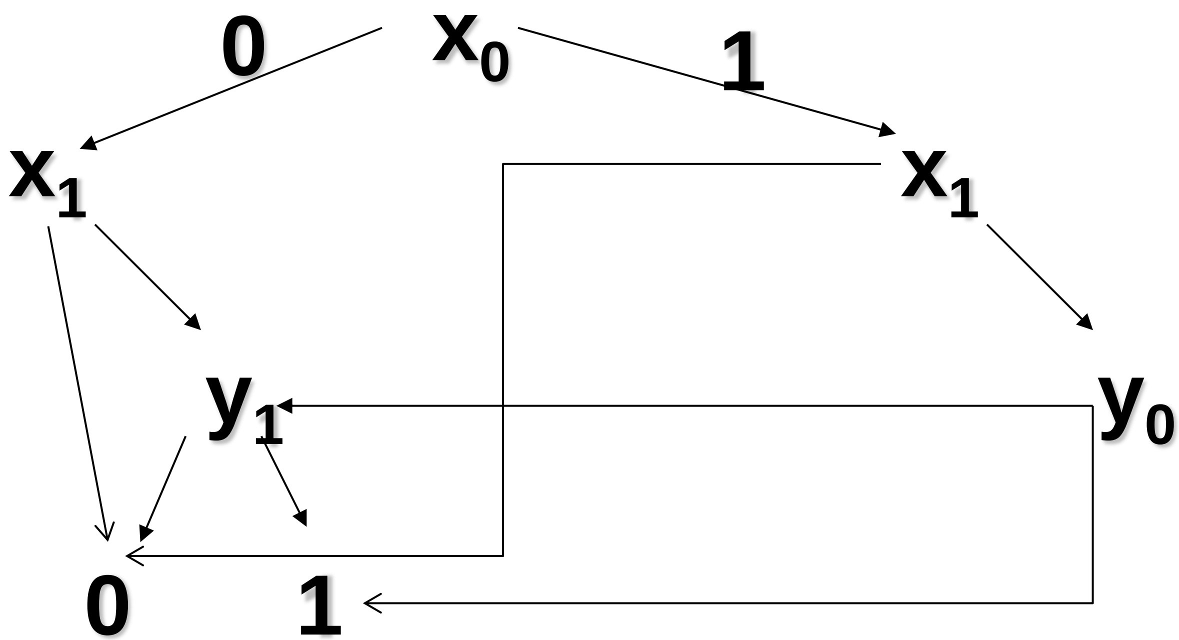
$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$



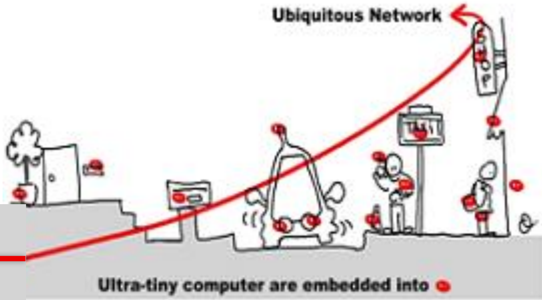
Model Checking Implementation (2)



$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$

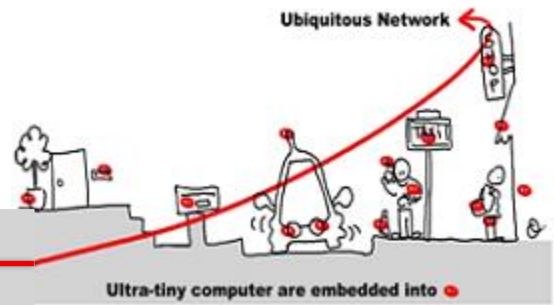


Model Checking Implementation(3)



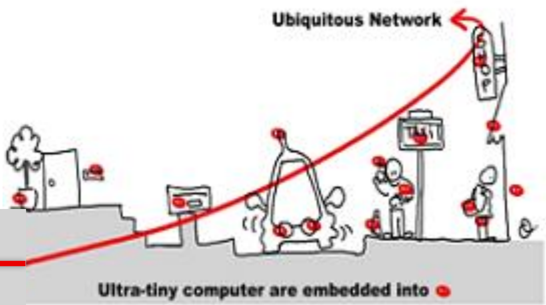
- Implicit representation of the of states set and of the transition relation of automata with BDD.
- BDD allows
  - canonical representation
  - test of emptiness immediate (bdd =0)
  - complementarity immediate (1 = 0)
  - union and intersection not immediate
  - Pre immediate

Model Checking Implementation (4)



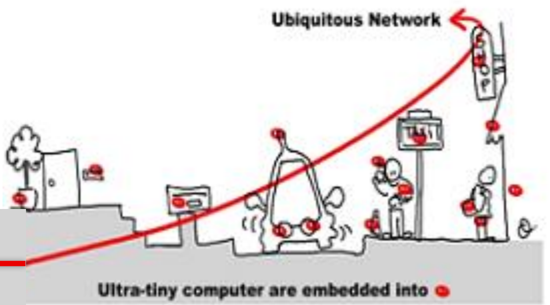
- But BDD efficiency depends on the number of variables
- Other method: SAT-Solver
  - Sat-solvers answer the question: given a propositional formula, is there exist a valuation of the formula variables such that this formula holds
  - first algorithm (DPLL) exponential (1960)

Model Checking Implementation (4)



- SAT-Solver algorithm:
  - formula ➔ CNF formula ➔ set of clauses
  - heuristics to choose variables
  - deduction engine:
    - propagation
    - specific reduction rule application (unit clause)
    - Others reduction rules
  - conflict analysis + learning

Model Checking Implementation (5)



- SAT-Solver usage:
  - encoding of the paths of length k by propositional formulas
  - the existence of a path of length k (for a given k) where a temporal property Φ is true can be reduce to the satisfaction of a propositional formula
  - theorem: given Φ a temporal property and $\mathcal{M}$ a model, then $\mathcal{M} \models \Phi \Rightarrow \exists\, n$ such that $\mathcal{M} \models_n \Phi$ $(n < |S| \,.\, 2^{|\Phi|})$

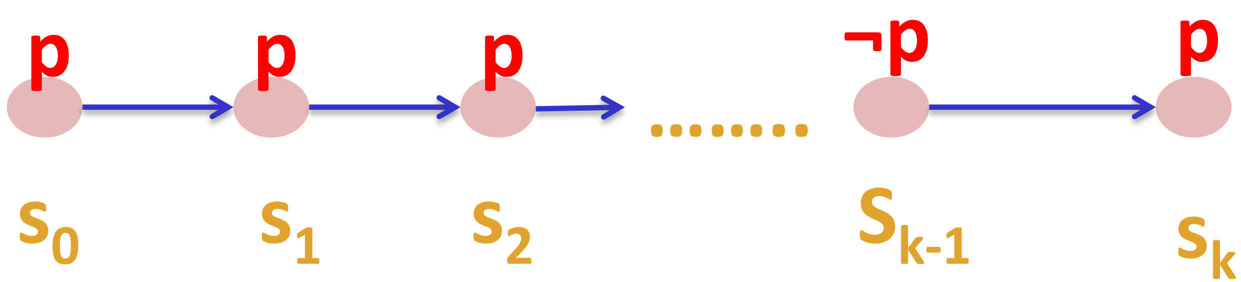
Bounded Model Checking

- SAT-Solver are used in complement of implicit (BDD based) methods.
- $\mathcal{M} \models \Phi$
  - verify $\neg \Phi$ on all paths of length k (k bounded)
  - useful to quickly extract counter examples

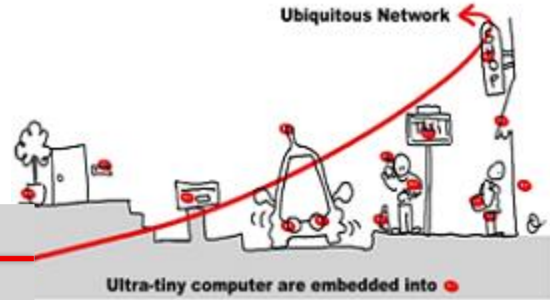
Bounded Model Checking

Given a property p

Is there a state reachable in *k* steps, which satisfies ¬p ?



Bounded Model Checking



The reachable states in *k* steps are captured by:

$$I(s_0)\ \wedge\ T(s_0,s_1)\ \wedge\ \ldots\ldots\ldots\ \wedge\ T(s_{k-1}, s_k)$$

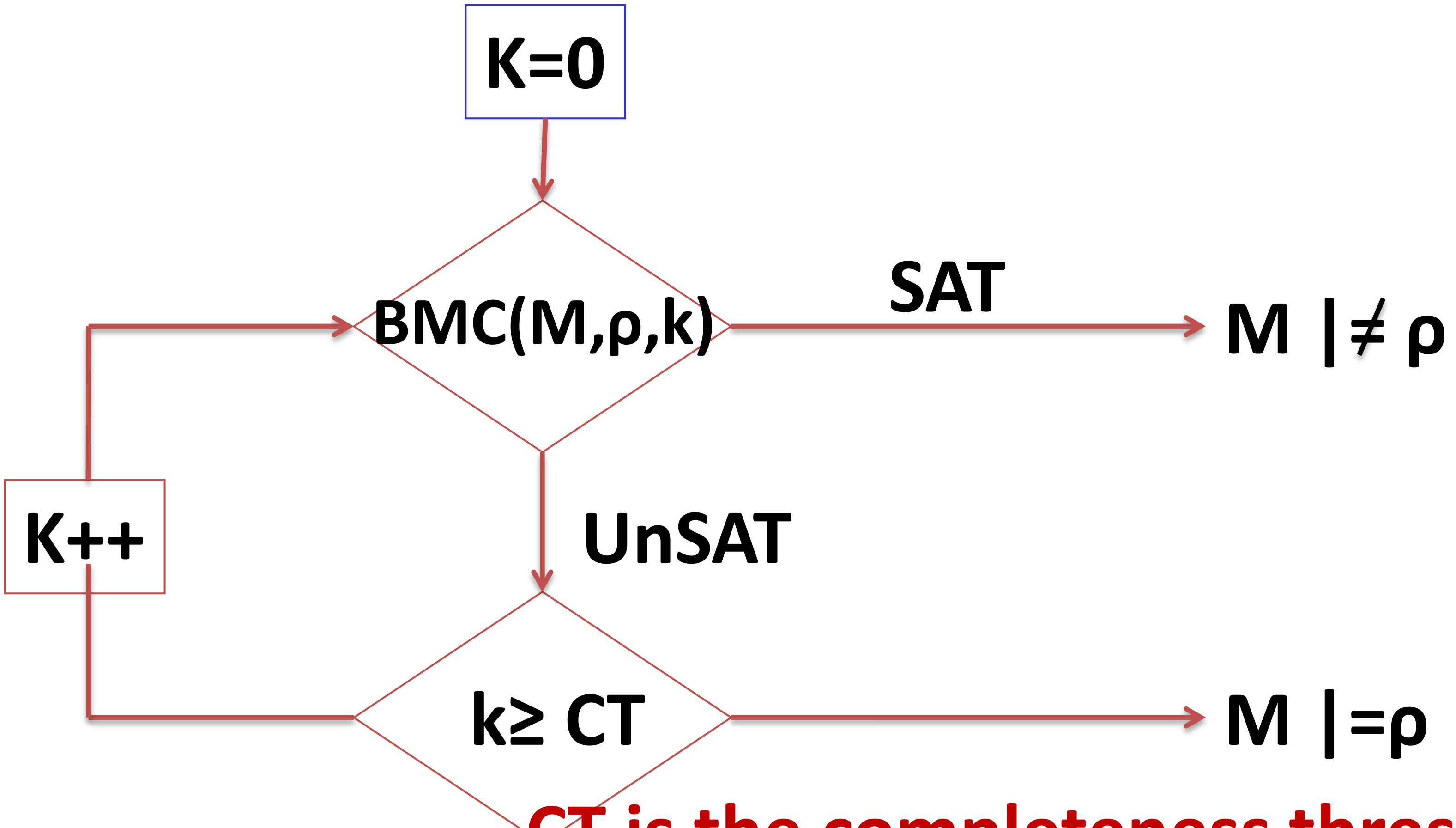
The property **p** fails in one of the k steps

$$\neg p(s_0) \vee \neg p(s_1) \vee \neg p(s_2) \ldots\ldots \vee \neg p(s_{k-1}) \vee \neg p(s_k)$$

The safety property **p** is valid up to step k iff Ω(k) is unsatisfiable:

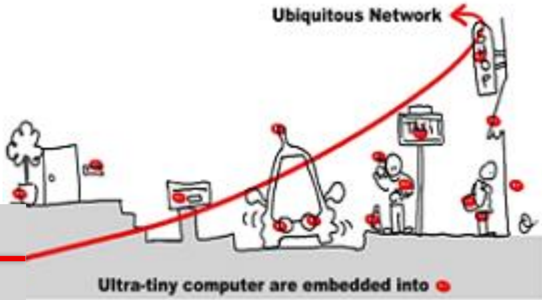
$$\Omega(k) = I(s_0) \wedge \left(\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \right) \wedge \left(\bigvee_{i=0}^{k} \neg p(s_i) \right)$$

Bounded Model Checking



CT is the completeness threshold

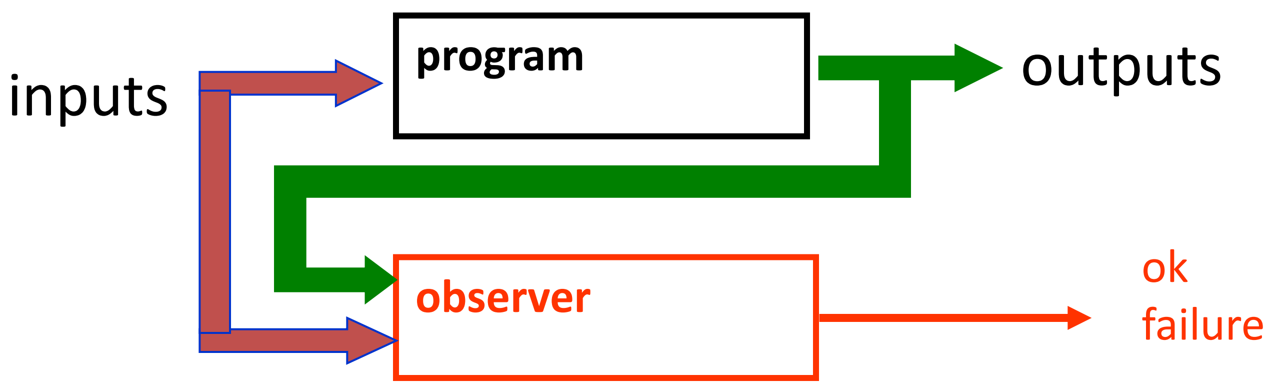
Bounded Model Checking



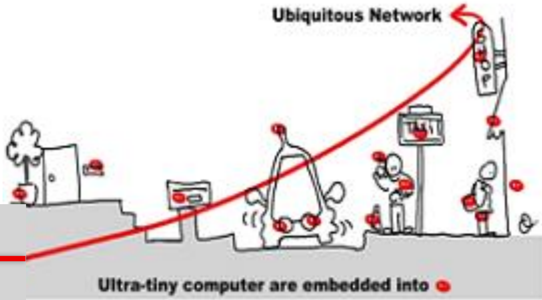
- Computing CT is as hard as model checking.
- Idea: Compute an over-approximation to the actual CT
  - Consider the system *as a graph.*
  - Compute *CT from structure of the graph.*
- Example: for **AGρ** properties, CT is the longest shortest path between any two reachable states, starting from initial state

Model Checking with Observers

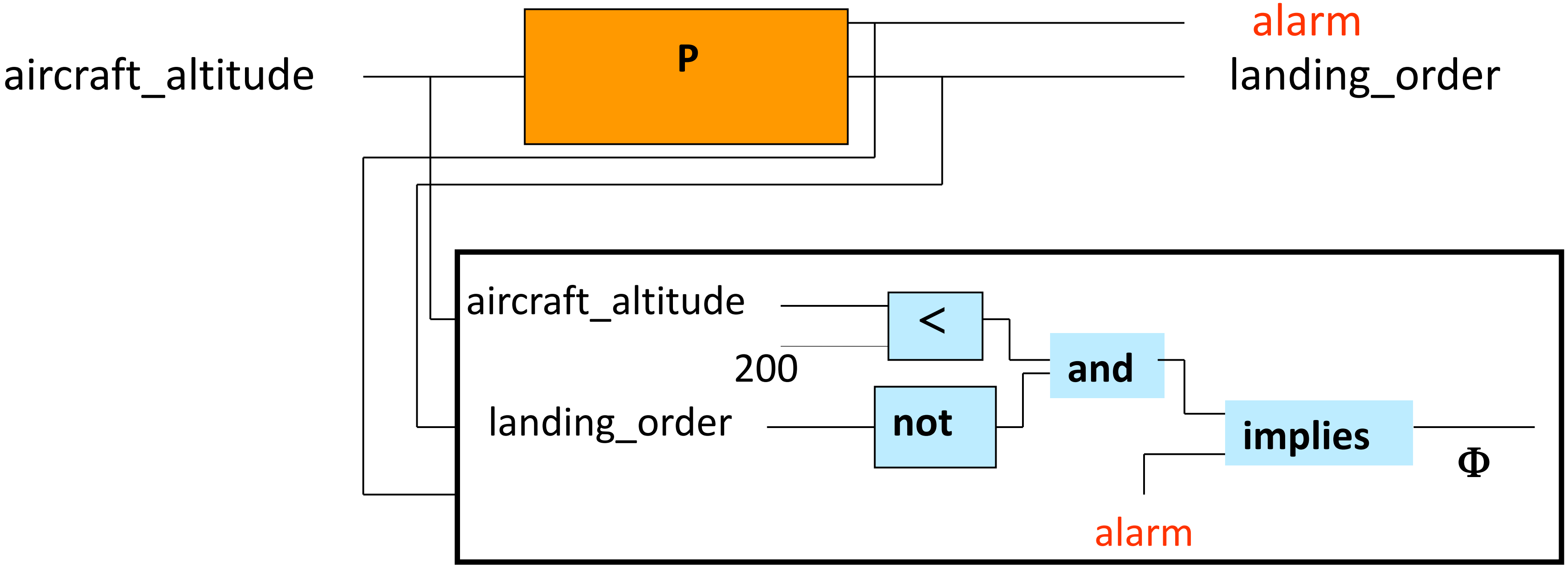
- Express safety properties as observers.
- An observer is a program which observes the program and outputs ok when the property holds and failure when its fails



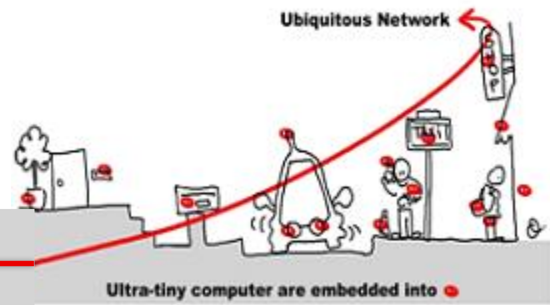
Model Checking with observers (2)



P: aircraft autopilot and security system



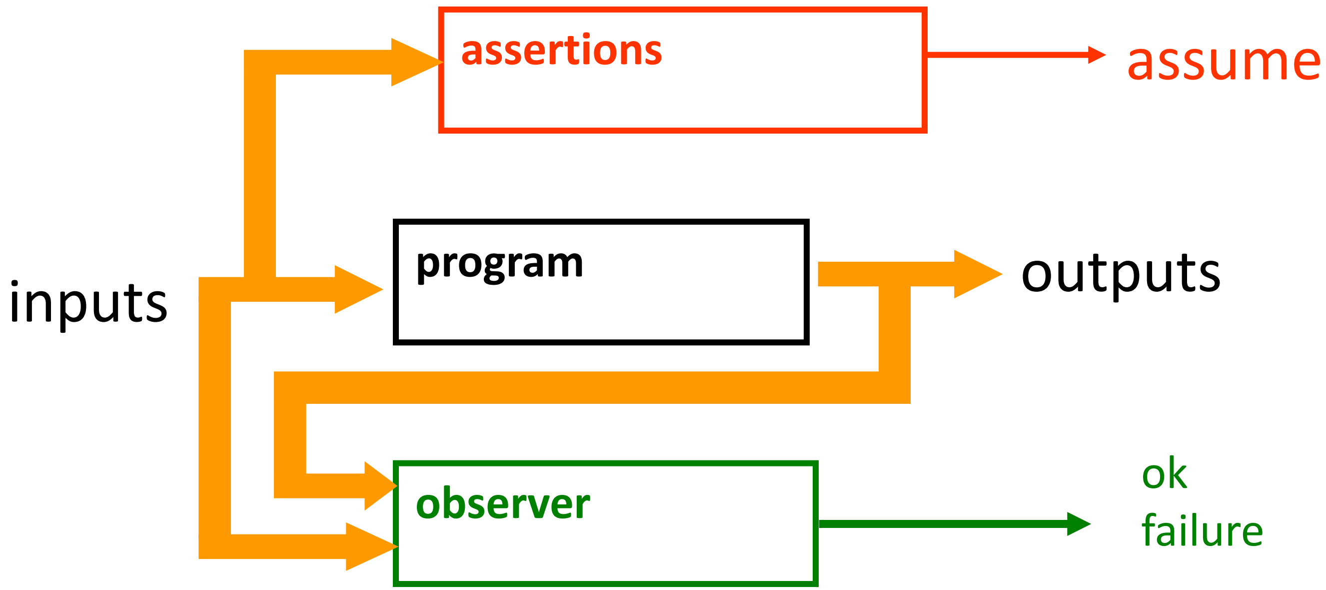
Properties Validation



- 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

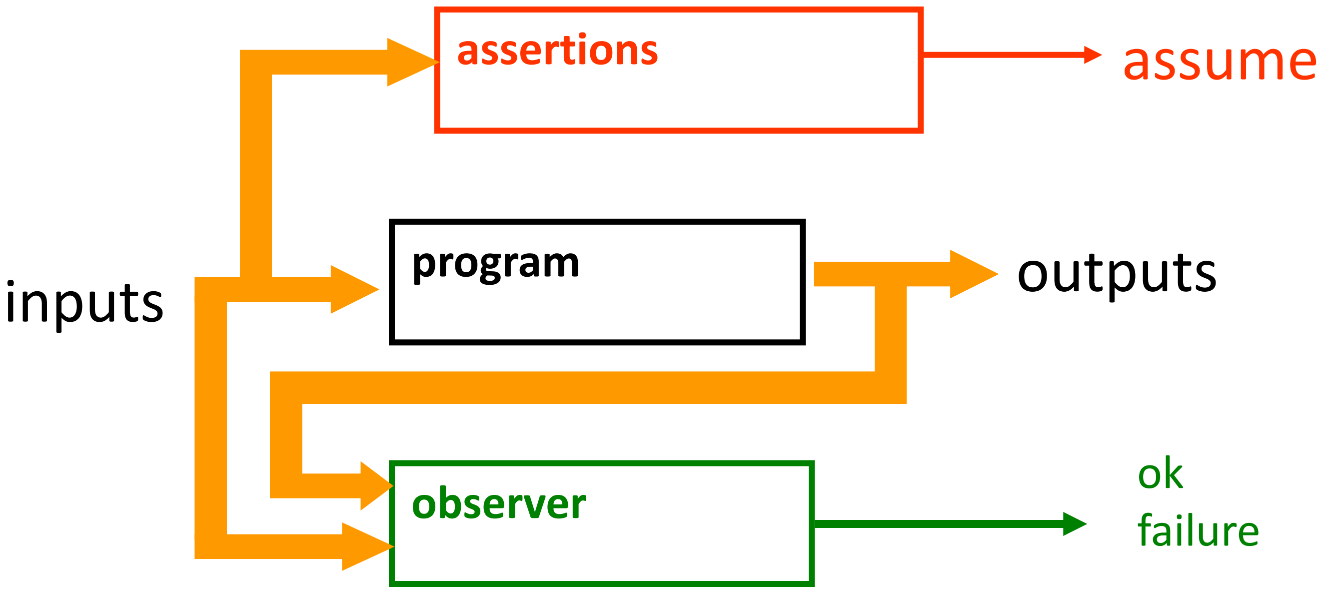
Properties Validation (2)

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

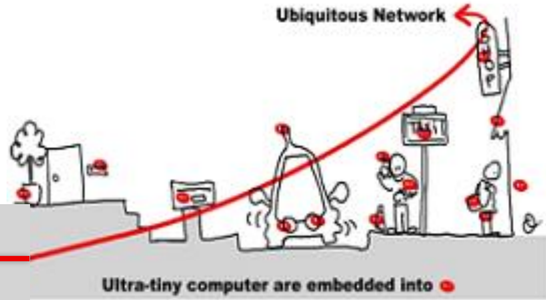


Properties Validation (3)

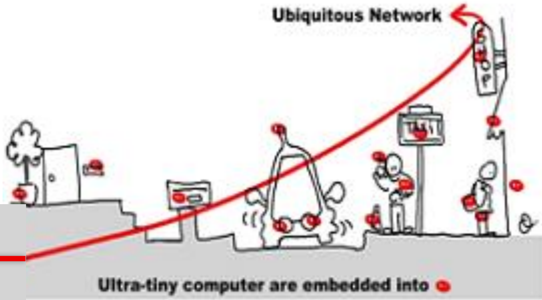
- if assume remains true, then ok also remains true (or failure false).



Outline



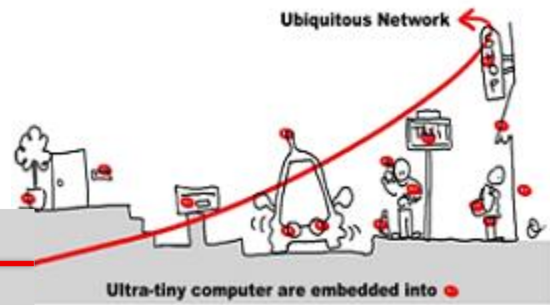
1. Critical system validation
2. Model-checking solution
   1. Model specification
   2. Model-checking techniques
3. Application to component based adaptive middleware
   1. Middleware critical component as synchronous models to allow validation
   2. The Scade and CLEM solutions



Practical Issues

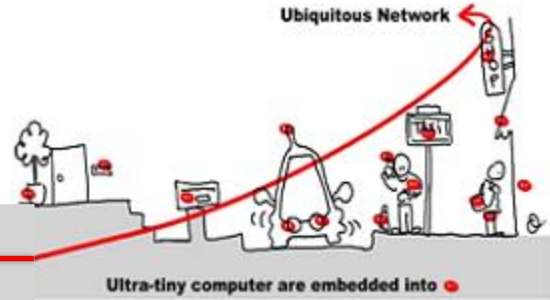
Application to Component Based Adaptive Middleware for Ubiquitous Computing

Component Modeling



- Adaptive middleware (as Wcomp) component listen to input events and provide output methods in reaction.
- They could be critical and response time sensitive
  - They should support formal validation
  - They should be deterministic
- Component behavior specification as **synchronous model**

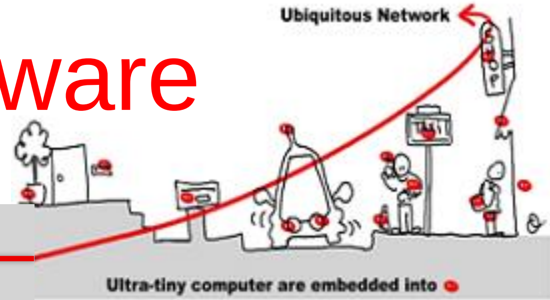
Synchronous Models



To sum up :

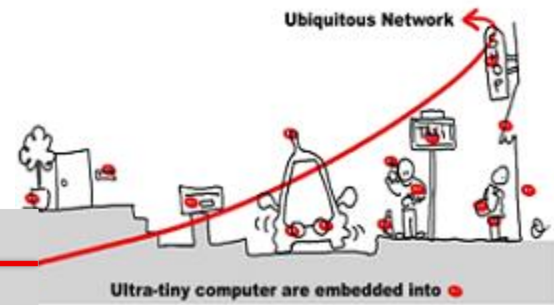
1. Synchronous models can be designed as **event-driven controllers** or **as data flow operator networks**
2. They always represent automata
3. Model-checking techniques apply

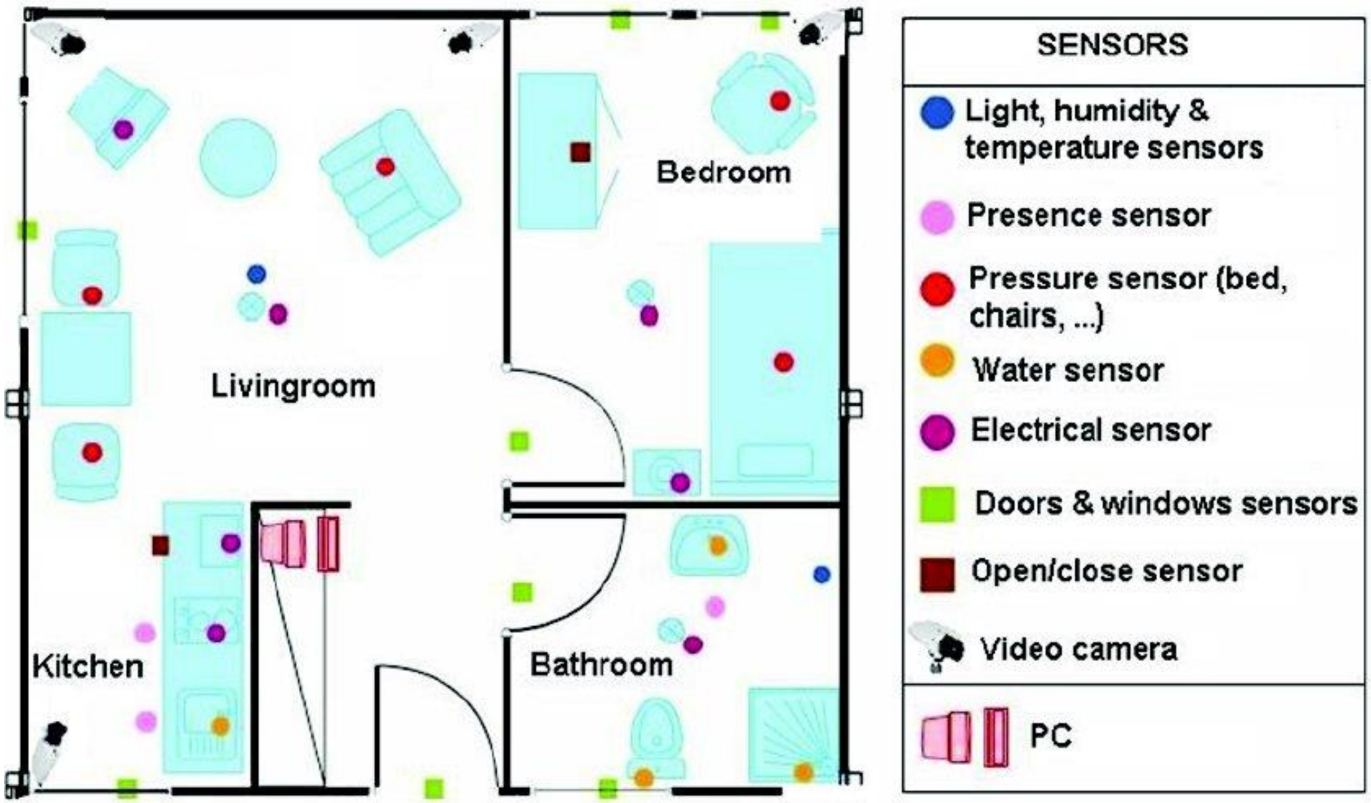
Application to Adaptive Middleware



- Our goal is to validate critical component of component based adaptive middleware for ubiquitous computing
- critical component will provide a synchronous model of their behaviors to allow model-checking techniques application as validation
- This synchronous model will be translated into a specific component called a **synchronous monitor**

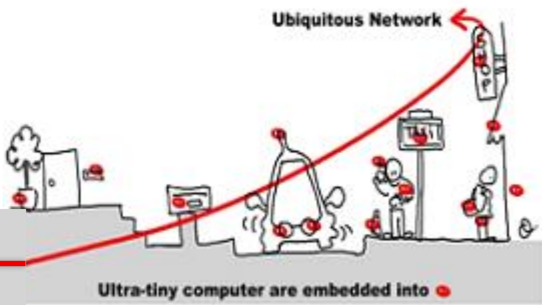
Use Case





Old adults monitoring in an instrumented home

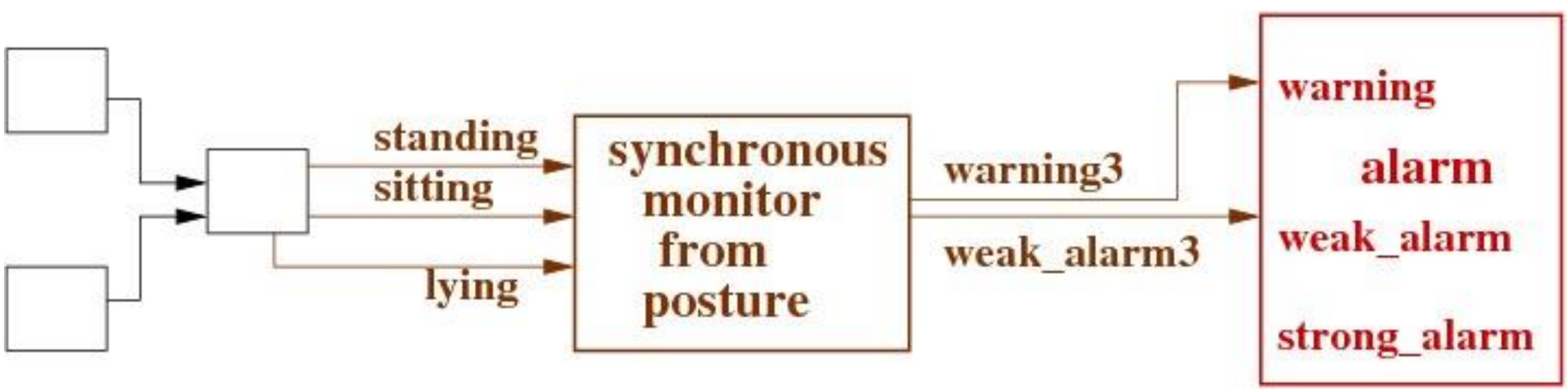
Use Case



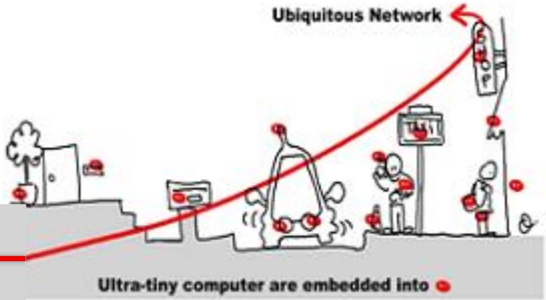
- **Use case**: observe kitchen usage
  1. Camera sensor (to locate the person)
  2. Fridge (contact sensor on the door) and a timer to know how long the door is opened
  3. Posture sensor (accelerometers) to know if the person is standing, sitting or lying
- **Goal**: send the appropriate alarm (strong, weak or warning)

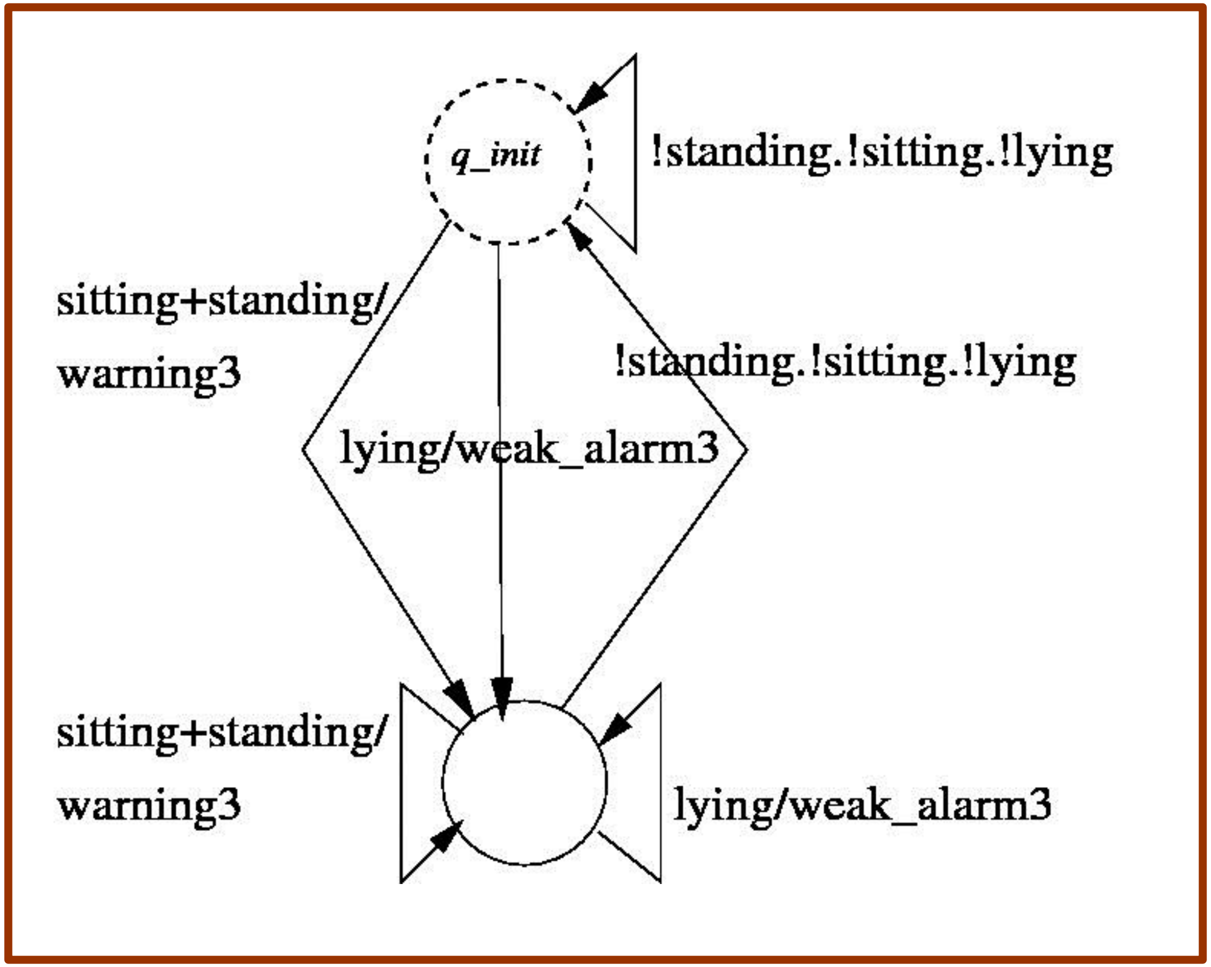
Use Case Implementation

- The **Alarm**, component is critical, 3 synchronous monitors will be introduced to specify the Alarm component behaviors w.r.t the fridge, the posture and the camera components



Use Case Implementation





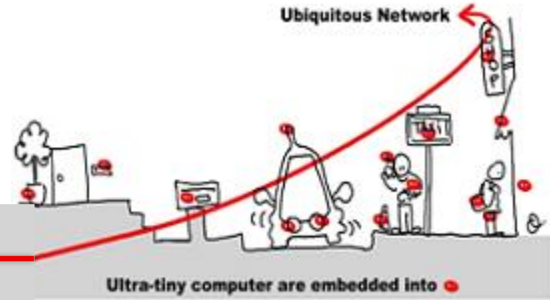
Posture synchronous monitor

The SCADE solution

- How design the posture component ?
- How validate its behaviors ?
- How introduce it in the overall design ?

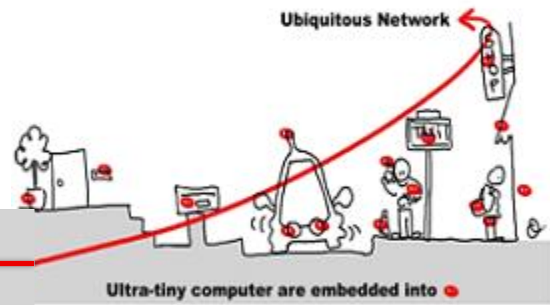
Rely on **Synchronous toolkit**

SCADE: Safety-Critical Application Development Environment

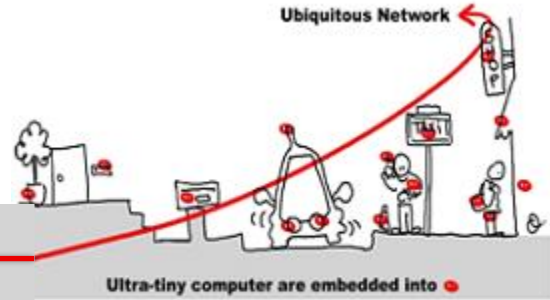


- Scade has been developed to address safety-critical embedded application design
- The Scade suite KCG code generator has been qualified as a development tool according to DO-178B norm at level A.

SCADE



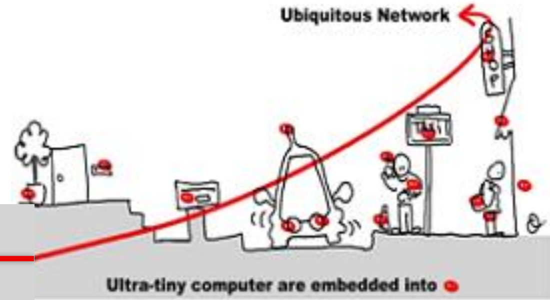
- Scade has been used to develop, validate and generate code for:
  - avionics:
    - Airbus A 341: flight controls
    - Airbus A 380: Flight controls, cockpit display, fuel control, braking, etc,..
    - Eurocopter EC-225 : Automatic pilot
    - Dassault Aviation F7X: Flight Controls, landing gear, braking
    - Boeing 787: Landing gear, nose wheel steering, braking



SCADE

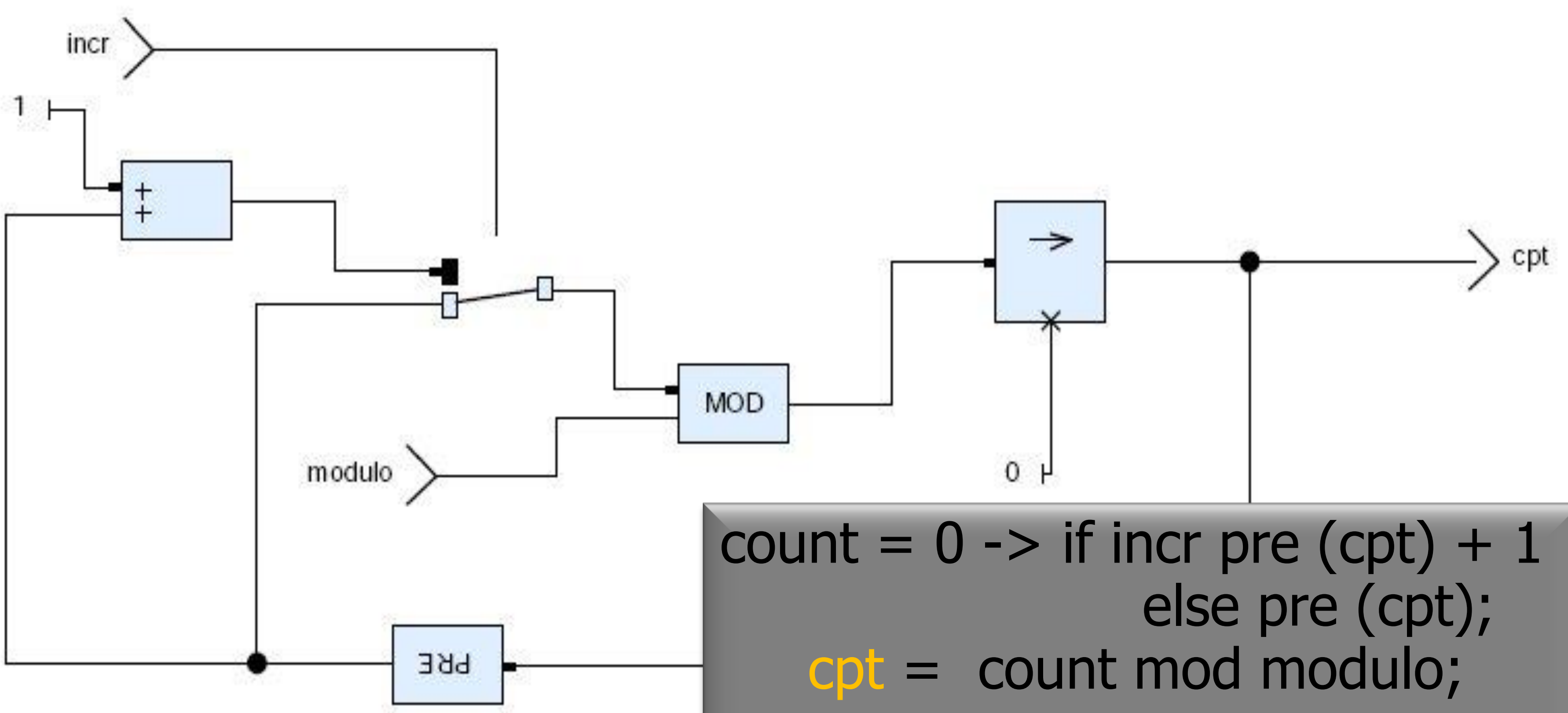
- System Design
  - Both data flows and state machines
- Simulation
  - Graphical simulation, automatic GUI integration
- Verification
  - Apply observer technique
- Code Generation
  - certified C code

Modulo Counter



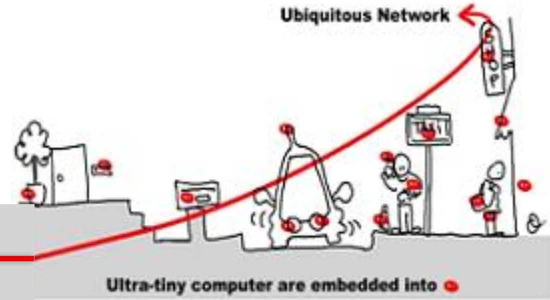

```
operator MCounter (incr:bool; modulo : int)
                        returns (cpt:int);
  var count : int;

    count = 0 -> if incr pre (cpt) + 1
                 else pre (cpt);
    cpt =  count mod modulo;
```

Modulo Counter



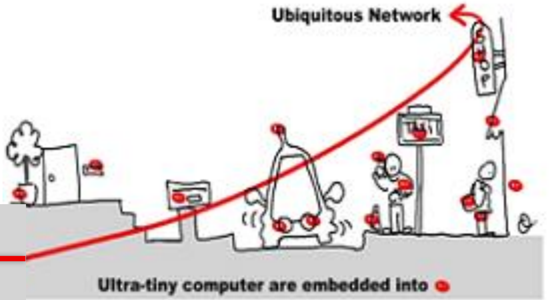
count = 0 -> if incr pre (cpt) + 1
else pre (cpt);
cpt = count mod modulo;

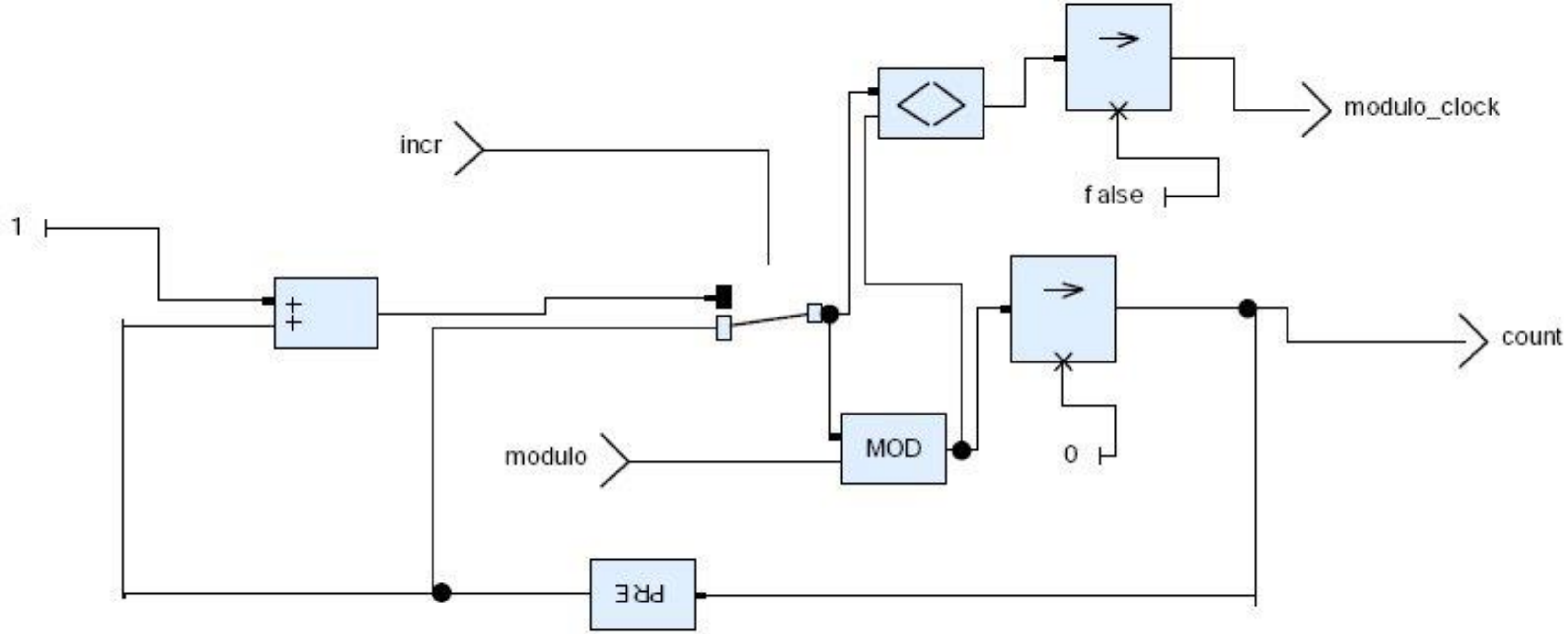
Modulo Counter Clock



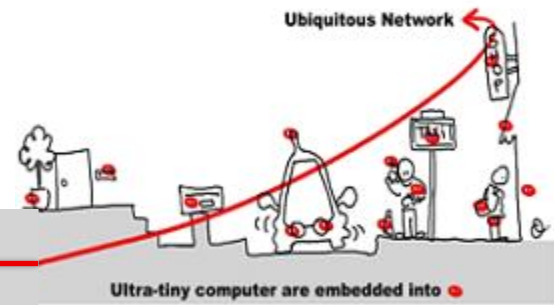

```
operator MCounterClock (incr:bool;
                         modulo : int)
              returns(cpt:int;
                         modulo_clock: bool);
  var count : int;
   count = 0 -> if incr pre (cpt) + 1
                else pre (cpt);
   cpt =  count mod modulo;
    modulo_clock = count <> cpt;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

Modulo Counter Clock



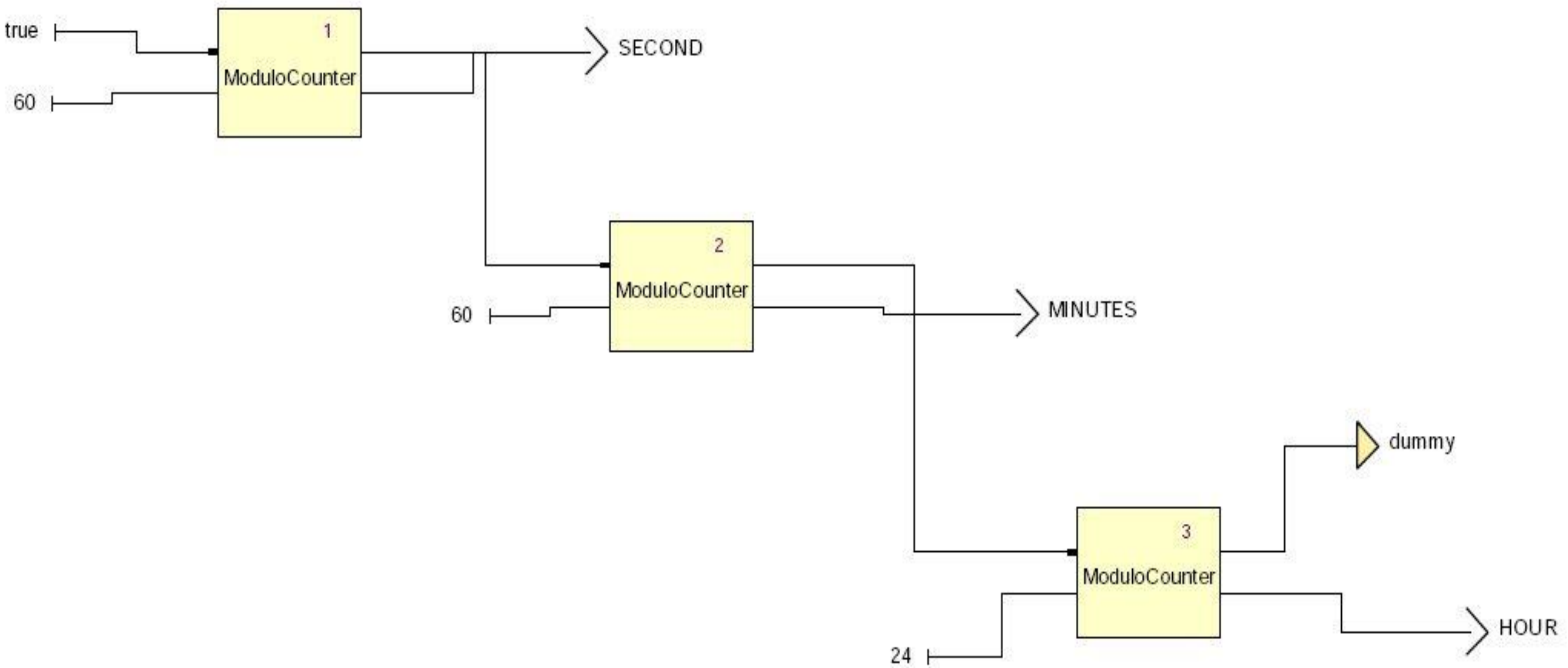


Timer

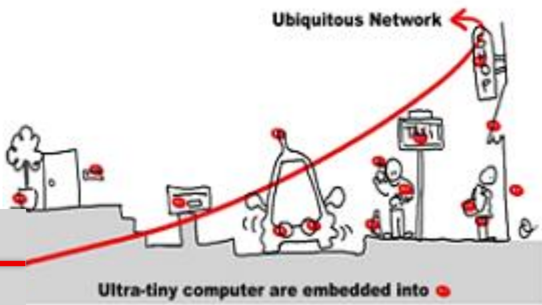



```
operator Timer returns (hour, minute, second:int);
var hour_clock, minute_clock, day_clock : bool;

  (second, minute_clock) = MCounterClock(true, 60);
  (minute, hour_clock) =
                    MCounterClock(minute_clock,60);
  (hour, dummy_clock) =
                    MCounterClock(hour_clock, 24);
```

Timer


true
60
1
ModuloCounter
SECOND
2
ModuloCounter
60
MINUTES
dummy
3
ModuloCounter
24
HOUR

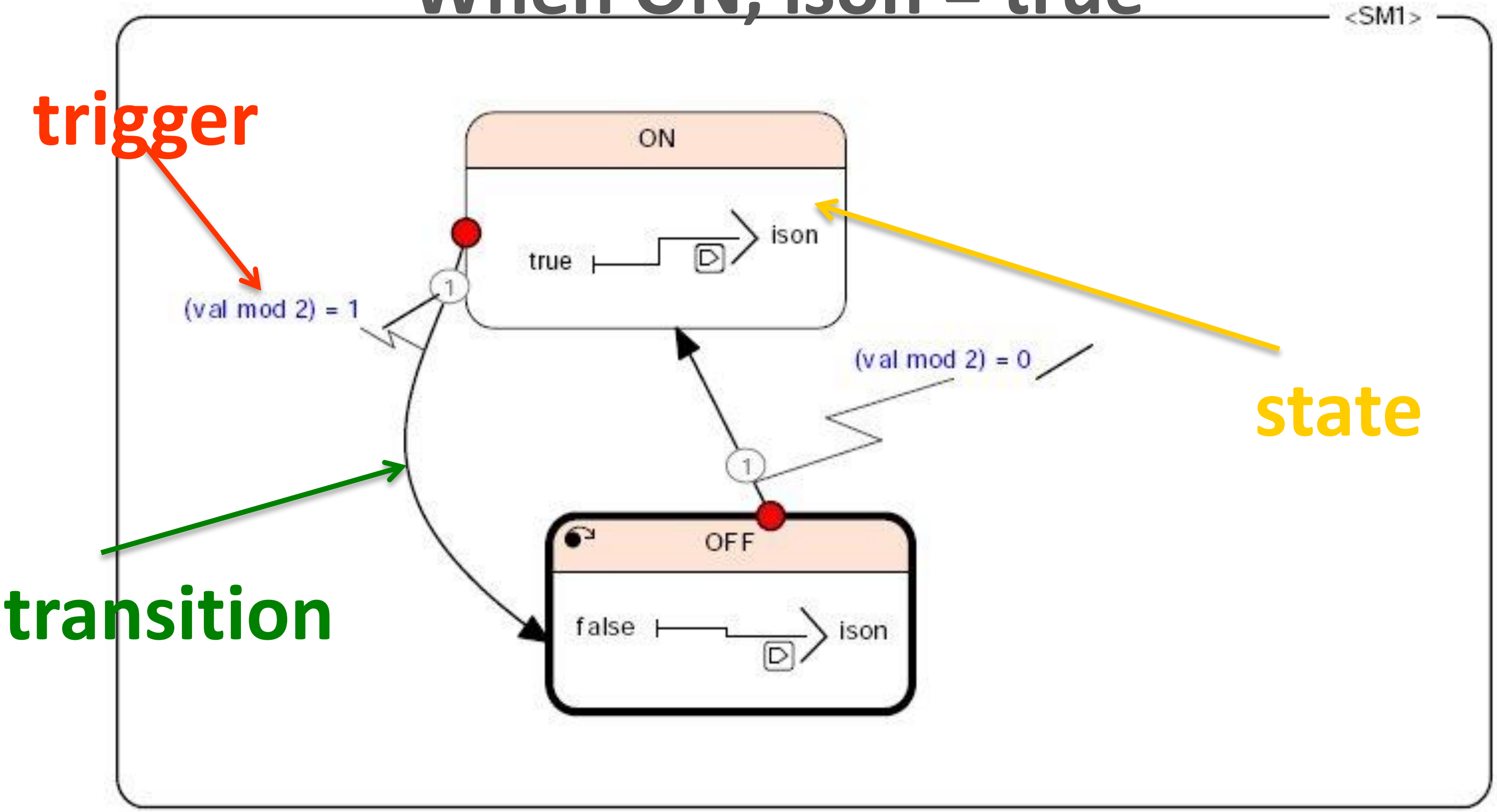
SCADE: state machines



- Input and output: same interface
- States:
  - Possible hierarchy
  - Start in the initial state
  - Content = application behavior
- Transitions:
  - From a state to another one
  - Triggered by a Boolean condition

SCADE: state machines

When ON, ison = true

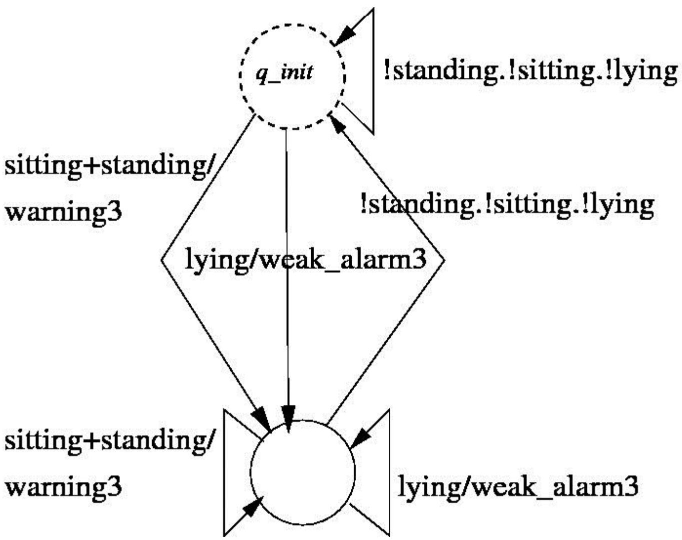


When off, ison = false

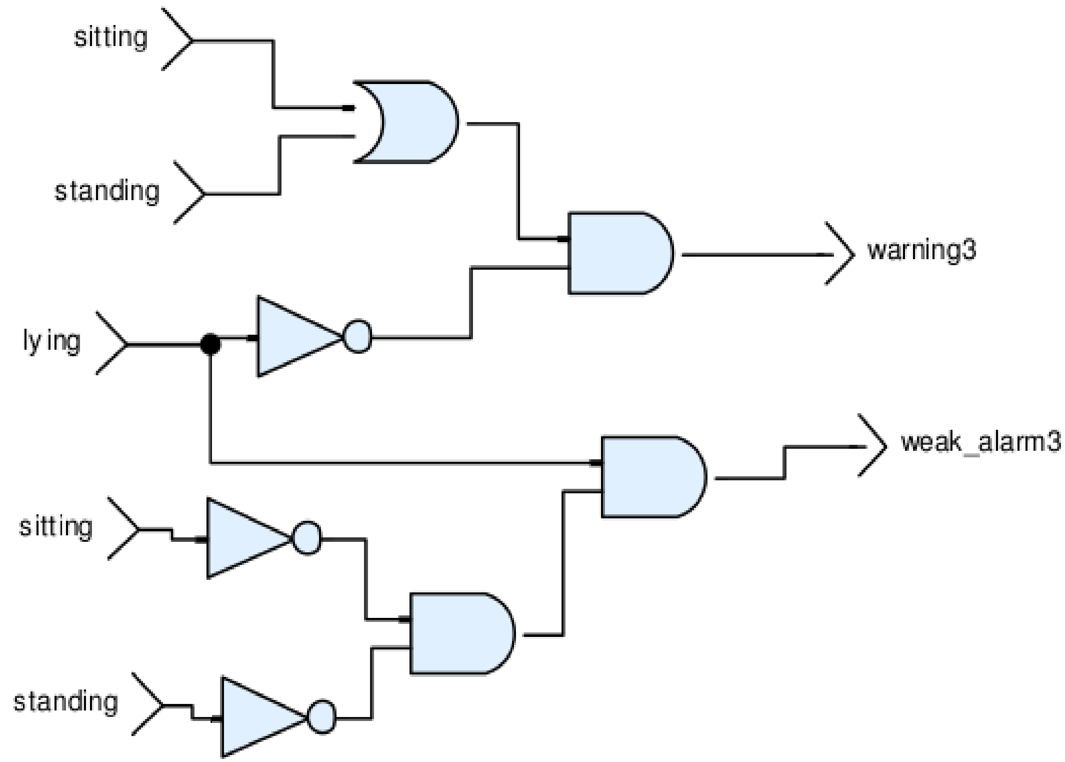
SCADE: model checking

Observer technique

posture model



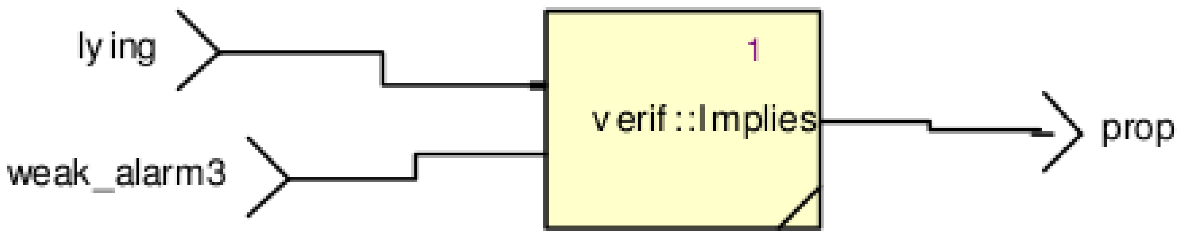
posture model specification in scade

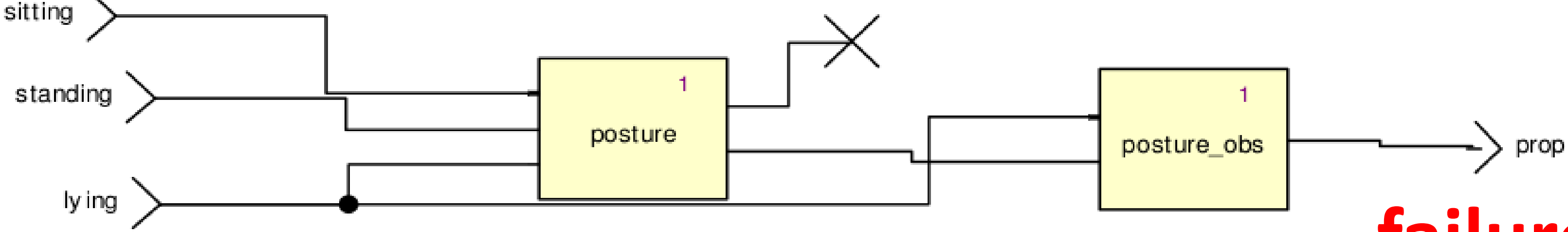


SCADE: model checking

Observer technique

posture observer

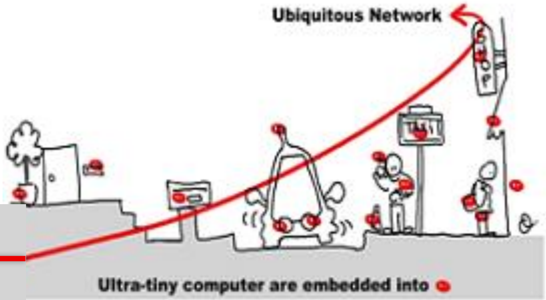




failure

posture verification

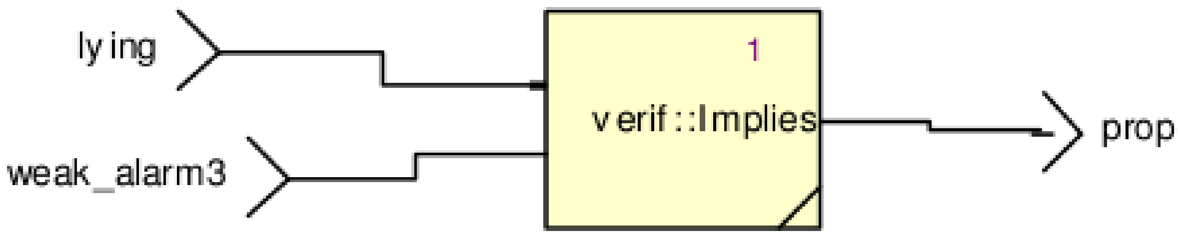
lying: true; sitting:true;standing:true

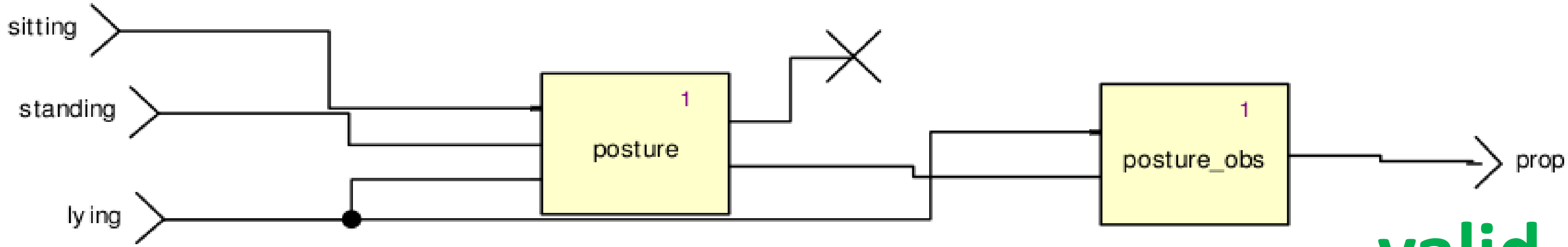


SCADE: model checking

Observer technique

posture observer



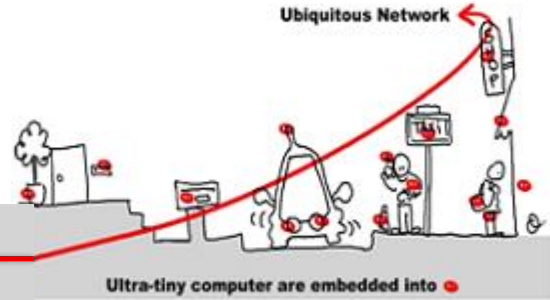


valid

posture verification

assume (lying # sitting # standing)

SCADE: code generation

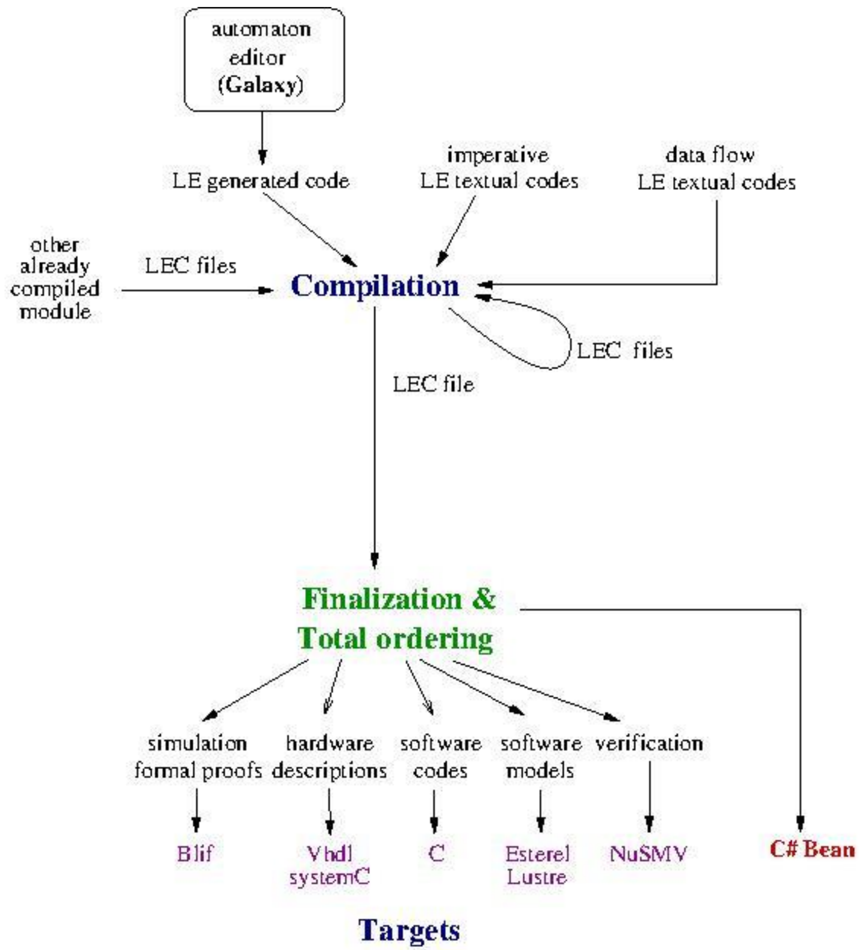


- KCG generates certifiable code (DO-178 compliance)
- Clean code, rigid structure (possible integration)
- Interfacing potential with user-defined code (c/c++)

CLEM versus SCADE

- SCADE suite:
  - Complex design environment
  - C code not embedded into C# bean easily
  - closed compilation environment
- Solution: use CLEM toolkit to specify and verify synchronous monitor before integration:
  - own compilation means
  - C# code generation

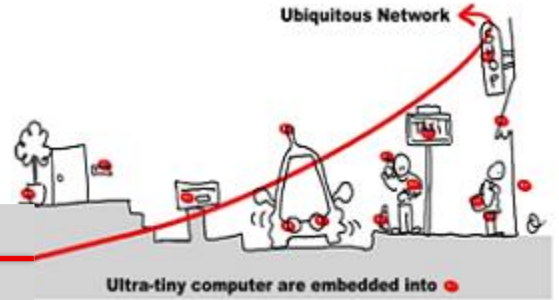
CLEM ISSUE



CLEM is a toolkit around the LE synchronous language offering:

- Modular compilation
- Simulation
- Verification
- Code generation for hardware and software targets (C#)

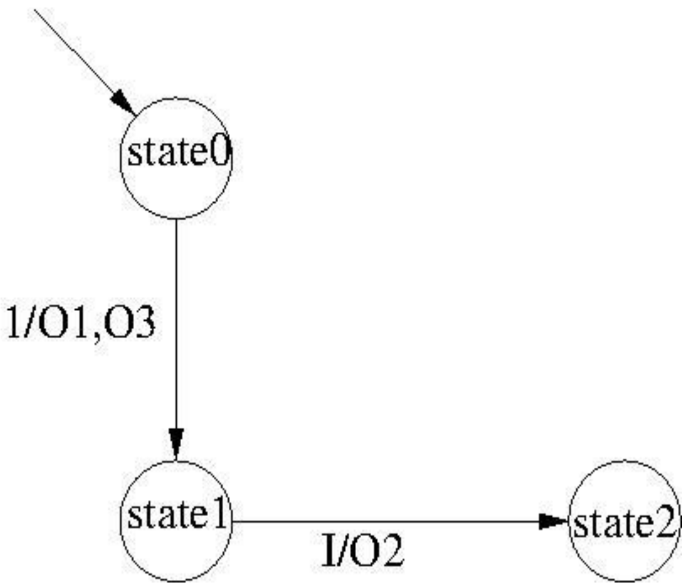
LE Language



- LE synchronous language
  - Textual imperative language (~ Esterel)
    - Usual synchronous languages operators:
      - || ; abort ; strong abort; sequence (>>); present; loop; emit
      - wait pause
    - run to call external module
  - Explicit Mealy machine (automata designed with Galaxy)
  - Implicit Mealy machine (~data flow)

LE Language

```
module Parallel:
Input:I;
Output: O1, O2,O3;
  emit O1
||
  wait I >> emit O2
||
   emit O3
end
```



LE Language

module Parallel:

Input:I;

Output: O1, O2,O3;

Mealy Machine

Register:

X0: 0: X0next;

X1: 0 : X1next;

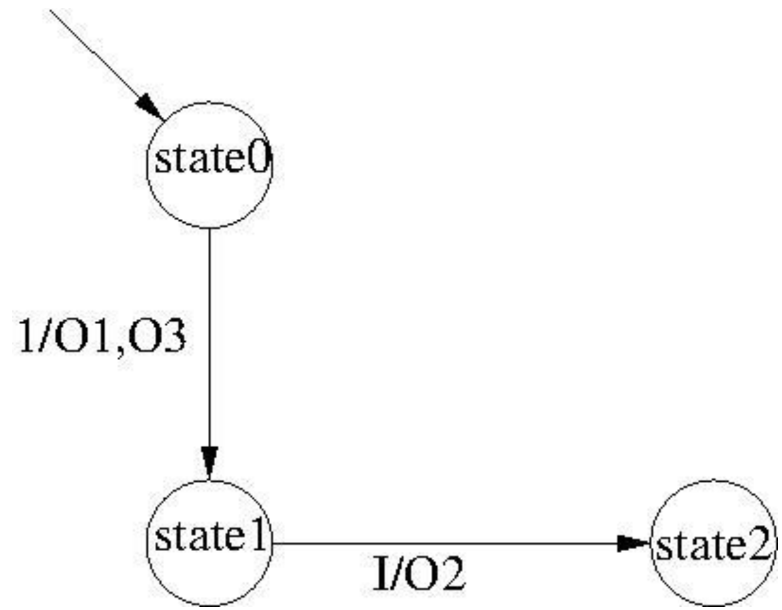
X0next = X0 and not X1;

X1next = X0 and X1 or not X1 and I
or not X0 and X1;

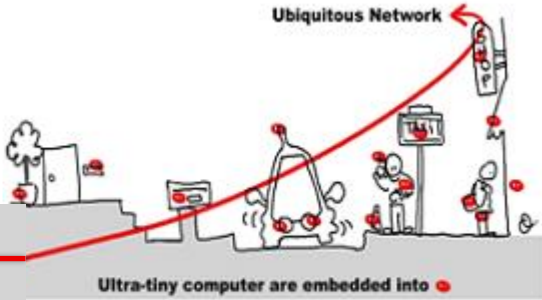
O1 = not X0 and not X1;

O2 = X0 and not X1 and I;

O3 = not X0 and not X1;



LE Compilation



- Compilation into implicit Mealy machines (Boolean equation systems with registers)
- Compilation ⇒ sort equation systems
- Challenge: modular compilation ?
  - ⇒ face causality problem
  - causality = no evaluation cycle in equation systems
  - total order prevents modularity
  - issue: compute partial orders

LE Compilation

```
module first:
Input: I1,I2;
Output: O1,O2;
loop {
 pause >>
 {
 present I1 {emit O1}
 ||
 present I2 {emit O2}
 }
end
```

```
module second:
Input: I3;
Output: O3;
loop {
 pause >> present I3 {emit O3}
}
end
```

```
module final:
Input: I;
Output O;
local L1,L2 {
 run first[ L2\I1,O\O1,I\I2,L1\O2]
 ||
 run second[ L1\I3,L2\O3]
}
end
```

O2= I2

O1 = I1

O3 = I3

L1 = I

O = L2

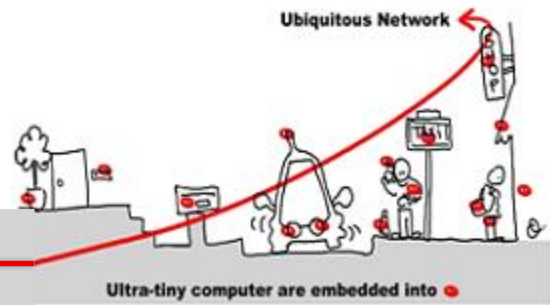
L2 = L1

L1 = I

O = L2

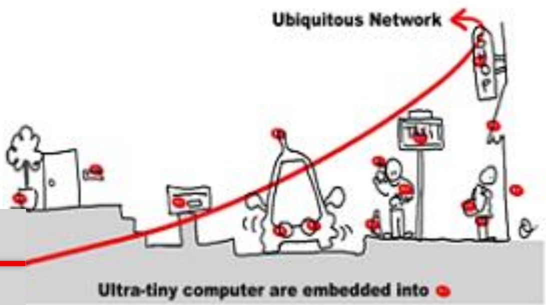
L2 = L1

LE Compilation



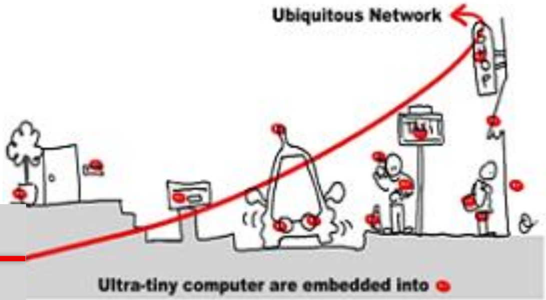
- Sorting algorithms:
  1. Apply CPM on dependency graphs of equation systems to compute ranges of evaluation levels for variables (efficient)
  2. apply fix point theory:
     - Compute variable evaluation levels as fix point of a monotonic increasing function
     - Uniqueness of fixpoints we can consider a global sorting as well as a local and separate sorting

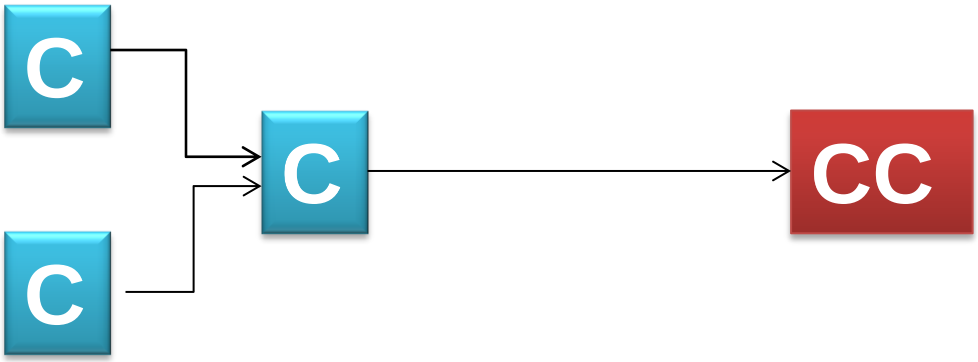
CLEM Simulation and Verification



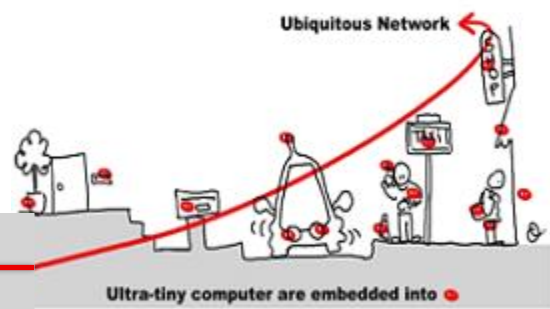
- Simulation:
  - Based on blif_simul an interpretor for blif code generated by CLEM
- Verification:
  1. NuSMV model checker (code generated)
  2. blif_check for small application

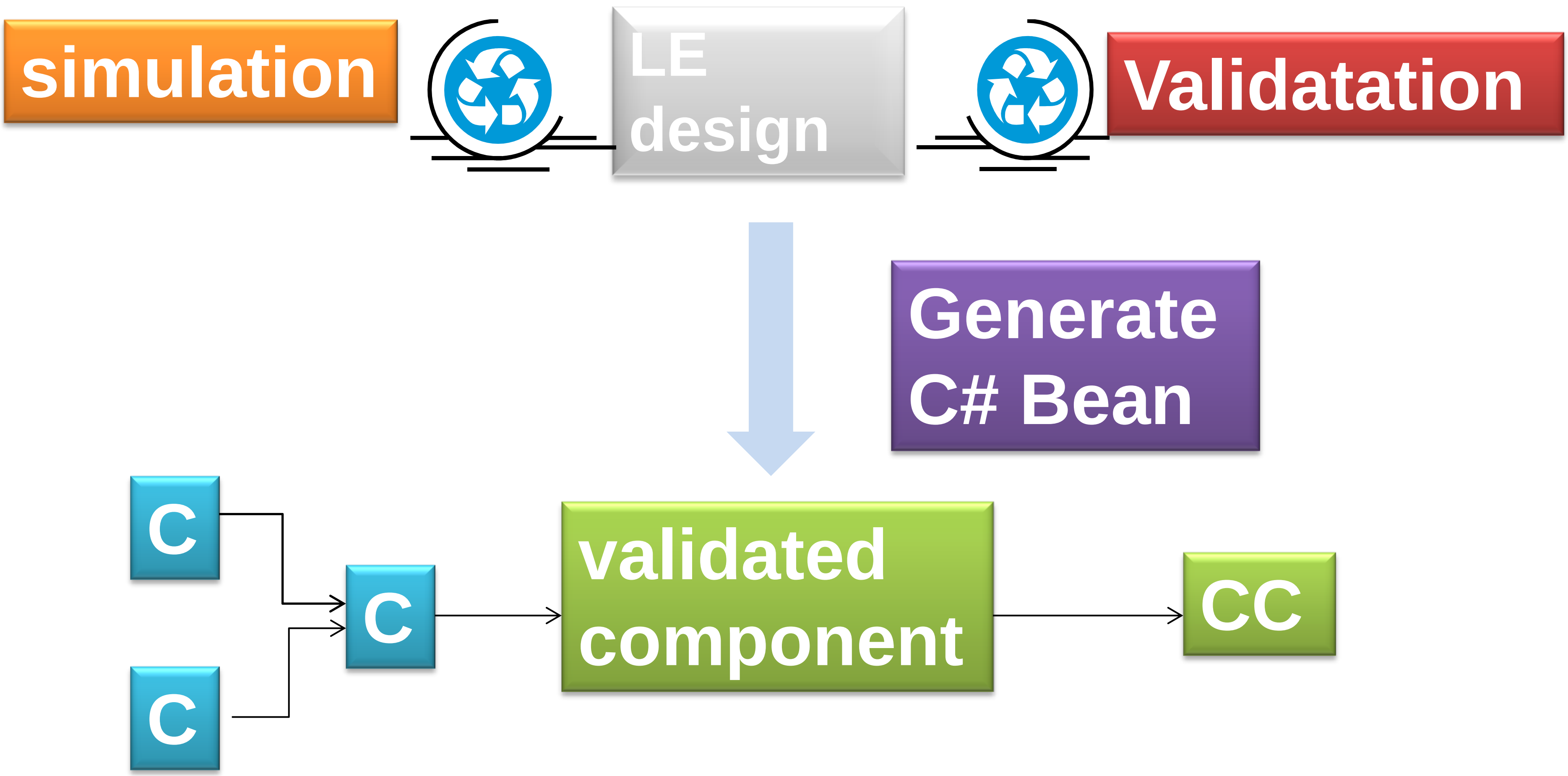
Critical Component Validation with CLEM



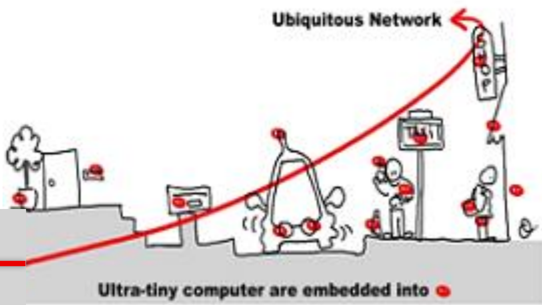


Critical Component Validation with CLEM



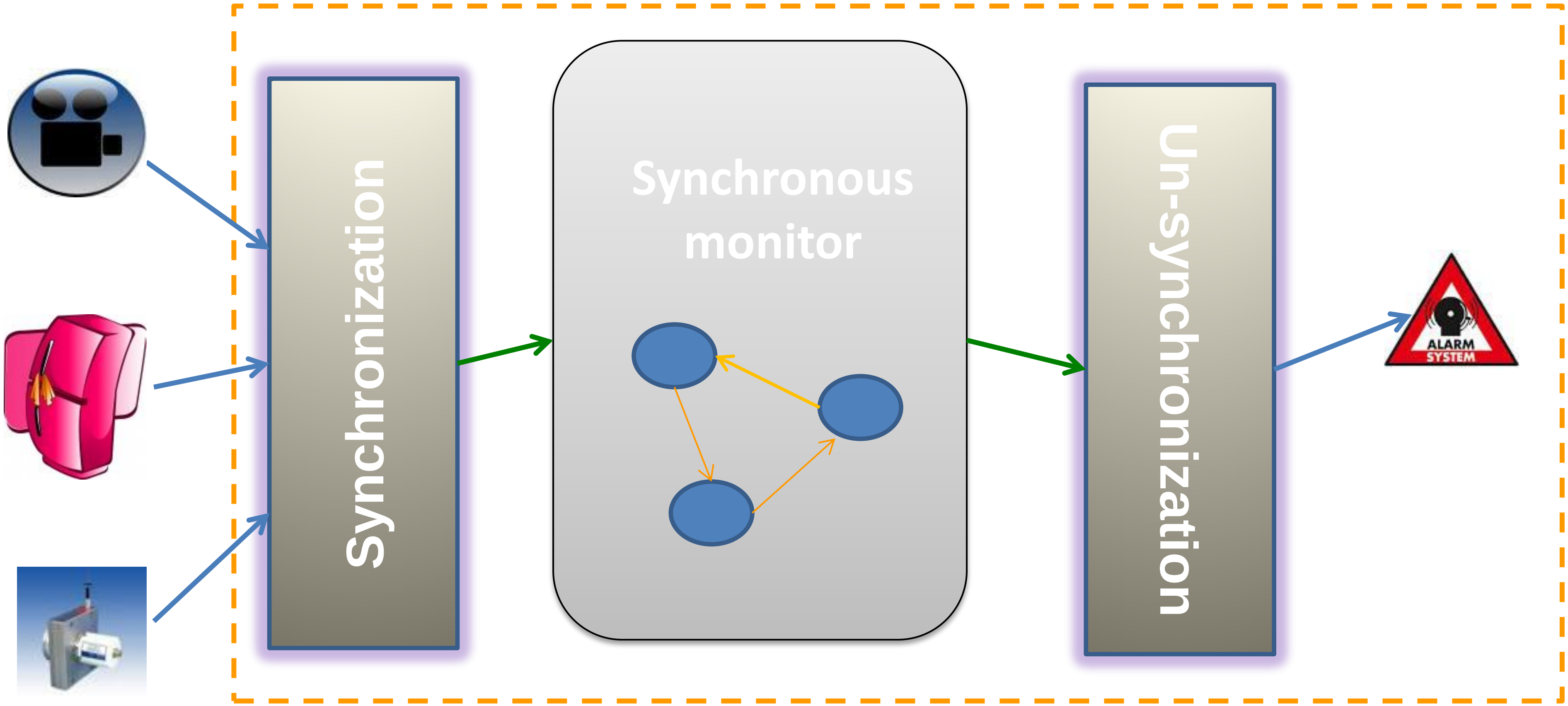


C# Bean Generation



- C# Bean implements synchronous monitor in Wcomp
- Communication is asynchronous in WComp
- $\Rightarrow$
  - need of a synchronizer to collect asynchronous events and build the logical event for the synchronous monitor
  - need for the reverse operation to plunge the outputs of the instant into asynchronous events

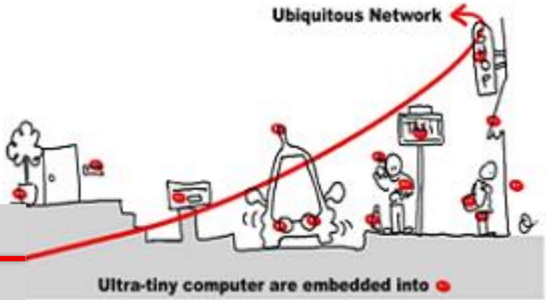
C# Bean Generation



asynchronous data

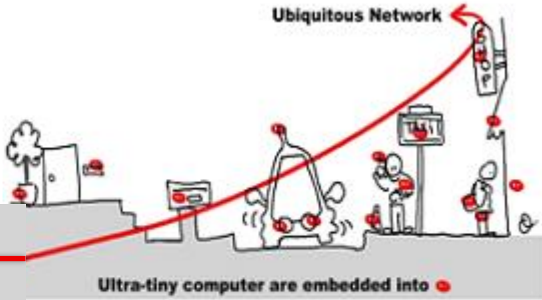
synchronous data

Asynchrony/Synchrony



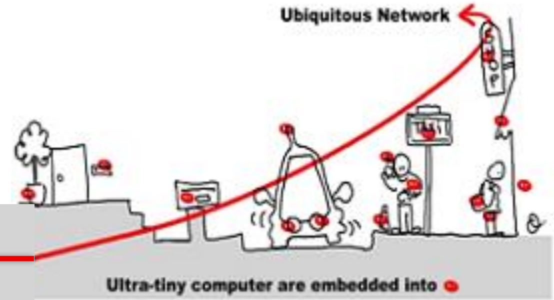
- Synchronization goal:
  1. generate the set of synchronous input events that characterizes the synchronous logical instant.
  2. Define an exchange format to allow communication between synchronous monitors and asynchronous components
- Un-synchronization goal:
  1. Generate the set of asynchronous output events from synchronous output events computed by the synchronous monitor.

Asynchrony/Synchrony

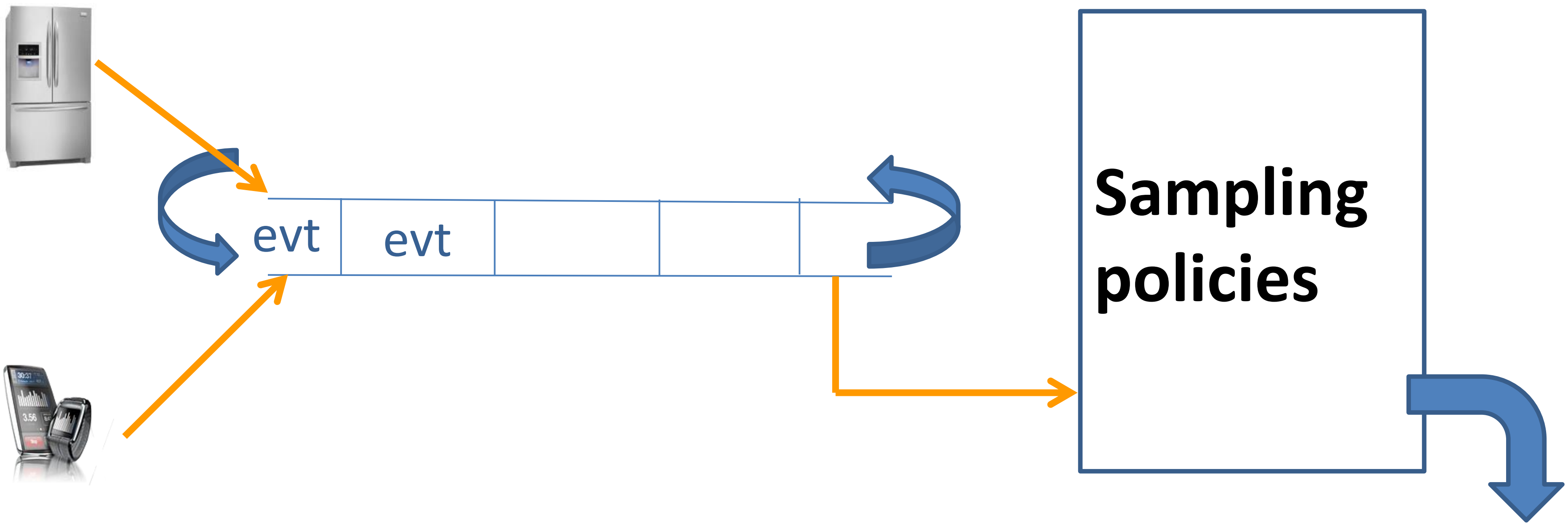


- How define the logical instant ?
  - The synchronization phase should be generic and allow to take into account several types of devices.
  - Introduction of a generic structure to represent events coming from different sensors:
    - name, presence, value type, value, elapsed time
    - apply several sampling policies : elapsed time, occurrence, average

Asynchrony/Synchrony

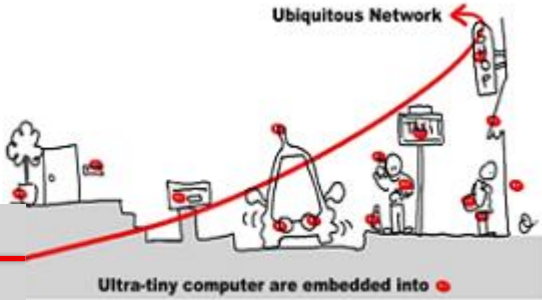


- How define the logical instant ?



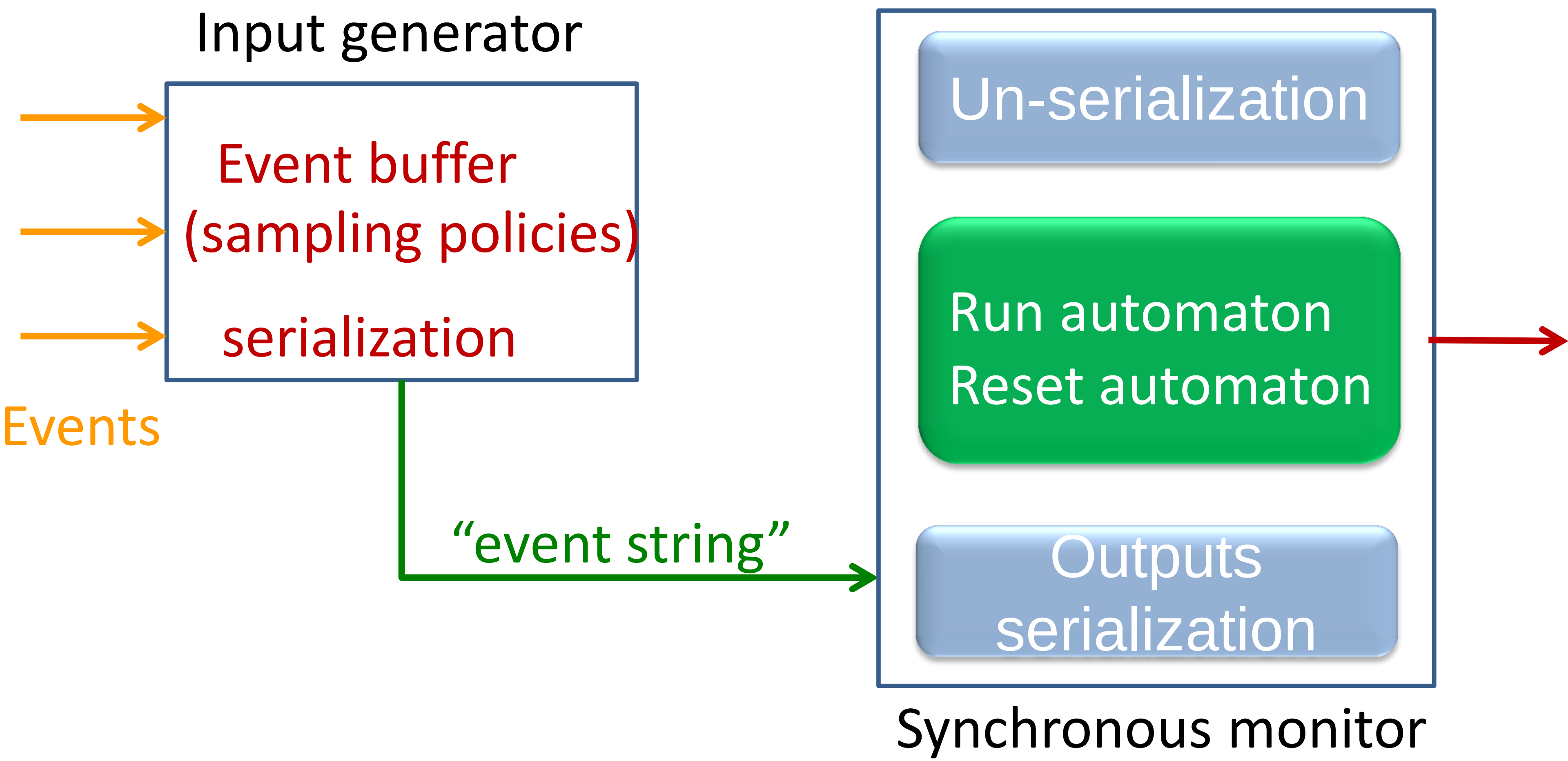
Synchronous instant

Asynchrony/Synchrony

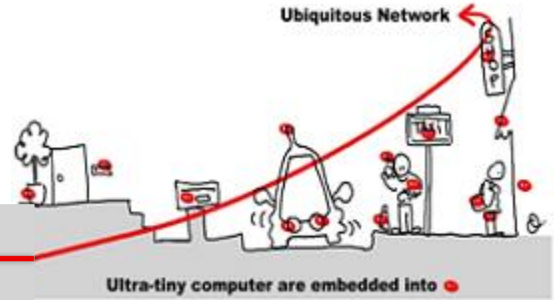


- Exchange format to get a means to establish communication between input methods and output events in Wcomp.
- ⇒ Serialization/Deserialization of events. Two serialization proposals:
  1. “ [<name> = <occurrence>,[<type>, <valeur>]?;]+”
     - a = false; b = true; v = true, int, 7;”
  2. [“<name>”<occurrence> <type> <valeur>”]+
     - “a false” “b true” “v true int 7”

Asynchrony/Synchrony



Asynchrony/Synchrony



Synchronous monitor

- Un-serialization
- Run automaton
  Reset automaton
- Outputs serialization

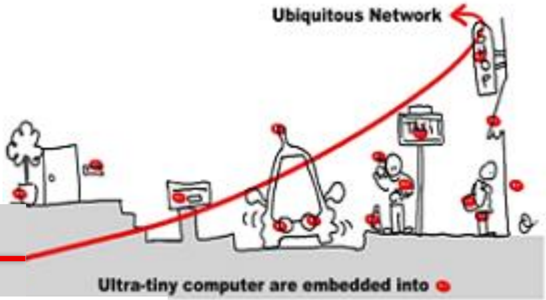
→

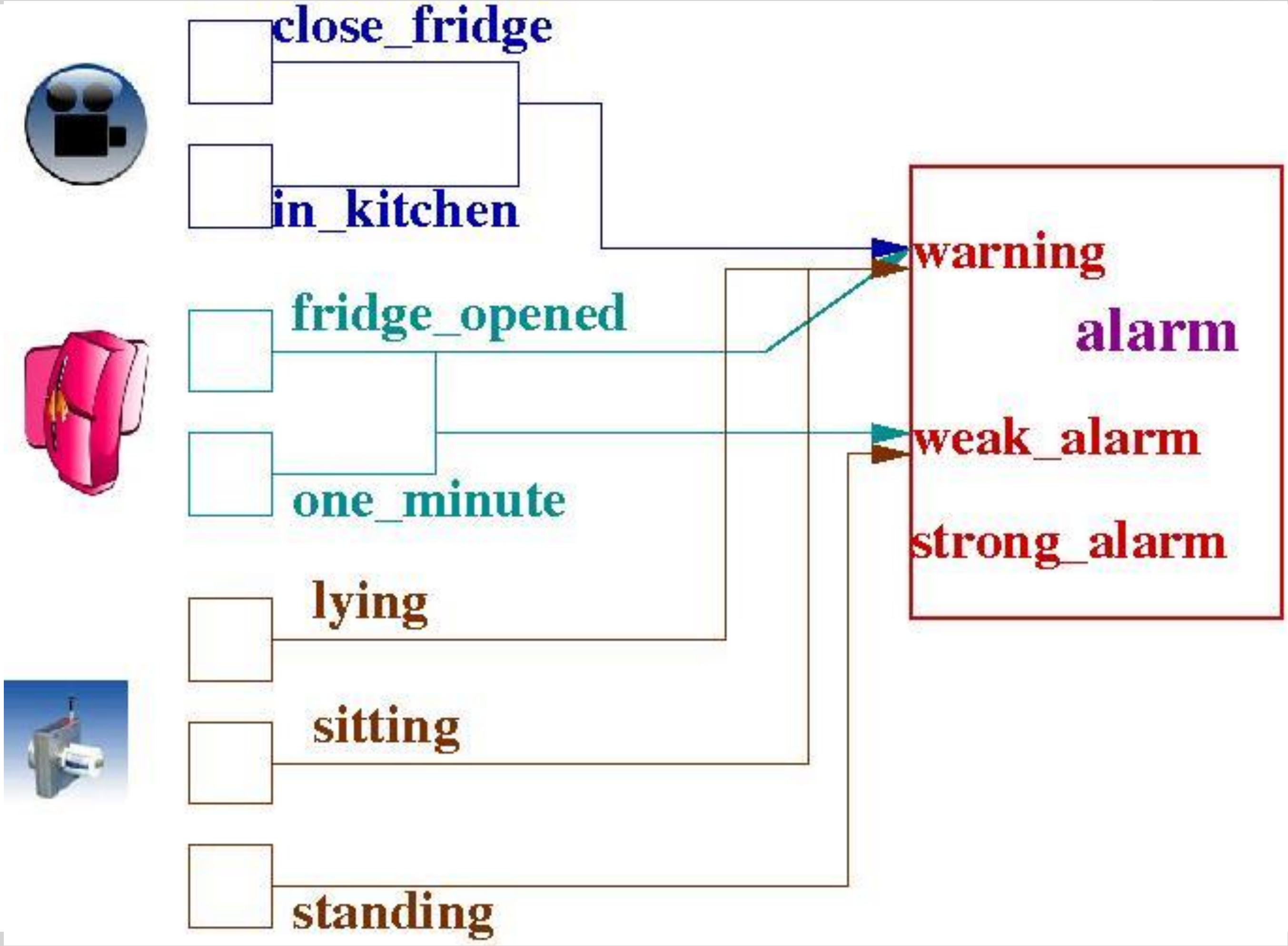
Ouputs generator

- Un-serialization (string → events)
- Sending Policies

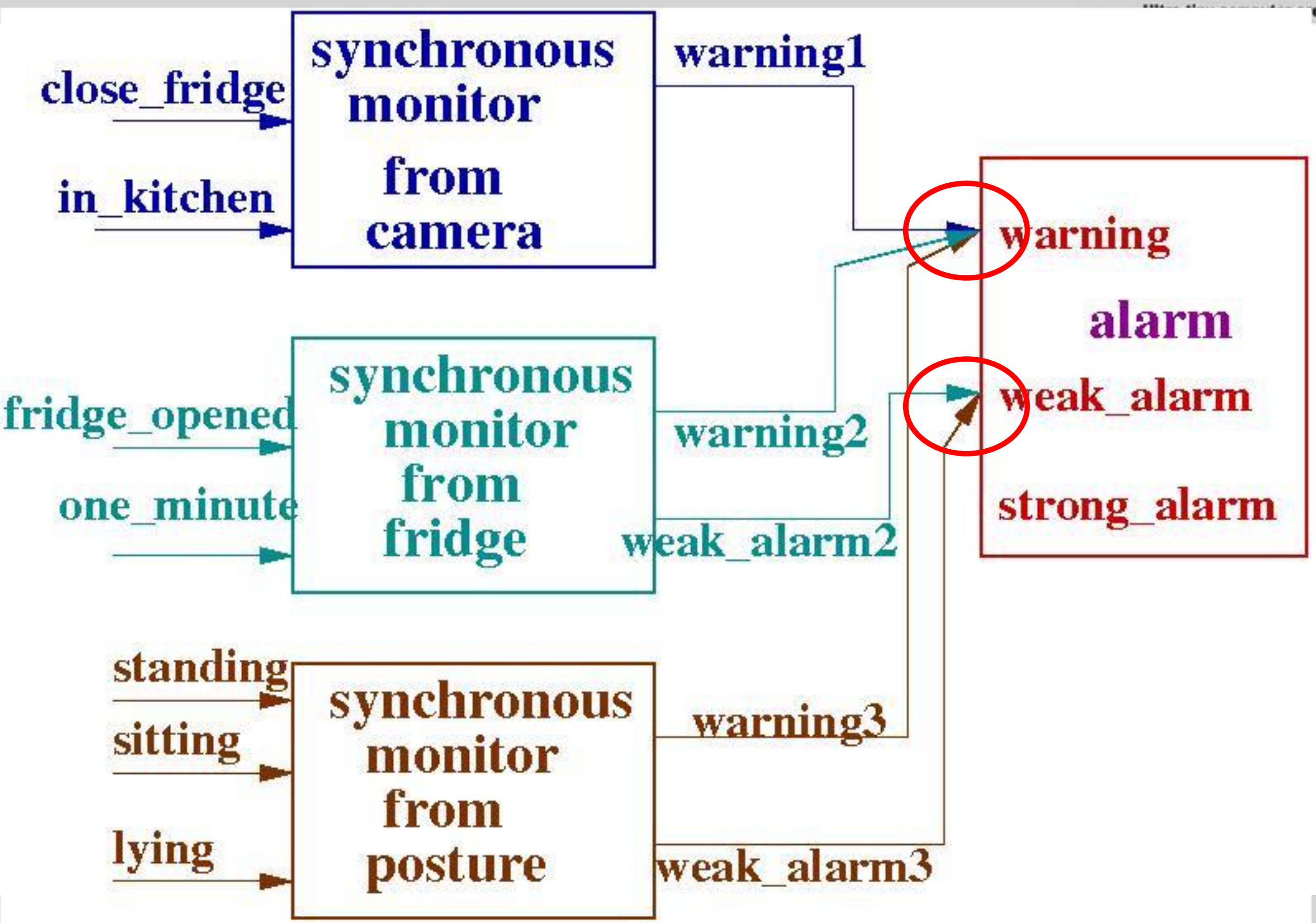
→ Asynchronous events

Synchronous Monitor Composition

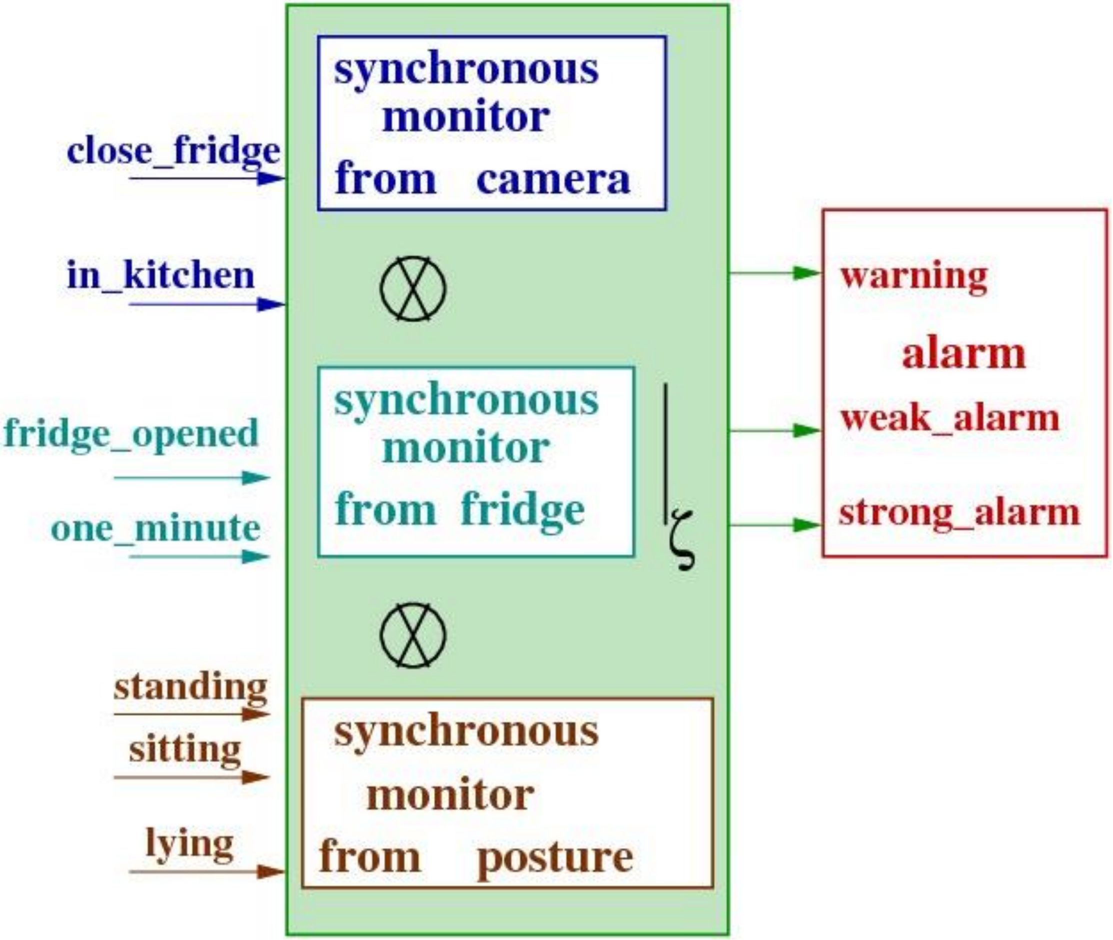




Synchronous Monitor Composition



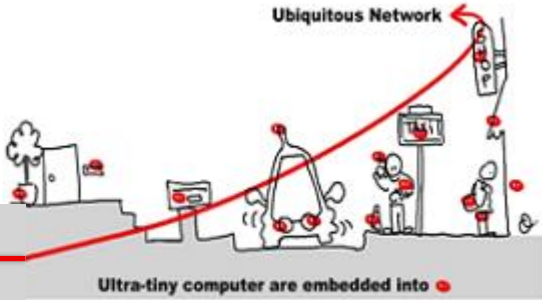
Synchronous Monitor Composition



Solution:

composition under constraint : $\otimes|_{\zeta}$

Synchronous Monitor Composition



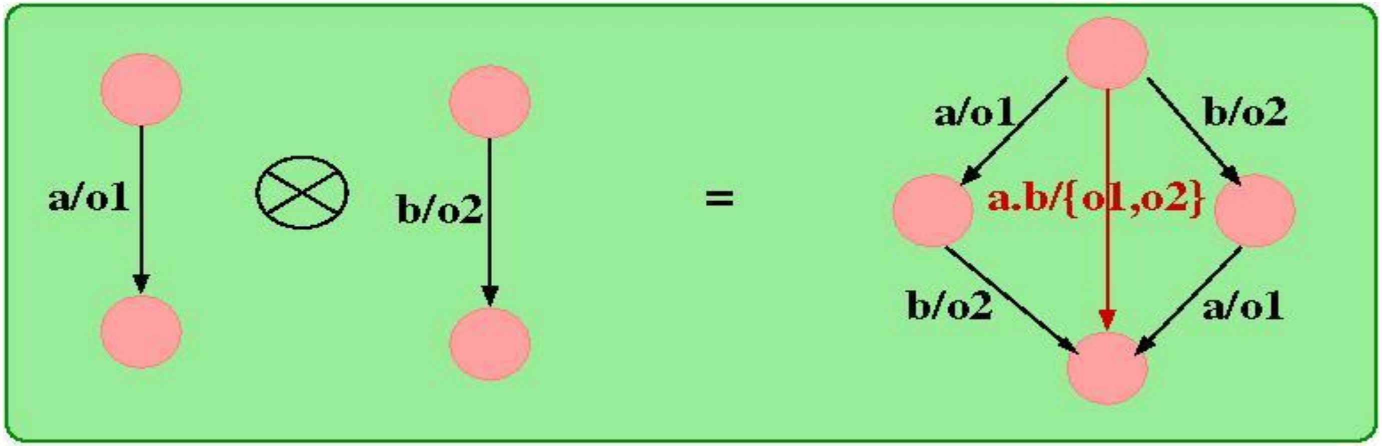
$\otimes|_{\zeta}$ = synchronous product + constraint function

The constraint function tells us how multiple accesses are combined

Property : $\otimes|_{\zeta}$ preserves safety property:

M_1 verifies Φ then $M_1 \otimes|_{\zeta} M_2$ verifies Φ also

Synchronous Monitor Composition

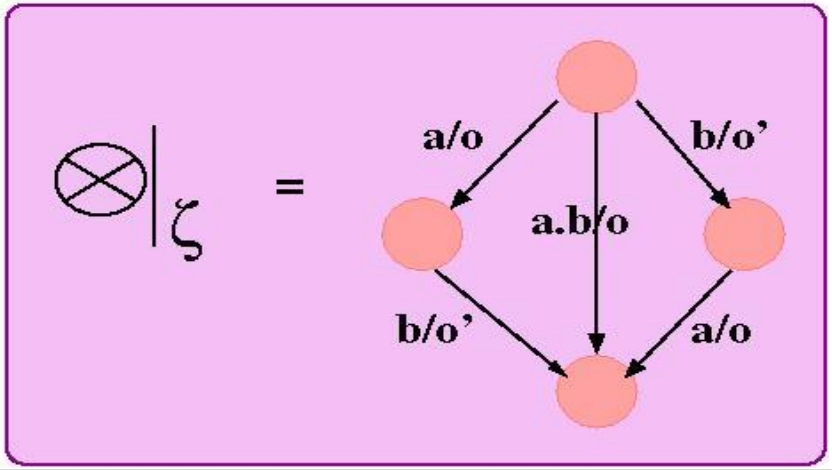


$O = \{o, o'\}$

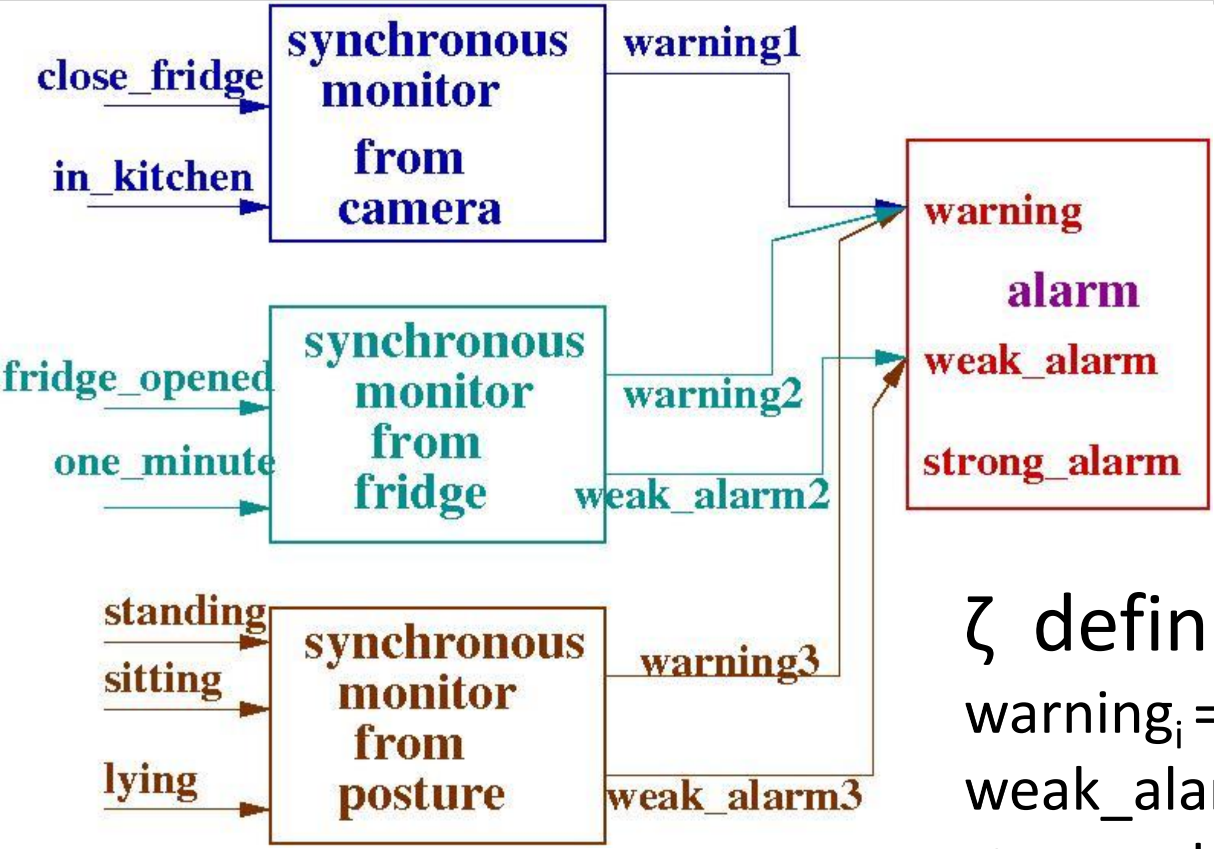
ζ: o1 → o

o2 → o'

{o1,o2} → o



Synchronous Monitor Composition

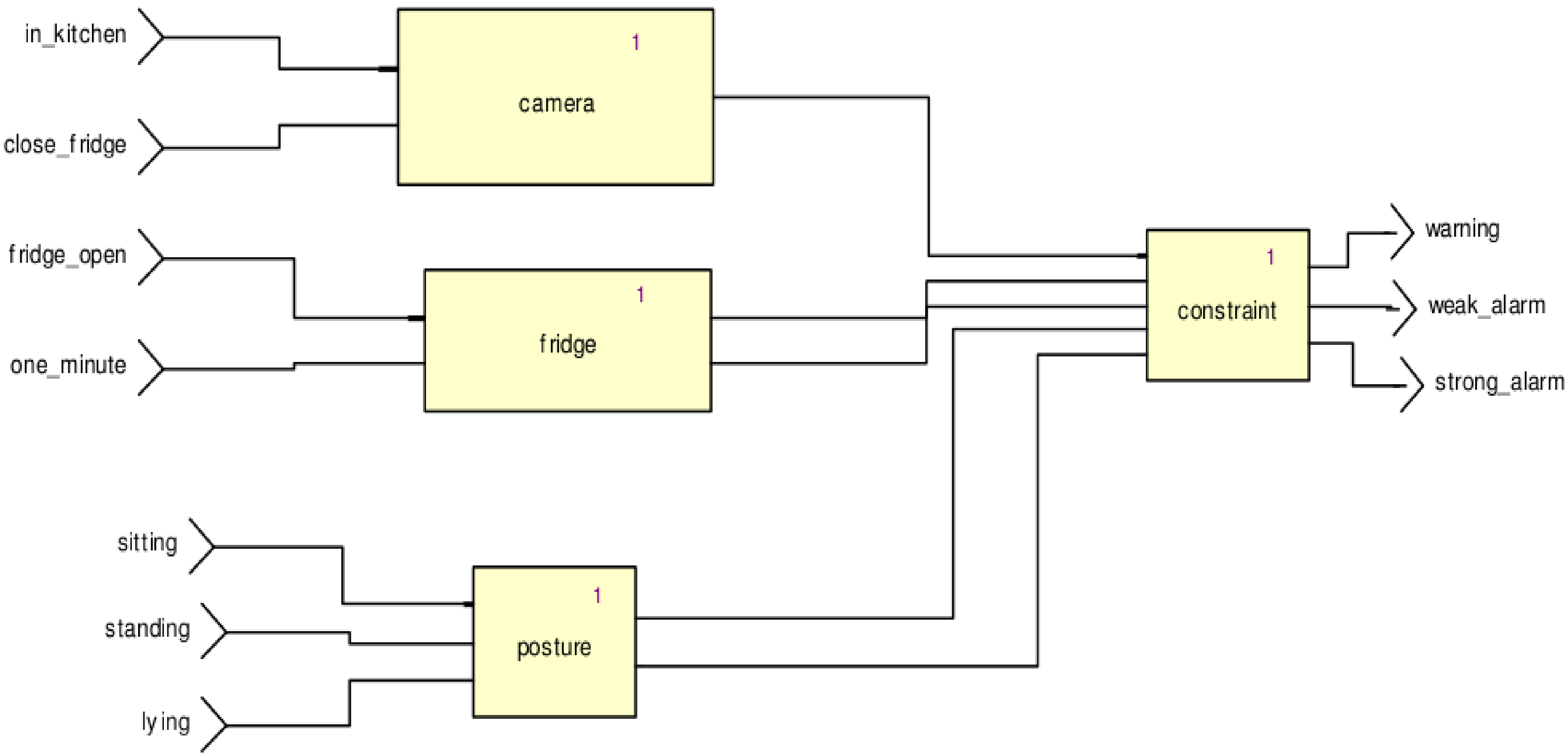


ζ definition:

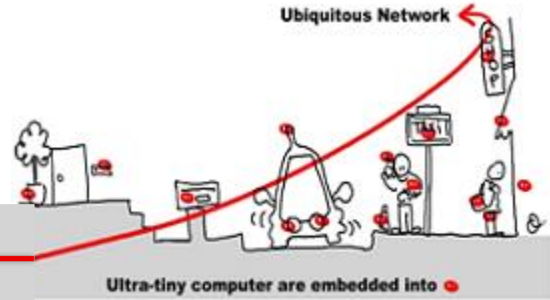
$warning_i$ = warning

$weak\_alarm_2$ & weak_alarm3 = strong_alarm

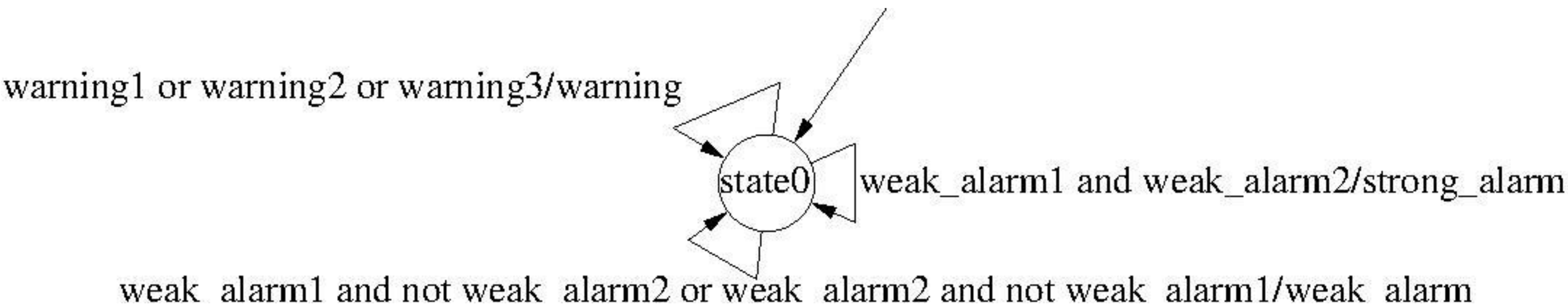
Synchronous Monitor Composition



Synchronous Monitor Composition

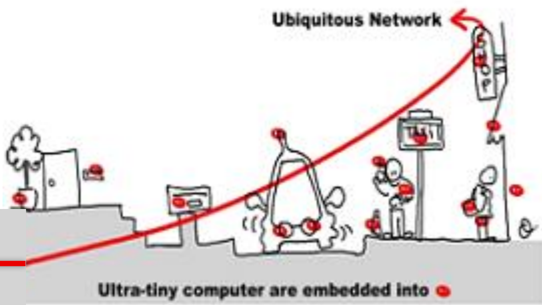


Explicit Mealy machine:



weak_alarm$_2$ & weak_alarm$_3$ implies strong_alarm

Synchronous Monitor Composition

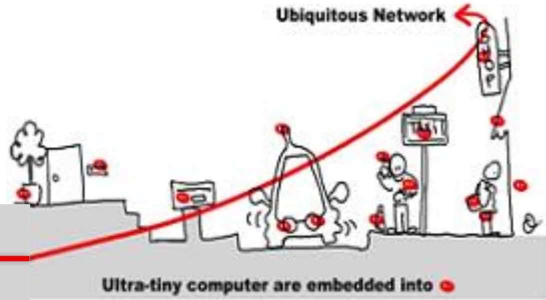


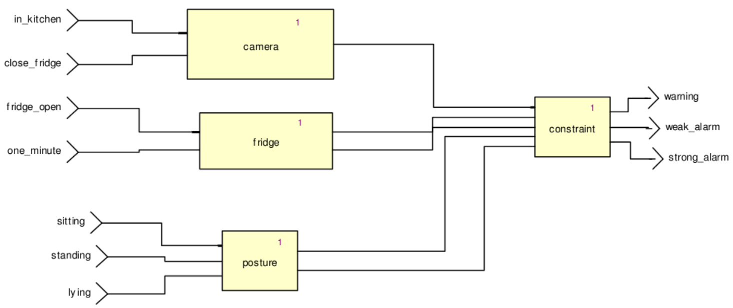
Implicit Mealy machine:

module **constraint**:
Input: warning1, warning2, warning3, weak_alarrm1, weak_alarm2;
Output: warning, weak_alarm, strong_alarm;
Mealy Machine
warning = warning1 or warning2 or warning3;
weak_alarm = weak_alarm1 and not weak_alarm2 or
weak_alarm2 and not weak_alarm1
strong_alarm = weak_alarm1 and waek_alarm2
end

weak_alarm$_2$ & weak_alarm$_3$ implies strong_alarm

Use case Implementation in WComp





validated alarm Bean