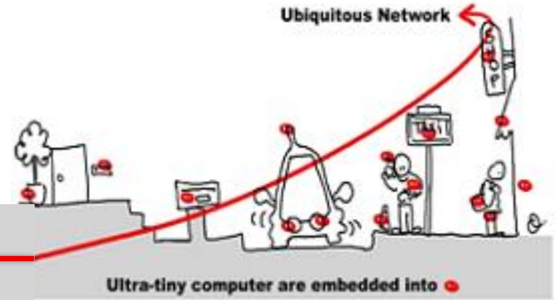


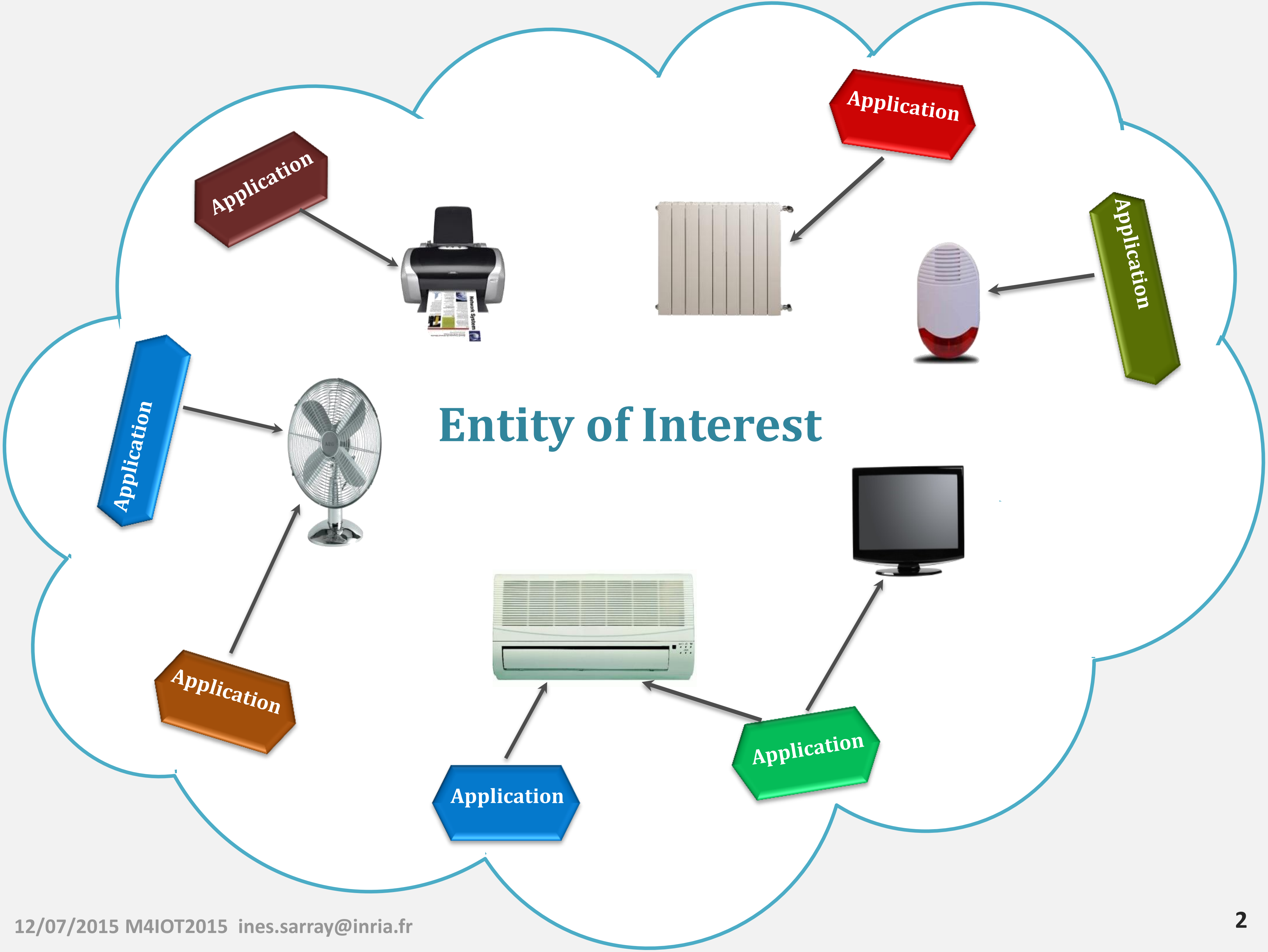
Safety in Middleware for IoT

Annie Ressouche

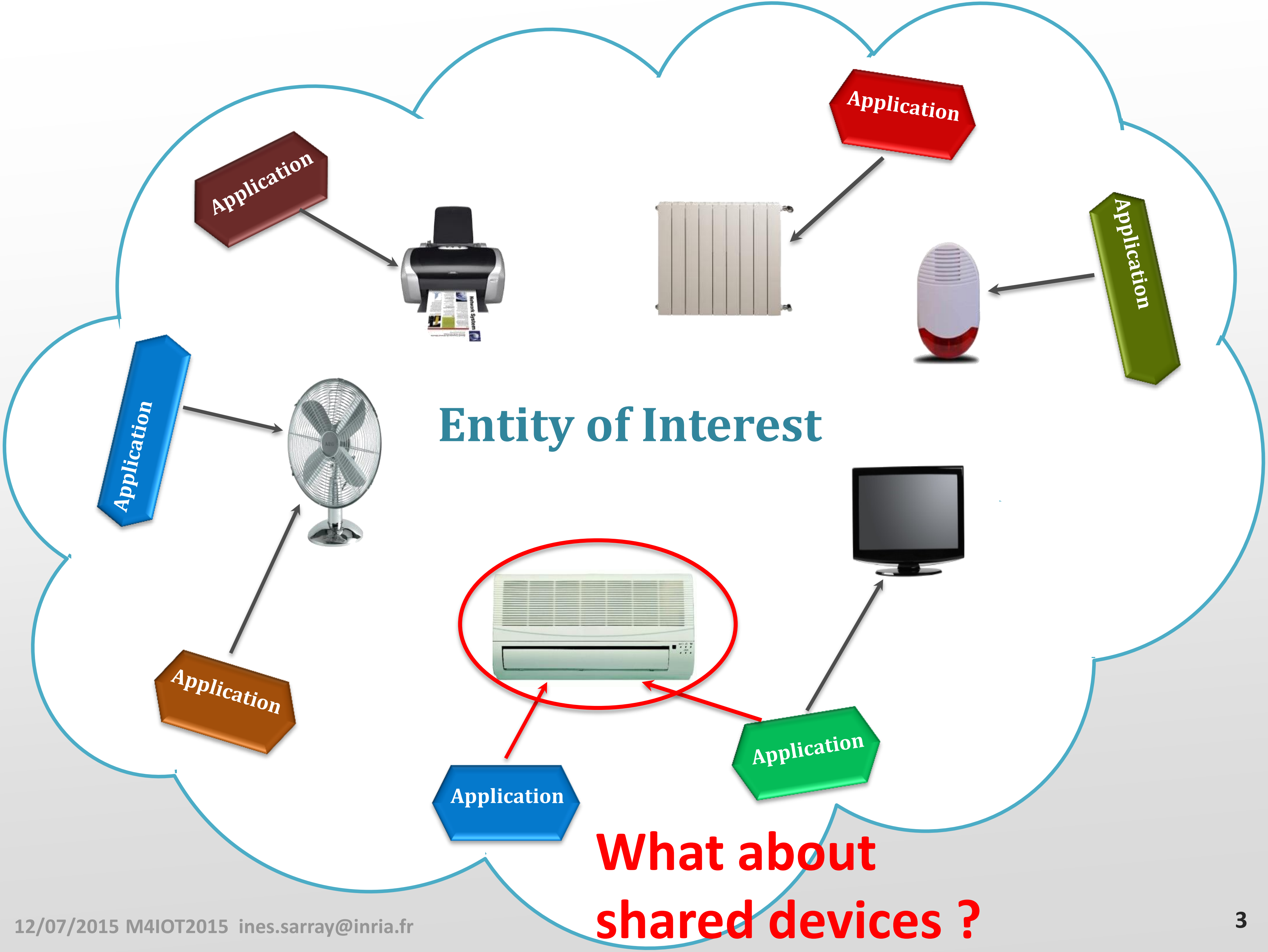
Inria-sam (stars)

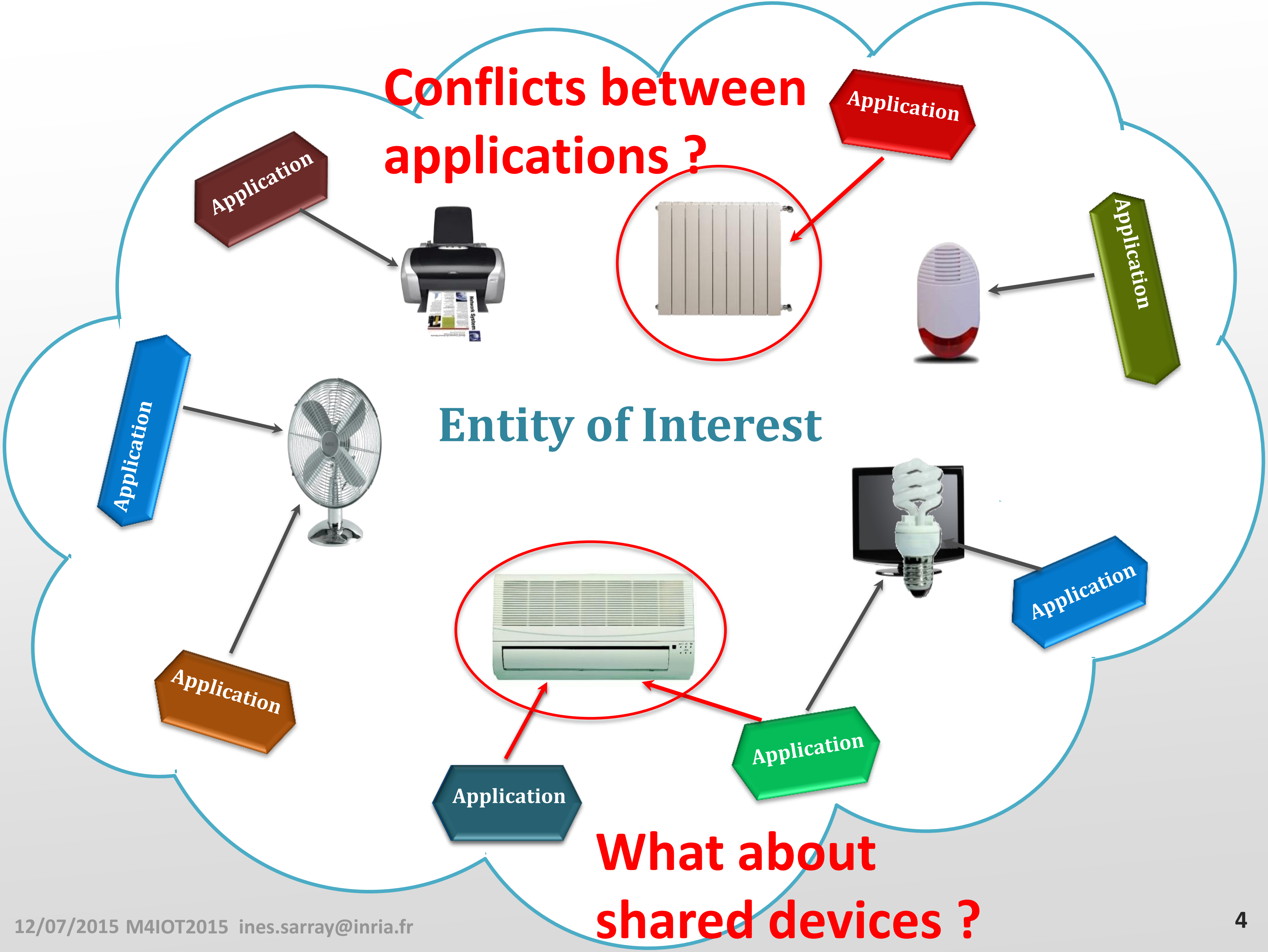
annie.ressouche@inria.fr

http://www-sop.inria.fr/members/Annie.Ressouche/teaching.html

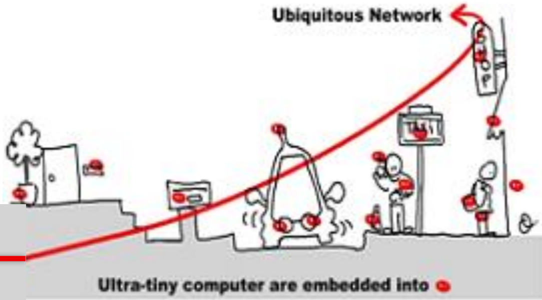


12/07/2015 M4IOT2015 ines.sarray@inria.fr


Application
Application
Application
Application
Entity of Interest
Application
Application
Application
Application
What about shared devices ?

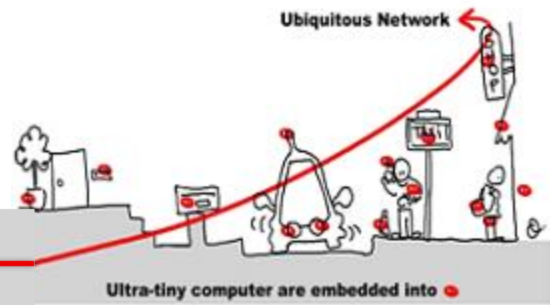

Conflicts between applications ?
Application
Application
Application
Application
Entity of Interest
Application
Application
Application
Application
Application
What about shared devices ?

Introduction



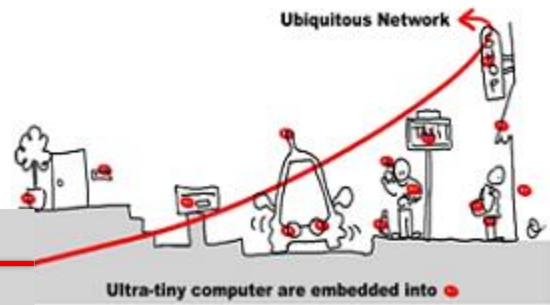
- How to maintain consistency in spite of concurrent accesses by multiple services and multiple applications to a common Entity of Interest ?
- How to deal with dynamic context changes ?
- Solution: apply general techniques used to develop critical software

Outline



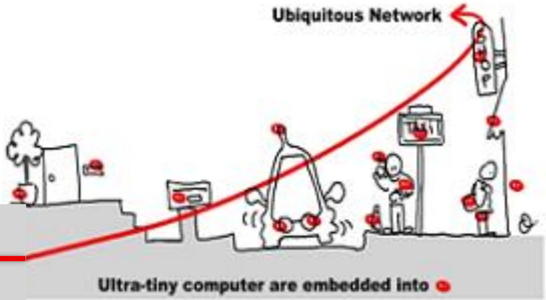
1. Critical system validation
2. Model-checking solution
    1. Model specification
    2. Model-checking techniques
3. Application to middleware for IoT
    1. Introduction in middleware design of synchronous components to allow validation
    2. Synchronous/asynchronous issue

Outline



1. Critical system validation
2. Model-checking solution
    1. Model specification
    2. Model-checking techniques
3. Application to component based adaptive middleware
    1. Introduction in middleware design of synchronous components to allow validation
    2. Synchronous/asynchronous issue

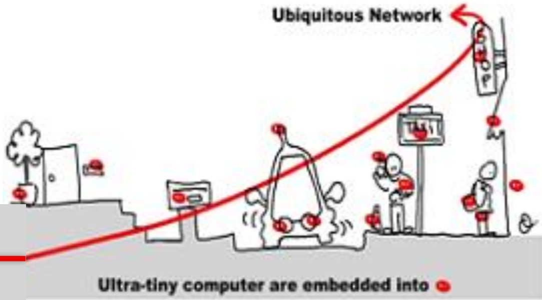
Critical Software



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

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

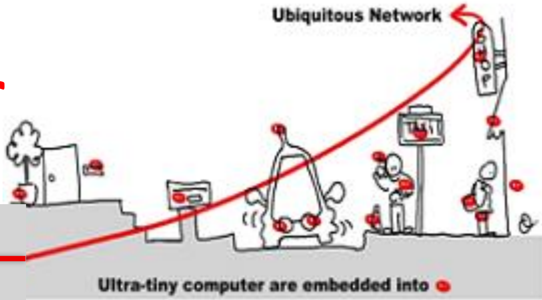
…



Critical Software

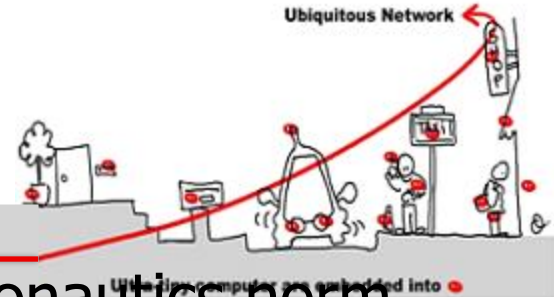
- In addition, other consequences are relevant to determine the critical aspect of software:
  - Financial aspect
    - Loosing equipment, bug correction
    - Equipment callback (automotive)
  - Bad advertising

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

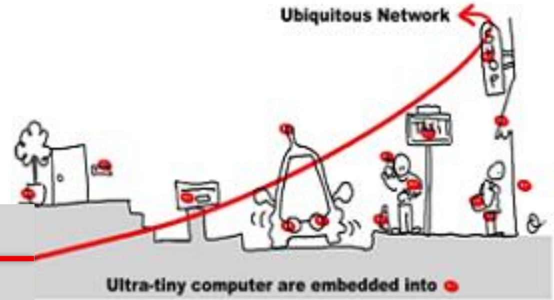


Example of the aeronautics norm DO178B:

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

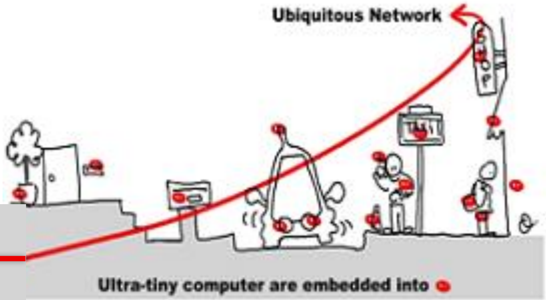
A	Catastrophic (human life loss)
B	Dangerous (serious injuries, loss of goods)
C	Major (failure or loss of the system)
D	Minor (without consequence on the system)
E	Without effect

Software Classification

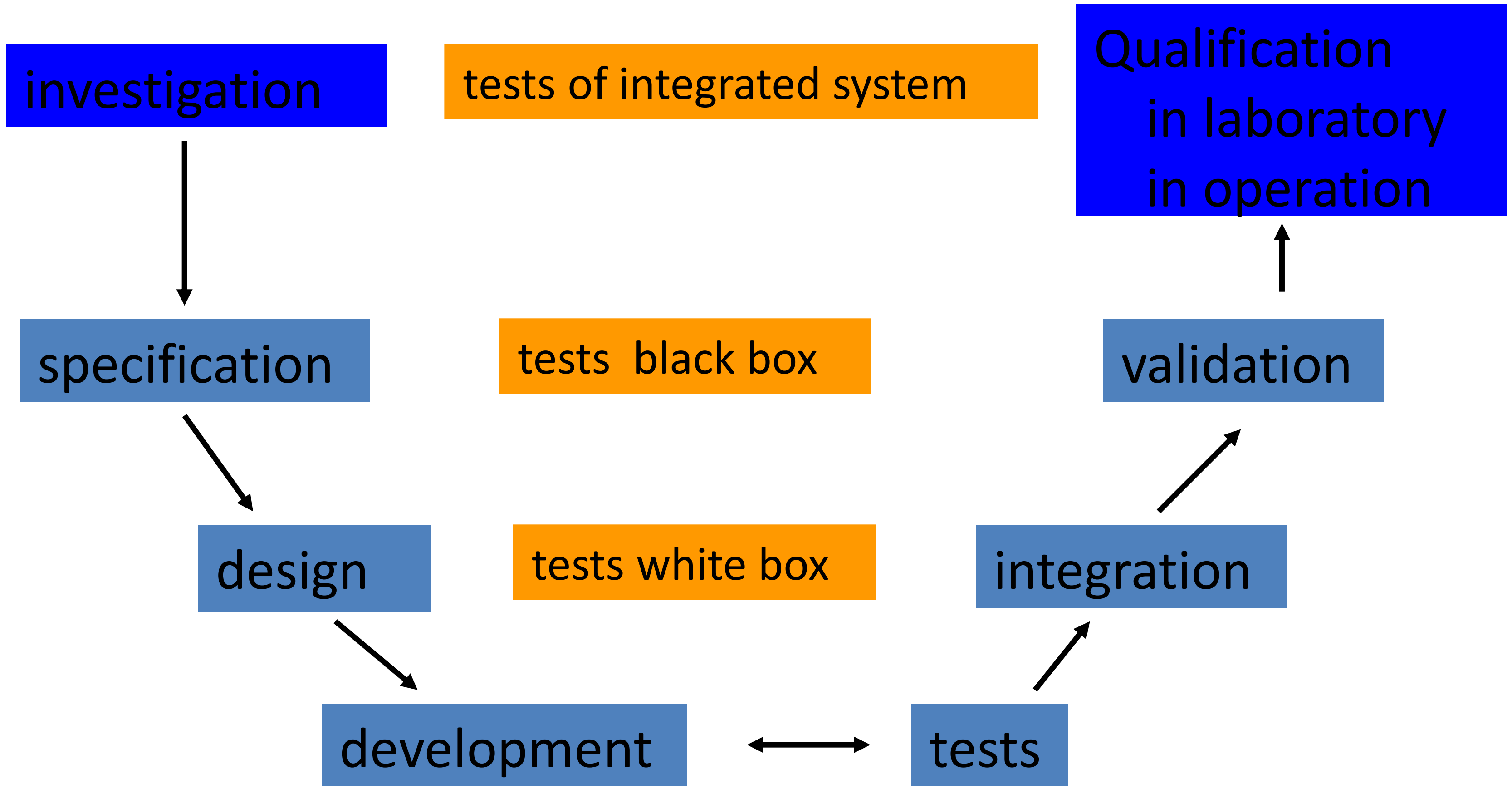


Minor			acceptable situation	
Major				
Dangerous	Unacceptable situation			
catastrophic	10^{-3} / hour	10^{-6} / hour	10^{-9}/hour	10^{-12}/hour
probabilities	probable	rare	very rare	very improbable

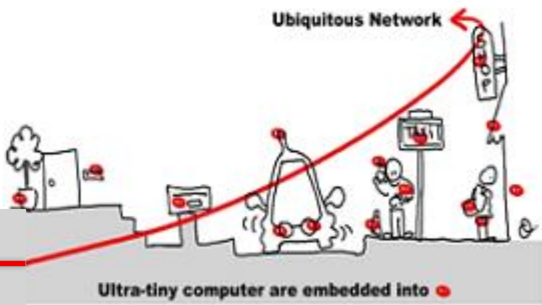
How Develop critical software ?



Classical Development U Cycle

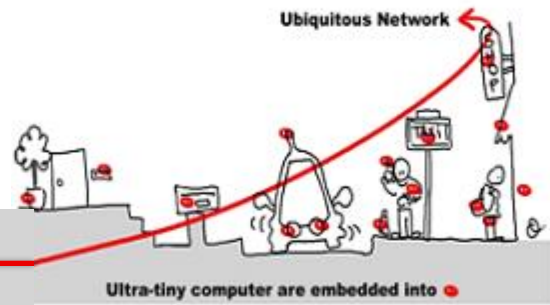


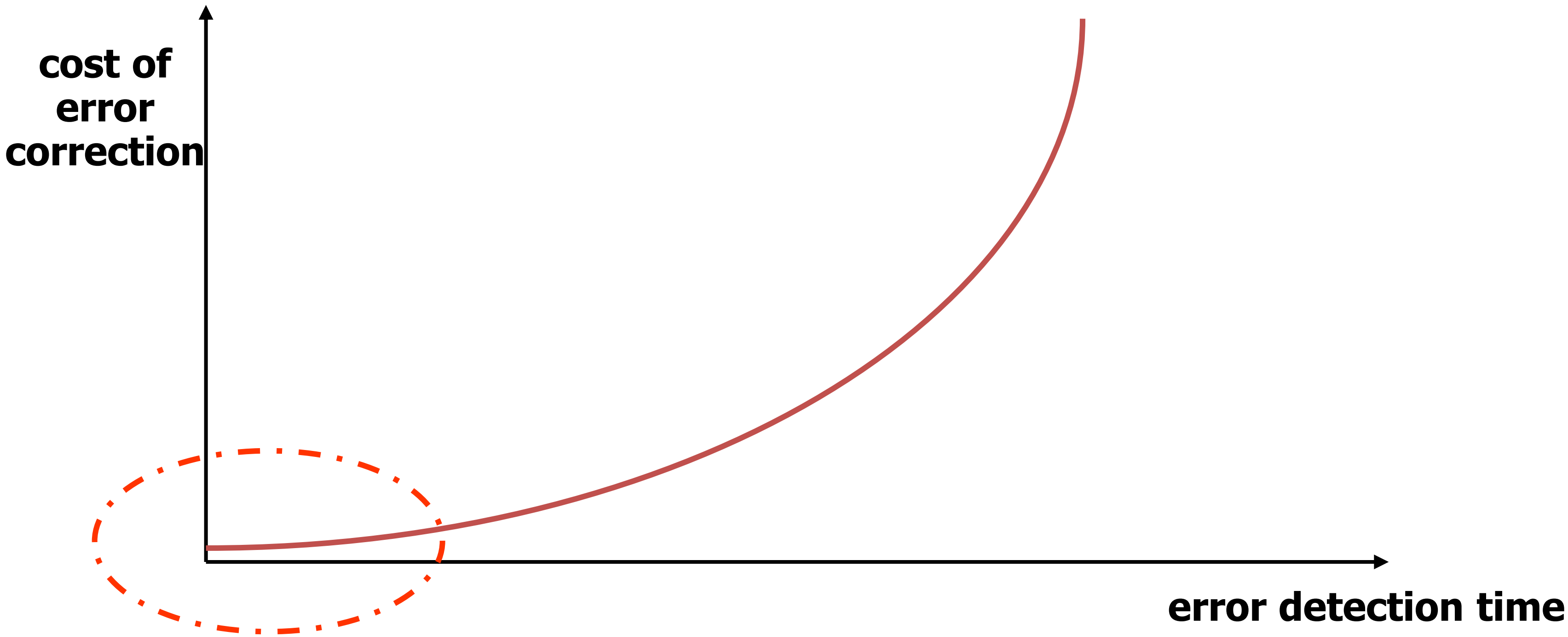
How Develop Critical Software ?



- 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

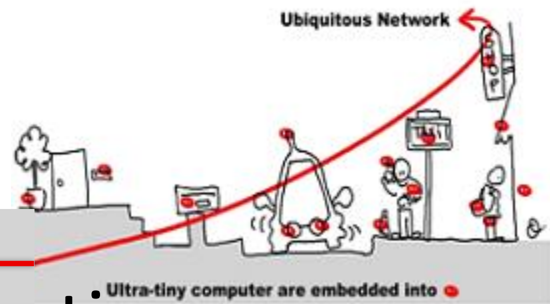




Put the effort on the upstream phase

development based on models

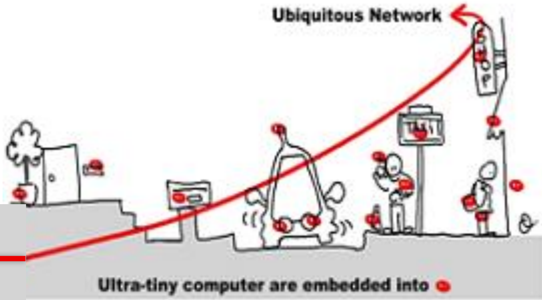
How Develop Critical Software ?



- 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

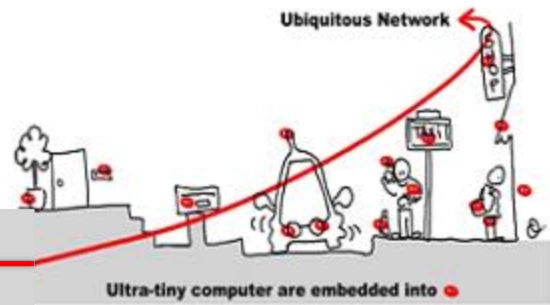
⇒ Formal model usage

Critical Software Specification



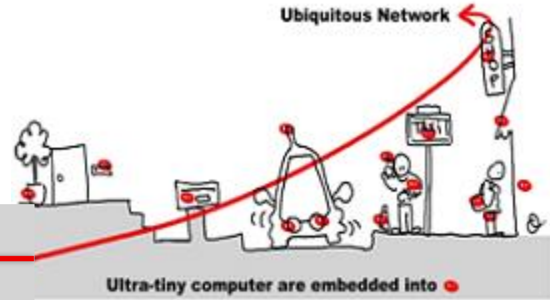
- First Goal: must yield a formal description of the application needs:
  - Standard to allowing communication between computer science engineers and non computer science ones
  - General enough to allow different kinds of application:
    - Synchronous (and/or)
    - Asynchronous (and/or)
    - Algorithmic

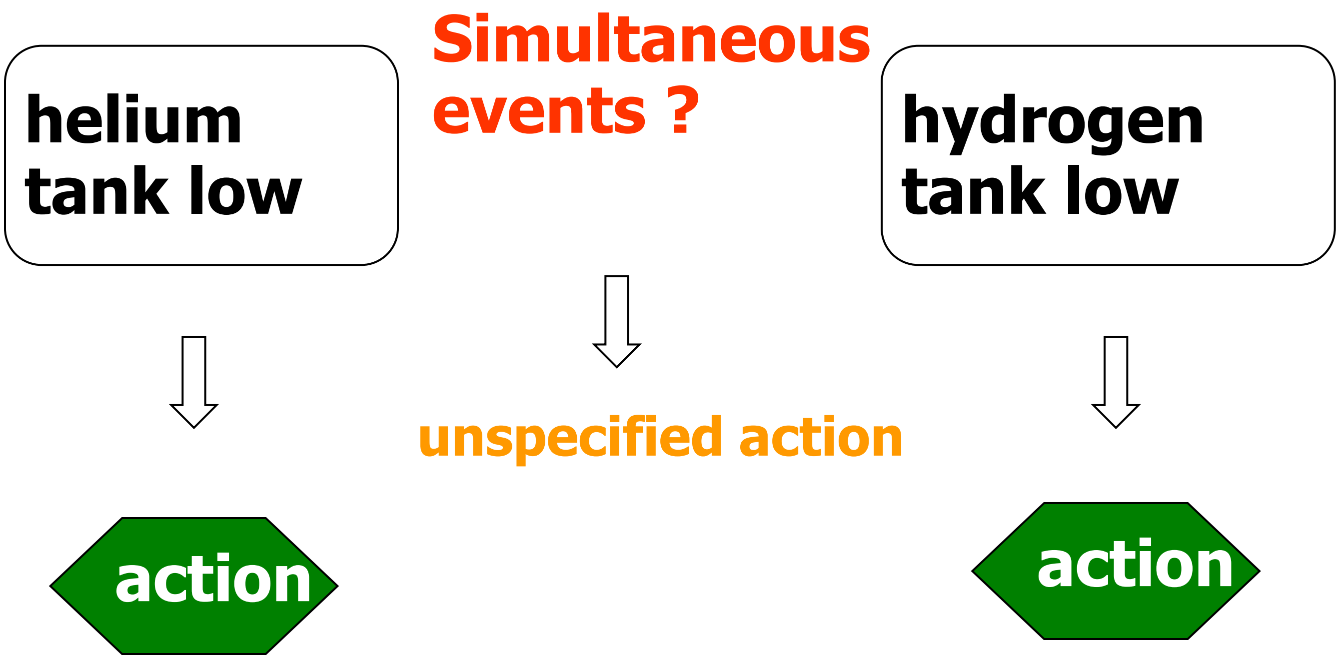
Critical Software Specification



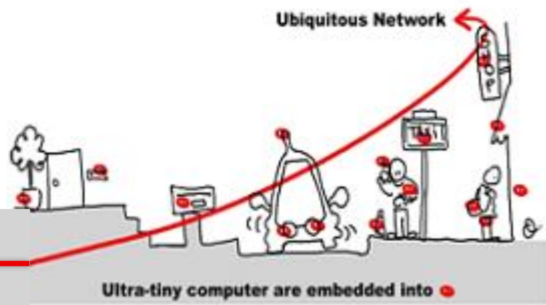
- Second Goal: allowing errors detection carried out upstream:
  - Validation of the specification:
    - Coherency
    - Completeness
    - Proofs
  - Test
    - Quick prototype development
    - Specification simulation

Critical Software Specification



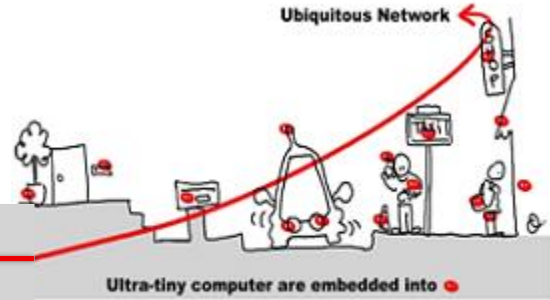


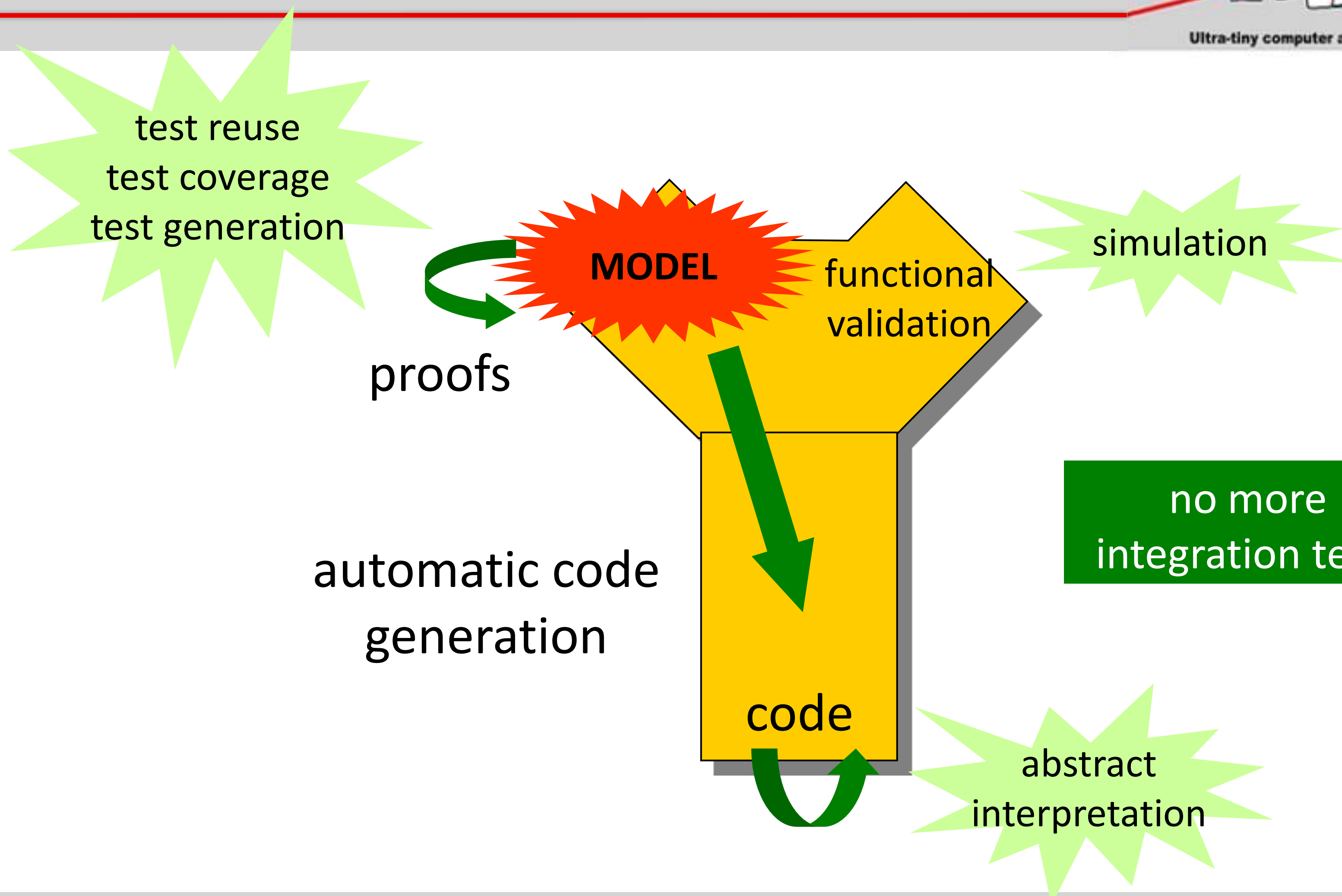
Critical Software Specification



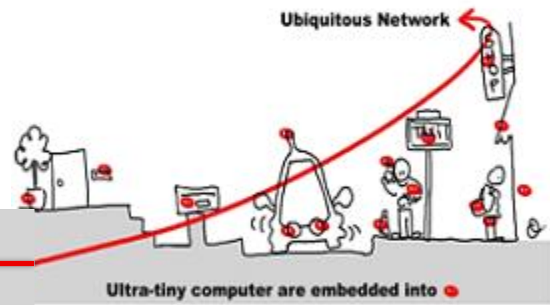
- Third goal: make easier the transition from specification to design (refinement)
  - – Reuse of specification simulation tests
  - – Formalization of design
  - – Code generation
    - Sequential/distributed
    - Toward a target language
    - Embedded/qualified code

How Develop Critical Software



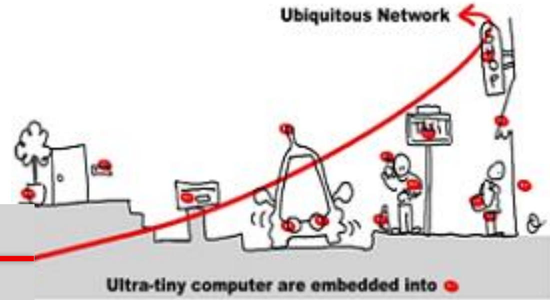


Critical Software Validation



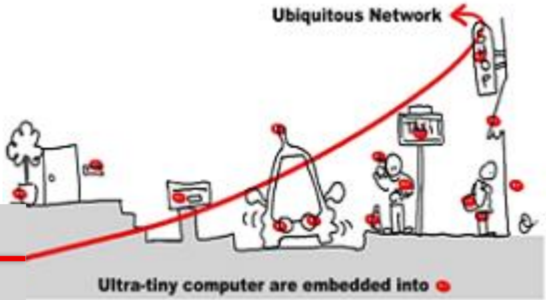
- What is a correct software?
  - No execution errors, time constraints respected, compliance of results.
- Solutions:
  - At model level :
    - Simulation
    - Formal proofs
  - At implementation level:
    - Test
    - Abstract interpretation

Validation Methods



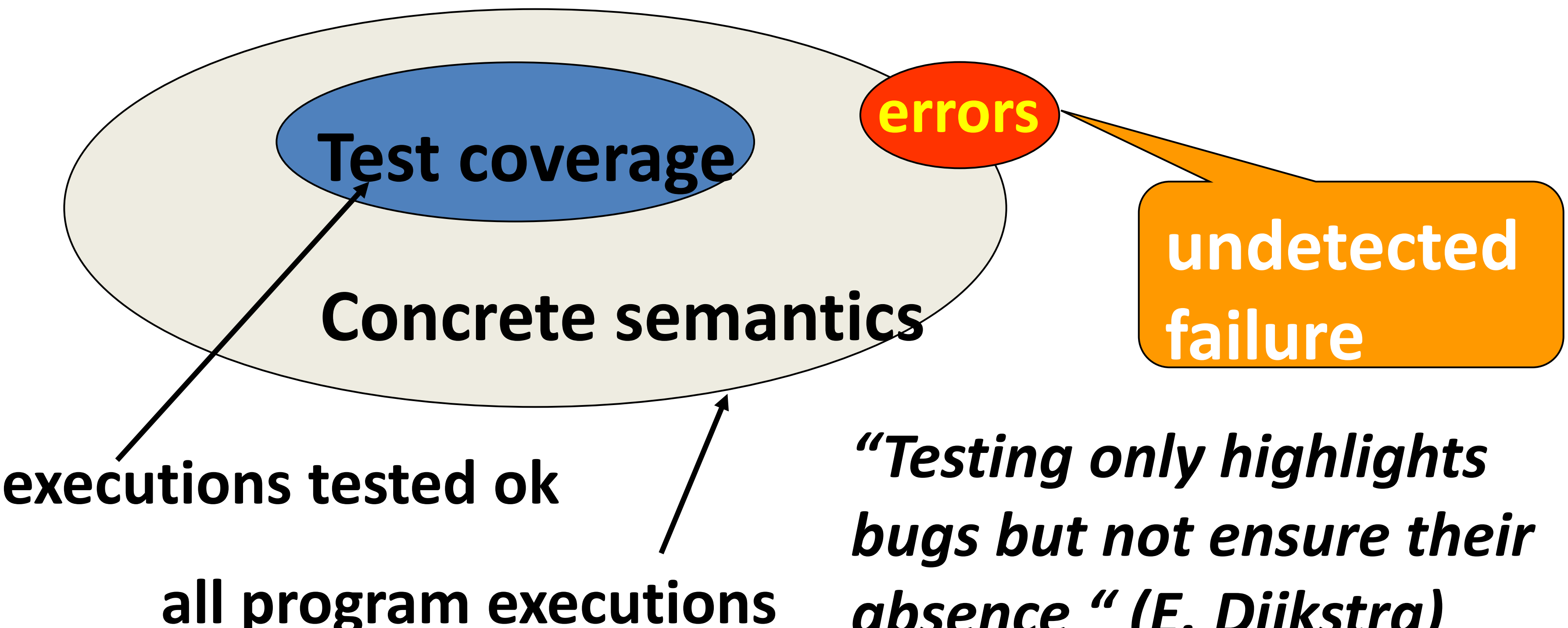
- Testing
  - Run the program on set of inputs and check the results
- Static Analysis
  - Examine the source code to increase confidence that it works as intended
- Formal Verification
  - Argue formally that the application always works as intended

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

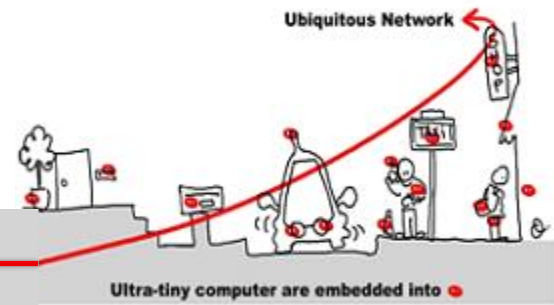


"Testing only highlights bugs but not ensure their absence " (E. Dijkstra)

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: }
```

$$x1 = [1,1]$$

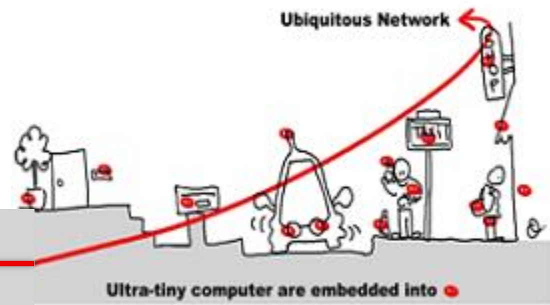
$$x2 = x1 \cup x3 \cap [-\infty, 999]$$

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

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

Abstract interpretation theory ⇒ values are fix point equation solutions.

Formal Verification

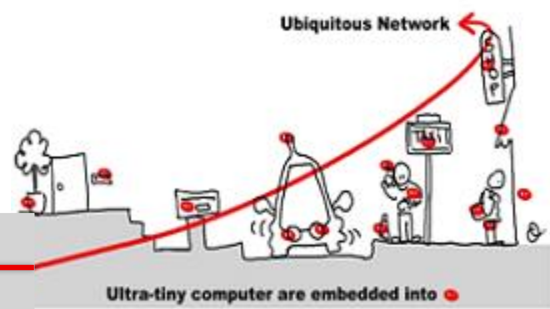


- What about functional validation ?
  - Does the program compute the expected outputs?
  - Respect of time constraints (temporal properties)
  - Intuitive partition of temporal properties:
    - Safety properties: something bad never happens
    - Liveness properties: something good eventually happens

Safety and Liveness Properties

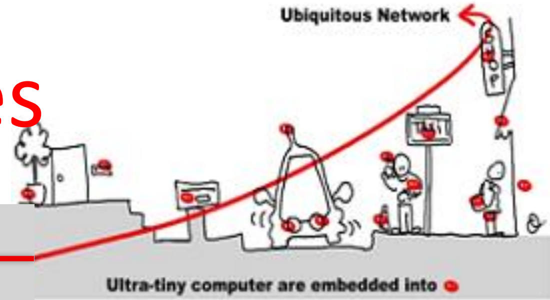
- Example: train timetable
  - Count the difference between marks and seconds
  - Decide when the train is ontime, late, early
    - **ontime** : difference = 0
    - **late** : difference > 3 and it was ontime before or difference > 1 and it was already late before
    - **early** : difference < -3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



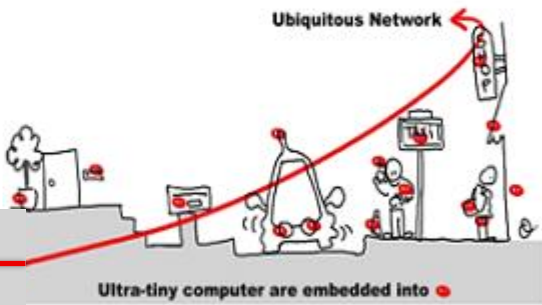
Some properties:

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

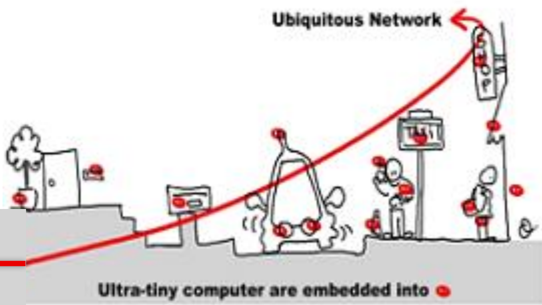
Property 4 : liveness (refer to unbound future)

Outline



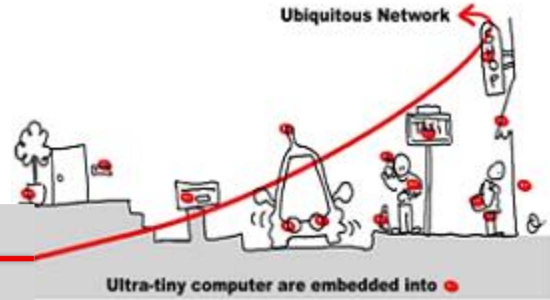
1. Critical system validation
2. Model-checking solution
   1. Model specification
   2. Model-checking techniques
3. Application to middleware for IoT
   1. Introduction in middleware design of synchronous components to allow validation
   2. Synchronous/asynchronous issue

Safety and Liveness Properties Checking



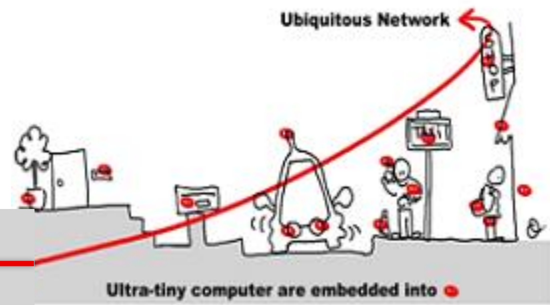
- Use of model checking technique
- Model checking goal: prove safety and liveness properties of a system in analyzing a model of the system.
- Model checking techniques require:
  - model of the system
  - express properties
  - algorithm to check properties againts the model ($\Rightarrow$ decidability)

Model Checking Techniques



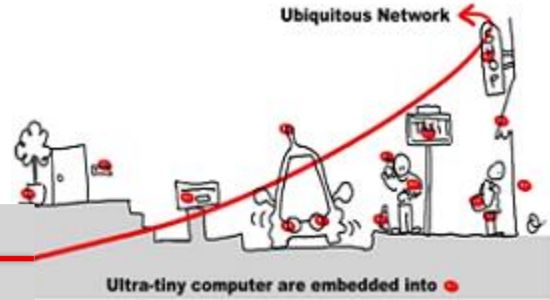
- Model = automata which is the set of program behaviors
- Properties expression = temporal logic:
  - LTL : liveness properties
  - CTL: safety properties
- Algorithm =
  - LTL : algorithm exponential wrt the formula size and linear wrt automata size.
  - CTL: algorithm linear wrt formula size and wrt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

Model Checking Model



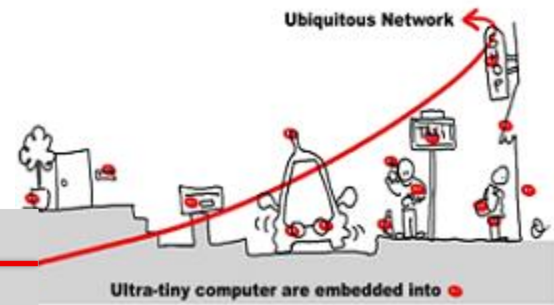
- Model = finite state machine (automata) which is the set of program behaviors
- Kripke structure:
  - non deterministic automata
  - Oriented graph
  - Nodes are program states
  - To each state , a set of atomic (basic) properties is associated

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



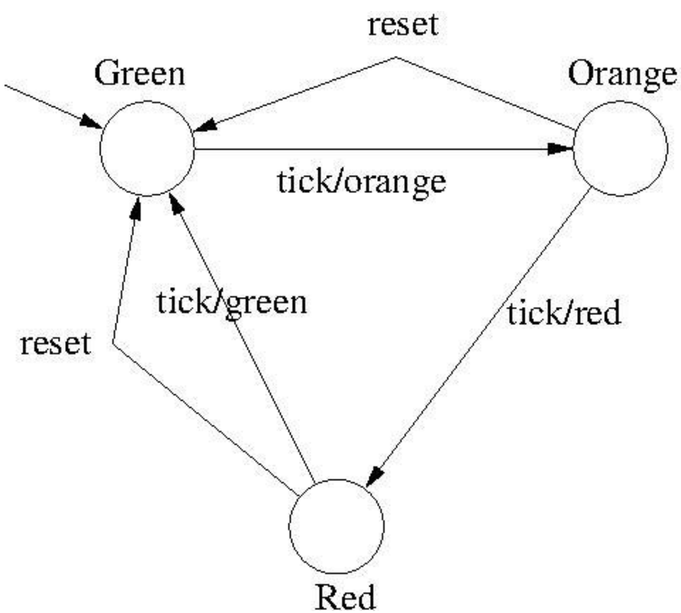
- Model = Mealy automata which is the set of program behaviors (deterministic)
- A Mealy automata is composed of:
  1. A finite set of states ($\mathbb{Q}$)
  2. A finite alphabet of triggers ($\mathbb{T}$)
  3. A finite alphabet of actions ($\mathbb{A}$)
  4. An initial state ($q^{init} \in \mathbb{Q}$)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

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

Model Specification

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

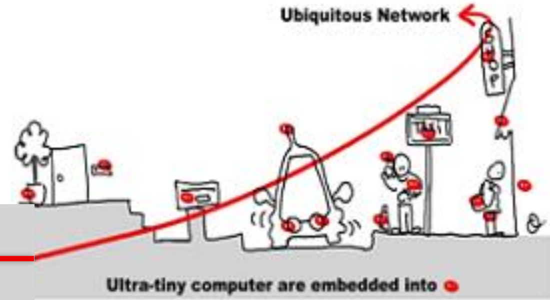
Example: **Traffic Light**



trigger: tick, reset

action:green,orange,red

Model Specification

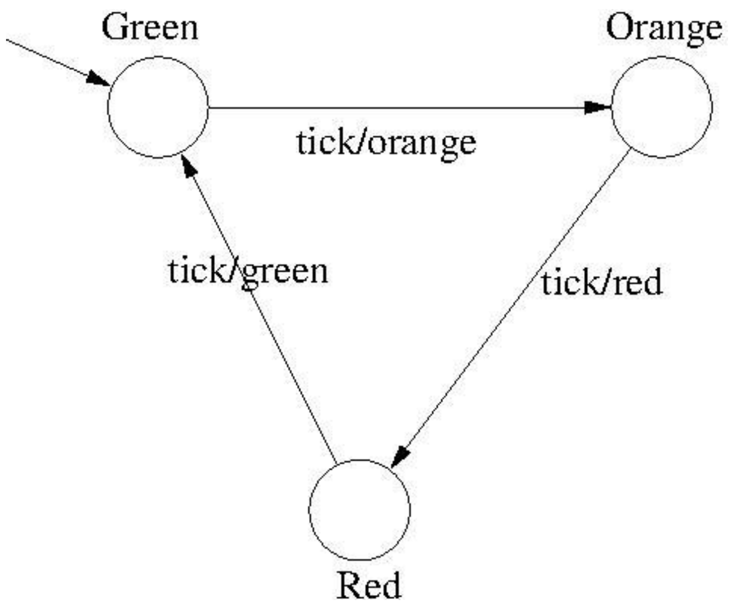


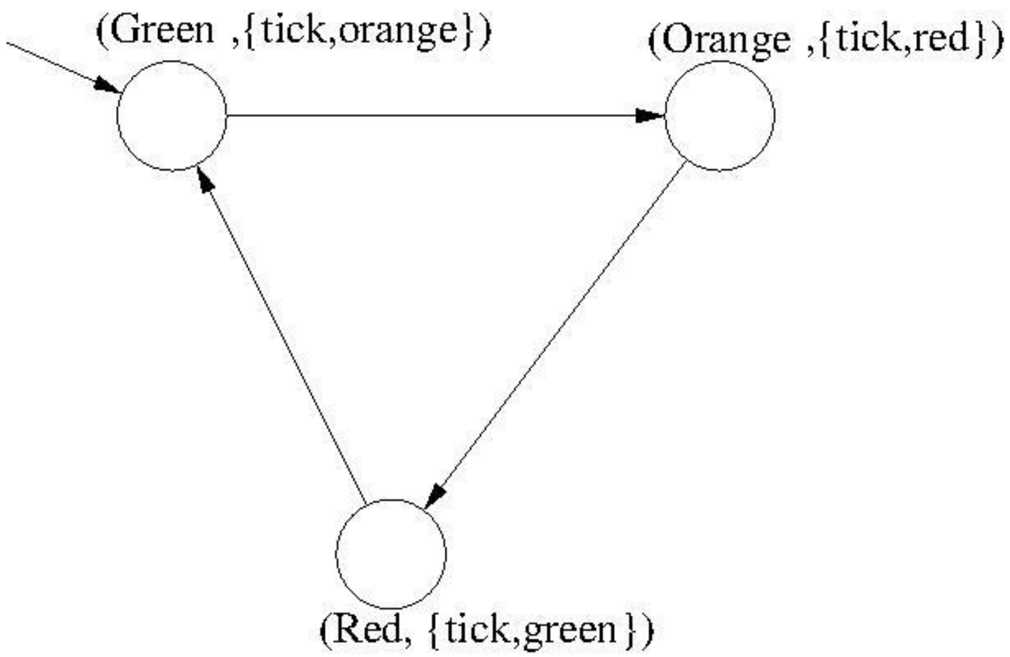
Mealy automata = Kripke structure

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

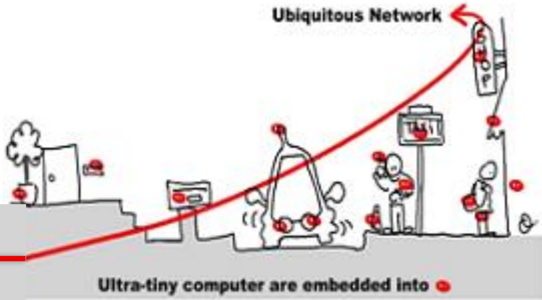
Model Specification

Mealy automata = Kripke structure



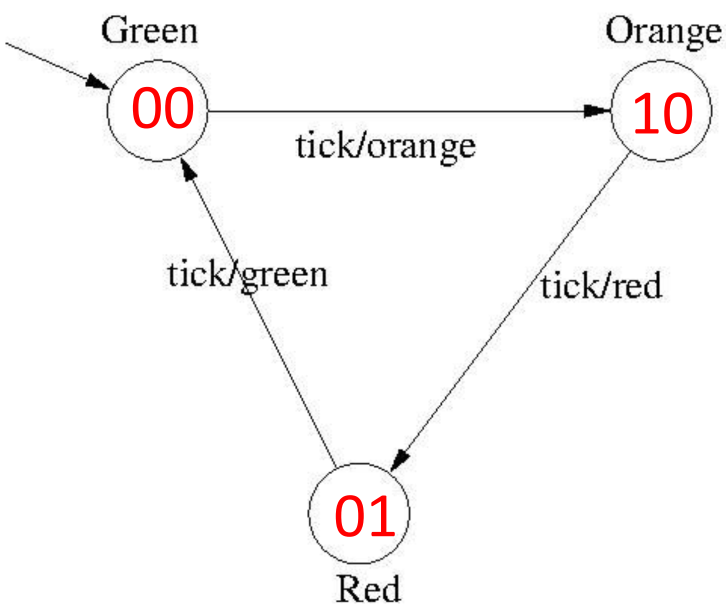


Implicit vs Explicit Mealy Machine



- Mealy automata is an explicit Mealy Machine
- Implicit representation as Boolean equation system with registers.
- $M = \langle Q, q^{init}, T, A, \delta, \lambda \rangle \quad \xi(M) = \langle T \cup A, R, D \rangle$:
  - 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;

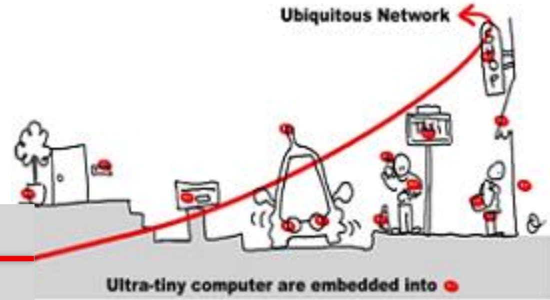
X1next = X0;

orange = not X0 and not X1 and tick;

green = not X0 and X1 and tick;

red = X0 and not X1 and tick;

Model Checking

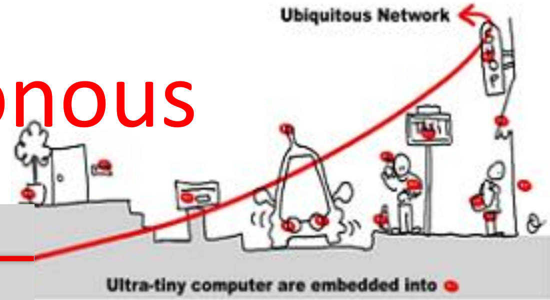


How design Mealy automata ?

Use synchronous languages to specify critical systems.

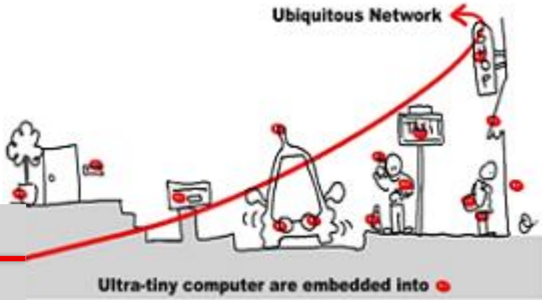
Synchronous programs = Mealy automata

Model Specification with Synchronous Languages



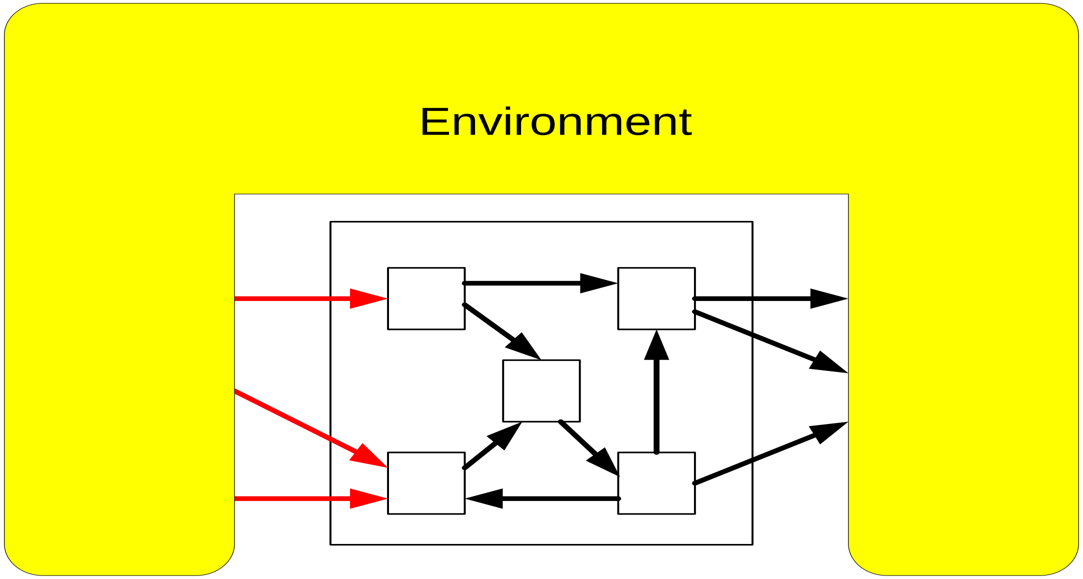
1. Synchronous languages have a **simple formal model** (a finite state machine) making formal reasoning tractable.
2. Synchronous languages support **concurrency** and offer an implicit or explicit means to express parallelism.
3. Synchronous languages are devoted to design **reactive systems**.

Determinism & Reactivity



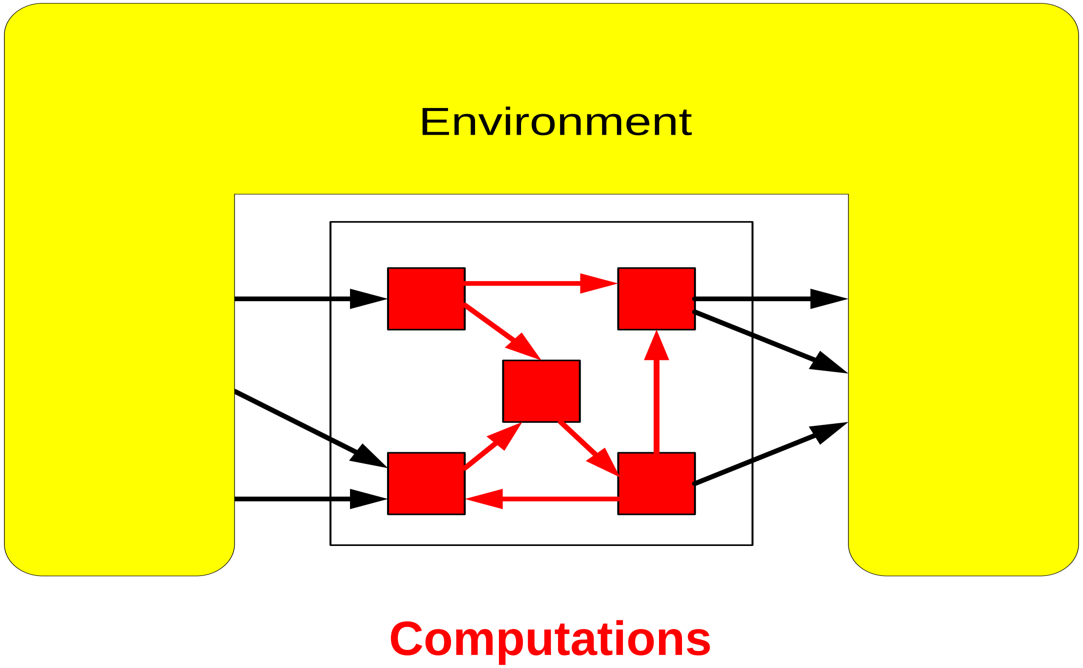
- Synchronous languages are deterministic and reactive
- Determinism:
  - The same input sequence always yields the same output sequence
- Reactivity:
  - The program must react(*) to any stimulus
  - Implies absence of deadlock
    - (*) *Does not necessary generate outputs, the reaction may change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

Synchronous Reactive Programs (1)

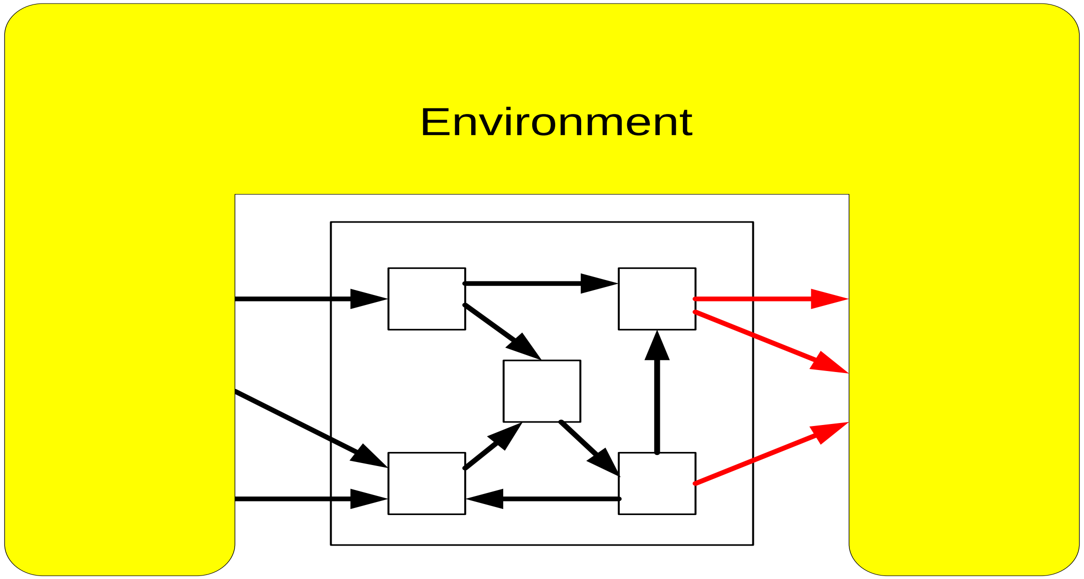


Read

Synchronous Reactive Programs (1)



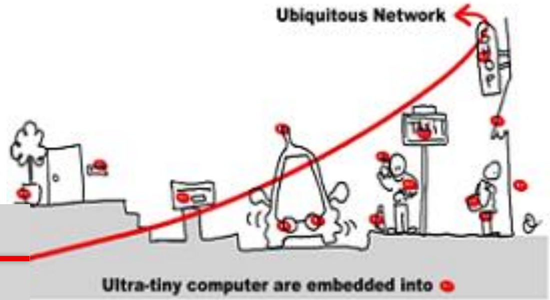
Synchronous Reactive Programs (1)

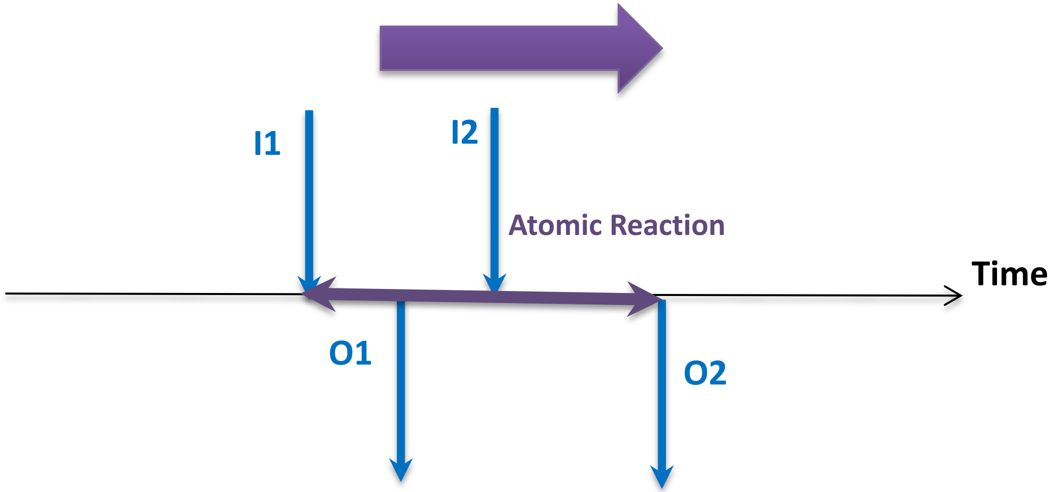


Write

Atomic execution: read, compute, write

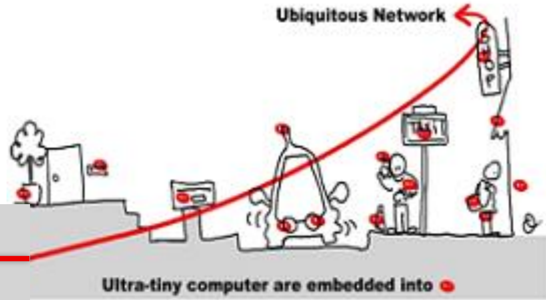
Synchronous Modelling





- Atomic execution of the reaction
- Logical time
- Well founded
- Liable to formal analysis

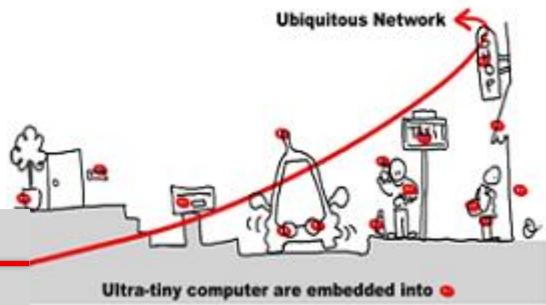
Synchronous Hypothesis



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

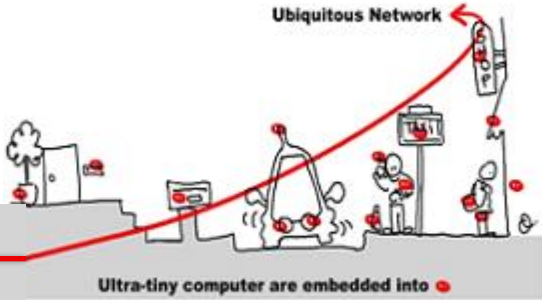
  Use N as time base
- A reaction executes in one instant.
- Actions that compose the reaction may be partially ordered.

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.

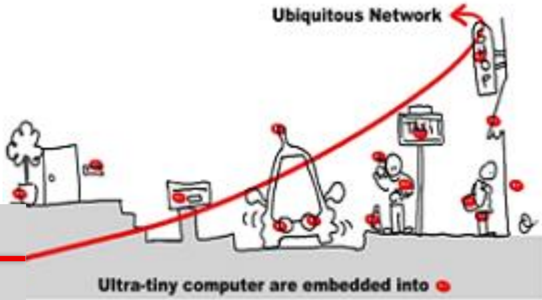
Reactive ?



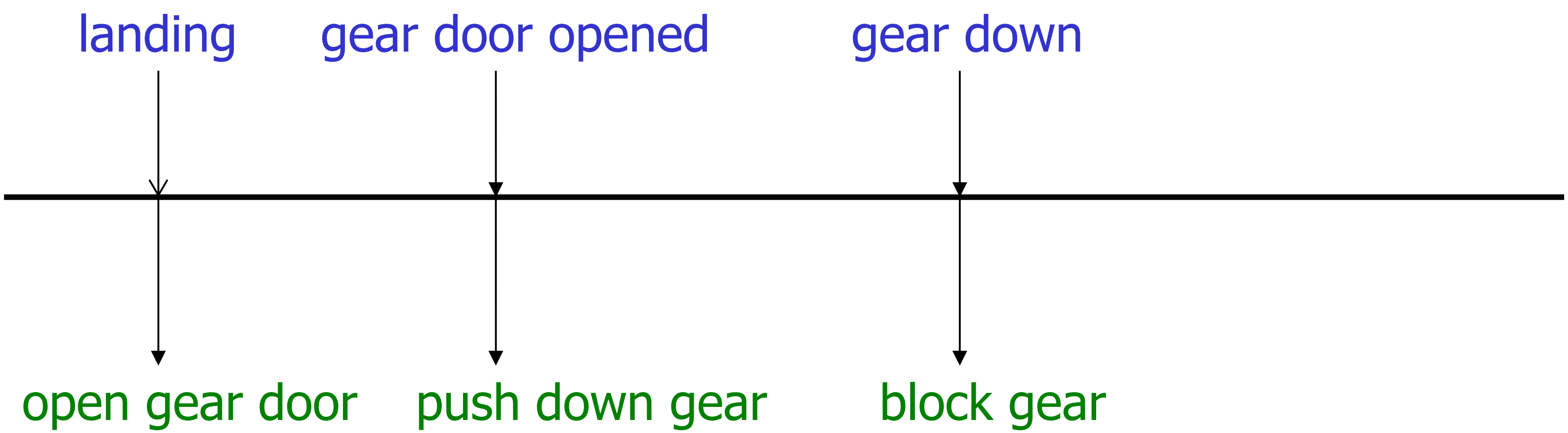
- Different ways to "react" to the environment:
  - Event driven system:
    - Receive events
    - Answer by sending events
  - Data flow system:
    - Receive data continuously
    - Answer by treating data continuously also

Some systems have components of both kinds

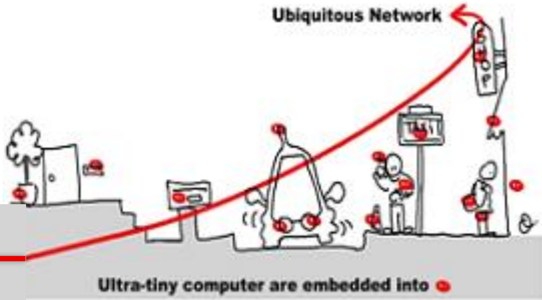
Event Driven Reactive System



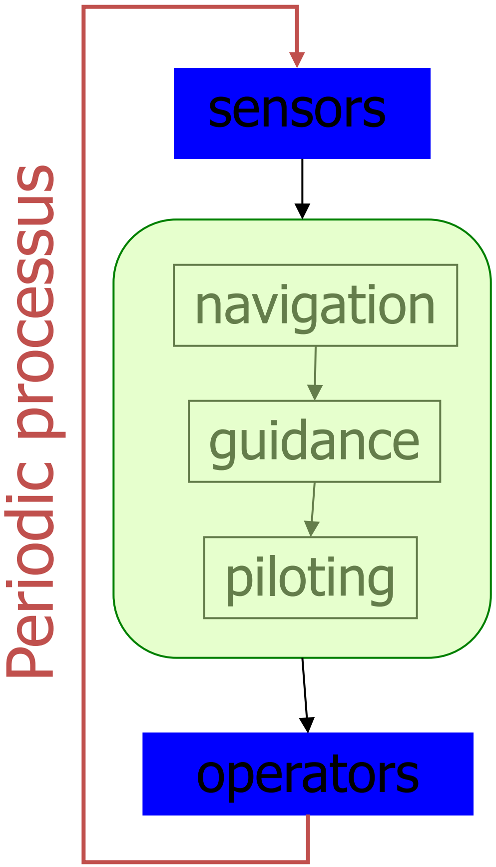
Langing gear management



Data Flow Reactive System (Example)

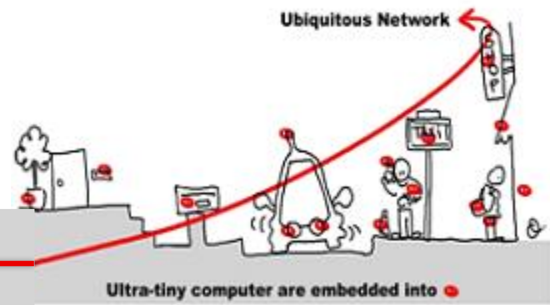


Control/Command vehicle



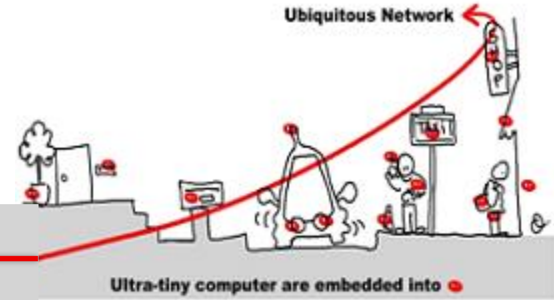
- get measures
- where am I ?
- where go I ?
- command computation
- command to operators

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

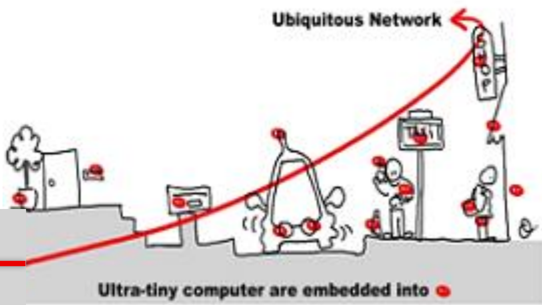
Imperative Language



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

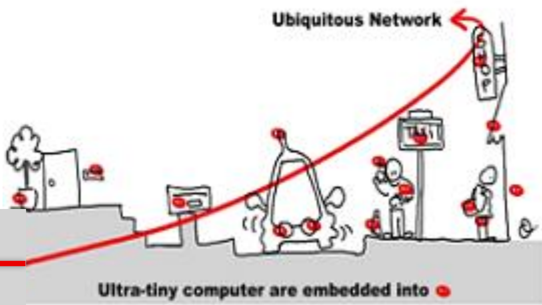
1. Listen input and output events
2. Specific operators to deal with the logical time (await)
3. Test of presence or absence of signals (present)
4. Synchronous parallelism (||)
5. Emit to change the environment (emit S)
6. Usual operators (loop, abort when)

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

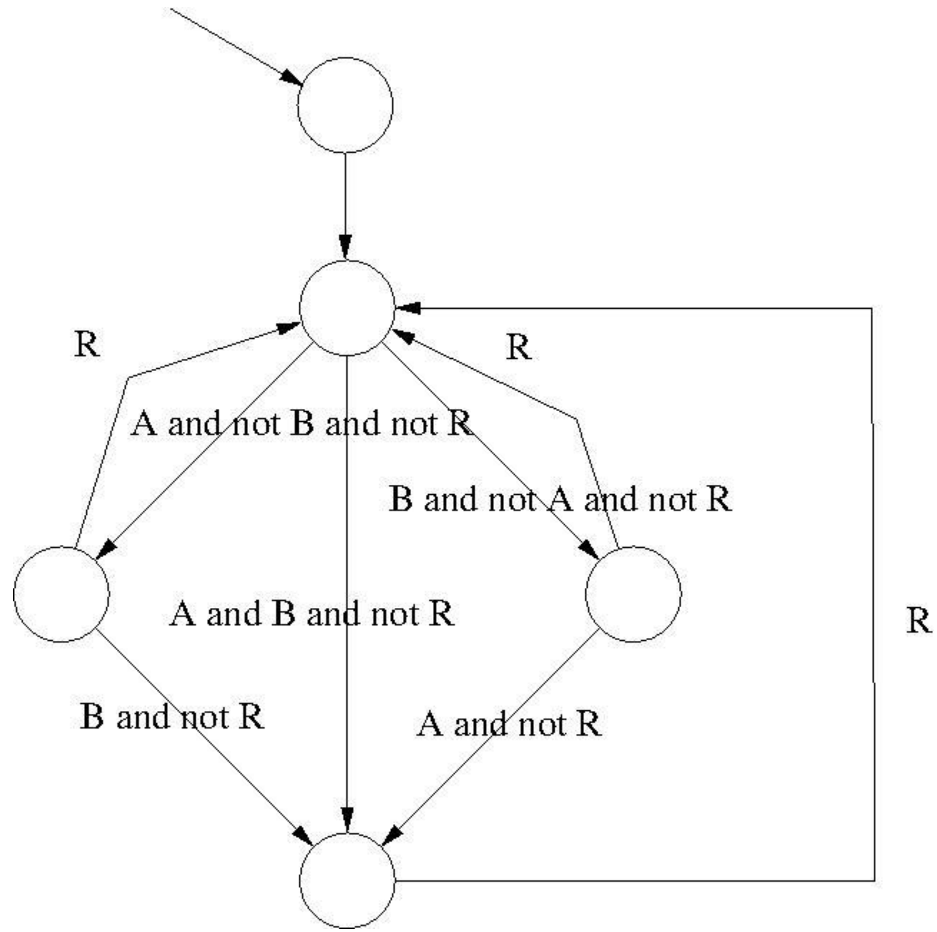



```
every Morning do
  repeat NumberOfLaps times
    abort
      abort  sustain Walk when 100 Meter;
      abort
        every Step do emit Jump end every
      when 15 Second;
      sustain Run
    when Lap
  end repeat
end every
```

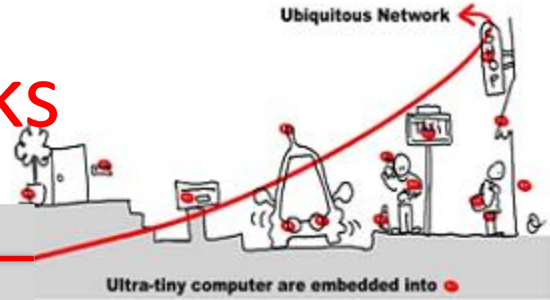
sequence

Esterel program = Mealy Machine

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

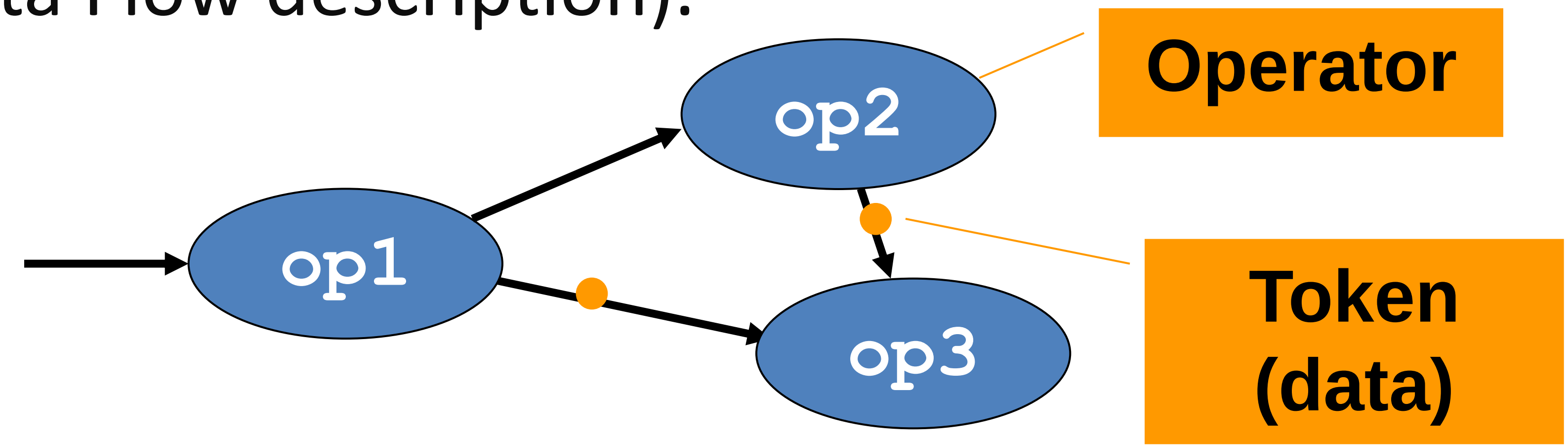


Data flow = Operator Networks

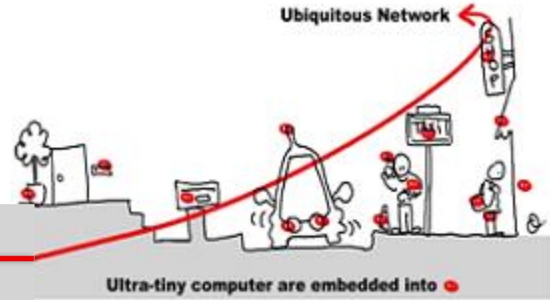


Data flow programs can be interpreted as networks of operators.

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



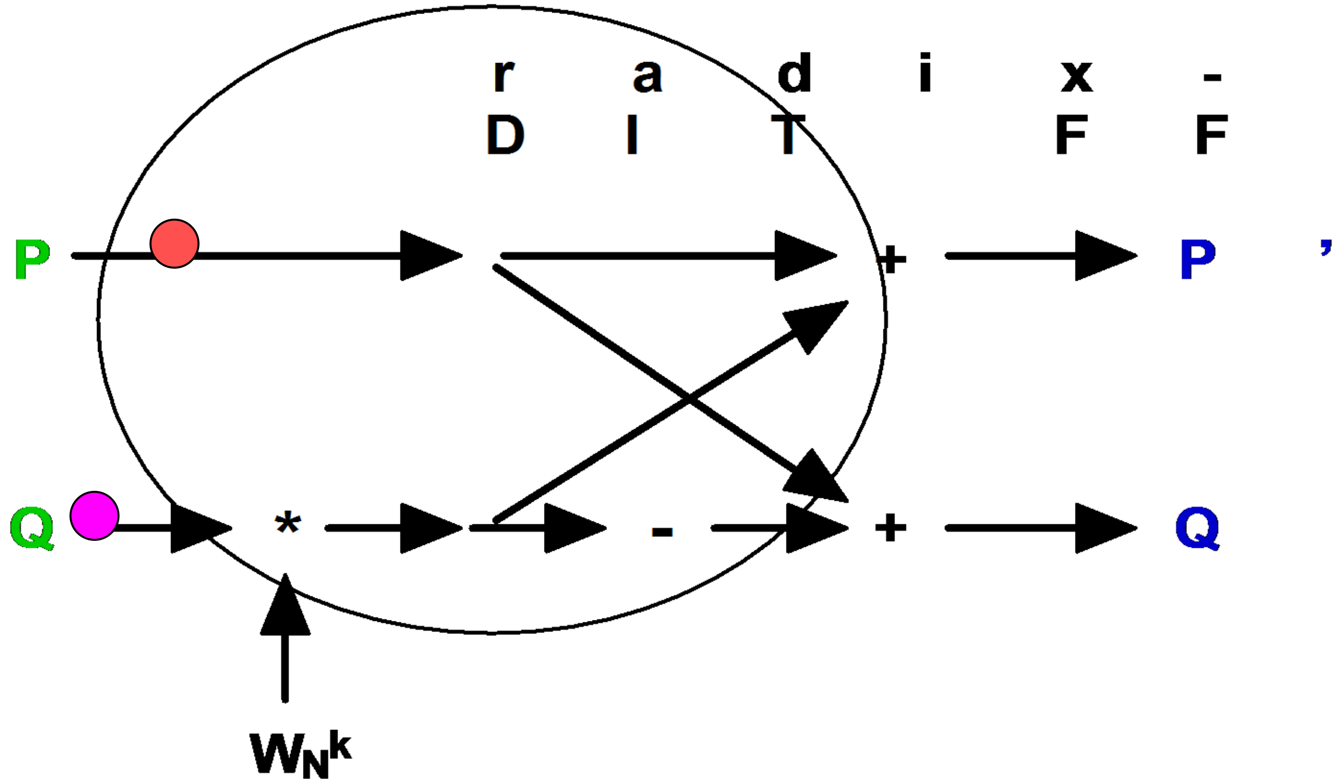
Flows, Clocks



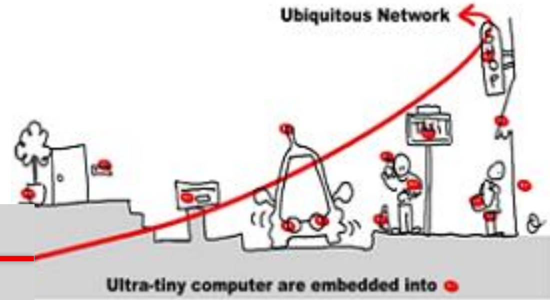
- A flow is a pair made of
  - A possibly infinite sequence of values of a given type
  - A clock representing a sequence of instants

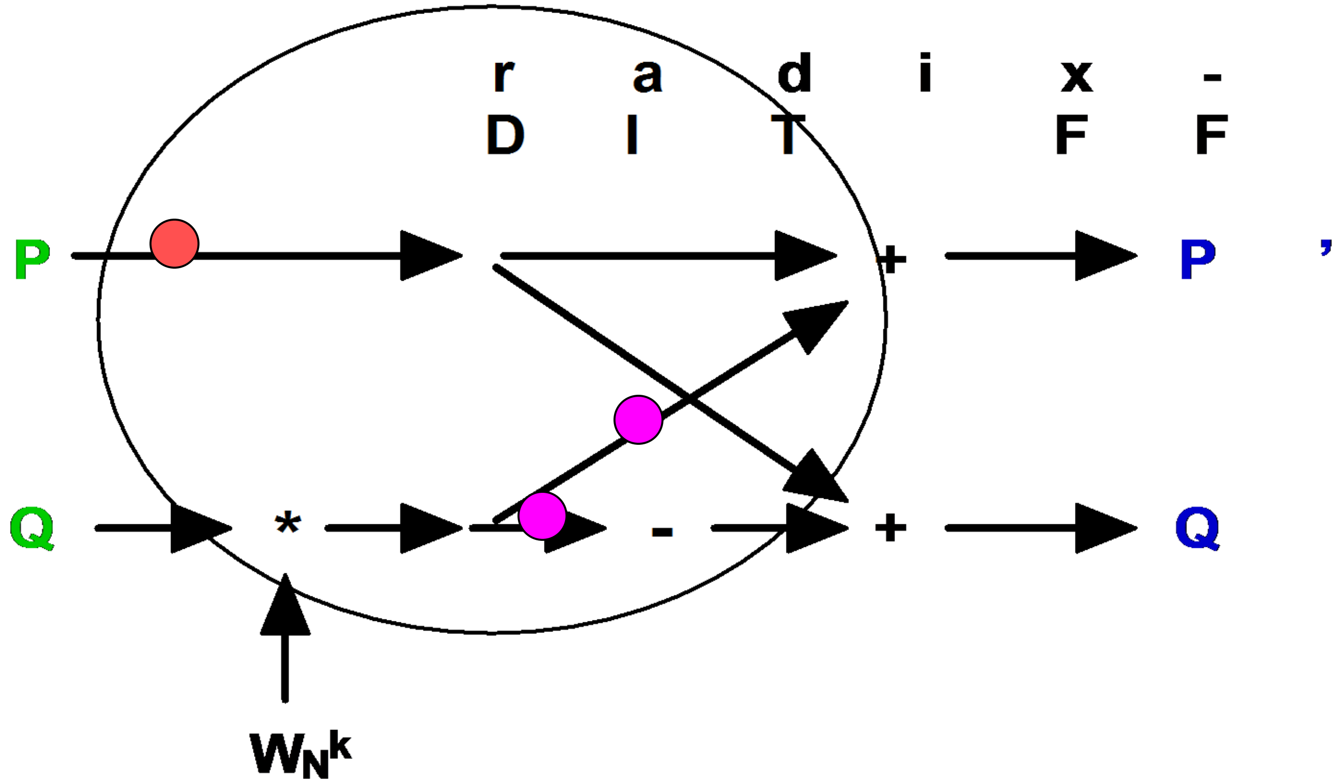
$$\mathbf{X:T} \qquad (x_1, x_2, \dots, x_n, \dots)$$

An example of Data Flow

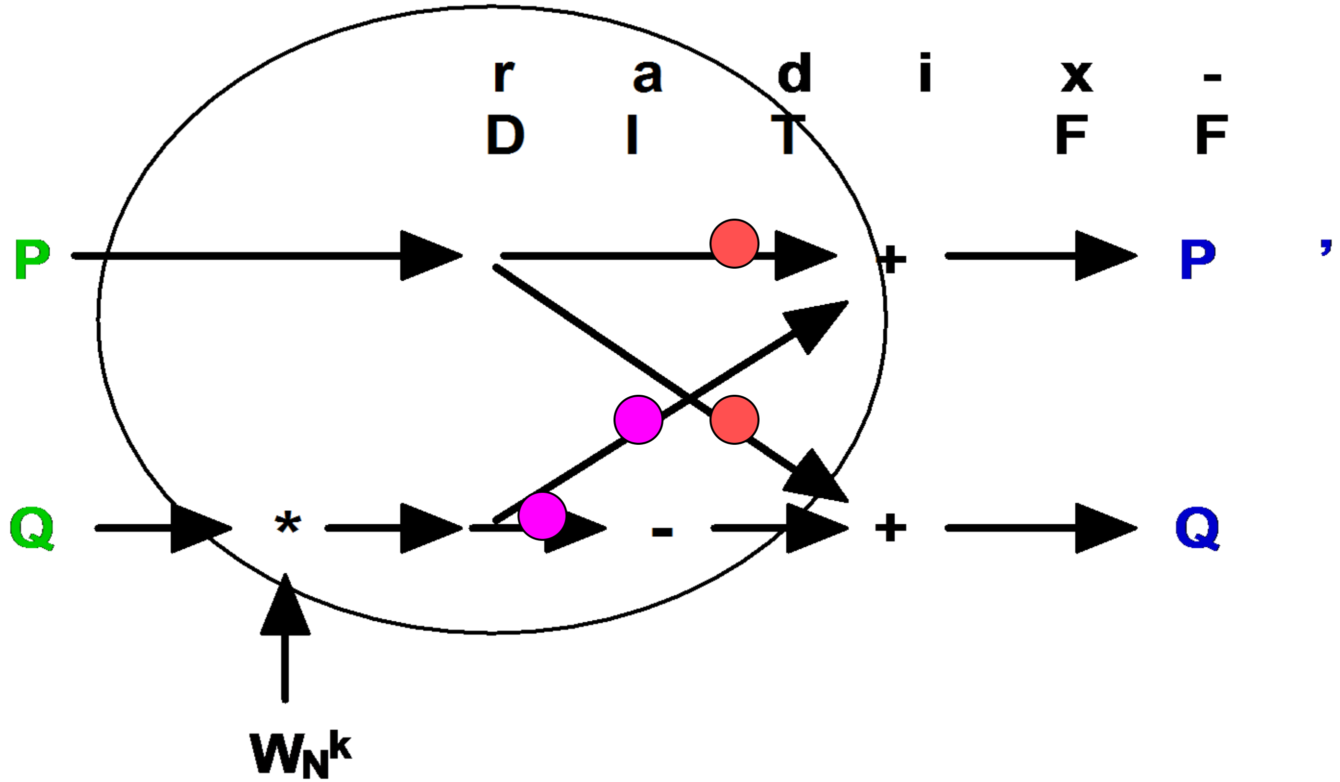


Data Flow

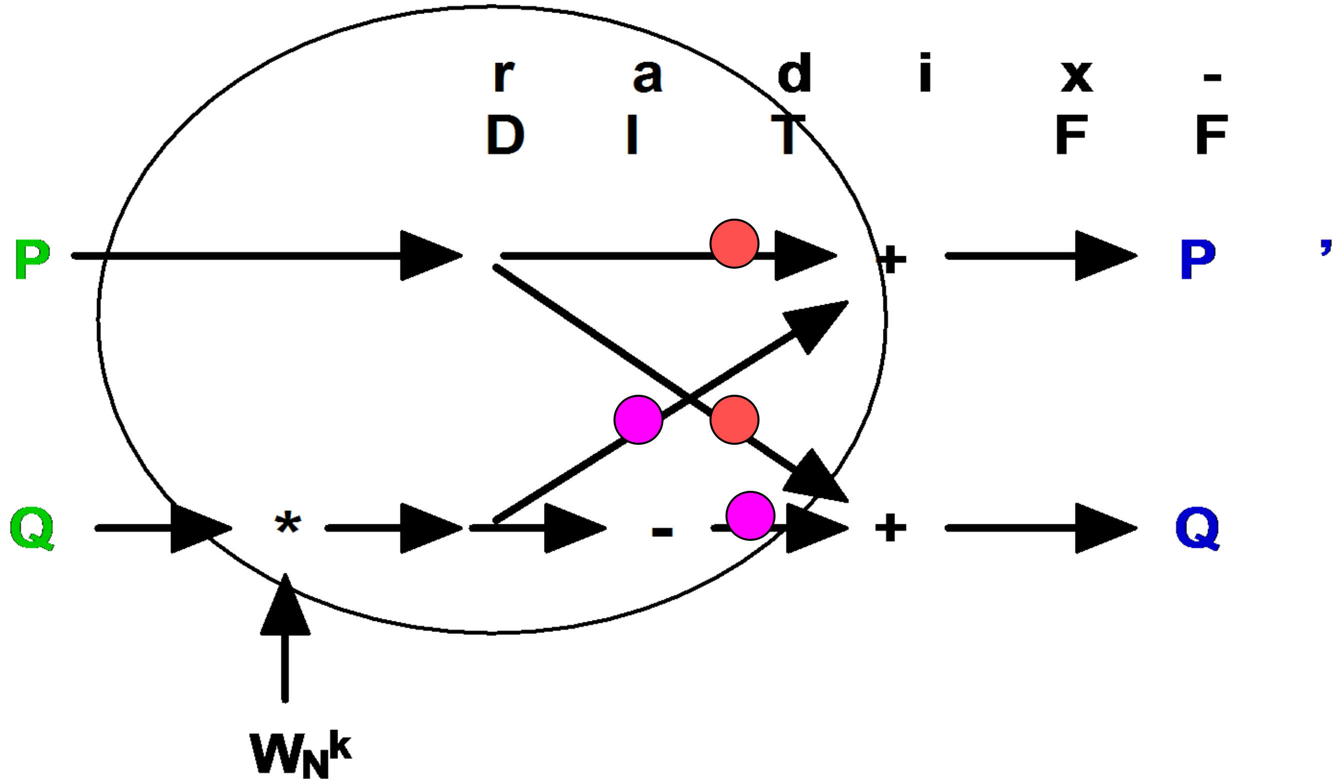

Ubiquitous Network
Ultra-tiny computer are embedded into


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

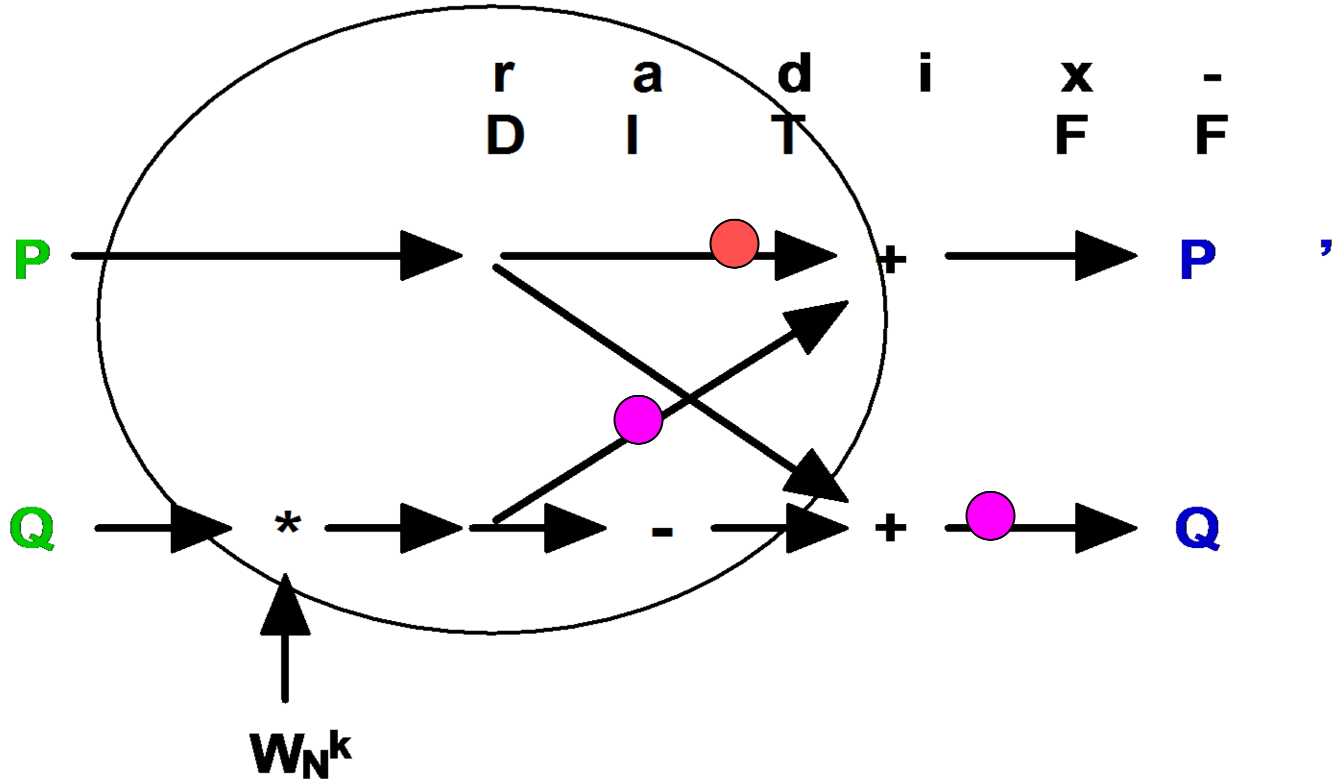
Data Flow



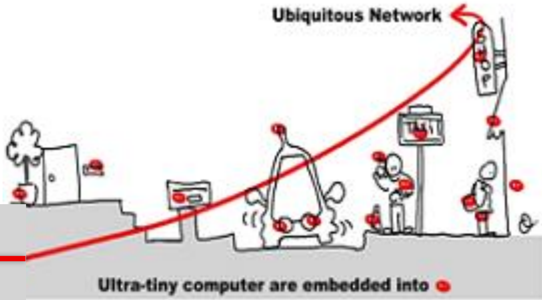
Data Flow



Data Flow

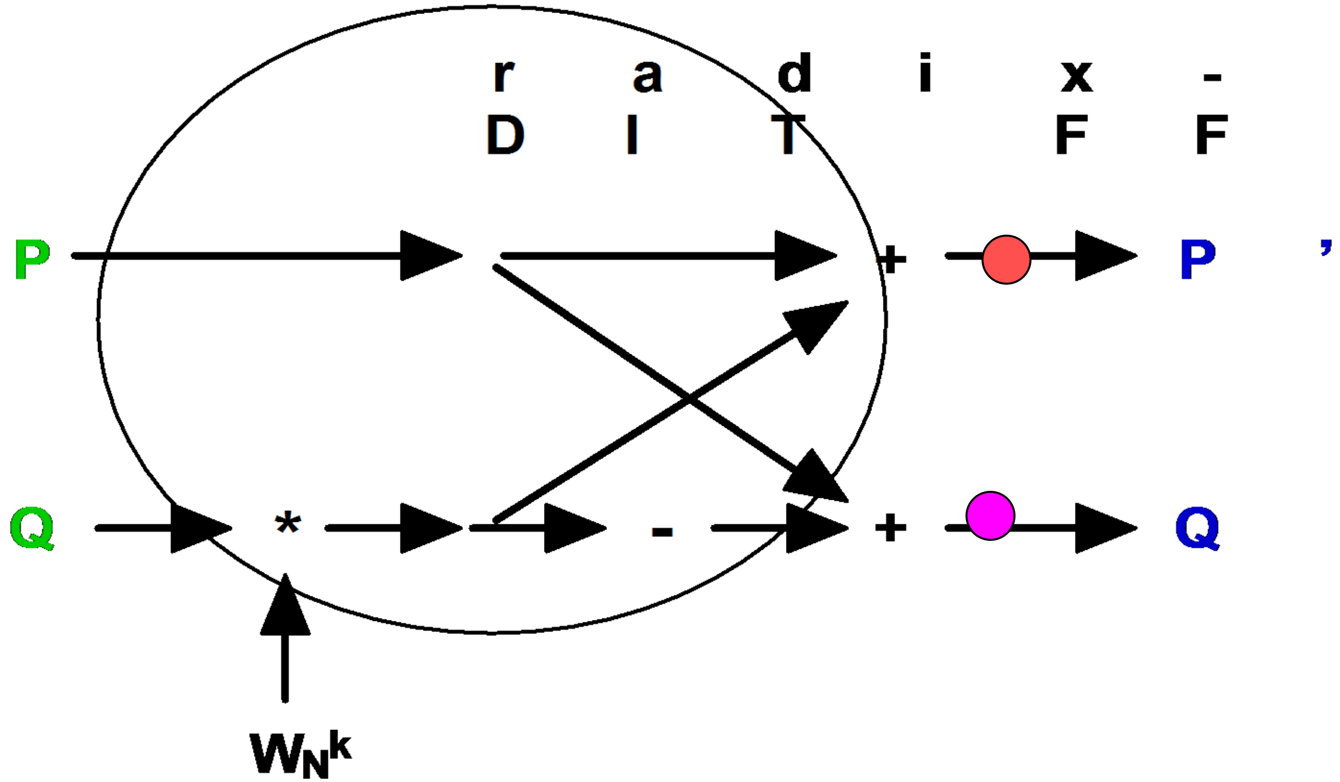


r a d i x -
D I T F F
P
Q
*
-
+
+
P ’
Q
W_N^k

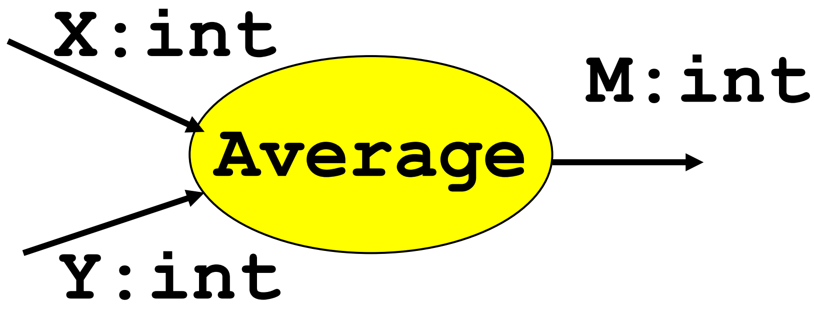


Ubiquitous Network
Ultra-tiny computer are embedded into

Data Flow



Data Flow Synchronous Languages



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

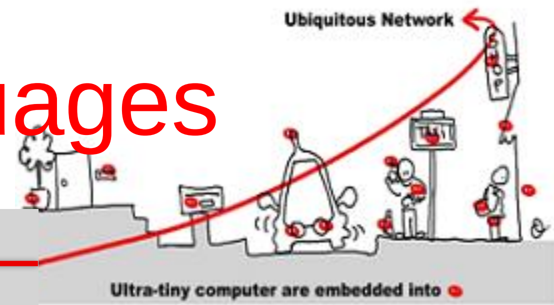
$$M = (X + Y)/2$$

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

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

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

Data Flow Synchronous Languages



Memorizing to take the past into account:

1. pre (**previous**):

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

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

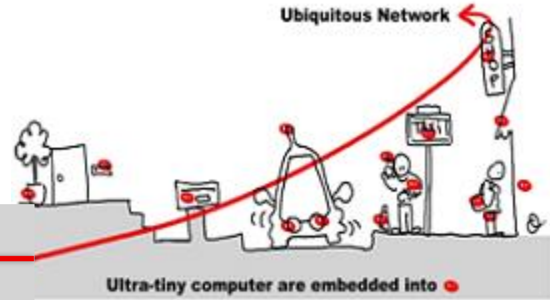
   nil undefined value denoting uninitialized memory

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

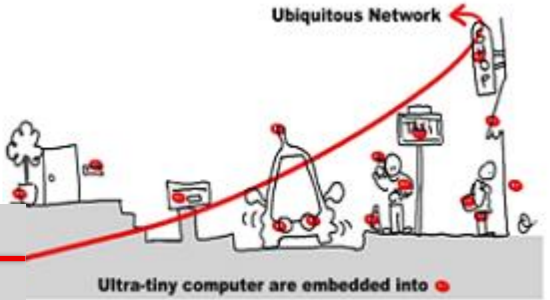


n= 0 → pre(n) + 1

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):

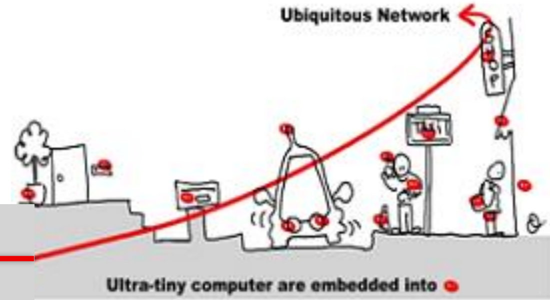
$$c = init \rightarrow pre(c) + 2$$

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,....)

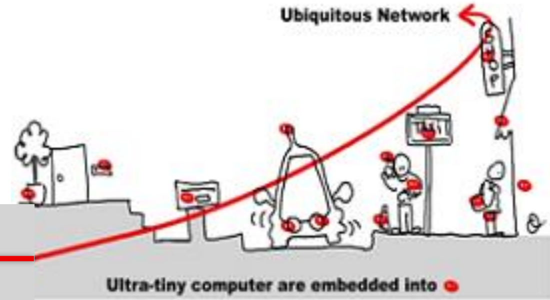
operator DoubleCall (even:bool) returns (n:int)

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

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

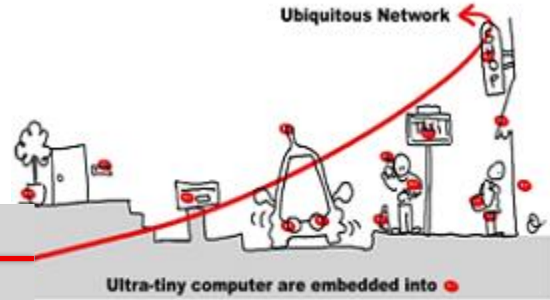
(1,3,4,6,9,11,12,14,17)

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):

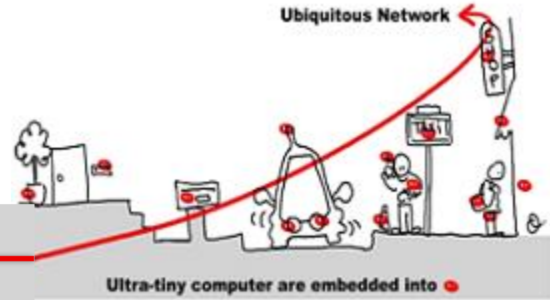
count: 0 1 2 3 1 2 3……

cpt = 0 1 2 0 1 2 0…….

modulo_clock = ff ff ff tt ff ff tt ….

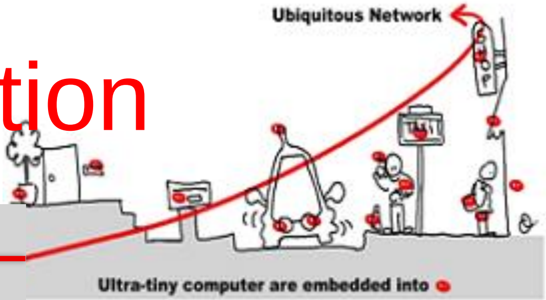
```
var count : int;
    count = 0 -> if incr pre (cpt) + 1
                 else pre (cpt);
    cpt =  count mod modulo;
    modulo_clock = count != cpt;
```

Timer



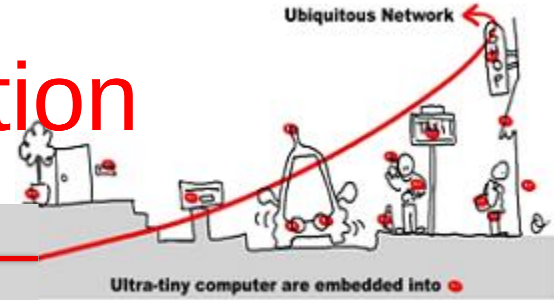

```
operator Timer returns (hour, minute, second:int);
var hour_clock, minute_clock, day_clock : bool;

(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



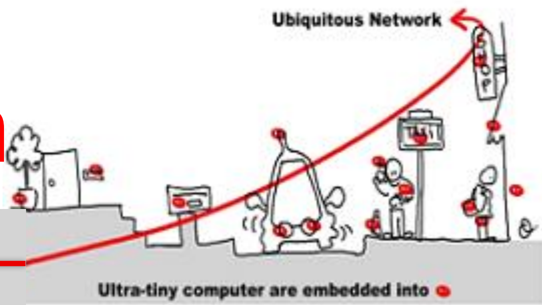

```
operator WD (set, reset, deadline:bool)
                        returns (alarm:bool);
var is_set:bool;
  alarm = is_set and deadline;
  is_set = false -> if set then true
                              else if reset then false
                                      else pre(is_set);
  assert not(set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

initial

S0: alarm := false;

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

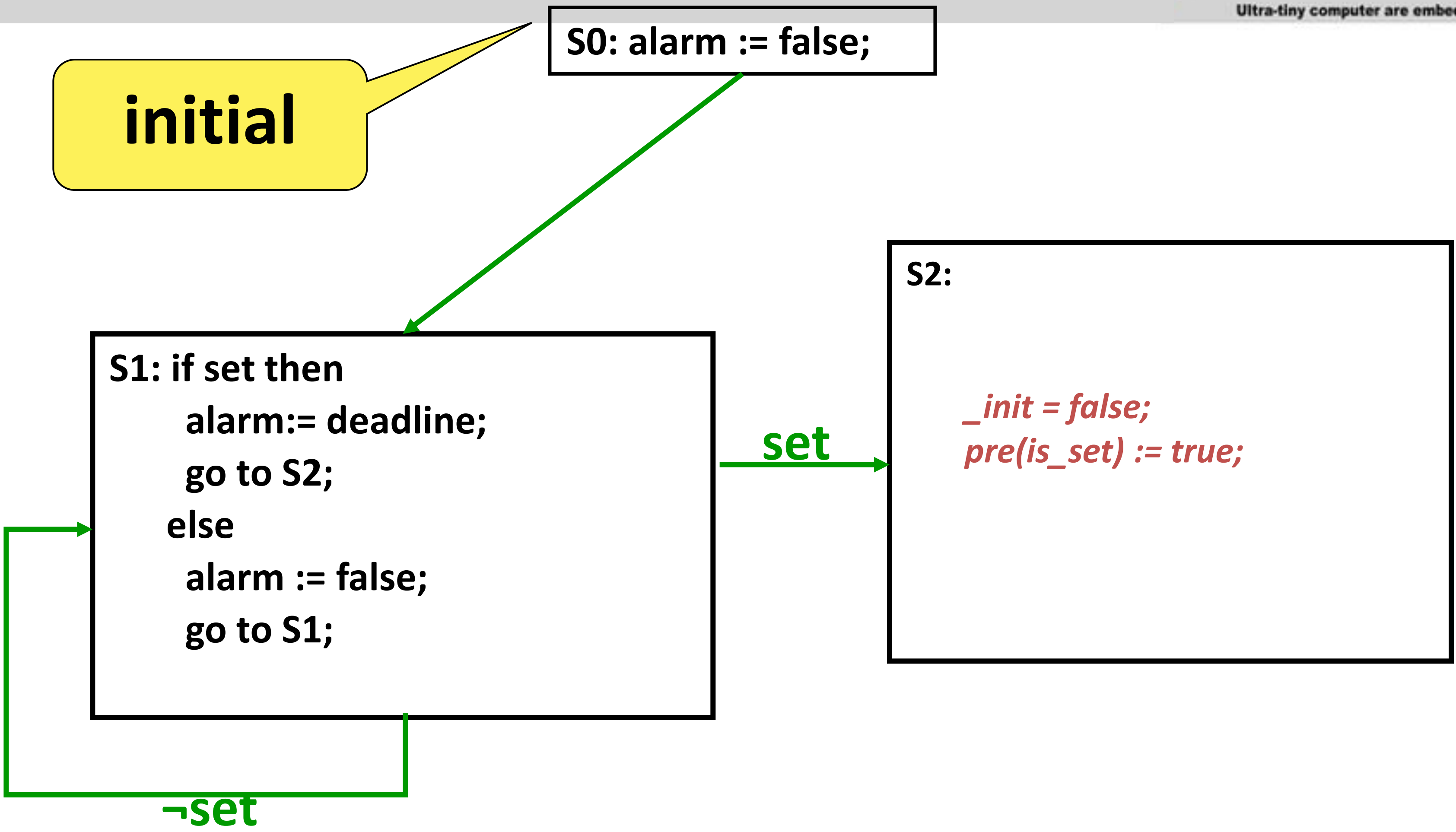
set

¬set

S2:

```
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

S0: alarm := false;

initial

```
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
```

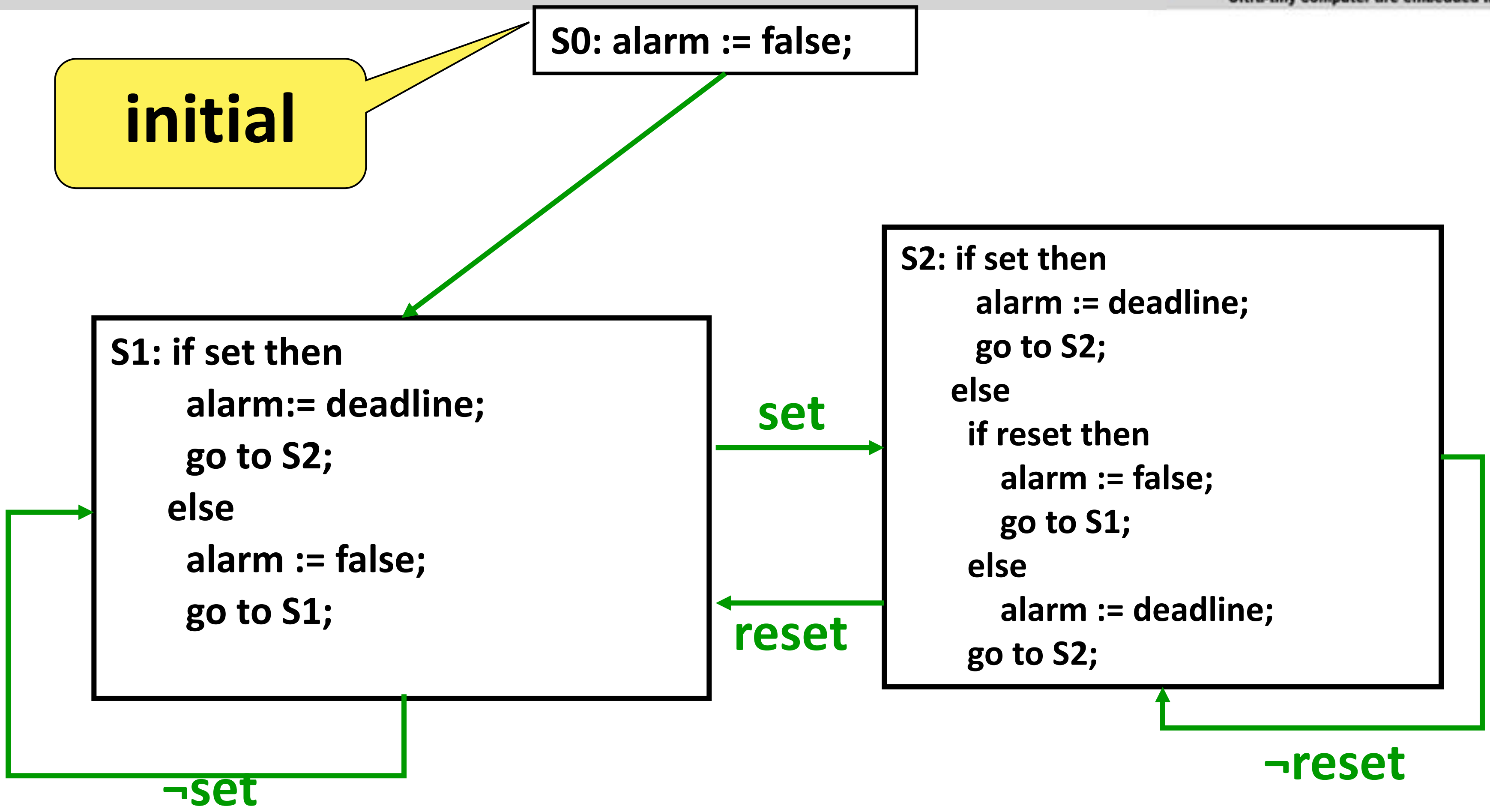
set

reset

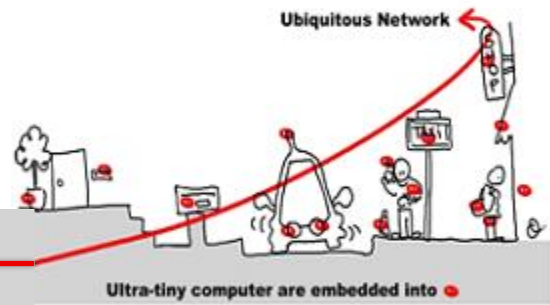
```
S2: if set then
      alarm := deadline;
      go to S2;
    else
     if reset then
       alarm := false;
       go to S1;
     else
       alarm := deadline;
     go to S2;
```

¬reset

Lustre Program Compilation

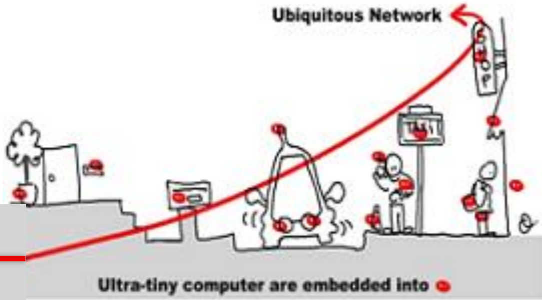


Model Checking Technique



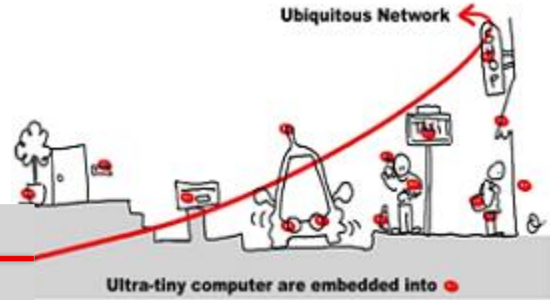
- Model = automata which is the set of program behaviors
- Properties expression = temporal logic:
  - LTL : liveness properties
  - CTL: safety properties
- Algorithm =
  - LTL : algorithm exponential wrt the formula size and linear wrt automata size.
  - CTL: algorithm linear wrt formula size and wrt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

Properties Checking



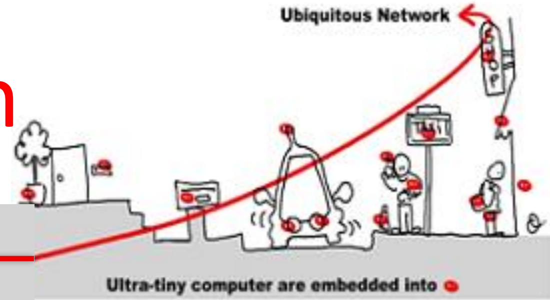
- Liveness Property Φ :
  - $\Phi \Rightarrow$ automata $B(\Phi)$
  - $\mathbb{L}(B(\Phi)) = \varnothing$ decidable
  - $\Phi \mid= \mathcal{M}$: $\mathbb{L}(\mathcal{M} \otimes B(\sim\Phi)) = \varnothing$

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($\varnothing$) the set of states where $\varnothing$ is true.

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

Algorithm:

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

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

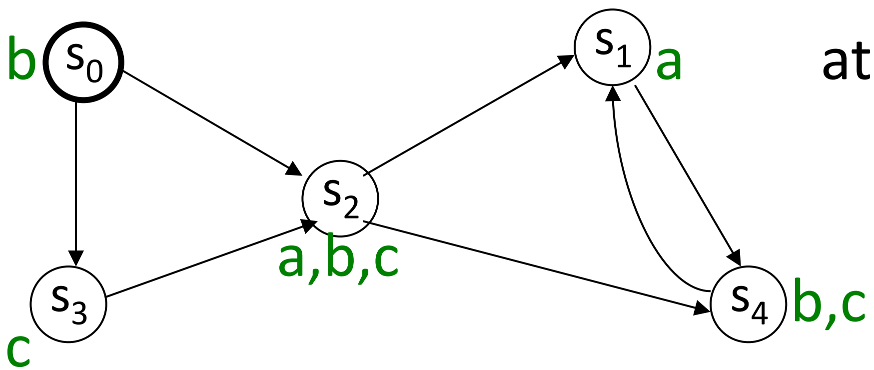
$Sat(\Phi 1 \text{ or } \Phi 2) = Sat(\Phi 1) \cup Sat(\Phi 2)$

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

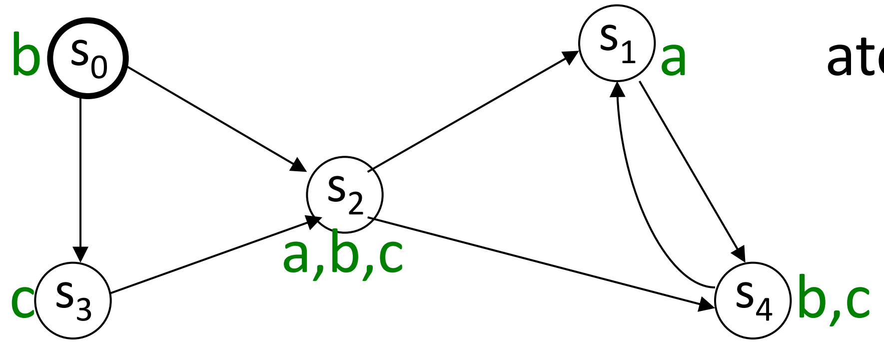
$gfp\ (\Gamma(x) = \text{Sat(a or b)} \cap \text{Pre}(x))$

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

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

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

Example



atomic formulas: a, b, c

EG (a or b)

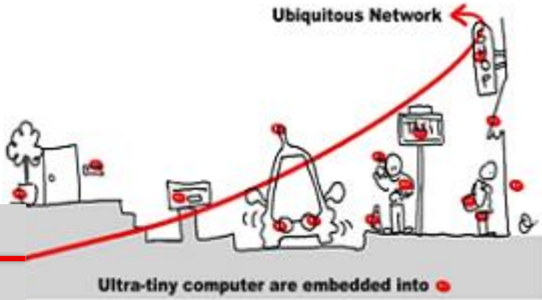
$\Gamma(\{s_0, s_1, s_2, s_3, s_4\}) = \{s_0, s_1, s_2, s_4\}$

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

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

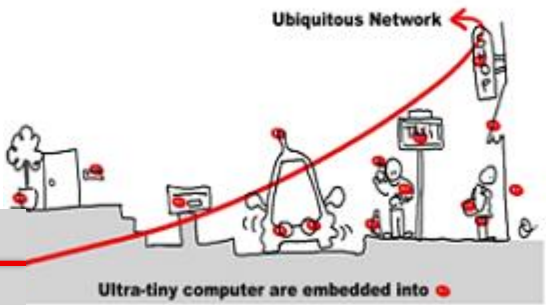
$S_0 \models \text{EG(a or b)}$

Model Checking Implementation



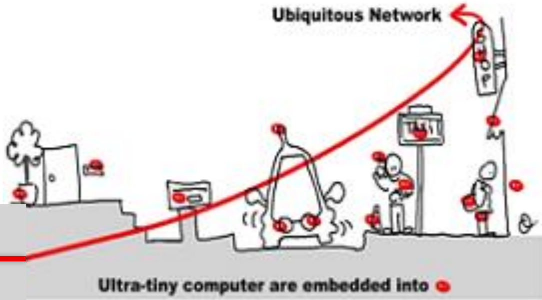
- Problem: the size of automata
- Solution: symbolic model checking
- Usage of BDD (Binary Decision Diagram) to encode both automata and formula.
- Each Boolean function has a unique representation
- Shannon decomposition:
  - $f(x_0,x_1,\ldots,x_n) = f(1, x_1,\ldots, x_n) \vee f(0, x_1,\ldots,x_n)$

Model Checking Implementation

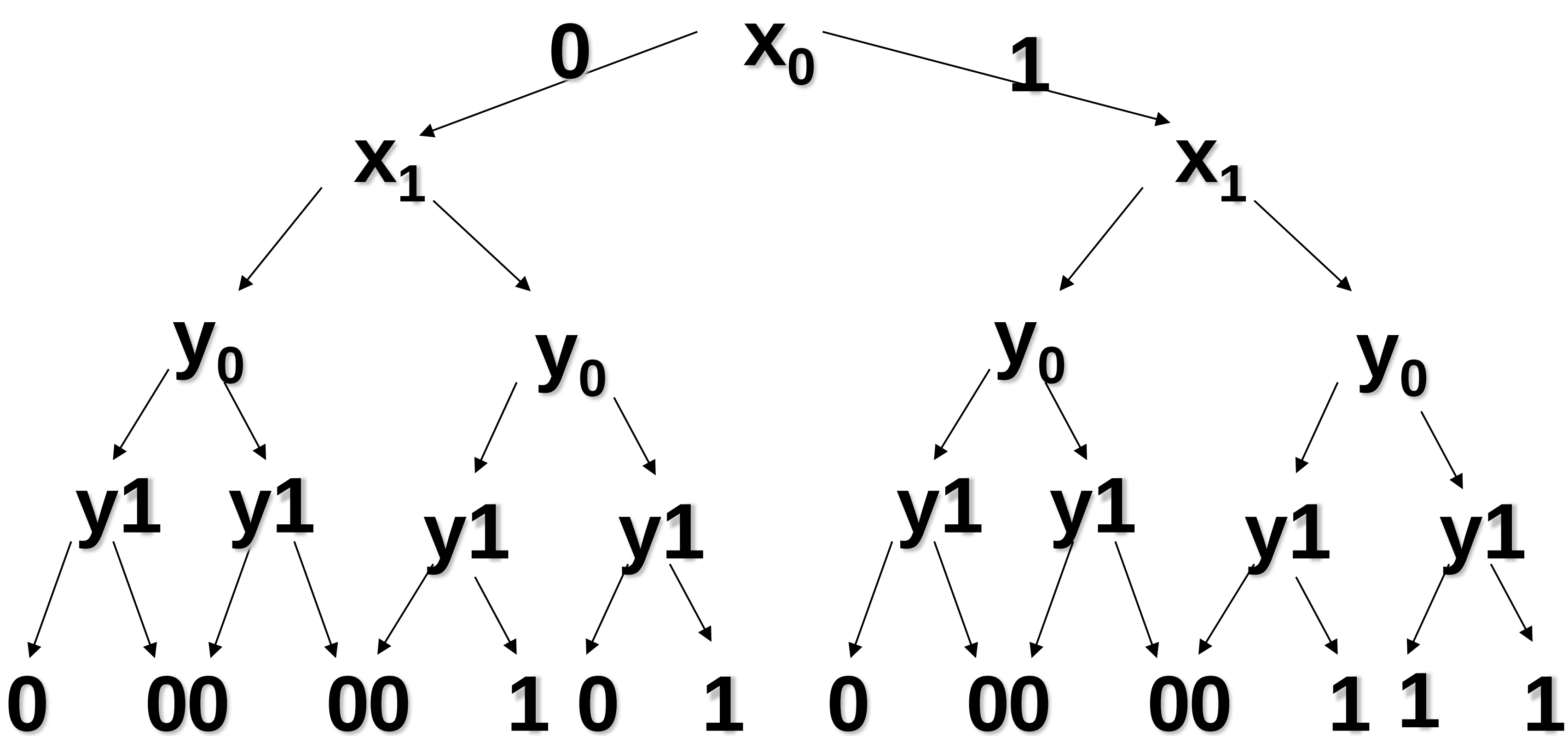


- When applying recursively Shannon decomposition on all variables, we obtain a tree where leaves are either 1 or 0.
- BDD are:
  - A concise representation of the Shannon tree
  - no useless node (if x then g else g $\Leftrightarrow$ g)
  - Share common sub graphs

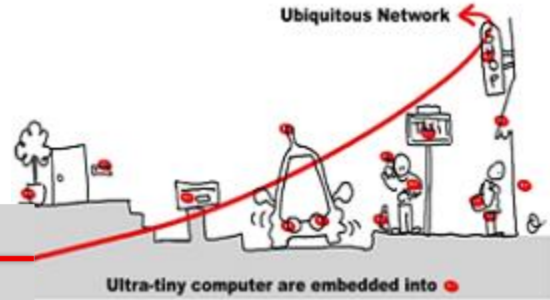
Model Checking Implementation (2)



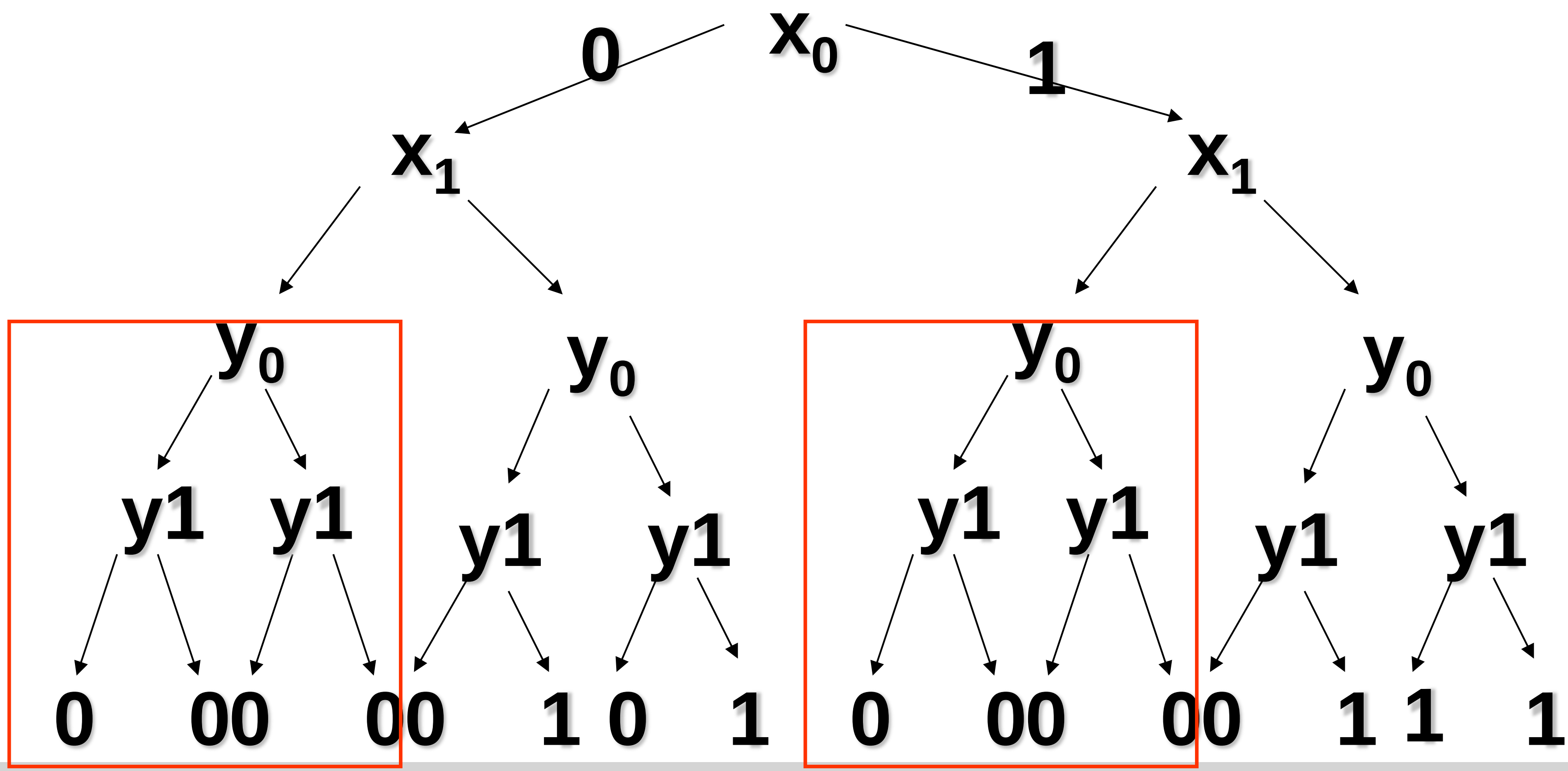
$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$



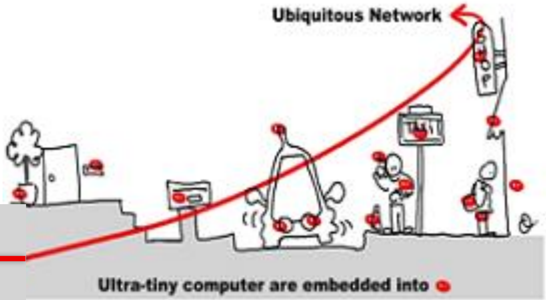
Model Checking Implementation (2)



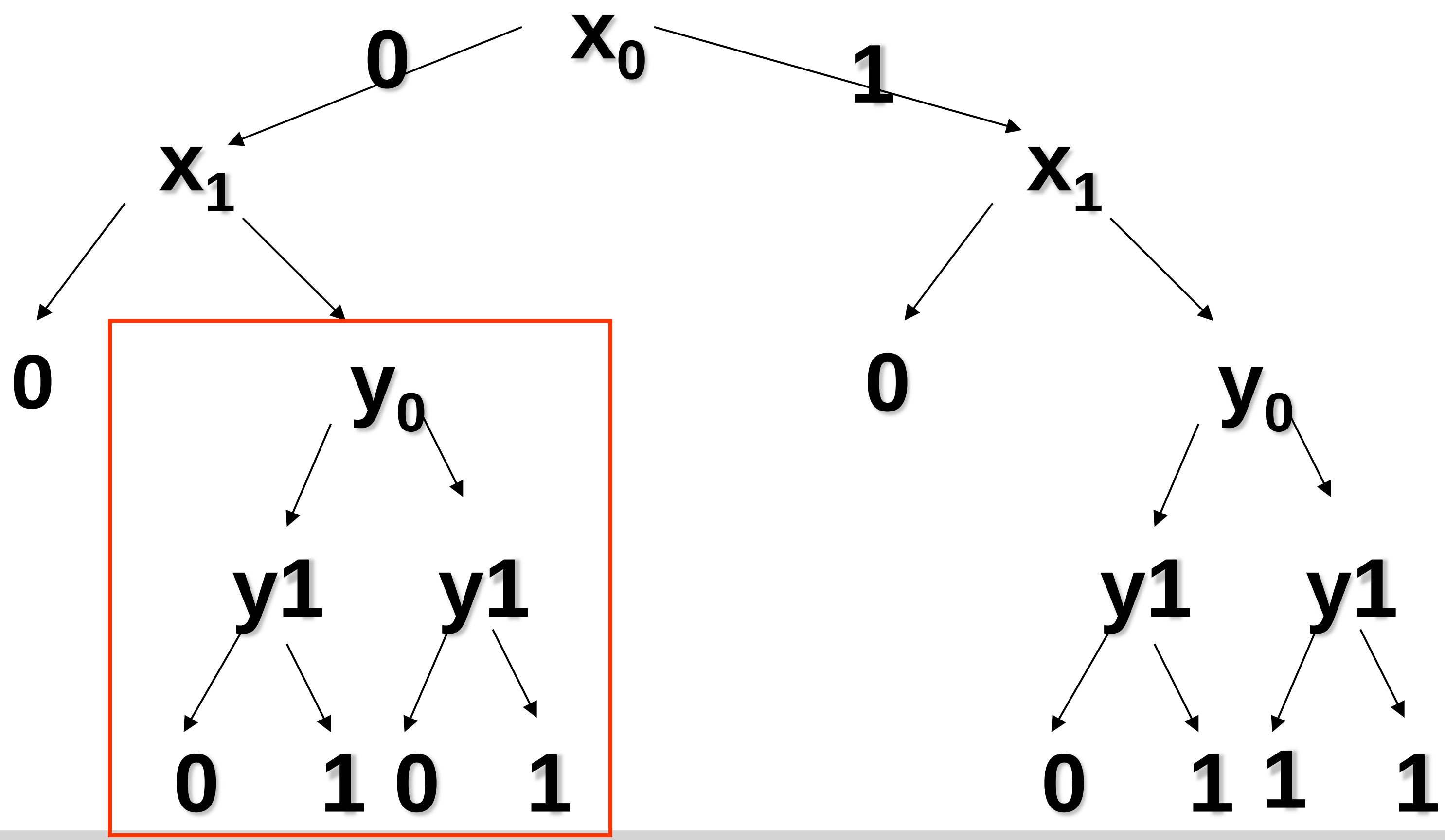
$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$



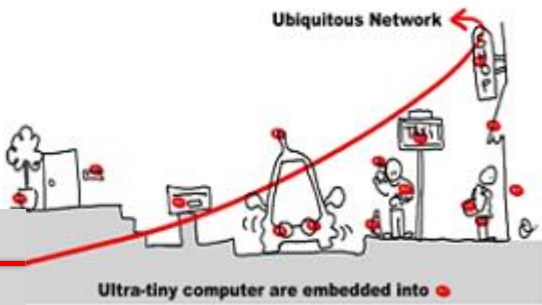
Model Checking Implementation (2)



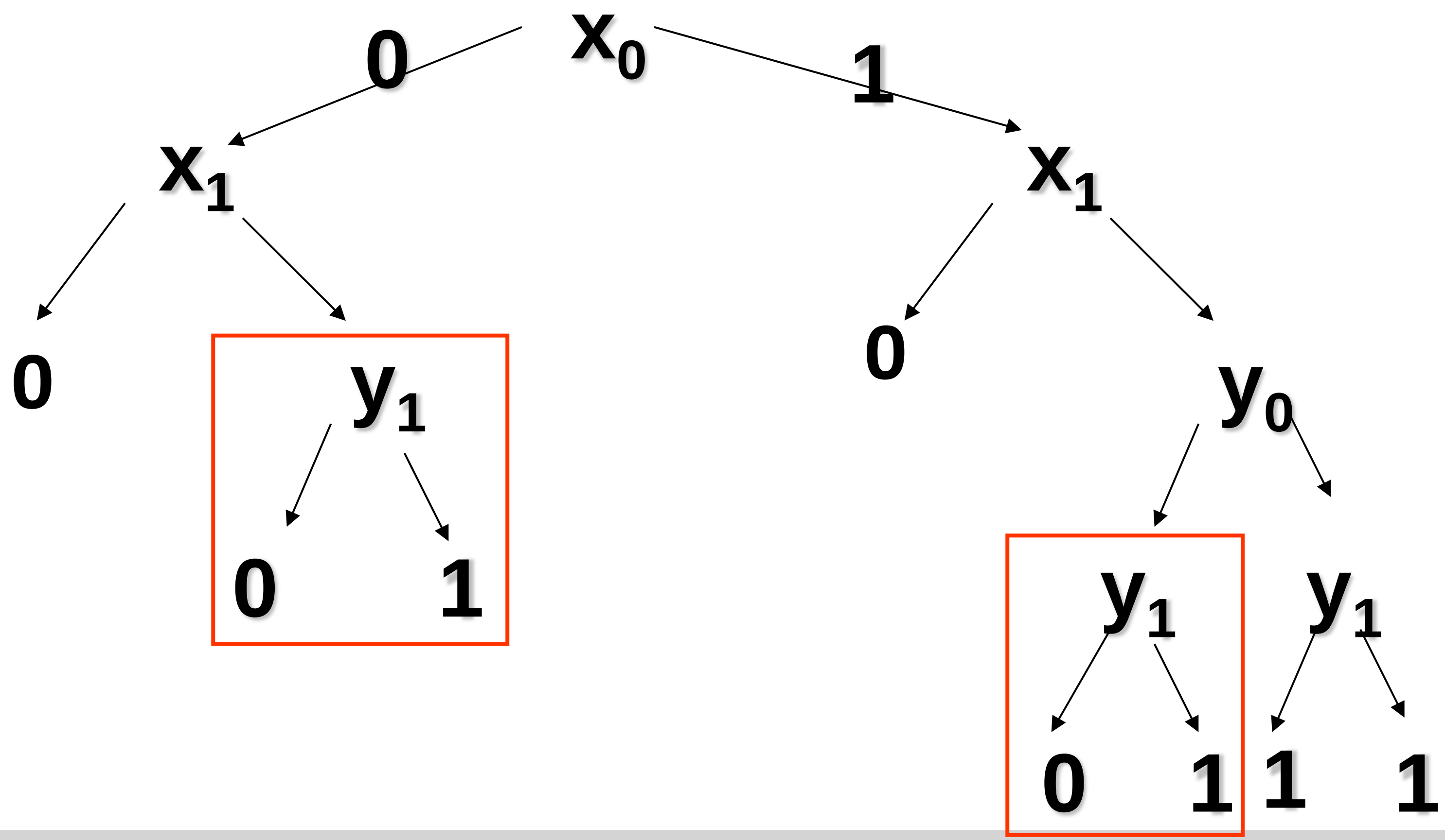
$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$



Model Checking Implementation (2)

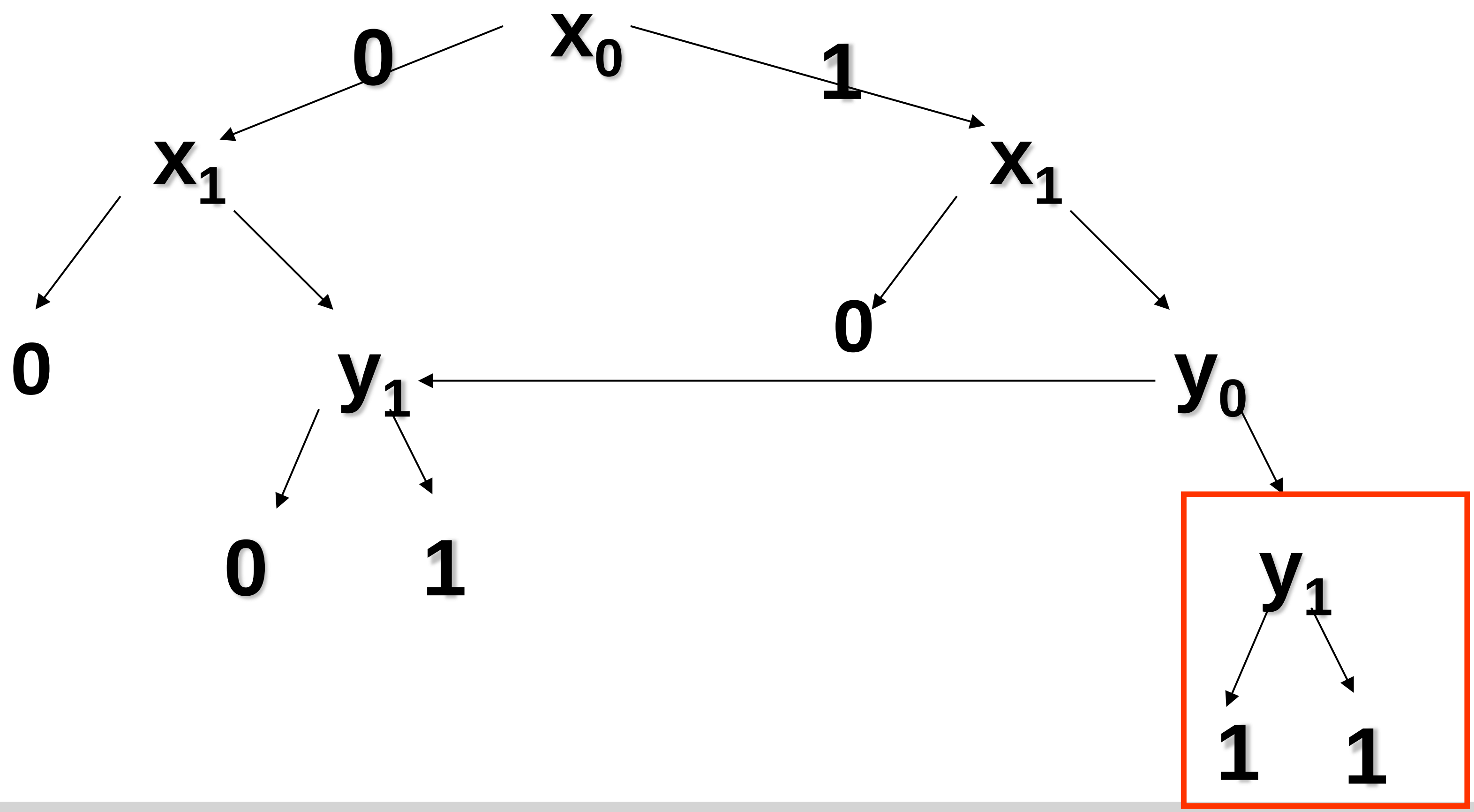


$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$



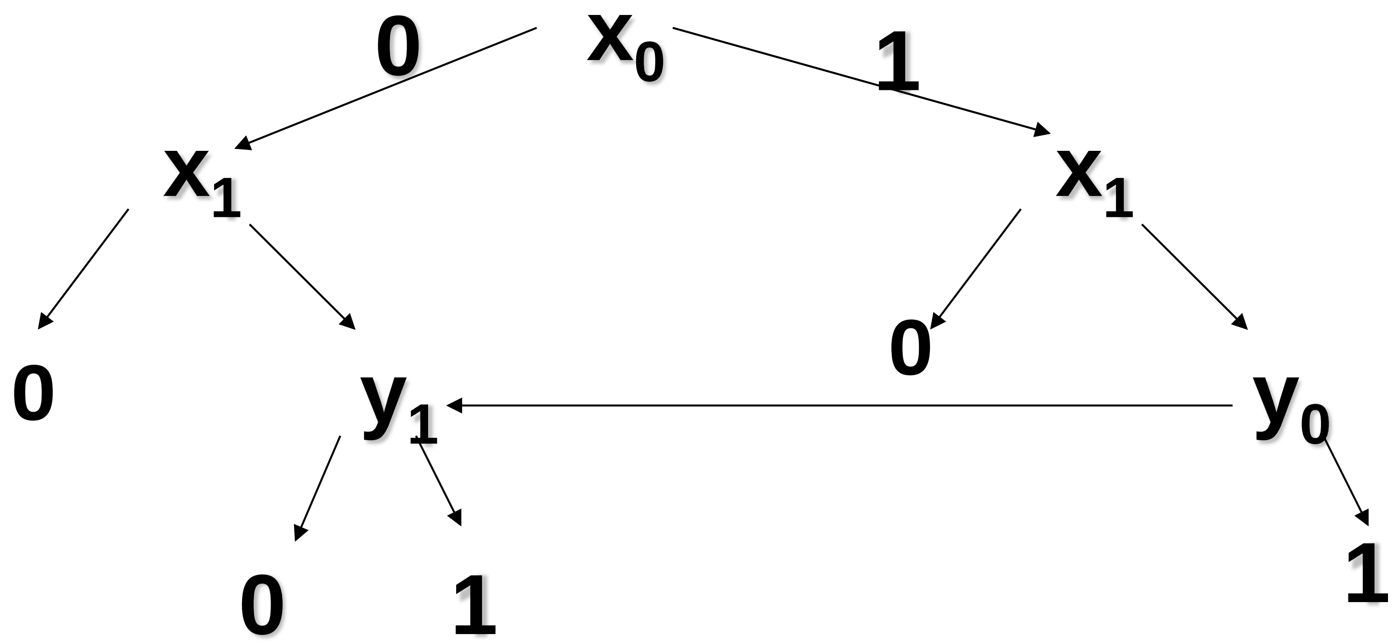
Model Checking Implementation (2)

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



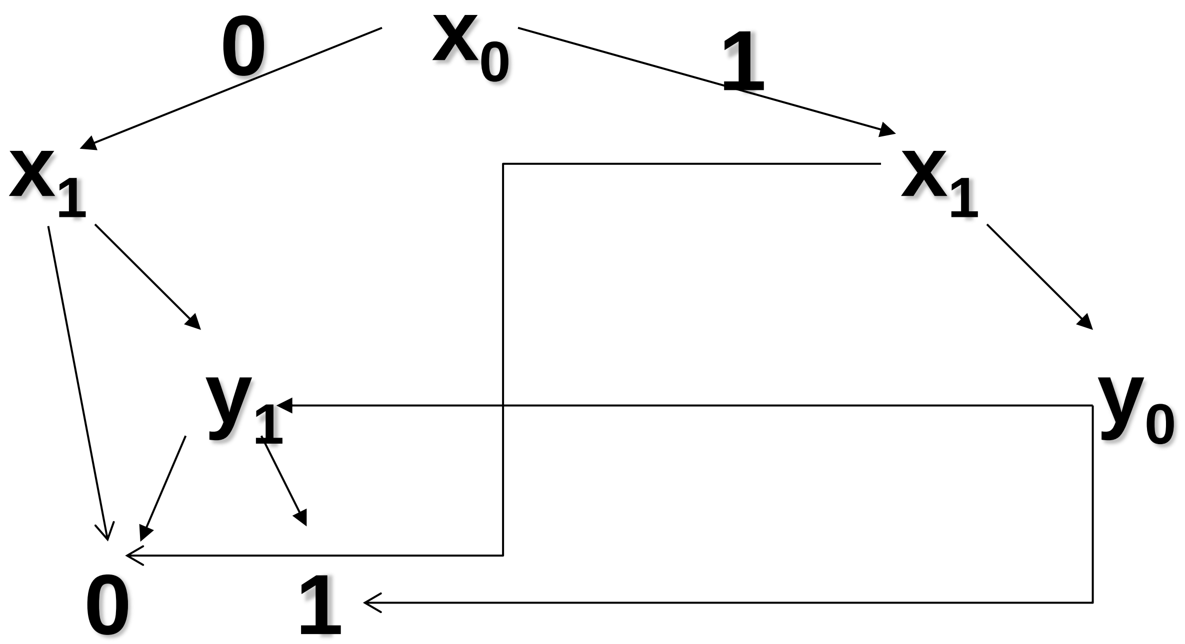
Model Checking Implementation (2)

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

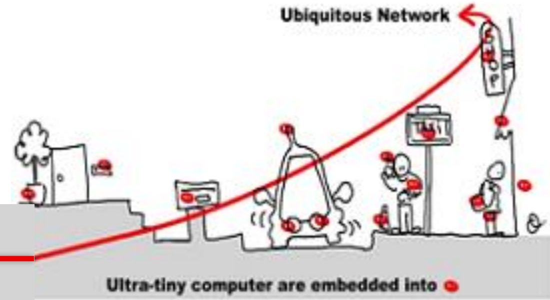


Model Checking Implementation (2)

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

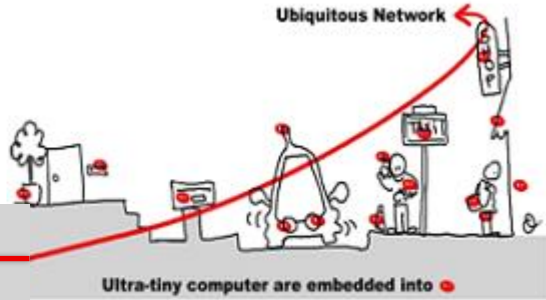


Model Checking Implementation(3)



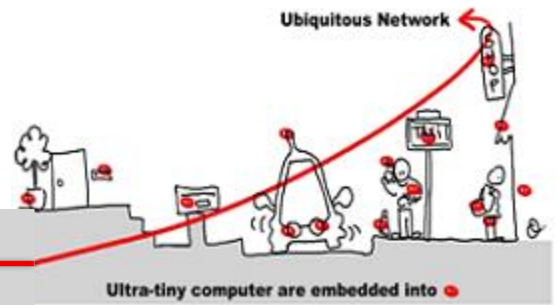
- Implicit representation of the of states set and of the transition relation of automata with BDD.
- BDD allows
  - canonical representation
  - test of emptiness immediate (bdd =0)
  - complementarity immediate (1 = 0)
  - union and intersection not immediate
  - Pre immediate

Model Checking Implementation (4)



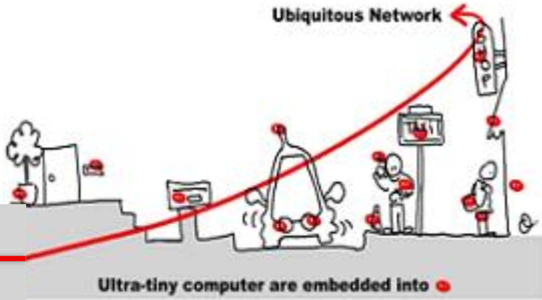
- But BDD efficiency depends on the number of variables
- Other method: SAT-Solver
  - Sat-solvers answer the question: given a propositional formula, is there exist a valuation of the formula variables such that this formula holds
  - first algorithm (DPLL) exponential (1960)

Model Checking Implementation (4)



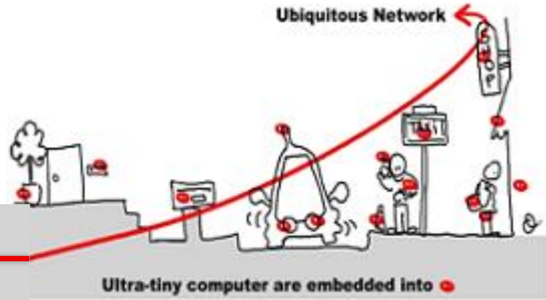
- SAT-Solver algorithm:
  - formula ➔ CNF formula ➔ set of clauses
  - heuristics to choose variables
  - deduction engine:
    - propagation
    - specific reduction rule application (unit clause)
    - Others reduction rules
  - conflict analysis + learning

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 $\mathcal{M} \models_n \Phi$ ($n < |S| . 2^{|\Phi|}$)

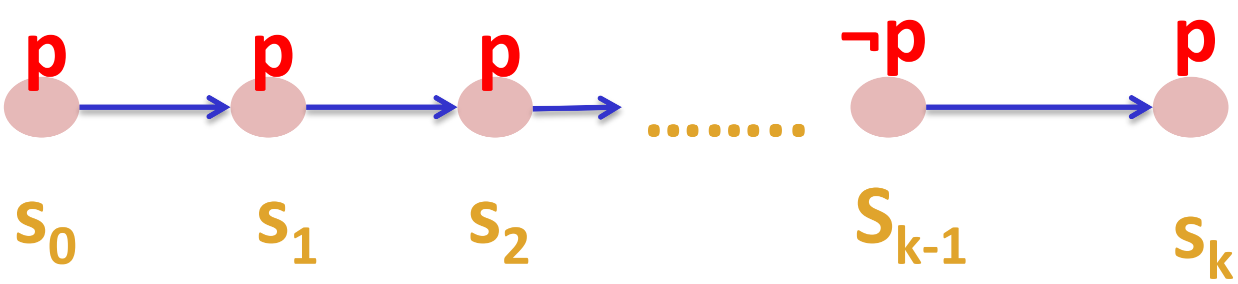
Bounded Model Checking



- SAT-Solver are used in complement of implicit (BDD based) methods.
- $\mathcal{M} \mid= \Phi$
  - 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:

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

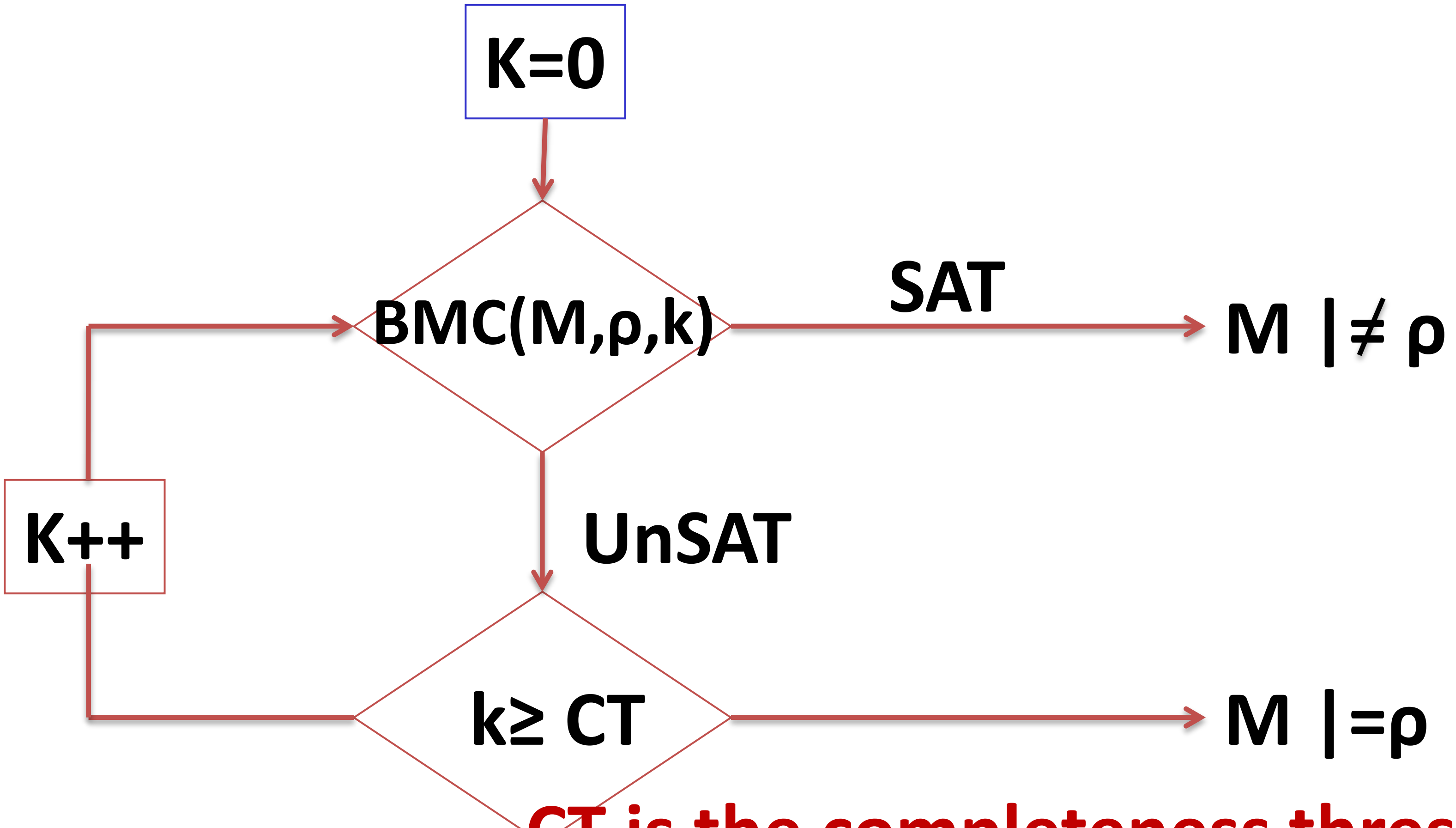
The property **p** fails in one of the k steps

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

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

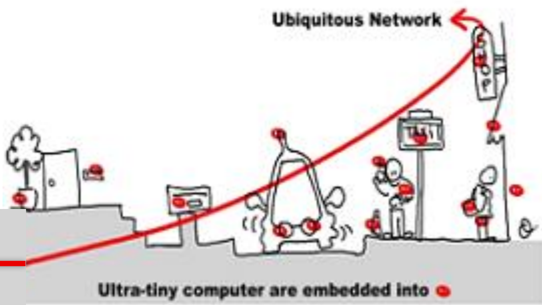
$$\Omega(k) = I(s_0) \wedge \left(\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \right) \wedge \left(\bigvee_{i=0}^{k} \neg p(s_i) \right)$$

Bounded Model Checking



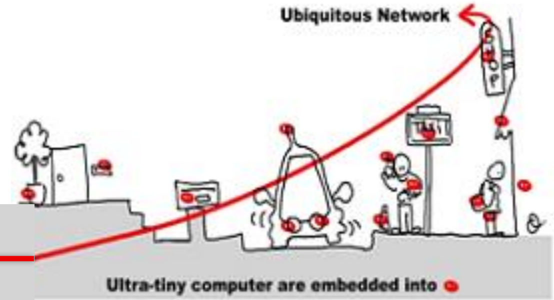
CT is the completeness threshold

Bounded Model Checking

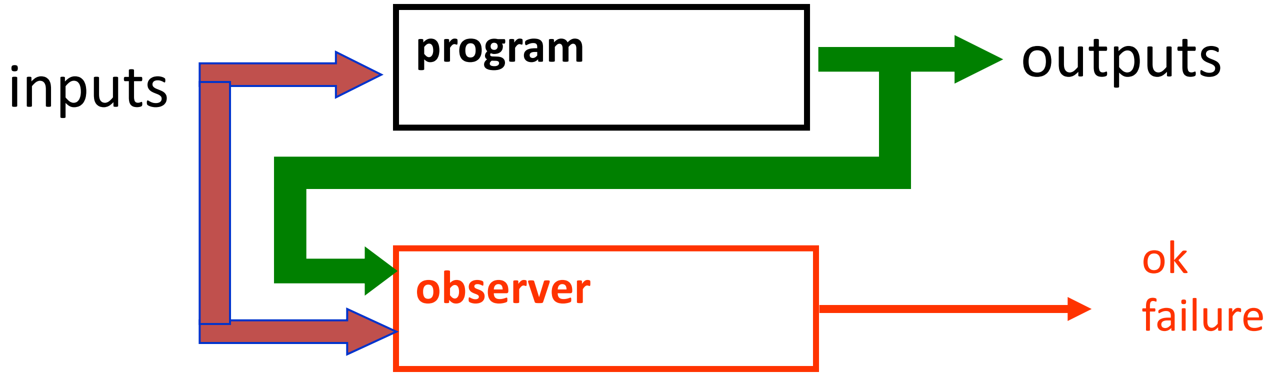


- Computing CT is as hard as model checking.
- Idea: Compute an over-approximation to the actual CT
  - Consider the system *as a graph.*
  - Compute *CT from structure of the graph.*
- Example: for **AGρ** properties, CT is the longest shortest path between any two reachable states, starting from initial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

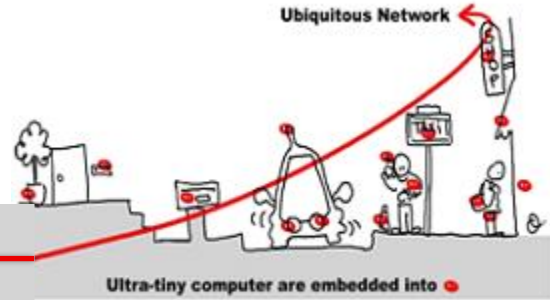
Model Checking with Observers



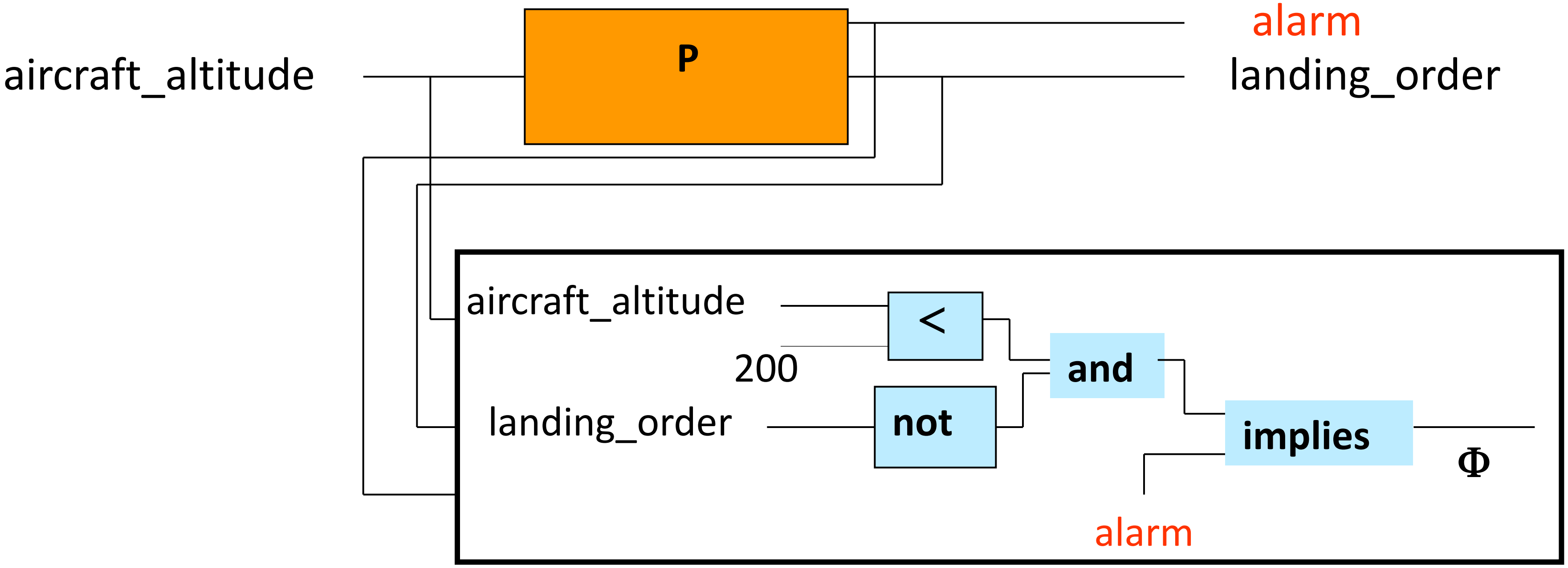
- Express safety properties as observers.
- An observer is a program which observes the program and outputs ok when the property holds and failure when its fails



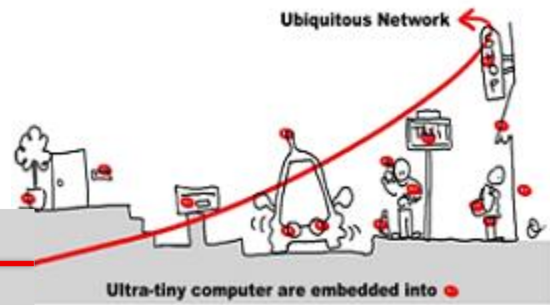
Model Checking with observers (2)



P: aircraft autopilot and security system

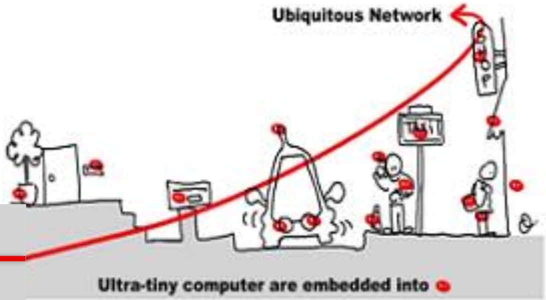


Properties Validation

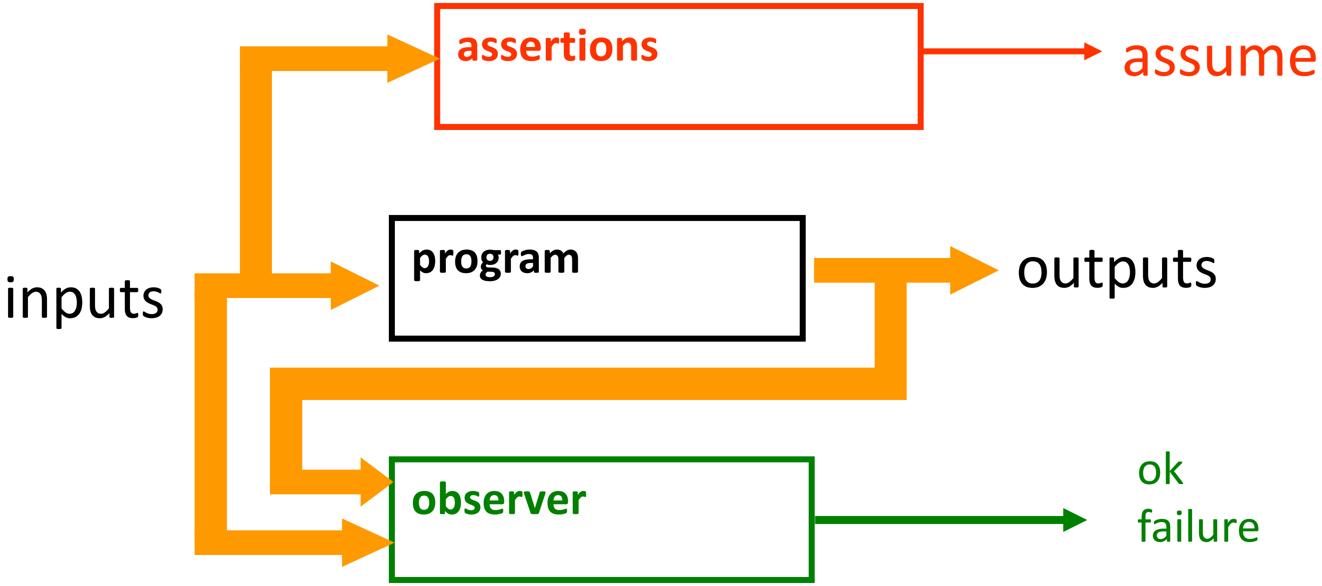


- Taking into account the environment
  - without any assumption on the environment, proving properties is difficult
  - but the environment is indeterminist
    - Human presence no predictable
    - Fault occurrence
    - …
  - Solution: use assertion to make hypothesis on the environment and make it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

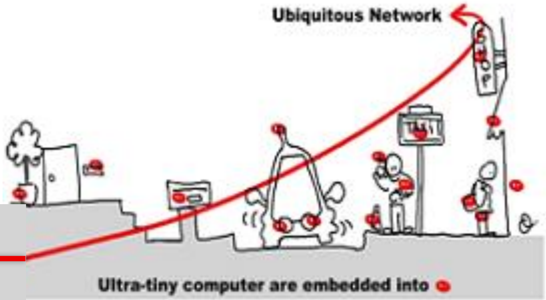
Properties Validation (2)



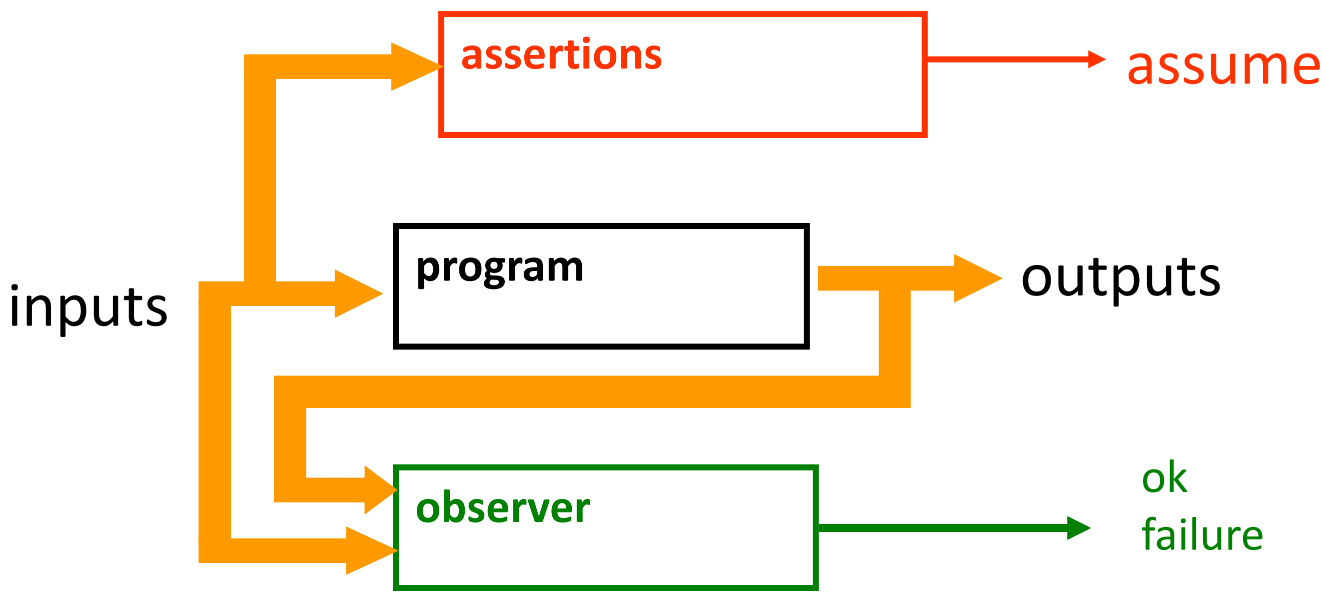
- Express safety properties as observers.
- Express constraints about the environment as assertions.



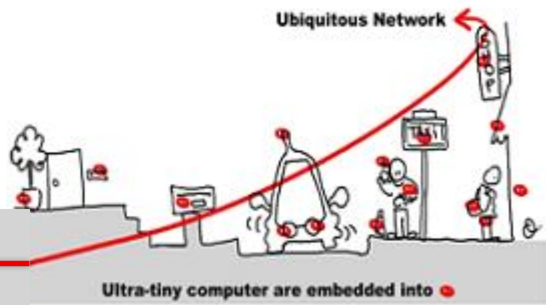
Properties Validation (3)



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

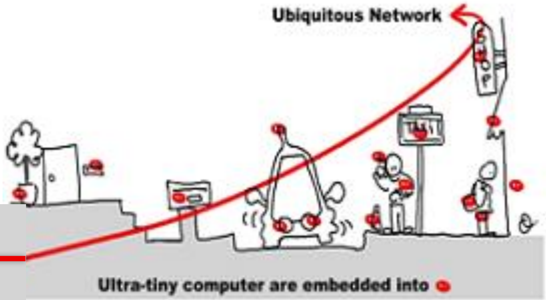


Outline



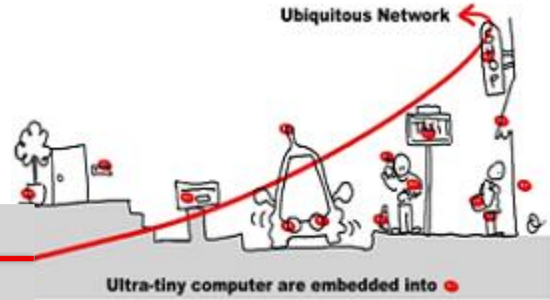
1. Critical system validation
2. Model-checking solution
    1. Model specification
    2. Model-checking techniques
3. Application to middleware for IoT (~Wcomp)
    1. Introduction in middleware design of synchronous components to allow validation
    2. Synchronous /asynchronous issues

Practical Issues



Application to Middleware for IoT

Practical Issues



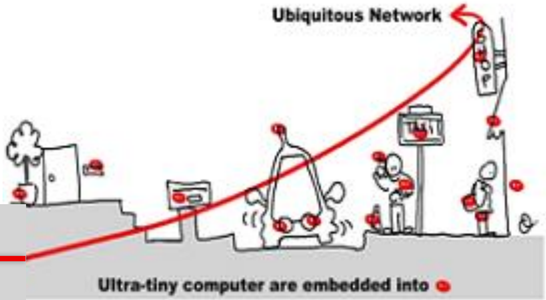
Our challenges are:

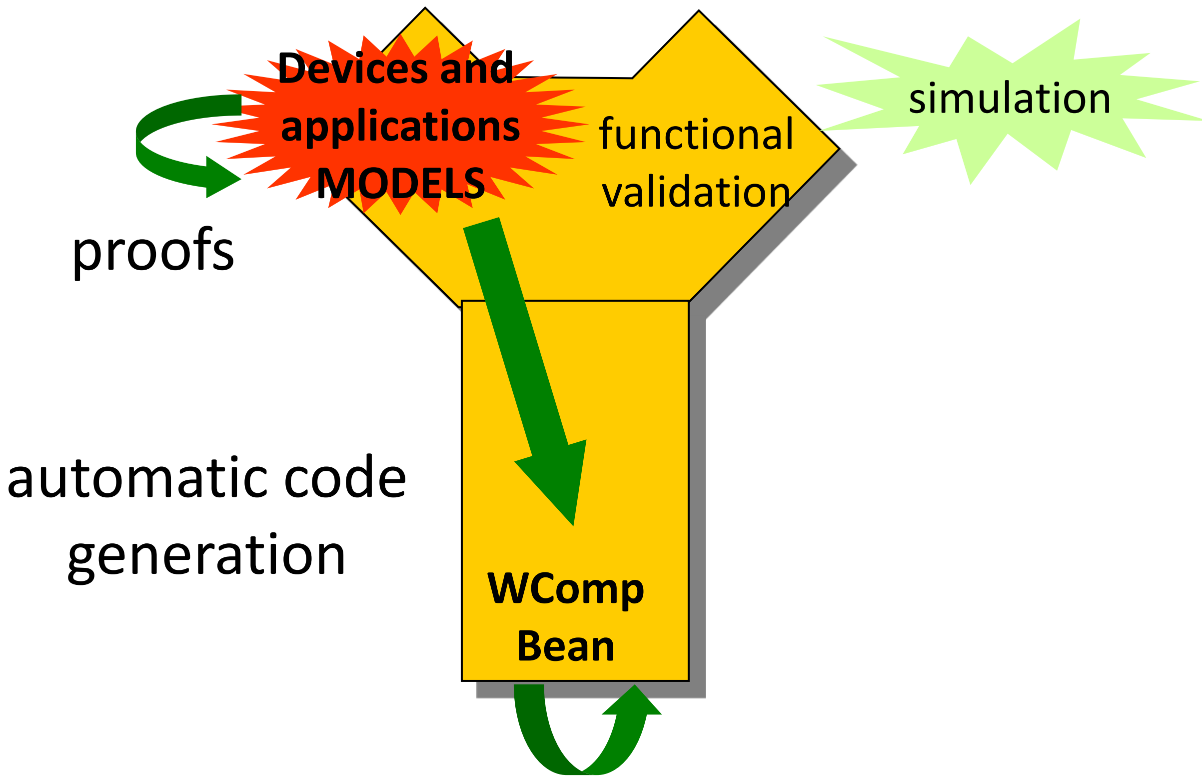
•How to maintain consistency in spite of concurrent accesses by multiple services and multiple applications to a common Entity of Interest ?

• How to deal with dynamic context changes ?

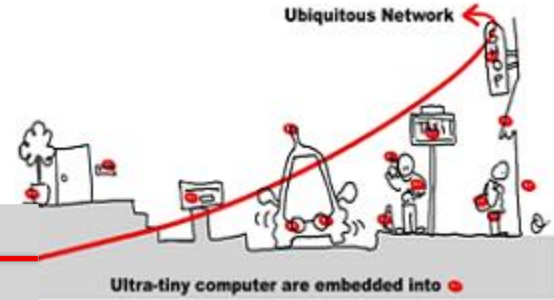
•Introduce in Middleware specific components (synchronous components) on which model checking technique applies

Application to Middleware





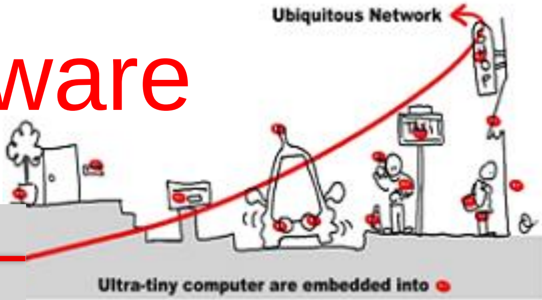
Synchronous Models



To sum up :

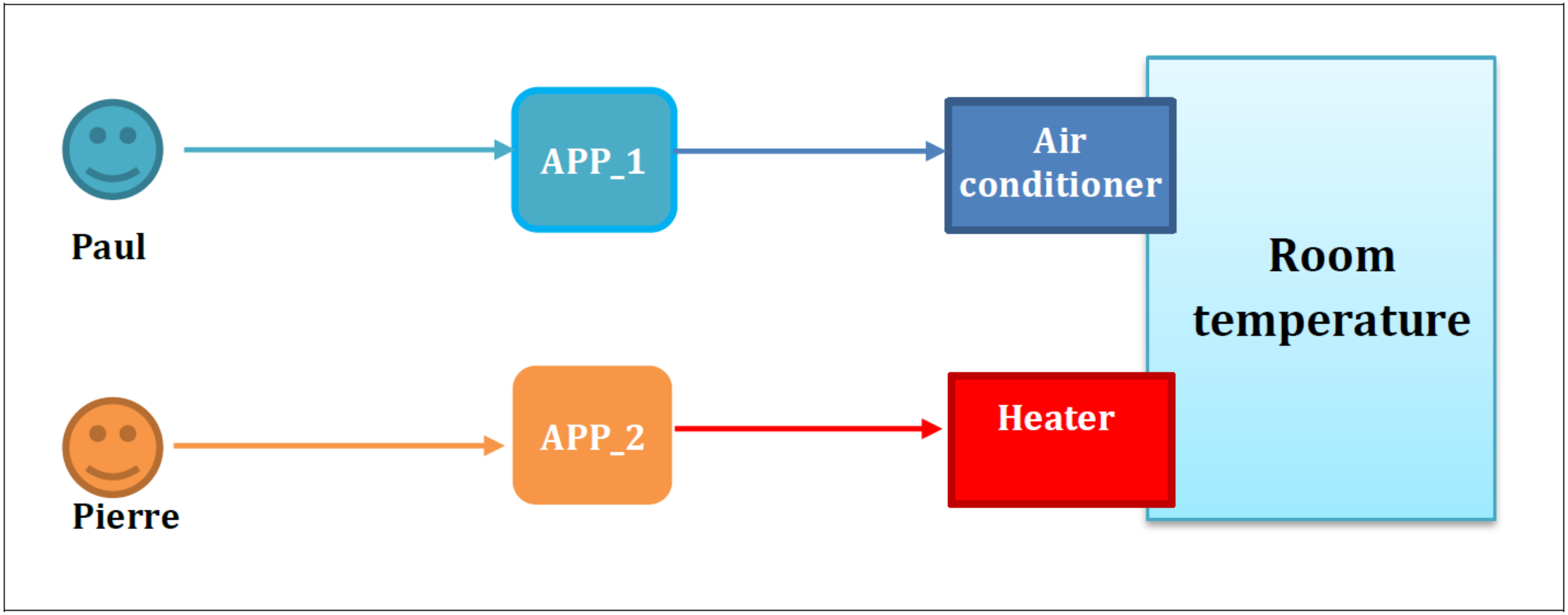
1. Synchronous models can be designed as **event-driven controllers** or **as data flow operator networks**
2. They always represent automata
3. Model-checking techniques apply

Application to Adaptive Middleware



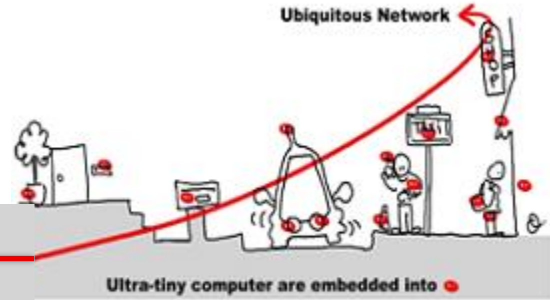
- Our goal is to ensure safety for applications using and managing services.
- Devices will have a synchronous component to allow model-checking techniques application as validation
- Synchronous component to express constraints between concurrent services
- Synchronous parallelism as composition

Use Case



Entity of interest: temperature controlled room

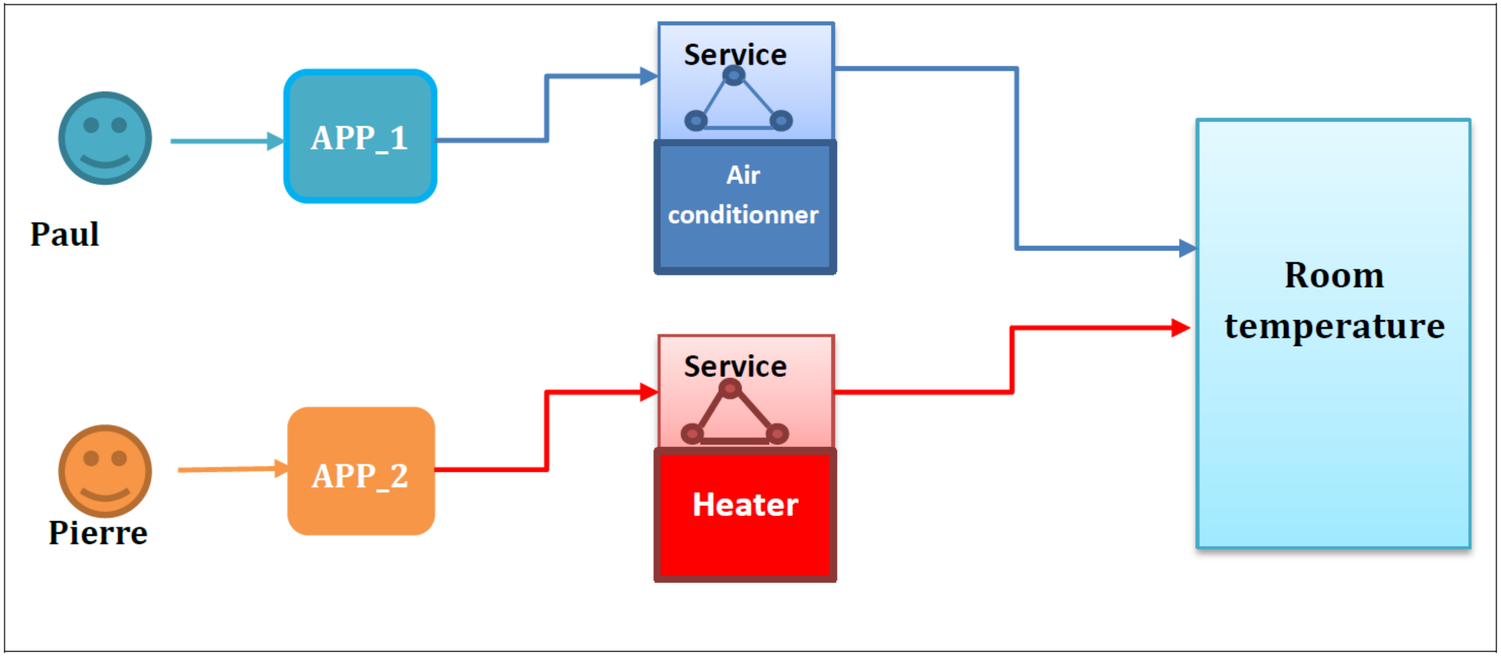
Use Case

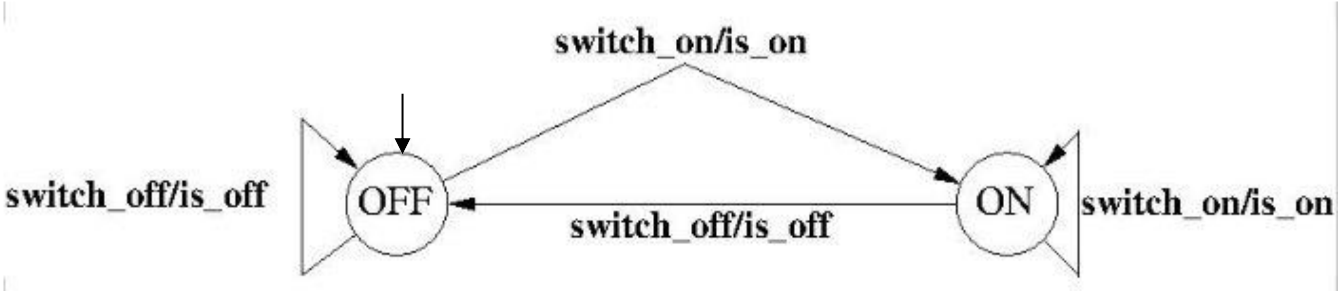


- **Use case**: manage room temperature
  1. Temperature controlled by 2 internet objects: air condionner and heater
  2. Two applications use these devices:
     1. APP1: to cool the room
     2. APP2: to warm the room

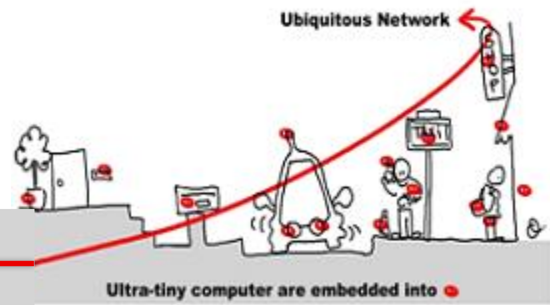
     simultaneous
  3. Constraints:
     - APP1 is launch by Paul smartphone
     - APP2 is launch by Pierre smartphone
     - The air conditioner and the heater cannot be switch on simultaneously

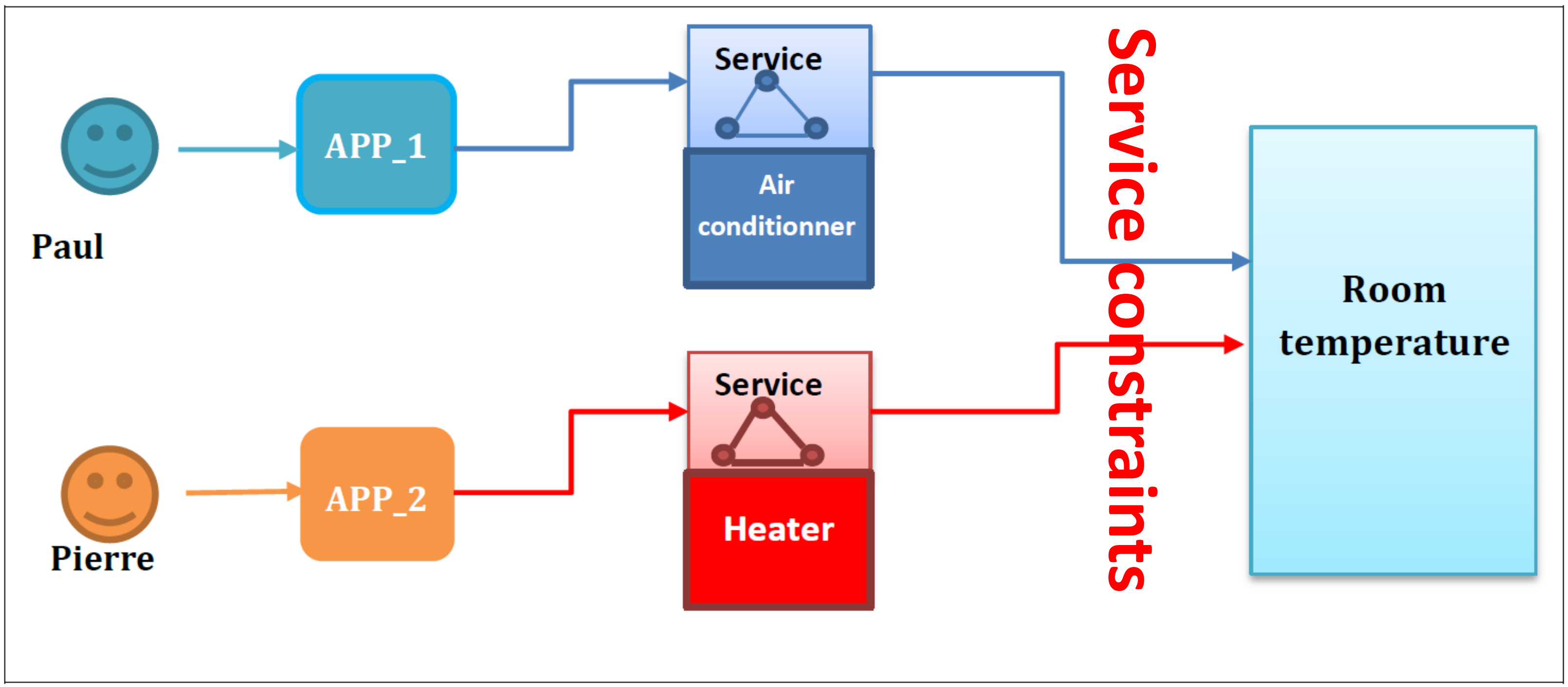
Use Case Implementation





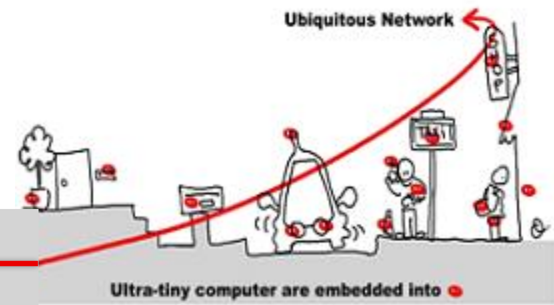
Use Case Implementation





Application constraints

Use Case Implementation

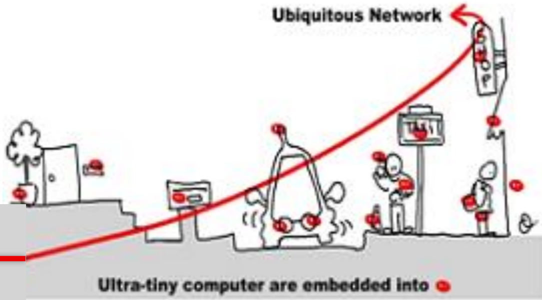


How specify the Heater synchronous model ?

How specify both device and application constraints as synchronous models ?

Solution: use a synchronous language

First Solution: SCADE

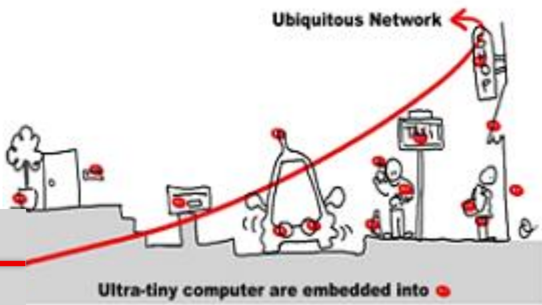


- Scade (Safety-Critical Application Development Environment) has been developed to address safety-critical embedded application design
- The Scade suite KCG code generator has been qualified as a development tool according to DO-178B norm at level A.

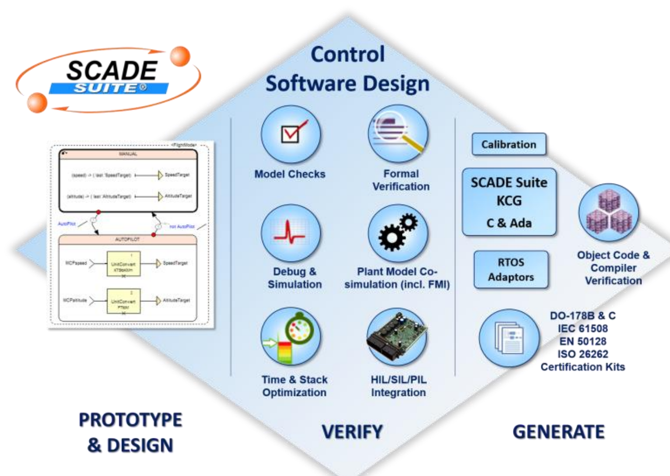
SCADE

- 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

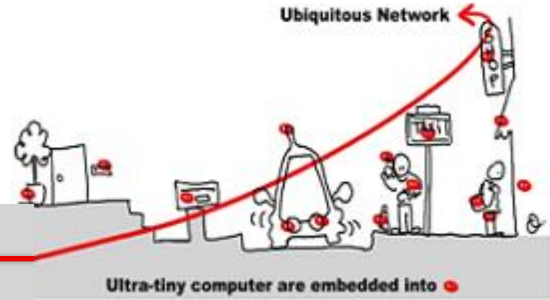
SCADE



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

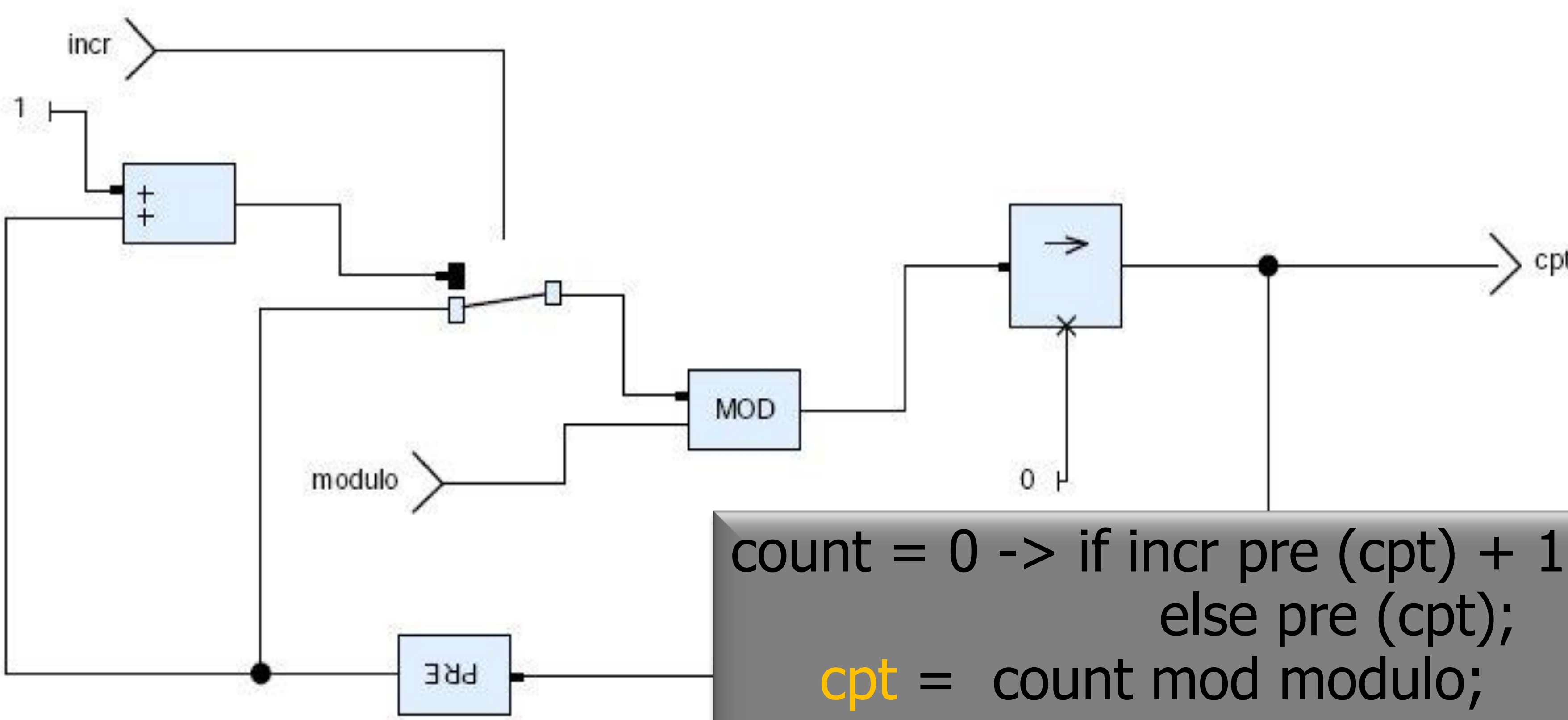


Modulo Counter



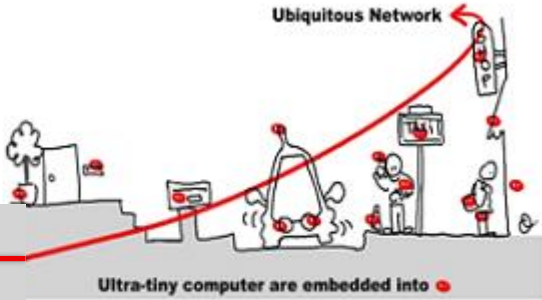

```
operator MCounter (incr:bool; modulo : int)
                              returns (cpt:int);
   var count : int;

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

Modulo Counter



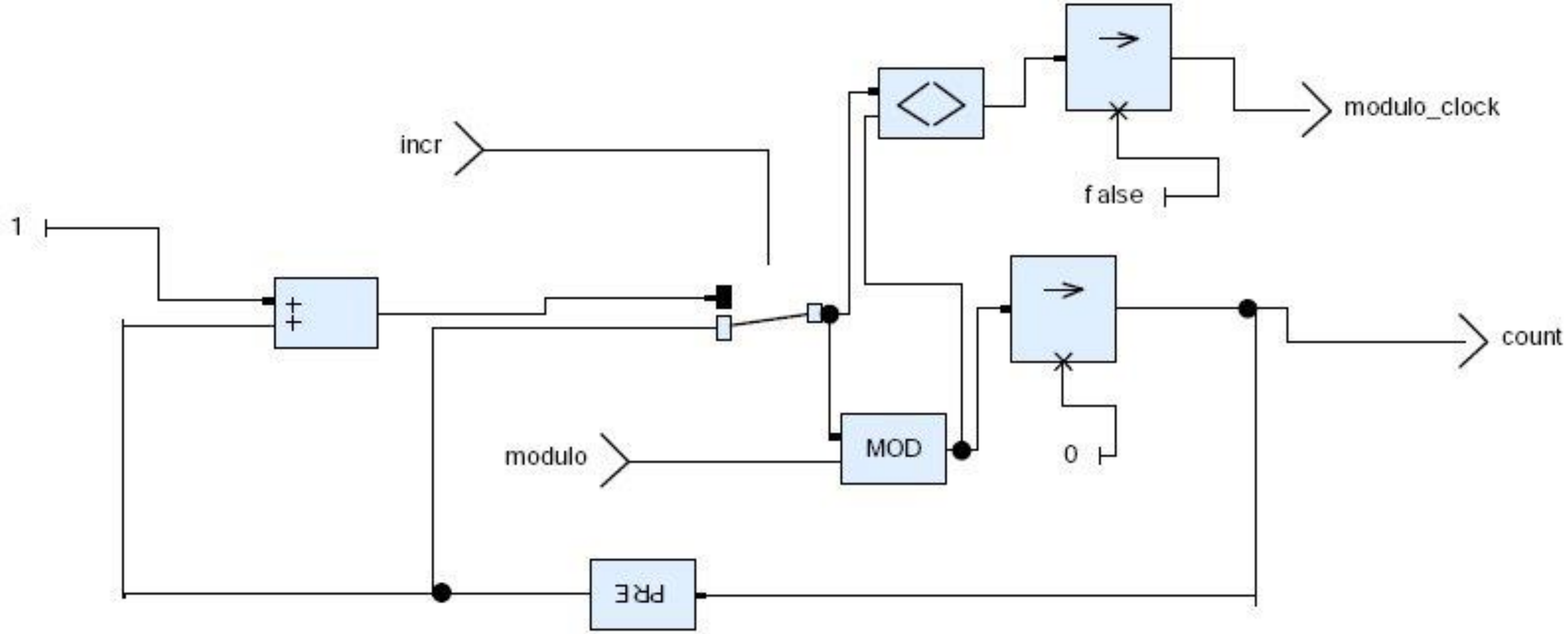

```
count = 0 -> if incr pre (cpt) + 1
                 else pre (cpt);
    cpt =  count mod modulo;
```

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;
```

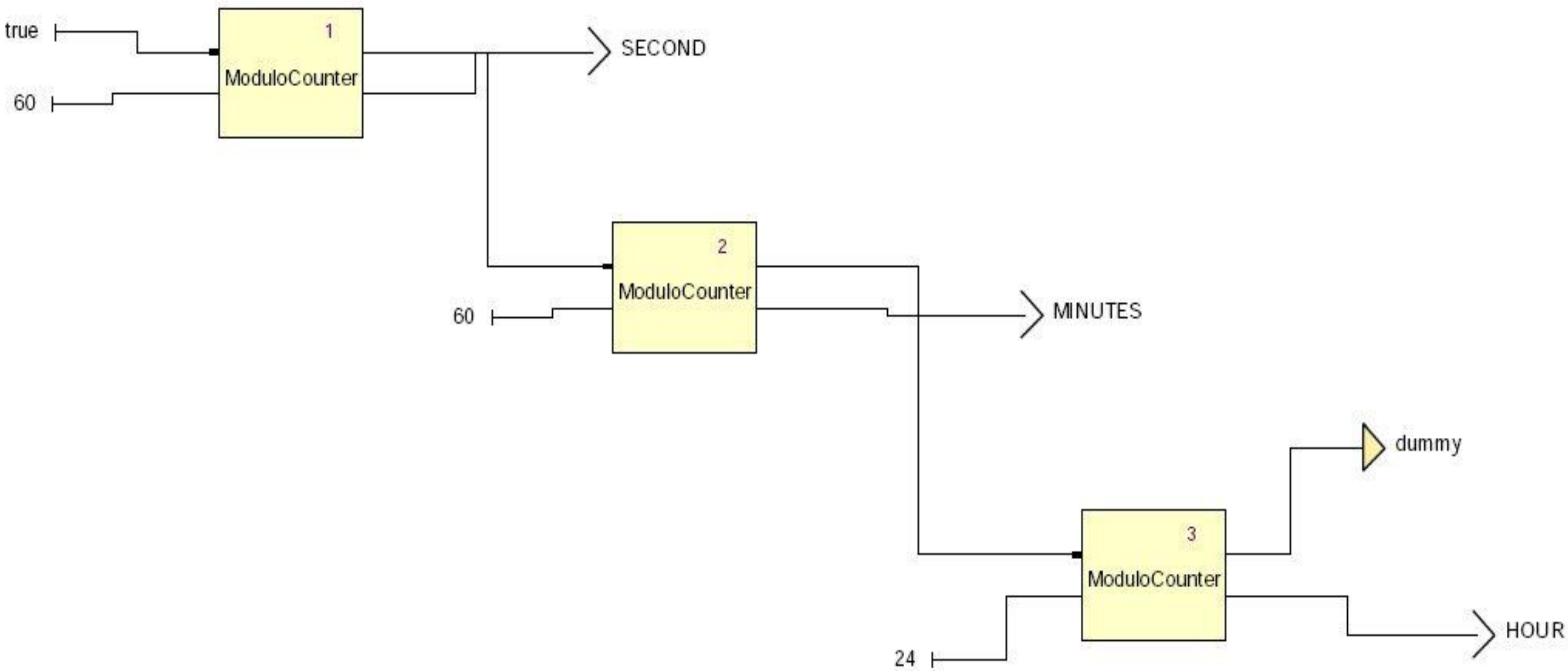
Modulo Counter Clock



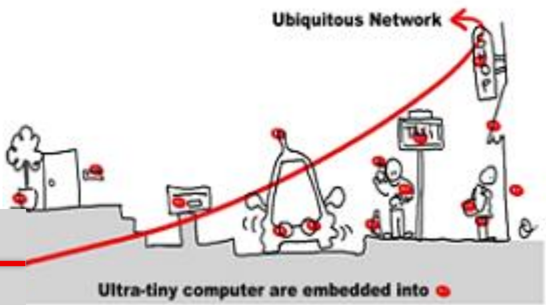
Timer

```
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

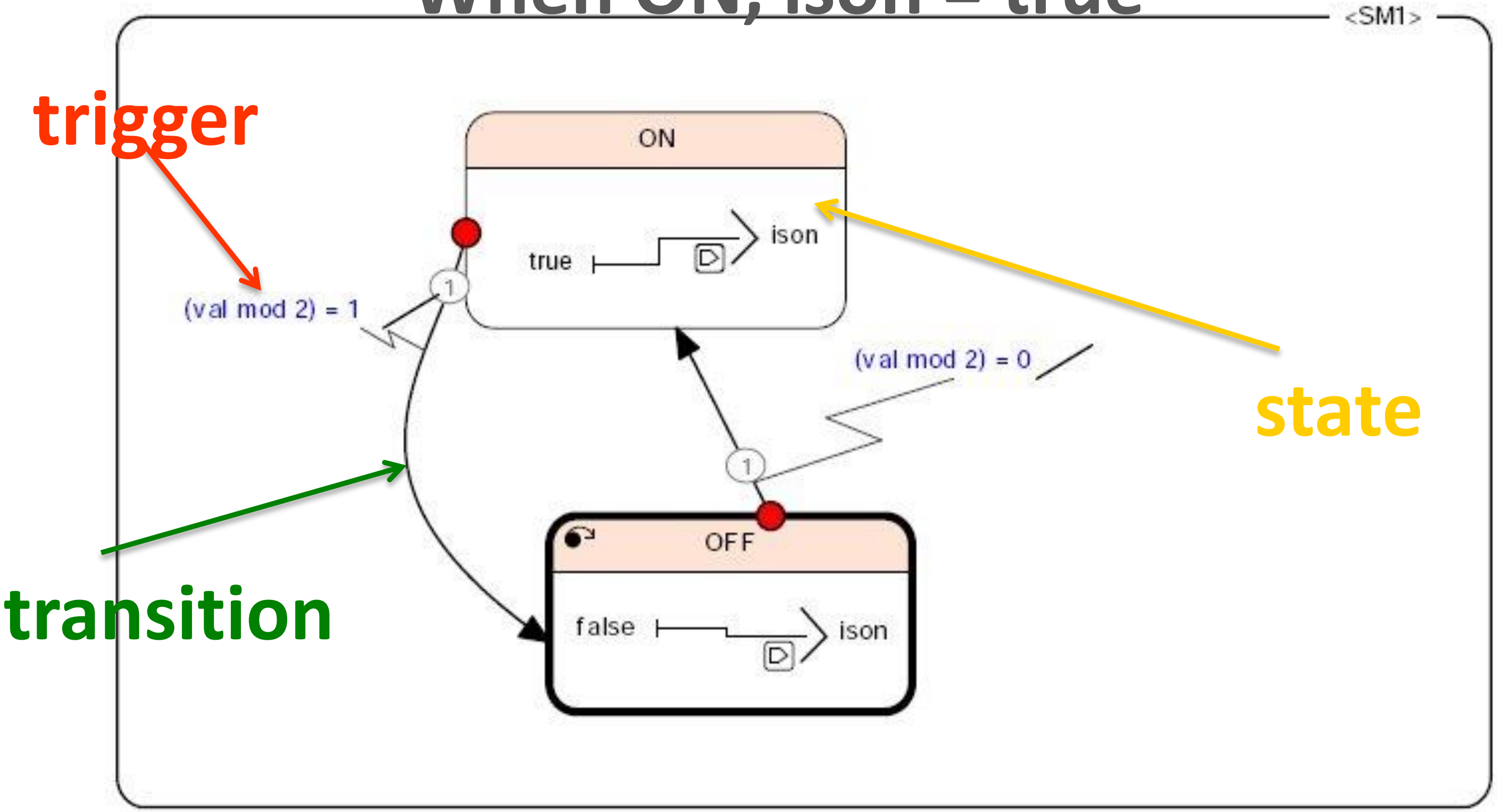
SCADE: state machines



- 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

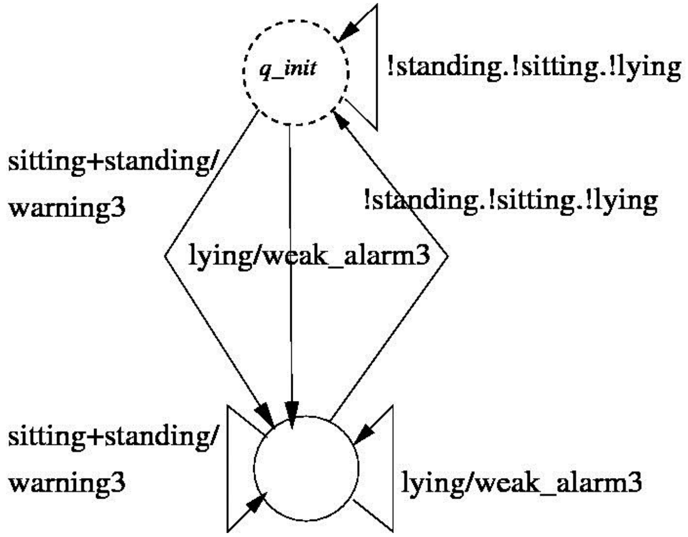


When off, ison = false

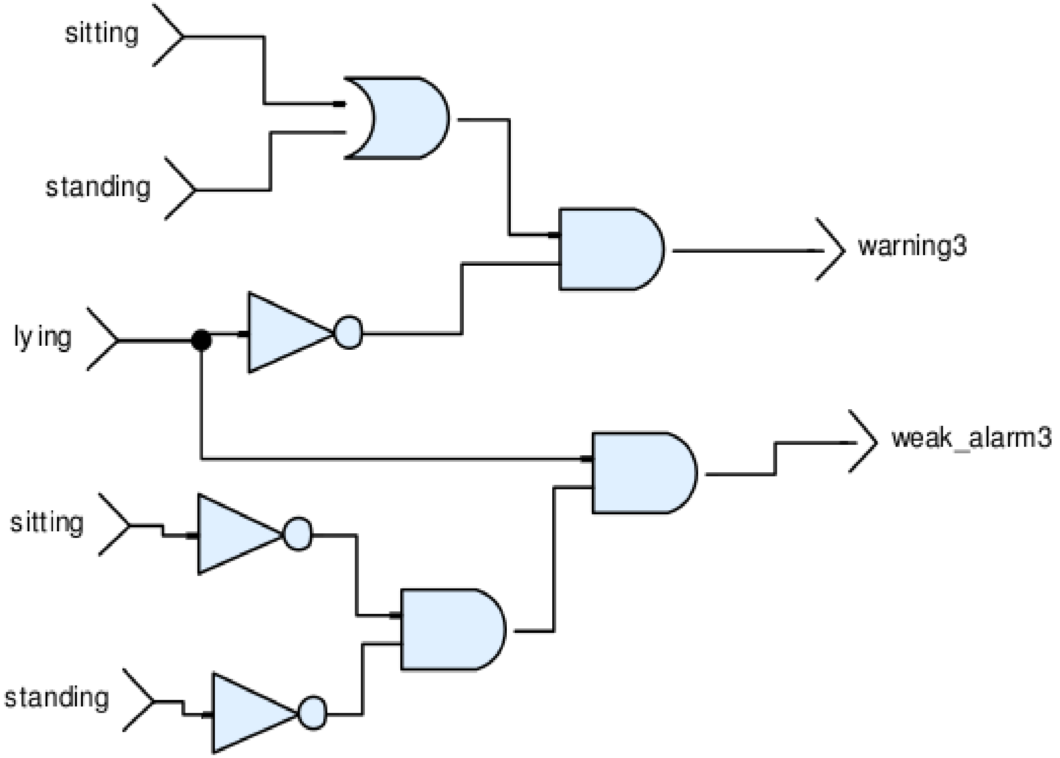
SCADE: model checking

Observer technique

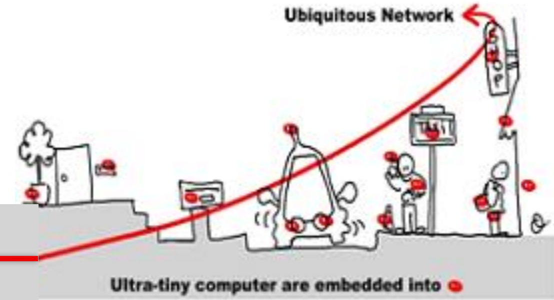
posture model



posture model specification in scade

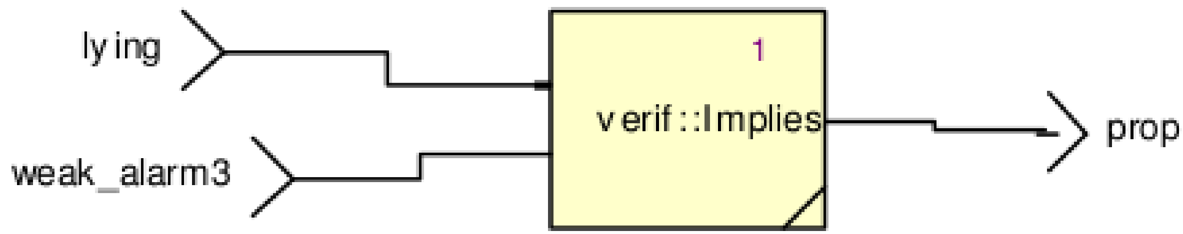


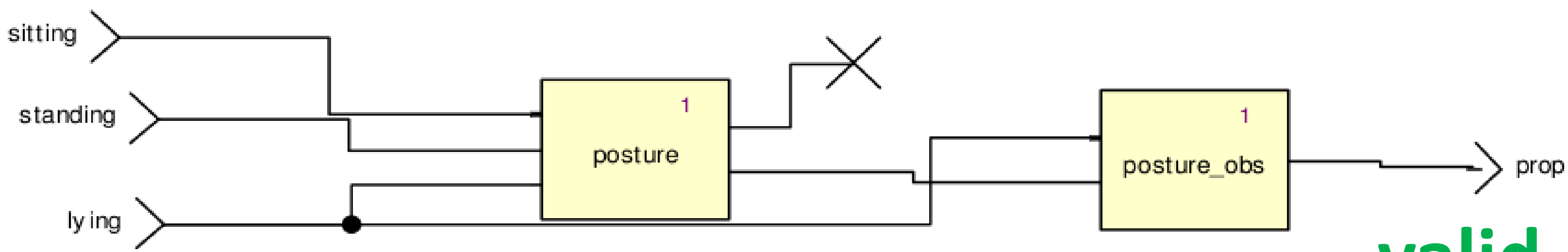
SCADE: model checking



Observer technique

posture observer



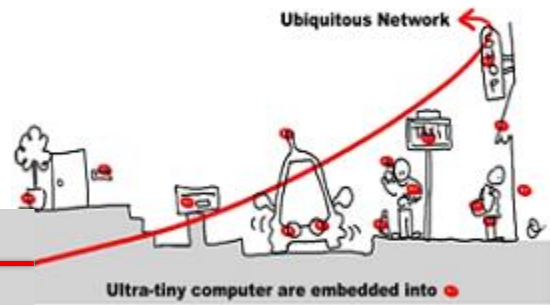


valid

posture verification

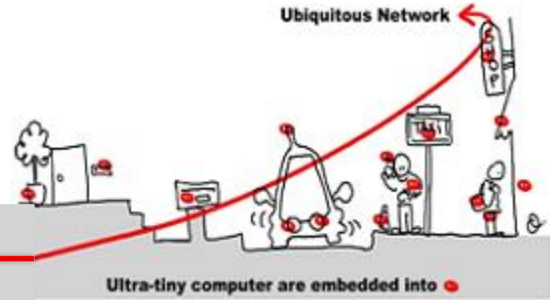
assume (lying # sitting # standing)

SCADE: code generation



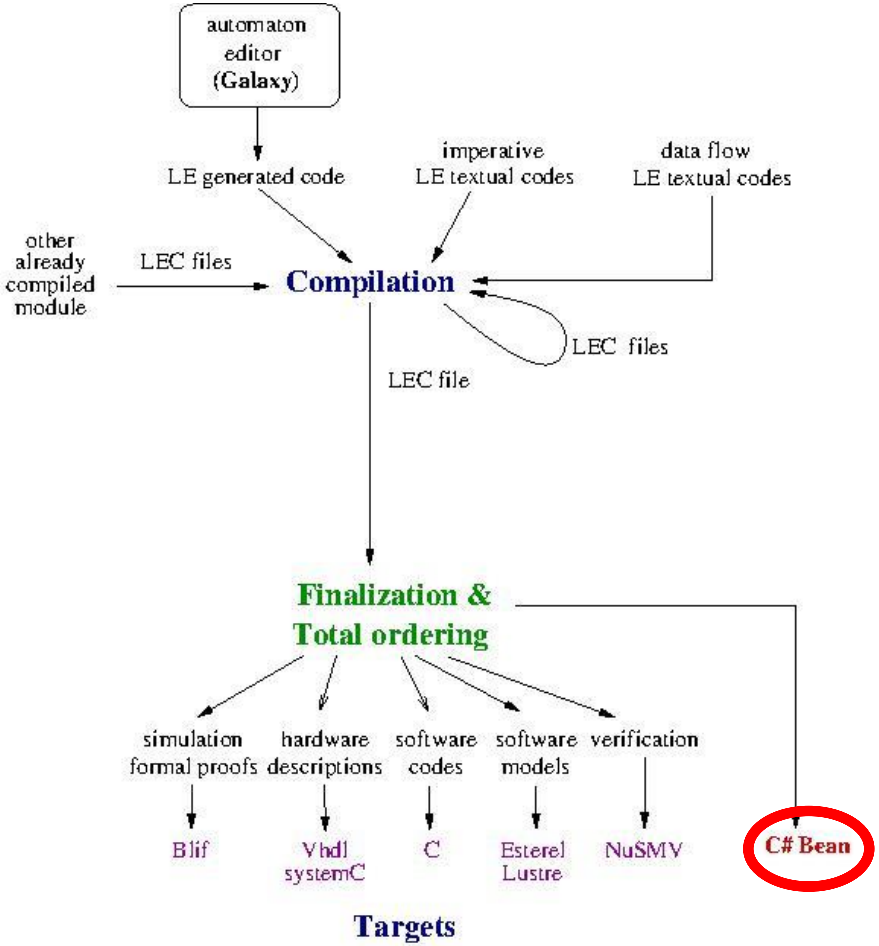
- KCG generates certifiable code (DO-178 compliance)
- Clean code, rigid structure (possible integration)
- Interfacing potential with user-defined code (c/c++)

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

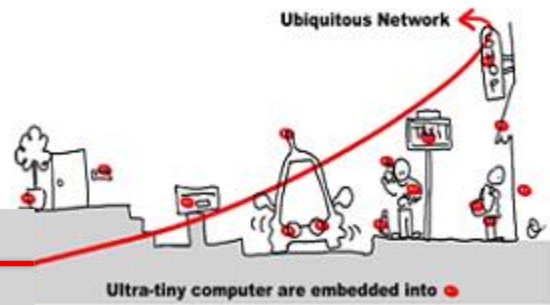
CLEM ISSUE



CLEM is a toolkit around the LE synchronous language offering:

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

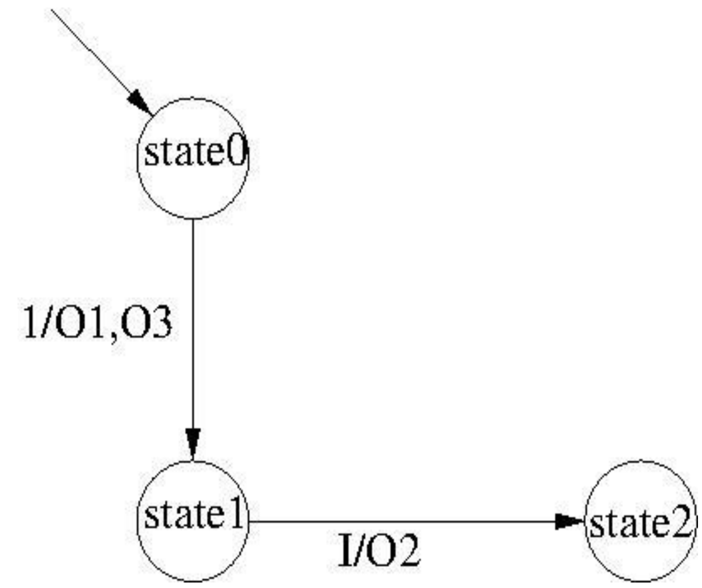
LE Language



- 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;

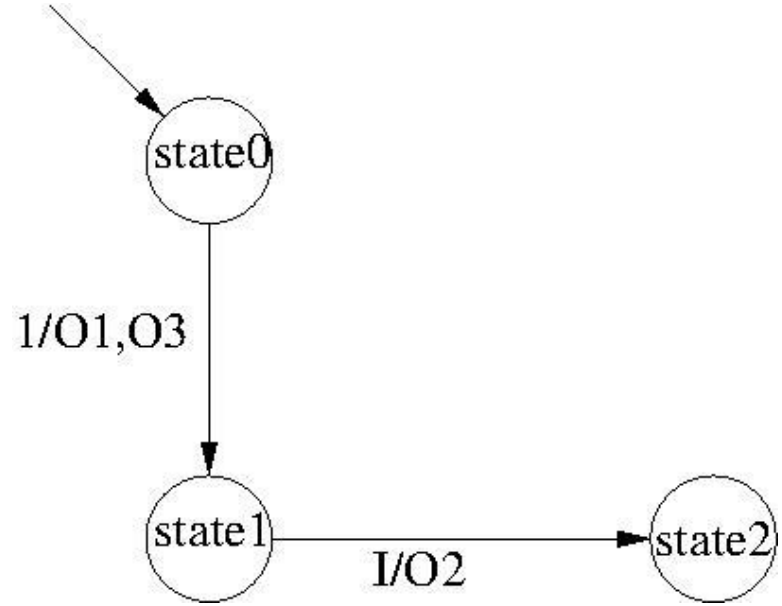
X1next = X0 and X1 or not X1 and I

or not X0 and X1;

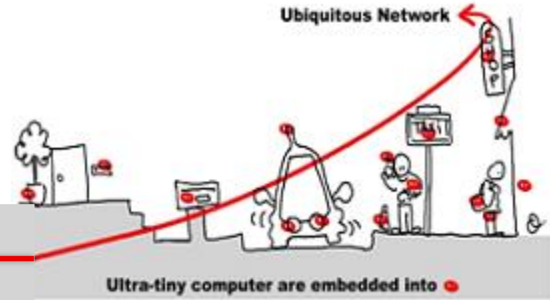
O1 = not X0 and not X1;

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

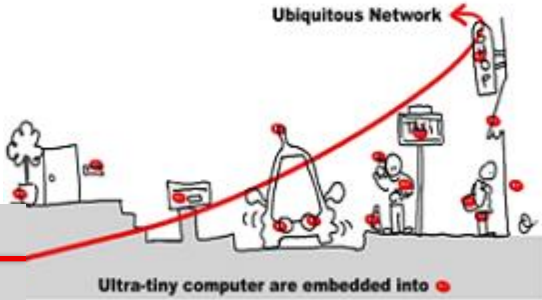
L2 = L1

L1 = I

O = L2

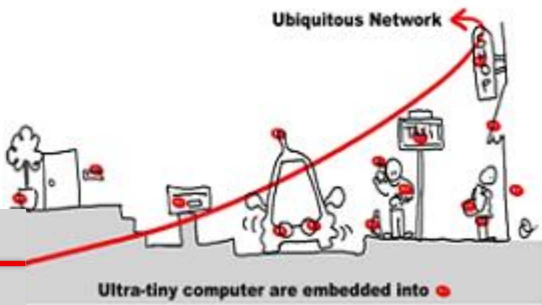
L2 = L1

LE Compilation



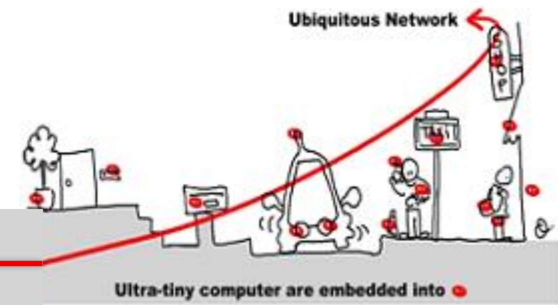
- Sorting algorithms:
  1. Apply CPM on dependency graphs of equation systems to compute ranges of evaluation levels for variables (efficient)
  2. apply fix point theory:
     - Compute variable evaluation levels as fix point of a monotonic increasing function
     - Uniqueness of fixpoints we can consider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

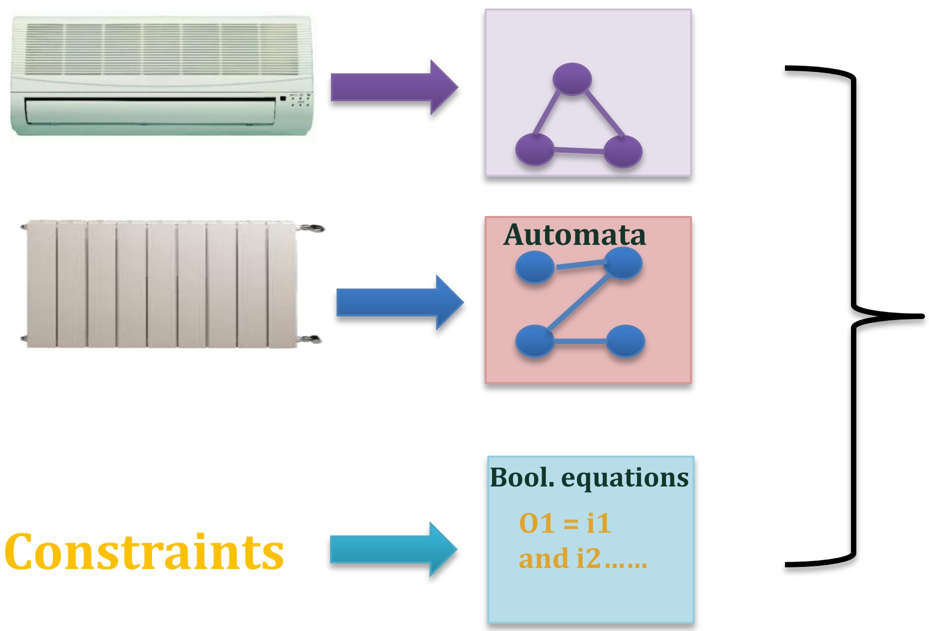
CLEM Simulation and Verification



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

Synchronous Component Design with CLEM





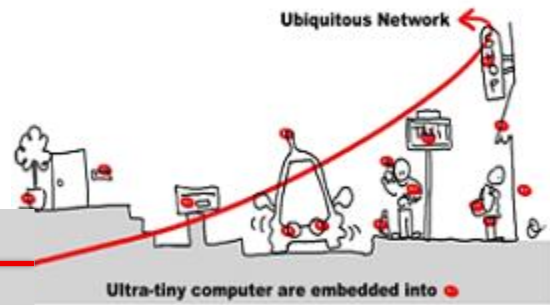
Synchronous modeling

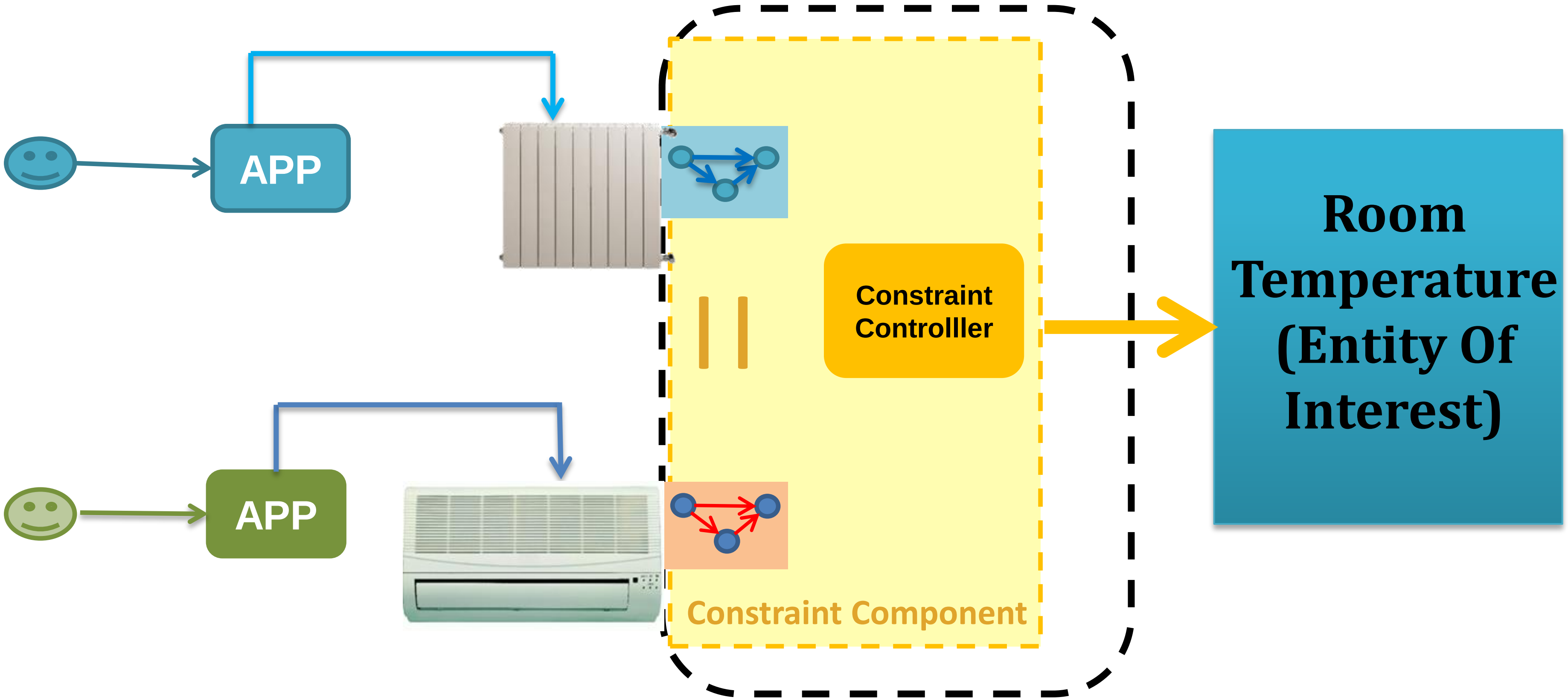
Explicit Mealy machine designed with Galaxy

or

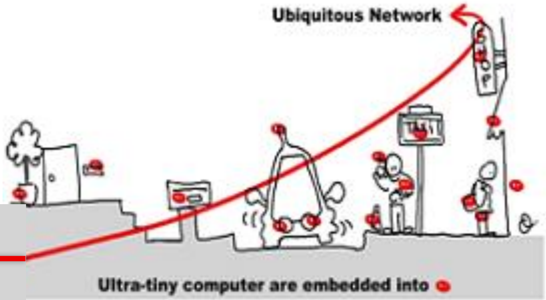
Implicit Mealy machine designed as Boolean equations in Clem

Solution:Constraint Component

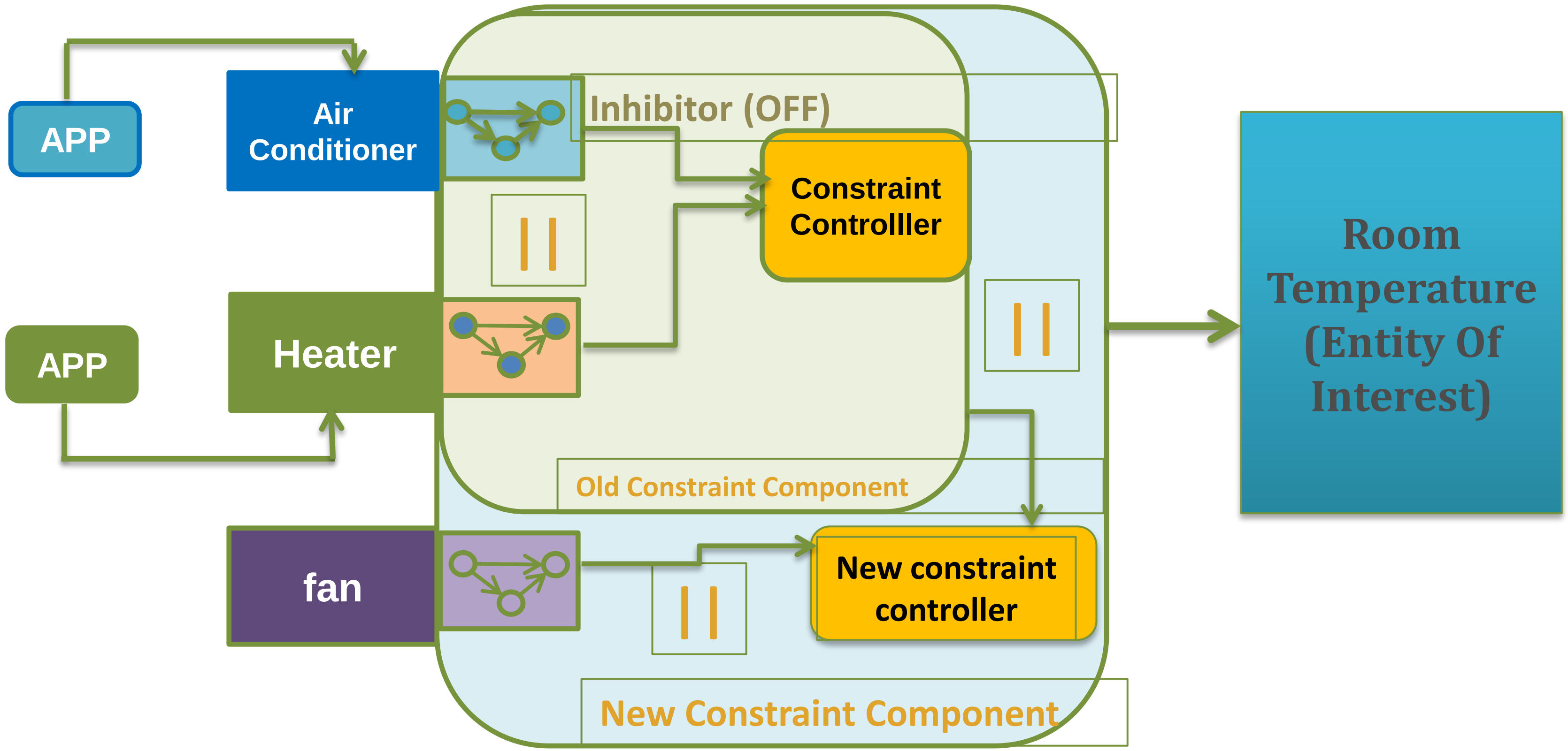




Solution: Constraint Component

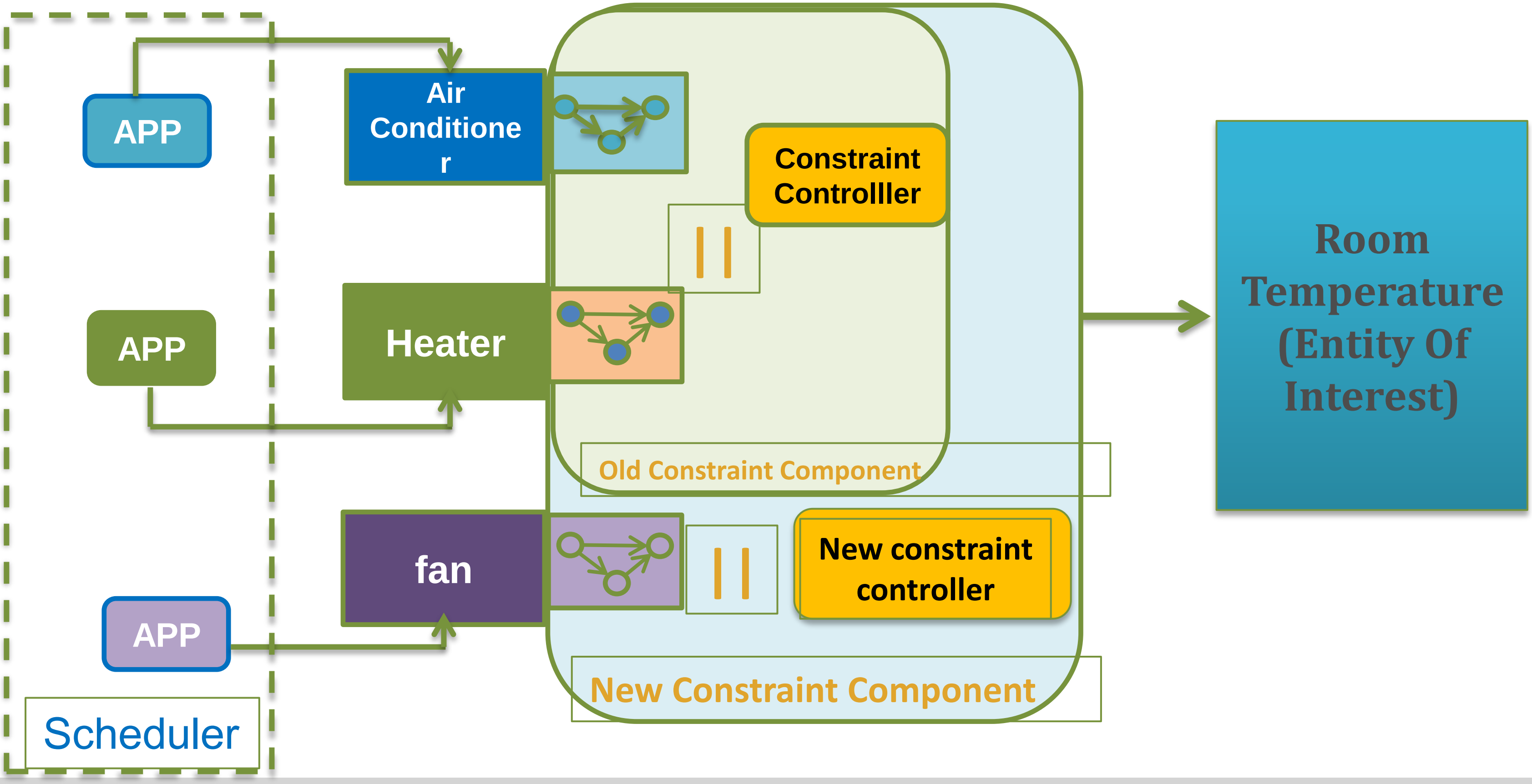


Dynamicity: Appearance of a new device

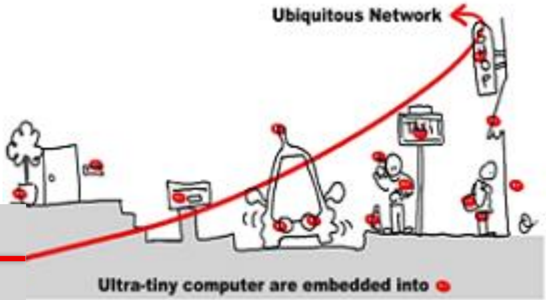


Solution: Constraint Component

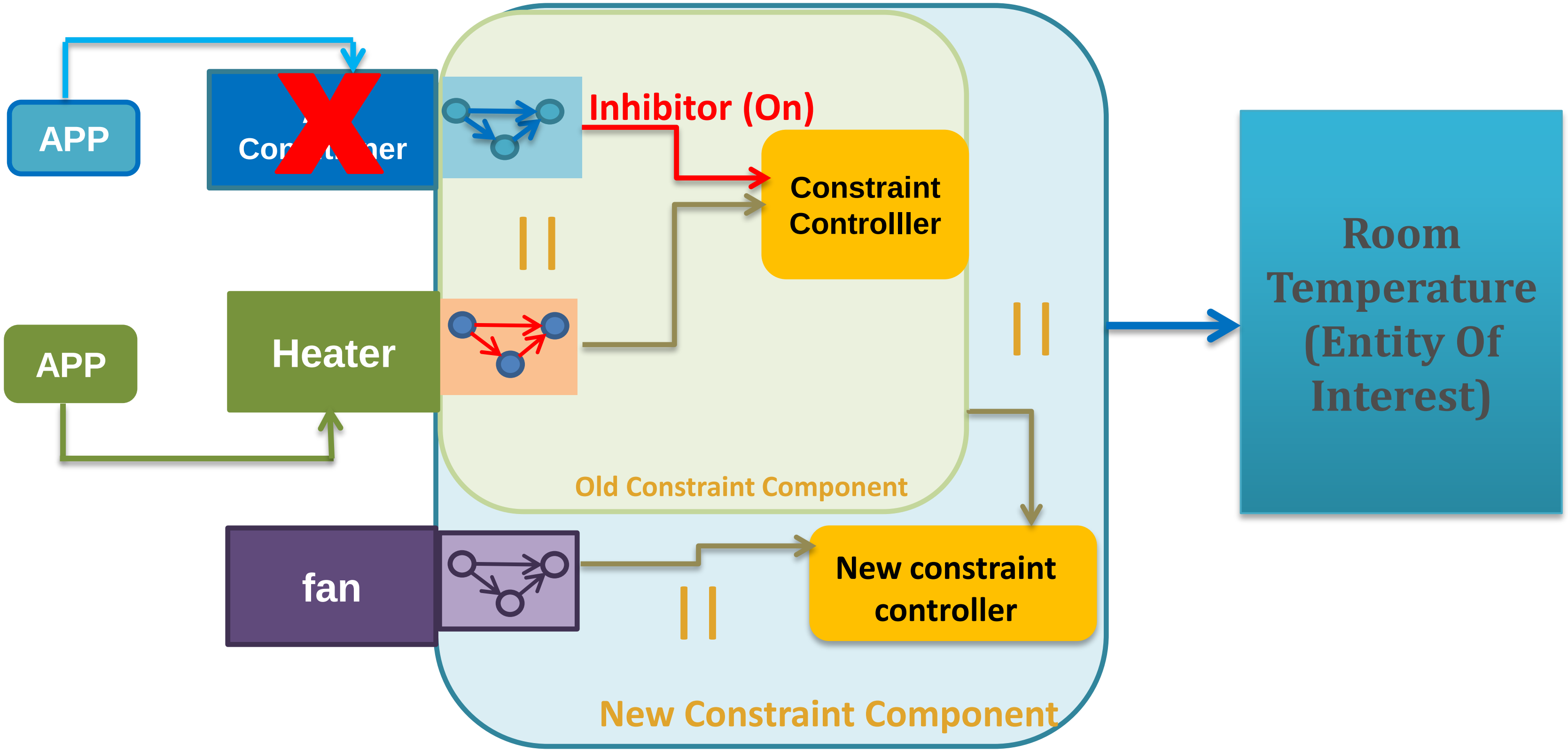
Dynamicity: Appearance of a new Application



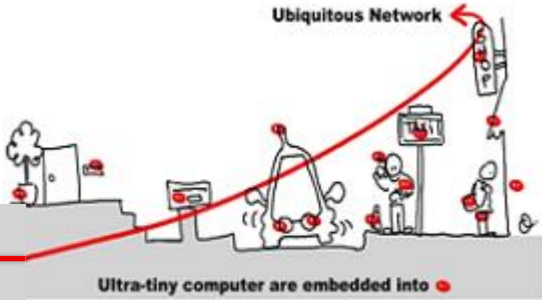
Solution: Constraint Component



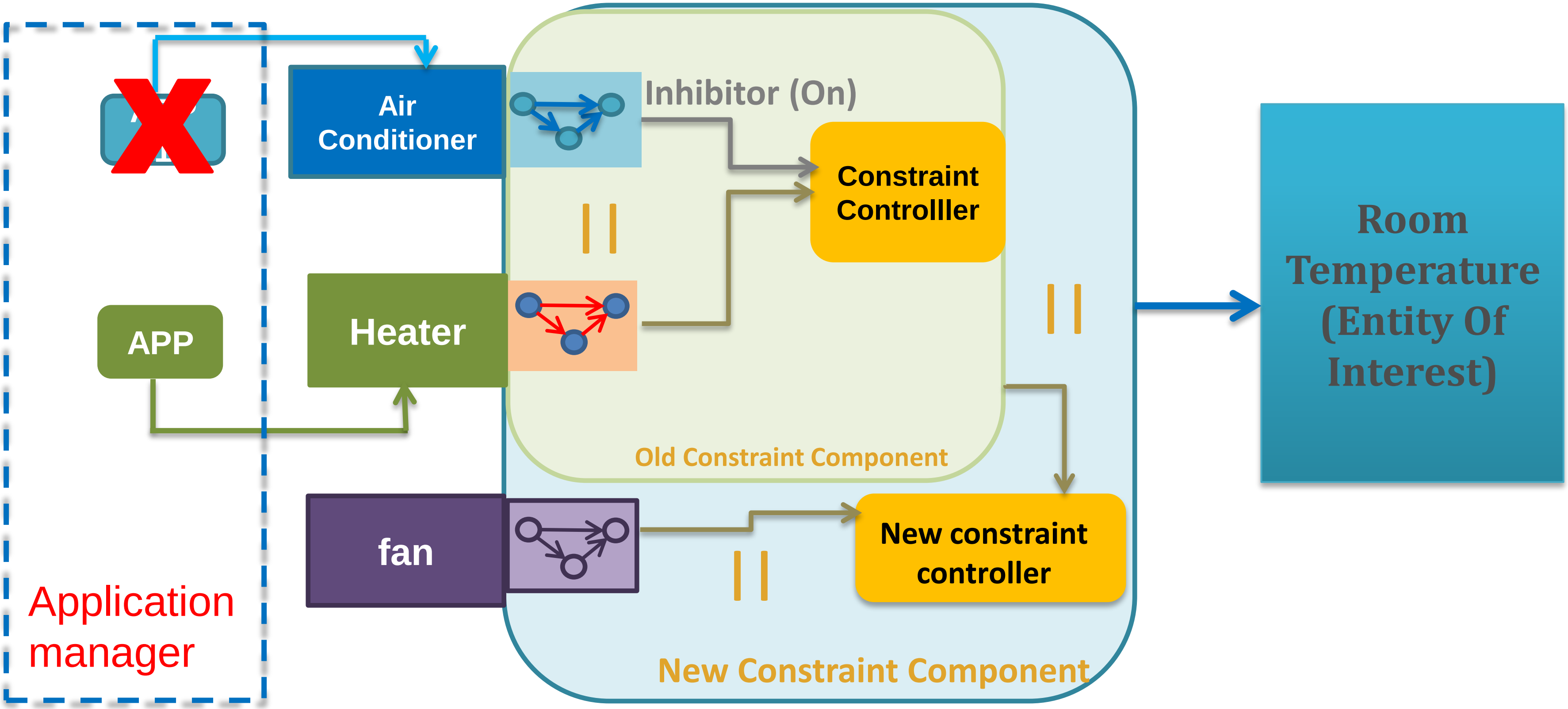
Dynamicity: Disappearance of a device



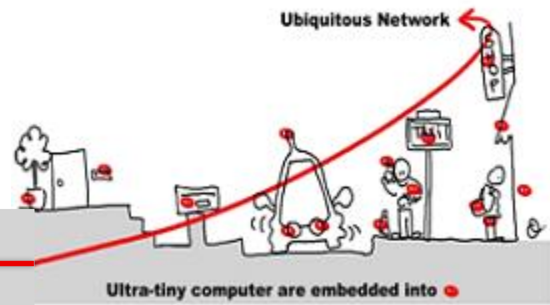
Solution: Constraint Component



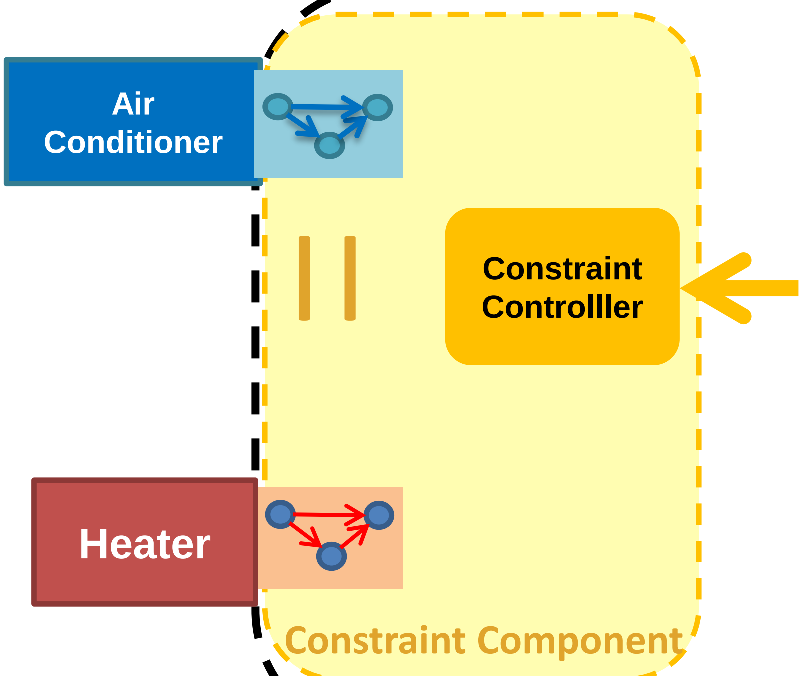
Dynamicity: Disappearance of an application



Constraint Controller Design



Automatic generation of the constraint controller ?



DCL = Description Constraint Language

TypeDevice:
AirConditioner : n, Heater : m;

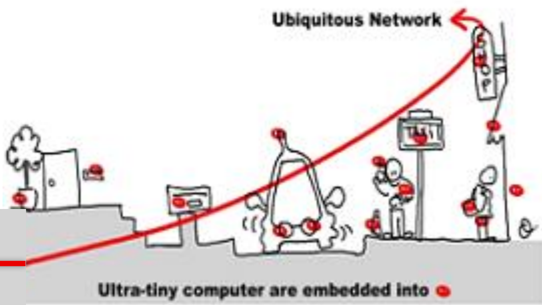
Constraints:
AirConditioner:is_on = Or i in [0..n-1] device[i].is_on and Forall j in [n..n+m-1] (device[j].is_off) ;

AirConditioner:is_off = And i in [0..n] device[i].is_off ;

Heater:is_on = Or i in [n..n+m-1] device[i].is_on and Forall j in [0..n-1](device[j].is_off) ;

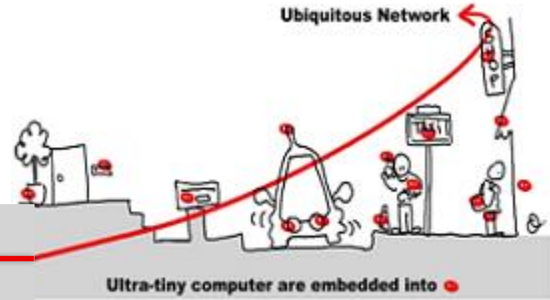
Heater:is_off = And i in [n..n+m-1] device[i].is_off ;

Description Constraint Language



- ✓Need of only application and device types
- ✓Generic constraints description to manage multiple accesses
- ✓Generation of CLEM implicit Mealy machines describing constraint controller behaviors
- ✓Dealing with dynamic environments changes : appearance and disappearance of applications/devices.

Validation with CLEM

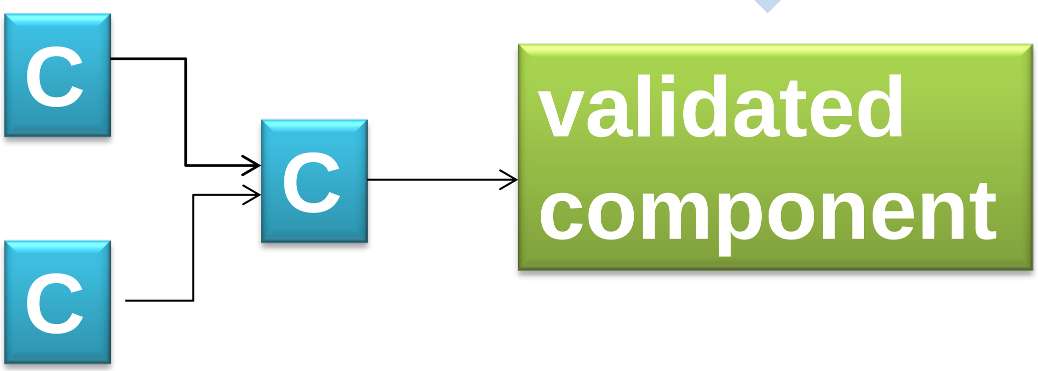


simulation

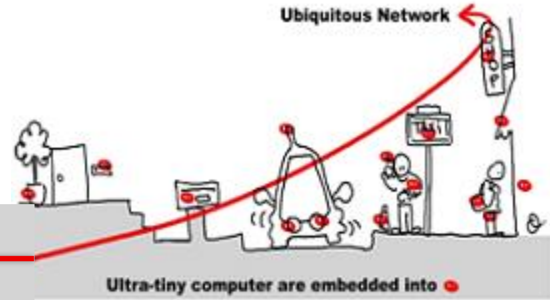
LE design

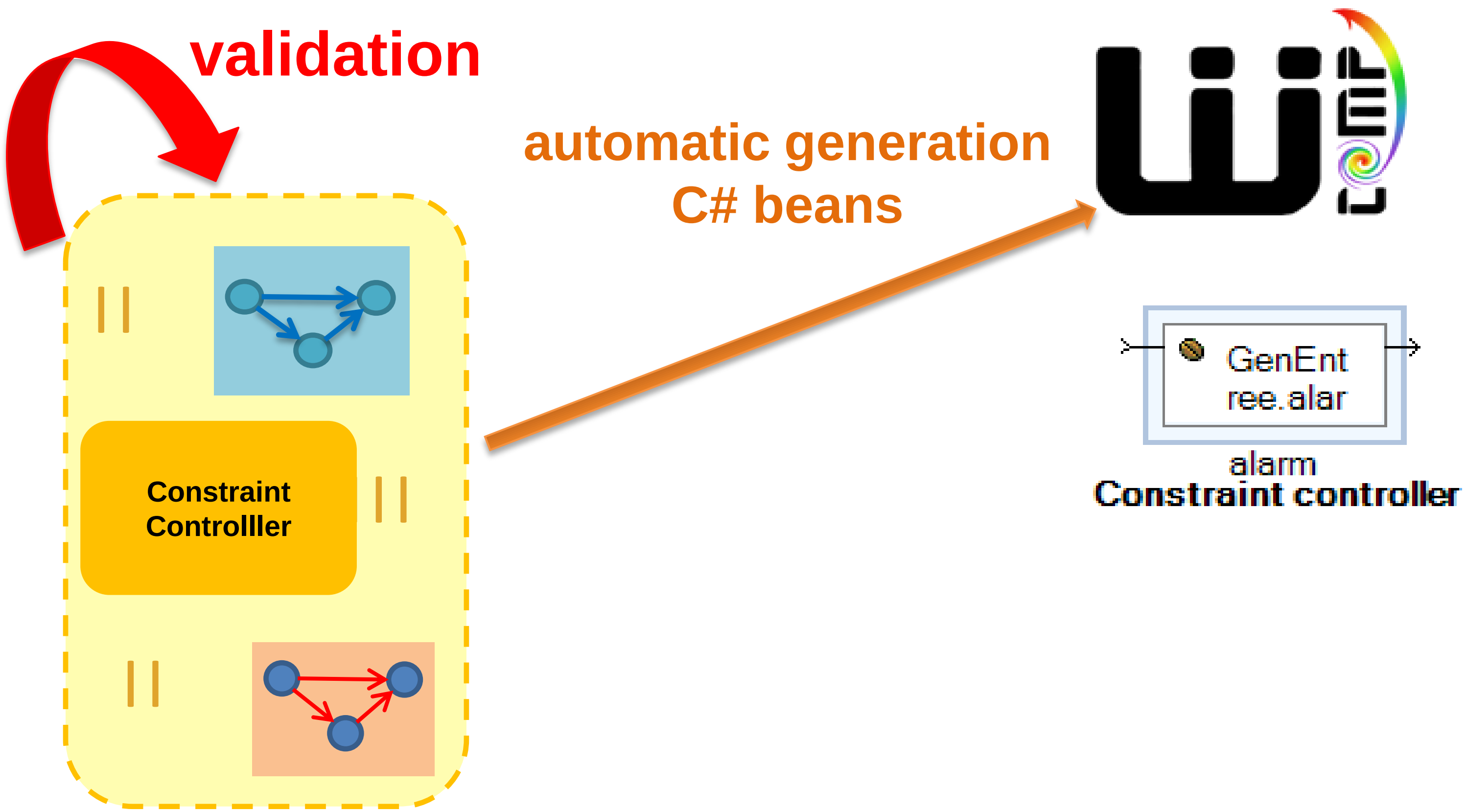
Validatation

Generate C# Bean

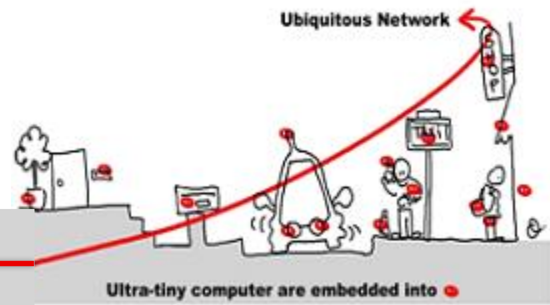


Application to WComp

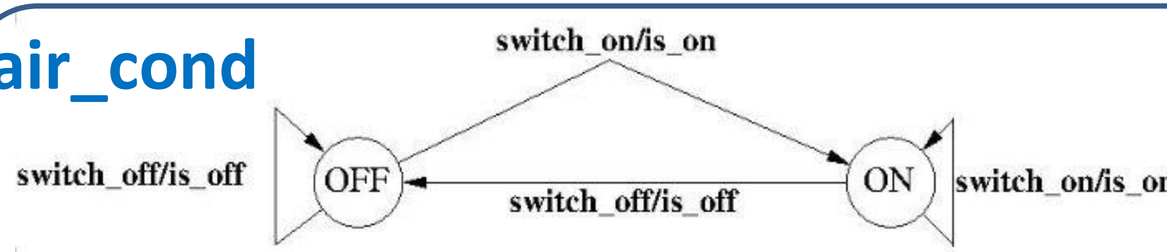




Use Case Issue in CLEM



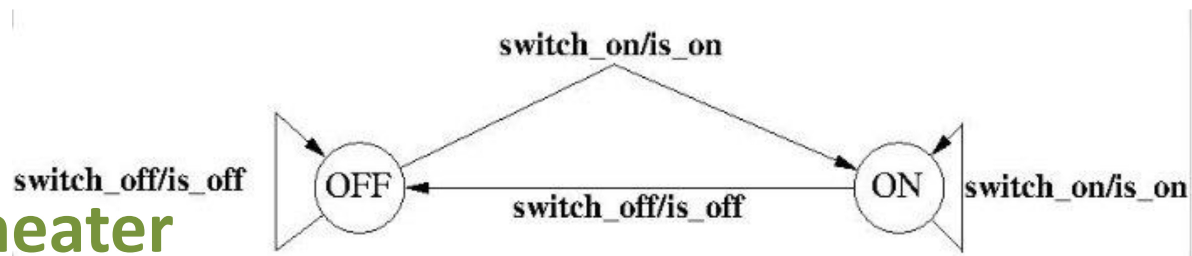
air_cond



Galaxy

air_cond_swith_on = switch_on ; air_condi_switch off = switch_off
air_cond_is_on = is_on ;air_cond_is_off = is_off

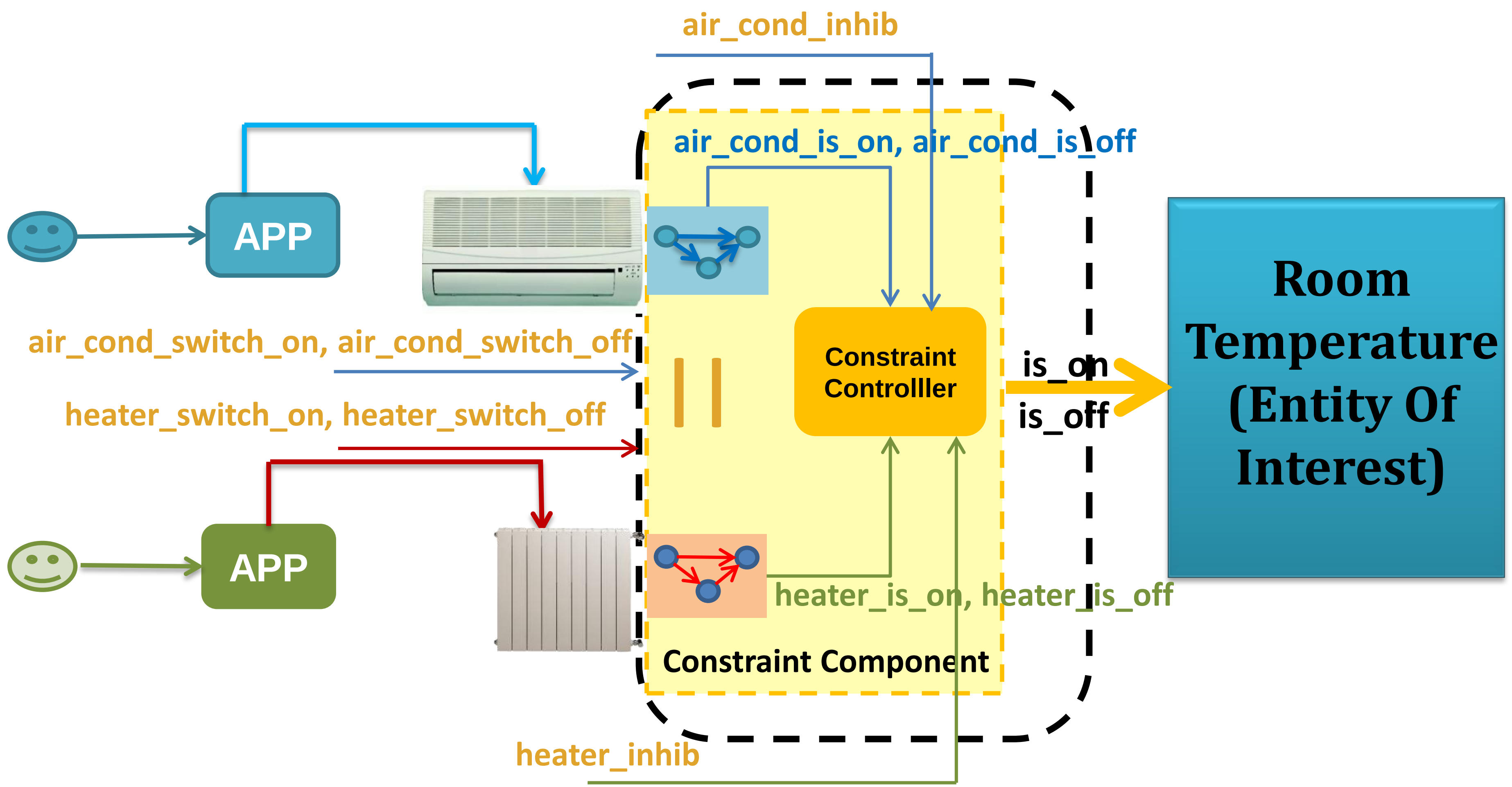
heater_swith_on = switch_on ; air_heater_switch off = switch_off
heater_is_on = is_on ; heater_is_off = is_off



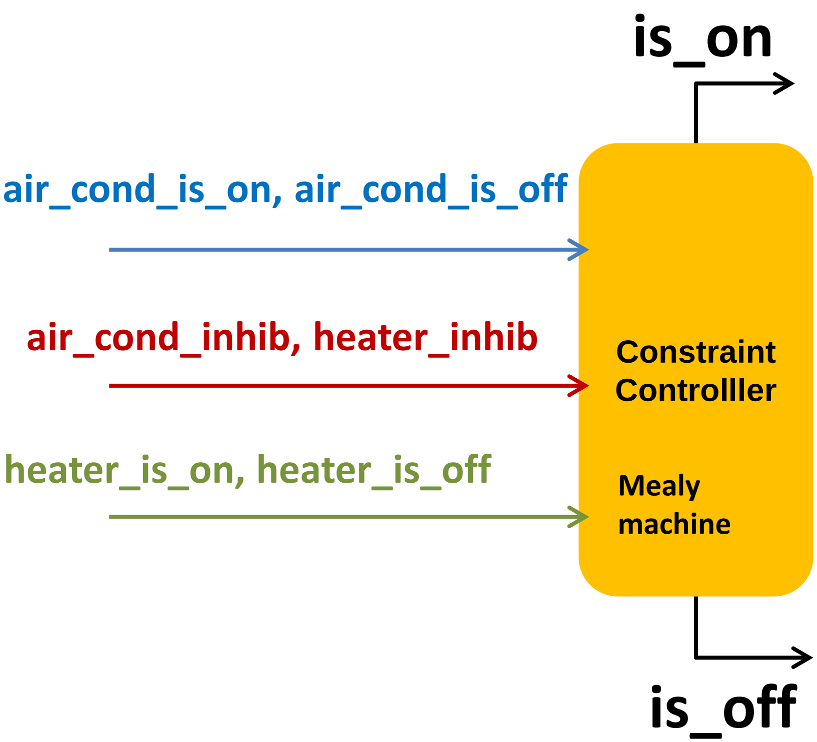
heater

Galaxy

Use Case in CLEM

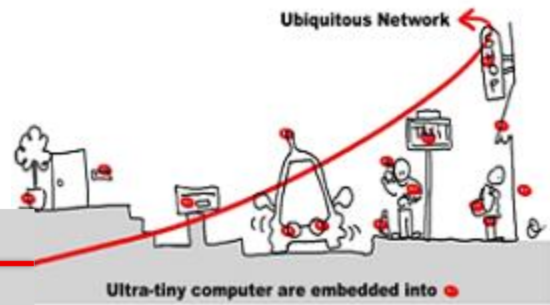


Use Case in CLEM



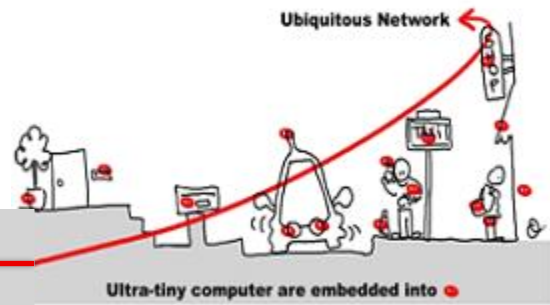

```
module ConstraintController :
 Input: air_cond_is_on, air_cond_is_off,
        heater_is_on, heater_is_off,
        air_cond_inhib, heater_inhib;
 Output: is_on, is_off;
 local ac_is_on, ac_is_off, h_is_on, h_is_off
{
  Mealy machine:
    ac_is_on = air_cond_is_on and not  air_cond_inhib;
    ac_is_off = air_cond_is_off anf not  air_cond_inhib;
    h_is_off = heater_is_off and not heater_inhib;
    h_is_off = heater_is_off and not heater_inhib;

    is_on = (ac_is_on and not h_is_on) or
            (h_is_on and not ac_is_on);
    is_off = h_is_off and ac_is_off;
}
end
```

Use Case in CLEM




```
module ConstraintComponent:
 Input: air_cond_switch_on, air_cond_switch_off, air_cond_inhib, heater_switch_on,
           heater_switch_off, heater_inhib;
 Output: is_on, is_off;
 local air_cond_is_on, air_cond_is_off, heater_is_on, heater_is_off
{
   run AC_H_model[air_cond_switch_on\ switch_on, air_cond_switch_off\switch_off,
                              air_cond_is_on\is_on, air_cond_is_off\is_off]
||
  run AC_H_model[heater_switch_on\switch_on, heater_switch_off\switch_off,
                             heater_is_on\is_on, heater_is_off\is_off]
||
  run ConstraintController
}
end
```

C# Bean Generation



LE Constraint Component

Validation (CLEM blif_check):

air_cond_switch_on and heater_switch_off => is_on

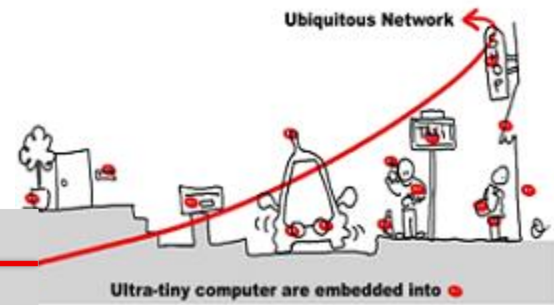
air_cond_inhib and heater_inhib => not is_on

C# Bean Generation

run automaton

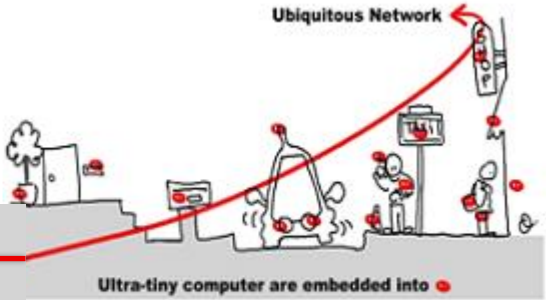
reset automaton

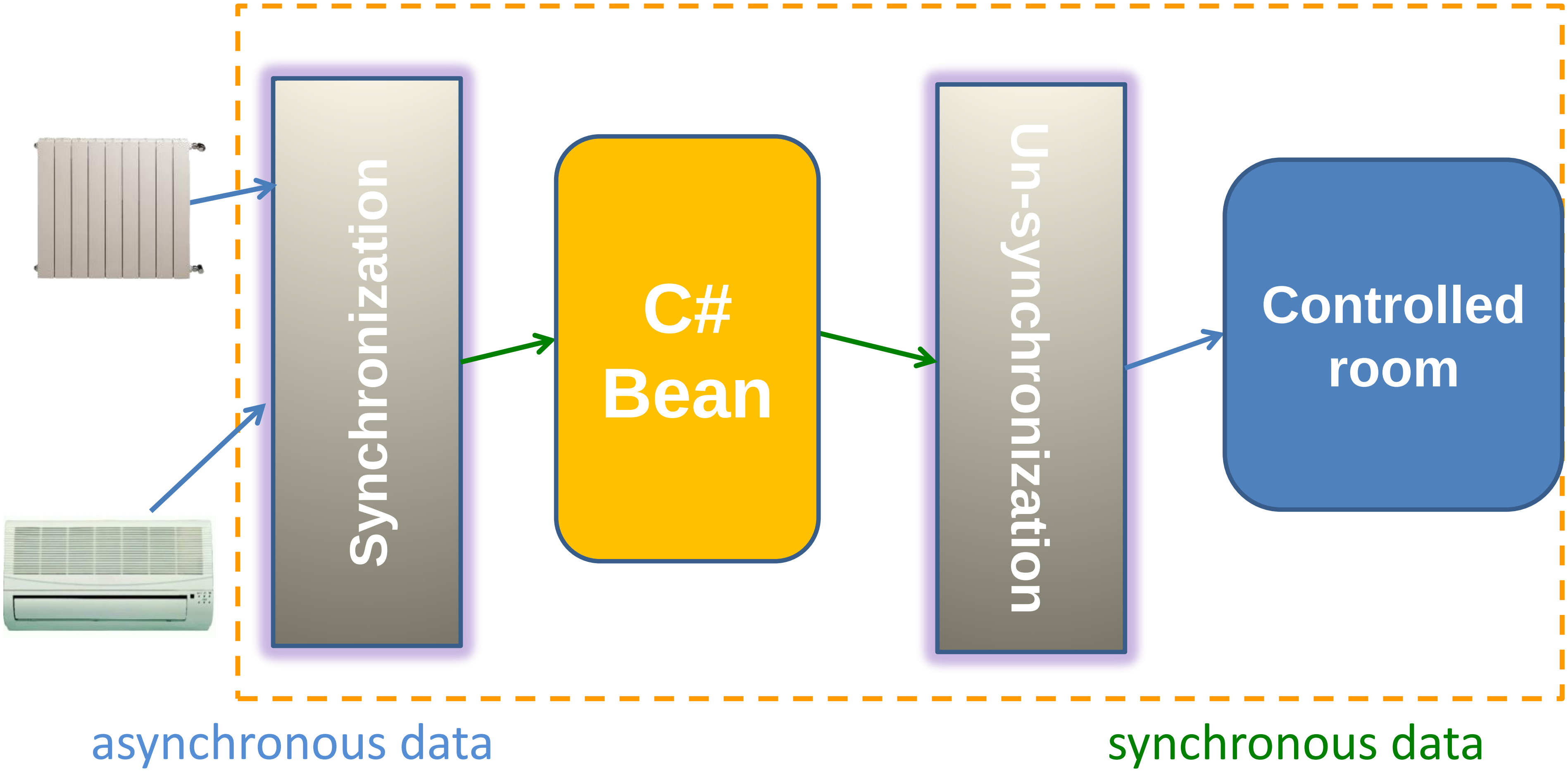
C# Bean Integration



- C# Bean implements synchronous component in Wcomp
- Communication is asynchronous in WComp
- ⇒
  - 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

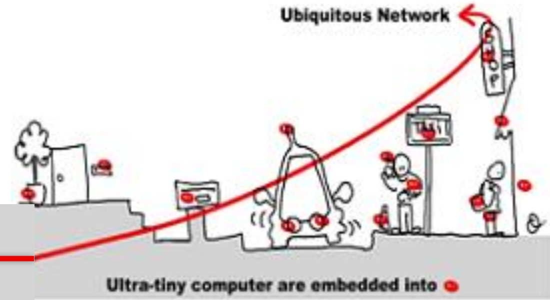




Asynchrony/Synchrony

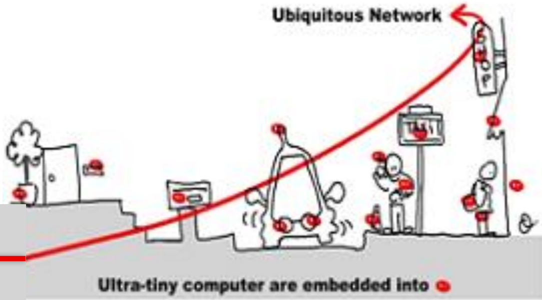
- Synchronization goal:
  1. generate the set of synchronous input events that characterizes the synchronous logical instant.
  2. Define an exchange format to allow communication between synchronous monitors and asynchronous components
- Un-synchronization goal:
  1. Generate the set of asynchronous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

Asynchrony/Synchrony

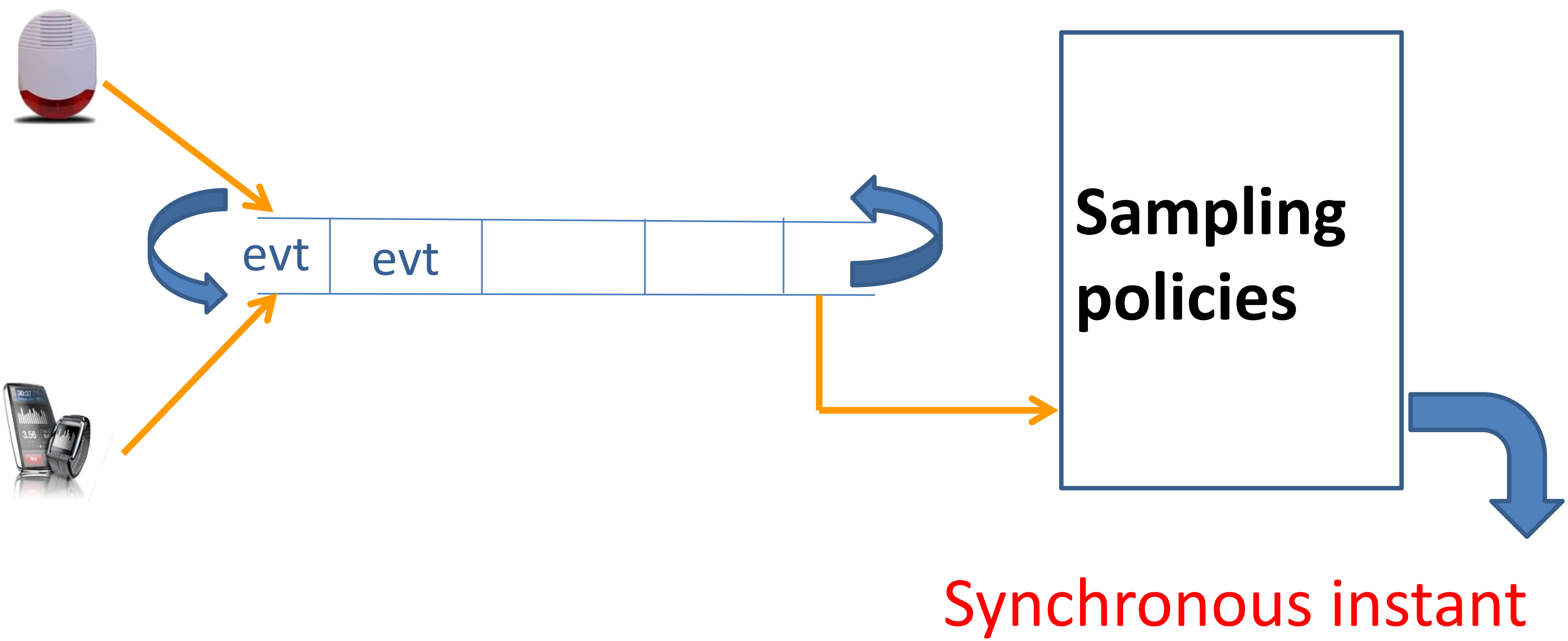


- How define the logical instant ?
  - The synchronization phase should be generic and allow to take into account several types of devices.
  - Introduction of a generic structure to represent events coming from different sensors:
    - name, presence, value type, value, elapsed time
    - apply several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

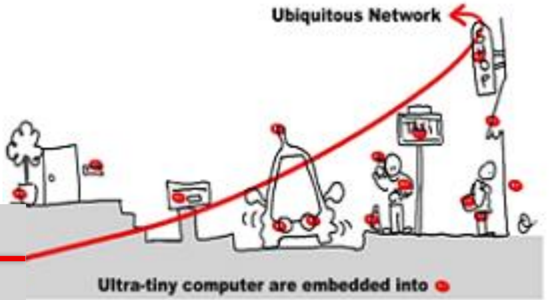
Asynchrony/Synchrony



- How define the logical instant ?

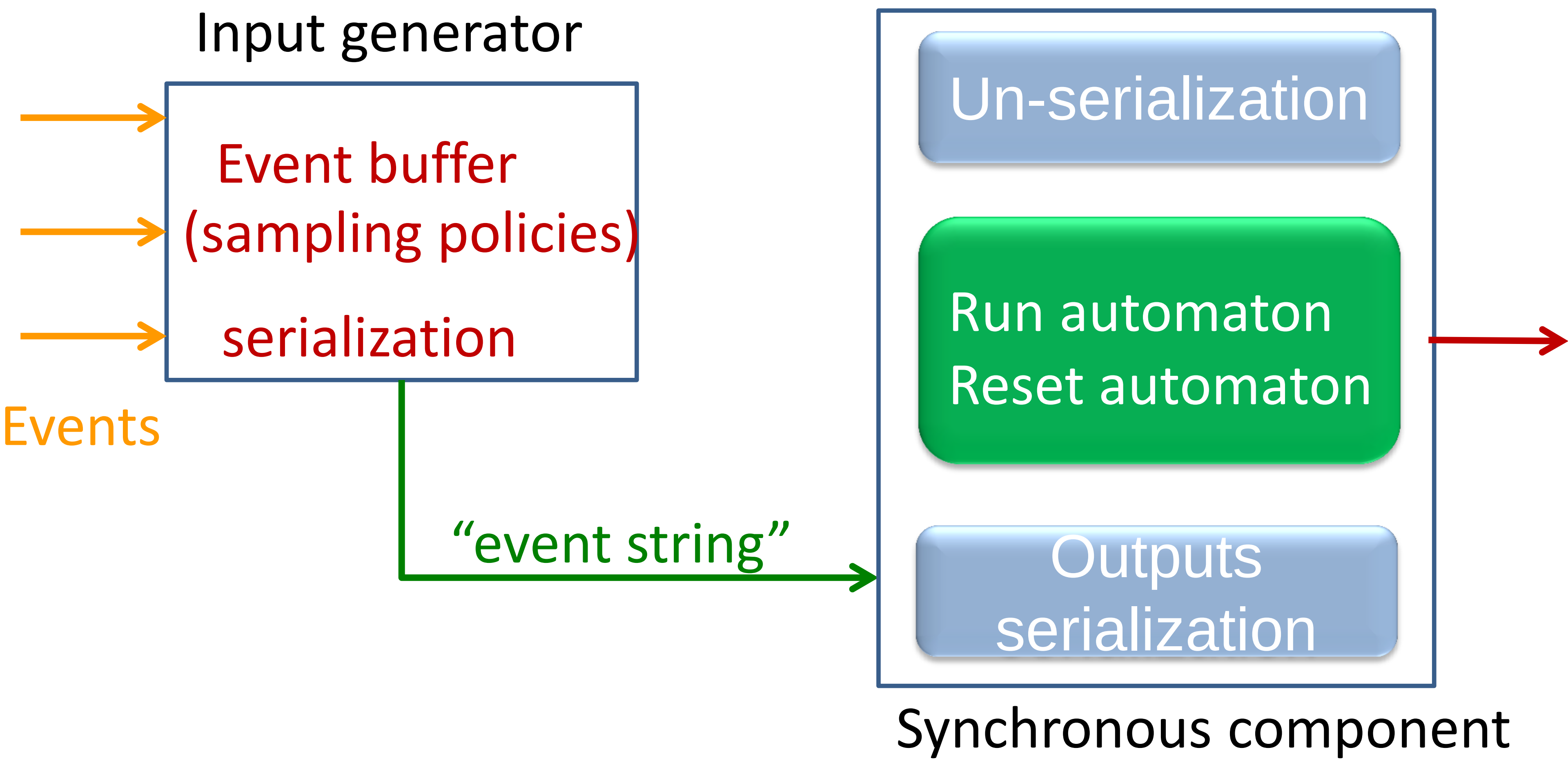


Asynchrony/Synchrony



- 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