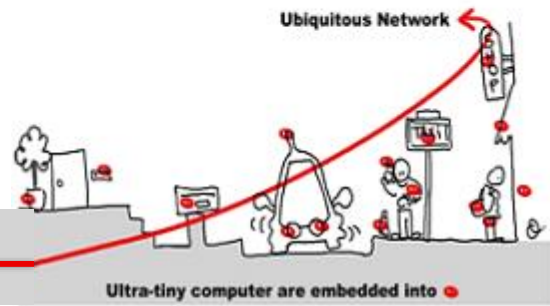


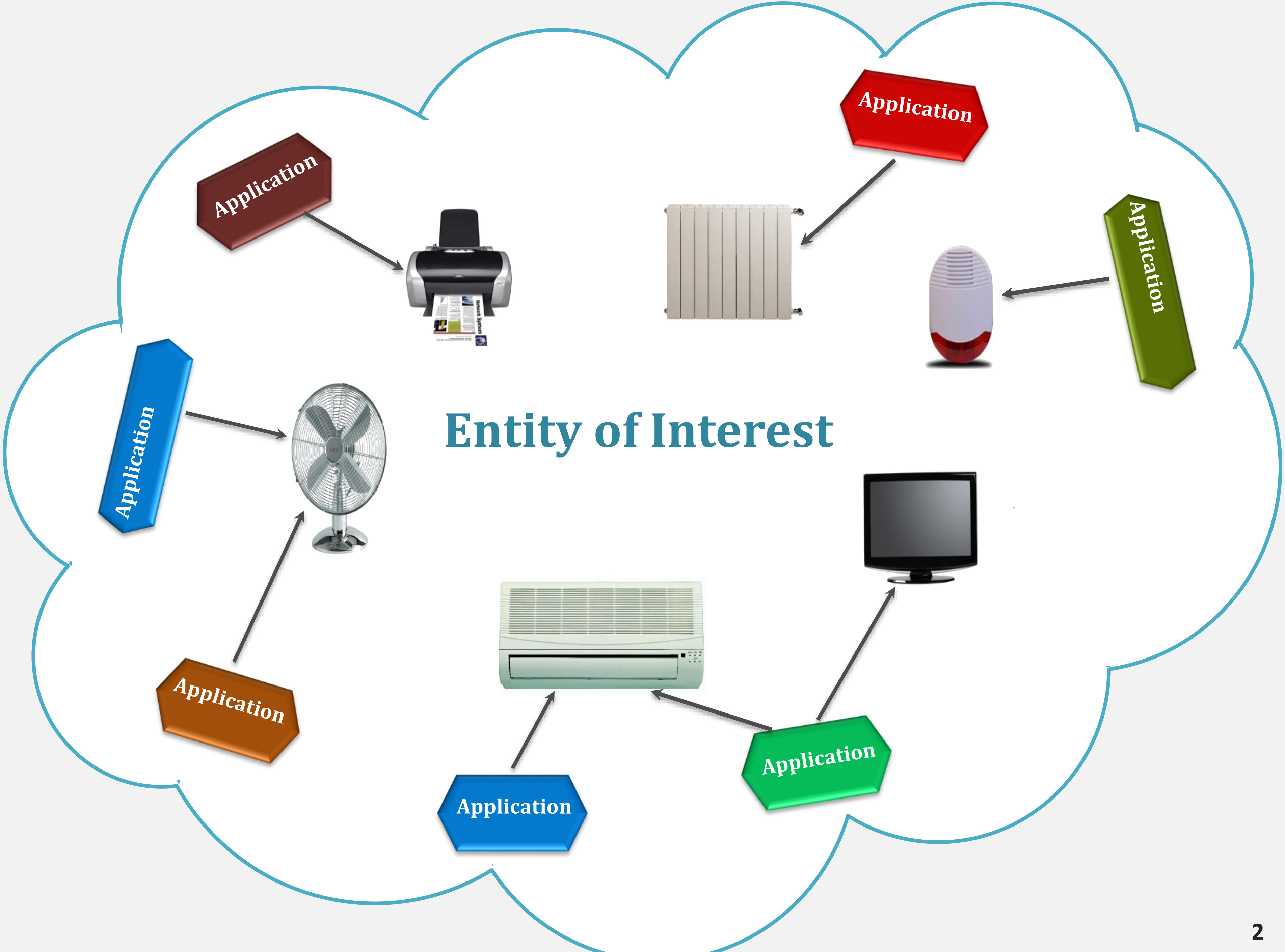

# Synchronous language for formal validation - application to CEP (complex event processing)

Annie Ressouche & Ines Sarray

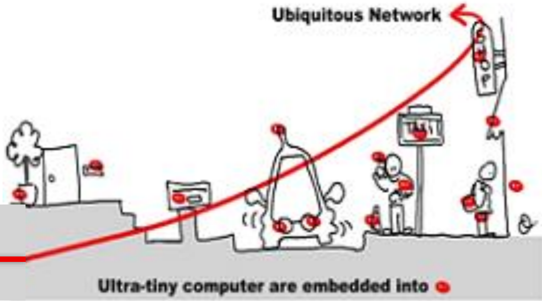
Inria-sam (stars)

{annie.ressouche, ines.sarray}@inria.fr

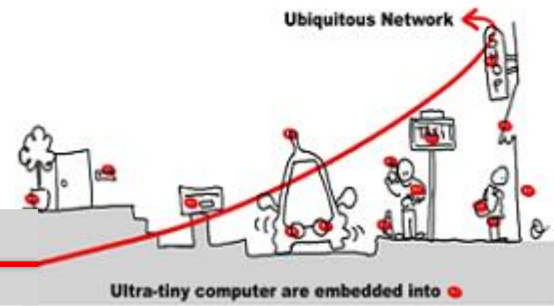
http://www-sop.inria.fr/members/Annie.Ressouche/teaching.html


Application
Application
Application
Application
Entity of Interest
Application
Application
Application
Application

# Introduction



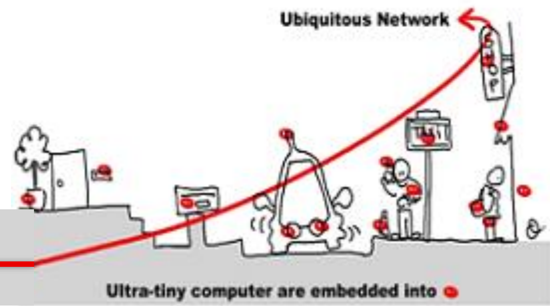

- Middleware for IoT may be used to design critical applications.
- How ensure a correct behavior of applications and services sharing same device accesses ?
- Apply general techniques used to develop critical software




# Outline

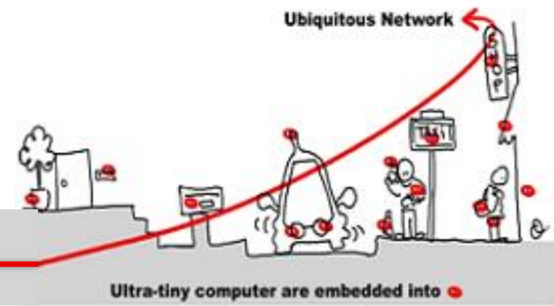
1. Critical system validation
2. Model-checking solution
    1. Model specification
    2. Model-checking techniques
3. Application to middleware for IoT
    1. Introduction in middleware design of synchronous components to allow validation
    2. Synchronous/asynchronous issue

# 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

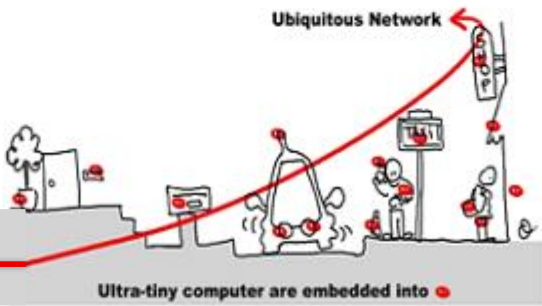



A critical software is a software whose failing has serious consequences:

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

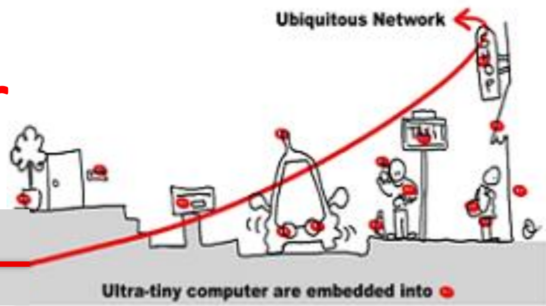
…

# 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)

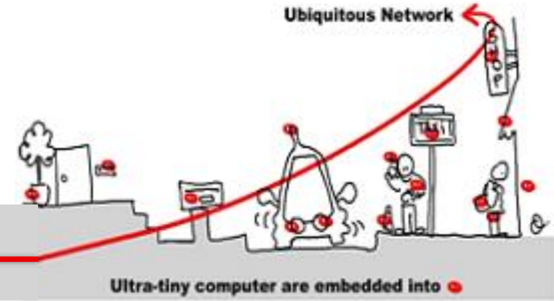
# Software Classification

Example of the aeronautics norm DO178B:

| | |
|---|---|
| **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 |

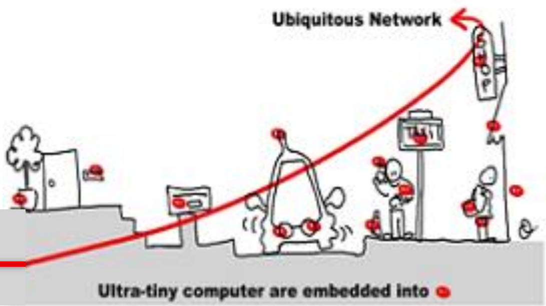
Depending of the level of risk of the system, different kinds of verification are required

# Software Classification

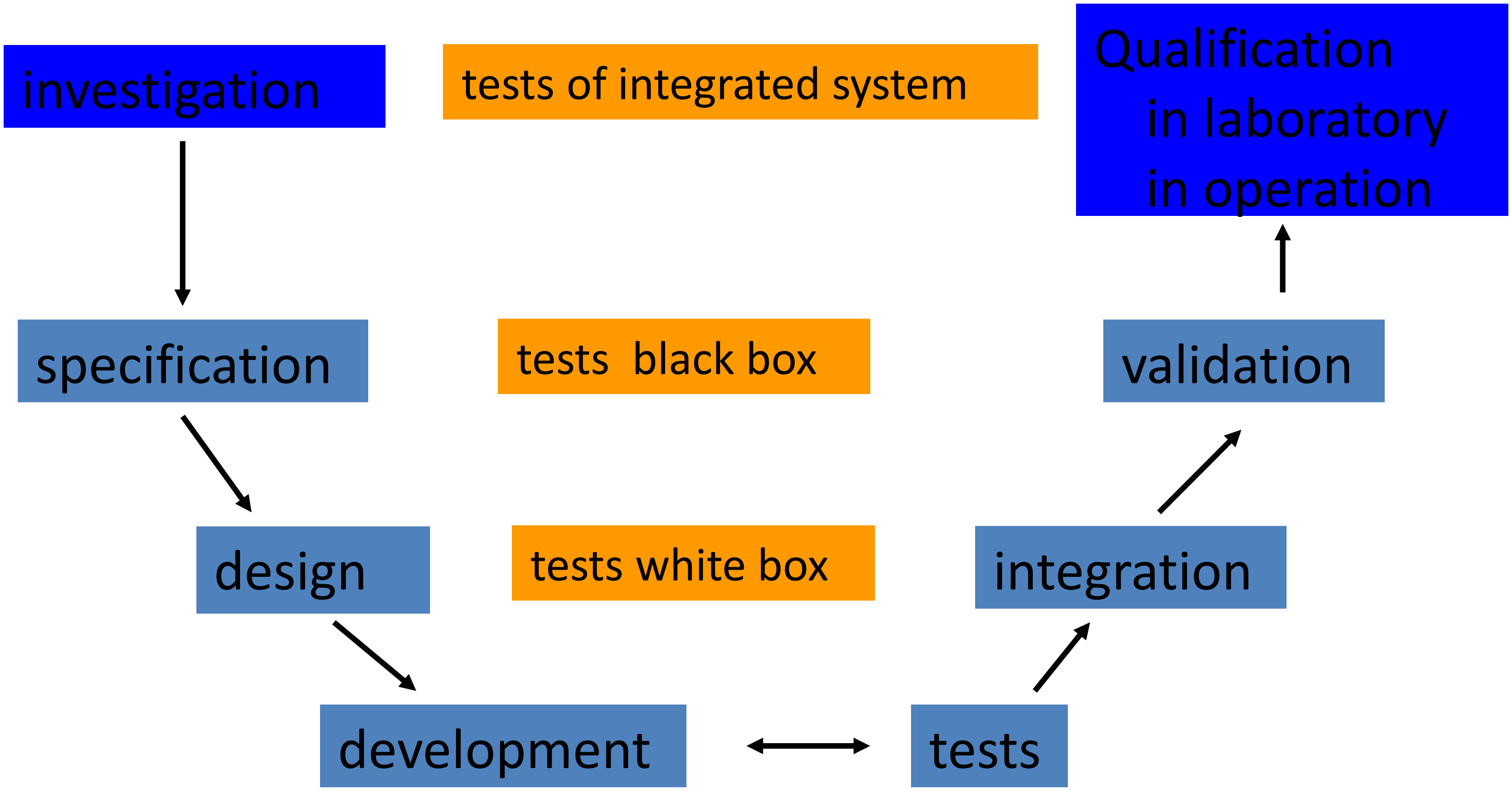



| | | | | |
|---|---|---|---|---|
| Minor | | | acceptable situation | |
| Major | | | | |
| Dangerous | Unacceptable situation | | | |
| catastrophic | $10^{-3}$ / hour | $10^{-6}$ / hour | $10^{-9}$/hour | $10^{-12}$/hour |
| *probabilities* | probable | rare | very rare | very improbable |

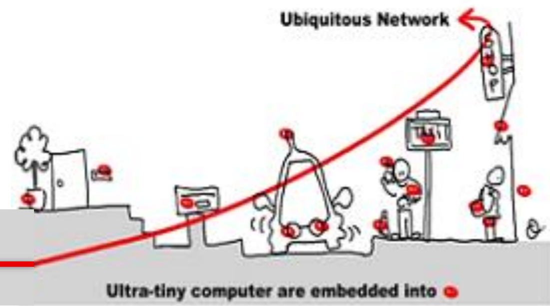
# How Develop critical software ?




Classical Development U Cycle

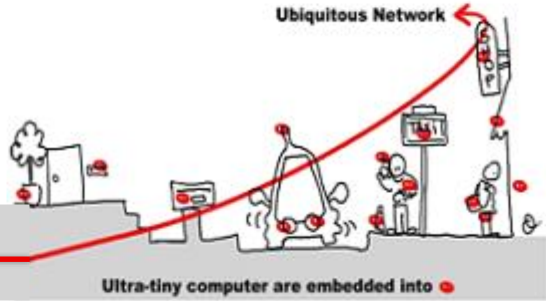


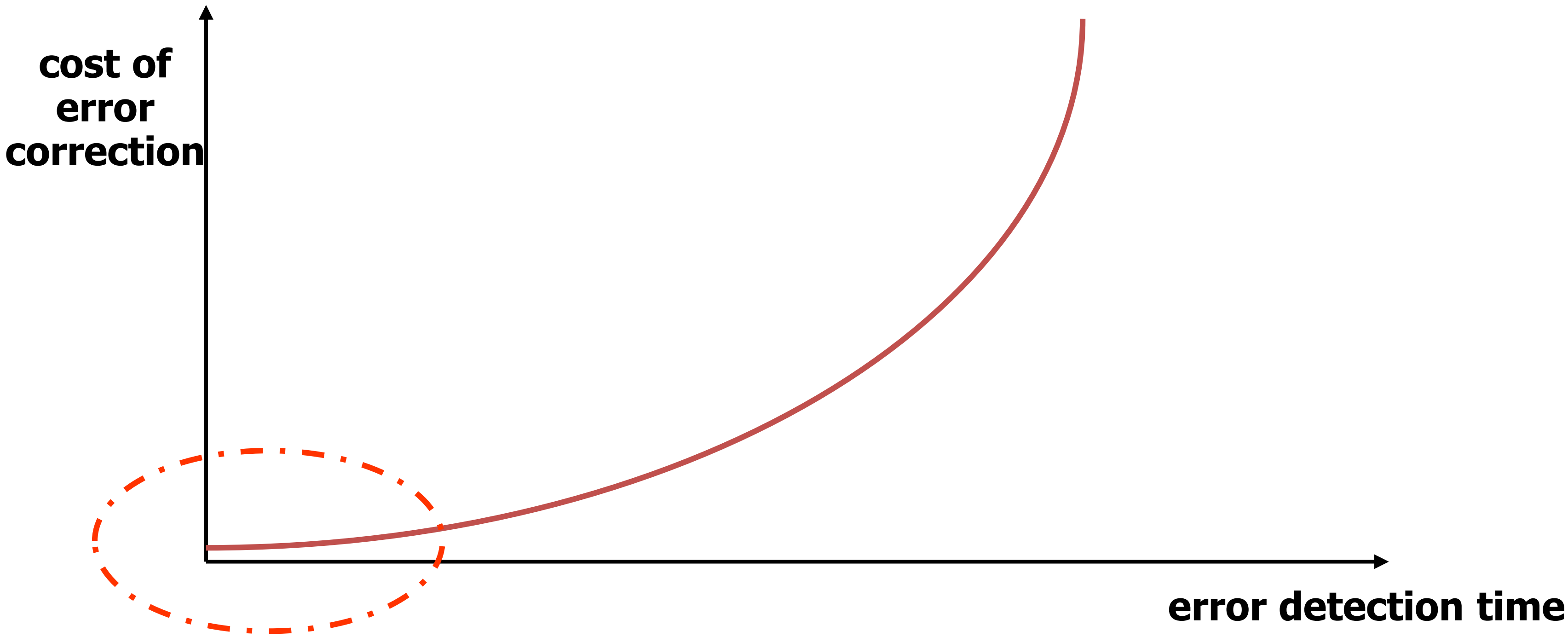
# How Develop Critical Software ?




- 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

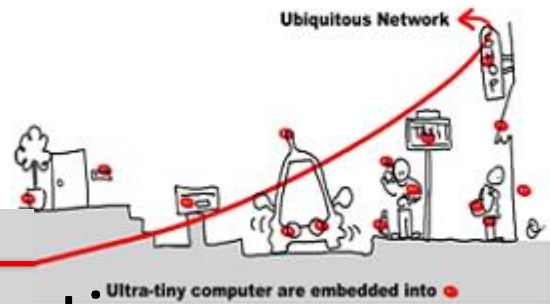



**Put the effort on the upstream phase**

↳ **development based on models**

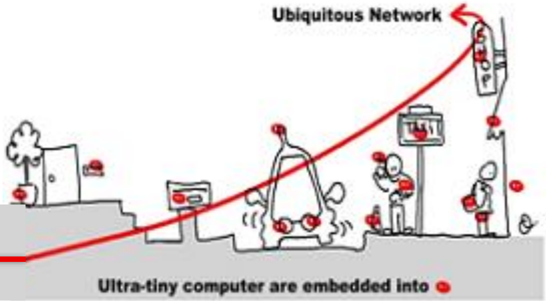
# How Develop Critical Software ?




- 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

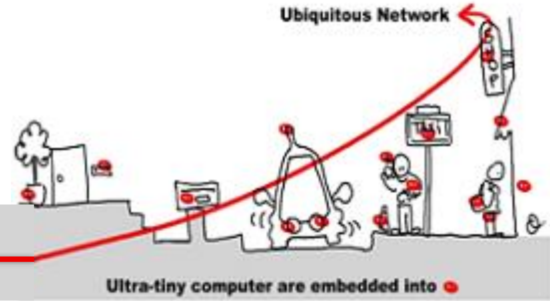
⇒ Formal model usage

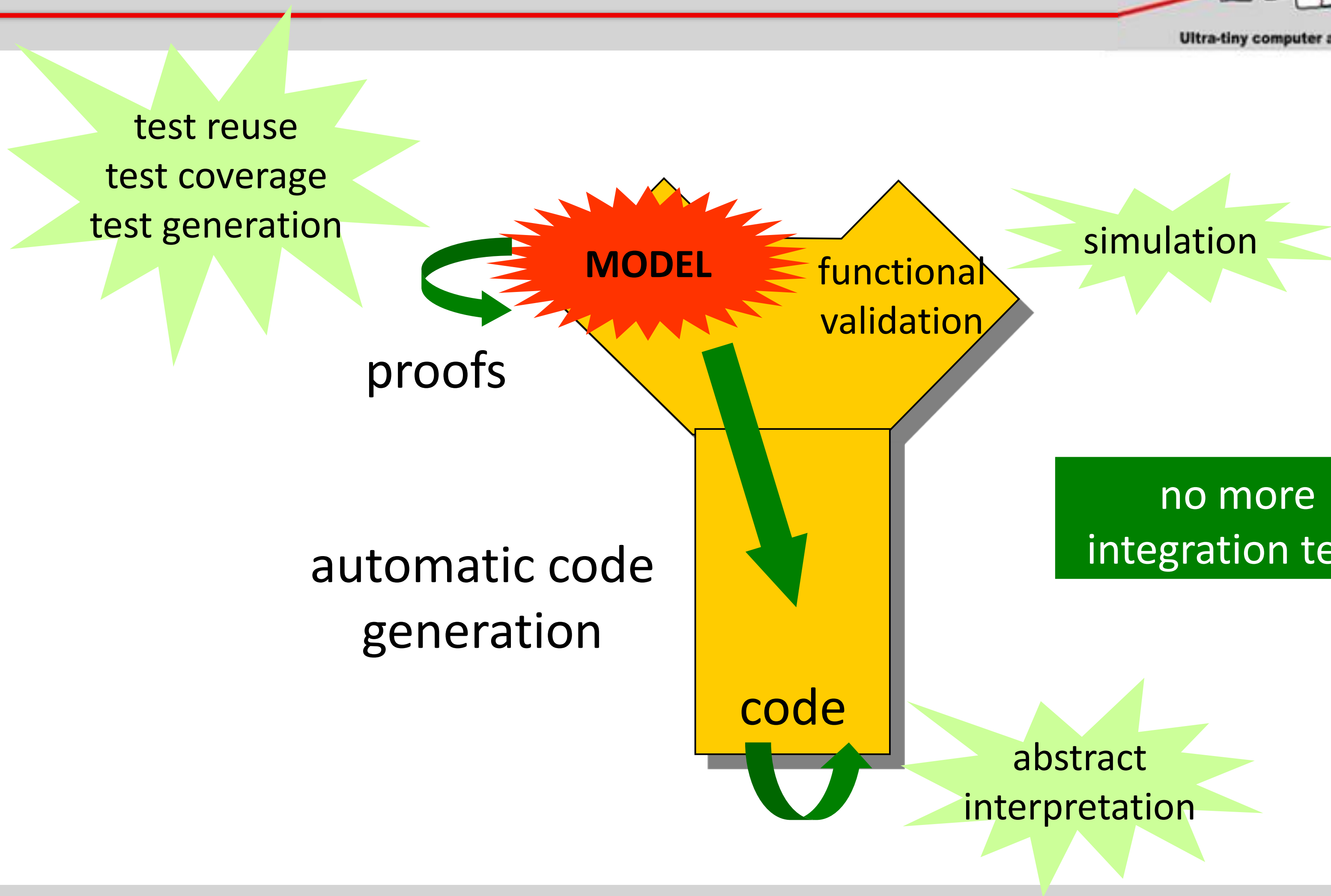
# Critical Software Specification



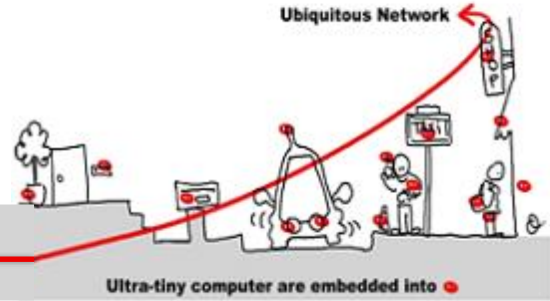

- First goal: must yield a formal description of the application needs.
- Second goal: allowing errors detection carried out upstream.
- Third goal: make easier the transition from specification to design

# How Develop Critical Software



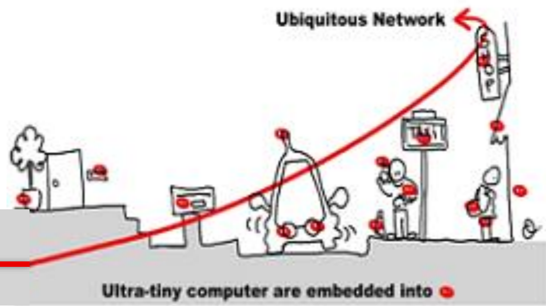



# Critical Software Validation



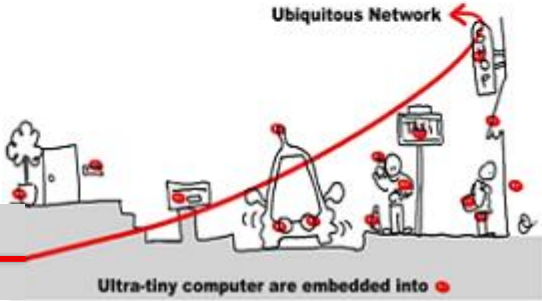

- What is a correct software?
  - No execution errors, time constraints respected, compliance of results.
- Solutions:
  - At model level :
    - Simulation
    - Formal proofs
  - At implementation level:
    - Test
    - Abstract interpretation

# Validation Methods



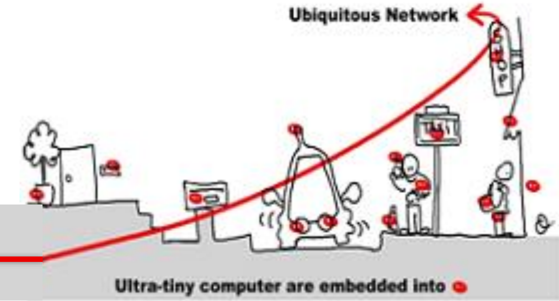

- Testing
  - Run the program on set of inputs and check the results
- Static Analysis
  - Examine the source code to increase confidence that it works as intended
- Formal Verification
  - Argue formally that the application always works as intended

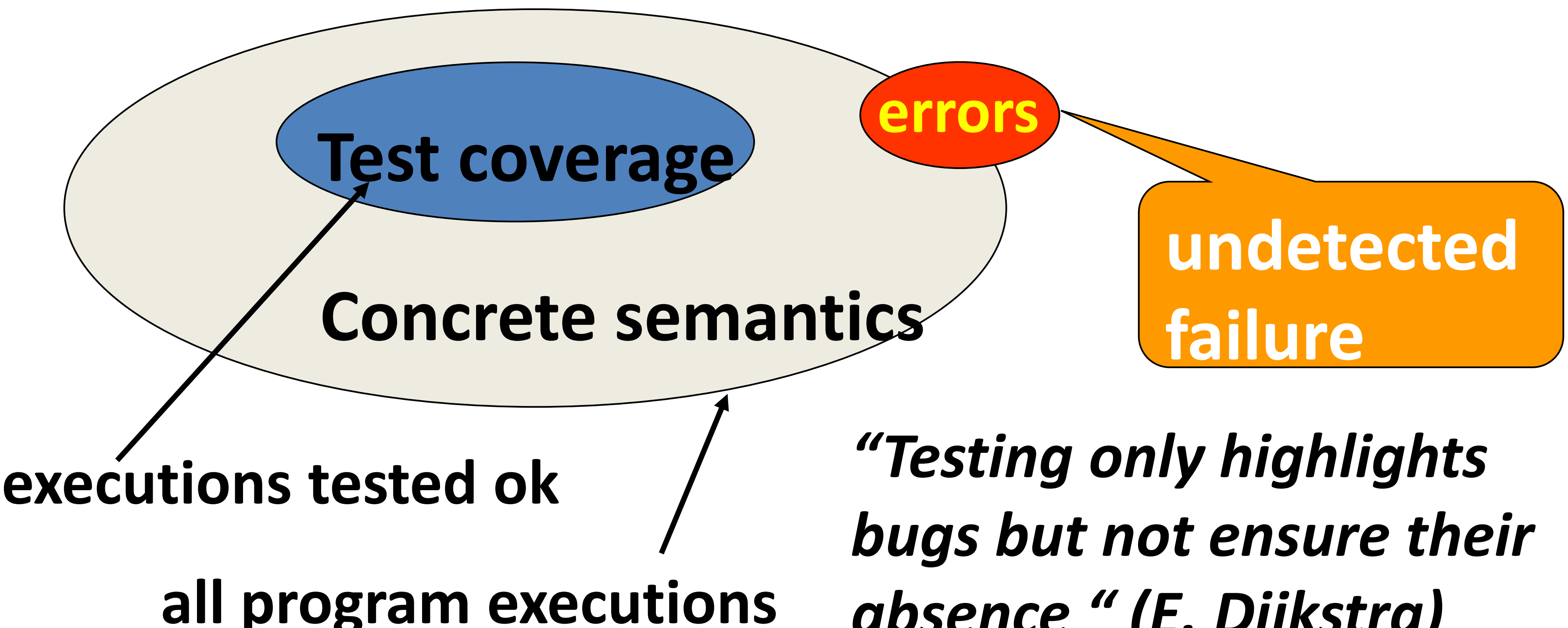
# Testing




- 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.

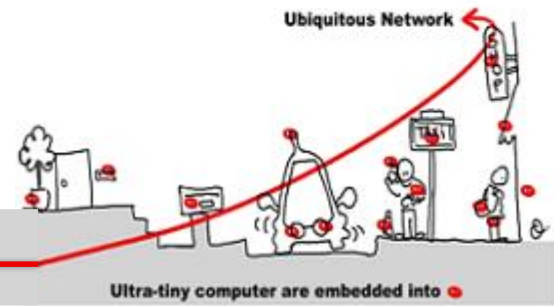
# Program Testing



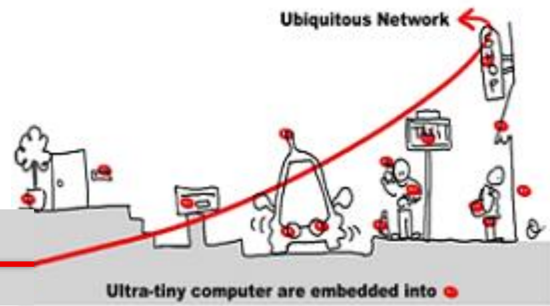

***"Testing only highlights bugs but not ensure their absence " (E. Dijkstra)***

# 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 :
  - $\mathfrak{A}$(P) “correct” $\Rightarrow$ P correct
  - But $\mathfrak{A}$(P) “incorrect” $\Rightarrow$ ?

# Static Analysis: example




abstraction: integer by intervals

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

$$x1 = [1,1]$$

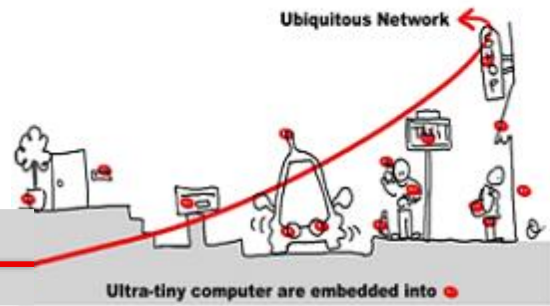
$$x2 = x1 \cup x3 \cap [-\infty, 999]$$

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

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

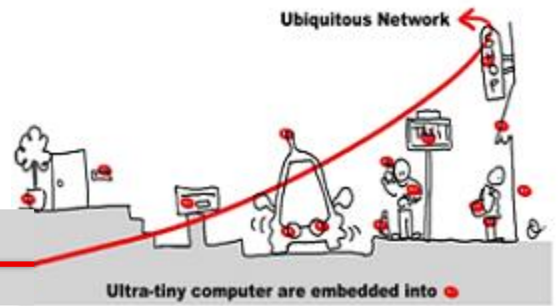
**Abstract interpretation theory ⇒ values are fix point equation solutions.**

# Formal Verification



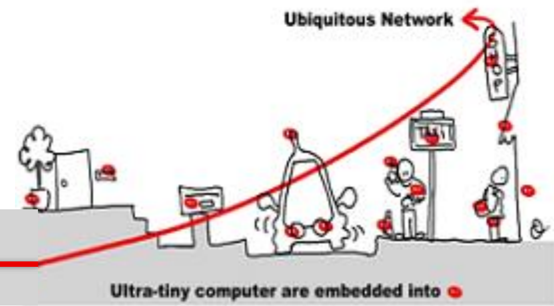

- What about functional validation ?
  - Does the program compute the expected outputs?
  - Respect of time constraints (temporal properties)
  - Intuitive partition of temporal properties:
    - Safety properties: something bad never happens
    - Liveness properties: something good eventually happens

# Safety and Liveness Properties



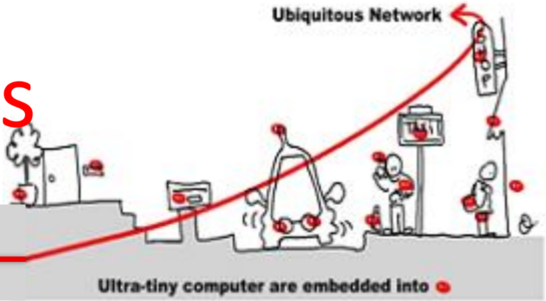

- Example: train timetable
  - Count the difference between marks and seconds
  - Decide when the train is ontime, late, early
    - **ontime** : difference = 0
    - **late** : difference > 3 and it was ontime before or difference > 1 and it was already late before
    - **early** : difference < -3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



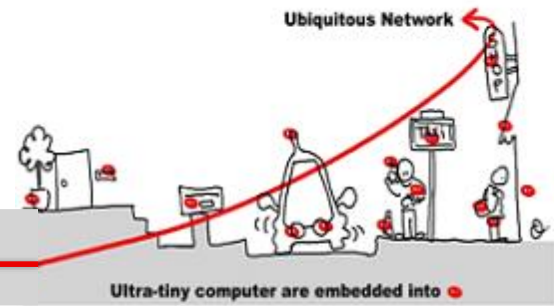

Some properties:

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

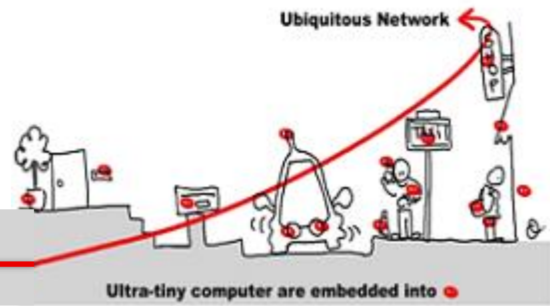
Property 4 : liveness (refer to unbound future)

# Outline



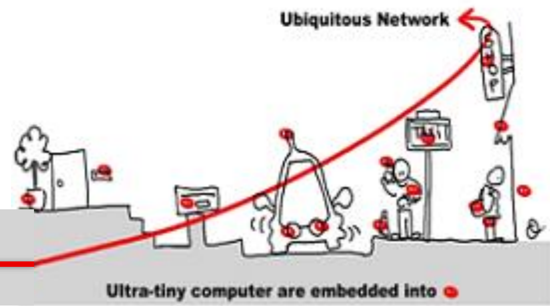

1. Critical system validation
2. Model-checking solution
    1. Model specification
    2. Model-checking techniques
3. Application to middleware for IoT
    1. Introduction in middleware design of synchronous components to allow validation
    2. Synchronous/asynchronous issue

# Safety and Liveness Properties Checking



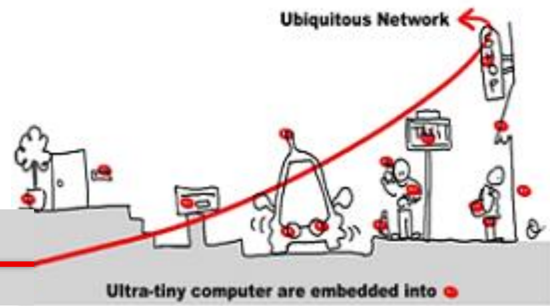

- Use of model checking technique
- Model checking goal: prove safety and liveness properties of a system in analyzing a model of the system.
- Model checking techniques require:
  - model of the system
  - express properties
  - algorithm to check properties againts the model ($\Rightarrow$ decidability)

# Model Checking Techniques



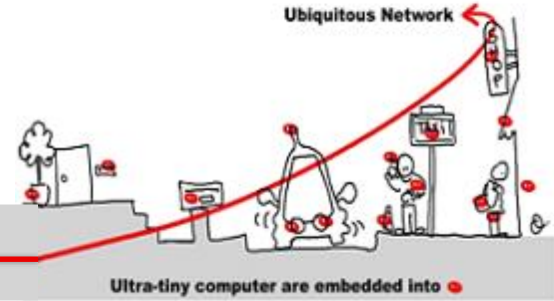

- Model = automata which is the set of  program behaviors
- Properties expression = temporal logic:
  - LTL : liveness properties
  - CTL: safety properties
- Algorithm =
  - LTL : algorithm  exponential wrt the formula size and linear wrt automata size.
  - CTL: algorithm linear wrt formula size  and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

# Model Checking Model



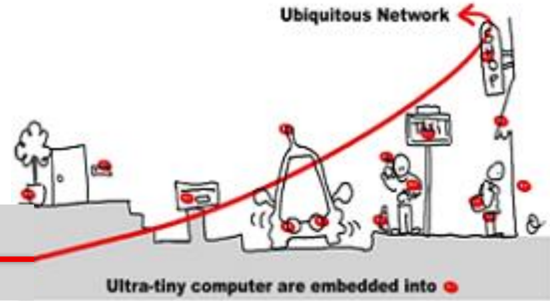

- Model = finite state machine (automata) which is the set of program behaviors
- Kripke structure:
  - non deterministic automata
  - Oriented graph
  - Nodes are program states
  - To each state , a set of atomic (basic) properties is associated

# 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 ?

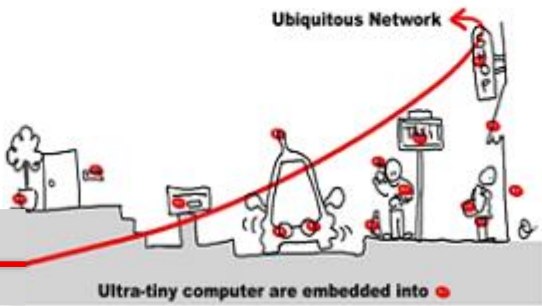
# Model Specification




- 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 $\delta: \mathbb{Q} \times \mathbb{T} \rightarrow \mathbb{Q}$
  6. An output function $\lambda : \mathbb{Q} \times \mathbb{T} \rightarrow 2^{\mathbb{A}}$

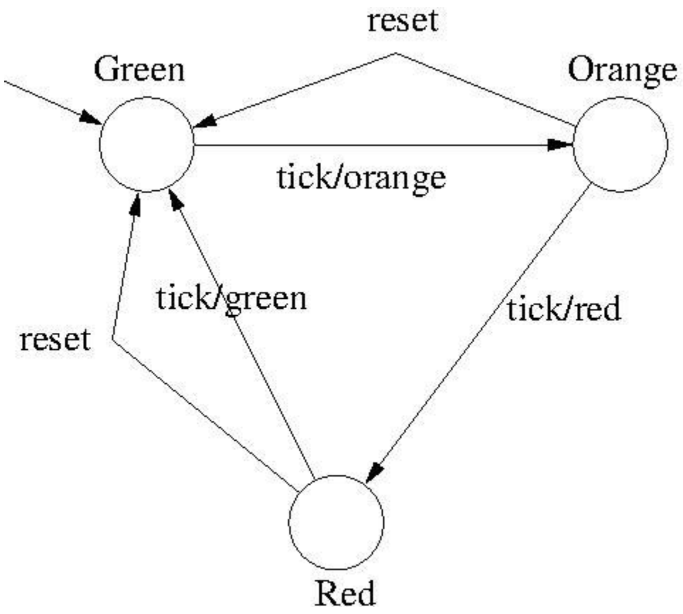
Notation: a transition is denoted $q_1 \xrightarrow{t/a} q_2$

# Model Specification




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

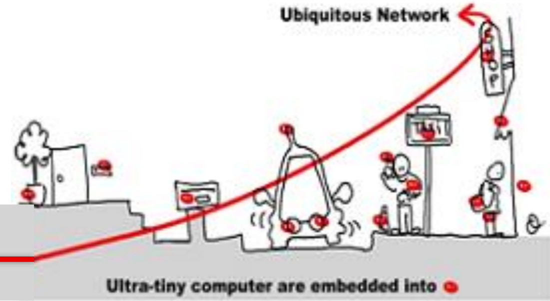
Example: **Traffic Light**




trigger: tick, reset

action:green,orange,red

# Model Specification

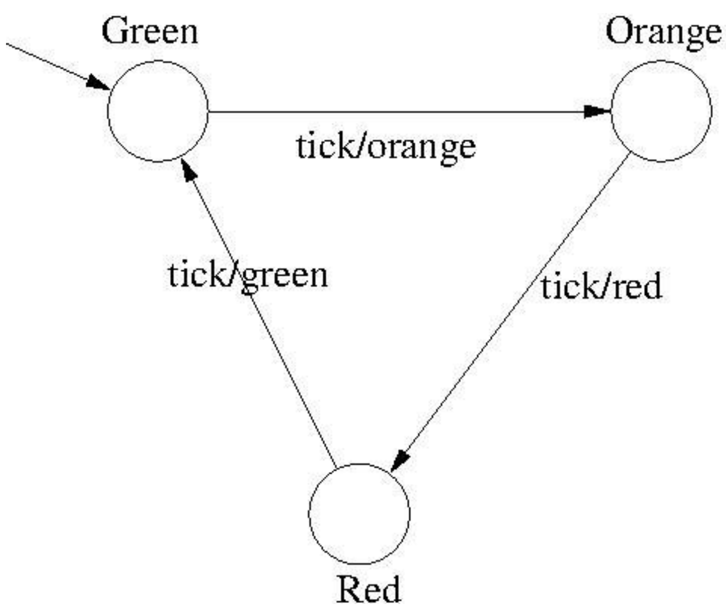


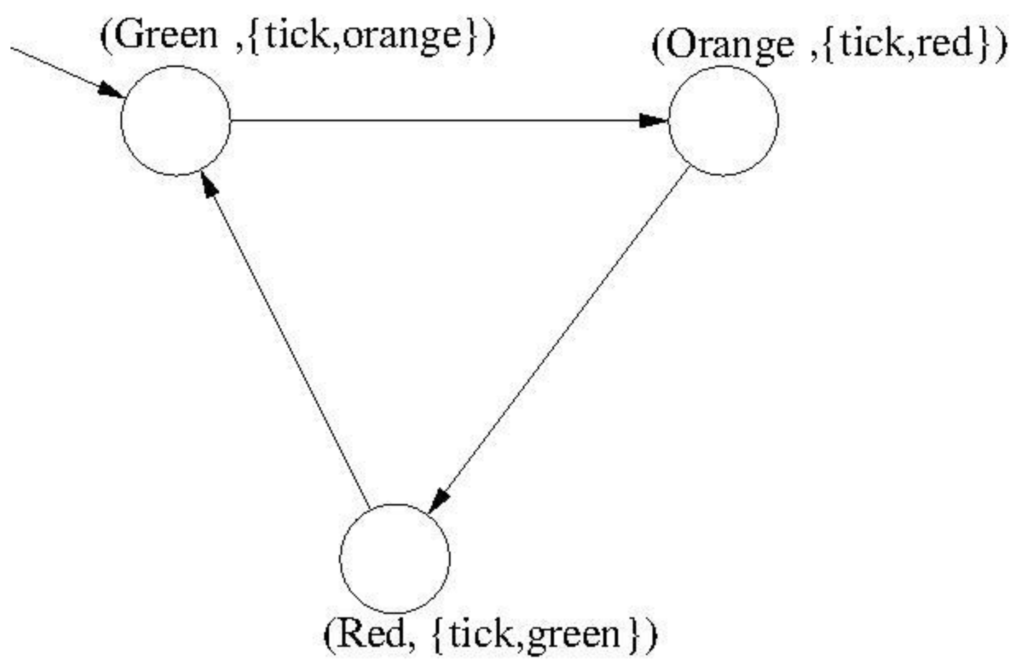

## Mealy automata = Kripke structure

- $\mathbb{AP} = \mathbb{T} \cup \mathbb{A}$
- $\mathbb{S} \subseteq \mathbb{Q} \times 2^{\mathbb{AP}}$ ; $\{(q, v) \mid \exists\, q \xrightarrow{t/a} q'$ 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

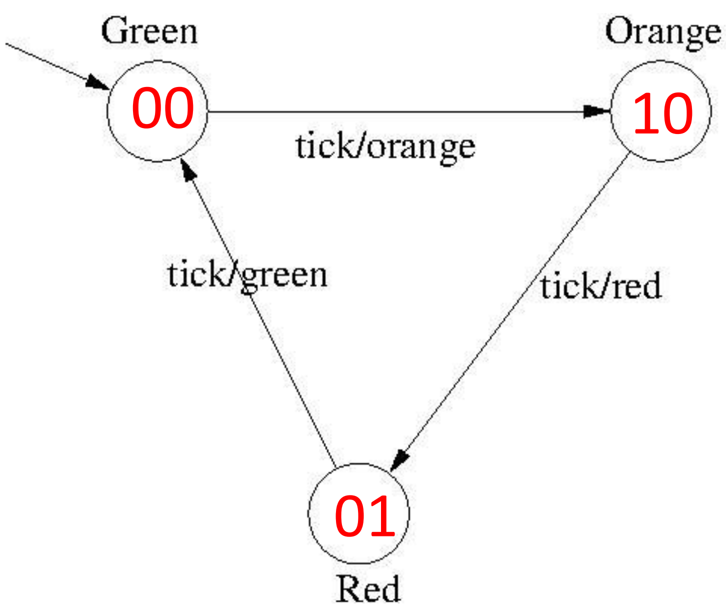


# Implicit vs Explicit Mealy Machine

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

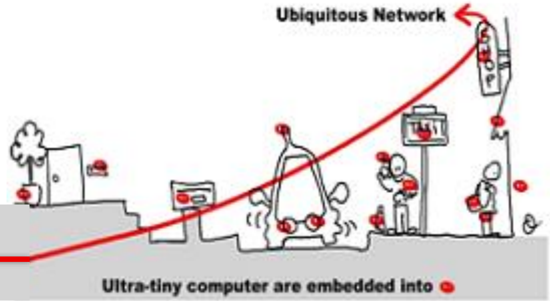
X1next = X0;

orange = not X0 and not X1 and tick;

green = not X0 and X1 and tick;

red = X0 and not X1 and tick;

# Model Checking

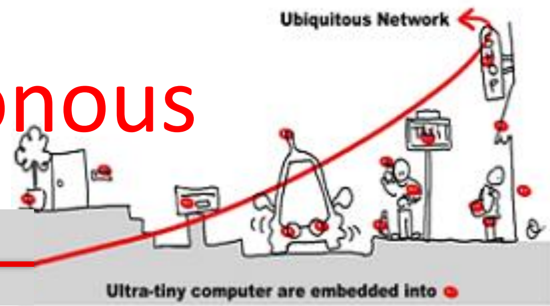



How design  Mealy automata ?

Use synchronous languages to specify critical systems.

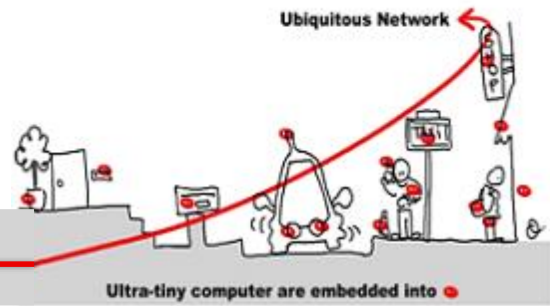
Synchronous programs = Mealy automata

# Model Specification with Synchronous Languages



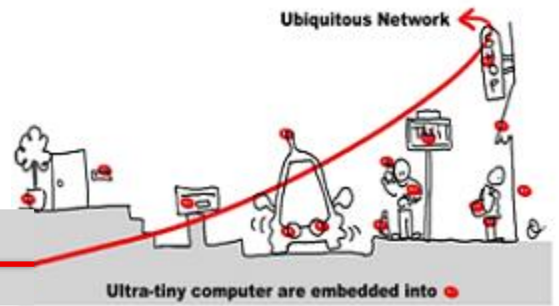

1. Synchronous languages have a **simple formal model** (a finite state machine) making formal reasoning tractable.
2. Synchronous languages support **concurrency** and offer an implicit or explicit means to express parallelism.
3. Synchronous languages are devoted to design **reactive systems**.

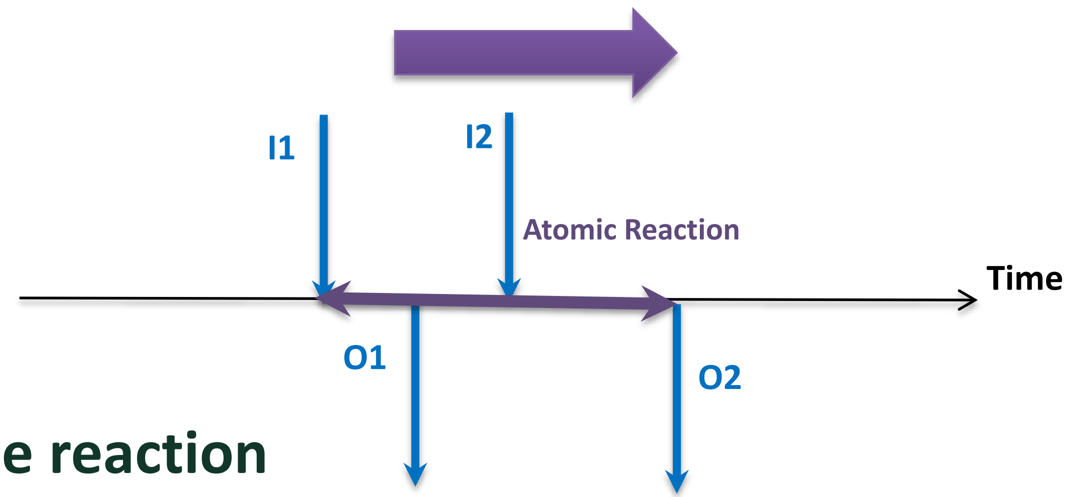
# Determinism & Reactivity




- 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 Modelling



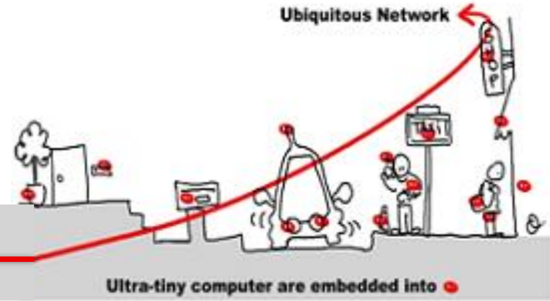

- **Atomic execution of the reaction**
- **Logical time**
- **Well founded**
- **Liable to formal analysis**

# Synchronous Hypothesis

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

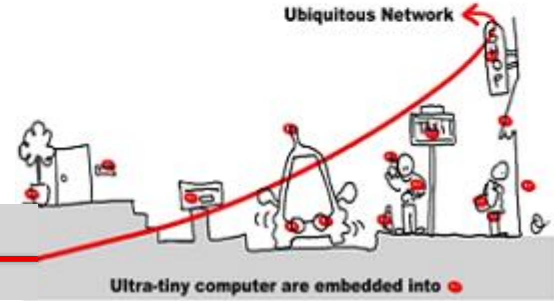
  } Use N as time base
- A reaction executes in one instant.
- Actions that compose the reaction may be partially ordered.

# 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.

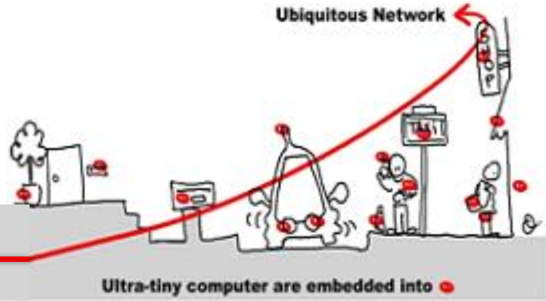
# Reactive ?



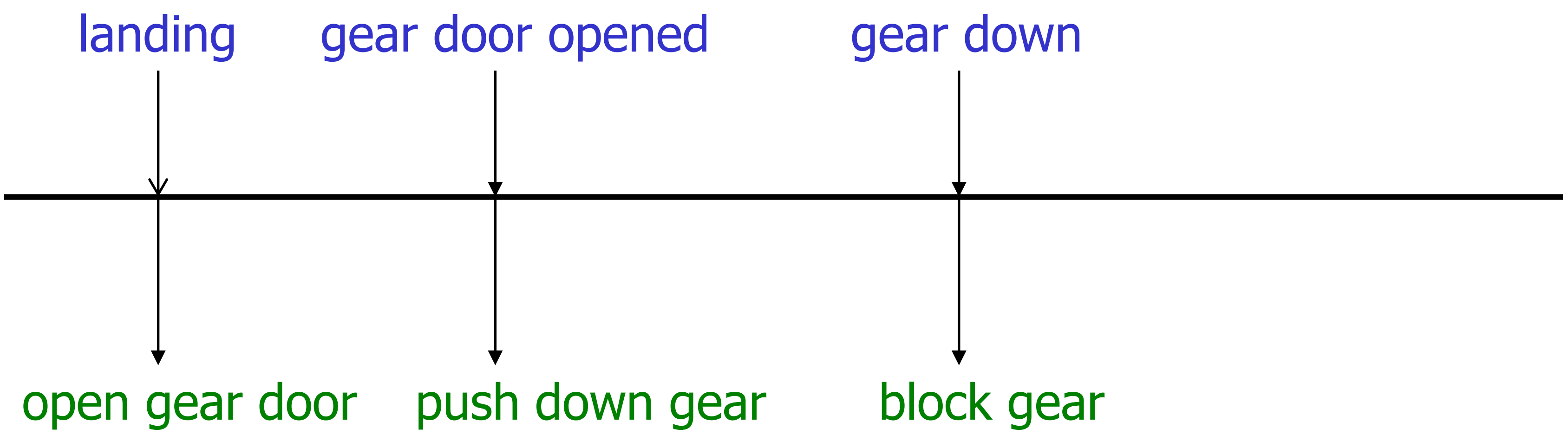

- Different ways to “react” to the environment:
  – Event driven system:
    - Receive events
    - Answer by sending events
  – Data flow system:
    - Receive data continuously
    - Answer by treating data continuously also

**Some systems have components of both kinds**

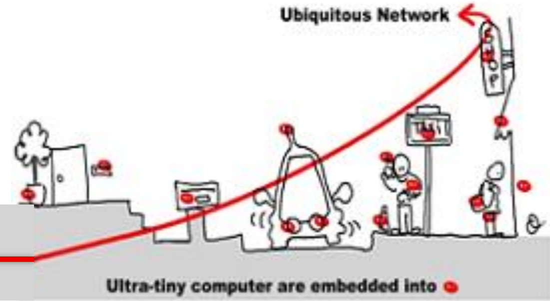
# Event Driven Reactive System



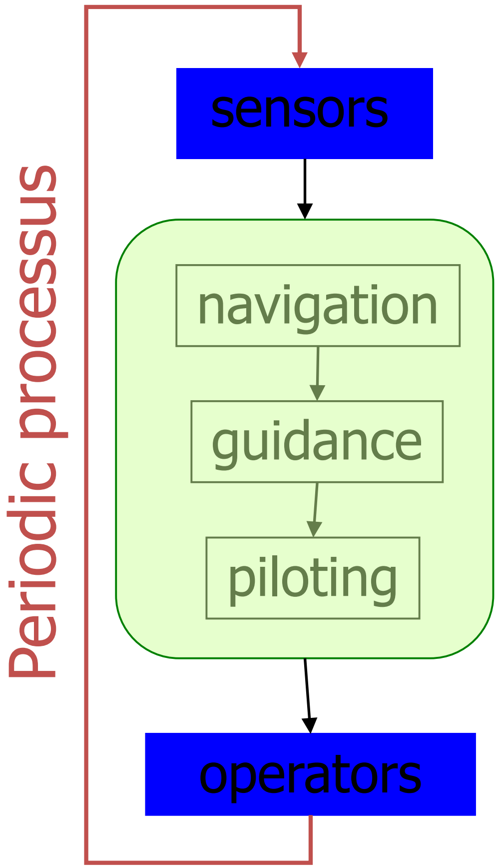

## Langing gear management



# Data Flow Reactive System (Example)




## Control/Command vehicle



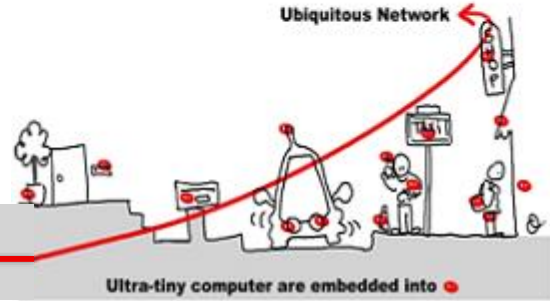

- get measures
- where am I ?
- where go I ?
- command computation
- command to operators

# 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

Synchronous programs = Mealy Automata

# 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

@Inbook{Clavel2007, author="Clavel, Manuel and Dur{\'a}n, Francisco and Eker, Steven and Lincoln, Patrick and Mart{\'i}-Oliet, Narciso and Meseguer, Jos{\'e} ar
@Inbook{Clavel2007, author="Clavel, Manuel and Dur{\'a}n, Francisco and Eker, Steven and Lincoln, Patrick and Mart{\'i}-Oliet, Narciso and Meseguer, Jos{\'e}

# Properties Checking

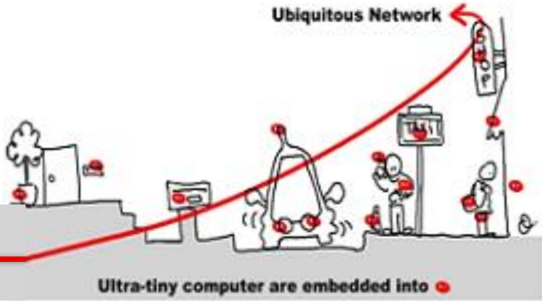

- Liveness Property $\Phi$ :
  - $\Phi \Rightarrow$ automata $B(\Phi)$
  - $\mathbb{L}(B(\Phi)) = \varnothing$ decidable
  - $\Phi \mid= \mathcal{M}$ : $\mathbb{L}(\mathcal{M} \otimes B(\sim\Phi)) = \varnothing$

Reference:

"LTL Model Checking, in All About Maude- A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic"

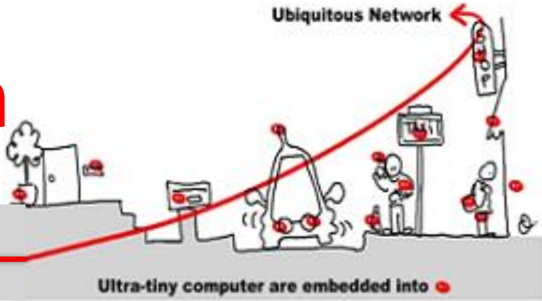
Pages 385-418, Ed: Springer Berlin Heidelberg

# Safety Properties




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

# Safety Properties Verification




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

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

Algorithm:

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

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

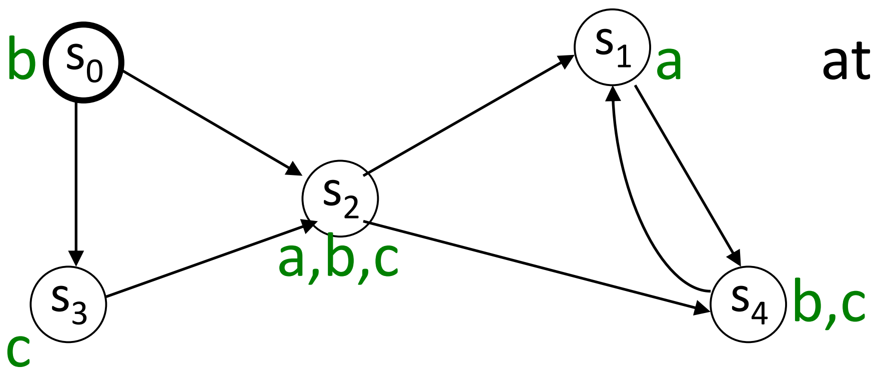
$Sat(\Phi 1 \text{ or } \Phi 2) = Sat(\Phi 1) \cup Sat(\Phi 2)$

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

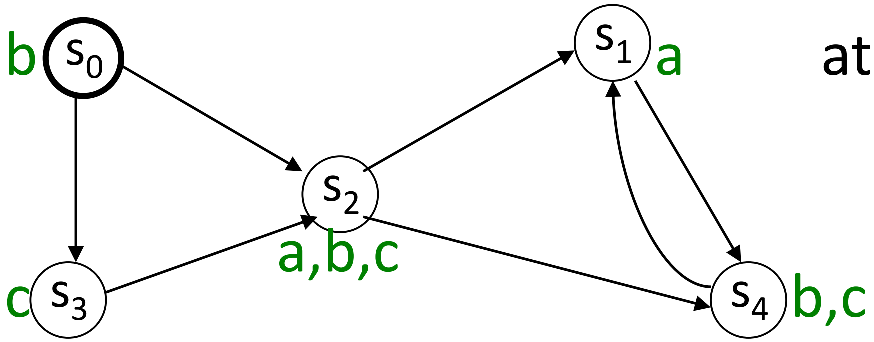
$$gfp\ (\Gamma(x) = \text{Sat(a or b)} \cap \text{Pre}(x))$$

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

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

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

# Example




atomic formulas: a, b, c

EG (a or b)

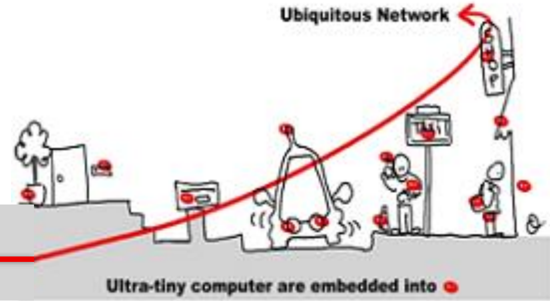
$\Gamma(\{s_0, s_1, s_2, s_3, s_4\}) = \{s_0, s_1, s_2, s_4\}$

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

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

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

# Model Checking Implementation

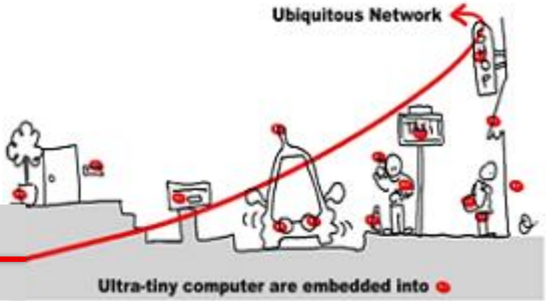



- Problem: the size of automata
- Solution: symbolic model checking
- Usage of BDD (Binary Decision Diagram) to encode both automata and formula.
- Each Boolean function has a unique representation
- Shannon decomposition:
    - $f(x_0,x_1,\ldots,x_n) = f(1, x_1,\ldots, x_n) \vee f(0, x_1,\ldots,x_n)$

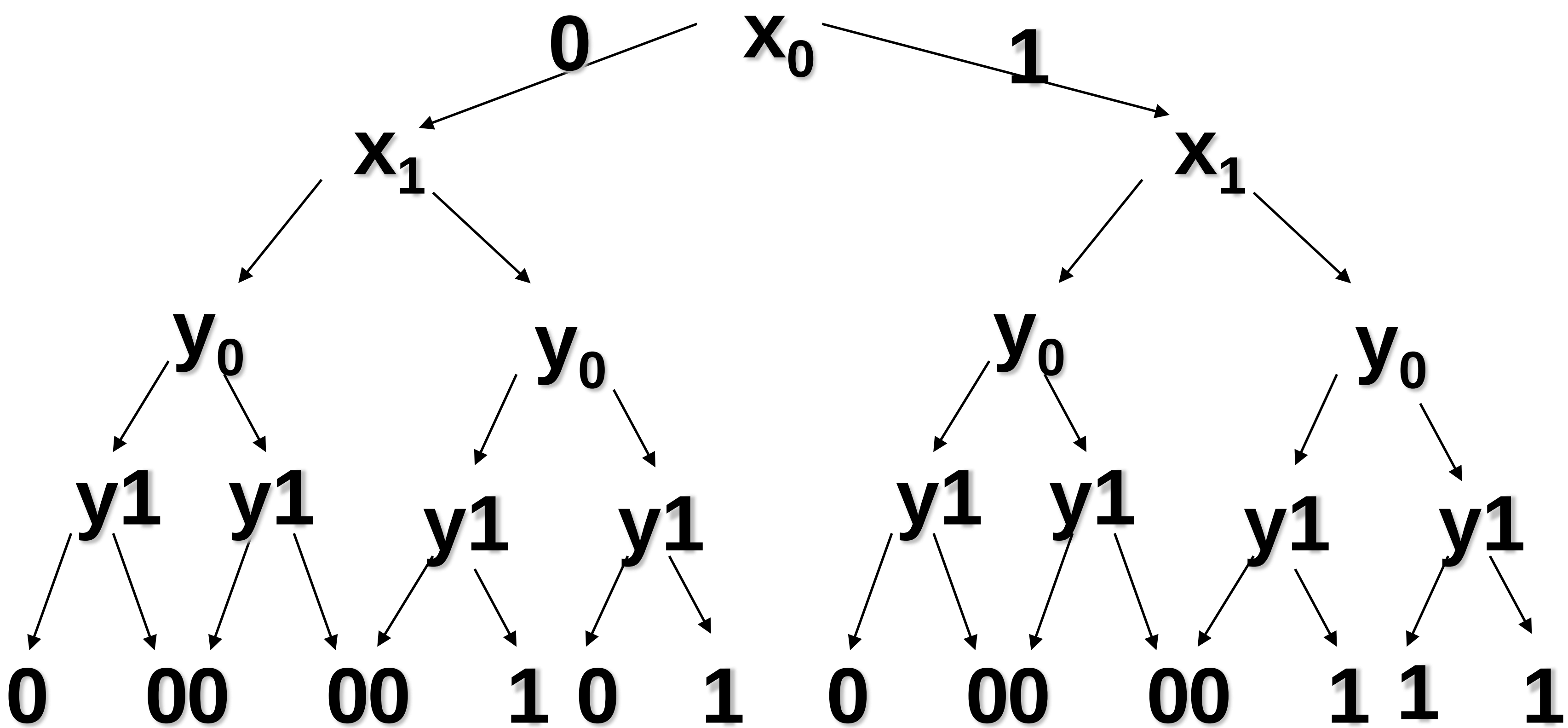
# Model Checking Implementation

- 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 ⇔ g)
  - Share common sub graphs

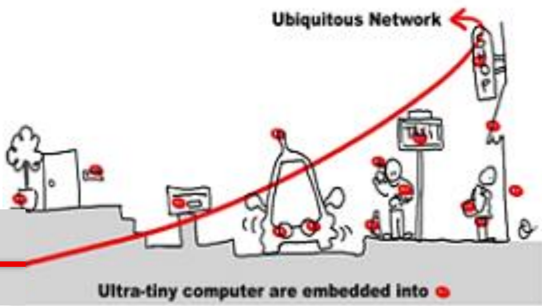
# Model Checking Implementation (2)



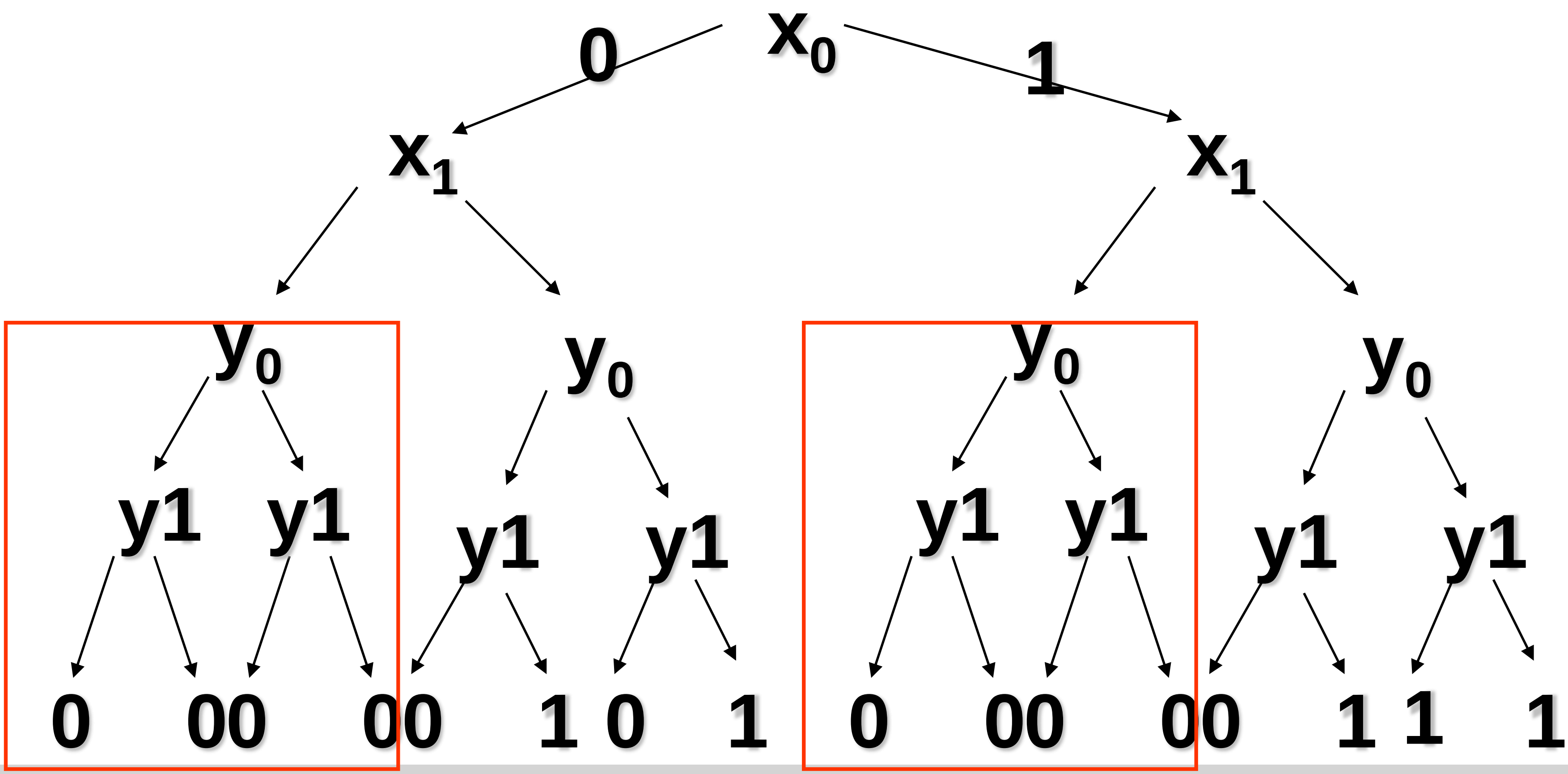

**$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$**



# Model Checking Implementation (2)

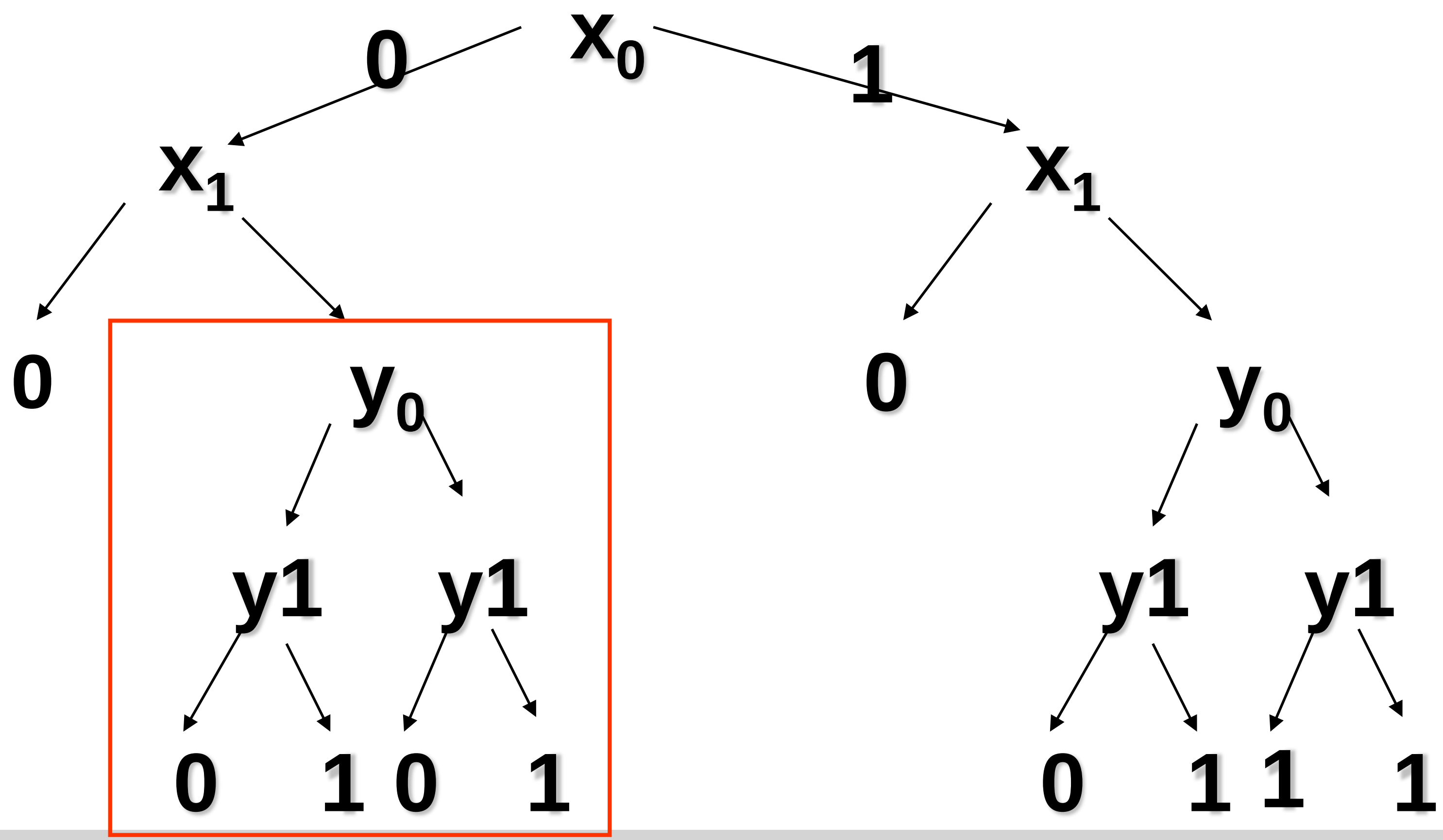



$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$



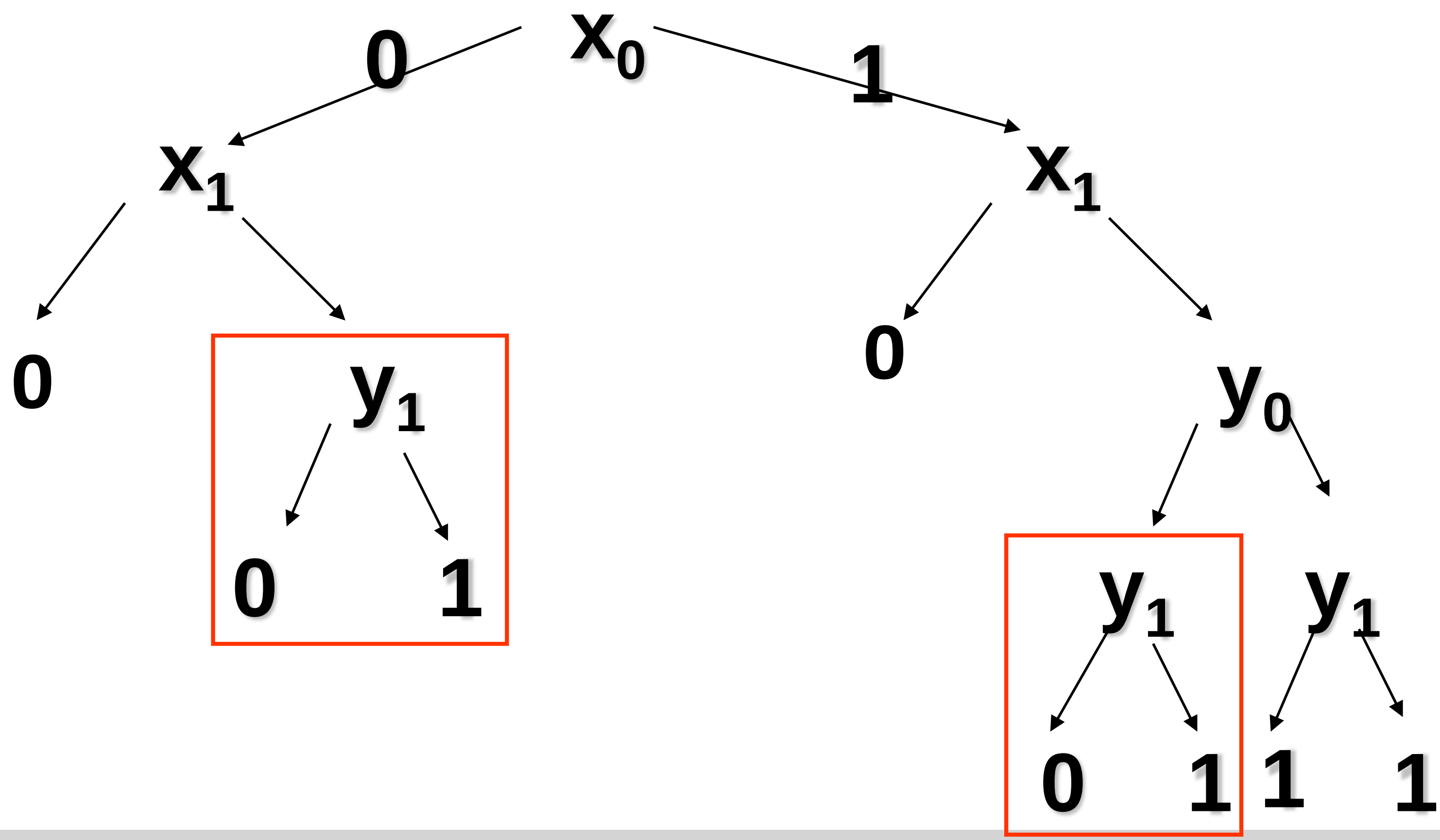
# Model Checking Implementation (2)

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



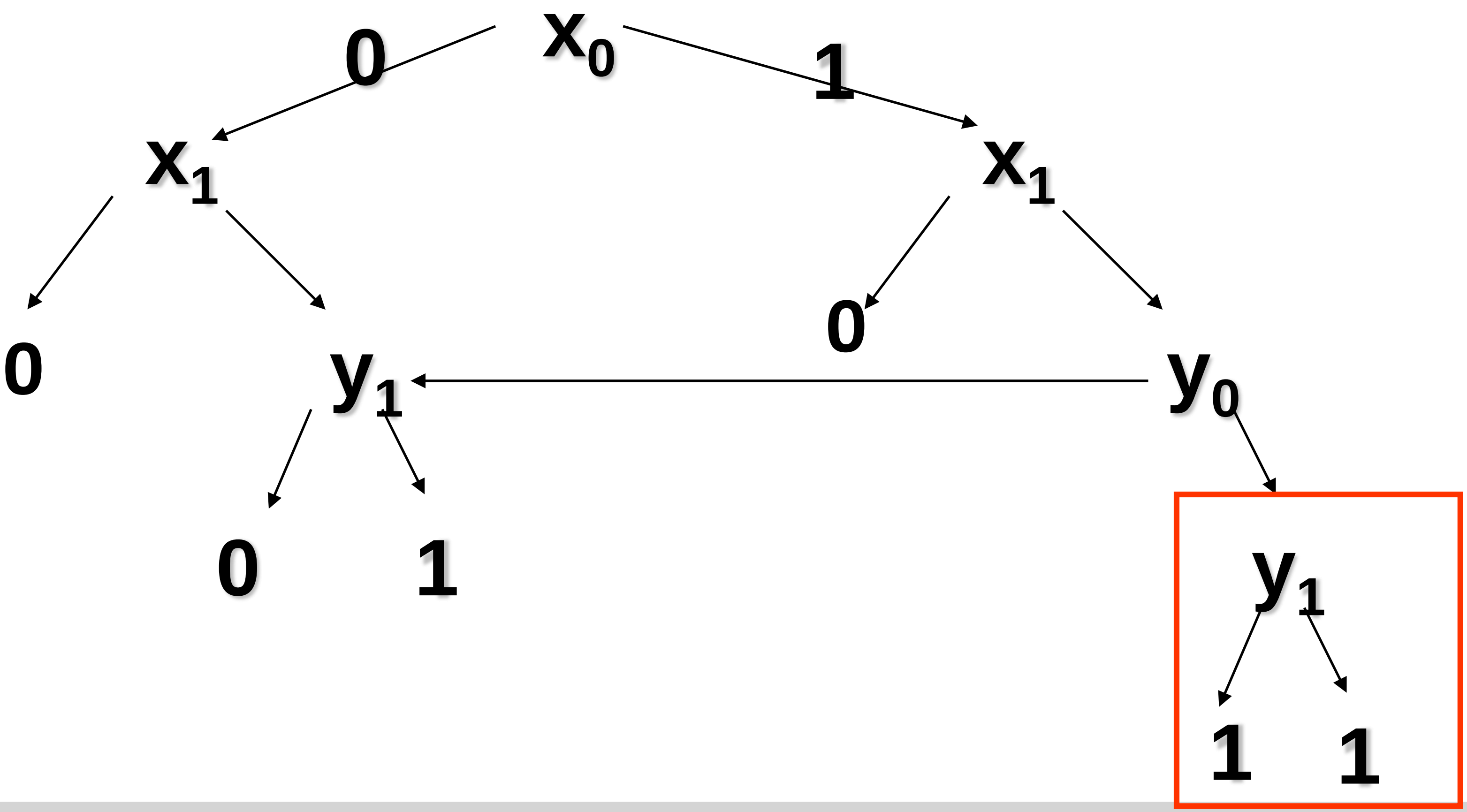
# Model Checking Implementation (2)

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

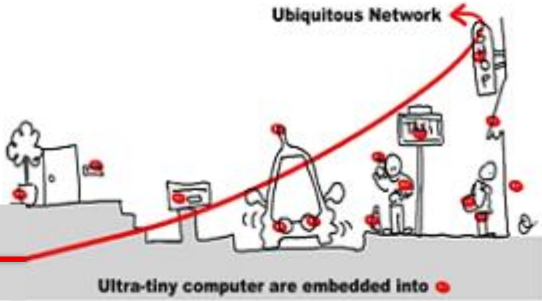


# Model Checking Implementation (2)

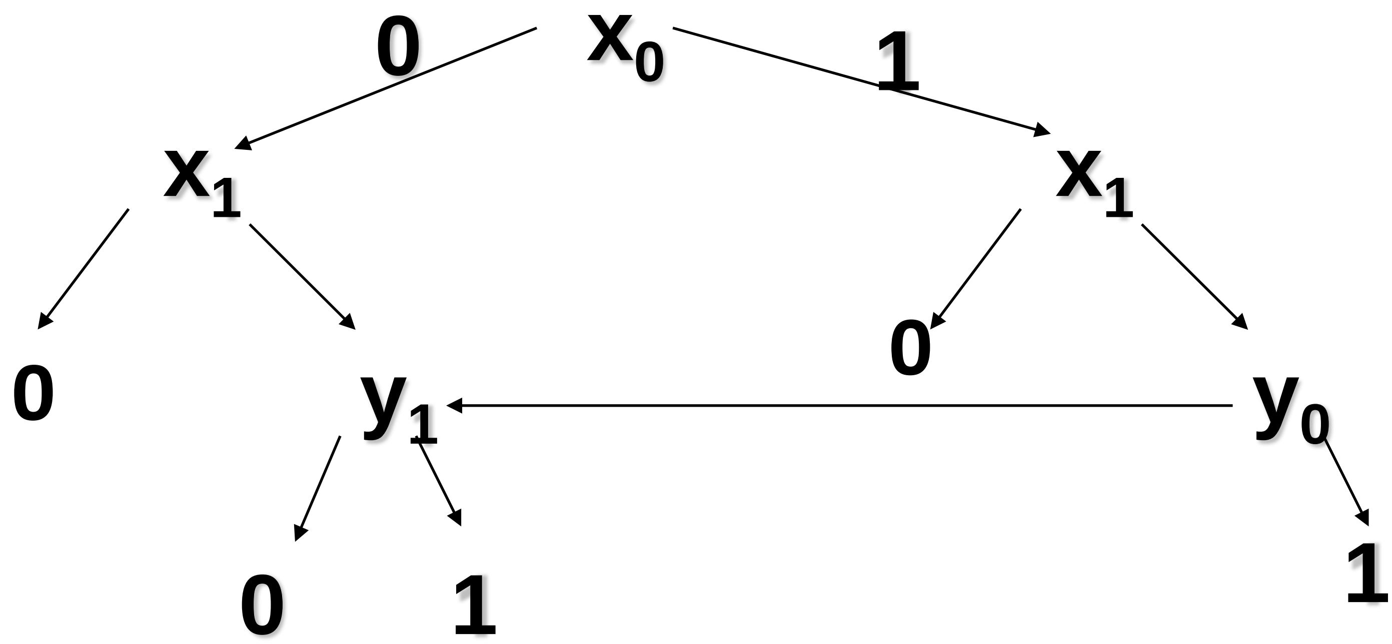
$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$



# Model Checking Implementation (2)

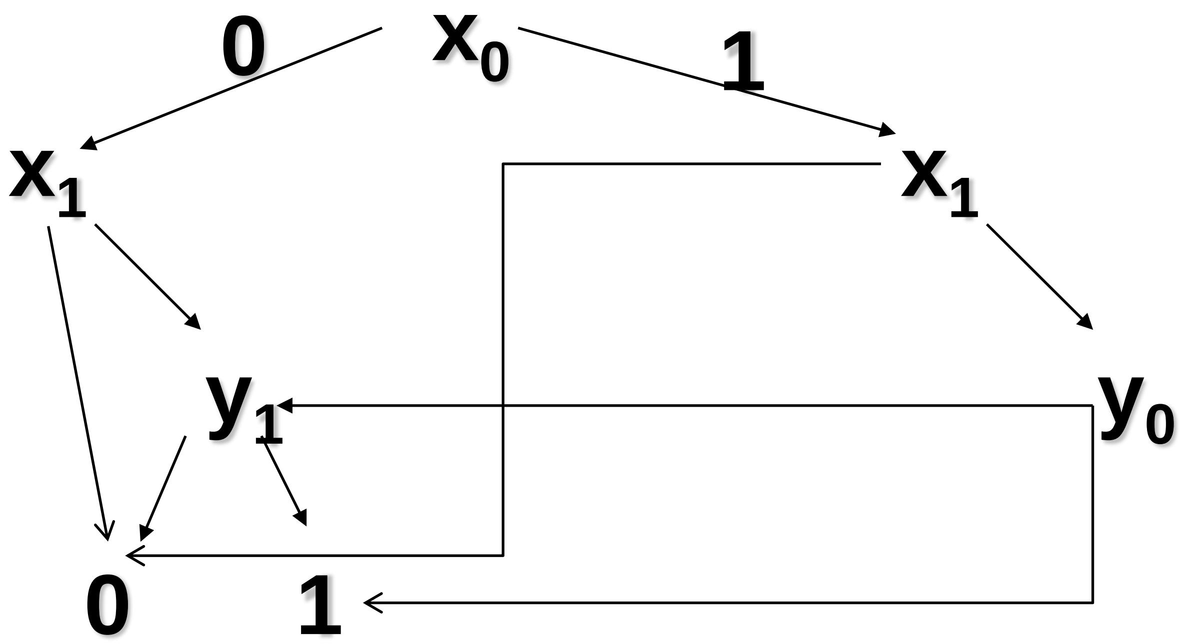



$(x_1 \wedge y1) \vee (x_0 \wedge y_0 \wedge x_1)$

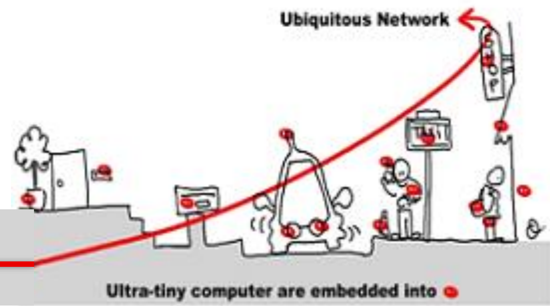


# Model Checking Implementation (2)

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

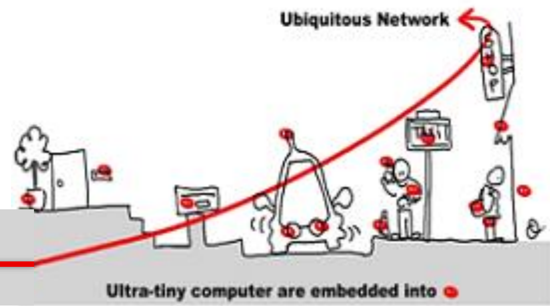


# Model Checking Implementation(3)




- 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)

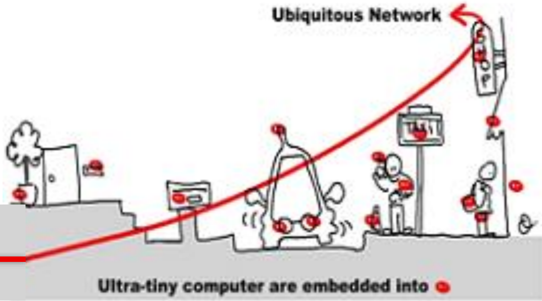



- But BDD efficiency depends on the number of variables
- Other method: SAT-Solver
  - Sat-solvers answer the question: given a propositional formula, is there exist a valuation of the formula variables such that this formula holds
  - first algorithm (DPLL) exponential (1960)

# Model Checking Implementation (4)

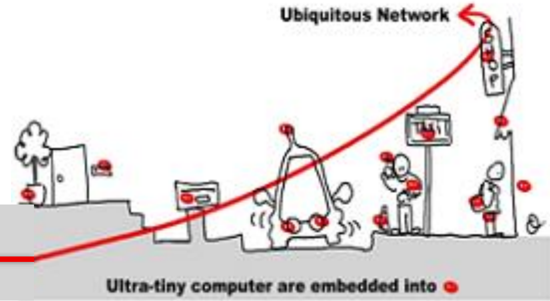
- SAT-Solver algorithm:
  - formula ➔ CNF formula ➔ set of clauses
  - heuristics to choose variables
  - deduction engine:
    - propagation
    - specific reduction rule application (unit clause)
    - Others reduction rules
  - conflict analysis + learning

# 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 $\Phi$ is true can be reduce to the satisfaction of a propositional formula
  - theorem: given $\Phi$ a temporal property and $\mathcal{M}$ a model, then $\mathcal{M} \;|\!= \Phi \Rightarrow \exists\, n$ such that $\mathcal{M} \;|\!=_n \Phi$ ( $n < |S| \,.\, 2^{|\Phi|}$)

# Bounded Model Checking

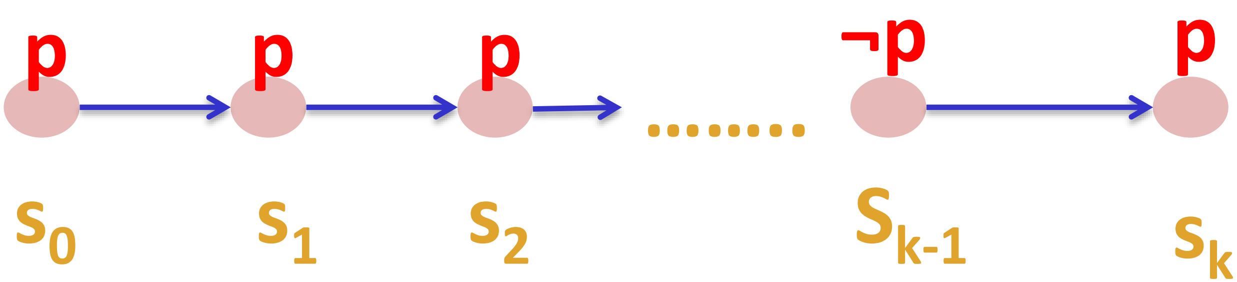



- SAT-Solver are used in complement of implicit (BDD based) methods.
- $\mathcal{M} |= \Phi$
  - verify $\neg \Phi$ on all paths of length k (k bounded)
  - useful to quickly extract counter examples

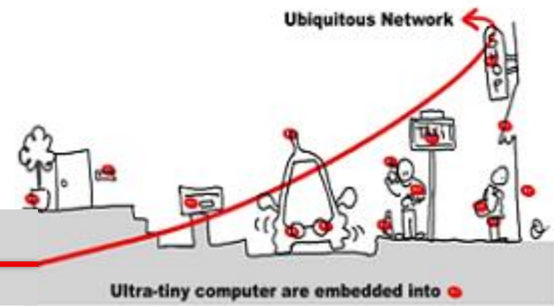
# Bounded Model Checking

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:

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

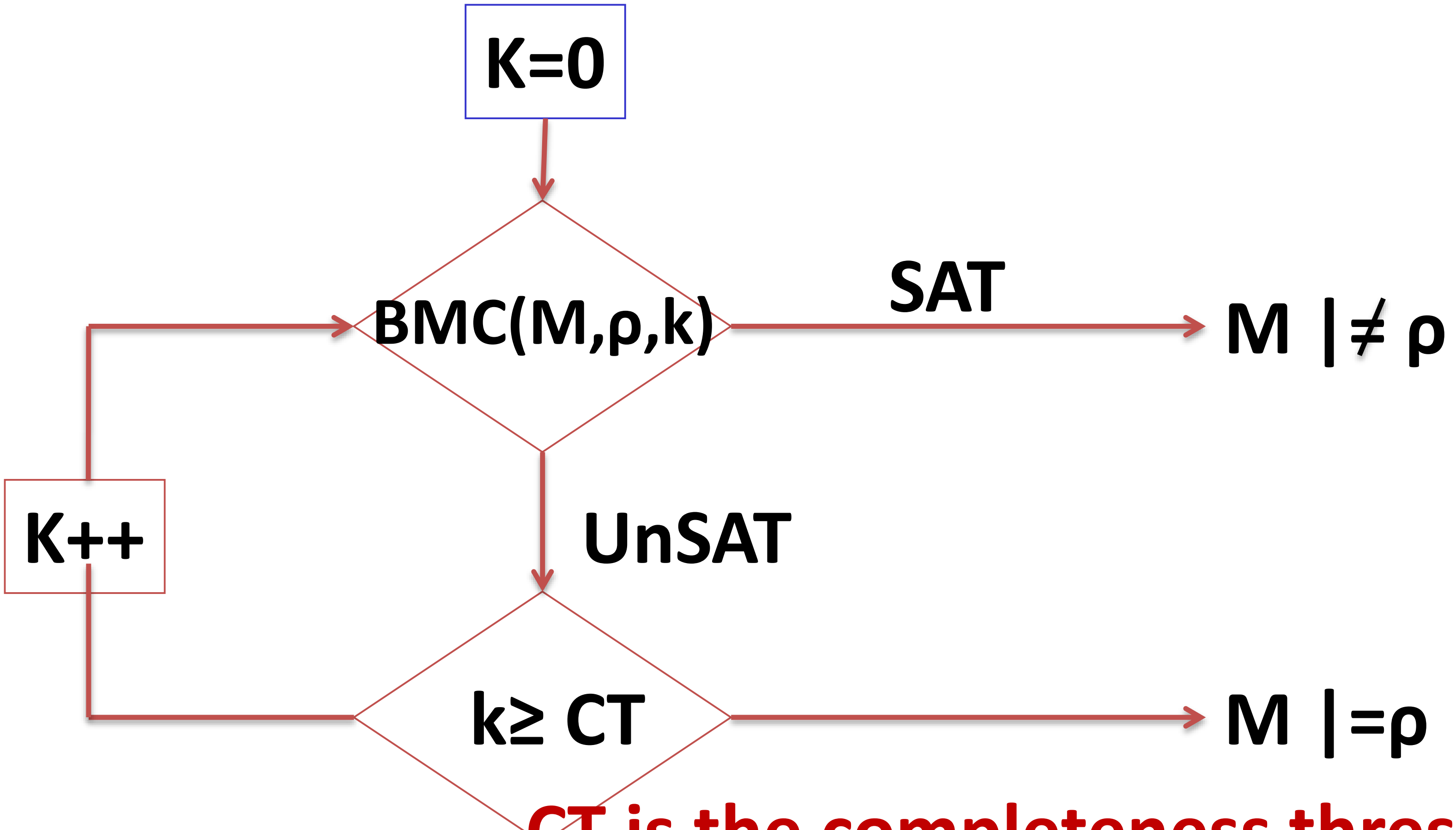
The property **p** fails in one of the k steps

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

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

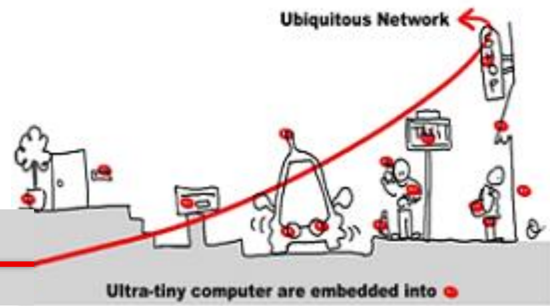
$$\Omega(k) = I(s_0) \wedge \left( \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \right) \wedge \left( \bigvee_{i=0}^{k} \neg p(s_i) \right)$$

# Bounded Model Checking



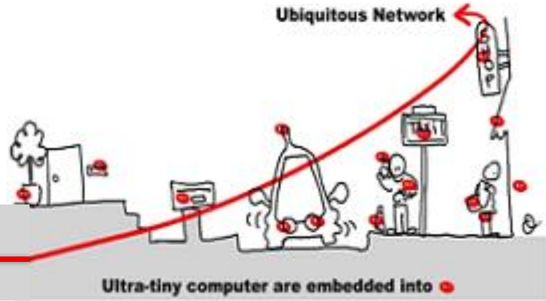

**CT is the completeness threshold**

# Bounded Model Checking

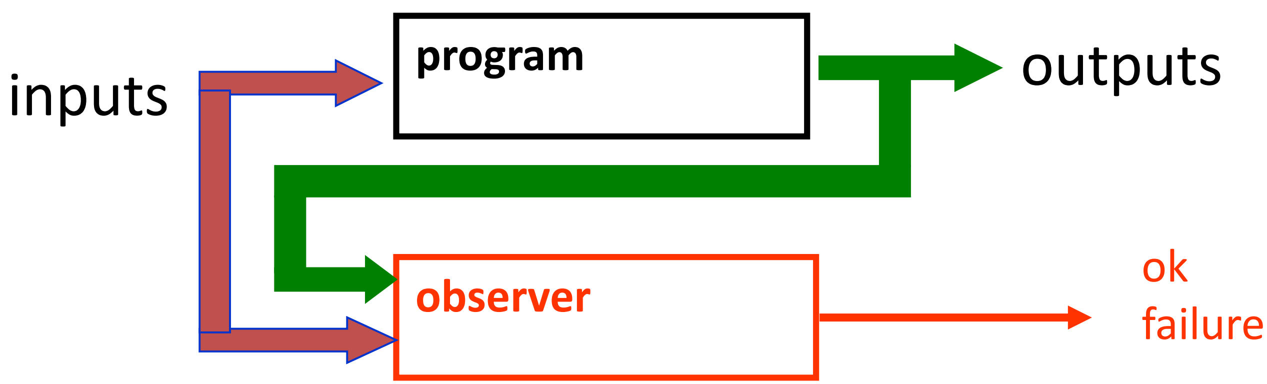



- Computing CT is as hard as model checking.
- Idea: Compute an over-approximation to the actual CT
  - Consider the system *as a graph.*
  - Compute *CT from structure of the graph.*
- Example: for **AGρ** properties, CT is the longest shortest path between any two reachable states, starting from initial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

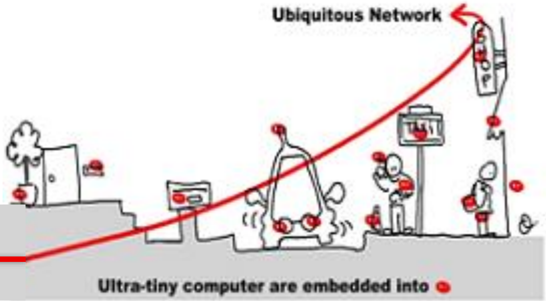
# Model Checking with Observers



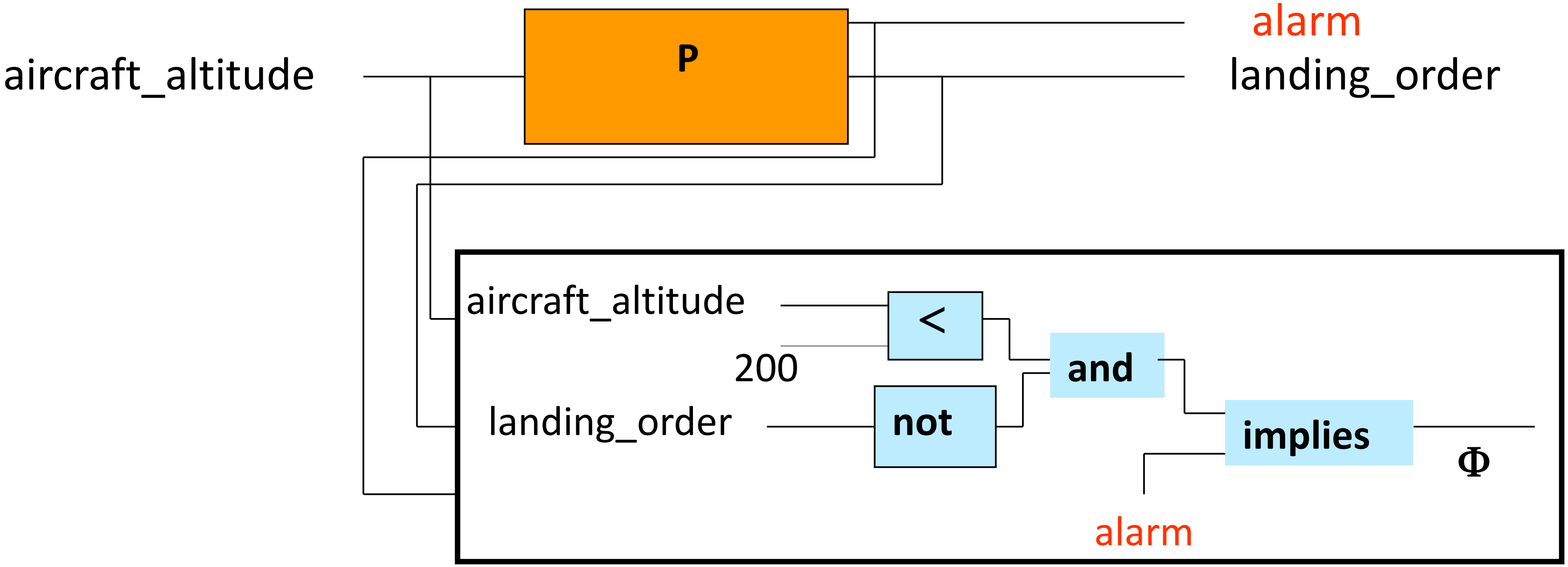

- Express safety properties as observers.
- An observer is a program which observes the program and outputs ok when the property holds and failure when its fails



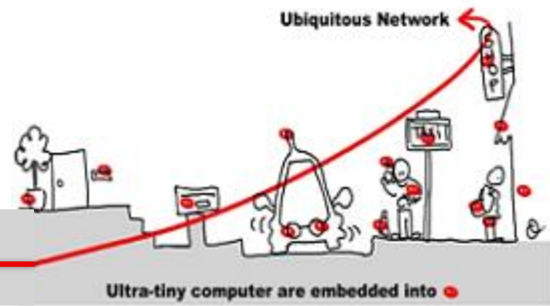
# Model Checking with observers (2)




**P**: aircraft autopilot and security system

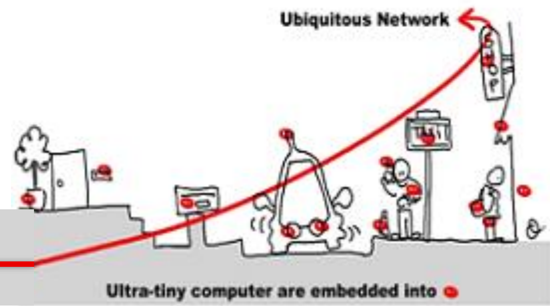


# Properties Validation

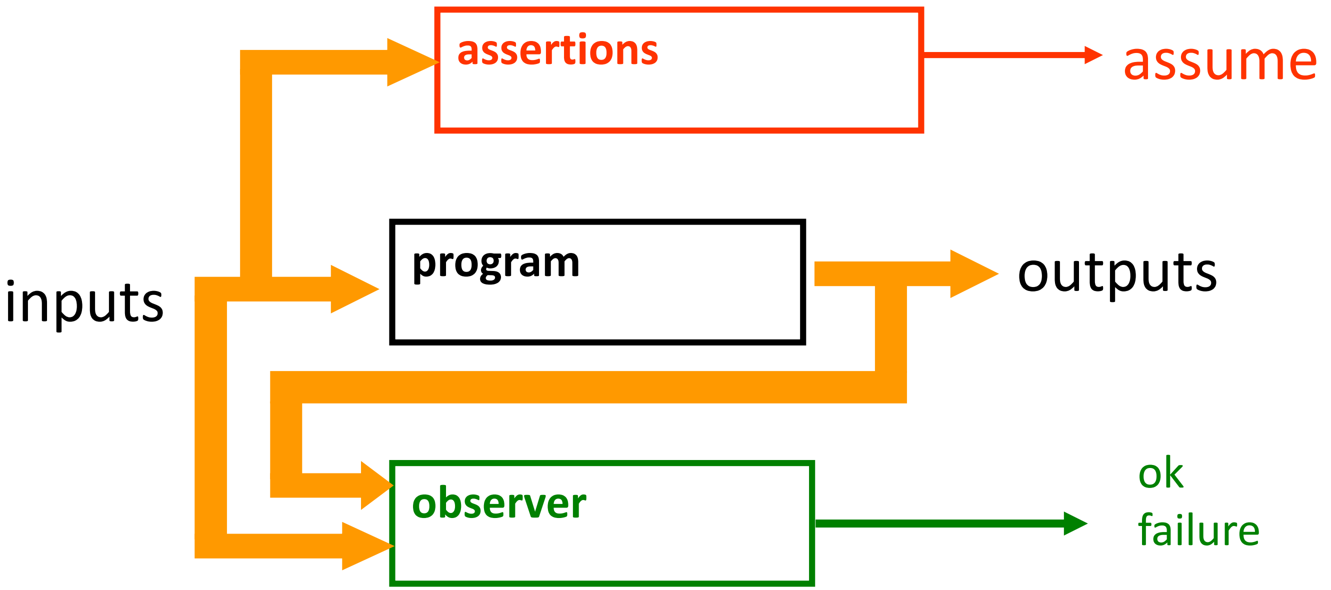



- Taking into account the environment
  - without any assumption on the environment, proving properties is difficult
  - but the environment is indeterminist
    - Human presence no predictable
    - Fault occurrence
    - …
  - Solution: use assertion to make hypothesis on the environment and make it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

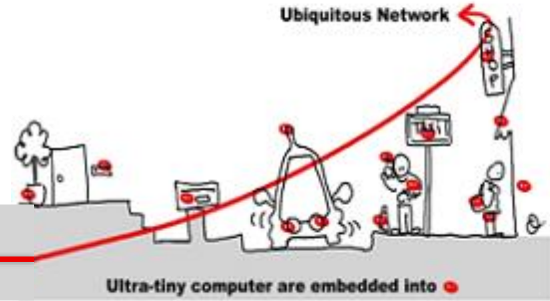
# Properties Validation (2)



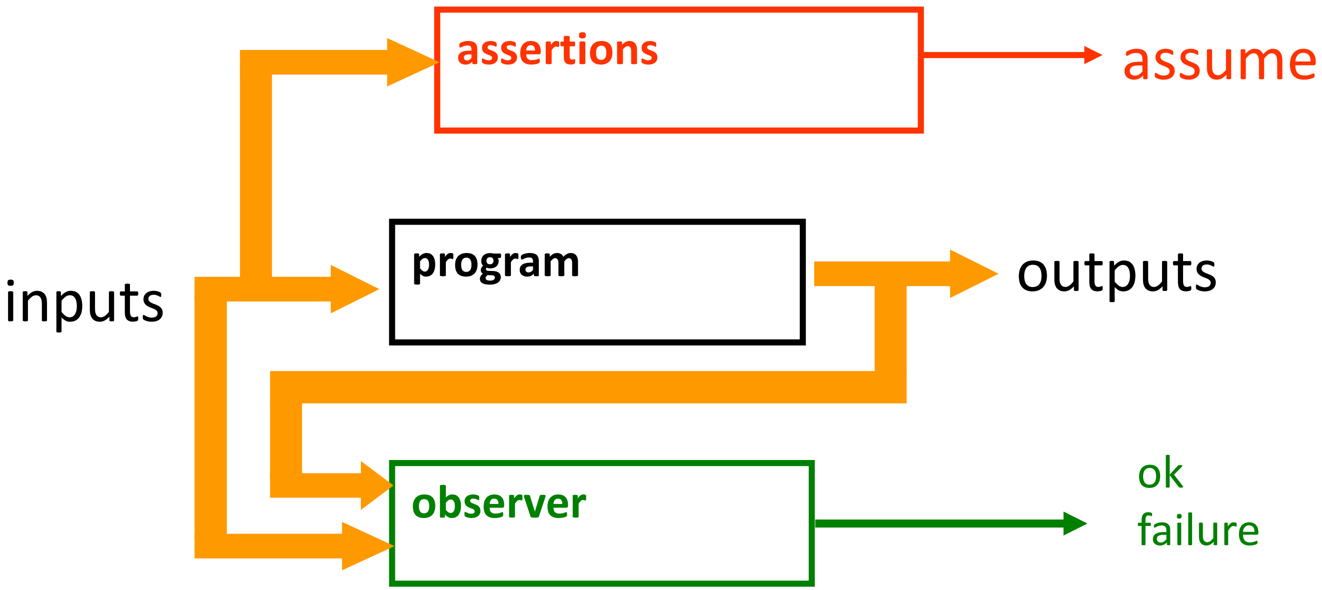

- Express safety properties as observers.
- Express constraints about the environment as assertions.



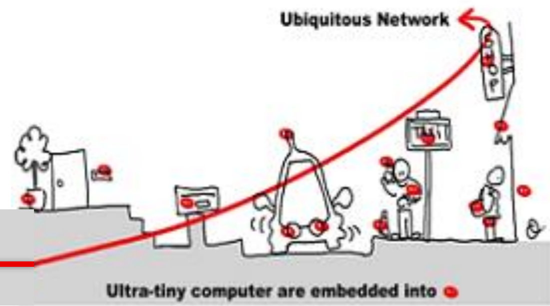
# Properties Validation (3)




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

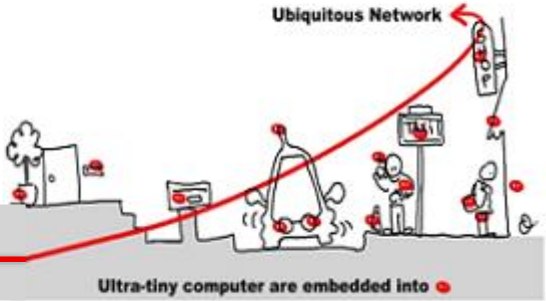


# Outline



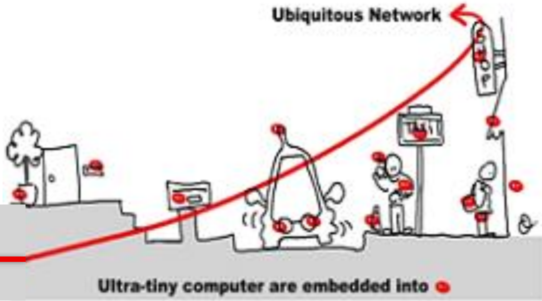

1. Critical system validation
2. Model-checking solution
    1. Model specification
    2. Model-checking techniques
3. Application to middleware for IoT
    1. Introduction in middleware design of synchronous components to allow validation
    2. Synchronous /asynchronous issues

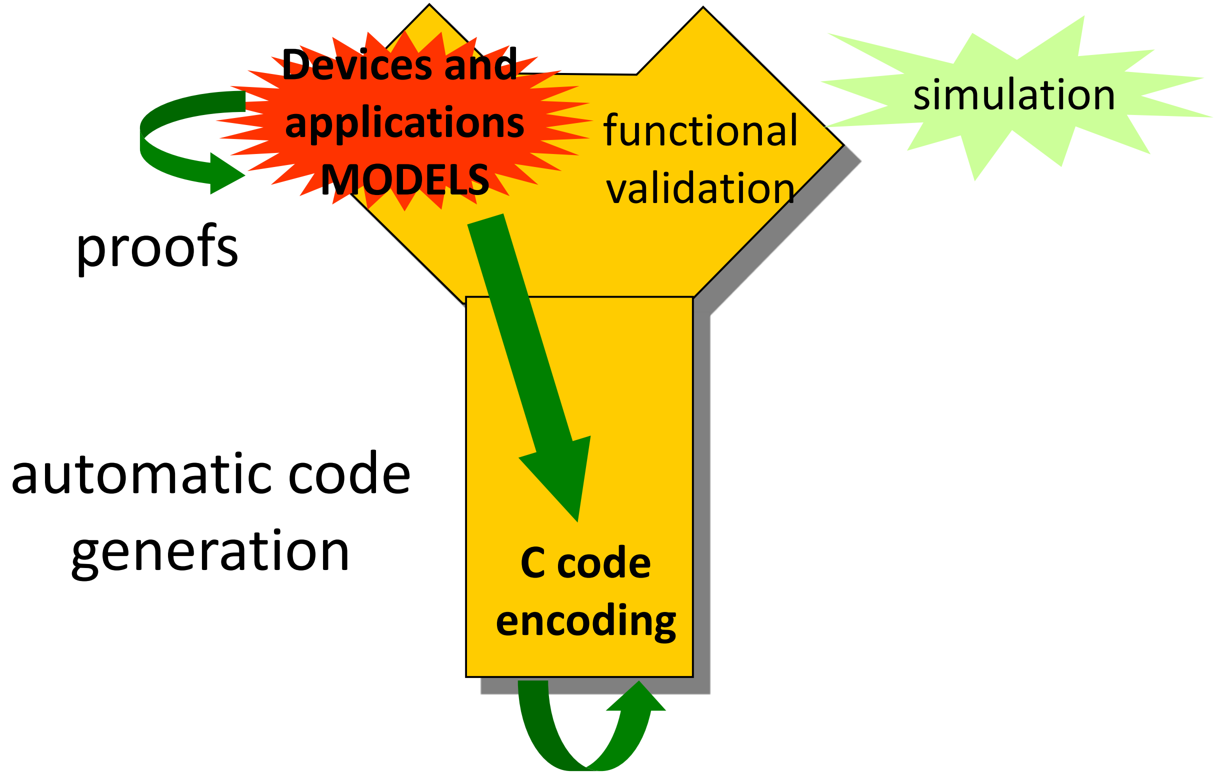
# Practical Issues



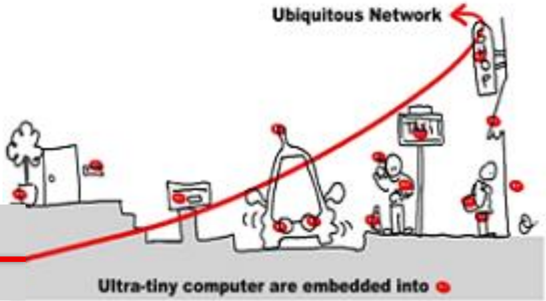

## Application to Middleware for IoT

# Application to Middleware



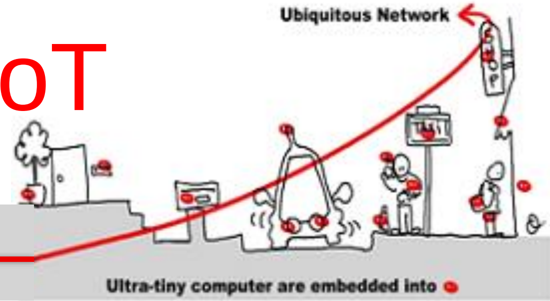
# Synchronous Models




To sum up :

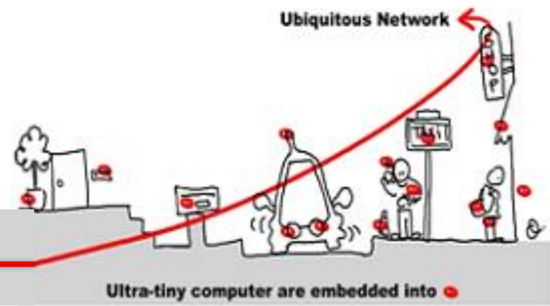
1. Synchronous models can be designed as **event-driven controllers** or **as data flow operator networks**
2. They always represent automata
3. Model-checking techniques apply

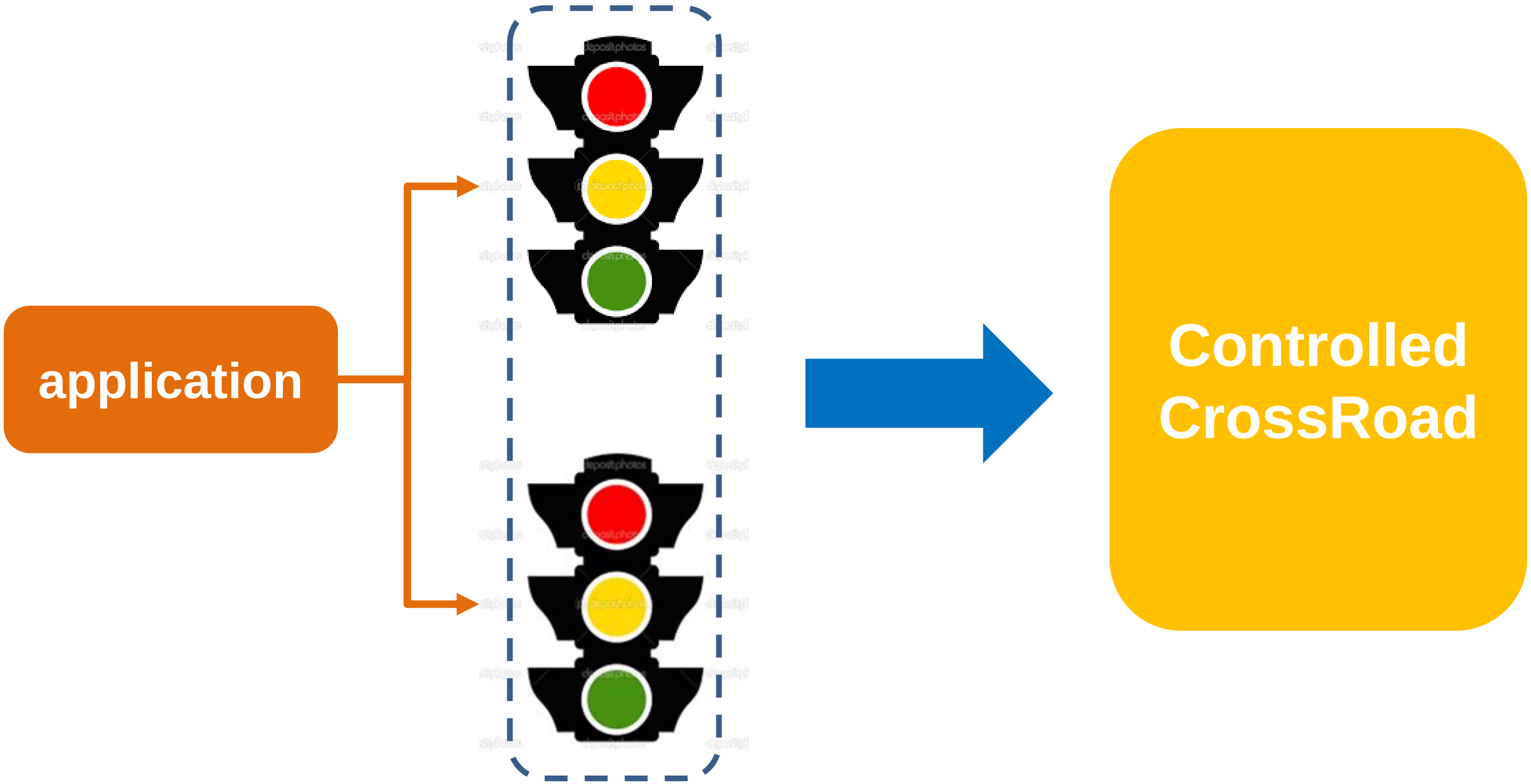
# Application to Middleware for IoT



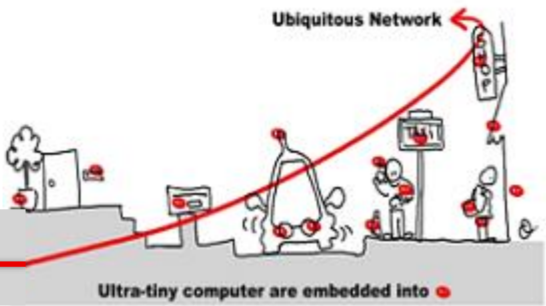

- Our goal is to ensure safety for applications using and managing services.
- Devices will have a synchronous component to allow model-checking techniques application as validation
- Synchronous component to express constraints between concurrent services
- Synchronous parallelism as composition

# Use Case


Ubiquitous Network
Ultra-tiny computer are embedded into

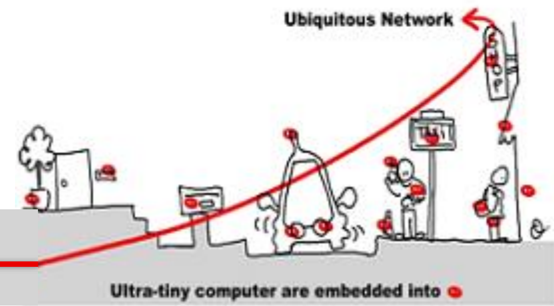

application
Controlled CrossRoad

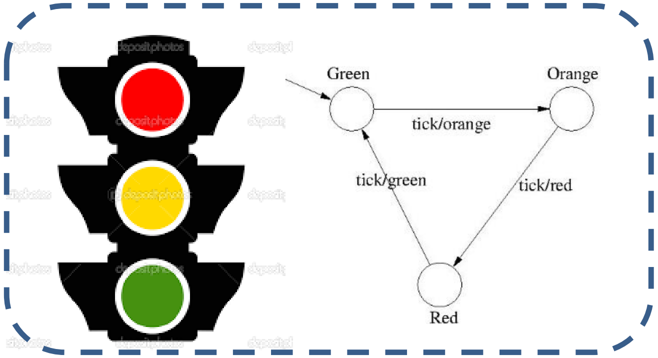
# Use Case

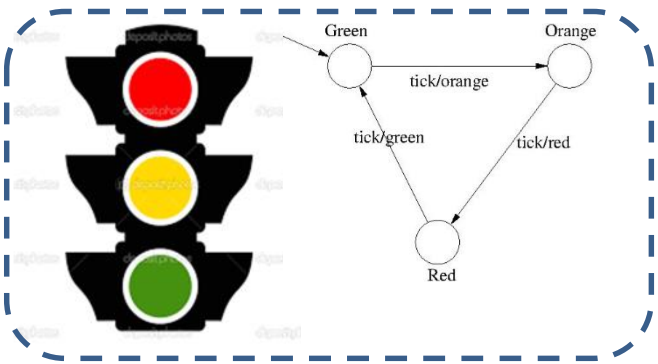



- **Use case**: manage a crossroad
  1. 2 roads (EW and NS) with a traffic light each
  2. Each traffic light has 3 exclusive outputs: red, yellow, green.
  3. Constraints:
     - ❖ each traffic light works following the sequence: green -> yellow -> red
     - ❖ traffic lights work in a consistent way (no 2 green lights simultaneously)

# Use Case Implementation



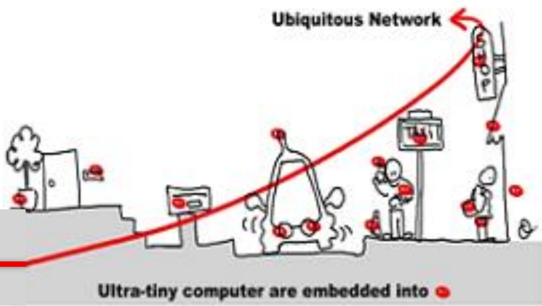




+ CONSTRAINTS

Controlled CrossRoad Component

# Use Case Implementation

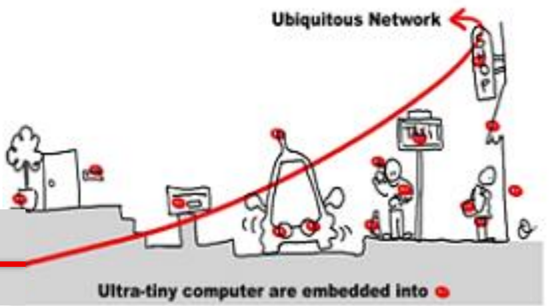



How specify  the traffic light synchronous model ?

How specify both device and application constraints as synchronous models ?

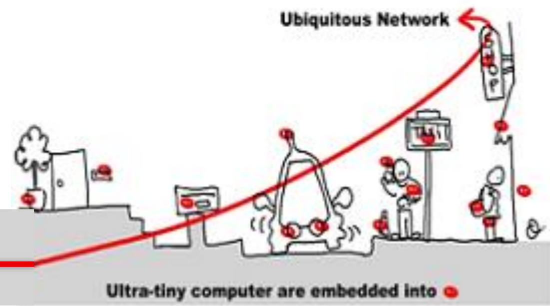
Solution: use a synchronous language

# First Solution: SCADE



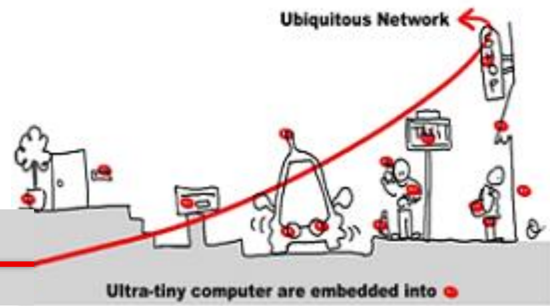

- Scade (Safety-Critical Application Development Environment) has been developed to address safety-critical embedded application design
- The Scade suite KCG code generator has been qualified as a development tool according to DO-178B norm at level A.

# SCADE

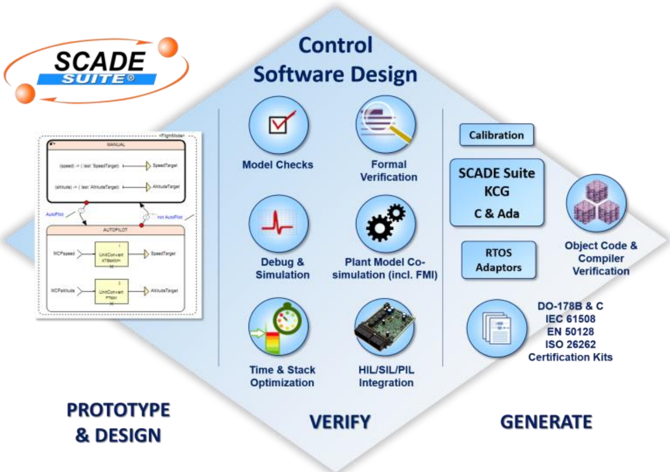



- Scade has been used to develop, validate and generate code for:
  - avionics:
    - Airbus A 341: flight controls
    - Airbus A 380: Flight controls, cockpit display, fuel control, braking, etc,..
    - Eurocopter EC-225 : Automatic pilot
    - Dassault Aviation F7X: Flight Controls, landing gear,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

# SCADE



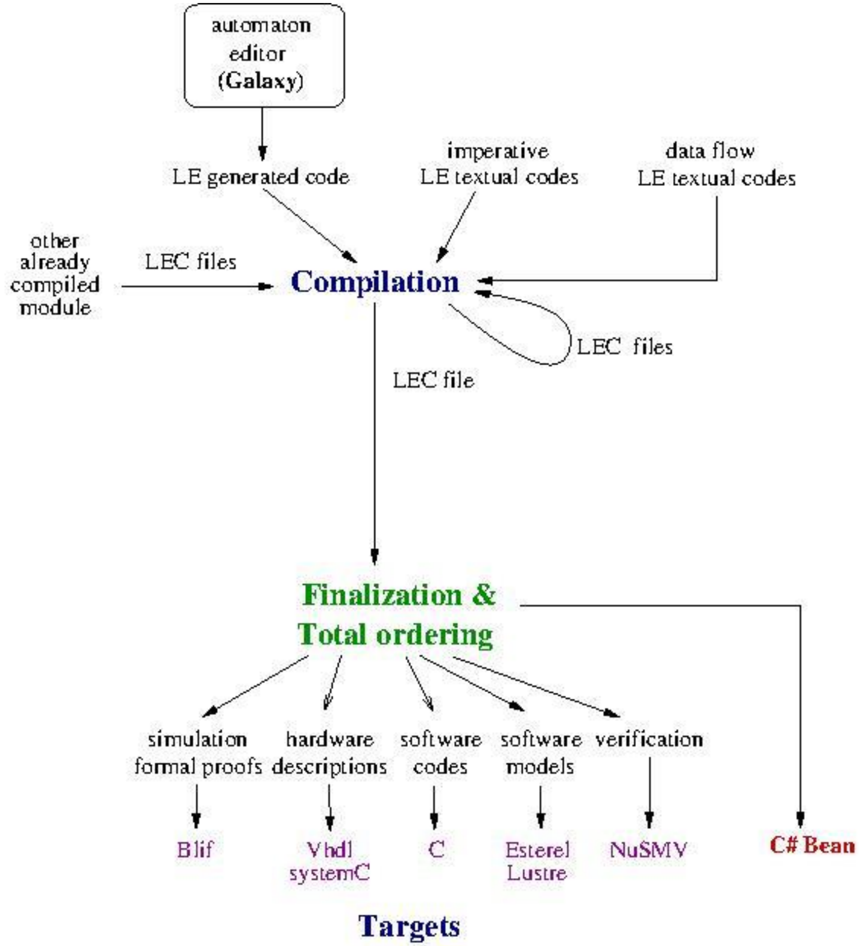

- System Design
  - Both data flows and state machines
- Simulation
  - Graphical simulation, automatic GUI integration
- Verification
  - Apply observer technique
- Code Generation
  - certified C code



# CLEM versus SCADE

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

# CLEM ISSUE




CLEM is a toolkit around the LE synchronous language offering:

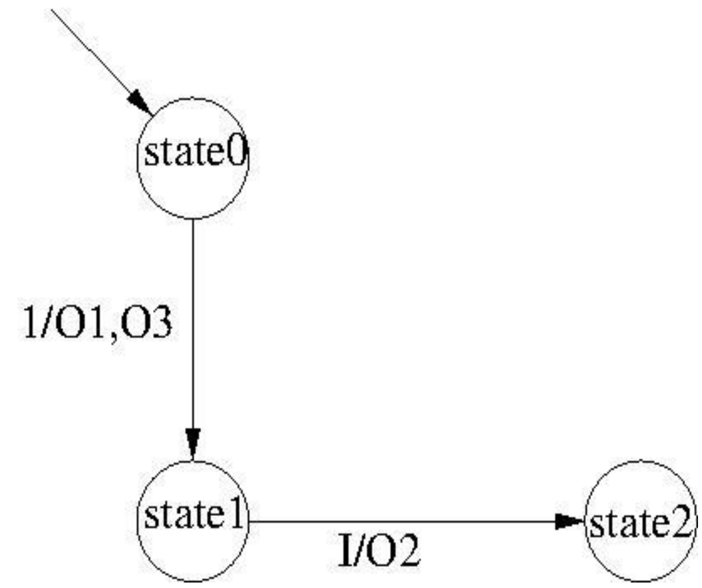
- Modular compilation
- Simulation
- Verification
- Code generation for hardware and software targets (C)

# LE Language

- LE synchronous language
  - Textual imperative language
    - 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;

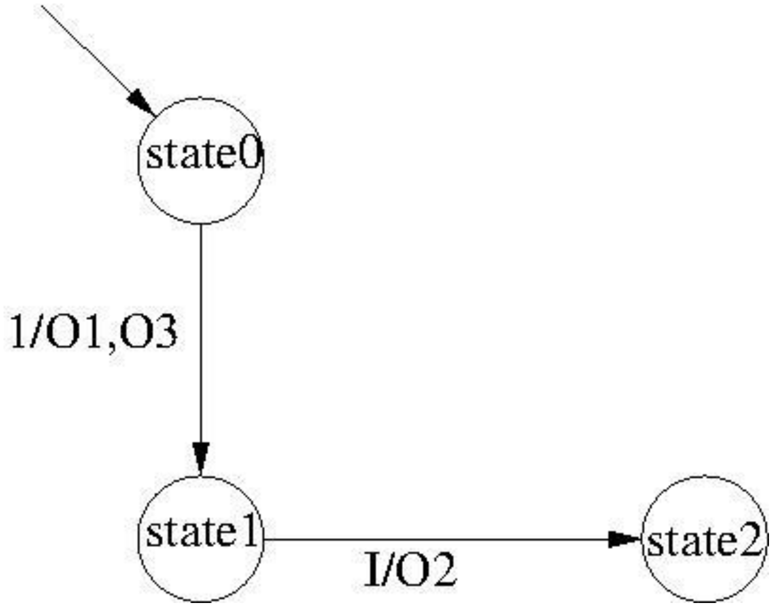
X0next = X0 and not X1;

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

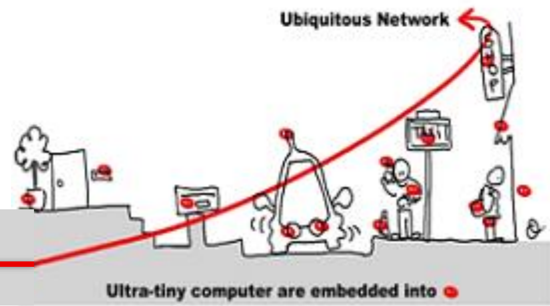
O1 = not X0 and not X1;

O2 = X0 and not X1 and I;

O3 = not X0 and not X1;

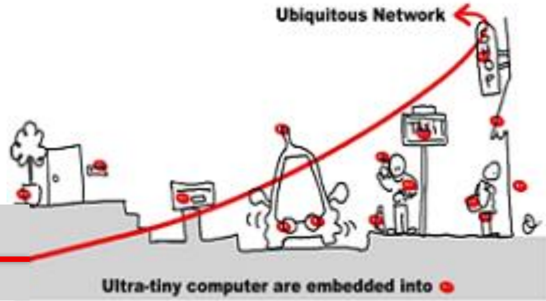


# LE Compilation



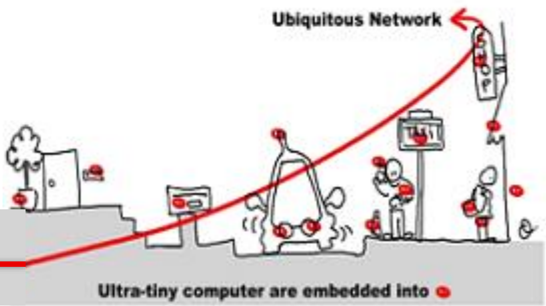

- Compilation into implicit Mealy machines (Boolean equation systems with registers)
- Compilation $\Rightarrow$ sort equation systems
- Challenge: modular compilation ?
  - $\Rightarrow$ face causality problem
  - causality = no evaluation cycle in equation systems
  - total order prevents modularity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

# LE Compilation



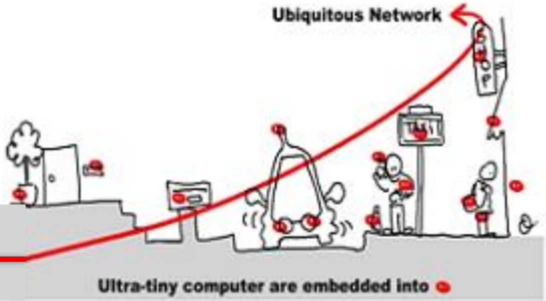

- Sorting algorithms:
  1. Apply CPM on dependency graphs of equation systems to compute ranges of evaluation levels for variables (efficient)
  2. apply fix point theory:
     - Compute variable evaluation levels as fix point of a monotonic increasing function
     - Uniqueness of fixpoints  we can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

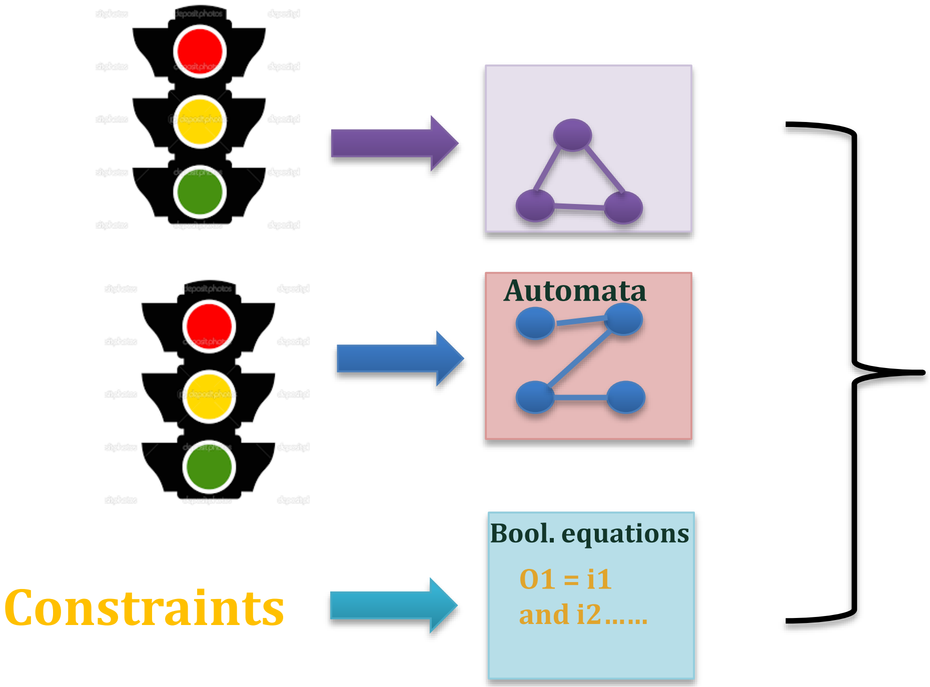
# CLEM Simulation and Verification




- 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



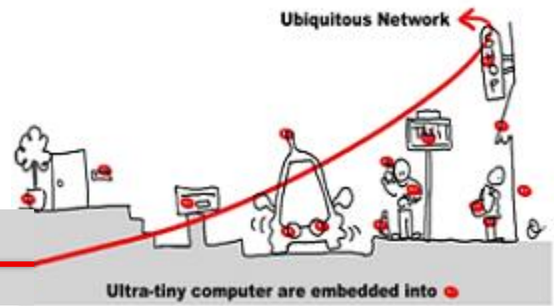




**Synchronous modeling**

↓

**Explicit Mealy machine designed with Galaxy**
**or**
**Implicit Mealy machine designed as Boolean equations in Clem**

# Validation with CLEM




simulation

LE design

Validatation

Generate C code

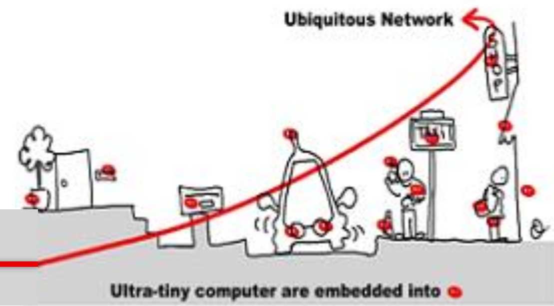
C

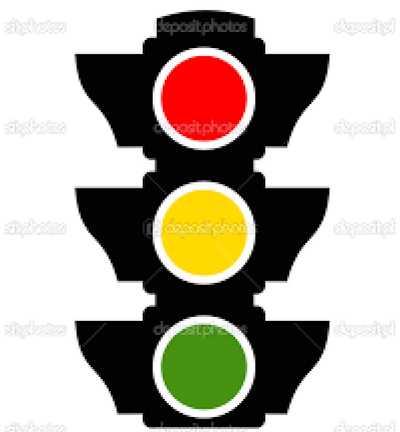
C

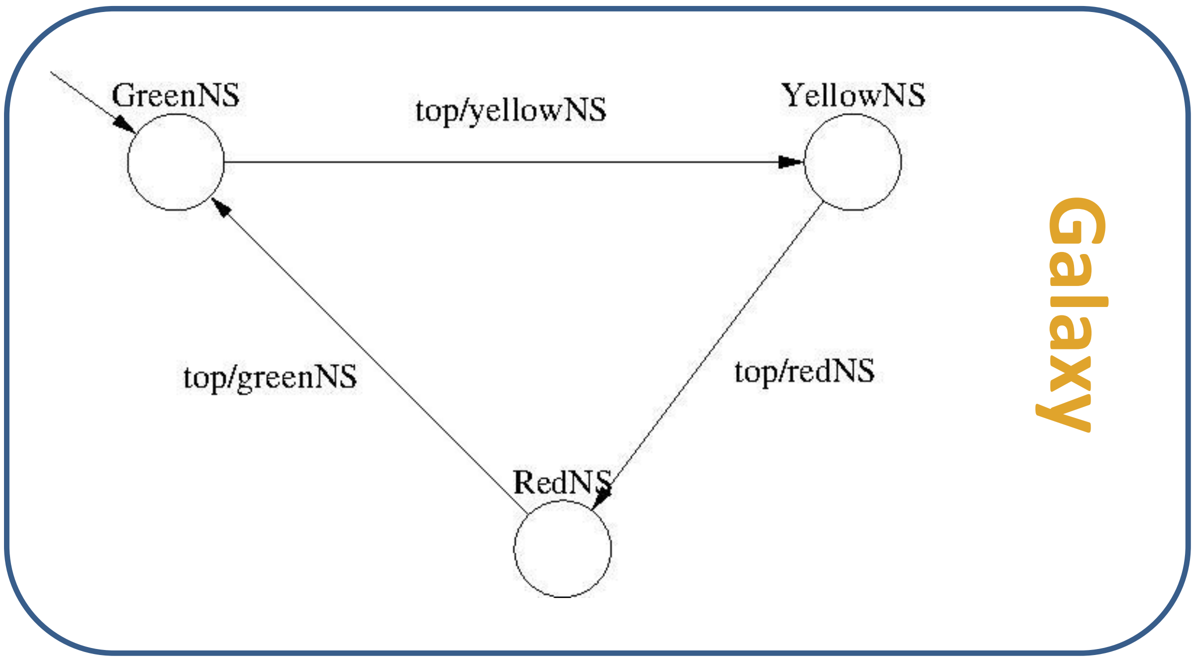
C

validated component

# Use Case Issue in CLEM

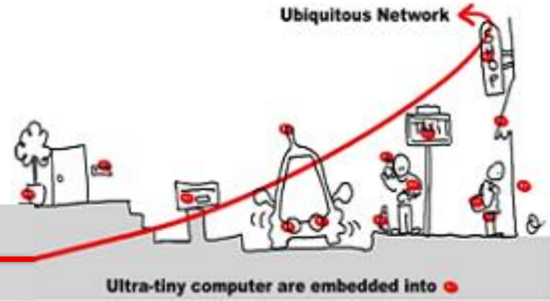


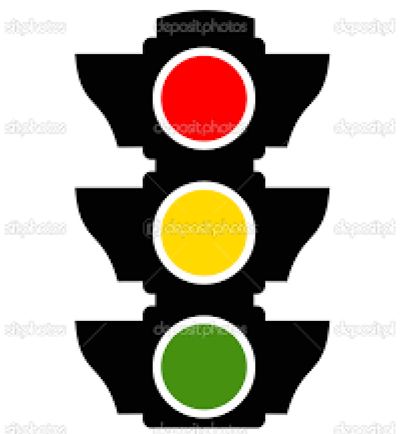





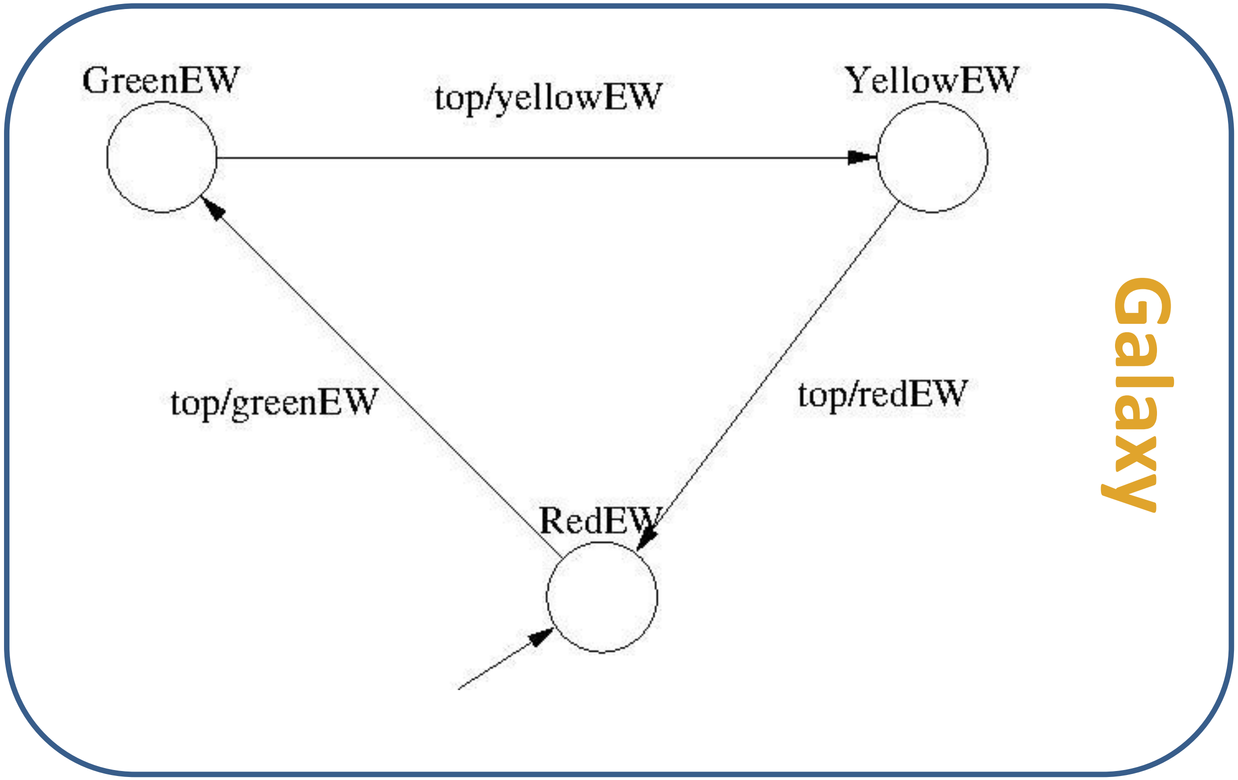

TrafficLight NS

# Use Case Issue in CLEM



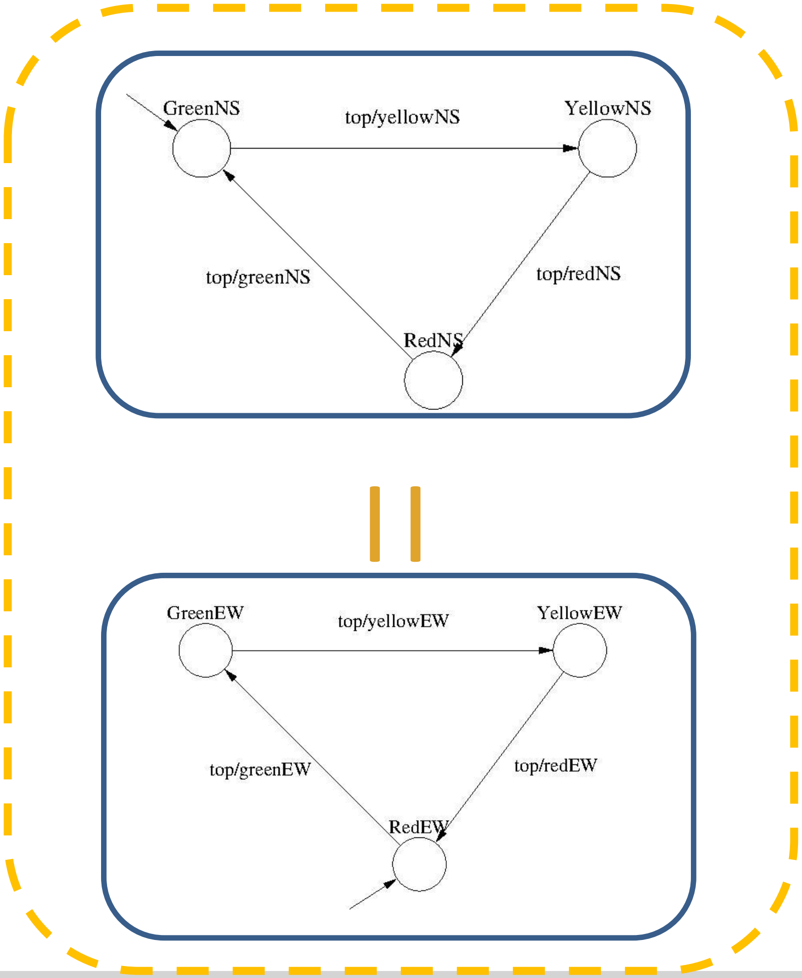



TrafficLight EW



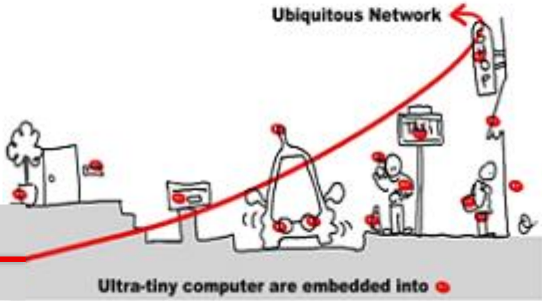
# Use Case Issue in CLEM

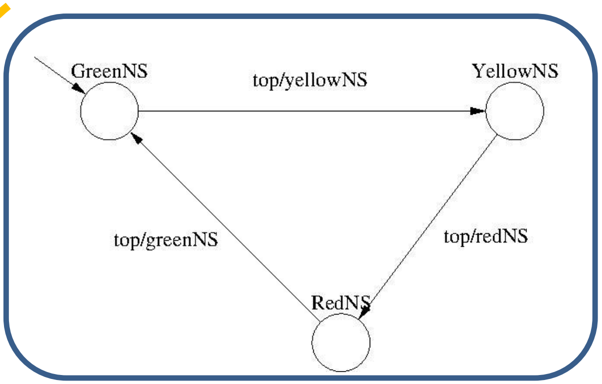
top



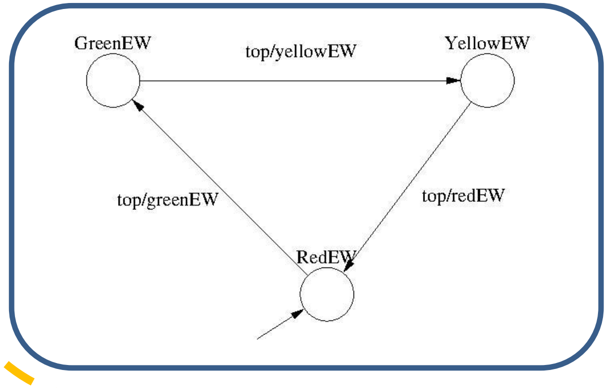

greensNS
redNS
yellowNS
greenEW
redEW
yellowEW

# Verification in CLEM

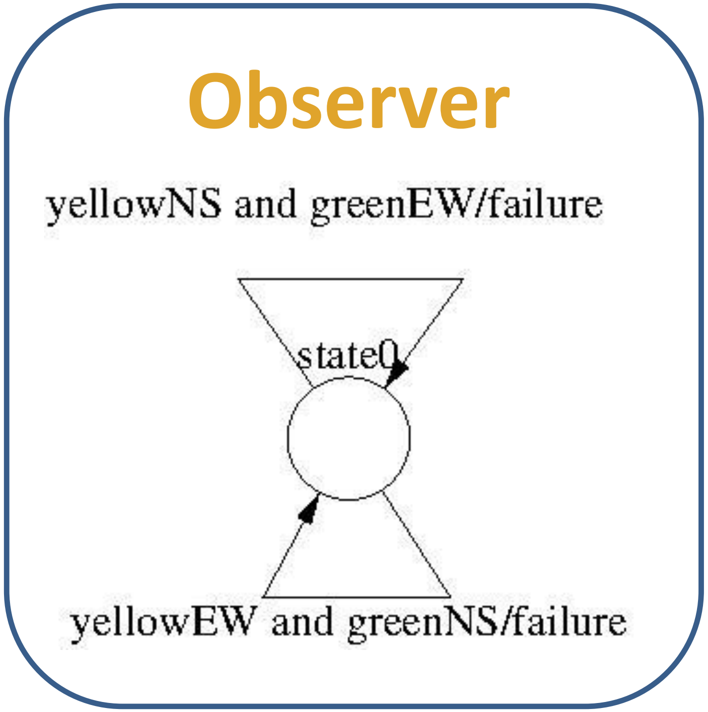



||



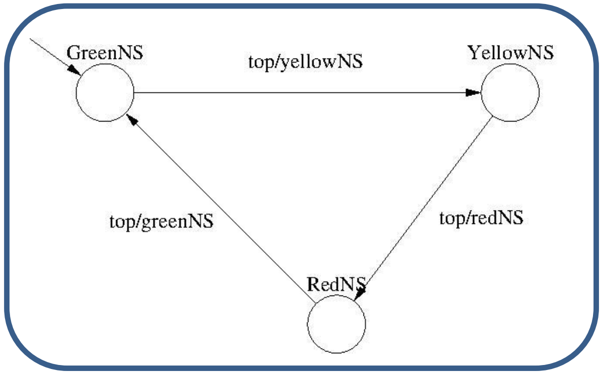

||



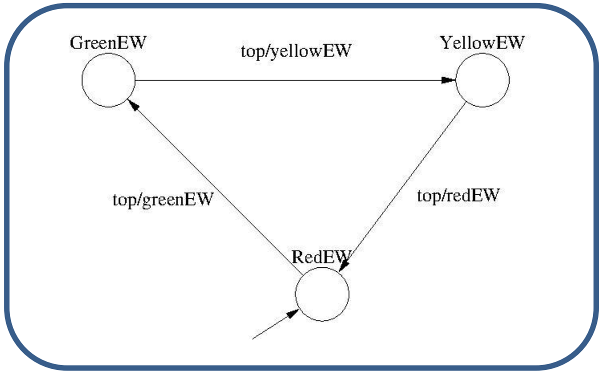

**NuSMV:**

**AG(!failure) ⟶ false**

# Use Case Issue in CLEM



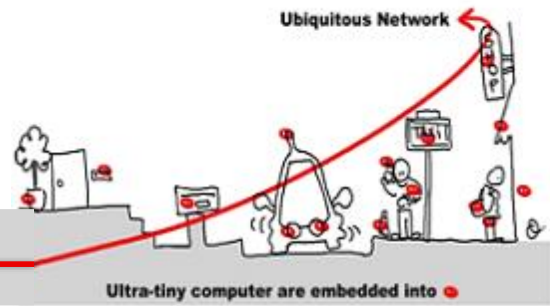

||




||

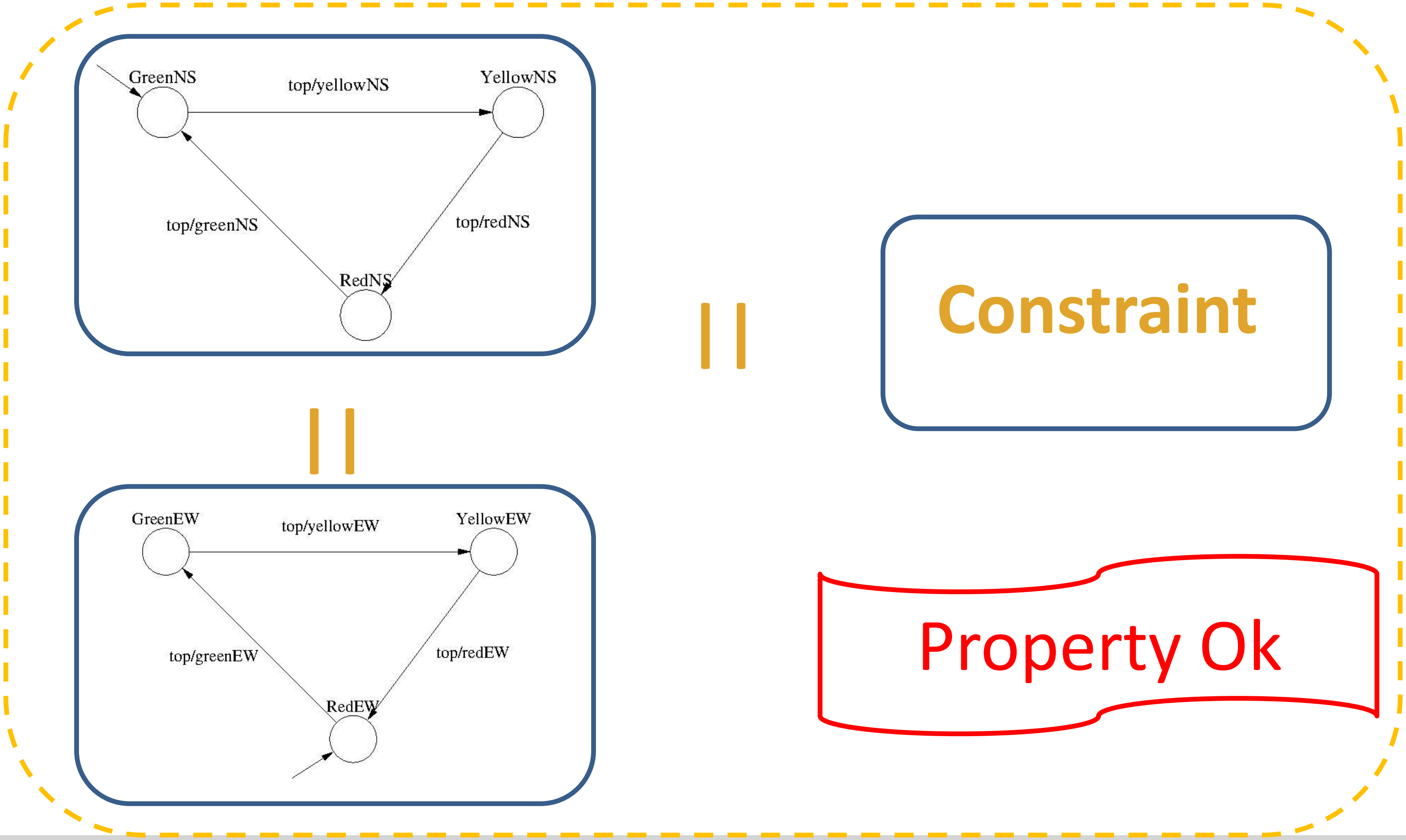
**Constraint**

# Constraint Expression in CLEM




```
module CrossRoadConstraint:
Input: greenNS, redNS, yellowNS, greenEW, redEW, yellowEW;
Output: greenNSC, redNSC, yellowNSC, greenEWC, redEWC,
yellowEWC;
local isNS, isEW
{
Mealy Machine
  isNS = greenNS or redNS or yellowNS;
  isEW = greenEW or redEW or yellowEW;
  greenNSC = greenNS and isEW;
  redNSC = redNS and isEW;
  yellowNSC = yellowNS and isEW;
  greenEWC = greenEW and isNS;
  redEWC = redEW and isNS;
  yellowEWC = yellowEW and isNS;
}
end
```

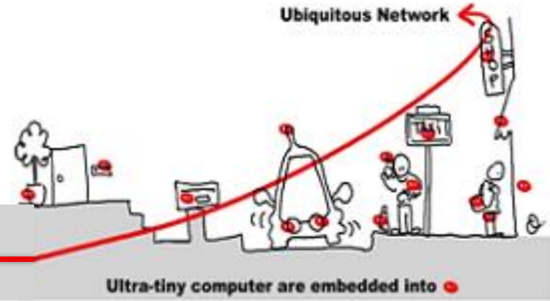
# Use Case Issue in CLEM



# LE Validated Component

```
module CrossRoad:
Input: top
Output: greenNSC, redNSC, yellowNSC, greenEWC, redEWC,
yellowEWC;
local greenNS, redNS, yellowNS, greenEW, redEW, yellowEW
{
run TrafficLightNS
||
run TrafficLightEW
||
run CrossRoadConstraint
}
end
```

# C Code Generation




**LE Validated CrossRoad** → 

## C Code Generation

**run automaton**

**reset automaton**

### CrossRoad.h:

extern void CrossRoad_reset_automaton
(int top, int*yellowNS, int*redNS, int*greenNS, int*yellowEW, int*greenEW, int*redEW);
extern void CrossRoad_automaton
(int top, int*yellowNS, int*redNS, int*greenNS, int*yellowEW, int*greenEW, int*redEW);

### CrossRoad.c:

Register definition as global variables; CrossRoad_reset_automaton; CrossRead_automaton.

# Creating a CEP using MQTT Approach