Critical Real Time Software Verification

A. Ressouche*

(*) Inria Sophia Antipolis-Méditerranée

Outline

1. Critical software design
2. Critical software validation techniques
3. Model checking
   1. Model specification
   2. Synchronous languages
   3. Scade
4. Model Checking Technique
   1. Property definition
   2. Validation technique

Critical Software

- Roughly speaking a critical system is a system whose failure could have serious consequences
- Nuclear technology
- Transportation
  - Automotive
  - Train
  - Avionics
- ....

Critical Software (2)

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

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

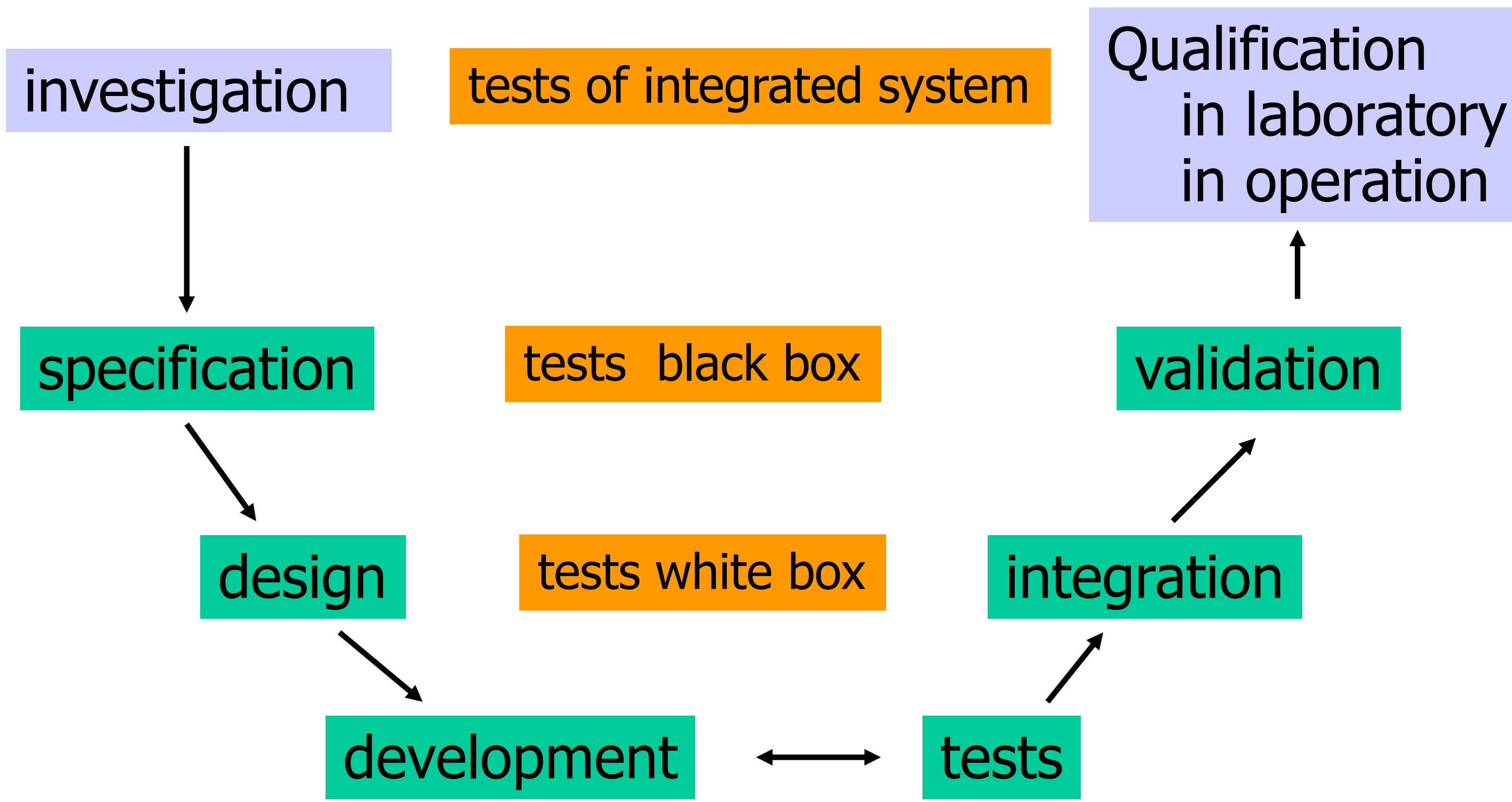
Depending of the level of risk of the system, different kinds of verification are required

Software Classification (avionics)

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 ?

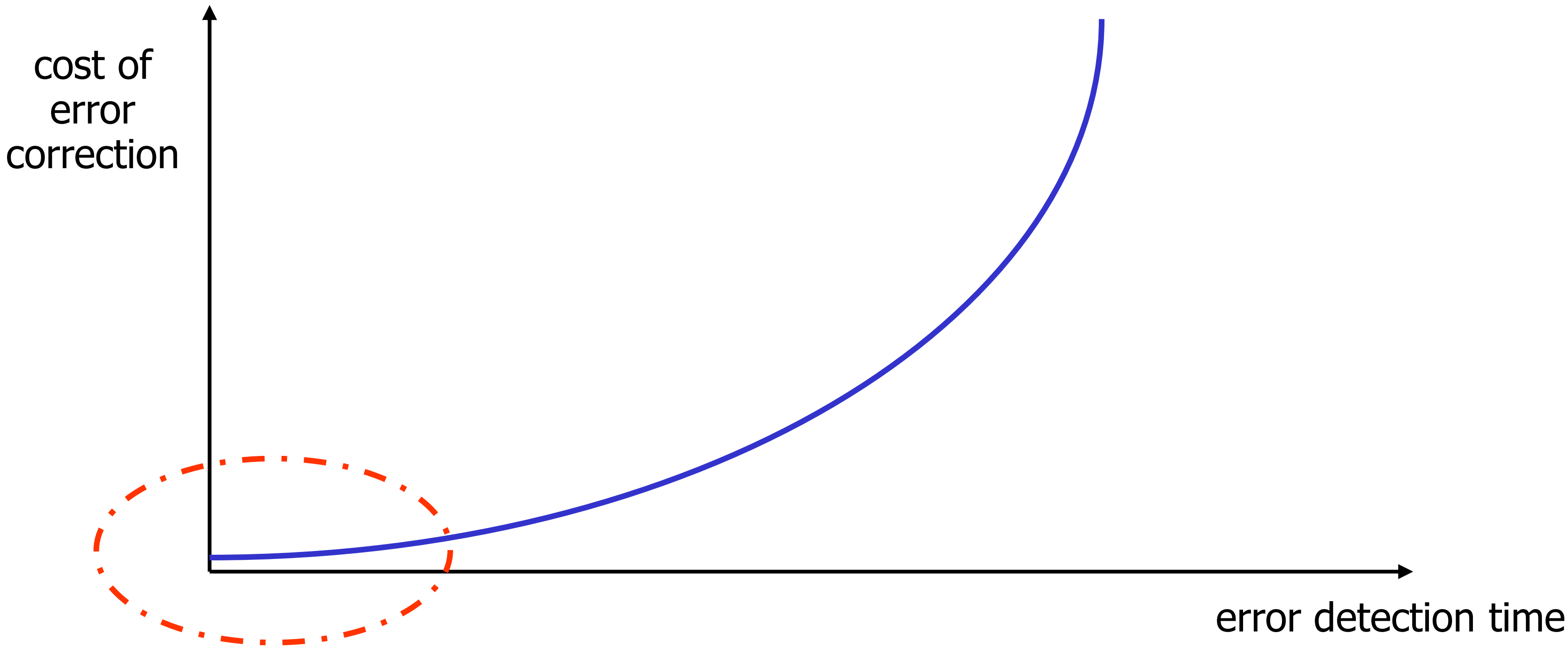
Classical Development V Cycle



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, more 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
    - $\Rightarrow$ specific domain engineers
  - Allowing application development
    - Coherency
    - Completeness
  - Allowing application functional validation
    - Express properties to be validated

$\Rightarrow$ Formal models usage

Critical software specification

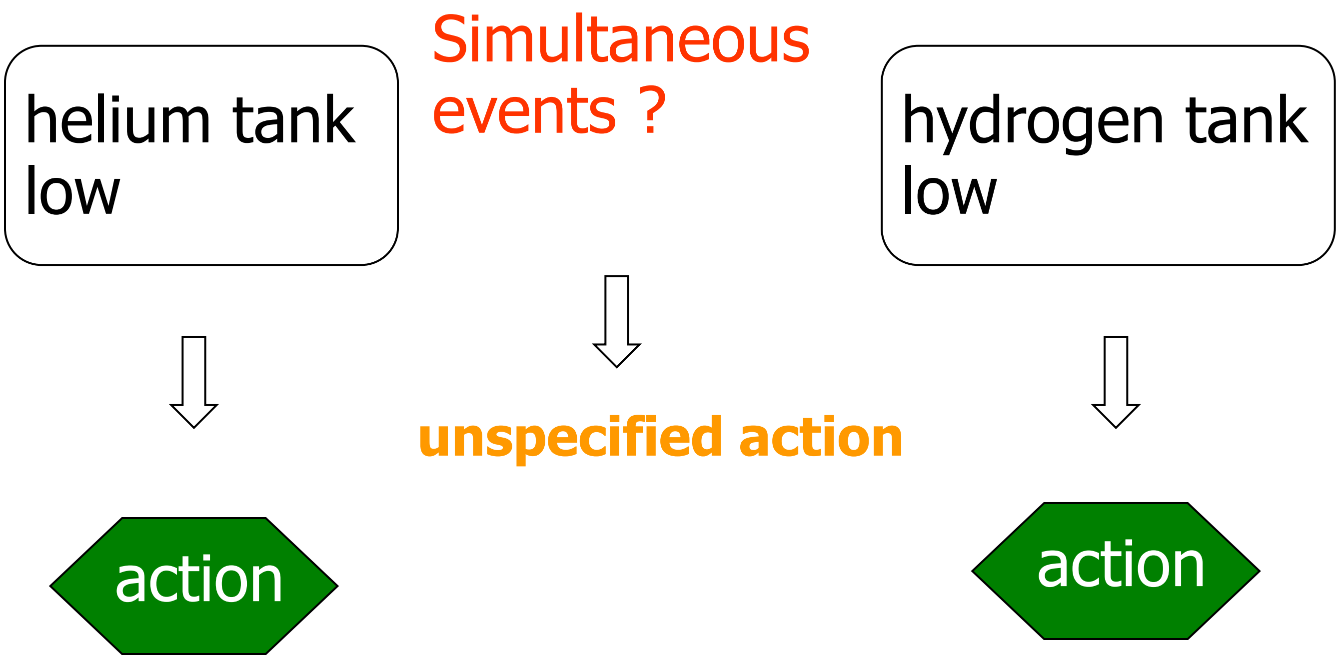
- First Goal: must yield a formal description of the application needs:
  - Standard to allowing communication between computer science engineers and non computer science ones
  - General enough to allow different kinds of application:
    - Synchronous (and/or)
    - Asynchronous (and/or)
    - Algorithmic

Critical software specification

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

Example of non completeness

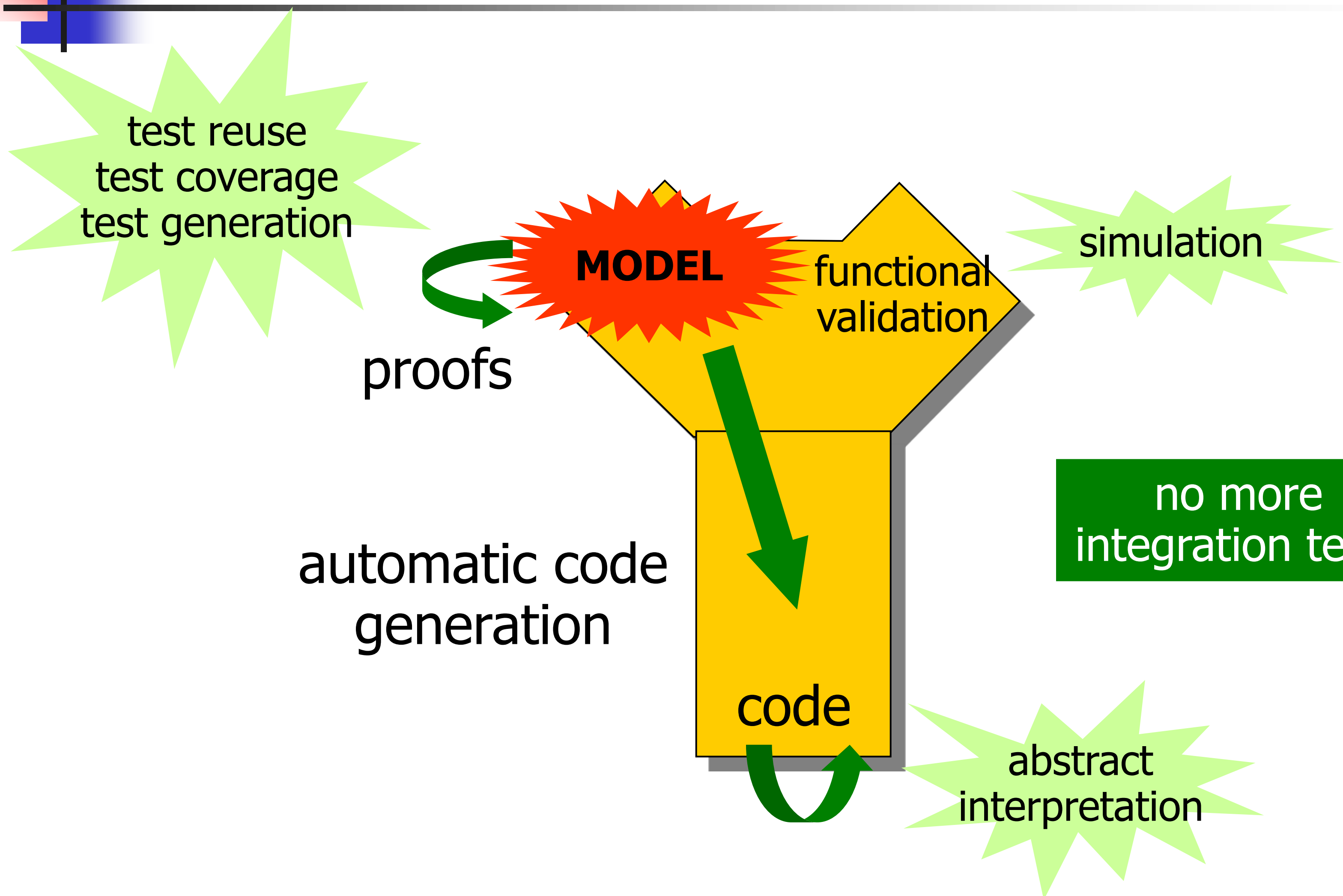
From Ariane 5:



Critical Software Specification (3)

- 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

Relying on Formal Methods



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

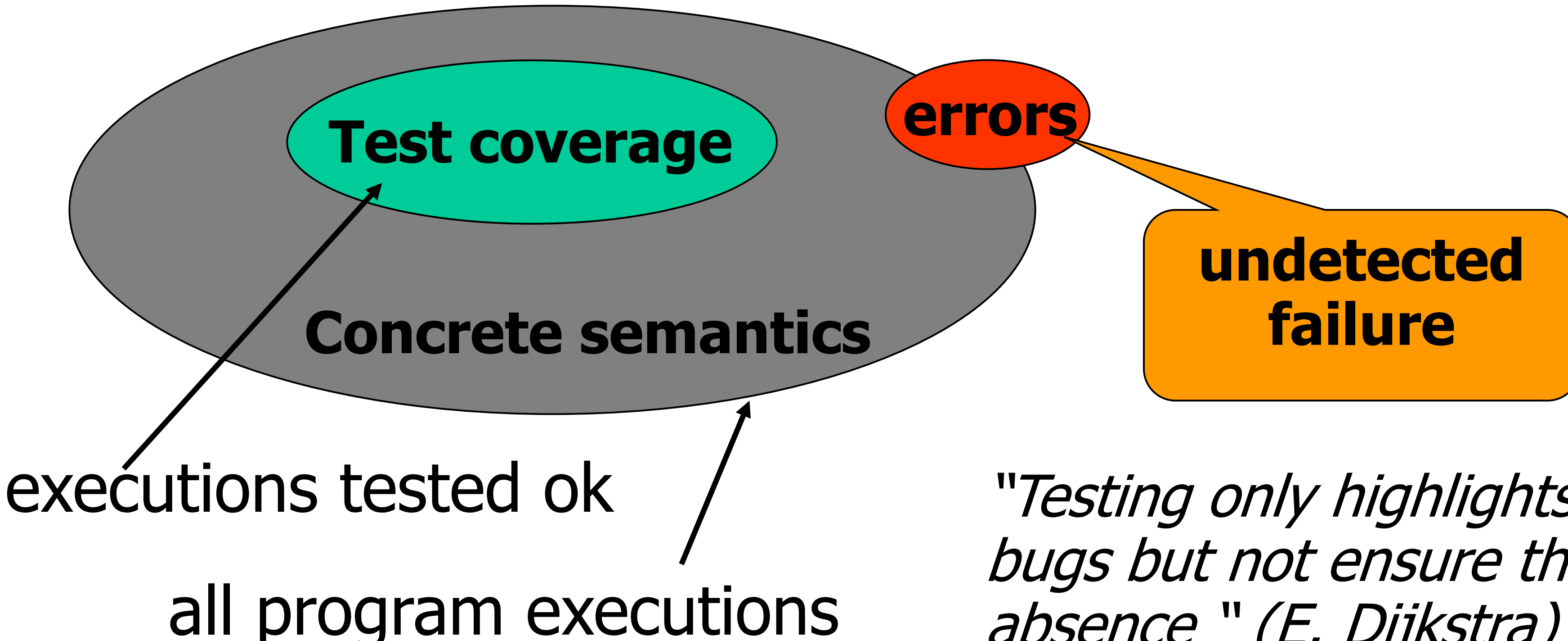
- Testing
  - Run the program on set of inputs and check the results
- Static Analysis
  - Examine the source code to increase confidence that it works as intended
- Formal Verification
  - Argue formally that the application always works as intended

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.

Testing (2)

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: }

$x_1 = [1,1]$

$x_2 = x_1 \cup x_3 \cap [-\infty, 999]$

$x_3 = x_2 \oplus [1,1]$

$x_4 = x_1 \cup x_3 \cap [1000, \infty]$

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

Formal verification

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

Safety and Liveness Properties

- Example: the beacon counter in a train:
  - Count the difference between beacons 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 ontime 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

It refers to unbound future

Safety and Liveness Properties Checking

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

Model Checking Techniques

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

Model Checking Model Specification

- Model = automata which is the set of program behaviors

Model Specification

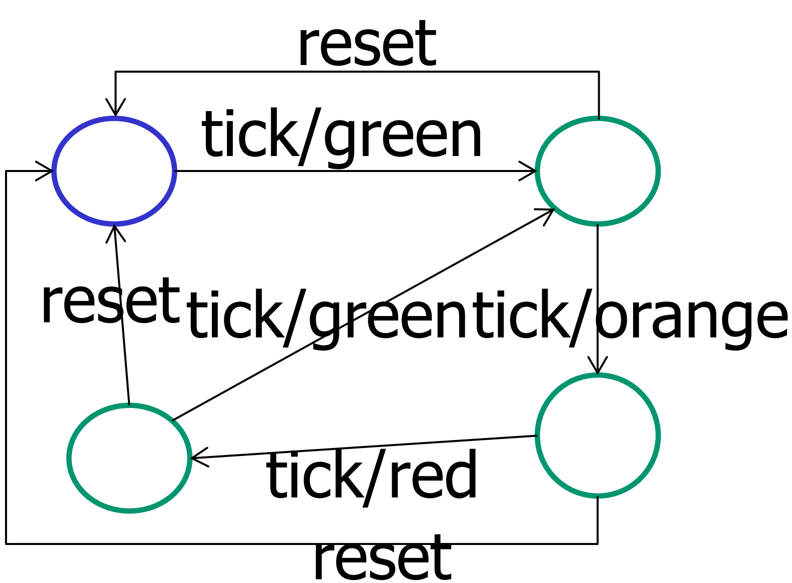
- Model = automata which is the set of program behaviors
- An automata is composed of:
  1. A finite set of states ($\mathbb{Q}$)
  2. A finite alphabet of actions ($\mathbb{A}$)
  3. An initial state ($q^{init} \in \mathbb{Q}$)
  4. A transition relation ($\mathbb{R}$ in $\mathbb{Q} \times \mathbb{Q}$)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

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

Model Specification

- Model = automata which is the set of program behaviors

Example: Traffic Light



trigger: tick, reset

action:green,orange,red

Model Specification

Model Specification

How design automata as system behaviors ?

Use synchronous languages to specify critical systems.

Synchronous programs = automata

Model Specification with Synchronous Languages

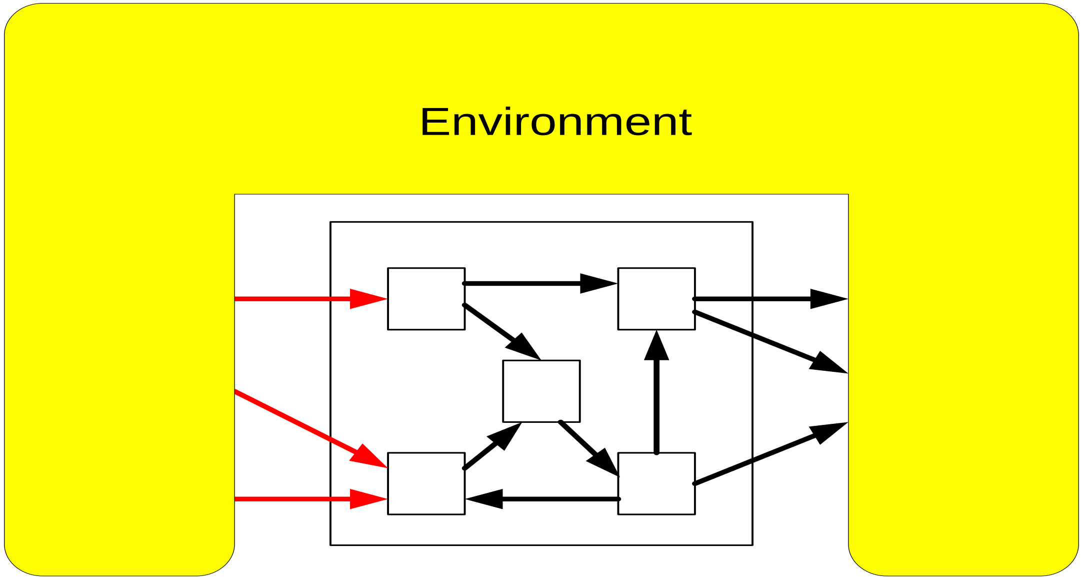
1. Synchronous languages have a simple formal model (a finite automaton) making formal reasoning tractable.
2. Synchronous languages support concurrency and offer an implicit or explicit means to express parallelism.

3. Synchronous languages are devoted to design reactive real-time systems.

Determinism & Reactivity

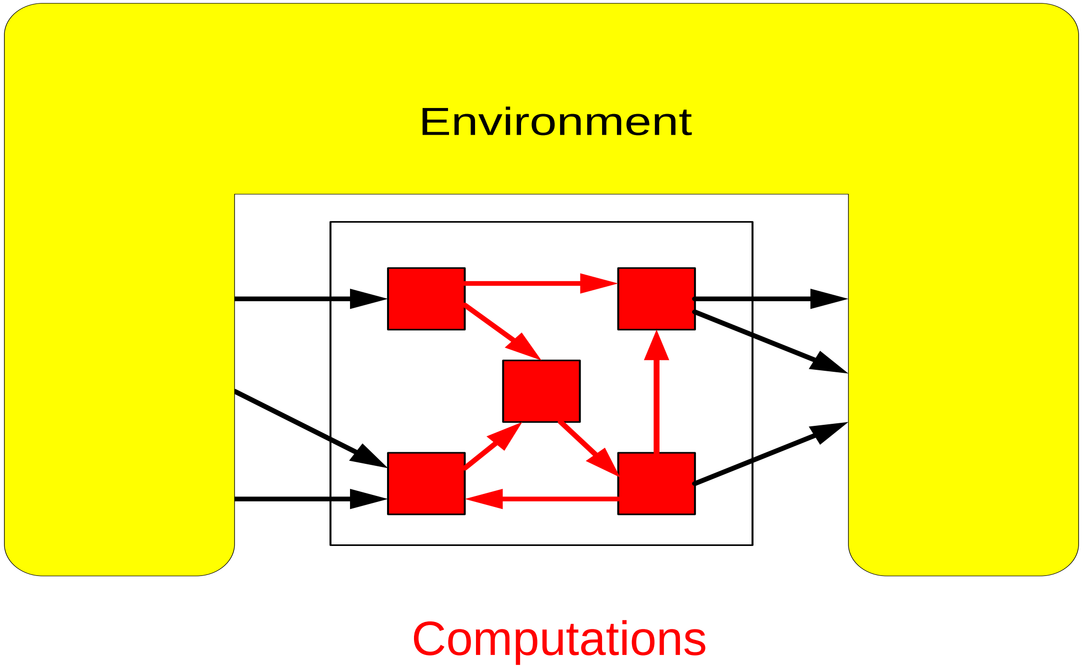
- Synchronous languages are deterministic and reactive
- Determinism:
  - The same input sequence always yields the same output sequence
- Reactivity:
  - The program must react(*) to any stimulus
  - Implies absence of deadlock
    - (*) *Does not necessary generate outputs, the reaction may change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

Synchronous Reactive Systems (1)

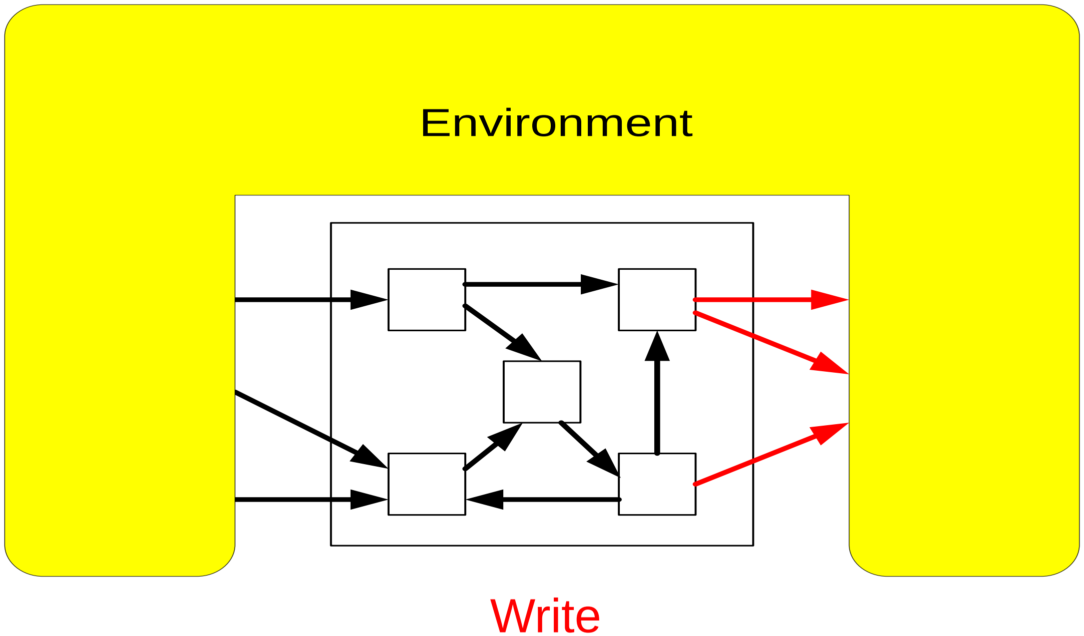


Read

Synchronous Reactive Systems (2)



Synchronous Reactive Systems (3)



Atomic execution: read, compute, write

Synchronous Hypothesis

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

Use N as time base

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

Synchronous Hypothesis

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

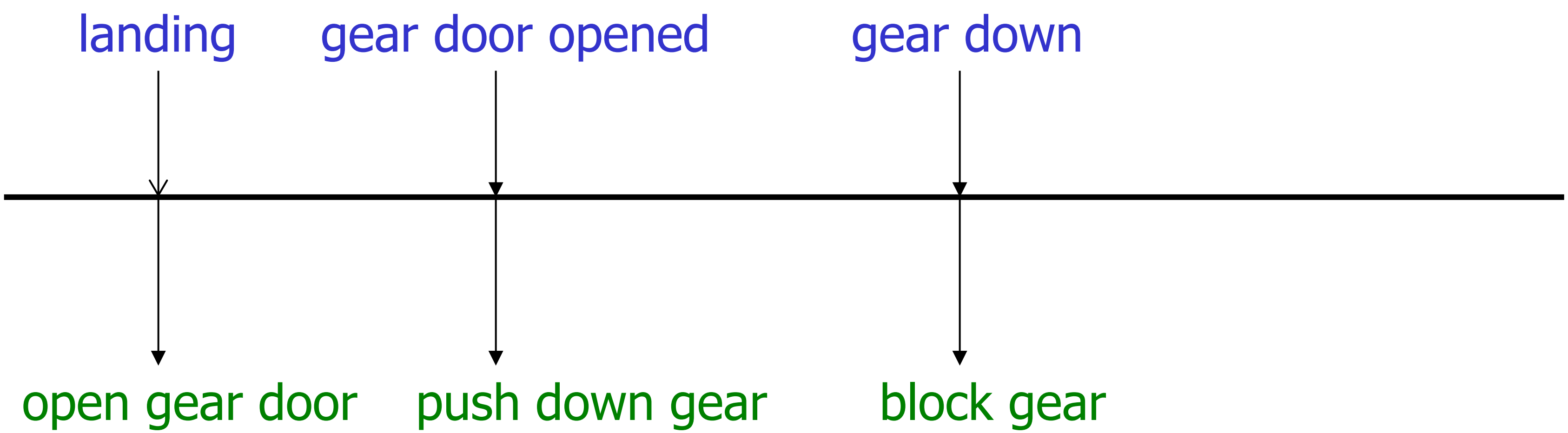
Reactive ?

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

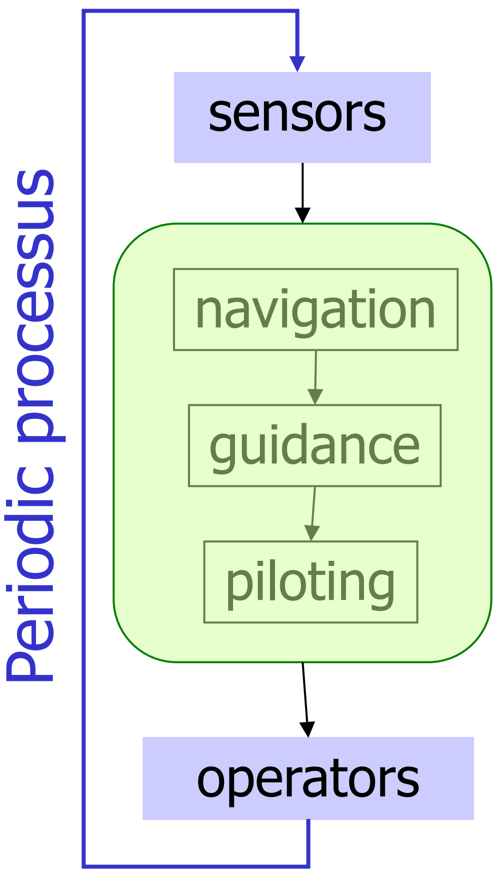
Some systems have components of both kinds

Event Driven Reactive System

Langing gear management



Data Flow Reactive System (Example)



Control/Command vehicle

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

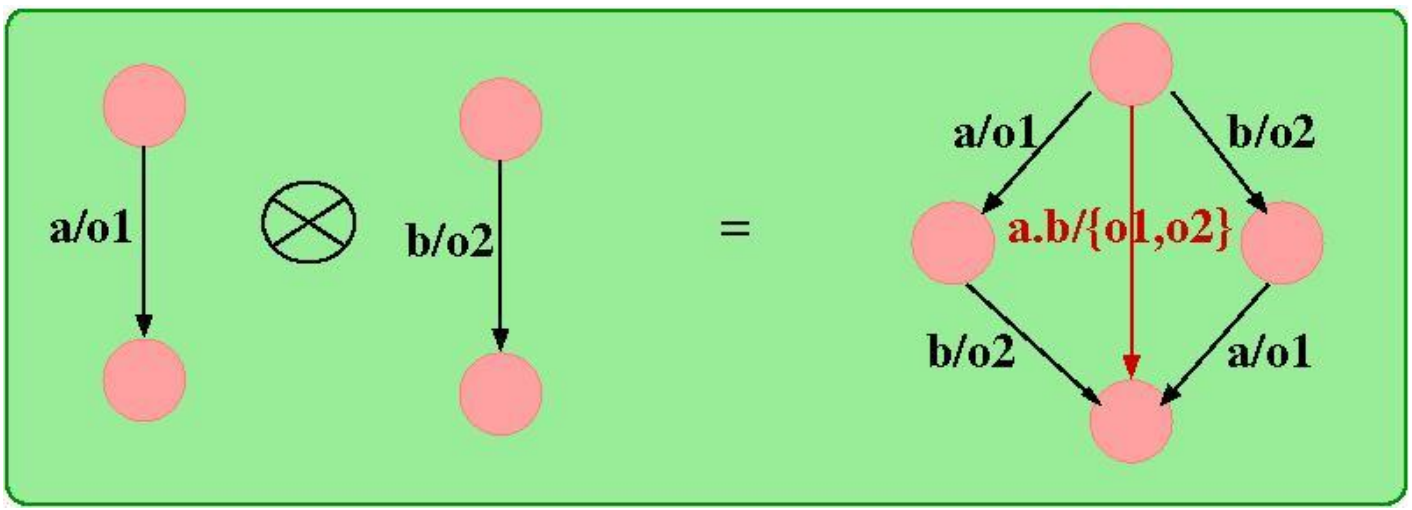
Imperative and Declarative languages

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

Event Driven = FSM

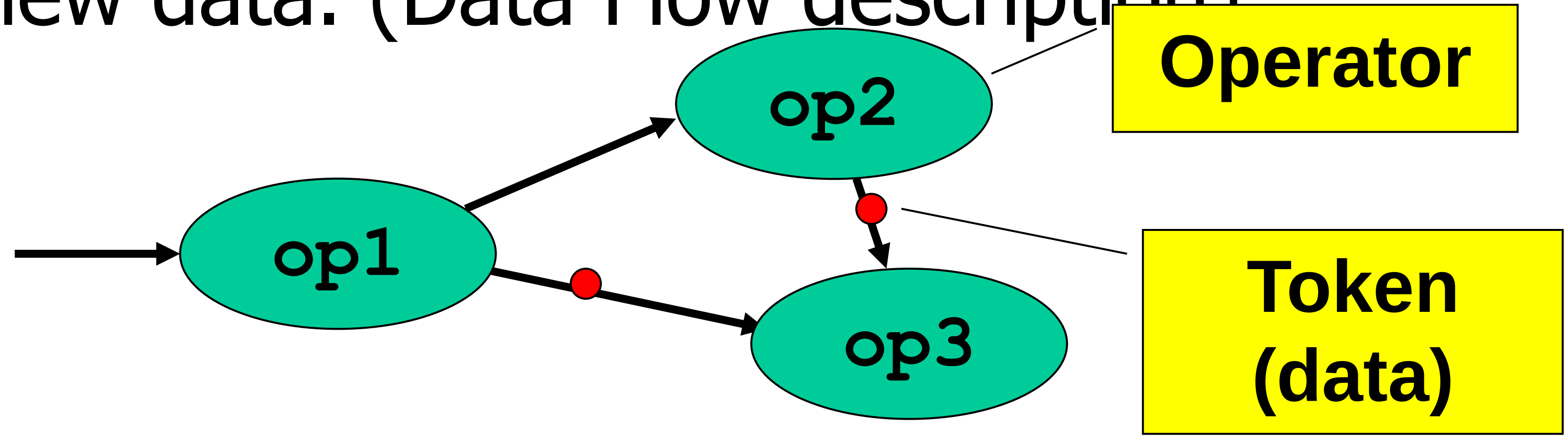
Event driven applications can be designed:

1. As simple finite sate machines (= automata)
2. As the synchronous product of finite state machines

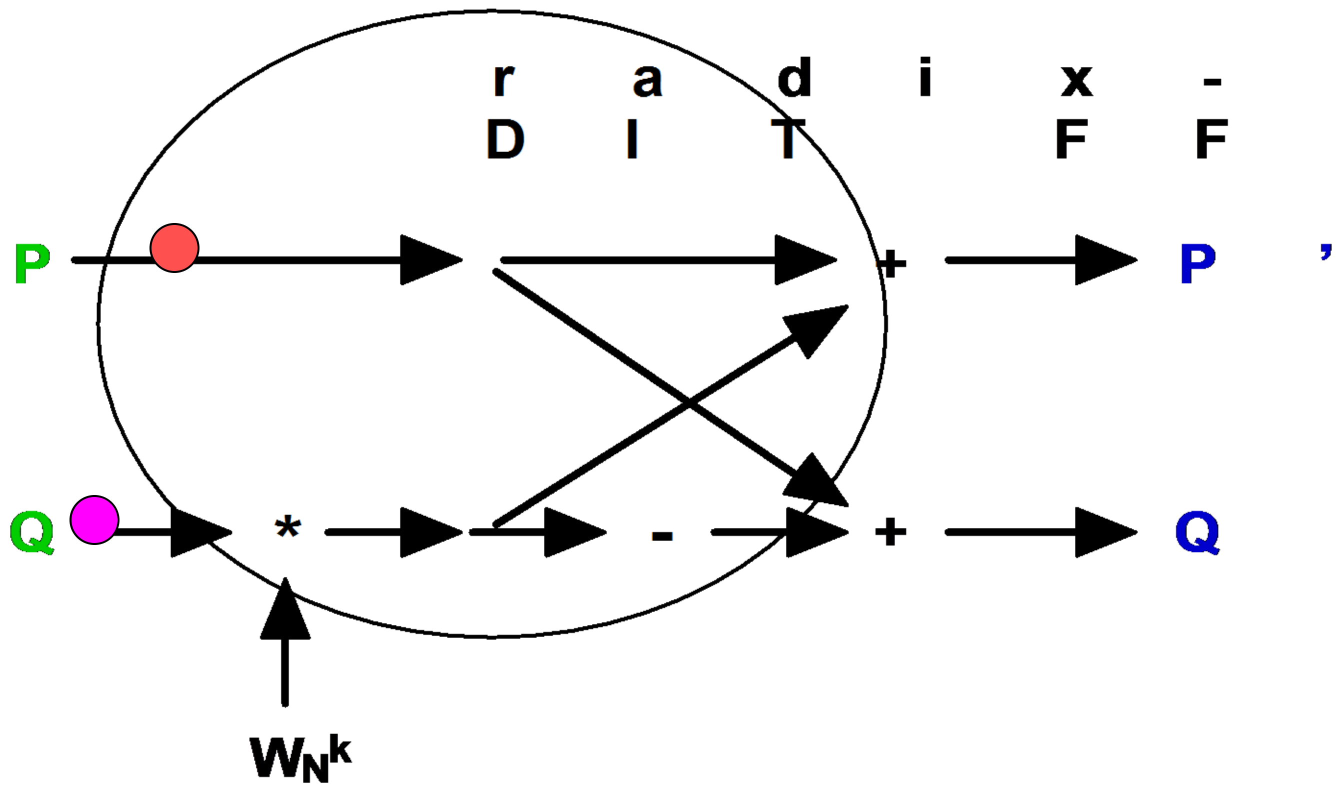


Data Flow = Operator Networks

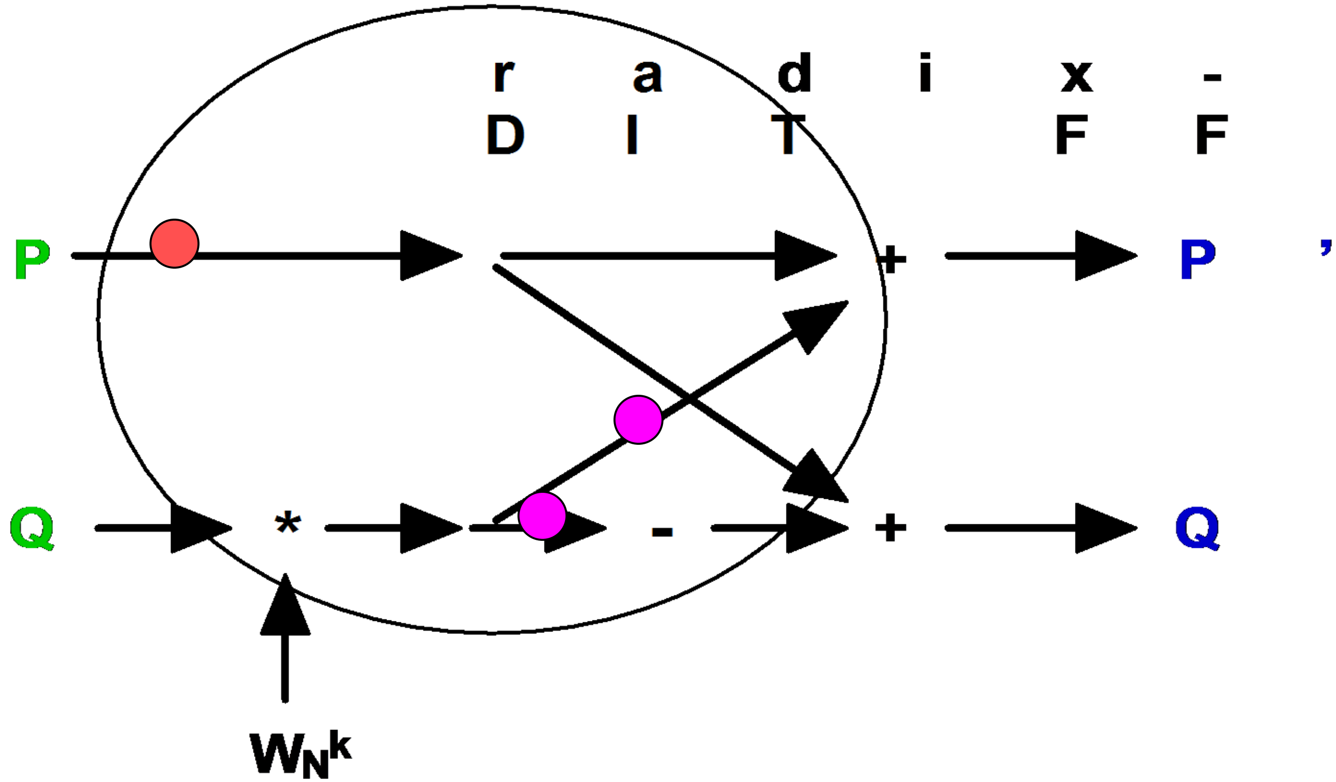
- LUSTRE programs can be interpreted as networks of operators.
- Data « flow » to operators where they are consumed. Then, the operators generate new data. (Data Flow description)



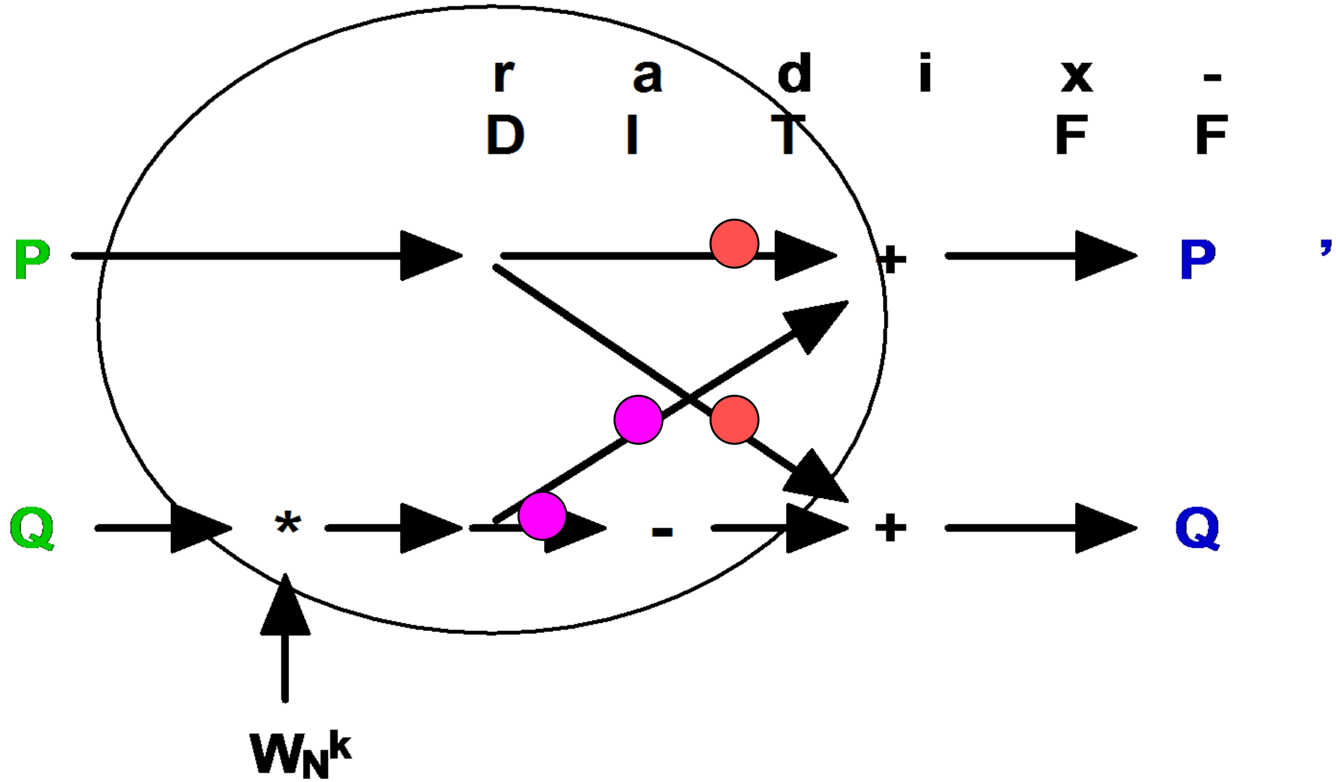
An example of Data Flow



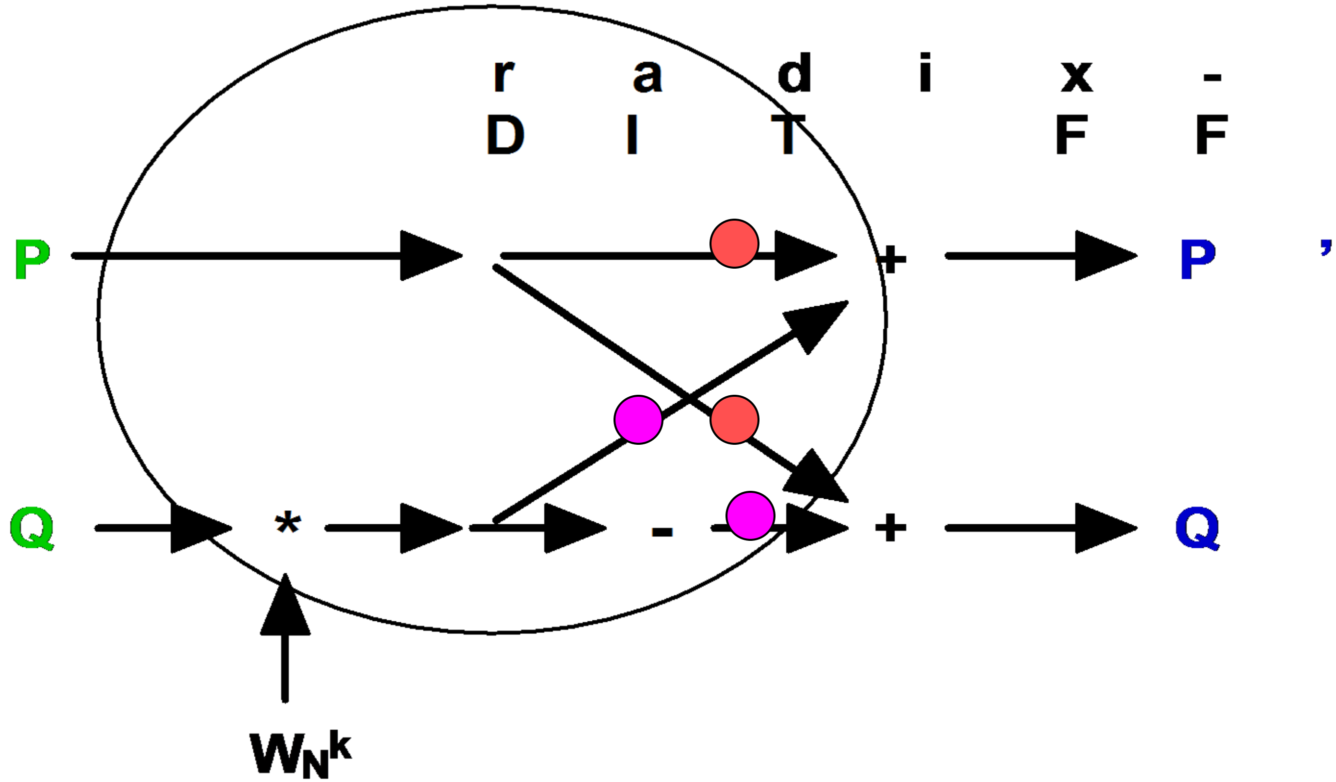
Data Flow



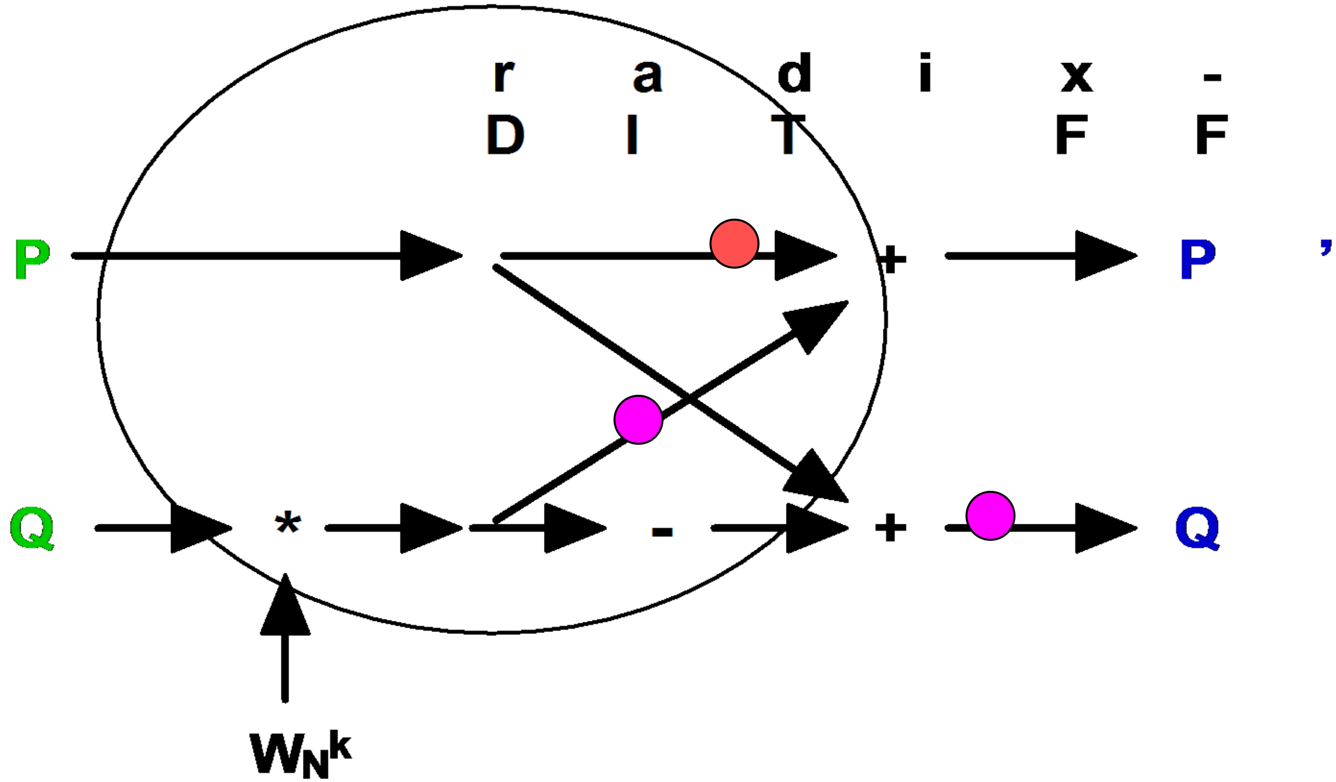
Data Flow



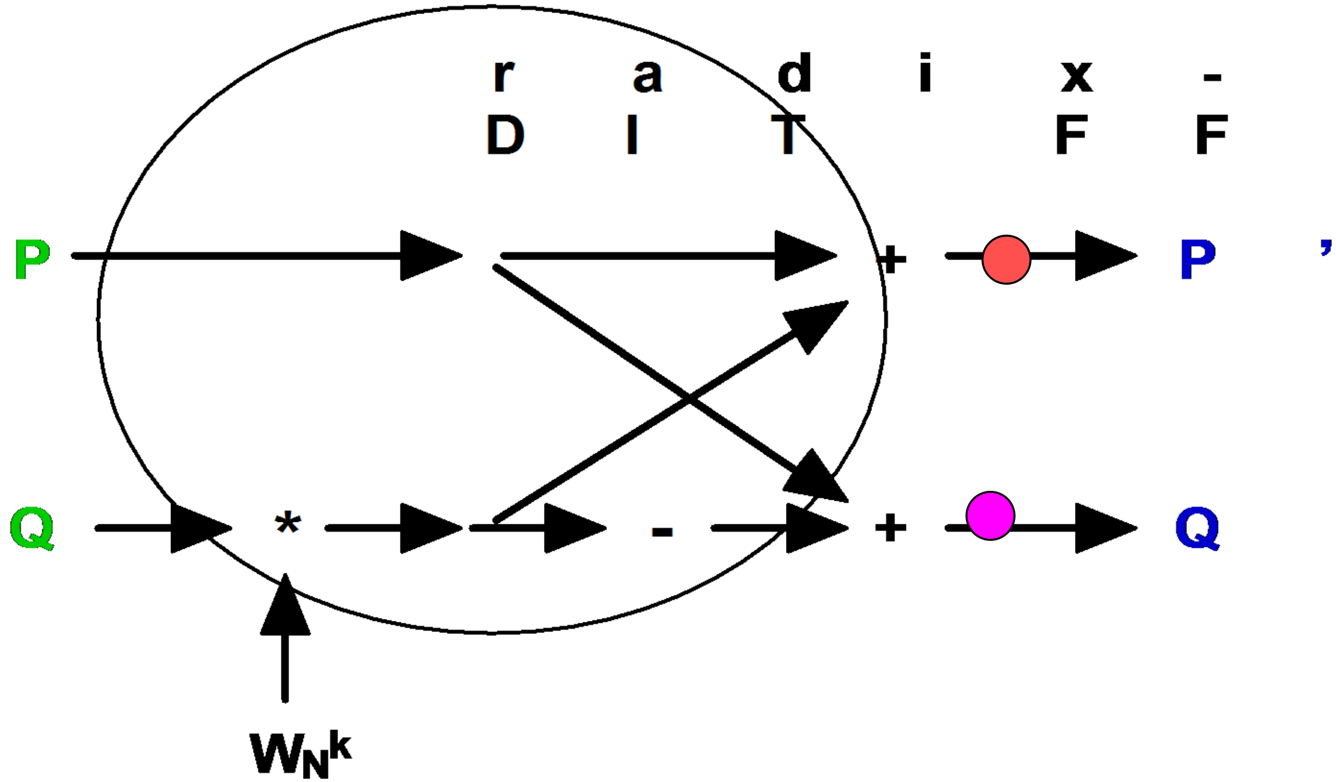
Data Flow



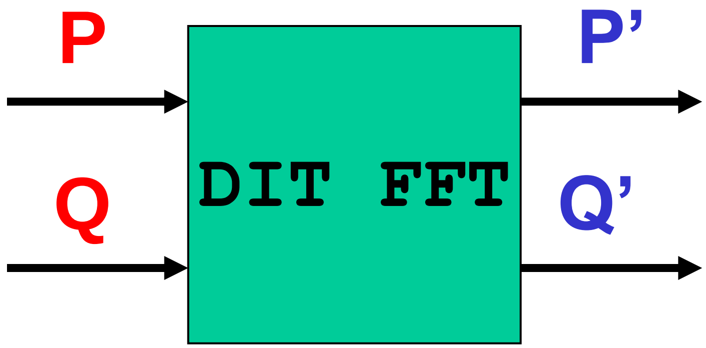
Data Flow



Data Flow



Functional Point of View



$$P' = P + W_N^k * Q$$

$$Q' = P - W_N^k * Q$$

Flows, Clocks

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

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

Data Flow Synchronous Languages

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

Data Flow Synchronous Languages

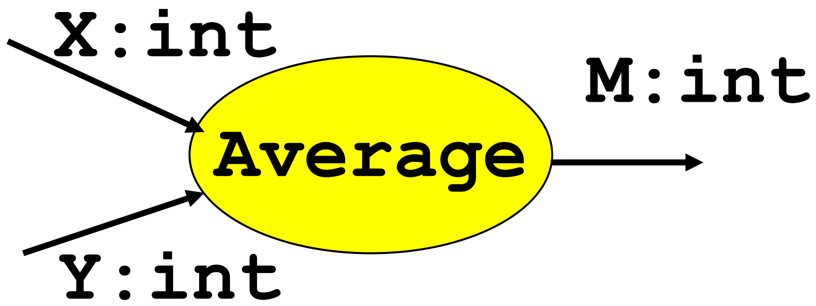
Substitution principle:

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

Definition principle:

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

Data Flow Synchronous Languages



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

$M = (X + Y)/2$

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

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

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

Expressions

Constants

`0, 1, ..., true, false, ..., 1.52, ...`

real

int

bool

\+
Imported types and operators

$$C:\alpha \equiv \forall k \in N, C_k = C$$

« Combinational » Operators

Data operators

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

« Point-wise » operators

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

« Combinational » Operator IF

- if operator

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

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

« Combinational » Operator IF

- if operator

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

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

Memorizing

Take the past into account!

pre (previous):

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

Undefined value denoting uninitialized memory: nil

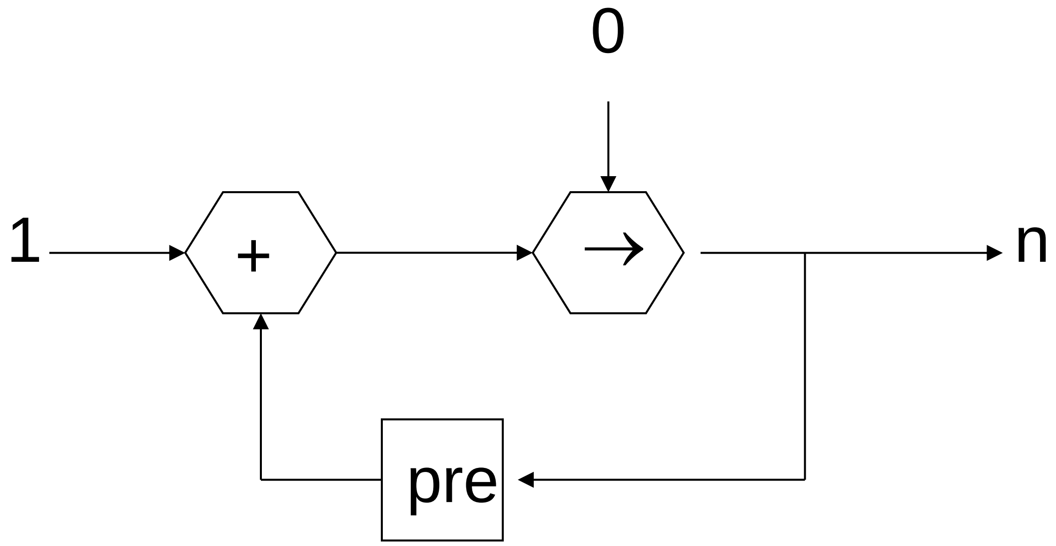
-> (initialize): sometimes call "followed by"

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

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

« Sequential » Examples

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



Sequential » Examples

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;

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)

Recursive definitions

Temporal recursion

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

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

Instantaneous recursion

e.g.: X = 1.0 / (2.0 – X)

Forbidden in Lustre, even if a solution exists!

Be carefull with cross-recursion.

Clocks

Basic clock

Discrete time induced by the input sequence

Derived clocks (slower)

`when` (filter operator):

`E when C` is the sub-sequence of **E** obtained by keeping only the values of indexes e_k for which c_k**`=true`**

Examples of clocks

Basic cycles	1	2	3	4	5	6	7	8
C1	true	false	true	true	false	true	false	true
Cycles of C1	1		2	3		4		5
C2	false		true	false		true		true
Cycles of C2			1			2		3

Example of sampling

nat,odd:int

halfBaseClock:bool

nat = 0 -> pre nat +1;

halfBaseClock =

true -> not pre halfBaseClock;

odd = nat **when** **halfBaseClock**;

`nat` is a flow on the basic clock;

`odd` is a flow on `halfBaseClock`

Exercice: write even

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);
```

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

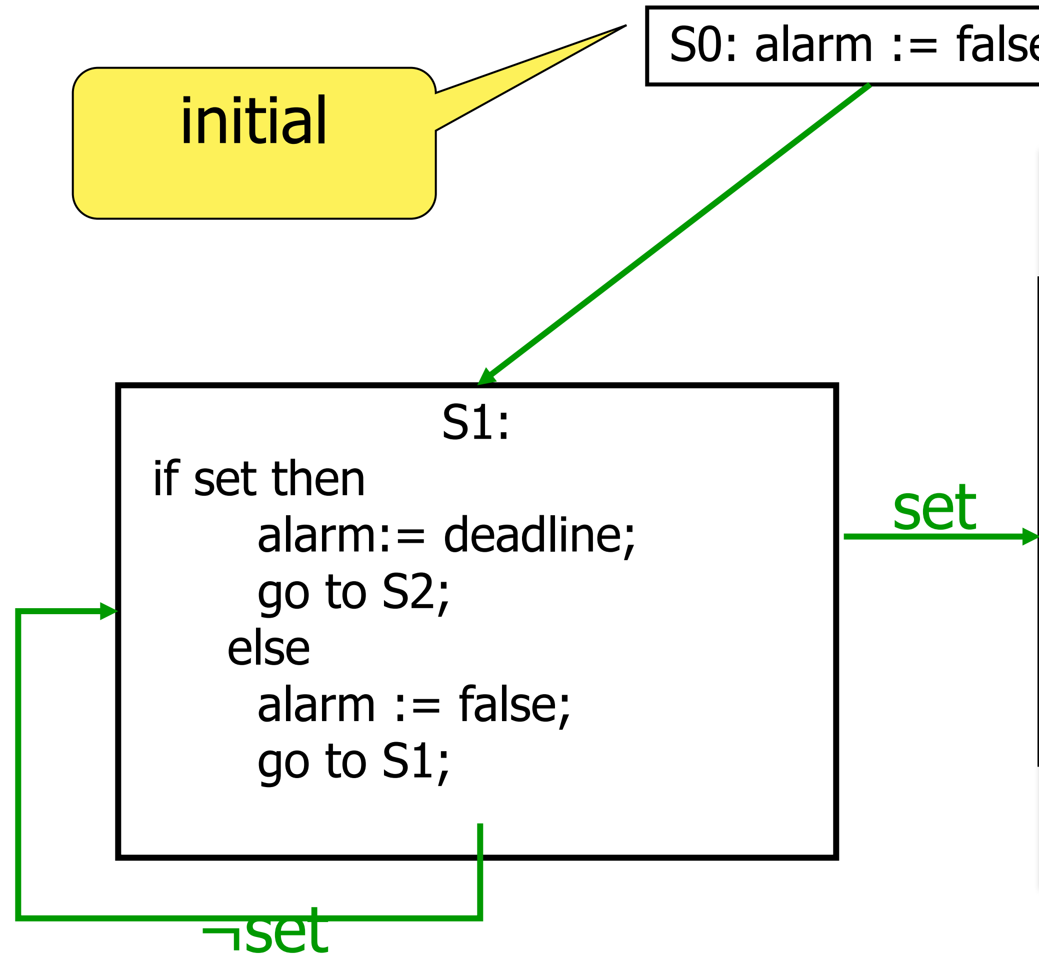
S0: alarm := false;

S1:

_init := false
pre_is_set= false

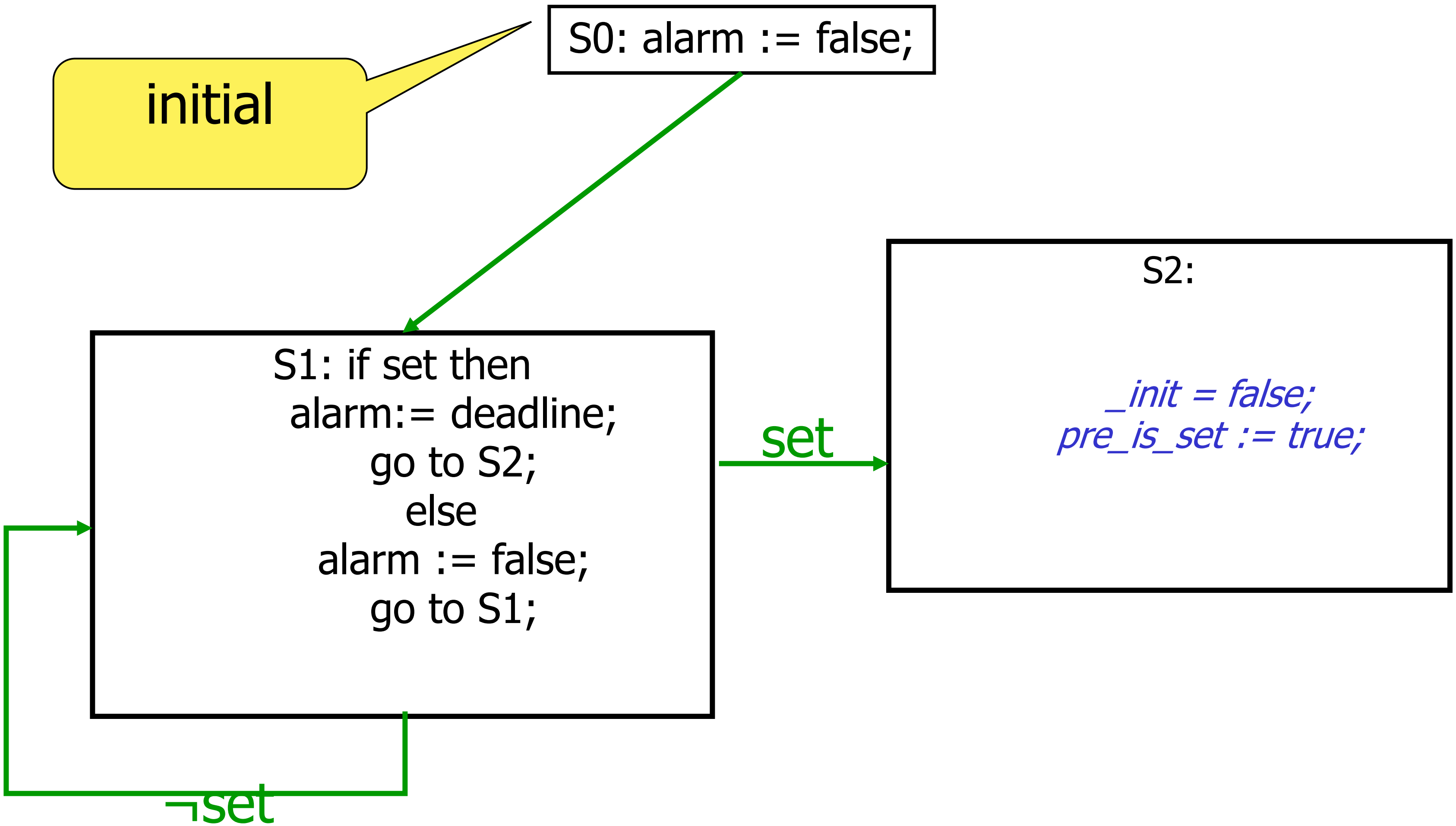
```
if _init then // first instant (or reaction)
   is_set := false; alarm := false
_init := false;
else   // following reactions
   if set then is_set := true
   else
      if reset then is_set := false;
      endif
   endif
   alarm := is_set and deadline;
endif
```

Data Flow Program Compilation

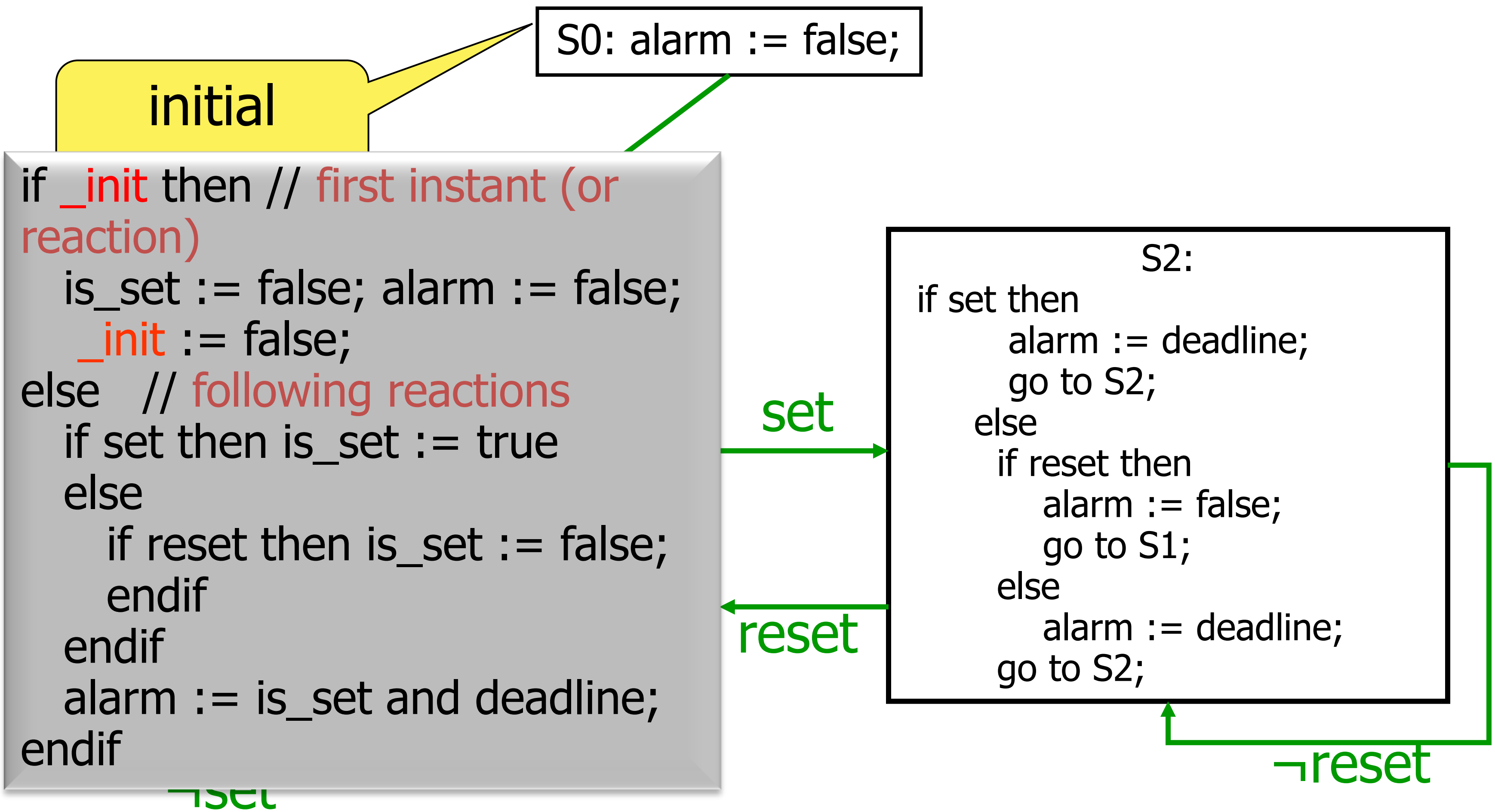



```
if _init then // first instant (or reaction)
   is_set := false; alarm := false;
    _init := false;
else   // following reactions
   if set then is_set := true
   else
      if reset then is_set := false;
      endif
   endif
   alarm := is_set and deadline;
endif
```

Data Flow Program Compilation

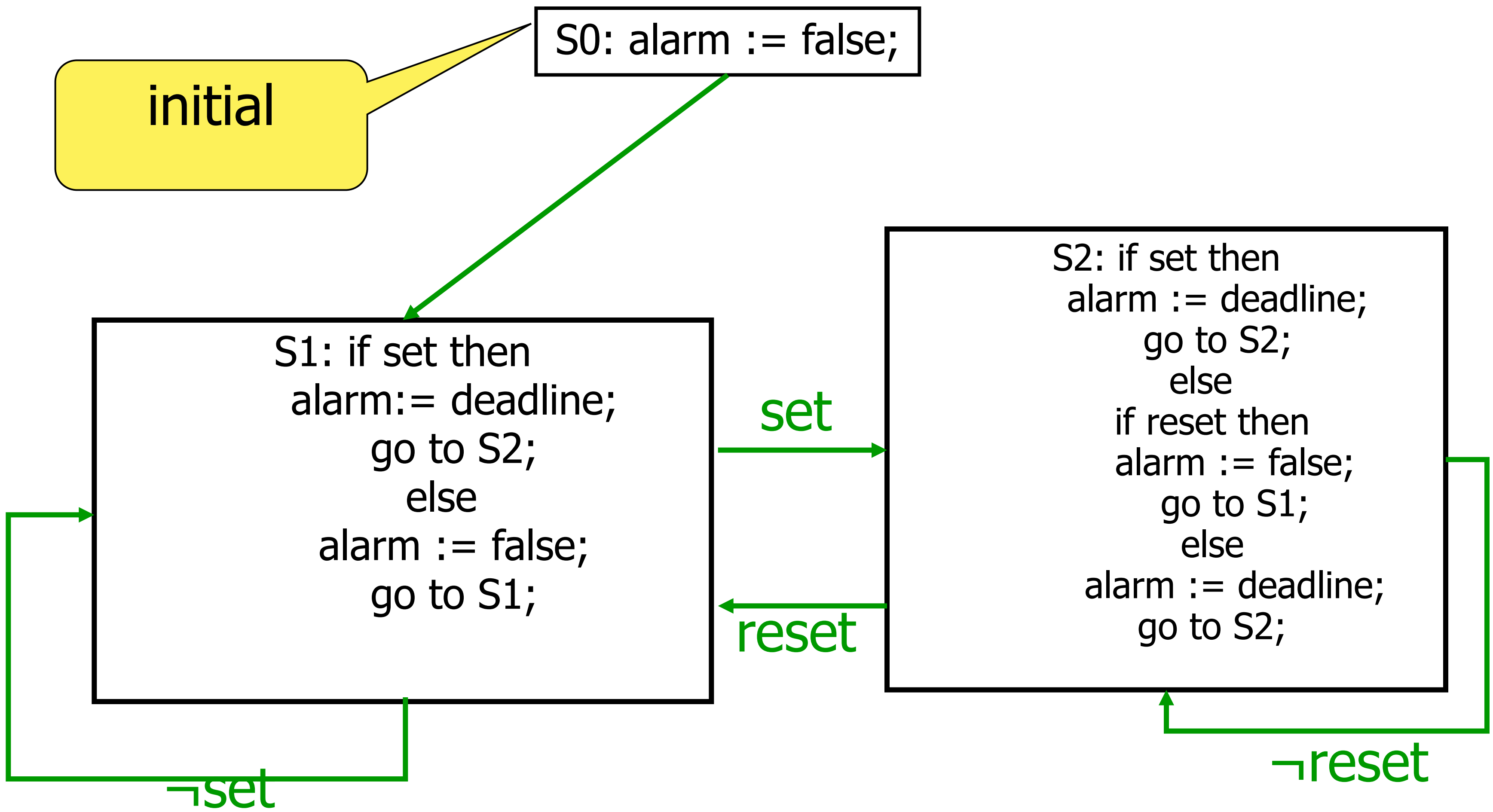


Lustre Program Compilation



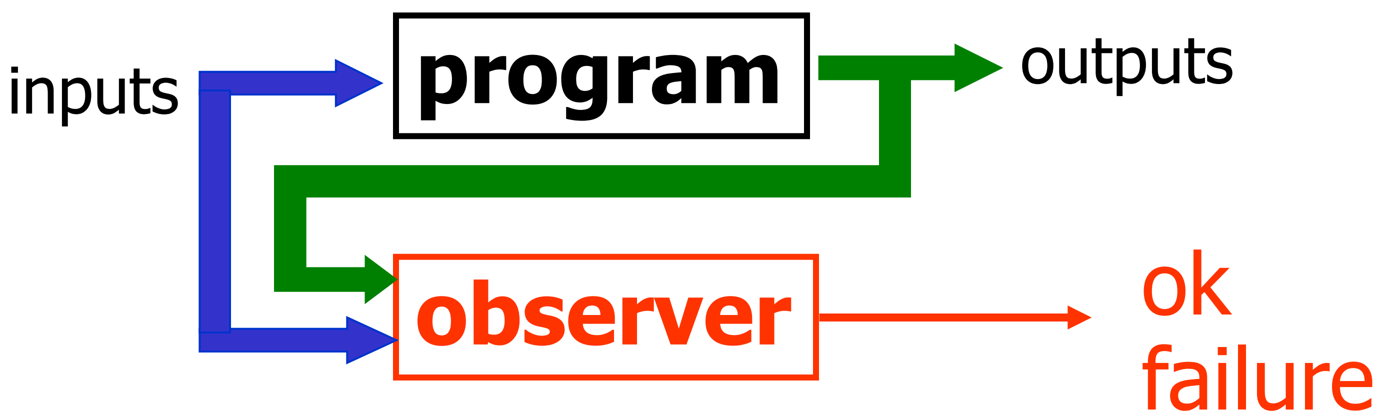

```
if _init then // first instant (or reaction)
   is_set := false; alarm := false;
    _init := false;
else   // following reactions
   if set then is_set := true
   else
      if reset then is_set := false;
      endif
   endif
   alarm := is_set and deadline;
endif
```

Lustre Program Compilation



Model Checking of Data Flow programs 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

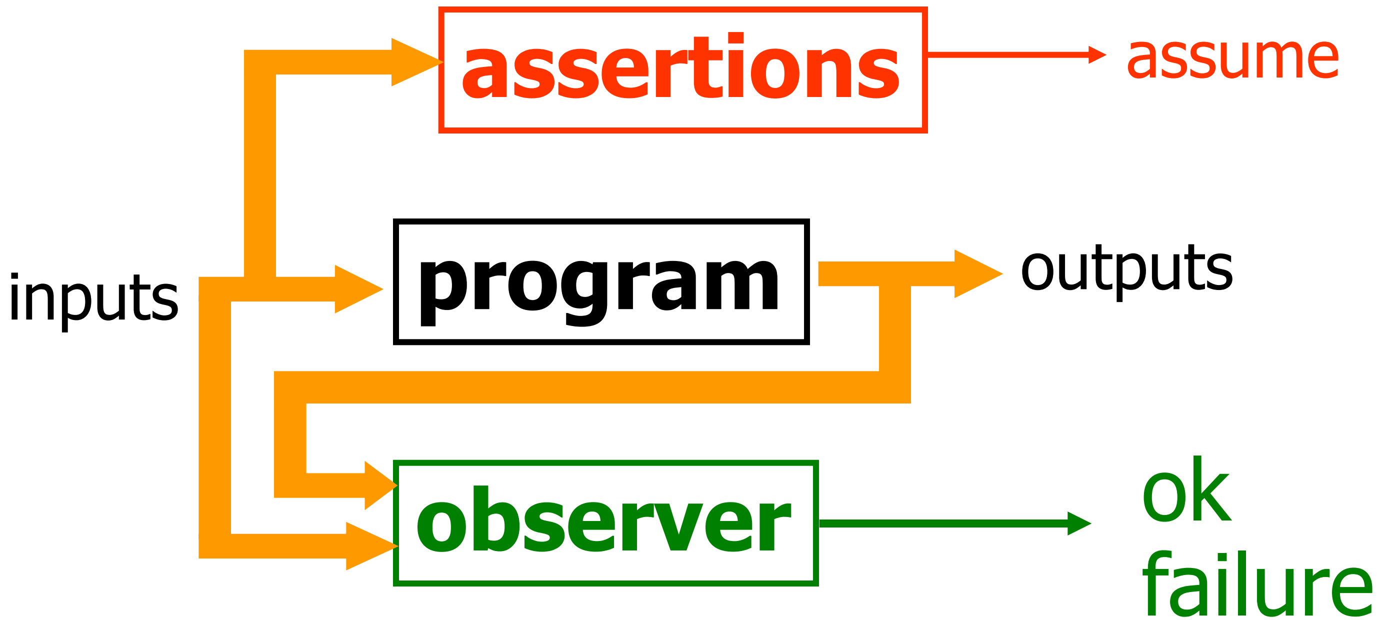


Properties Validation

- 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

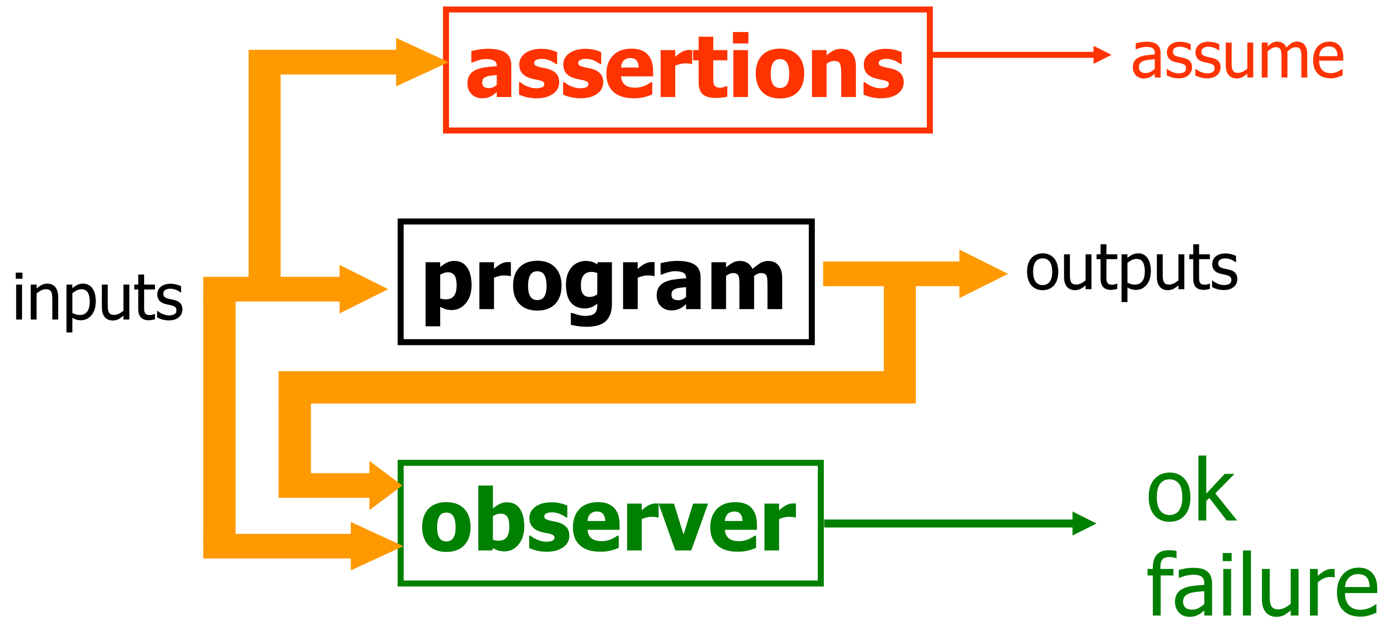
Properties Validation (2)

- 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).



Safety and Liveness Properties

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

operator **train** (sec, bea : bool) returns (ontime, early, late: bool)

```
diff = (0 ->pre diff) + (if bea then 1 else 0) + (if sec then -1 else 0);
        early = (true -> pre ontime) and (diff > 3) or
                (false -> pre early) and (diff > 1);
        late  = (true -> pre ontime) and (diff < -3)  or
                (false -> pre late) and (diff < -1);
        ontime = not (early or late);
```

Train Safety Properties

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

Train Assumptions

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

SCADE: Safety-Critical Application Development Environment

- Scade has been developped 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 Aviaation F7X: Flight Controls, landing gear, braking
    - Boeing 787: Landing gear, nose wheel steering, braking

SCADE

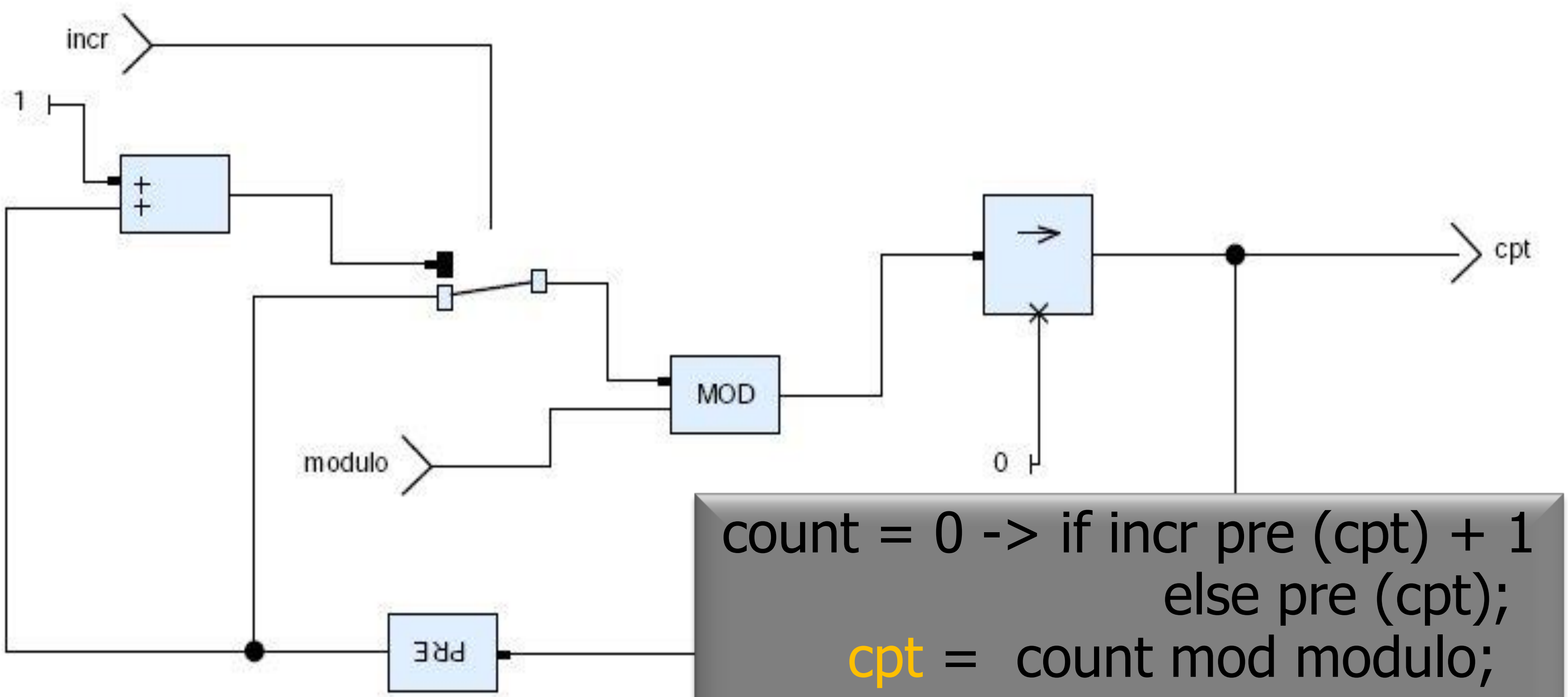
- System Design
  - Both data flows and state machines
- Simulation
  - Graphical simulation, automatic GUI integration
- Verification
  - Apply observer technique
- Code Generation
  - certified C code

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

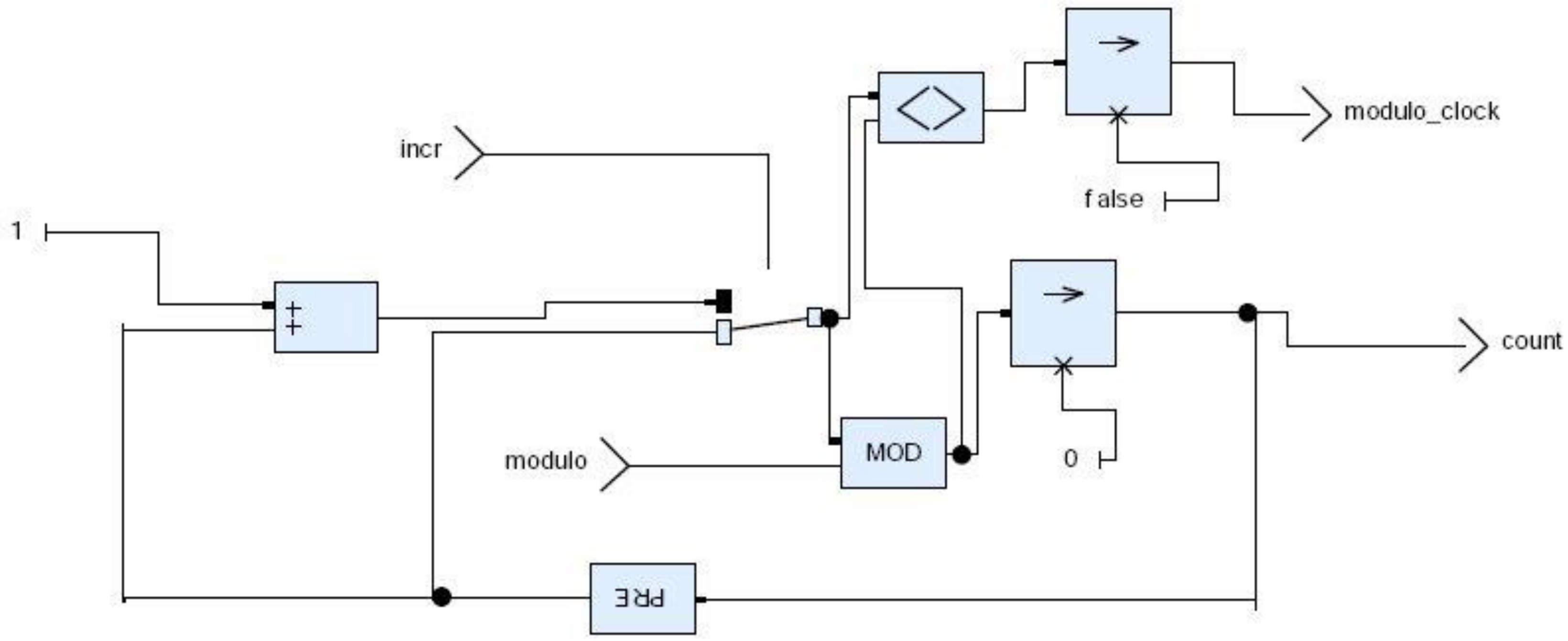



```
count = 0 -> if incr pre (cpt) + 1
                    else pre (cpt);
      cpt =  count mod modulo;
```

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

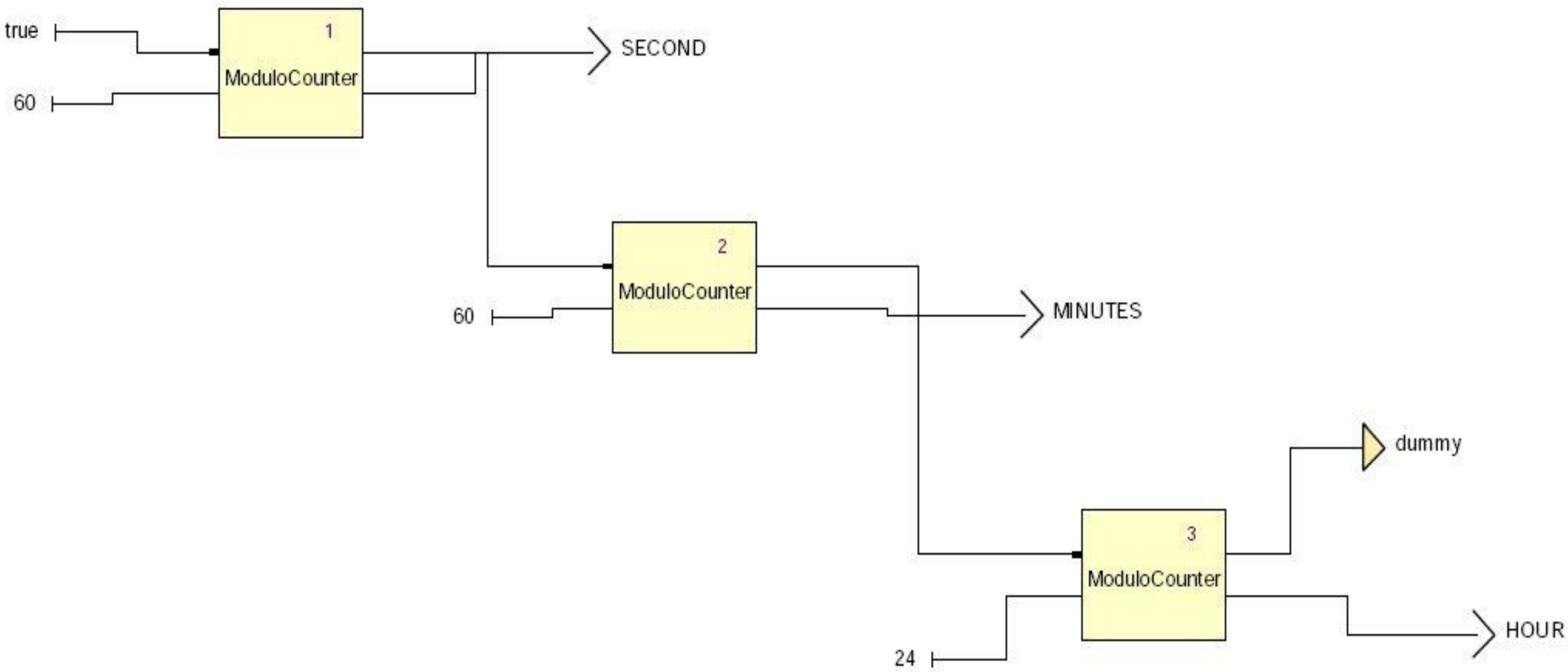


Timer

```
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



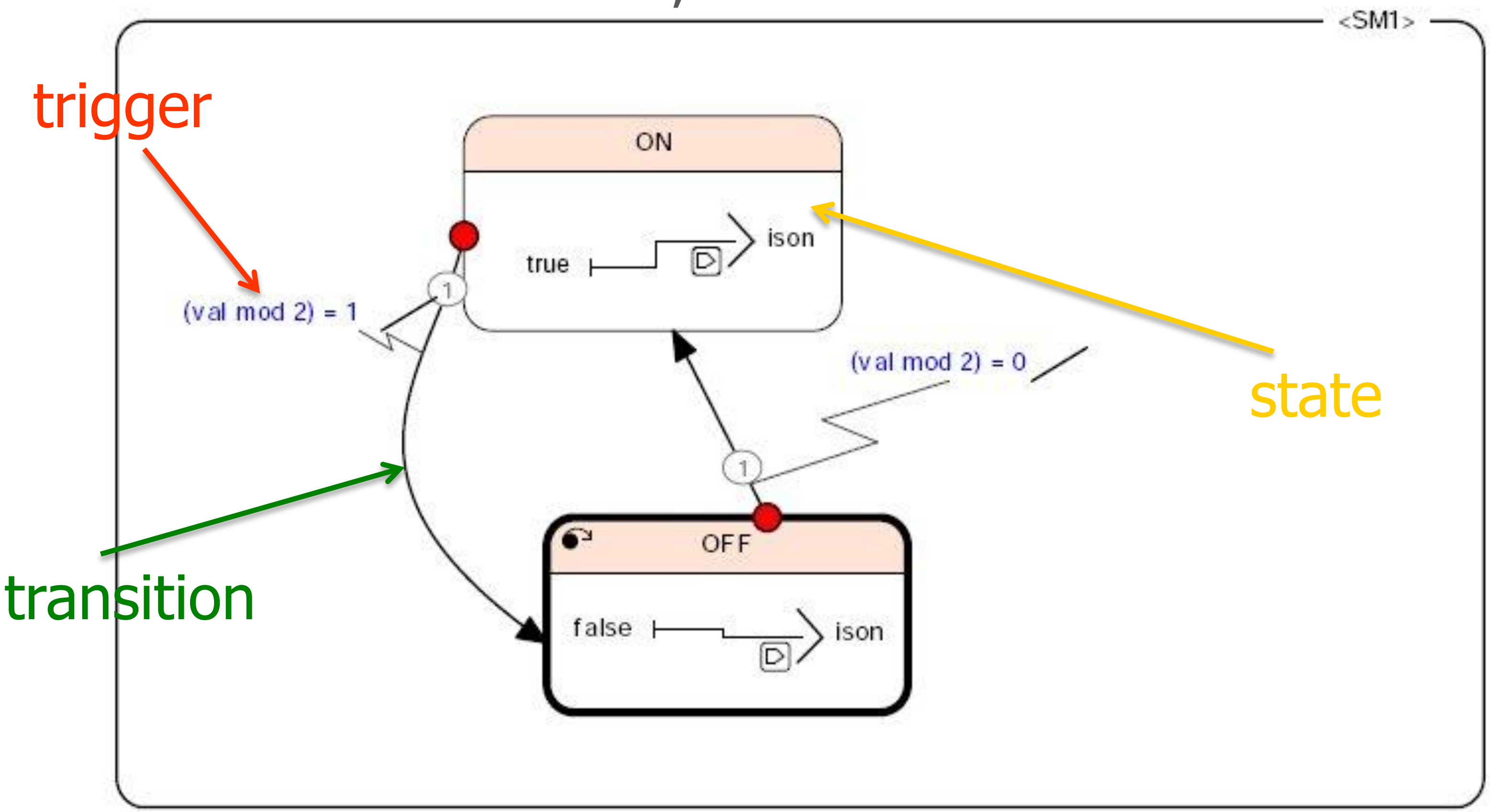
true
60
1
ModuloCounter
SECOND
2
ModuloCounter
60
MINUTES
dummy
3
ModuloCounter
24
HOUR

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

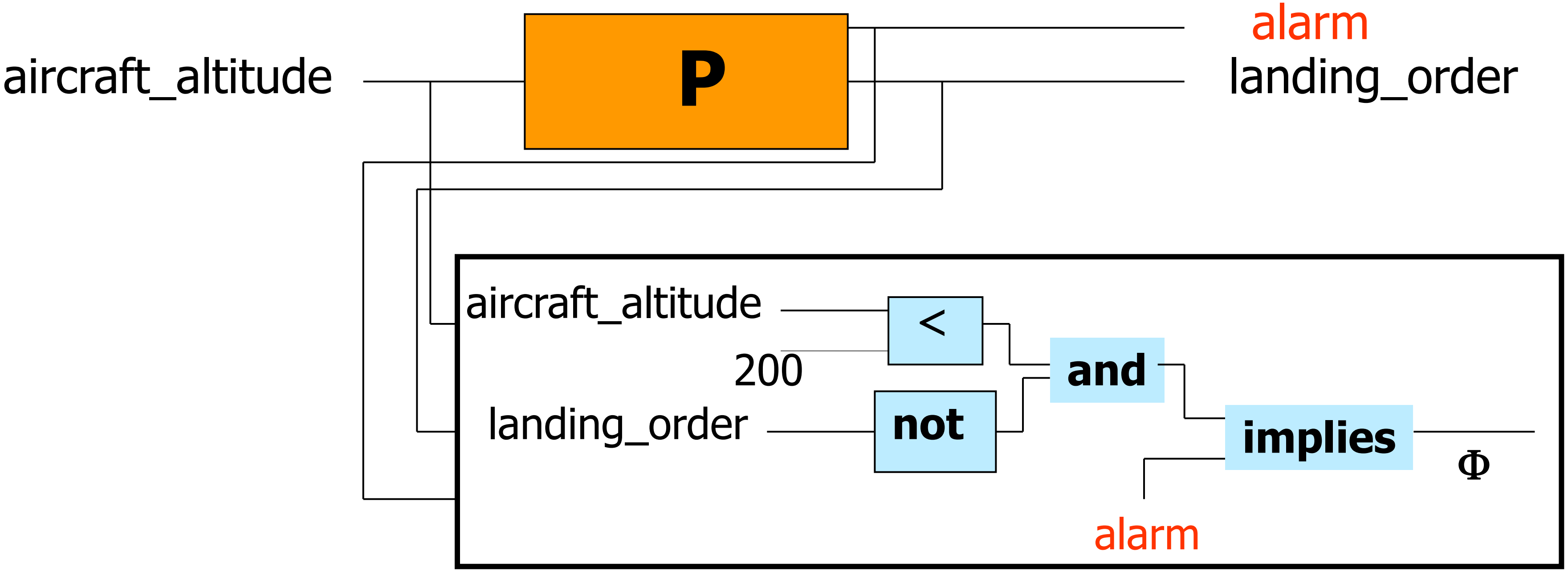


When off, ison = false

SCADE: model checking

Observers in Scade

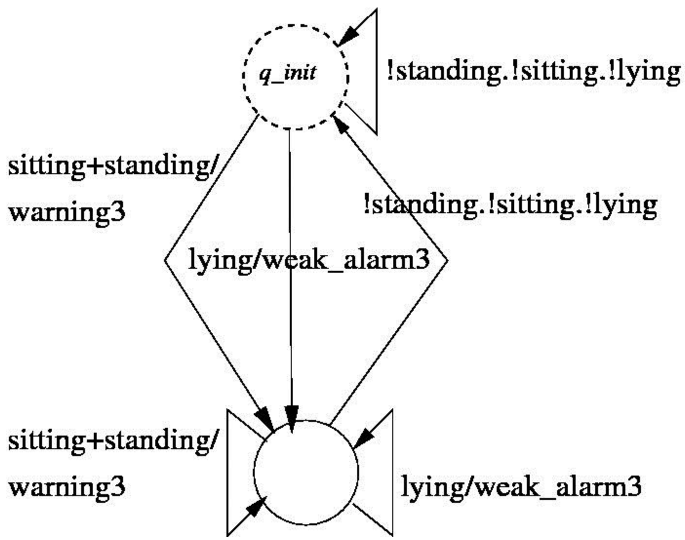
P: aircraft autopilot and security system



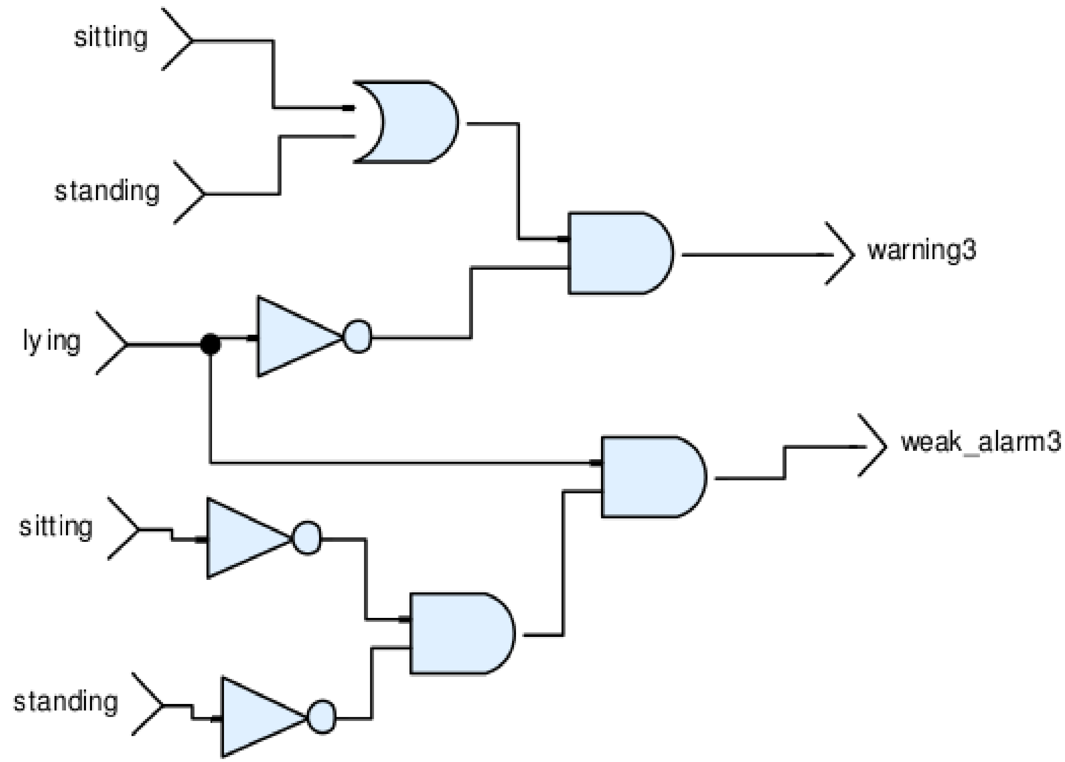
SCADE: model checking

Observer technique

posture model



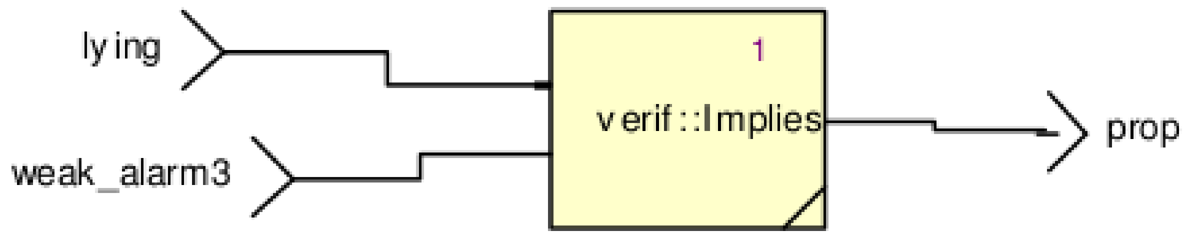
posture model specification in scade

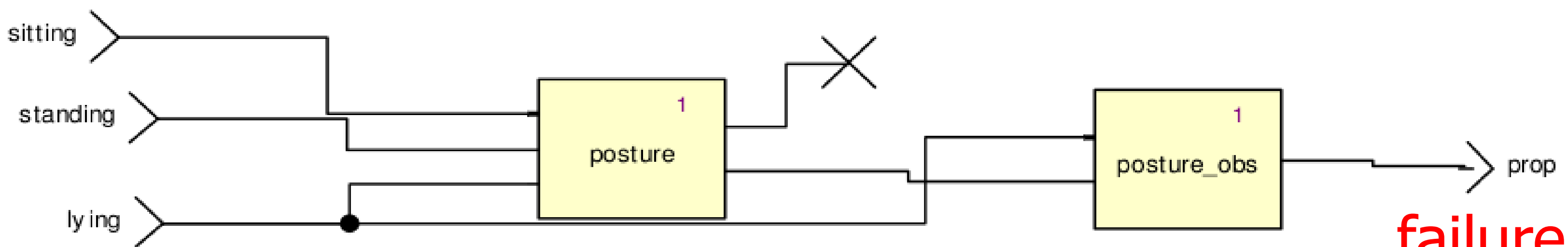


SCADE: model checking

Observer technique

posture observer





failure

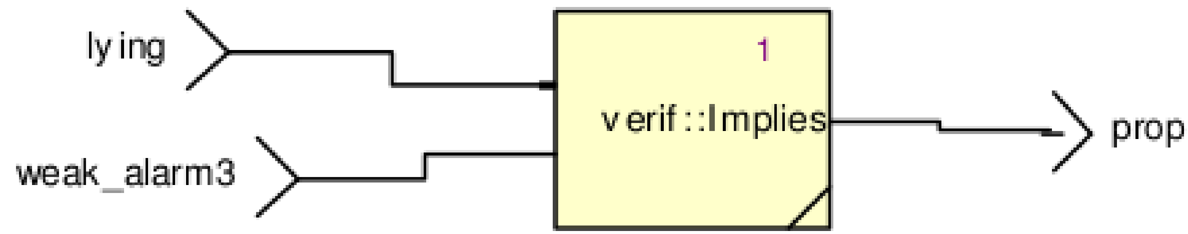
posture verification

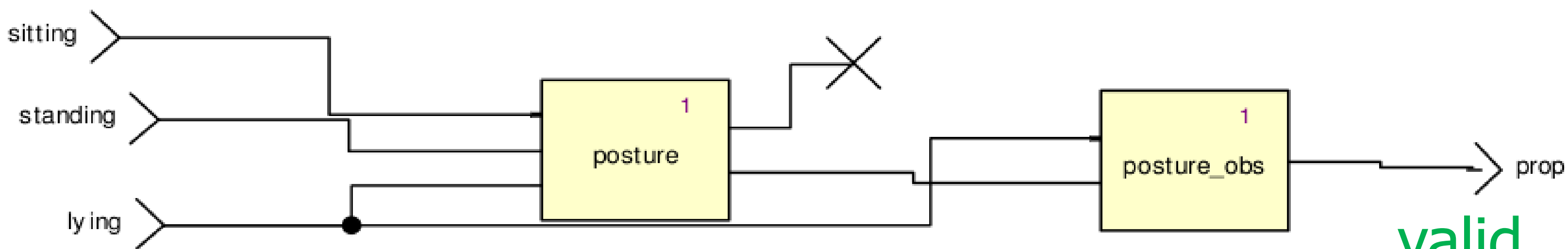
lying: true; sitting:true;standing:true

SCADE: model checking

Observer technique

posture
observer





valid

posture verification

assume (lying # sitting # standing)

SCADE: code generation

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

SCADE: code generation structures

- InC_<operator_name>
  - structure C
  - one member for each input
- OutC_<operator_name>
  - Structure C
  - one member for each output and each state
  - Other members for output/state computations

SCADE: code generation structures

- Reaction function
  - for a transition (or a reaction) computes the output and the new state
  - void <operator_name> (Inc_<operator_name> * inC, outC_<operator_name>* outc)
- Reset function
  - To reset the reaction and the structures
  - void <operator_name>_reset (outC_<operator_name>* outc

SCADE: code generation files

- Generated files
  - <operator_name>.h : type and function declarations for code integration
  - <operator_name>.c : implementation of reaction and reset functions
  - kcg_types.(h,c) to define types in C
  - kcg_conts.(h,c) to define contants

CHECKING TEMPORAL PROPERTIES

Properties Checking

- Liveness Property Φ :
  - $\Phi \Rightarrow$ automata $B(\Phi)$
  - $\mathbb{L}(B(\Phi)) = \emptyset$ décidable
  - $\Phi \models \mathcal{M} : \mathbb{L}(\mathcal{M} \otimes B(\sim\Phi)) = \emptyset$
- Scade allows only to verify safety properties, thus we will study such properties verification techniques.

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 (1)

- Mathematical framework:
  - S : finite state, ($\mathcal{P}$(S), $\subseteq$) is a complete lattice with S as greater element and $\emptyset$ as least one.
  - f : $\mathcal{P}(S) \longrightarrow \mathcal{P}(S)$:
    - f is monotonic iff $\forall$ x,y $\in \mathcal{P}$(S), $x \subseteq y \Rightarrow f(x) \subseteq f(y)$
    - f is $\cap$-continue iff for each decreasing sequence $f(\cap\, x_i) = \cap\, f(x_i)$
    - f is $\cup$-continue iff for each increasing sequence $f(\cup\, x_i) = \cup\, f(x_i)$

Safety Properties Verification (2)

- Mathematical framework:
  - if S is finite then monotonic $\Rightarrow$ $\cap$-continue et $\cup$-continue.
  - x is a fix point iff of f iff $f(x) = x$
  - x is a least fix point (lfp) iff $\forall y$ such that $f(y) = y$, $x \subseteq y$
  - x is a greatest fix point (gfp) iff $\forall y$ such that $f(y) = y$, $y \subseteq x$

Safety Properties Verification (3)

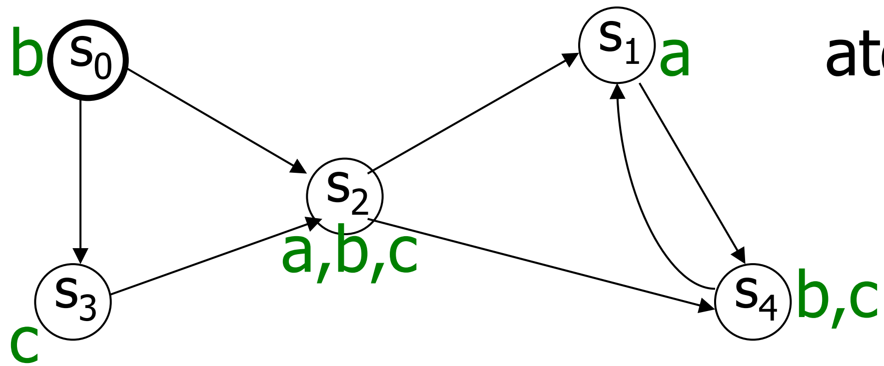
- Theorem:
  - f monotonic $\Rightarrow$ f has a lfp (resp glp)
  - $lfp(f) = \cup f^n(\emptyset)$
  - $gfp(f) = \cap f^n(S)$

Fixpoints are limits of approximations

Safety Properties Verification (4)

- We call Sat($\varnothing$) the set of states where $\varnothing$ is true.
- $\mathcal{M} \models \varnothing$ iff $s_{init} \in$ Sat($\varnothing$).
- Algorithm:
  - $Sat(\Phi) = \{ s \mid \Phi \models s \}$
  - $Sat(\text{not } \Phi) = S \backslash Sat(\Phi)$
  - $Sat(\Phi 1 \text{ or } \Phi 2) = Sat(\Phi 1) \cup Sat(\Phi 2)$
  - $Sat(EX\ \Phi) = \{ s \mid \exists t \in Sat(\Phi),\ s \rightarrow t \}$ (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)

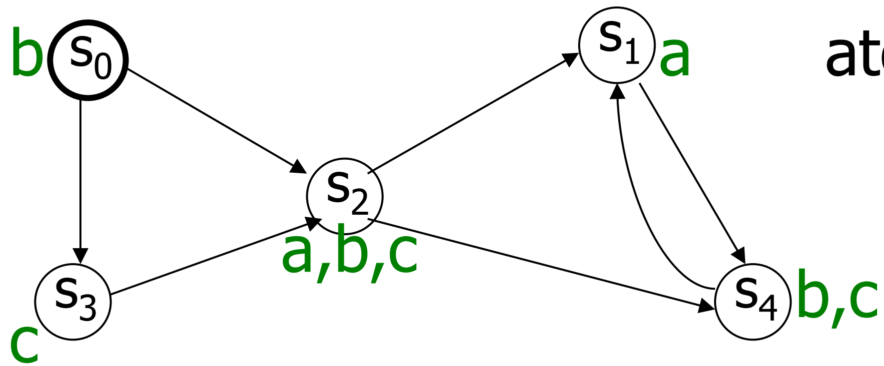
$$gfp\,(\Gamma(x) = \mathrm{Sat}(\Phi) \cap \mathrm{Pre}(x))$$

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

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

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

Example



atomic formulas: a, b, c

EG (a or b) $\quad \Gamma(\{s_0, s_1, s_2, s_{3,} s_4\}) = \{s_0, s_1, s_2, s_4\}$

$\Gamma(\{s_0, s_1, s_2, s_4\}) = \text{Sat (a or b)} \cap \text{Pre}(\{s_0, s_1, s_{2,}, s_4\})$

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

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

Model checking implementation

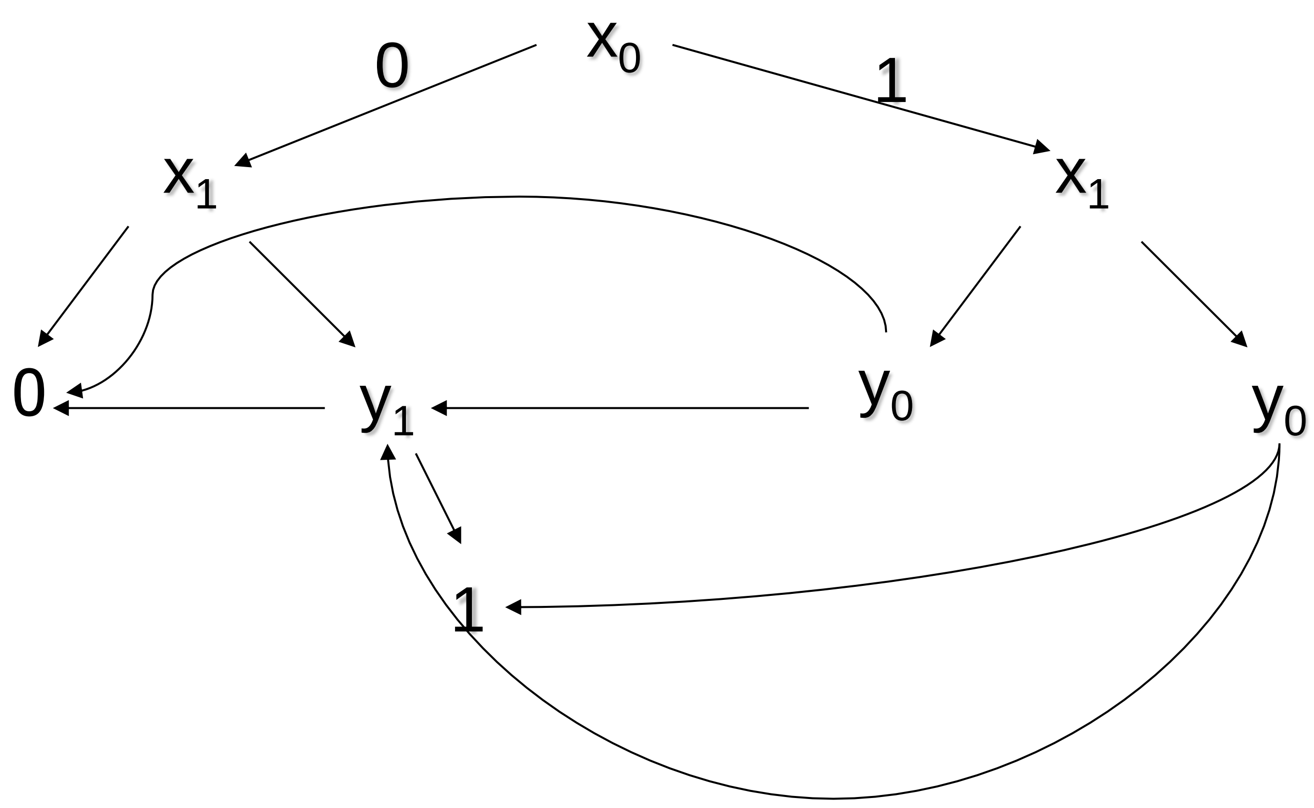
- Problem: the size of automata
- Solution: symbolic model checking
- Usage of BDD (Binary Decision Diagram) to encode both automata and formula.
- Each Boolean function has a unique representation
- Shannon decomposition:
  - $f(x_0,x_1,\ldots,x_n) = f(1, x_1,\ldots., x_n) \vee f(0, x_1,\ldots,x_n)$

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

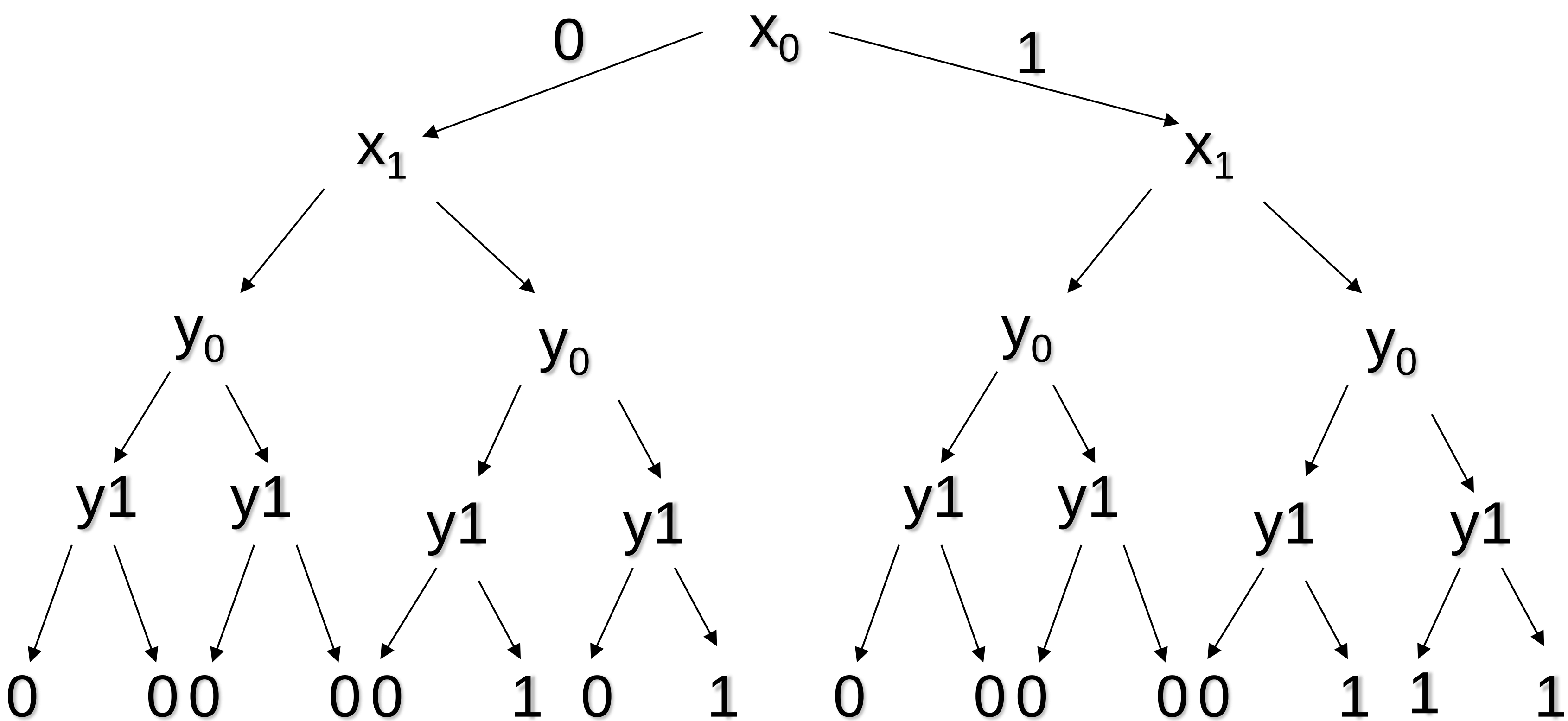
Model Checking Implementation (2)

$(x_1 \wedge x_0) \vee ((x_1 \vee y_1) \wedge (x_0 \wedge y_0))$



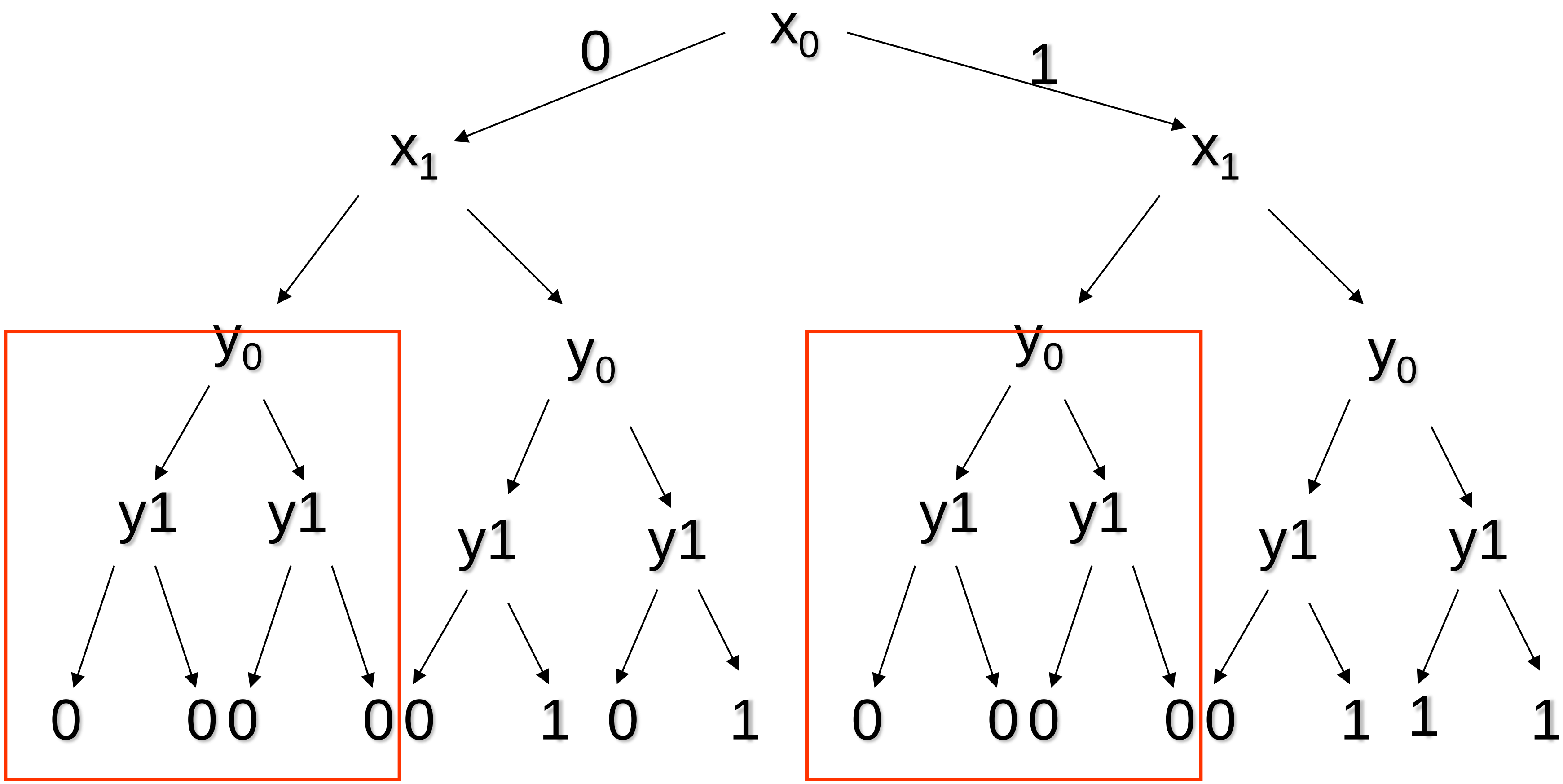
Model Checking Implementation (2)

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



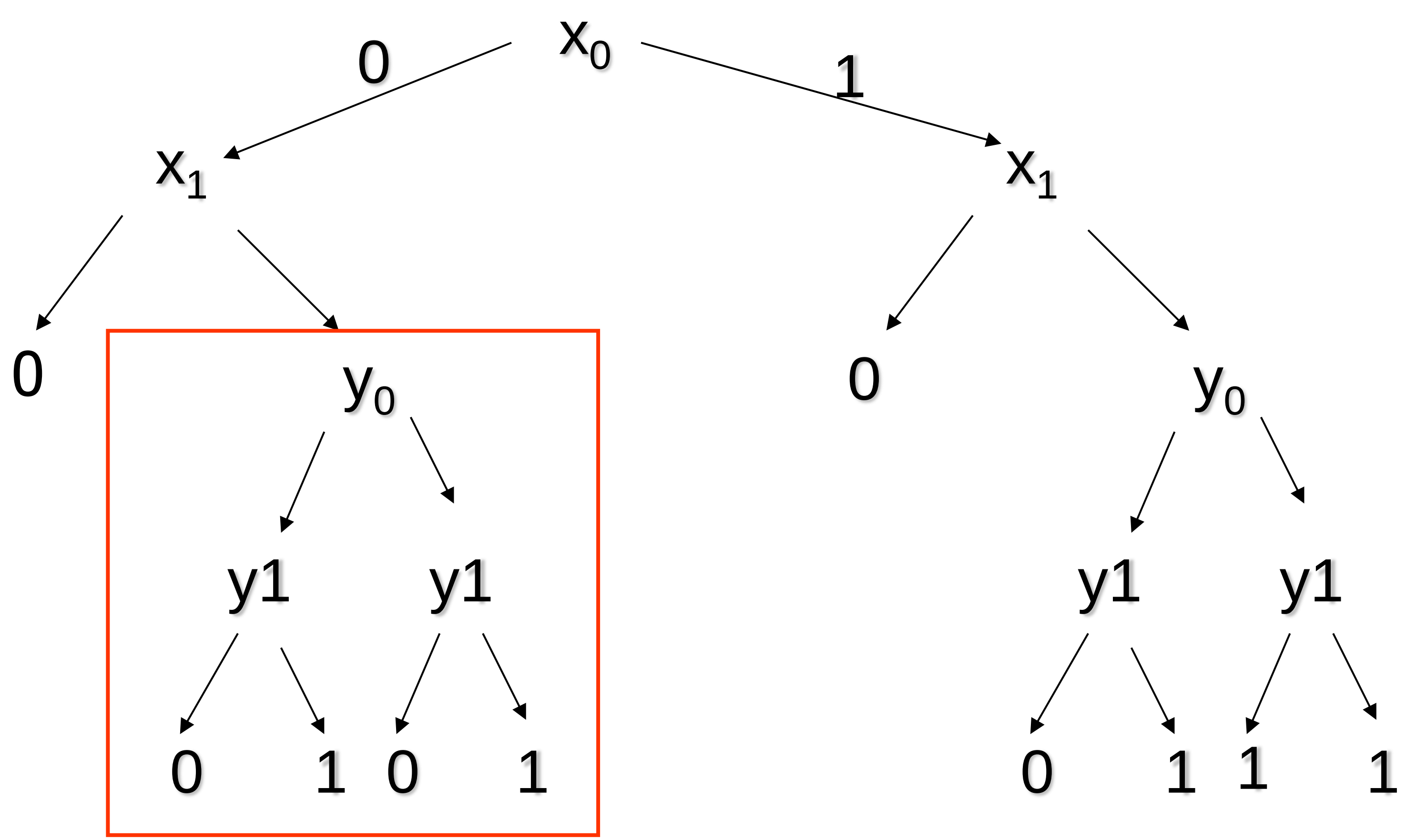
Model Checking Implementation (2)

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



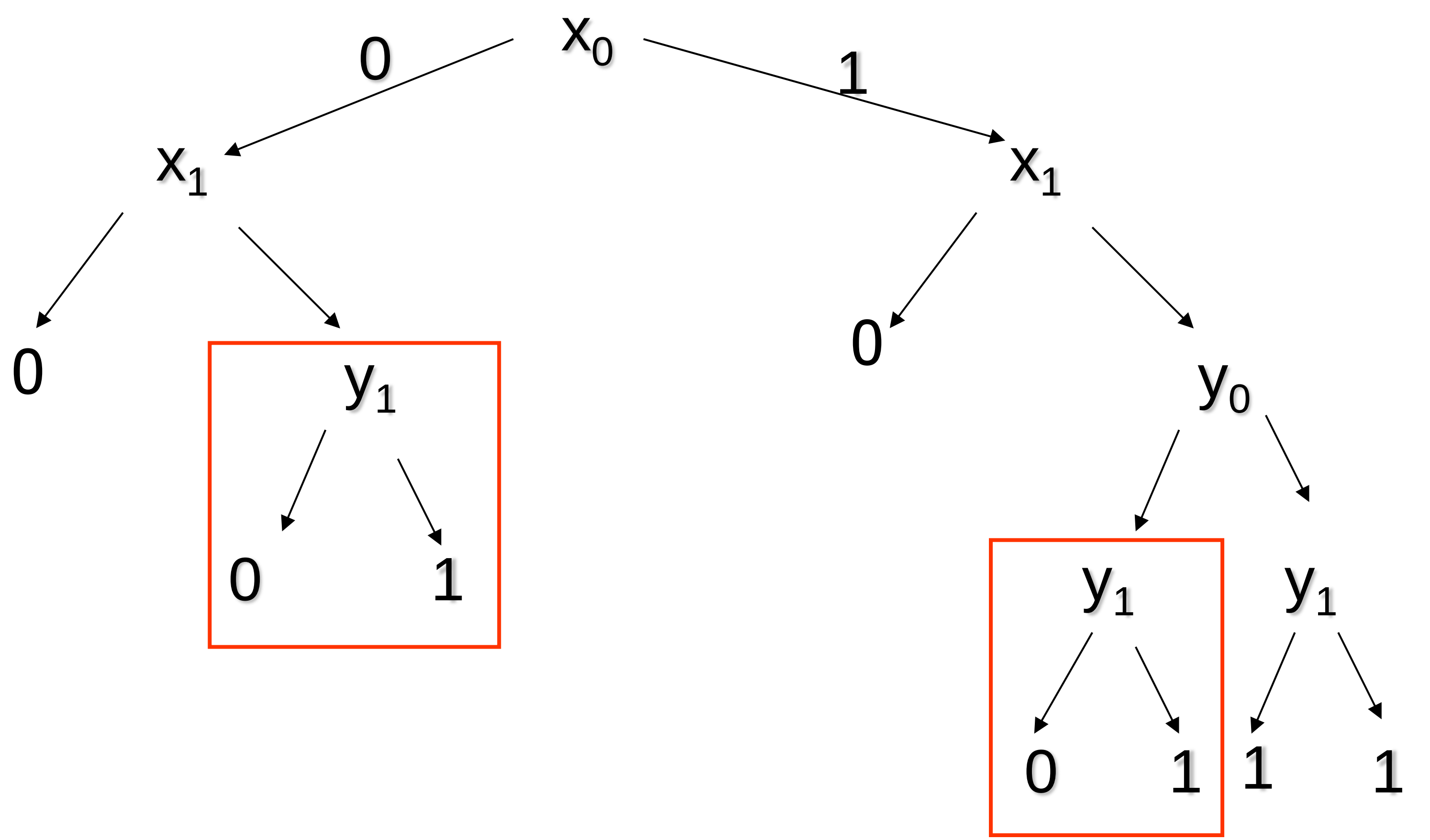
Model Checking Implementation (2)

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



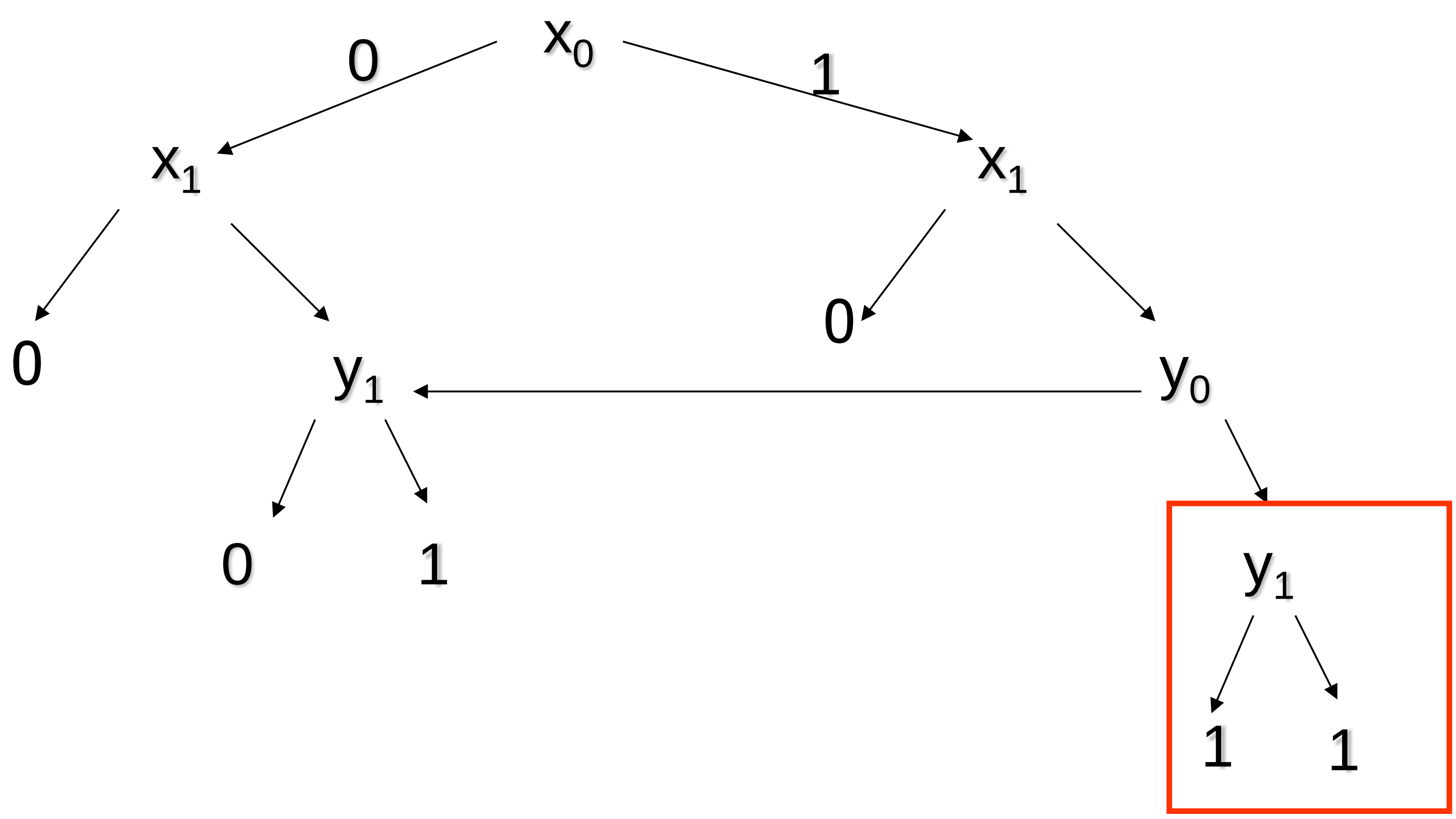
Model Checking Implementation (2)

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



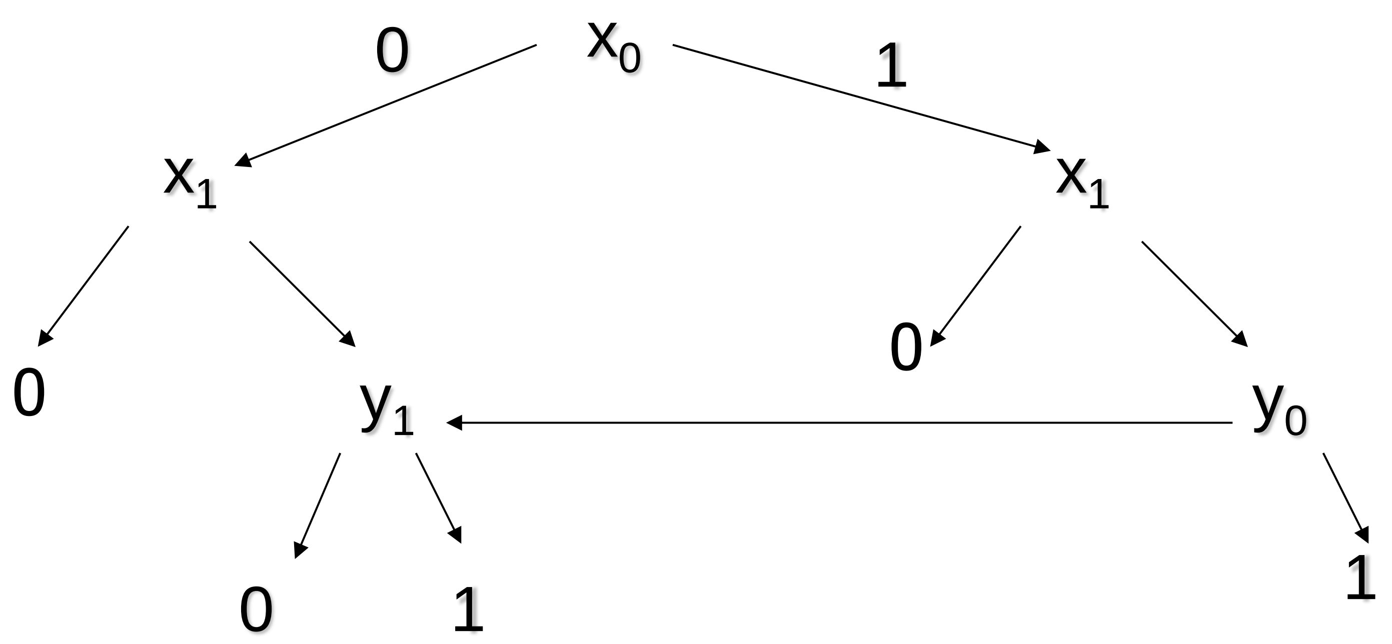
Model Checking Implementation (2)

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



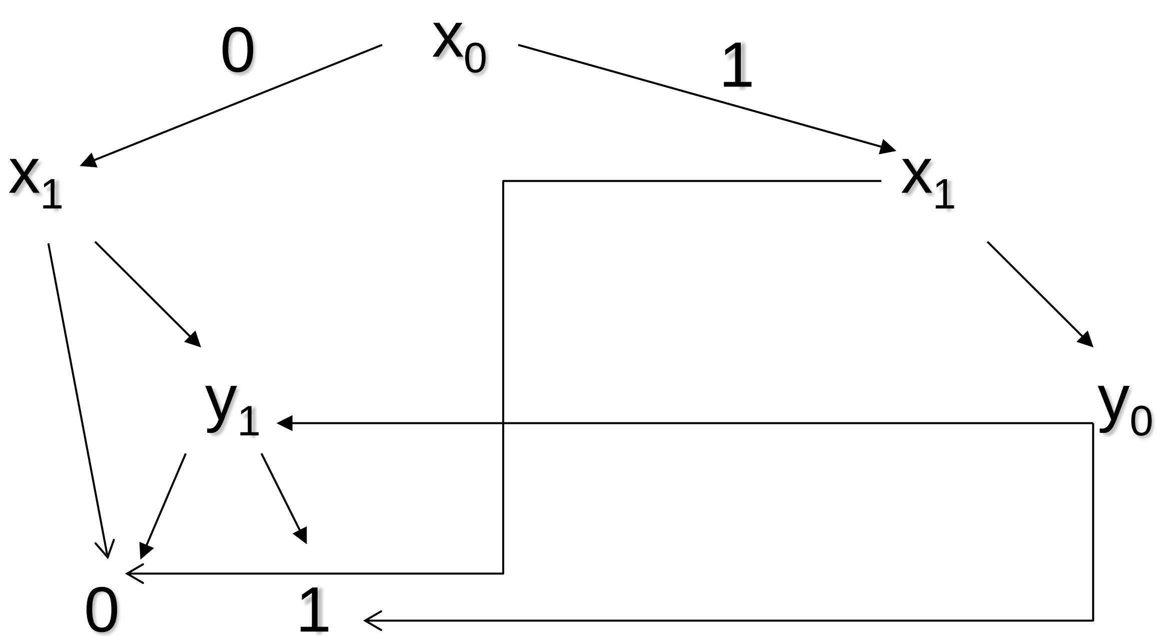
Model Checking Implementation (2)

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



Model Checking Implementation (2)

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



Model Checking Implementation(3)

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

Model Checking Implementation (4)

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

Model Checking Implementation (4)

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

Model Checking Implementation (5)

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

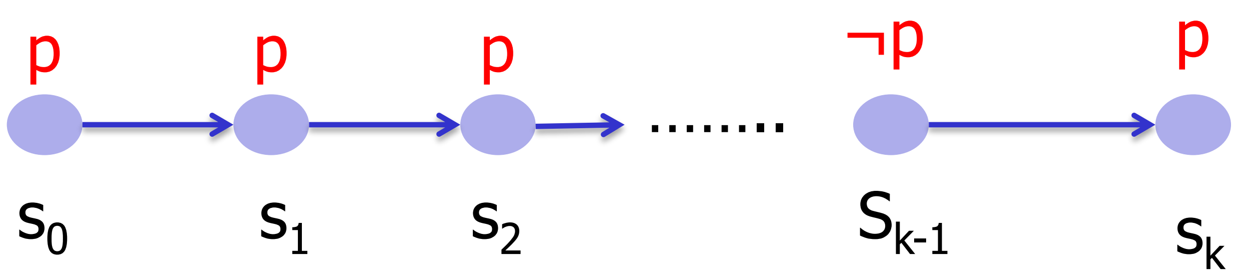
Bounded Model Checking

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

Bounded Model Checking

Given a property p

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



Bounded Model Checking

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

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

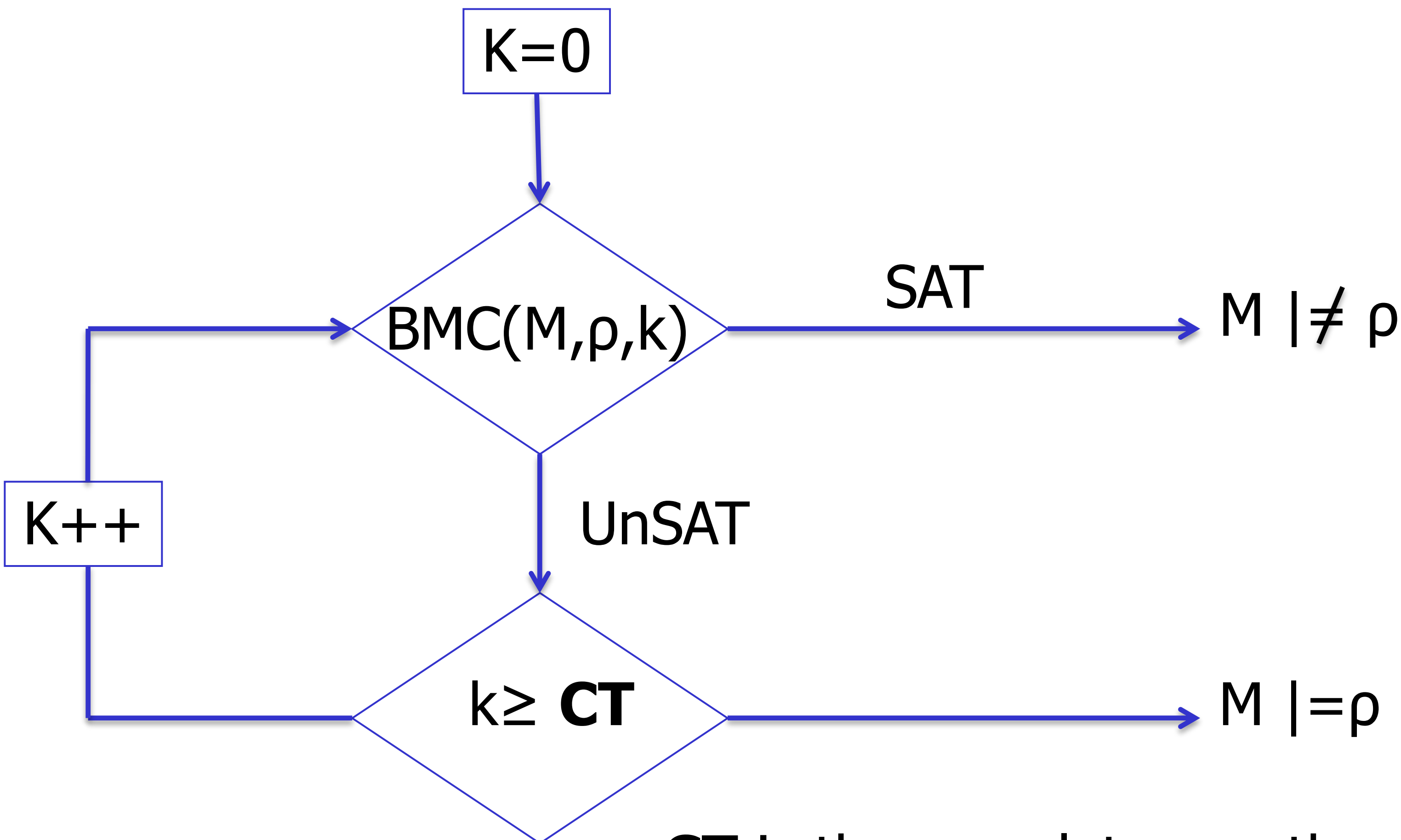
The property p fails in one of the k steps

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

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

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

Bounded Model Checking



CT is the completeness threshold