IF observers - goals

- control model generation process
  - cut off generation of irrelevant states
  - act as dynamic scheduler
  - model the environment
  - inject faults

- express properties operationally
  - linear, safety, timed
the observer entity

- extended timed automaton (~ IF process)
  - executed in parallel with an IF system
- no signal queue
  - may send signals (→ act as environment, inject faults)
- can observe
  - every part of a system (variables, states, queues…)
  - events occurred during the previous system step
    (I/O, fork/kill, message delivery…) → { observable event set }
- can modify
  - variables, queues…
state observation

- special operators, functions, types
  - process-in-state operator
  - queue observation functions
    - get_length(), get_signal_at()...
    - generic signal type, signal cast operators

- data observation
  - import of variables (~ IF processes)
  - unrestricted by export clauses
event observation

- the **match** clause
  ≈ input clause for observed events (acts on the event set)
- retrieves data related to the event
  - parameter values, PIDS of involved processes...
- examples:
  - **match input** ODATA (x,n) in RX
  - **match output** ODATA (x,n) from TX via R2 to RX
  - **match fork** (newpid) RX in Daemon
  - **match kill** (newpid) RX in Daemon
  - **match deliver** ODATA(x,n) from R2
  - **match informal** “advance window” in TX
control & properties

- **specific actions**
  - cut -- stops all system execution
  - flush -- forces the erase of the event set

- **state classification**
  - ordinary, error, success states
  - optionally used in state space exploration

- **observer classification**
  - pure $\subseteq$ cut $\subseteq$ intrusive

[more on the semantics...]
example: alternating bit

- **Property**: every time a \( put(m) \) is received, the transmitter does not return in the state idle until a \( get(m) \) with the same \( m \) is issued by the receiver

```plaintext
pure observer safety1;
var m data;
var n data;

state idle #start ;
  match input put(m);
  nextstate wait;
endstate;

state wait;
  provided ((transmitter)0) instate idle;
  nextstate err;
  match input put(m);
  nextstate wait;
match output get(n);
  nextstate dec;
endstate;

state dec #unstable ;
  provided n = m;
  nextstate idle;
  provided n <> m;
  nextstate wait;
endstate;

state err #error ;
endstate;
endobserver;
```