Property-based design with HORUS / SYNTHORUS

Dominique Borrione, Negin Javaheri, Katell Morin-Allory, Yann Oddos, Alexandre Porcher

Radboud University, Nijmegen

March 27, 2013

Functional specifications with assertions

- **Assertion:**
  - **A design property that is declared to be true**
  - **Declarative style**
  - **An assertion states expected facts about the design or its environment**

Assert A implies next[2] (B = C)

- **Assertions are used to express functional design intent**
  - **Expected input-output behavior**
  - **Constraints on inputs**
  - **Forbidden behavior**
  - **Architectural properties (fairness, deadlock)**

Use of Assertions

- **Where should you place assertions?**
  - **White-box: in the design itself (simulation, formal verification)**
  - **Black-box: observe interface constraints and protocols from the outside**
  - **Test-bench support: verification during simulation/emulation or offline**

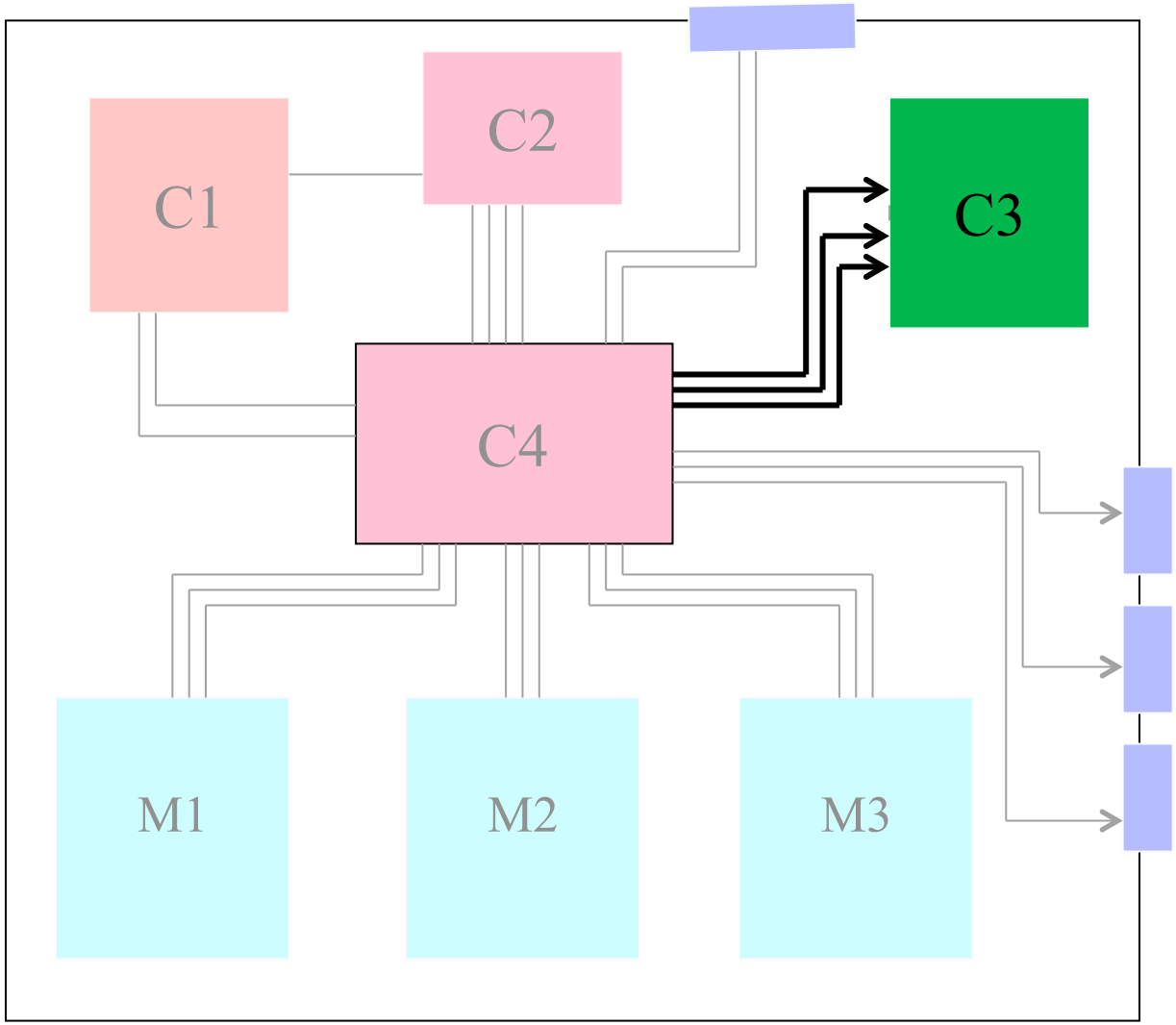
Processing Assertions

- Assertion evaluation techniques
  - Simulation
  - Emulation
  - Formal Verification
  - Test patterns Generation
  - On-line monitoring

Some implementations

- **Simulation: all modern simulators implement PSL/SVA monitors**
  - **Check properties during simulation**
  - **Much easier for debugging than formal tools**
- **Automatic Property Checking**
  - **Industrial Model Checkers: Magellan, Incisive, 0-in, Jasper Gold, etc...**
  - **University tools: RAT, Cando**
- **Emulation, on-line test**
  - **FoCs: first industrial tool**
  - **Many industrial tools restricted to OVL**
  - **Academic tools: MBAC (McGill), Horus (TIMA), …**

Assume-Guarantee Paradigm



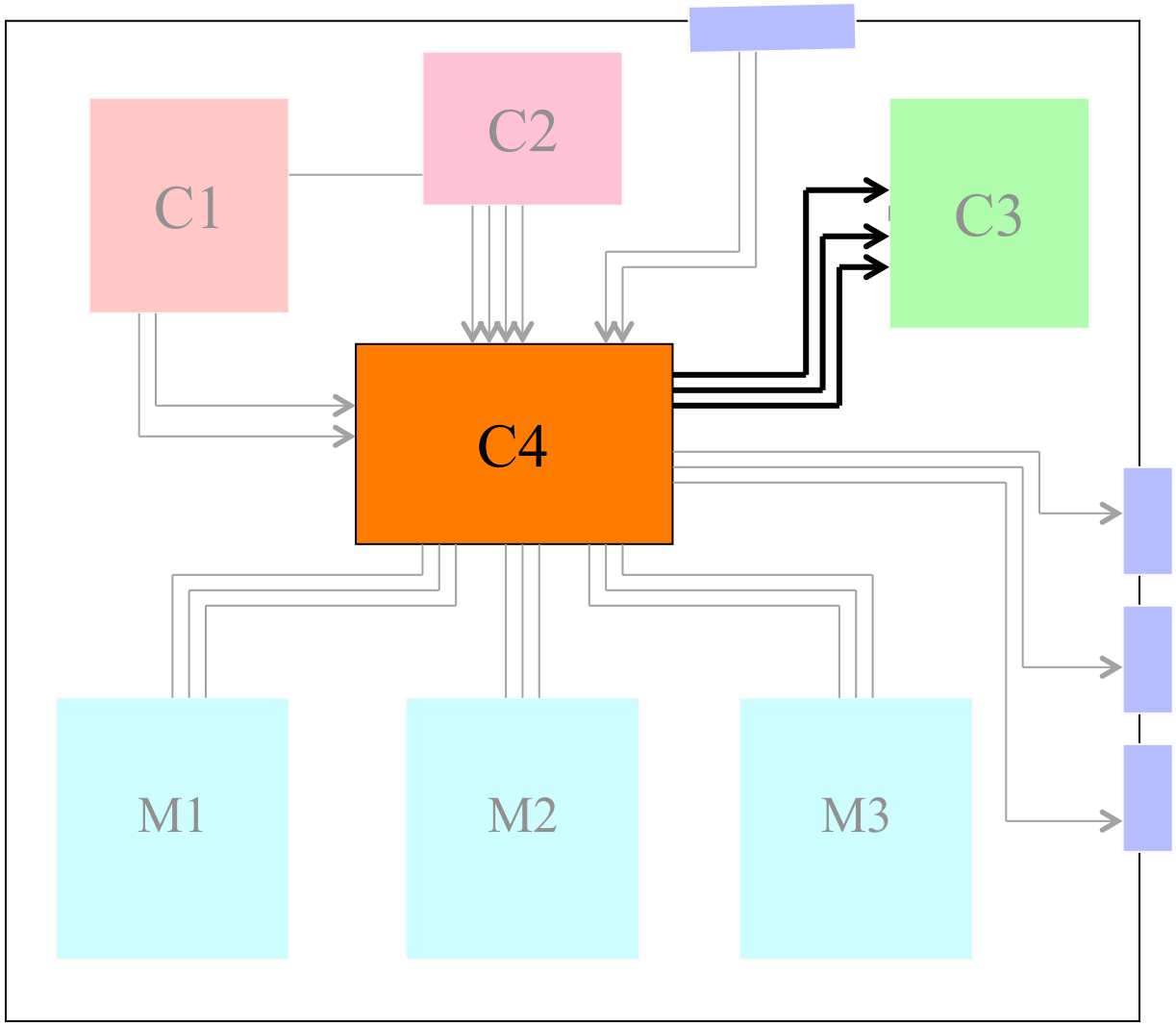
Properties used to describe:

- **environment behavior assumptions**

Example

Assume
Always A -> { B ; C }

Assume-Guarantee Paradigm



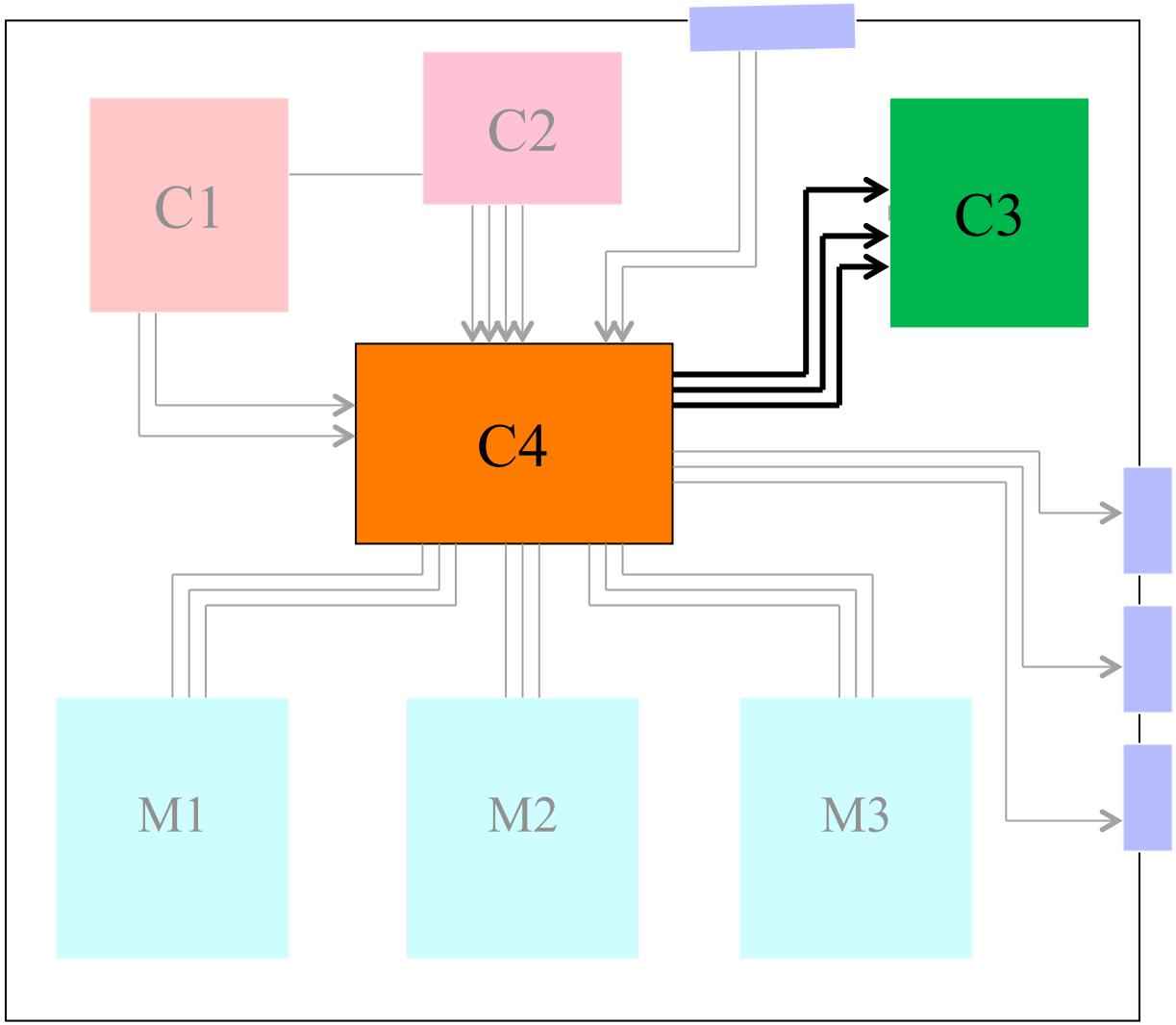
Properties used to describe:

- **design behavior assertions**

Example

Assert
Always A -> { B ; C }

Assume-Guarantee Paradigm



Same property used to describe:

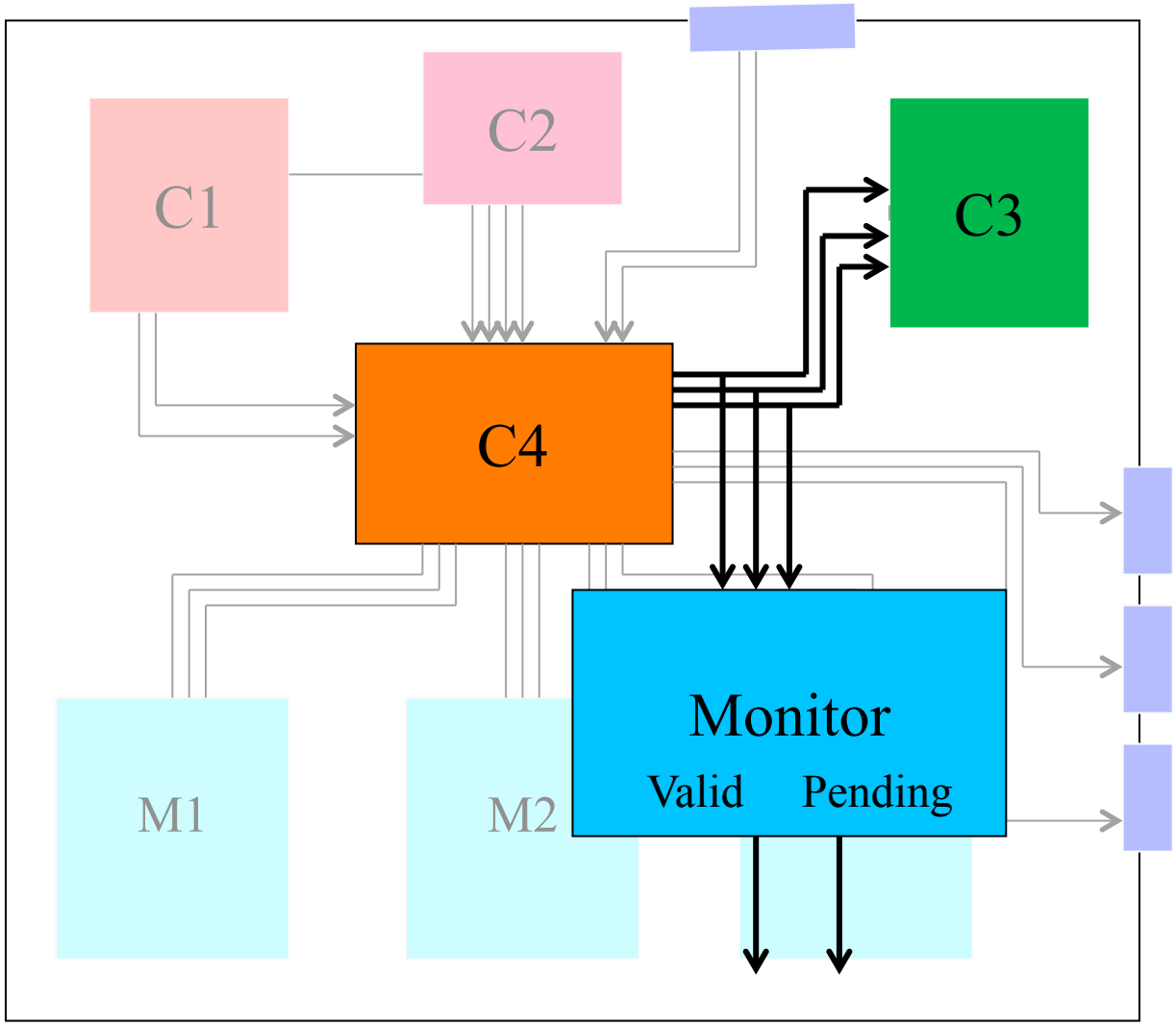
- **environment behavior assumption for C3**
- **design behavior assertion for C4**

For online verification, synthesize assumptions and assertions in hardware

Example

Always A -> { B ; C }

Assume-Guarantee Paradigm



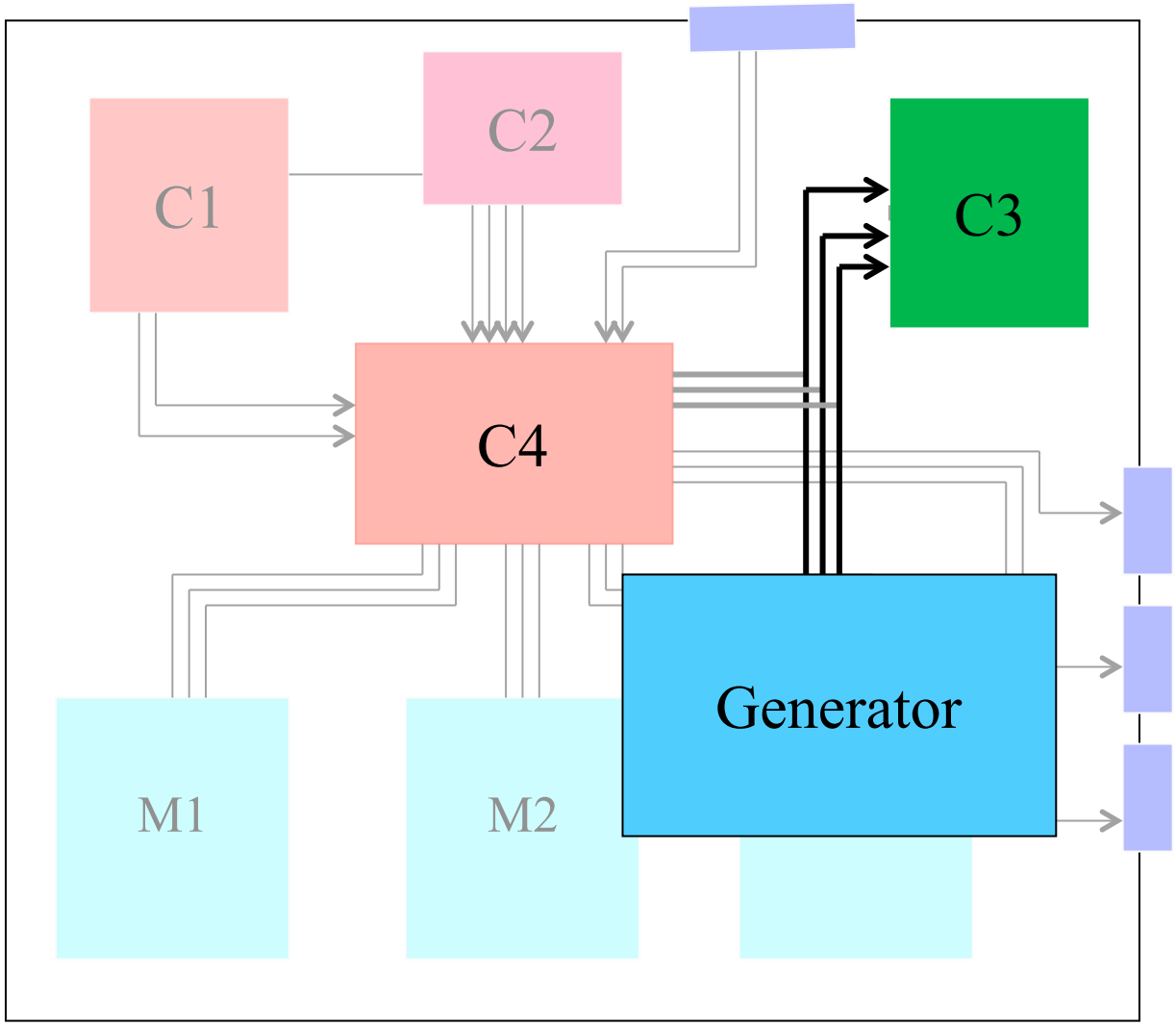
For online verification

synthesize assertions as monitors

Example

Assert
Always A -> { B ; C }

Assume-Guarantee Paradigm



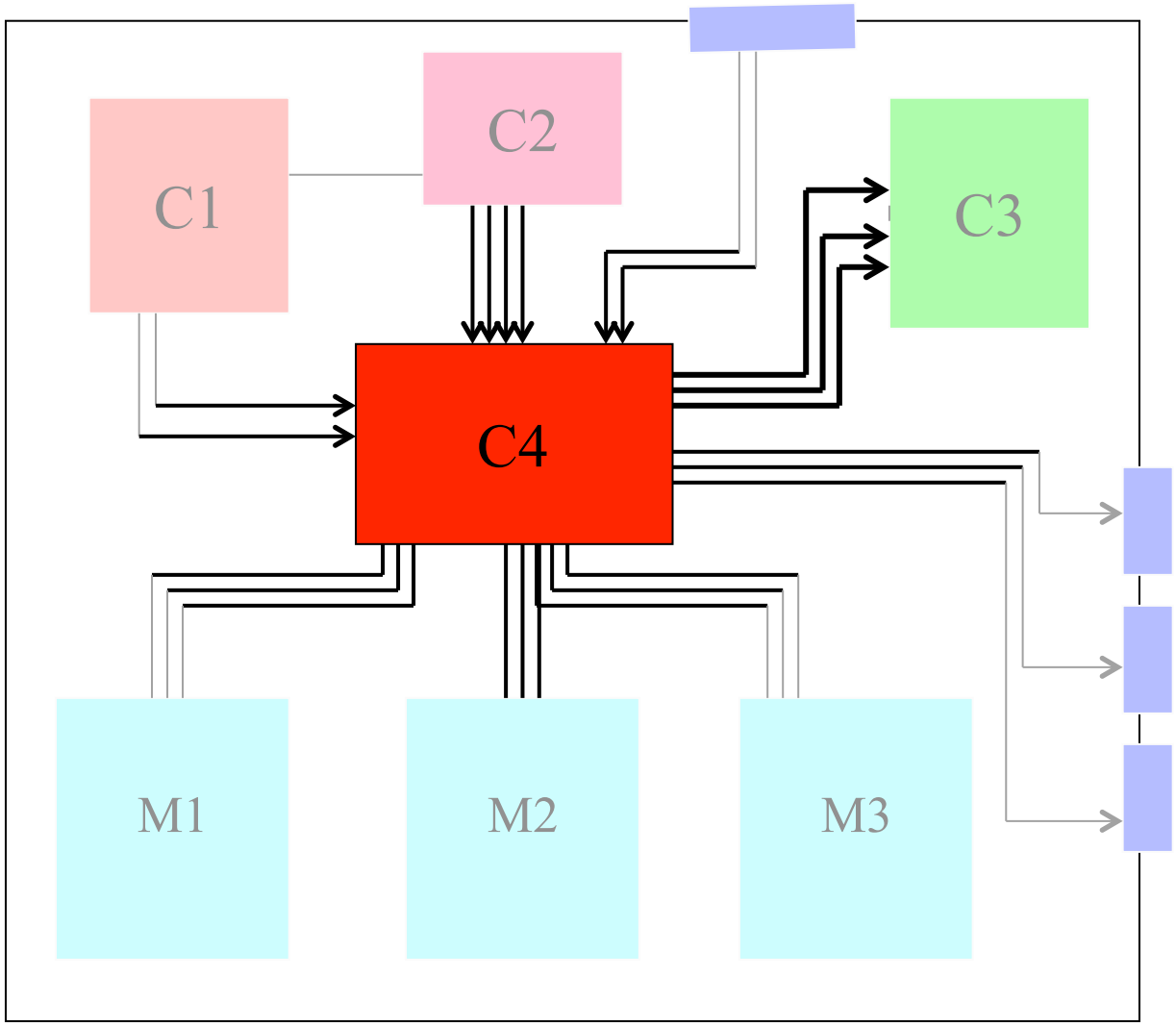
For online verification

synthesize assumptions as generators

Example

Assume
Always A -> { B ; C }

Automatic module design from assertions



Properties completely specify the design behavior:

- **Input-output relations**
- **Over time**
- **For all interface signals**

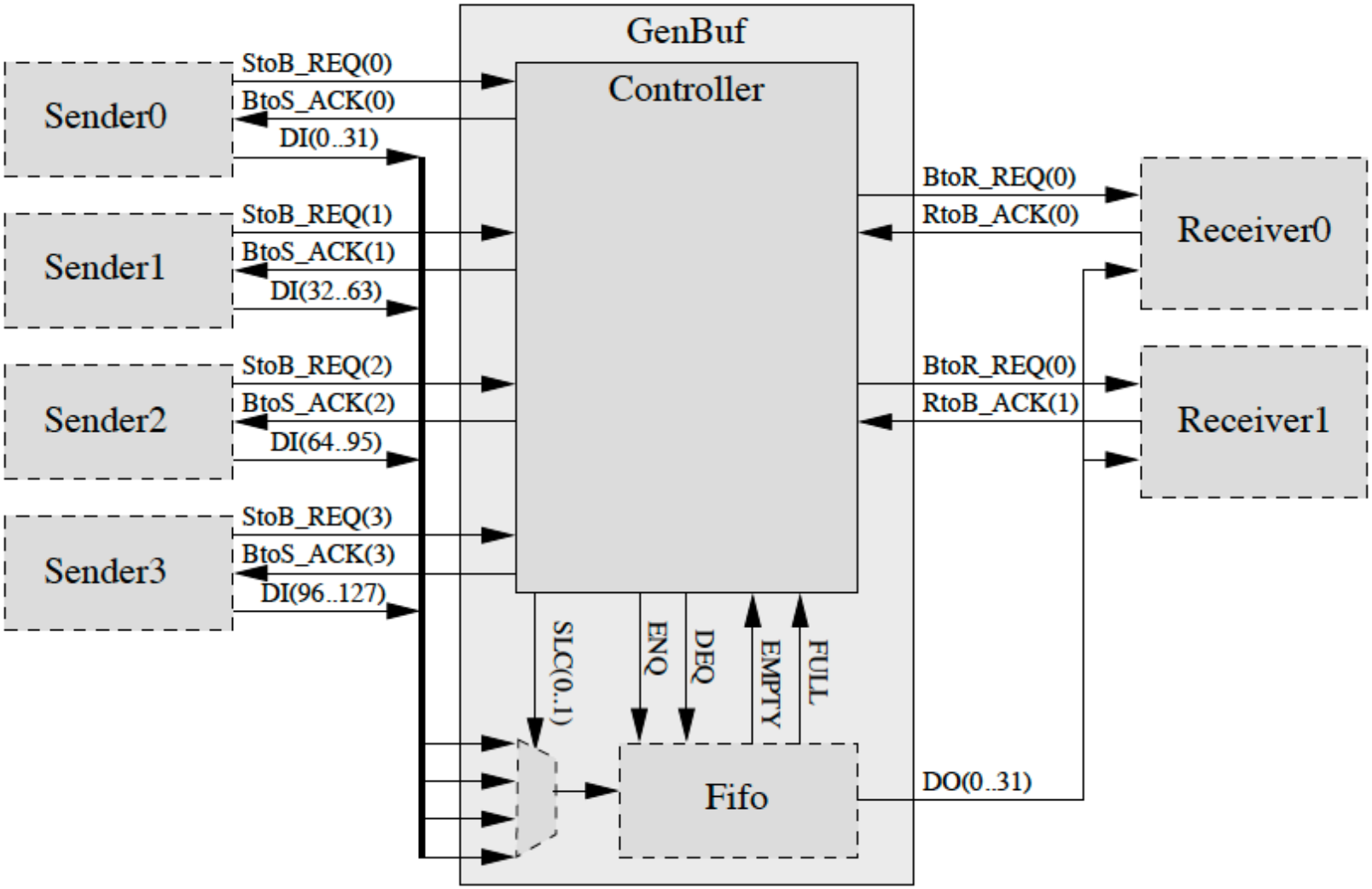
Outline

- A simple running example
- Brief introduction to Property Specification Languages
- Proven correct hardware verification IP's from PSL
- Automatic Hardware Generation from PSL: problems and partial solutions
- Conclusion

Generalized Buffer

- **Sources**
  - **https://www.research.ibm.com/haifa/projects/verification/RB_Homepage/tutorial3/GenBuf_english_spec.htm**
  - **R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, M. Weiglhofer: « Specify, Compile, Run: Hardware from PSL », Electronic Notes in Theoretical Computer Science, vol190, N°4, 2007**
- **GenBuf**
  - **Queues words of (32 bit) data sent by 4 senders to 2 receivers**
  - **Depth 4 FIFO**
  - **Interface GenBuf <-> senders: 4-phase handshaking protocol**

Generalized Buffer



Outline

- A simple running example
- **Brief introduction to Property Specification Languages**
- Proven correct hardware verification IP's from PSL
- Automatic Hardware Generation from PSL: problems and partial solutions
- Conclusion

Temporal properties

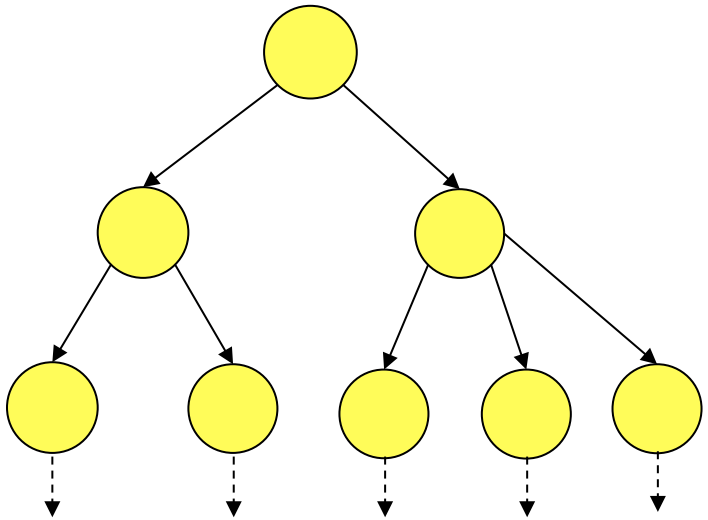
Used to assert desirable features (safety, liveness, absence of deadlock, absence of starvation …) of a circuit description.

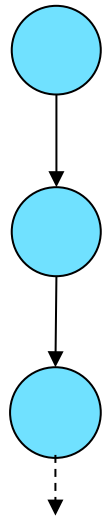
Partial specifications, expressed in some temporal logic (LTL, CTL, PSL, SystemVerilog, etc.).

Verified over all the states of one FSM, reachable from the initial state.

Looking at TIME

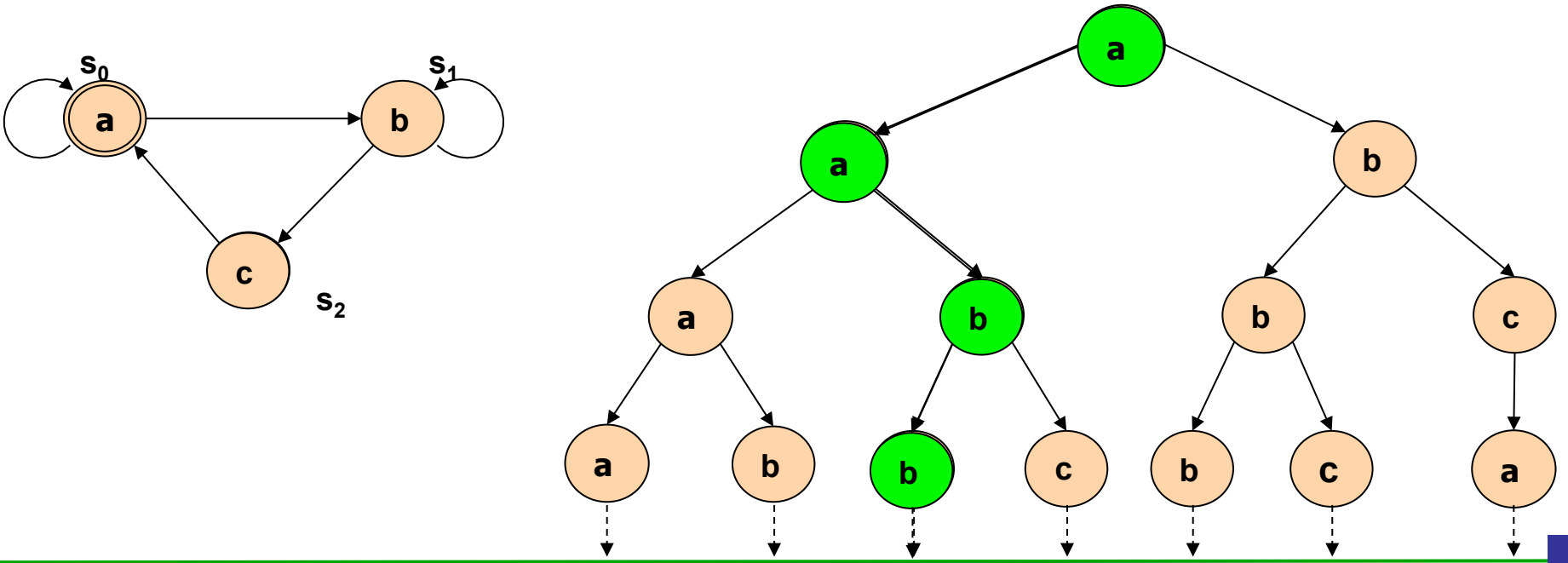
- **Linear** vs. **branching** time
- **Discrete** vs. continuous time
- **Time points** vs. time intervals
- Past time vs. **future time reasoning**





Computation Tree

- Obtained by unrolling the FSM
- Used to define the CTL Semantics
- Path view to the structure



The limits of readability

- **Each time a request is raised and the acknowledge is low, the acknowledge is raised within 3 to 5 cyles**

$$G\ (\ \neg REQ \wedge \neg ACK \wedge X(REQ \wedge \neg ACK) \rightarrow$$
$$X\ \ (X\ (\neg ACK) \wedge XX\ (\neg ACK) \wedge$$
$$(\ XXX\ ACK \vee\ XXXX\ ACK \vee\ XXXXX\ ACK)\ \ ))$$

A more user friendly expression

- **Each time a request is raised and the acknowledge is low, the acknowledge is raised within 3 to 5 cyles**

- **With sequential regular expressions**

  always {not REQ and not ACK; REQ and not ACK} |=>

  {not ACK[*2 to 4]; ACK}

- **With temporal operators**

  always rose (REQ) and not fell(ACK) and not ACK ->

  (next_a[1 to 2] (not ACK) and (next_e [3 to 5] ACK)

Brief History of PSL

- **Sugar 1.0**
  - **94: syntactic sugar for CTL for formal verification with RuleBase (IBM)**
  - **95: addition of regular expressions**
  - **97: automatic generation of simulation checkers**
- **Sugar 2.0**
  - **01: from CTL, moved to linear-time LTL semantics**
  - **02: selected by Accelera for IEEE standardisation**
- **03: Reference Manual: PSL 1.0**
- **05: IEEE standard**
- **08: Included in VHDL standard**

4 layers of language in PSL

- **Boolean layer**
  - rose (REQ) and not fell(ACK) and not ACK
- **Temporal layer**
  - Sequences of values over time {not ACK[*2 to 4]; ACK}
  - Temporal relations between signals next_a[1 to 2] (not ACK)
- **Verification layer**
  - Directives to the software: assert, assume, fairness, cover
- **Modeling layer**

  REQ <= REQ1 or REQ2

assert always rose (REQ) and not fell(ACK) and not ACK ->
(next_a[1 to 2] (not ACK) and (next_e [3 to 5] ACK)

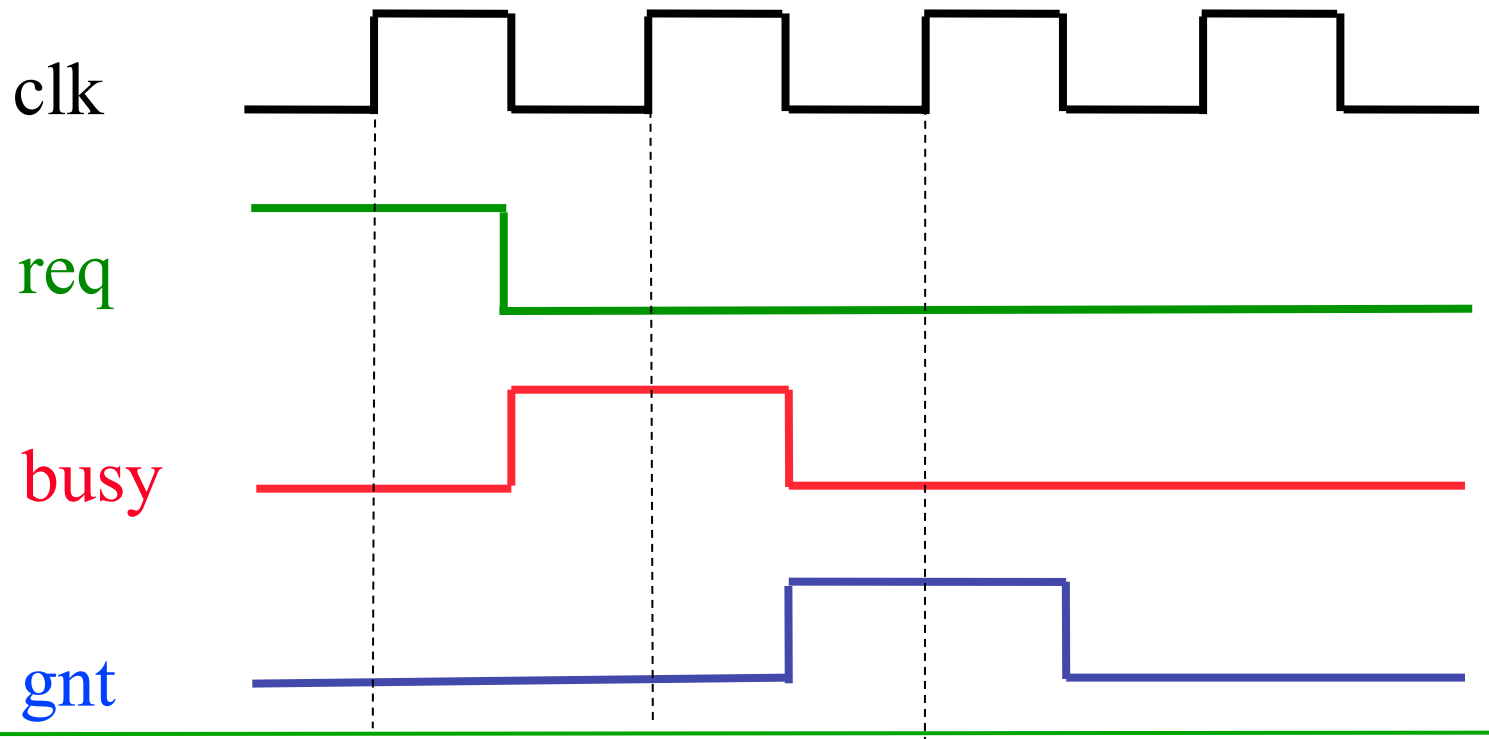
SERE Construction

- A SERE is a sequence of Boolean expressions over one or more signals
- If only a signal name appears, it means the value of the signal is TRUE
- In the following slides, we assume a default rising clock edge

SEREs – Example1

A SERE **describes** a set of sequences of states
(which are represented using timing diagrams)

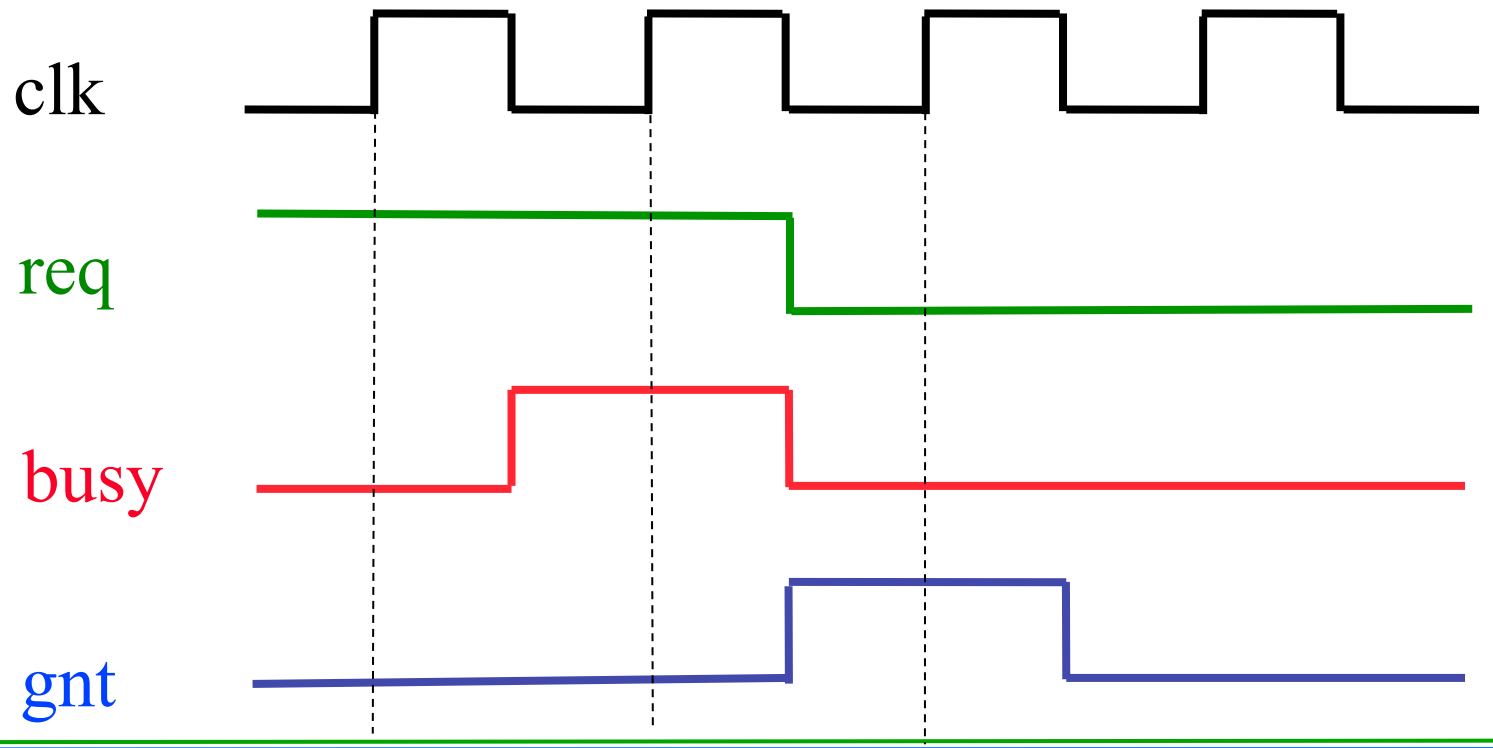
{req; busy; gnt}



SEREs – Example1

This diagram is also described by the same SERE

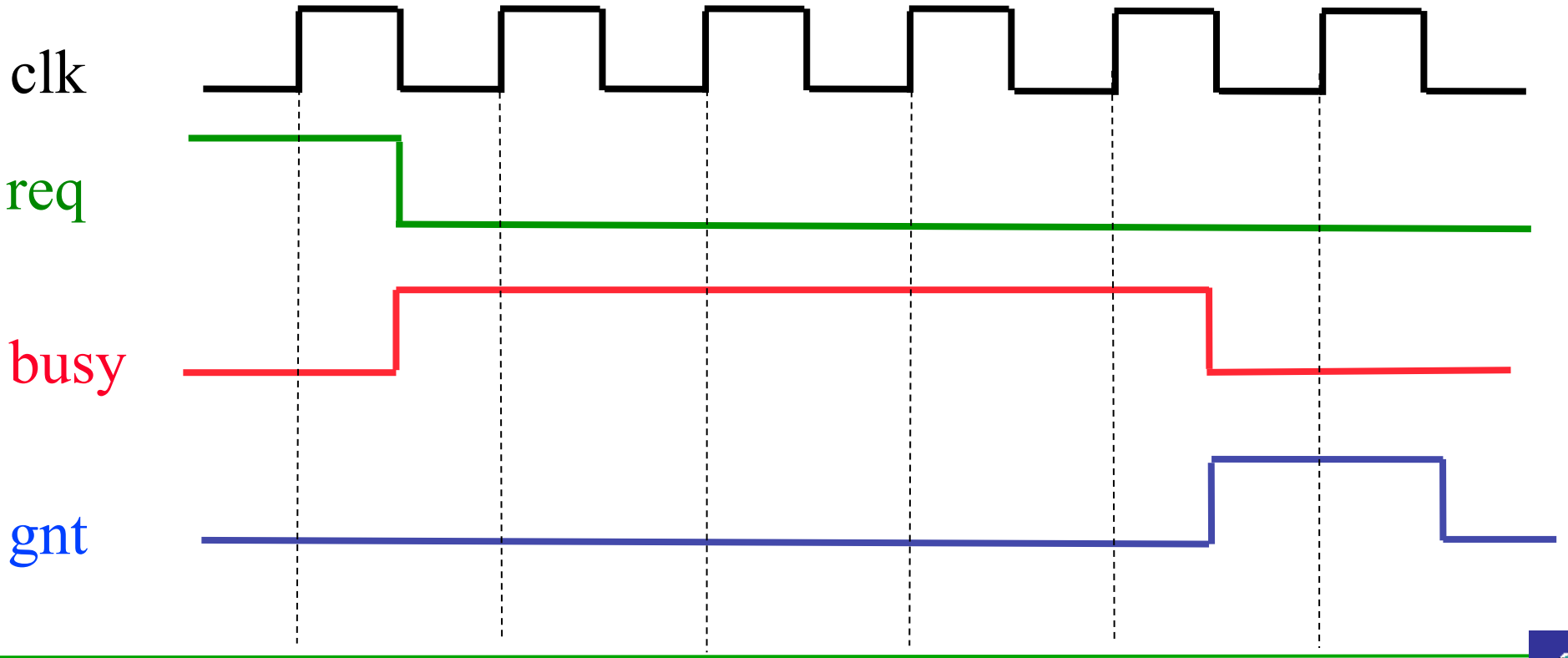
{req; busy; gnt}



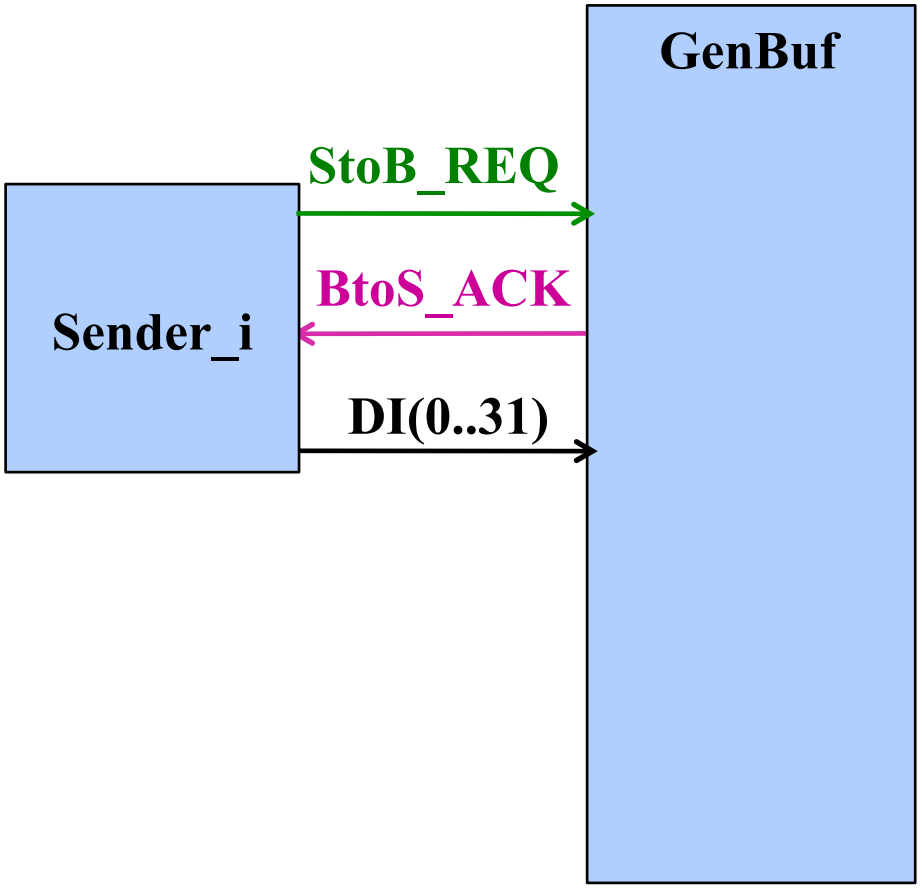
SEREs – Example 2

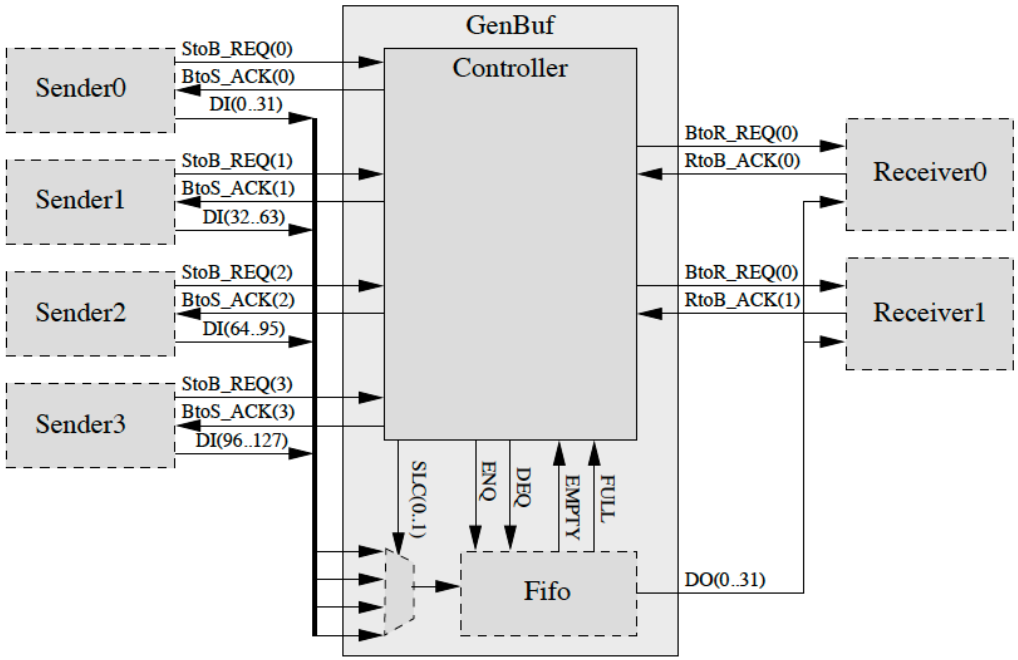
{req; busy [*4]; gnt}

signal busy holds 4 times



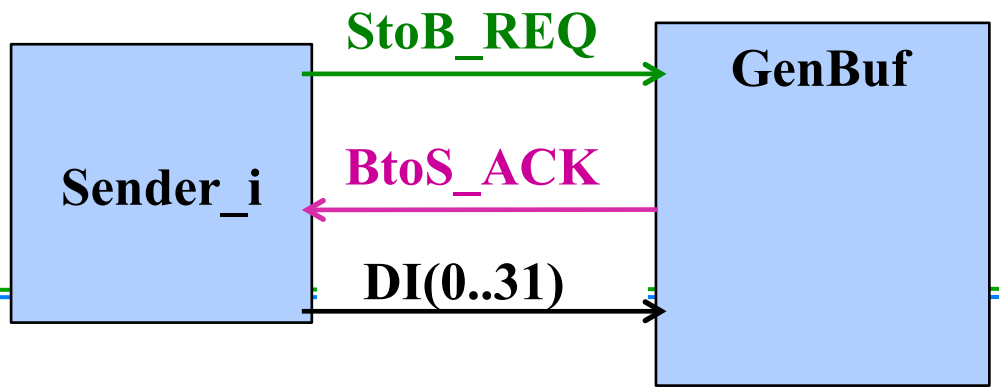
Temporal operators: GenBuf Example





Genbuf specification: the 4-phase handshake protocol with the senders

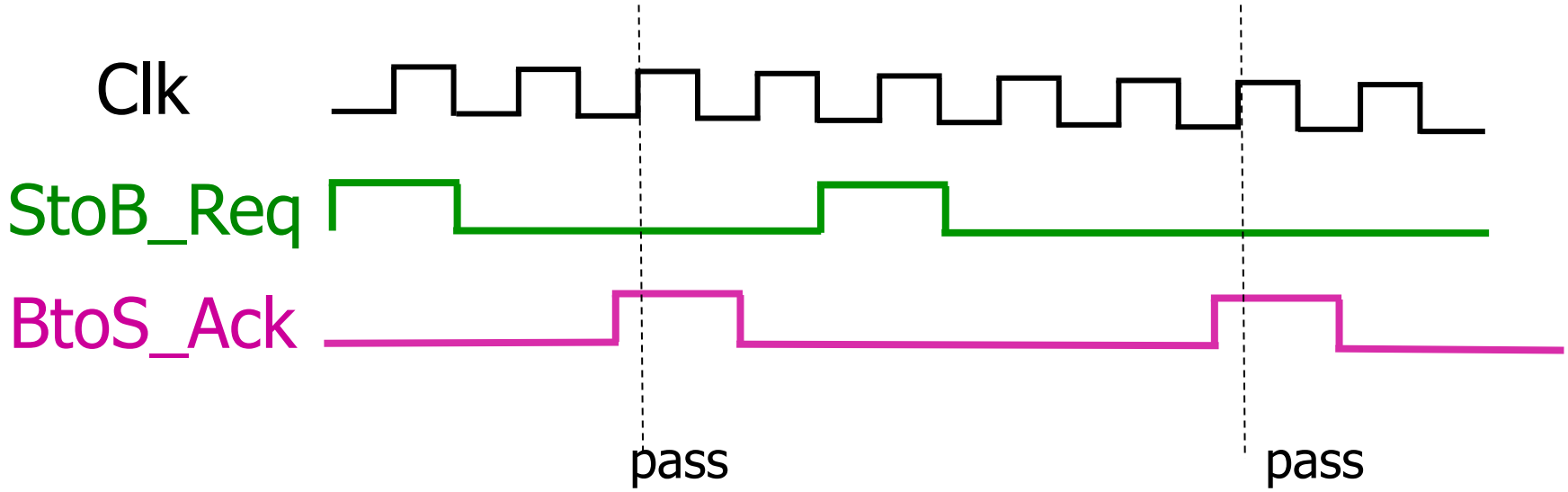
- Sender i initiates a transfer by raising StoB_REQ(i)
  One cycle later, the sender puts the data on its data bus DI
- To service the sender, GenBuf reads the data from the data bus and raises BtoS_ACK(i).
- One cycle after, the sender should lower StoB_REQ(i).
  From this point onwards, the data on the data bus is considered invalid.
- The end of the transaction is marked by GenBuf lowering BtoS_ACK(i).
- Note: GenBuf may hold BtoS_ACK(i) high for several cycles before lowering it.
- A new transaction may begin one cycle after BtoS_ACK(i) has become low.



A request from a sender is always acknowledged

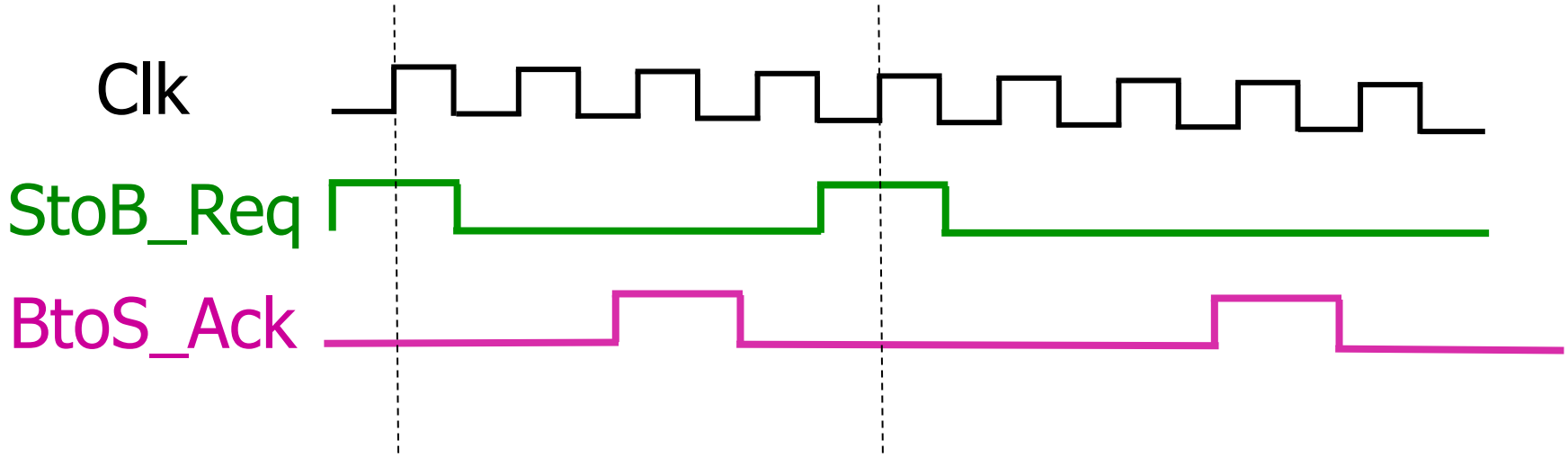
default clock is rising_edge (Clk);

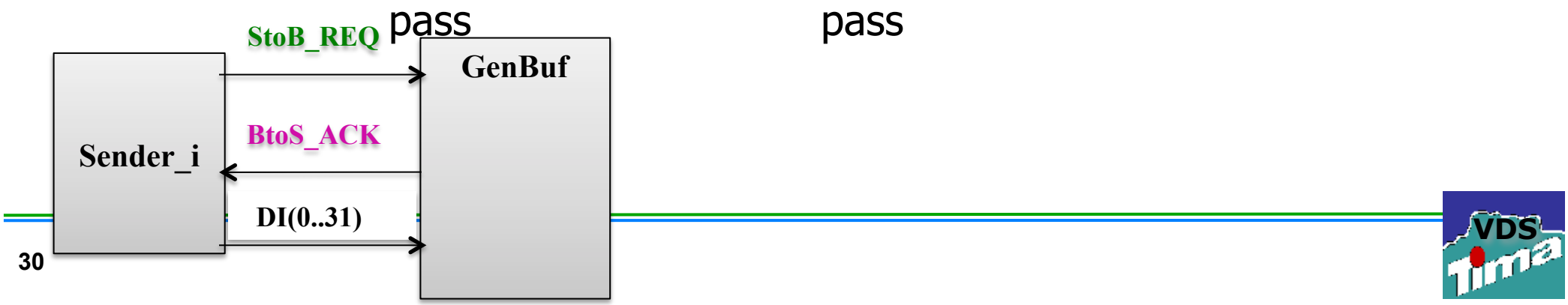
forall i in {0:3} : always (StoB_REQ(i) -> eventually! BtoS_ACK(i))



A request may only be raised when acknowledge is down

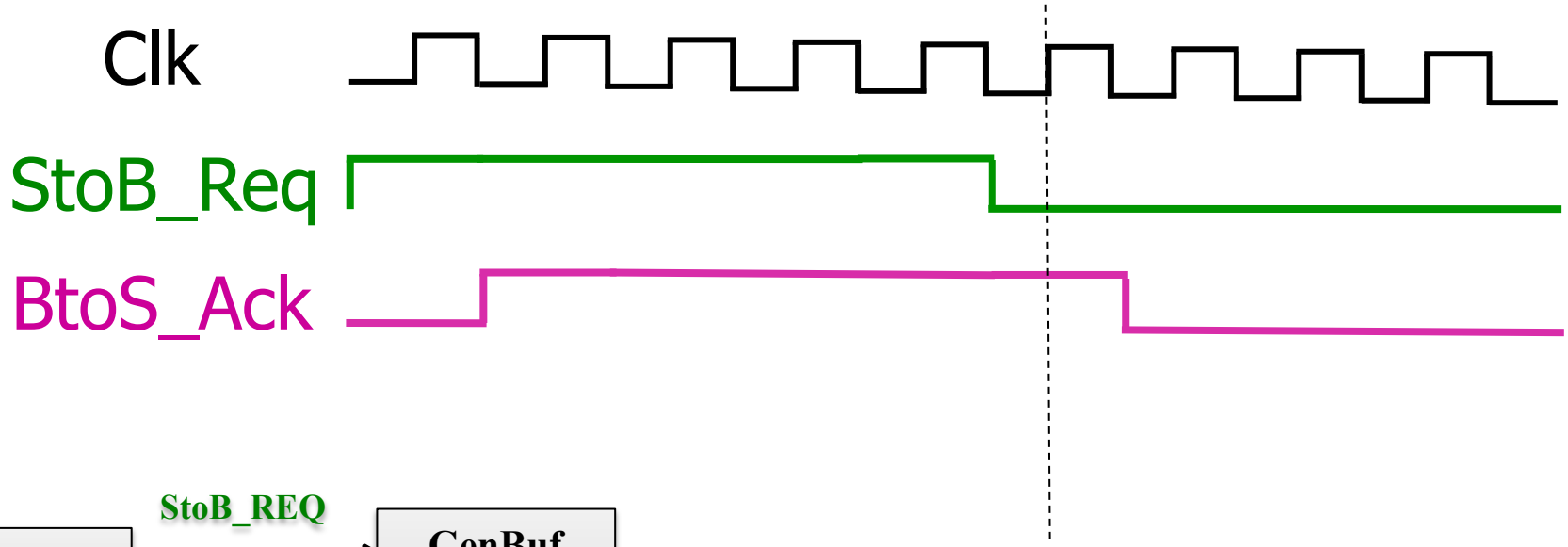
```
default clock is rising_edge (Clk);
forall i in {0:3} : always (rose(StoB_REQ(i) -> not BtoS_ACK(i);
```

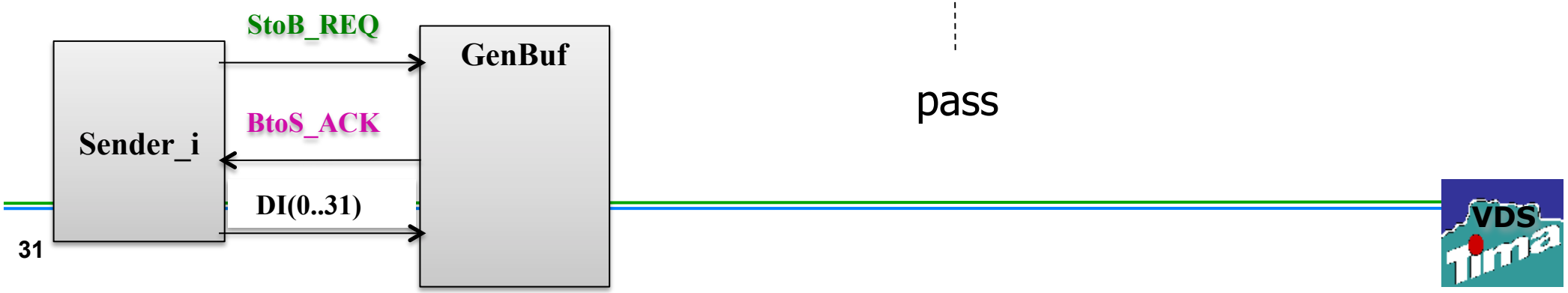




An acknowledge is not deasserted unless the sender deasserts its request first

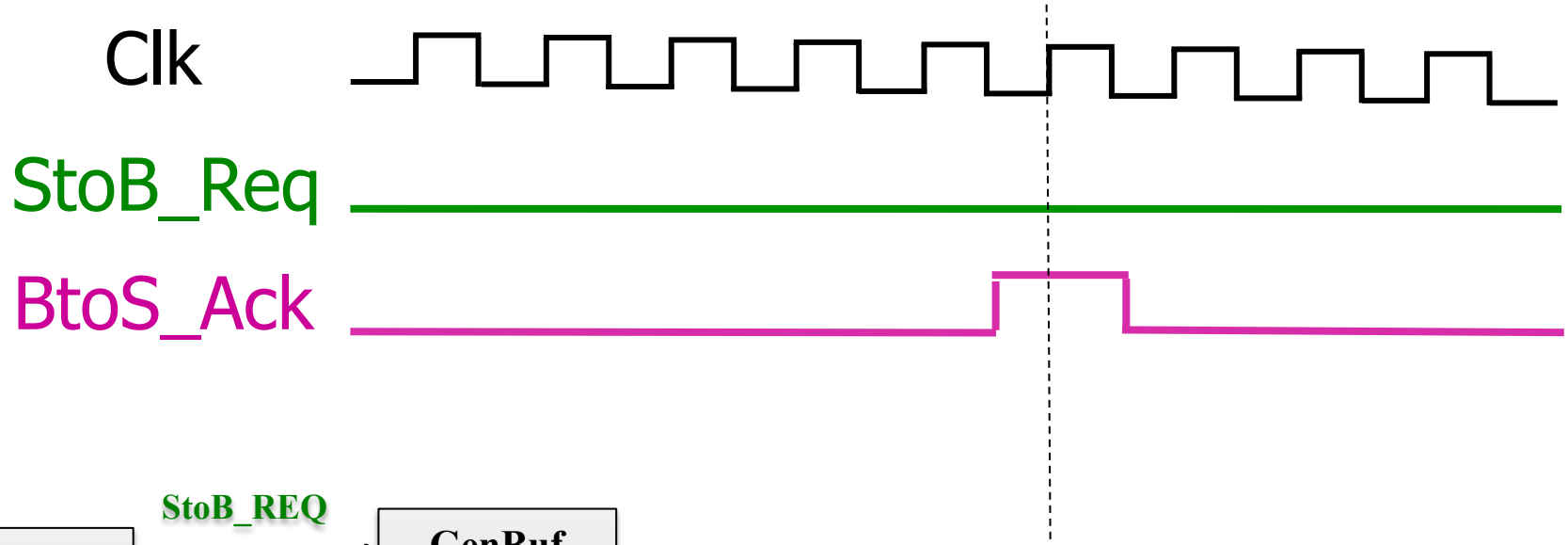
forall i in {0:3} : always ((BtoS_ACK(i) and StoB_REQ(i)) → next! BtoS_ACK(i))





There is no acknowledge if there is not a request

forall i in {0:3} : always ((not BtoS_ACK(i) and (not StoB_REQ(i) -> next! (not BtoS_ACK(i))



Complete specification of the 4-phase protocol

forall i in {0:3} :

always ((BtoS_ACK(i) and StoB_REQ(i)) -> next! BtoS_ACK(i))

always ((not BtoS_ACK(i) and StoB_REQ(i)) -> next! StoB_REQ(i))

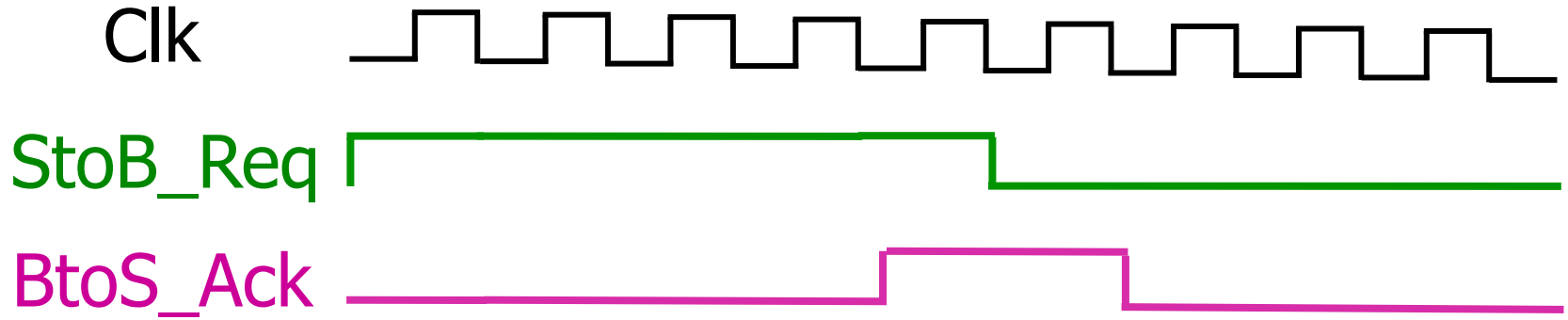
always (StoB_REQ(i) -> eventually! BtoS_ACK(i))

always (not StoB_REQ(i) -> eventually! not BtoS_ACK(i))

always (rose(StoB_REQ(i) -> not BtoS_ACK(i))

always (BtoS_ACK(i) -> next! not StoB_REQ(i))

always ((not BtoS_ACK(i) and (not StoB_REQ(i) -> next! (not BtoS_ACK(i))

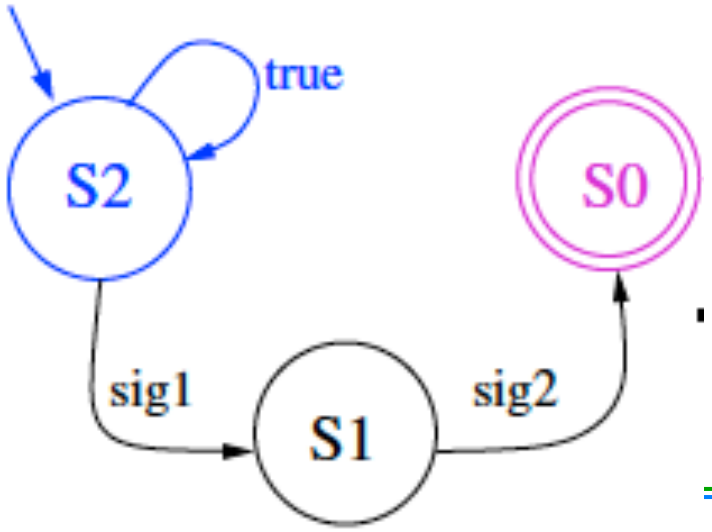


Outline

- A simple running example
- Brief introduction to Property Specification Languages
- **Proven correct hardware verification IP's from PSL**
- Automatic Hardware Generation from PSL: problems and partial solutions
- Conclusion

Automata-theoretic Approach

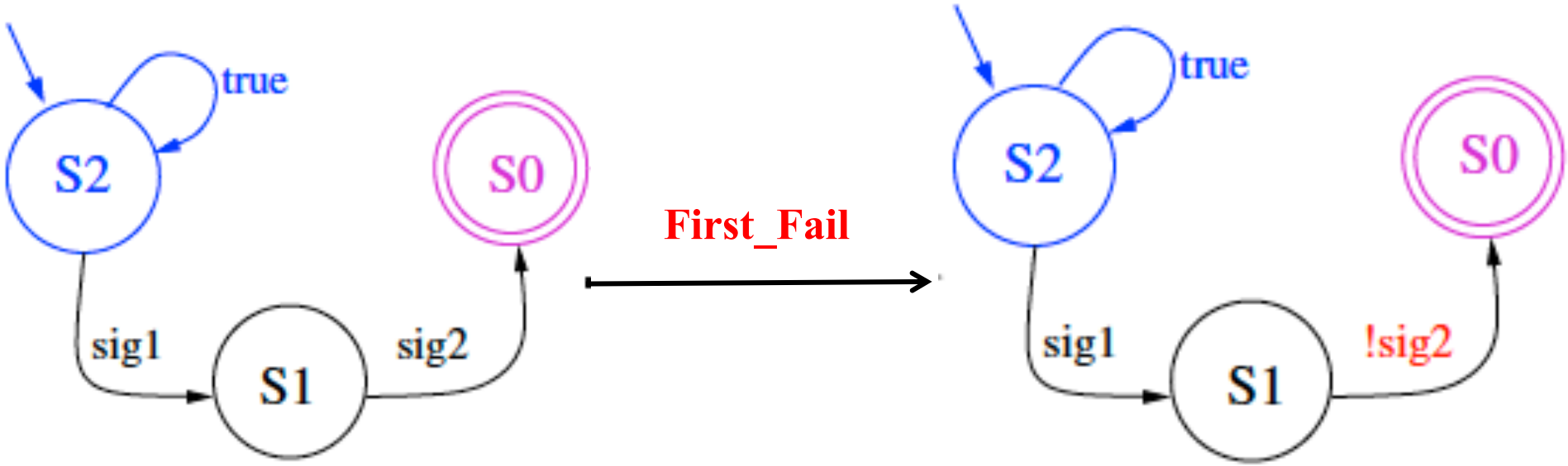
- The set of traces that satisfy a property P is interpreted as the words of an infinite language
- Build a non-deterministic automaton that recognizes all the traces that satisfy the property
- Property P2 : always (sig1 -> next sig2)



Automata-based method in MBAC (Boulé 08)

- **Transform recognizing automaton into failure automaton (may modify the automaton structure)**
- **Efficiently produce synthesizable RTL checker.**
- **Most efficient technique for SERE's**

Property P2 : always (sig1 -> next sig2)

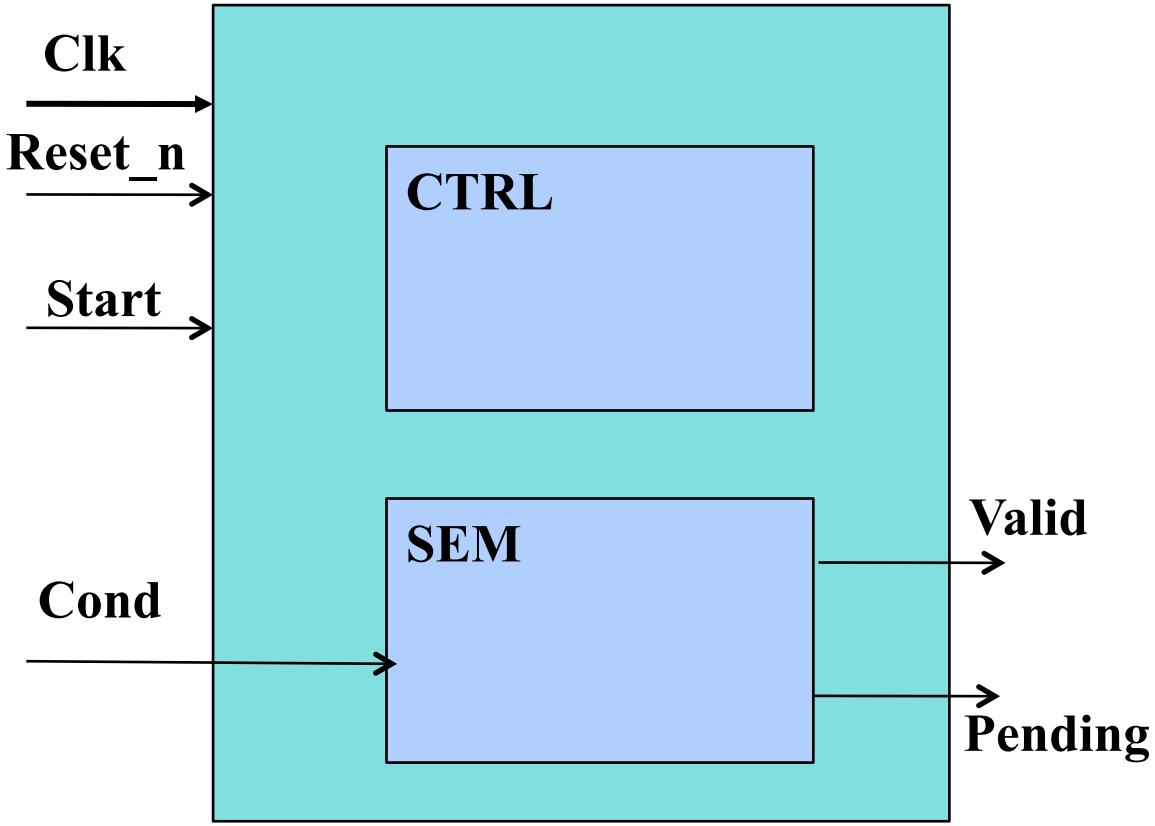


Principles of the construction in HORUS

- A monitor for Property P is a synchronous design detecting dynamically all the violations of P
- A generator for Property P is a synchronous design producing sequences of signals complying with P
- Both types of IP's based on
  - A library of primitive modules
  - An interconnection scheme directed by the syntax tree of P

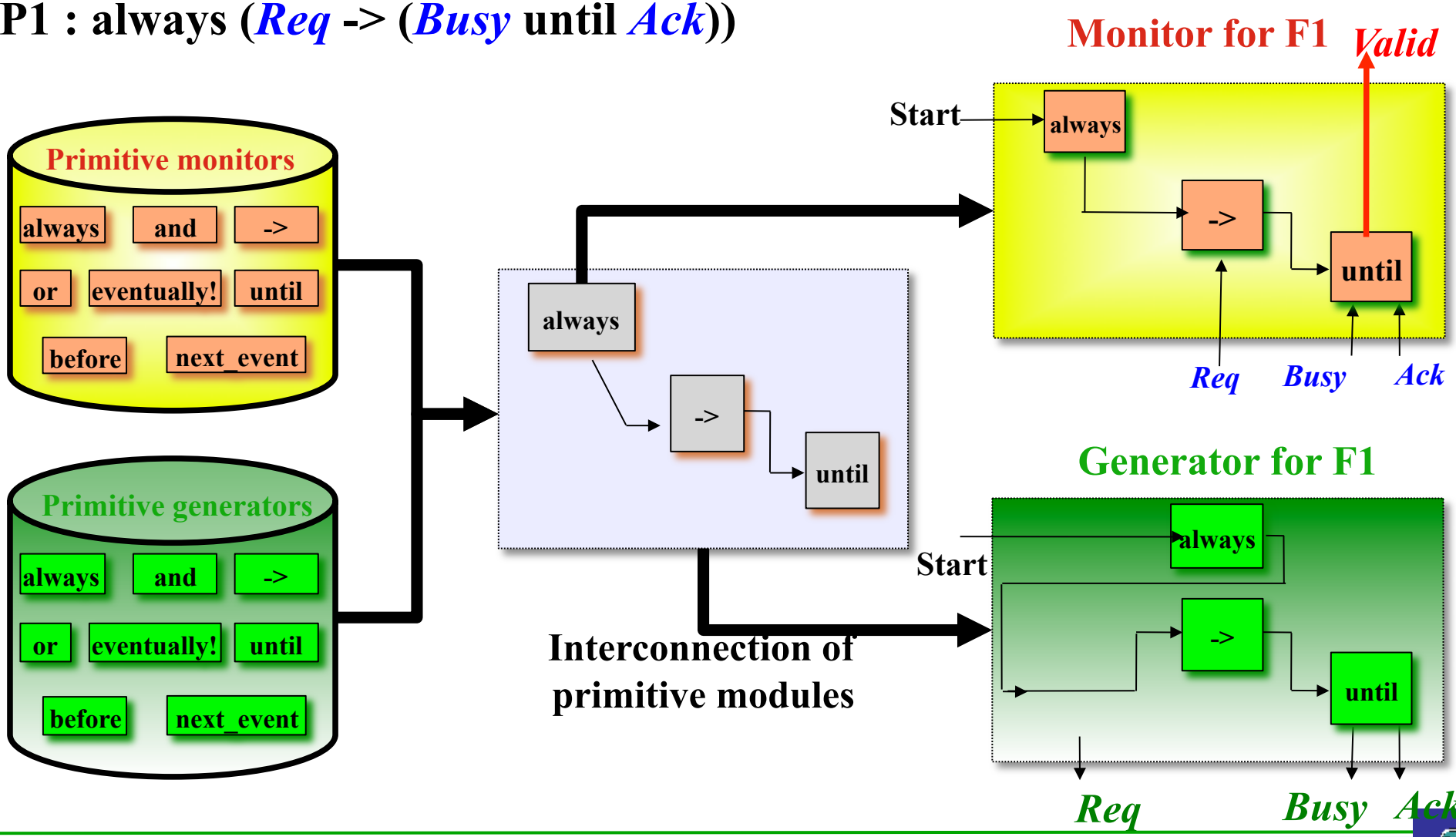
Most efficient technique for temporal operators

Primitive Modules



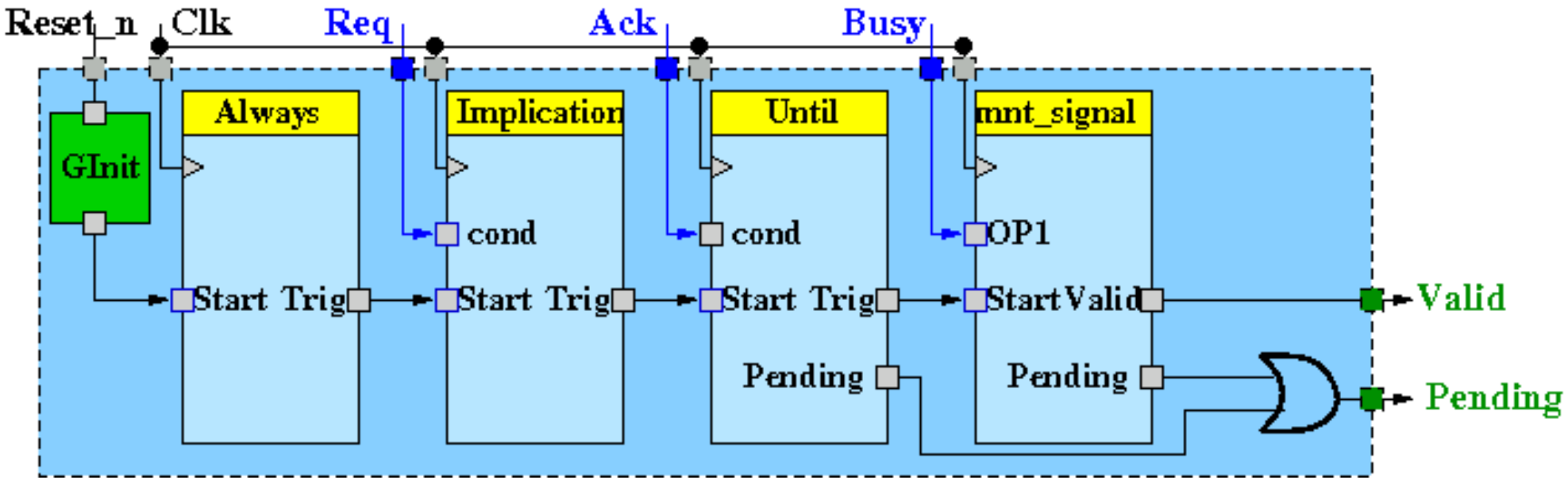
Verification IP Synthesis

P1 : always (*Req* -> (*Busy* until *Ack*))

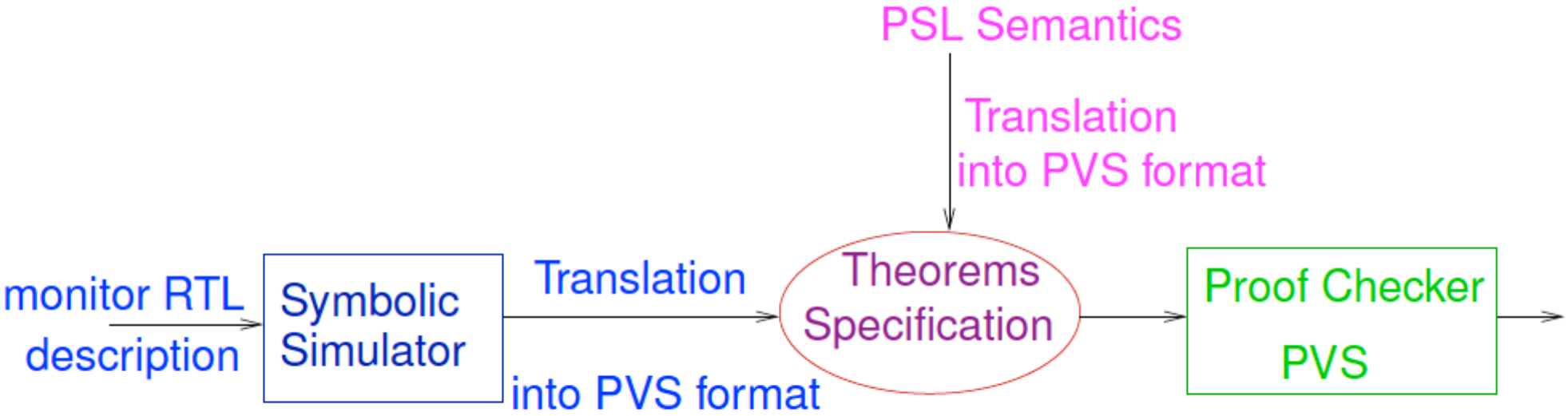


Interconnect of Primitive Modules

assert always (Req -> (Busy until! Ack))



Proof of correctness of the library modules



PSL semantics modeled in PVS: signal

- **PSL semantics are defined on traces.**
- **Signals are seen as functions over discrete time**
- **Semantic definition of a signal monitoring**

```
Valid_Signal (t:Nat): Boolean =
( if t=0 then True
  elsif  Reset_N = False then True
  elsif Start (t-1) = True then Expr (t-1)
  else True
  end if
)
```

PSL semantics modeled in PVS: operator

PSL semantics are defined on trace.
Example: semantics of $\mathrm{Next}(e)$

e	0	0	1	1	0	1	1
time	0	1	2	3	4	5	6

$$\mathrm{Sem}_{\mathrm{Next}}(e, 0, 6) = False$$
$$\mathrm{Sem}_{\mathrm{Next}}(e, 4, 6) = True$$

- A mapping:

$$\begin{aligned} \mathrm{Sem}_{\mathrm{Next}} : (PSL \times \mathbb{N} \times \mathbb{N}) &\rightarrow \mathbb{B} \\ e, t_0, T &\mapsto T - t_0 \geq 1 \wedge \mathrm{Sem}(e, t_0 + 1, T) \end{aligned}$$

- Translation into PVS

```
SemNext(e:PSL,t0:nat,T:upfrom(t0)):boolean=
   T-t0>=1 AND  Sem(e,t0+1,T)
```

Theorem to be proved

Formally the equivalence modeling for any property is given by the expresion:

$$\forall P, \forall [t_0, T], \text{Hypothesis}(P, t_0, T) \implies$$
$$\text{SemFormula}(P, t_0, T) \iff$$
$$(\text{MonitorFormula}(P, t_0, T) \wedge \neg \mathit{Pending}(T+1))$$

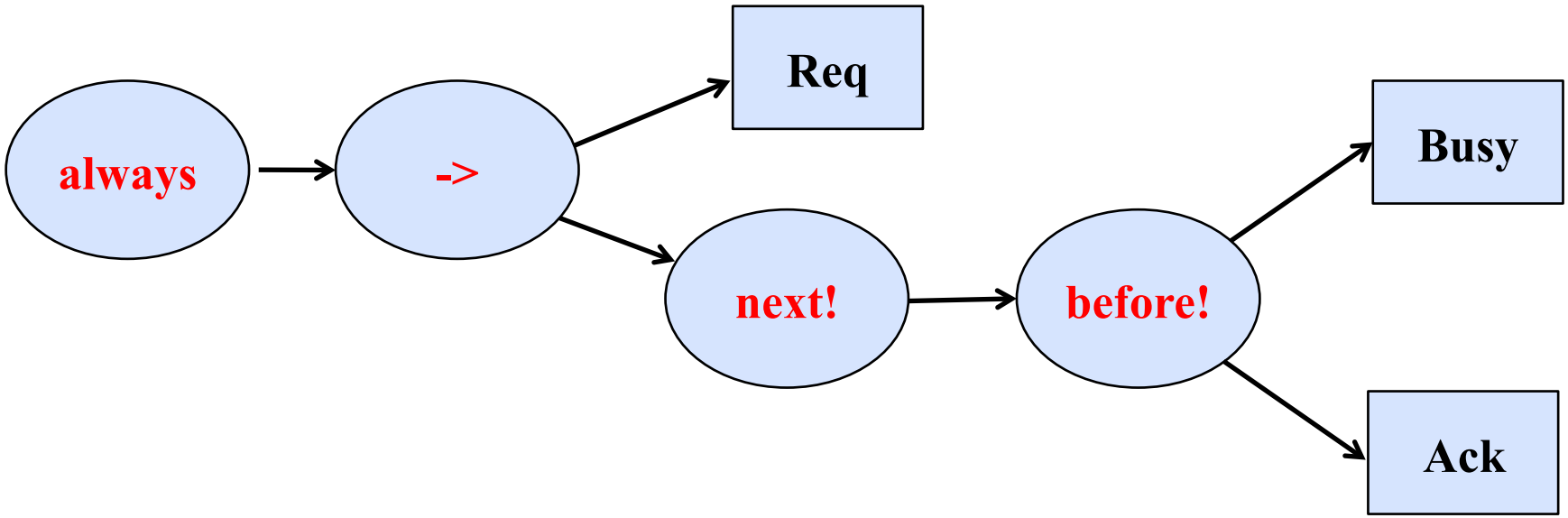
$$\text{SemFormula}(P, t_0, T) = \forall t \in [t_0, T], \text{Start}(t) \implies \text{Sem}(P, t, T)$$

$$\text{MonitorFormula}(P, t_0, T) = \forall t \in [t_0, T], \neg\,\text{Pending}(t) \implies \text{Valid}(t+1)$$

$\text{Hypothesis}(P, t_0, T) \rightsquigarrow$ no active reset on $[t_0, T]$

Semantics of a property: induction over the structure

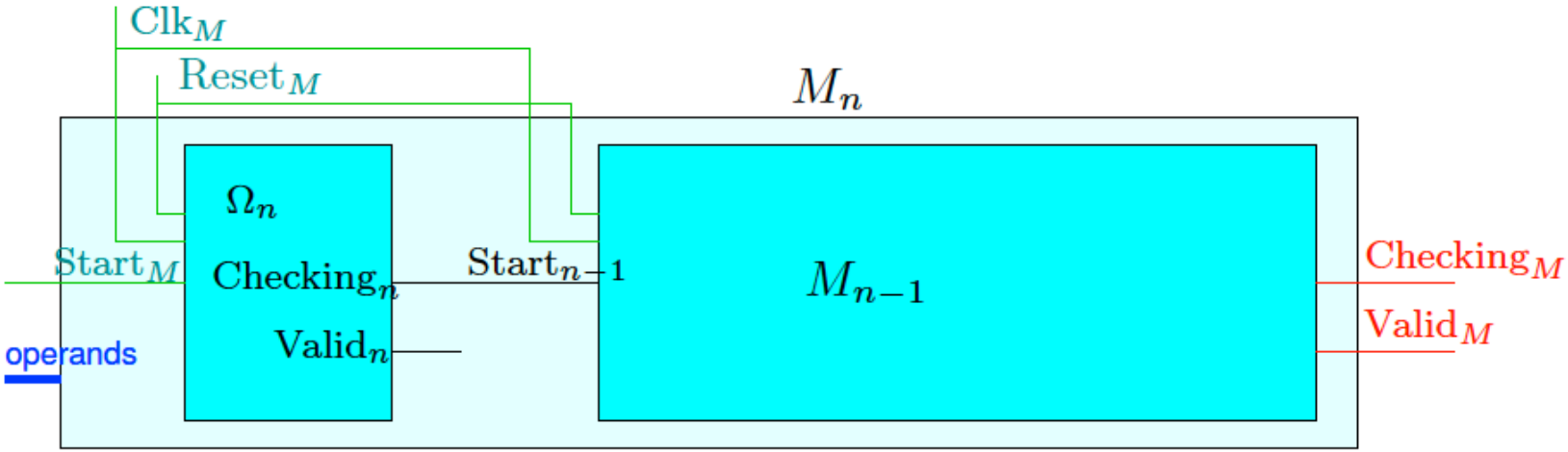
- assert always (Req -> next! (Busy before! Ack))



Proof of the interconnection

By induction on the depth of P.

$P = \Omega_n \ldots \Omega_1 \mathrm{op}_1 \ldots \mathrm{op}_n$



- **Base case**: equivalence proof for each elementary monitor in the library
- **Induction case**: proof of the construction method (based on substitution and generalization).

Proof figures

- For each operator: one theorem for the base case, two for the induction step
- 250 lines, 80 typed proof obligations
- From 10 instructions to a thousand instructions
- While proving the library:
  - all weak operators were found correct
  - two strong operators had a bug
- The interconnection method is proven correct

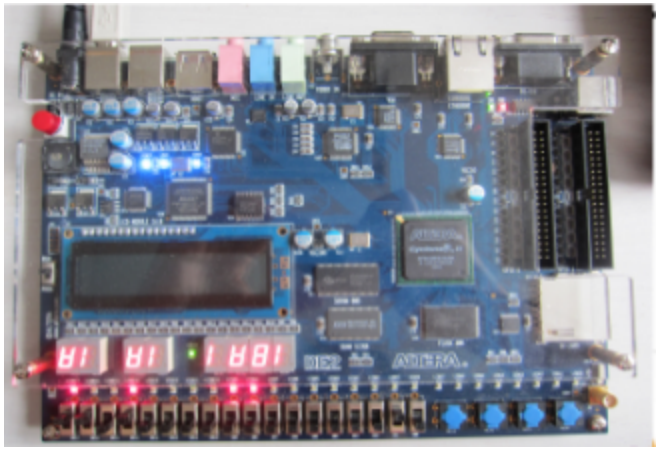
Outline

- A simple running example
- Brief introduction to Property Specification Languages
- Proven correct hardware verification IP's from PSL
- **Automatic Hardware Generation from PSL: problems and partial solutions**
- Conclusion

Considerations

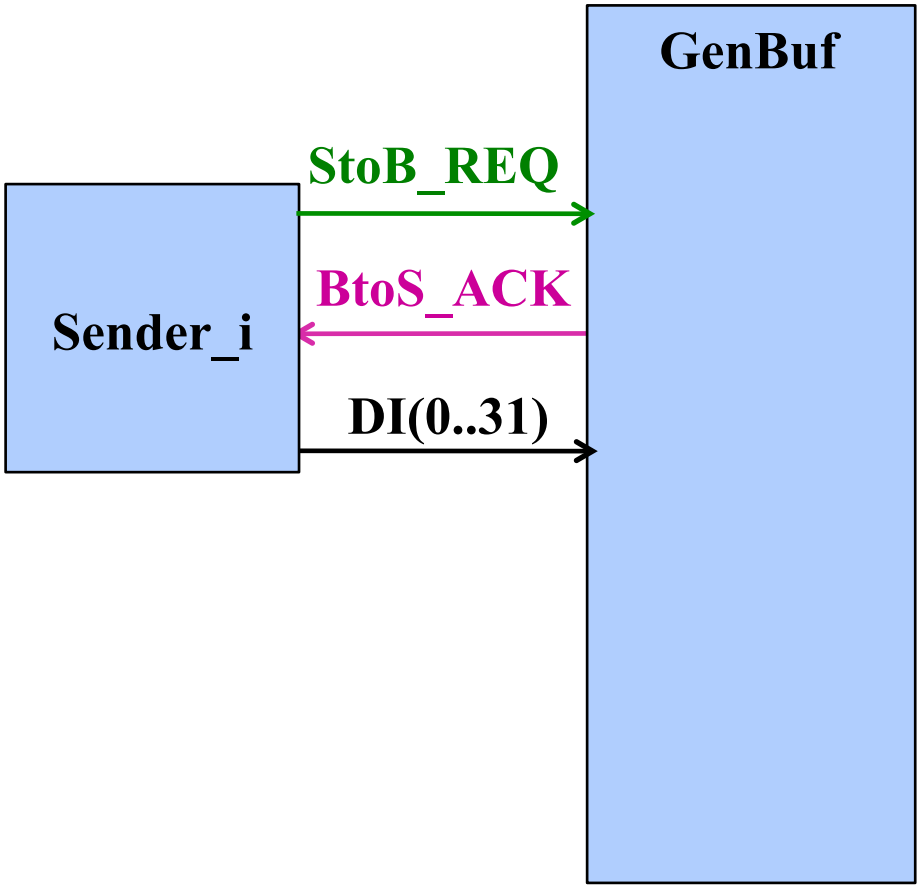
- **Goal: Automatic production of a circuit prototype from PSL specifications**

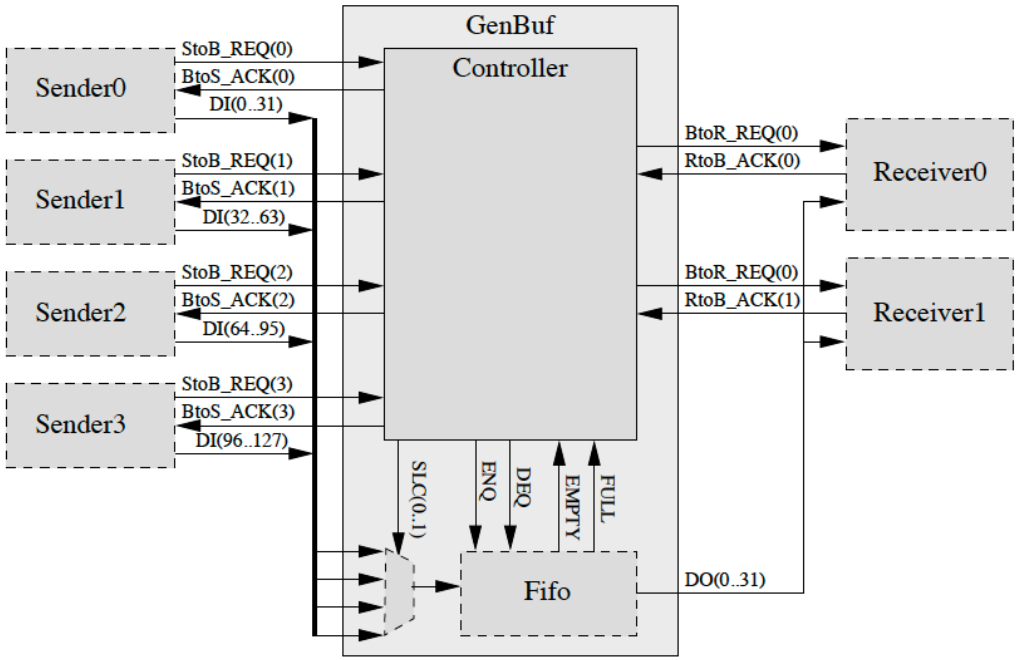
assert always ...
assert always ...



- **Two hypotheses**
  - **Ensuring that the specification is complete (on-going work at Chalmers Univ. and TIMA)**
  - **Ensuring that the specification has no contradiction (RAT, on going work at TIMA)**
- **New look at verification IP's: reactants**

Back to Genbuf: generation of Controller





Back to the 4-phase protocol: Phase 1: suppress assume

forall i in {0:3} :

assert always ((BtoS_ACK(i) and StoB_REQ(i)) -> next! BtoS_ACK(i))

assume always ((not BtoS_ACK(i) and StoB_REQ(i)) -> next! StoB_REQ(i))

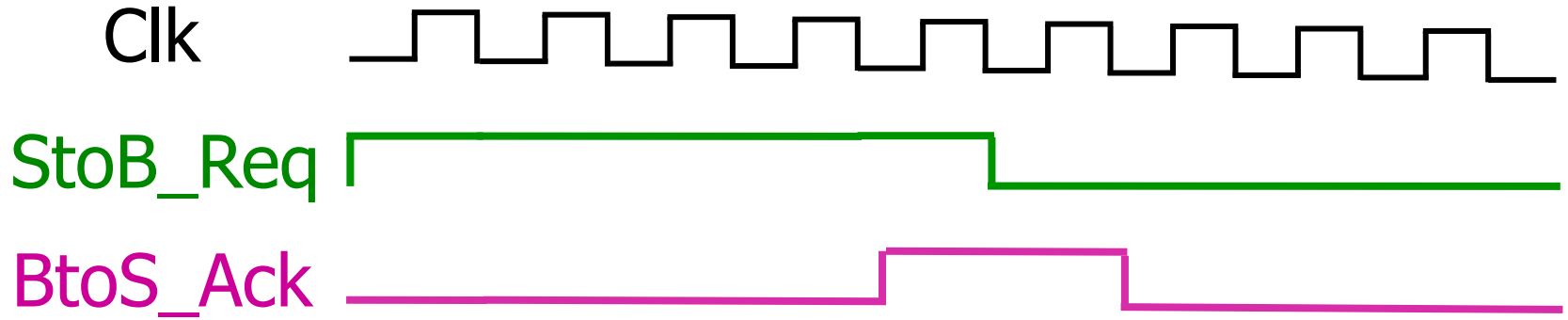
assert always (StoB_REQ(i) -> eventually! BtoS_ACK(i))

assert always (not StoB_REQ(i) -> eventually! not BtoS_ACK(i))

assert always (rose(StoB_REQ(i) -> not BtoS_ACK(i))

assume always (BtoS_ACK(i) -> next! not StoB_REQ(i))

assert always ((not BtoS_ACK(i) and (not StoB_REQ(i) -> next! (not BtoS_ACK(i))



Phase 2: top level annotation using port direction

forall i in {0:3} :

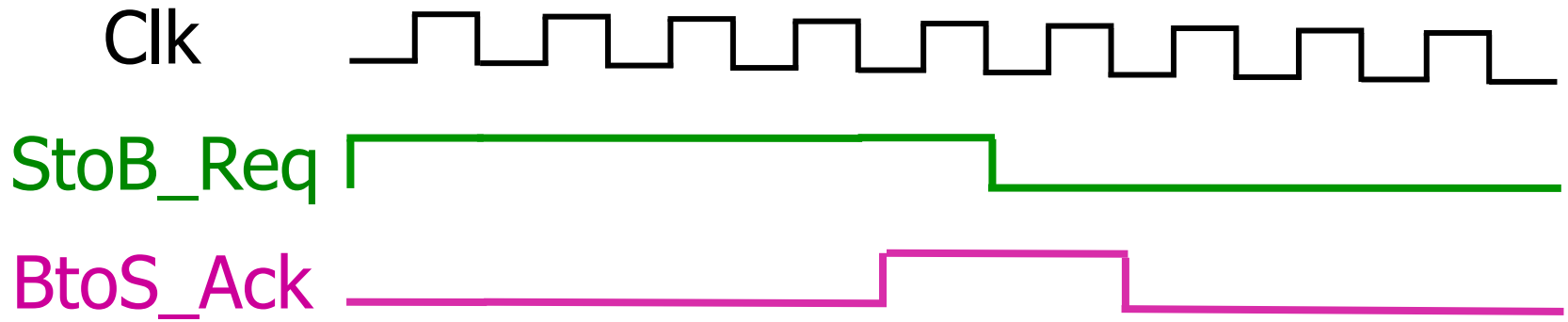
assert always ((BtoS_ACK(i) and StoB_REQ(i)) -> next! BtoS_ACK(i))

assert always (StoB_REQ(i) -> eventually! BtoS_ACK(i))

assert always (not StoB_REQ(i) -> eventually! not BtoS_ACK(i))

assert always (rose(StoB_REQ(i) -> not BtoS_ACK(i))

assert always ((not BtoS_ACK(i) and (not StoB_REQ(i) -> next! (not BtoS_ACK(i))



Phase 3: annotation at property level

forall i in {0:3} :

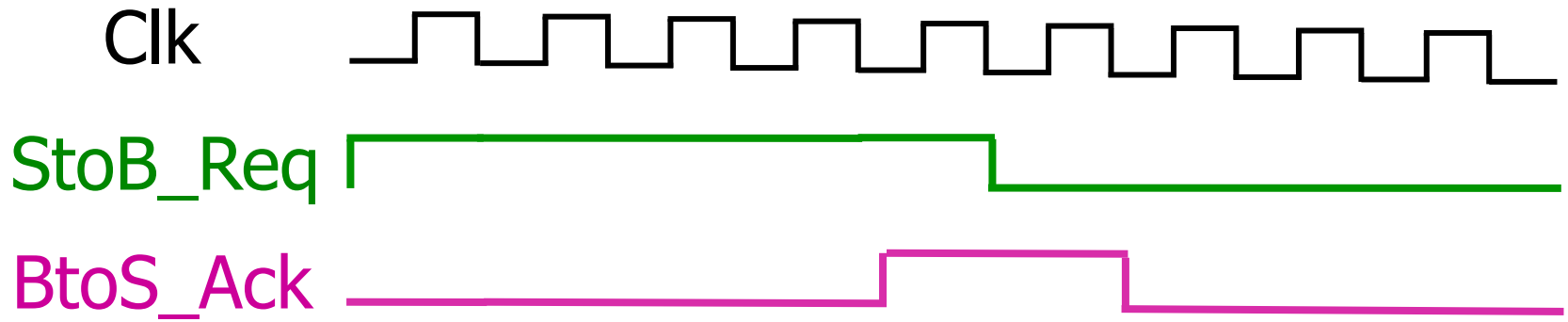
P1: assert always ((BtoS_ACK(i) and StoB_REQ(i)) -> next! BtoS_ACK(i))

P2: assert always (StoB_REQ(i) -> eventually! BtoS_ACK(i))

P3: assert always (not StoB_REQ(i) -> eventually! not BtoS_ACK(i))

P4: assert always (rose(StoB_REQ(i) -> not BtoS_ACK(i))

P5: assert always ((not BtoS_ACK(i) and (not StoB_REQ(i) -> next! (not BtoS_ACK(i))



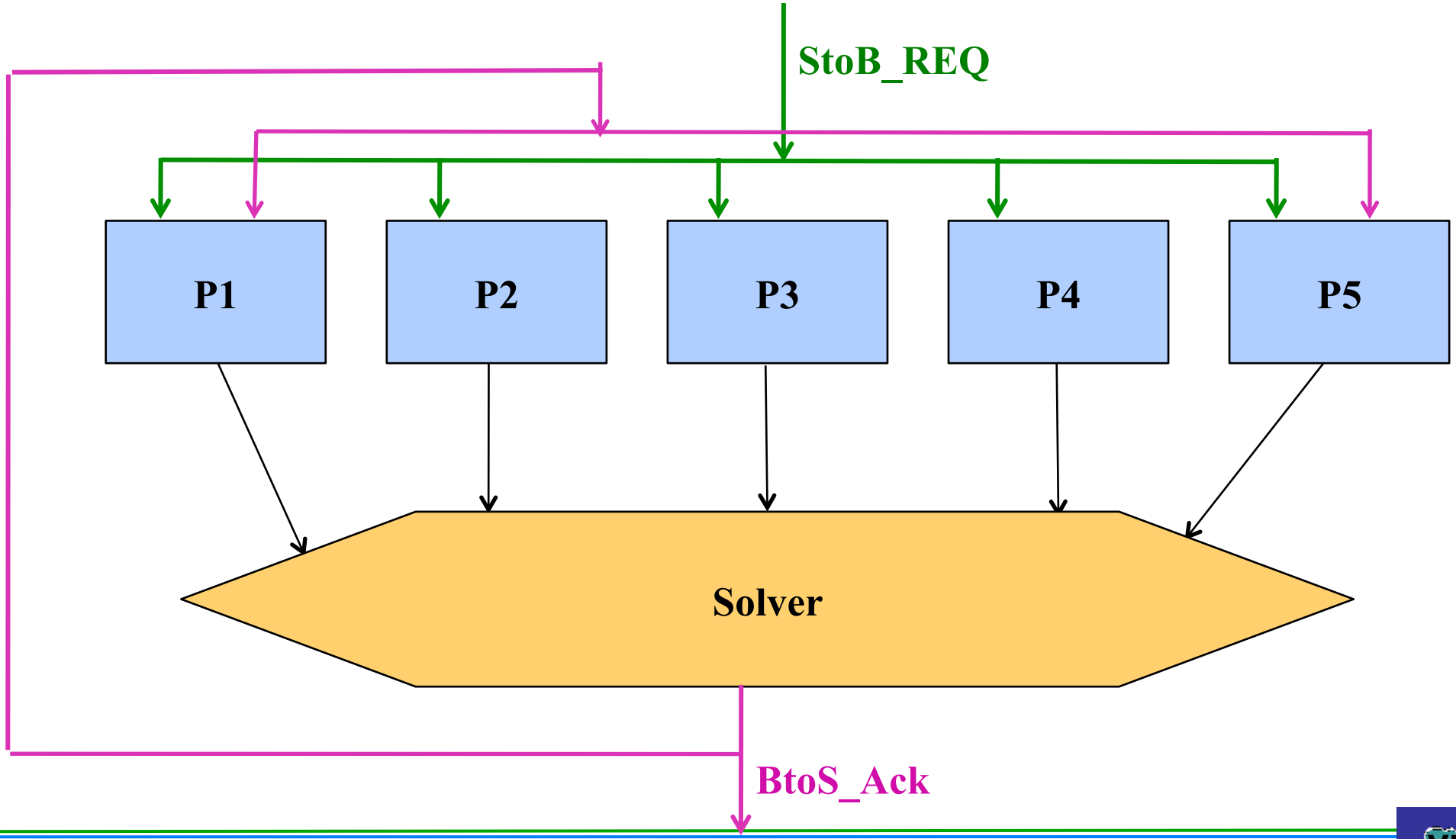
Considerations

P1: assert always ((BtoS_ACK(i) and StoB_REQ(i)) -> next! BtoS_ACK(i))

P2: assert always (StoB_REQ(i) -> eventually! BtoS_ACK(i))

- **Each asserted property constrains at least one signal**
- **In a property, some signals are read, others are generated.**
- **Several properties may name the same signal**
- **Some properties observe the signal, other properties generate the signal**
- **An algorithm has been defined to annotate the signals as IN or OUT for each property, based on a dependency graph**
- **When a signal is generated by 2 or more properties, there is a solver**

Compiled 4-phase controller



Conclusions

- HORUS = Verification IP's for Monitoring: done
- SYNTHORUS = Prototyping from PSL
  - Compilation status
    - FL operators only
    - Generated: Logical signals only
  - On going
    - Checking specification completeness and coherence
    - Optimization of compiled code
    - Definition of a synthesizable PSL subset
  - Future works
    - SERE's
    - Arithmetic signals

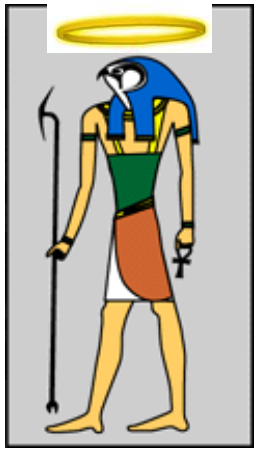
The full story

Màat

Synchronous Monitors

D. Borrione, Miao Liu,
Katell Morin-Allory

SyntHorus



Property based design
RTL

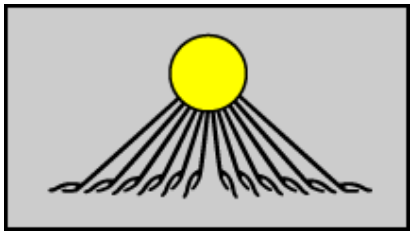
D. Borrione, Negin Javaheri,
Katell Morin-Allory,
Axandre Porcher

Property
refinement

Isis

TLM Monitors

L. Pierre, L. Ferro,
Z. Bel Hadj Amor



Aton

Synchronous Generators

Yann Oddos


Interface
Specification
Debug
Performances
Systems
Embedded
Simulation
NoCs
PSL
SoC
Emulation
Flexibility
Design for test
Yann Oddos - 2009
Synthesis
High-level
Low Power
HDL Description
Formal Verification

Thanks