Formal Verification of Systems-on-Chip – Industrial Practices

Wolfgang Kunz

Department of Electrical & Computer Engineering
University of Kaiserslautern, Germany

kunz@eit.uni-kl.de

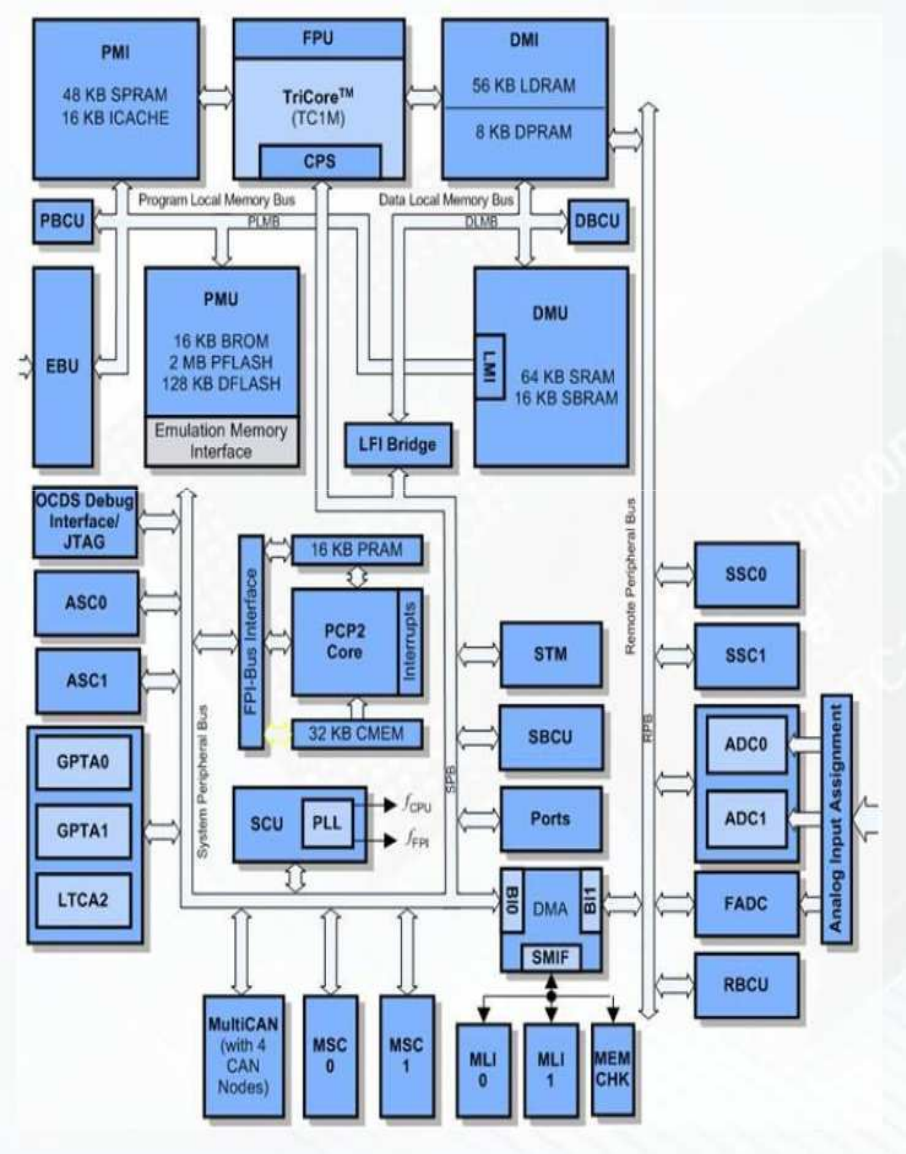
Industrial Experiences

Formal verification of Systems-on-Chip in industrial practice

---

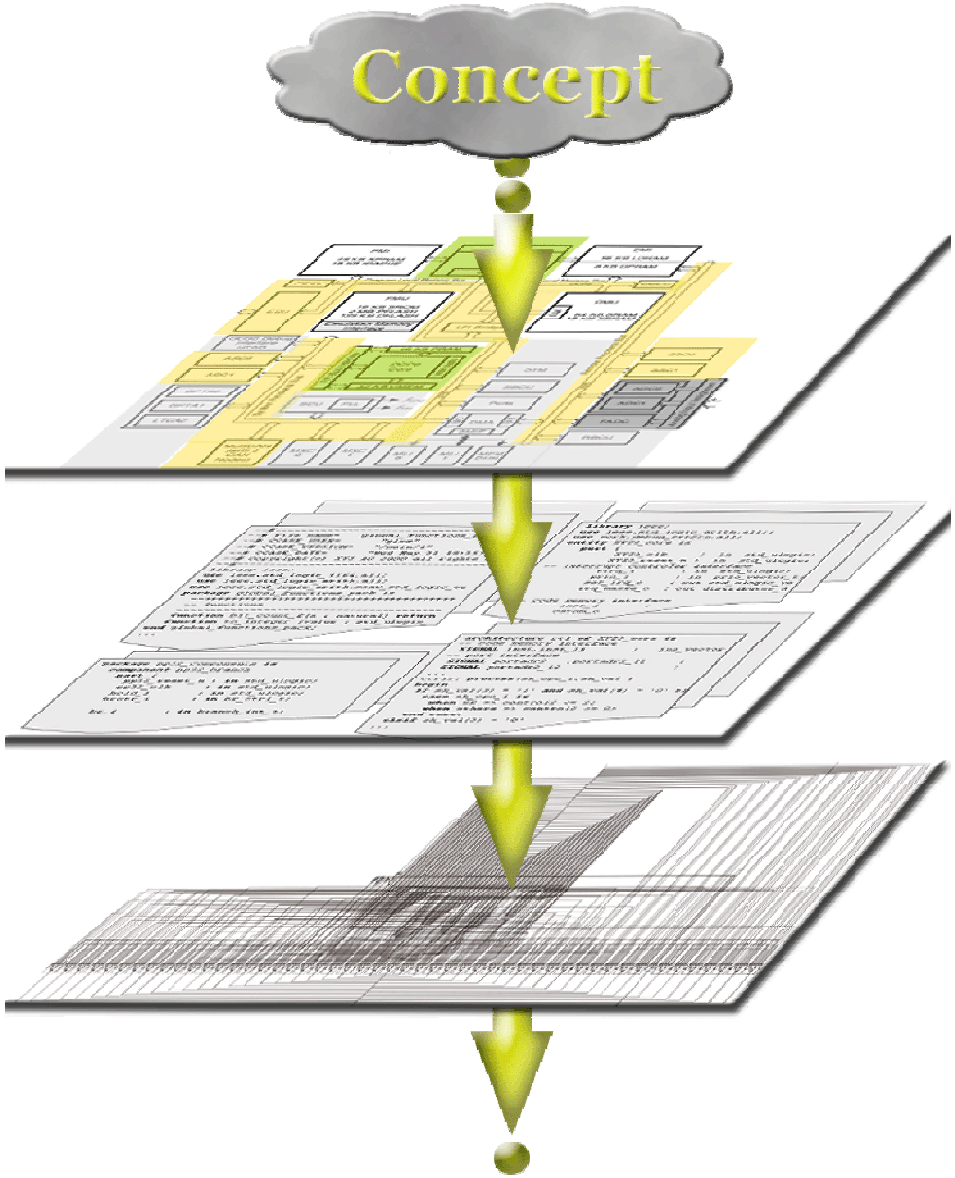
Industrial partners:

Example: SoC for automotive application



- processors
- hardware accelerators
- memories
- I/O controllers
- mixed signal blocks
- communication structures

SoC Design Flow



Early phase

- set up and assess functional prototypes

Architecture

- model and explore architectural choices
- specify modules and communication for target architecture

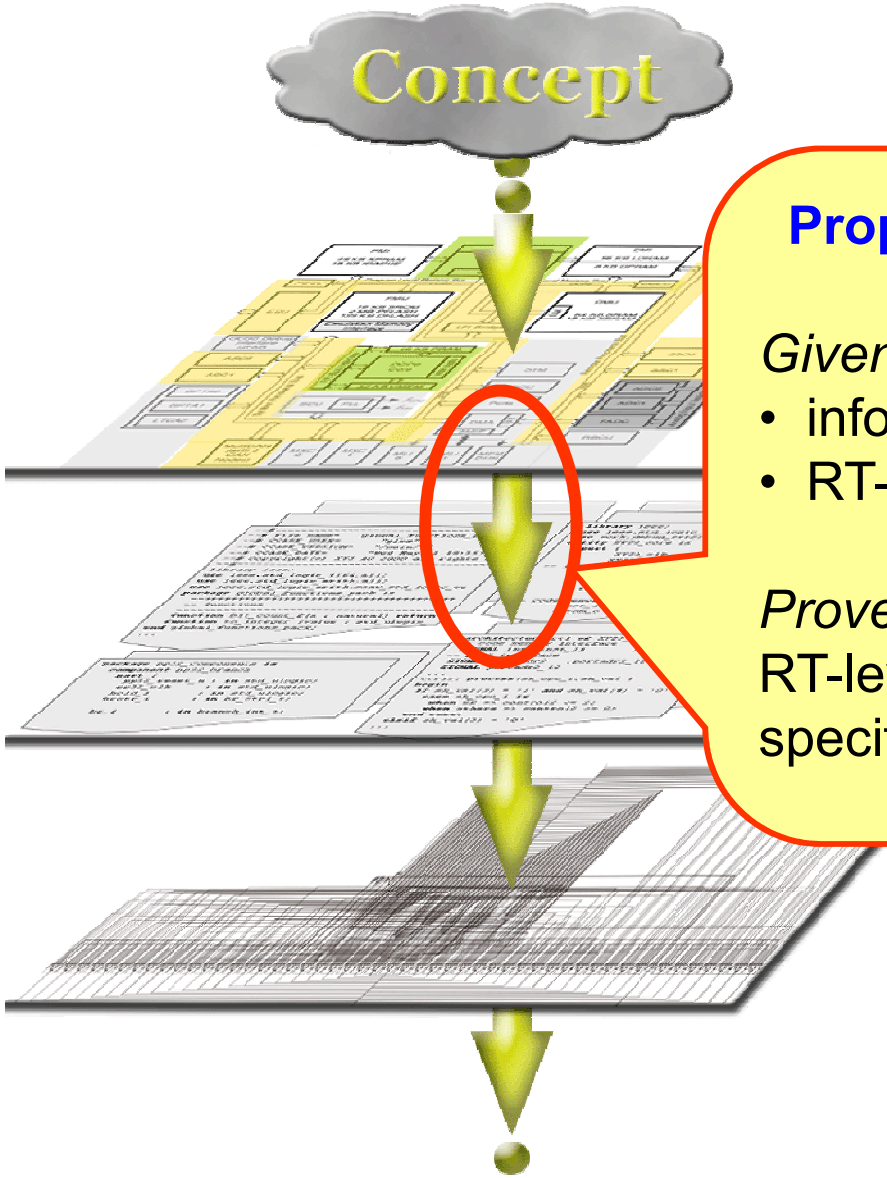
Design (RT)

- Register-Transfer (RT) description of modules
- system integration, communication structures

Implementation

- Synthesis and optimization
- test preparation

SoC Design Flow



Early phase

- set up and assess functional

Property Checking

Given:

- informal specification
- RT-level circuit description

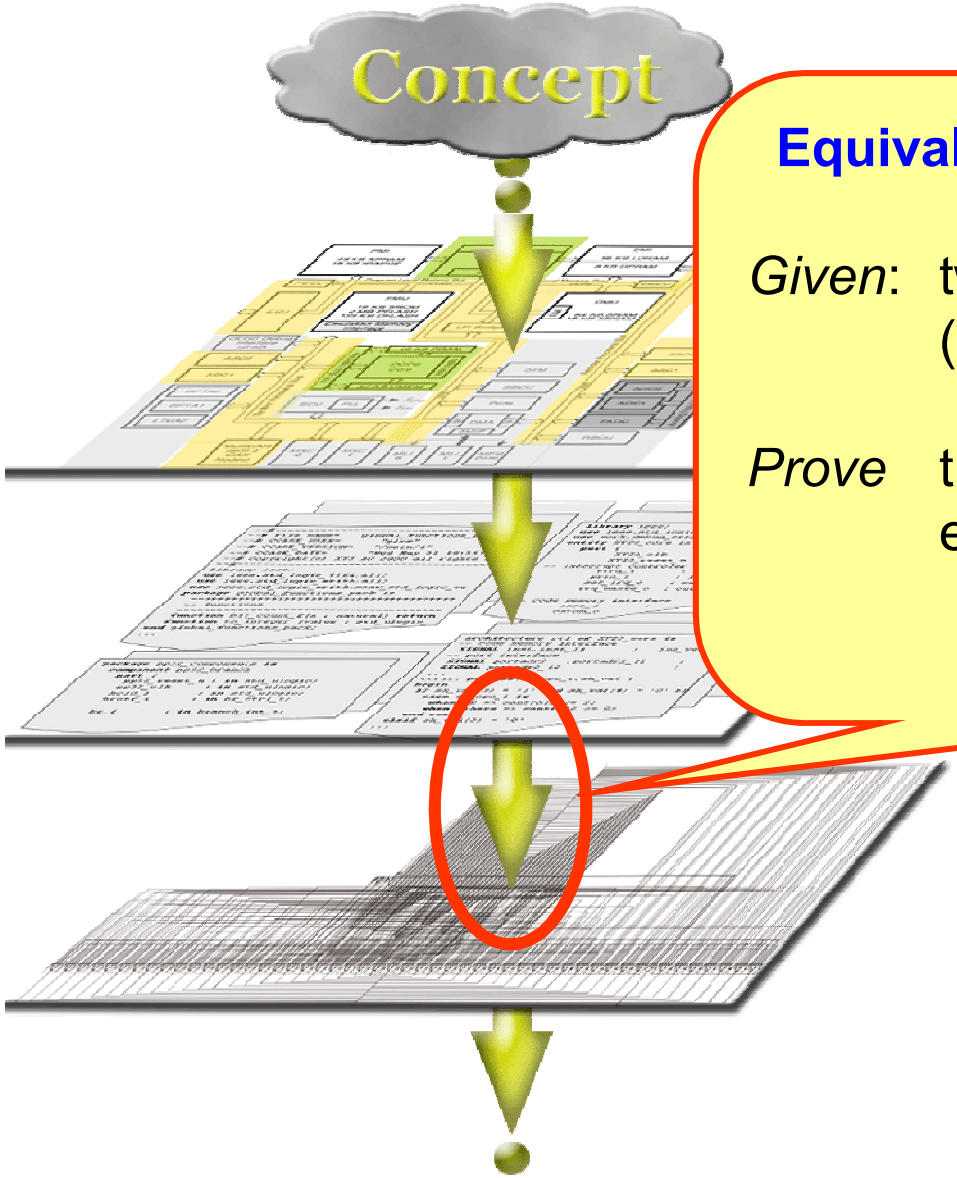
Prove (by checking properties) that the RT-level design description fulfills the specification

structures

Implementation

- Synthesis and optimization
- test preparation

SoC Design Flow



Early phase

Equivalence Checking

Given: two design descriptions (e.g. 1x RTL, 1x Gatelevel)

Prove that both designs are functionally equivalent

of modules

- system integration, communication structures

Implementation

- Synthesis and optimization
- test preparation

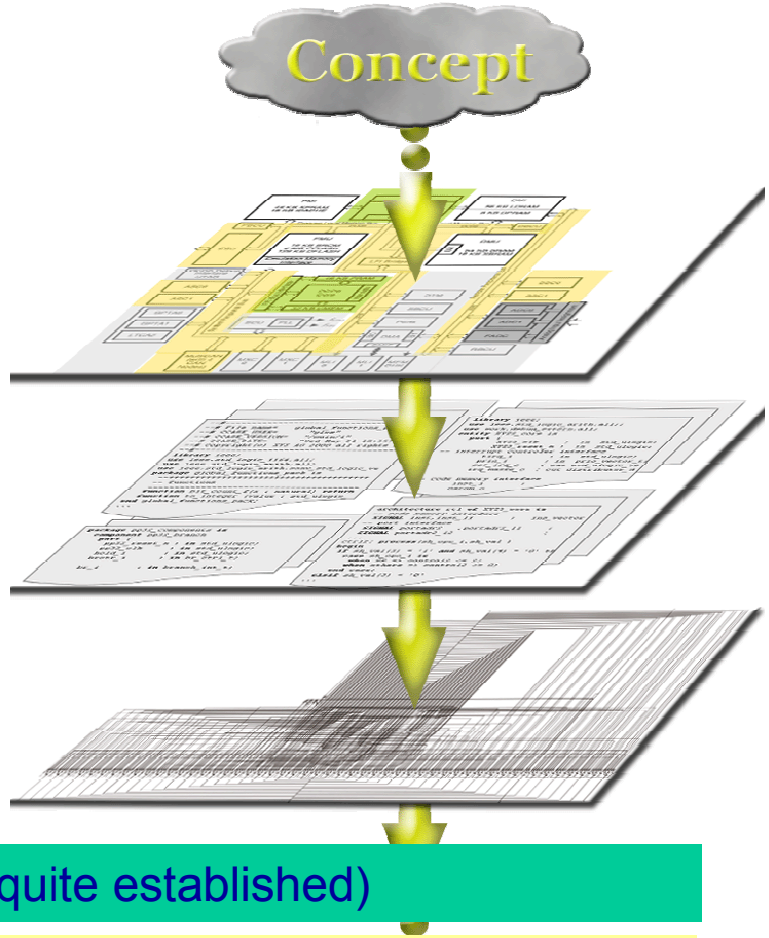
Verification Tasks

The hot spot for property checking

Given:

informal specification of modules and communication between modules (protocols)

Implementation at the register-transfer (RT) level in Verilog or VHDL (hardware description languages)

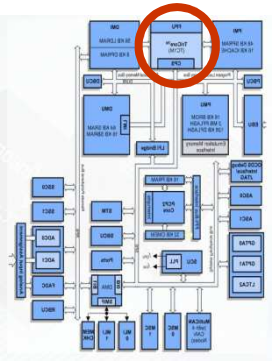


Approach:

Verify each module individually	(FV quite established)
Verify interfaces between modules	(FV feasible but more advanced)
Verify global behavior of entire chip	(not yet)

RT-level module verification

A typical property for RT-level module verification:



$$AG(a \rightarrow c)$$

---

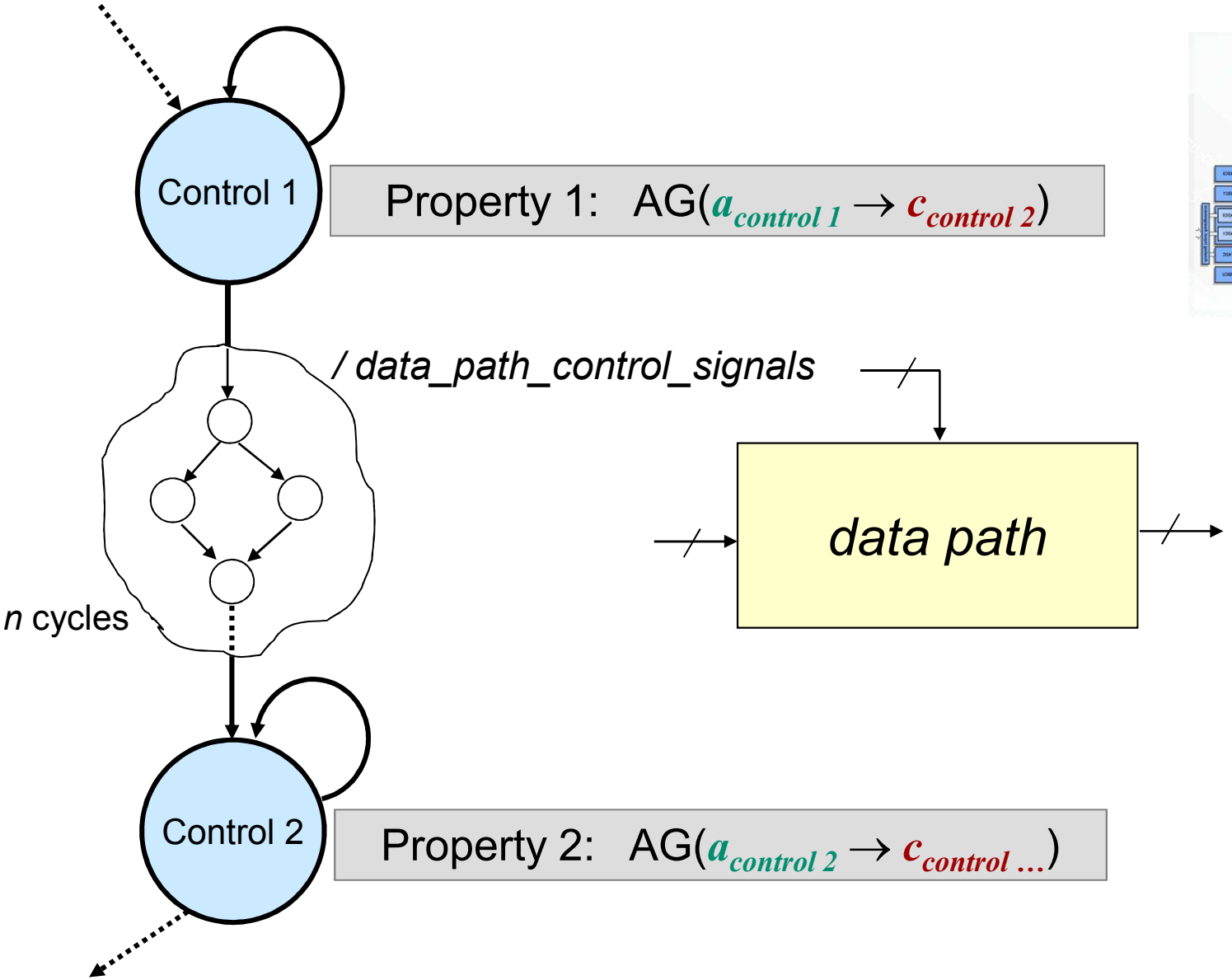
a : assumptions

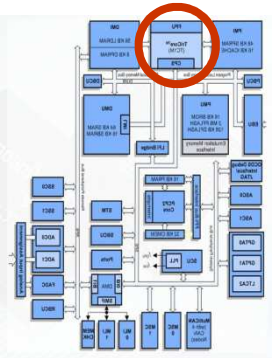
- module is in some control state *V*
- certain inputs *X* occur

c : commitments

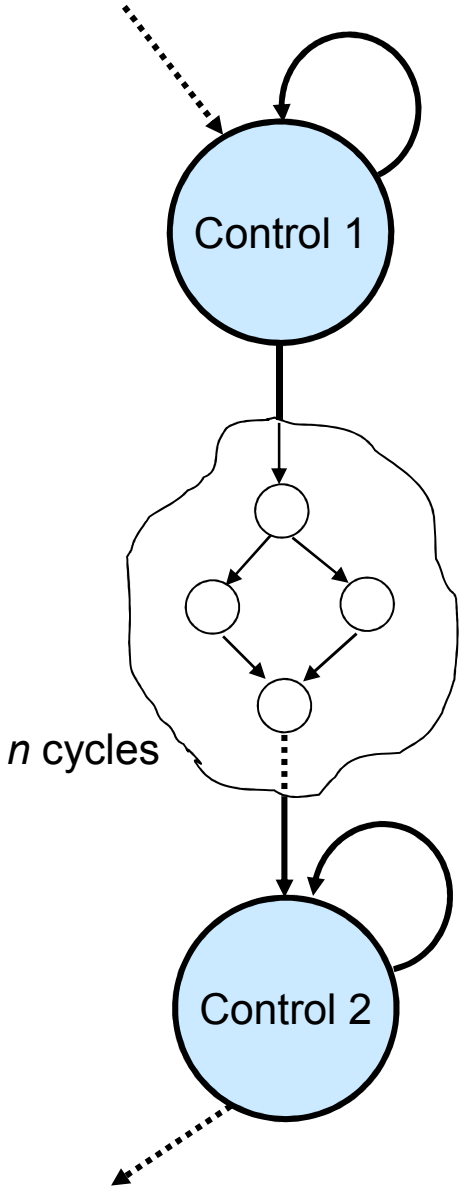
- module goes into certain control state *V'*
- certain outputs *Y* occur

RT-level module verification: operation by operation





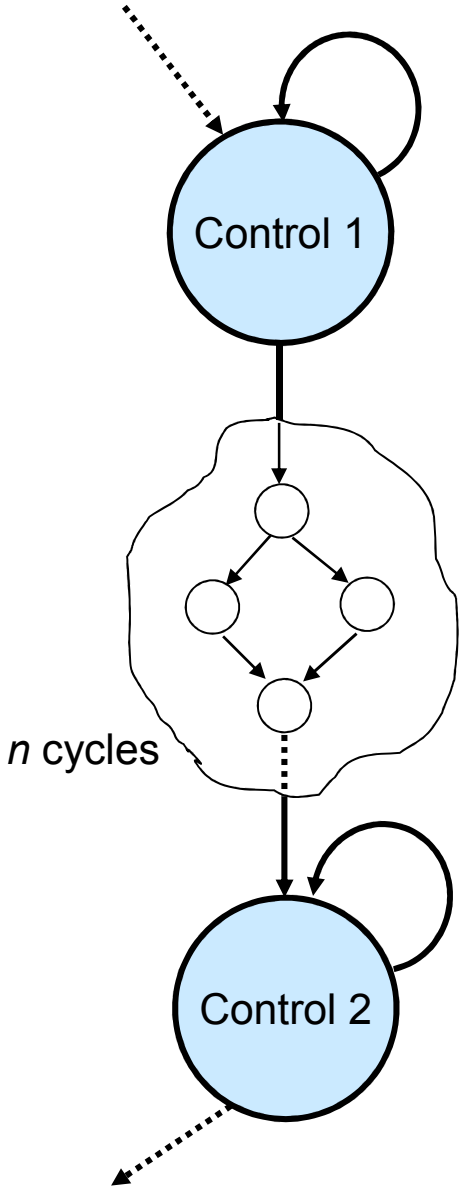
RT-level module verification: operation by operation



Typical methodology for Property Checking of SoC modules:

- Adopt an operational view of the design
- Each operation can be associated with certain “important control states” in which the operation starts and ends
- Specify a set of properties for every operation, i.e., for every important control state
- Verify the module *operation by operation* by moving along the important control states of the design
- The module is verified when every operation has been covered by a set of properties

RT-level module verification



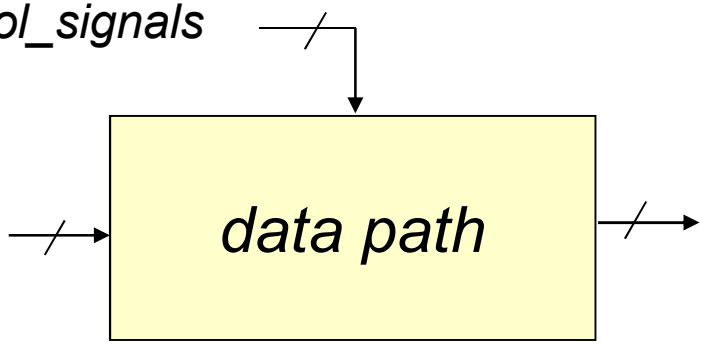
$AG(a_{control\ 1} \rightarrow c_{control\ 2})$

$a_{control\ 1}$: $a_{control\ 1}(V) \wedge \prod_{j=0}^{j=n} \mathbf{AX}^j(a_j(X))$

$c_{control\ 2}$: $\prod_{j=0}^{j=n} \mathbf{AX}^j(c_j(X,Y)) \wedge \mathbf{AX}^n(c_{control\ 2}(V))$

V: state variables

/ data_path_control_signals



Property for RT-level module verification

$$\mathrm{AG}(a \rightarrow c)$$

$$a = a_{start}(V) \wedge \prod_{j=0}^{j=n} \mathbf{AX}^{j}(a_j(X))$$

$$c = \prod_{j=0}^{j=n} \mathbf{AX}^{j}(c_j(X,Y)) \wedge \mathbf{AX}^{n}(c_{end}(V))$$

property *myExample* **is**
  assume:
    at ***t***: $a_{start}(V)$; //starting state //
    at ***t***: $a_0(X)$;
    at ***t*+1**: $a_1(X)$;
    at …
    at ***t*+*n***: $a_n(X)$;
  prove:
    at ***t***: $c_0(X,Y)$;
    at ***t*+1**: $c_1(X,Y)$;
    at …
    at ***t*+*n***: $c_n(X,Y)$;
    at ***t*+*n***: $c_{end}(V)$; //ending state //
end property;

V: state variables, X: inputs, Y: outputs

Property for RT-level module verification

Assumptions a()

- ***we start in a certain control state***
- ***a certain input sequence arrives***

Commitments c()

- ***certain input/output relations hold***
- ***operation ends in a certain control state***

property *myExample* **is**
  assume:
    at ***t***: $a_{start}(\boldsymbol{s})$; //starting state //
    at ***t***: $a_0(\boldsymbol{x})$;
    at ***t+1***: $a_1(\boldsymbol{x})$;
    at …
    at ***t+n***: $a_n(\boldsymbol{x})$;
  prove:
    at ***t***: $c_0(\boldsymbol{x},\boldsymbol{y})$;
    at ***t+1***: $c_1(\boldsymbol{x},\boldsymbol{y})$;
    at …
    at ***t+n***: $c_n(\boldsymbol{x},\boldsymbol{y})$;
    at ***t+n***: $c_{end}(\boldsymbol{s})$; //ending state //
end property;

$\boldsymbol{s}$: state variables, $\boldsymbol{x}$: inputs, $\boldsymbol{y}$: outputs

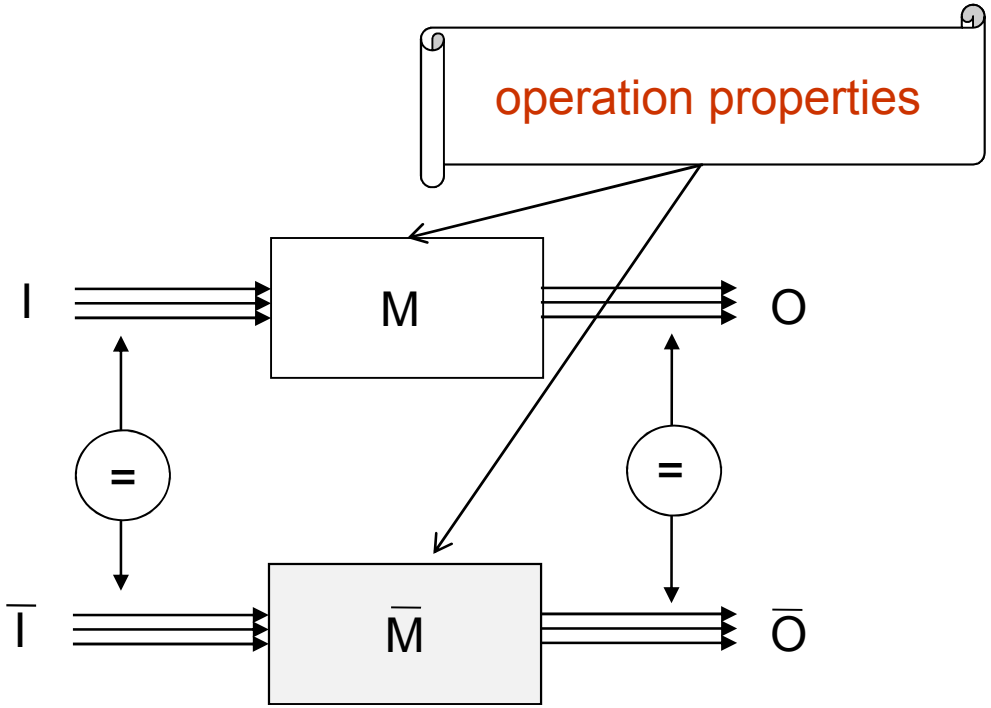
Formal Module Verification

Usage model: "Automated code inspection"

Code review: verification engineer inspects code of chip designer

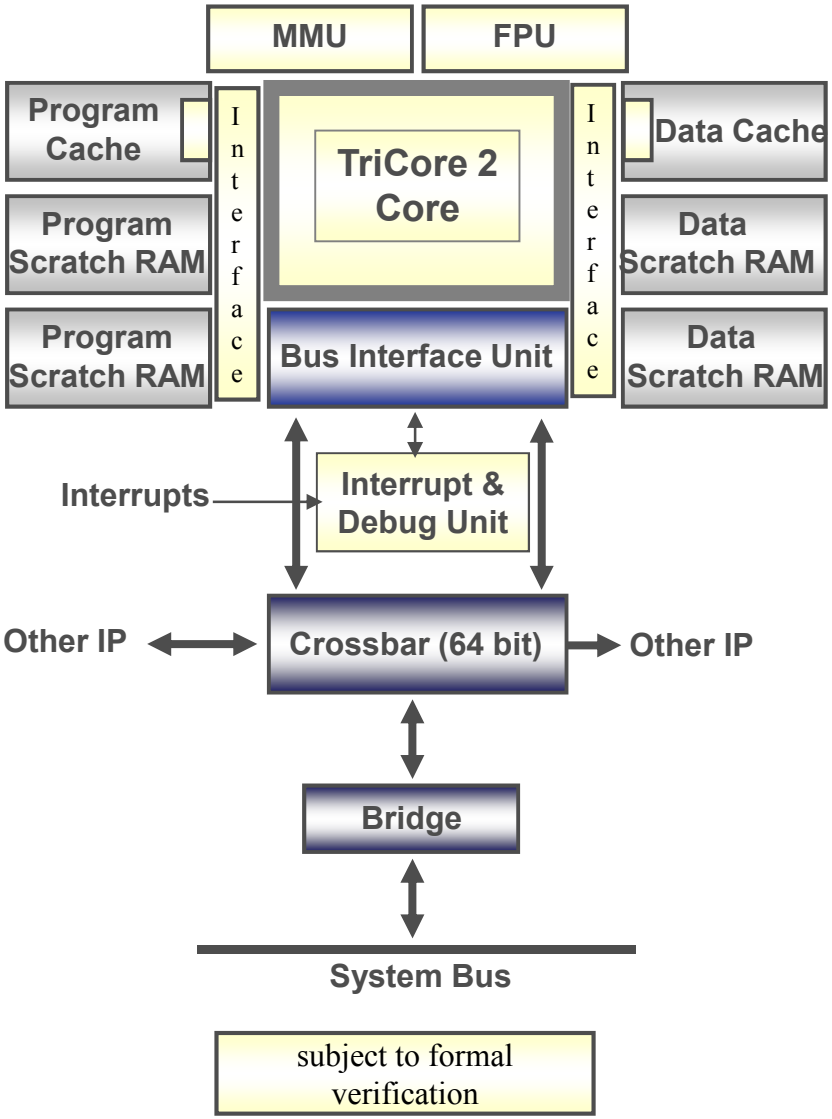
- Looks at RT code and seeks explanation in specification
- Formulates hypothesis on behavior of implementation, formulates this hypothesis in terms of property that can be checked automatically
  - If property fails, design error is detected, or, verification engineer improves his understanding of implementation and specification and corrects his property
  - Every true property documents a piece of correct design behavior
- Walks through the code, *operation by operation*, and covers each piece of code by appropriate hypotheses
- Process is continued until implementation code is completely covered by properties (metrics available to check completeness!)

Completeness Criterion



A set of operation properties is called *complete*, if it uniquely determines the I/O behavior of a circuit, i.e., every two circuits M, $\overline{M}$ fulfilling the properties are sequentially equivalent.

TriCore 2 Microprocessor System of Infineon



Architectural characteristics

- unified 32-Bit-RISC/DSP/MC architecture
- 853 instructions
- 6-stage superscalar pipeline
- multithreading extensions
- coprocessor support/floating point unit

Current Implementation

- 0.13 micron technology
- 3 mm^2 core area/8 mm^2 hardmacro area
- typical frequency ~ 500 MHz
- typical compiled code 1.5 MIPS / MHz
- 2 MMACS/MHz, 0.5 mW/MHz @ 1.5 V

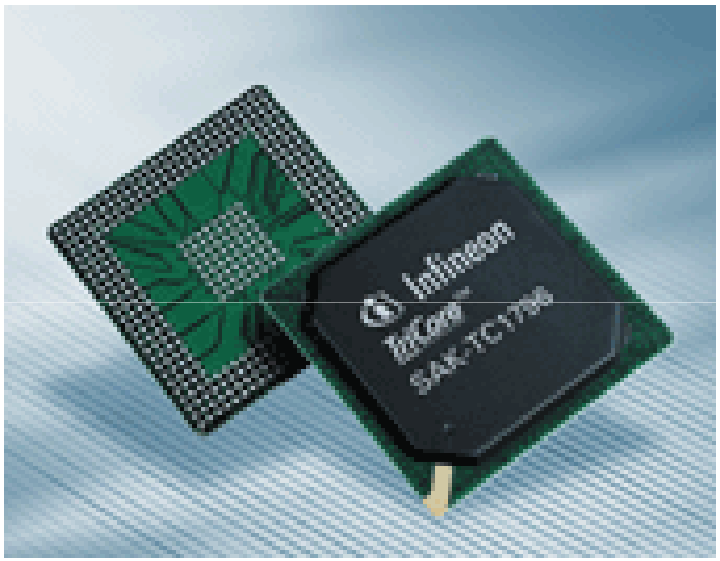
Deployment

- primarily in automotive high-end

Infineon Tricore 2 project – Example

Every instruction of the processor is verified by formulating a property (or set of properties) describing its behavior

MAC Unit: multiply, multiply/add, multiply/subtract saturation, rounding, shift-bits



e.g.

- MUL.H
  - packed multiply
  - 2 parallel multiplications
  - 8 variants +12 special cases
  - 16 bit operands
  - 64 bit result

- MADD(S).Q
  - multiply/add in Q-format
  - 40 variants + 24 special cases
  - 32/16 bit operands
  - 64/32 bit results
  - some variants with saturation

Verification of processor pipelines

Goal Prove that instructions are performed correctly

Example

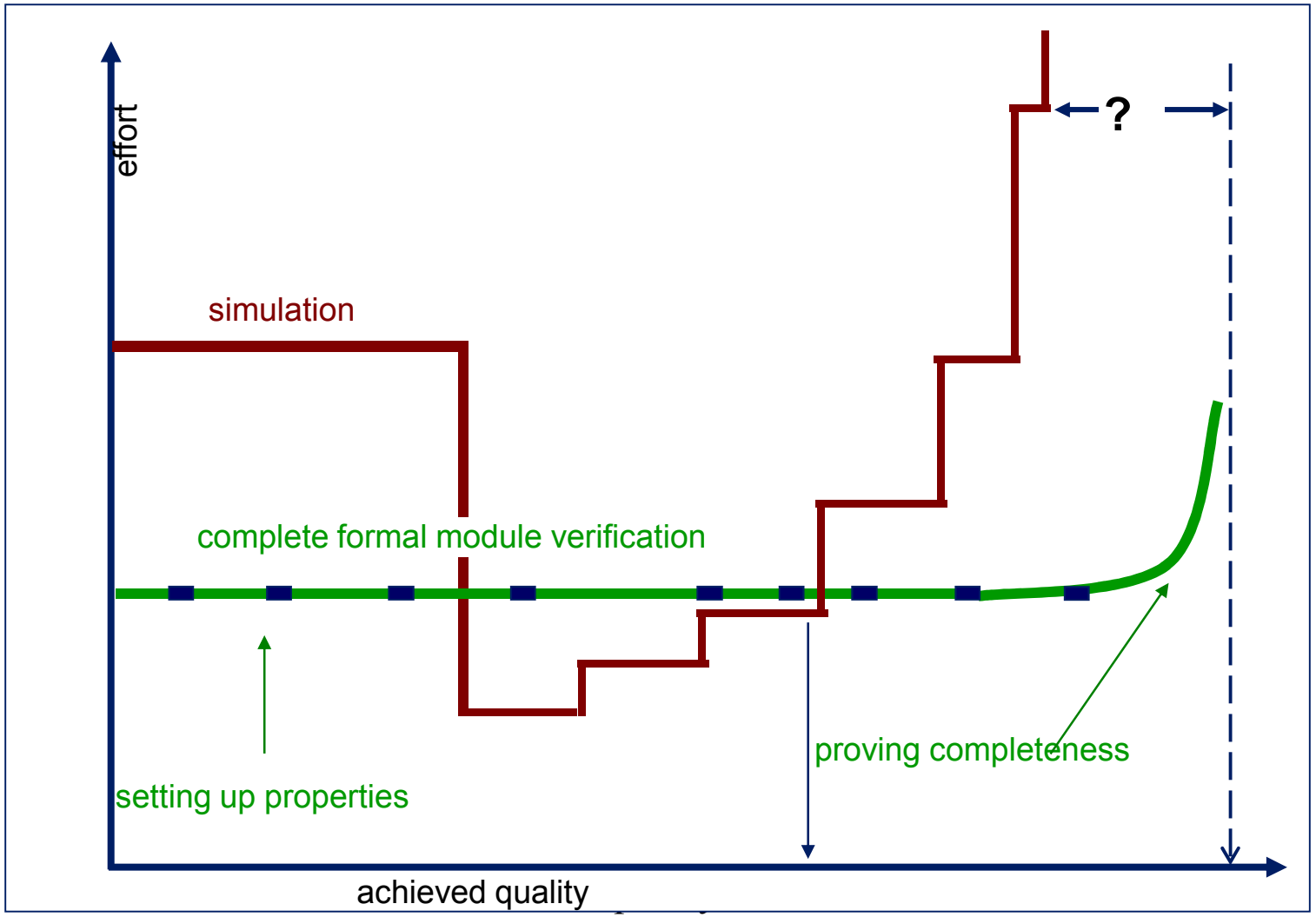
Property in ITL (InTerval Language): "assumption + commitment"

"assumptions"

"commitments"

```
theorem mul;      // packed half word
                     multiplication
assume:
  at t: command_dec(MUL,op1,op2);
  during[t,t+3]: no_reset;
  during[t,t+3]: no_cancel;
  …
prove:
  at t+3: ip_res[31:0]
             == op1[15:0]*op2[15:0];
  at t+3: ip_res[63:32]
             == op1[31:16]*op2[31:16];
  at t+5: decode_next_instruction;
end
```

Simulation vs. Complete Formal Module Verification



The Tricore processor – some results

Performance of property checking

- 99.9 % of properties run in less than 2 minutes on solaris machine
- current property suite runs in 40 hours on 1 solaris machine

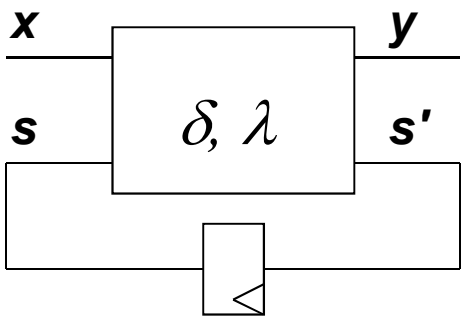
Productivity

- 2k LoC per person month exhaustively verified

Quality

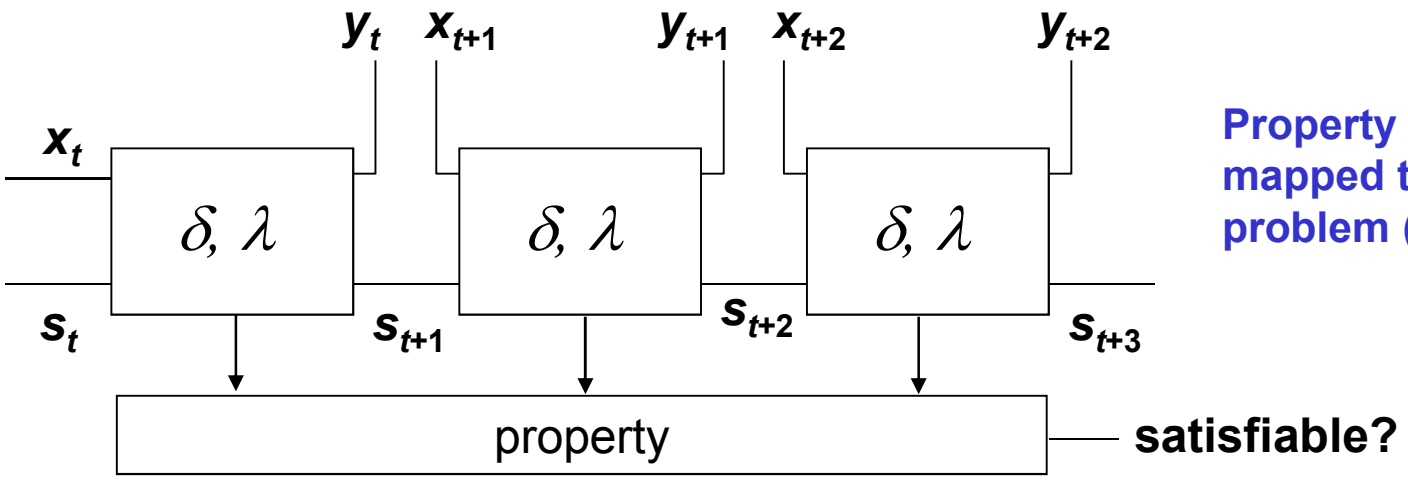
- formal techniques identified bugs that are hard or impossible to find by conventional technique
- drastic reduction of errata sheets seems realistic

New-Generation Property Checking



Computation and representation of state sets are very hard !

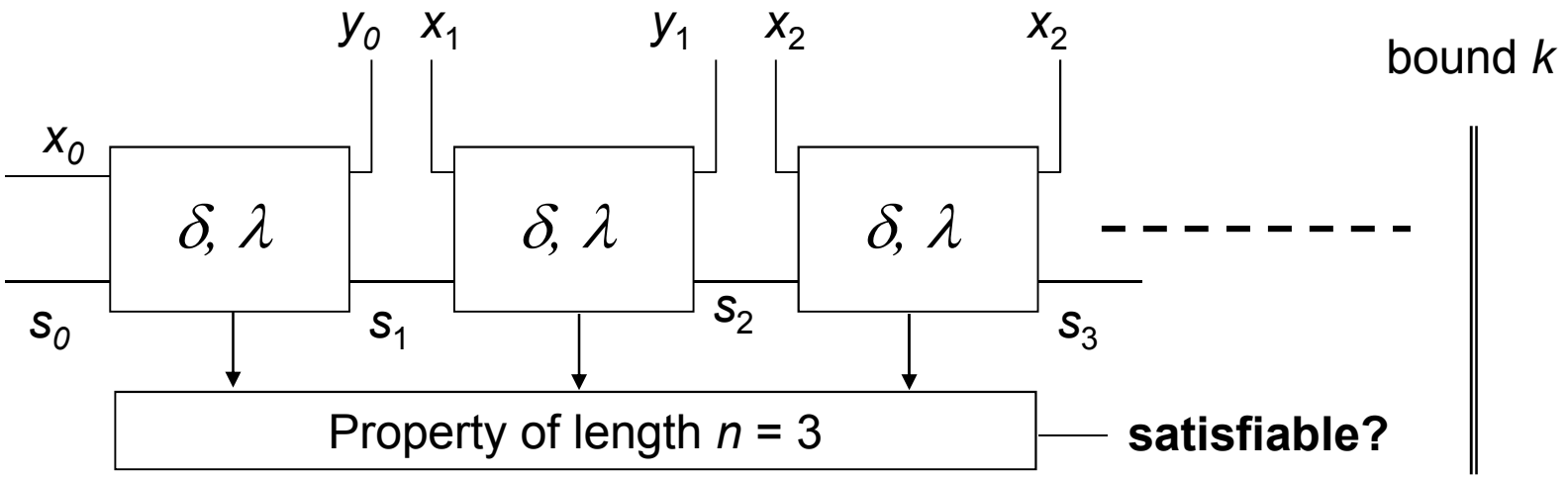
Consider machine in selected time window



Property checking mapped to satisfiability problem (SAT)

Bounded Model Checking

Properties are proved for finite time interval!



generate SAT instance:

$$I(s_0) \wedge \bigwedge_{i=0}^{k+n} \tau(s_i, x_i, s_{i+1}) \wedge \bigvee_{t=0}^{k} [p]^t$$

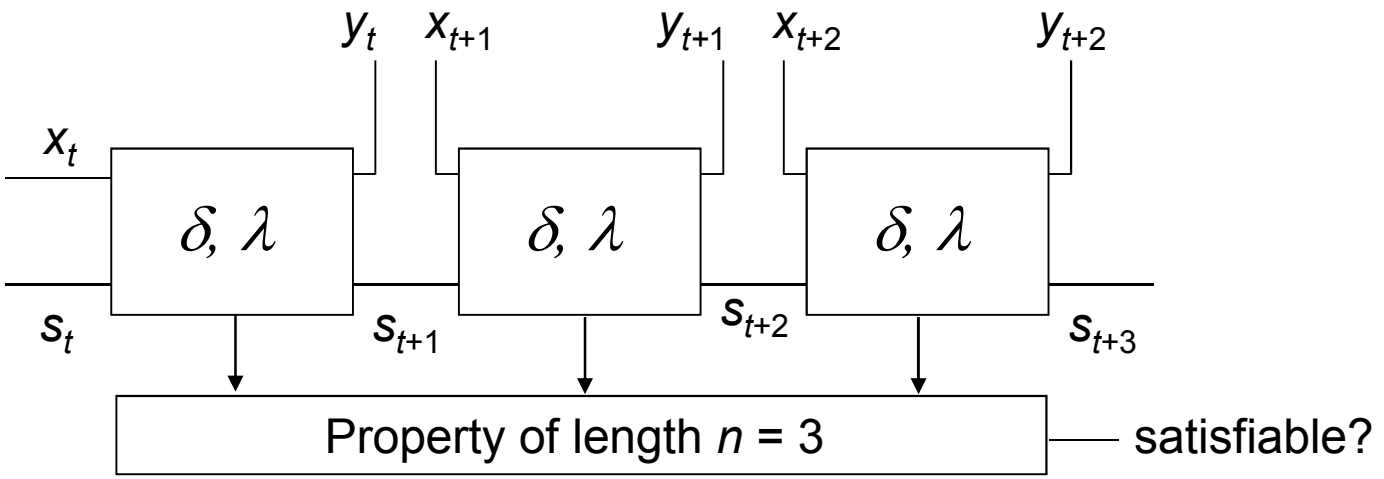
initial states

transition relation unrolled k+n times

propositional formula for internal formula of AGp, k instances

Modified formulation

Proving safety properties (AG*p*) using bounded circuit model



generate SAT instance:

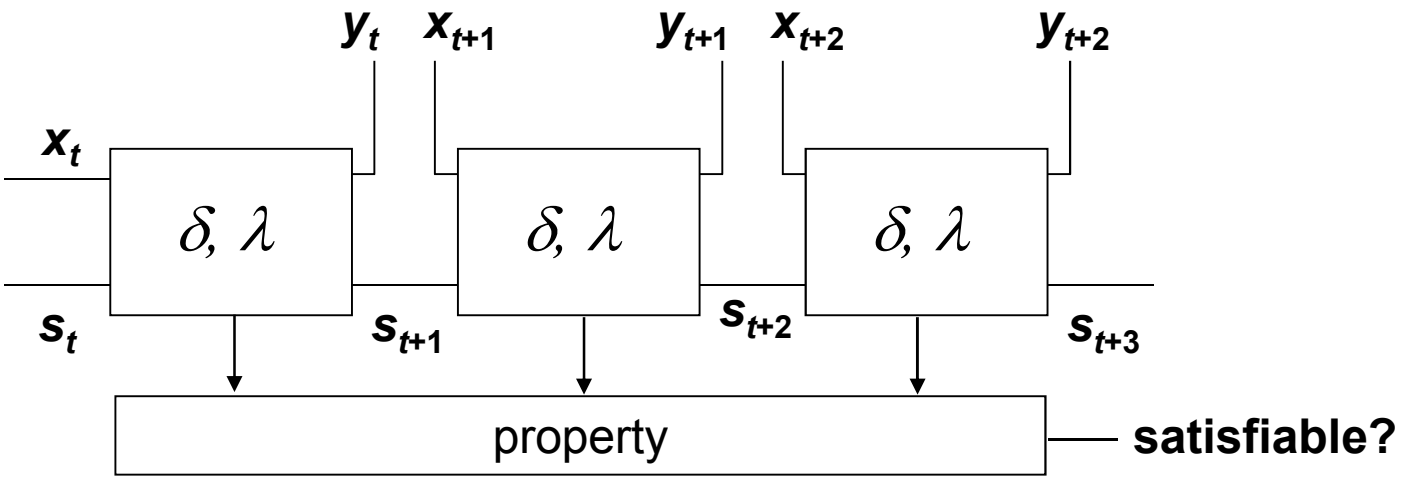
„Interval Property Checking (IPC)“
[Siemens: 1995]

$$\bigwedge_{i=t}^{t+n} \tau(s_i, x_i, s_{i+1}) \wedge [p]^t$$

transition relation unrolled n times

one instance of propositional formula for property

Interval Property Checking

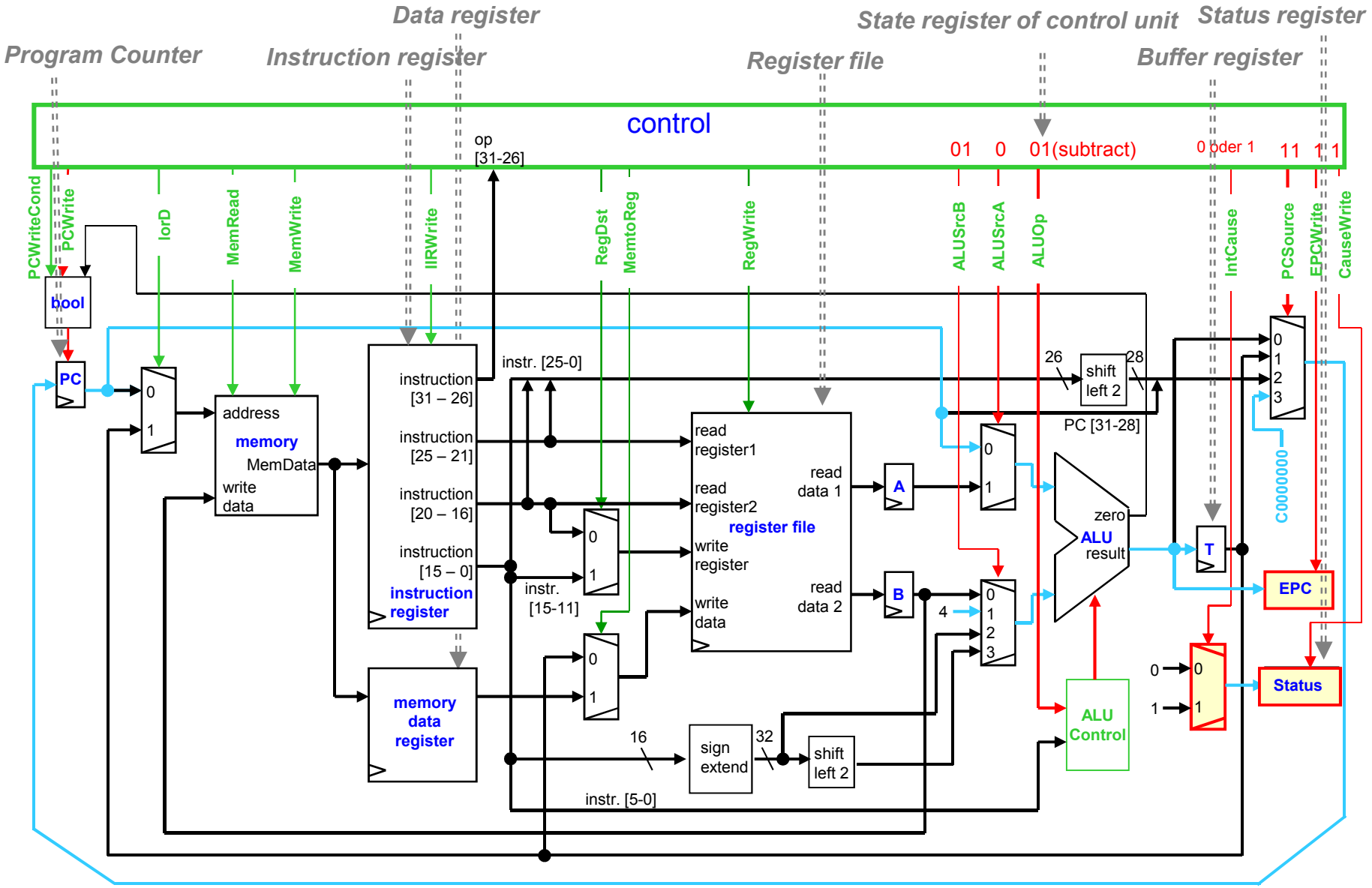


Interval Property Checking essentially means that we prove safety properties constructing a certain combinational circuit and solve a SAT problem for it.

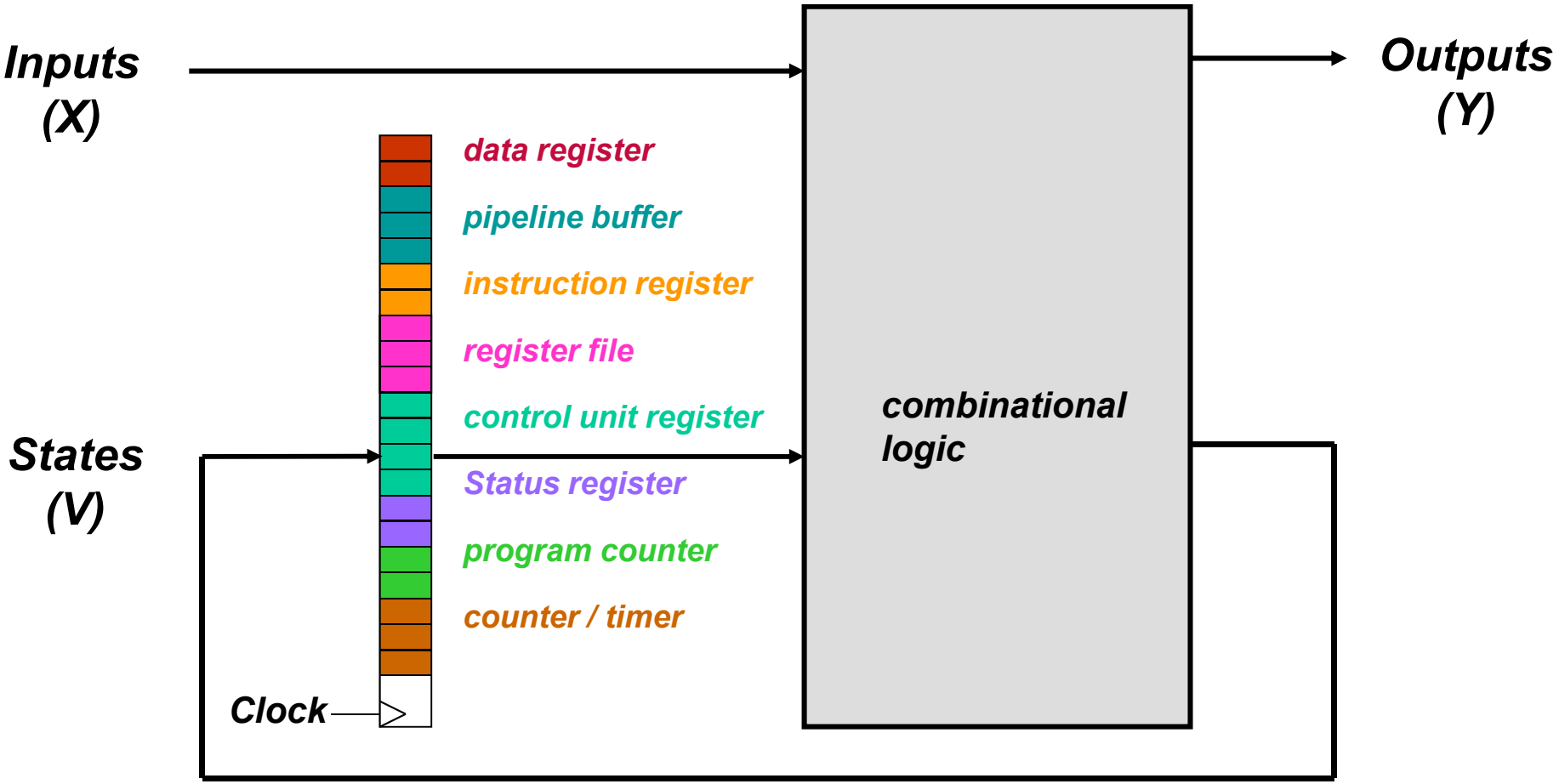
So, what about all the classical notions of

- reachability analysis
- representations and operations for large state sets
- finite state machine traversal techniques
-…

Example: Registers of an SoC-module



Representation of SoC Module as Moore- or Mealy machine



The registers of the SoC module correspond to different segments in the global state vector *V*.

Operation property

Note:

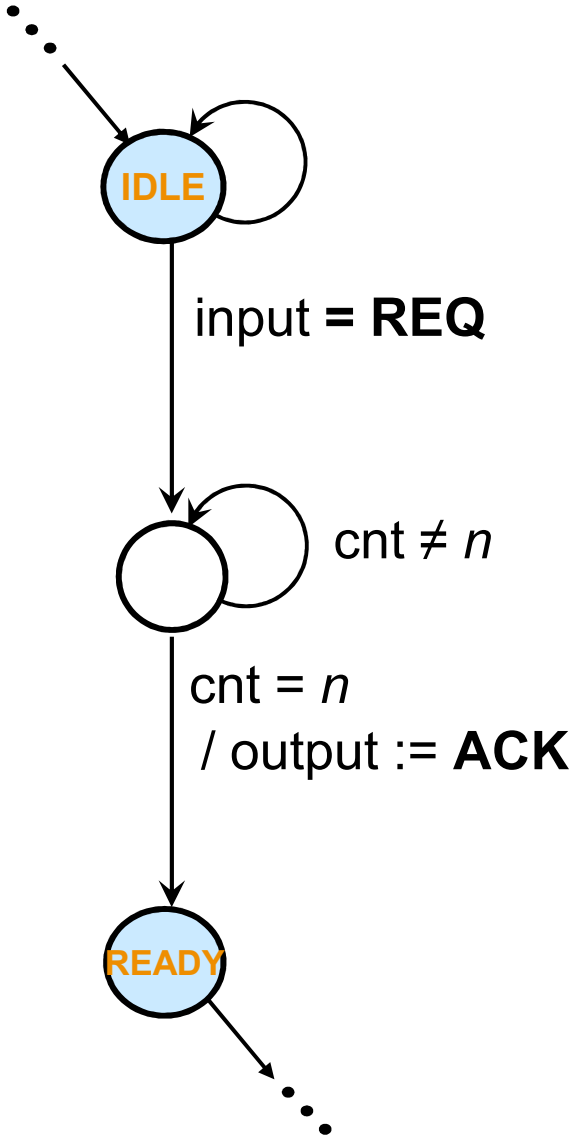
In general, operational properties specify the register contents only for a subset of the SoC registers.

e.g., a property may specify the opcode bits of the instruction register as well as some bits of the control unit registers. Nothing is said about all other registers.

property *myExample* **is**
  assume:
    at ***t***: $a_{start}(V)$; //starting state //
    at ***t***: $a_0(X)$;
    at ***t*+1**: $a_1(X)$;
    at …
    at ***t*+*n***: $a_n(X)$;
  prove:
    at ***t***: $c_0(X,Y)$;
    at ***t*+1**: $c_1(X,Y)$;
    at …
    at ***t*+*n***: $c_n(X,Y)$;
    at ***t*+*n***: $c_{end}(V)$; //ending state //
end property;

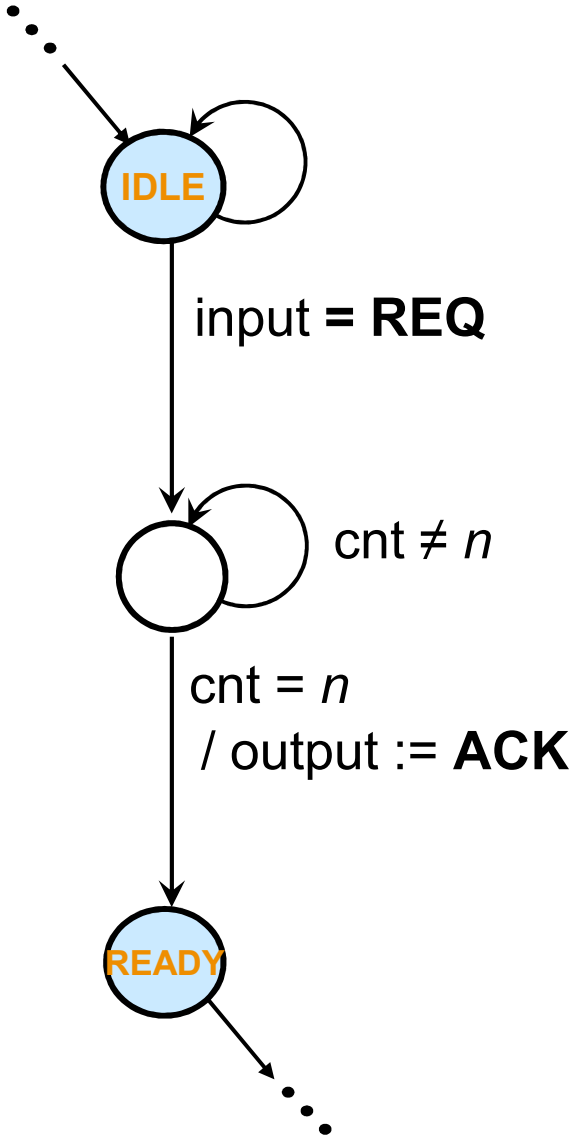
V: state variables, X: inputs, Y: outputs

Example: Verifying communication structures



FSM describes a transaction in a request/acknowledge protocol. System waits for input "request". If it arrives a counter is started. When the counter has counted up to *n* an acknowledge is given and the FSM goes into state READY.

Example

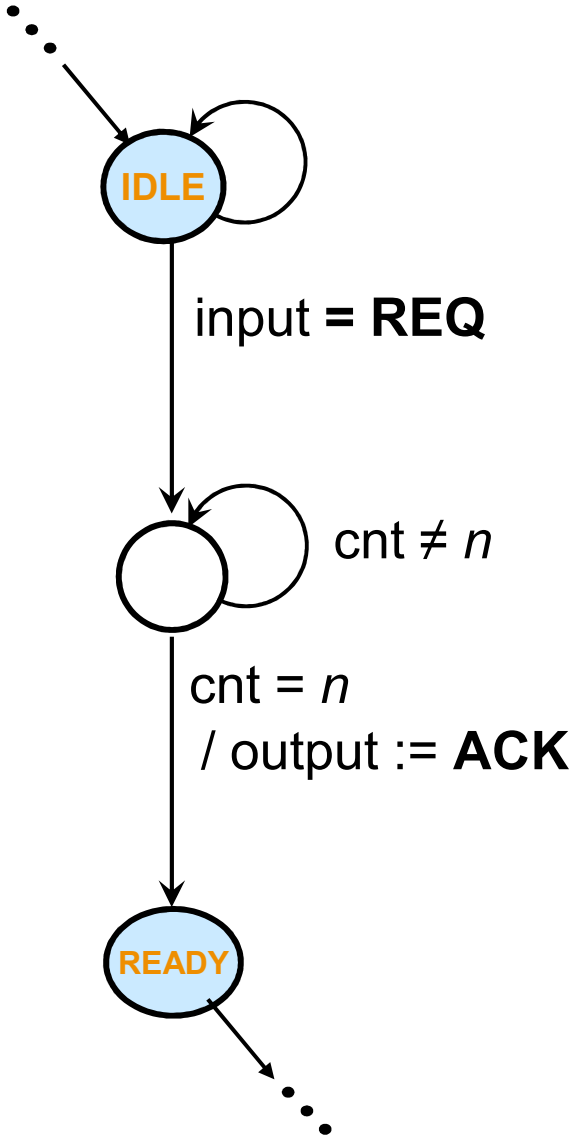


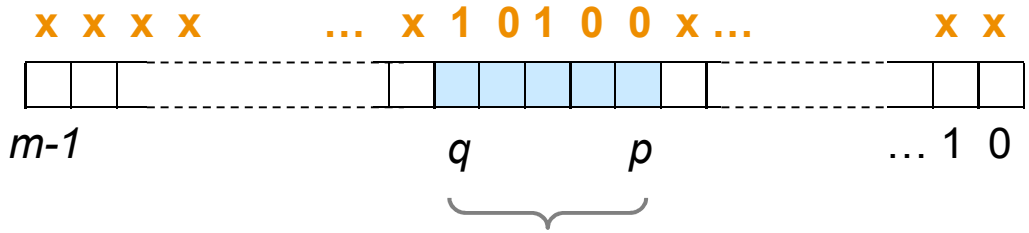
Property

assume:
at *t*: (state = IDLE && input = **REQ**)
prove:
at *t*+*n*: (state = READY && output = **ACK**)

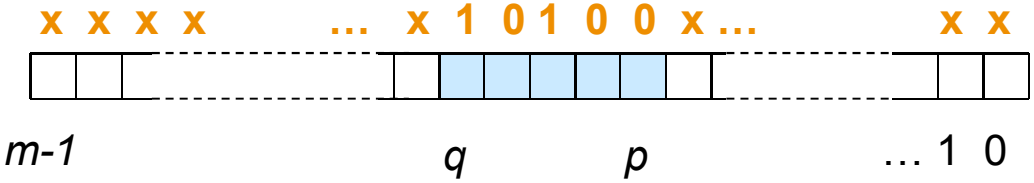
Operational property with IDLE and READY as starting and ending states

Example



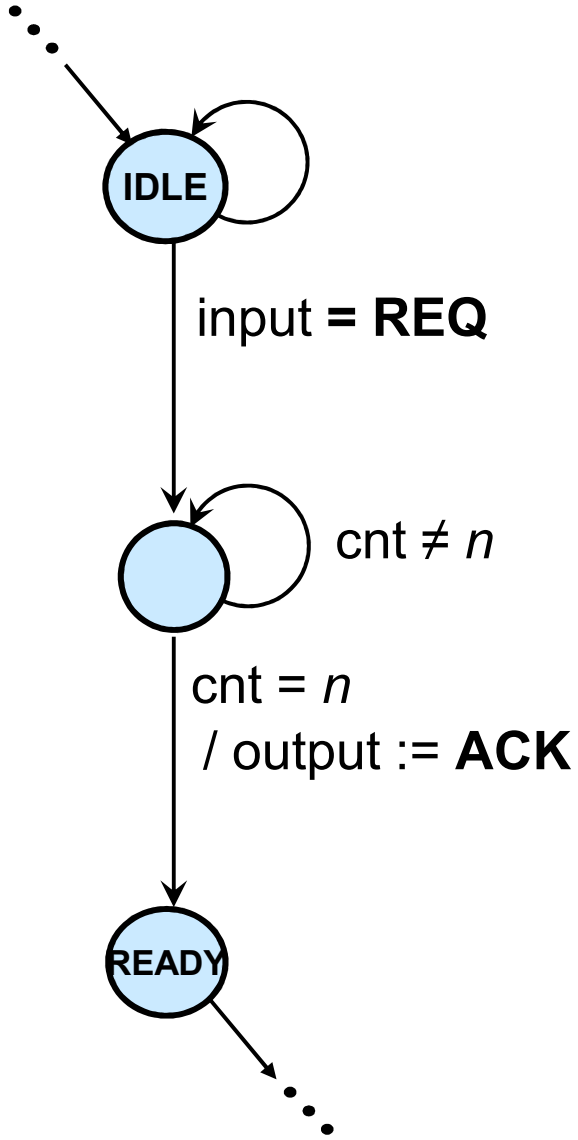


IDLE and READY are specified by asserting certain state bits in the global state vector



global state vector with m bits

Example



Property

assume:
at *t*: (state = **IDLE** && input = **REQ**)
prove:
at *t*+*n*: (state = **READY** && output = **ACK**)

False!
Counterexample:

READY after n-1 cycles but not later

Example

Verification engineer analyzes situation:

Inspection of counterexample

at time t: counter value is 1 when the controller is in IDLE, but should be 0

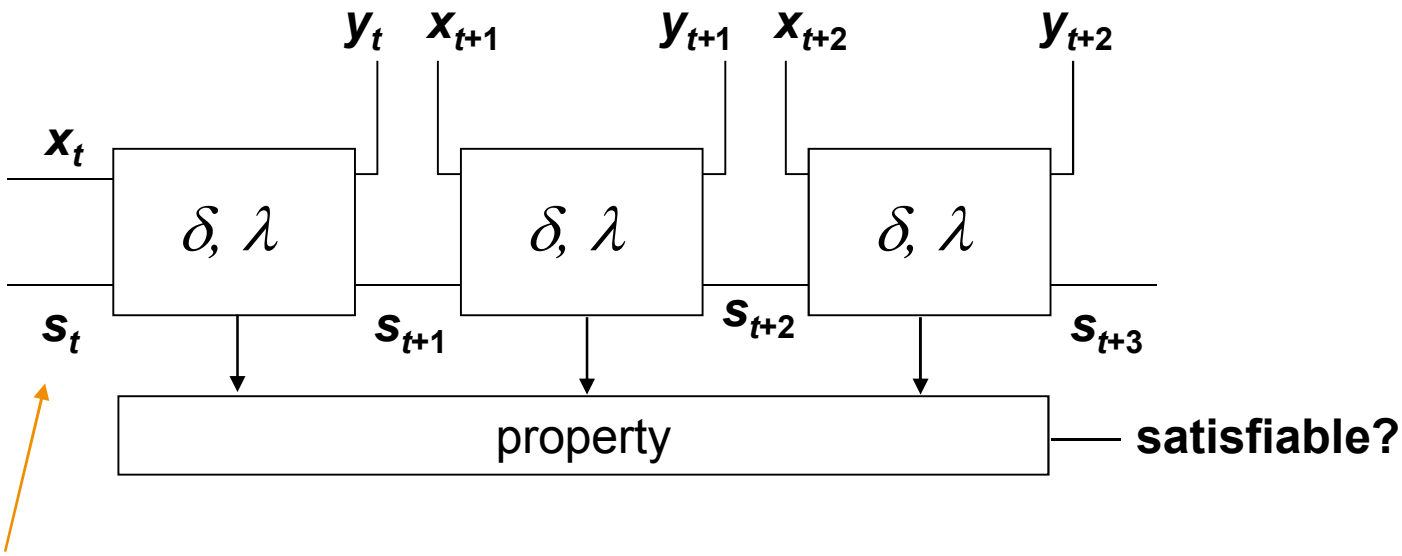
⇒ Is there a bug? Forgot to initialize the counter properly?

Inspection of design

Design is correct! Counter is always 0 when controller is in state IDLE

⇒ The tool's answer is wrong! ("False Negative")

Example



At time *t*:

counter value is 1 when controller is in IDLE

This is possible in our computational model, even if it is not possible in the real design!

Note: there are no restrictions on s_t

$\Rightarrow$ all binary code words are considered to be reachable states at time *t* !

IPC: The reachability problem

Do we still need this stuff?

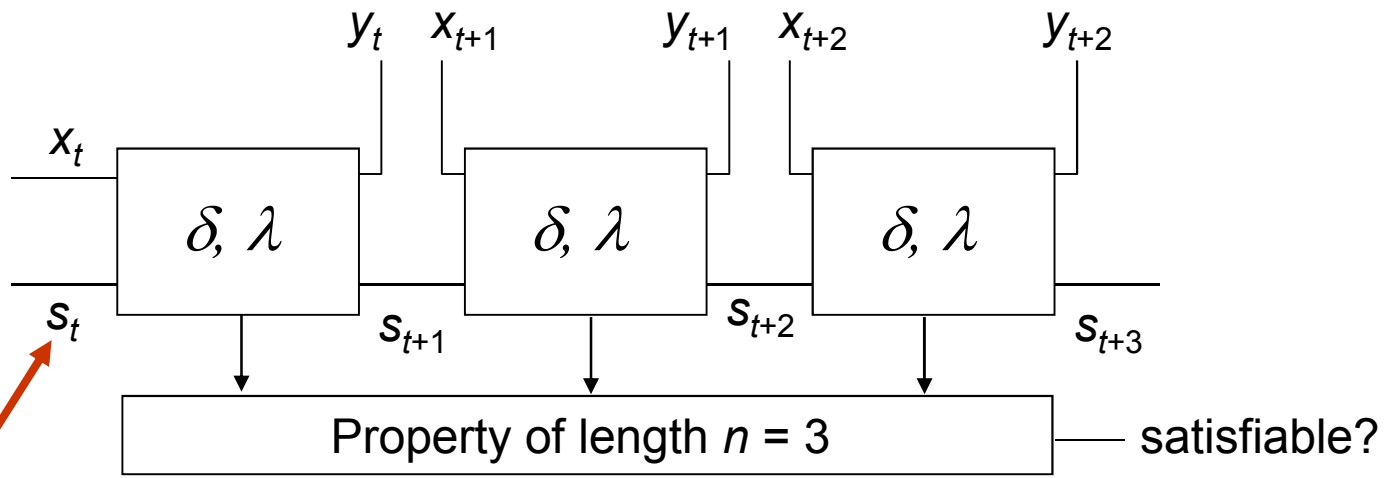
- reachability analysis
- representations and operations for large state sets
- finite state machine traversal techniques

But then

we are back in the 90s
and we can only handle small designs …

IPC with invariants

Proving safety properties (AG*p*) by IPC with invariants



need to add reachability information here!

generate SAT instance:

$$\phi(s_t) \wedge \bigwedge_{i=t}^{t+n} \tau(s_i, x_i, s_{i+1}) \wedge [p]^t$$

Invariant

(special case: ϕ = 1)

transition relation unrolled n times

one instance of propositional formula for property

Invariants

The notion of an „invariant“

Definition:

A set of states *W* in a finite state machine *M* is called *invariant* if *W* contains all states being reachable from *W*.

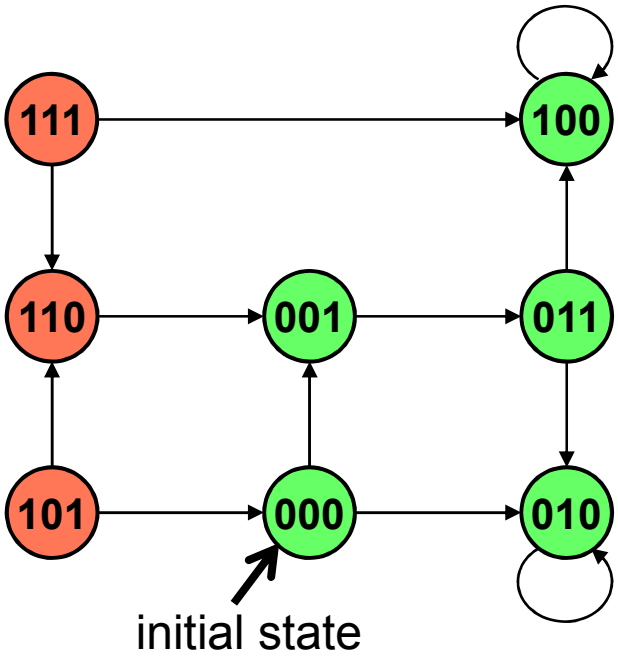
Example:

The set of all reachable states in *M* is an invariant.

Can there be other invariants than the reachable state set *R*?

Invariants

Example: FSM with 3 state variables



Reachable states:

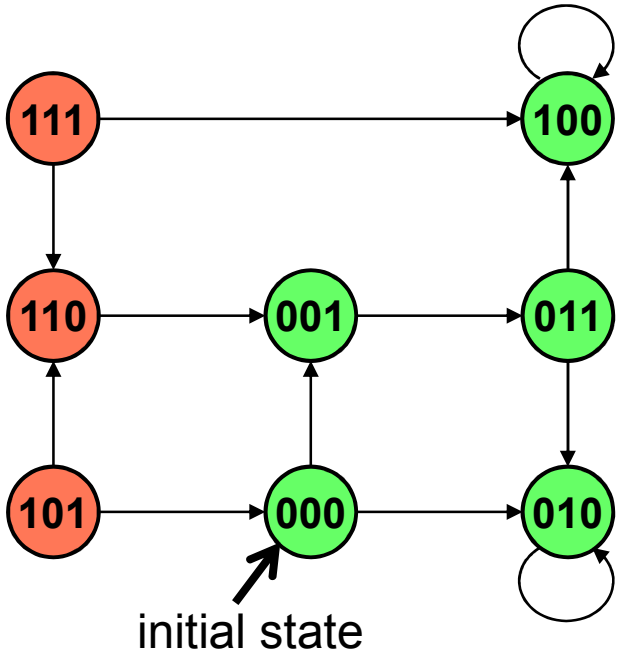
R = {000, 001, 010, 011, 100}

Unreachable states:

U = {101, 110, 111}

Invariants

Example (continued)



Invariants:

$W_1 = R =$ {000, 001, 010, 011, 100}
W_2 = {001, 011, 010, 100}
W_3 = {010}
W_4 = {011, 010, 100}
W_5 = {100}
W_6 = {101, 110, 000, 001, 010, 011, 100}
W_7 = {110, 001, 011, 010, 100}
W_8 = {111, 110, 001, 011, 010, 100}
W_9 = {010, 100}
W_{10} = {111, 101, 110, 000, 001, 010, 011, 100}

Proving Safety Properties with Invariants

Let *p* be a Boolean formula. We want to prove that *p* holds in every reachable state of the system ("safety property", in CTL: AG*p*).

If the formula *p* holds for some invariant *W* that includes the initial state, then, *p* holds in every reachable state of the system, i.e., the system fulfills this safety property.

Which invariants of the previous example can be useful to prove the safety property?

W_1, W_6, W_{10}

E.g., consider W_6:

$W_6 \supseteq R$ "W_6 over-approximates the reachable state set"

Proving Safety Properties with Invariants

Over-approximating the state space

For any state set *W*

- which is an invariant and
- which includes the initial state

it must hold that $W \supseteq R$.

Obviously, if a property holds for a superset of the reachable state set, it must also for the reachable state set itself.

Therefore, we can prove safety properties based on invariants that over-approximate the reachable state set.

Proving Safety Properties with Invariants

False negatives

But, what if the property fails for an invariant *W*, with $W \supseteq R$?

Then, we need to distinguish:

1) Property fails for one or more reachable states
(e.g. states 000, 001, 010, 011, 100 in W_6)

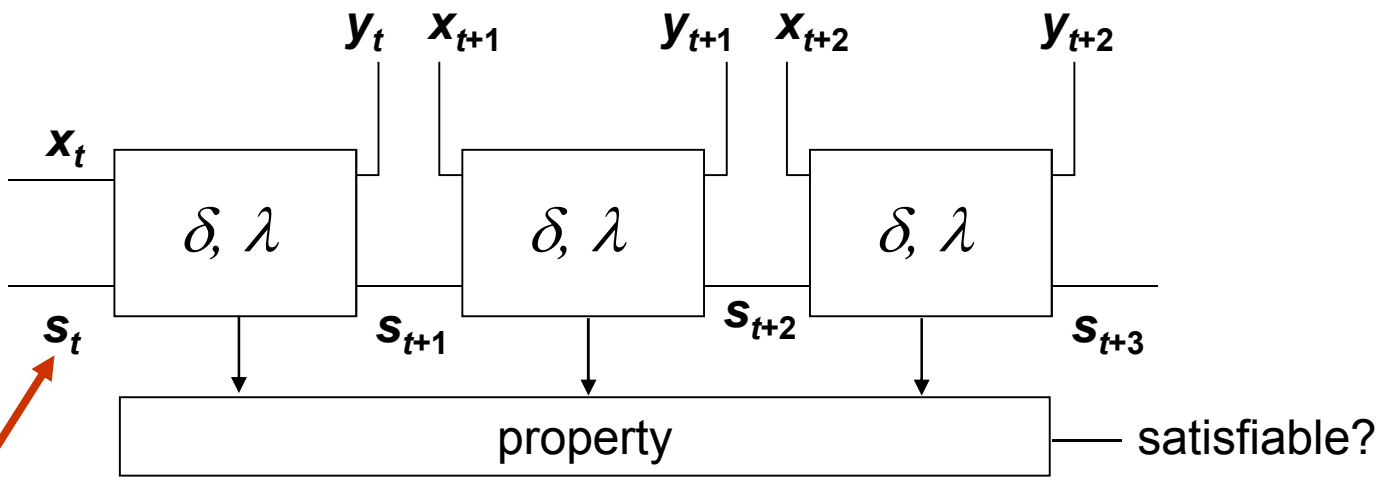
   $\Rightarrow$ there is a bug in the design (“True Negative”)

2) Property fails only for one or more unreachable states
(e.g. states 101, 110 in W_6)

   $\Rightarrow$ there is no bug in the design (“False Negative“)

   The counterexample is “spurious”, i.e., it is based on states that are unreachable in the design. Fortunately, the verification engineer can usually recognize this by inspection.

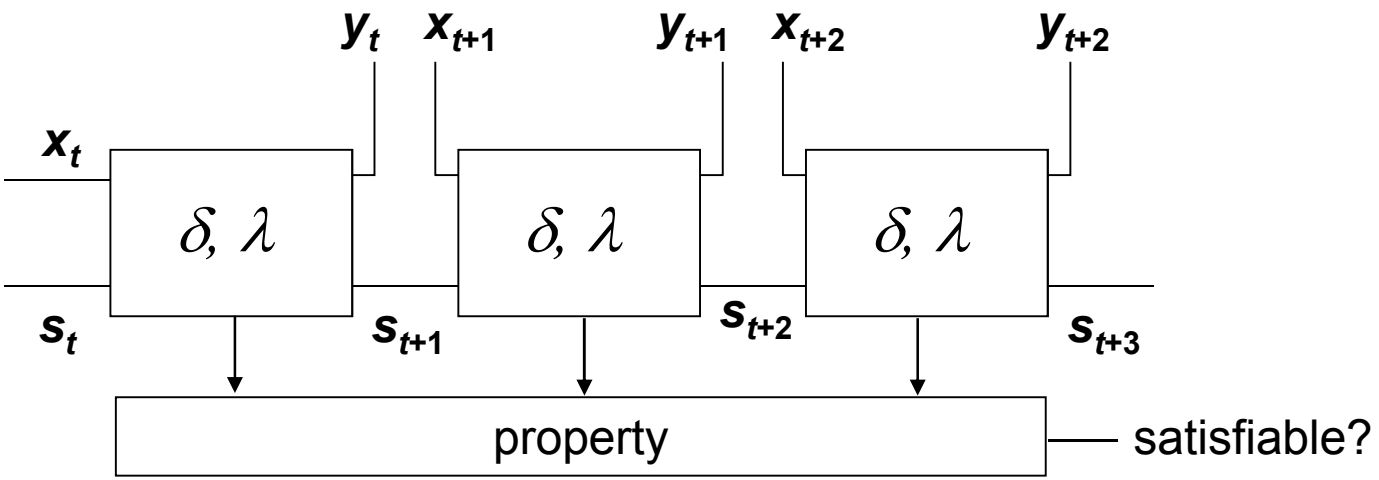
Interval Property Checking with Invariants



May need to add
reachability information here!

This reachability information is added in terms of an invariant!

IPC: The reachability problem



Which states are reachable in this model?

at time *t*: all binary state codes (includes unreachable states)
at time *t*+1: only those states that are the image of some state code
at time *t*+2: only those states that are the image of an image of a state code
…

Invariants in IPC

Proving AGp

$$\phi(s_t) \wedge \bigwedge_{i=t}^{t+n} \tau(s_i, x_i, s_{i+1}) \wedge [p]^t$$

Invariants: sets of states closed under reachability

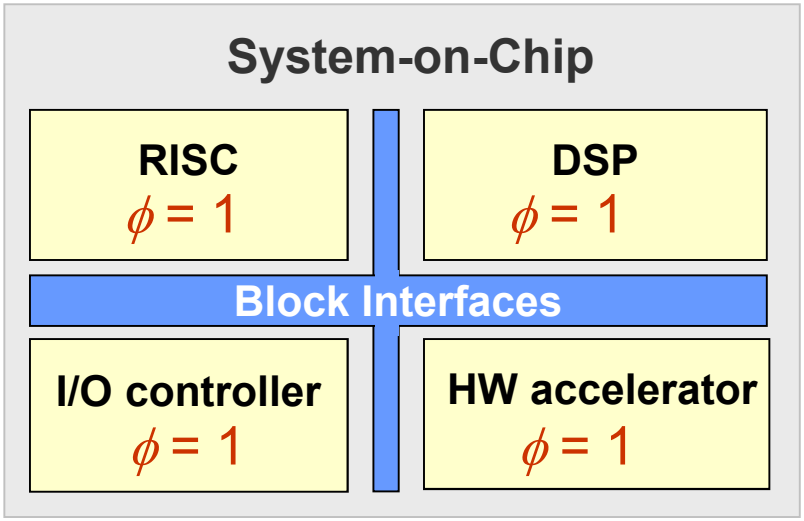
Invariants in this formulation compared to invariants in most other model checking techniques:

- can be weaker
- can be of simpler syntactic forms
- are more intuitive to the designer

⇒ common practice to derive invariants manually

IPC: the standard case

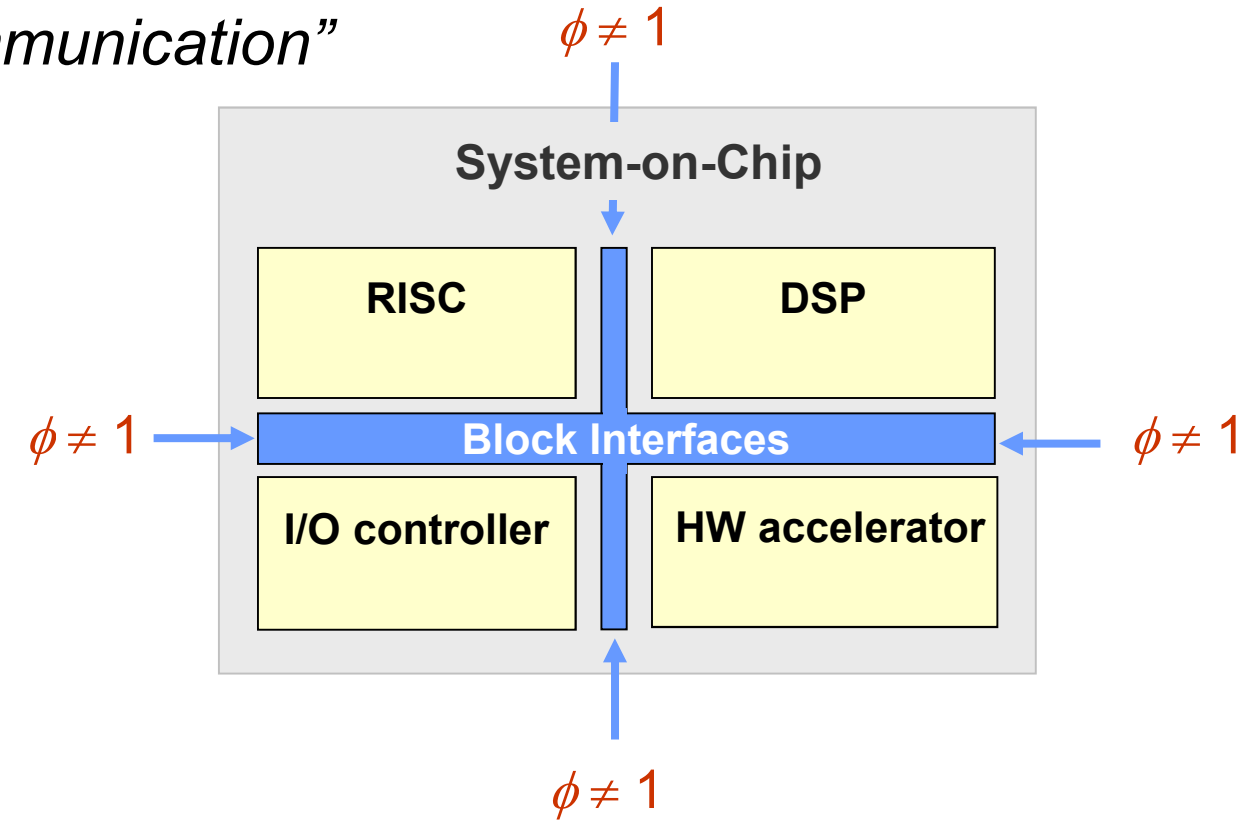
The good cases…
"computation"



$\phi = 1$ holds in most cases when verifying individual modules

IPC with invariants

The difficult cases…
"communication"



$\phi \neq 1$ for implementations of SoC protocols !

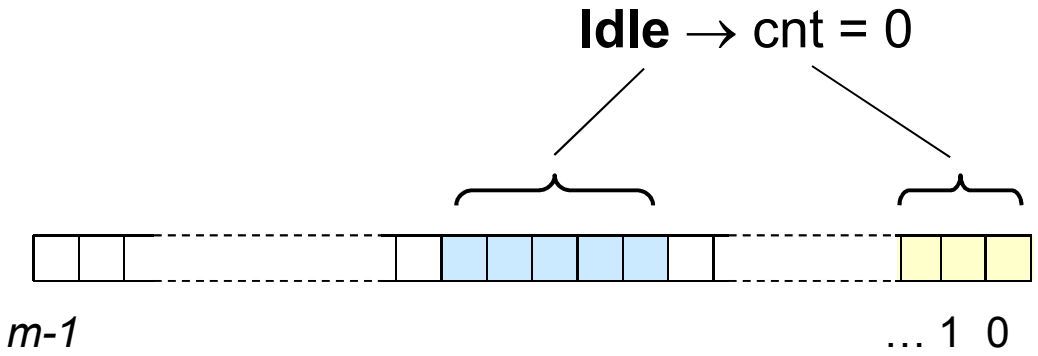
IPC with Invariants

In industrial practice invariants are often described implicitly:

all states of the code space that fulfill certain “constraints”

Example constraint: the counter value is 0 whenever the controller is in state IDLE.

This means: the designer sets up constraints (e.g. implications, equivalences) which he/she expects to hold in the design. These constraints implicitly describe a state set. Then, we try to prove that the state set characterized by the constraints is an invariant.

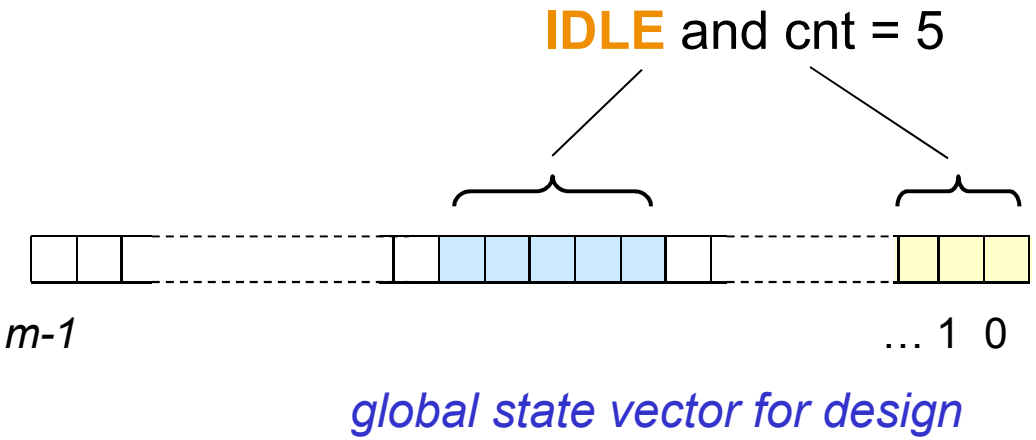


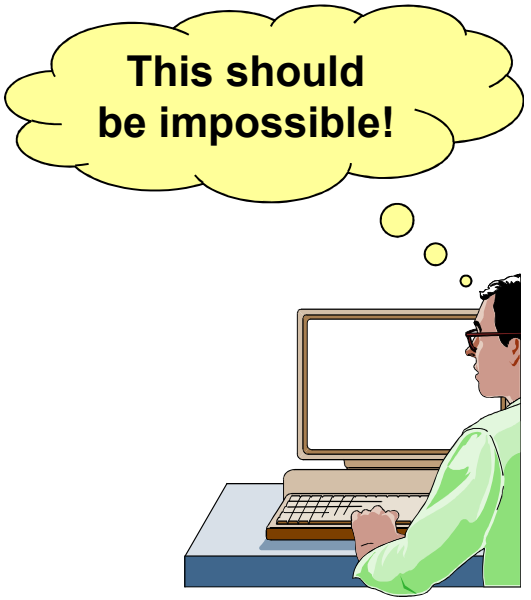
global state vector for design

IPC with Invariants

Tool produces counterexample, false negative? How to proceed in practice?

Step 1:
Inspect counterexample: check e.g. whether important states are combined with “weird”, possibly unreachable states in other parts of the design

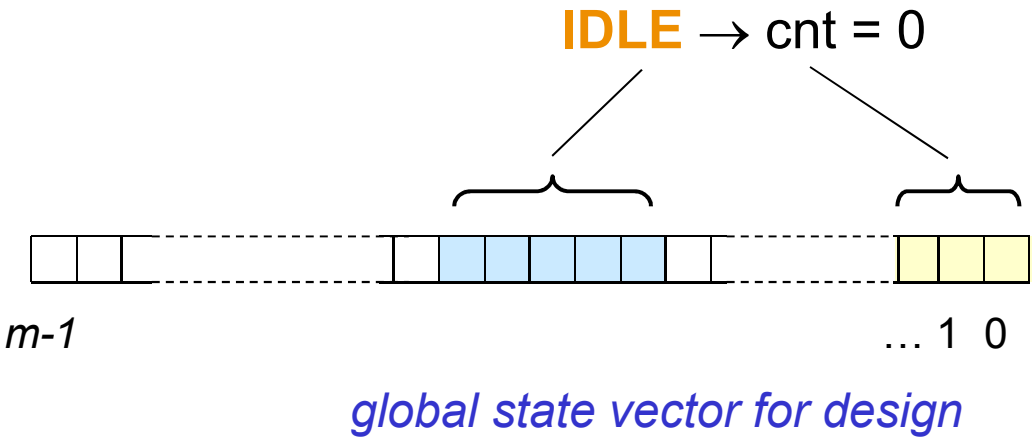




IPC with Invariants

Tool produces counterexample, false negative? How to proceed in practice?

Step 2:
Formulate a “reachability constraint” that you expect to hold for the design.



IPC with Invariants

Tool produces counterexample, false negative? How to proceed in practice?

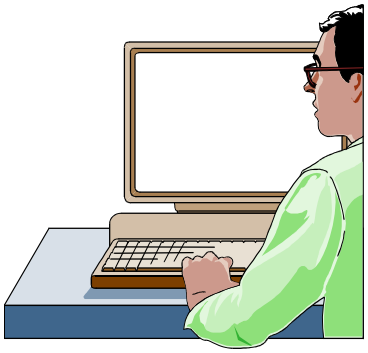
Step 3:
Prove the reachability constraint by induction.

property 1 (base case)

Assume: initial state
Prove: IDLE → cnt = 0

property 2 (induction step)

Assume:
at t: IDLE → cnt = 0
Prove:
at t+1: IDLE → cnt = 0



Hence, the state set characterized by (IDLE → cnt = 0) includes the initial state (base case) and is an invariant (induction step).

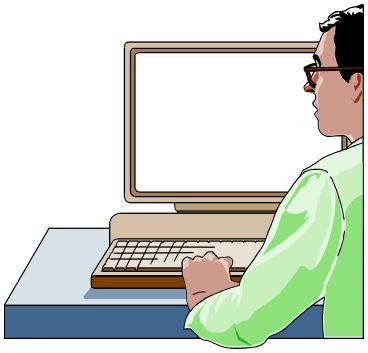
IPC with Invariants

Tool produces counterexample, false negative? How to proceed in practice?

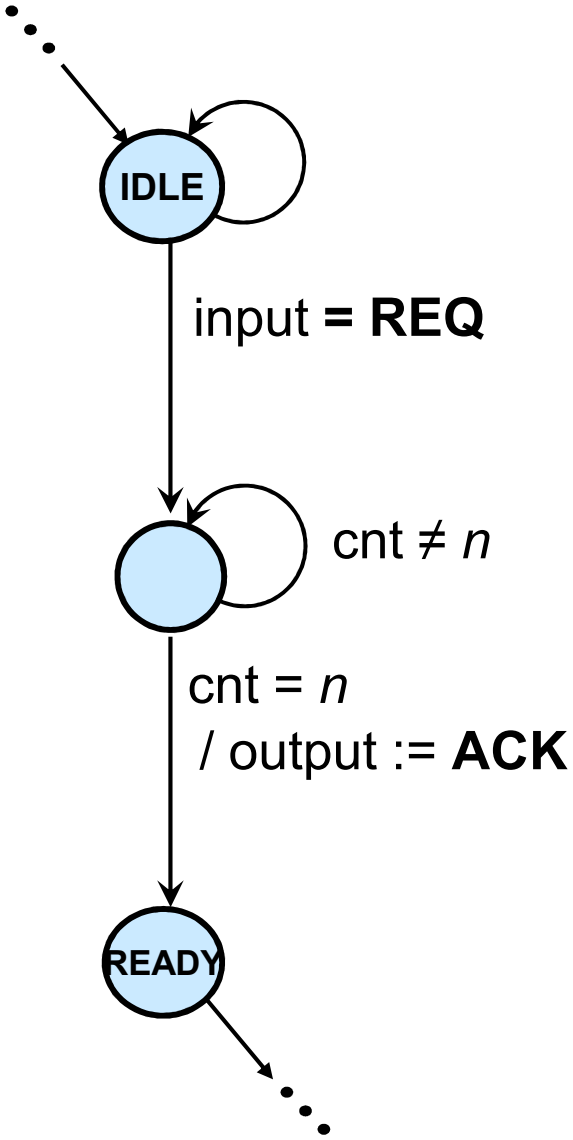
Step 4:
Prove the original property using the reachability constraint. This means that you prove the property for all states of the design that fulfill the constraint. Since the constraint is proved to be valid for the design it means that your proof is based on a state set that is an invariant and which includes the initial state.

Property

assume:
  at *t*: (state = **IDLE** && input = **REQ**
  && cnt = 0)
prove:
  at *t*+*n*: (state = **READY** && output = **ACK**)



Advanced Feature in OSS 360MV

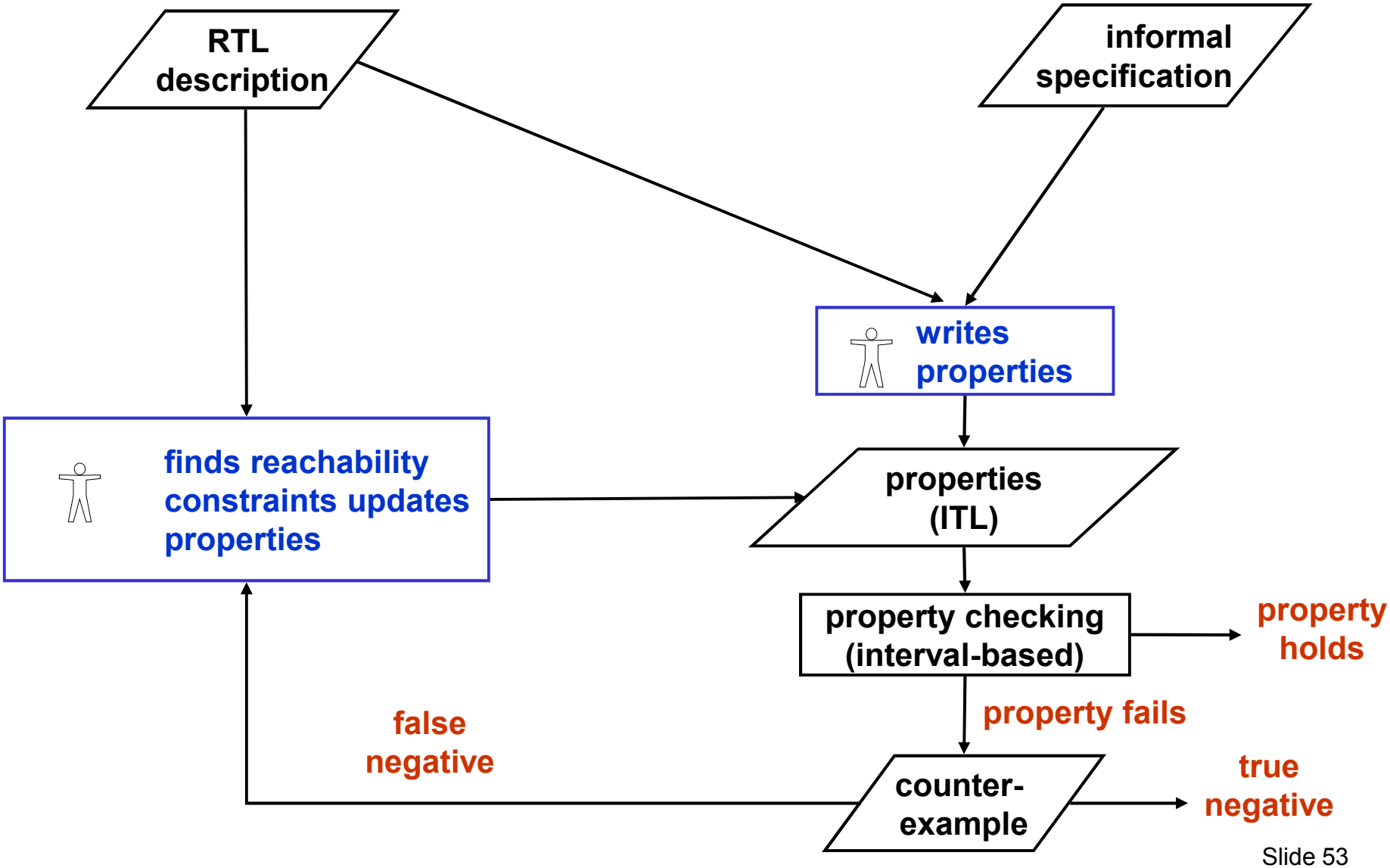



```
assertion reachabiliy_constraints :=
  if state = IDLE then cnt = 0 end if;
end assertion;

property improved is
dependencies: reachability_constraints;
assume:
   at t:      state = IDLE and input = REQ;
prove:
   at t+n:  state = READY and output = ACK;
end property;
```

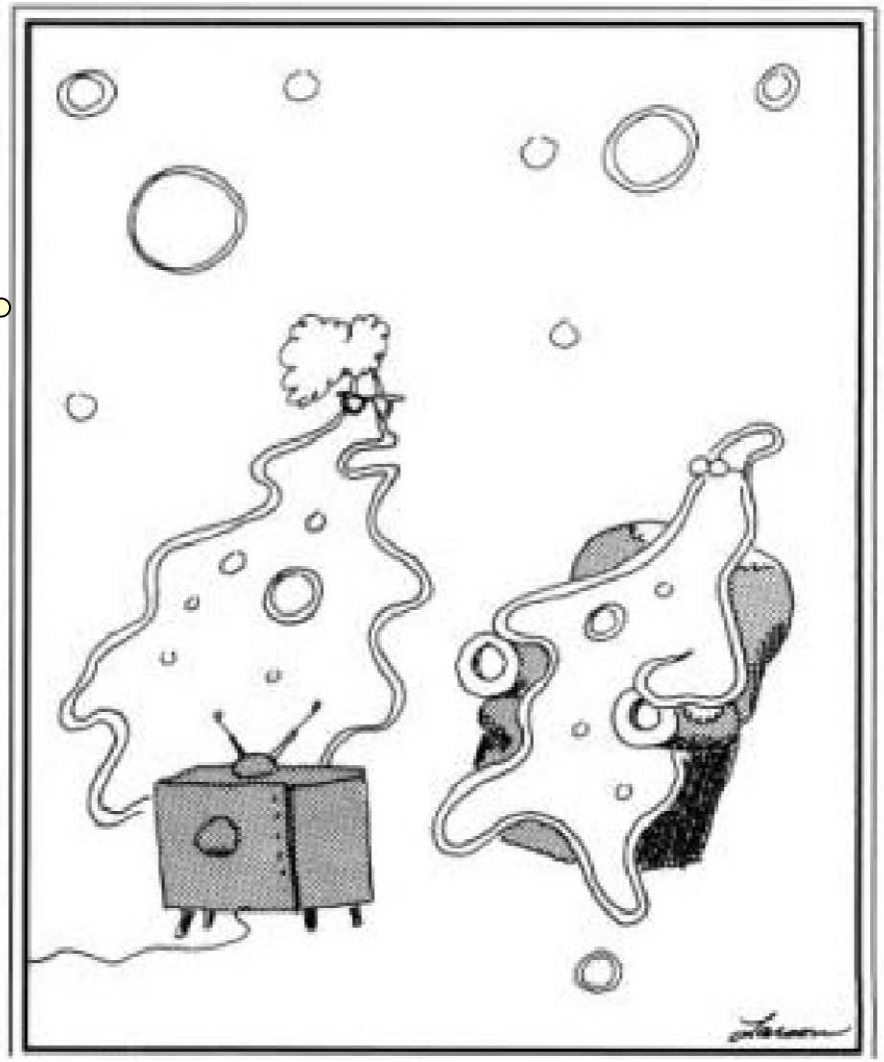
Property proven !

Methodology



Industrial Experiences

How to integrate FV in ESL-based design flows?



"Stimulus, response! Stimulus, response! Don't you ever think?"

Verification engineer's preferences

White-box versus black-box verification

Electronic System Level (ESL)

The problem

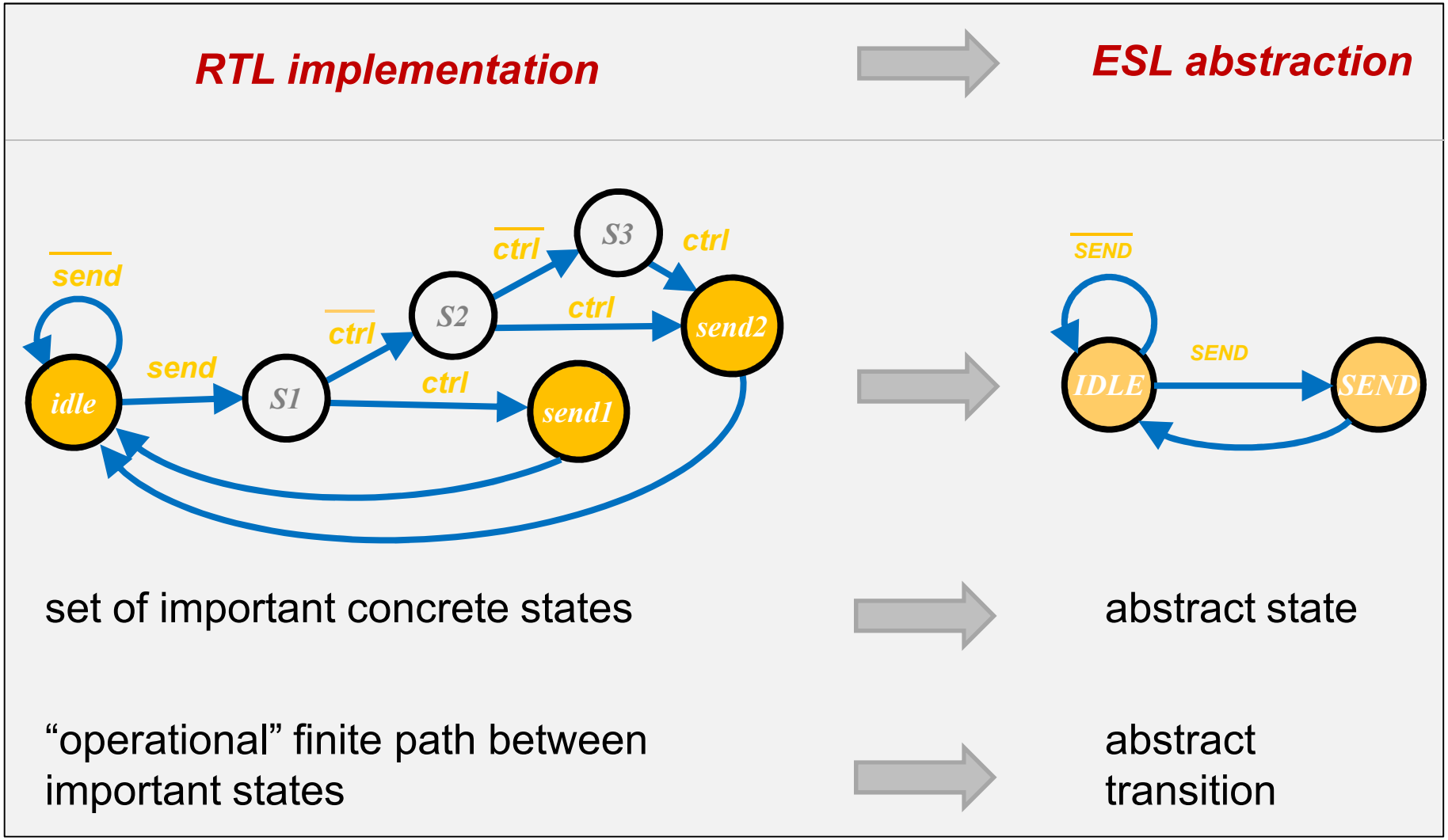
System level models are usually created in addition to the other models, e.g. for virtual prototyping

→ high costs (in spite of IP re-use)

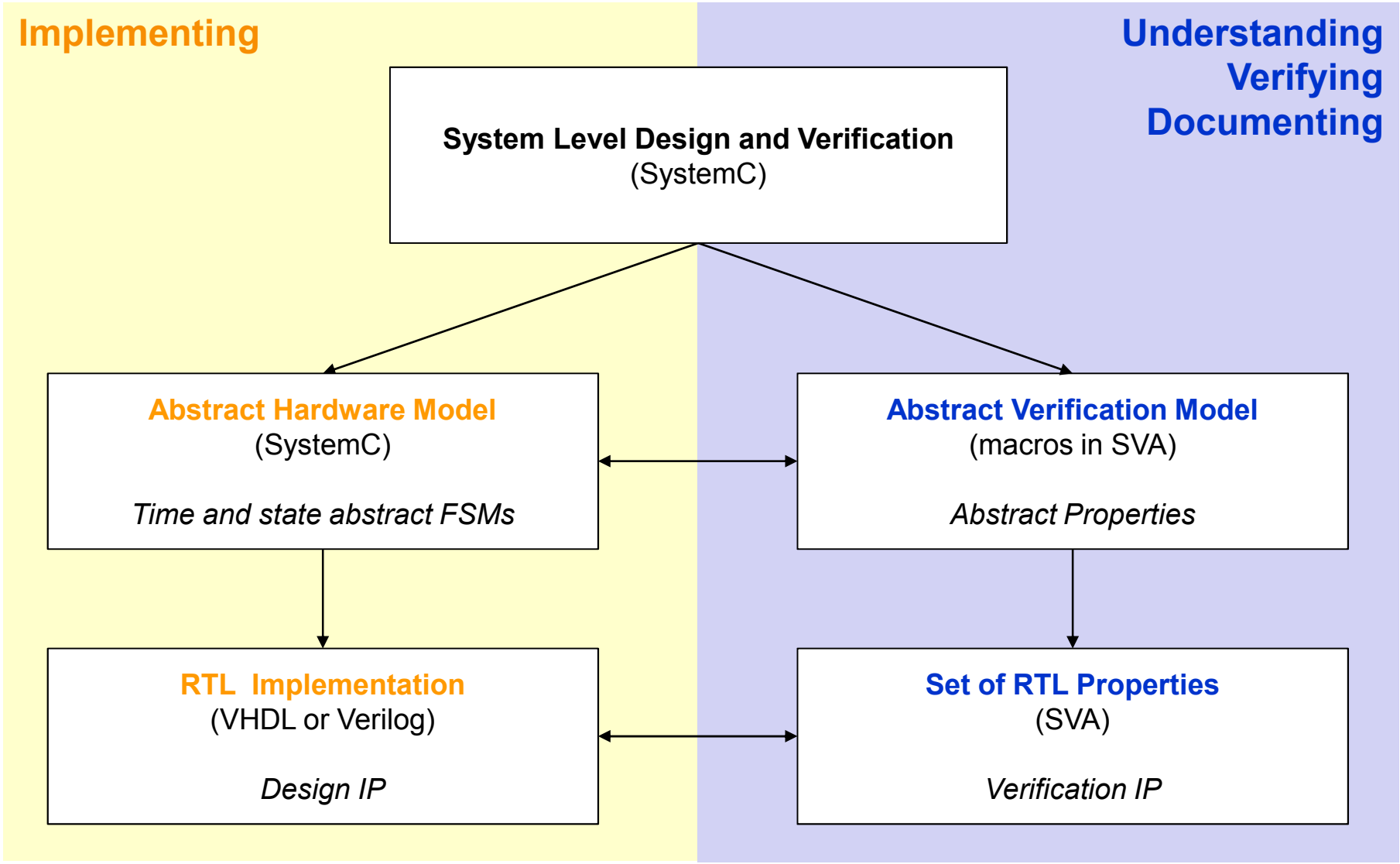
Semantic Gap: high-level synthesis applicable only in niches, no formal relationship between high-level models and concrete RTL implementation

→ ESL models do not reduce costs of RTL design
→ ESL models do not reduce costs of RTL verification

ESL – RTL: Closing the semantic gap by property checking



Future Flow?


Implementing
Understanding
Verifying
Documenting
System Level Design and Verification
(SystemC)
Abstract Hardware Model
(SystemC)
Time and state abstract FSMs
Abstract Verification Model
(macros in SVA)
Abstract Properties
RTL Implementation
(VHDL or Verilog)
Design IP
Set of RTL Properties
(SVA)
Verification IP

Conclusion

- Formal SoC verification relies on a sophisticated combination of methodologies and proof engines
- More than “bug hunting”: the result of formal RTL verification should be the soundness of the ESL model:
  - verify global system behavior at the system level (and get rid of RTL chip-level simulation!)
  - verify local register transfers (operations) at the RT level
- New challenges and opportunities for property checking in ESL-based design flows