Chapter 3

Interval Property Checking

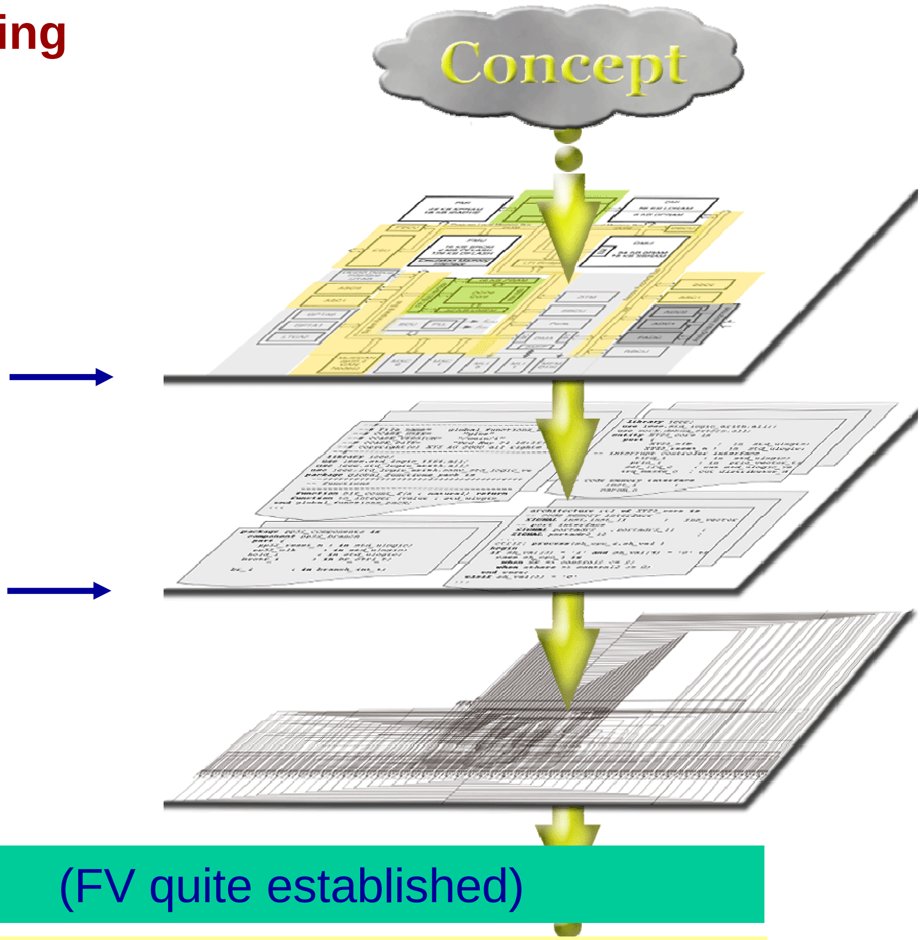
Verification Tasks

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)



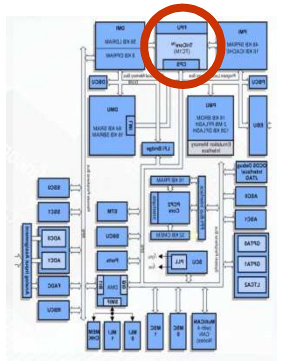
Approach:

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

RT-level module verification by IPC

A typical property for RT-level module verification:

$$a \rightarrow c$$



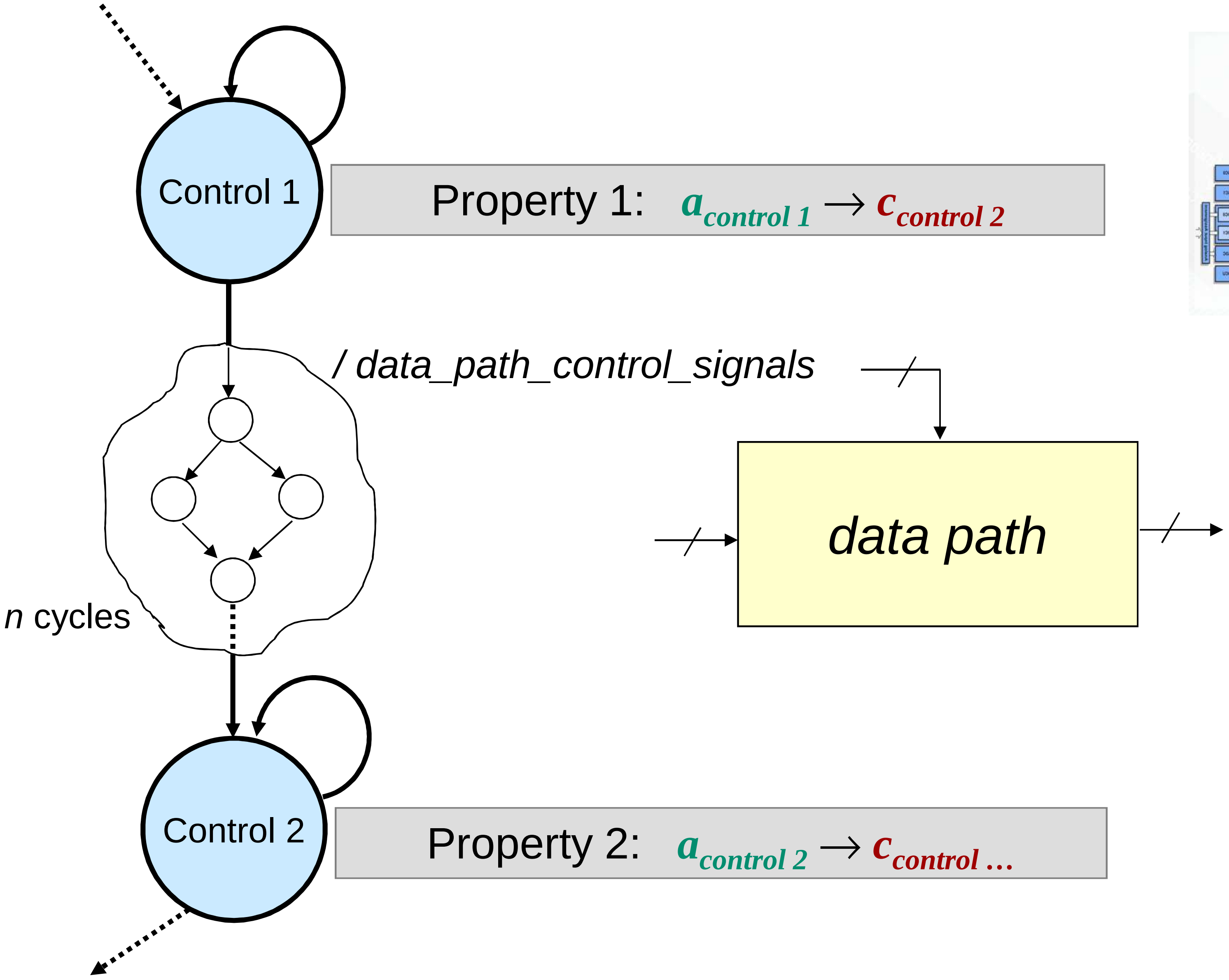
a : assumptions

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

c : commitments

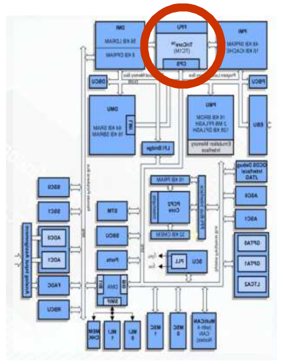
- module goes into certain control state *V'*
- certain outputs *Y* occur

RT-level module verification: operation by operation

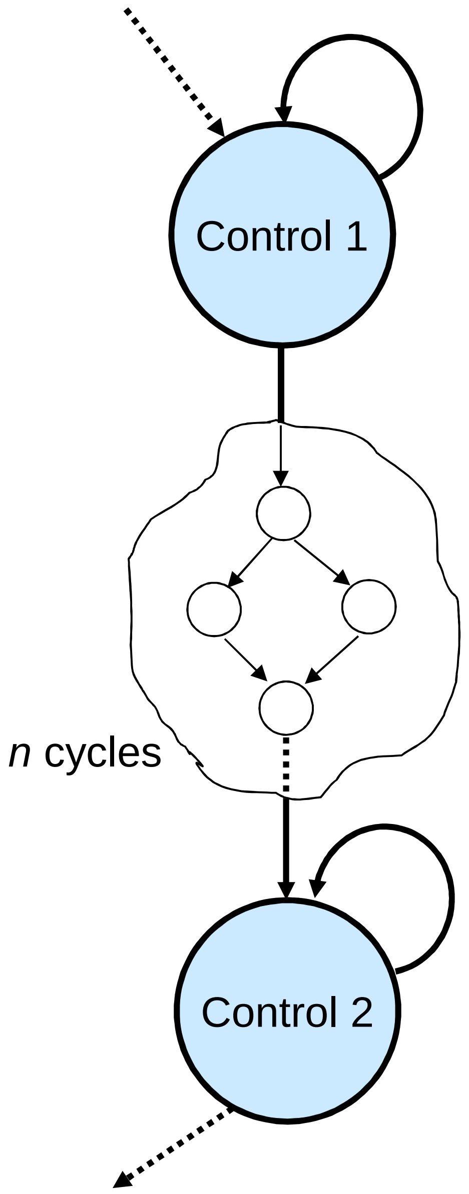


Property 1: $a_{control\ 1} \rightarrow c_{control\ 2}$

Property 2: $a_{control\ 2} \rightarrow c_{control\ \ldots}$



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

Property for RT-level module verification

SVA Operation property

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;

   ##0  $a_{start}(\boldsymbol{s})$   and        //starting state //
   ##0  $a_0(\boldsymbol{x})$      and
   ##1  $a_1(\boldsymbol{x})$      and
   ## ..
   ##n  $a_n(\boldsymbol{x})$
implies
   ##0 $c_0(\boldsymbol{x},\boldsymbol{y})$      and
   ##1 $c_1(\boldsymbol{x},\boldsymbol{y})$      and
   ## …
   ##n $c_n(\boldsymbol{x},\boldsymbol{y})$      and
   ##n $c_{end}(\boldsymbol{s})$;               //ending state //
endproperty;
```

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

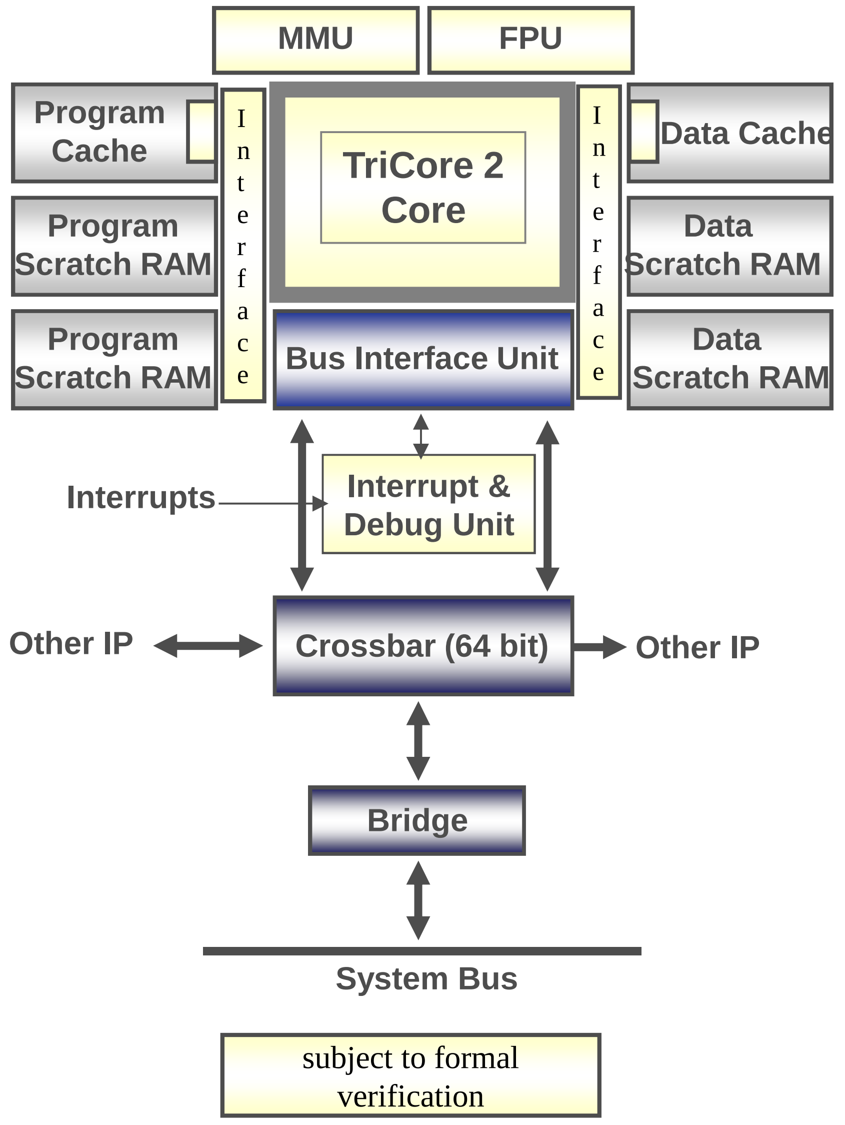
Formal Module Verification

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 needed to check completeness!)

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

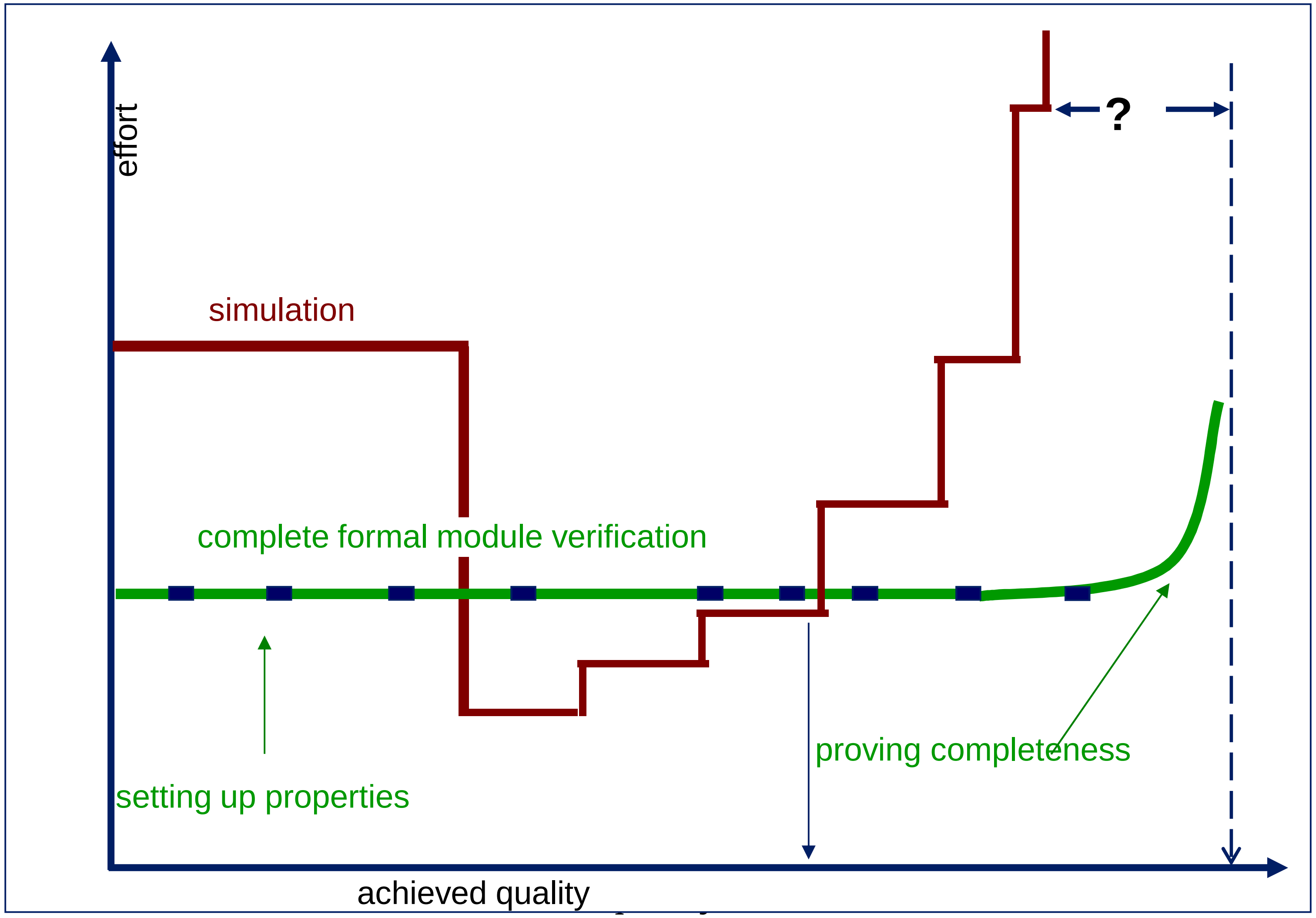
Current Implementation

- 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

Simulation vs. Complete Formal Module Verification



The Tricore processor – some project 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

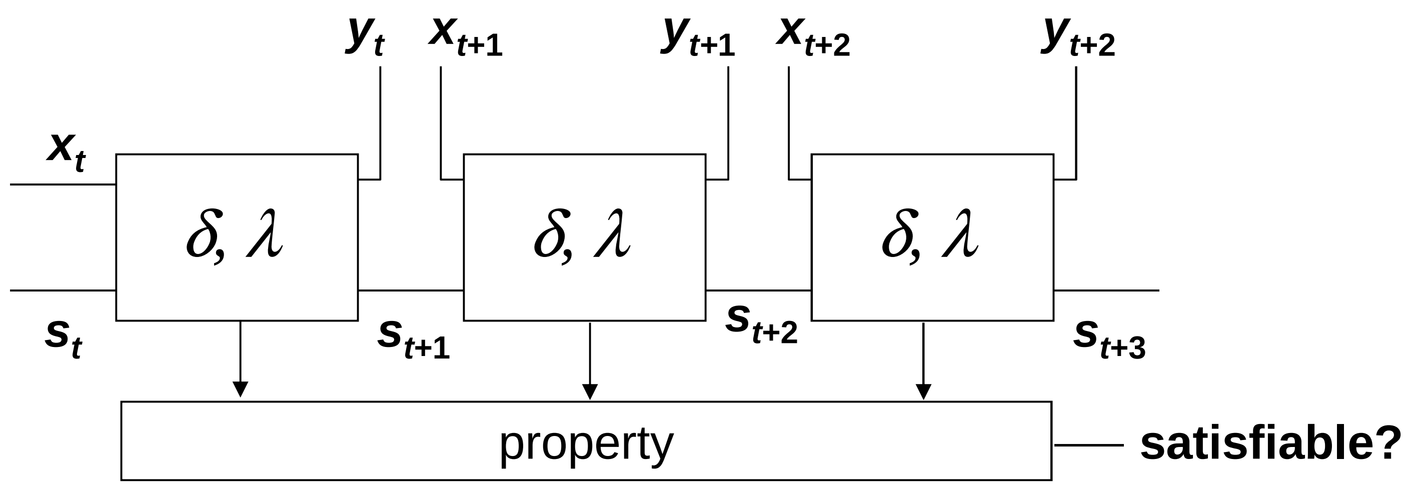
Productivity

- 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

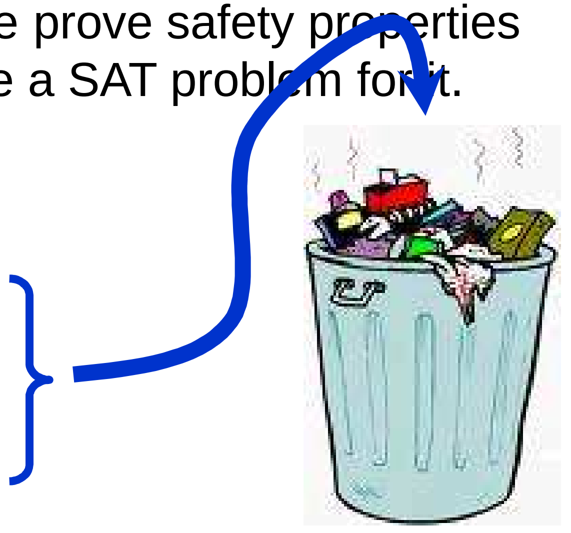
New-Generation Property Checking



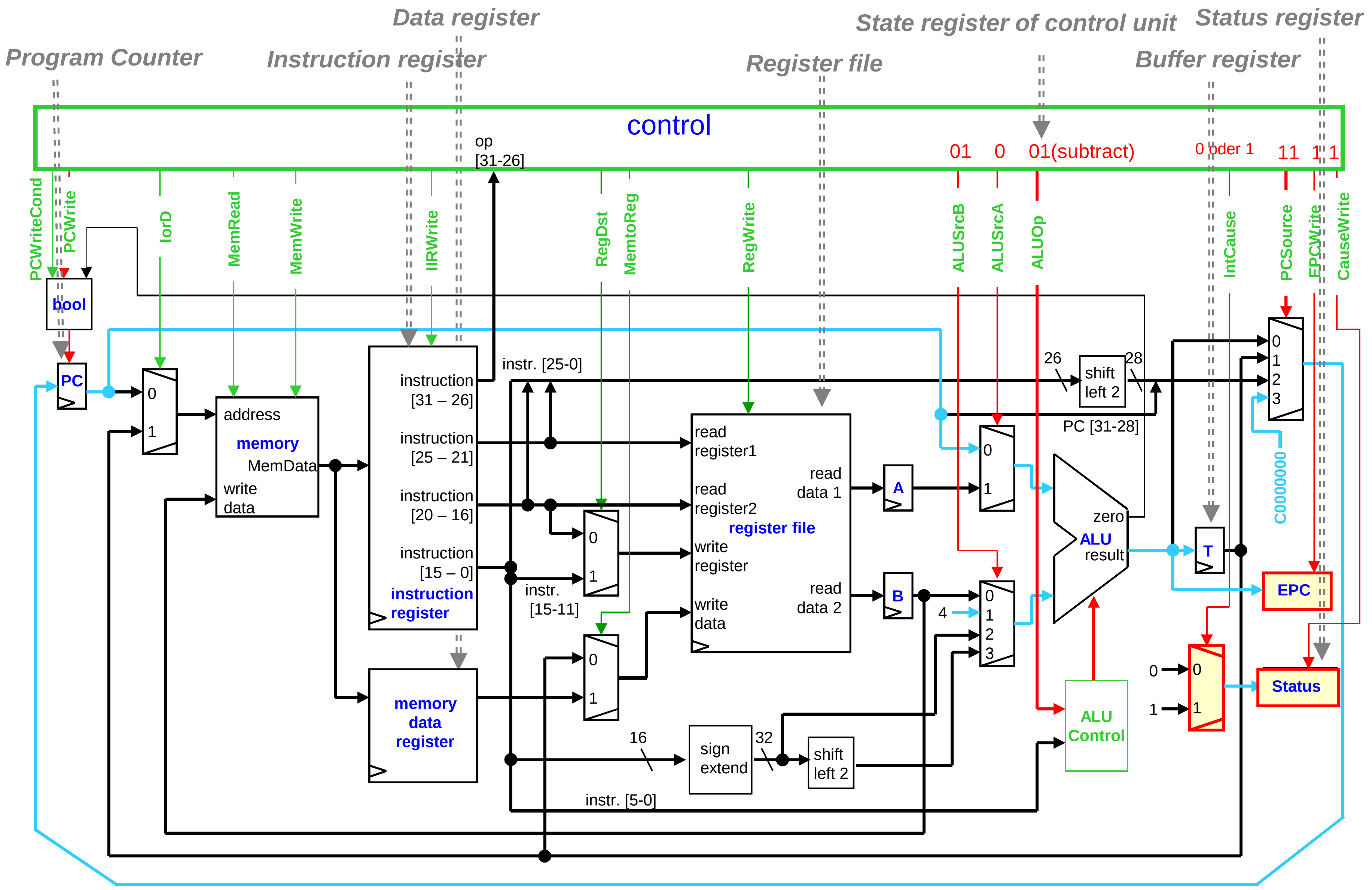
Interval Property Checking essentially means that we prove safety properties constructing a certain combinational circuit and solve a SAT problem for it.

So, what about all the classical notions of

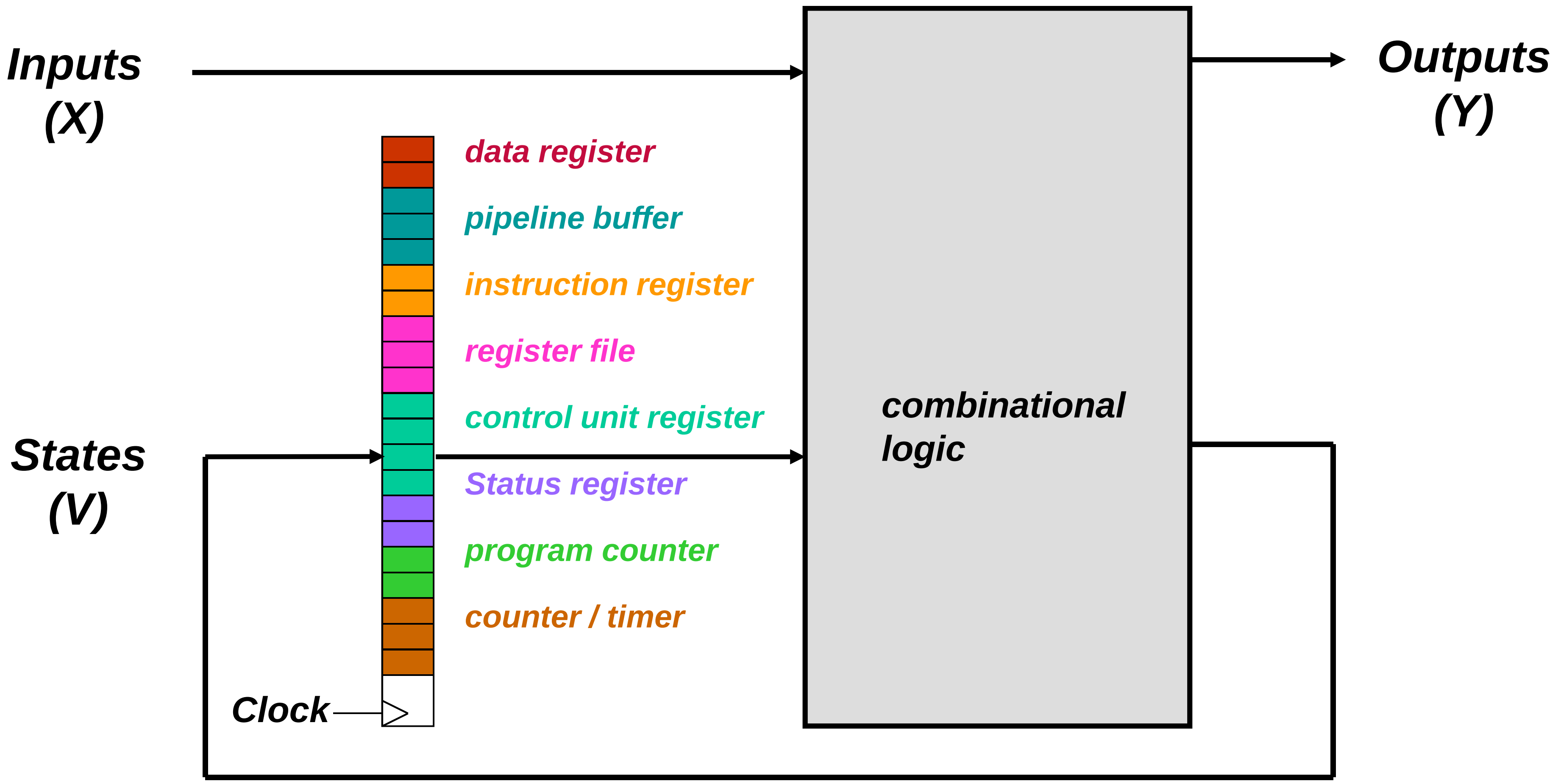
- reachability analysis
- representations and operations for large state sets
- finite state machine traversal techniques
-…



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.*

SVA operation property

Note:

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

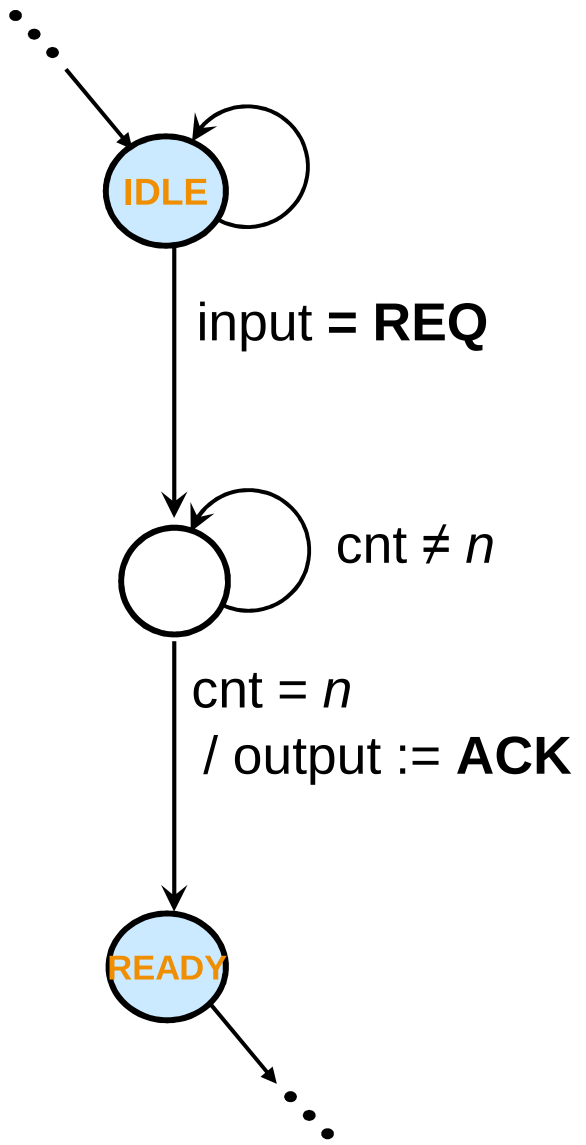
e.g., a property may specify the opcode bits of the instruction register as well as some bits of the control unit registers. Nothing is said about all other registers.

```
property myExample;

    ##0  a_start(S)  and  //starting state //
    ##0  a_0(X)      and
    ##1  a_1(X)      and
    ## ..
    ##n  a_n(X);
implies
    ##0  c_0(X,Y)    and
    ##1  c_1(X,Y)       and
    ## …
    ##n  c_n(X,Y)    and
    ##n  c_end(S);          //ending state //
endproperty;
```

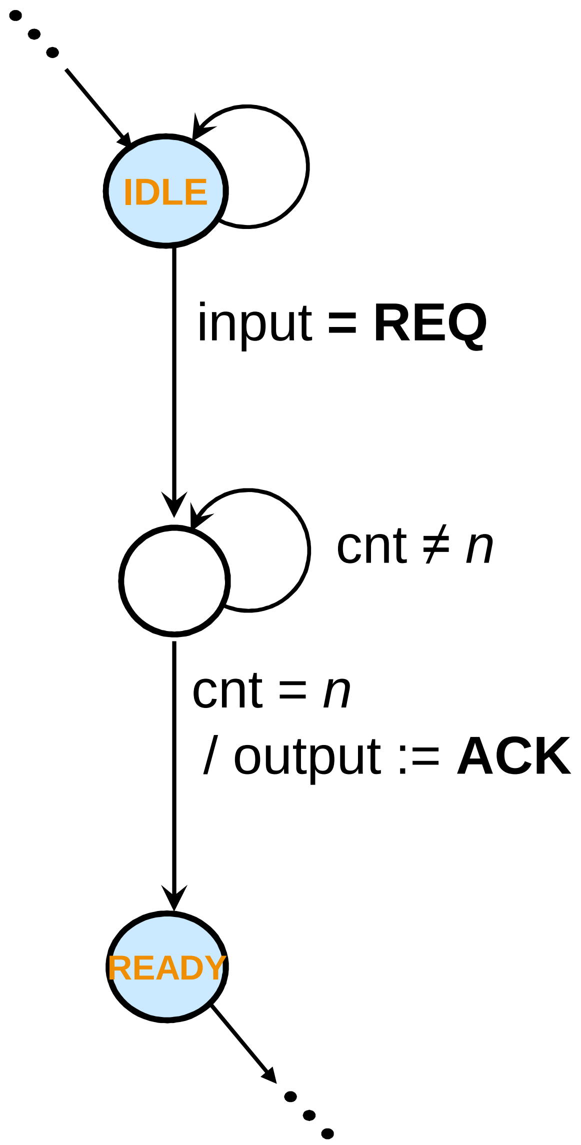
S: state variables, X: inputs, Y: outputs

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

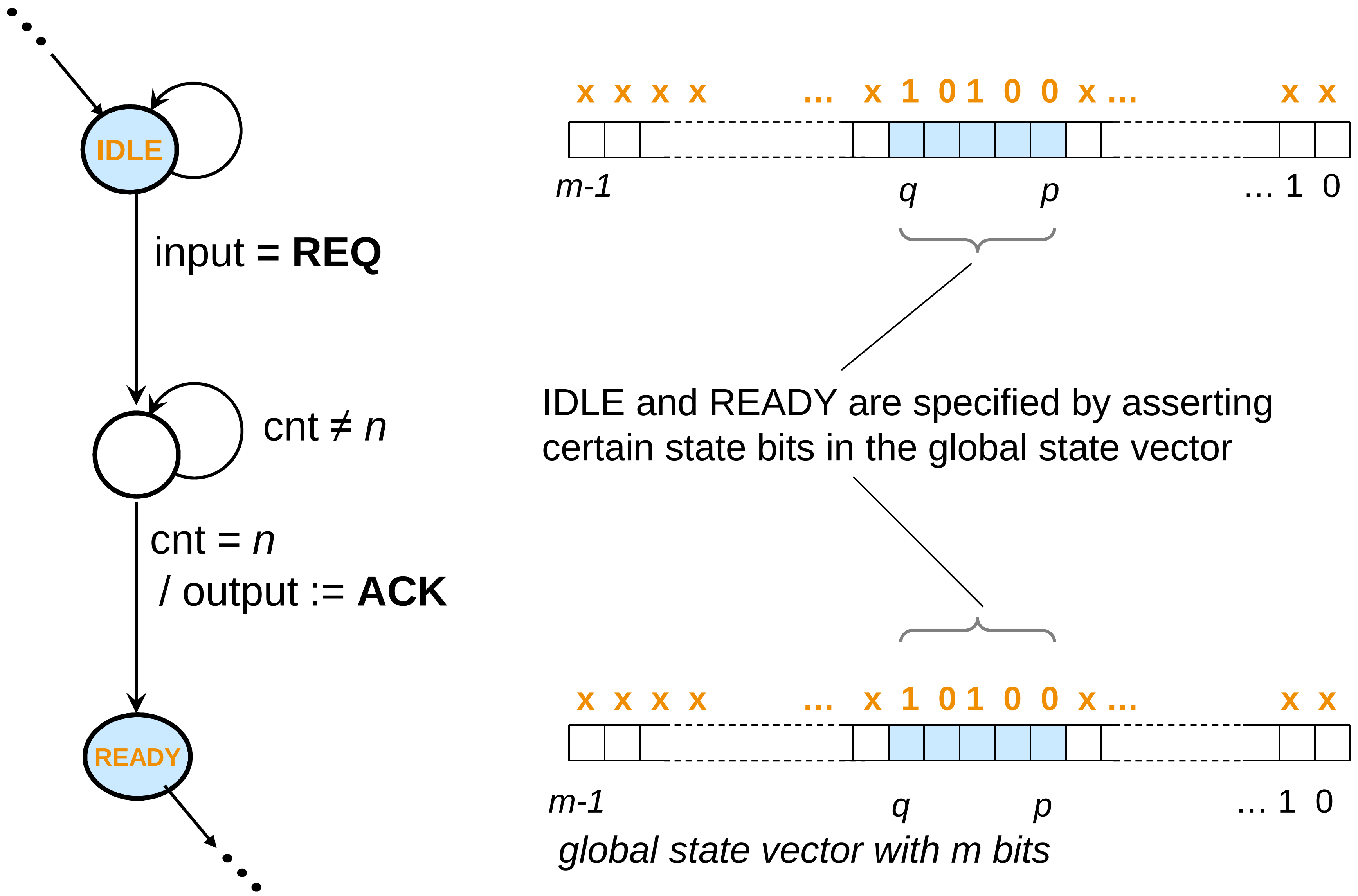


Property

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

Operation property with IDLE and READY as starting and ending states

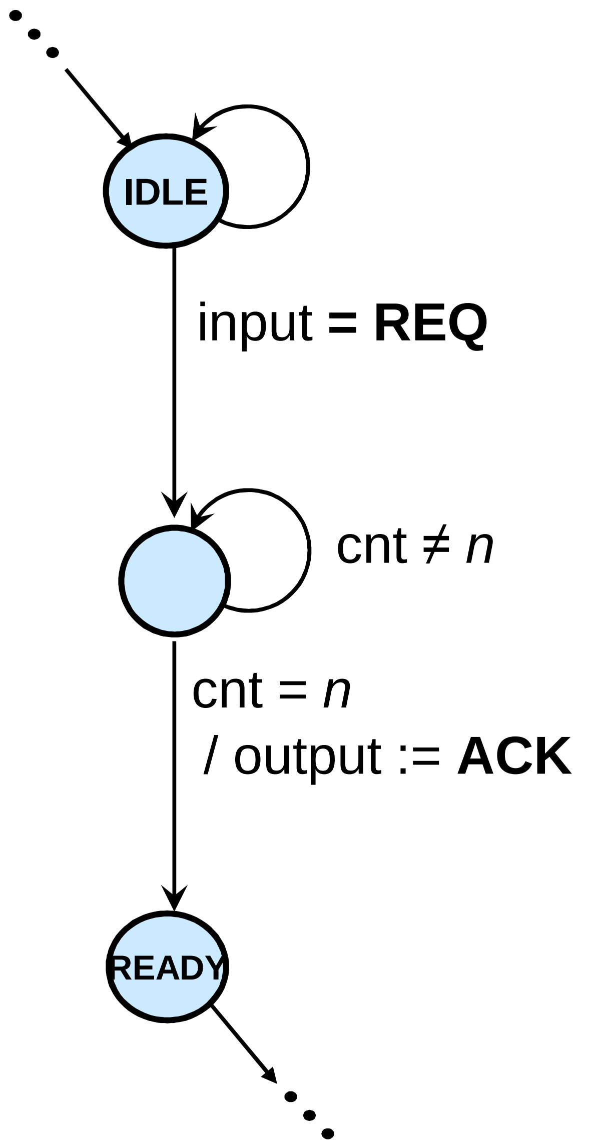
Example



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

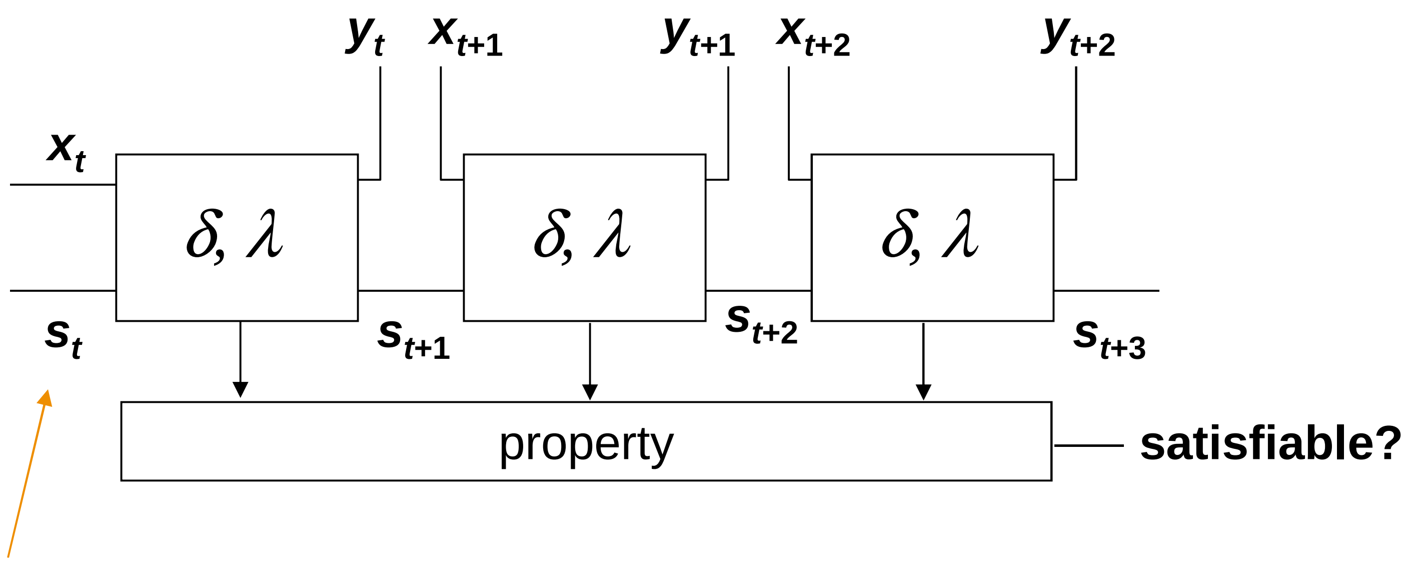
⇒ Is there a bug? Forgot to initialize the counter properly?

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

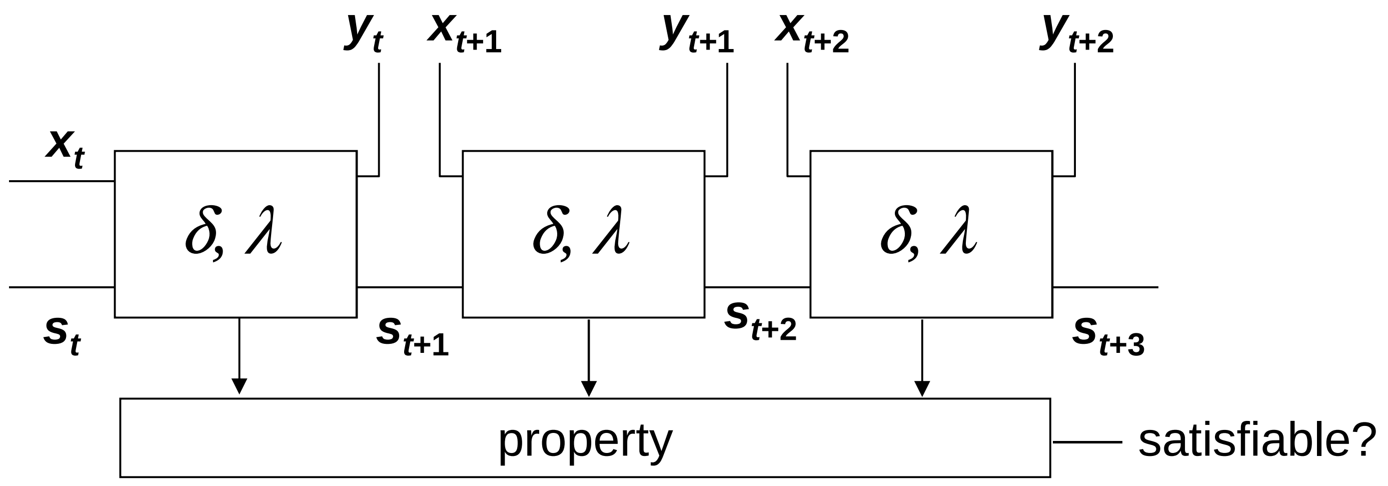
Do we still need this stuff?

- 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 …

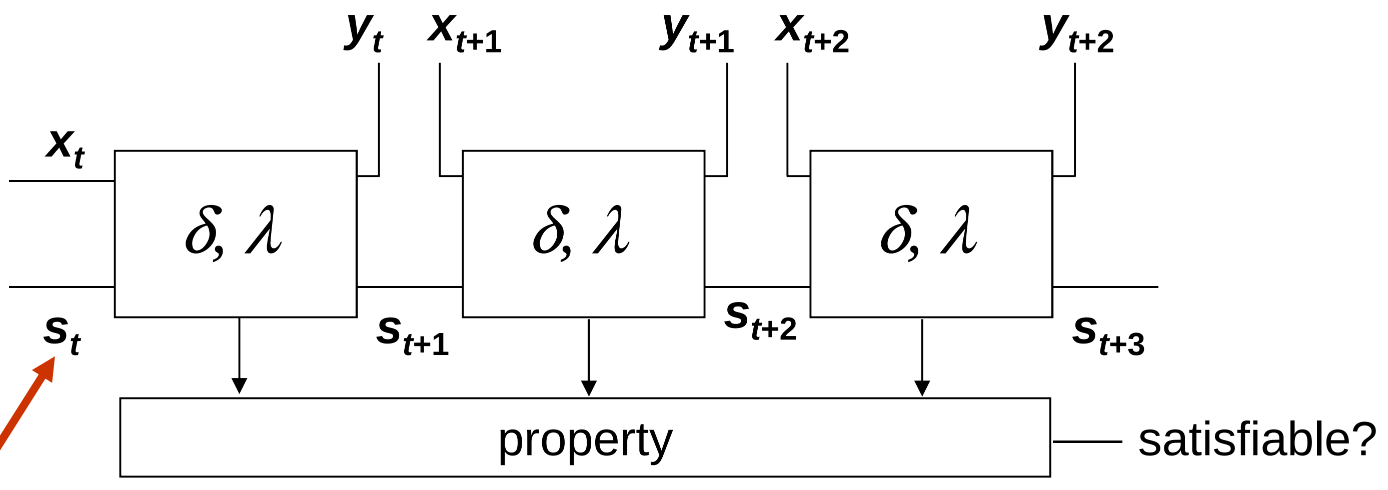
Interval Property Checking: The reachability problem



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
…

Interval Property Checking: The reachability problem



May need to add
reachability information („invariants“) here!

The iterative circuit model implicitly represents “some” reachability information but it generally over-approximates the reachable state space.

It practice we can often fix the problem by adding some reachability information at the beginning of the iterative circuit model. This can be done manually or, sometimes, automatically.

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.

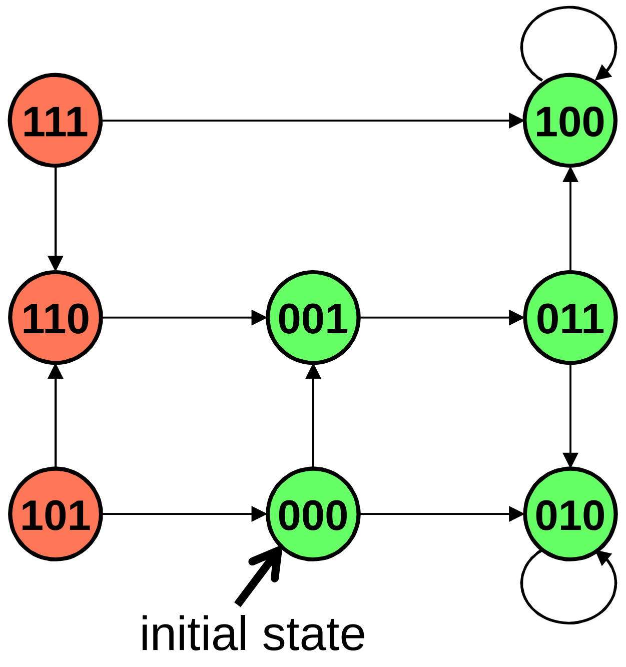
Example:

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:

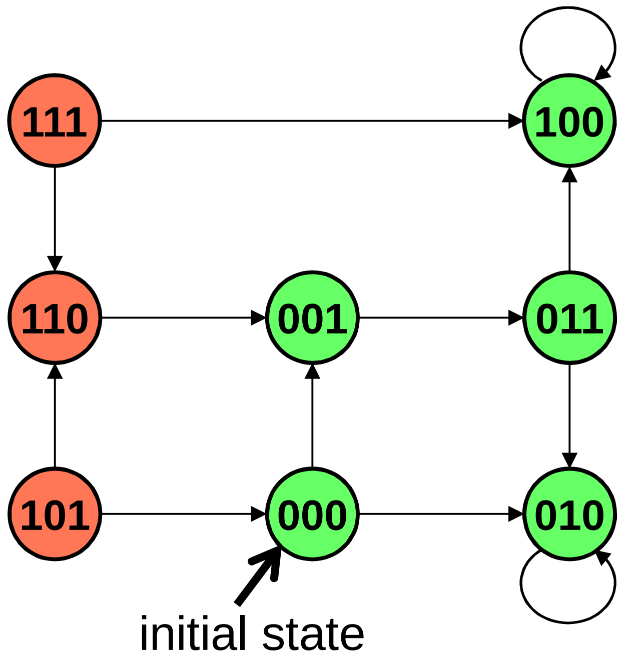
R = {000, 001, 010, 011, 100}

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 Properties with Invariants

Let *p* be a Boolean formula. We want to prove that *p* holds in every reachable state of the system.

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 property.

Which invariants of the previous example can be useful to prove the 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 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 properties based on invariants that over-approximate the reachable state set.

Proving 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)

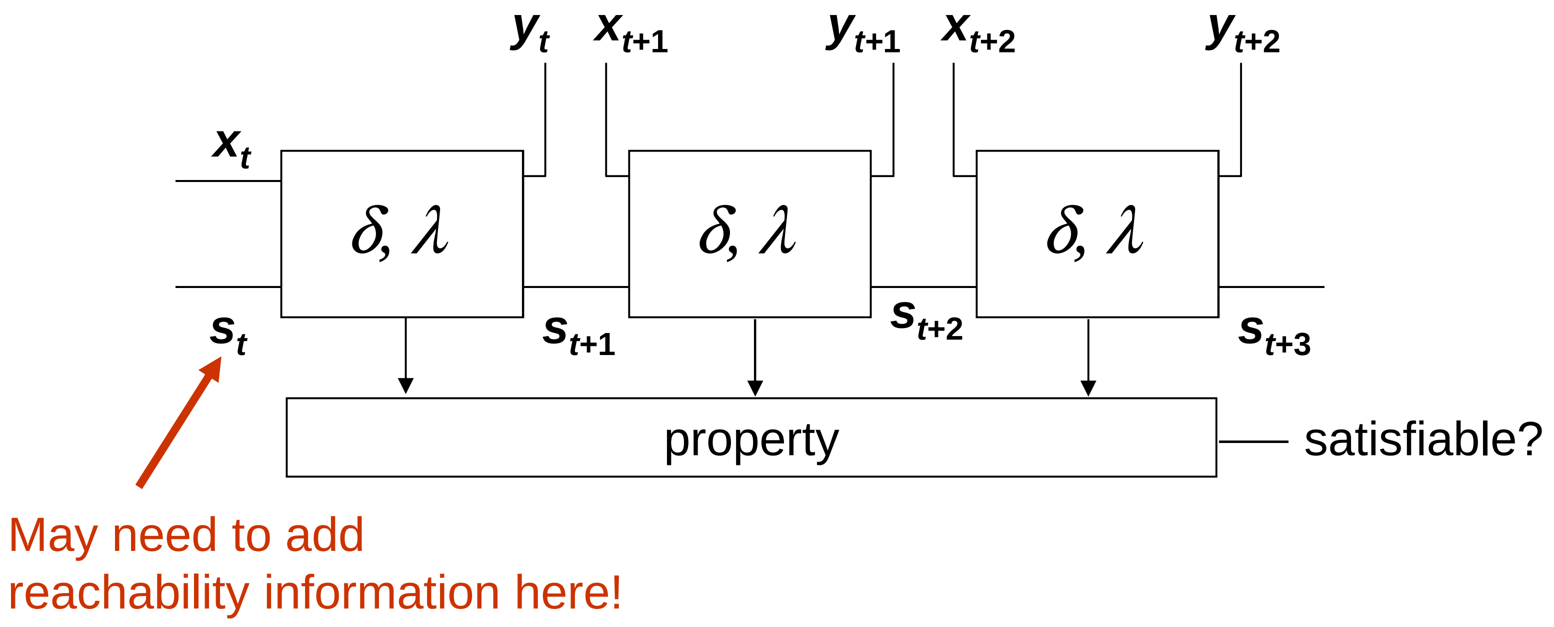
$\Rightarrow$ there is a bug in the design ("True Negative")

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

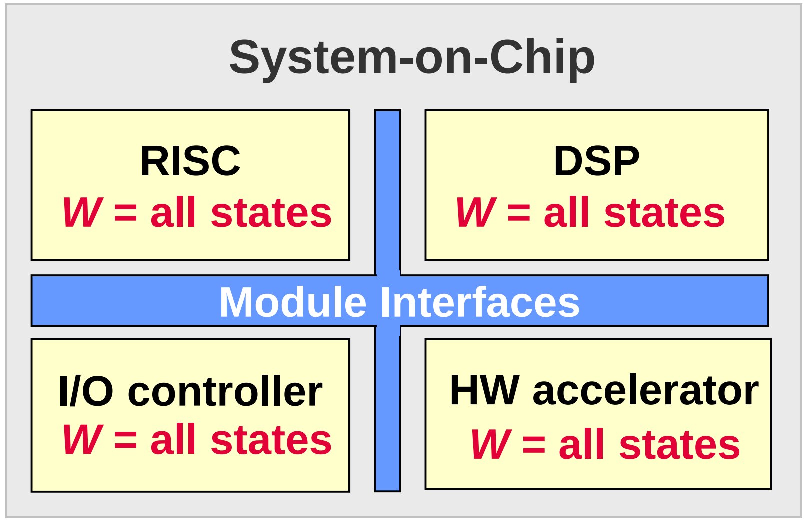


This reachability information is added in terms of an invariant!

Interval Property Checking

The good cases…

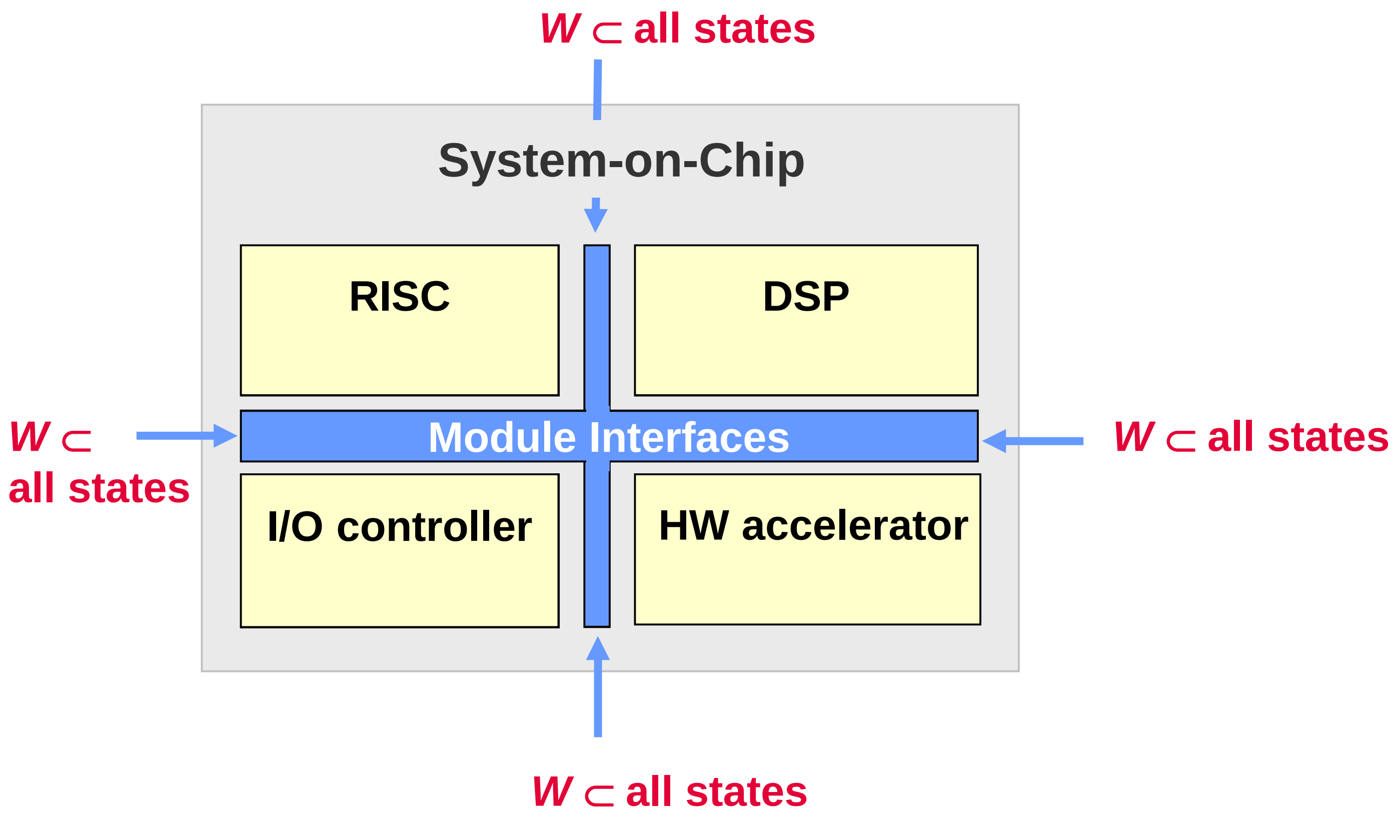
In the most simple case, no reachability information at all is required at the beginning of the iterative circuit model, i.e., W = all states.
(W_{10} in previous example)



W = all states holds in most cases when verifying computational modules (processors, hardware accelerators, …)

Interval Property Checking

The difficult cases…



For communication modules (implementations of SoC protocols) the invariants often need to be refined: $W \subset$ all states

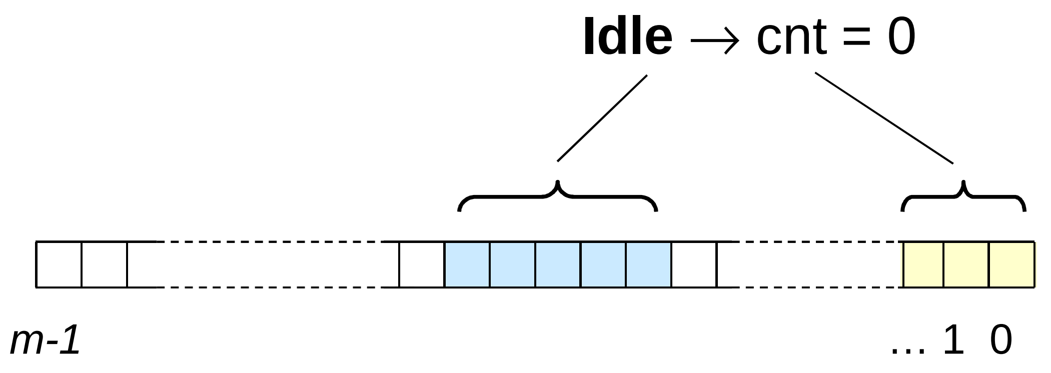
IPC with Invariants

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.

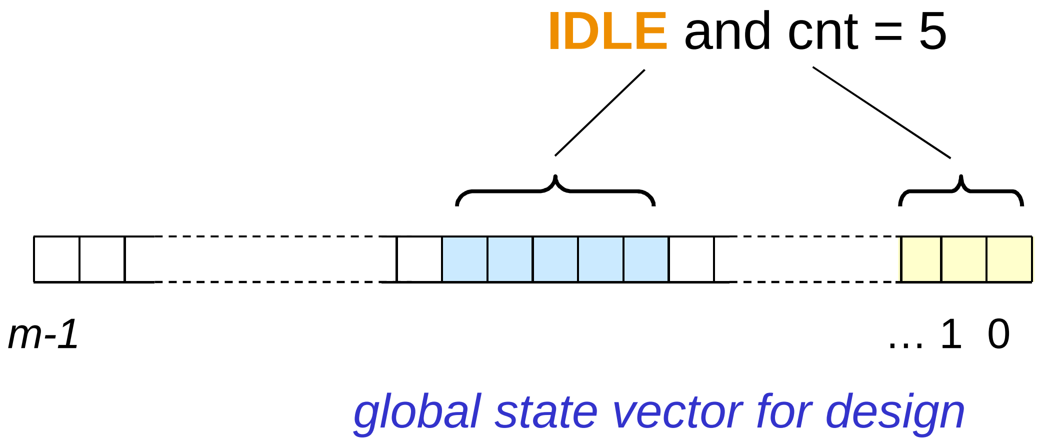


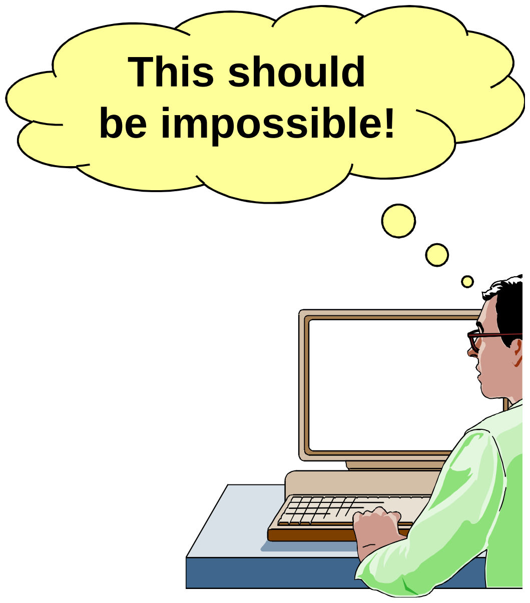
global state vector for design

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

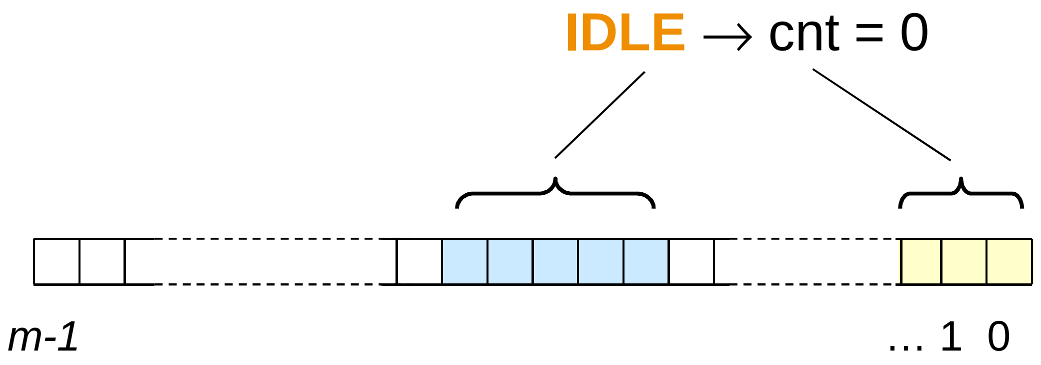




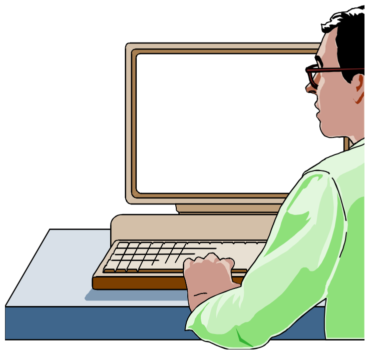
IPC with Invariants

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

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



global state vector for design



IPC with Invariants

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

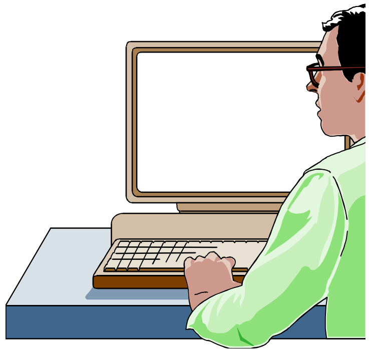
Step 3:
Prove the reachability constraint by induction.

property 1 (base case)

Assume: initial state
Prove: $IDLE \rightarrow cnt = 0$

property 2 (induction step)

Assume:
at t: $IDLE \rightarrow cnt = 0$
Prove:
at t+1: $IDLE \rightarrow cnt = 0$



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

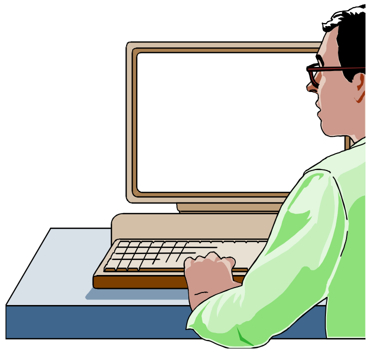
IPC with Invariants

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

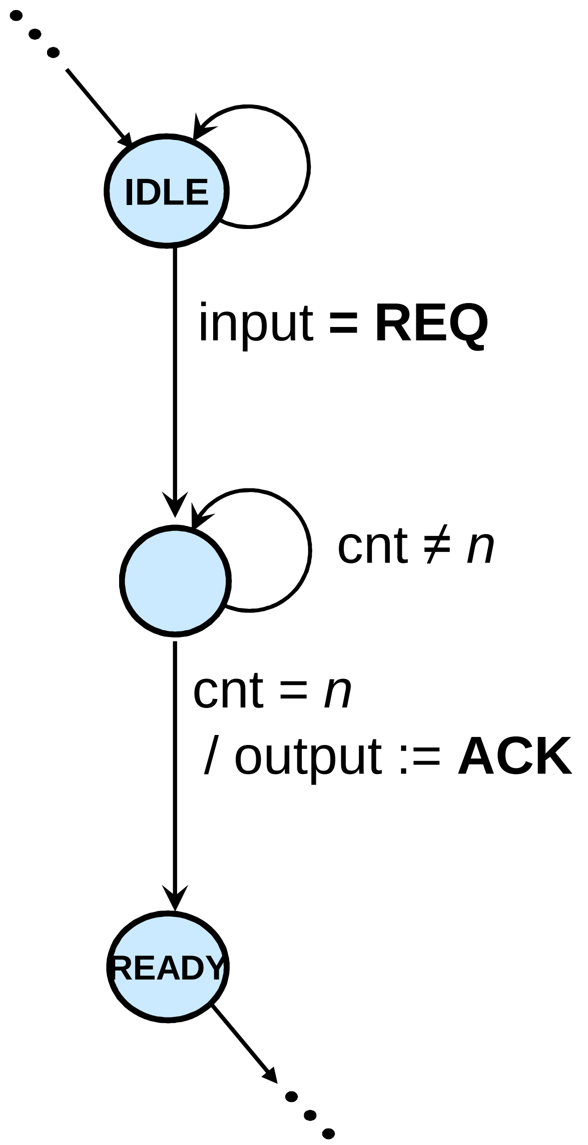
Step 4:
Prove the original property using the reachability constraint. This means that you prove the property for all states of the design that fulfill the constraint. Since the constraint is proved to be valid for the design it means that your proof is based on a state set that is an invariant and which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

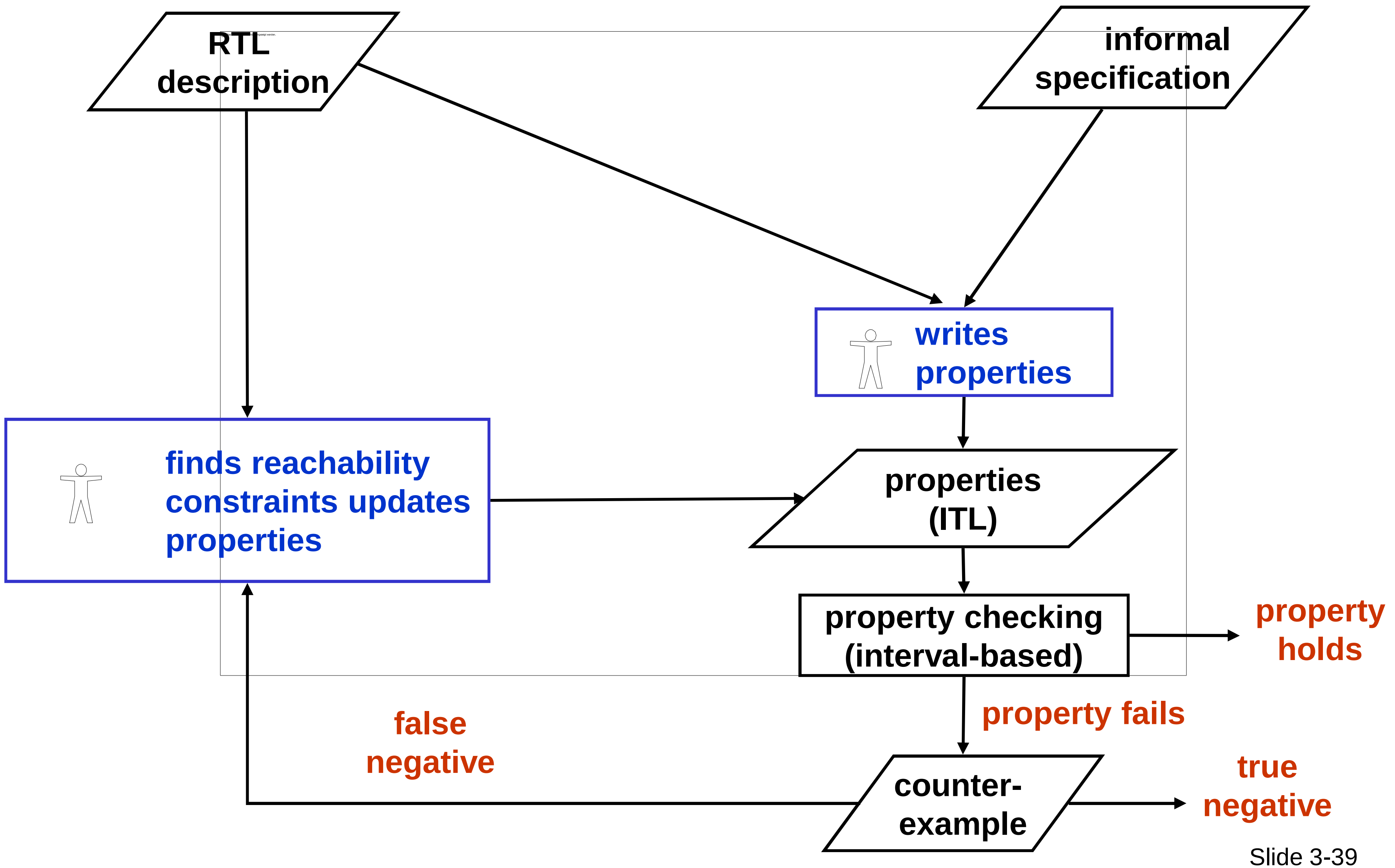


```
assertion reachabiliy_constraints :=
  if state = IDLE then cnt = 0 end if;
end assertion;

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

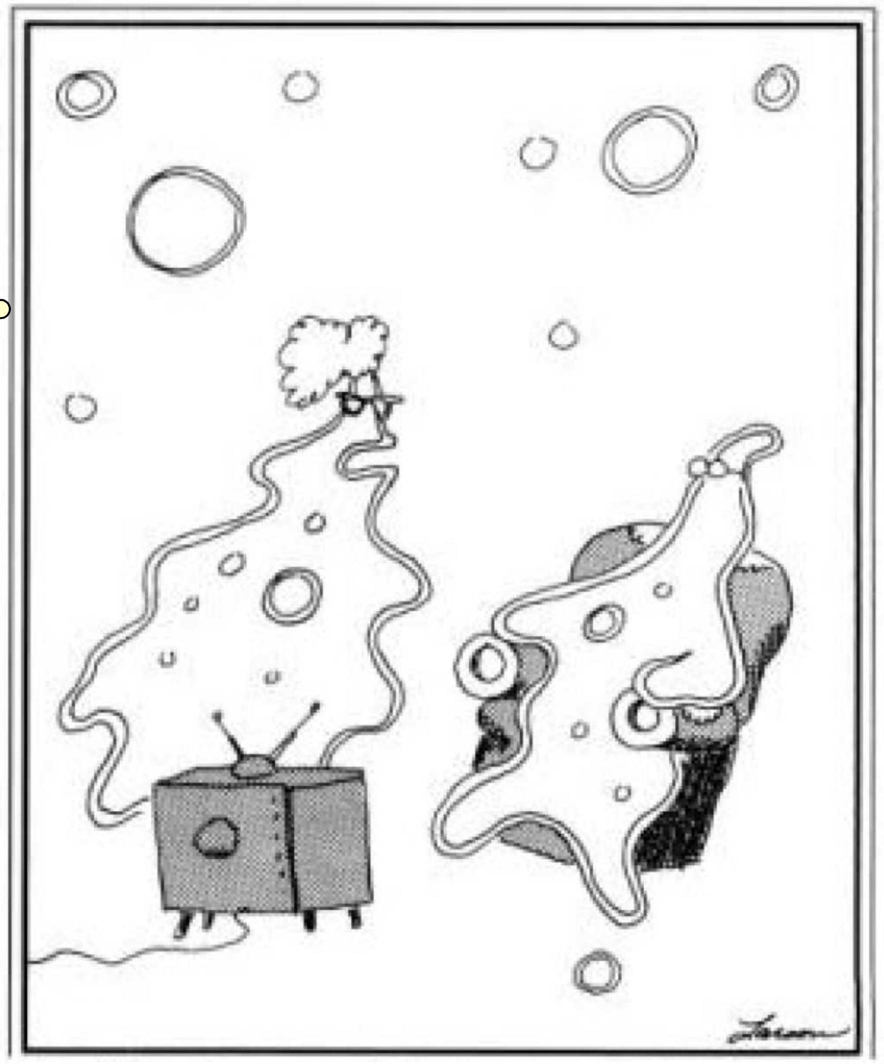


Industrial Experiences

How to integrate FV in ESL-based design flows?

Verification engineer's preferences

White-box versus black-box verification



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

Electronic System Level (ESL)

The problem

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

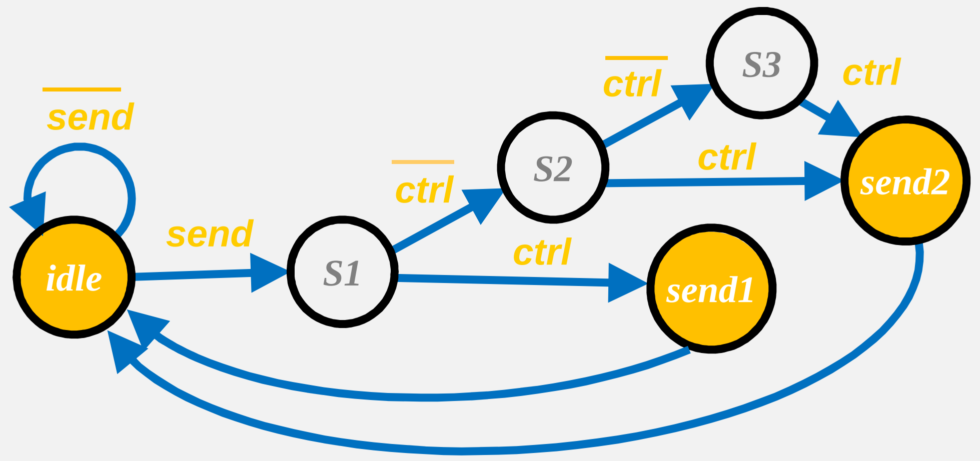
$\rightarrow$ high costs (in spite of IP re-use)

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

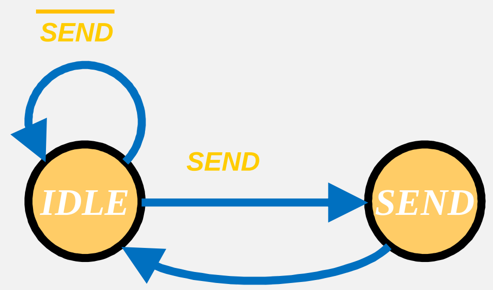
$\rightarrow$ ESL models do not reduce costs of RTL design
$\rightarrow$ ESL models do not reduce costs of RTL verification

ESL – RTL: Closing the semantic gap by property checking

RTL implementation → ***ESL abstraction***



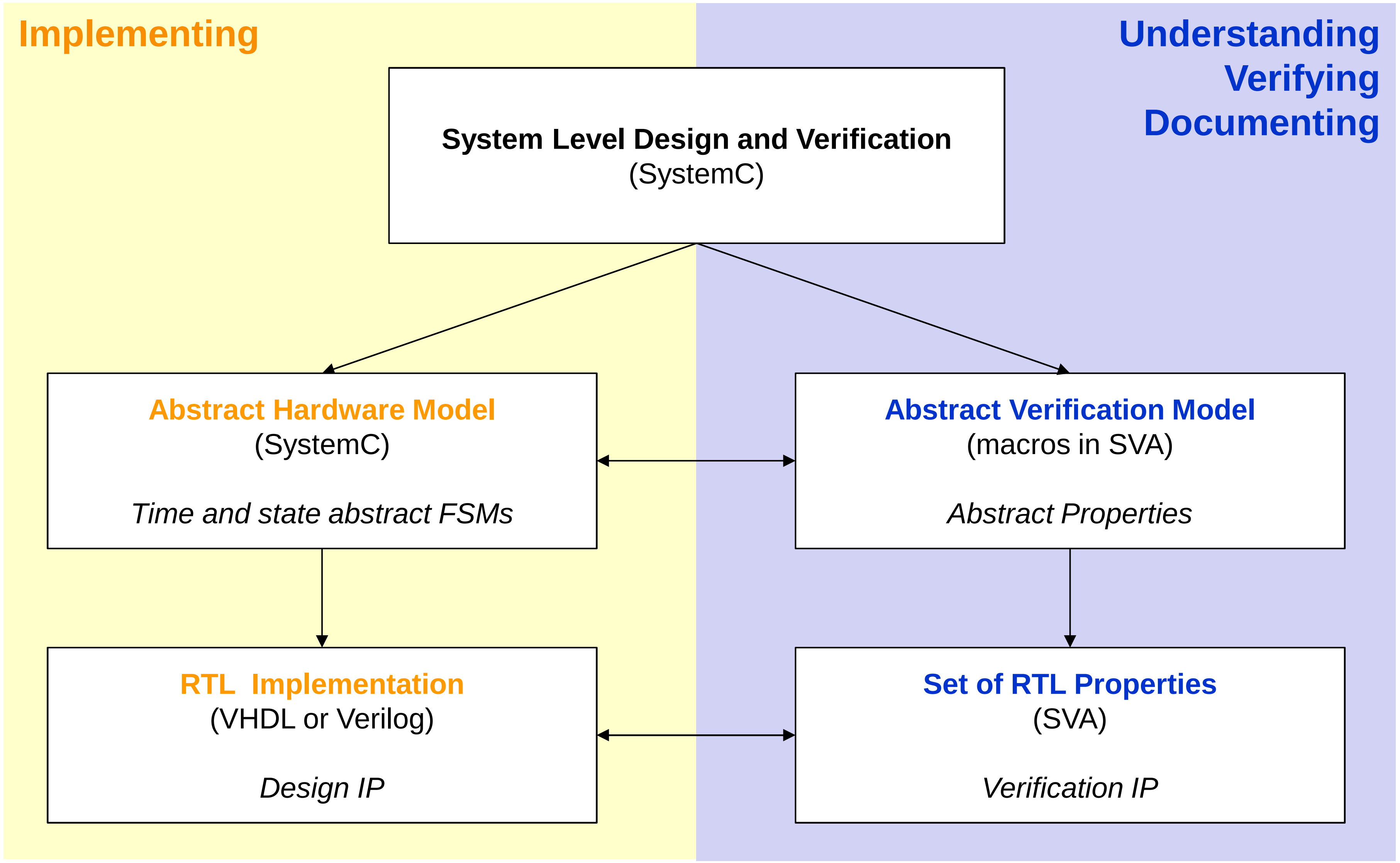
→



set of important concrete states → abstract state

“operational” finite path between important states → abstract transition

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

Research Directions

- 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
- But coverage in property checking is key!

Welcome

to a project, internship or Master's thesis!!

- **conducted in Kaiserslautern**
- **possibly in collaboration with industrial companies**

Research Collaborations:

OneSpin SOLUTIONS

Infineon technologies

Alcatel·Lucent

BOSCH

Audi

intel®

IBM®