The Big Hurdles for Formal Verification in Practice –

Can we overcome them in system-level design flows?

Wolfgang Kunz

Joint work with: Dominik Stoffel, Joakim Urdahl

Electronic Design Automation
Dept. of Electrical and Computer Engineering

TU Kaiserslautern, Germany

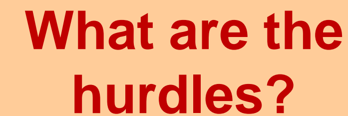
Formal Property Checking

After decades of high-caliber research:

- powerful proof algorithms can handle SoC modules of realistic size
- adequate specification languages (e.g. PSL, SVA) are available

After years of industrial development:

- sophisticated property checking methodologies are available
- past industrial case studies have shown high productivity of FV at the RTL

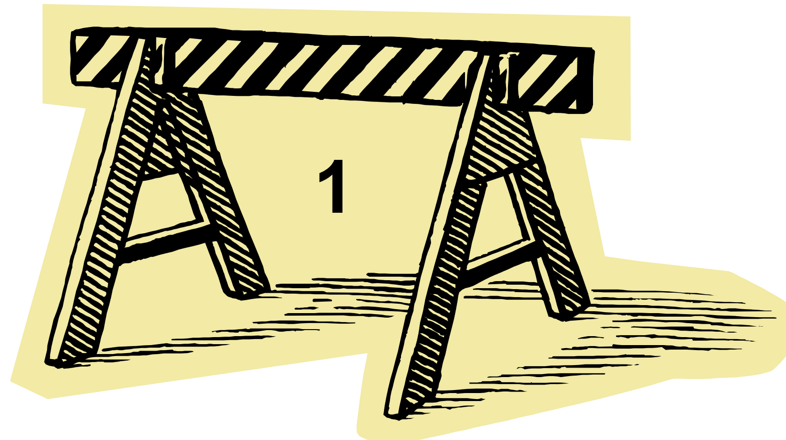


But:

- formal property checking is *only moderately successful*

RTL Property Checking

Double Efforts



Formal **+** **Simulation**

Formal
- examine corner cases
- thorough code analysis using coverage metrics

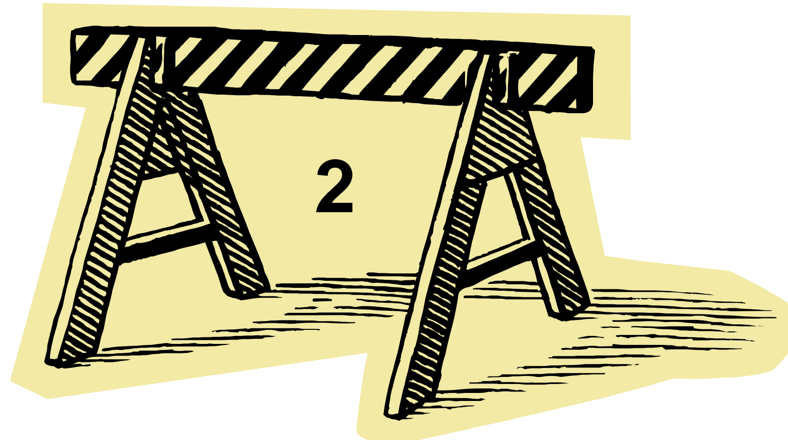
Scope: SoC module-level

Simulation
- probe functional behaviors

Scope: chip-level

RTL Property Checking

Limited Use of formal Verification IP impedes amortizing its costs



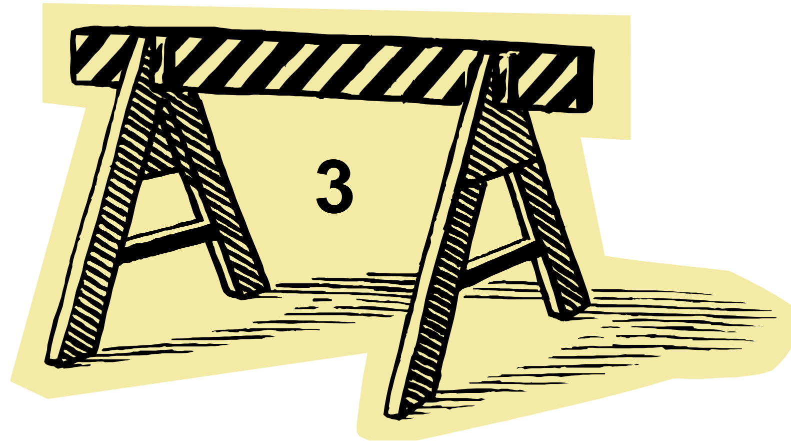
Formal Verification IP (property suite)

examines *functional* correctness, but what about *non-functional* design goals?

- low-power
- safety / reliability
- performance
- …

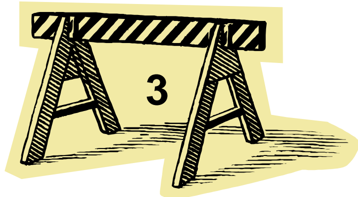
RTL Property Checking

Limited Acceptance with Verification Engineers

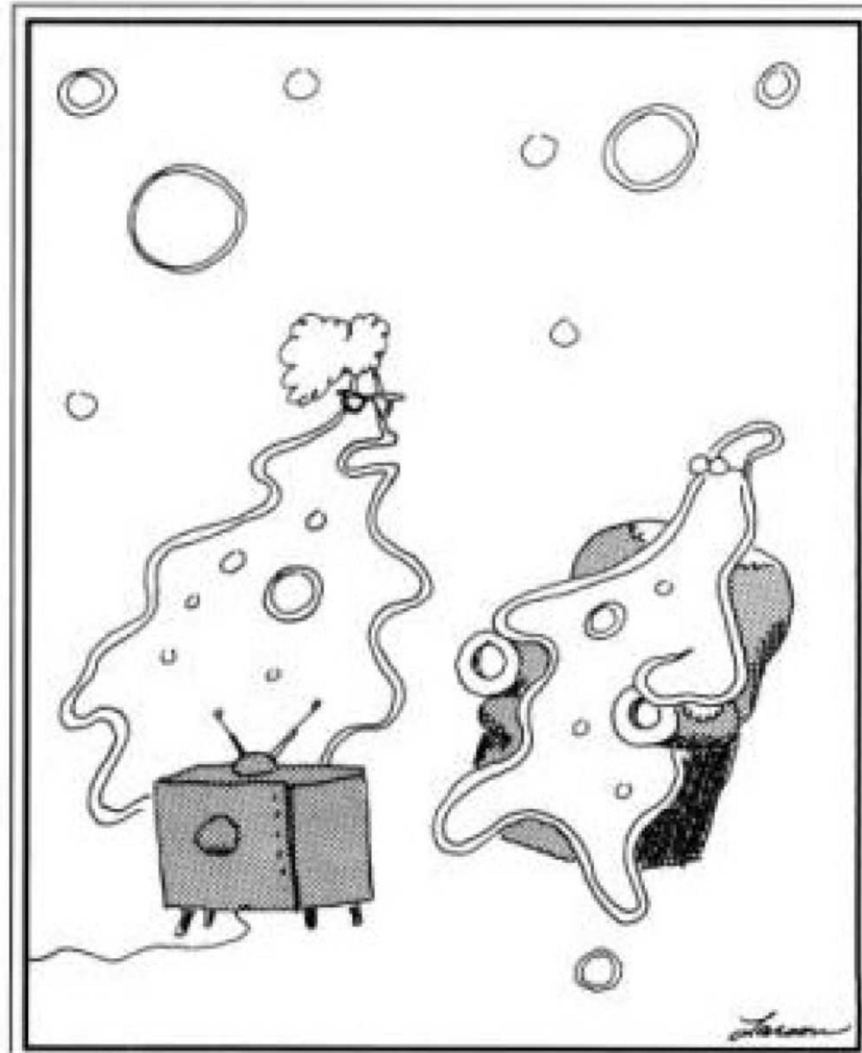


… in spite of proven capabilities in finding critical bugs missed by simulation!

Preferences in the Verification Flow



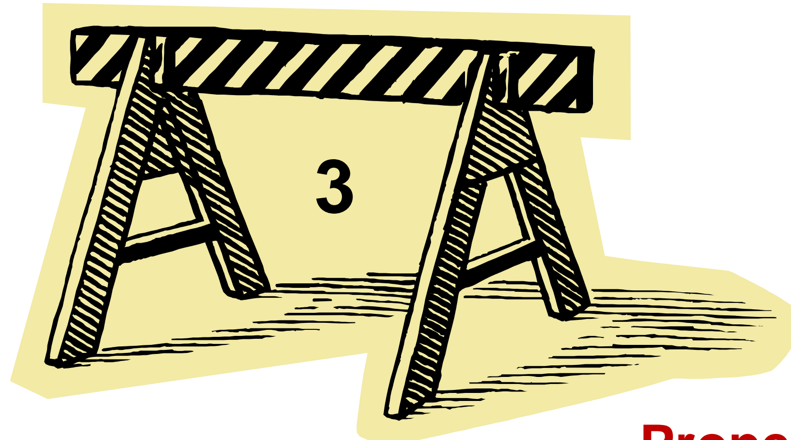
White box versus black box verification



“Stimulus, response! Stimulus, response! Don‘t you ever think?”

RTL Property Checking

Limited Acceptance with Verification Engineers



Property Checking is actually more like *design* than like *verification* !

For good reasons:

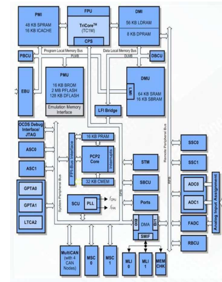
- Verification traditionally (and ideally) is black box (as is the case with simulation)
- Property Checking is white box

Moving FV closer to Design

- Industrial practice: Assertion-Based Verification (ABV)
  - instrument RTL code with (local) assertions, prove formally when possible, simulate otherwise
- Integrate FV into System-Level Design
  - abstraction from implementation details mitigates white box / black box problem

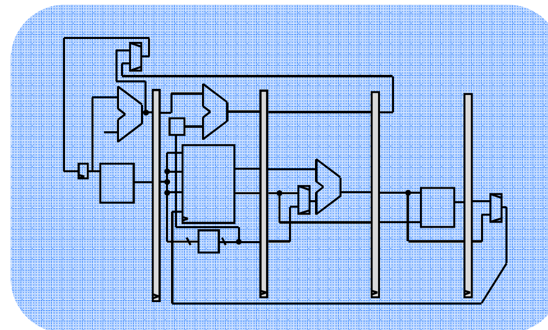
Bottom-up SoC Verification

System Level



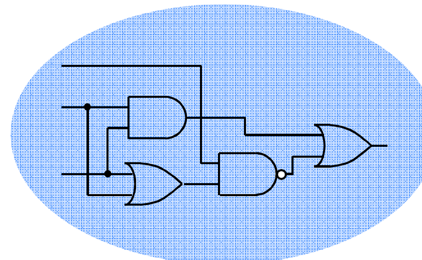
Verify entire system:
- simulation
- property checking

Register Transfer Level



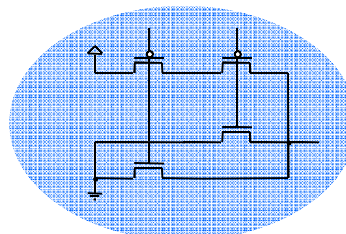
Verify microarchitecture:
- simulation
- property checking

Gate Level



Equivalence checking

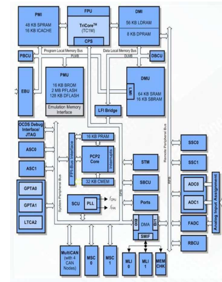
Transistor Level



Verification of electrical behavior à la SPICE

Bottom-up SoC Verification

System Level



Verify entire system:
- simulation
- property checking

But this is not really how it is in practice:

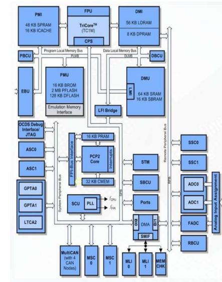
we do not verify the system behavior at the system level
but at the RT level

Why?

we do not trust our system models as we trust, e.g., our gate models

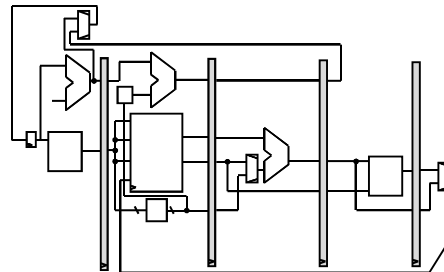
Bottom-up SoC Verification

System Level



Verify entire system:
- simulation
- property checking

Register Transfer Level



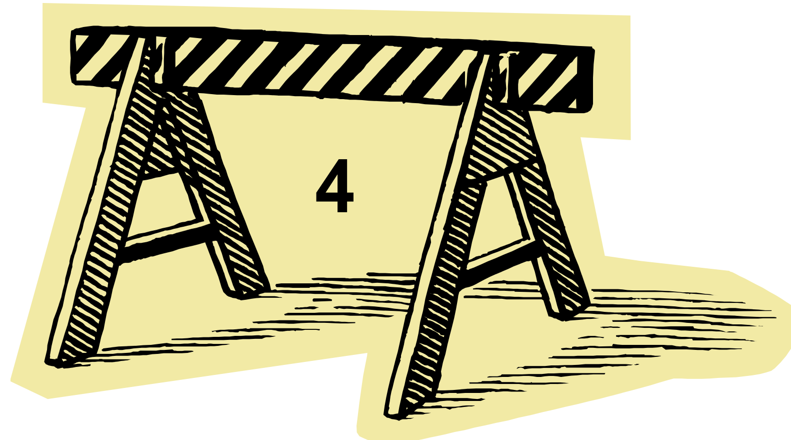
Chip-level simulation at RTL is needed!

⇒ RTL remains the reference for system verification

„Verification Gap": verification dominates design costs

System Level Property Checking

"Semantic Gap" between System Level and RTL



No formal relationship between high-level models and RTL implementation

→ RTL design process as costly as before
→ RTL verification as costly as before

Notable exception: certain data path applications where high-level equivalence checking is applicable

Can we overcome the hurdles in system-level design flows?

To do:

- Create abstract Verification IP already during system level design
- Refine it during System-to-RTL design process

→ white box design knowledge easily available

→ shift global verifications tasks from RTL to System Level and avoid double efforts at the RTL

→ leverage verification IP already during RTL design phase to support aggressive optimization w.r.t non-functional design targets

- **But only if:** the semantic gap between RTL and System Level gets closed!

Closing the semantic gap between system level and RT level

The Idea

define the semantics of the system model in terms of objects of the RTL description

by

formulating properties for the RTL model in a standard property language, e.g. SVA

The properties describe behavior closely related to register transfers and not the system behavior (since they describe the RTL model)!

Path Predicate Abstraction (PPA)

The notion of path predicate abstraction is used to describe the formal relationship between system model and RTL model.

Our agenda for the next slides:

- path predicate abstraction is defined as a specific relationship between two (partially) colored graphs

Operational Graph Coloring

Definition: operational graph coloring

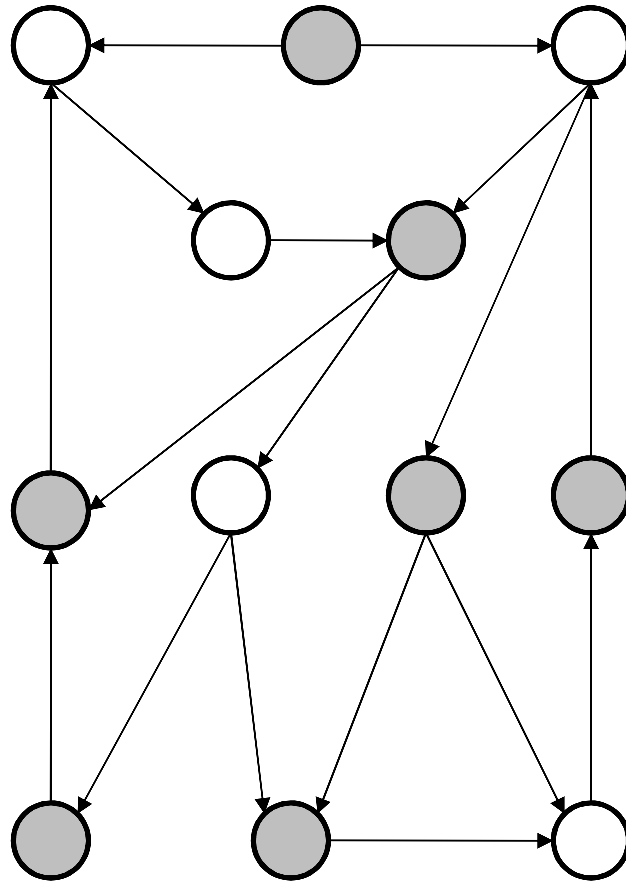
Consider a graph $G = (V, E)$ and a coloring function c: $W \rightarrow \{c_1, c_2, \ldots\}$, with $W \subseteq V$. A path $(v_0, v_1, \ldots, v_n)$ such that $v_0, v_n \in W$ and $v_1, \ldots, v_{n-1} \in V \backslash W$ is called *operational path* in G.

W must be chosen and labeled with colors $\{c_1, c_2, \ldots\}$ such that:

1. every cyclic path in G contains at least one node from W
2. for every operational path $(v_0, v_1, \ldots, v_n)$ and $u_0 \in W$ such that $c(u_0) = c(v_0)$ there must exist an operational path $(u_0, u_1, \ldots, u_m)$ in G with $c(u_m) = c(v_n)$

A graph G with such a coloring function c is called ***operationally colored*** graph.

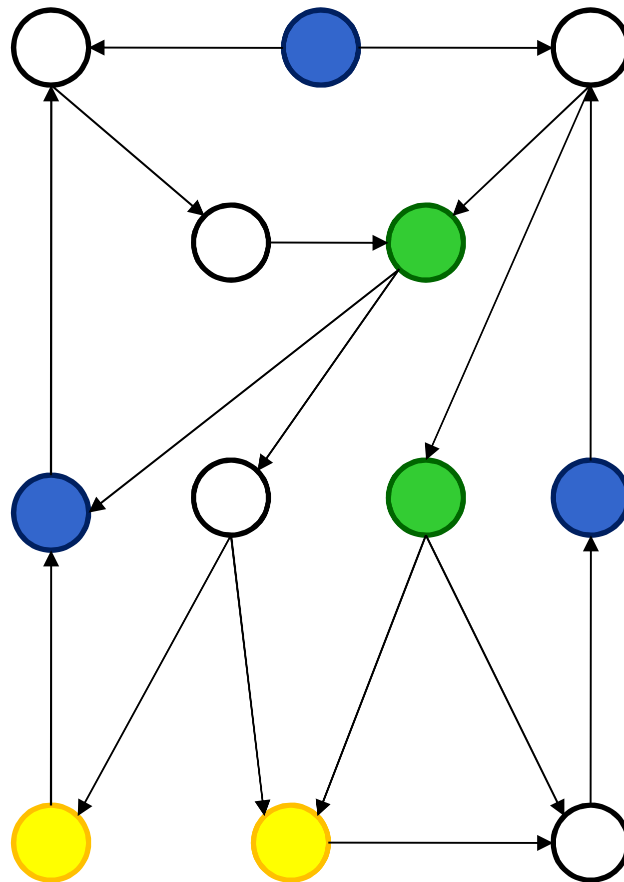
Example



Given a graph G where nodes in W are shaded grey. Find an operational graph coloring with colors:

- **blue**
- **green**
- **yellow**

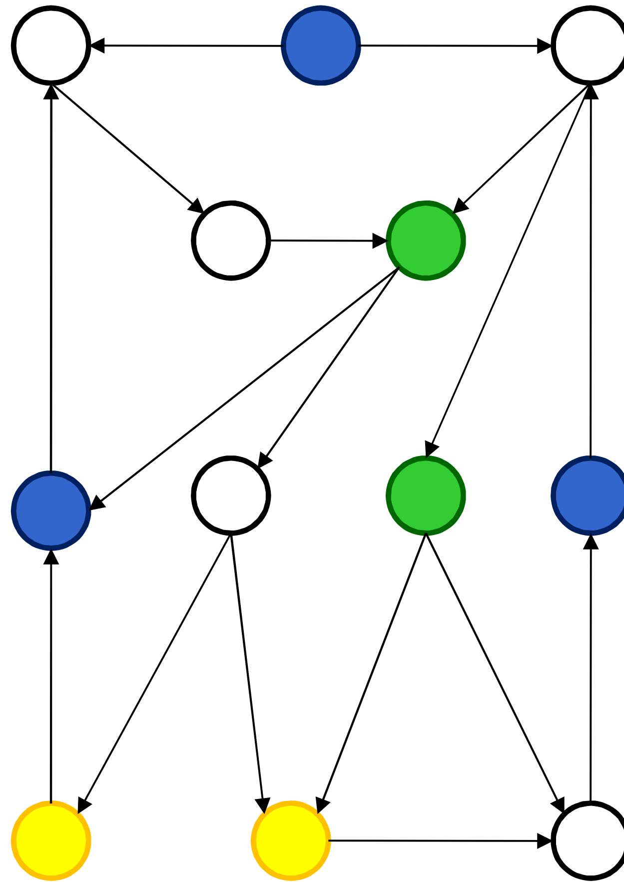
Example



Given a graph G where nodes in W are shaded grey. Find an operational graph coloring with colors:

- **blue**
- **green**
- **yellow**

Example – cont'd.



"Operations":

- **blue** goes to **green**
- **green** goes to **blue**
- **green** goes to **yellow**
- **yellow** goes to **blue**

Path Predicate Abstraction (PPA)

Definition: **path predicate abstraction**

We consider a graph $G = (V, E)$ with an operational coloring function $c\colon W \rightarrow \{c_1, c_2, \ldots\}$, with $W \subseteq V$.

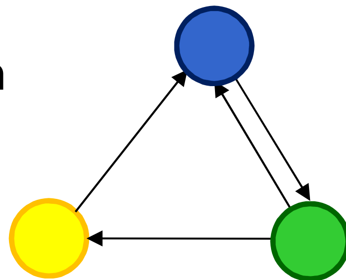
A graph $\hat{G} = (\hat{V}, \hat{E})$ such that

- $\hat{V} = img_c(W)$
- For any two nodes $u, w \in W$ it is $(c(u), c(w)) \in \hat{E}$ if and only if there is an operational path $(u, \ldots, w)$ in G

is called *path predicate abstraction of* G

Example – cont'd.

Path predicate abstracted graph



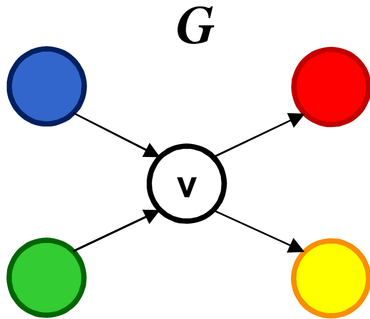
"Operations":

- **blue** goes to **green**
- **green** goes to **yellow**
- **green** goes to **blue**
- **yellow** goes to **blue**

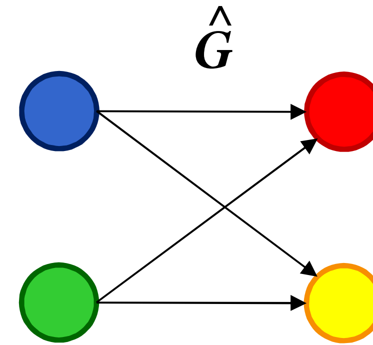
Soundness Theorem

Let $\hat{G}$ be a path predicate abstraction of a graph G. Then, for every (finite or infinite) path in G corresponding to a sequence of colored states $(w_0, w_1, w_2, \ldots)$ that lie on that path there exists an abstract path $(c(w_0), c(w_1), c(w_2), \ldots)$ in $\hat{G}$, and vice versa.

Path Predicate Abstraction vs. Stuttering Bisimulation



graph with operational coloring



path predicate abstraction

Can we find a coloring for the uncolored node v in G such that the relation between G and $\hat{G}$ can be described as a stuttering bisimulation?

If v is colored

- green, this wrongly introduces an edge from blue to green in $\hat{G}$
- blue, this wrongly introduces an edge from green to blue in $\hat{G}$
- red, this wrongly introduces an edge from red to yellow in $\hat{G}$
- yellow, this wrongly introduces an edge from yellow to red in $\hat{G}$

Hence, **not every path predicate abstraction of an operationally colored graph can be described in terms of a stuttering bisimulation.**

Path Predicate Abstraction for FSMs

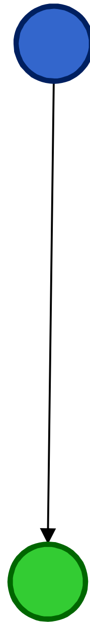
The notion of an „operation“:

- as old as digital design
- provides a natural view on RTL designs
  (operations are multi-cycle register transfers)
- can create link between RTL and higher levels?

Property checking can be used to create an operational coloring on state transition graphs.

Operation Property

blue goes to green

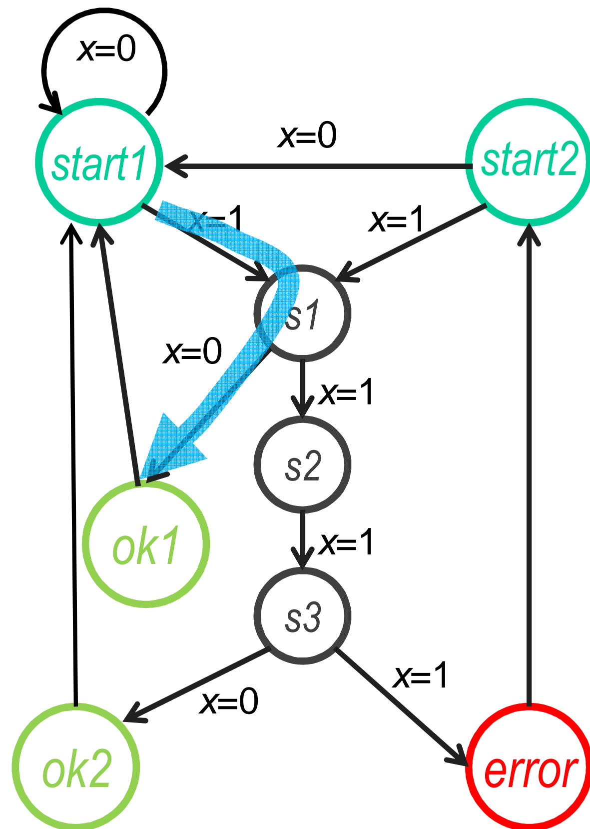


```
property my_operation;

        a_start(S) and        //starting state //
   ##0  a_0(X)     and        // trigger…
   ##1  a_1(X)     and
   ## ..
   ##n  a_n(X);                          …//
implies
        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

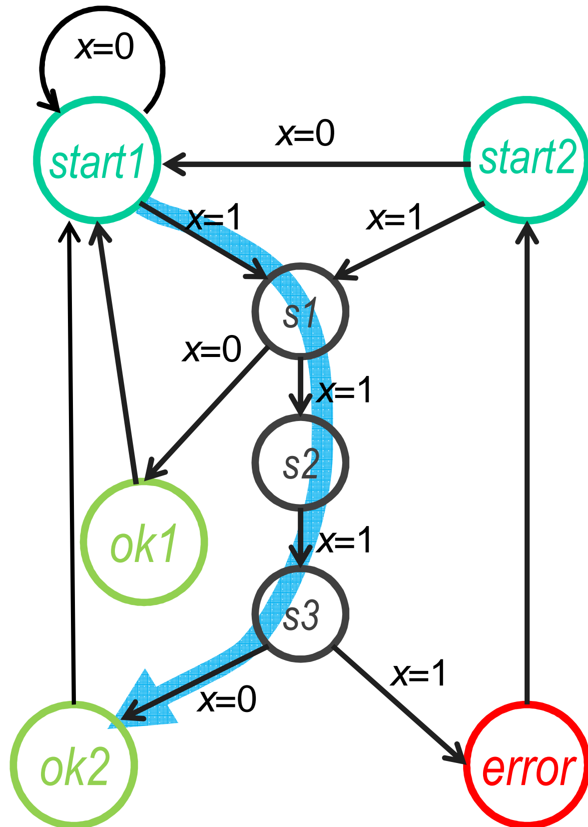
Finite State Machine (Ex.)



Operation "short":
Start state: *start*
End state: *ok*

Trigger sequence:
at t=0: x=1
at t=1: x=0

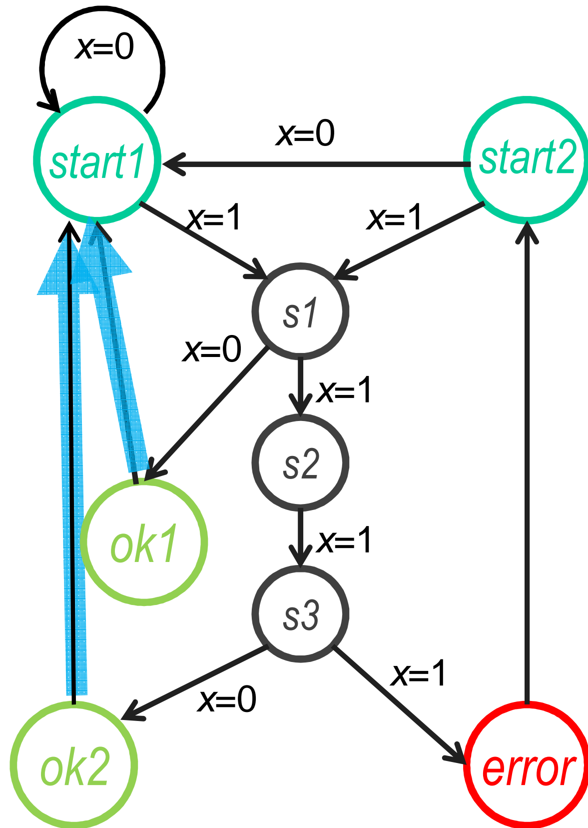
Finite State Machine (Ex.)



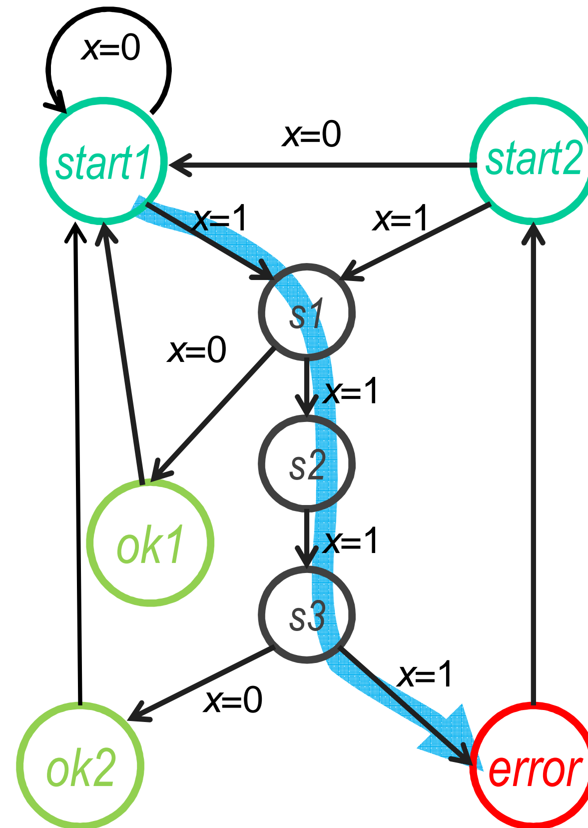
Operation "long":
Start state: *start*
End state: *ok*

Trigger sequence:
at t=0: x=1
at t=1: x=1
at t=2: x=1
at t=3: x=0

Finite State Machine (Ex.)

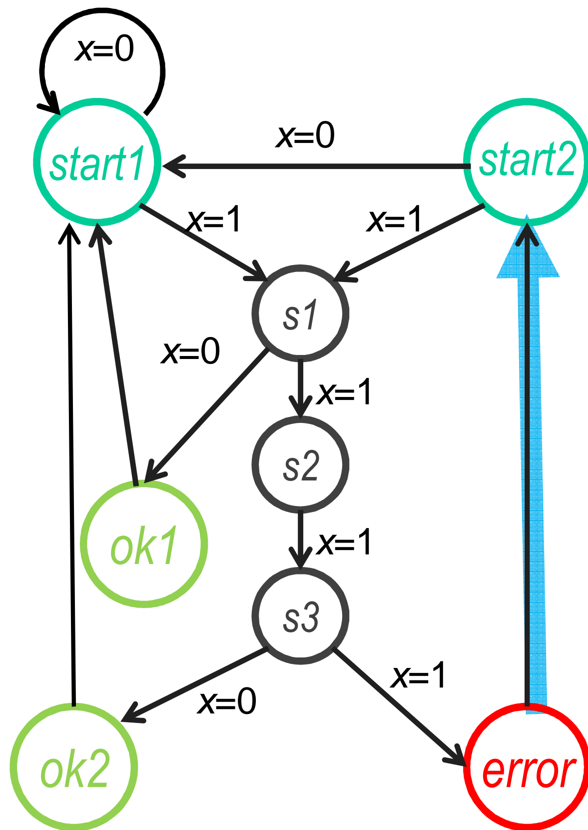


Operation “return”

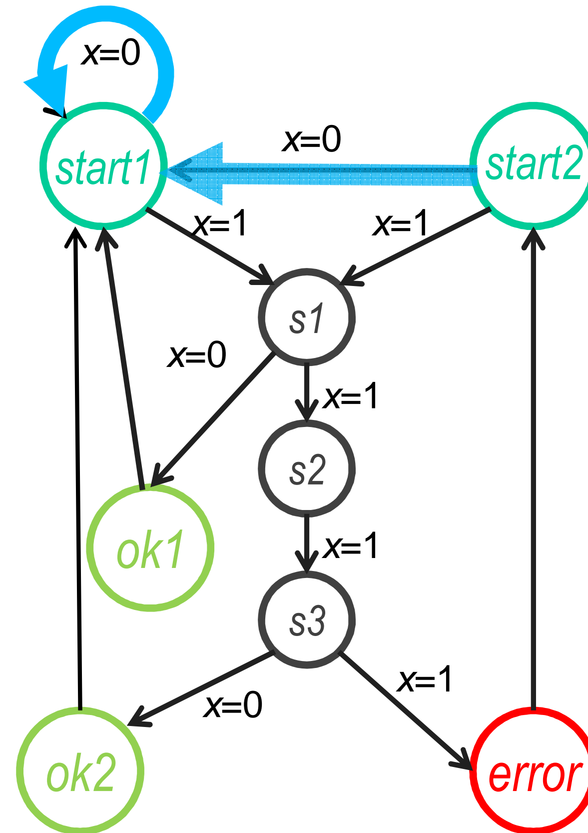


Operation “fail”

Finite State Machine (Ex.)

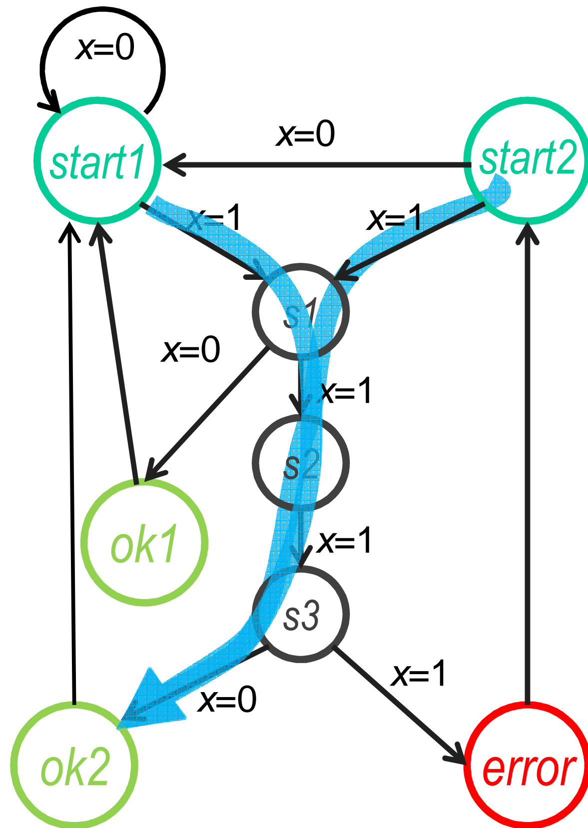


Operation “recover”



Operation “idle”

Operation “Long”




```
property long;
            // starting states
      ##0   (state == start) and
            // trigger sequence
      ##0   (x == 1) and
      ##1   (x == 1) and
      ##2   (x == 1) and
      ##3   (x == 0)
implies
            // output sequence
      ##1   (output == s1) and
      ##2   (output == s2) and
      ##3   (output == s3) and
      ##4   (output == ok) and
            // ending states
      ##4   (state == ok)
end property;
```

Creating the Abstraction

x=0
start1
x=0
start2
x=1
x=1
s1
x=0
x=1
s2
ok1
x=1
s3
x=1
x=0
ok2
error

```
property long;
        ##0   STATE_start() and
        ##3   TRIGGER_long()
implies
        ##4   OUTPUT_long() and
        ##4   STATE_ok()
end property;
```

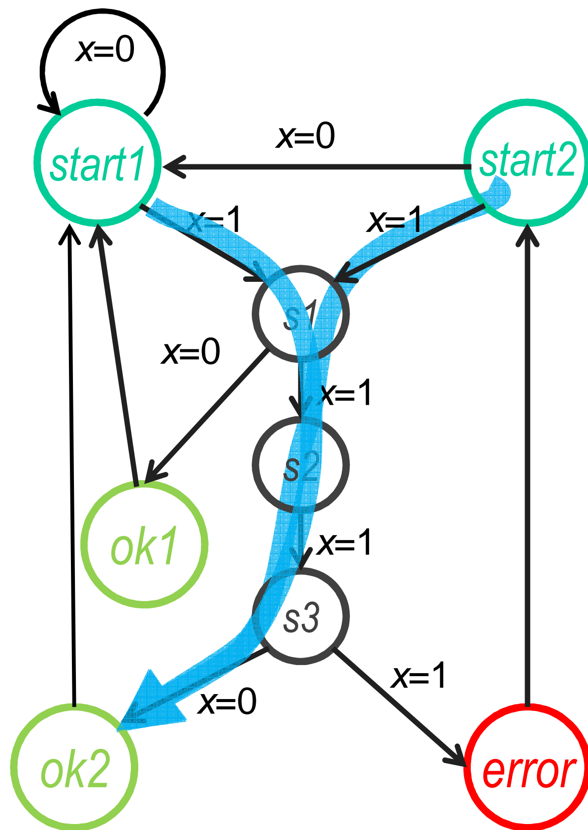
```
function STATE_start();
        STATE_start() = (state == start);
end function;
```

```
function TRIGGER_long();
        TRIGGER_long() = ($past(x,3)==1) && $past(x,2)==1 && x==0);
end function;
```

```
function OUTPUT_long();
        OUTPUT_long() = ($past(output,3)==s1) && $past(output,2)==s2
                && $past(output,1)==s3) && output==ok);
end function;
```

```
function STATE_ok();
        STATE_ok() = (state == ok);
end function;
```

Creating the Abstraction



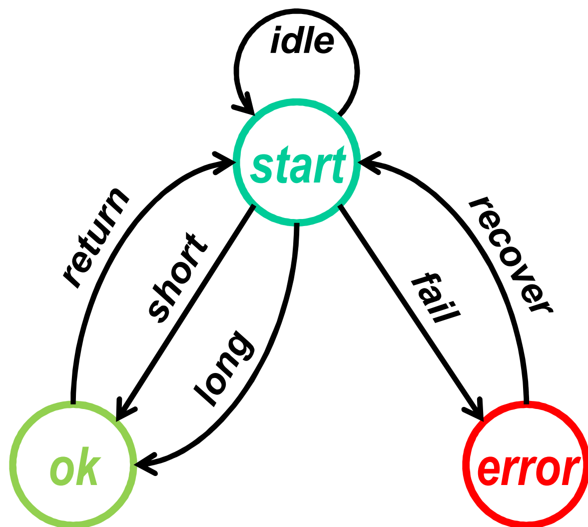

```
property long;
        ##0   STATE_start() and
        ##3   TRIGGER_long()
implies
        ##4   OUTPUT_long() and
        ##4   STATE_ok()
end property;
```

Two types of macros:

- *state predicates*
  represent important control situation (set of states)
- *sequence predicates*
  represent sets of sequences of inputs, outputs and states

Predicates constitute *operational coloring* of the state transition graph

Creating the Abstraction




```
property long;
      ##0   STATE_start() and
      ##3   TRIGGER_long()
implies
      ##4   OUTPUT_long() and
      ##4   STATE_ok()
end property;
```

Two types of macros:

- *state predicates*
  represent important control situation (set of states)
- *sequence predicates*
  represent sets of sequences of inputs, outputs and states

Predicates constitute *operational coloring* of the state transition graph

$\Rightarrow$ soundness of path predicate abstraction

Soundness w.r.t. LTL

	Abstract formula	Concrete formula
(1)	$\hat{s}_{M,i}$	η_i
	$\hat{x}_{M,j}$	$\Psi \wedge \iota_j$
	$\hat{y}_{M,k}$	$\Psi \wedge \mu_k$
(2)	$\mathsf{X}\,\hat{f}$	$\mathsf{X}(\neg\Psi\,\mathsf{U}\,(\Psi \wedge f))$
	$\mathsf{F}\,\hat{f}$	$\mathsf{F}(\Psi \wedge f)$
	$\mathsf{G}\,\hat{f}$	$\mathsf{G}(\Psi \Rightarrow f)$
	$\hat{g}\,\mathsf{U}\,\hat{f}$	$(\Psi \Rightarrow g)\,\mathsf{U}\,(\Psi \wedge f)$
(3)	$\neg\hat{f}$	$\neg f$
	$\hat{f} \wedge \hat{g}$	$f \wedge g$
	$\hat{f} \vee \hat{g}$	$f \vee g$

Compositionality Path Predicate Abstraction

Communication between Modules

- **System level:** designers think in terms of „events“
- **RTL:** designers think in terms of register transfers / operations

⇒ asynchronous and synchronous communication mechanisms at the RTL are mapped to asynchronous handshakes at the system level

[1] Urdahl, J.; Stoffel, D.; Kunz, W., "Path Predicate Abstraction for Sound System-Level Models of RT-Level Circuit Designs“, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,, vol.33, no.2, pp.291-304,* Feb. 2014

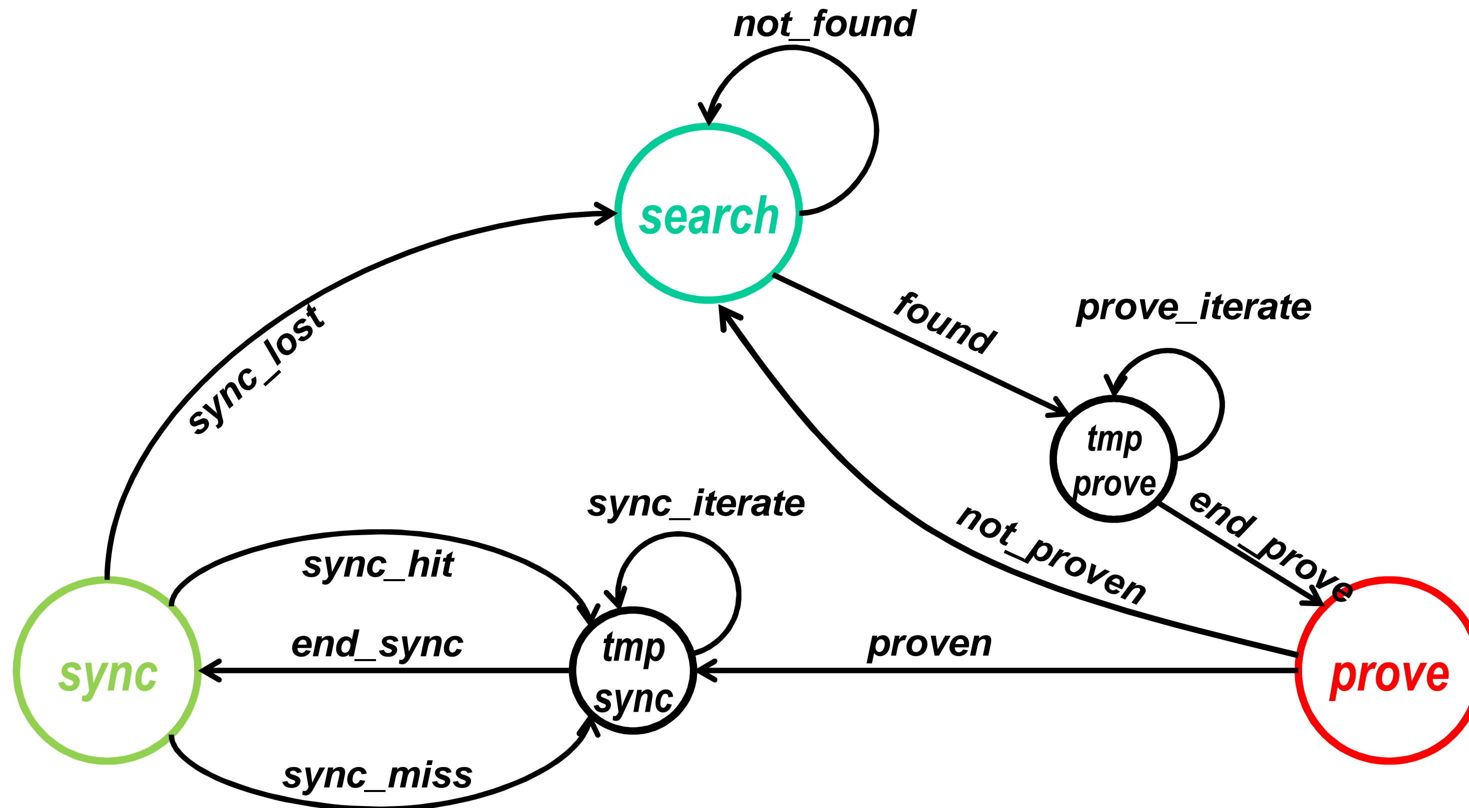
[2] Urdahl, J.; Stoffel, D.; Kunz, W., "System- versus RT-Level Verification of Systems-on-Chip by Compositional Path Predicate Abstraction“, *Technical Report,Technische Universität Kaiserslautern*, May 2014

Industrial case study

Alcatel Lucent SONET / SDH Framer

- 27,000 LoC (VHDL)
- Aligns an incoming bit stream in SONET / SDH frames
- Determines synchronization status
- Circuitry to output and setup performance measurements (including a micro-processor interface)
- Generic implementation for different SONET / SDH rates

Formalized Abstract State Model



Path Predicate Abstraction
soundly models RTL behavior by SVA properties

Results: Case Study 1

Sound system level model of Alcatel-Lucent Framer

Module	#inputs	#outputs	#state bits	#LoC
Framer	549 / 9	280 / 7	47213 / 11	27k / 122
Monitor	20 / 13	6 / 3	30 / 12	850 / 80

Work effort: 2 person months

Public domain demonstrator available at:
http://www.eit.uni-kl.de/eis/forschung/ppa

Results: Case Study 2

Flexible Peripheral Interconnect (*FPI*) bus
(owned by Infineon Technologies)

- modular system
  - BCU
    - arbiter, starvation prevention, debugging unit, ...
  - master agent $\times N$
  - slave agent $\times M$
- communicational, complex and highly optimized

A path predicate abstraction was constructed for each module

Results: Case Study 2

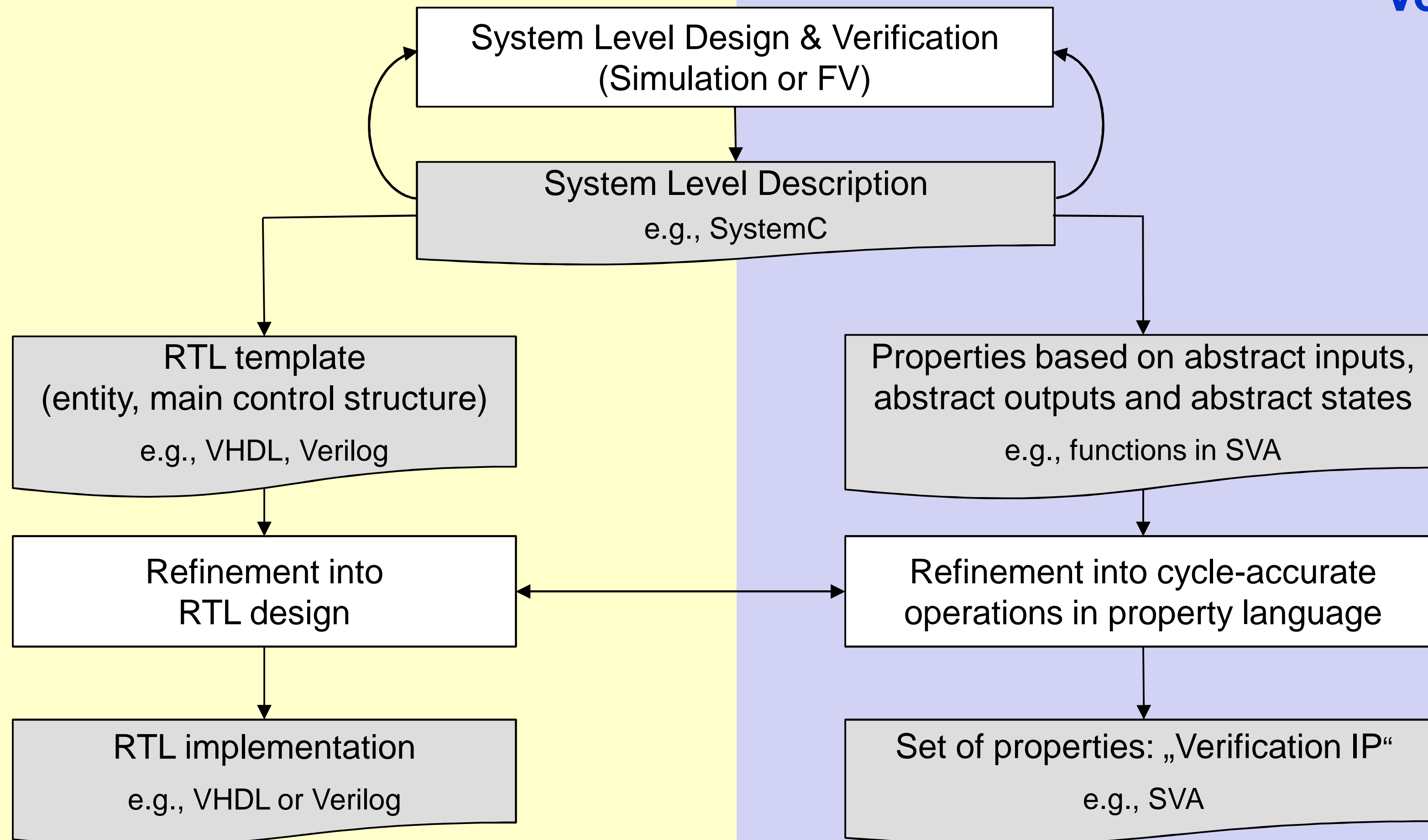
The path predicate abstracted FPI Bus modules were realized in PROMELA

Module	#inputs		#outputs		#state bits		#LOC	
	RTL	Sys.	RTL	Sys.	RTL	Sys.	RTL	Sys.
MasterAgent	199	17	202	13	292	17	3568	91
SlaveAgent	199	10	202	5	292	7	3568	49
BCU	258	9	215	4	941	14	8966	41

Composed system: 3 × master, 3 × slave, 1 × BCU

- proving freedom of deadlocks with SPIN
  about 3 minutes using 3GB of memory
- a requested transaction is executed correctly in the system, $G(a \Rightarrow Fc)$
  about 1 minute using 0.5GB of memory

Top-Down Design Flow



Optimization Potential

Signals \ Operations / Time points	prove_sync_iterate t=0	sync_iterate t=0	marker_not_found t=0	marker_found t=0	prove_sync_end t=0	sync_end t=0	sync_hit t=0	sync_lost t=0	sync_lost t=1	sync_not_proven t=0	sync_proven t=0	sync_miss t=0	sync_iterate_last t=0
data_in_buf_reg_lsb													
dt_path_frmout													
ror_reg		T				T	T	T			T	T	T
sync_found_twice_reg		T			T	T	T	T		T	T	T	
search_result	T	T		T		T	T	T			T	T	T
sync_frm_start						T		T			T		T
no_sync_found		T			T	T	T	T	T	T	T	T	
frm_in_pulse						T	T	T	T			T	T
search_frm_cnt													
search_cnt_start		T		T		T	T	T			T	T	T
oof_counter	T	T		T			T	T			T	T	T
oof_msg	T	T		T			T	T	T	T	T	T	T
search_pos							T	T		T	T		
start_new_search_ff	T			T		T	T				T	T	T
new_sync_found	T	T		T		T	T				T	T	T
pos_reached						T	T	T	T			T	T
new_sync		T				T	T				T	T	
index_found		T				T	T				T	T	

Top-Down Sonet/SDH Framer

- A complete redesign of the framer (besides the µP interface)
  - Minimal buffering of the incoming stream
  - Power-aware process sensitivities
- Total energy consumption (dynamic and static) reduced by about 50% compared to the original
- Correct refinement of the system model (formal proof)

Conclusion

The Big Hurdles of Formal Verification in Practice – Can we overcome them in system-level design flows?

Yes, if we successfully address these issues:

- Property Checking needs to adopt a new role:
  - more than “bug hunting”: the result of formal RTL property checking should be the soundness of the Electronic System Level (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

Conclusion

The Big Hurdles of Formal Verification in Practice – Can we overcome them in system-level design flows?

Yes, if we successfully address these issues:

- Integrate property-based abstractions into top-down design methodology:
  - Automatically generate properties at system level
  - Semi-automatically refine them into RTL
  - Leverage Verification IP to reach non-functional design targets
- Most technology is available, methodology is key!