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

What are the hurdles?
Double Efforts

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

*Scope*: SoC module-level

**Simulation**
- probe functional behaviors

*Scope*: chip-level
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
- ...
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?”
Limited Acceptance with Verification Engineers

For good reasons:

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

Property Checking is actually more like design than like verification!
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
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
“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)!
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
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 $v_1, \ldots, v_{n-1} \in V \setminus W$ 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.
Given a graph $G$ where nodes in $W$ are shaded grey. Find an operational graph coloring with colors:

- blue
- green
- yellow
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 $c: W \rightarrow \{ c_1, c_2, \ldots \}$, 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 $G$**
Soundness Theorem

Let $\tilde{G}$ 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 $\tilde{G}$, and vice versa.

"Operations":
- blue goes to green
- green goes to yellow
- green goes to blue
- yellow goes to blue
Path Predicate Abstraction vs. Stuttering Bisimulation

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**

**property** my_operation;

\[ a_{\text{start}}(S) \quad \text{and} \quad // \text{starting state} // \]
\[ ##0 \quad a_0(X) \quad \text{and} \quad // \text{trigger…} \]
\[ ##1 \quad a_1(X) \quad \text{and} \]
\[ .. \]
\[ ##n \quad a_n(X); \quad // \text{…} // \]

**implies**

\[ c_0(X, Y) \quad \text{and} \]
\[ ##1 \quad c_1(X, Y) \quad \text{and} \]
\[ .. \]
\[ ##n \quad c_n(X, Y) \quad \text{and} \]
\[ ##n \quad c_{\text{end}}(S); \quad // \text{ending state} // \]

**endproperty**;

\( S \): state variables, \( X \): inputs, \( Y \): outputs

---

**blue goes to green**

- Blue circle
- Green circle
- Transition arrow
Finite State Machine (Ex.)

Operation “short”:
Start state: \( \text{start} \)
End state: \( \text{ok} \)

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

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

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

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

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
⇒ **soundness** of path predicate abstraction

---

```
property long;
    ##0  STATE_start() and
    ##3  TRIGGER_long()
implies
    ##4  OUTPUT_long() and
    ##4  STATE_ok()
end property;
```
### Soundness w.r.t. LTL

<table>
<thead>
<tr>
<th>Abstract formula</th>
<th>Concrete formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_{M,i}$</td>
<td>$\eta_i$</td>
</tr>
<tr>
<td>$x_{M,j}$</td>
<td>$\Psi \land t_{j}$</td>
</tr>
<tr>
<td>$y_{M,k}$</td>
<td>$\Psi \land \mu_k$</td>
</tr>
<tr>
<td>$X \hat{f}$</td>
<td>$X(\neg \Psi \cup (\Psi \land f))$</td>
</tr>
<tr>
<td>$F \hat{f}$</td>
<td>$F(\Psi \land f)$</td>
</tr>
<tr>
<td>$G \hat{f}$</td>
<td>$G(\Psi \Rightarrow f)$</td>
</tr>
<tr>
<td>$\hat{g} \cup \hat{f}$</td>
<td>$(\Psi \Rightarrow g) \cup (\Psi \land f)$</td>
</tr>
<tr>
<td>$\neg \hat{f}$</td>
<td>$\neg f$</td>
</tr>
<tr>
<td>$\hat{f} \land \hat{g}$</td>
<td>$f \land g$</td>
</tr>
<tr>
<td>$\hat{f} \lor \hat{g}$</td>
<td>$f \lor g$</td>
</tr>
</tbody>
</table>
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


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
Sound system level model of Alcatel-Lucent Framer

<table>
<thead>
<tr>
<th>Module</th>
<th>#inputs</th>
<th>#outputs</th>
<th>#state bits</th>
<th>#LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framer</td>
<td>549 / 9</td>
<td>280 / 7</td>
<td>47213 / 11</td>
<td>27k / 122</td>
</tr>
<tr>
<td>Monitor</td>
<td>20 / 13</td>
<td>6 / 3</td>
<td>30 / 12</td>
<td>850 / 80</td>
</tr>
</tbody>
</table>

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
The path predicate abstracted FPI Bus modules were realized in PROMELA

<table>
<thead>
<tr>
<th>Module</th>
<th>#inputs</th>
<th>#outputs</th>
<th>#state bits</th>
<th>#LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>MasterAgent</td>
<td>199</td>
<td>17</td>
<td>202</td>
<td>13</td>
</tr>
<tr>
<td>SlaveAgent</td>
<td>199</td>
<td>10</td>
<td>202</td>
<td>5</td>
</tr>
<tr>
<td>BCU</td>
<td>258</td>
<td>9</td>
<td>215</td>
<td>4</td>
</tr>
</tbody>
</table>

**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 ⇒ Fc) about 1 minute using 0.5GB of memory
Top-Down Design Flow

System Level Design & Verification (Simulation or FV)

- System Level Description
  - e.g., SystemC
  - RTL template (entity, main control structure)
    - e.g., VHDL, Verilog
    - Refinement into RTL design
    - RTL implementation
      - e.g., VHDL or Verilog

- Properties based on abstract inputs, abstract outputs and abstract states
  - e.g., functions in SVA
  - Refinement into cycle-accurate operations in property language
  - Set of properties: "Verification IP"
    - e.g., SVA
## Optimization Potential

<table>
<thead>
<tr>
<th>Operations</th>
<th>Time points</th>
<th>prove_sync_iterate</th>
<th>sync_iterate</th>
<th>marker_not_found</th>
<th>marker_found</th>
<th>prove_sync_end</th>
<th>sync_end</th>
<th>sync_hit</th>
<th>sync_lost</th>
<th>sync_lost</th>
<th>sync_not_proven</th>
<th>sync_proven</th>
<th>sync_miss</th>
<th>sync_iterate_last</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signals</td>
<td></td>
<td>data_in_buf_reg_lsb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dt_path_frmout</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ror_reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sync_found_twice_reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>search_result</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sync_frm_start</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>no_sync_found</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>frm_in_pulse</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>search_frm_cnt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>search_cnt_start</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>oof_counter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>oof_msg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>search_pos</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>start_new_search_ff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>new_sync_found</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pos_reached</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>new_sync</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>index_found</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data in the table represents the potential for optimization across various operations and time points.
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!