Formal Verification of Systems-on-Chip – Industrial Experiences and Scientific Perspectives

Wolfgang Kunz

Department of Electrical & Computer Engineering
University of Kaiserslautern
Outline

- Basics of formal verification: the bounded and the unbounded paradigm
- Industrial experiences
- Abstraction techniques in property checking
- Perspectives: Abstractions for System-Level Design and Verification
Formal Property Checking - Overview

Formal Verification of SoCs

SoC Design Flow

- System Requirements Analysis
- Equivalence Checking

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

Prove that both designs are functionally equivalent
Formal Verification of SoCs

SoC Design Flow

- System Requirements Analysis
- System Architecture Design
- Hardware Property Checking
  Given:
  - informal specification
  - RT-level circuit description
  Prove (by checking properties) that the RT-level design description fulfills the specification

- Micro Architecture Design (for all modules and interfaces)
- RTL Design
- Logic Design
- Physical Design
**Property Checking**

**Basic Approaches**

The „unbounded“ paradigm:

- (Classical) Model Checking

*the world of*

- Kripke structures
- state space exploration
- BDDs (Binary Decision Diagrams)
- automatic abstraction/refinement techniques
- handling systems with a few hundred state variables
Formal Verification of SoCs

Property Checking

Basic Approaches

The „bounded“ paradigm:

- Bounded Model Checking
- Interval Property Checking
- K-Step-Induction

_the world of_

- unrolled FSMs
- SAT (satisfiability solving)
- intuitive invariants
- sophisticated methodology
- handling systems with thousands of state variables
A **Kripke model** is a quintuple $K = (S, S_0, R, A, l)$ with

- $S$: a finite set of states
- $S_0$: a set of initial states with $S_0 \subseteq S$
- $R$: a transition relation, $R \subseteq S \times S$, describing all possible state transitions of the model
- $A$: a set of atomic formulas, $A = \{p, q, \ldots\}$ that can each assume the values true or false
- $l$: a valuation function, $l: A \rightarrow 2^S$. It specifies for every formula in $A$ the set of all states in $S$ for which the atomic formula is valid (true)

The Kripke model can be easily obtained from the finite state machine model of a digital circuit.
Model Checking

Kripke Model

atomic formulas: \( A = \{g, y, r\} \)
Model Checking

The Language

**Computation Tree Logic (CTL)**

atomic formulas: f, g, h, … (can be true or false)

propositional operators: \( \neg \) (not) \( \land \) (and) \( \lor \) (or)

modal operators: \( E \) (\text{"it is possible"}) \( A \) (\text{"always"})

\( (\exists) \) \( (\forall) \)

temporal operators: \( X \) (next) \( F \) (future) \( G \) (globally) \( U \) (until)
Model Checking

The Language

Computation Tree
Examples:

**EG**\( r \): 
\( s_0 \models EG r \iff \) there exists a path \((s_0, s_1, s_2, \ldots)\) such that \(s_0 \models r\), \(s_1 \models r\), \(s_2 \models r\), ...

**AF**\( y \): 
\( s_0 \models AF y \iff \) for all paths \((s_0, s_1, s_2, \ldots, s_i, \ldots)\) there is an \(i\) with \(i \geq 0\) such that \(s_i \models y\)

**EX**\( g \): 
\( s_0 \models EX g \iff \) there exists a path \((s_0, s_1, s_2, \ldots)\) such that \(s_1 \models g\)
General procedure to verify a property in a Kripke model $K$:

- formulate property as CTL formula $f$
- evaluate formula, i.e., calculate all states of the Kripke model for which the formula is valid
- property is verified if and only if $S_0 \subseteq f$
\( y_{i+1} = p \lor \text{EX} y_i, \quad y_0 = \{ \} = \text{false} \)
Theorem: For given state sets $p$ and $q$ of a Kripke model, the state sets $\text{EF}p$, $\text{EG}p$ and $\text{E}(p \cup q)$ are the fixed points of an iteration over $y_i$. For each formula, in the fixed point it holds that $y_{i+1} = y_i$.

The CTL operators and their fixed point iterations are:

- $\text{EF}p$: $y_{i+1} = p \lor \text{EX}y_i$, $y_0 = \text{false}$
- $\text{EG}p$: $y_{i+1} = p \land \text{EX}y_i$, $y_0 = \text{true}$
- $\text{E}(p \cup q)$: $y_{i+1} = (q \lor (p \land \text{EX}y_i))$, $y_0 = \text{false}$

**Symbolic Model Checking:** fixed point iterations on state sets can be implemented by symbolic representations using Binary Decision Diagrams (BDDs)
Example
Consider a Finite State Machine $M$ with a transition relation $R$: 

![Finite State Machine Diagram]

- States: 0000, 0001, 1101, 0100, 1110
- Transitions: 0000 → 0001, 0001 → 1101, 1101 → 0100, 0100 → 1110
Example
Suppose the two least significant bits are considered as free inputs:

\[ S \]

\[ 0000 \rightarrow 0001 \rightarrow 1101 \rightarrow 1110 \rightarrow 0100 \]

The two most significant bits span an abstract machine

\[ M \rightarrow \hat{M} \]
**Example**

The abstract machine

The two most significant bits span an abstract machine $\hat{M}$.

This abstract machine has been obtained by „Localization Reduction“
Abstractions in Model Checking

Example

The abstract machine

\[ S \]

00

01

11

But:
also unreachable states are abstracted into the abstract states

The good news:
- less states
- smaller sequential depth
Abstractions in Model Checking

Example

\[ \hat{S} \]
Abstractions in Model Checking

Example

The abstract machine

Safety property: $\text{AG}(\hat{s} \neq 10)$ \textbf{fails!}

Abstraction is sound with respect to properties of type $\text{AG}p$ ("safety properties"): if property holds on abstract model it also holds on concrete design

but false negatives can occur if counterexamples are based on unreachable states

Possible solution: iterative, counter-example guided abstraction / refinement
Industrial Experiences

- CTL and related languages have strongly influenced today’s industrial property languages:
  - Property Specification Language (PSL)
  - System Verilog Assertions (SVA)

- Complexity issues:
  - state explosion problem
  - sequential depth may cause large number of fixed point iterations
  - abstraction techniques not always robust

- Designs with more than a few 100 state variables cannot be handled robustly

- Industrial practice: Assertion-Based Verification (ABV)
  - instrument RTL code with (local) assertions, prove formally when possible, simulate otherwise
The bounded paradigm

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!

Bounded proof:
property holds for $k$ cycles starting from initial state $s_0$
Interval Property Checking

Reachable states?

\[ \delta, \lambda \]

Property of length \( n = 3 \)

unbounded proof:
- property formulated over finite time window always holds starting from any state

But: false negatives possible
A typical property for RT-level module verification:

\[ \text{AG}(a \rightarrow c) \]

\[ a : \text{ assumptions} \]
- module is in some control state \( S \)
- certain inputs \( X \) occur ("trigger")

\[ c : \text{ commitments} \]
- module goes into certain control state \( S' \)
- certain outputs \( Y \) occur
RT-level module verification: operation by operation

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

Property 2: $AG(a_{control\ 2} \rightarrow c_{control\ ...})$

RTL-level module verification: operation by operation
**RT-level module verification**

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
Completeness Checking

Prove that a set of operation properties is “complete”, i.e., uniquely describes all behaviors of a deterministic finite state machine.

Complete Property Set: two different FSMs fulfilling all properties of the set are sequentially equivalent (in the signals mentioned in the verification plan).
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.

\[ AG(a \rightarrow c) \]

**Note:**

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

\[ AG(a \rightarrow c) \]

\[ S: \text{state variables}, \ X: \text{inputs}, \ Y: \text{outputs} \]
IPC moves along abstract states

Specify properties in terms of main states

\[
\text{property } \text{myExample is}
\]
\[
\text{assume:}
\]
\[
\begin{align*}
\text{at } t: & \quad \hat{S} = \hat{s}_1; & \text{ //starting state //} \\
\text{at } t: & \quad a_0(X); \\
\text{at } t+1: & \quad a_1(X); \\
\text{at } & \ldots \\
\text{at } t+n: & \quad a_n(X);
\end{align*}
\]
\[
\text{prove:}
\]
\[
\begin{align*}
\text{at } t: & \quad c_0(X, Y); \\
\text{at } t+1: & \quad c_1(X, Y); \\
\text{at } & \ldots \\
\text{at } t+n: & \quad c_n(X, Y); \\
\text{at } t+n: & \quad \hat{S} = \hat{s}_2; & \text{ //ending state //}
\end{align*}
\]
\[
\text{end property;}
\]
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**

**Formal Property Checking - Overview**

**Formal Verification of SoCs**

**Example**

```
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

Input = REQ

cnt ≠ n

cnt = n
/ output := ACK

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

Global state vector with m bits:

\[
\begin{array}{cccccccc}
\ldots & x & x & x & x & \ldots & x & 1 & 0 & 1 & 0 & 0 & x & \ldots & x & x \\
m-1 & q & p & \ldots & 1 & 0 & 0 & x & \ldots & x & x
\end{array}
\]
Example

Formal Property Checking - Overview

Formal Verification of SoCs

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

False negative?
“Invariant”: state set closed under reachability

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.

\[
\text{IDLE} \rightarrow \text{cnt} = 0
\]

\[
\begin{array}{c}
\text{m-1} \\
\ldots \\
1 \\
0 \\
\end{array}
\]

global state vector for design
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.

This should be impossible!

**Global state vector for design**

- IDLE and $cnt = 5$
- $m-1$, ..., 1, 0
Tool produces counterexample, false negative? How to proceed in practice?

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

$$\text{IDLE} \rightarrow \text{cnt} = 0$$

*global state vector for design*
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**

**assertion** reachability_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!*
Industrial Experiences

- The bounded paradigm: SoC modules with thousands of state variables can be handled

- Operation-based formulation:
  - Systematic way of completely describing design by properties
  - Reachability analysis mostly avoided

- Industrial practice: Gap-Free Formal Verification
  - Properties provide a complete documentation and verification of SoC module
  - Sophisticated methodology
Industrial Experiences

Verification Engineer’s preferences

White-box versus black-box verification

“Stimulus, response! Stimulus, response! Don’t you ever think?”
The future of FV will be decided …

… by how we will handle system-level design and verification

- Electronic System Level (ESL)
- SystemC
- Virtual Prototyping
- High Level Synthesis
- …
...are used in industry, for example, to

- to parallelize HW and SW development ("virtual prototyping")
- to evaluate complex signal processing algorithms
  - functional correctness
  - performance bottlenecks

*but ad hoc, in niches and only for parts of the design!*
System Level Models

The problem

System level models and tools are quite domain-specific

→ only used in niches
→ tool support not always adequate

System level models are usually created in addition to the other models

→ high costs (in spite of IP re-use)

Semantic Gap: no formal relationship between high-level models and concrete RTL implementation

→ RTL design process as costly as before
→ RTL verification as costly as before
The future?

Terminology developed in FV over decades seems useful to structure ESL design flow:

- Create a **formal relationship** between system level and RTL
  - behavior verified at system level is also guaranteed for RTL implementation

- Methodology for design refinements from system level such that formal relationship is preserved
  
  e.g.
  - transitions in system-level FSMs are translated into RTL operations

- „Model-based“ HW verification: system-level description yields verification model for implementation
  - low-level properties derived easily from system level models
Thank you for your attention!