Formal Verification of Systems-on-Chip

Wolfgang Kunz

Department of Electrical & Computer Engineering
University of Kaiserslautern, Germany
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 prototypes

Property Checking

Given:
- informal specification
- RT-level circuit description

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

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

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 not mature)
Verify global behavior of entire chip (FV out of reach)
RT-level module verification

A typical property for RT-level module verification:

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

**$a$**: 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

Property 1: $\forall_{\text{control } 1} \rightarrow c_{\text{control } 2}$

Property 2: $\forall_{\text{control } 2} \rightarrow c_{\text{control } ...}$

$\forall$ cycles

Data Path
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($a_{\text{control 1}} \rightarrow c_{\text{control 2}}$)

$a_{\text{control 1}} : a_{\text{control}} \land (V) \land \prod_{j=0}^{j=n} AX^{j}(a_{j}(X))$

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

$V$: state variables

/ data_path_control_signals

data path
Property for RT-level module verification

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

\[ a = a_{\text{start}}(V) \land \prod_{j=0}^{j=n} AX^j(a_j(X)) \]

\[ c = \prod_{j=0}^{j=n} AX^j(c_j(X,Y)) \land AX^n(c_{\text{end}}(V)) \]

**Interval Temporal Logic™**

**property myExample is**

**assume:**
- at \( t \): \( a_{\text{start}}(V) \); //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 \( t+n \): \( c_{\text{end}}(V) \); //ending state //

end property;

\( V \): state variables, \( X \): inputs, \( Y \): outputs
Property for RT-level module verification

**Interval Temporal Logic™**

```plaintext
property myExample is
  assume:
    at t: a_{\text{start}}(s);  // 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_{\text{end}}(s);  // ending state //
end property;
```

\( s \): state variables, \( x \): inputs, \( y \): outputs

**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
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

- Simulation
- Complete formal module verification
- Setting up properties
- Proving completeness

Effort vs. Achieved quality graph showing the comparison between simulation and 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
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!

\[ \delta, \lambda \]

\[ s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \]

Property of length \( n = 3 \)

satisfiable?

generate SAT instance:

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

- initial states
- transition relation unrolled \( k+n \) times
- propositional formula for internal formula of AGp, \( k \) instances
Modified formulation

Proving safety properties (AGp) using bounded circuit model

\[
\begin{align*}
\delta, & \lambda \\
\delta, & \lambda \\
\delta, & \lambda \\
\end{align*}
\]

Property of length \( n = 3 \) satisfiable?

generate SAT instance:

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

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

transition relation unrolled \( n \) times

one instance of propositional formula for property
Interval Property Checking essentially means that we construct 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
The registers of the SoC module correspond to different segments in the global state vector $V$. 
**ITL operational 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.

---

**Interval Temporal Logic™**

```plaintext
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

input = REQ

cnt ≠ n

cnt = n
/ output := ACK

READY

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

Global state vector with m bits:

\[
\begin{array}{cccccccc}
\_ & \_ & \_ & \_ & \_ & \ldots & 1 & 0 & 1 & 0 & 0 & x & \ldots & x & x \\
\_ & \_ & \_ & \_ & \_ & \ldots & q & p & \ldots & 1 & 0 \\
\end{array}
\]
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
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 …
Proving safety properties (AGp) by IPC with invariants

generate SAT instance:

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

- **Invariant**
- transition relation unrolled \( n \) times
- one instance of propositional formula for property

(special case: \( \phi = 1 \))

need to add reachability information here!
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?
Example: FSM with 3 state variables

Reachable states: 
\[ R = \{000, 001, 010, 011, 100\} \]

Unreachable states: 
\[ U = \{101, 110, 111\} \]
Invariants:

\[ W_1 = R = \{000, 001, 010, 011, 100\} \]
\[ W_2 = \{001, 011, 010, 100\} \]
\[ W_3 = \{010\} \]
\[ W_4 = \{011, 010, 100\} \]
\[ W_5 = \{100\} \]
\[ W_6 = \{101, 110, 000, 001, 010, 011, 100\} \]
\[ W_7 = \{110, 001, 011, 010, 100\} \]
\[ W_8 = \{111, 110, 001, 011, 010, 100\} \]
\[ W_9 = \{010, 100\} \]
\[ W_{10} = \{111, 101, 110, 000, 001, 010, 011, 100\} \]
Let $p$ be a Boolean formula. We want to prove that $p$ holds in every reachable state of the system ("safety property", in CTL: $\text{AG}p$).

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.
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

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

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…

\[ \phi = 1 \]

holds in most cases when verifying individual modules
IPC with invariants

The difficult cases…

φ ≠ 1

System-on-Chip

RISC

DSP

Block Interfaces

I/O controller

HW accelerator

φ ≠ 1

φ ≠ 1

φ ≠ 1

φ ≠ 1

φ ≠ 1

for implementations of SoC protocols!
Typical invariants in industrial practice:

All states of the code space that fulfill certain “reachability constraints”.

**Example:** The set of all states such that the counter value is 0 whenever the controller is in state **IDLE**.

This means: we do not need to specify an invariant explicitly. Instead, we can describe it implicitly by a set of constraints for the state variables (e.g. implications, equivalences) that is proved to hold for all states.

**IPC with Invariants**

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

$m-1 \quad \ldots \quad 1 \quad 0$

*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

This should be impossible!
Tool produces counterexample, false negative? How to proceed in practice?

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

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

$m-1 \quad ... \quad 1 \quad 0$

*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 $\rightarrow$ cnt = 0

**property 2 (induction step)**

Assume: at t: IDLE $\rightarrow$ cnt = 0
Prove: at t+1: IDLE $\rightarrow$ cnt = 0
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 $t$: $(\text{state} = \text{IDLE} \land \text{input} = \text{REQ} \land \text{cnt} = 0)$

**prove:**
- at $t+n$: $(\text{state} = \text{READY} \land \text{output} = \text{ACK})$
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!
Methodology

- RTL description
- informal specification
- writes properties
- finds reachability constraints updates properties
- properties (ITL)
- property checking (interval-based)
- counter-example
- property holds
- property fails
- true negative
- false negative
IPC moves along abstract states

Specify properties in terms of main states

property myExample is
assume:
  at t: \hat{\mathbf{V}} = \hat{s}_1;  //starting main-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: \hat{\mathbf{V}} = \hat{s}_2;  //ending main-state //
end property;
Methodology using Main-FSM

Specifying properties based on the main-FSM

• is intuitive,
• global understanding of design is sufficient
• code inspection restricted to main-FSM

But:

• it does not solve the false negative problem
Traversal: transition by transition

Idea: “Transition-by-Transition (TBT)” traversal
compute the sets $S_{\hat{s}}$ step by step according to the transitions of the main FSM

\[
S_{\hat{s}_3} = \text{img}_{\hat{s}_1 \rightarrow \hat{s}_3}(S_{\hat{s}_1}) \cup \text{img}_{\hat{s}_2 \rightarrow \hat{s}_3}(S_{\hat{s}_2})
\]
Methodology

RTL description → automatic extraction → main FSM → informal specification

TBT-traversal

finds reachability constraints updates properties

properties (ITL)

property checking (interval-based)

property holds

false negative

property fails

true negative

counter-example
Industrial case study

Industrial design (flash-memory controller using AMBA-flavor protocol):
  493 state variables, 23 properties to prove
  – compliance with protocol specification
  – correct execution of functional operations
main FSM: 38 states, 103 transitions (extracted automatically by Debussy®)

Verification effort without proposed approach:
  approximately 1 person month
  70% of the time needed for code inspection to refine reachability constraints

Results with proposed approach:
  38 reachability constraints identified: used as assumption in properties
  total CPU time: 0:07:10h

all properties proved, no manual refinements needed!
Conclusion

FV for modules getting established, what's next?

Correctness by integration…

- Better leverage of low level guarantees in high level verification
- Better synergy between automatic and manual abstractions

- FV of interfaces (communication)
- FV of modules (computation)

50% algorithms plus 50% methodology!