Open Source Formal Verification

Introduction to formal verification

Yann Thoma

Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

July 2025

Verification goal

Answer two questions

- Does it work?
- Are you sure?
- Really?
- Well, really really sure?
- No kidding?

Formal verification

- Checking the system against properties
- Proof of the adequation between the system and its specifications
- Advantages
  - Formal proof, so irrefutable if properties are well written
  - Allows to validate part of a system that can then be used in simulation without further tests
- Disadvantages
  - Maybe not adequate for all systems
  - Better with control than data
  - A specific know-how
- Could need to be complemented by functional verification

Thinking about properties

- A big shift towards properties instead of behavior and scenarios
- Examples of properties:
  - If a request is issued, then an acknowledgement shall arrive after 3 cycles
  - If the number of push into a FIFO equals the number of pop, then empty signal shall be high
  - Address and data buses shall be stable throughout a write operation
  - The system shall never reach that state if it is in this other state
- Another way of thinking about a design

$\Rightarrow$ A good way of catching bugs

Properties

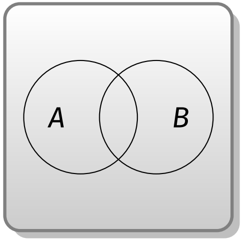
- Properties are based on
    - logic operations
    - temporal behaviors
    - A bit a logic, mathematically speaking

Implication

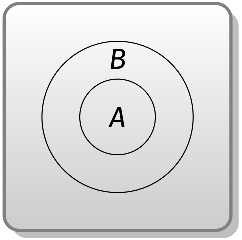
- A lot of properties will consist of an implication
- If A is observed, then B should be true
    - A and B could be simple boolean equations, or complex sequences running for several clock cycles
- Mathematically, an implication is represented as $\Rightarrow$
    - "*If A then B*" is equivalent to $A \Rightarrow B$

Implication and Venn diagram

Classical Venn diagram for two variables



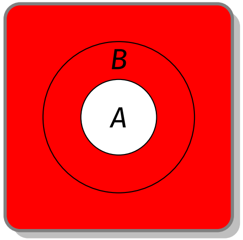
Representation of $A \Rightarrow B$



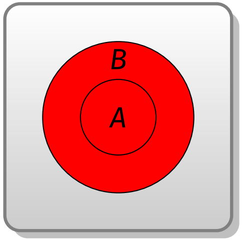
Implication: A $\Rightarrow$ B $\equiv \overline{A}$ or B

- Imagine:
  - $A \equiv$ is a crow, and $B \equiv$ is black
  - Is a crow $\Rightarrow$ is black
  - Space: everything that exists on earth

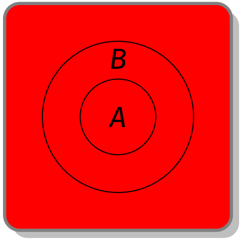
🟥 $\overline{A}$



🟥 B

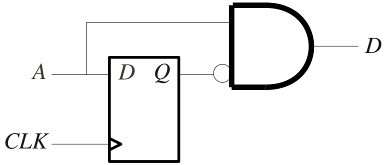


🟥 $\overline{A}$ or B



Formal verification: Simple example

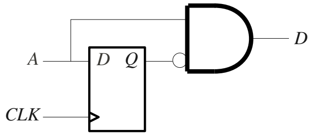
Rising edge detector



```
entity Detector is
port(
    Clk_i : in  std_logic;
    A     : in  std_logic;
    D     : out std_logic
    );
end Detector;
architecture behave of Detector is
    signal reg_s: std_logic;
begin
    process(clk_i)
    begin
        if rising_edge(clk_i) then
            reg_s <= A;
        end if;
    end process;
    D <= A and not(reg_s);
end behave;
```

Formal verification: Simple example

Rising edge detector



Properties that can be verified

If $A = 1$ and A was 0 at the previous cycle, then $D = 1$
If $A = 1$ then $D = 0$ at the next clock cycle
If $A = 0$ then $D = 0$
If $D = 1$ then $A = 1$
If $D = 1$ then $A = 0$ at the previous clock cycle
... ?

Can we prove one of these properties?

Formal verification: Simple example

Rising edge detector

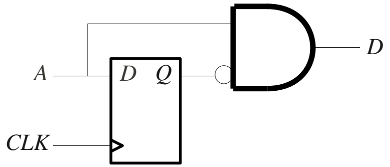
Property to verify

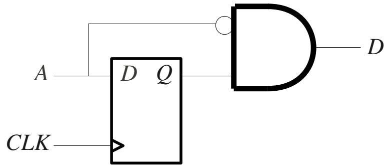
If $A = 1$ then $D = 0$ at the next clock cycle

$\Leftrightarrow (A_{t-1} = 1) \Rightarrow D_t = 0$

$\Leftrightarrow A_{t-1} \Rightarrow \overline{D_t}$

$\Leftrightarrow \overline{A_{t-1}} + \overline{D_t} = 1$ (because $A \Rightarrow B$ is identical to $\overline{A} + B$)



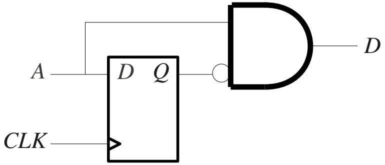


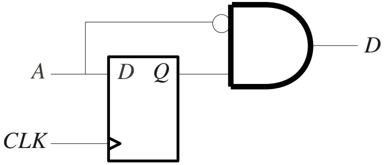
- What design is the correct one?

Formal verification: Simple example

Rising edge detector

- Property to verify: $\overline{A_{t-1}} + \overline{D_t} = 1$





- $D_t = A_t \cdot \overline{A_{t-1}}$
- So

$$\begin{aligned}\overline{A_{t-1}} + \overline{D_t} &= \overline{A_{t-1}} + \overline{A_t \cdot \overline{A_{t-1}}} \\ &= \overline{A_{t-1}} + \overline{A_t} + A_{t-1} \\ &= 1\end{aligned}$$

- $D_t = \overline{A_t} \cdot A_{t-1}$
- So

$$\begin{aligned}\overline{A_{t-1}} + \overline{D_t} &= \overline{A_{t-1}} + \overline{\overline{A_t} \cdot A_{t-1}} \\ &= \overline{A_{t-1}} + A_t + \overline{A_{t-1}} \\ &= \overline{A_{t-1}} + A_t \neq 1\end{aligned}$$

Formal verification: Simple example

Be careful with implications

Warning

$A \Rightarrow B$ is not the same as $B \Rightarrow A$

$A \Rightarrow B$ is not the same as $\overline{A} \Rightarrow \overline{B}$

Being here today $\Rightarrow$ Being an FPGA developer

vs

Being an FPGA developer $\Rightarrow$ Being here today

vs

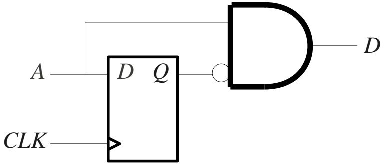
Not being here today $\Rightarrow$ Not being and FPGA developer

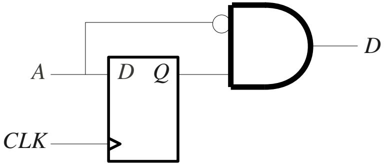
Formal verification: Simple example

Rising edge detector

Property to verify

If $D = 1$ then $A = 1$





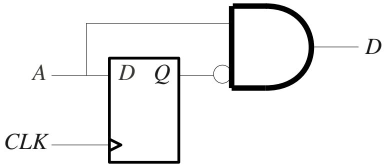
- What design is the correct one?

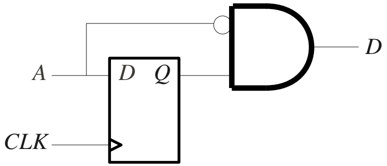
Formal verification: Simple example

Rising edge detector

Property to verify

If $A = 1$ then $D = 1$





- What design is the correct one?

Formal verification : Software tools

- Hopefully we do not need to do everything by hand, some software can help
- QuestaFormal (Closed Source)
  - PSL or SVA
  - Validates properties
  - Or finds a counterexample and shows a waveform
  - Very close to model checking technics
- SBY (Open Source)
  - PSL
  - Validates properties
  - Or finds a counterexample and shows a waveform
  - Various solvers (model checking or provers)
- ...

Formal verification

Languages

- Formal verification needs properties to be written
  - Useful for:
    - Assertions
    - Assumptions
    - Coverage
- Languages
  - PSL (Property Specification Language) : *language-agnostic*
  - SVA (SystemVerilog assertions) : Inherent to the language
  - OVL (Open Verification Library) : Built on top of VHDL/SVA

From Model checking to digital systems verification

- Model checkers or provers exploit some temporal logic
  - Linear Temporal Logic
  - Computational Tree Logic
  - ... Or variations on the themes

Linear Temporal Logic

ϕ ::=	$\top$	true
	$\bot$	false
	$\neg(\phi)$	negation
	p	proposition
	$(\phi \wedge \phi)$	conjunction
	$(\phi \vee \phi)$	disjunction
	$X\phi$	next time ϕ
	$F\phi$	eventually ϕ (strong operator)
	$G\phi$	always ϕ
	$\phi U \phi$	$\phi U \psi$: ϕ until ψ (strong operator)

Computational Tree Logic

$\phi ::=$	$\top$	true
	$\bot$	false
	p	a proposition
	$\neg(\phi)$	negation
	$(\phi \wedge \phi)$	conjunction
	$(\phi \vee \phi)$	disjunction
	$(\phi \Rightarrow \phi)$	implication
	$(\phi \Leftrightarrow \phi)$	equivalence
	$AX\phi$	In all subsequent states ϕ holds
	$EX\phi$	There exist a subsequent state such that ϕ holds
	$AF\phi$	In all futures there is at least one state where ϕ holds
	$EF\phi$	There exist at least a future in which ϕ holds
	$AG\phi$	ϕ holds in every state in the future
	$EG\phi$	There is one path on which ϕ holds in every state
	$A\,[\phi U \phi]$	ϕ holds until ψ on every path
	$E\,[\phi U \phi]$	There exist one path on which ϕ holds until ψ

PSL/SVA

LTL or?

- PSL
    - LTL-style : close to LTL (until, next, eventually, ...)
    - CTL-style : close to CTL (existence operator), called Optional Branching Extension (OBE), but it is not available in formal verification tools
    - SERE-style : exploits regular expressions
- SVA
    - LTL-style
    - SERE-style
- Differences
    - A lot of properties can be expressed as LTL or SEREs

Assertions in 5 minutes

In PSL

- Assertions need properties
- Properties need sequences and implications
- Sequences need ... a language

Sequence

Examples

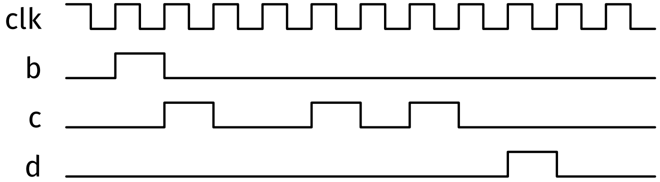
```
-- a followed by b
sequence seq_a is {a;b};

-- c followed by d disabled
sequence seq_b is {c;not d};
```

Sequence

Example

```
-- b followed by c active 3 times
-- (with c inactive in-between) followed by d
sequence seq_2 is {b;c[=3];d};
```



Property

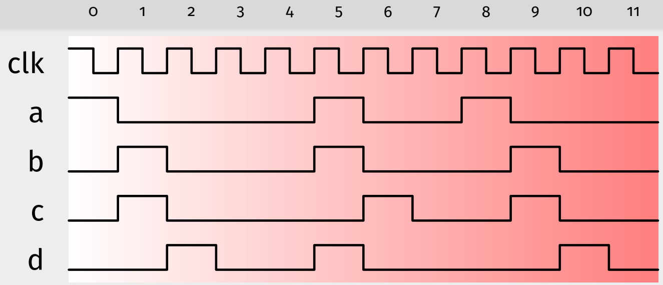
- Based on sequences and implications

Example

```
property prop is (seq_a |=> seq_b);
```

Implication operator

Example



Assertions

```
property {a;b} |-> {c;d};
```
The right-hand side starts directly

```
property {a;b} |=> {c;d};
```
The right-hand side starts one cycle after

Usage of properties

- Properties can be used by:
  - `assert` to verify a property
  - `assume` to specify assumptions on the environment. Helps reducing the state space.
  - `cover` to monitor properties

Exemples

```
a0 : assert prop;

a1 : assume never (a = b);

a2 : cover {a;b;c};
```

Mindset

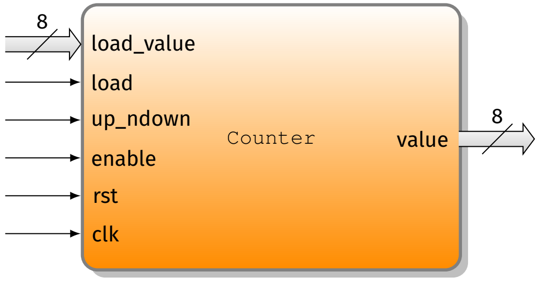
- A verification engineer should
  - Test interesting corner cases
  - Think about everything that could go wrong
  - Try to break the design
- Potential issues with simulation and tesbenches
  - Usage of code close to the design one for the reference model
  - Some corner cases can be hard to reach (or thought of)
- Thinking as properties
  - Another way to see the design
  - Less likely to be close to the design code
  - Even if the developer is the verification engineer, it offers a way to think differently about the system

Examples

Verification of a counter

DUV

- 8-bit counter
  - Parallel load
  - Incrementing
  - Decrementing
  - Hold



Verification of a counter

Testbench

1. Directed tests
   - Defined scenarios with borderline cases
2. Random generation of inputs
   - Coverage for verifying all borderline cases have been covered
     - Max to 0, or 0 down to Max

Verification of a counter

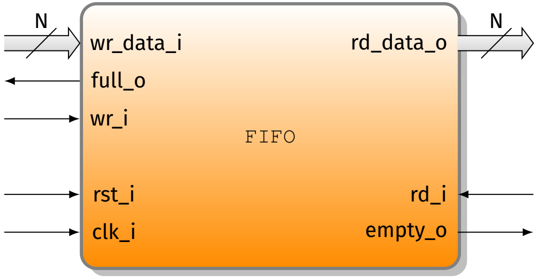
Properties to check

- `p_load`: If `load` is active, at the next clock cycle the output shall be the current input
- `p_hold`: If `load` and `enable` are active, the counter shall keep its value
- `p_incr`: If `load` is inactive, `enable` is active and `up_ndown` is active, the counter shall be incremented
- `p_decr`: If `load` is inactive, `enable` is active and `up_ndown` is inactive, the counter shall be decremented

Verification of a FIFO

DUV

- Simple FIFO
- Synchronous read/write
- Two outputs for the FIFO status (empty/full)



Verification of a FIFO

Testbench

1. Directed tests
    - Checks what happens when the FIFO is empty
    - Checks what happens when the FIFO is full
2. Random tests
    - Random commands
    - Coverage to be sure every relevant condition has been observed
        - Simultaneous read/write

Verification of a FIFO

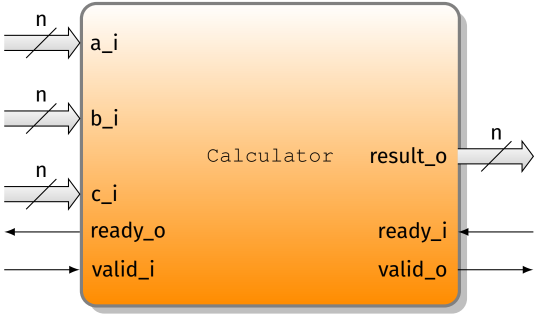
Formal proof

- Properties to verify
  - P1. If the number of writes equals the number of reads, then `empty_o` shall be active
  - P2. If the difference between the number of writes and the number of reads equals the FIFO size, then `full_o` shall be active
  - P3. The n^th^ data written shall be the n^th^ to go out
  - P4. `full_o` and `empty_o` shall never be active at the same time

Calculator

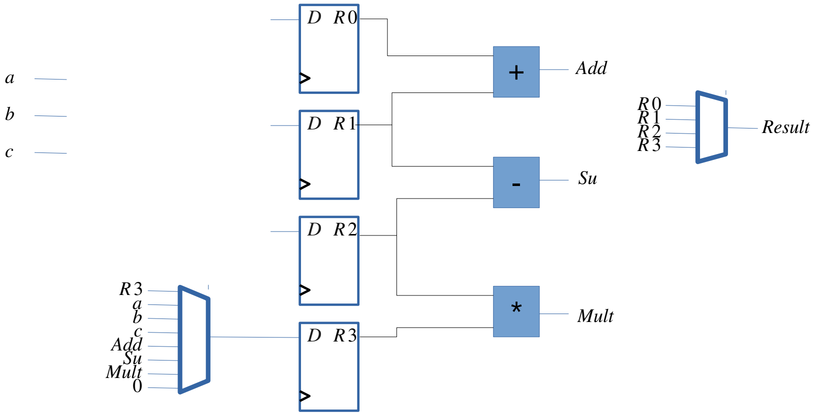
DUV

- A module able to compute the following function:
- $F(a, b, c) = a + a + b - c$
  - Could be any kind of arithmetic computation
- Different architectures:
  - Datapath-control
  - Pipelined
- A single testbench to validate different architectures



Verification of a calculator

Datapath



Verification of a calculator

Formal proof

- Properties to verify
  - P1. When a computation is launched, then after a certain time the result shall be available at the output
  - P2. The result of a computation shall correspond to the calculation (verification of data)
    - The nth result shall correspond to the nth input data
  - P3. When no computation is under way, then `ready_o` shall be active

Testbench vs assertions

Lines of code

Project	Lines TB	Lines Assertions
Counter	200	4
FIFO	200	30
FIFO multi	300	45
Calculator	350	50

Relevant lines

Synthesis

Design for verification

- Generic parameters as much as possible
  - Modular and reusable code (standard)
  - Allows to reduce the search space for formal proofs

$\Rightarrow$ Please use generics as much as possible
Or constants in a package

Synthesis of properties

- Some properties can be synthesized
  - For verification
    - Embedded in an emulation
  - For design
    - Generates a design compliant with the properties

Synthesis of a design

Automata approach

- European project PROSYD (`http://www.prosyd.org/`), 2004-2006
- From PSL properties, build a Mealy machine that matches the properties
  1. A two-player game played between a system and an environment
  2. The game structure is a multi-graph that represents the input variables and the state variables
     - Each node represents a combination of input and state variables that are valid
     - Each edge represents the transitions from a set of input and state variables to another set of valid input and state variables.
  3. If the environment wins, then the specification is *unrealizable*, and if the system wins, they extract a BDD representing this system
  4. In case the specification is *realizable*, the corresponding BDD is synthesized.
- Use cases: A generalized buffer and an AMBA bus arbiter
- Issues : Size of the design (and applicability)

Synthesis of a design

Modular approach

- Dominique Borrione and her team (Grenoble, France)
- Based on a library of monitors and generators
    - Corresponding to PSL operators
- Monitors detect correct or wrong behavior
- Generators produce sequences
- Allows to interpret LTL-style and SERE-style assertions
- Way more efficient in terms of size

Why not using that?

- Issue of generating a synthesizable design based on properties:
- Properties need to be complete
- The smallest incompleteness jeopardizes the process

Conclusion about formal verification

- The learning curve is quite steep
- Then pretty efficient
- Fewer number of code lines
- Good for control paths
- More difficult for data paths
- Another way of thinking than during design

$\Rightarrow$ Let’s do this!