1DT109 - Accelerating systems with FPGAs

Formal verification

Riccardo De Masellis

Uppsala University

October 4, 2021

Why verification?

To have an hardware free of bugs

Why verification?

To have an hardware free of bugs

- What is a bug?

Why verification?

To have an hardware free of bugs

- What is a bug?
- How do you define “correct behaviour?” (extensionally, intentionally?)

Specification using english

“This playground is forbidden to who: is shorter than 130cm or younger than 8 years old, if alone;

Specification using english

“This playground is forbidden to who: is shorter than 130cm or younger than 8 years old, if alone; or is shorter than 110cm or younger than 6 years old if accompanied”.

Specification using english

“This playground is forbidden to who: is shorter than 130cm or younger than 8 years old, if alone; or is shorter than 110cm or younger than 6 years old if accompanied”.

Example: formal verification of a counter

Implement a synchronous `counter` that counts up to 4 with an `enable` signal.

Example: formal verification of a counter

Implement a synchronous `counter` that counts up to 4 with an `enable` signal.

Ambiguous!

- Does the `counter` start from zero?
- if `enable`=1 then next value is previous value +1, but
- if `enable` $\neq 1$? Shall we reset? Or next value is equal to previous value?

Why FORMAL verification

(Un-formal) verification and testing are not exhaustive, in general:

- hand-written test benches;
- constrained random simulations.

Why FORMAL verification

(Un-formal) verification and testing are not exhaustive, in general:

- hand-written test benches;
- constrained random simulations.

Formal verification guarantees (more or less) absence of bugs, by:

- having unambiguous properties/specifications;
- analyzing all possible system behaviours.

Importance of HW verification

- In certain cases, especially in Embedded Systems, we can have critical components;
- Fixing HW is more expensive than fixing SW (e.g., Intel's bug).

Indeed, it was HW industry pushed the development of formal verification techniques, which is nowadays always used (for HW).

Importance of HW verification

- In certain cases, especially in Embedded Systems, we can have critical components;
- Fixing HW is more expensive than fixing SW (e.g., Intel's bug).

Indeed, it was HW industry pushed the development of formal verification techniques, which is nowadays always used (for HW).

Question

Is it possible to always guarantee that HW/SW is free of bugs (theoretically/practically)?

Digression: the halting problem

The halting problem (proved by A. Turing, 1936)

There is no program $halt(\cdot,\cdot)$ such that given as input any program $P(\cdot)$ and any input x, $halt(R,x)$ returns 1 is $R(x)$ terminates and 0 otherwise.

Digression: the halting problem

The halting problem (proved by A. Turing, 1936)

There is no program $halt(\cdot, \cdot)$ such that given as input any program $P(\cdot)$ and any input x, $halt(R, x)$ returns 1 is $R(x)$ terminates and 0 otherwise.

Proof intuition (informal) by contradiction.

- Suppose *halt* exists.
- Take the following program:

```
def R(x):
 if halt(R, x) then loop forever;
```

Digression: the halting problem

The halting problem (proved by A. Turing, 1936)

There is no program $halt(\cdot, \cdot)$ such that given as input any program $P(\cdot)$ and any input x, $halt(R, x)$ returns 1 is $R(x)$ terminates and 0 otherwise.

Proof intuition (informal) by contradiction.

- Suppose *halt* exists.
- Take the following program:

```
def R(x):
 if halt(R, x) then loop forever;
```

- now, if halt(R) is true (meaning: R terminates), then R loops forever, contradiction;
- therefore hypothesis on existence of *halt* is faulty.

Halting problem: in practice

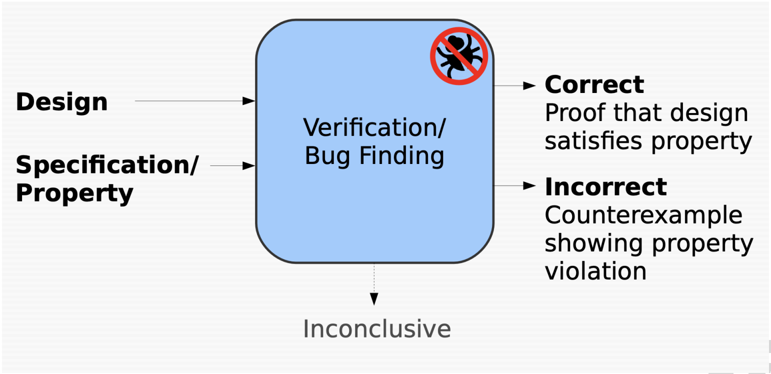
All programs that runs on a machine have finite memory, therefore finite inputs, thus in principle they could be formally verified.

Halting problem: in practice

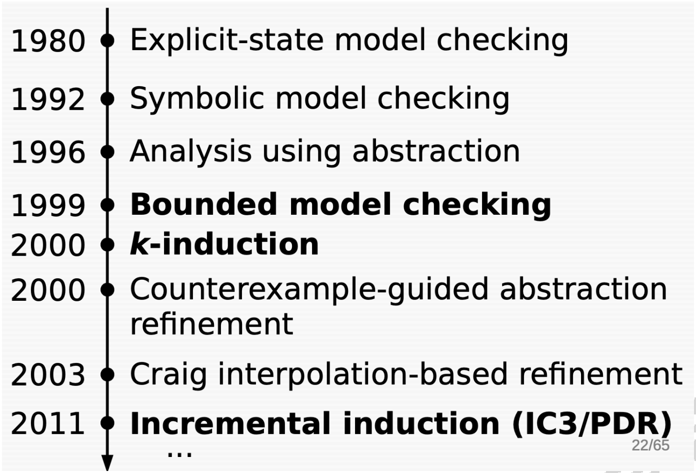
All programs that runs on a machine have finite memory, therefore finite inputs, thus in principle they could be formally verified.

Analyze extensively the behaviour of a program, where states are all possible combination for values of variables.

What formal verification does


Design
Specification/ Property
Verification/ Bug Finding
Correct
Proof that design satisfies property
Incorrect
Counterexample showing property violation
Inconclusive

Some techniques



1980 Explicit-state model checking

1992 Symbolic model checking

1996 Analysis using abstraction

1999 **Bounded model checking**

2000 ***k*-induction**

2000 Counterexample-guided abstraction refinement

2003 Craig interpolation-based refinement

2011 **Incremental induction (IC3/PDR)**

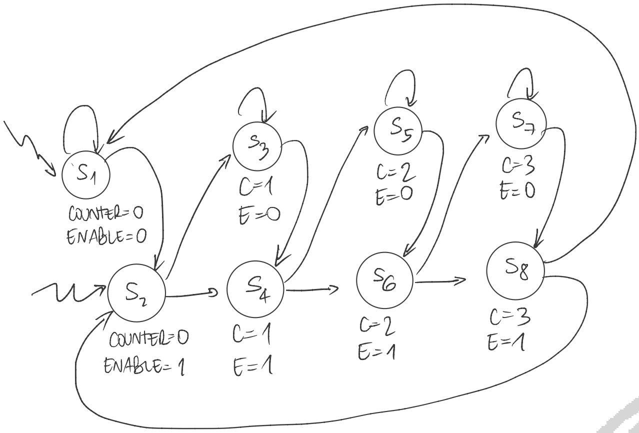
...

22/65

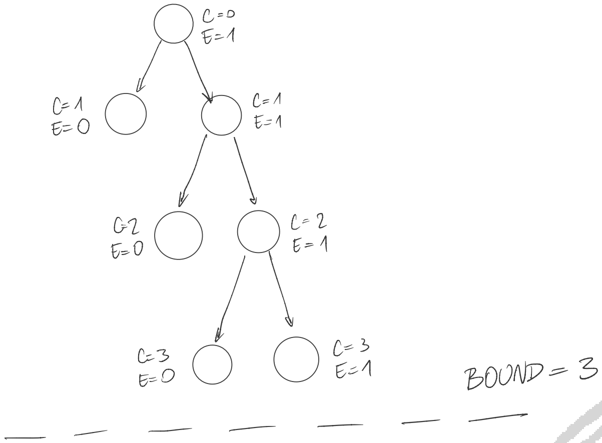
Explicit model checking, example

```
module counter(
  output [1:0] out, input enable, input clk);

  reg [1:0] count;
  assign out = count;

  initial count = 0;

  always @(posedge clk)
    if(enable)
     count = count + 1;
endmodule
```

Explicit model checking, example cont'd



Bounded model checking, example, cont'd



How to express properties

In a formal language.

We will use (restricted) temporal logic, which main operators are:

- always, in every state the property holds;
- next, in the next state the property holds;
- concatenation of *n* next, namely, after *n* steps a property hold.

How to express properties

In a formal language.

We will use (restricted) temporal logic, which main operators are:

- always, in every state the property holds;
- next, in the next state the property holds;
- concatenation of *n* next, namely, after *n* steps a property hold.

Full temporal logics are more expressive:

- until, something must hold until something else becomes true;
- ...

Explicit model checking (intuition)

- Each formula is some sort of “pattern”/automaton.

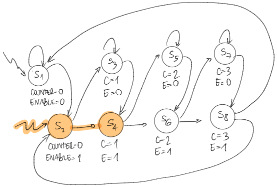
E.g., $\texttt{counter} = 0 \rightarrow \textit{next}\ \texttt{counter} = 2$

Explicit model checking (intuition)

- Each formula is some sort of “pattern”/automaton.

  E.g., $\texttt{counter} = 0 \rightarrow \textit{next}\ \texttt{counter} = 2$

- The algorithm checks if your model satisfies the pattern (by graph algorithms or by automata-based reasoning).

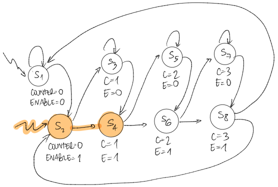


Explicit model checking (intuition)

- Each formula is some sort of “pattern”/automaton.

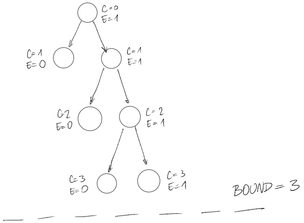
  E.g., $\texttt{counter} = 0 \rightarrow \textit{next}\ \texttt{counter} = 2$

- The algorithm checks if your model satisfies the pattern (by graph algorithms or by automata-based reasoning).



- And returns (ideally):
  - true if all executions satisfy the properties or
  - false, and a counterexample trace.

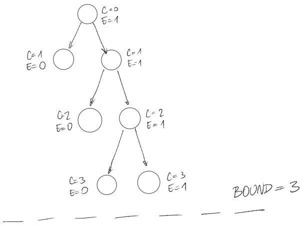
Bounded model checking (intuition)



- Relations between states is represented as a (constraint) boolean formula $R(c, e, c', e')$:

$$(c = 0 \land e = 1 \leftrightarrow c' = 1) \land (c = 0 \land e = 0 \leftrightarrow c' = 0) \land \dots$$

Bounded model checking (intuition)



- Relations between states is represented as a (constraint) boolean formula $R(c, e, c', e')$:

$$(c = 0 \wedge e = 1 \leftrightarrow c' = 1) \wedge (c = 0 \wedge e = 0 \leftrightarrow c' = 0) \wedge \ldots$$

- We unfold $R(c, e, c', e')$ a number of time equal to the bound (makes use of $2 \cdot 3 = 8$ variables):

$$\begin{array}{c}(c_0 = 0 \wedge e_0 = 1 \leftrightarrow c_1 = 1) \wedge \ldots \\ (c_1 = 0 \wedge e_1 = 1 \leftrightarrow c_2 = 2) \wedge \ldots\end{array}$$

- we add the property in conjunction and the initial condition:

$$\neg(c_0 = 0 \wedge c_0 = 2) \vee \neg(c_1 = 0 \wedge c_2 = 2) \vee \neg(c_2 = 0 \wedge c_3 = 2) \vee \ldots \wedge c_0 = 0$$

- if sat, then the property does not hold (truth assignment is the counterexample).

Differences between model checking techniques

- Explicit-state model checking suffers of state-explosion problem;
- symbolic model checking alleviates the problem;
- bounded model checking does not verify that all executions satisfies the property, as only bounded-depth executions are checked;
- k-induction use mathematical induction to prove that all executions satisfy the property (although not all properties are inductive).

Language we will use: `assert`

Conjunction of Always formulas:

`assert property(`Φ`)` where Φ can be:

Language we will use: `assert`

Conjunction of Always formulas:

$$\texttt{assert property}(\Phi) \quad \text{where } \Phi \text{ can be:}$$

- atomic formula, such as:

$$\texttt{unlock}, \texttt{count} < 4, \ldots$$

Language we will use: `assert`

Conjunction of Always formulas:

$$\texttt{assert property}(\Phi) \quad \text{where } \Phi \text{ can be:}$$

- atomic formula, such as:

$$\texttt{unlock}, \texttt{count} < 4, \ldots$$

- boolean combination of atomic formulas, e.g.,

$$\texttt{count} < 4 \wedge \texttt{code} \neq 3'b000$$

Language we will use: `assert`

Conjunction of Always formulas:

$$\texttt{assert property}(\Phi) \quad \text{where } \Phi \text{ can be:}$$

- atomic formula, such as:

$$\texttt{unlock}, \texttt{count} < 4, \ldots$$

- boolean combination of atomic formulas, e.g.,

$$\texttt{count} < 4 \wedge \texttt{code} \neq 3'b000$$

- (n-)next formulas:

$$\texttt{code} \neq 3'b000 \texttt{ |=> } \texttt{count} = 0$$

$$\texttt{code} \neq 3'b000 \texttt{ |-> } \#\#3 \ \texttt{count} = 0$$

Language we will use: `assume`

- `asserts` are properties we want to check (for every input);
- `assume` are assumptions (on the inputs).

`assume property` implies `assert property`

Language we will use: `assume`

- `asserts` are properties we want to check (for every input);
- `assume` are assumptions (on the inputs).

`assume property` implies `assert property`

```
assume property(enable = 0)
assert property(count |=> count)
```

Model checking verilog code

- Time is marked by the clock (combinatorial circuits are instantaneous, as in behavioural simulation);
- Inputs are selected by the model checker in all possible ways;
- When a (or more) input(s) changes, a new "stable" state is computed.

In practice

We will use the EBMC[1] model checker[2]. It can perform bounded model checking or incremental induction.

[1] `http://www.cprover.org/ebmc/`

[2] `http://logicrunch.it.uu.se:4096/~wv/ebmc/`

In practice

We will use the EBMC[1] model checker[2]. It can perform bounded model checking or incremental induction.

For each module `M` we want to formally verify, we write a verification module `ReqM` which will have a set of assert properties used to verify `ReqM`.

In EBMC you can choose between:

- bounded model checking (and set the bound);
- k-induction.

[1] `http://www.cprover.org/ebmc/`

[2] `http://logicrunch.it.uu.se:4096/~wv/ebmc/`

Example, counter

```
module counter(
  output [1:0] out, input enable, input clk);
...
endmodule

module counterReq(
       input enable, input clk);

 wire [1:0] out;
 counter our_count(out, enable, clk);

 assume property (...)
 assert property (...)
 assert property (...)

 endmodule
```

Suggestions

1. Most properties relate past values with new values: use registers in the `Req` module to save the past values.
2. Avoid latches at all costs in the design!