Accelerating Systems with Programmable Logic Comp.

A Hardware Verification Tutorial

May 4, 2020

Philipp Rümmer
Uppsala University
Philipp.Ruemmer@it.uu.se

Guest lecture

Mahdad Davari:
Custom Silicon Solutions at Ericsson

Wednesday May 6th, 10:15 – 12:00
(usual Zoom Meeting)

The projects

- Hardware is on the way to you, probably you will get it sometime this week. Let us know if you have not received it by **May 13!**
- Each package contains a return envelope, which you can use after the course to send back the boards
- We will introduce & start the project next week
  (email with more information to come)

Outline of this lecture

- Functional Verification, test benches
- SystemVerilog assertions
- Automated assertion checking: EBMC

- Lab 3

- Bounded Model Checking
- *k*-induction

Verifying Designs is Important ...

- Hardware is created that takes care of possibly critical functions
    - In particular in Embedded Systems
- Mistakes can be expensive
    - Intel's FDIV bug: ~1/2 billion \$ (needed to recall defective processors)
- Rule of thumb: ~70% of development time is spent with verification (same for hardware as for software)

What can be done?

- Testing + Simulation (~90% of effort)
  - Hand-written test benches that exercise interesting scenarios
    As done in the labs!
    (standard, but not very scalable)
  - Constrained-random simulation
    Randomly generate inputs, but only those satisfying given constraints
    (standard in industry)

You have already seen this!

- In Vivado/Verilog:
  Test benches written in Verilog
  (often using unsynthesizable code)
- In Vivado HLS:
  Test benches written in C
- In Chisel:
  Test benches written in Scala
  (also using random testing)

Figure 15-1. Traditional Verification Flow

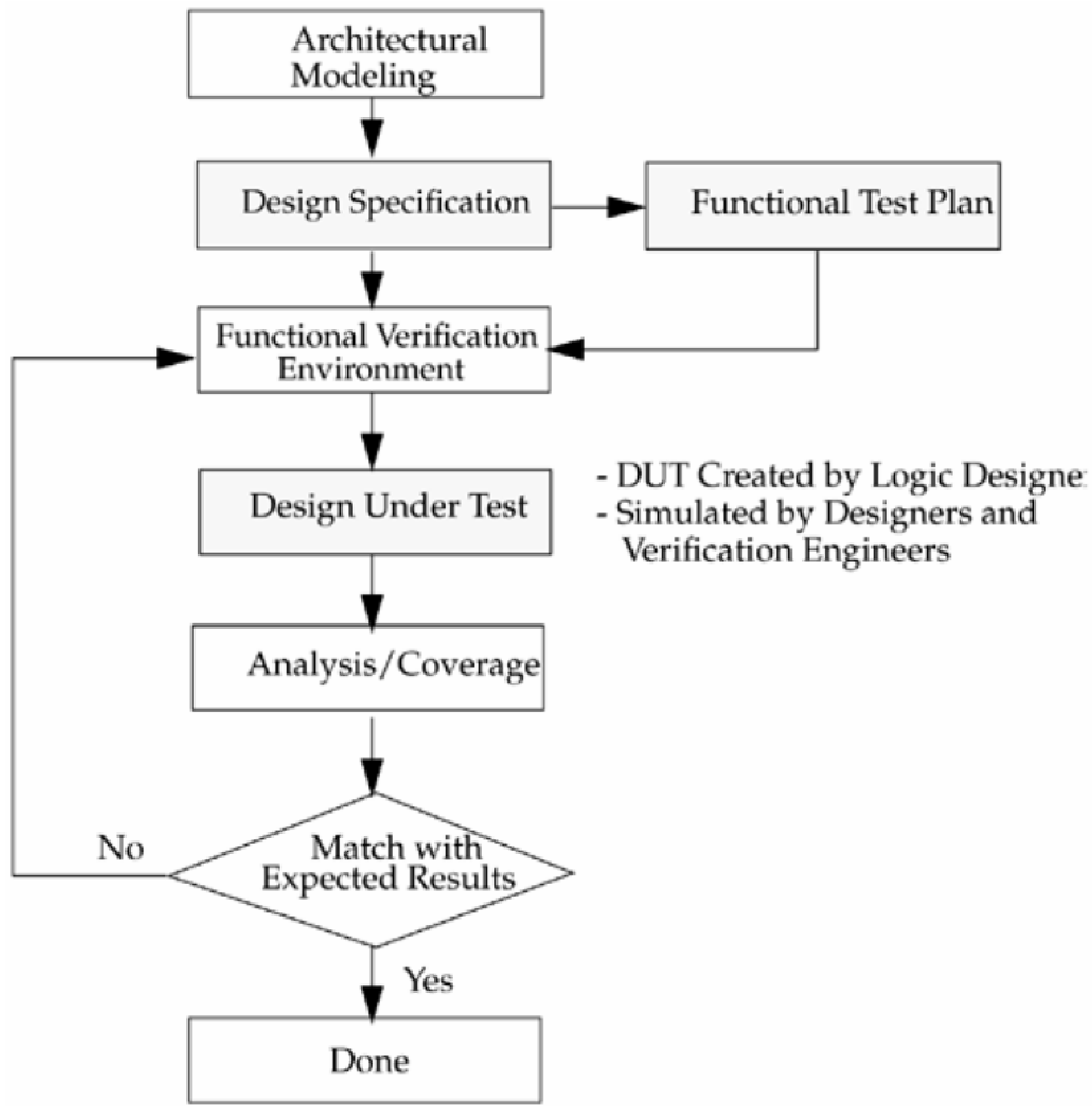
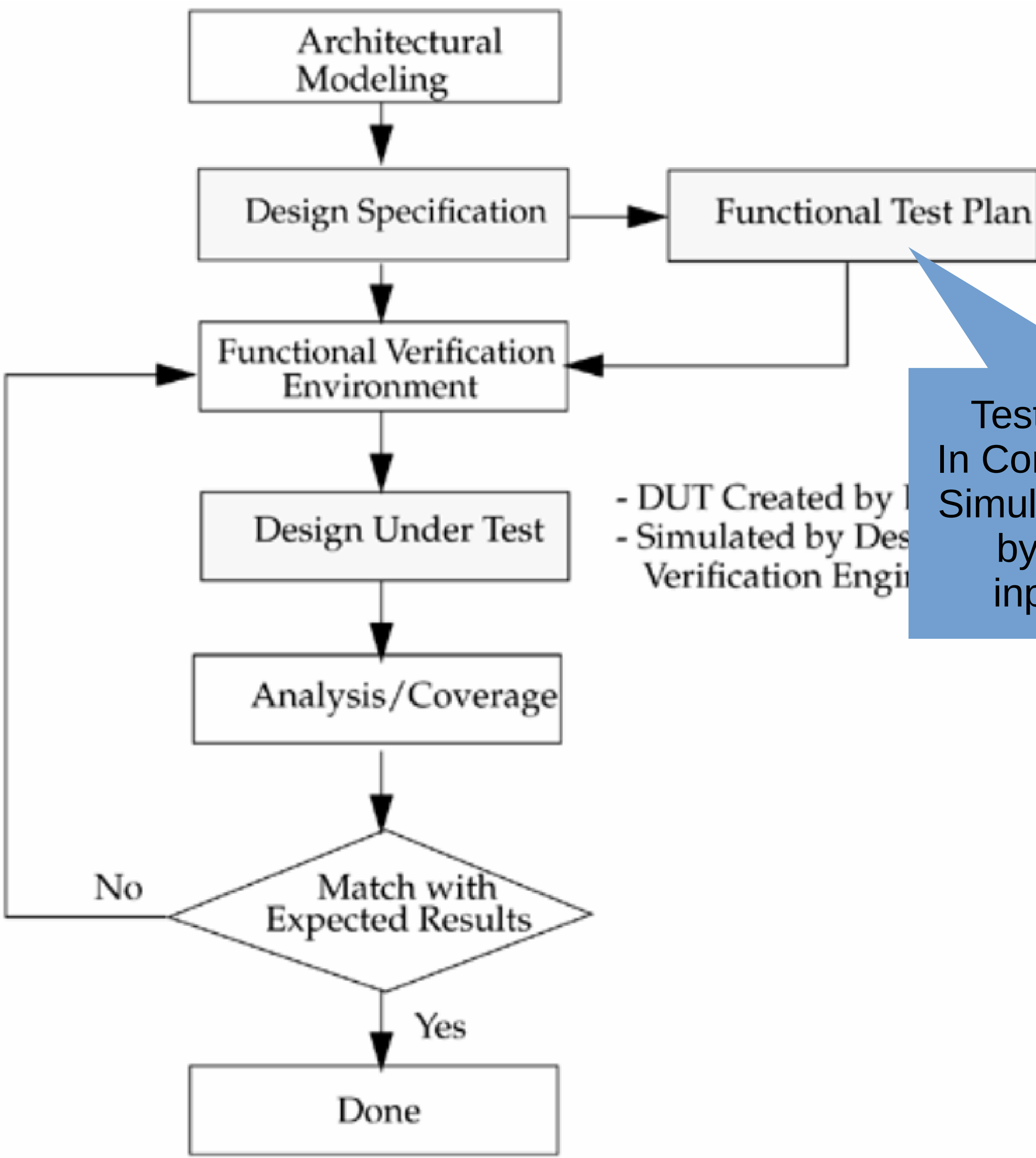
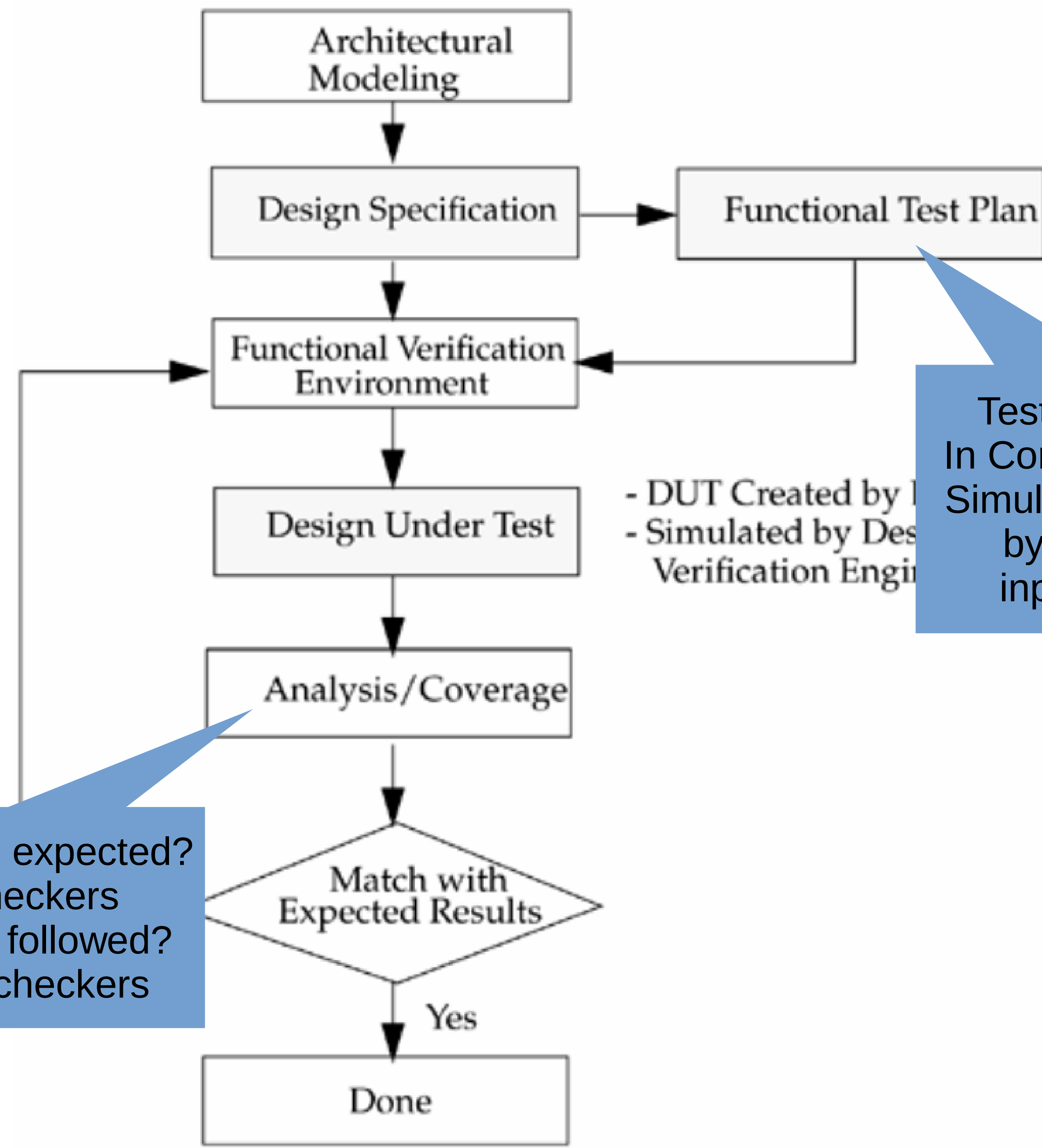


Figure 15-1. Traditional Verification Flow



Test data is defined.
In Constrained-Random Simulation, represented by contraints and input distributions

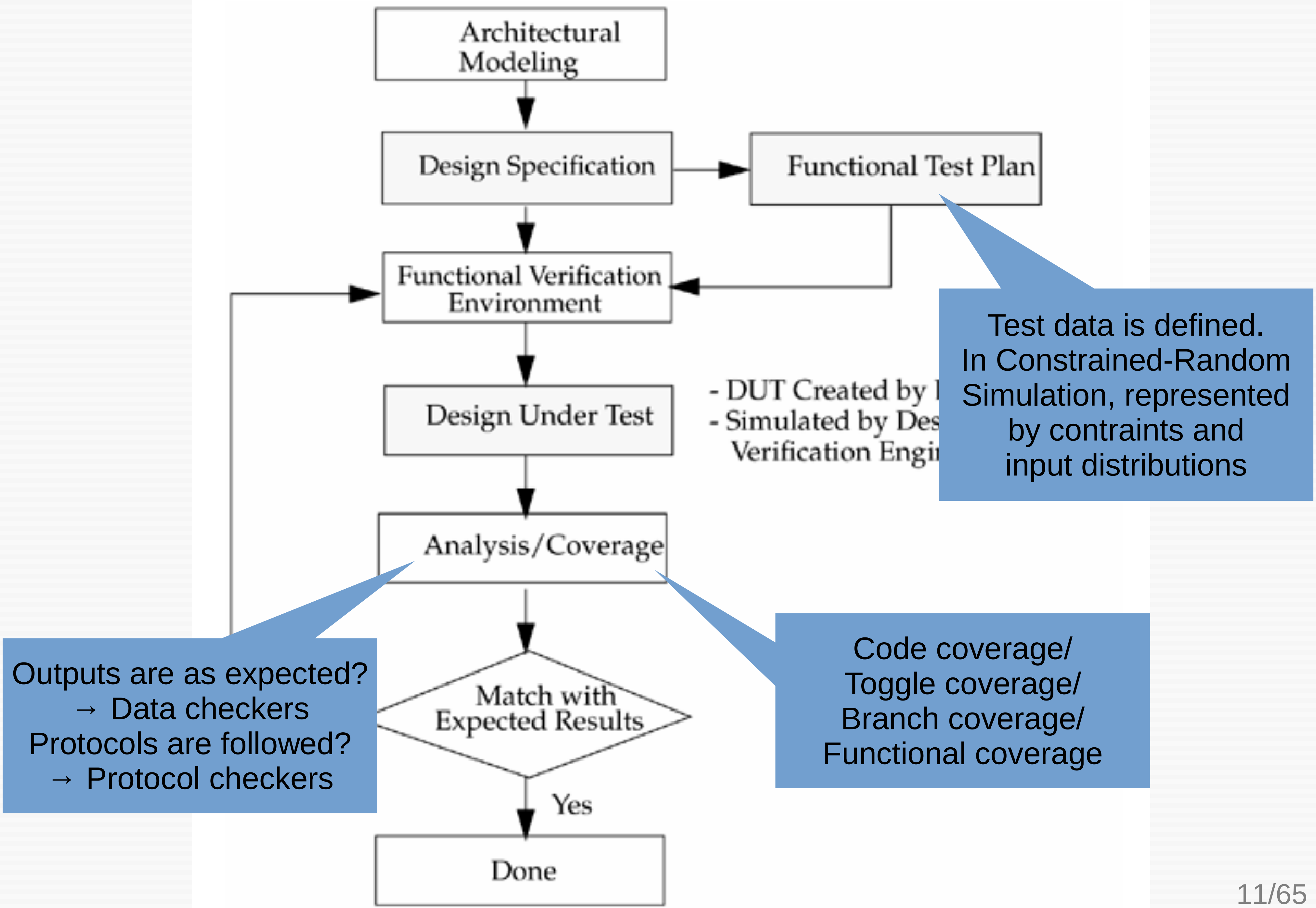
Figure 15-1. Traditional Verification Flow



Test data is defined.
In Constrained-Random Simulation, represented by contraints and input distributions

Outputs are as expected?
→ Data checkers
Protocols are followed?
→ Protocol checkers

Figure 15-1. Traditional Verification Flow



What can be done? (2)

- Formal verification (~10% of effort)
  - Statically analyse **all** possible behaviours
  - Most rigorous way to develop systems
  - Main bottleneck today: needs **properties/specifications:** What is a system supposed to do?
  - Focus of this lecture

What can be done? (3)

- Equivalence checking
    - Instance of formal verification

What can be done? (3)

- Equivalence checking
  - Instance of formal verification
  - Vertically: check that synthesis/place/route preserves behaviour

What can be done?

Have seen something similar in HLS: **co-simulation** is used to check conistency of C and RTL design

- Equivalence checking
  - Instance of formal verification
  - Vertically: check that synthesis/place/route preserves behaviour

What can be done? (3)

- Equivalence checking
  - Instance of formal verification
  - Vertically: check that synthesis/place/route preserves behaviour
  - Horizontally: check that modifications of a component preserve behaviour

What can be done? (3)

- Equivalence checking
  - Instance of formal verification
  - Vertically: check that synthesis/place/route preserves behaviour
  - Horizontally: check that modifications of a component preserve behaviour

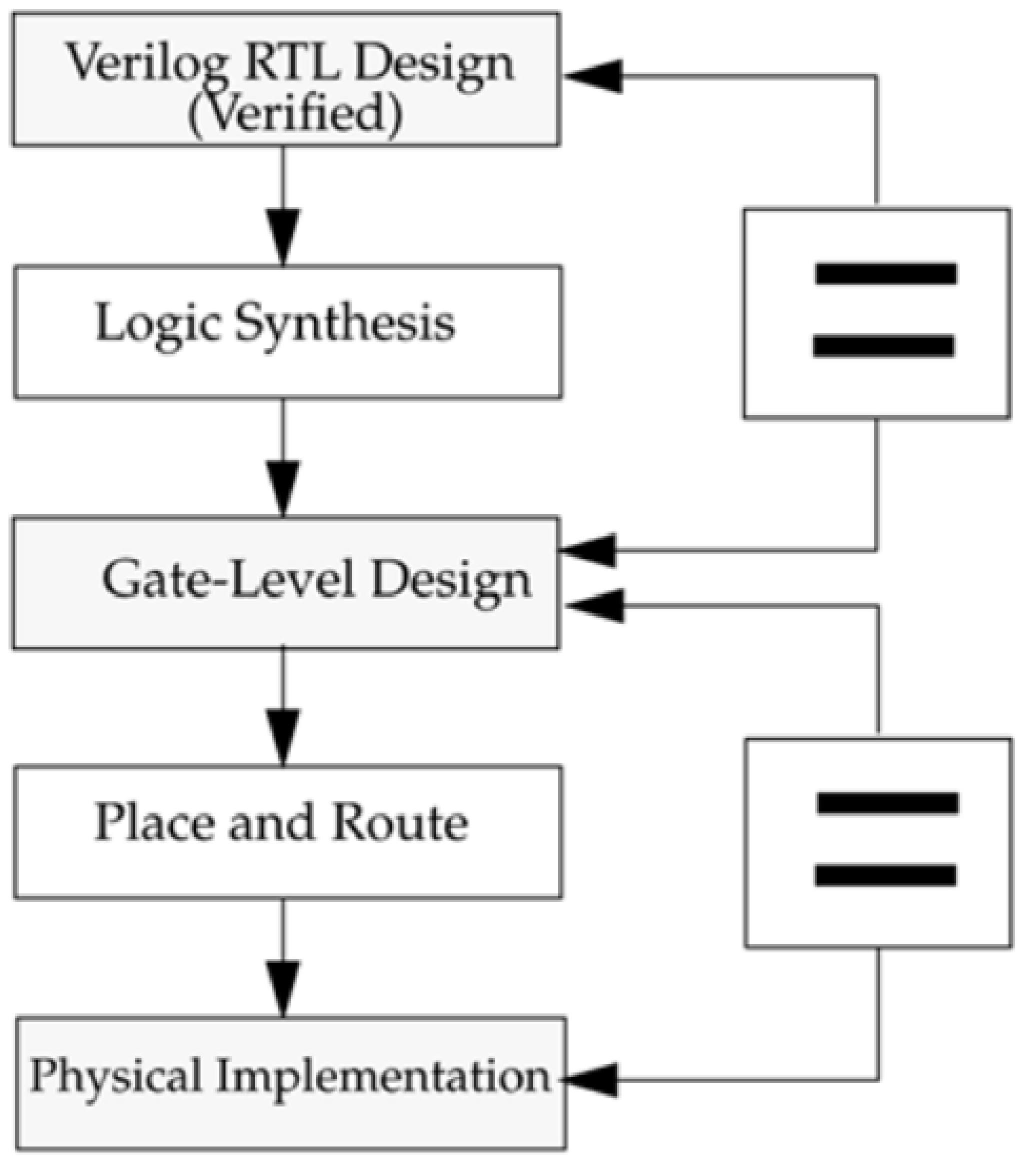
Used when manually fixing/tuning the output of synthesis

What can be done? (3)

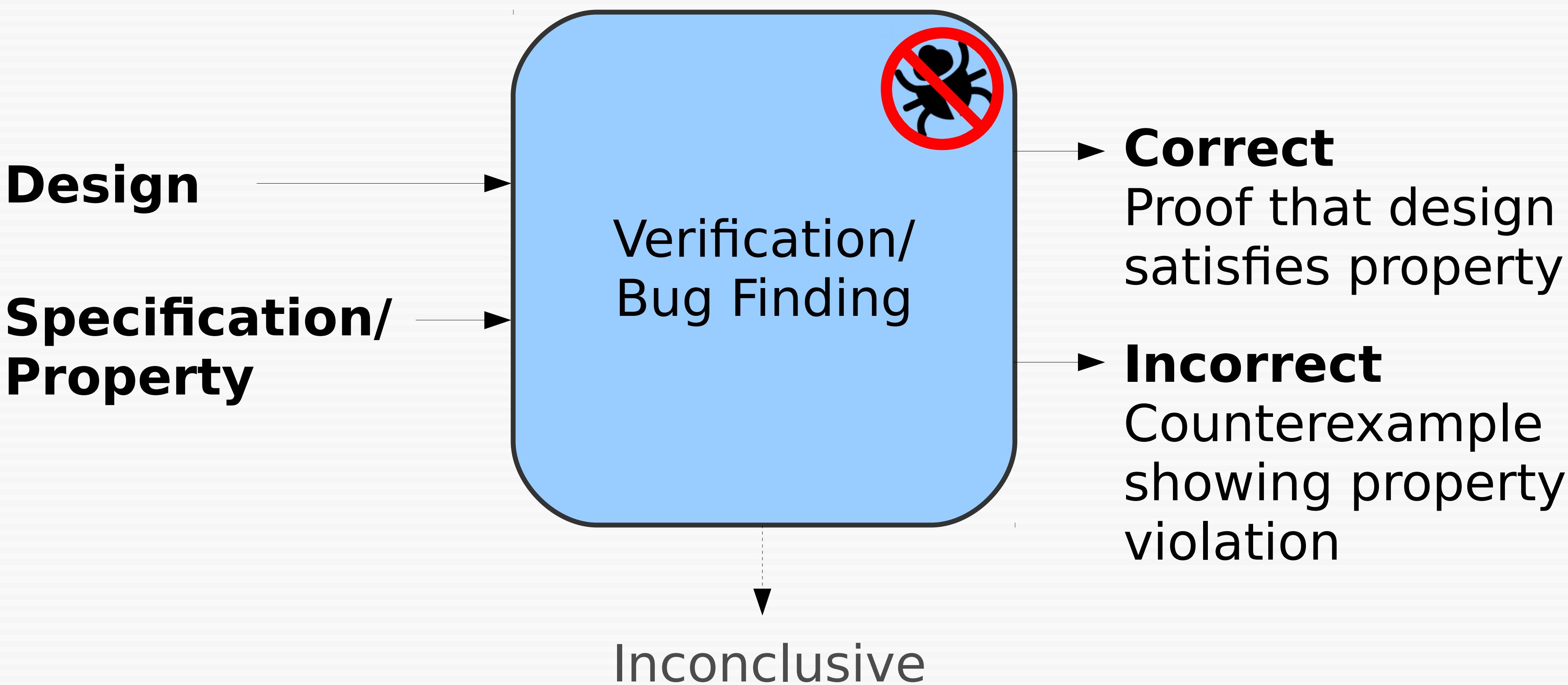
- Equivalence checking
    - Instance of formal verification
    - Vertically: check that synthesis/place/route preserves behaviour
    - Horizontally: check that modifications of a component preserve behaviour

    - Both are **extremely** important in industry

Figure 15-10. Equivalence Checking



Formal Functional Verification



Time-line: Methods in Hardware Verification

- 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)**
- ...

Examples 1

Observers/Monitors

- A module `ReqM` containing the asserted properties of a module `M`
- `ReqM` instantiates `M`
- `ReqM` has the same inputs as `M`
- Outputs of `M` become local wires in `ReqM`

Observers/Monitors (2)

```
module Max(input [7:0] a,
           input [7:0] b,
           output [7:0] m,
           input clk);
  assign m = a > b ? a : b;
endmodule

module ReqMax(input [7:0] a,
              input [7:0] b,
              input clk);
  wire [7:0] m;
  Max max(a, b, m, clk);
  assert property (m == a || m == b);
  always @(posedge clk) begin
    assert (m >= a && m >= b);
  end
endmodule
```

The EBMC Tool

- A verification tool for Verilog designs
  - Bounded model checking
  - *k*-induction
  - Symbolic model checking
- Partial support for SystemVerilog assertions
- http://www.cprover.org/ebmc/
- Web interface: http://logicrunch.it.uu.se:4096/~wv/ebmc/

The EBMC Tool

- A verification tool for Verilog designs
  - Bounded model checking
  - *k*-induction
  - Symbolic model checking
- Partial support for SystemVerilog assertions
- http://www.cprover.org/ebmc/
- Web interface: http://logicrunch.it.uu.se:4096/~wv/ebmc/

We will mostly focus on this!

What does “SUCCESS” mean?

- Intuitively:
  A **mathematical proof** has been found that given properties cannot be violated
  (but only **up to the specified bound)**
- Different from testing and simulation:
  - **All** possible inputs and scenarios have been considered
  - **However**: the assumption is made that compiler/synthesis/hardware are correct

What does “FAILURE” mean?

- For some inputs, an assertion violation can occur within the specified bounds

BMC vs *k*-Induction

- Bounded Model Checking only analyses system up to a certain bound
  - Here, *k* first cycles
- *k*-Induction tries to verify properties for any depth
  - But sometimes fails, and will then return UNKNOWN

- More about both methods later

SystemVerilog Assertions

- `assert`: can be used in behavioural blocks, **assert** properties in specific cycles
- `assert property`: continuously **assert** some property ("concurrent assertion")
- `assume property`: continuously **assume** some property
- https://www.design-reuse.com/articles/10907/using-systemverilog-assertions-in-rtl-code.html

Difference between `assume` and `assert`?

- `assume`: what does a module assume about its environment?

- `assert`: which properties is a module supposed to satisfy?

Examples 2

Temporal requirements

- Examples so far are on the **propositional** level
- Interesting requirements often contain **temporal** aspects; their statements span multiple cycles of system execution

- SystemVerilog has temporal operators; more general stuff can be encoded

Examples 3

Temporal SystemVerilog Assertions

- `A |=> B`
  - Similar as implication `|->`, but evaluate `B` in the **next cycle**
  - "non-overlapping"
- `##<`*n*`> A`
  - Evaluate `A` n cycles in the future
  - (only partially supported by EBMC)
  - `A |=> B` is the same as `A |-> ##1 B`
- (assertions are always tied to a clock!)

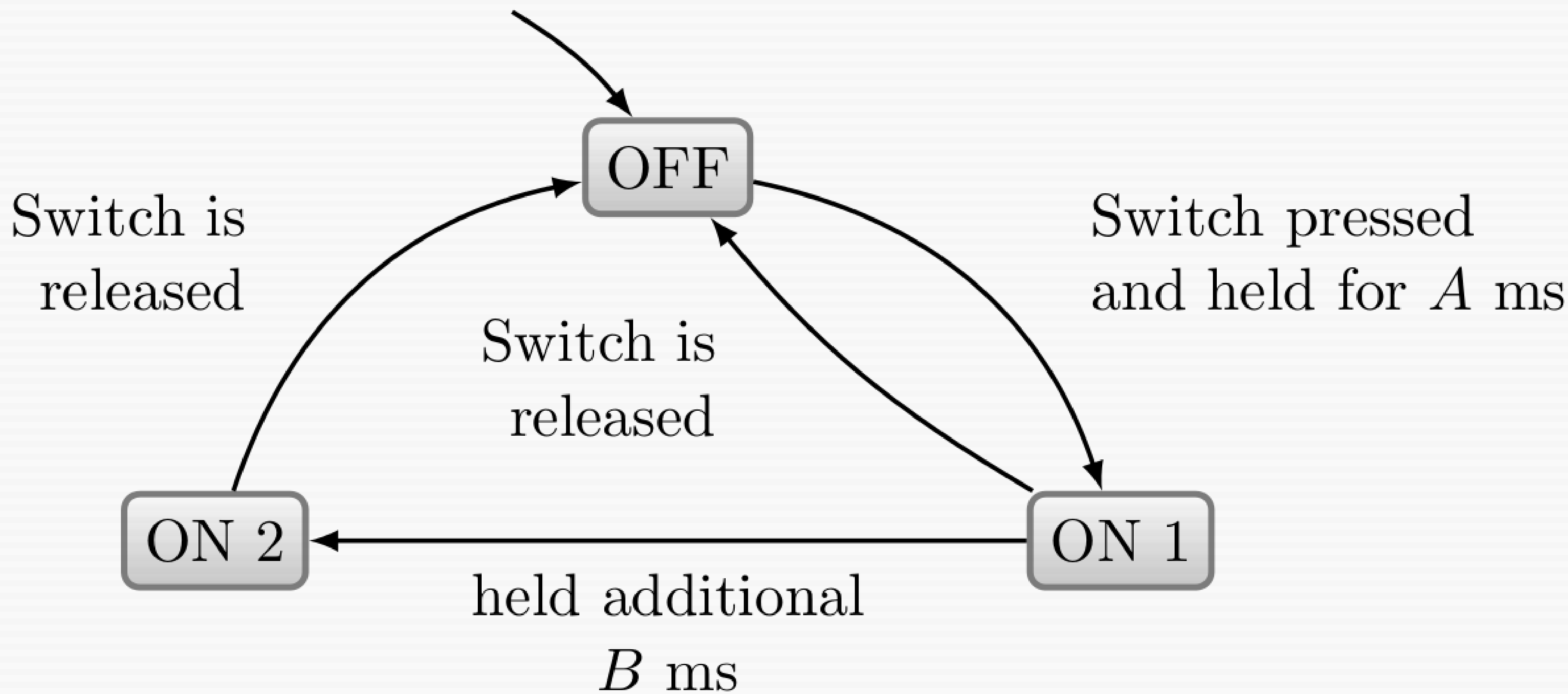
More General Patterns of Temporal Reasoning

- “`X` has happened”
- “`X` never happens for more than `Y` consecutive cycles”
- “Since `X` happened, `Y` has been true”

Safety vs. Liveness

- Different classes of requirements
- Safety:
  - **“Something bad never happens.”**
- Liveness:
  - **“Eventually, something good happens.”**
- **Looking into the past, we can only express safety properties!**
- (But bounded liveness can be done)

MultiStateSwitch



Properties

- **R1:** `On1` and `On2` are never true at the same time
- **R2:** If `On2` is true, then `On1` has been true sometime in the past
- **R3:** If `On2` is true and the button is not released, `On2` stays true

Rigorous Design

- **Requirements** are first formulated as text (in, say, English)
- Textual requirements are translated to **formal expressions**
- Formal requirements are put in an **observer** or **monitor** (similar to a test bench or stimulus)
- Correctness of design is checked using **testing** or **model checking**

From Text to Assertions

- Textual requirements often use patterns with commonly understood meaning
- But: text is not always unambiguous; writing good/precise requirements can be difficult

Common English patterns

English	Logic	SystemVerilog (similar for C)
A and B A but B	A & B	A && B
A if B A when B A whenever B	...	...
if A, then B A implies B A forces B		
only if A, B B only if A		
A precisely when B A if and only if B		
A or B either A or B		
A or B		

Ambiguous;
to clarify, write
“either A or B”
or
“A or B, or both”

Common English patterns

English	Logic	SystemVerilog (similar for C)
A and B A but B	A & B	A && B
A if B A when B A whenever B	B => A	B |-> A !B || A
if A, then B A implies B A forces B	A => B	A |-> B
only if A, B B only if A	B => A	B |-> A
A precisely when B A if and only if B	A <=> B	A == B
A or B either A or B	A (+) B (exclusive or)	A != B
A or B	A v B (logical or)	A || B

Ambiguous;
to clarify, write
“either A or B”
or
“A or B, or both”

Lab 3: Implementing & Verifying a Door Lock

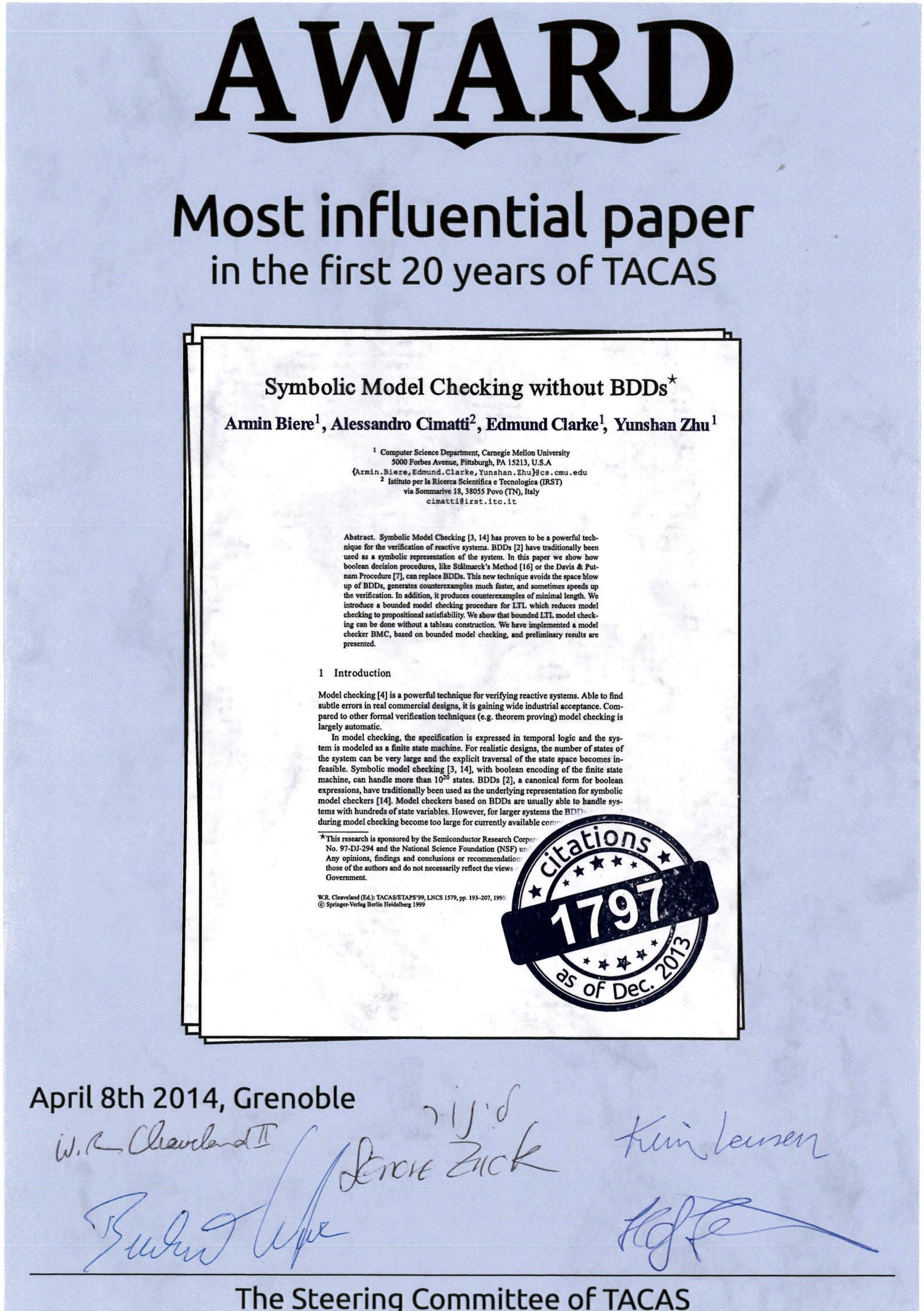
Main techniques of EBMC

- **Bounded model checking**
  - Constraint solving to detect error traces/counterexamples
  - Internally uses a SAT solver
  - Standard technique when designing hardware
- ***k*-Induction**
  - Strong form of mathematical induction
  - Prove that requirements hold

Bounded model checking

- *Idea:* search for bugs in programs/systems up to some depth; but otherwise reason **fully precisely**
- Tailored to showing **reachability** (e.g., **finding bugs**), not so much **unreachability**
- The workhorse of formal hardware verification

BMC Problem
Decide whether an error can be reached within the first k execution steps of a program/system.



AWARD

Most influential paper
in the first 20 years of TACAS

Symbolic Model Checking without BDDs*

Armin Biere[1], Alessandro Cimatti[2], Edmund Clarke[1], Yunshan Zhu[1]

[1] Computer Science Department, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, U.S.A
{Armin.Biere,Edmund.Clarke,Yunshan.Zhu}@cs.cmu.edu
[2] Istituto per la Ricerca Scientifica e Tecnologica (IRST)
via Sommarive 18, 38055 Povo (TN), Italy
cimatti@irst.itc.it

Abstract. Symbolic Model Checking [3, 14] has proven to be a powerful technique for the verification of reactive systems. BDDs [2] have traditionally been used as a symbolic representation of the system. In this paper we show how boolean decision procedures, like Stålmarck's Method [16] or the Davis & Putnam Procedure [7], can replace BDDs. This new technique avoids the space blow up of BDDs, generates counterexamples much faster, and sometimes speeds up the verification. In addition, it produces counterexamples of minimal length. We introduce a bounded model checking procedure for LTL which reduces model checking to propositional satisfiability. We show that bounded LTL model checking can be done without a tableau construction. We have implemented a model checker BMC, based on bounded model checking, and preliminary results are presented.

1 Introduction

Model checking [4] is a powerful technique for verifying reactive systems. Able to find subtle errors in real commercial designs, it is gaining wide industrial acceptance. Compared to other formal verification techniques (e.g. theorem proving) model checking is largely automatic.

In model checking, the specification is expressed in temporal logic and the system is modeled as a finite state machine. For realistic designs, the number of states of the system can be very large and the explicit traversal of the state space becomes infeasible. Symbolic model checking [3, 14], with boolean encoding of the finite state machine, can handle more than 10^{20} states. BDDs [2], a canonical form for boolean expressions, have traditionally been used as the underlying representation for symbolic model checkers [14]. Model checkers based on BDDs are usually able to handle systems with hundreds of state variables. However, for larger systems the BDD
during model checking become too large for currently available com

* This research is sponsored by the Semiconductor Research Corpor
No. 97-DJ-294 and the National Science Foundation (NSF) un
Any opinions, findings and conclusions or recommendatio
those of the authors and do not necessarily reflect the views
Government.

W.R. Cleaveland (Ed.): TACAS/ETAPS'99, LNCS 1579, pp. 193–207, 1999

citations
1797
as of Dec. 2013

April 8th 2014, Grenoble

The Steering Committee of TACAS

ETAPS Test of Time Award 2017
(Awarded in the Uppsala Castle)

Bounded model checking

- Every Verilog design can be represented as a Boolean circuit/equation
- E.g.:

```
module Counter(output reg [15:0] c,
               input clk);
  initial c = 0;
  always @(posedge clk) c = c + 1;
endmodule
```

↓

$$c_0 = 0$$
$$c_{i+1} = c_i + 1$$

Bounded model checking (2)

- We can duplicate the circuits to generate counterexamples for properties
- Let's say, we try to prove for the counter that

  “*c* is always less than 10”

  (does not hold)

Bounded model checking (3)

- Generate *k* copies of the circuit:

$$c_0 = 0$$
$$c_{i+1} = c_i + 1$$

→

$$\begin{aligned} c_0 &= 0 \\ c_1 &= c_0 + 1 \\ c_2 &= c_1 + 1 \\ c_3 &= c_2 + 1 \\ &\vdots \\ c_{15} &= c_{14} + 1 \end{aligned}$$

Bounded model checking (4)

- Check whether new circuits imply property:

$$\begin{aligned} c_0 &= 0 \\ c_1 &= c_0 + 1 \\ c_2 &= c_1 + 1 \\ c_3 &= c_2 + 1 \\ &\vdots \\ c_{15} &= c_{14} + 1 \end{aligned} \quad \Longrightarrow \quad \begin{gathered} c_0 \leq 10 \land c_1 \leq 10 \\ \land \cdots \land \\ c_{15} \leq 10 \end{gathered}$$

- A SAT solver can check this quickly ... and produce a counterexample

Bounded model checking (5)

- Bounded model checking can often show very quickly that some requirement **does not hold**

- **What if a requirement holds?**
    - Second technique in EBMC: *k*-induction

What is *k*-induction?

Imagine Fibonacci numbers ...

$$
\begin{aligned}
f_0 &= 0 \\
f_1 &= 1 \\
f_2 &= 1 \\
&\vdots \\
f_{i+2} &= f_i + f_{i+1}
\end{aligned}
$$

Let's prove that all Fibonacci numbers are non-negative:

$$\forall i.\ f_i \geq 0$$

$$f_0 = 0$$
$$f_1 = 1$$
$$f_2 = 1$$
$$\vdots$$
$$f_{i+2} = f_i + f_{i+1}$$

Proof using standard induction

- To show $\forall i.\ f_i \geq 0$
  we prove:
    - Base case: $f_0 \geq 0$
    - Step case: $f_i \geq 0 \Rightarrow f_{i+1} \geq 0$

Proof using standard induction

- To show $\forall i.\ f_i \geq 0$
  we prove:
  - Base case: $f_0 \geq 0$
  - Step case: $f_i \geq 0 \Rightarrow f_{i+1} \geq 0$

- *Does not work for Fibonacci numbers*

Induction with two base cases *(2-induction)*

- To show $\forall i.\ f_i \geq 0$
  we can also prove:
  - Two base cases:

    $$f_0 \geq 0,\ f_1 \geq 0$$

  - "Simpler" step case:

    $$f_i \geq 0 \wedge f_{i+1} \geq 0 \ \Rightarrow\ f_{i+2} \geq 0$$

- *Works for Fibonacci numbers!*

k-Induction

- Generalises 2-induction to *k* base cases
- **Can be used to verify properties/requirements *P* of sequential circuits!**
    - **Base case:** prove that *P* holds in cycles 0, 1, 2, ..., (*k-1)*
    - **Step case:** assume that P holds in cycle *i*, *i*+1, *i*+2, ..., *i*+*k*-1, then prove that *P* also holds in cycle *i*+*k*

Non-inductive properties

- For some properties *P*, it can happen that step case fails, even though P always holds → *P* is **not inductive**

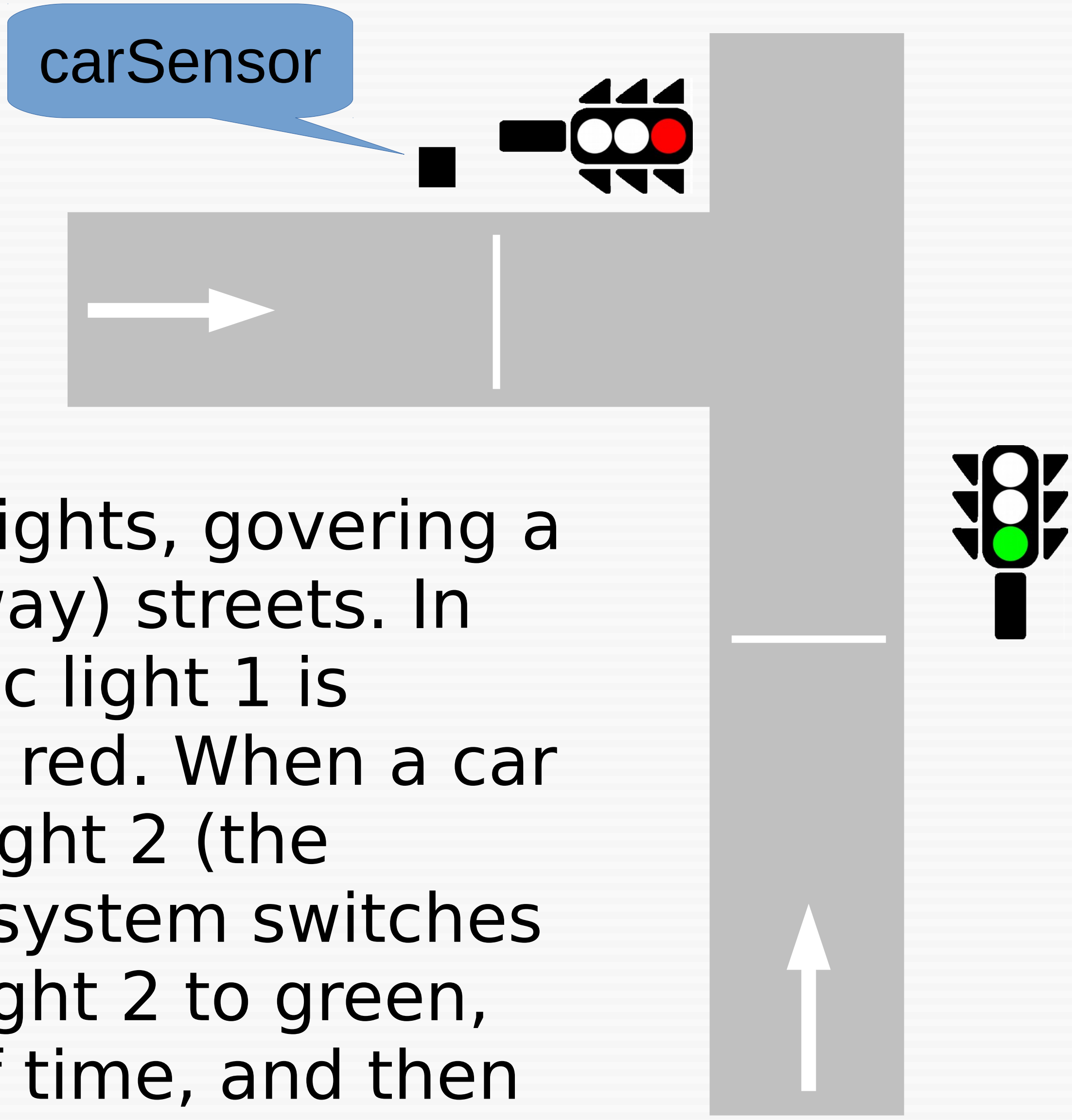
- E.g., $\forall i.\ f_i \geq 0$ is not inductive for *k*=1 (but for *k*=2)
- Some properties are not inductive for any *k*!

What to do in case of non-inductive properties?

- Method 1: **strengthen** the property P
  - verify not only P, but a stronger property **P & Q**

- Method 2: make the program to be verified more **defensive**
  - handle some cases that cannot actually occur
    → EBMC might not be able to detect that the cases cannot occur

Exercise: back to the Traffic lights



System of two traffic lights, govering a junction of two (one-way) streets. In the default case, traffic light 1 is green, traffic light 2 is red. When a car is detected at traffic light 2 (the carSensor input), the system switches traffic light 1 to red, light 2 to green, waits some amount of time, and then switches back to the default situation.

Some Traffic light Properties

- For each traffic light, no red and green at the same time
- Some signal is always shown
- It cannot happen that the two traffic light are green at the same time

Further reading

- A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, 1999: “Symbolic Model Checking without BDDs”
- Sheeran, Singh, Stålmark, 2000: “Checking Safety Properties Using Induction and a SAT-Solver”