# COMSM0115 Design Verification:
# **Checking**

# Kerstin Eder

University of
BRISTOL

Department of
COMPUTER SCIENCE

# Last Time

- **High-level Verification**
  - sn and e
  - A2 demo
- **Stimuli Generation**

# Checking - Outline

- Motivation
- Issues in checking
  - When to check
  - What to check
- Checking technologies
  - Reference models
  - Scoreboards
  - (Rule-based checking)
    - (Assertions)
- Assertion-based verification (ABV) – later ☺

# The Yin-Yang of Verification

- Driving and checking are the yin and yang of verification

  – We cannot find bugs without creating the failing conditions

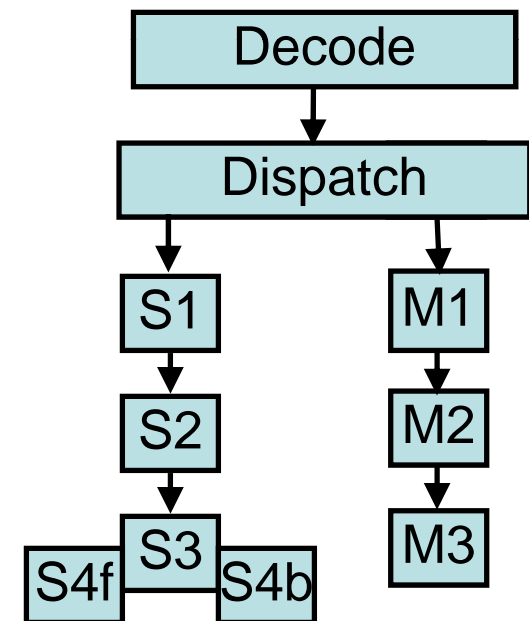  – We cannot find bugs without detecting the incorrect behavior
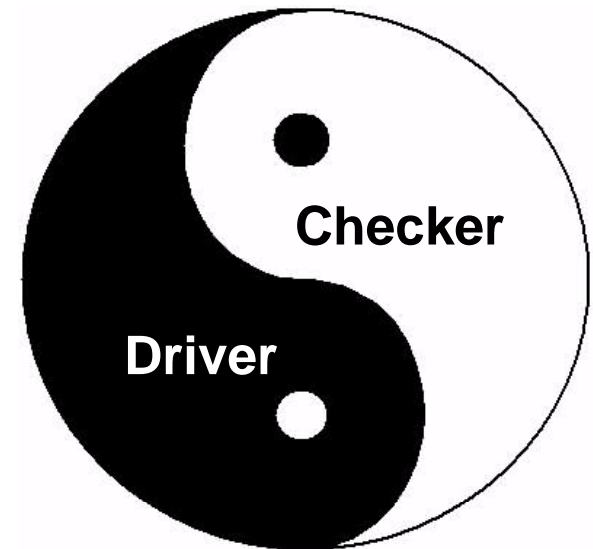
# Ideal Checking

- In theory – detect deviation from expected behavior as soon as it happens and where it happens
  - No need to worry about "disappearing errors"
  - Easy to debug – the checker points to the bug
- This is not easy (even if we ignore many practical aspects) because in many cases we understand that something bad happened only in retrospect
  - Several "good" behaviors collide to create a bad behavior
- And what about the bugs we are not looking for

# "Good" Behavior Collision

- At cycle 1000 fdiv F1, F2, F3 is dispatched to the M unit
  - It reaches stage M2 at cycle 1001
  - Its execution time is 60 cycles
- At cycle 1023 fld F1,100(G2) is dispatched to the S unit
  - It reaches stage S2 at cycle 1024
- The data returns from the cache at cycle 1060
- At cycle 1061 the fdiv is ready to write
  - It moves to stage M3
- At cycle 1061 the fld is ready to write
  - It moves to stage S3
- Both instruction write to the same register together

```
        Decode
          │
          ▼
       Dispatch
        │       │
        ▼       ▼
       S1      M1
        │       │
        ▼       ▼
       S2      M2
        │       │
        ▼       ▼
   S4f S3 S4b  M3
```

# "Good" Behavior Collision

- There are many possible causes for the problem

# Practical Aspects

- **Ground rules**
  - Only black-box checking is allowed
- **The cost of implementation and maintenance**
  - Against the cost of debugging
- **The cost of mistakes**
  - Misdetection
    - We failed to detect a bug that was exposed by the stimuli.
  - False alarm
    - We mistakenly flagged a good behavior as bad.
  - Which is more expensive?

# When to Check?

- **Checking can be done at various stages of the verification job**
  - During simulation
    - On-the-fly checking
  - At the end of simulation
    - End-of-test checking
  - After the verification job finishes
    - External checking
- **Checking at each stage has its own advantages and disadvantages**

# On-the-fly Checking

- Checking is done **while the simulation is running**
- The DUV is continuously monitored to detect erroneous behavior
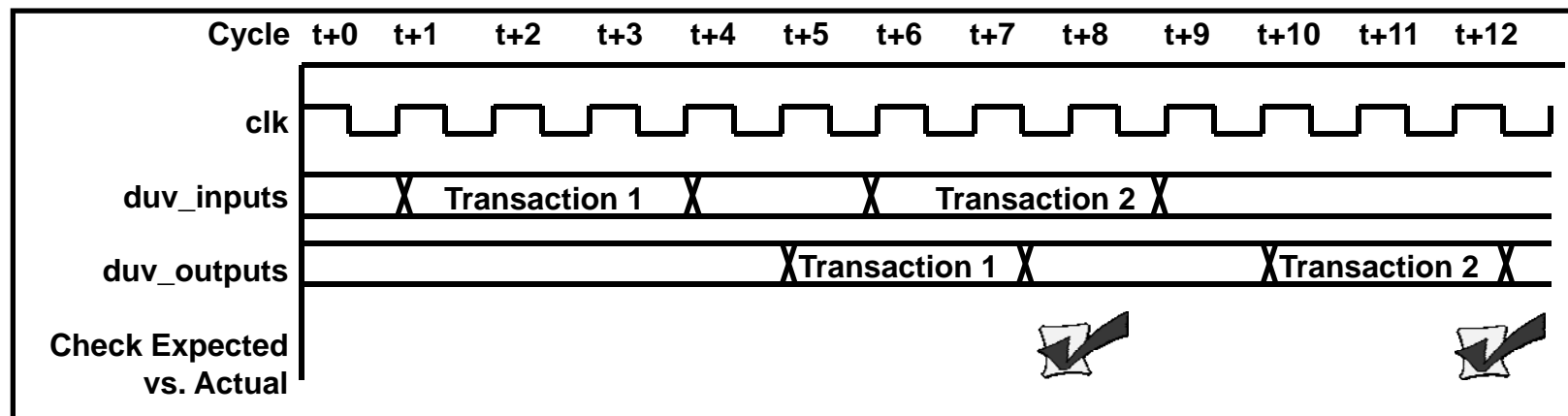
# On-the-fly Checking

- **Advantages**
  - Detection can be as close as possible (in time and space) to the bug source
  - Can stop test as soon as bug occurs; no wasted simulation cycles
  - Do not require large traces and external tools to do the checking

- **Disadvantages**
  - May **slow down simulation**
  - Checking is limited to allowed time and space complexity
  - Need to plan the checking in advance
    - To add a new checker, we need to rerun simulation

# End-of-test Checking

- Checking is done **at the end of simulation**
- The checker checks the state of internal and external resources and makes sure that they are correct
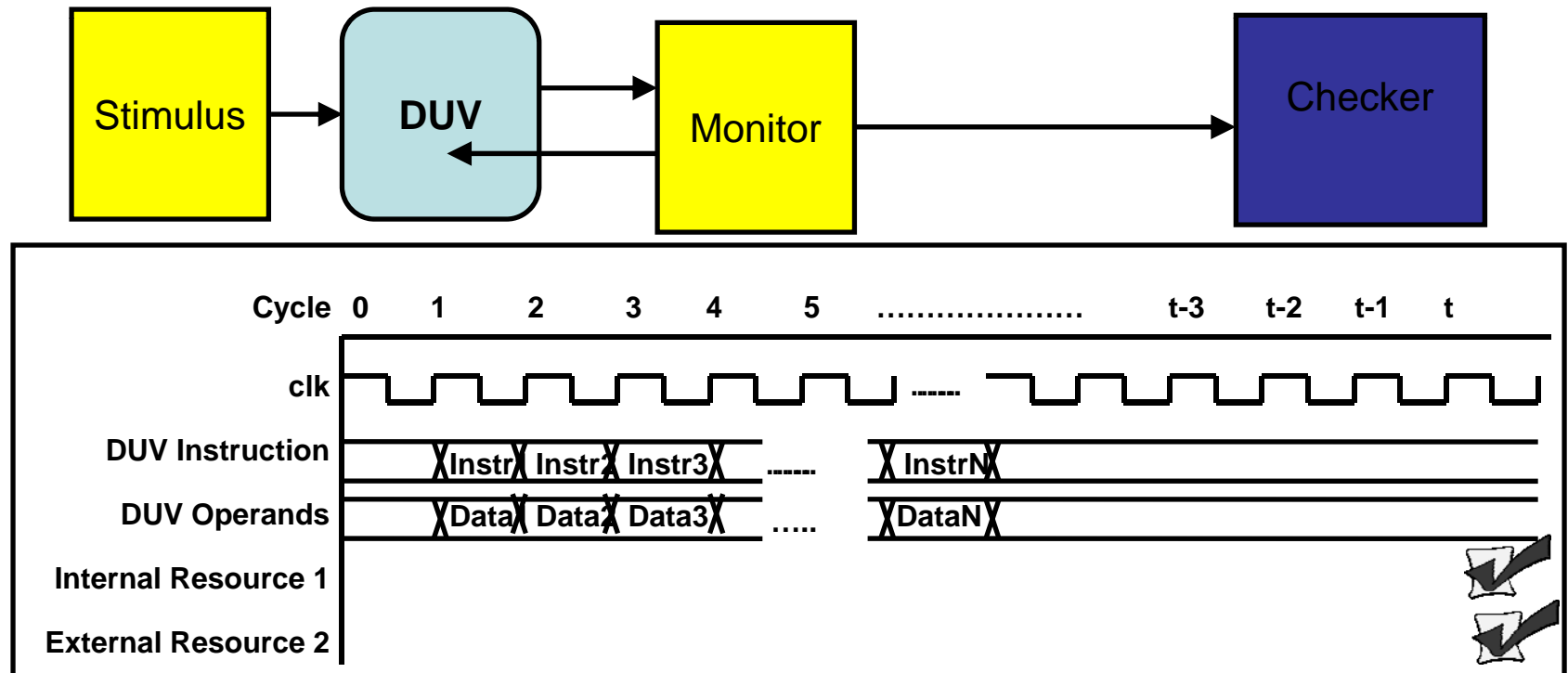
# End-of-test Checking

- **Disadvantages**
  - Provides limited checking capabilities
    - Static look at the state of resources at the end of the test
  - High probability of masking bugs by rewriting to the resources
  - Hard to detect performance bugs
    - Correct things are happening, but not at the right time
  - Hard to correlate symptoms to bugs
    - Hard to debug
- **Advantages**
  - Simpler than other forms
    - May not require a deep understanding of the DUV
  - Reduces probability of false alarms
    - Caused by disappearing bad effects

# External Checking (Monitors)

- **Monitors** keep internal resources values and behaviors in trace files
- Checking is done by an external program that examines these files
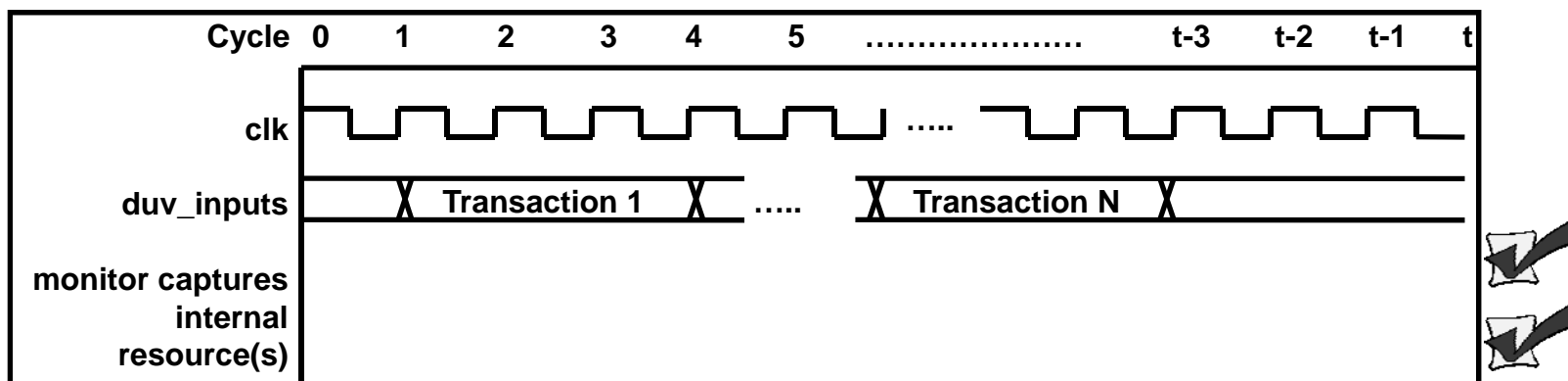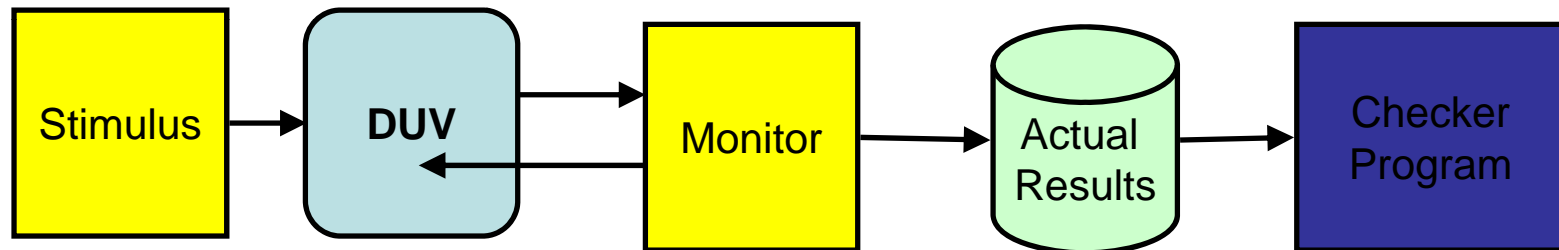
# External Checking

- **External checking separates the checking from the simulation**
  - We can perform any check we want without rerunning the simulation
    - As long as the data is in the trace files
  - We can perform more complicated checks
    - Use longer history, process events out-of-order
  - We can combine information coming from different sources
    - For example, different verification environments

In theory, external checking has all the powers of on-the-fly checking plus end-of-test checking - plus more

(Trace size and amount of traced facilities is a practical limitation.)

# What to Check

- **There are five main sources of checkers**
  - The inputs and outputs of the design (specification)
  - The architecture of the design
  - The microarchitecture of the design
  - The implementation of the design
  - The context of the design

- **Note that the source of checkers and their implementation are two different issues**

# Coarser Classification – The What And the How
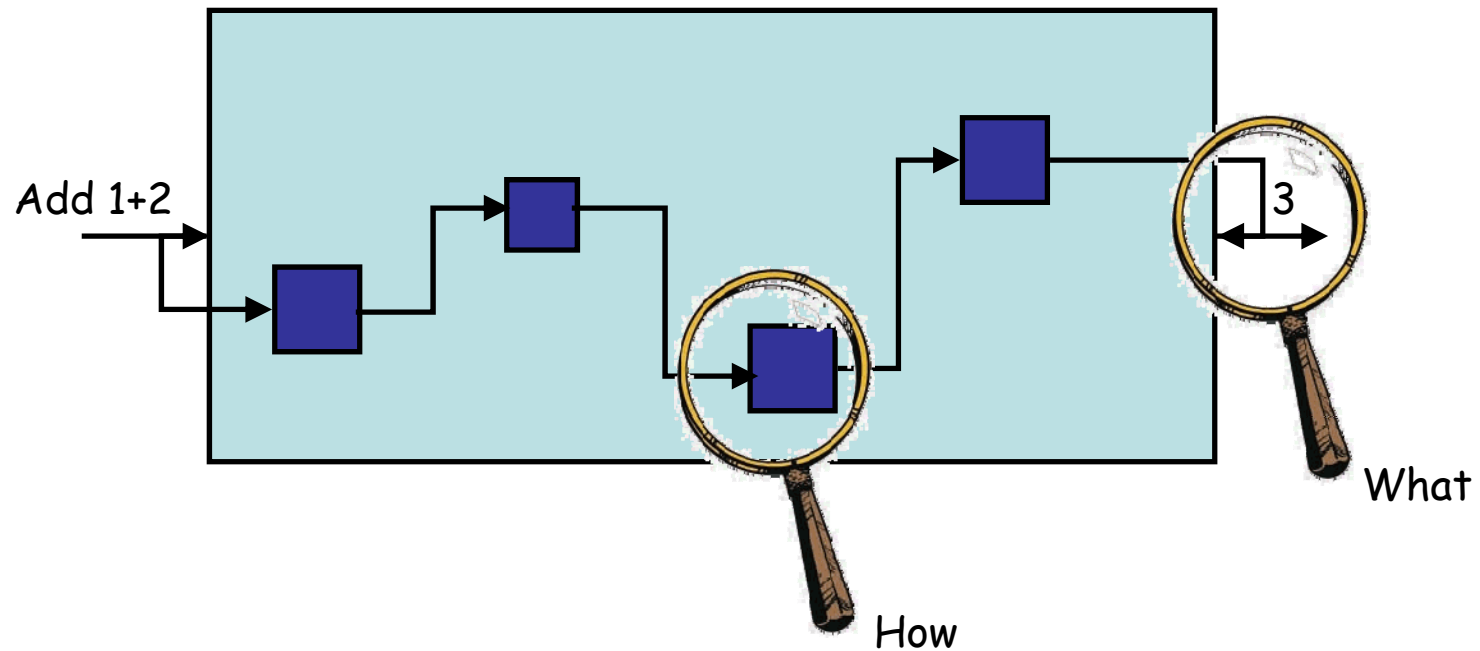
# Checking the What

- **Check the final outcome of a behavior**
  - Data oriented
    - But not limited to data
  - Usually based on higher level of abstraction
    - Checking is less tight
    - Requires less familiarity with the DUV
    - Less false alarms, more misdetections
  - Low correlation between failure and bugs
    - Harder for debugging
    - Can find "unexpected" bugs

# Checking the How

- **Check *how* things are done internally**
  - Control oriented
  - Usually at lower levels of abstraction
    - Closer to implementation
  - More false alarms, less misdetections
  - Tighter relations between failure and bugs

# Checking Technologies

# Stimuli Generation and Checking

- In general, checking should be isolated from the stimuli generation
  - **Modularity** – ability to replace the stimuli generator
  - **Reusability** – ability to use the checkers at higher level of the design hierarchy
- Exceptions
  - Self-checking tests
  - Golden vectors

- The stimuli generation can assist checking by improving observability
  - Help transfer events from dark corners to the spotlight

# Scoreboards

- Scoreboards are smart data structures that keep track of events in the DUV during simulation

- Usually, scoreboards are global
  - One scoreboard per verification environment

- Scoreboard are not checking mechanisms, but
  - The main purpose of using scoreboards is for checking
  - In practice, many checkers are implemented inside scoreboards
  - There are many typical checks that are done with scoreboards

# Scoreboard Operation
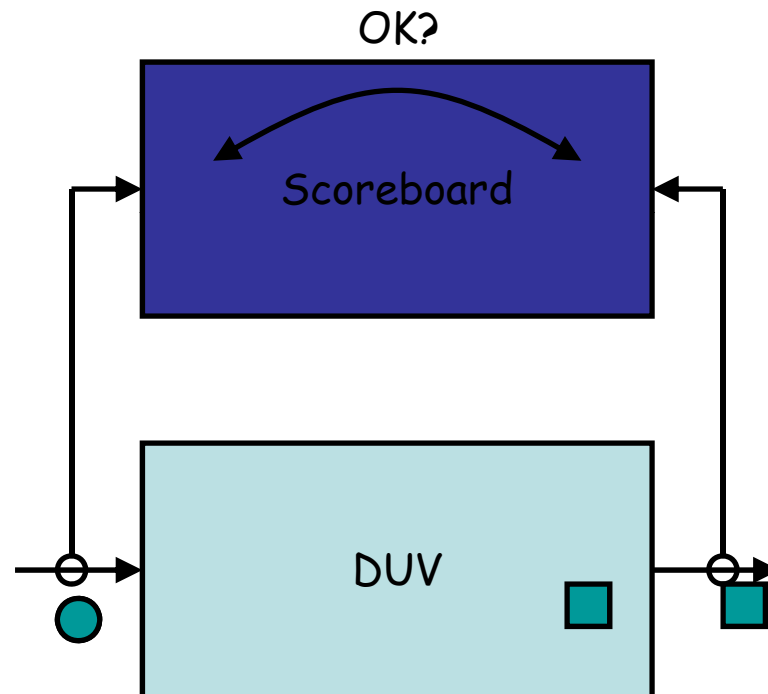
# Scoreboards

- **Sources of information to the scoreboard**
  - Primarily, the inputs and outputs of the DUV
  - Internal events can also be used

- **Types of checks done with scoreboard**
  - Matching between inputs and outputs
    - Nothing is lost
      - Input with no matching output
    - Nothing is born
      - Output with no matching inputs
    - Data matching
  - Timing rules
    - Delay from input to output is within limits
  - Ordering rules

- **Scoreboards are very useful in data flow designs**
  - Routers and calc1

# Scoreboarding in e - 1

- Assume: DUV does not change order of packets.
  - Hence, first packet on scoreboard has to match received packet.

```
import packet_s;
unit scoreboard {
   !expected_packets : list of packet_s;
   add_packet(p_in : packet_s) is {
      expected_packets.add(p_in);
   };

   check_packet(p_out : packet_s) is {
      var diff : list of string;
       -- Compare physical fields of first packet on scb with p_out.
       -- Report up to 10 differences.
      diff = deep_compare_physical(expected_packets[0], p_out, 10);
                              check that (diff.is_empty())
             else dut_error(``Packet not found on scoreboard'',
   diff);
      -- If match was successful, continue.
      out(``Found received packet on scoreboard.'');
      expected_packets.delete(0);
   };
};
```

# Scoreboarding in e - 2

Recording a packet on the scoreboard:

Extend driver such that

– When packet is driven into DUV call `add_packet` method of scoreboard.

■ Current packet is copied to scoreboard.

– It is useful to define an event that indicates when packet is being driven.

Checking for a packet on the scoreboard:

Extend receiver such that

– When a packet was received from DUV call `check_packet`.

■ Try to find the matching packet on scoreboard.

– It is useful to define an event that indicates when a packet is being received.

# Side Note – Graceful End-of-test

- **Checking that nothing is lost is very important**
- If an input does not have a matching output, how can we distinguish between two cases
  - The input is lost or hopelessly stuck in the DUV
  - The DUV did not have enough time to handle the input
- Possible solution – Start a timer when a new input enters the DUV
  - If the timer expires, that input is lost or stuck
  - But, what if the delay cannot be bound?
- Alternative (or complementary) solution – stop the inputs before the end of the test and let the design clean itself
  - Because there are no new inputs, things that are stuck inside have a chance to get free

# Reference Models

- **A reference model is an oracle that tells how the DUV should behave**
  - Usually in the form of an alternative implementation

- **It runs in parallel to the DUV, using the same inputs and provides the checking mechanisms with information about the expected behavior**
  - Checking is done by comparing the expected behavior to the actual one

- **Pure reference models can run independently of the DUV**
  - But not all reference models are pure (example later)

# Reference Model Operation
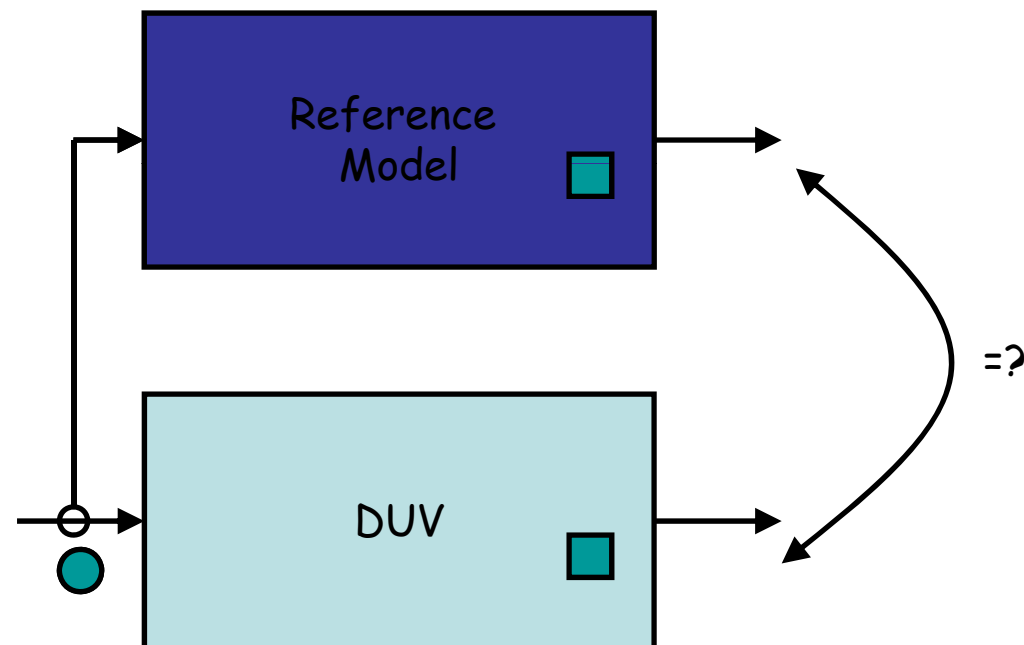
# Reference Models

- **Reference models have many uses**
  - Checking
  - Aids for stimuli generation
  - "Smart" BFM – imitate the function of the DUV
  - Vehicles for SW development

- What can we check with a reference model
  - In principal, anything
  - In practice it depends on the level of details and accuracy of the reference model
    - And how much of its behavior we are willing to expose

# Levels of Abstraction

- **The level of abstraction in a reference model dictates the type of information we can get out of it for checking**
  - Functionally accurate models can be used only to check correctness of data, usually at the end of the test or at well defined points in time
    - Timing, order, and other checks need other means
  - Cycle accurate models can be used for checking all aspects of I/O behavior
  - Cycle accurate and latch accurate models can be used also for checking the internal state of the DUV
    - The book calls this type of model deep function reference model

# Impure Reference Model

- Sometimes it is impossible (or very hard) for the reference model to duplicate significant decisions made by the DUV

- Possible solution:
  Use information from the DUV to assist the reference model!
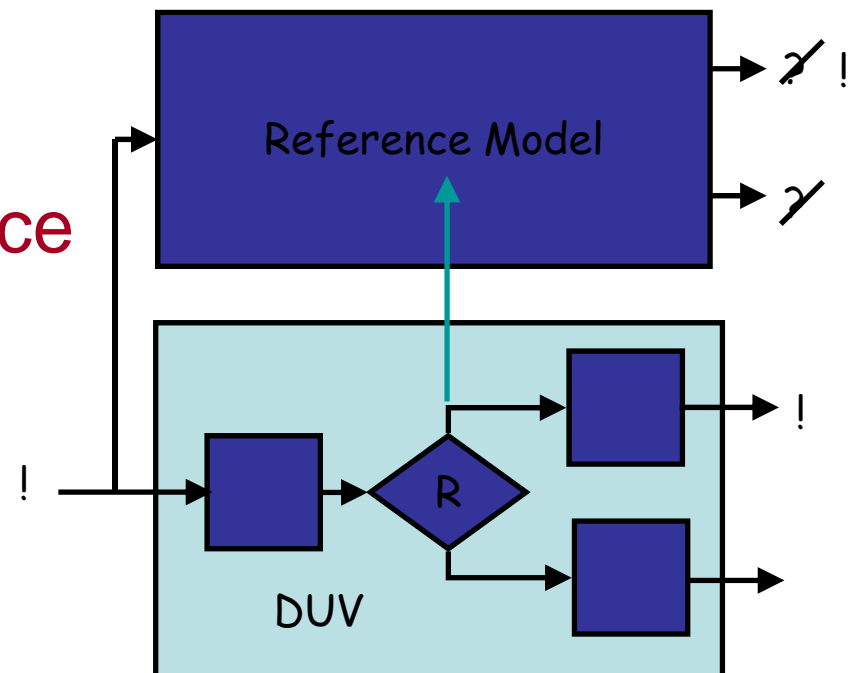
# Rule-based Checking

- **Checks that a set of rules hold in the DUV**
- **Essentially, all checking is rule-based**

if (not something) then error

- **Something can be**
  - Value of a register matches value in reference model
  - Data in a packet at the DUV output matches data in the input as stored in the scoreboard
  - response_out == 0 $\rightarrow$ data_out == 0
- **Rule-based checking usually refers to the last case**

# Rule-based Checking

- Rules can come from many sources
  - All levels of the design process
    - Spec, high-level design, implementation
  - Behavior of neighboring units
- Rules checking can be implemented in many places
  - External checking tools
  - Various places in the verification environment
    - Interface monitors
    - Scoreboards
    - End-of-test checkers
  - In the DUV itself
- Rule-based checking that is embedded in the DUV code is called assertions
- Lecture on Assertion-Based Verification

# Putting Coverage, Generation and Checking together:

# **The Verification Environment**

(With many thanks to Cadence for providing the animations in this section.)

# Traditional Approach: Directed Testing

Verification engineer sets goals and writes directed test for each item in the
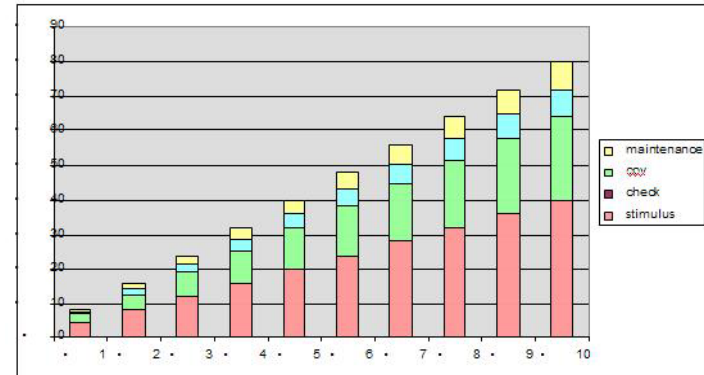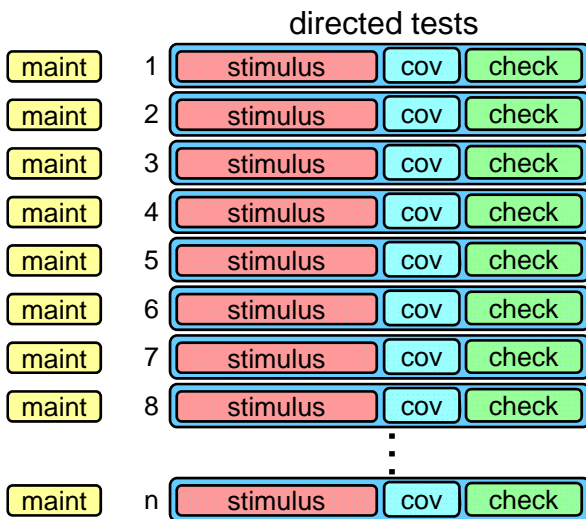Verification Plan:
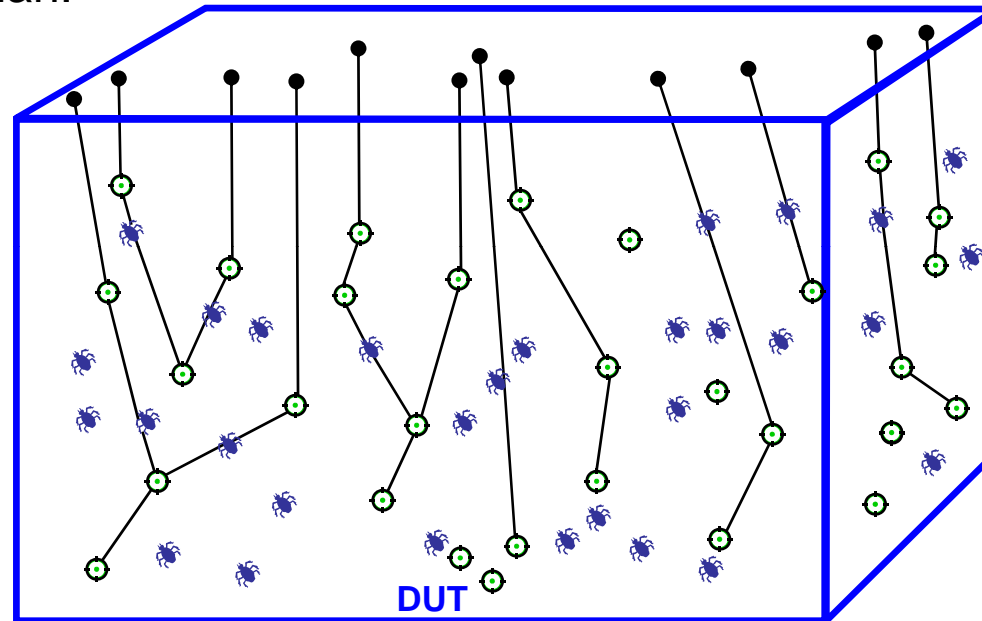


Redo if design changes

Automation ➤ Significant manual effort to write all the tests

Automation ➤ Work required to verify each goal was reached

Completeness ➤ Poor coverage of non-goal scenarios
… especially the cases that you didn't "think of"

# Directed Test Environment

- Composition of a directed test
  - Directed tests contain more than just stimulus.
  - Checks are embedded into the tests to verify correct behavior.
  - The passing of each test is the indicator that a functionality has been exercised.
- Reusability and maintenance
  - Tests can become quite complex and difficult to understand the intent of what functionality is being verified
  - Since the checking is distributed throughout the test suite, it is a lot of maintenance to keep checks updated
  - It is usually difficult or impossible to reuse the tests across projects or from module to system level
- The more tests you have the more effort is required to develop and maintain them
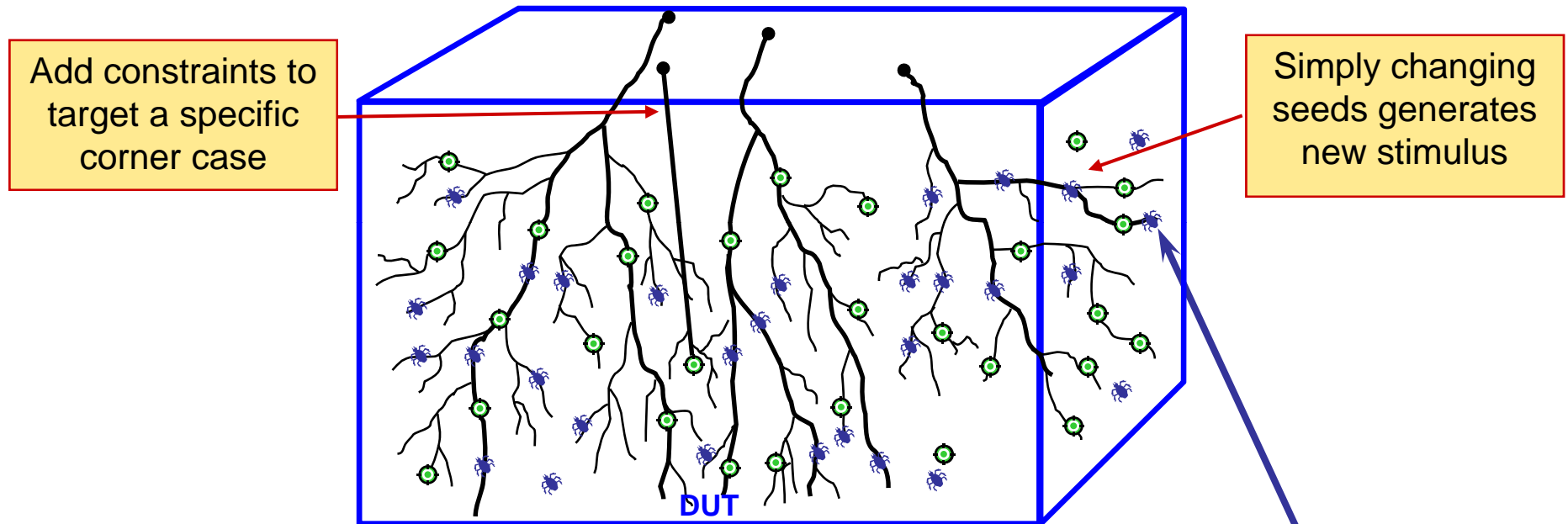


Directed test approach
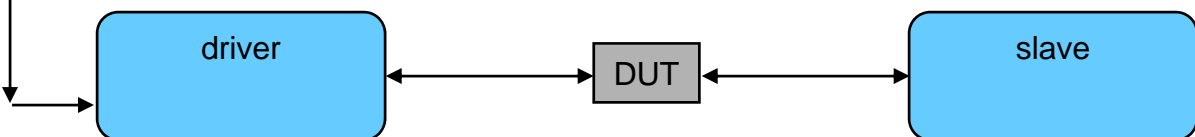
# Coverage Driven Verification Methodology:
## *Defining Coverage "Goals" Enables Automation*

Focuses on reaching **goal areas** (*versus execution of test lists*):

Add constraints to target a specific corner case

Simply changing seeds generates new stimulus

**DUT**

Constrained-random stimulus **generation** explores goal areas (& beyond)

**Coverage** shows which *goals* have been exercised and which need attention

(Self-Checking ensures proper DUT response.)    *Even for non-goal states!*

**Automation –** Constrained-random stimulus accelerates hitting coverage goals and exposing bugs.  Coverage and checking results indicate effectiveness of each simulation, which enables scaling many parallel runs.

# Coverage Driven Environment

- Composition of a coverage driven environment
  - Reusable stimulus sequences developed with "constrained random" generation
  - Running unique seeds allows the environment to exercise different functionality
  - Monitors independently watch the environment
  - Independent checks ensure correct behavior.
  - Independent coverage points indicate which functionality has been exercised.
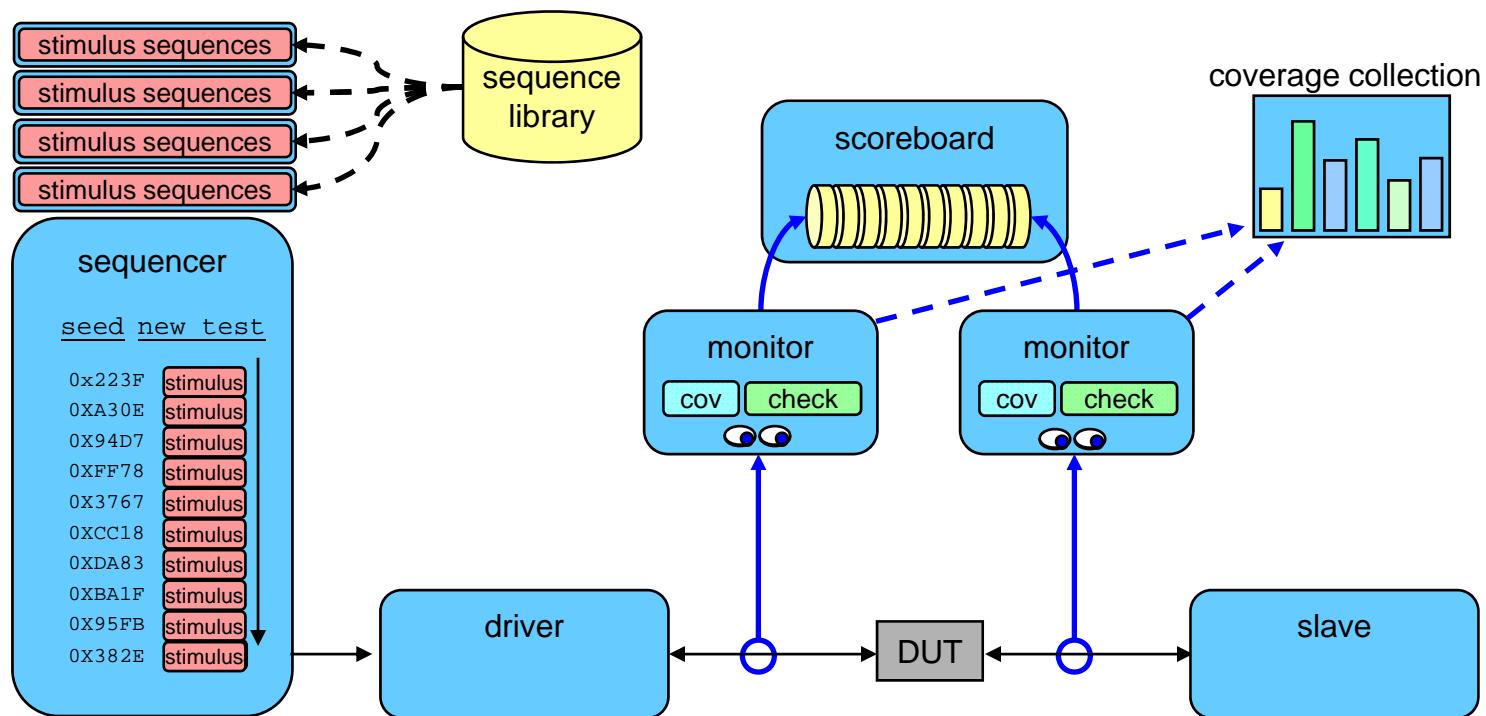
# Summary
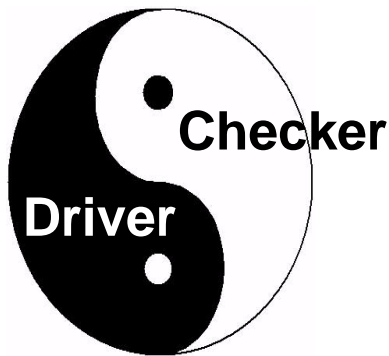


- **Stimuli Generation**
- Coverage and
- **Checking**

## Coverage Driven Verification Methodology

- Next: ABV