Test and Verification Solutions

Getting you to market sooner by providing easy access to outsourcing solutions

Introduction to Design Verification

Kerstin Eder, TVS
Introduction: Increasing Design Complexity

Multiple Power Domains, Security, Virtualisation
Nearly five million lines of code to enable Media gateway
Introduction: Shorter Time-to-market windows

- Shipment windows: 90s
- Shipment windows: early 2000s
- Shipment windows: today

Volume

Design

Time

Confidence

95%+

Desired

Actual

Risks
Quality
Predictability
Productivity

Time

Scheduled tapeout

Final tapeout
Introduction: Verification vs Validation

- **Verification:**
  - confirms that a system has a given input / output behaviour, sometimes called the transfer function of a system.

- **Validation:**
  - confirms that a system has a given behaviour, i.e.
  - validation confirms that the system’s transfer functions results in the intended system behaviour when the system is employed in its target environment, e.g. as a component of an embedded system.
“Design Verification is the process used to demonstrate the correctness of a design w.r.t. the requirements and specification.”

Types of verification:
- Functional verification
- Timing verification
- ...
- What about performance?
Conceptual representation of the verification process

- Important question: What are you verifying?

- Choosing a common origin and reconvergence point determines what is being verified and what type of method is best to use.

- The specification of a FIFO (.doc)
- The implementation of a FIFO (.v)

- Transformation
- Verification

- Higher Abstraction Level
- Lower Abstraction Level
In practice, the specification is often a document written in a natural language by individuals of varying degrees of ability to communicate.

An individual (or group of individuals) must read and interpret the specification and transform it into the design.

Human introduced errors are introduced by (mis)interpretation.

DANGER: When a designer verifies her/his own design – then she/he is verifying her/his own interpretation of the design – not the specification!
Introduction: Verification Independence

- Verification should be kept **independent** from Design
  - Verification engineers refer to the specification in disputes with the design team
  - Designers and Verification engineers both interpret the specification

Verification relies on both not making the same interpretation mistake!
Introduction: Functional Verification Approaches

- **Verification**
  - **Static**
    - Reviews
    - Code Analysis
      - Linters
    - Equivalence Checking
  - Formal
    - Property Checking
  - Dynamic
    - Simulation
    - Prototyping
      - Silicon
      - FPGA
    - Dynamic Formal
      - Theorem Proving
Introduction: Observability and Controllability

- **Observability**: Indicates the ease at which the verification engineer can identify when the design acts appropriately versus when it demonstrates incorrect behavior.

- **Controllability**: Indicates the ease at which the verification engineer creates the specific scenarios that are of interest.
Introduction: Levels of Observability

- Black Box

- White Box

- Grey Box
Introduction: Formal Property Checking

Properties of a design are formally proven or disproved.

- Used to check for generic problems or violations of user-defined properties of the behaviour of the design.
- Usually employed at higher levels of abstractions.
- Properties are derived from the specification.
- Properties are expressed as formulae in some (temporal) logic.
- Checking is typically performed on a Finite State Machine model of the design.
  - This model needs to be derived from the RTL.

```plaintext
under env_constraint if condition then expectation
```

Property checking can also be performed on higher levels of abstraction.
Introduction: Role of Simulation

- Most widely used verification technique in practice
- Complexity of designs makes exhaustive simulation impossible in terms of cost/time
  - Engineers need to be selective
  - Employ state of the art coverage-driven verification methods
  - Test generation challenge
- Simulation can drive a design deep into its state space
  - Can find bugs buried deep inside the logic of the design
- Understand the limits of simulation:
  - Simulation can only show the presence of bugs but can never prove their absence!
Introduction: Simulation vs Functional Formal Verification

In practice, capacity limits and completeness issues restrict formal verification to selected parts of the design.

Naïve interpretation of exhaustive formal verification: Verify ALL properties.

Only selected parts of the design can be covered during simulation.

In practice, capacity limits and completeness issues restrict formal verification to selected parts of the design.

Introduction: How big is Exhaustive?

- Consider simulating a typical CPU design
  - 500k gates, 20k DFFs, 500 inputs
  - 70 billion sim cycles, running on 200 linux boxes for a week
  - How big: $2^{36}$ cycles

- Consider formally verifying this design
  - Input sequences: cycles $2^{\text{inputs+state}} = 2^{20500}$
  - What about X’s: $2^{15000}$ (5,000 X-assignments + 10,000 non-reset DFFs)
  - How big: $2^{20500}$ cycles ($2^{15000}$ combinations of X is not significant here!)

- That’s a big number!
  - Cycles to simulate the 500k design: $2^{36}$ (70 billion)
  - Cycles to formally verify a 32-bit adder: $2^{64}$ (18 billion billion)
  - Number of stars in universe:
    - $2^{70}$ ($10^{21}$)
  - Number of atoms in the universe:
    - $2^{260}$ ($10^{78}$)
  - Possible X combinations in 500k design:
    - $2^{15000}$ ($10^{4515} \times 3$)
  - Cycles to formally verify the 500k design:
    - $2^{20500}$ ($10^{6171}$)
Coverage Types

- **Code** coverage
- **Structural** coverage
- **Functional** coverage

- **Other classifications**
  - Implicit vs. explicit
  - Specification vs. implementation
Code Coverage - Basics

- Coverage models are based on the HDL code
  - Implicit, implementation coverage
- Coverage models are syntactic
  - Model definition is based on syntax and structure of the HDL
- Generic models – fit (almost) any programming language
  - Used in both software and hardware design
Implicit coverage models that are based on common structures in the code
- FSMs, Queues, Pipelines, …

State-machines are the essence of RTL design

**FSM coverage models** are the most commonly used structural coverage models

Types of FSM coverage models
- State
- Transition (or arc)
- Path
Coverage questions not answered by code coverage tools
- Did every instruction take every exception?
- Did two instructions access the register at the same time?
- How many times did cache miss take more than 10 cycles?
- Does the implementation cover the functionality specified?
- ...(and many more)

Code coverage indicates how thoroughly the test suite exercises the source code!
- Can be used to identify outstanding corner cases

Code coverage lets you know if you are not done!
- It does not indicate anything about the functional correctness of the code!

100% code coverage does not mean very much. 😒

Need another form of coverage!
Functional Coverage

- It is important to cover the **functionality** of the DUV.
  - Most functional requirements can’t easily be mapped into lines of code!

- **Functional coverage models** are designed to assure that various aspects of the functionality of the design are verified properly, they link the requirements/specification with the implementation.

- Functional coverage models are specific to a given design or family of designs.

- Models may cover
  - The design in terms of inputs and the outputs
  - The design’s internal states or micro-architectural features
  - Protocols
  - Specific scenarios from the verification plan
  - Combinatorial or sequential features of the design
Types of Functional Coverage

- **Discrete set of functional coverage tasks**
  - Set of unrelated or loosely related coverage tasks often derived from the requirements/specification
  - Often used for corner cases
    - Driving data when a FIFO is full
    - Reading from an empty FIFO
  - In many cases, there is a close link between functional coverage tasks and **assertions**

- **Structured functional coverage models**
  - The coverage tasks are defined in a structure that defines relations between the coverage task
  - **Cross-product** (Cartesian Product) most commonly supported
A cross-product coverage model is composed of the following parts:

1. A semantic **description** of the model (story)
2. A list of the **attributes** mentioned in the story
3. A set of all the **possible values** for each attribute (the attribute value **domains**)
4. A list of **restrictions** on the legal combinations in the cross-product of attribute values
## Combining Coverage Metrics

- **Do we need both code and functional coverage? YES!**

<table>
<thead>
<tr>
<th>Functional Coverage</th>
<th>Code Coverage</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Low</td>
<td>There is verification work to do.</td>
</tr>
<tr>
<td>Low</td>
<td>High</td>
<td>Multi-cycle scenarios, corner cases, cross-correlations still to be covered.</td>
</tr>
<tr>
<td>High</td>
<td>Low</td>
<td>Verification plan and/or functional coverage metrics inadequate. Check for “dead” code.</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
<td>Increased confidence in quality.</td>
</tr>
</tbody>
</table>

- **Coverage models complement each other.**
  - Mutation testing adds value in terms of test suite qualification.
- **No single coverage model is complete on its own.**
Case Study: Constrained pseudo random test generation

- Include
  - Advanced FIFO TB architecture
  - Self-checking
  - Scoreboards
  - Monitors
- Design size reduction: reduce size of FIFO to 2 entries
  - demonstrate fewer tests needed to get coverage
  - Same corner cases
  - Directed tests no longer work – promote parametrization in TB
HW assertions:

- **combinatorial** (i.e. “zero-time”) conditions that ensure functional correctness
  - must be valid at all times
    - “This buffer never overflows.”
    - “This register always holds a single-digit value.”

and

- **temporal conditions**
  - to verify sequential functional behaviour over a period of time
    - “The grant signal must be asserted for a single clock cycle.”
    - “A request must always be followed by a grant or an abort within 5 clock cycles.”
  - Need temporal assertion specification language!
    - System Verilog Assertions
    - PSL/Sugar
Property Types: Safety

- **Safety**: Nothing bad ever happens
  - The FIFO *never* overflows.
  - The system *never* allows more than one process to use a shared device simultaneously.
  - Requests are *always* answered within 5 cycles.

These properties can be falsified by a finite simulation run.
Property Types: Liveness

- **Liveness:** Something good will eventually happen
  - The system *eventually* terminates.
  - Every request is *eventually* answered.

In theory, liveness properties can only be falsified by an infinite simulation run.
  - Practically, we can assume that the “graceful end-of-test” represents infinite time.
    - If the good thing did not happen after this period, we assume that it will never happen, and thus the property is falsified.
Remember, simulation can only show the presence of bugs, but never prove their absence!

An assertion has never fired - what does this mean?
- Does not necessarily mean that it can never be violated!
- Unless simulation is exhaustive..., which in practice it never will be.
- It might not have fired because it was never evaluated.

Assertion coverage: Measures how often an assertion condition has been evaluated.
Completion Criteria

Common criteria for completion are:

- **Coverage targets**
  - (coverage closure)
- **Target metrics**
  - bug rate drop
- **Resolution of open issues**
- **Review**
- **Regression results**

- **Coverage Complete?**
  - Yes
  - No

- **Bug Rate Dropped**
  - Yes
  - No

- **No Open Issues**
  - Yes
  - No

- **Review**
  - Yes
  - No

- **Clean Regression**
  - Yes

**Ready to ship**
Test and Verification Solutions

Getting you to market sooner by providing easy access to outsourcing solutions

Example DUV

Kerstin Eder, TVS
Example DUV Specification

- **Inputs:**
  - wr indicates valid data is driven on the data_in bus
  - data_in is the data to be pushed into the DUV
  - rd pops the next data item from the DUV in the next cycle
  - clear resets the DUV
Example DUV Specification

**Outputs:**
- `data_out_valid` indicates that valid data is driven on the `data_out` bus
- `data_out` is the data item requested from the DUV
- `empty` indicates that the DUV is empty
- `full` indicates that the DUV is full
Example DUV Specification

- **Black Box Verification**
  - The design is a FIFO.
  - Reading and writing can be done in the same cycle.
  - Data becomes valid for reading one cycle after it is written.
  - No data is returned for a read when the DUV is empty.
  - Clearing takes one cycle.
  - During clearing read and write are disabled.
  - Inputs arriving during a clear are ignored.
  - Data written to a full DUV will be dropped.
  - The FIFO is 8 entries deep.
Properties of the DUV

Black box view:

- (P) empty and full are never asserted together.
- (P) After clear the FIFO is empty.
- (D) After writing 8 data items the FIFO is full.
- (D) Data items are moving through the FIFO unchanged in terms of data content and in terms of data order.
- (D) No data is duplicated.
- (D) No data is lost.

Assertions:

- Distinguish between protocol properties and design properties/coverage.
- Protocol coverage is more easily re-usable.
Basic techniques to combat Complexity

- **Collapse the size of the FIFO to only 2 (or 4) entries and demonstrate**
  - Impact on verification effort
  - Parametrization of design and tb is important

- **Reduce the width of the FIFO’s data path**
  - Show this reduces complexity of data coverage
  - Show this helps Formal Verification
The mechanics of an advanced test bench

- Test
- Functional Coverage
- Checker
- Monitor
- Assertions
- Code Coverage

Stimulus generator:
- constraint
- addr
- data

Driver:
- active
- passive

Design Under Test:
- assert

Coverage:
- assertions

Functional Coverage:
- coverage

Monitor:
- coverage

The diagram illustrates the interaction between various components in an advanced test bench, including the test phase, stimulus generator, driver, monitor, assertions, and code coverage.
Some hardware verification examples

CPU Verification

Instruction Stream Generator → assembler → CPU RTL → Compare → CPU C Model → Accuracy?
Some hardware verification examples

USB Verification

- **Packet Generator**
- **Driver**
- **DUT**
- **Response**
- **Scoreboard**

- **Assertions**
- **Coverage**
Bubble Sort “Proof of Concept” for SW Testing

• **Program Specification**
  – Input lists of integers, floats, ascii, etc.
  – Reject lists of mixed types
  – Convert unsorted lists to sorted lists

• **Can we test the program with constrained input generation?**
  – Generate valid and invalid inputs
  – Direct generation towards corner cases
  – Check outputs for correctness
    • Without re-writing an identical checker program
  – Measure what we have tested
Results of Bubble Sort “Proof of Concept”

Lists of
• Integers
• Floats
• Ascii
• etc

List Generator

Checkers

Software Under Test

Coverage Metrics

Lists

Constrain towards
• Empty lists
• Equal values
• Reverse ordering

• Check output list is ordered
• Output list contents == input list contents

• Empty List
• Reverse ordered
• Error cases (mix integers, floats, ascii)
• Etc.
Virtual System Level Test Environment

Event Stream Generator

Sensor Inputs

Software Under Test

Actuator Outputs

Checkers

Logfiles

Metrics
Virtual System Level Checkers

- Assert “never do anything wrong”
  - Always fail safe

- Assert “always respond correctly”
  - If A&B&C occur then check X happens
    - Assertion coverage “check A&B&C occurs” for free

- Analyse log files
  - Look for anomalies
    - Did the actuator outputs occur in the correct order
Functional Coverage

Requirements coverage

“Cross-product” coverage

A cross-product coverage model is composed of the following parts:
1. A semantic **description** of the model (story)
2. A list of the **attributes** mentioned in the story
3. A set of all the **possible values** for each attribute (the attribute value **domains**)
4. A list of **restrictions** on the legal combinations in the cross-product of attribute values

A **functional coverage space** is defined as the Cartesian product over the attribute value domains.

Situation coverage

Safety compliance (asureSIGN)

- **Managing Requirements**
  - Importing and editing requirements

- **Decomposing requirements to verification goals**

- **Tracking test execution**
  - Automating import of test results
  - Automate accumulation and aggregation of test results

- **Impact analysis**
  - Managing changes in requirements and tests

- **Demonstrating compliance to DO254 & DO178C**

- **Managing multiple projects**
asureSIGN™ at the heart of HW/SW V&V

- Requirements: Excel, Doors, Jira, etc.
- SystemC Simulation
- Hardware Simulation: Coverage (Cadence), Assertions (Mentor, Aldec, etc.)
- Directed test results
- Formal Verification: OneSpin
- Manual API
- Automated SW Test Tool
- SW Test Tools
- Matlab
- Lab Results
- Requirements Engineering tools

XML API
UCIS API
Run API

Decomposing requirements to features and tests

Import Requirements (Doors, Excel, Word, etc)

Edit Requirements

Map requirements to verification goals

The mapped verification goal

Sign off a requirement with a manual test (e.g. in the lab)
Safety compliance (asureSIGN)

- Managing Requirements
  - Importing and editing requirements

- Decomposing requirements to verification goals

- Tracking test execution
  - Automating import of test results
  - Automate accumulation and aggregation of test results

- Impact analysis
  - Managing changes in requirements and tests

- Demonstrating compliance to DO254 & DO178C

- Managing multiple projects
Tracking test execution: Automating import of test results

- Accellera standard
- asureSIGN
- UCIS
- XML
- API
- Manual Entry
- Manual Testing
- Software Test Tools
- Hardware Verification Results
- Hardware Simulation
- Formal Verification
Automate accumulation and aggregation of test results

Accumulate results over multiple regressions

Record results from each test

Aggregate results through the hierarchy

Define and track against interim milestones (based on % of requirements tested)
Safety compliance (asureSIGN)

- Managing Requirements
  - Importing and editing requirements

- Decomposing requirements to verification goals

- Tracking test execution
  - Automating import of test results
  - Automate accumulation and aggregation of test results

- Impact analysis
  - Managing changes in requirements and tests

- Demonstrating safety compliance – for example
  - DO254/178C, ISO26262, IEC 60601, IEC 61508, EN 50128, IEC 61513

- Managing multiple projects
Demonstrating compliance to DO254 & DO178C

- Export XML for import back into Doors, etc.
- Export PDF report for audit

Select level of detail

Pid = unique reference to requirement in external tool

Export Metadata such as
- Tool version numbers
- Configuration data
- Data owners