Test and Verification Solutions

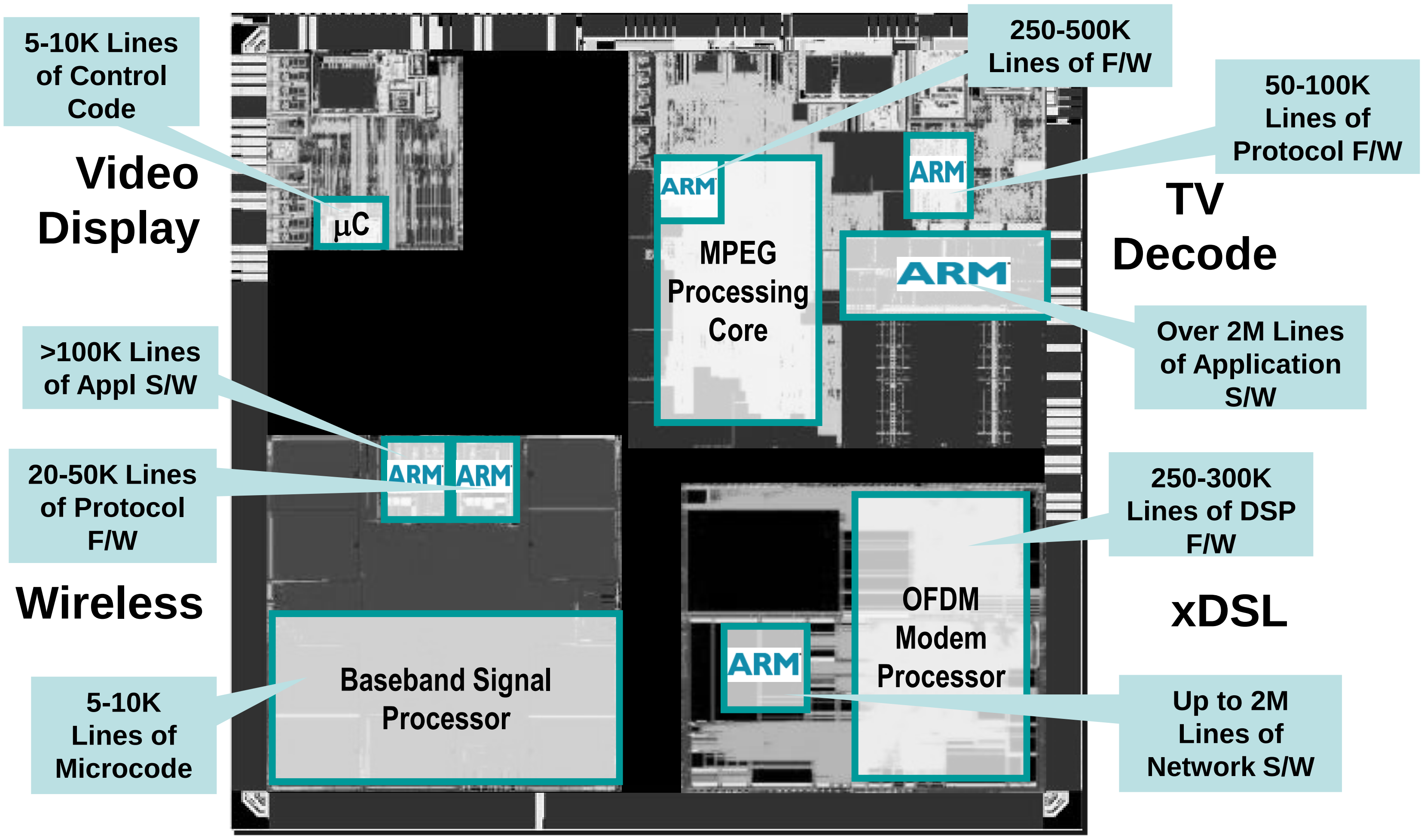
TVS

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

Introduction to Design Verification

Kerstin Eder, TVS

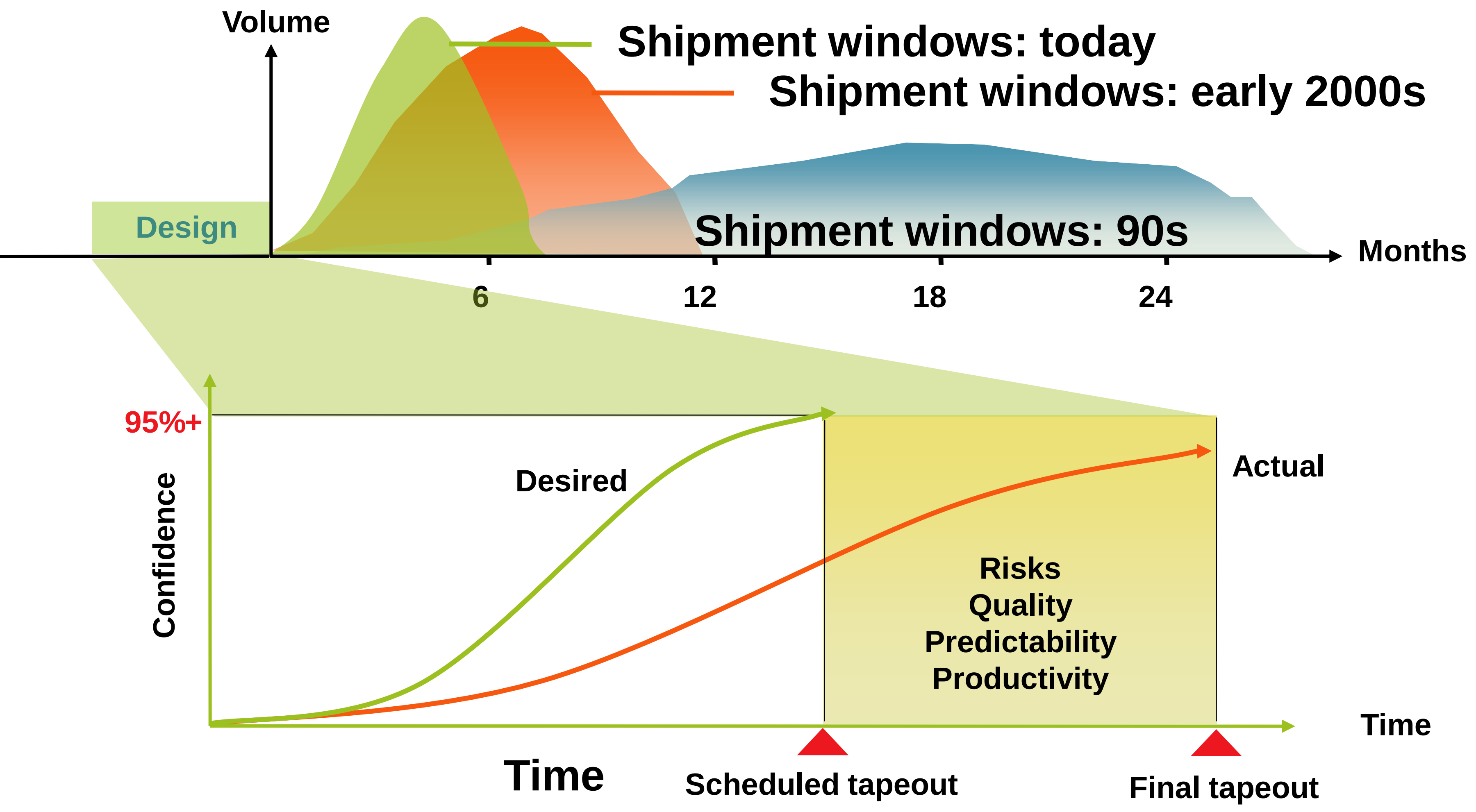
Introduction: Increasing Design Complexity


5-10K Lines of Control Code
Video Display
µC
250-500K Lines of F/W
ARM
MPEG Processing Core
ARM
50-100K Lines of Protocol F/W
TV Decode
ARM
Over 2M Lines of Application S/W
>100K Lines of Appl S/W
ARM
ARM
20-50K Lines of Protocol F/W
Wireless
250-300K Lines of DSP F/W
xDSL
OFDM Modem Processor
ARM
Baseband Signal Processor
5-10K Lines of Microcode
Up to 2M Lines of Network S/W

Multiple Power Domains, Security, Virtualisation
Nearly five million lines of code to enable Media gateway

CONEXANT

Introduction: Shorter Time-to-market windows



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.

Introduction: What is Design Verification?

“DesignVerification is the process used to demonstrate the correctness of a design w.r.t. the requirements and specification.”

Types of verification:

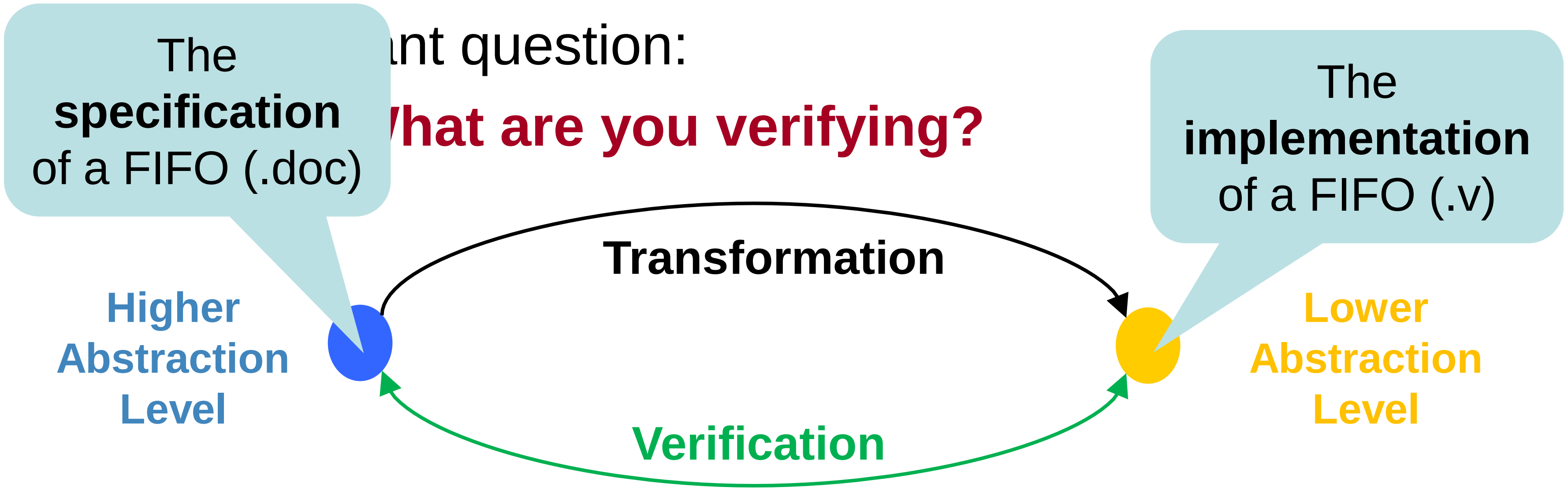
- Functional verification
- Timing verification
- ...
- What about performance?

Introduction: Reconvergence Models [Bergeron]

Conceptual representation of the verification process

nt question:

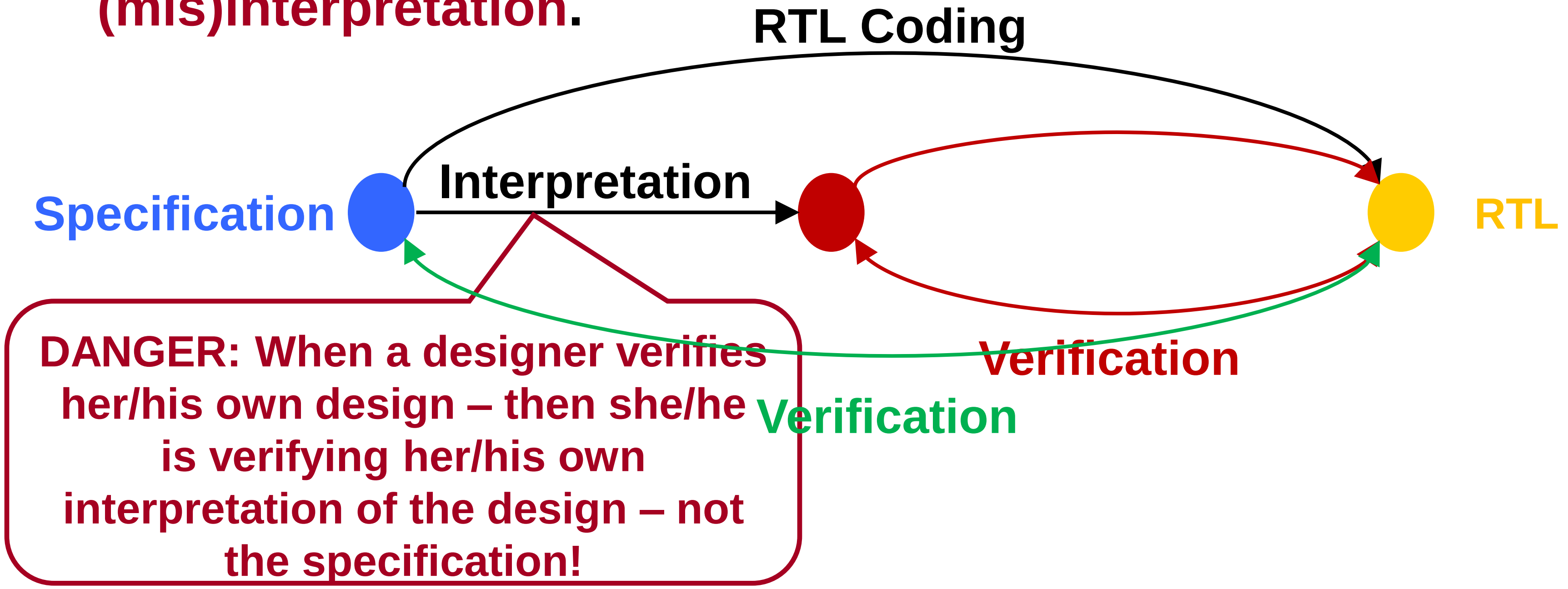
hat are you verifying?


The **specification** of a FIFO (.doc)
The **implementation** of a FIFO (.v)
Transformation
Higher Abstraction Level
Lower Abstraction Level
Verification

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

Introduction: The Human Factor in Verification

- 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.

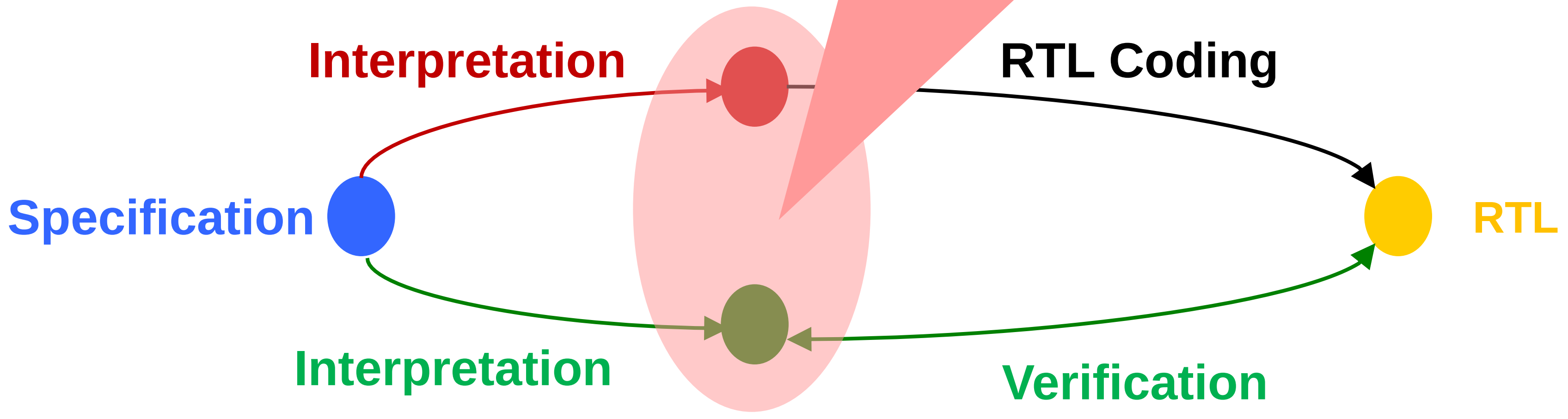


DANGER: When a designer verifies her/his own design – then she/he is verifying her/his own interpretation of the design – not the specification!

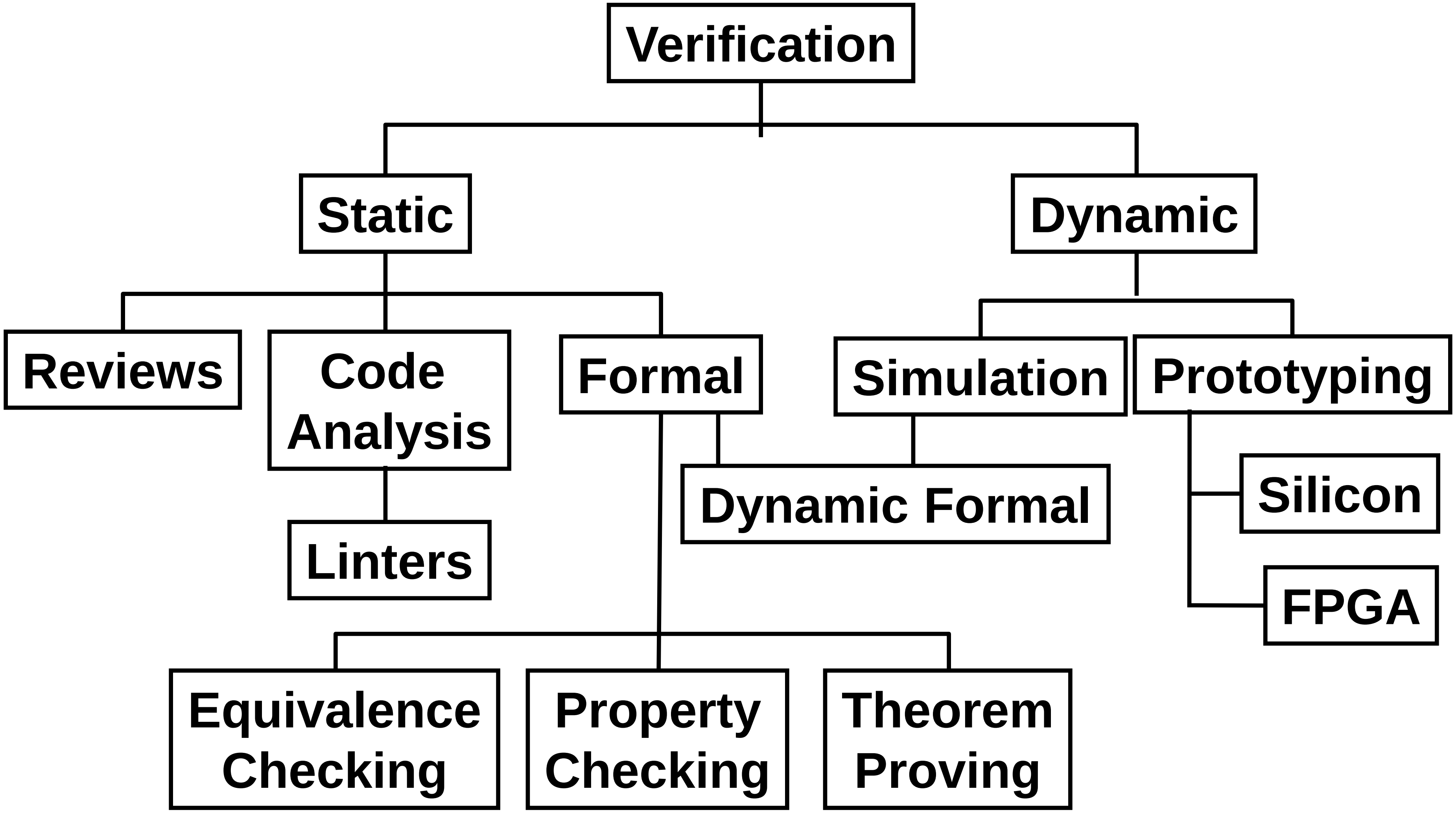
Introduction: Verification Independence

- Verification should be kept **independent** from Design
  - Verification engineers refe
    design team
  - Designers and Verification
    specification

Verification relies on both not making the same interpretation mistake!



Introduction: Functional Verification Approaches



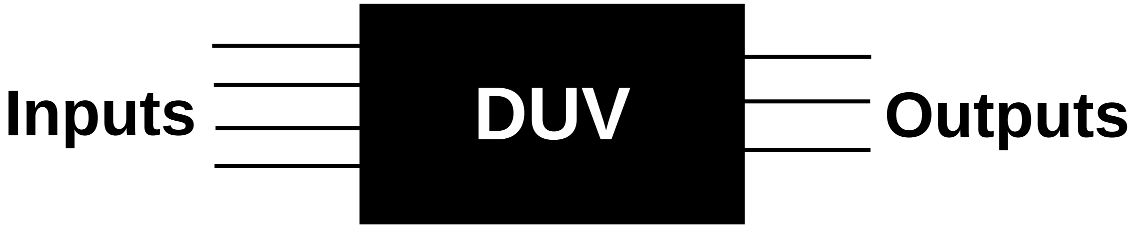
Introduction: Observability and Controllability

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

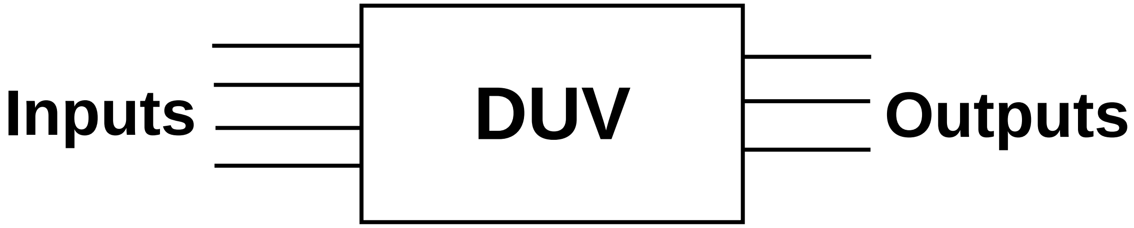
- **Controllability: Indicates the ease at which the verification engineer creates the specific scenarios that are of interest.**

Introduction: Levels of Observability

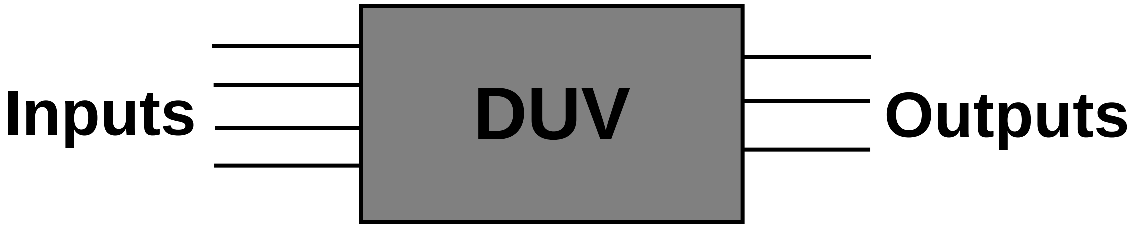
- Black Box



- White Box



- Grey Box



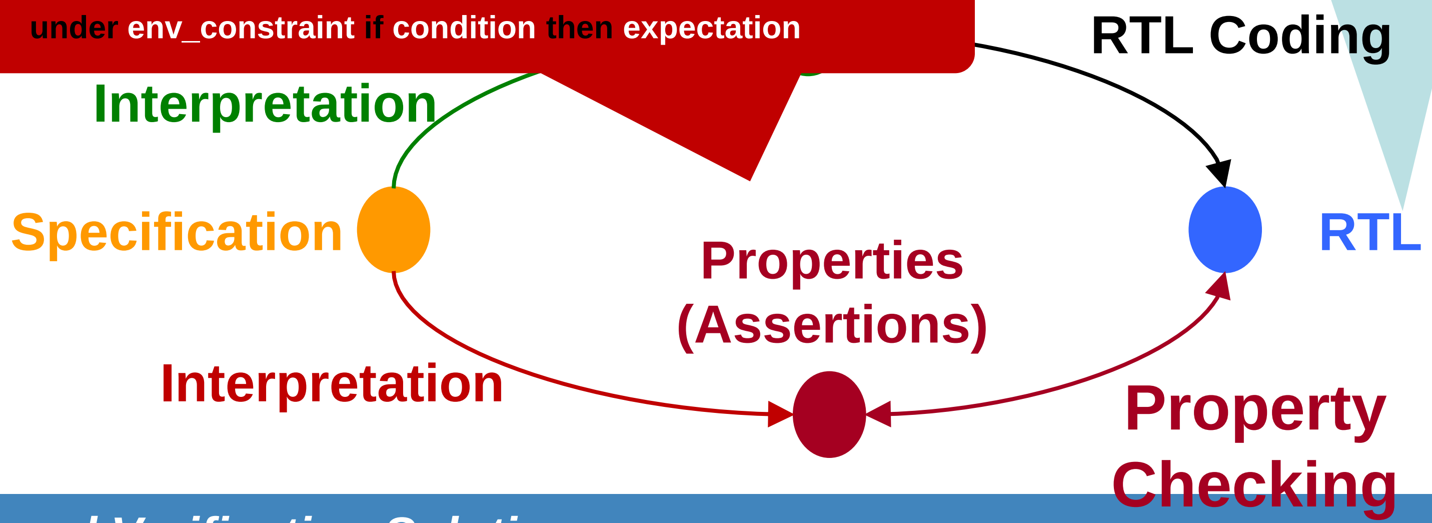
Introduction: Formal Property Checking

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.

Property checking can also be preformed on higher levels of abstraction.

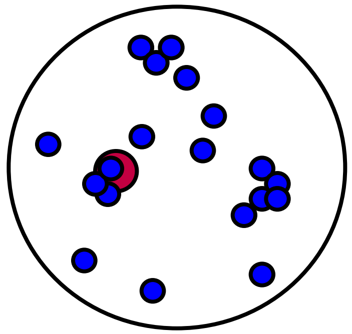
under env_constraint if condition then expectation

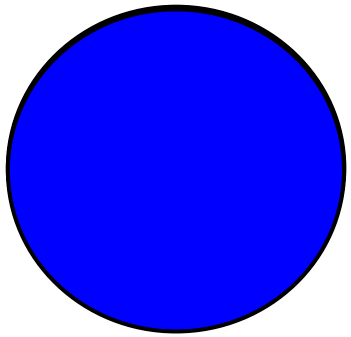


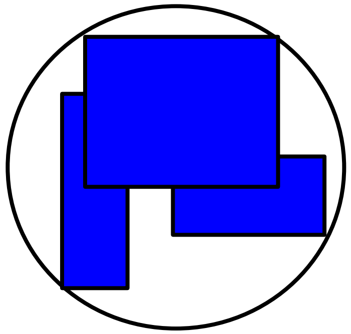
Introduction: Role of Simulation

- **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







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

Naïve interpretation of exhaustive formal verification: Verify ALL properties.

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

[B. Wile , J.C. Goss and W. Roesner, "Comprehensive Functional Verification – The Complete Industry Cycle", Morgan Kaufman, 2005]

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:	$\mathbf{2^{36}}$	(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} x 3)
– Cycles to formally verify the 500k design:	$\mathbf{2^{20500}}$	(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

Structural Coverage – FSM Coverage

- **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

Code/Structural Coverage - Limitations

- **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

Cross-Product Coverage Model

[O Lachish, E Marcus, S Ur and A Ziv. Hole Analysis for Functional Coverage Data. In proceedings of the 2002 Design Automation Conference (DAC), June 10-14, 2002, New Orleans, Louisiana, USA.]

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!**

Functional Coverage	Code Coverage	Interpretation
Low	Low	There is verification work to do.
Low	High	Multi-cycle scenarios, corner cases, cross-correlations still to be covered.
High	Low	Verification plan and/or functional coverage metrics inadequate. Check for "dead" code.
High	High	Increased confidence in quality.

- **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

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.

Assertion Coverage

- **Remember, simulation can only show the presence of bugs, but never prove their absence!**

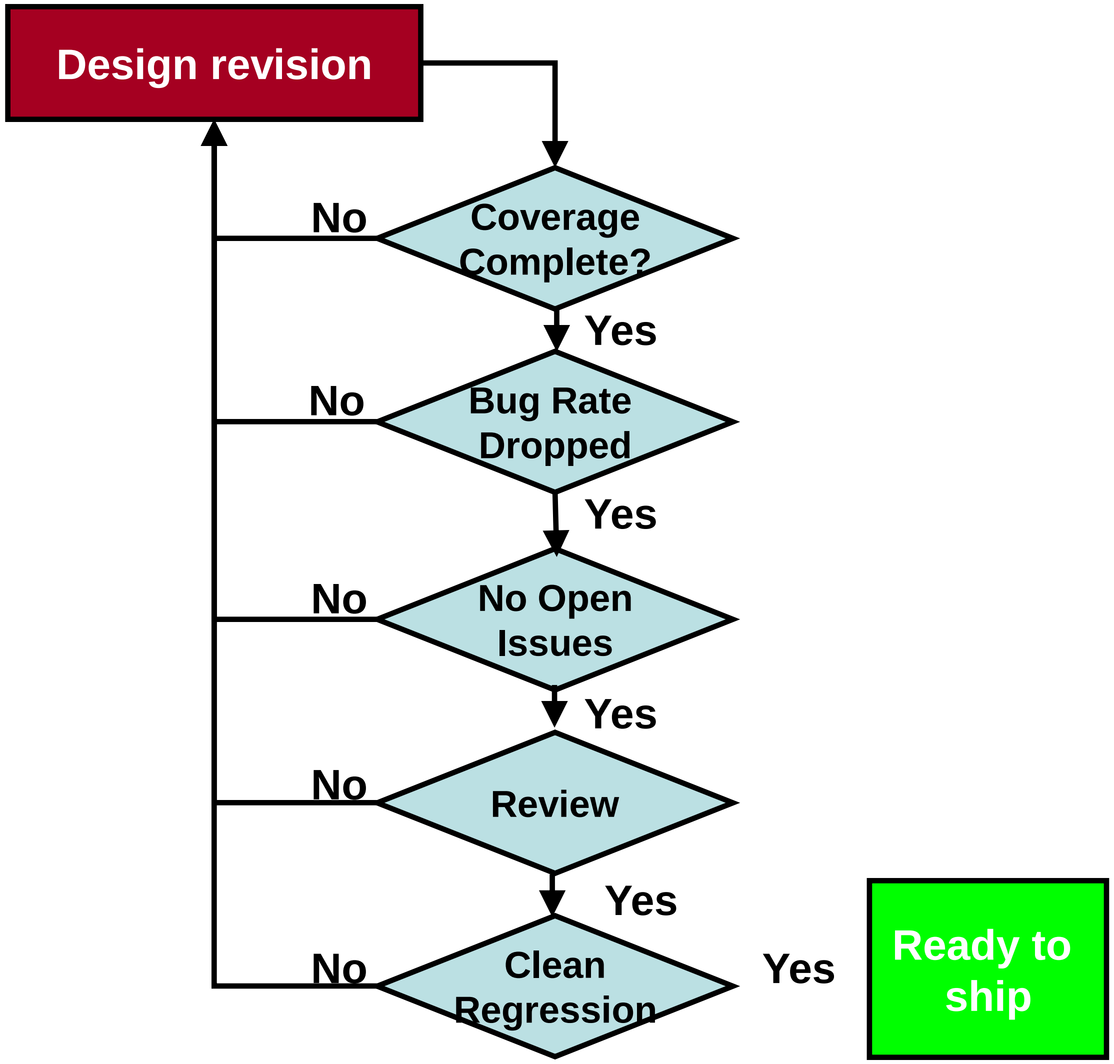
- **An assertion has never fired - what does this mean?**
  - Does not necessarily mean that it can never be violated!
  - Unless simulation is exhaustive..., which in practice it never will be.
  - It might not have fired because it was never evaluated.

- **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**



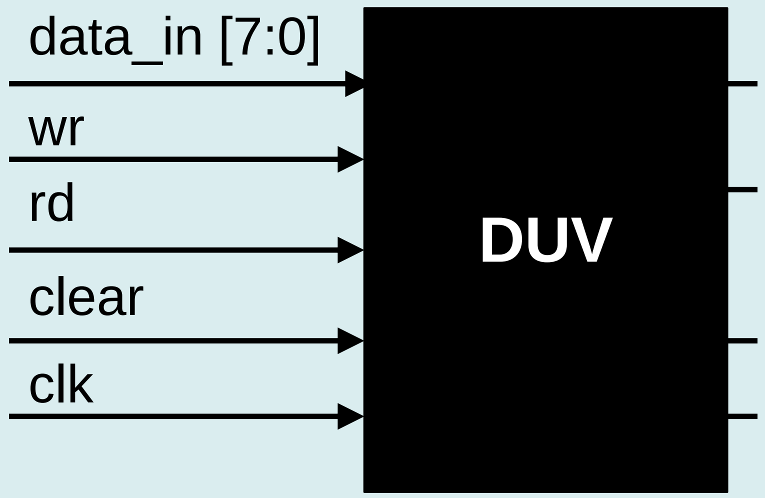
Test and Verification Solutions

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

Example DUV

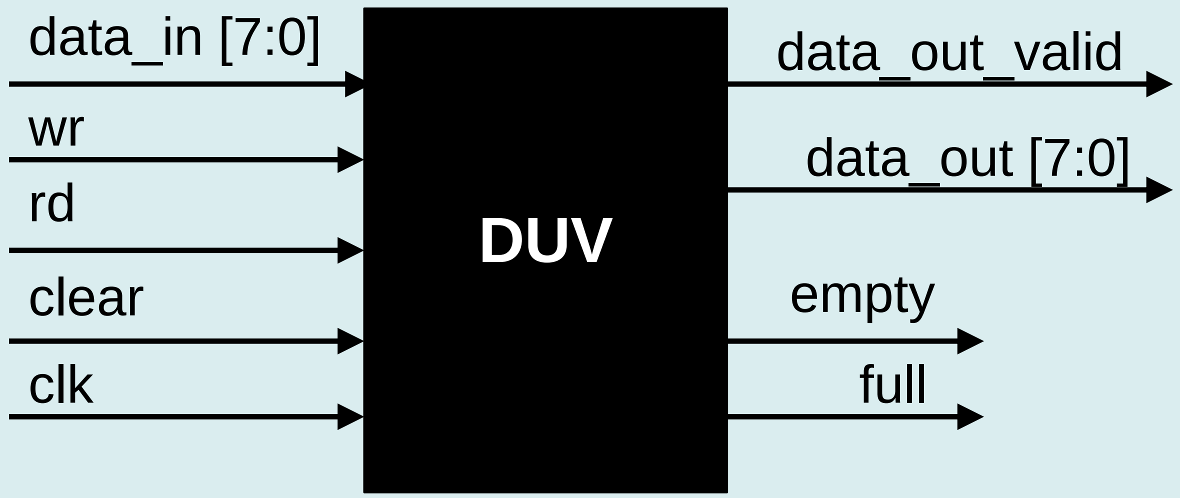
Kerstin Eder, TVS

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.

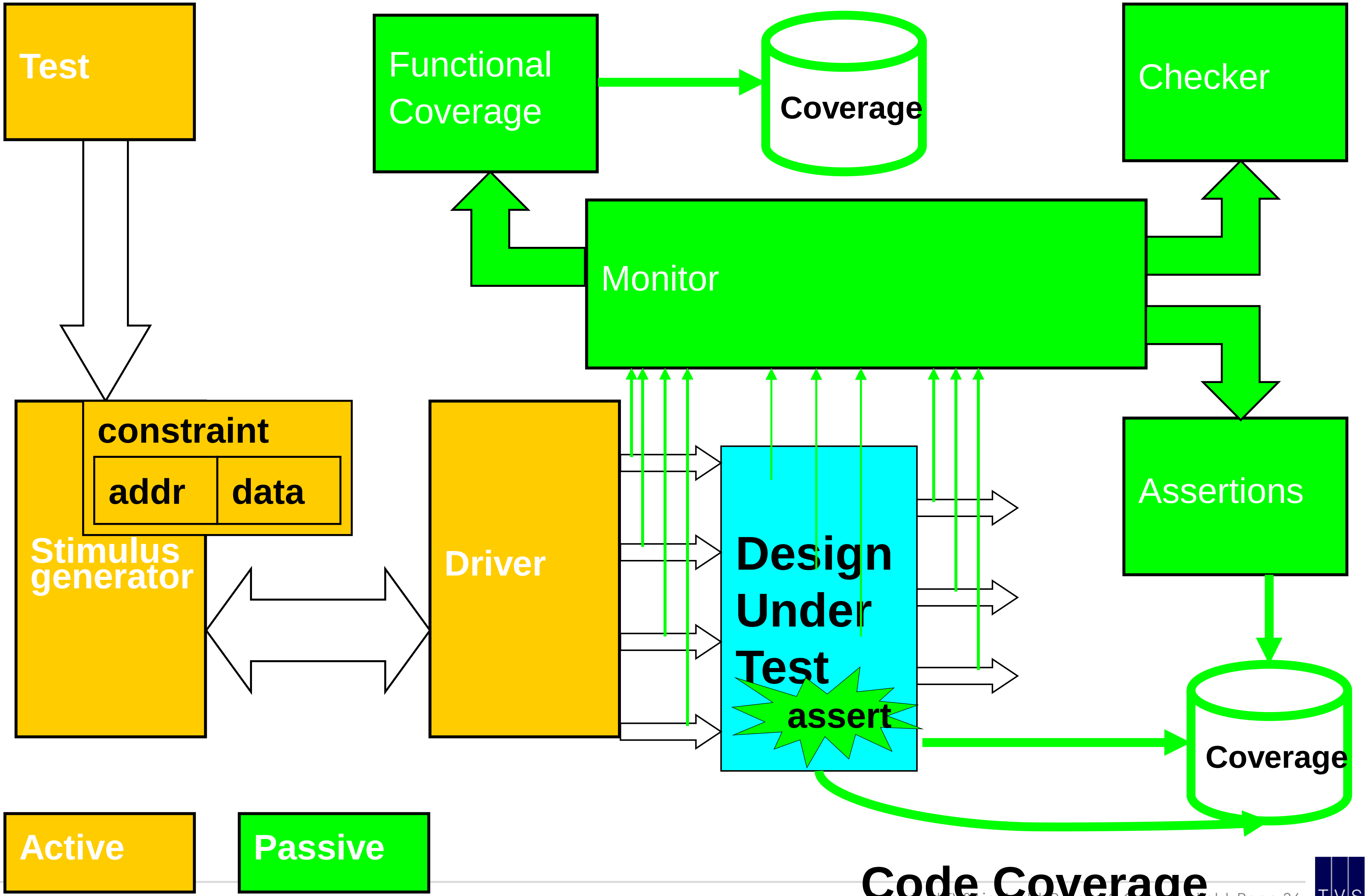
Assertions:

- Distinguish between protocol properties and design properties/coverage.
- Protocol coverage is more easily re-usable.

Basic techniques to combat Complexity

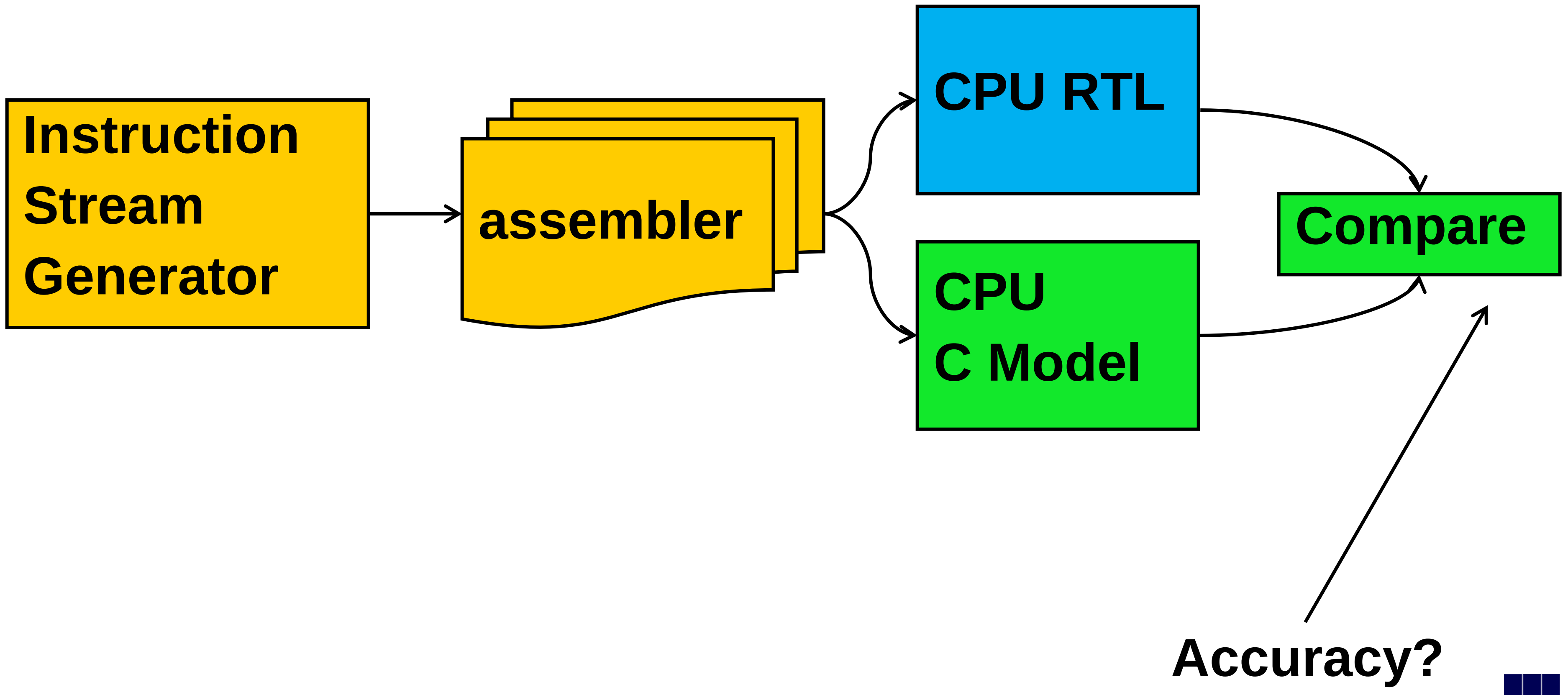
- **Collapse the size of the FIFO to only 2 (or 4) entries and demonstrate**
  - Impact on verification effort
  - Parametrization of design and tb is important
- **Reduce the width of the FIFO's data path**
  - Show this reduces complexity of data coverage
  - Show this helps Formal Verification

The mechanics of an advanced test bench



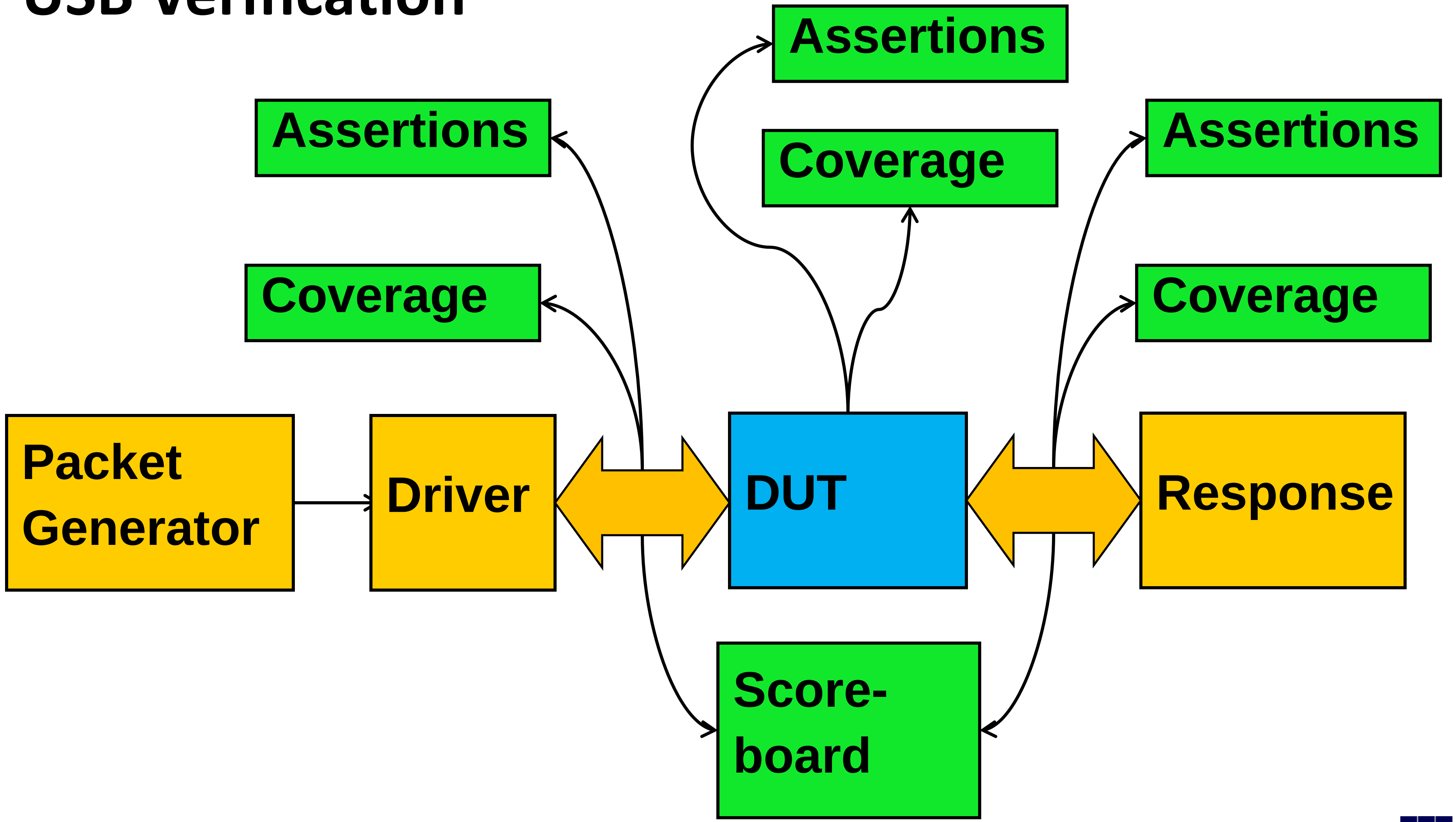
Some hardware verification examples

CPU Verification



Some hardware verification examples

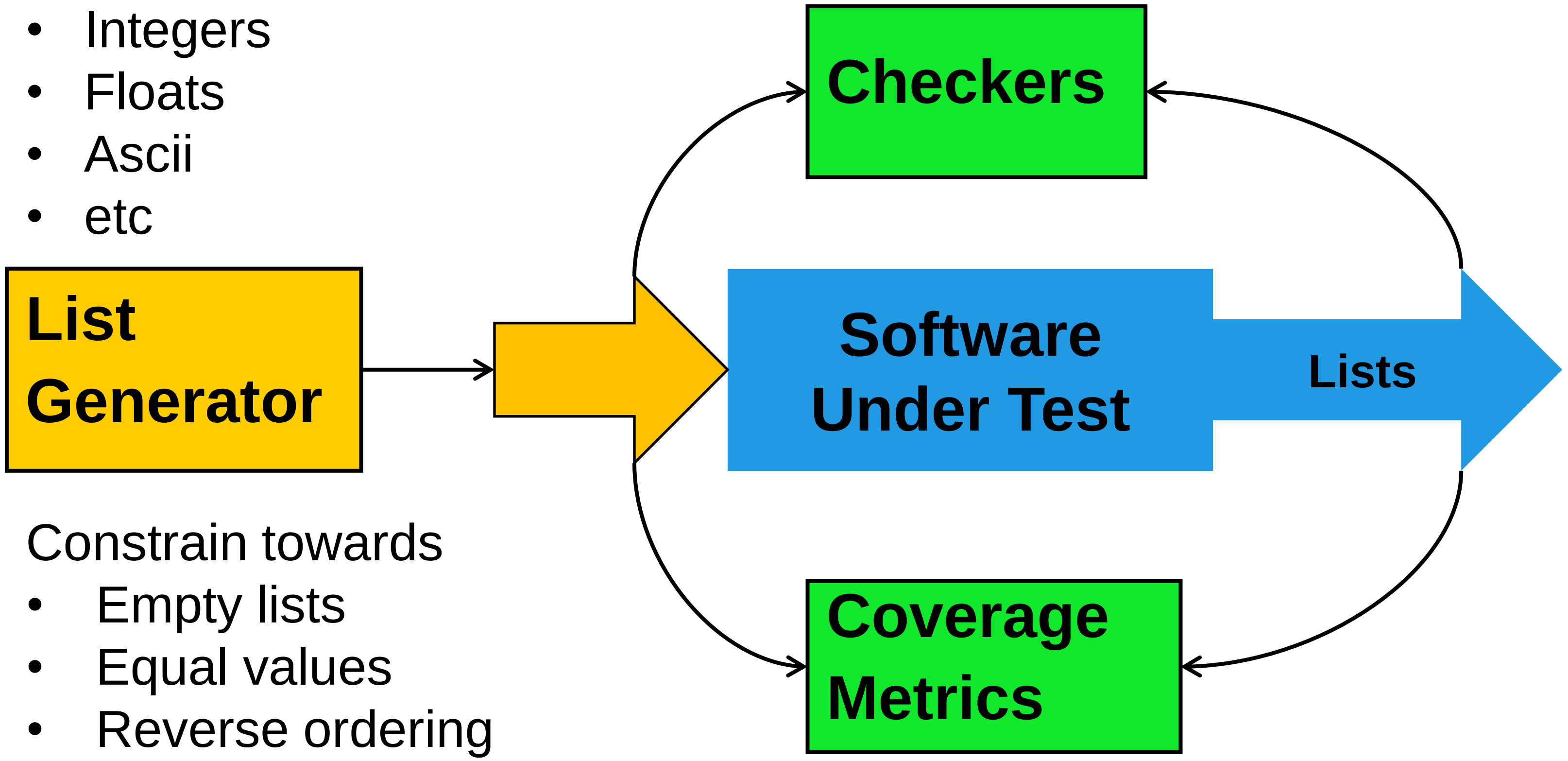
USB Verification



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"



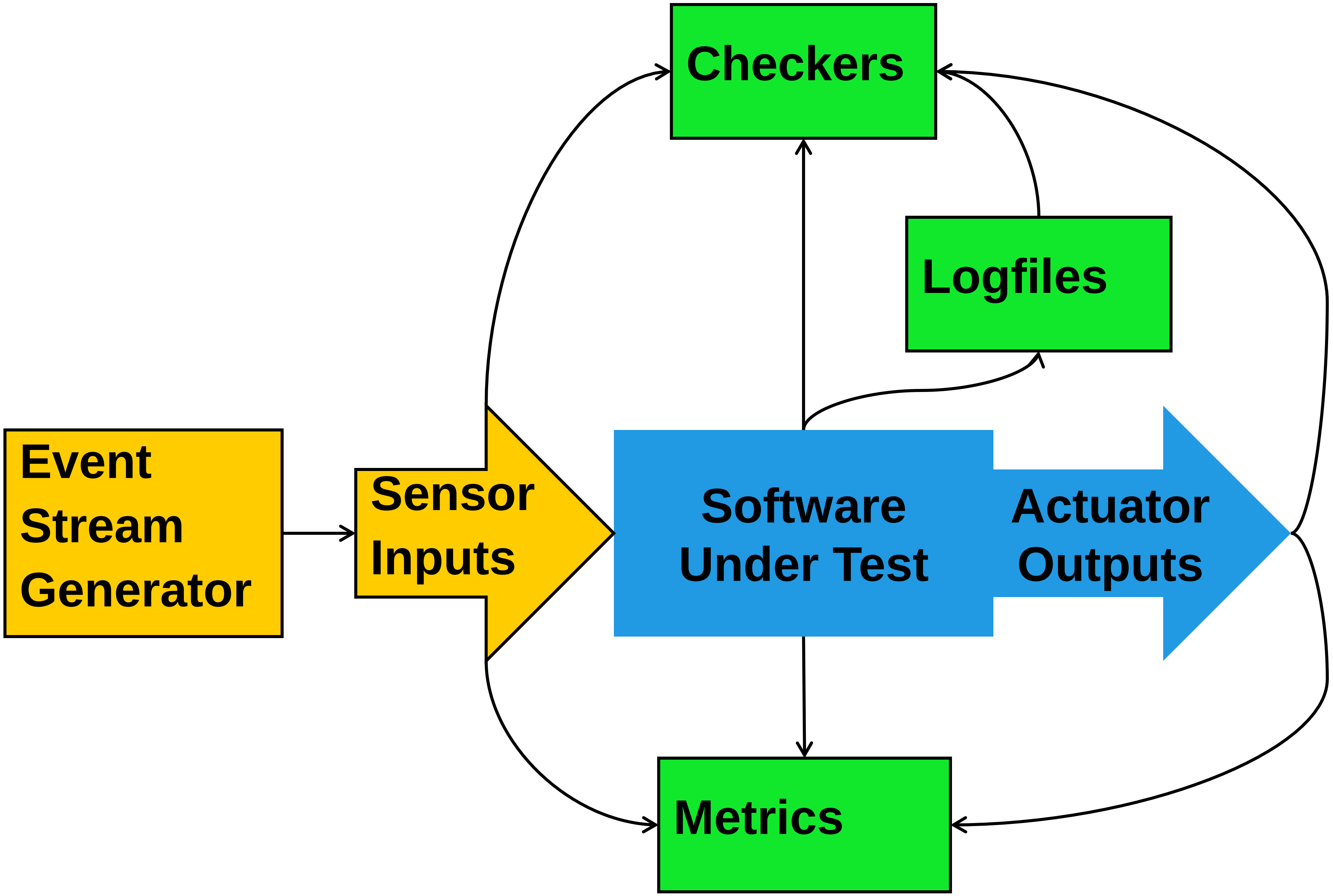
- Check output list is ordered
- Output list contents == input list contents

Lists of
- Integers
- Floats
- Ascii
- etc

Constrain towards
- Empty lists
- Equal values
- Reverse ordering

- Empty List
- Reverse ordered
- Error cases (mix integers, floats, ascii)
- Etc.

Virtual System Level Test Environment



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

From Kerstin Eder of the University of Bristol

- **Requirements coverage**
- **“Cross-product” coverage**

[O Lachish, E Marcus, S Ur and A Ziv. Hole Analysis for Functional Coverage Data. Design Automation Conference (DAC), June 10-14, 2002, New Orleans, Louisiana, USA.]

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**

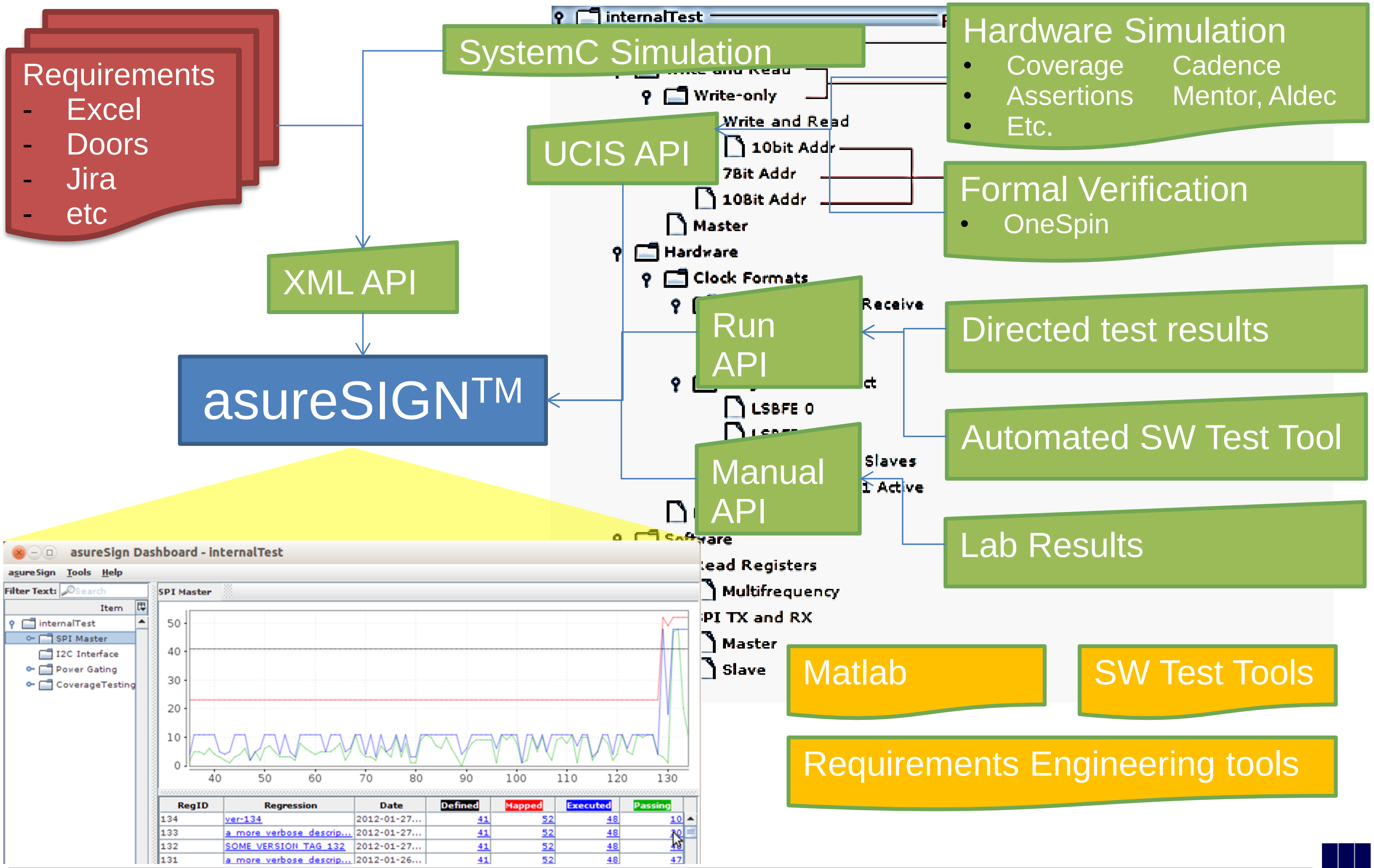
[R Alexander et al. Situation coverage – a coverage criterion for testing autonomous robots. University of York, 2015]

	┬	┤	┘	┐	─	╷	╴	┼	┴	├	└	┌	│	╵	╶
Car															
Bike															
HGV															
Ped															

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



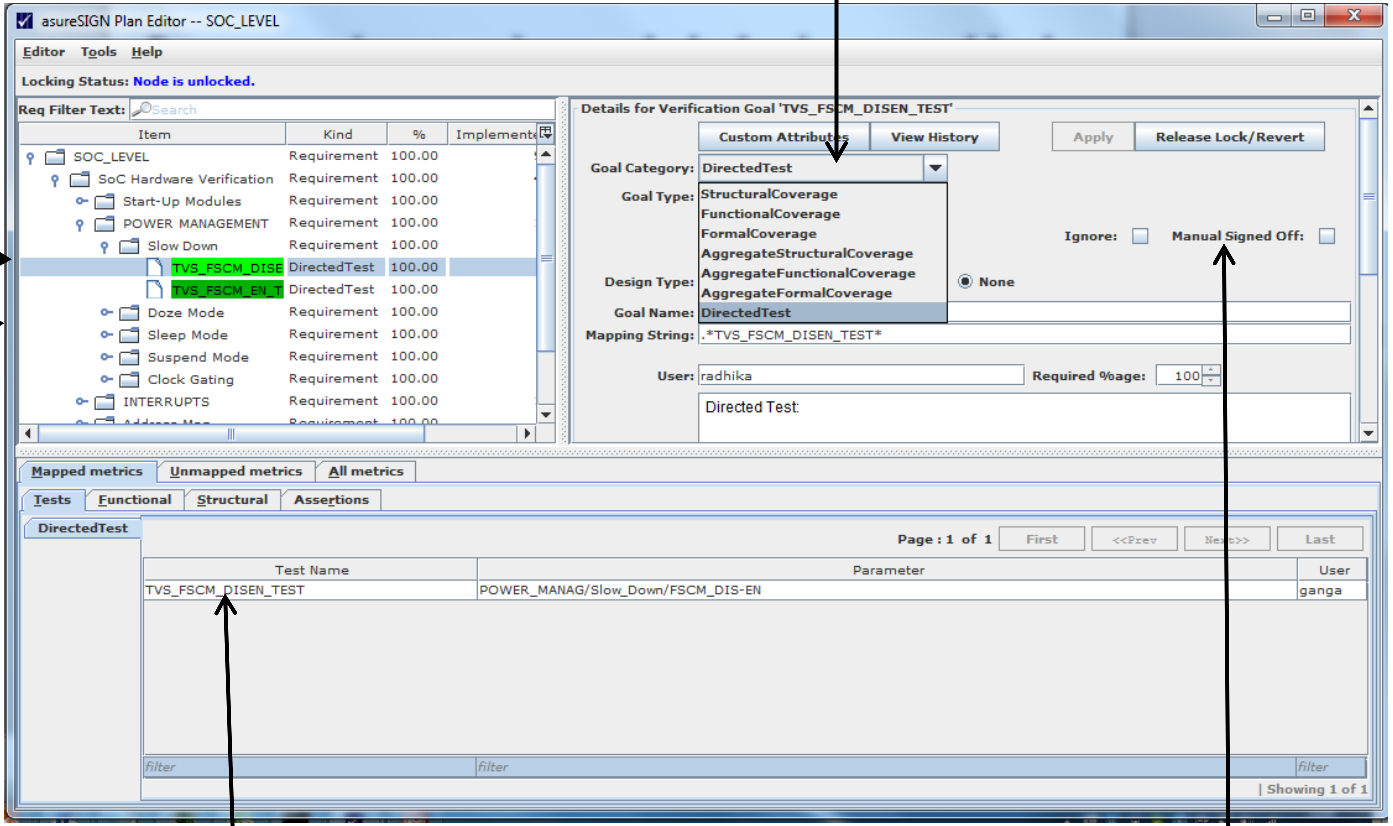
RegID	Regression	Date	Defined	Mapped	Executed	Passing
134	ver-134	2012-01-27...	41	52	48	10
133	a more verbose descrip...	2012-01-27...	41	52	48	30
132	SOME VERSION TAG 132	2012-01-27...	41	52	48	48
131	a more verbose descrip...	2012-01-26...	41	52	48	47

Decomposing requirements to features and tests

Import Requirements (Doors, Excel, Word, etc)

Map requirements to verification goals

Edit Requirements



asureSIGN Plan Editor -- SOC_LEVEL

Editor Tools Help

Locking Status: Node is unlocked.

Req Filter Text: Search

Item	Kind	%	Implemente
SOC_LEVEL	Requirement	100.00	
SoC Hardware Verification	Requirement	100.00	
Start-Up Modules	Requirement	100.00	
POWER MANAGEMENT	Requirement	100.00	
Slow Down	Requirement	100.00	
TVS_FSCM_DISE	DirectedTest	100.00	
TVS_FSCM_EN_T	DirectedTest	100.00	
Doze Mode	Requirement	100.00	
Sleep Mode	Requirement	100.00	
Suspend Mode	Requirement	100.00	
Clock Gating	Requirement	100.00	
INTERRUPTS	Requirement	100.00	

Details for Verification Goal 'TVS_FSCM_DISEN_TEST'

Goal Category: DirectedTest

Goal Type: StructuralCoverage / FunctionalCoverage / FormalCoverage / AggregateStructuralCoverage / AggregateFunctionalCoverage / AggregateFormalCoverage / DirectedTest

Ignore: ☐ Manual Signed Off: ☐

Design Type: ● None

Goal Name: DirectedTest

Mapping String: .*TVS_FSCM_DISEN_TEST*

User: radhika Required %age: 100

Directed Test:

Mapped metrics | Unmapped metrics | All metrics

Tests | Functional | Structural | Assertions

DirectedTest

Page : 1 of 1

Test Name	Parameter	User
TVS_FSCM_DISEN_TEST	POWER_MANAG/Slow_Down/FSCM_DIS-EN	ganga

| Showing 1 of 1

The mapped verification goal

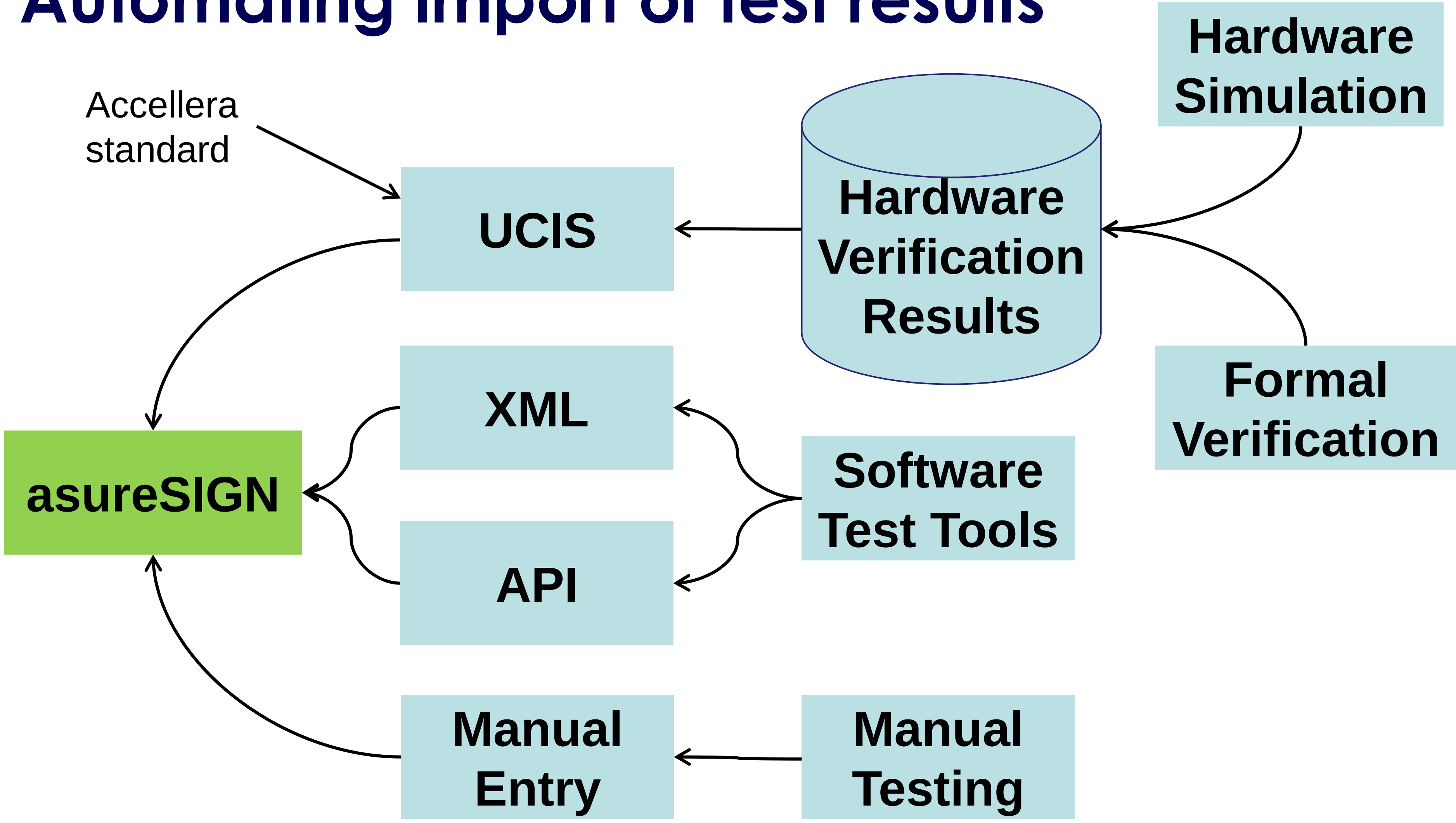
Sign off a requirement with a manual test (e.g. in the lab)

Safety compliance (asureSIGN)

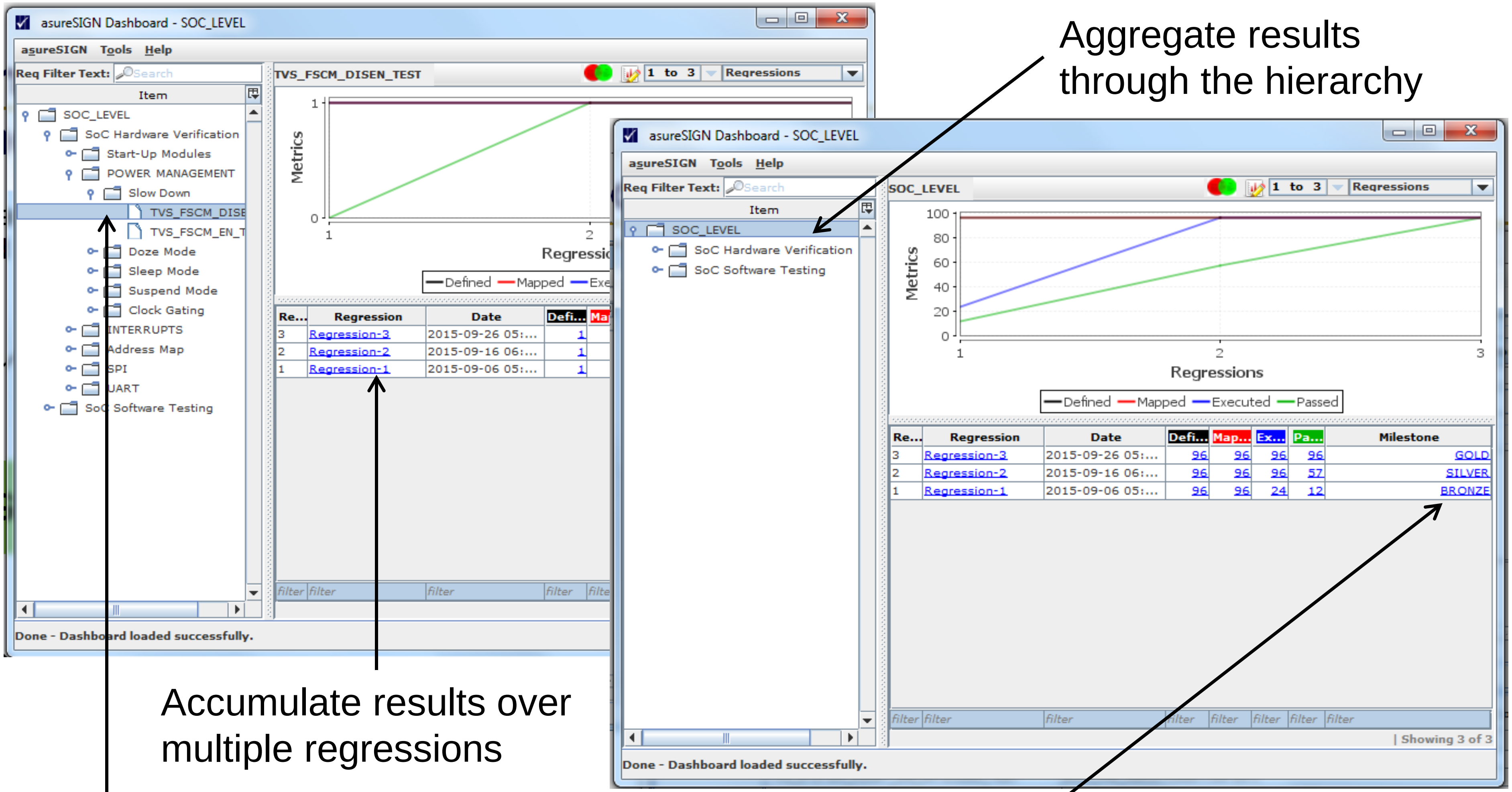
- **Managing Requirements**
  - Importing and editing requirements
- **Decomposing requirements to verification goals**
- **Tracking test execution**
  - Automating import of test results
  - Automate accumulation and aggregation of test results
- **Impact analysis**
  - Managing changes in requirements and tests
- **Demonstrating compliance to DO254 & DO178C**
- **Managing multiple projects**

Tracking test execution:

Automating import of test results



Automate accumulation and aggregation of test results



Re...	Regression	Date	Defi...	Ma
3	Regression-3	2015-09-26 05:...	1	
2	Regression-2	2015-09-16 06:...	1	
1	Regression-1	2015-09-06 05:...	1	

Re...	Regression	Date	Defi...	Map...	Ex...	Pa...	Milestone
3	Regression-3	2015-09-26 05:...	96	96	96	96	GOLD
2	Regression-2	2015-09-16 06:...	96	96	96	57	SILVER
1	Regression-1	2015-09-06 05:...	96	96	24	12	BRONZE

Aggregate results through the hierarchy

Accumulate results over multiple regressions

Record results from each test

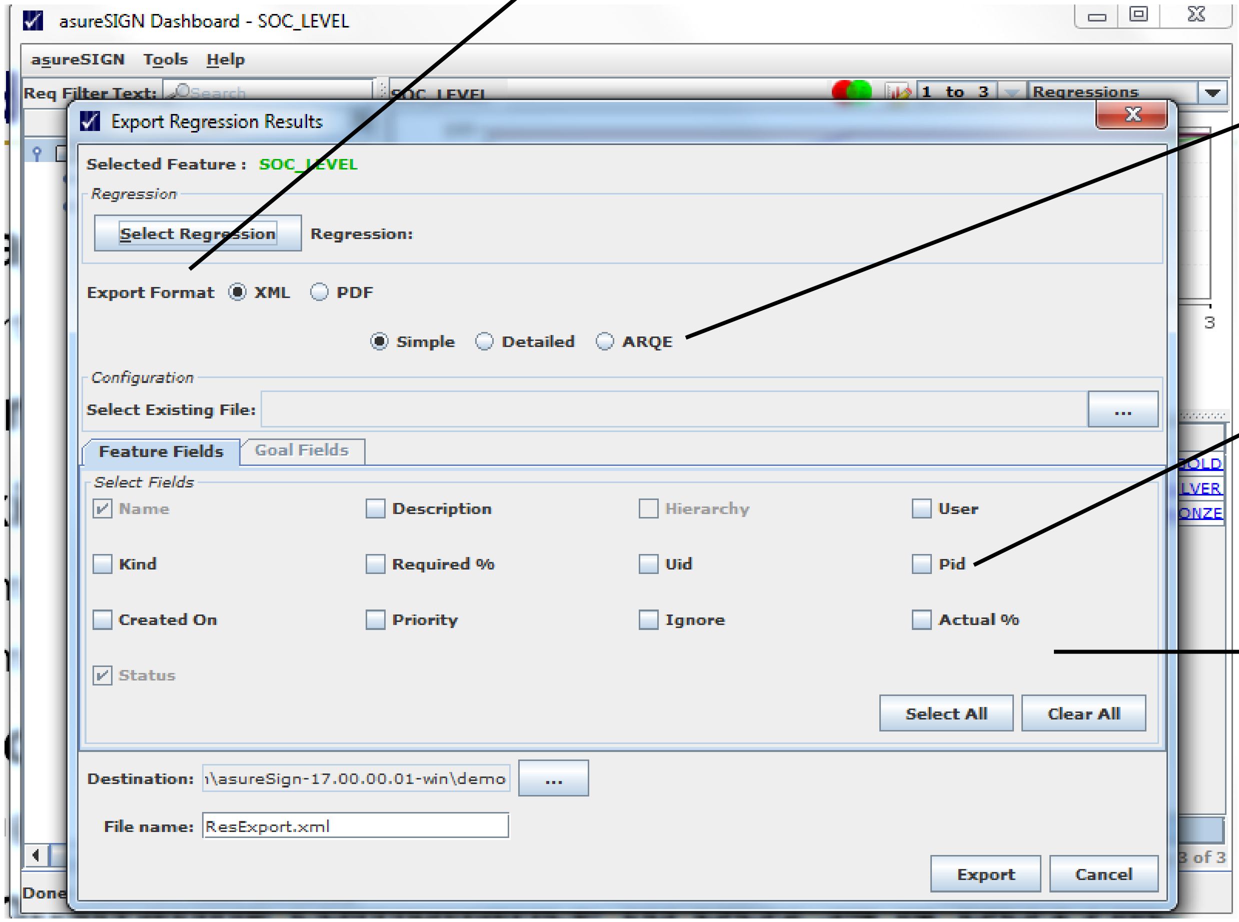
Define and track against interim milestones (based on % of requirements tested)

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



Selected Feature : SOC_LEVEL

Regression

Regression:

Export Format [✓ XML] PDF

[✓ Simple] Detailed ARQE

Configuration

Select Existing File:

Feature Fields | Goal Fields

Select Fields

✓ Name	Description	Hierarchy	User
Kind	Required %	Uid	Pid
Created On	Priority	Ignore	Actual %
✓ Status			

Destination: \asureSign-17.00.00.01-win\demo

File name: ResExport.xml

Select level of detail

Pid = unique reference to requirement in external tool

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