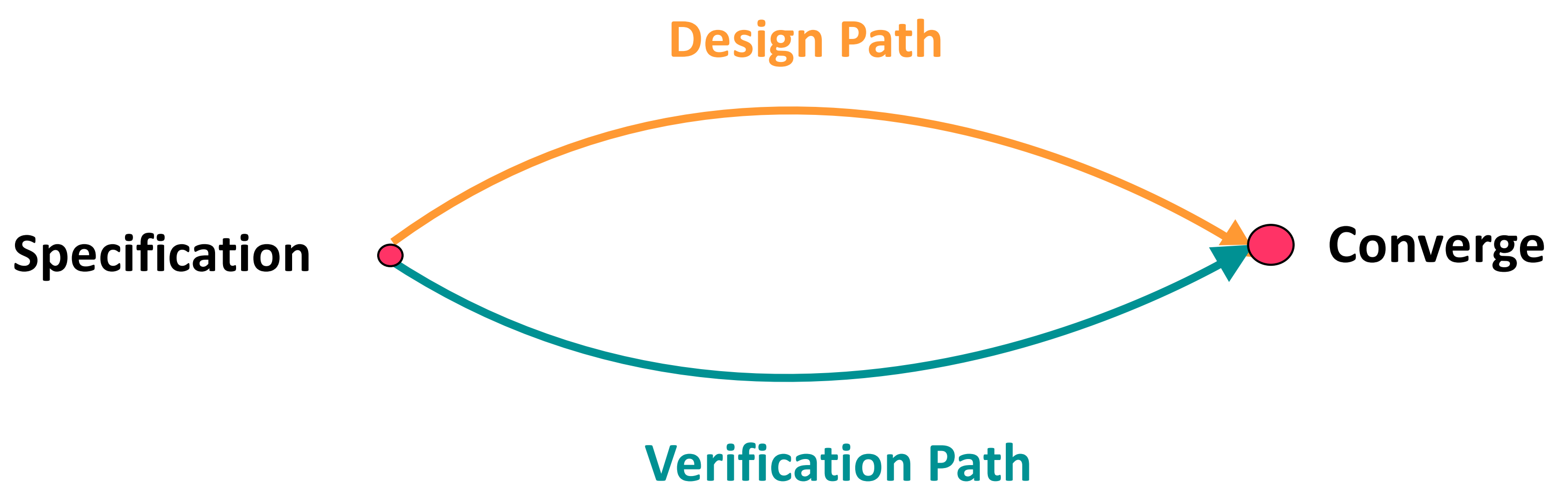
# Lecture 22:
# Introduction to Design Verification

**Mark McDermott**
**Electrical and Computer Engineering**
**The University of Texas at Austin**

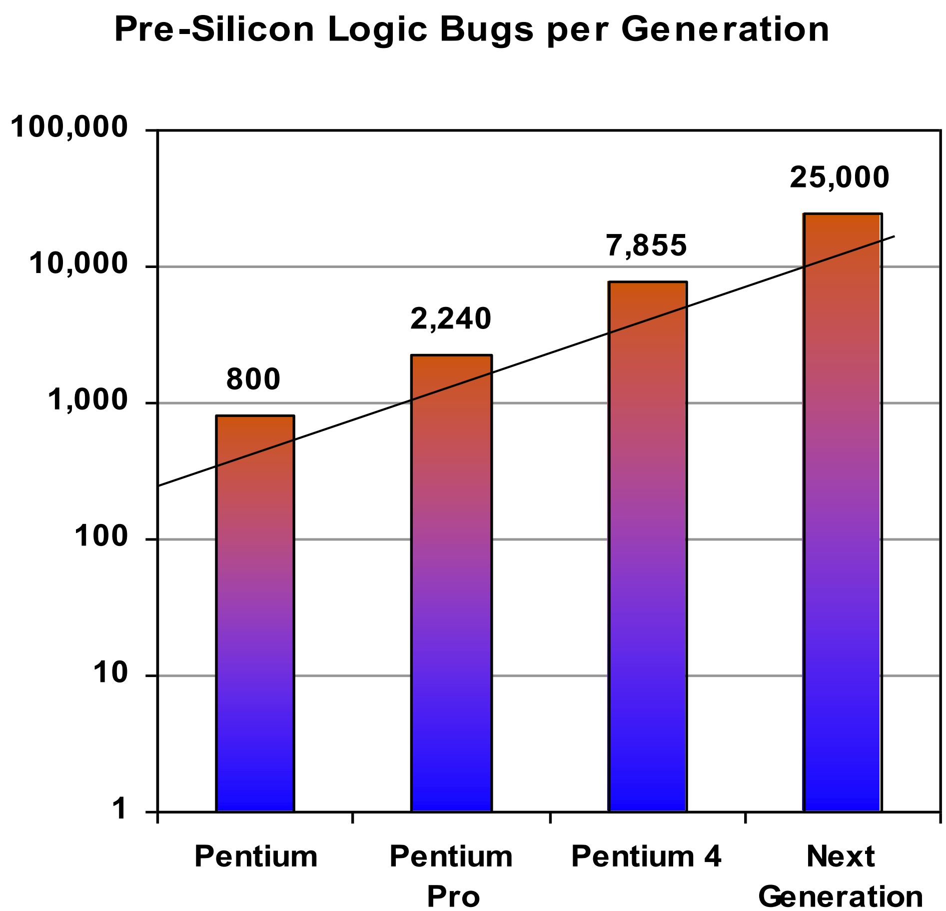
**Mike Warner**
**Mentor Graphics**

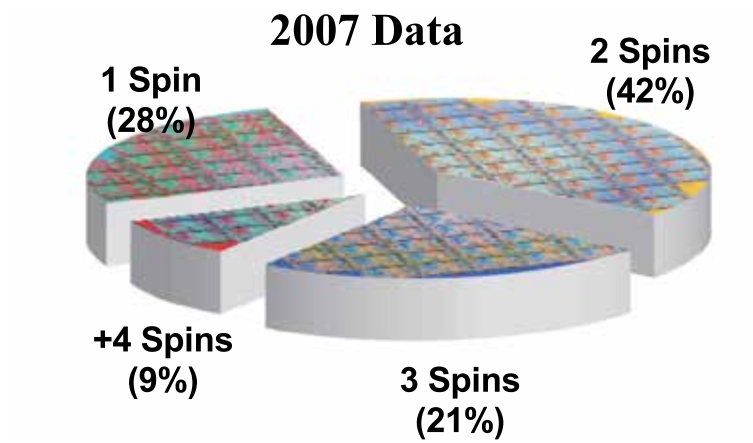
# What is functional verification?

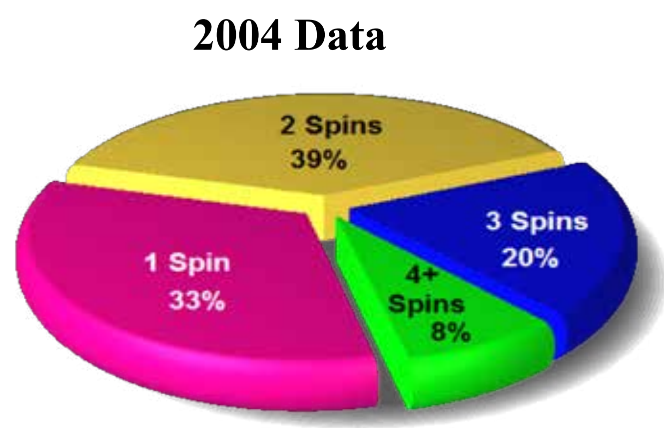
- **Verification is the process of insuring the intent of the specification is preserved in the implementation**



# Why should I care?



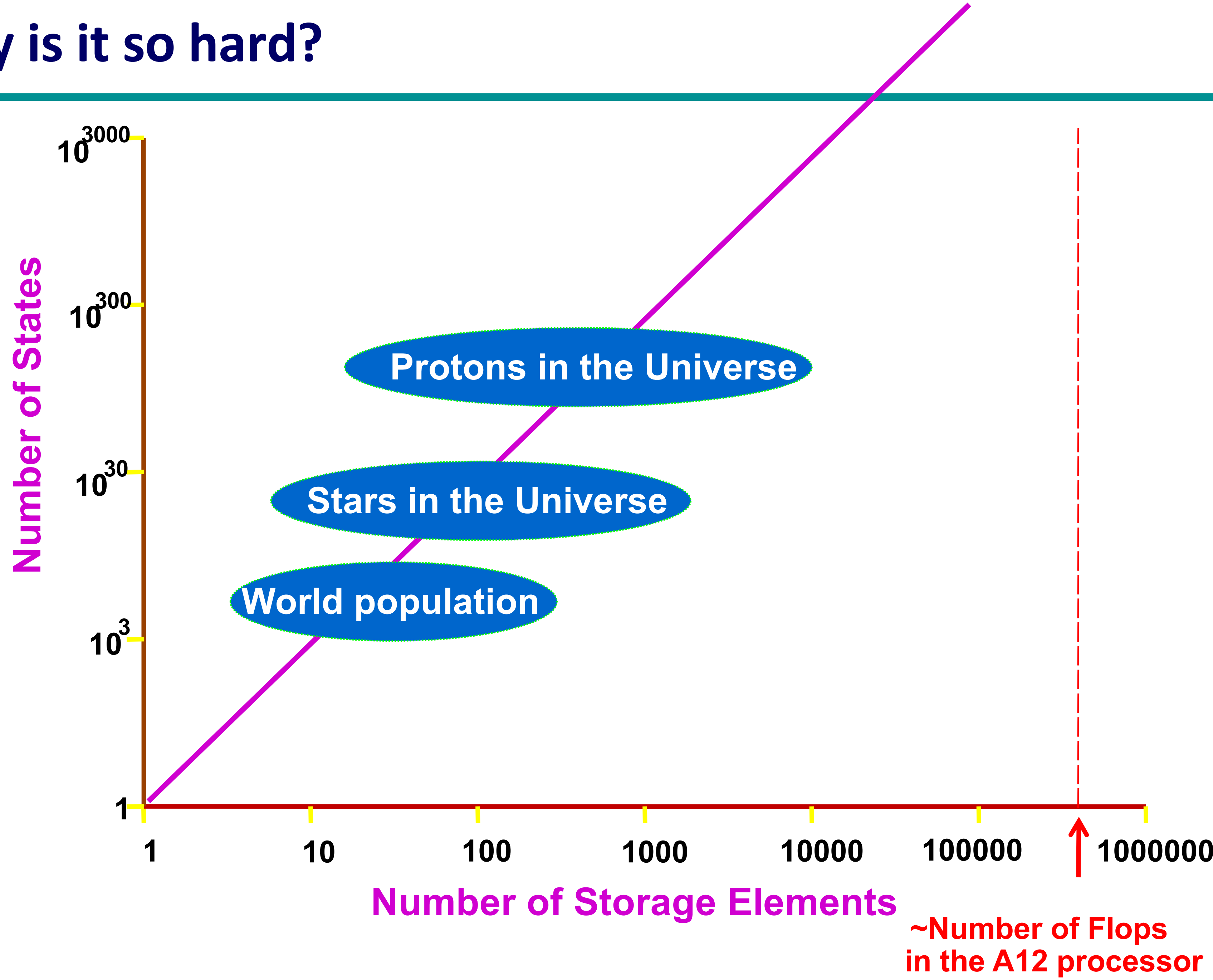

*Source: Tom Schubert, Intel "High Level Formal Verification of Next –Generation Microprocessors" DAC 2003*
*Intel Corporate Web Site "Moore's Law … http://www.intel.com/technology/silicon/mooreslaw/index.htm*

# So what's the big deal?

| Design Phase | Architecture | Specification | Logic Design | Verification | HW-SW Co-Verification | Post-Si Test | Customer Return | Total Bug Detection Cost (based on 1000 bug project) |
|---|---|---|---|---|---|---|---|---|
| Cost of Finding Bug | $1 | $10 | $100 | $1,000 | $10,000 | $100,000 | $1,000,000 | |
| Bug Injection Distribution | 10% | 30% | 60% | 0% | 0% | 0% | 0% | |
| Bug Distribution Best In Class | 5% | 10% | 20% | 64% | 1% | 0.1% | 0.01% | $961,050 |
| Bug Distribution Second Fiddle | 5% | 10% | 20% | 54% | 10% | 1% | 0.1% | $3,561,050 |
| Bug Distribution Breaking Even | 5% | 10% | 20% | 44% | 15% | 5% | 1% | $16,961,050 |
| Bug Distribution Fire Sale | 5% | 10% | 20% | 38% | 20% | 10% | 2% | $32,401,050 |

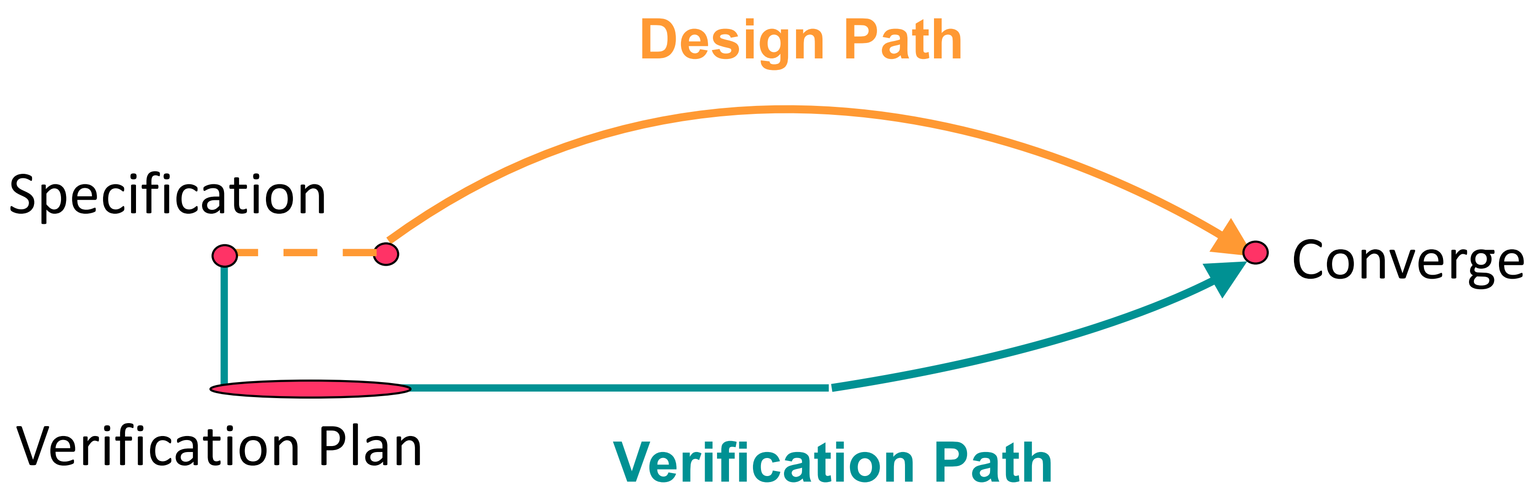
Cost of poor verification

# Why is it so hard?



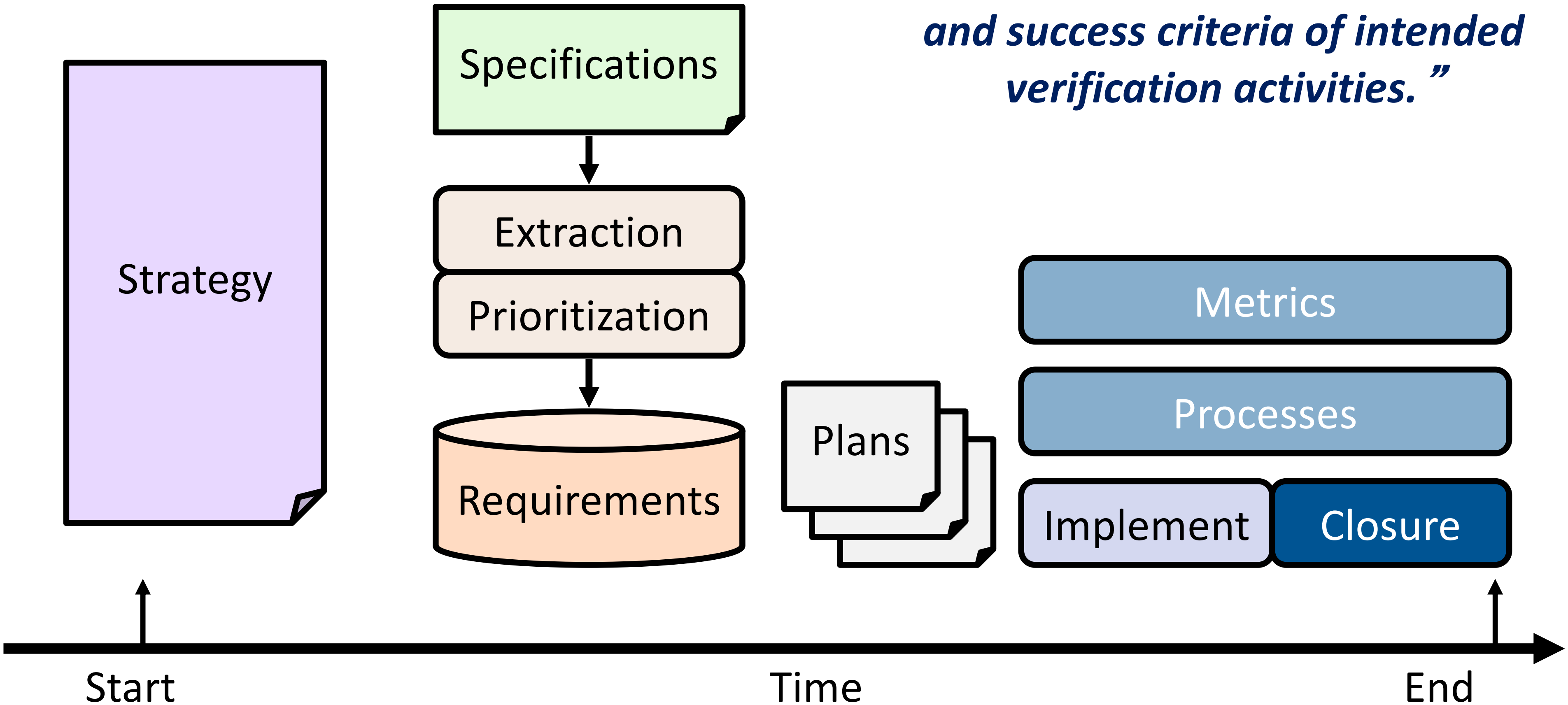
# Verification Reality

- **Verification averages 70% of design effort (50%-80%)**
- **Why?**
  - **Verification is open loop, never ending, non-deterministic**



# What is a Verification Plan?

*"A document describing the methodology, environment, resources, requirements, priorities and success criteria of intended verification activities."*

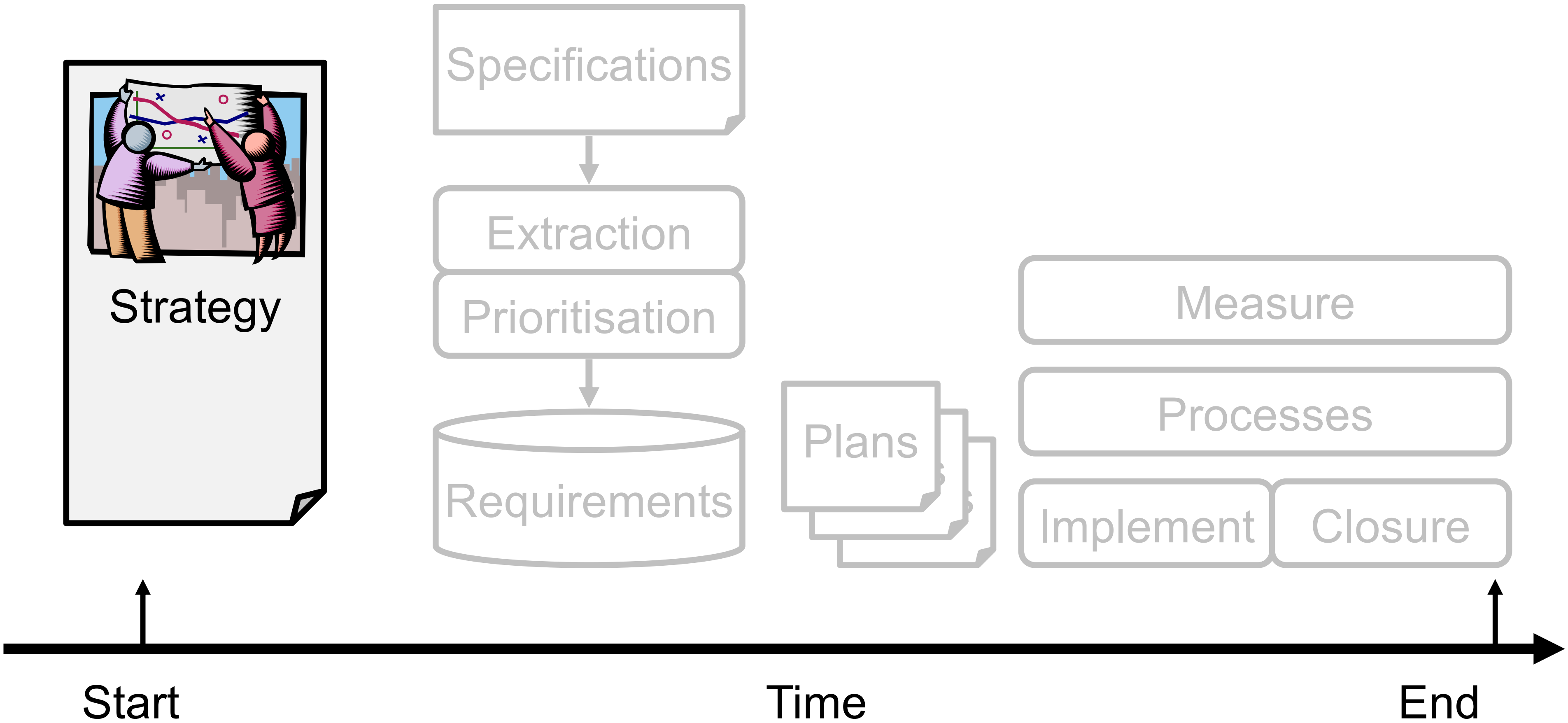



ANSI/IEEE Standard 829-1983 for Software Test Documentation

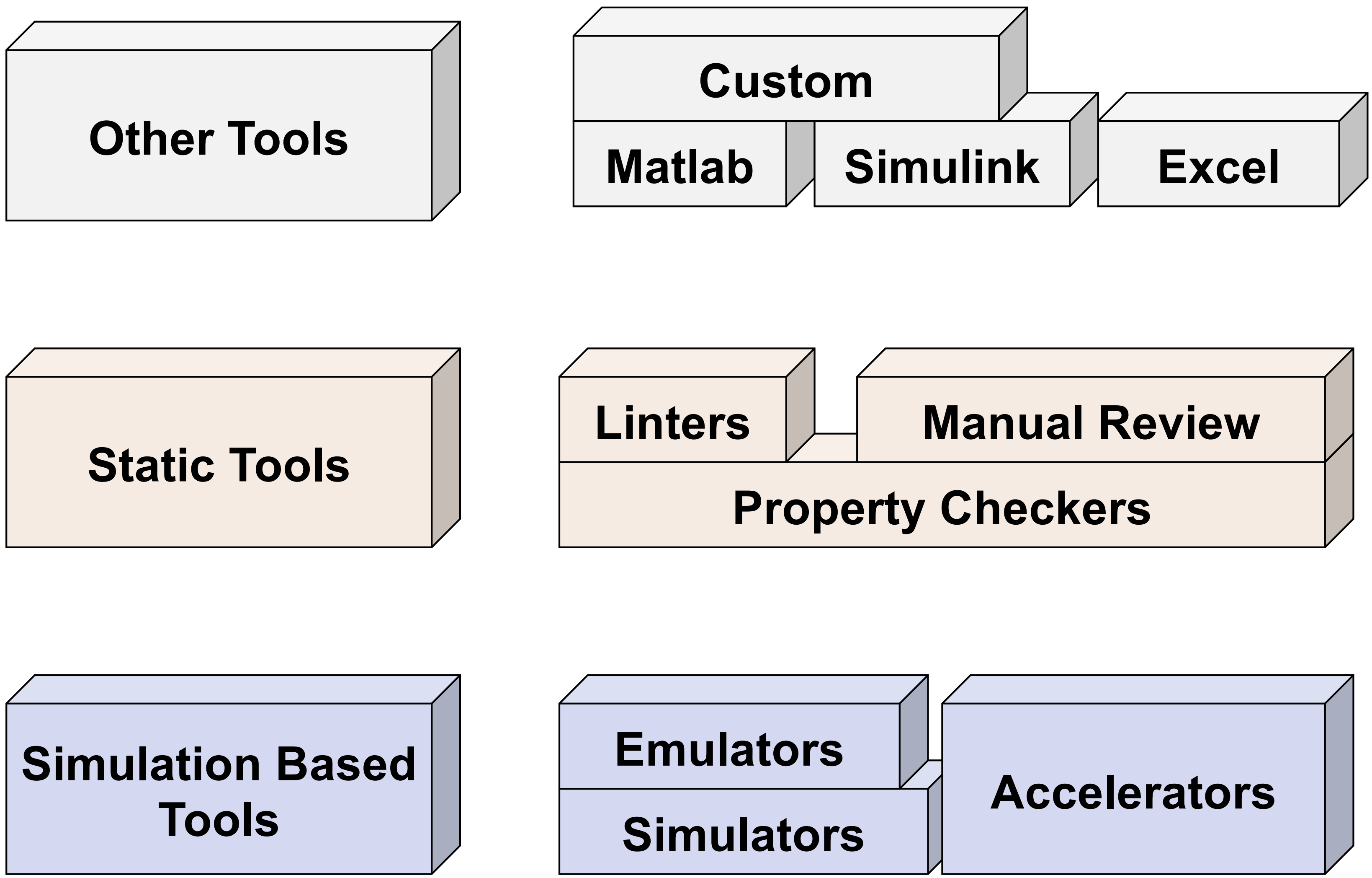
# Context: Strategy

This is <u>how</u> we go about doing verification <u>in general</u>

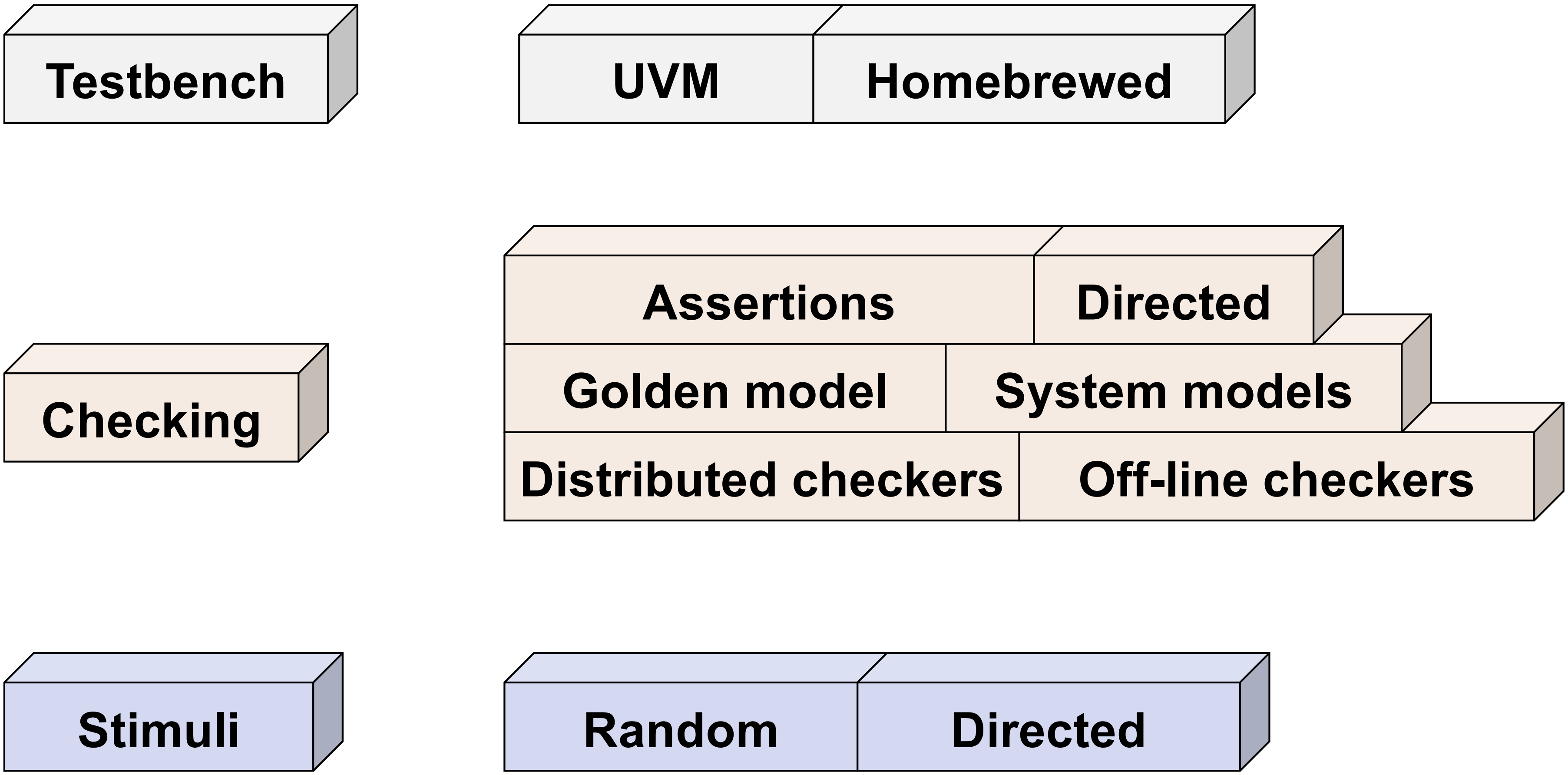
These are our goals, tools, technologies and processes



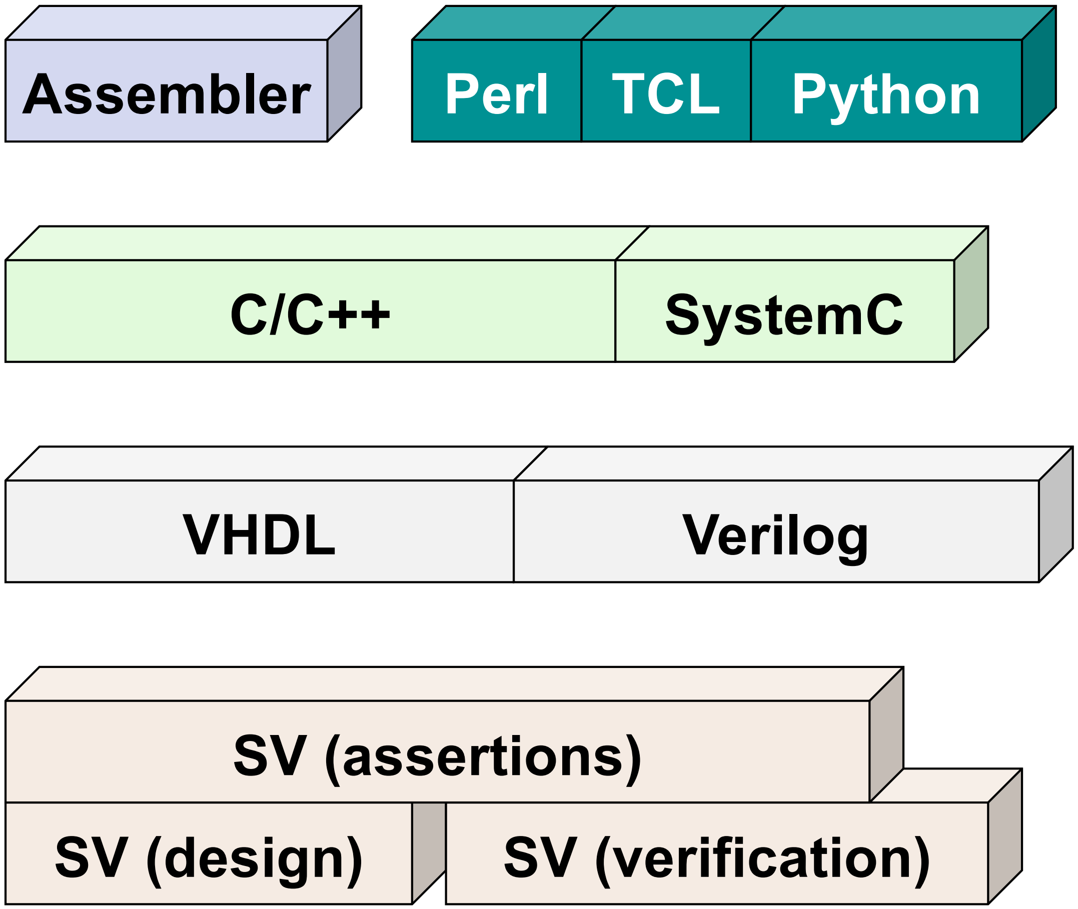
# Strategy: Tools



# Strategy: Methodology



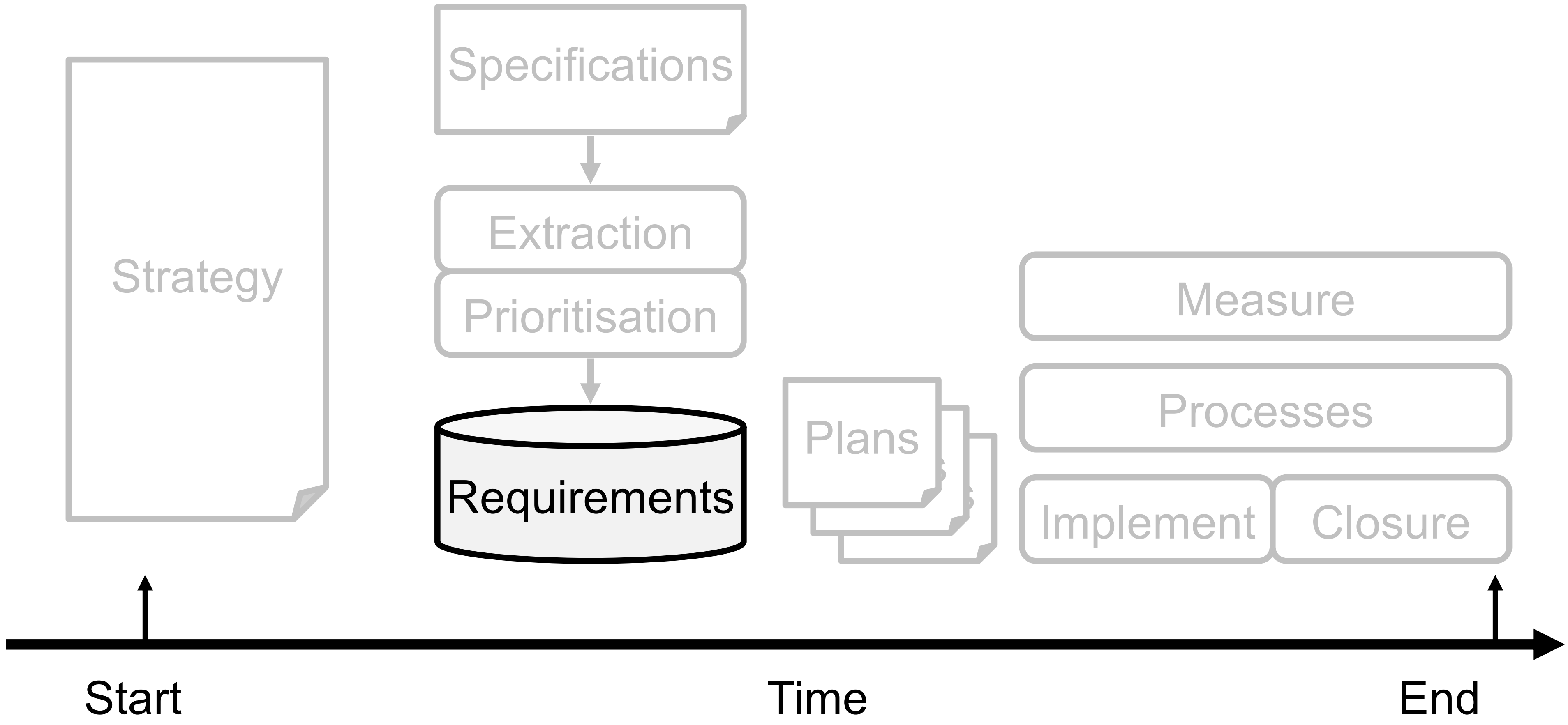
# Strategy: Language



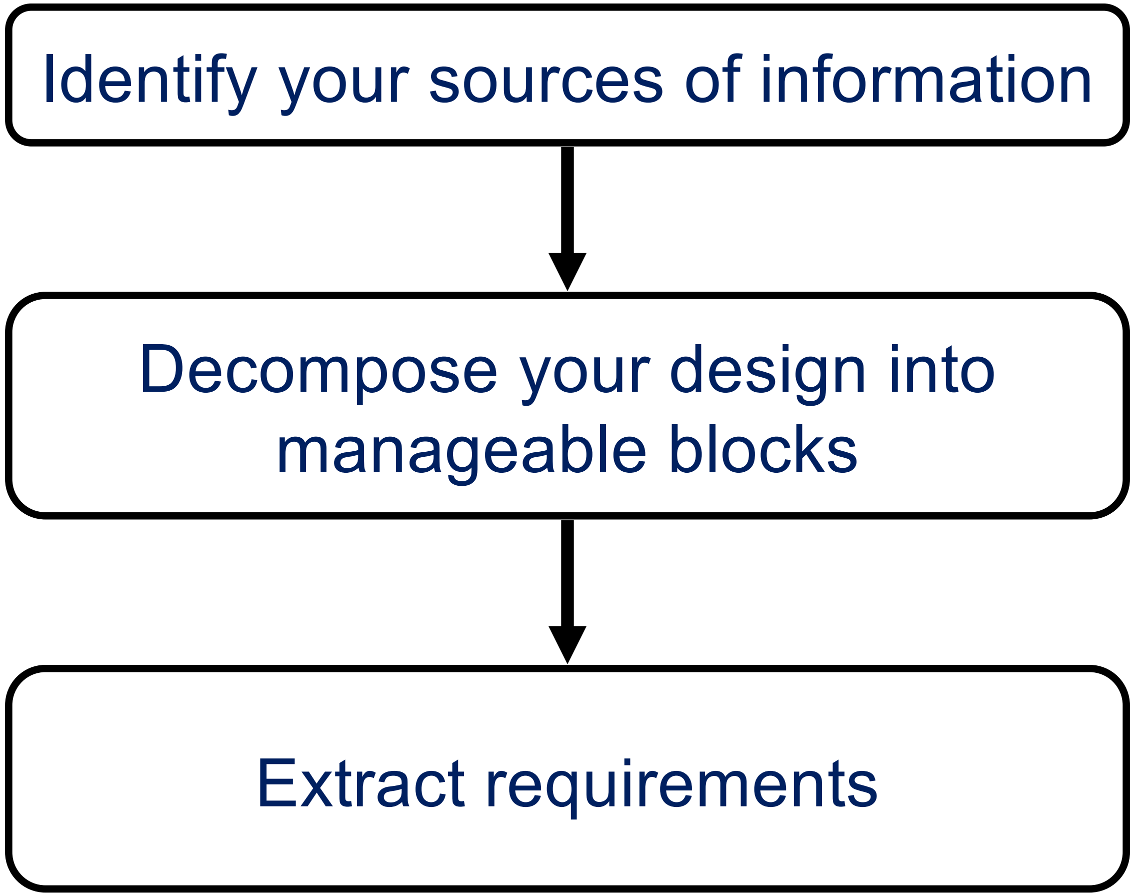
# Context: Requirements

**This is what we need to verify in a particular design**

**These are the features we need to hit and how important they are**

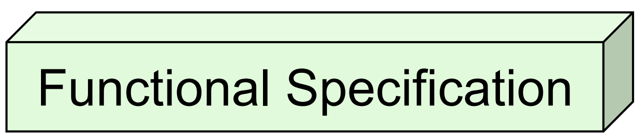


# Requirements: Extraction



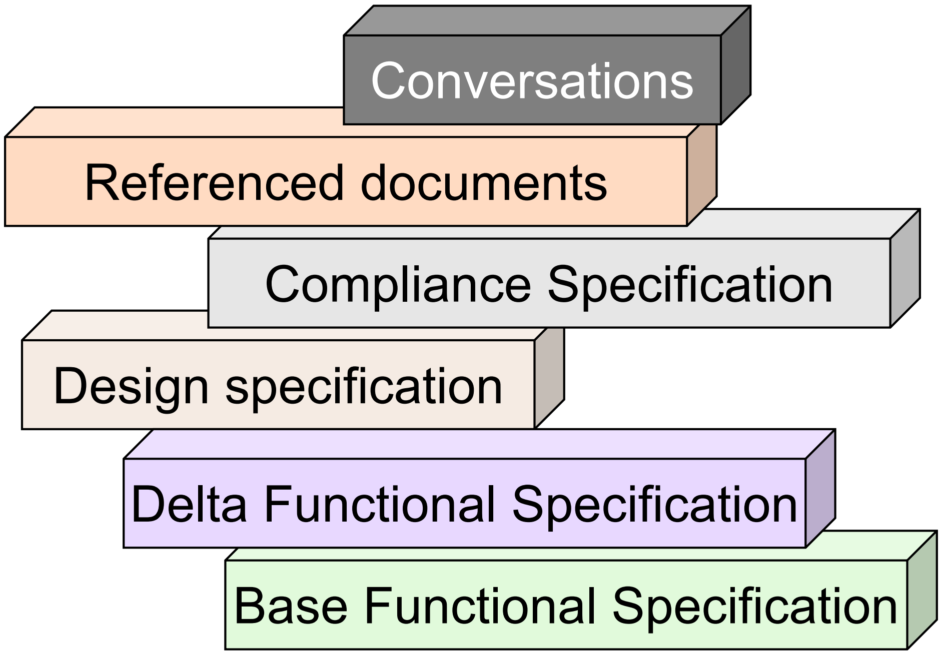
# Requirements: Sources of Information

- **Where should you go to get your requirements?**
- **Functional specification is the obvious choice**
  - although be aware that the required information for one feature might come from several places
- **In theory, it contains all you need**



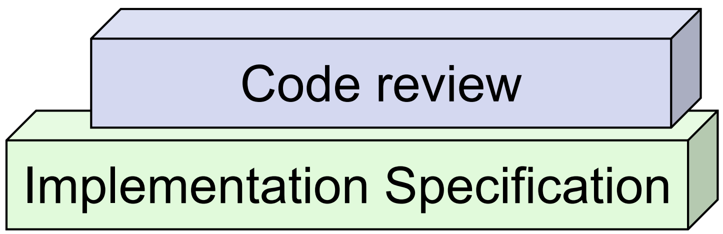
# Requirements: Sources of Information

- **How many functional specifications though?**



# Requirements: Sources of Information

- **What about implementation details?**
  - the functional specification might not mention the architecture used in a block, but it still has to be verified to check corner case handling
    - FIFO full flags
    - Counter wraparound
    - Local shared buffer arbiters



# Requirements: Block Level

| Possible subsections | Useful places to look |
|---|---|
| Major and Minor Modes of operation | The specification headers, module key features, register fields, configuration parameters |
| Exception handling | Interrupts, status bits, illegal inputs, "what if" questions |
| Interfaces | Pin list, module key features, timing diagrams |
| Registers | The register map |
| Interrupts | The register map, pin list, the module key features, status bits |
| Implementation | The implementation specification, the RTL, the designer |

# Requirements: Implementation

- **Finite State Machines**
  - encoding (one hot or only legal values)
  - enters reset state on reset
- **Interfaces**
  - usage assumptions not violated
  - outputs become disabled during reset
- **FIFOs and stacks**
  - full and empty flags used correctly
- **Registers**
  - Read-only bits can never be written
  - reset to defaults
  - updated correctly from bus
- **Arithmetic under- or over-flow**
- **Signals remain mutually exclusive**
- **Clock synchronization blocks**

# Requirements: Read Between the Lines

- The specification will not have all the details
  - learn to read between the lines
  - what wasn't mentioned that should have been?
  - what feasible error conditions should be dealt with?

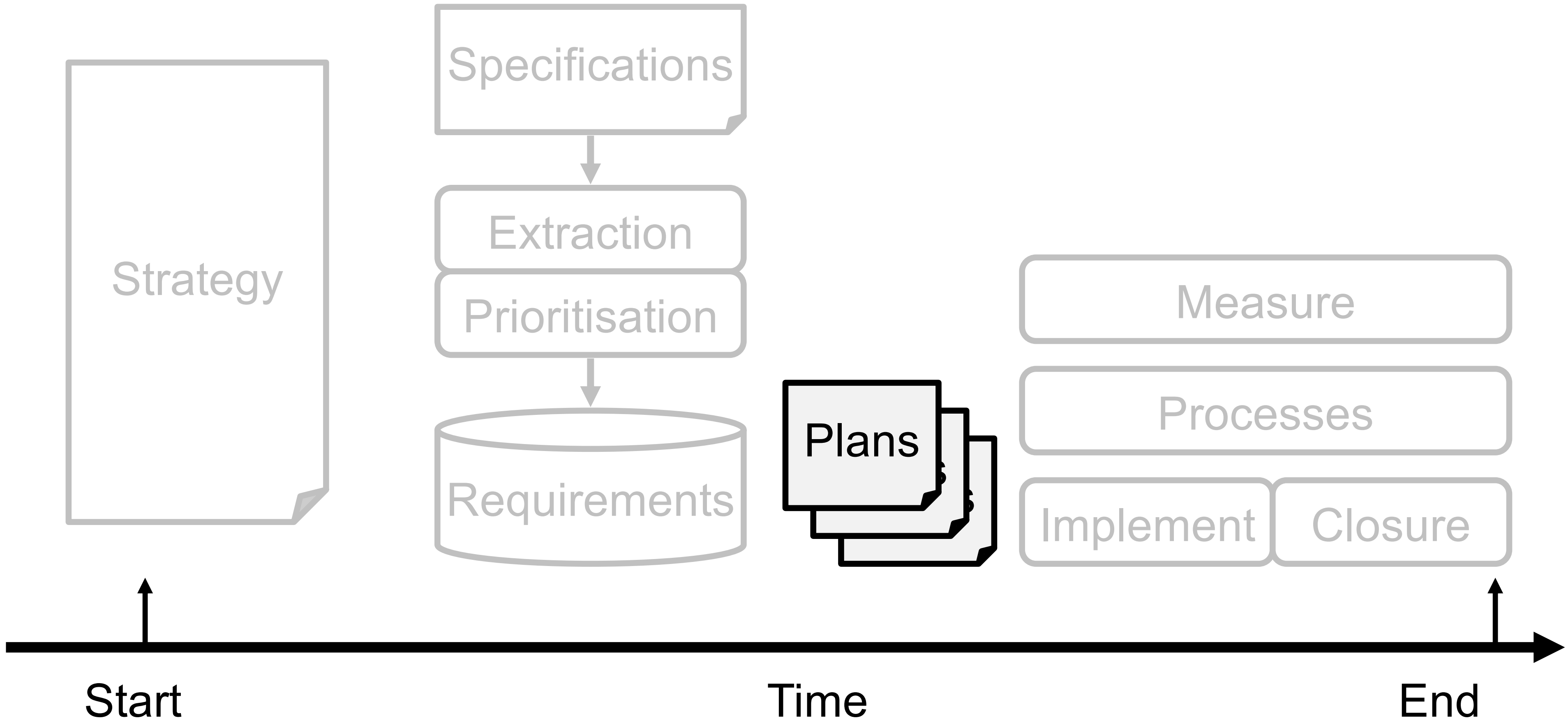
- "When the transfer is completed, the DMA engine will assert an interrupt (if enabled) or go to the idle state. If the auto restart bit (ARS) is set, it immediately restarts the operation"

- What if interrupts are enabled and ARS is set?
  - actually, the specification doesn't mention if the interrupt needs to be cleared to start a channel
  - experience says that this might be a reasonable thing to do

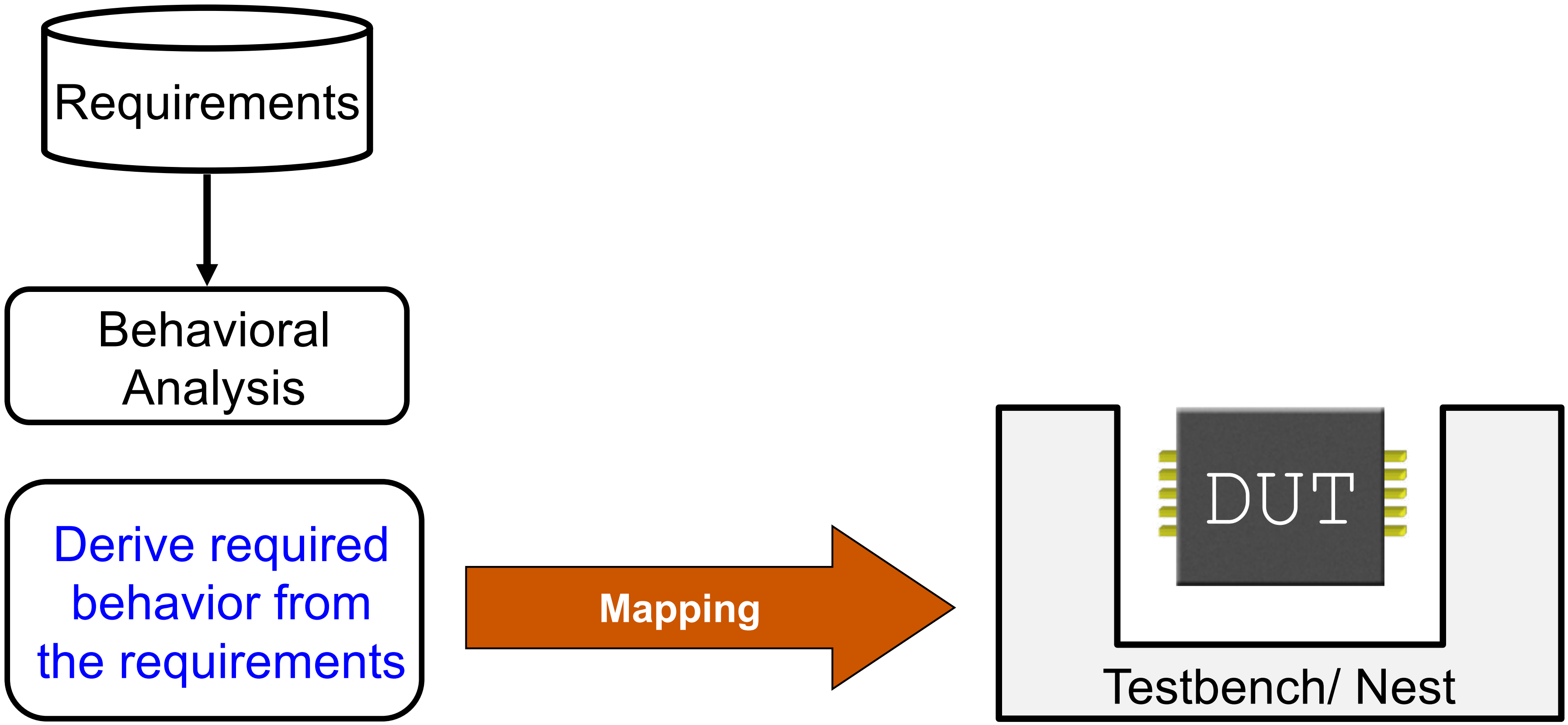
# Context: Plans

This is exactly how we will verify a particular design

The plan(s) contain the specific steps



# Plans: Verification Requirements



# Plans: What is a verification requirement?

- **A verification requirement is the systematic set of checks, coverage points and stimuli corresponding to a specific design requirement**

- **It's something about the design that you want to stimulate, check and cover before you'd be happy saying that the design has been verified**

- **Extracted before verification planning or implementation begins**

# Plans: Example

The serial receive block has four buffers. The block checks for the parity and validity of the data frame on the RXD input and then writes correct data into its buffers.

Requirement

Check that RXD data is being properly written into buffers taking into consideration the parity and validity of the data.

Check

| <u>Using</u> | | <u>Using</u> | | <u>Using</u> |
|---|---|---|---|---|
| Data = 5 bits,<br>Data = 6 bits,<br>Data = 7 bits,<br>Data = 8 bits | X | Parity = OFF,<br>Parity = EVEN,<br>Parity = ODD | X | Stop Bits = 1,<br>Stop Bits = 2 |

Functional Coverage

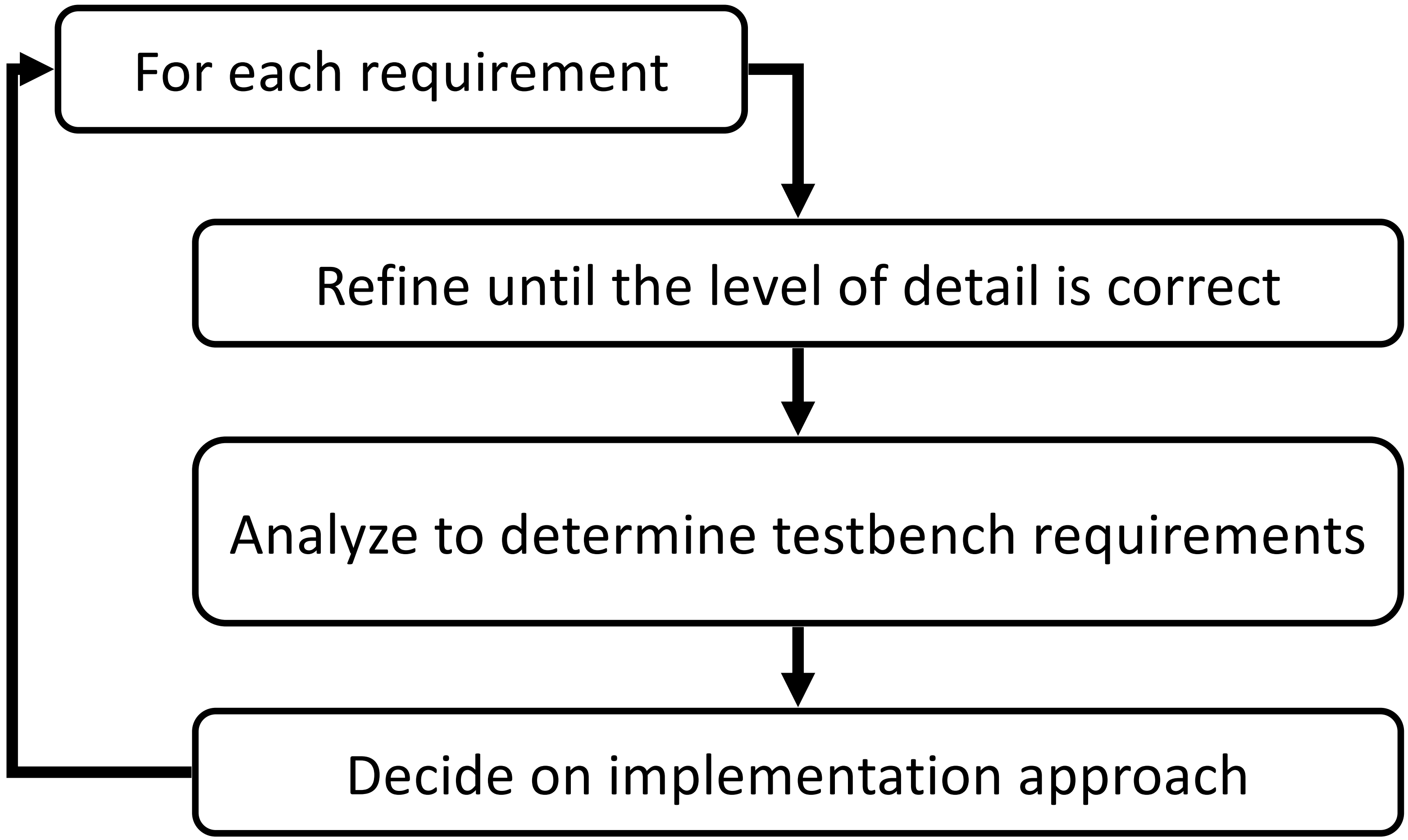
| Data[1:8] | Parity[1:3] | Stop Bits[1:2] | Stall[0:10] | Error[0:1] |
|---|---|---|---|---|

Constraints

# Plans: Behavioral Analysis

- To map each requirement to a checker or an assertion
  - and to understand what is required from the testbench to do so
- To map each requirement to functional coverage
  - and to understand what is required from the testbench to do so
- To identify extra testbenches, actors or components that are required
- To cross link everything to provide traceability from the functional specification to the verification code

# Plans: Behavioral Analysis



# Plans: Refining your Requirements

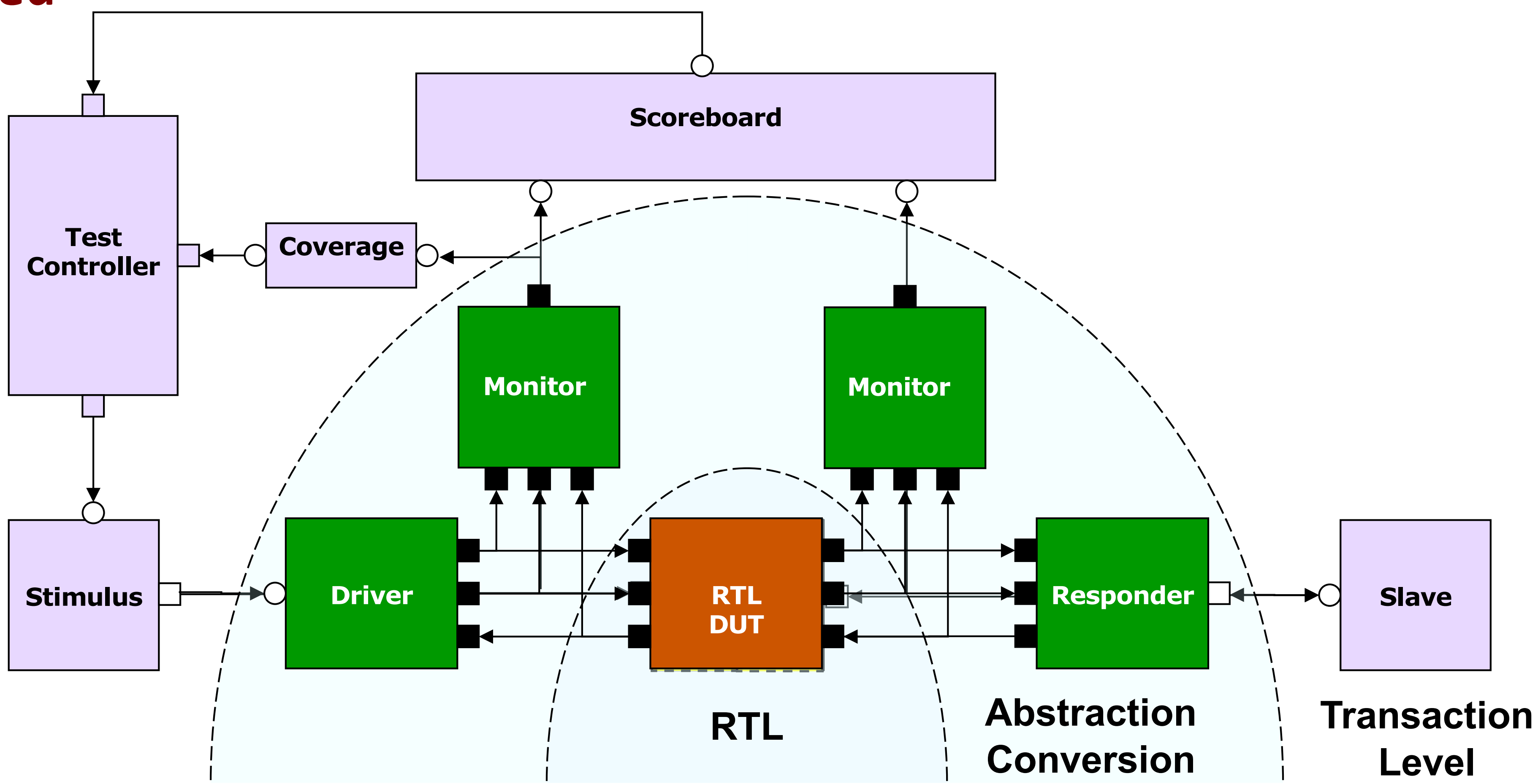
- **Get the level right**
  - too high and you don't really know what you're checking
  - too low and you're checking things that the specification didn't specify, or maintaining too many requirements
- **Reword to make explicit**
  - "Check that the arbiter selects the correct channel"
  - instead of "Check that the arbiter works"
  - or "channel arbitration"
- **Accuracy**
  - avoid cycle accurate checks
  - avoid verifying what you don't have to

# Plans: What level?

| | Requirement |
|---|---|
| ✗ | Check that the DMA works |
| ✗ | Check that data is transferred correctly by the DMA |
| ✓ | Check that data is fetched from the correct source address |
| ✓ | Check that data is sent to the correct destination address |
| ✓ | Check that the sent data is equal to the fetched data |
| ✗ | Check that data is fetched from the specified source address (for the first transfer in a block) and then from incremented addresses for subsequent transfers in a block when INC_SRC is 1 |

# Plans: Verification Requirements to Testbench Arch.

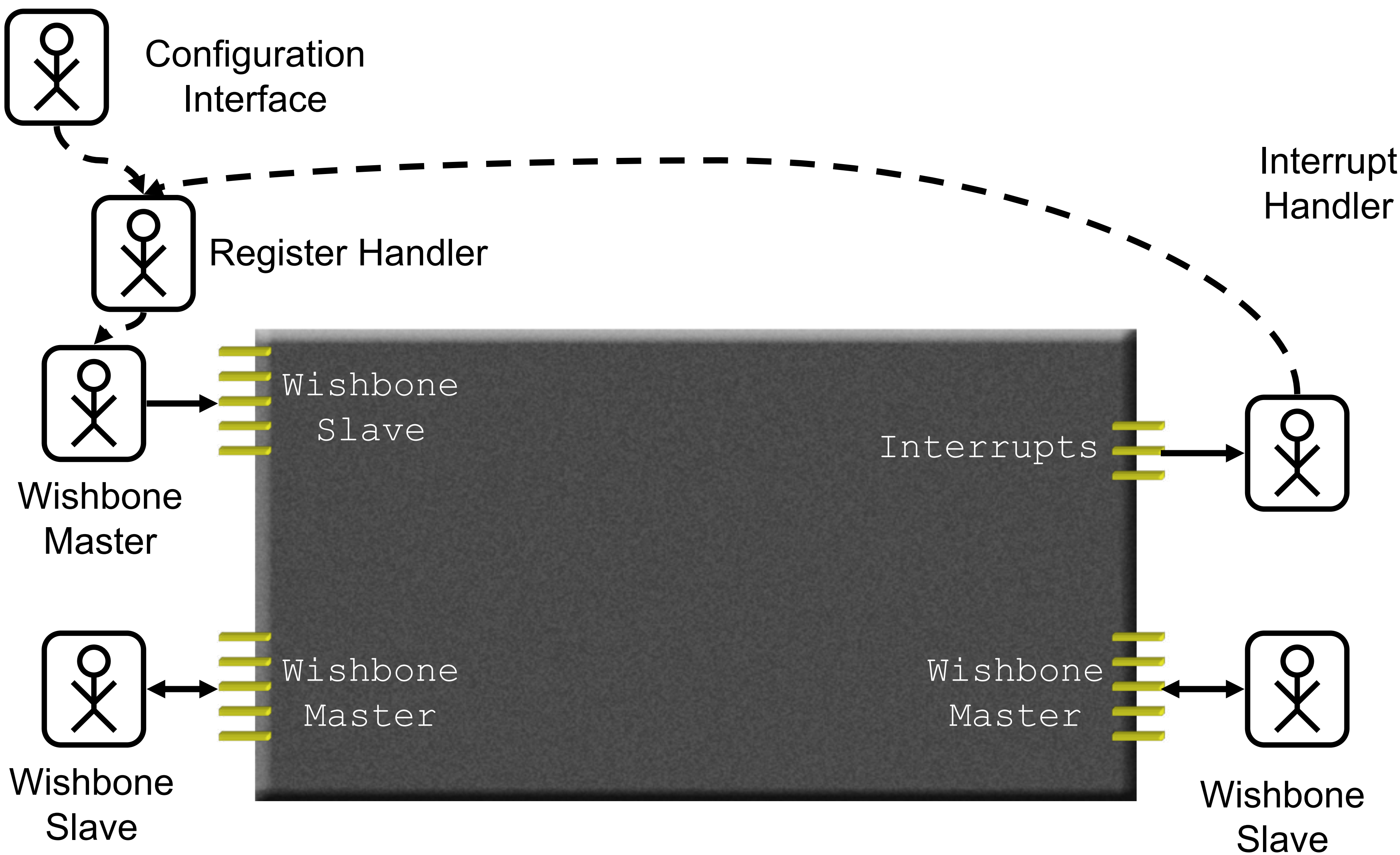
- To determine a general structure for a testbench that can exercise this DUT
- To build a list of verification components and actors that you will need



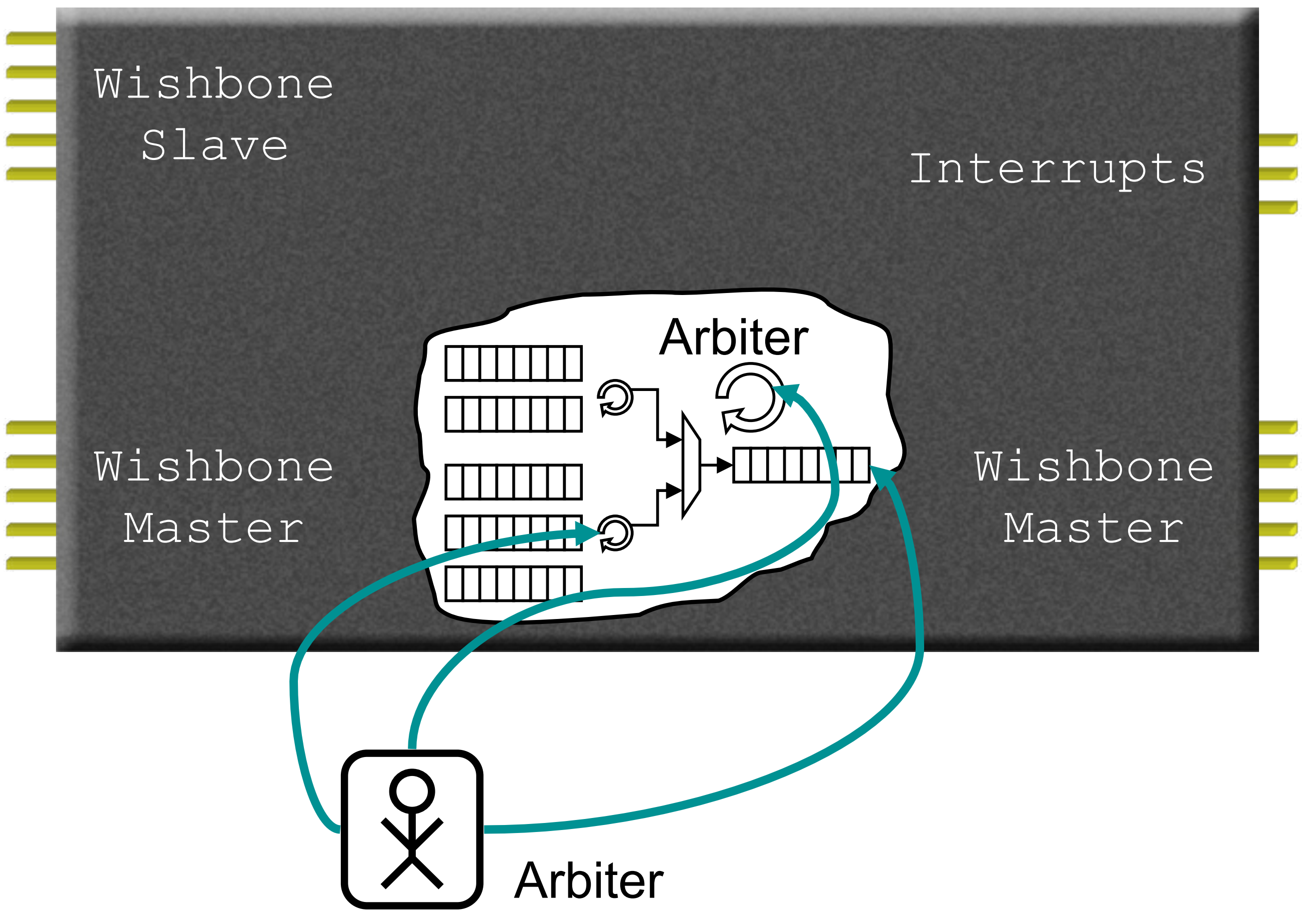
# Plans: Finding Actors

- Start with a block diagram of the DUT in a system
  - the DUT could be the entire design, a cluster or a single block
- You will need an “External” Actor to interact with each DUT interface
- If you plan to just do black box verification, then these are probably all the actors you need
- If you are doing white or grey box testing, then you will need actors for interfaces within the DUT

# Plans: External Actors



# Plans: Internal Actors



# Plans: TLM Data Structures

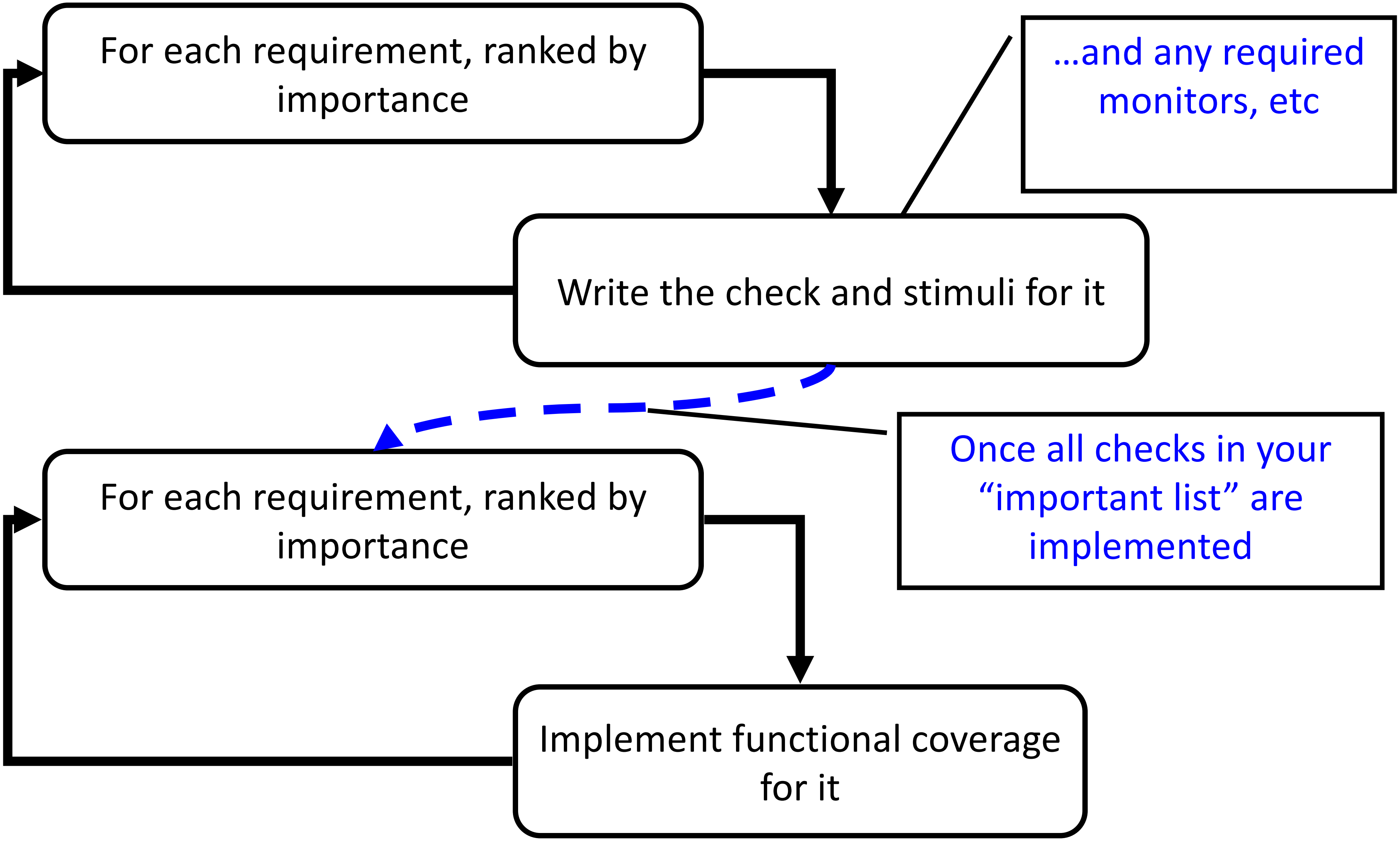
- **For each actor, think about the transactions that will pass between**
  - it and the DUT
  - it and other actors
  - its internal components

| Bus Transaction | Register Transaction | Configuration Transaction | Monitor Transaction |
|---|---|---|---|
| `Address`<br>`Data`<br>`Burst Type`<br>`Size` | `Address[]`<br>`Data[]` | `DMA mode`<br>`Channel Number`<br>`Priority`<br>`Src Address`<br>`Dst Address`<br>`Size` | `Status`<br>`Channel Number`<br>`Priority`<br>`#Bytes so far`<br>`#Bytes to go`<br>`Data[]` |

# Plans: Mapping Deliverables

- You should now have a good idea of the Actors you need
  - and therefore the sub-components per actor
  - and the transactions you need
- …and a rough idea about the topology of the testbench
- The detail is missing though, and for that, you need to map results of behavioural analysis onto actors

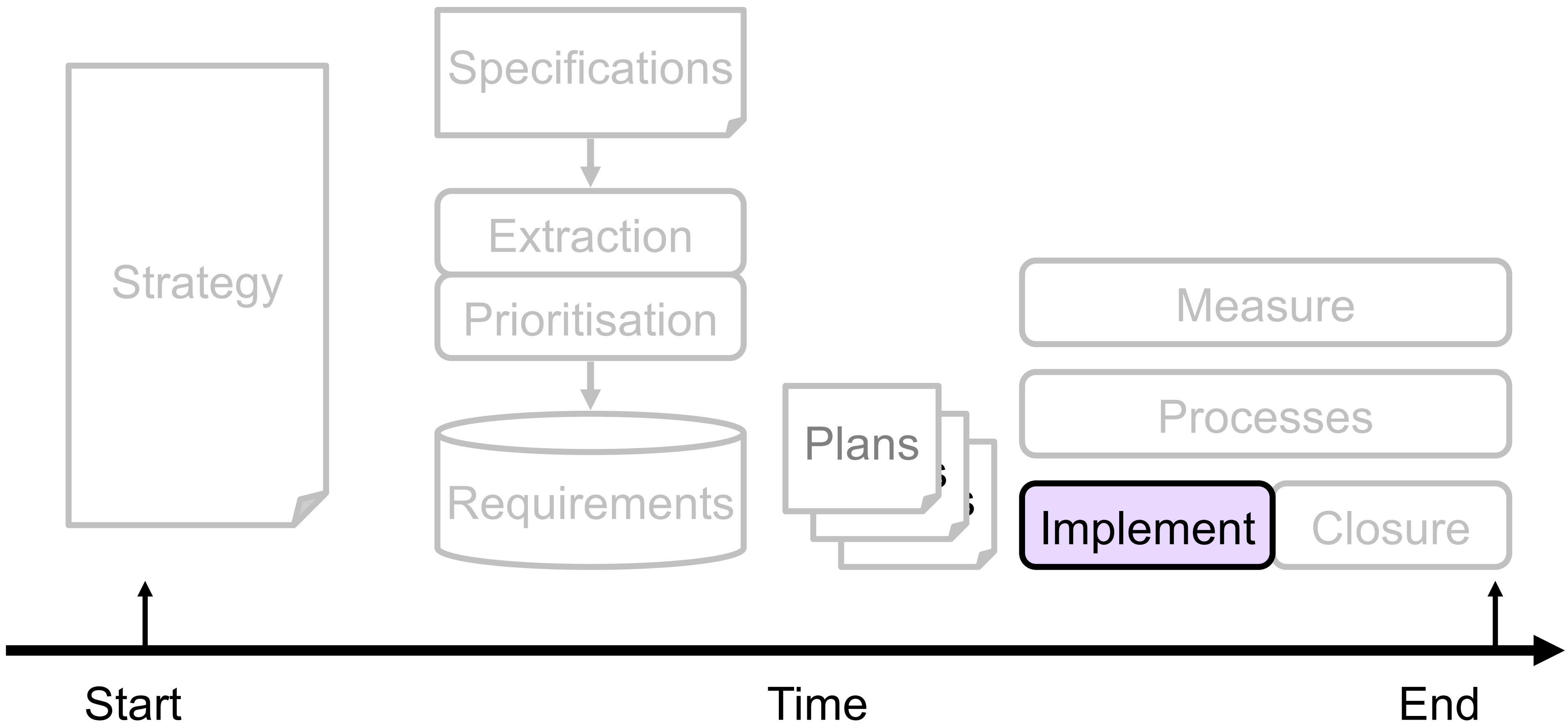
# Plans: Implementation



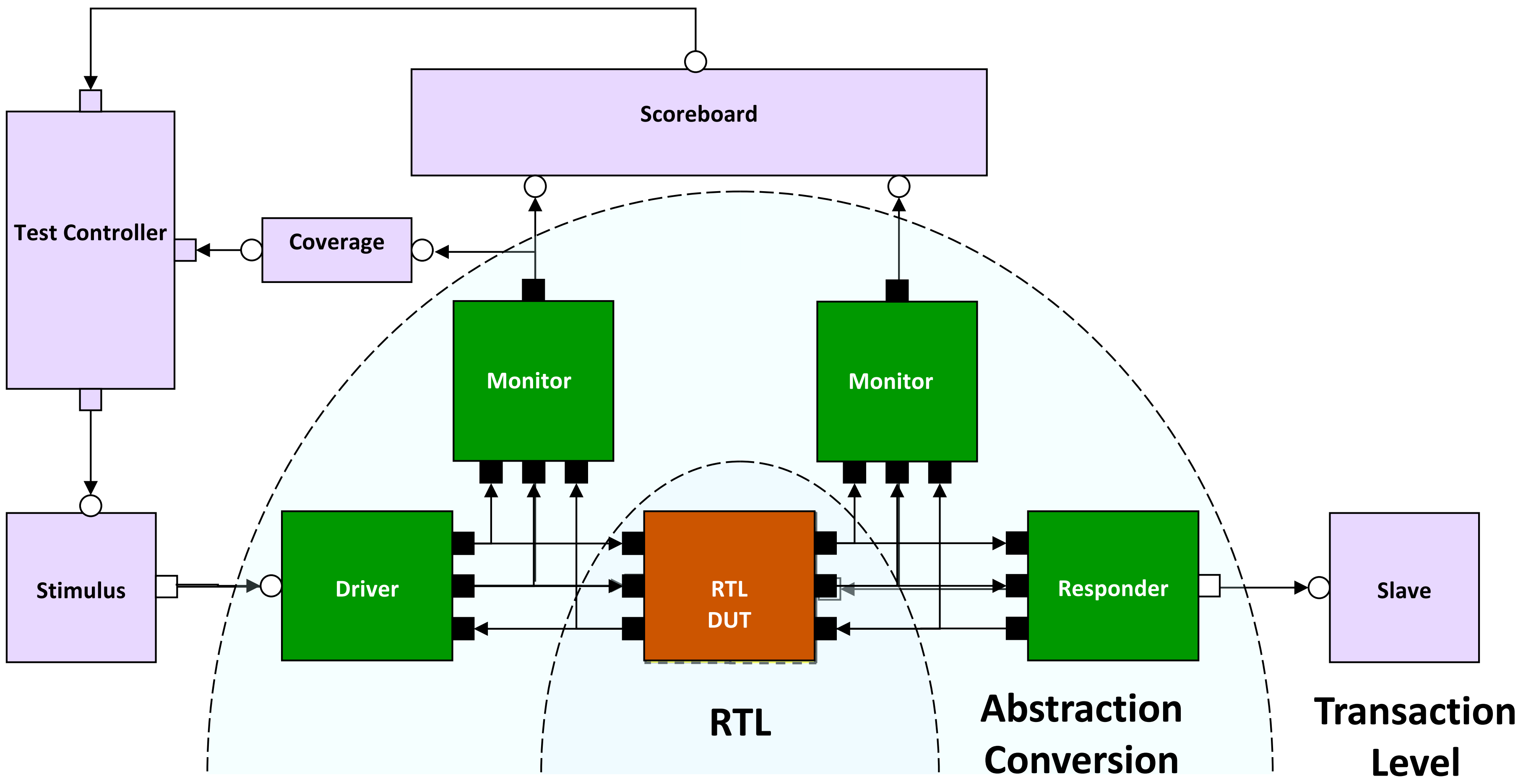
# Context: Implementation

This is the testbench implementation we will use to verify the design

The implementation provides the means by which we will stimulate, check and cover the design

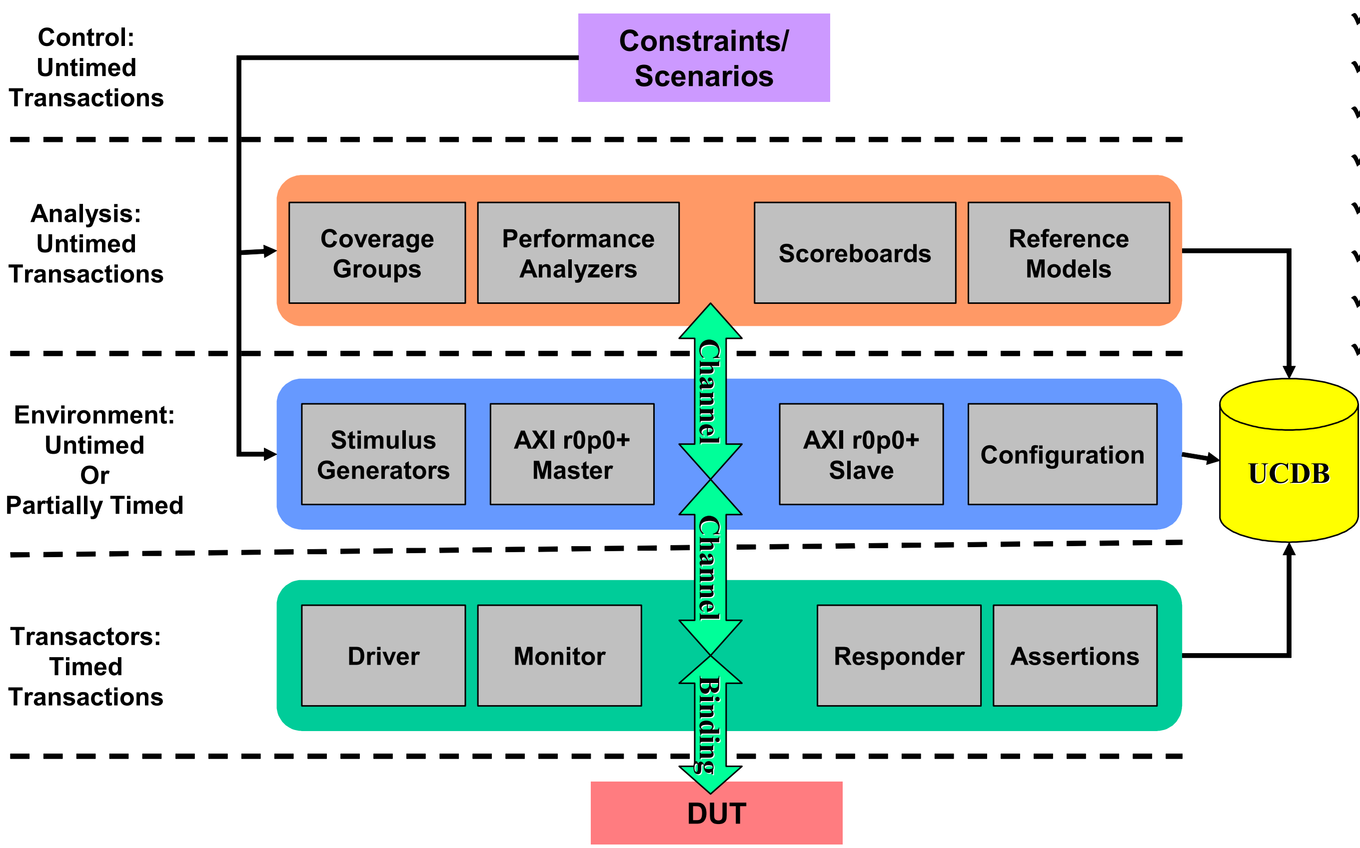


# Implement: Testbench Architecture



# Implement: HVL Base Classes

## UVM Layering



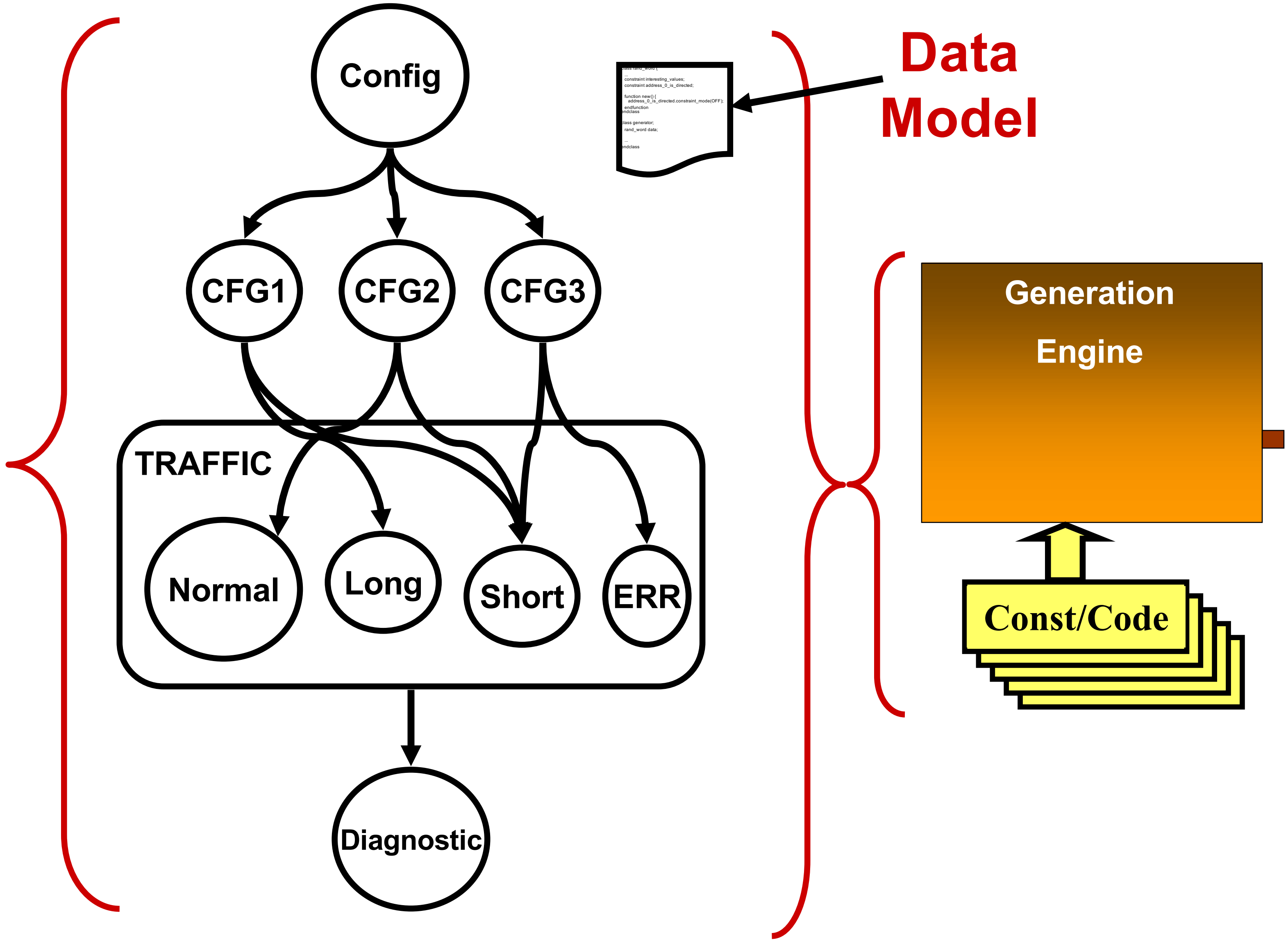

## UVM Base Classes

- ✓ Base class for messaging
- ✓ Base class for basic transactions
- ✓ Base class for channels
- ✓ Base class for burst transactions
- ✓ Base class for Transactors
- ✓ Base class for drivers
- ✓ Base class for monitors
- ✓ Base class for memory management
- ✓ Base class for scenario generation

# Implement: Random Generation Engine

**FLOW MODEL:**

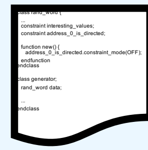
**Captures the flow of the chip**



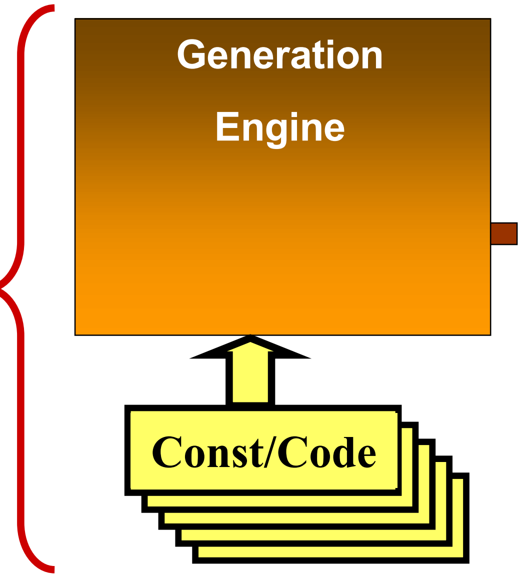
# Implement: Random Generation Engine

Control Model

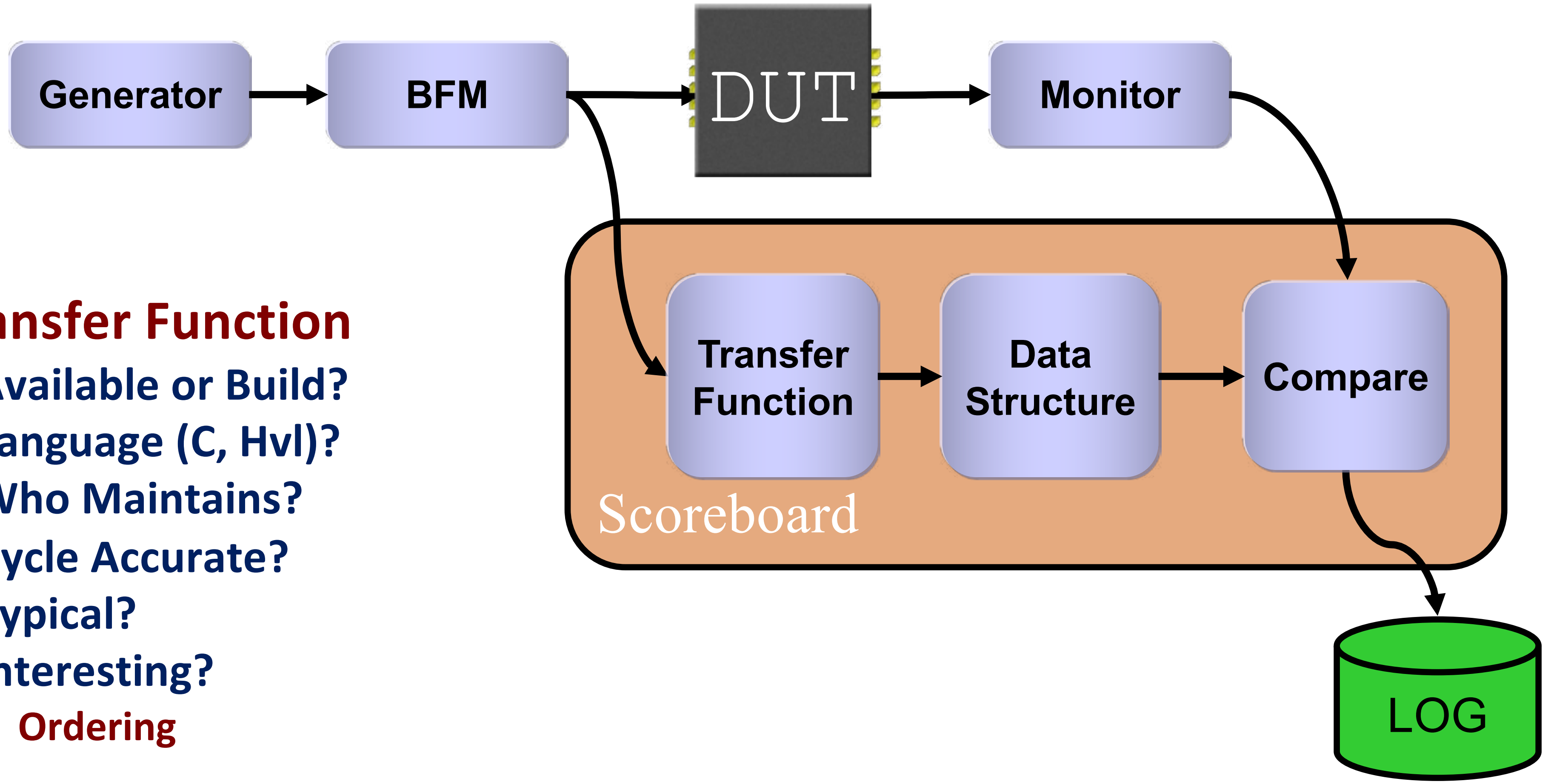
```
module GenEng ();
initial
  forever begin : loop
    randsequence (main)
      main : cfg, traffic, diagnostic;
      cfg : {
        randcase
          cfg1: …;
          cfg2: …;
          cfg3: …;
           …
        endcase
      }; // cfg
      traffic : {
        fork
          …
        join none
      }; // traffic
      diagnostic : {…};
    endsequence
  end // forever
end /init
endmodule
```



Data Model



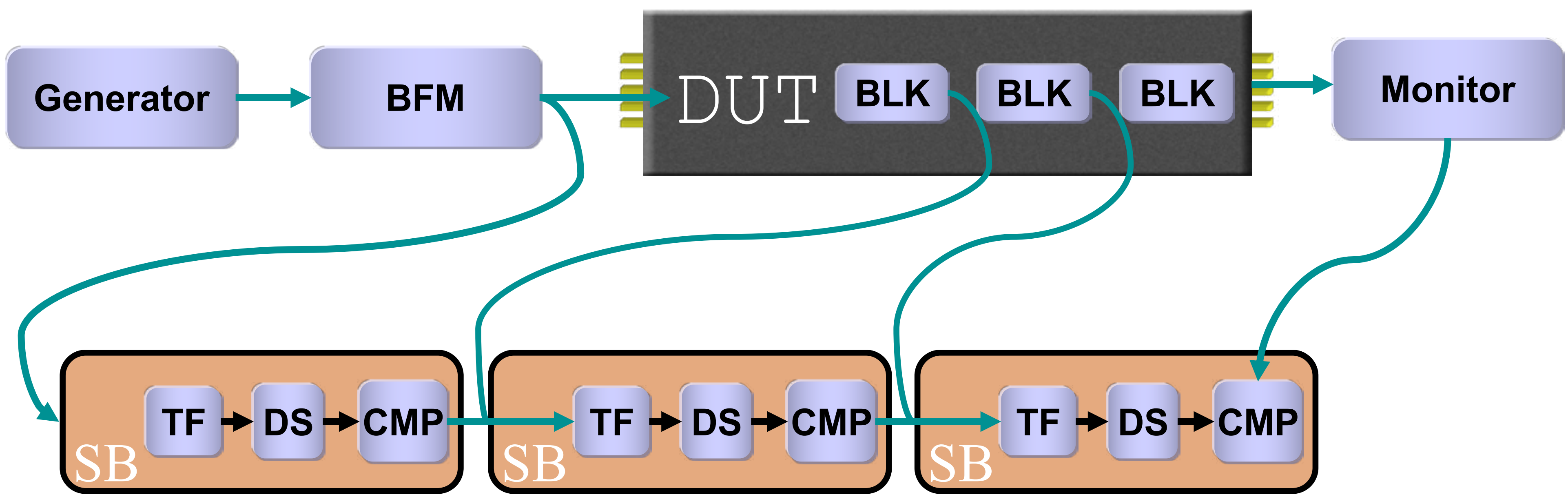
# Implement: Scoreboard



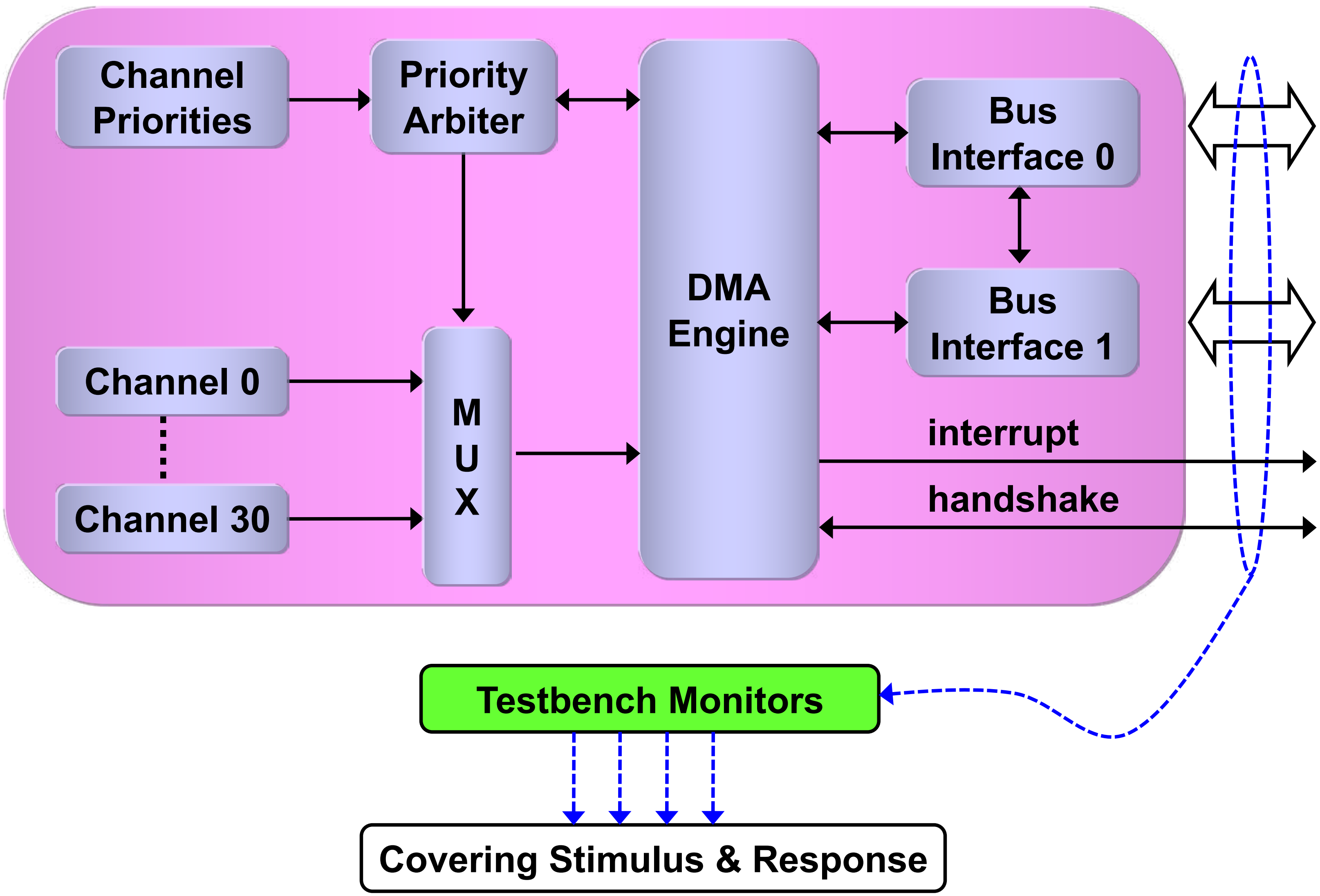

- **Transfer Function**
  - Available or Build?
  - Language (C, Hvl)?
  - Who Maintains?
  - Cycle Accurate?
  - Typical?
  - Interesting?
    - Ordering

# Implement: Scoreboard Architecture

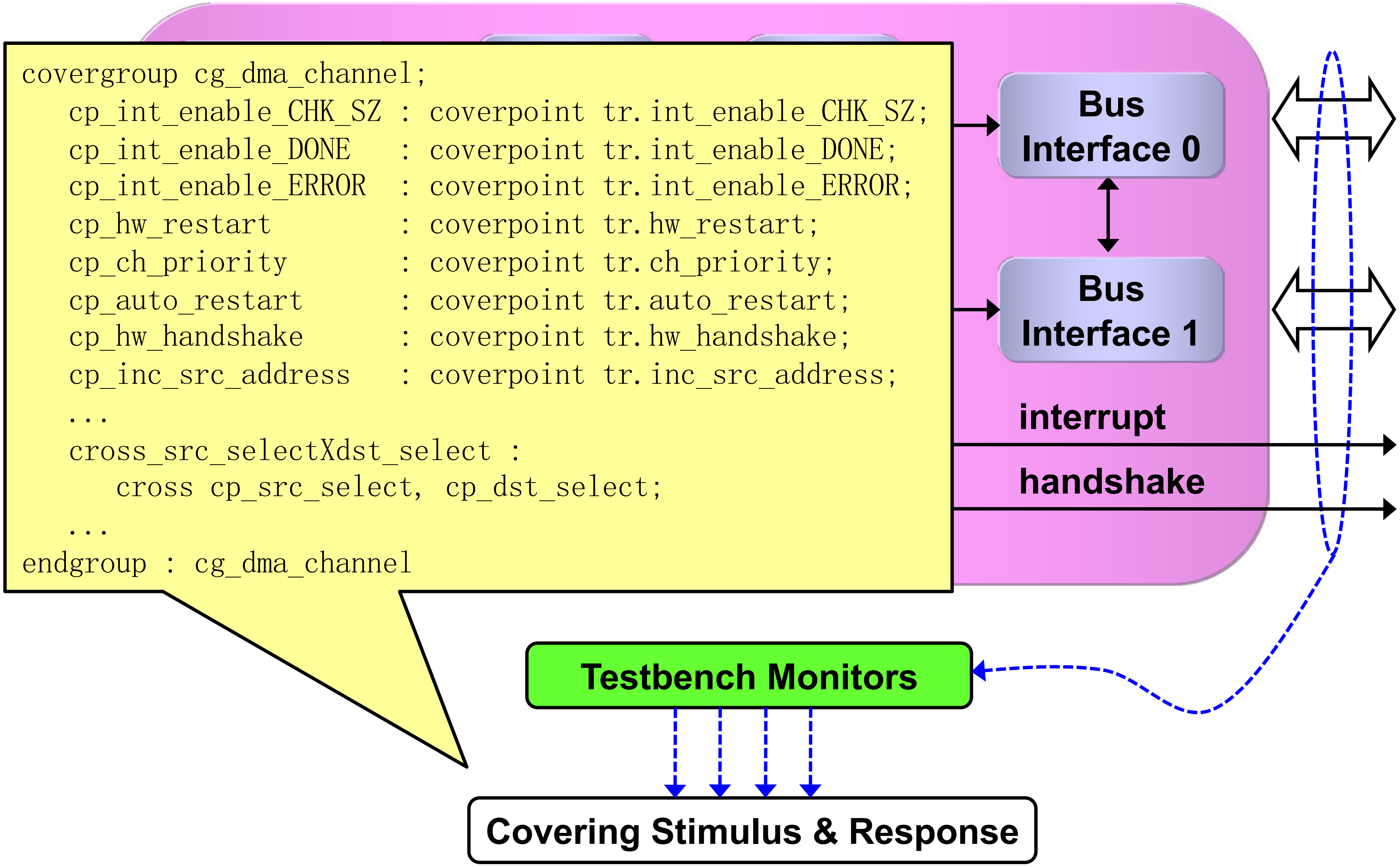
- One big scoreboard that is a model of the entire chip
- Divide up into smaller, easier scoreboards daisy-chained together



# Implement: Coverage

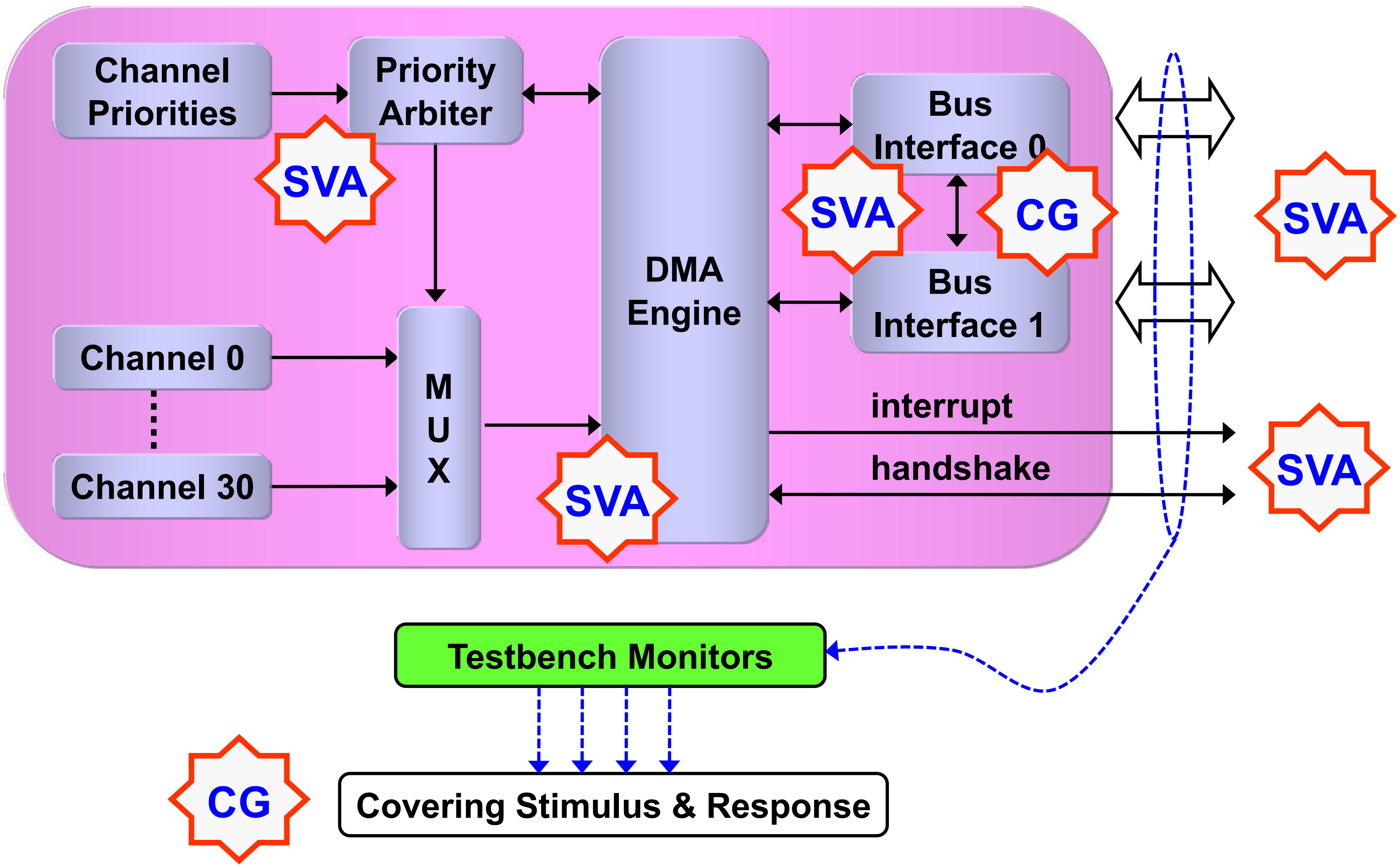


# Implement: Coverage




```
covergroup cg_dma_channel;
   cp_int_enable_CHK_SZ : coverpoint tr.int_enable_CHK_SZ;
   cp_int_enable_DONE   : coverpoint tr.int_enable_DONE;
   cp_int_enable_ERROR  : coverpoint tr.int_enable_ERROR;
   cp_hw_restart        : coverpoint tr.hw_restart;
   cp_ch_priority       : coverpoint tr.ch_priority;
   cp_auto_restart      : coverpoint tr.auto_restart;
   cp_hw_handshake      : coverpoint tr.hw_handshake;
   cp_inc_src_address   : coverpoint tr.inc_src_address;
   ...
   cross_src_selectXdst_select :
      cross cp_src_select, cp_dst_select;
   ...
endgroup : cg_dma_channel
```

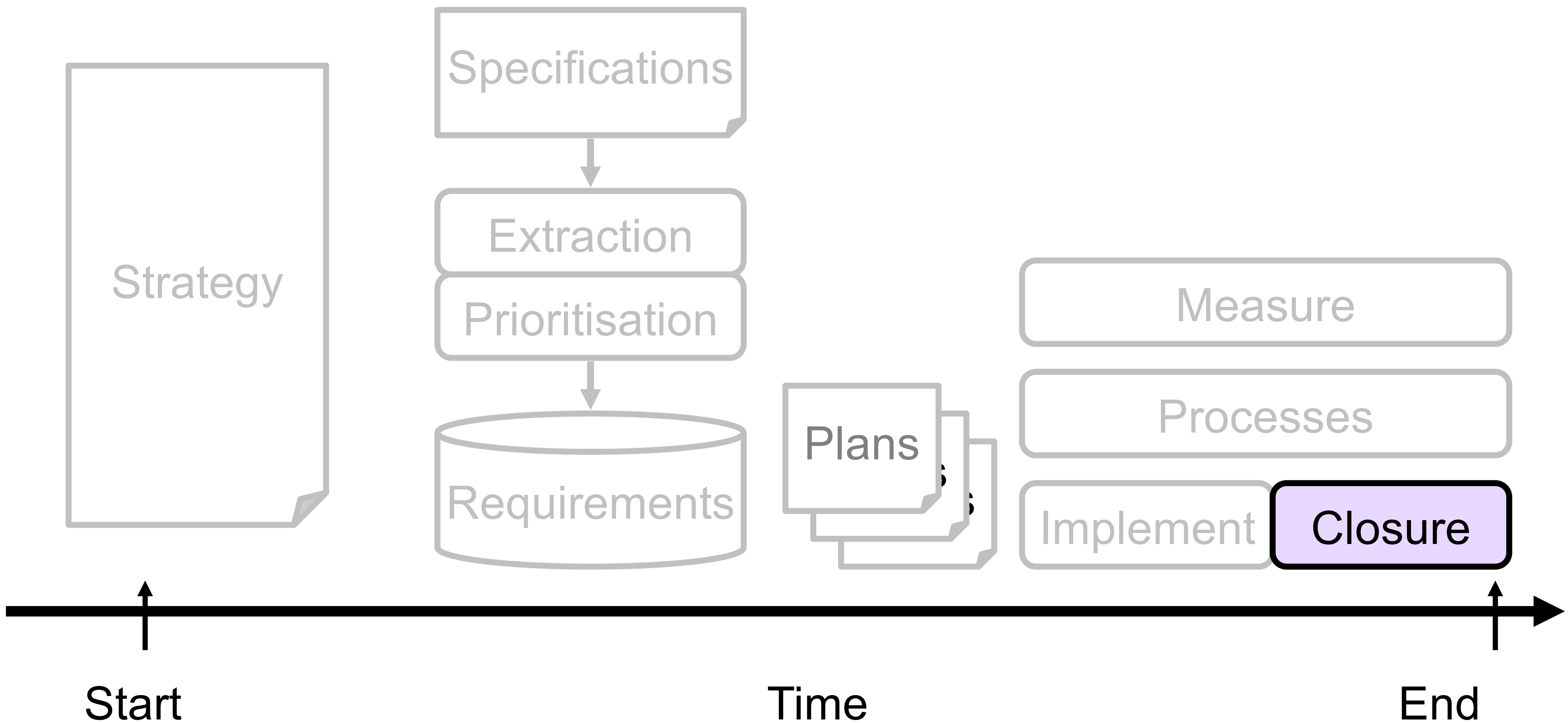
# Implement: Coverage Model



# Context: Closure

This is the mechanics of how we will verify the design

The closure process ensures we move toward tape-out in a methodical way taking into consideration various metrics.



# Closure: Verification Planning Metrics

- **Requirements: DRs & VRs**
  - Status: Written, Refined, Reviewed, Signed off, Implemented, Covered
  - Updated: Changed, Re-Prioritized
  - Stuck: Outstanding issues
  - Testcases: Total # Directed & Random (# Seeds) Written/Passing/Failing
  - Assertion Density

# Closure: Testbench Development Metrics

- **Lines of Code**
  - **Estimated/Actual SLOC** (S-Curve) **vs. Time/Phases/Release**
  - **% Change SLOC** (HDL vs. HVL) **vs. Time/Phases/Release**
  - **HVL Bugs vs. SLOC vs Time**
  - **HVL Mean Time Between Failure (MTBF)**
  - **Total Cycles per Regression vs. Time**
  - **Cycles per Second** (CPS) **vs. Release**
  - **Ratio of SLOC to Comment Lines of Code (CLOC)**
  - **Kind of code: Base or Extended, VI or testcase**

# Closure: Bug Discovery Metrics

- **Table**
  - **Total Test Cases/Seeds**
  - **% testcases Pass/Fail**
- **Graph**
  - **HDL Bugs vs. Time**
    - **Total**
    - **By Unit, Subsystem and Chip**
  - **Bug Pipeline vs. Time**
  - **Average Bug Resolution Time** (Open to Close)
  - **HDL Mean Time Between Failure** (MTBF)
- **Bug Schema** (Historical)
  - **Type** (e.g. Spec, HVL, HDL, Firmware, Software, tool, library, etc.)
  - **Source** (e.g. block, sub-system, chip, prototype, samples, etc.)
  - **Discovery Method** (e.g. visual inspection, assertions, reference model, etc.)
  - **Stimulation Method** (e.g. directed test case, constraint random, irritator, etc.)
  - **Severity**

# Closure: Coverage Population Metrics

- **Functional Coverage**
  - Total Coverage Bins vs. Implemented vs. Populated
  - Coverage Groups/Bins vs. Time/Release
  - Coverage Properties vs. Time/Release
  - Merged By Unit, Subsystem, Chip

- **Structural Coverage**
  - Code/Line Coverage (Statement, Branch, Condition, Path, Toggle)
  - FSM Coverage

# Tying it all together

- **Design Specification**
  - 32-bit general purpose scalar processor
  - 5 stage pipeline – can experience stalls in any stage
  - 16 x 32-bit general purpose registers
  - Register forwarding
  - Op-codes
    - Register based addition: add src1, src2, dst
    - Register based addition with saturation: sadd src1, src2, dst
    - nop
  - What would be a reasonable data model?
  - What would be a reasonable control model?
  - What would be a reasonable correctness model?
  - What would be a reasonable coverage model?