Overwiev

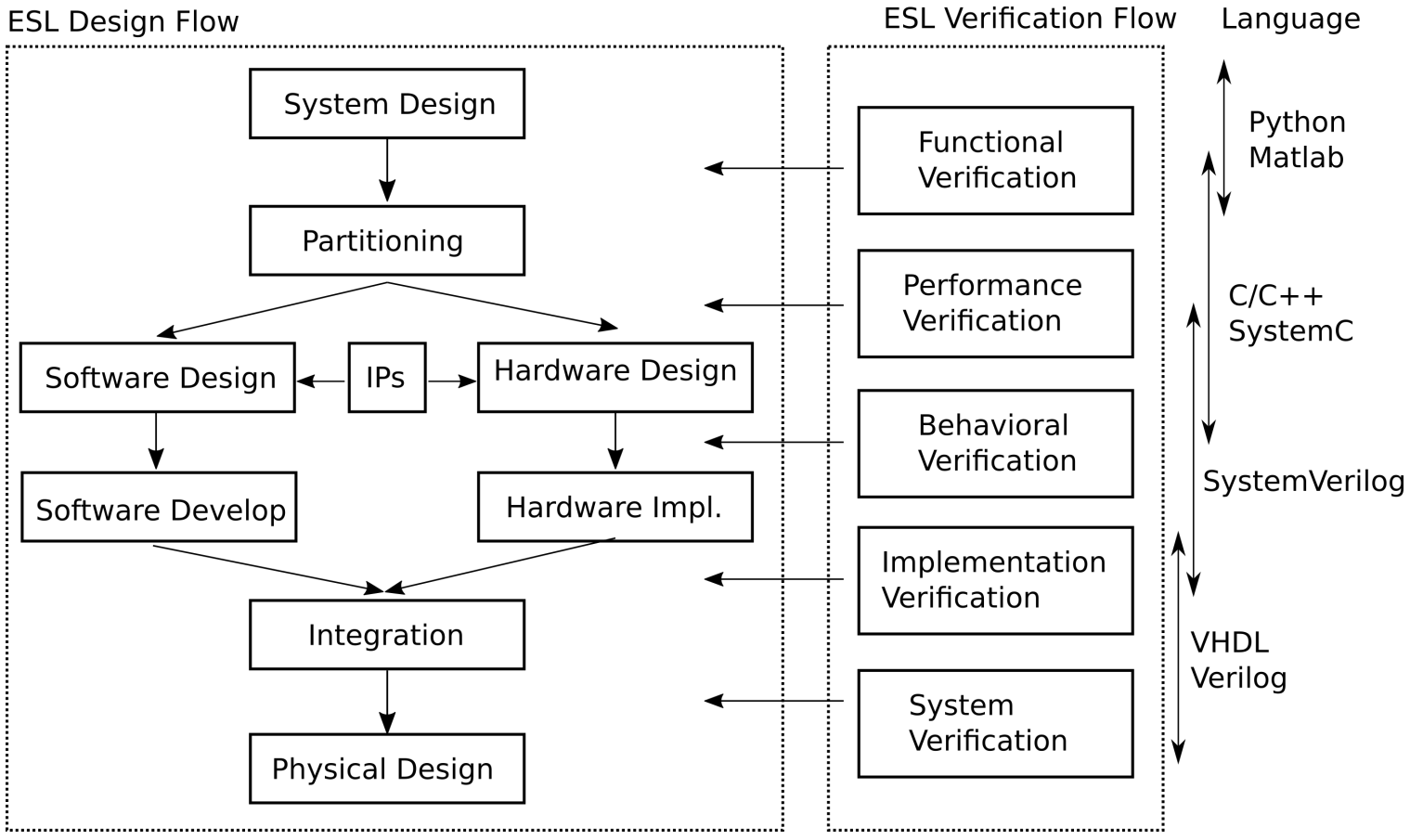
Post-Partitioning Verification I

- Verification means making sure the **implementation is correct**, ensuring that it conforms to the **requirements** recorded in the **specification**.
- Post-partitioning verification is introduced into the ESL design flow to **discover design errors** that are introduced **before** RTL and software implementation.

Facets of Verification I

- A facet of verification is a **particular type** of verification that needs to be performed and includes such things as **functionality, architecture, performance, timing, and implementation**.
- Although it is possible to do all of this on an **RTL model**, it has already been stated that this is both **inefficient** and in many cases **ineffective**.

Facets of Verification II



- **Assuming** that we have a **models of the system**, the verification will be performed at **every level**.

Facets of Verification III

- The **failure** to adopt an ESL verification methodology mean that for most companies, functional verification gets performed **later** in the flow.
- Although **abstract HW/SW co-simulation** has been available for many years, **few people** do performance verification at **abstract-behavioral** level because of a **lack of models**, and it is also deferred until later in the flow.
- **RTL** is where most companies **start the verification** process today. This includes system-level functional verification, performance verification, and implementation verification. **These models** contain **more detail** than is necessary to perform these types of verification and result in a large amount of **wasted effort**.

Facets of Verification IV

- To **avoid** this **verification inefficiency**, it is important to understand the **different aspects of verification** at each stage of the design process. To **verify behavior**, we **do not need** to be concerned about **every clock**, just as the verification of the RTL implementation does not need detailed timing.
- The verification process begins with **verification planning** and ends with **failure and coverage analysis**.

Overwiev

Verification Planning I

- The purpose of **verification planning** is to formulate a **strategy** and associated tactics to verify the hardware and software components of the **post-partitioned design**.
- The outcome of the process is a **verification plan**. The verification plan must answer two questions: first, "What is the **scope** of the verification problem?" and second, "What is the **solution** to the verification problem?"
- Verification plan outline:
  1. Introduction
  2. Functional Requirements
     1. Functional Interfaces
     2. Core Features
  3. Design Requirements
     1. Design Interfaces
     2. Design Cores

Verification Planning II

4. Verification Views
5. Verification Environment Design
    1. Coverage
    2. Checkers
    3. Stimuli
    4. Monitors

- **Introduction** should briefly **introduce the DUV** and discuss the **general verification strategy** to be used.
- The next two sections of the plan partition the **opaque box** and **clear box** DUV features. "Opaque box"– sometimes referred to as "black box"–refers to observing the DUV solely from its **external I/O interface**, whereas "clear box"–also known as "white box"–refers to observing **the internals** of the DUV.

Verification Planning III

- The **opaque** box features of the design are recorded in the **"Functional Requirements"** section, interface features in **"Functional Interfaces,"** and core features in the section of the same name.
- **Clear** box requirements are recorded in the **"Design Requirements"** section, again partitioned into interface and core features.
- The **"Verification Views"** section groups time-based and function-based references to other plan sections with their own goals.
- Finally, **"Verification Environment Design,"** with its four subsections–"Coverage," "Checkers," "Stimuli," and "Monitors"–is the **functional specification** for the **verification environment**, both its dynamic and static aspects.

What Is the Scope of the Verification Problem? I

- To discover the magnitude and particulars of the verification problem we face, we need to discover the **intended behavior of the design** - functional and design specification.
- We expose recorded design intent through **specification analysis**.

Specification Analysis I

- Through specification analysis, we discover the **feature set** of the design and its **corner cases**.
- A **corner case** is one or more data values or sequential events that, in combination, lead to a **substantial change** in design behavior.
- There are two approaches to specification analysis: **bottom-up analysis and top-down analysis**. These terms refer to the two ends of an abstraction spectrum, its bottom and top.
- The **bottom** of the abstraction spectrum contains the **most detail** and the **least ambiguity**. It is also requires **the most information to describe**: written text or illustrations and diagrams.
- The **top** of the abstraction spectrum has the **least amount** of detail and the **most ambiguity**. However, it requires **the least amount of information** to convey.

Bottom-up Specification Analysis I

- A relatively **small specification**, perhaps consisting of 20 pages or less, may be reviewed chapter by chapter, paragraph by paragraph, and sentence by sentence. This is referred to as **bottom-up specification analysis**
- As we examine each element of the specification, we identify the **features and attributes** described therein.
- A **feature** is a **behavioral requirement** with associated **attributes** that contribute to its definition..
- As we walk through the specification, each **feature name** is **recorded** in the appropriate section of the **verification plan**, along with its description.
- The **feature description** not only explains the feature in a sentence or two, but it serves as the **functional specification for the coverage model** that will quantify the feature.

Top-down Specification Analysis I

- Top-down analysis **abstraction specification** through the human mind. **A small team** of engineers familiar with the specification and managers concerned with project schedule and resources **is assembled** and tasked with contributing to the **verification plan**.
- **Features** and their associated **attributes**, just as with bottom-up planning, are recorded in the nascent **verification plan**
- The small team consists of
  - The system architect
  - RTL designer
  - Software engineer
  - Manager concerned with labor and schedule
  - Verification engineer
- Whether bottom-up or top-down analysis is used to extract the design features, the description of each feature is distilled into a **semantic description**. That description is the first step in the top-level design of the **feature's associated coverage model**.

Coverage Model Top-Level Design I

- Just as any model is an approximation or simplified example of a reality, a **coverage model** is an **approximation** of a subset of the **intended behavior of the DUV**.
- A coverage model is an approximate description of part of the behavior of a hardware or software design, whose purpose is to **quantify some region of design behavior** in order to know what **has and has not been exercised**.
- During **coverage model design**, as each **feature** of the design and its **attributes** is **identified and recorded** in its respective section of the verification plan, the **size and fidelity** of the coverage model to quantify its scope are chosen.
- The **size** of the model is defined by the **number of points** it contains, where each **point** is itself defined by the **values of a set of attributes**.

Coverage Model Top-Level Design II

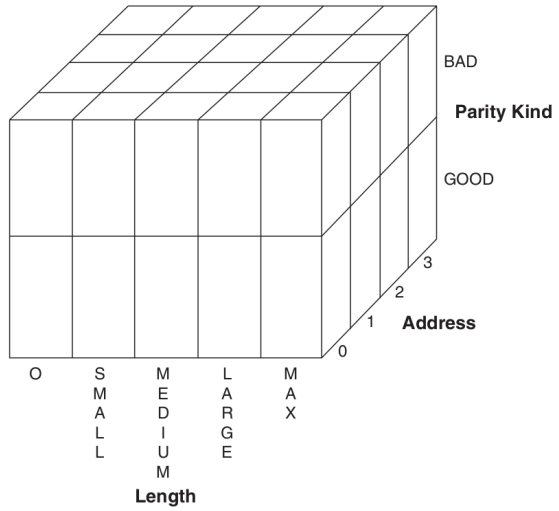
- Model is usually refined to a model of greater and **greater fidelity as the project proceeds**, time and resources permitting.
- The **first** step of coverage model design is **top-level design**:
  - Semantic description
  - Attributes
  - Attribute relationships
- **The semantic description** captures the **essence of a feature** in **one or two sentences** and serves as the functional specification for the coverage model. It should concisely describe the purpose of the feature and what influences its behavior.

Coverage Model Top-Level Design III

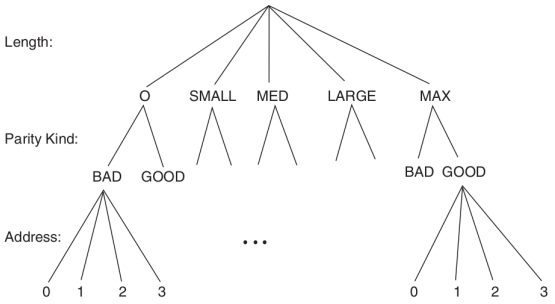
- **The attributes** – many identified during specification analysis – that contribute to the definition of the feature are **recorded**, along with their relevant values and sampling times. The values of an attribute fall into two overlapping sets: the **physical space and valid space**. The physical space contains all of the values that may be encoded by the storage element containing the attribute value. The valid space contains those values intended to be used by the DUV.
- For each attribute, we need to choose a **sampling time**, a designated time or event – periodic or aperiodic – to capture the current value of the attribute.
- The second time associated with a coverage model is the **correlation time**. This is the time at which the most recently **sampled values** of the attributes are stored as a set in a **coverage database**.
- Last, the **relationship among the attributes** – matrix, hierarchical, or hybrid structural – is chosen for subsequent refinement.

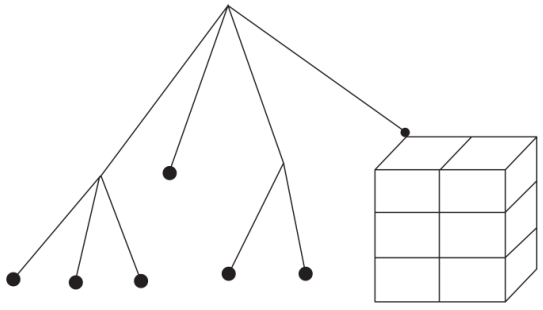
Coverage Model Top-Level Design IV

- In a **matrix coverage model**, each **attribute defines a dimension** of the model and each attribute value defines a position on an axis.
- In a **hierarchical coverage model**, each **attribute defines a level in the hierarchy** of the model and each attribute value defines a branch in the inverted tree.
- A **hybrid structural coverage model** is a **composition** of the matrix and hierarchical models

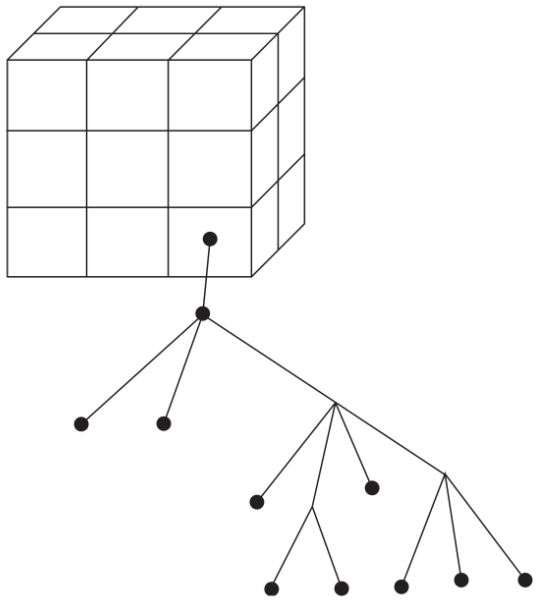


Coverage Model Top-Level Design V





Coverage Model Top-Level Design VI



Coverage Model Detailed Design I

- Coverage model detailed design is responsible for **mapping** the top-level coverage model design **to the verification environment**.
  1. **What** must be **sampled** for each attribute value?
  2. **Where** in the verification environment or DUV should the value be **sampled**?
  3. **When** should the data be **sampled and correlated**?
- The "**what**" in the first question refers to a **DUV register, signal, or variable**.
- The "**where**" in the second question refers to which **architectural element of the verification environment** should be responsible for **sampling** the attribute value.
- The "**when**" in the third question above refers to a DUV- or verification environment- **relative time** that defines when attribute **sampling and correlation** should be performed.

Hybrid Metric Coverage Models I

- Coverage models may use **multiple metric sources** such as:
  - specification and implementation metrics
  - implicit and explicitmetrics
  - metrics from the data and temporal domains
  - data sources such as simulation and formal analysis
- An **implicit metric is inherent in the source** from which the metric is derived.
- **An explicit metric is one chosen by the engineer** that is not implicit in the metric source.
- A **specification metric** is one derived from a **natural language specification**, such as a functional or design specification.
- An **implementation metric** is a measure selected from the **design-under-verification**

Hybrid Metric Coverage Models II

- A **data metric** is the **value** of a storage element, variable, signal, or parameter in the design.
- A **temporal metric** is the value of an absolute **point in time**.
- Verification **method** is either **dynamic (simulation) or static (formal)**.

Hybrid Metric Coverage Models III

Metric Kind	Metric Source	Metric Domain	Verification Method	Example
implicit	specification	data	(n/a)	Control register specification paragraph number five.
implicit	specification	temporal	(n/a)	Instruction issue rate specification sentence number three.
implicit	implementation	data	dynamic	Boolean expression of a C++ “if” statement.
implicit	implementation	data	static	Third sequential condition of an assertion.
implicit	implementation	temporal	dynamic	Cycles elapsed from VHDL statement three to statement seven.
implicit	implementation	temporal	static	Range of clocks elapsed evaluating a concurrent assertion.
explicit	specification	data	(n/a)	Specified audio output amplitude range.
explicit	specification	temporal	(n/a)	Specified range of R/F signal frequency jitter.
explicit	implementation	data	dynamic	Value of packet in top of queue.
explicit	implementation	data	static	Compression value referenced by an immediate assertion.
explicit	implementation	temporal	dynamic	Clocks elapsed during a single “do...while” loop iteration.
explicit	implementation	temporal	static	Cycles specified by a particular property.

What Is the Solution to the Verification Problem? I

- The solution will usually be a combination of **static and dynamic verification methods**.
- A static verification method is one that relies solely on **static (formal) analysis** to demonstrate that a certain feature is properly implemented. These methods include **model checking and theorem proving**, which may provide **exhaustive** verification coverage.
- A **dynamic** verification method, such as **simulation**, requires **input stimulus** to be injected into a simulated DUV to expose bugs, but **never** provides **complete** verification coverage except in very small software modules or logic blocks.
- Although **static** verification is suitable for **block- and cluster-level RTL environments**, it is **not applicable** to post-partitioned **TLMs** because the size and complexity of TLMs exceed the capacity of current formal analysis tools.

What Is the Solution to the Verification Problem? II

- Commercial solutions may be available in the form of **verification intellectual property (VIP)**. VIP is a pre-verified verification component for a standard interface or core. It is available in the form of executable verification environments as well as assertion libraries, suitable for both simulation and formal analysis. All VIP today are **targeted** at hardware verification **at the RT level**.

Stimulus Generation I

- The most effective source for the bulk of post-partitioned model verification is **constrained random stimulus generation**. The functional requirements of input stimuli, defined by the DUV functional specification, are recorded in the "Verification Environment Design" section of the verification plan.
- The **functional constraints** will be **implemented as generation constraints** in the verification environment. Those constraints that further restrict generated stimuli to those most likely to activate DUV corner cases – verification constraints – are also recorded in the verification plan.
- The functional and verification constraints together are used to implement an **autonomous verification environment**. An autonomous verification environment is a verification environment, capable of **achieving 100% coverage without any external input**, such as tests.

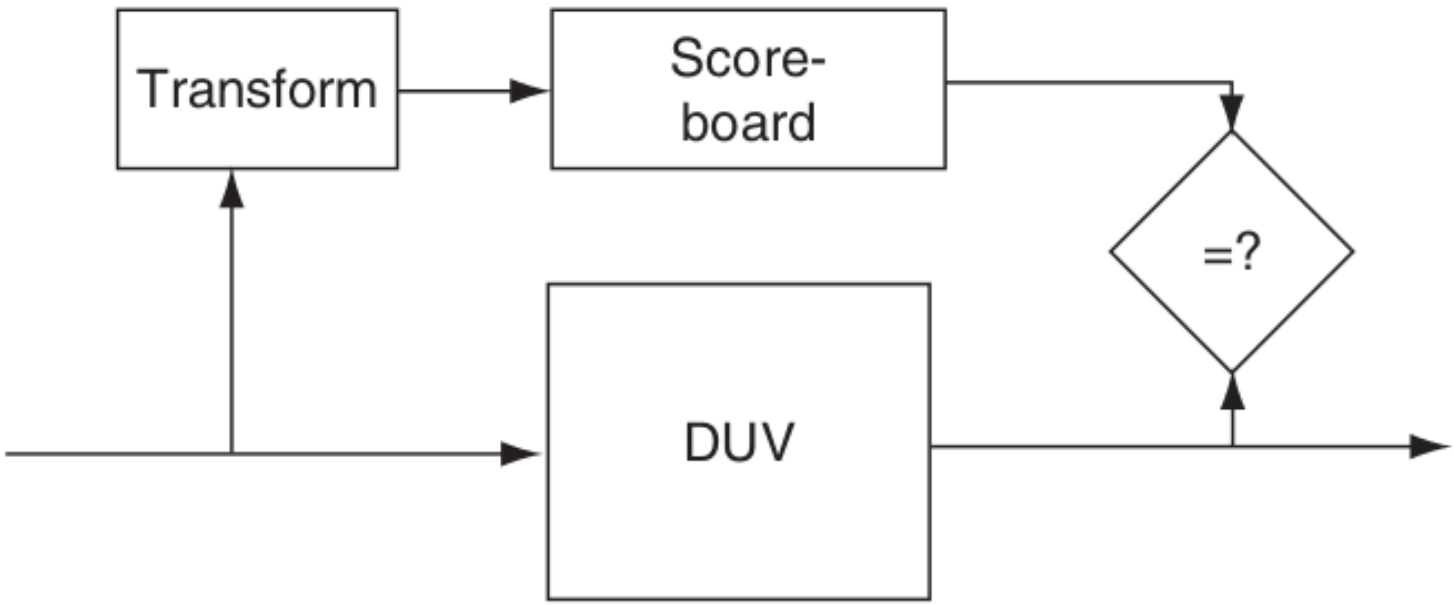
Stimulus Generation II

- This coverage includes both **functional and code coverage**. The **motivation** for building such an environment is that the **Return-On-Investment (ROI)** in verification productivity is substantially higher than that of a directed environment.
- In addition to designing the stimulus aspect of the verification environment to autonomously achieve the desired verification goals, an opportunity exists to **automate the feedback path from coverage measurement to stimulus generation through dynamic constraint adaptation**.
- **Directed tests** are typically used to wring the **first** few show-stopper bugs from the post-partitioned models. These tests are known as **"bring-up tests"** because they are used to bring up the models, that is, get the basic functionality working.

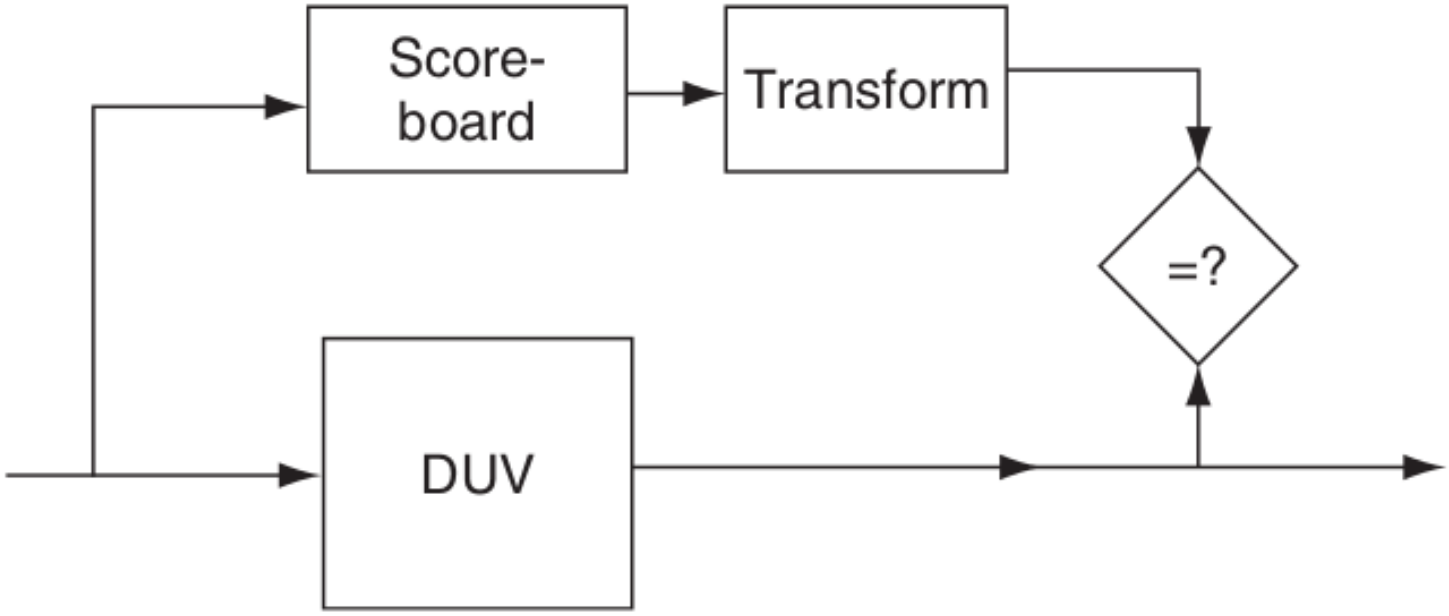
Response Checking I

- The mirror image of coverage measurement in a dynamic verification environment is response checking. The commonly used response checking techniques are **reference models, scoreboards, and distributed data and temporal checkers**. Data and temporal checkers are often implemented using assertions.
- A **reference model** is a **pre-verified executable representation of the DUV**, most commonly used for checking a subsystem, full chip, or major software component.
- The reference model is written at a **higher abstraction level** than the DUV and **their states compared** when their **time domains are synchronized**. For example, a common reference model is the ISS.
- A **scoreboard** is a data structure used to store either **expected results or injected stimuli** of a DUV that performs data transformation, and subsequently compares the data transformed by the DUV to that of the scoreboard.

Response Checking II



Response Checking III



- **Distributed data and temporal checkers** are a third approach to the design of a response checking aspect. They may be implemented using **procedural code, concurrent assertions, or immediate assertions**. An assertion is an expression stating a **safety** (invariant), **liveness** (eventuality), or **fairness** property. These types of checks have the advantage that they may be inserted **very close to the source of a DUV bug**, thereby reducing debugging time.