

# "Improving Shareholder Value by Separating Verification from Design"

The semiconductor industry continues to strive to manage the ever increasing risk of leaving a corner case bug that becomes the next front page story for EETimes. With cost of failure fast becoming a common agenda point in many semiconductor board room meetings around the world, how can you deliver to the ever growing demands of increasing shareholder value?

Managing scarce resources to deliver sustainable competitive advantage is the platform upon which most of today's strategic thinking is built. Yet in the complex world of semiconductor product design we continue to see the promotion of the "jack of all trades" designer in the false belief that it actually reduces costs.

**Separating verification from design is a natural evolution** to the necessary specialization that has become functional verification today. Why do accounting regulations demand that you employ armies of auditors to review your end of year accounts? When, not so many years ago, you could get away with your own accountants completing this function.

**Auditors are much the same as verification engineers.** They are approaching the problem from a completely different perspective. They do not come with the cognitive incompetence of **"I know that's right, because I produced it!"** The global company graveyard is littered with the tomb-stones of many household names that have made this mistake, Enron included.

In this presentation, Verisity will deliver a solution to provide unique value that can be generated when you separate the concerns of functional verification from design. Reducing the cost of failure risks and significantly improving the effectiveness of your scarce engineering resource by automating the process of verification itself.

*Presenter at edaForum04, Dresden, Germany, December 9th/10th 2004:*

*Coby Hanoach*

*Senior Vice President of Sales*

*Verisity Design, Inc.*

# High-level Verification

## State-of-the-art Verification Methodology

■ Tools: Specman Elite and e now from Cadence (who bought Verisity in April 2005)

💡 Reduction in time and resources required for verification.

## Basics of **e language**.

💡 For local configuration info see Exercise 4: *Intro to Specman Tutorial*.

[Please work your way through the on-line Specman intro tutorial.]

[Credits: The material for this lecture is adapted from Verisity training material.]

# Specman Elite

- **Automates** verification process.
- Provides **functional coverage analysis**.

 **Raises level of abstraction and enhances productivity.**

- Easy capture of design spec. - high level.
  - Build complete *verification* environments!
- Create lots of tests quickly and effectively.
- Design **self-checking testbenches**.
  - Including protocol checking!
- **Accurately track verification progress**.
  - Automatic coverage collection.
  - Promotes *coverage-driven* verification methodology.

---

# SN Main Enabling Technologies

## Constraint-driven Test Generation

- Control over automatic test generation.
- Capture constraints from spec and verification plan.

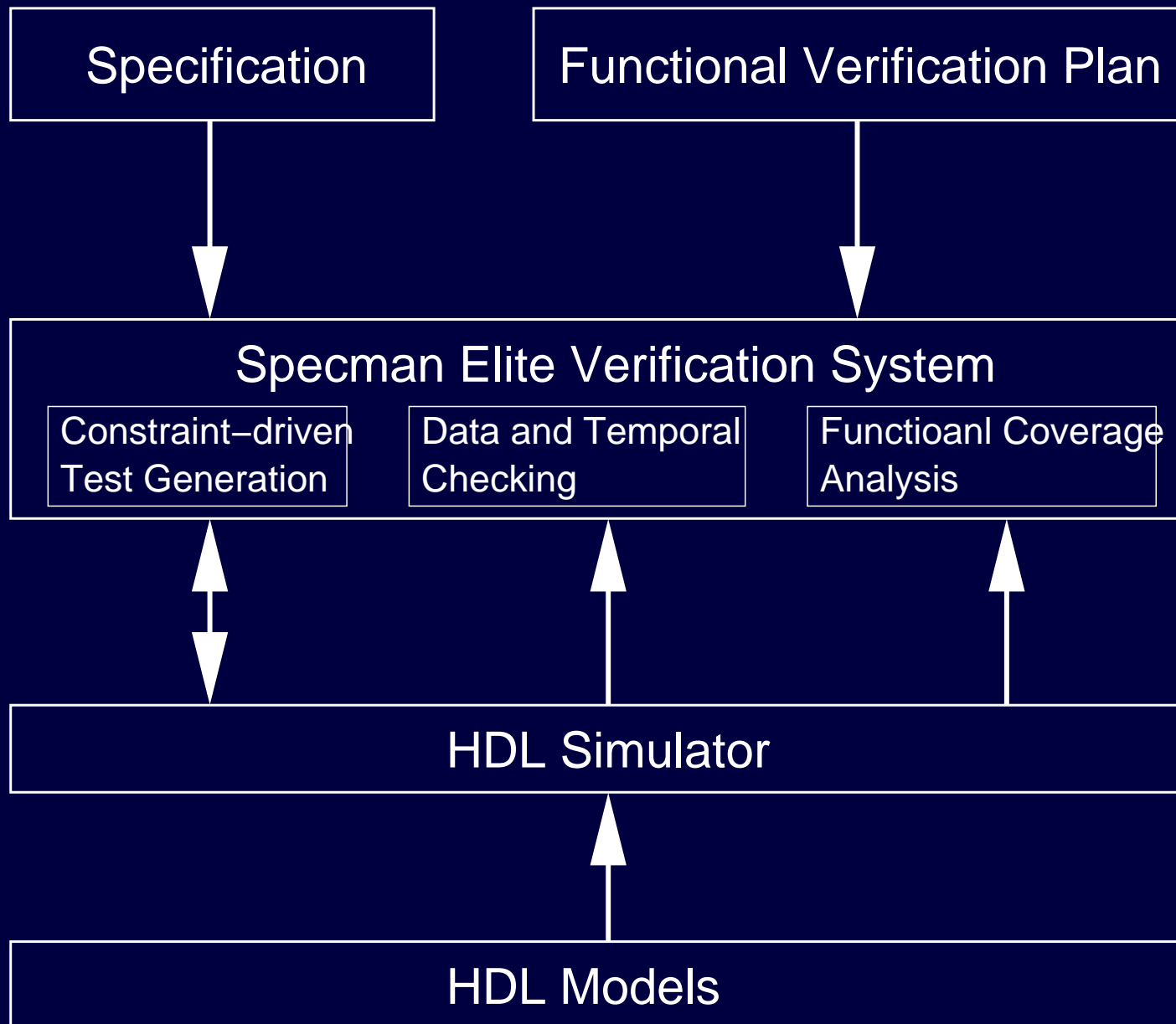
## Data and Temporal Checking

- Self-checking modules ensure data correctness and temporal properties.
- Checks are always active.
  - Unless turned off by: `set check IGNORE ;`:-)

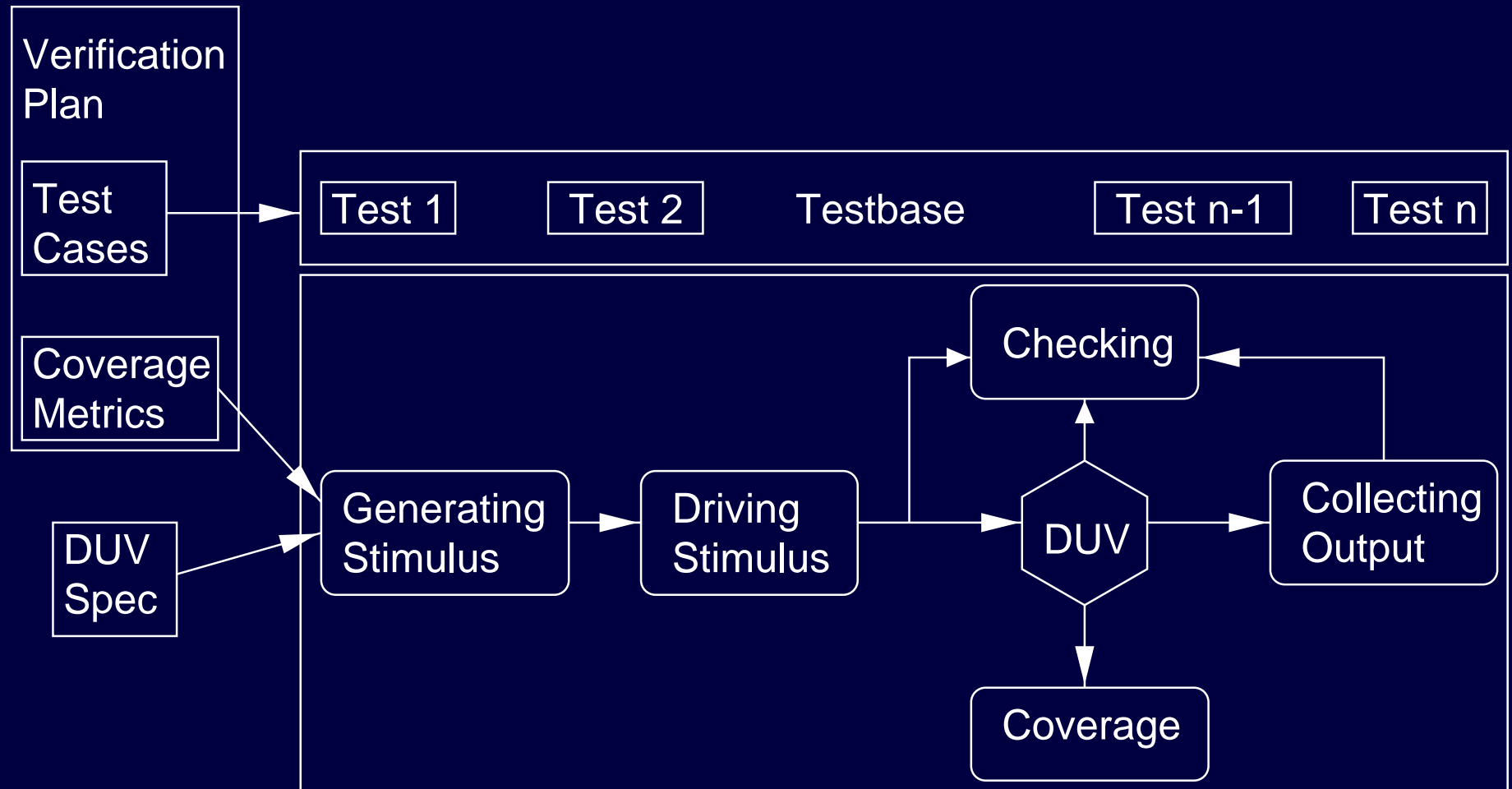
## Functional Coverage Analysis

- Measure progress of verification against coverage metrics.

# SN Verification Automation

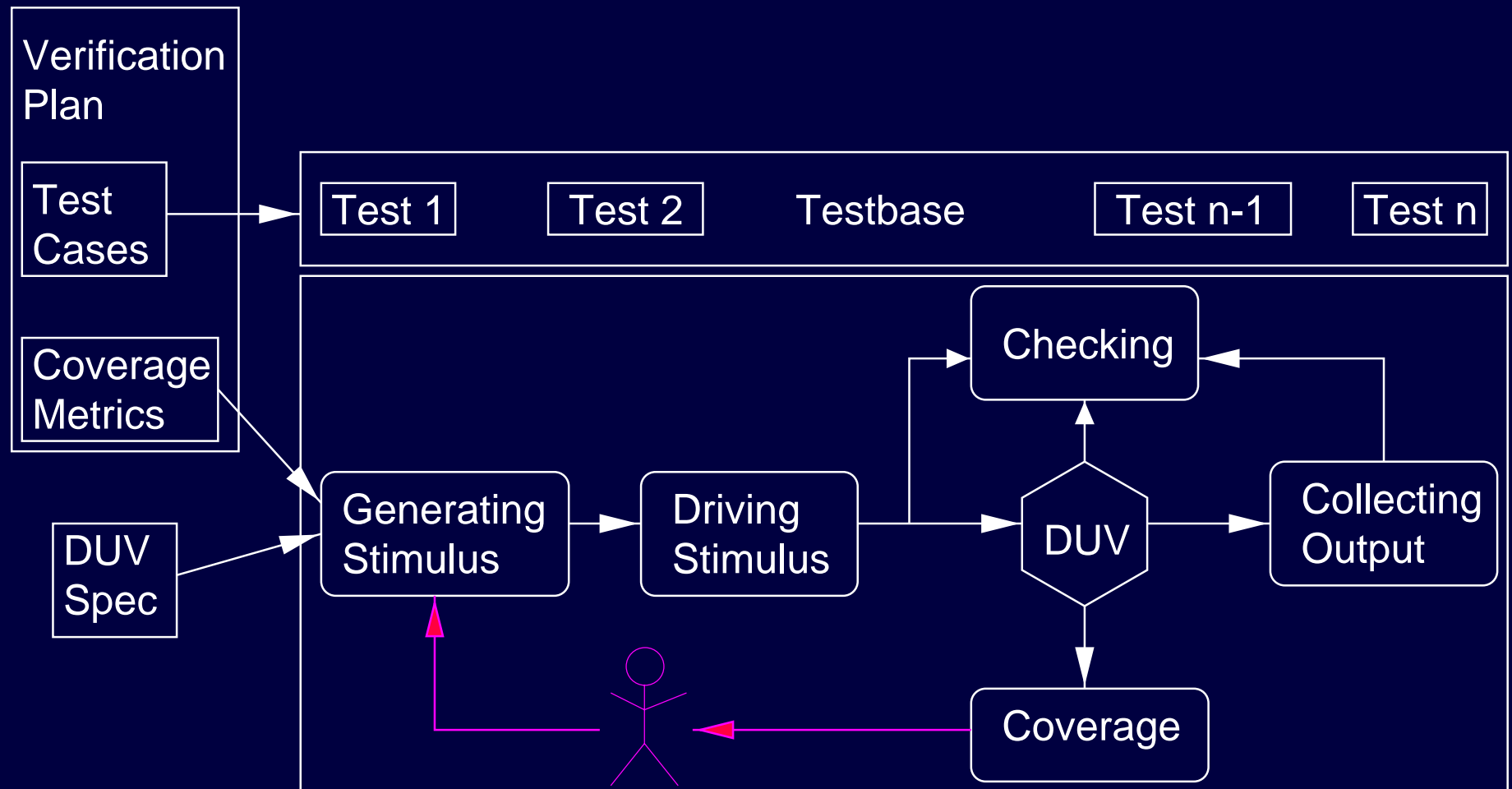


# Complete SN Verification Process



The key is in the **Verification Plan!**

# Complete SN Verification Process



The key is in the **Verification Plan!**

**Coverage-driven generation!**

# Basics of the e Language

**High-level language** for writing verification environments:

- test benches
- coverage collection
- test generation and checking

 *An e component is a representation of the “rest of the world” as seen from an interface of the design under verification (DUV).*

## **e supports:**

- modular **aspect-oriented** design
- high-level data types
- **pseudo-random constrained-based** data generation
- events
- high-level checking
- checking of basic **timing properties**



# Aspect-oriented Programming

**AOP is *next step up* from object-oriented programming.**

- Testcases have specific purposes:
  - Does parity check on packets work?
  - Are timing properties of transmission protocol valid?
- Both are different concerns: They are orthogonal!



**Two aspects of same application DUV.**

**AOP provides mechanisms to separate these two concerns into separate aspects of the verification environment.**

- Well-defined techniques for adding declarations, inserting or replacing code from the outside of a class, without editing the original class.

---

# On-line Help

**All Specman and e language help is on-line:**

- e language reference
- Command reference for Specman Elite
- User guide, etc.



*Look at a quick demo during lecture!*

# File Format

- An e code segment is enclosed with a **begin-code marker** `<'` and an **end-code marker** `'>`.
- Both the begin-code marker and the end-code markers must be placed at the beginning of a line (left-most), with no other text on that same line.

■ e code segment:

```
<'
import cpu_test_env;
'>
```

- Several code segments can appear in one file, each segment consists of one or more statements.

# Comments

- e files begin with a comment!
- This comment ends when first begin-code marker `<'` is found.
- Comments in code segments can be marked with `--` or `//`.

 Use end-code `'>` and begin-code `<'` markers to write several consecutive lines of comment in the middle of code segments.

# Predefined Constants

Constant	Description
TRUE	For Boolean variables and expressions.
FALSE	For Boolean variables and expressions.
NULL	For structs: specifies a NULL pointer. For character strings: specifies empty string.
UNDEF	Indicates NONE where an index is expected.
MAX_INT	largest 32-bit int ( $2^{31} - 1$ )
MIN_INT	smallest 32-bit int ( $-2^{31}$ )
MAX_UINT	largest 32-bit uint ( $2^{32} - 1$ )

# Syntactic Elements

**Statements** are top-level constructs.

- Valid within `<'` and `'>` markers.
- **Statements always end with a semicolon “;”!**

**Struct members** are second-level constructs.

- Valid only within a `struct` definition.

 Associated with dynamic constructs of a testbench e.g. stimulus.

- (There are also **Units** which are associated with testbench constructs such as drivers/checkers/scoreboards. They exist for the duration of the simulation.)

**Actions** are third-level constructs.

- Valid only when associated with a struct member, such as a method or an event.

**Expressions** are lower-level constructs.

- Can be used only within another e construct.

# Statements

Key statement types:

- **struct**: Defines a new data structure.
- **unit**: Defines a new unit.)
- **type**: Defines an enumerated type or subtype.
- **extend**: Extends a previously defined struct or type.
- **define**: Extends language. ■ `define OFFSET 5;`
- **import**
- ... (more, see on line doc)



**Imports must be first (after defines).**

Otherwise, order is not critical.

# Structs vs Units:

Structs are the most basic building blocks in e.

- Used to keep data and operations together.
  - packets, instructions, frames
- Can be created at run-time, i.e. they are dynamic.
  - Data can be generated on-the-fly.

**Units are a special kind of struct.**

- Units are **static**! Can be generated during test phase only.
- Optional mapping to HDL path.
- Used for generators/checkers/monitors, **bus functional models (BFMs)**, self-checking structures, overall testbench.
  - **BFMs** package all bus functional procedures of an interface, i.e. all transactions supported by the interface. The transactions are abstracted from a physical-level interface to a procedural interface. BFMs can be used to generate stimulus as well as to check the DUV response.



# Struct and Struct Members

Members are 2nd-level constructs: Valid only within a struct definition.

- A simple struct for packets to be used in comms protocol:

```
type packet_kind: [atm, eth];
struct packet {
  len: int;
  keep len < 256;
  kind: packet_kind;
};
```

- **keep**: Specifies rules for constraints to influence data generation.

- Another example struct for transactions:

```
struct transaction {
  address: uint;
  data: list of uint;
  transform(multiple:uint) is empty;
};
```

# Struct Members

**Field:** Defines data entry to be member of enclosing struct with explicit data type.

**Method:** Defines operational procedure that can manipulate fields of enclosing struct and access run-time values in DUV.

**Subtype declaration:** Defines instance of parent struct in which specific members have particular values or behaviour.

- Use `when` for conditional constraints on possible values of a field.

**Constraint declaration:** Influences distribution of values generated for data entries and the order in which values are generated. ■ `keep`

**Coverage declaration:** Defines functional verification goals and collects data on how well the testbench is meeting these goals.

- `cover event-type is coverage-item-definition; ...;`

**Temporal declaration:** Defines events and their associated actions. ■ `event`

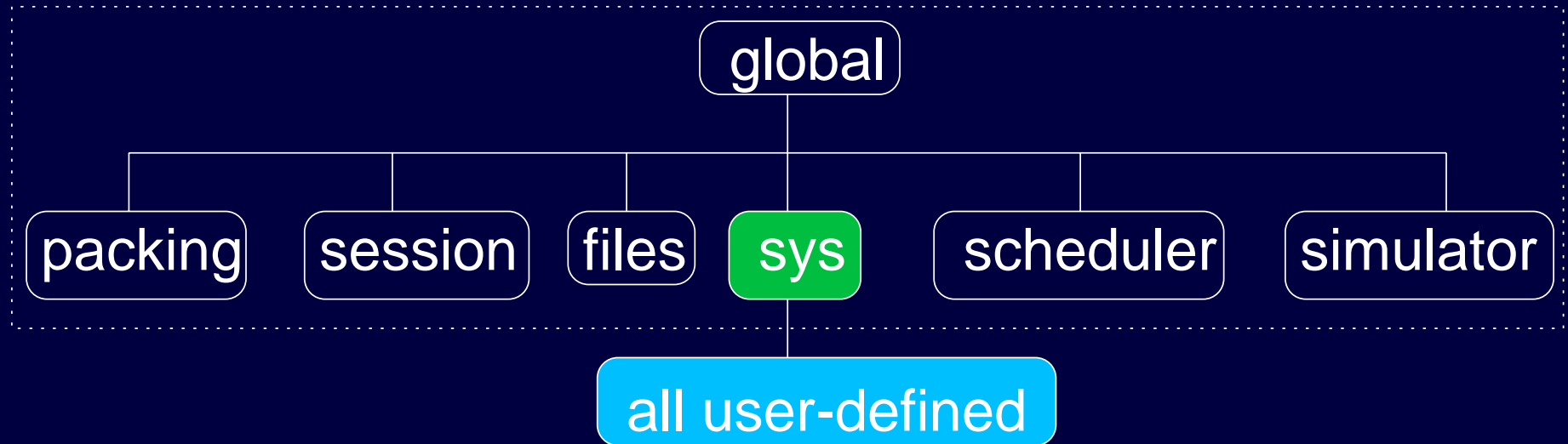
```

type PCICommandType: [ IO_READ=0x2, IO_WRITE=0x3,
                        MEM_READ=0x6, MEM_WRITE=0x7 ];
struct pci_transaction like transaction {
    command : PCICommandType;
    keep soft data.size() in [0..7];
    dual_address: bool;
    when dual_address pci_transaction {
        address2: uint;
    };
    bus_id: uint;
    event initiate;
    on initiate {
        out("An event has been initiated on bus ", bus_id);
    };
    cover initiate is {
        item command;
    };
    transform(multiple:uint) is only {
        address = address * multiple;
    };
};

```

# Predefined Structs

An e environment contains by default a number of predefined structs (and of course some user-defined ones).



💡 The system struct **sys** is the root for user-defined structs.

- Must instantiate **user-defined structs** under **sys**.
- Contents of **sys** can be viewed via SN GUI.
- Similar to **main** in C. ;-)

# Instantiation under sys

💡 Every user-defined struct (including units) must be instantiated as a (sub)field of **sys**.

```
struct packet {  
    address : uint (bits : 2);  
    payload : uint (bytes : 64);  
};
```

```
unit router_bfm {  
    packets : list of packet;  
};
```

```
extend sys {  
    router : router_bfm is instance;  
};
```

# Generating Input for Verification

## When to generate stimulus?

- **On-the-fly:**

- Easy to react to the current state of the DUV.
  - \* Makes reaching corner cases easier.
- Easier concurrency control over multiple channels.
- Small memory footprint.

- **Pre-run**

- Can be easier for complex sequences with inter-dependencies.
- May be compulsory.
  - Assembler program: Must first be compiled and loaded!

- **Mixed: Pre-run and On-the-fly**

- Processor Verification - Pre-run: sequence of instructions
- Processor Verification - On-the-fly: interrupts

Specman supports all of these methods.

# Generation with SN

## Pre-run (in Generate phase):

- Use **Generate** or **Test** command
  - Test calls Generate command!
- Recursively generates *everything* under `sys`.
- BEWARE: Can consume a lot of memory!

## On-the-fly (at run-time):

- Use `gen` action. ■ `gen gen-item [keeping {...}]`
- Allows to dynamically generate values based on DUV state.

# Fields

**Syntax:** `[!][%] field-name[: type] [[min-val .. max-val]] [((bits | bytes):num)]`

! Denotes an *ungenerated* field.

% Denotes a *physical* field.

The type for the field can be any scalar type, string, struct, or list.

(bits | bytes: num) specifies width of field in bits or bytes.


```
type NetworkType: [IP=0x0800, ARP=0x8060] (bits: 16);
  struct header {
    %address: uint (bits: 48);
    %length: uint [0 .. 32];
  };
  struct packet {
    hdr_type: NetworkType;
    %hdr: header;
    is_legal: bool;
    !counter: uint;
  };
```



**Field order is important!** It is the packing order for physical fields.



# Ungenerated Fields

- Marked with **!**.
  - Values for this field are **not generated automatically**.
  - **Useful for fields that:**
    - Are explicitly assigned values during verification.
    - Must contain values whose computation is too complicated to be expressed with constraints.
  - Ungenerated fields get **default initial value:** 0 for scalars, NULL for structs and empty list for lists.
  - Ungenerated fields whose value is from a range (e.g. [20..30]) get initialized to the first value in range.
-  If the field is a struct it won't be allocated and none of the fields in it will be generated.

# Packing: Driving Stimulus into the DUV

`pack( )` function:

- `pack(option:pack option, item: exp, ...)`: list of bit
- Specman Elite system function.
- Conversion from higher-level data structure to bit stream required by DUV.

```
■ i_stream = pack(packing.high, opcode, op1, op2);
```

**pack options are:** `packing.high`, `packing.low` or `NULL`

- `packing.high`: 1st item at MSB
- `packing.low`: 1st item at LSB
- `NULL`: Use global default - set initially to `packing.low`.

**item:** A legal e expression that is a path to a scalar or a compound data item, such as a struct, field, list, or variable.

# Packing High

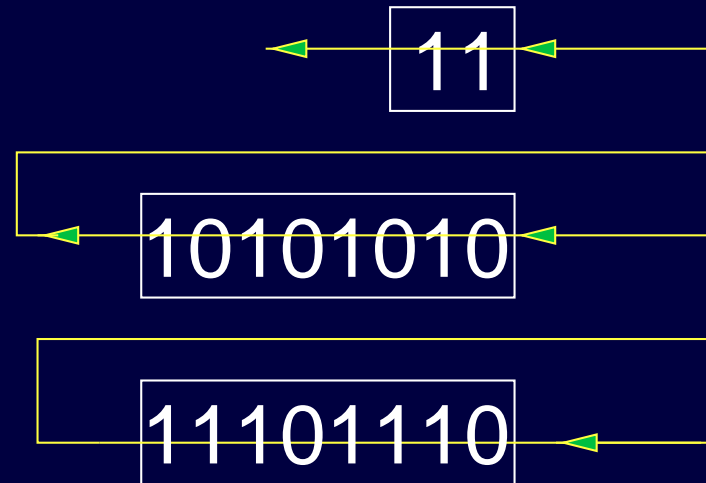
**packing.high: 1st item at MSB**

```
i_stream = pack(packing.high, addr, data);
```

```
packet.addr = 2'b11;
```

```
packet.data[0] = 0xaa;
```

```
packet.data[1] = 0xee;
```



```
17.....0  
i_stream = 11 10101010 11101110
```

# Packing Low

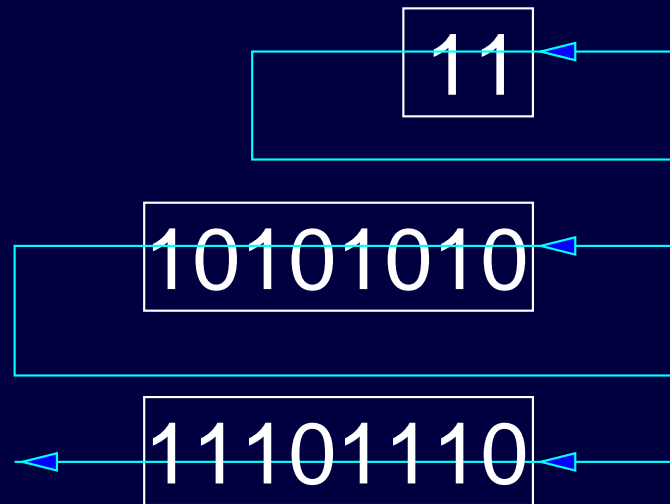
**packing.low: 1st item at LSB**

```
i_stream = pack(packing.low, addr, data);
```

```
packet.addr = 2'b11;
```

```
packet.data[0] = 0xaa;
```

```
packet.data[1] = 0xee;
```




```
17.....0  
i_stream = 11101110 10101010 11
```

# Physical Fields

- Marked with %.
- Physical fields are packed when the struct is packed.
- Used for fields that represent data that will be sent to HDL design in the simulator.
- Non-physical fields are called virtual fields.
  - They are not packed automatically when the struct is packed.  
(– They can be packed individually if needed.)
- If no range is specified, width of field is determined by field's type.
- If the field's type does not have a known width, you must use (bits | bytes: num) syntax to define the width. (Important for packing!)

# Limitations of e's AOP Implementation

- Everything can be extended!
  - So more discipline and structure is required.
- Fields can only be appended:
  - Fields are physically appended to existing fields
  - Might create a problem when packing!
- Variance control fields: Extensions can only be specified for a single value of the control field.
  - instructions `add` and `sub` with feature that applies to both
    - Needs to be specified for both or use trick!  See next slide!
- Methods can only be appended, prepended or replaced.
- Aspects are order-dependent (on loading).

## Extensions via *variance control fields* can only be specified for a **single value of the control field!**

- To get around this, introduce an additional *virtual* field.
- This field controls common extensions.

### ■ Extension to an instruction struct (for Calc\_1 design):

```
type opcode_t : [ NOP, ADD, SUB, INV, INV1, SHL, SHR ] (bits : 4);

struct instruction_s {

    %cmd_in : opcode_t;
    %din1    : uint (bits:32);
    %din2    : uint (bits:32);

    !resp    : uint (bits:2);
    !dout    : uint (bits:32);

    check_response(ins : instruction_s) is empty;

}; // struct instruction_s

extend instruction_s {

    is_a_shift : bool;
    keep is_a_shift == opcode in [SHL, SHR];

    when is_a_shift instruction_s {
        // Common extension to SHL and SHR goes below.
        ...
    }
}
```

---

# Specman Elite Tutorial

**DUV:** simple CPU (ALU, 4 regs, PC, PC\_Stack, fetch/exec FSM)

- **Interface:** clock, reset, instruction [8 bit]

## Learn how to:

- Design the verification environment
- Define DUV interfaces
- Generate a simple test
- Drive and check the DUV
- Generate constraint-driven tests
- Define and analyse test coverage
- Create corner case tests
- Create temporal and data checks
- Analyse and bypass bugs

*108 pages. A really easy “learn by doing” lab. Takes about 2h. :)*



---

# We have now covered

- Basics of e language.
- Please do Specman Elite Tutorial with CPU DUV.

## Next:

- Assignment 2 - Intro to .e code and verification method.
- Hands-on session with demo.