2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

What Ever Happened to AOP ?

James Strober, P.Eng
Ciena Corporation
3500 Carling Avenue
Ottawa, Ontario, K2K 3C8 Canada
jstrober@ciena.com

Corey Goss
Cadence Design Systems Inc.
1130 Morrison Drive, Suite 240
Ottawa, Ontario K2H 9N6 Canada
cgoss@cadence.com

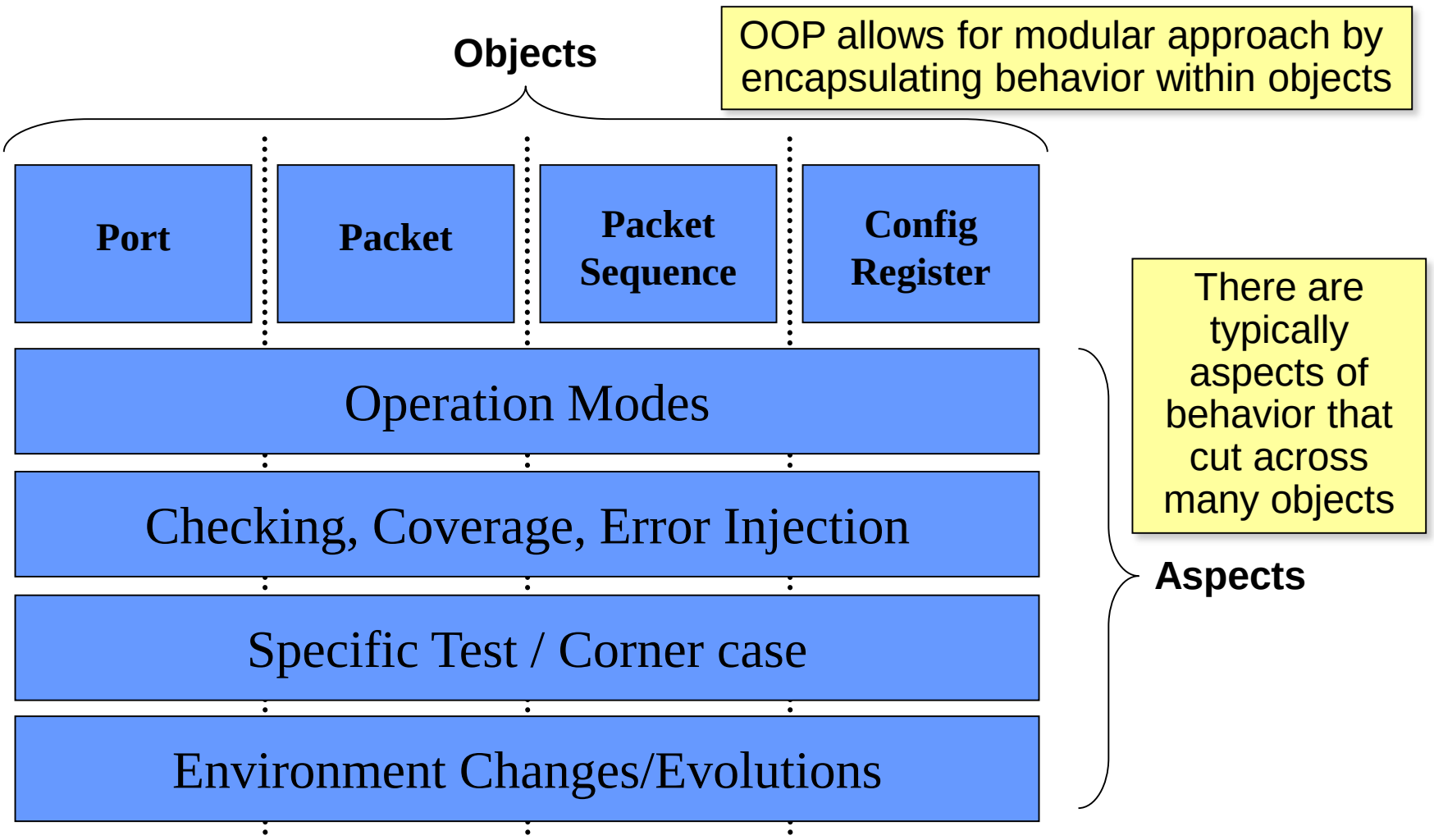
What is Aspect Oriented Programming (AOP) ?

- From Wikipedia:

  Aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns

  –Cross-cutting concerns are aspects of a program that affect other concerns. These concerns often cannot be cleanly decomposed from the rest of the system in both the design and implementation, and can result in either scattering (code duplication), tangling (significant dependencies between systems), or both.

Orthogonal Relationship Between Aspects & Objects



History of AOP

- Emerged from a **need** to better modularize and address cross cutting concerns
- Many believe that AspectJ AOP extension to Java 2001 was the first AOP language
- Has since been adopted by several languages including:
  - Perl, Python, Ruby, Groovy, C++, COBOL, Java, Matlab, Prolog, Smalltalk, XML and many others
- Has led to an emerging discipline of “Aspect Oriented SW Development” AOSD
  - http://en.wikipedia.org/wiki/Aspect-oriented_software_development

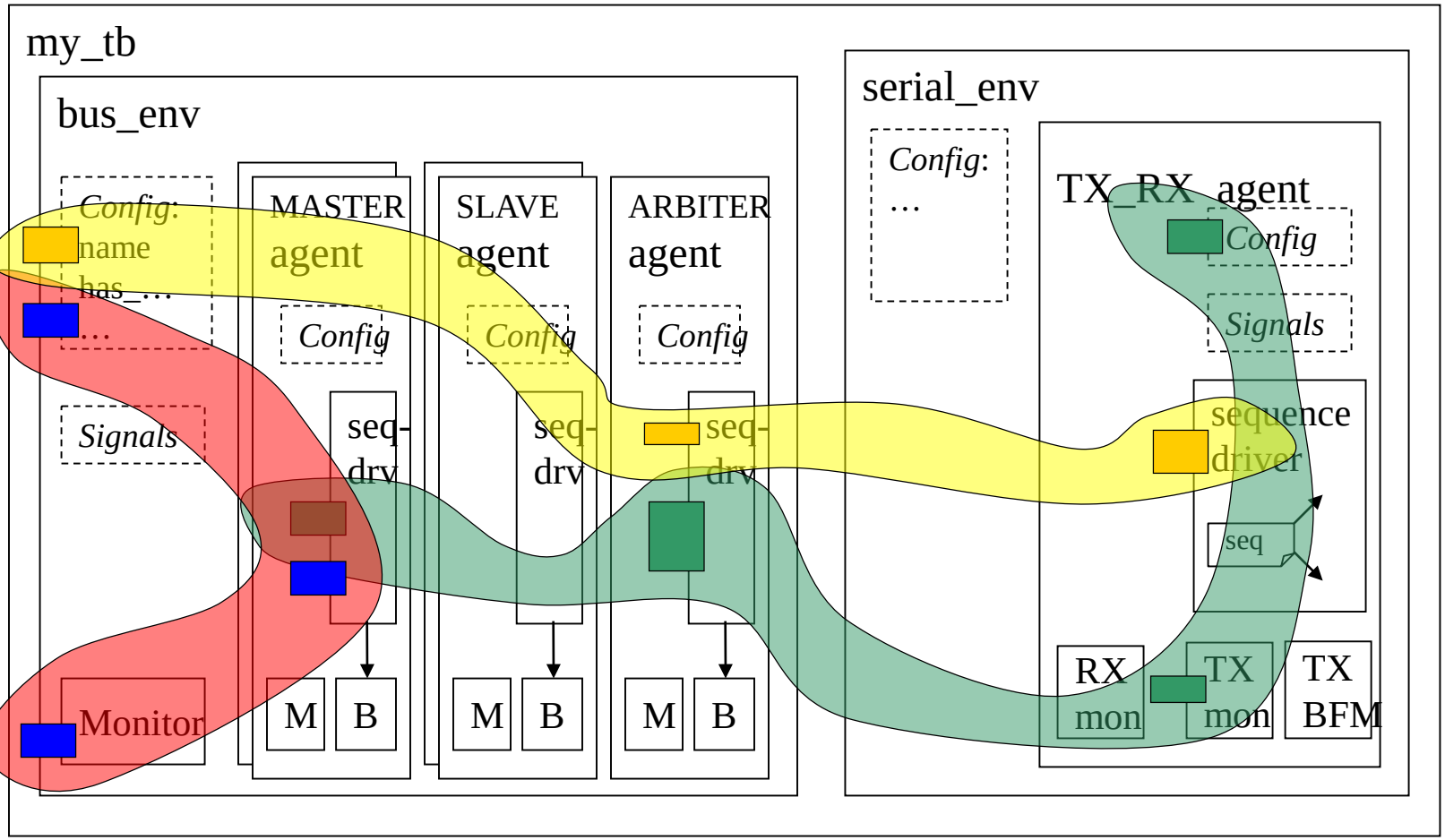
History for Verification

- Cross cutting and modularization concerns have been present in our industry since inception
  - As a result, we have had AOP languages for much longer than the SW industry as a whole
- ***e***, created by Verisity Design Inc. in 1993 is natively AOP
- AOP extensions subsequently added to OpenVera
- Now, our industry is moving towards SV, an OOP language, which is a **significant step back** for advanced verification
- AOP proposed to SystemVerilog 2012 but rejected

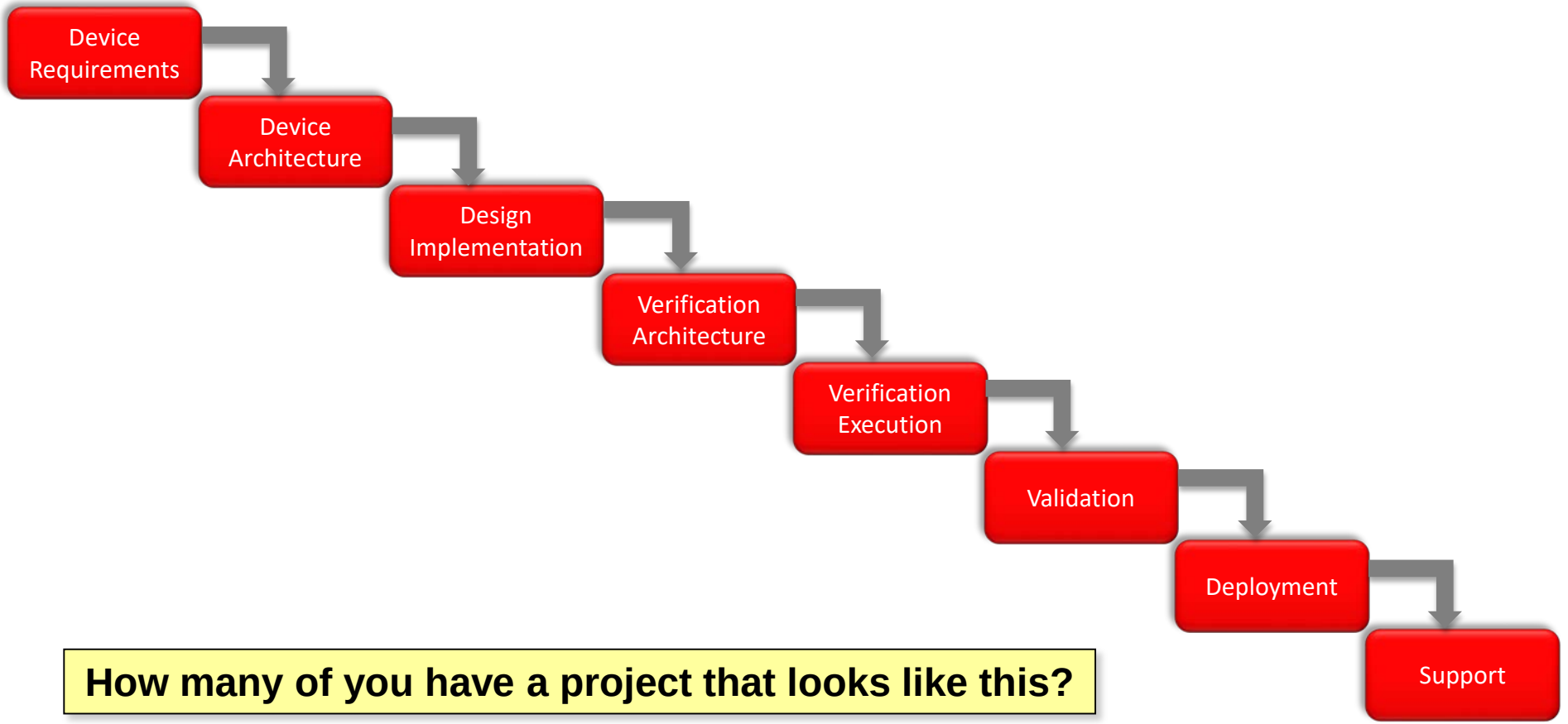
Power of AOP for Verification

- Efficient handling of cross-cutting concerns is particularly well-suited to:
  - Mitigating the impact of product feature churn
  - Addressing real-world verification architecture challenges
  - Verification closure
  - Code re-use
  - Debugging

Cross Cutting – What's in a testcase?

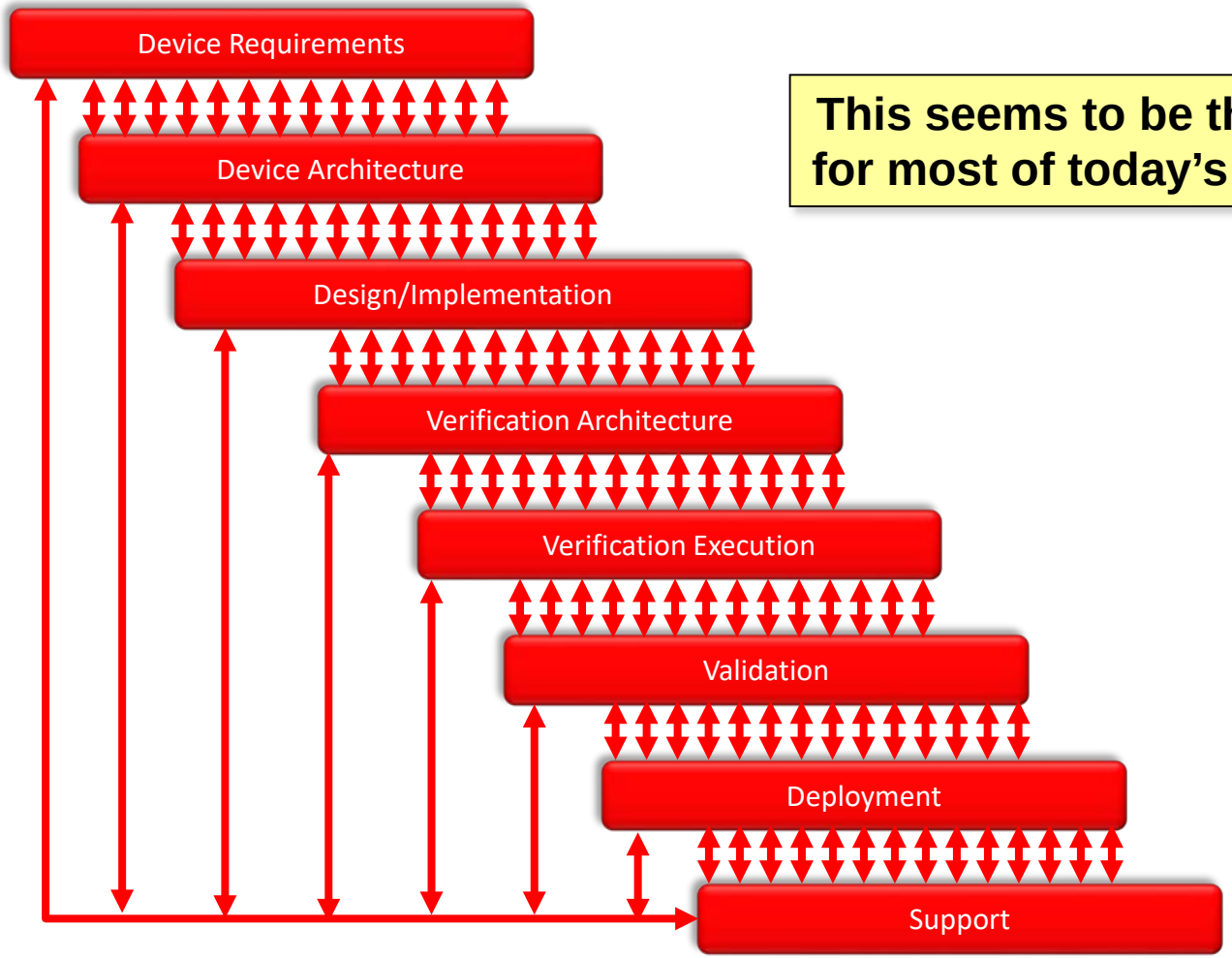


Device Development Ideal Waterfall Flow



How many of you have a project that looks like this?

Realistic Parallel Waterfall Flow



This seems to be the norm for most of today's projects

OOP for VE Architecture

- OOP is well-suited to address the ideal flow but NOT the realistic flows. Why?
  - We can architect the Verification Environment (VE) for anticipated scalability and flexibility requirements **ONLY** when they are known
  - We can allow for flexibility through judicious application of inheritance and encapsulation and implementation of predefined hooks but **ONLY** based on known requirements
  - VE architecture can be detrimentally sensitive to architectural churn

AOP for VE Architecture

- AOP is well-suited to address realistic flows. Why?
  - Probability of arriving at a first order approximation of final VE at project onset is usually low
  - AOP provides a framework for efficient re-architecture and re-work
  - AOP constructs provide hooks without a need to anticipate or predict their necessity and without disturbing the original code base
  - Feature addition, changes, pruning as well as arbitrary variants can be handled safely and efficiently

VE Churn Example: New 64 BIT driving protocol

- Midway through the project, a new 64 bit driving protocol must be supported
  - Current environment supports 8, 16 and 32 bit bus speeds
  - New driving protocol will affect several objects:
    - Configuration: Adjust the clock speed
    - Driver: Need to segment/drive 64 bits at a time
    - Monitor: Need to receive 64 bits at a time

Note: The code examples in this paper are based on an imaginary AOP verification language.

VE Churn Example: AOP extensions for new packet

Crosscutting

Aspect

```
extend bus_speed_t: [BITS_64];
```

enumeration extended to include new bus speed

```
extend project_config_objects_c (BITS_64'bus_speed) {
   constraint clock_cg is also {
      clock_speed == serial_clock_frequency/64;
};};
```

configuration class extended and constraint group modified

```
extend project_driver_c (BITS_64'bus_speed) {
   task drive_packet(…) is only {
    …
};};
```

drive protocol extended to account for new bus speed

```
extend project_monitor_c (BITS_64'bus_speed) {
   task receive_packet(…) is only {
   …
};};
```

monitoring protocol extended to account for new bus speed

VE Churn Example: New packet version

- Midway through the project, a new packet format must be supported
  - Current environment supports VI and VII packets

```
type version_t: [VI, VII];
class packet_c {
   rand version_t version;
   rand byte payload [];
};
```

  - New packet must:
    - Have payload within the range of 100 to 200 bytes
    - By default, contain a parity byte after the payload
    - Be driven 16 bits at a time into the DUT with no inter packet gap
      - Current packets are 8 bits at a time

VE Churn Example: AOP extensions for new packet

Aspect

```
extend packet_c (VIII'version) {
   rand bit contains_parity;
   constraint v3_pkt_constraints {
      payload.size() inside [200:100];
      contains_parity == 1'b1;};
   byte parity;  //only present for VIII packets
};};
```

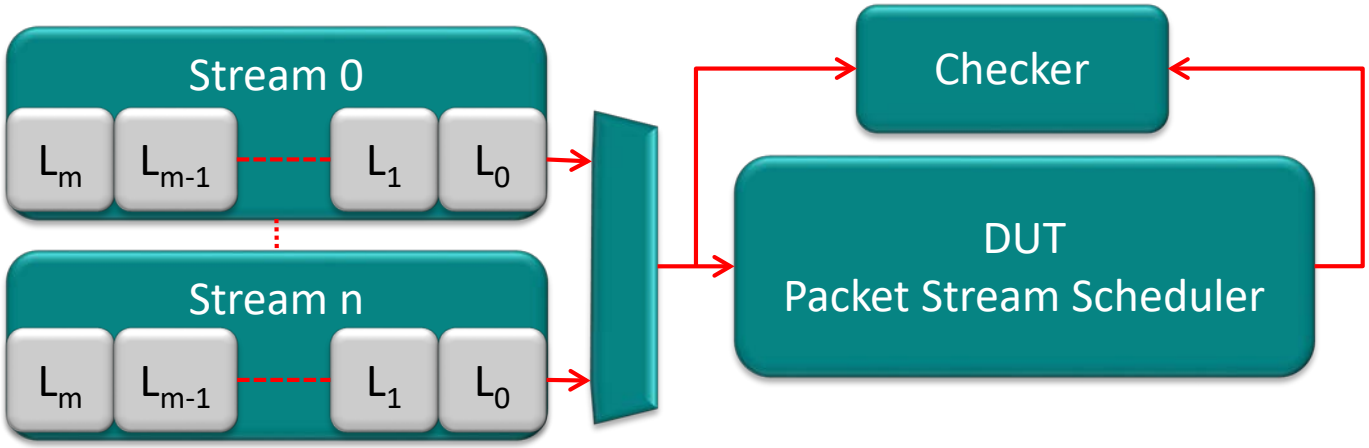
base packet easily extended for VIII packets

Crosscutting

```
extend port_c (VIII'version) {
   constraint VIII_port {
      bus_width == 16;
      inter_pkt_gap == 0; // no gaps for VII
};};
```

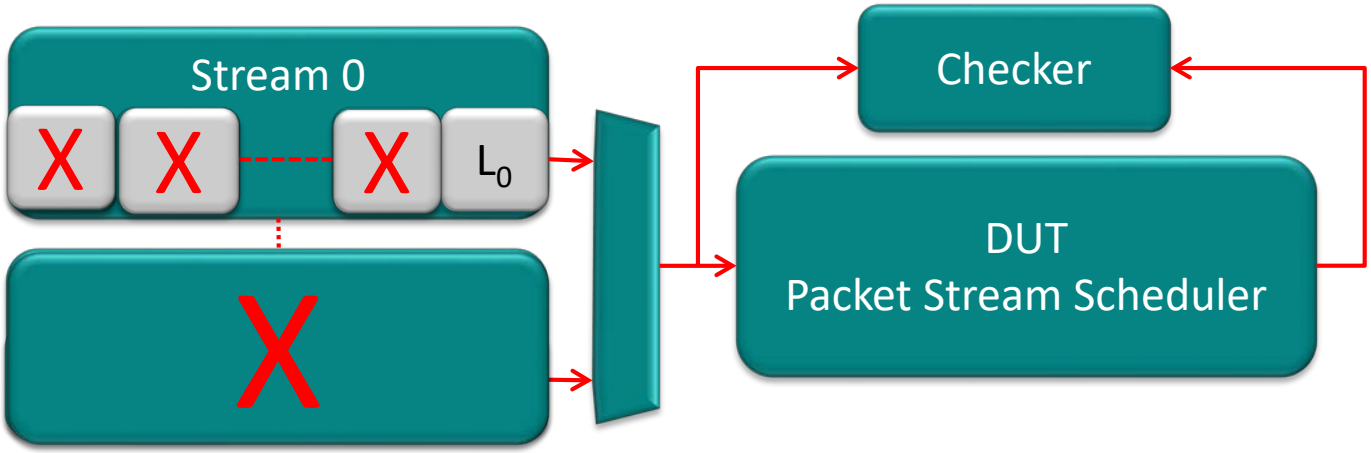
Port must also be modified to support new VIII packet format

OOP - Linear Scalability



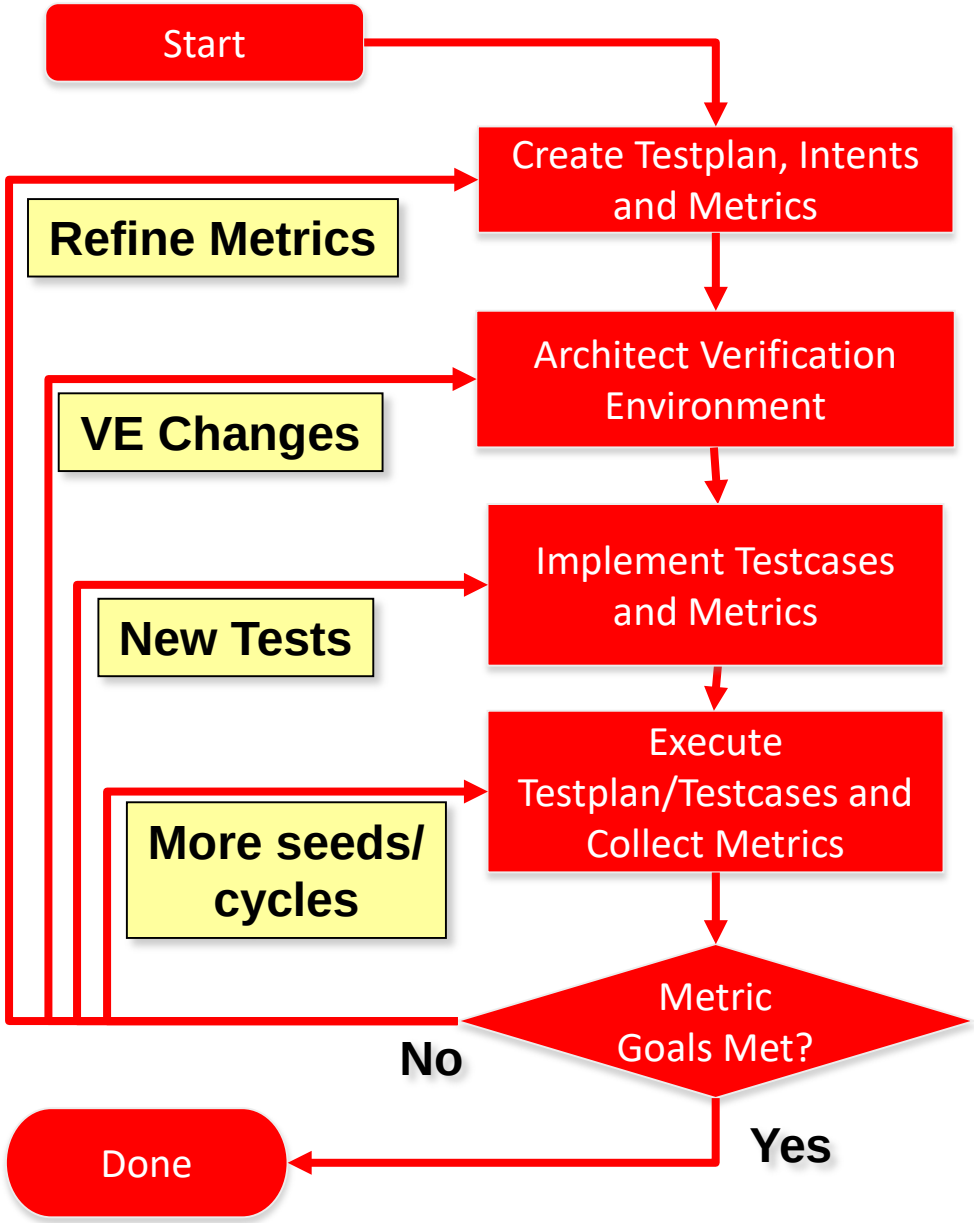
- Consider a scalable testbench for a packet stream scheduler
- Stimulus for each stream is generated by an instance of a legacy multi-layer stack of verification components
- Conceptually scalable but scale practically limited by memory and performance constraints
- What happens if we need to transcend this practical limit to hit a maximum scale boundary that is orders of magnitude greater than our normal range of testing ?

AOP Continuum of Controllability



- Layers can be collapsed into lower functionality L_0 to achieve maximum scale using AOP
- To hit the maximum scale boundary there is also the option to give L_0 limited multi-stream capability using AOP
- To hit the boundary, use AOP to trade-off functionality for memory, performance, and scale within the narrow scope of 1 testcase or verification intent
- AOP enables a “Continuum of Controllability”

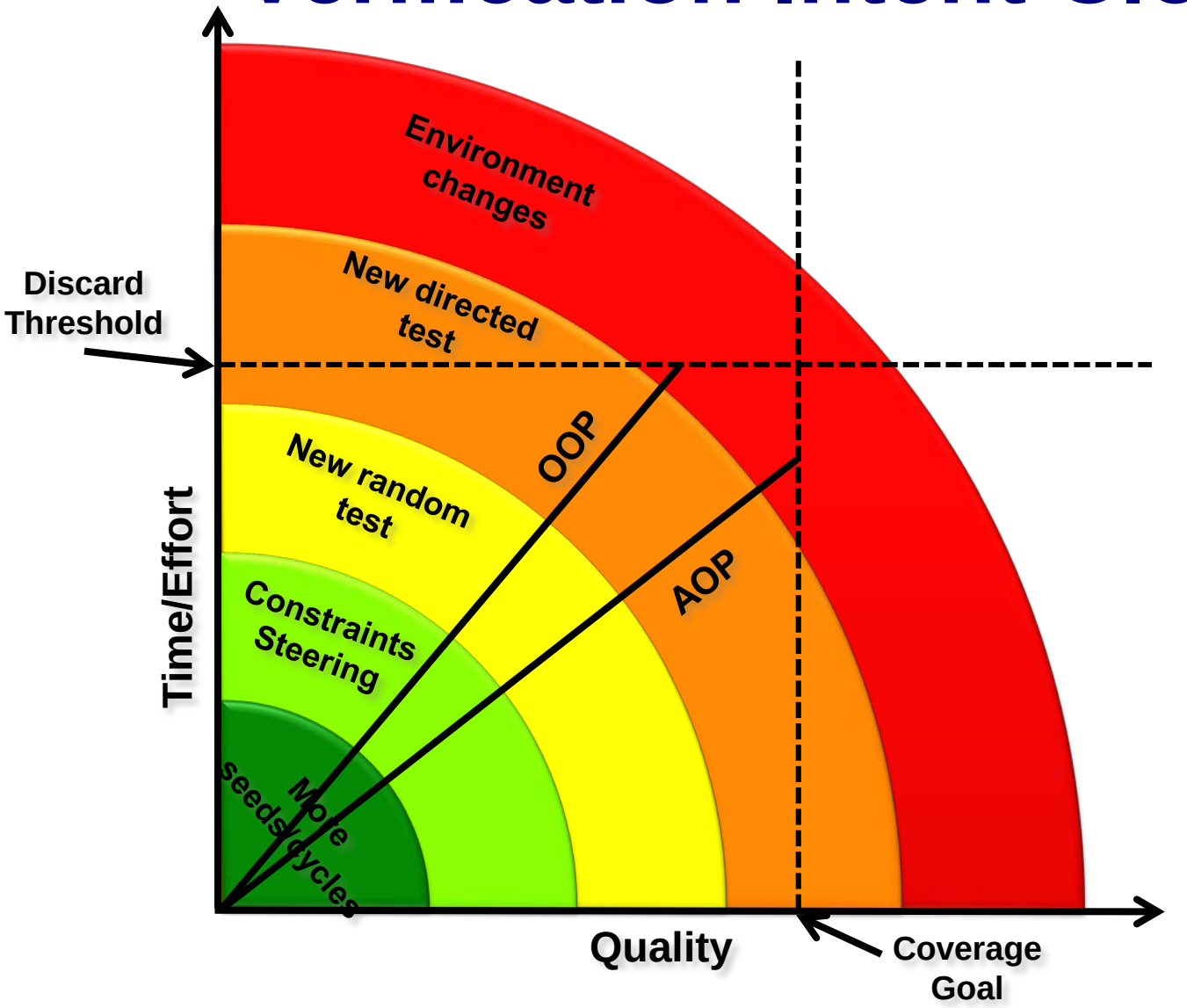
Verification Closure



Empirical 90/10 Target

- Verification Coverage and Metrics Closure is one of the most significant risks to project schedule/quality
- We should be targeting to hit 90% of verification metrics through randomized tests before resorting to directed verification
- 90% can typically only be achieved with significant verification planning discipline and verification architecture expertise
- Even at 90%, the effort to close the remaining 10% can easily explode

Verification Intent Closure



AOP for hard to hit coverage

- AOP is particularly well suited for targeting typically hard-to-hit coverage:
  - Out of scope from original VE architecture
  - Complex design interactions that present controllability and/or visibility challenges
  - Scenarios requiring complex synchronization, orchestration, and alignment of configuration
    - Alignment of Earth, Moon and Stars

Efficiency and Quality

- Consider that for any given verification project we may have anywhere from hundreds to thousands of individual verification intents to close
- Even a marginal improvement in coverage closure efficiency using AOP will have an appreciable impact on project schedule and quality
- Verification is the bottleneck, we need all the help we can get

Coding Efficiency Examples

- Yes, **almost** anything can be done in any language
  - Just a matter of how much effort
- Consider a simple “Hello World!” example:

```
+
+
+
…
+
.
>
+
+
.
```

brainf*ck – 106 lines

```
class my_class;
   function new();
      $display(“Hello World”);
   endfunction
endclass
my_class myc = new();
```

OOP (SV) – 6 lines

```
extend global {
   run() is also {
      out(“Hello World”);
};};
```

AOP – 4 lines

Coding Efficiency Example (OOP vs. AOP)

- Stream a single random packet into our DUT
  - Example uses sequences (common in today's TB's)
  - Example is far from complete code
    - Assume that all other code is in place for driving/monitoring, checking, etc.

Coding Efficiency Example: AOP

- Recall our previous packet definitions:

```
type version_t: [VI, VII];
class packet_c {
   rand version_t version;
   rand byte payload [];
};
```

```
extend version_t: [VIII];
extend packet_c (VIII'version) {
   rand bit contains_parity;
   byte parity;
};};
```

- Now, the additional code:

```
extend my_sequence_c (RANDOM'kind) {
   rand packet_c item;
   function new();
      item = new();
      assert(randomize(item));
   endfunction
};};
```

No need to consider version when randomizing as all AOP extensions are of the base type

AOP – 6 lines

Coding Efficiency Example: SV OOP with UVM

- Red colour indicates extra code compared to AOP

```
import uvm_pkg::*;
typedef enum {VI, VII, VIII} version_t;
class packet_c;
   `uvm_object_utils(packet_c)
      …
   `uvm_object_utils_end
   rand version_t version;
   rand byte payload [];
endclass
```

Import UVM to access factories

Intrusive addition of new version

Need to register class with factory to allow for type overrides (see next slide)

```
class viii_packet extends packet_c;
   `uvm_object_utils(packet_c)
      …
   `uvm_object_utils_end
   rand bit contains_parity;
   byte parity;
endclass
```

Need to create a new derived class to represent new variant

Need to register new class with factory as well

Coding Efficiency Example: SV OOP with UVM

- Red colour indicates extra code compared to AOP

```
class my_random_sequence extend my_sequence_c;
   rand version_t version;
   rand packet_c item;
   function void new();
      assert(randomize(version));
      case(version)
         VI,VII: item = new();
         VIII: item = v3_packet::type_id::create(…);
      endcase
      assert(item.randomize());
endclass
```

Add new field to randomize the packet version first, before assigning the correct packet class

We need to use the UVM factory to pull in the correct derived class
Note: Must rewrite this logic if a new version is added

OOP SV – 18+ lines

- It is typical to see > 30% less code using AOP languages

AOP for Debug

- AOP constructs are particularly well suited for debugging. Why ?
  - Extend any part of the VE to add visibility for instrumentation
  - Extend any part of the VE to increase controllability
  - Selectively change any part of the VE functionality for exploration and what if analysis
  - All can be layered on top of existing code without changing the code base
  - Reproduce escapes seen in validation that require precise orchestration

Debug Coding Example

- Recall our previous example (new 64 bit bus speed)
- After making our code changes, we notice:
  - 64 bit data was not being read correctly by DUT
    - Can see data on bus, but bit ordering appears incorrect
  - Data appears on rising edge when falling is expected
    - Drivers clocking event might be incorrectly defined
- Let's see how AOP can help us:
  - Debug how packets are being converted into bits
  - Try out a new fix to the clocking event used to drive data

Debug Coding Example: The base code definition

```
extend project_driver_c (BITS_64'bus_speed) {

  event clock is rise(clock_sig);

  task drive_packet(p: project_packet_c) is only {
     packet_as_64_bit_words =
        transform_packet(p, config.endienness);
     req_sig = 1'b1;
     @dut_ready;
     req_sig = 1'b0;
     data_valid_sig = 1'b1;
     for(int i=0; i<packet_as_64_bit_words.size();i++) {
        data_signal = packet_as_64_bit_words[i];
        @clock;
     };
     data_valid_sig = 1'b0;
};};
```

Clock seems incorrect

bits within every word seem incorrect

Debug Coding Example: AOP Extensions

```
extend project_driver_c (BITS_64'bus_speed) {

   event clock is only fall(clock_sig);

   function [63:0] transform_packet [](p:project_packet_c,
                                       endianness:endianness_t) is only {
      print p;
      print endianness;
      proceed();
      print result;
   };

   event data_sig_change is change(data_sig);
   on data_sig_change {
      print hex(data_sig);
   };
};
```

Overwrite the clock

Print packet and endianness before transformation and resulting array of 64 bit words

Print each value applied to the DUT's input data signal

Discipline and Expertise

"**With great power comes great responsibility**"

-Voltaire

- There are two major, but related, criticisms of AOP from OOP programmers
  1. OOP requires a more structured approach
     - Enforces more careful planning up front, which is better
  2. AOP languages result in "Spaghetti code"
     - Extensions are hard to manage/maintain

OOP is More Structured Therefore, Better Than AOP

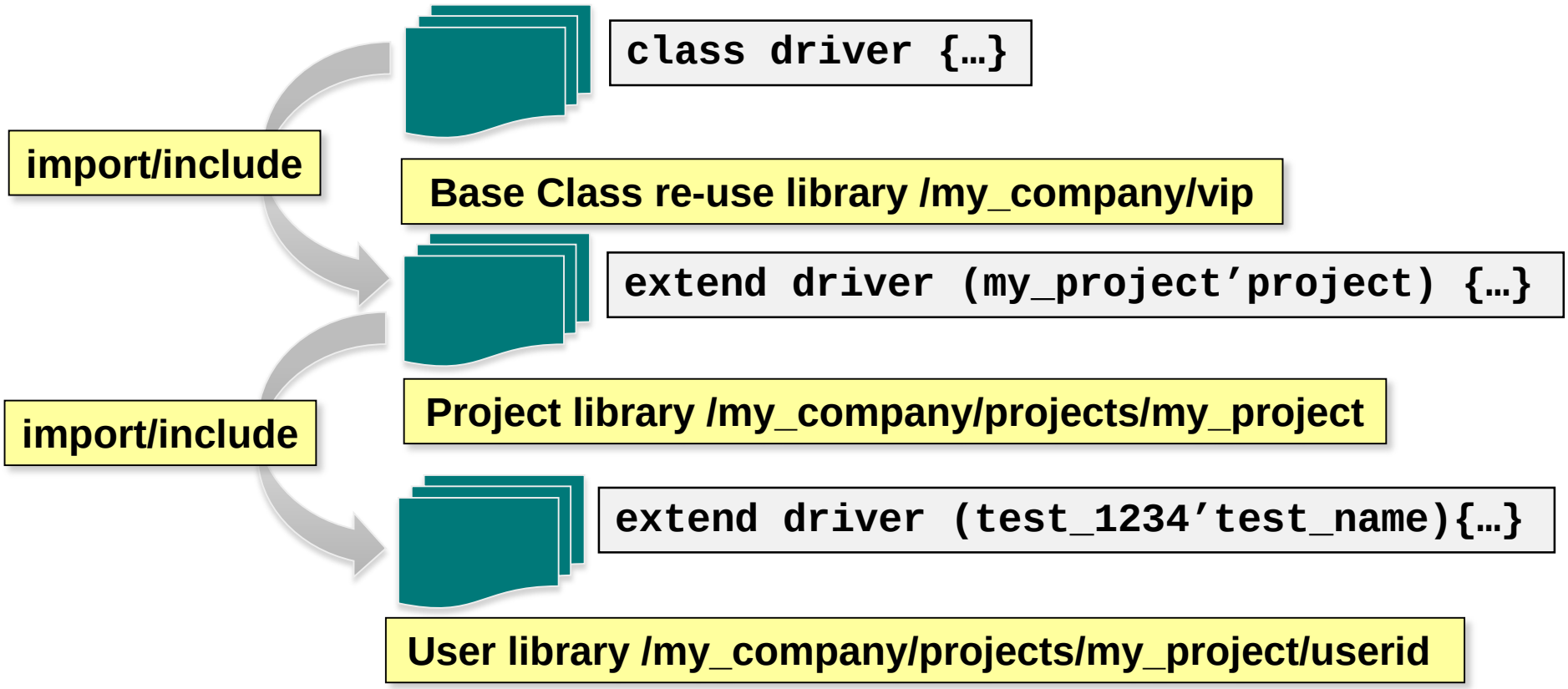
- AOP is a superset of OOP
  - Can be considered as OOP++
  - There is nothing in OOP that you cannot do with a capable AOP language
    - Though, as we have seen, the opposite is not true
- In the functional verification space, more structure does not equate to enhanced productivity
  - Cannot possibly implement an architecture of all needed features at project onset
  - Reduced flexibility, causing un-needed re-architecting

AOP Languages Result in "Spaghetti Code"

- For inexperienced programmers, this can be true
- AOP allows for new methodologies in code management
  - Traditional OOP: One class per file
  - AOP: Can break up files/functionality in many ways
    - One class per file
    - One class extension per file
    - One feature per file (many extensions to VE to support it)
    - Base class vs. project/user specific files
- Need to decide on a methodology that is right for you

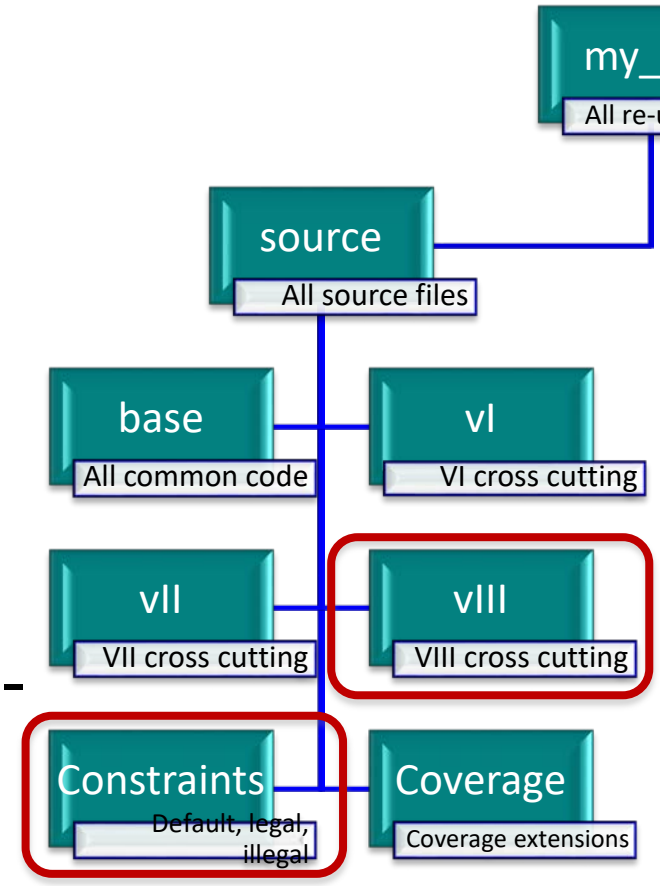
Reducing "Spaghetti Code"

- Files can be stored in several locations and assembled into the final via compilation order



Improving File Organization

- AOP allows for improved functional partitioning
- Can separate functionality based on any number of concerns
- Allows for greater re-use flexibility



Users of the VIP need only include the functionality they need when re-using

Example: Project only supports VIII packets, only need to include relevant files and constraints

Does AOP really deliver ?

- AOP is a relatively recent innovation which is still maturing
- Adoption within the realm of Software development is becoming more widespread with AOP being implemented using AspectJ for JAVA, Aquarium for Ruby, and Spring for .NET
- For an interesting example of a real-world case that gives a true sense of the power of AOP please see :
  - http://ramnivas.com/blog/?p=19

Summary and Recommendations

- For verification engineers using HDL languages today, OOP offers new, much needed functionality
- For verification engineers accustomed to using AOP languages, OOP is a **significant step backward**
- AOP is highly beneficial to our world of ever changing specs and requirements
  - Allows verif. engineers to keep up with pace of change
- While the SW industry advances towards AOP, our industry (who pioneered AOP) is walking away

Summary and Recommendations

- We recommend:
  - The relevant standards bodies, committees, vendors and verification community as a whole re-examine the overall benefits of AOP languages and create a roadmap for creatively re-adopting AOP
  - All avenues should be explored on the spectrum from leveraging existing mature languages to defining a next generation language to tackle tomorrow's verification challenges including HLS, SW Driven verification, MS, Formal