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

***Abstract*- In this paper, the concept of Aspect Oriented Programming (AOP) will be reviewed. The advantages of using AOP in the context of real world verification projects are explored including application to late churn development, flexible and efficient environment configuration, special purpose orchestration and synchronization, coverage closure and debug. Efficiencies and advantages with respect to code reduction and test case creation will be presented. The focus will be to present AOP as a complementary, beneficial and necessary superset of OOP. The functional superset that AOP brings is absolutely needed in our industry to address modern hardware verification needs.**

I. INTRODUCTION

For over twenty years, the power of Aspect Oriented Programming (AOP) has been applied successfully in the functional verification of real designs. AOP is a native feature of the ***e*** Hardware Verification Language (HVL) and is also supported by Accellera's OpenVera HVL. Currently, SystemVerilog is being adopted by many companies for verification; however, adoption of AOP has been deferred and/or bypassed in recent years during SystemVerilog's evolution. The challenges in verifying today's large and complex designs have steadily increased yet SystemVerilog, a key language in the verification space, has reduced functionality compared to languages two decades older. The time has come to have a closer look at the potential value of bringing back AOP to address modern productivity challenges.

A. What is AOP?

The following definition is excerpted from [1]:

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.

To explain this concept using a simple real-world example, consider a Verification Component (VC) that must drive a particular interface of a Design Under Test (DUT). This VC must support a variety of bus sizes (BITS_1, BITS_8, BITS_16, etc.). The bus size can be thought of as an "aspect". Once the user configures the bus size to be BITS_8 (8 bit bus), a number of things must happen within the VC in order to properly control its behavior. Some examples of such items are:

- Constraints must be applied to the configuration objects (to set clock speeds, bus endian-ness, etc.)
- The driving/monitoring protocol must be configured to drive/sample an 8 bit bus
  - Data chunks should be driven 8 bits at a time
  - Endian-ness must be considered
  - Loop indexes must be adjusted to accommodate the correct number of iterations
- The DUT registers must be configured properly such that the DUT is aware it is receiving 8 bit data

The above items can be thought of as "cross-cutting concerns". One simple change to the bus size aspect affects a number of concerns (configuration objects, driving/monitoring protocol, register programming, etc.) that cut across the VC in many locations. A key capability of AOP is the ability to dynamically insert or control behavior of a functional unit (concern) without the need to override and/or duplicate the code base. Modifications are non-intrusive and selectively compiled into the system as needed. This is both complementary and different from an Object Oriented Programming (OOP) approach in that it allows for

the dynamic application of polymorphic behavior without the necessary overhead, structure and hierarchy of one or more newly derived objects. It also avoids the need to provide heretofore unavailable and unanticipated hooks.

In AOP, the granularity for code modification is much finer than can be provided through an exclusively OOP approach. One key consideration is that hooks for changing the behavior of an object do not have to be defined up front when architecting an AOP based environment. Another key differentiator is the ability to better modularize and organize cross-cutting behavior through file and functional partitioning.

As applied to various implementations in the HVL arena, these, and other features mentioned later in this paper, translate into a highly flexible, code-efficient and powerful framework for arbitrarily changing the behavior of any object without the need and expense of architecting predefined hooks and at a fine level of control across varied scopes. The ability to keep pace with constantly changing requirements and specifications throughout a project is critical in order to deliver on time and AOP directly addresses this need.

B. History of AOP for Verification

If we look back at the evolution of verification languages, we find that the first language designed specifically for functional hardware verification was an AOP language, ***e***, created by Verisity Design Inc. in 1993. To the best of the authors' knowledge, the Vera HVL was created several years later by Sun Microsystems. Vera was created as an OOP language, but was later modified to handle AOP extensions. Because the AOP capabilities were introduced only after the OOP foundation was created, this led to limitations when compared to the functionality of ***e***. The early 2000's were a critical time for verification from an EDA vendor perspective, with each major verification tool provider providing its own proprietary language solution. Due to its rich feature set, the ***e*** language proliferated and Vera (minus the AOP extensions) was donated to the Accellera SystemVerilog initiative around the year 2002. Three years later, SystemVerilog was born. What is concerning is that, even in the early 2000's, the industry was moving toward AOP (due to the need for AOP features in our space); however, the SystemVerilog standard did not include them. It is the authors' opinion that bypassing the inclusion of AOP capabilities in the SystemVerilog language created a significant leap backwards in terms of functionality for advanced verification users.

It is the authors' understanding that several years later, in the 2012 revision of SystemVerilog, AOP features were proposed to the IEEE committee. The proposal was submitted fairly late in the approval process and, at the time, viewed as incomplete by committee members. The committee spent time discussing the proposal. However, there were many technical issues raised (enough to fill at least an entire meeting) and, due to a need to focus on content that had a high probability of making the 2012 revision, the proposal was not raised again.

The test bench constructs of SystemVerilog were created as an OOP language, built on top of a HDL design language. This understandably introduced inconsistencies in the resulting language. At this point in time, merging the existing constructs of SystemVerilog together with any new AOP extensions and outlining their interoperability would only lead to further inconsistencies and complications from a language design perspective.

II. BENEFITS OF AOP

A. Addressing Verification Architecture Challenges

The problem of architecting a verification environment for the complex devices of today is a daunting one for a variety of reasons, some of which we will cover in this section. Realistically, relying on purely OOP concepts often falls short due to language limitations. In the following sections, we will compare and contrast an ideal device development flow, where OOP may be well suited, to a realistic flow, where AOP concepts significantly enhance flexibility and productivity. These are not representative of all existing flows but the critical dimensions addressed can be considered universal nonetheless.

i. Device Development Flows

Take for example the device development flow described in Figure 1**.** This depiction is an idealized version of the familiar *Waterfall* development flow for software development projects.

With this idealized flow, up-front visibility into the entire set of device requirements as well as DUT architecture/interface is assumed and one can make the argument that OOP principles suffice in describing a suitably flexible and scalable verification environment that should satisfy the needs of verifying the requirements.

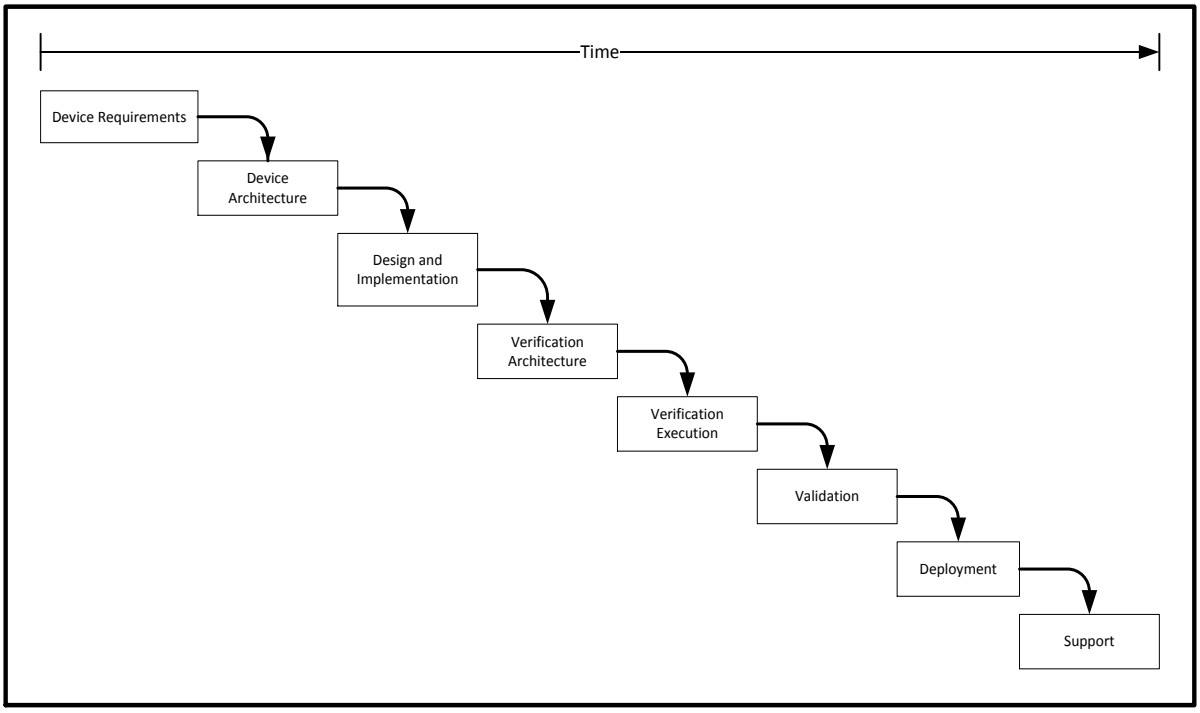

Figure 1 Sequential Development Flow

Realistically, the flow in Figure 1 almost never occurs in an actual project. Projects invariably experience iterations and churn at each phase in the above diagram, causing feedback loops that require significant and non-trivial changes to the verification environment architecture throughout the entire project. One must be prepared to support augmentation, pruning, and changes across various scopes as well as hybridized versions of the first iteration. A significant portion of the churn may be addressed through refinement and/or wholesale changes of the verification environment using traditional OOP polymorphism. Having access to AOP constructs provide an extra degree of freedom in solving the critical trade-off puzzle of time, resources, and quality. In a real-world roll-out of a complex device, the development flow will more closely resemble the modified *Waterfall* description shown in Figure 2**.**

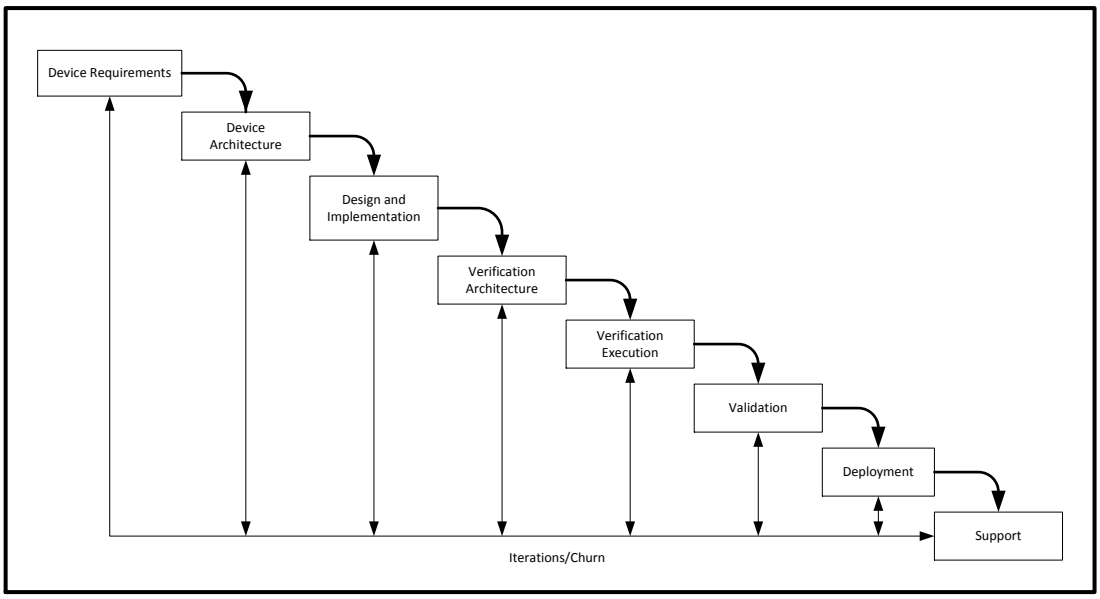

Figure 2 Development Flow with Iterations and Churn

To ensure maximum productivity, development of both the verification environment and the design often occurs in parallel. The resulting real-world development model is the parallel modified *Waterfall* approach as shown in Figure 3.

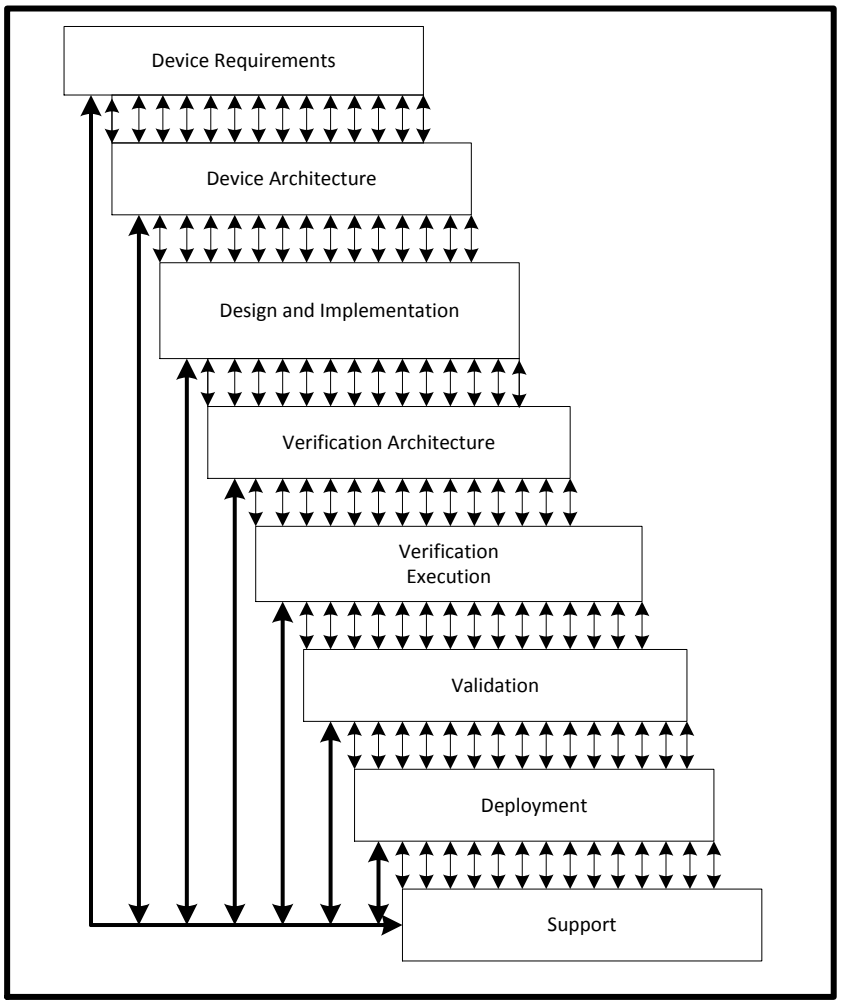

Figure 3 Parallel Development Flow with Iterations and Churn

If we compare this model to the original model of Figure 1, the prospects at arriving at even a first-order approximation of our final verification environment at project onset is very low. It is for this reason that the OOP paradigm often breaks down, causing significant re-work of the verification environment throughout the project. Any changes to the environment introduce increasing risk as the project progresses. Through its ability to introduce non-intrusive code changes, AOP is extremely well suited to address unexpected churn and cross-functional iterations.

AOP constructs give us a degree of agility in addressing the dimensions of both unexpected churn and parallel development concerns. Continuing the example introduced earlier in the paper, let's say that a new bus speed, BITS_64, was introduced midway through the project. Using AOP, we can create a single, standalone file that contains all of the needed functionality to support this new bus speed. The original code can remain intact; therefore, quality and stability is not compromised:

```
// File: bits_64_functionality

//extend the existing enumerated type to include the new value
extend bus_speed_t [BITS_64];

//extend the configuration class only for the BITS_64 bus_speed
//to add a new constraint to a previously declared constraint block
extend project_config_objects_c (BITS_64'bus_speed) {
   constraint clock_cg is also {clock_speed == serial_clock_frequency/64;};
   …
};
extend project_driver_c (BITS_64'bus_speed) {
   // modify the drive protocol to account for the new bus speed
   task drive_packet(…) is only {
      …
   };
};
extend project_monitor_c (BITS_64'bus_speed) {
   // modify the monitoring protocol to account for the new bus speed
   task receive_packet(…) is only {
      …
   };
};
```

Figure 4 Bus Speed Aspect Example

The above code would only affect the extended classes if the user configures the bus speed to be BITS_64. Also, a key thing to note here is, the above code would only affect the environment if the bits_64_functionality file was compiled along with the existing VC code. Assuming that the user has implemented a configuration scheme that allows for automatic propagation of the configuration parameters throughout the environment objects, the user would need only to set the bus_speed in the configuration object and all other VC objects would adjust accordingly.

Of course, one can argue that these issues can be addressed with a purely OOP approach; however, depending on the magnitude of the changes, the time and effort burden may result in a significant and unacceptable quality compromise.

ii. Legacy Verification Intellectual Property (VIP)

Leveraging existing VIP (developed both in-house and externally provided) of widely varying quality is a reality for many verification teams. Because these VIP's are typically comprised of legacy code, these teams often do not have the luxury of integrating components that are well-maintained, architected for re-usability, easy to configure, or adequate from a feature perspective.

In order to leverage the VIP for their own specific project, these teams need mechanisms for quickly integrating, configuring, and extending/trimming the functionality of legacy VIP's. AOP provides the capabilities to do this quickly. This is especially valuable when the VIP functionality to be leveraged is of too narrow/broad focus and does not meet the specific needs of the project. Layering AOP extensions on the verification environment, selected test cases, or a combination of both are all viable options.

iii Churn Example: Method-based Sequences

In order to understand the value proposition that AOP represents, a simple yet realistic example of churn is presented here. Elements of this example are necessarily simple but applicable nonetheless.

a. AOP Constructs

In the interest of keeping language neutral; we will be using our own set of imaginary constructs for an imaginary AOP based verification language. Let's say, within our environment, we have created a task used for performing a hard reset on our device. This task is part of a larger set of tasks that will be introduced as part of a generalized test flow later in this section. Consider the following class in Figure 5.

```
class my_test_c {
   …
   task do_hard_reset() {…};
};
```

Figure 5 my_test_c class

Next, let's say that, for tests running the DUT in BITS_64 bus_speed, the reset sequence looks slightly different. In order to easily extend only the tests running in BITS_64 mode, we must first identify an aspect (bus_speed) and the class of interest. Let us define an ***extend*** directive to accomplish this goal as shown in Figure 6. This is called an AOP ***Introduction***.

```
extend my_test_c (BITS_64'bus_speed) {
   …
};
```

Figure 6 Bus Speed Aspect for my_test_c class

Let us next define constructs required to specify behavior that executes before, after or in lieu of the execution of pre-existing tasks (Figure **7**). In AOP parlance, we would call each of these an ***Advice***.

```
extend my_test_c (BITS_64'bus_speed) {
   …
   // code will be executed before code loaded previously
   task do_hard_reset() is first {
      flush_driver_buffers();
   };

   // code will be executed after code loaded previously
   task do_hard_reset() is also {
      delay 10ms;
   };

   // code will replace code loaded previously
   // the proceed task will execute all previously loaded code
   task do_hard_reset() is only {
      …
      proceed(); // invoke previously defined functionality
      …
   };

   // introduce a new task that will be called during
   // a hard reset
   task flush_driver_buffers() {
      …
   };
};
```

Figure 7 AOP Advices

We can see from the above example the relative ease in which we can use AOP to modify existing methods to tweak functionality slightly to meet the specific needs of a project or user.

b. Problem Description

As mentioned earlier, consider a class encapsulation of a modular and step-by-step test flow consisting of a number of steps that every test must execute (Figure 8). We view each step as represented by a method hook that can be customized through traditional OOP concepts of polymorphism and can be applied using a factory override.

We make the assumption here that our test flow started out simple and well defined.

```
class my_test_c {

   string test_name;
   integer instance_id;

   task do_reset {…};
   task do_config{…};
   task do_training{…};
   task do_traffic{…};
   task do_checking{…};
   task do_cleanup {…};

   task run_test {
      do_reset();
      do_config();
      do_training();
      do_traffic();
      do_checking();
      do_cleanup();
   };
};
```

Figure 8 my_test_c example class

At an advanced point in the verification plan execution it becomes apparent that the test flow must be changed, pruned, or augmented to satisfy a specific verification requirement.

c. Problem Solution

Based on our proposed subset of AOP constructs, one can demonstrate this problem is readily and easily solved. To decrease risk and increase the utility of this approach, let us assume we have a mechanism (an aspect) to easily differentiate the test case context. For maximum control, it is also desirable to have an aspect to discriminate between different instances of the same object (Figure 9).

```
// conditionally extend only tests whose test_name string
// field set to "corner case 1" and whose instance_id
// integer field is set to 1
extend my_test_c ("corner_case 1"'test_name 1'instance_id) {
   task wait_for_clock{…};

   // add a new step before an advice
   task do_reset() is first {
      wait_for_clock();
   };

   // prune a step to remove functionality completely
   task do_training()  is only {};

   // introduce new tasks
   task do_custom_training {…};

   //change/augment the ordering of task calls
    task run_test() is only {
      do_reset();
      do_config();
      do_traffic();           // send traffic before executing training
      do_reset();             // add extra reset
      do_config();            // configure again
      do_special_training();  // do customized training
      do_reset();             // add extra reset
      do_checking();
      do_cleanup();
   };
```

Figure 9 Method Based Sequence Churn

It's worth noting that the above example, and many AOP concepts, can be solved using OOP functionality. However, the non-intrusiveness, scalability, flexibility, and code efficiency of the AOP approach has many advantages.

B. AOP for Verification Closure

i. Verification Flow

Let's examine a more detailed breakdown of the verification pieces of the development flow discussed previously. Consider the flow chart in Figure 10**.**

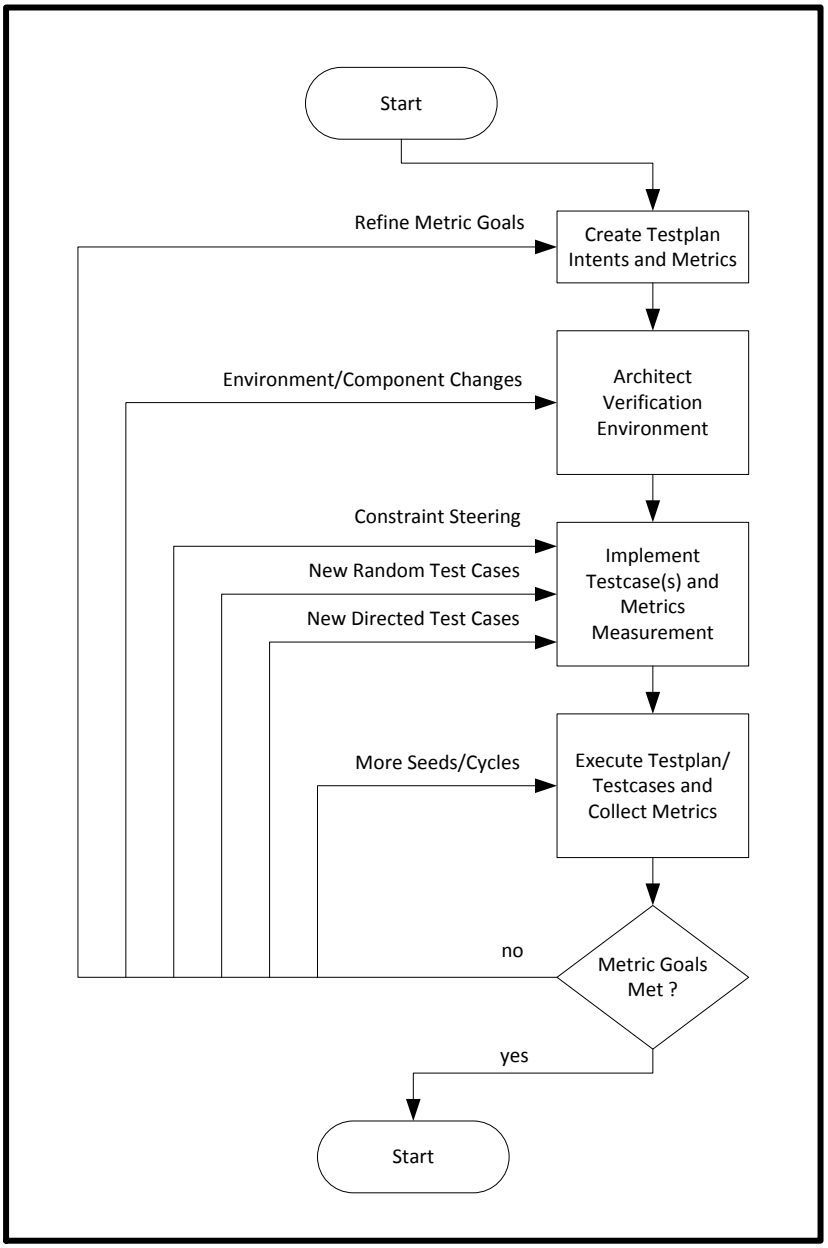

Figure 10 Verification Flow

The key complexity not represented in this flow is what criteria engineers use to move from one arc to another. Traditionally this decision would be based on an intimate knowledge of the verification intent being addressed, the controllability of the verification environment, the implementation of the design and the resource constraints.

For the sake of this discussion, consider an alternate view of the above flow shown for a specific intent (Figure 11). This diagram factors in the element of identifying a target coverage goal and a time/effort budget to achieve the goal.

The necessity to move from one arc to another is largely dictated by the capacity of the existing verification environment and test cases to effectively address the complete set of verification intents. Realistically, we know from our verification architecture discussion in section II.A., it is generally improbable to achieve the required level of completeness without significant churn.

In terms of effort, resources, and schedule risk, a key point in this flow is usually the critical decision of whether or not to prune a hard-to-achieve verification metric (such as a coverage goal) or write a complex directed test case to hit the metric being measured. Let us assume, for the sake of this discussion, that across all of our verification intents we cover 90% of the definable metrics we've described to measure completeness of our verification problem before hitting the directed test case decision threshold. Based on personal experience of the author, this is generally only achievable with significant verification planning discipline and verification architecture expertise.

A less apparent nuance that gets lost in analyzing the schedule risks of coverage closure but can have significant repercussions on effort, schedule, and productivity is *when* we make the decision to prune or do directed testing. Experience and hindsight indicates that timing of this critical decision point is usually executed later than it should be and is seldom optimal.

It is also worth noting that a key risk is the decision to prune coverage goals because they are too complex to achieve within the context of the existing VE, test cases and a resource profile. A pragmatic approach to these types of verification problems is to mitigate risk by reducing the coverage burden but not eliminating it. In other words, doing something is better than doing nothing.

What is the nature of intents or metrics that tend to arrive at this critical juncture? These types of intents usually fall into a few categories. One category of metric hole follows intuitively from the verification environment development model we've discussed. Specifically, these are out-of-scope from the verification environment development. They typically consist of verification requirements that have been added late in the verification process. The root cause of these "missed" requirements can be myriad ranging from human error to product redefinition. Their omission may or may not have been avoidable, however this is of little consequence once we've reached this point.

Another class of hard-to-hit coverage arises due to complex interactions between elements of the design that present challenges to controllability and visibility. A good example of this type of requirement is one in which an improbable but real scenario must be reproduced that involves complex synchronization, orchestration, and alignment of configuration amongst several components in the existing test bench. Sometimes these types of scenarios only surface in a lab environment on a test chip. These are described as corner cases that require the alignment of Earth, Moon, and Sun, so to speak.

This class of verification problem presents complexity in simply conceiving of the verification requirement. Executing only the set of conceived verification intents is often inadequate and poses significant risk of an escape. Trying to address these problems within the scope of the verification environment using OOP techniques is often cumbersome and can quite easily require a tremendous amount of effort and expertise.

It is at the critical "discard or directed" juncture in the verification closure flow that AOP is particularly well-suited. The control and flexibility allowed by AOP lowers the bar in terms of burden of effort to achieve these types of verification metrics. The availability of this option also lends itself to "divide and conquer" on these holes. Also, through creative application of AOP techniques, one can often flexibly adjust the intent of the original verification requirements to provide a compromise solution that appreciably mitigates the risk of an escape. Without access to this powerful tool, options become limited. Considering a fixed resource profile, any factor which moves us closer to the "directed" vs. "discard" end of the spectrum will result in higher level of product quality.

We can see this effect graphically represented in Figure 11 whereby the time and effort efficiency is inversely proportional to the slope of the individual programming paradigms (OOP and AOP). For any individual verification project one might have to manage hundreds or even thousands of verification intents. Even for a well-architected VE and set of test cases that achieves our 90% target we can expect to cross the directed threshold for a large volume of intents of which many will fall into the hard-to-hit camp. Even a marginal improvement in coverage closure efficiency using AOP will have an appreciable impact on project schedule and quality.

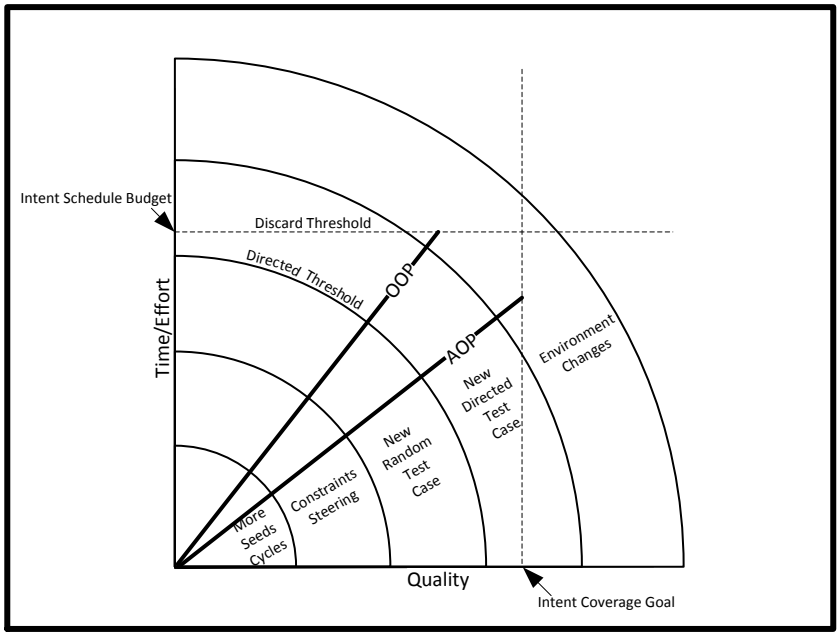

Figure 11 Verification Intent Closure

As a simple example to illustrate AOP's ability to modify our defined coverage model let's say that, for a project, there is a good amount of coverage already implemented within one of the VIP components we are re-using. Though most of the coverage is relevant, our project, like all projects, has special needs. We can extend pre-existing coverage metrics and directly add/remove or refine as follows:

```
// extend the packet class and add/prune/modify
// as needed to suit our own project-specific needs

extend project_packet_c (MY_PROJECT'project) {

   covergroup packet_type @transmit_packet is also {

      // add a new coverpoint for new functionality
      special_error_type: coverpoint error_type;

      // remove the LARGE packet_type from coverage as
      // our project does not support it
      packet_types: coverpoint packet_type is also {
         value LARGE is only omit;
      };

      // redefine a pre-existing coverage point to
      // have new project-specific bins
      preamble_size: coverpoint preamble_size is only {
         bins small = {0,7};          // was previously 0,16
         bins medium = {8,15};        // was previously 17,23
         bins small = {16,23};        // was previously 24,35
      };
   };
};
```

Figure 12 Coverage Pruning using AOP

ii. Verification Closure: Temporal Orchestration Example

A simple demonstration of applying AOP principles to effect temporal orchestration across different test bench objects is presented here. Elements of this technique may be employed to directly reproduce specific stimulus or it may be used in conjunction with functional coverage to enhance the probability of exposing a desired scenario.

a. Problem Description

Two test bench components described in Figure 13 are generally independently configured and controlled but need to be aligned in order to realize or increase the likelihood of exposing and observing a specific scenario. In this example, a new data squelching feature has been added very late in the project. If 50 packets in a row (difficult to achieve this scenario using completely randomized data) are sent into the DUT on the parallel interface with bit 1 of the reserved header field set to 1'b1, then this must trigger the output serial interface to squelch its output data (continuous stream of 1'b1) within 100 clock cycles. We have two environment objects that need to interact in order to verify this functionality.

```
//Input parallel monitor that is collecting packets
class project_parallel_monitor_c {
   task receive_pkt() {…};
};

// Output serial monitor whose data must be squelched
class project_serial_monitor_c {
...
};
```

Figure 13 Objects requiring synchronization

b. Problem Solution

At an advanced stage of the verification process, the need for tight coordination has presented itself or is discovered after architecting and implementing the components involved. The requirement is fairly narrow (i.e. one directed test case) and re-architecting the temporal framework of these components is effort prohibitive. In the code below, we use the power of AOP to simply extend the base classes of the needed environment objects to add the functionality needed from an external file.

```
// extend the parallel monitor to detect the squelch condition
// and then trigger the start of the condition within the
// serial monitor
extend project_parallel_monitor_c ("squelch_case"'test_name) {

   //50 packets in a row with reserved[1] set to squelch data
   integer num_pkts_to_squelch = 50;
   integer squelch_pkt_counter = 0;

   task receive_pkt() is also {
      if(pkt.header.reserved[1] == 1'b1) {
         squelch_pkt_counter++;
      }
      else {
       squelch_pkt_counter=0;
      };

      if (squelch_pkt_counter == num_pkts_to_squelch) {
         //trigger squelch event in serial monitor
         ->top.serial_agent.monitor.input_squelch_occured;
         squelch_pkt_counter = 0;
      };
   };
};


// extend the serial monitor to check for the squelching
// of data appropriately
extend project_serial_monitor_c ("squelch_case"'test_name) {

   //20 cycles of continuous 1'b1 will mean that output data is squelched
   event output_squelch_detected is repeat[20](data == 1'b1);
   event input_squelch_occured;
   integer squelch_cycles = 100;

   // check to ensure that, once squelch condition has been received, we detect
   // squelched data within 100 cycles of the clock event
   expect @input_squelch_occured -> [0:squelch_cycles-1] @output_squelch_detected else {
      error("Squelch out of range ... did not occur within ", squelch_cycles," cycles");
   } @clock;
};
```

Figure 14 AOP Synchronization Solution

Keep in mind that this example is simplified. In larger environments, there may be many instances of the parallel/serial monitors. In reality we may have to synchronize many different objects of disparate architectural quality across many different scopes and levels of hierarchy. The key is that AOP gives us a toolset to discriminate and weave these concerns together in a code-efficient manner.

C. Debug

As mentioned previously, AOP allows the user to tweak and tune the environment in a non-intrusive manner, which can help to dramatically improve debug turnaround time. From within a standalone file compiled on top of the Verification Environment (VE), the user can non-intrusively extend VE classes, methods (functions/tasks), enumerations, events and many other items to include additional debug information or to overwrite existing functionality completely to test out new fixes to the code.

Example #1:

Let's say that, after our initial implementation of the driving code for the new BITS_64 driving protocol (referenced earlier) within our driver class, we noticed that the data for each packet was not being presented correctly to the DUT. A simplified version of our driving code may look something like this:

```
extend project_driver_c (BITS_64'bus_speed) {
   …
   // modify the drive protocol to account for the new bus speed
   task drive_packet(p: project_packet_c) is only {
      …
      packet_as_64_bit_words = transform_packet(p, config.endienness);
      req_sig = 1'b1;       // request data to be sent to DUT
      @dut_ready;           // wait for DUT to be ready to receive data
      req_sig = 1'b0;
      data_valid_sig = 1'b1;// indicate valid data on bus
      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;// indicate valid data no longer on bus
      …
   };

   // function takes in a packet and outputs an array of 64 bits words
   function [63:0] transform_packet[] (p: project_packet_c, endienness: endienness_t) {
   …
   };
};
```

Figure 15 Driver Code Example

From a small extension to the environment, we could debug the driving protocol in a number of ways. For example, we could print the value applied to the data_signal every time there was a change, or print the packet before and after the transformation to 64 bit values through the code below:

```
extend project_driver_c (BITS_64'bus_speed) {
   event data_sig_change is change(data_signal);  //fires upon a new value
   on data_sig_change {
      print hex(data_signal);  // on block executes every trigger event
   };

   // debug the transformation on the packet to ensure
   // that the endienness is not the issue
   task transform_packet(p:project_packet_c,endienness: endienness_t) is only {
      print p;       // print packet before the transform
      proceed();     // execute the previously loaded code
      print result;  // print the resulting array of 64 bit values
   };
};
```

Figure 16 AOP Debug Instrumentation

Example #2:

Let's say that, within our environment, there was a queue/array being used by a scoreboard when checking for packet matches between input and output packets. (Figure 17):

```
class project_scoreboard_c {
   event clock is rise(clock);  //define clocking event
   project_packet_c packet_list [];
   …
   task compare(packet: project_packet_c) {
      q_packet: project_packet_c = packet_list.pop_front();
      if (q_packet.payload !== packet.payload ) {
        error("Mismatch!!  Input packet: ", packet, " and packet_list: ", q_packet);
   };
};
```

Figure 17 Scoreboard Example

If a mismatch on the packet payload occurred, the user could expect to see a simple debug message with the two packets being printed to the screen. For debugging this issue, one might wonder if **any** packet in the queue actually matched the input packet. In a short extension from within a file compiled with the environment, the user could include the following simple code to print all items in the scoreboards packet list to the screen, each time the compare() method was called (Figure 18).

```
extend project_scoreboard_c {
   task compare(packet: project_packet_c) is also {
      foreach(packet_list[i]) {
         print it;  //print each item to the screen
      };
   };
};
```

Figure 18 Compare Method

In another example, let's say that you suspected that the scoreboard's clocking event may have been defined incorrectly. The user could test out a new fix using the following few lines of code loaded along with the other simulation files (Figure 19):

```
extend project_scoreboard_c {
   event clock is only fall(clock); //overwrite incorrect clocking event
};
```

Figure 19 Clock Override

Using the power of AOP, the user can add debug information through simple extensions, all done from private files, without affecting any other user on the project and with no changes to the base code files.

Now, there are situations that occur where an RTL bug is encountered that may not be fixed for some time due to, let's say, overload on the RTL designer. Using AOP extensions, the user can create a small extension to the environment that can be loaded by all users to steer the environment away from the bug until the fix is in place. This is typically called a bug bypass. For example, let's say the DUT cannot handle the situation where, in high speed mode, back to back packets whose payloads are larger than 1000 bytes are sent. This scenario can be temporarily excluded from stimulus generation by including the following code within a file and loading it along with the environment until the bug is fixed (Figure 20).

```
extend project_base_sequence_c {
   constraint bug_bypass_b2b_large_pkts is {
      (config.mode == HIGH_SPEED && prev.payload.size() > 1000)
                           => cur_item.payload.size() <= 1000;
};
```

Figure 20 Bug Workaround

Of course, it should go without saying that any bug workaround should be removed before the project tapes out. If you are using functional coverage as a metric, bug workarounds should create holes in your coverage, flagging to users that they should be removed.

III. DISCIPLINE AND EXPERTISE

Many critics of AOP will make the claim that OOP requires a more structured approach and, hence, is better than AOP. AOP is a superset of OOP and can be considered, really, as OOP++. There should be nothing in OOP that you cannot do with a capable AOP language. In the functional verification space, more structure does not necessarily equate to enhanced productivity. The famous Voltaire quote "With great power comes great responsibility" could not be truer when it comes to AOP languages. Some critics of AOP also claim that the collection of extensions to base classes that AOP allows will result in "spaghetti code" that is hard to follow, organize, debug and maintain. These are valid concerns and, for a novice code developer; there may be some truth to that statement. However, any experienced programmer should realize both the power that AOP provides and, as a result, the need to set appropriate methodology in place to harness that power. In fact, with more and more software languages adopting AOP features, an entire development movement named Aspect Oriented Software Development (AOSD) [5] has emerged around how to best handle code scattering, tangling and other potential issues.

One simplified, yet effective, way to manage your project files is outlined here; however, there are surely many other ways.

- ⇨ **Re-usable portions**: These files contain functionality that should be present for any user of the code. They should reside within a re-use directory accessible by all project teams. For example, if

you were building a USB traffic generator, you would want all of the USB features of the USB standard implemented within the re-usable code and stored for all to use.

- ⇨ **Project specific portions**: These files contain functionality that is specific to your particular project. They should reside within the directory structure of your project. These files can import the re-usable code, and then modify that code, through extensions, to meet the needs of your specific project. For example, your project may not support certain transfer types in the USB specification. Using AOP, you can add extensions to the transfer class and add a constraint to never send specific types. Also included here are your verification environment files as well as project specific verification components.

- ⇨ **User specific portions**: These files contain functionality specific to a particular test writer. Individual testers may add extensions to test out a new fix to the verification environment, debug a scenario, bypass a bug or steer the environment towards hitting new scenarios to fill coverage holes.

All of the above code should reside in separate directories to provide clear differentiation of functionality and an intuitive partitioning scheme. In addition to file management, it is recommended that users take advantage of linting technologies to flag problematic code early in the project, before they have had a chance to propagate. Linters can be used to flag items such as too many extensions to a particular class/method, too many extensions in one file, etc. Some AOP languages also possess the powerful concept of reflection which, among many other benefits, allows the user to write their own set of custom linting properties for their project to enforce very specific style/performance guidelines.

IV. SUMMARY AND RECOMMENDATIONS

The SystemVerilog language was born through a merger of multiple language constructs. This resulted in a quirky and, at times, inconsistent single language for both design and verification. For verification engineers previously using Hardware Description Languages (HDL's) for verification, the verification constructs added brought new, much needed functionality that allows for significant productivity enhancements. However, for verification engineers accustomed to using AOP verification languages, SystemVerilog presents users with serious limitations. At a time when the productivity challenges of verification are intensifying, one might argue that we've taken a significant step back by letting AOP fall between the cracks.

AOP is highly beneficial and directly applicable to our world of ever-changing specifications and features and allows verification engineers to remain productive and keep up with the pace of change. At one point, our industry was on the forefront of advancing the AOP paradigm in order to address a broad spectrum of verification-centric challenges. While we've let AOP fall by the wayside, the software industry, as a whole, has continued to recognize the benefits and accelerated the rate of adoption to conceive of modern languages and methodologies.

From a user's perspective, we are in dire need of improvements to address the verification gap. This paper's recommendation is for the relevant standards bodies, committees, vendors and verification community as a whole to examine the overall benefits of AOP languages and to 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. Such an approach presents the added opportunity of employing a holistic view that could also consider other impending concerns such as high level synthesis, metrics collection management, software driven verification, analog mixed signal verification, and formal verification.

REFERENCES

[1] Wikipedia,"Aspect-oriented programming", 16 December 2014, http://en.wikipedia.org/wiki/Aspect-oriented_programming
[2] T.Tsai, "Aspect Oriented Support in SystemVerilog", EDA Industry Working Groups, IEEE DASC P1800, 16 December 2014, http://www.eda.org/sv-ieee1800/Meetings/2010/February/Presentations/Tony%20Tsai%20Presentation.pdf
[3] Synopsys Inc., "OpenVera LRM", August 2013
[4] Tutorialspoint, "SDLC Waterfall Model", 16 December 2014, http://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm
[5] Wikipedia, "Aspect-oriented software development", 9 February 2015, http://en.wikipedia.org/wiki/Aspect-oriented_software_development