2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Unleashing Portable Stimulus Productivity with a PSS Reuse Strategy

M. Ballance
Mentor, a Siemens Business
8005 SW Boeckman Rd
Wilsonville, OR, 97070

INTRODUCTION

Creating sufficient tests to verify today's complex designs is a key verification challenge, and this challenge is present from IP block-level verification all the way to SoC validation. The Accellera Portable Test and Stimulus Standard (PSS) promises to boost verification reuse by allowing a single description of test intent to be reused across IP block, subsystem, and SoC verification environments, and provides powerful language features to address verification needs across the verification levels and address the specific requirement of verification reuse. However, much as powerful object-oriented features in the Java and C++ languages didn't automatically result in high-quality reusable code, the PSS standard's language features on their own do not guarantee productive reuse of test intent. Judiciously applied, reuse of design IP and test intent can dramatically reduce rework and avoid mistakes introduced during the rework process. In addition, just as reuse of design IP accelerates the creation of new designs, reuse of test intent accelerates the creation or new test scenarios. However, effective reuse of test intent requires up-front planning, in the same way that reuse of design IP or software code does. Without a well-planned process, reuse can backfire and require more work without providing proportionate benefits. This paper will help you to design a PSS reuse strategy that matches the goals and profile of your organization, and maximizes the benefits you receive by adopting PSS.

ANATOMY OF A PORTABLE STIMULUS DESCRIPTION

The PSS language was designed with the requirements of test intent reuse, and automated test creation in mind. The requirement to allow test intent to be reused across a variety of very different platforms drove the PSS language to enable a clean and clear distinction between test intent and test realization, as shown in Figure 1. In a PSS description, test intent specifies the high-level view of *what* behavior is to be exercised. PSS test intent is captured in a declarative manner. Declarative descriptions, as we've seen from the use of the declarative constraint description in SystemVerilog, lend themselves very nicely to reuse and automation.

Both of these requirements are well-served by declarative language features. Declarative languages deal with the *what* rather than that *how* by specifying rules that bound the legal space of *what* can happen. If you've used SystemVerilog constraints, you've used a declarative language to specify rules for legal stimulus data values. The PSS language extends the data-centric declarative description that SystemVerilog provides to the scenario level, allowing rules to be captured that not only specify data relationships, but also specify temporal relationships between scenario elements called *actions*.

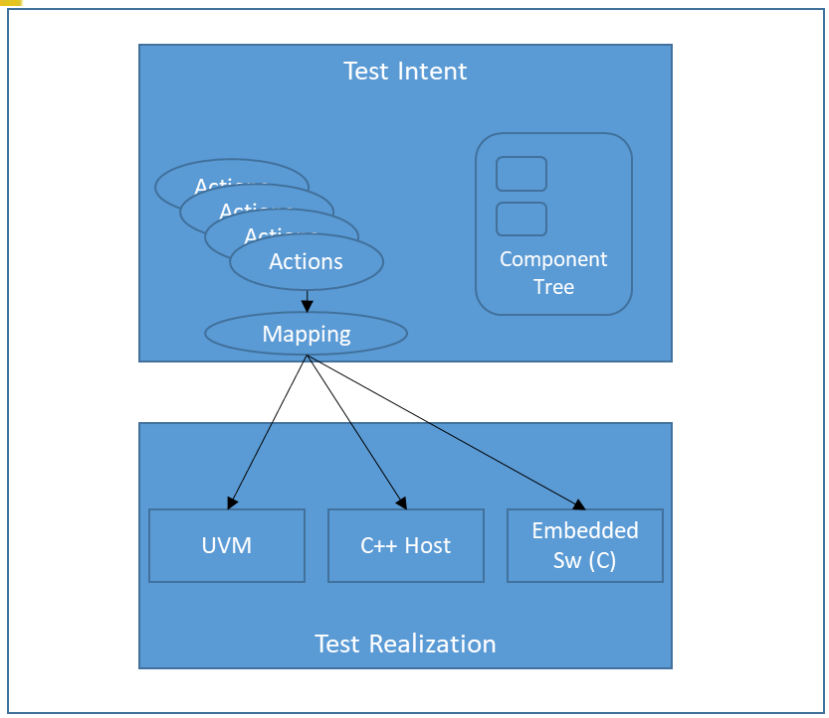

Figure 1 - Anatomy of a Portable Stimulus Description

Having a high-level declarative description isn't sufficient on its own, however. Ultimately, tests need to interact with the design being verified at the much lower level of registers and interrupts. Test realization is the code that interfaces between the high-level test intent and the lower-level details of the target platform. This code has a much lower need to enable automation, and verification environments often have significant test-realization code available already that can be leveraged. As a consequence, test realization code for portable stimulus test intent is nearly always implemented in existing imperative languages, such as SystemVerilog, C++, or C.

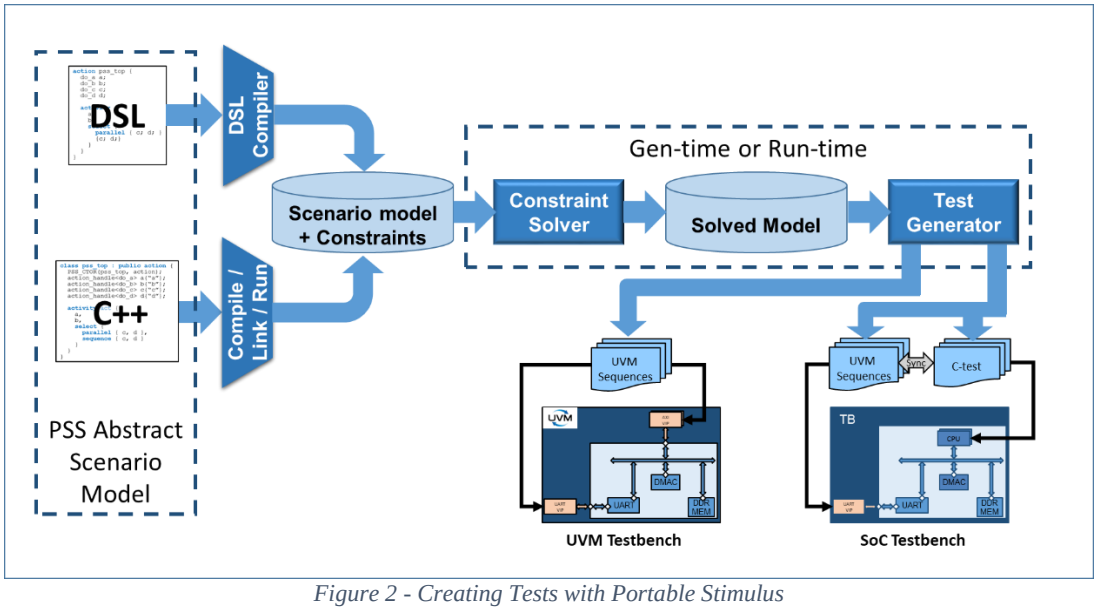

Figure 2 - Creating Tests with Portable Stimulus

Ultimately, of course, the purpose of a portable stimulus description is to generate tests that can be run against the design. Figure 2 shows a typical tool flow for a portable stimulus description. In the case of a host-based simulation-type environment, the PSS description will often be executed on-the-fly as the simulation runs. In this case, the portable-stimulus engine can be seen as providing constraint-solver functionality. In the case of an SoC environment, driven by tests running on the embedded processor, the PSS description will invariably be executed ahead of time to generate a set of simple tests that can be efficiently executed on the embedded processor.

In both cases, a key aspect of portable stimulus is the separation between the high-level test intent and the specific tests generated by tool automation.

THREE MEANINGS OF PORTABLE

As you approach designing your Portable Stimulus application strategy, it's useful to consider three meanings of Portable, and how each of these meanings factors into your current and future plans for applying Portable Stimulus.

Vertical reuse is what often comes to mind when thinking about portable test intent. The concept here is to enable test intent to be developed early – typically at the IP block level – and reused across the verification flow from subsystem to system level. Vertical reuse of test intent boosts the productivity of creating test scenarios at the subsystem and SoC level with a robust library of reusable content developed at IP block and subsystem level. Reuse of test intent and realization dramatically reduces the amount of rework required at subsystem and SoC level, also reducing the number of bugs introduced at these levels due to rework. While the benefits of vertical test intent reuse are impressive, implementing this sort of reuse requires significant organizational commitment due to the requirement that IP development teams produce reusable test intent for downstream teams to use. Portable stimulus descriptions will need to be created for existing IPs.

Horizontal reuse with portable stimulus enables test intent reuse across projects where the design being verified is a variant of a design previously verified with portable stimulus. The declarative nature of a portable stimulus test intent description dramatically simplifies the task of adjusting the functionality that needs to be verified by adjusting the rules captured in the test intent instead of manually inspecting and updating a suite of directed tests.

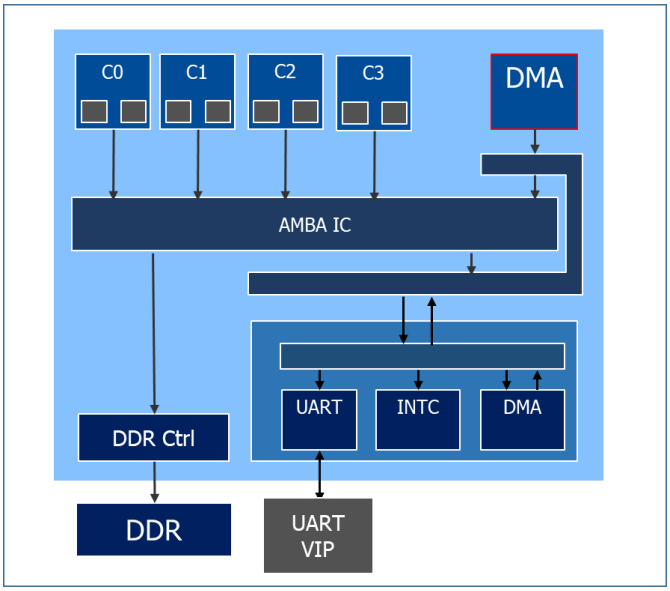

Figure 3 - Horizontal Reuse Example

Consider the example shown in Figure 3. Different variants of this SoC may contain different numbers of DMA engines. With a suite a directed tests, we would need to inspect and update all of the directed tests to ensure that they tested the SoC with the appropriate number of DMA engines. With a declarative portable-stimulus description, we

can simply adjust the rules to capture the available number of DMA engines, and re-generate a suite of tests that will test the SoC.

The final meaning of portable may seem just a bit counterintuitive: portability of test techniques. Consider SystemVerilog constrained-random testing. This technique, and the language supporting it, have been very valuable at raising verification productivity and quality. However, these techniques are still largely only available in simulation-based environments for verification of designs in Verilog and VHDL. These techniques aren't available in environments for verifying C++ designs for use with high-level synthesis (HLS). These techniques are also unavailable for creating embedded software tests, because embedded systems are typically too resource-constrained to meaningfully run a full SystemVerilog simulator and solver.

Portable stimulus makes test techniques, such as automated constrained-random test creation portable across a wide variety of target environments – from host-based environments, such as simulation and C++ verification environments, to resource-constrained embedded systems. This ability to use the same advanced verification techniques in environments where these techniques were not previously available may be reason enough to adopt portable stimulus – quite independent of the other types of portable previously described.

It's important to consider these three aspects of portability as you craft your PSS adoption strategy and decide which of these aspects of portability are significant to your organization, and the relative priority of those that are important to your organization. This prioritization will help your organization focus resources on enabling the aspects of portability that will bring the most benefit.

EXAMPLE OVERVIEW

A very simple example will be used across the balance of this paper to explain concepts. The SoC-level design, shown in Figure 3, contains a quad-core RISC-V processor, a peripheral subsystem, and several other controllers.

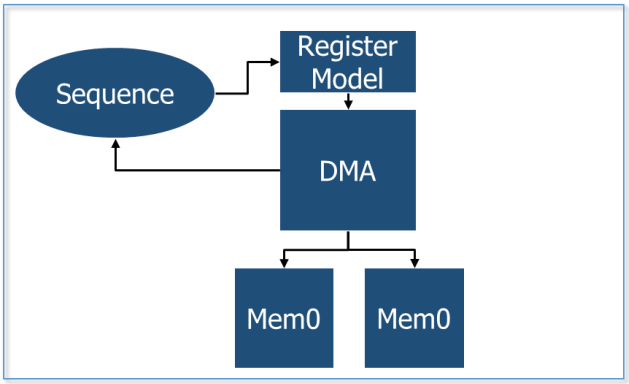

Figure 4 - DMA Engine Block-level Environment

We will look at the DMA IP at the block level. Figure 4 shows a block diagram of the block-level verification environment. We will also look at the subsystem-level design that incorporates the DMA engine along with a UART and an interrupt controller (shown in Figure 5).

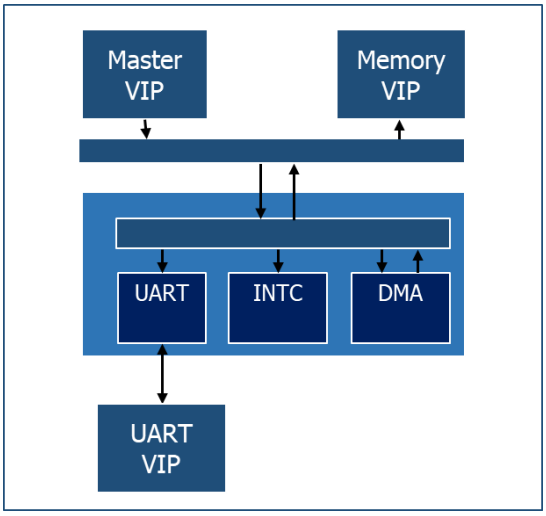

Figure 5 - Peripheral Subsystem Verification Environment

Identifying Existing Reuse Opportunities

After determining which portability aspects of portable stimulus make the most sense to pursue, it's time to take inventory of the elements you require to implement portable stimulus descriptions. It's also time to take inventory of the assets already available within your organization. Organizations have a wealth of information on the design and verification environment that can be used to in implementing a portable stimulus environment.

Constraints

PSS descriptions are heavily constraint-based, since they are declarative specifications. As a consequence, existing descriptions that are also declarative can often be converted into a PSS format and leveraged in creating PSS test intent.

SystemVerilog constraints are a good source of constraints to jump-start a PSS description. The format of SystemVerilog constraints has sufficient similarities to PSS constraints that reuse is often as simple as copying and pasting SystemVerilog constraints into the PSS model. Typical targets for reuse here are configuration classes that specify the rules for configuring IP and subsystem operation modes.

Constraints are often in a form that isn't immediately recognizable. Think about a spreadsheet that specifies the memory map for a SoC. With a little bit of work, this information that has been captured in a machine-readable format can easily be converted into PSS constraints to specialize accesses targeted to different memory regions.

Test Realization

Existing environments have a wealth of test realization, though often in a form that needs to be modified a bit to work with a PSS description.

In UVM environments, look for utility sequences that perform simple operations on an IP: setting its configuration, performing an operation, etc. Sometimes these sequences are created with random constraints and variables. In other cases, tasks are provided with arguments to control the different operation modes. In both cases, this test realization can easily be leveraged by a PSS description.

```
task init single transfer(
    int unsigned            channel,
    int unsigned            hw ctrl,
    int unsigned            src,
    int unsigned            inc src,
    int unsigned            dst,
    int unsigned            inc dst,
    int unsigned            sz,
    int unsigned            trn sz
    );
    wb dma ch ch = m regs.ch[channel];
    uvm status e status;
    uvm reg data t value;


    // Disable the channel
    ch.CSR.read(status, value);
    value[0] = 0;
    ch.CSR.write(status, value);

    // These registers are volatile. Read-back the content
    // so the register model knows to re-write them
    ch.A0.read(status, value);
    ch.A1.read(status, value);

    ch.A0.write(status, src);
    ch.A1.write(status, dst);

    ch.AM0.write(status, 'hfffffffc);
    ch.AM1.write(status, 'hfffffffc);
```

Figure 6 - SystemVerilog Test Realization Reuse

Figure 6 shows a code snippet from an existing SystemVerilog task within a virtual sequence that is used to setup the DMA engine to perform a transfer on a given channel. This code could be leveraged with a PSS model to interface with the DMA engine. Note that, because this is UVM, this task uses a UVM register model to access registers within the DMA engine.

```
void wb_dma_drv_init_single_xfer(
        wb_dma_drv_t        *drv,
        uint32_t            ch,
        uint32_t            src,
        uint32_t            inc_src,
        uint32_t            dst,
        uint32_t            inc_dst,
        uint32_t            sz
        ) {
    uint32_t sz_v, csr;
    fprintf(stdout, "begin xfer: drv=%p ch=%d\n", drv, ch);
    fflush(stdout);

    csr = WB_DMA_READ_CH_CSR(drv, ch);

    csr |= (1 << 18); // interrupt on done
    csr |= (1 << 17); // interrupt on error
    if (inc_src) {
        csr |= (1 << 4); // increment source
    } else {
        csr &= ~(1 << 4);
    }
    if (inc_dst) {
        csr |= (1 << 3); // increment destination
    } else {
        csr &= ~(1 << 3); // increment destination
    }

    csr |= (1 << 2); // use interface 0 for source
    csr |= (1 << 1); // use interface 1 for destination

    csr |= (1 << 0); // enable channel

    // Setup source and destination addresses
    WB_DMA_WRITE_CH_A0(drv, ch, src);
    WB_DMA_WRITE_CH_A1(drv, ch, dst);

    sz_v = WB_DMA_READ_CH_SZ(drv, ch);
    sz_v &= ~(0xFFF); // Clear tot_sz
    sz_v |= (sz & 0xFFF);
    WB_DMA_WRITE_CH_SZ(drv, ch, sz_v);

    // Start the transfer
    WB_DMA_WRITE_CH_CSR(drv, ch, csr);

    drv->status[ch] = 1;
}
```

Figure 7 - Embedded C Test Realization Reuse

Figure 7 shows a similar C function for programming the DMA engine to perform a single transfer. We can also leverage this code to provide test realization for PSS test intent for the DMA engine.

Note that the two sets of existing code are similar but not the same. We'll need to determine how best to interface to these from our PSS.

BUILDING REUSABLE PSS LIBRARIES

As you consider creating PSS content internal to your organization, it's worth thinking about common data structures. PSS as a standard is fairly new to the industry at the time that this paper was written and, consequently, doesn't have a standardized library of common data structures and other reusable types. It is still highly advisable to try to establish a reusable library of common types within your organization. By their nature, PSS descriptions frequently use very similar data structures -- for example, a memory buffer that has an address and size.

```
struct data_mem_t {
    rand bit[31:0]      addr;
    rand bit[31:0]      sz;
}

buffer data_mem_b : data_mem_t {
}
```

Figure 8 - Reusable Data-buffer Type

Don't take the chance that three people responsible for three different IPs will all define the same memory buffer. This would make it quite difficult to combine the three PSS models in a subsystem or SoC-level environment. Instead,

define common types, like that shown in Figure 8, and ensure that people creating PSS content in your organization reuse these common types and are able to contribute to the common type library.

It's helpful to establish some per-IP methodology with respect to creating PSS content. I recommend that all PSS actions for a given IP derive from a common IP-specific abstract action type, as shown in Figure 9.

```
abstract action dma_dev_a : pvm_dev_a {
    // All transfers involve a channel
    rand bit[7:0] in [0..7] channel;
    // Size of each transfer
    rand bit[4] in [1,2,4] trn_sz;

}

action mem2mem_a : dma_dev_a {
    input data_ref mem_b    dat_i;
    output data_ref mem_b   dat_o;

    constraint {
}

action dev2mem_a : dma_dev_a {
    output data_ref mem_b   dat_o;
    input data_ref s        info_i;

    rand bit[31:0]          src_addr;

    constraint {
}
```

Figure 9 - IP-Specific Common Base Action

A key PSS feature is type extension that allows content to be inserted in a PSS type without modifying the type itself. Having a common base type for all actions related to a given IP provides a common type to which extensions intended to apply to all actions for a given IP can be applied.

Reusable Data Generation and Checking

Results checking is one aspect of testing that varies significantly across the IP-block to SoC verification continuum. At the block level, it's common to use detailed scoreboard-based checking that looks at details of how an operation was carried out, as well as its overall result. At the SoC level, that level of visibility into the design isn't feasible, and result checking tends to be based on the overall result of the operation.

Defining a checking strategy that will be usable from IP to SoC will be important if vertical-reuse portability is a high priority for your organization. In this case, it is highly recommended to focus on building the types of checks that retain validity at the SoC level into the PSS description. Typically these checks will be based on in-memory data, and will focus on the overall success (or failure) of an operation.

It is always possible to augment functional checks with implementation checks. For example, at the block level, the DMA engine operation can be checked from a portable stimulus perspective by purely-functional checks (ie is the data at the destination the same as the data at the source). The block-level scoreboard can still be active in checking the details of how the DMA transfer was carried out. This strategy can be extended to the subsystem and SoC level as well. For example, bringing performance-checking scoreboards in at the SoC level.

Making Test Realization Reusable

Having multiple implementations of test realization is effectively mandatory. It's important for verifiers working in UVM to be able to take advantage of the services, such as a register model, that UVM provides. At the same time, it's

important for verifiers working with embedded software to be able to take advantage of the register-access mechanisms (packed data structures, bit fields, etc) that they are familiar with.

Define Common APIs

That said, it is beneficial to maximize the commonality between the different implementations of test realization. Designing a common API that can be used by all implementations is a first step in this direction.

```
task mem2mem(
    int unsigned            channel,
    int unsigned            src,
    int unsigned            dst,
    int unsigned            sz,
    int unsigned            trn sz);

    `uvm_info(get name(), $sformatf("--> mem2mem channel=%0d src='h%08h dst='h%08h sz=%0d",
            channel, src, dst, sz), UVM LOW);
    init single transfer(channel, 0, src, 1, dst, 1, sz, trn sz);
    wait complete irq(channel);
    `uvm_info(get name(), $sformatf("<-- mem2mem channel=%0d src='h%08h dst='h%08h sz=%0d",
            channel, src, dst, sz), UVM LOW);
endtask
```

Figure 10 - Common DMA API (SV)

Figure 10 shows an API for use by a DMA action that is built on top of the SV tasks reused from the block-level verification environment.

```
void wb_dma_dev_mem2mem(
        uint32 t            devid,
        uint32 t            channel,
        uint32 t            src,
        uint32 t            dst,
        uint32 t            sz,
        uint32 t            trn sz) {
    wb dma dev t    *drv = (wb dma dev t *)uex get device(devid);
    uint32 t csr, sz v;
    uex info low(0, "--> wb dma dev mem2mem %s channel=%d src=0x%08x dst=0x%08x sz=%d",
            drv->base.name, channel, src, dst, sz);
    // Disable the channel
    csr = uex ioread32(&drv->regs->channels[channel].csr);
    csr &= ~(1);
    uex iowrite32(csr, &drv->regs->channels[channel].csr);

    // Program channel registers
    csr = uex ioread32(&drv->regs->channels[channel].csr);
```

Figure 11 - Common DMA API (C)

Figure 11 shows an implementation of the same functionality in C for use in an embedded-software environment. Keeping a functionally-equivalent API simplifies the task of mapping from PSS to the various implementations of test realization.

If vertical reuse is of high importance, it's important to consider whether it's worth investing in an environment-compatibility layer, like the UEX hardware-access layer shown in Figure 12. The UEX hardware-access layer [1] provides a C API for accessing platform memory and threading capabilities in several ways. Using a compatibility layer like this enables test realization code for use in an embedded-software environment to be developed and debugged much earlier in the verification process, and reused across more of the verification process.

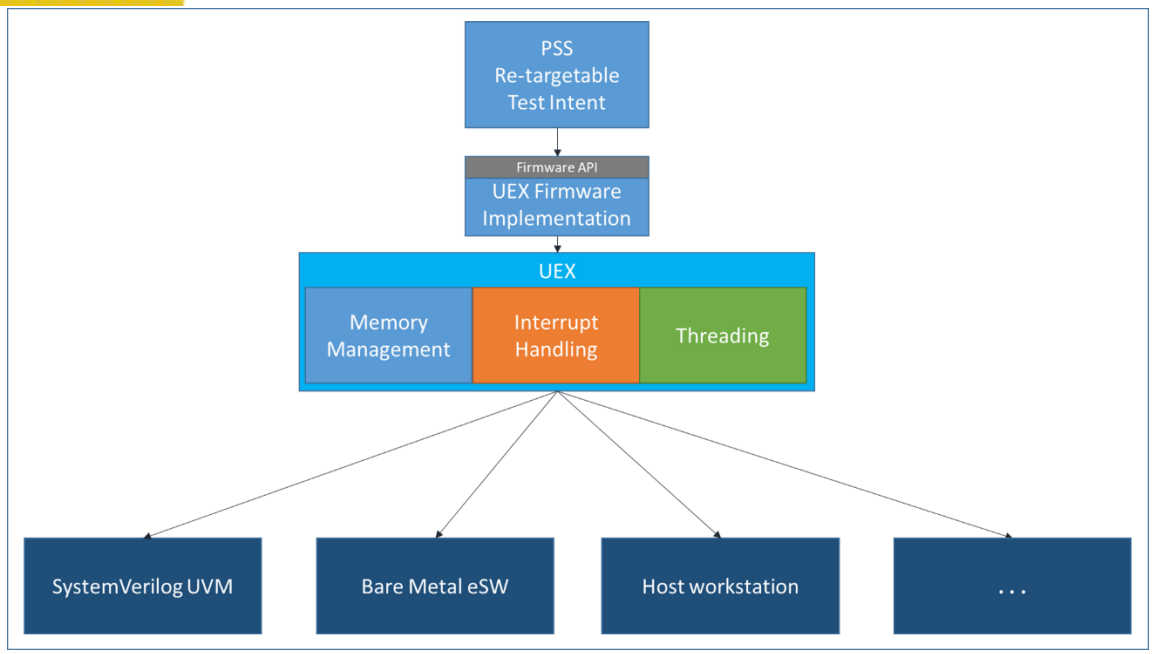

Figure 12 - UEX Hardware Access Layer

Whether it's a compatibility layer that spans several platforms, or a series of environment-specific APIS, it is important to consider how the test realization for different IPs will cooperate. The test realization code for all IPs will likely need to access memory. The test realization for many IPs will require notification when an interrupt occurs. In production code, an operating system provides the glue that connects the driver code for various IPs. In a verification environment, whether UVM or embedded software, something much more lightweight is required.

```
static void wb_dma_dev_irq(struct uex_dev_s *devh) {
    wb_dma_dev_t *dev = (wb_dma_dev_t *)devh;
    uint32_t i;
    uint32_t src_a;

    src_a = uex_ioread32(&dev->regs->int_src_a);

    // Need to spin through the channels to determine
    // which channel to activate
    for (i=0; i<8; i++) {
        if (src_a & (1 << i)) {
            // Read the CSR to clear the interrupt
            uint32_t csr = uex_ioread32(&dev->regs->channels[i].csr);
            uex_event_signal(&dev->xfer_ev[i]);
            dev->status[i] = 0;
        }
    }
}
```

Figure 13 - DMA IRQ Routine using UEX API

Figure 13 shows an interrupt-service routine for the DMA IP that uses the UEX API to read the DMA registers and notify waiting routines that a DMA transfer is complete. The UEX library enables this same code to run in a UVM, embedded bare-metal software environment, as well as an OS-based environment. This reuse of test realization code enables early debug of code for accessing IPs, and minimizes rework.

Specify a Common PSS Interface

Actions and test realization code for a type of IP are expected to interact with multiple instances of that IP. Specifying a common way to select, from the PSS layer, which IP instance is being accessed is important to ensuring uniformity across different test realization implementations.

```
component pvm dev c {

    bit[7:0]        devid;

    action pvm dev a {

    }

}
```

Figure 14 - Test Realization Base Component and Action

Figure 14 shows an example base component action that specifies a built-in field named *devid* that specifies which instance of an IP is being accessed by a given action. Defining a reusable base component and action type ensures that all PSS descriptions developed within your organization specify which IP instance is in use in the same way.

```
extend action wb_dma_c::mem2mem_a {
  exec body C = """
    wb_dma_dev_mem2mem({{comp.devid}}, {{channel}}, {{dat_i.addr}},
        {{dat_o.addr}}, {{dat_i.sz}}, {{trn_sz}});
  """;
}
```

Figure 15 - Referencing the Component devid Field

Figure 15 shows how the *devid* field is referenced from a PSS exec block for one of the DMA actions.

Minimize Data Exchange

One best practice when developing PSS test realization code is to minimize the volume of data exchanged between the PSS model and the test realization code. This best practice is shared by other languages that have a foreign language, such as SystemVerilog and Java [2]. Generally speaking, the PSS description is an executive that specifies the high-level view of operations for which the test realization will carry out the details.

Take, for example, the actions involved in a DMA transfer. Before transferring data from a memory location, that memory location should be initialized. Instead of writing a PSS description to fill in memory byte-by-byte, the PSS description shown in Figure 16 specifies a memory region to initialize, and delegates the details of how memory is initialized to the test realization function.

```
action gendata_a {
    input data_mem_b        dat_i;
    output data_ref_mem_b   dat_o;

    constraint dat_o.addr == dat_i.addr;
    constraint dat_o.sz == dat_i.sz;
}
```

Figure 16 - Action to Initialize Memory

```
void pvm_gendata(uint32_t ref, uintptr_t addr, uint32_t sz) {
    pvm_rand_t r;
    void *addr_p = (void *)addr;
    int i;

    pvm_rand_init(&r, ref);

    for (i=0; i<sz; i++) {
        uint8_t v = pvm_rand_next(&r);
        uex_iowrite8(v, addr_p+i);
    }
}
```

Figure 17 - C Test Realization to Initialize Memory

Figure 17 shows an implementation of the *gendata* functionality implemented in C. This delegation of responsibilities enables the PSS description to stay at a high level where declarative programming is most efficient, while delegating the detail work to an imperative language, which is most efficient at carrying out these tasks.

SUMMARY

Portable stimulus enables several types of test intent reuse: reuse across verification levels (vertical reuse), reuse across projects (horizontal reuse), and reuse of techniques across otherwise-unrelated environments. Selecting and prioritizing which of these benefits is attractive to your organization enables proper focus on what is important to enable those applications of portable stimulus. Performing an inventory of existing assets helps to ensure maximum benefit from previous investment. Developing an in-house methodology and PSS library helps to ensure that your organization uses common methodology. Finally, defining common APIs for test realization code and ensuring that test realization for different IPs can interoperate ensures that portable stimulus reuse is facilitated, and not limited, by test realization code.

All of these steps help to ensure that your organization can maximize the productivity benefits that Portable Stimulus offers.

REFERENCES

[1] M. Balance, “Managing and Automating Hw/Sw Tests from IP to SoC”, DVCon 2018, https://s3.amazonaws.com/verificationacademy-news/DVCon2018/posters/dvcon-2018_managing-and-automating-hw-sw-tests-from-ip-to-soc_poster.pdf

[2] M. Dawson, G. Johnson, A. Low, “Best Practices for using the Java Native Interface”, https://www.ibm.com/developerworks/library/j-jni/index.html