Designing PSS Environment Integration for Maximum Reuse

Matthew Ballance
Mentor, A Siemens Business
Portable Stimulus Vision

- Ambitious Scope
  - Portable across verification levels
  - Portable across verification engines
- Enables model-driven test creation
- Enables reuse of test intent
Test Intent vs Test Realization

- PSS is primarily a declarative language
  - Think “constraint-based”

- PSS separates test intent and realization
  - Test intent is declarative
  - Test realization is procedural

- Must address both test intent and realization
Let’s look at a simple example
  - DMA Engine with 8 channels
  - Basic memory-to-memory transfer

Test Intent focuses on
  - Pre- and post-conditions
  - Constraints on operation

Test realization
  - Programs registers
  - Waits for interrupts

```
action wb_dma_xfer_a : wb_dma_a {
  // The channel this transfer runs on
  rand bit[3] channel;

  // Total transfers to perform
  rand bit[16] tot_sz;

  // Bytes to transfer at a time (1, 2, 4)
}

action mem2mem_a : wb_dma_xfer_a {
  input data_mem_b dat_i;
  output data_mem_b dat_o;

  constraint {
    dat_i.sz == (tot_sz * trn_sz);
    dat_i.sz == dat_o.sz;
  }
}
```
Example – Test Realization

• PSS is very flexible with test realization
• Can specify precise code to generate
  – Can literally generate **anything**
• But, this has a cost for reuse
  – Specific language
  – Specific environment
  – Limited to pre-generated flat tests

```extend action wb_dma_c::mem2mem_a {
  exec body SV = ""
  begin
    wb_dma_descriptor desc =
      wb_dma_descriptor::type_id::create();
    desc.channel = {{channel}};
    desc.mode = 0;
    desc.inc_src = 1;
    desc.inc_dst = 1;
    desc.src_sel = 0;
    desc.dst_sel = 1;
    desc.tot_sz = {{tot_sz}};
    desc.trn_sz = {{trn_sz}};
    desc.chk_sz = 16;
    desc.src_addr = {{dat_i.addr}};
    desc.dst_addr = {{dat_o.addr}};

    start_item(desc);
    finish_item(desc);
  end
  """;
}
Example – Test Realization

• Calling an API improves reuse
  – Can pre-generated tests or generate on-the-fly
  – Better separation between Model and Realization

• Implementation code is nearly identical!

task mem2mem(
  bit[31:0] channel,
  bit[31:0] src,
  bit[31:0] dst,
  bit[31:0] tot_sz,
  bit[31:0] trn_sz);

wb_dma_descriptor desc =
  wb_dma_descriptor::type_id::create();

desc.channel = channel;
desc.mode = 0;
// . . .
desc.tot_sz = tot_sz;
desc.trn_sz = trn_sz;
desc.chk_sz = 16;
desc.src_addr = src;
desc.dst_addr = dst;

start_item(desc);
finish_item(desc);
endtask

function void mem2mem(
  bit[31:0] channel, bit[31:0] src,
  bit[31:0] dst, bit[31:0] tot_sz,
  bit[31:0] trn_sz);

import target function mem2mem;

extend action wb_dma_c::mem2mem_a {
  exec body {
    mem2mem(channel, dat_i.addr, 
              dat_o.addr, tot_sz, trn_sz);
  }
}

endfunction
Anatomy of Test Realization

- Reusable test realization goes beyond calling an API
- Three key aspects of test realization
  - API
    - Consistent interface to driver functionality
  - Configuration and context data
    - Configures driver code
    - Maintains state
  - Events
    - Notifies driver code to react to system

```c
void wb_dma_dev_mem2mem(
  wb_dma_dev_t *dev,
  uint32_t     channel,
  uint32_t     src,
  uint32_t     dst,
  uint32_t     sz,
  uint32_t     trn_sz);
```

typedef struct wb_dma_dev_s {
  wb_dma_regs_t     *regs;

  // Status flags for state of channels
  uint32_t     status[8];

  // Notification objects for interacting with IRQ
  pvm_event_t   xfer_ev[8];
} wb_dma_dev_t;

Test Realization Reuse Requirements

• PSS models elements compose easily
• Must be able to do the same with test realization
Test Realization Reuse Requirements

- PSS models easily support multiple IP instances
  - Component tree models IP instances
  - Maintains Action / Component instance association
- Must be able to do the same with test realization
Test Realization Reuse Requirements

• The PSS model is independent of execution platform
  – UVM, embedded software, etc

• Test realization API must support this as well

• Implies consistent API across platforms
Test Realization Best Practices

• Interact via scalar values
  – PSS modeling works well with scalar variables since they can be constrained
  – Scalar values are also treated similarly across platforms (UVM / eSW)

• Minimize data exchange
  – Best practice with most cross-language interfaces
  – Generally want data to move from Test Intent to Test Realization
Test Realization Best Practices

• Abstract Up
  – Helps to achieve goal of minimizing data exchange
  – Test Intent invokes high-level behavior, delegating details to Realization

• Use the procedural interface
  – Ensures PSS model is independent of realization language

• Use a test realization framework
  – Helps to ensure consistency
Test Realization Reuse Framework

• Base component type provides ‘devid’
  – Used in exec blocks to identify test-realization instance

```plaintext
component pvm_dev_c {
    int devid;
}

component wb_dma_c : pvm_dev_c {
    import pvm_types_pkg::*;
}

function void wb_dma_dev_mem2mem_d(
    bit[31:0] devid,
    bit[31:0] channel,
    bit[31:0] src,
    bit[31:0] dst,
    bit[31:0] tot_sz,
    bit[31:0] trn_sz);
import target function mem2mem;
extend action wb_dma_c::mem2mem_a {
    exec body {
        wb_dma_dev_mem2mem_d(comp.devid,
            channel,dat_i.addr,
            dat_o.addr, tot_sz, trn_sz
            );
    }
}
```
Test Realization Reuse Framework

- A base component is provided for UVM environments
- Each test-realization component must extend
- Test-realization class contains context data

```cpp
class wb_dma_dev extends pvm_dev;
    `uvm_object_utils(wb_dma_dev);
    wb_dma_reg_block m_regs;
    bit m_active[];
    semaphore m_sem[];
endclass
```
Test Realization Reuse Framework

- A macro is used to translate global task to Class-method calls
- Uses the devid passed from PSS to identify the appropriate object

```c
\`pvm_dev_task_decl_5(wb_dma_dev, mem2mem, uint32_t, uint32_t, uint32_t, uint32_t, uint32_t)
```

```c
task automatic wb_dma_dev::mem2mem_d(  
    uint32_t devid,  
    uint32_t p1,  
    uint32_t p2,  
    uint32_t p3,  
    uint32_t p4,  
    uint32_t p5);  
wb_dma_dev dev_inst;  
$cast(dev_inst, pvm_get_device(devid));  
  dev_inst.mem2mem(p1, p2, p3, p4, p5);
endtask
```

```c
task wb_dma_dev::mem2mem(  
    int unsigned channel,  
    int unsigned src,  
    int unsigned dst,  
    int unsigned sz,  
    int unsigned trn_sz);

    init_single_transfer(channel, 0, src, 1, dst, 1, sz, trn_sz);
    wait_complete_irq(channel);
endtask
```
Test Realization Reuse Framework

• Embedded software stores context data in a struct
  – Contains a data-structure instance (pvm_dev_t) with core data

• Test realization functions operate on this data structure

```c
typedef struct wb_dma_dev_s {
    pvm_dev_t dev;
    wb_dma_regs_t *regs;
    uint32_t status[8];
    pvm_event_t xfer_ev[8];
} wb_dma_dev_t;
```

```c
void wb_dma_dev_mem2mem(
    wb_dma_dev_t *drv,
    uint32_t channel,
    uint32_t src,
    uint32_t dst,
    uint32_t sz,
    uint32_t trn_sz);
```
Test Realization Reuse Framework

- A macro used to setup redirect functions
  - Accept device-id from PSS, call function with context data

```c
static inline void wb_dma_dev_mem2mem_d(
    uint32_t devid, uint32_t p1, uint32_t p2,
    uint32_t p3, uint32_t p4, uint32_t p5)
{
    wb_dma_dev_mem2mem((wb_dma_dev_t *)pvm_get_dev(devid),
        p1, p2, p3, p4, p5);
}
```

```c
void wb_dma_dev_mem2mem_d(
    uint32_t devid, uint32_t p1, uint32_t p2,
    uint32_t p3, uint32_t p4, uint32_t p5) {
    wb_dma_dev_mem2mem((wb_dma_dev_t *)pvm_get_dev(devid),
        p1, p2, p3, p4, p5);
}
```

```c
void wb_dma_dev_mem2mem(
    wb_dma_dev_t *dev,
    uint32_t channel,
    uint32_t src,
    uint32_t dst,
    uint32_t sz,
    uint32_t trn_sz);
```

```c
pvm_devid_api_decl_5(wb_dma_dev, mem2mem,
    uint32_t, uint32_t, uint32_t, uint32_t, uint32_t);
```

```c
static inline void wb_dma_dev_mem2mem_d(
    uint32_t devid, uint32_t p1, uint32_t p2,
    uint32_t p3, uint32_t p4, uint32_t p5) {
    wb_dma_dev_mem2mem((wb_dma_dev_t *)pvm_get_dev(devid),
        p1, p2, p3, p4, p5);
}
```
PSS 1.1. and Test Realization Reuse

• PSS 1.1 new features enable Portable Test Realization
  – Capture test realization predominantly in PSS, not C/C++ or SystemVerilog
  – Requires rewriting test realization
  – Once rewritten, can target any platform supported by the PSS processing tool

• Register Model
  – Provides a PSS abstraction for accessing registers

• Storage Allocation
  – APIs and data structures for managing and accessing memory

• Procedural Interface
  – PSS procedural “programming language”
  – Can write simple routines for managing IPs directly in PSS
Conclusion

• PSS defines modeling constructs that enable reuse and portability

• Must also ensure test realization is reusable and portable

• Following a few key best practices

• Using a test-realization reuse framework brings consistency

• Look for new opportunities in test realization reuse in PSS 1.1!