Making Legacy Portable with the Portable Stimulus Specification

Matthew Ballance, Mentor Graphics Corp.
Verification Stimulus Requirements

- Higher-level tests
  - Scenarios vs transactions
  - High test-creation productivity

- Automated tests across environments
  - UVM tests required at block, subsystem
  - Software-driven tests required at SoC
  - Target simulation, emulation, post-silicon

- Reuse of test intent across environments
  - Develop test intent at block level
  - Reuse block-level intent at subsystem
  - Reuse subsystem-level intent at SoC
What is Portable Stimulus?

• Accellera Portable Stimulus Working Group
  – Defining a Portable Stimulus Specification (PSS) language standard
  – Built from Portable Stimulus Specifications in use today

• A single representation of test intent that is reusable
  – By a variety of users
  – Across levels of integration
  – In a variety of execution platforms
  – Across different configurations

• Portable stimulus is not
  – A single forced level of abstraction
  – A replacement for all current testing activities
  – A monolithic representation
Anatomy of a PSS

Description

• Purely-declarative specification
  – Data structures
  – Random fields
  – Constraints

• Declarative-procedural
  – Operation sequence
  – Repetition
  – Dataflow relationships

• Interfaces to external code
  – Solver-helper functions
  – Test-realization utility code
Portable Stimulus and Legacy

- Reuse of legacy accelerates adoption
  - Build on existing, validated descriptions
  - Less content creation to get started

- Reuse Opportunities
  - SV and PSS declarative descriptions overlap
    • Can translate subset
  - PSS supports calling external behavioral code
    • Link to existing modeling code
    • Link to test-realization code
Legacy Descriptions

• Data structures and constraints (SV)
  – Transactions
  – Configuration

• Procedural modeling code
  – Compute expected results
  – Model behavior like memory allocation

• Procedural test-realization code
  – Program IPs
  – Interface with RTOS/OS
Reuse Opportunities

- Create input specification from legacy
- Reuse existing modeling code
- Reuse test-realization code
Declarative SV

• Block-level constraints
  – Configuration constraints
  – Operation-mode constraints

• Often reusable in higher-level scenarios
  – Configuring a multi-IP subsystem

• Reusable in PSS by translation
  – Extract SV variable / constraints
  – Translate to PSS syntax
Declarative SystemVerilog

- SV and PSS constraint syntax is nearly identical
- Modular, well-encapsulated constraints ease reuse

**Reuse Guidelines**
- Avoid inline constraints
- Avoid procedural enable/disable
- Avoid class-external references

```systemverilog
typedef enum {
    MODE_1,
    MODE_2,
    MODE_3,
    MODE_4
} mode_t;

class device_cfg;
    rand mode_t mode;
    rand bit[15:0] coeff1;
    rand bit[15:0] coeff2;

    constraint coeff_c {
        coeff1 <= coeff2;
    }

    constraint mode_c {
        if (mode == MODE_1 ||
            mode == MODE_2) {
            coeff1 == 0;
        }
    }
endclass
```

```pss
enum mode_t {
    MODE_1,
    MODE_2,
    MODE_3,
    MODE_4
}

struct device_cfg {
    rand mode_t mode;
    rand bit[15:0] coeff1;
    rand bit[15:0] coeff2;

    constraint coeff_c {
        coeff1 <= coeff2;
    }

    constraint mode_c {
        if (mode == MODE_1 ||
            mode == MODE_2) {
            coeff1 == 0;
        }
    }
}
```
Modeling Code

• Modeling code shapes generated stimulus
  – Computes values from randomized parameters
  – Computes expected results

• Interacts with the solve process

• Modeling code is ‘opaque’ to randomization process
  – Use care when controlling modeling-code results is desirable

• Example: memory allocation
  – If we just need a valid address, then reuse allocation function
  – If we need to constrain the address, then must code as constraints
Incorporating Modeling Code

- PSS provides a procedural interface to external code
  - Similar to SystemVerilog DPI
- PSS provides hooks to the solve process
  - pre_solve – before constraint solving
  - post_solve – after constraint solving
- Solver hook reference modeling code
- Example: address allocation
  - PSS tool solver selects buffer size
  - Allocation algorithm selects address

```verilog
import bit[31:0] alloc(bit[31:0] sz);

struct buf_addr {
    rand bit[15:0] sz;
    bit[31:0] addr;
}

exec post_solve {
    addr = alloc(sz);
}

component top_comp {
    action do_write {
        rand buf_addr buffer;
        // ...
    }
    action entry {
        do_write wr1, wr2;
        constraint c {
            wr1.buffer.sz != wr2.buffer.sz;
        }
        graph {
            w1;
            wr2;
        }
    }
}
```
Test Realization Code

• Implements low-level test intent details
  – Sets IP registers to setup device mode
  – Performs operation-status check

• Examples
  – SystemVerilog utility tasks
  – C utility functions
  – C driver stubs

• “body” exec specifies behavior
  – References external functions

```python
import void set_mode(mode_t m);
import void set_coeff(bit[3:0] coeff_num, bit[15:0] coeff);
import void init();

component device_comp {
    action do_device_cfg {
        rand device_cfg cfg;

        exec body {
            set_mode(cfg.mode);
            set_coeff(1, cfg.coef1);
            set_coeff(2, cfg.coef2);
            init();
        }
    }
}
```
Test Realization Code

Guidelines

• Consider cross-language reuse
  – Plan for symmetrical APIs in SV, C
  – Implement some APIs in C and reference in SV

• Use primitive data types where possible
  – Like SystemVerilog DPI, PSS works best with simple types

• Use care with memory management
  – Can store opaque pointers in PSS
  – Foreign language manages memory

```c
#include <stdint.h>

typedef enum {
    MODE_1,
    MODE_2,
    MODE_3,
    MODE_4
} mode_t;

void set_mode(mode_t m);
void set_coeff(uint8_t coeff_num, uint16_t coeff);
void init(void);
```
Portable Stimulus and Legacy

- Portable Stimulus
  - Raises abstraction level
  - Brings automated testing to new environments
  - Enables reuse across environments

- Emerging standard encourages reuse
  - Reuse declarative SystemVerilog code
  - Reuse modeling code
  - Reuse test-realization code

- Reuse of legacy accelerates adoption
  - Build on existing, validated descriptions
  - Less content creation to get started