Aspect Oriented Support in System Verilog

Tony Tsai

Introduction

- Complex designs face the following verification challenges (amongst others):
  - Large random space
  - Complex model required for checking/scoreboarding
- System Verilog provides powerful language for building complex verification environments
  - Support for constraint-random environments
  - OOP for modularity and scaling
- So what improvements are needed?
  - Test case development

Test Case

- Random environment still require test cases
  - Boundary conditions, important sequences, various operating modes, etc.
- Two different ways of writing a test:
  - Testcase constructs environment: Tests instantiate necessary components of the environment with appropriate settings
    - Duplication of code in tests
    - Difficult to manage in a large environment
  - Separate testbench and testcase: Test “poke” into the environment to set certain configurations
    - Simplifies test development
    - Scales better in large environment

Separating Out Test Case

- Majority of environment intelligence built into testbench
- Testbench environment provides necessary knobs for exploring the random space
- Test configures the knobs to the appropriate random space
- Test writer focus on testplan execution by tuning knobs

Current Methods

OOP

- Manually replace original definition of a class with derived class
- Easy to replace when building the environment
  - More appropriate for methodology where testbench and tests are not separated
- Instance-based replacement
  - Path to instance needs to be known (not scalable or reusable)
- Upfront planning required
  - Restrictions on code structure
  - Methods may need to be virtual

Current Methods

Factories

- Similar idea to OOP approach but methodology supported to provide easier replacements
- Heavy requirement on testbench environment
  - Need to follow methodology support
  - Timeline crucial in when an object is created – may complicate usage of object in different phases of timeline
  - Debugging may be difficult
- Upfront planning required
  - May be difficult to update an object to a factory object due to a new test requirement

Aspect Oriented Programming (AOP)

- Allows an aspect to alter the behavior of the base code
  - Adding or overriding constraints defined in a class
  - Adding or overriding class methods
- Not proposing to change SV to a pure AOP language
  - OOP is powerful enough to build complex testbench environments
  - Aspect in the testbench environment can be difficult to debug and maintain
- AOP usage most beneficial in testcases

AOP in Test Case

- Global replacement of class behavior
  - Instance based replacement can easily be implemented with an identifier
- No additional testbench environment or methodology support necessary
- Minimal upfront planning for AOP when developing testbench environment
  - No timeline requirement or hierarchy knowledge needed
- Scales well in large environments
- Factory support in the tool rather than in methodology

OOP/AOP

Adding Constraints

OOP

```
class new_pkt extends pkt;
   constraint c {
      pkt_length inside {[100:200]};
   }
endclass

class test ;
   task run(my_env env) ;
      new_pkt pkt = new;
      env.generator.xactor1.pkt = pkt;
      env.run();
   endtask: run
endclass: test
```

AOP

```
extends test_aop(pkt);
   constraint test0_aop {
      pkt_length inside {[100:200]};
   }
endextends
```

- Factories require even more code than OOP
  - Requires methodology as well as testbench environment support
  - Provides same flexibility as AOP

OOP/AOP

Modifying Methods

OOP

```
class new_blk_cfg extends blk_cfg;
   virtual function void configure_blk();
      expect_entry l_expect;
      super();
      l_expect = new(“blah”);
      expect_status(l_expect);
   endfunction
endclass

class test ;
   task run(my_env env) ;
   …
endtask: run
endclass: test
```

AOP

```
extends test_aop_1(blk_cfg);
   after function void configure_blk();
      expect_entry l_expect;
      l_expect = new(“blah”);
      expect_status(l_expect);
   endfunction
endextends
```

- AOP allows combining different aspects (n aspects)
  - Easy to run cross product of configs
- OOP/Factories require separate class for each configuration (up to 2^n classes)

Benefits in AOP in Test Cases

- Allows for scaling and reuse of environment
  - Aspect can be added to any environment class without modifications to the testbench
  - No restrictions placed on environment
- Fully backwards compatible with OOP (assuming use in test code)
  - Aspect file can be added in anytime
- Effective and efficient method of test writing
  - Test writer does not need to know details of testbench or even SV
  - Allows designers to help in testplan execution

Proposal

- Support AOP in System Verilog to allow for easier test case creation
  - Necessary as complexity increases in environments
- Use Open Vera AOP syntax as starting point
  - SV and OV syntax similar in other parts of the language
  - New syntax may need to be defined (e.g. parameterized class support)
  - Take step-wise approach to supporting AOP
- Not a replacement for OOP
- OVM/VMM/xVM can easily work AOP into methodology as an alternative for factories
  - Simplify implementation and usage for end-user

Questions


CISCO