Design & Verification Conference & Exhibition

February 28 – March 1, 2012

# Soft Constraints in SV : Semantics and Challenges

by

Mark Strickland

Verification Lead

Cisco Systems

# Agenda

- Current challenges
- Soft Constraint Defined
- Application of Soft Constraints
- Debugging Soft Constraints
- Benefit of Soft Constraints

# Constraint Modification Challenges

- It is desirable to centralize the intent of commonly needed constraints in the testbench, BUT some tests will need to apply constraints that conflict with common constraints
- Constraint modification techniques
    - Overriding constraint blocks through inheritance
    - Disable constraint block with constraint_mode()
    - Disable randomization for field with rand_mode()
    - Soft constraints
        - Reference: Approved for SystemVerilog IEEE 1800-2012 (Mantis 2987)

# Constraint Block Modification

- Overriding or constraint_mode(0) are applied to a constraint block (not individual constraint)
  - Block must contain only the constraints to be modified
  - Requires documentation or naming convention to identify blocks that are OK to ignore
- If a constraint should be applied except for a particular "randomize with", must use constraint_mode(0) before and constraint_mode(1) after that randomize call.
- If a constraint applied in "randomize with" could be ignored when a conflicting constraint is applied, neither solution can be applied.

# Randomization Control

- Using rand_mode(0) does target a specific field, but you can only set a field value, losing randomization.

# Modification Without Soft

- Discovering conflicting constraints involves debug time
- Fixing conflicting constraints cleanly requires a strict methodology having been applied to the original constraints
- When constraints are applied in “randomize with”, there may be no solution other than modifying the original code

# Semantics of Soft Constraints

- Hard constraints:
  - The solver must satisfy these constraints, or result a solver failure

    **constraint** x_constraint_1 { x < 10 };

- Soft constraints:
  - The solver is to satisfy these unless contradicted by a hard constraint or a soft constraint with higher priority (priority covered a little later)

    **constraint** x_constraint_2 { **soft** x < 5 };

# Discarding Soft Constraints

- Mantis 2987 also defines a mechanism to allow a test writer to disable all lower priority soft constraints on a variable

```
class M;
  rand int x;
  constraint a { soft x > 2; soft x < 10;
endclass

M obj = new();
obj.randomize() with {

   x inside { [0:20] };
}
```

*Solution: x = 3 .. 9*

```
class M;
  rand int x;
  constraint a { soft x > 2; soft x < 10;
endclass

M obj = new();
obj.randomize() with {
   disable soft x;
   x inside { [0:20] };
}
```

*Solution: x = 0 .. 20*

# Typical Constraint Scenario #1

- Say x is an enum with possible values MODE1, MODE2, MODE3. The most common mode is MODE1.

```
class M;
  rand int x;
  constraint common { soft x == MODE1;}
endclass

class N extends M;
  constraint special_test { x == MODE2; }
endclass
```

*Solution (for N): x = MODE2*

- Tests that need mode 1 do not need to add anything. Tests that need mode 2 can use x==MODE2.

# Typical Constraint Scenario #2

- Say x is an integer that can have a value between 0 and 50. Typically, x between 0 and 10 gives enough variation and faster simulation.

```
class M;
  rand bit[7:0] x,y;
  constraint legal {
    x inside {[0..50]};
    x > y;
  };
  constraint typ { soft x < 11; };
endclass
```

- Some tests must exercise the whole range {[0..50]}.

```
class M1 extends M;
  constraint test1 { disable soft x ; };
endclass
```

*Solution: x in 0..50*

```
class M2 extends M;
  constraint test2 { y == 20; };
endclass
```

*Solution: x in 21..255*

- One test adds constraint y ==20. The environment will adapt automatically to make the range for x be > 20.

# Typical Constraint Scenario #3

- Say x and y are 8-bit integers and the most DUT coverage is achieved when x > y.

```
class M;
  rand bit[7:0] x, y;
  constraint typ { soft x > y;};
endclass
```

A test requires y > 20.
The x > y still applies.

```
class M1 extends M;
  constraint test1 { y > 20; };
endclass
```

*Solution: x > y >20*

A test requires x < 10.
The x > y still applies.

```
class M2 extends M;
  constraint test2 { x < 10; };
endclass
```

*Solution: 10 > x > y*

A test requires both y > 20 and x < 10. Testbench adapts by ignoring the x > y constraint.

```
class M3 extends M;
  constraint test3 { x < 10;  y > 20;};
endclass
```

*Solution: x < y*

# Soft Constraint Priorities

- What happens when two soft constraints conflict?
- General answer: Later in the class or higher in the hierarchy wins (higher priority).

# Soft Constraint Priorities

- Higher priority given to constraints...
  1) That appear later in the same construct (constraint block, class, or struct)
  2) In out-of-body constraint blocks whose prototypes appear later in the class
  3) In container objects (class or struct) relative to constraints in its contained objects (rand class handles)
  4) In objects whose handles appear later in the container object
  5) In derived classes relative to constraints in their super classes
  6) Within inline constraint blocks relative to constraints in the class being randomized
  7) In later iterations within a foreach constraint

# Soft Constraint Priorities - Example

```
class M;
  rand int x;
  constraint a { soft x > 2; soft x < 10; }
endclass

class N extends M;
  constraint b;
  constraint c { soft x == 5; }
endclass

constraint N::b { soft x == 9; }

class Q ;
  rand N n;
  constraint d { soft n.x inside {[5:8]}; }
endclass

Q obj = new();
obj.randomize() with { soft n.x >= 7; };
```

Highest Priority

| | |
|---|---|
| x >= 7 | inline constraint |
| x inside { [5:8] } | Q::d in the container class |
| x == 5 | N::c (declared after N::b) |
| x == 9 | N::b out of body |
| x < 10 | M::a |
| x > 2 | M::a (appear before x<10) |

Lowest Priority

*Solution: x == 7 .. 8*

# Integration of other SV constraint controls

- Inheritance – different constraint block name

```
class M;
  rand int x;
  constraint a { soft x > 2; soft x < 10;
endclass

class N extends M;
  constraint b { soft x inside { [8:12] }; }
endclass

N obj = new();
obj.randomize();
```

*Solution: x = 8 .. 9*

- Inheritance – same constraint block name

```
class M;
  rand int x;
  constraint a { soft x > 2; soft x < 10;
endclass

class N extends M;
  constraint a { soft x inside { [8:12] }; }
endclass

N obj = new();
obj.randomize();
```

*Solution: x = 8 .. 12*

# Integration of other SV constraint controls

- constraint_mode
- rand_mode

```
class M;
  rand int x;
  constraint a { soft x > 2; soft x < 10;
  constraint b { soft x == 3; }
endclass

M obj = new();

obj.randomize();
```

*Solution: x = 3*

```
obj.b.constraint_mode(0);
obj.randomize();
```

*Solution: x = 3 .. 9*

```
class M;
  rand int x;
  constraint a { soft x > 2; soft x < 10;
  constraint b { soft x == 3; }
endclass

M obj = new();

obj.x.rand_mode(0);
obj.randomize();
```

*Solution: x = 0*

# Desirable Debugger Support

- Need to know if a soft constraint is dropped or honored
  - Some graphical way of displaying this information is useful for debug.
- Need to know <u>why</u> a soft constraint is dropped
  - This may include at least one hard constraint or a soft constraint of a higher priority
  - Some interactive mechanism to confirm that if a soft constraint were not dropped (e.g. by converting it to a hard constraint), there would be no solution.

# Conclusion – Benefits of Soft

- Clear syntax to identify individual constraints intended for typical application but not necessarily for always
- Automatic testbench adaptation when faced with conflicting testcase constraints
- Easy way to disable any unwanted soft constraints
- Well-defined semantics for interactions with hard constraints and other soft constraints