Soft Constraints for SystemVerilog

By Akiva Michelson – Ace Verification

What are Soft constraints:

Soft constraints are constraints which hold true unless contradicted by another constraint.

For example if I have the following constraints:

1. soft a == 10
2. a == 5
3. soft b inside 10 to 20
4. b inside 20 to 30

The resolution of these constraints are "a = 5 and b = 20"
In the first case constraint 1 is contradicted by constraint 2 so the second and non-soft constraint takes precedence, while in the case of b, both constraints can be resolved by the number 20.

The difference between a soft constraint failing, and a constraint failing is that in the case of a soft constraint, the alternate constraint takes precedence and the generation continues with no error.

Why use soft constraints:

Soft constraints are used to define *default* values, and *default distribution* for generated fields. For example if I have created an ethernet packet I might want to add in a few constraints to assure legal packets are generated by default.

Example:

```
constraint pkt_len {
        packet_length inside {[64:1518]};
}
constraint valid_crc {
        gen_valid_crc == 1;
}
```

By adding these non-soft constraints, the user of this packet class can generate the packets within these ranges. Any generation outside of these ranges will need special attention, in terms of either changing the values pre/post generation or turning the constraint off.

If, however, soft constraints were employed, additional constraints could override the default values, by adding new constraints like the following:

```
constraint test_constraint {
    gen_valid_crc == 0;
}

or
// In procedural code
packet.randomize() with packet_length == 1680;
```

Layered constraints:

In my experience, verification environments are most robust when they are built in a layered fashion.

A. Component layer
Absolute constraints of the class/object being constrained.
B. Knobs layer
Default values and distributions used in the current DUT's verification environment
C. Test Layer
Specific constraints for the current test (only if required)

For example:
In a packet generator there is a lowest level of constraints for what the packet generator is designed to handle.

Example of absolute constraints:

```
constraint absolute_c {
        pkt_length > 6;
        pkt_length < 10000;
 }
```

Example of environment constraints:

```
constraint env_c {
        pkt_length dist { 64 :/ 10 ; [65:128] :/ 60 ; [129:1000] :/ 30};
 }
```

Example of test constraints:

```
constraint t1_c {
        pkt_length == 64;
}
```


In most cases these three level of constraints live in harmony. When the test does not add a constraint the 'default' environment constraints are used, and when the test does use a constraint the constraint overrides the environment constraint.

The following examples show problems which may arise:
Test1

```
constraint t2_c {
        pkt_length dist { 1200 :/ 10 ; [1201:1300] :/ 10 ; [1301:1500] :/ 10};
 }
```

Test2

```
constraint t3_c {
        pkt_length == 15;
};
```

In both of these cases the test will contradict the environment constraints. In SystemVerilog the user is given the option to "turn off" the constraint. But this needs to be done in procedural code and the user needs to know the name of the constraint or constraints[1].

Solution:

Ace Verification has developed a small library for providing pseudo soft constraints for SystemVerilog users.

The package is available free of charge at the following link:
http://www.aceverification.com/softconstraints_pkg.htm

The library provides four macros which can be used as soft constraints.

```
`soft_eq(<Variable>, <value>)      // The variable is constraint to the value
`soft_gt(<Variable>, <value>)      // The variable is constrained to greater than the value
`soft_lt(<Variable>, <value>)      // The variable is constrained to less than the value
`soft_rng(<Variable>,<value>)      // Allows the user to define a 'soft' range for a variable
```

For example:

```
class example2;

   rand bit [7:0]  b1;
   rand bit [7:0]  b2;

   // Soft constraints (Can be overwritten by other constraints
   constraint c_1 {
      `soft_eq(b1,32); // b1 will be equal to 32
      `soft_lt(b2,48); // b2 will be generated less then 48
   }
endclass
```


[1] In my opinion turning off constraints is a poor programming practice which yields far more problems than it solves