

**Using Polymorphism &
Randomization in SystemVerilog**

- By Shailesh Dave





Using Polymorphism & Randomization in SystemVerilog

Some significant features of SystemVerilog are mentioned below, which make it a powerful tool for enabling faster and more comprehensive verification.

Polymorphism

Using this property, engineers can extend the definition of class using OOP. Users can derive a new class from a base-class and add more properties to class definition. Moreover, virtual methods allow overriding base class method definition in derived classes. This makes the class more versatile and the environment becomes configurable & reusable as shown in the example below:

```
class transaction;
    bit[7:0] data,address;
    virtual task transaction();//Empty definition
endclass : transaction

class read_transaction extends transaction;
    virtual task transaction();//Overrides task definition for read transaction sequences
        ..... //Read transaction sequences;
    endtask
endclass : transaction

class write_transaction extends transaction;
    virtual task transaction();//Overrides task definition for write transaction
sequences
    .... //Write transaction sequences;
    endtask
endclass : transaction

//Now we can use the above definitions
transaction tr[2];
read_transaction read_tr = new;
write_transaction write_tr = new;

tr[0] = read_tr ;
tr[1] = write_tr ;

tr[0].transaction();//Executes read transaction
tr[1].transaction();//Executes write transaction
```



Randomization

SystemVerilog supports a variety of constructs for randomization that are useful in generating random as well as directed random stimuli. SystemVerilog adds \$urandom_range() to generate random unsigned numbers in user defined range. In SystemVerilog each class has an inbuilt randomize(). Fields declared with `rand` or `randc` can be randomized using this inbuilt method. Moreover users can configure constraint blocks for class random properties that help generate directed random stimulus. Users can add inline construct using `randomize()` with `{}` construct to generate directed sequence. For Instance:

```
class data_struct;  
  rand bit[7:0] data;  
  constraint data_valid1{  
    data > 10;  
  }  
endclass : data_struct
```



```
data_struct data = new;  
integer result ;  
result = data.randomize(); //Generate data with value > 10  
result = data.randomize() with {data == 15}; //Generate data with value 15
```

Moreover SystemVerilog provides `randcase` to generate weight-based random cases and `randsequence` to generate random sequence of operation. This randomization facility helps users a long way in taking constrained-random approach and with functional coverage also a part of SystemVerilog, would mean Coverage-Driven Verification approach can easily be used.