# Chapter 6
# Randomization

## 6.1 Introduction

As designs grow larger, it becomes more difficult to create a complete set of stimuli needed to check their functionality. You can write a directed testcase to check a certain set of features, but you cannot write enough directed testcases when the number of features keeps doubling on each project. Worse yet, the interactions between all these features are the source for the most devious bugs and are the least likely to be caught by going through a laundry list of features.

The solution is to create test cases automatically using constrained-random tests (CRT). A directed test finds the bugs you think are there, but a CRT finds bugs you never thought about, by using random stimulus. You restrict the test scenarios to those that are both valid and of interest by using constraints.

Creating the environment for a CRT takes more work than creating one for directed tests. A simple directed test just applies stimulus, and then you manually check the result. These results are captured as a golden log file and compared with future simulations to see whether the test passes or fails. A CRT requires an environment to predict the result, using a reference model, transfer function, or other techniques, plus functional coverage to measure the effectiveness of the stimulus. However, once this environment is in place, you can run hundreds of tests without having to hand-check the results, thereby improving your productivity. This tradeoff of test-authoring time (your work) for CPU time (machine work) is what makes CRT so valuable.

A CRT is made of two parts: the test code that uses a stream of random values to create input to the DUT, and a seed to the pseudo-random number generator (PRNG), shown in Section 6.16.1 at the end of this chapter. You can make a CRT behave differently just by using a new seed. This feature allows you to leverage each test so each is the functional equivalent of many directed tests, just by changing seeds. You are able to create more equivalent tests using these techniques than with directed testing.

You may feel that these random tests are like throwing darts. How do you know when you have covered all aspects of the design? The stimulus space is too large to generate every possible input, so you need to generate a useful subset. In Chapter 9 you will learn how to measure verification progress by using functional coverage.

There are many ways to use randomization, and this chapter gives many examples. It highlights the most useful techniques, but choose what works best for you.

## 6.2 What to Randomize

When you think of randomizing the stimulus to a design, the first thing you may think of are the data fields. These are the easiest to create – just call `$random`. The problem is that this approach has a very low payback in terms of bugs found: you only find data-path bugs, perhaps with bit-level mistakes. The test is still inherently directed. The challenging bugs are in the control logic. As a result, you need to randomize all decision points in your DUT. Everywhere control paths diverge, randomization increases the probability that you'll take a different path in each test case.

You need to think broadly about all design input such as the following items.

- Device configuration
- Environment configuration
- Primary input data
- Encapsulated input data
- Protocol exceptions
- Delays
- Transaction status
- Errors and violations

### *6.2.1 Device Configuration*

What is the most common reason why bugs are missed during testing of the RTL design? Not enough different configurations have been tried! Most tests just use the design as it comes out of reset, or apply a fixed set of initialization vectors to put it into a known state. This is like testing a PC's operating system right after it has been installed, and without any applications; of course the performance is fine, and there are no crashes.

Over time, in a real world environment, the DUT's configuration becomes more and more random. In a real world example, a verification engineer had to verify a timedivision multiplexor switch that had 600 input channels and 12 output channels. When the device was installed in the end-customer's system, channels would be allocated and deallocated over and over. At any point in time, there was little correlation between adjacent channels. In other words, the configuration would seem random.

To test this device, the verification engineer had to write several dozen lines of Tcl code to configure each channel. As a result, she was never able to try configurations with more than a handful of channels enabled. Using a CRT methodology, she wrote a testbench that randomized the parameters for a single channel, and then put this in a loop to configure the whole device. Now she had confidence that her tests would uncover bugs that previously would have been missed.

### *6.2.2 Environment Configuration*

The device that you are designing operates in an environment containing other devices. When you are verifying the DUT, it is connected to a testbench that mimics this environment. You should randomize the entire environment, including the number of objects and how they are configured.

Another company was creating a PCI switch that connected multiple buses to an internal memory bus. At the start of simulation the customer used randomization to choose the number of PCI buses (1–4), the number of devices on each bus (1–8), and the parameters for each device (master or slave, CSR addresses, etc.). Even though there were many possible combinations, this company knew all had been covered.

### *6.2.3 Primary Input Data*

This is what you probably thought of first when you read about random stimulus: take a transaction such as a bus write or ATM cell and fill it with some random values. How hard can that be? Actually it is fairly straightforward as long as you carefully prepare your transaction classes. You should anticipate any layered protocols and error injection.

### *6.2.4 Encapsulated Input Data*

Many devices process multiple layers of stimulus. For example, a device may create TCP traffic that is then encoded in the IP protocol, and finally sent out inside Ethernet packets. Each level has its own control fields that can be randomized to try new combinations. So you are randomizing the data and the layers that surround it. You need to write constraints that create valid control fields but that also allow injecting errors.

### *6.2.5 Protocol Exceptions, Errors, and Violations*

Anything that can go wrong, will, eventually. The most challenging part of design and verification is how to handle errors in the system. You need to anticipate all the cases where things can go wrong, inject them into the system, and make sure the design handles them gracefully, without locking up or going into an illegal state. A good verification engineer tests the behavior of the design to the edge of the functional specification and sometimes even beyond.

When two devices communicate, what happens if the transfer stops partway through? Can your testbench simulate these breaks? If there are error detection and correction fields, you must make sure all combinations are tried.

The random component of these errors is that your testbench should be able to send functionally correct stimuli and then, with the flip of a configuration bit, start injecting random types of errors at random intervals.

### *6.2.6 Delays*

Many communication protocols specify ranges of delays. The bus grant comes one to three cycles after request. Data from the memory is valid in the fourth to tenth bus cycle. However, many directed tests, optimized for the fastest simulation, use the shortest latency, except for that one test that only tries various delays. Your testbench should always use random, legal delays during every test to try to find that (hopefully) one combination that exposes a design bug.

Below the cycle level, some designs are sensitive to clock jitter. By sliding the clock edges back and forth by small amounts, you can make sure your design is not overly sensitive to small changes in the clock cycle.

The clock generator should be in a module outside the testbench so that it creates events in the Active region along with other design events. However, the generator should have parameters such as frequency and offset that can be set by the testbench during the configuration phase.

Note that the methodology described in this book is for finding functional errors, not timing errors. Your constrained random testbench should not purposefully violate setup and hold and hold requirements. These are better validated using timing analysis tools.

## 6.3 Randomization in SystemVerilog

The random stimulus generation in SystemVerilog is most useful when used with OOP. You first create a class to hold a group of related random variables, and then have the random-solver fill them with random values. You can create constraints to limit the random values to legal values, or to test specific features.

You can randomize individual variables, but this case is the least interesting. True constrained-random stimuli is created at the transaction level, not one value at a time.

## 6.3.1 *Simple Class with Random Variables*

Sample 6.1 shows a packet class with random variables and constraints, plus testbench code that constructs and randomizes a packet.

**Sample 6.1** Simple random class

```
class Packet;
  // The random variables
  rand  bit [31:0] src, dst, data[8];
  randc bit [ 7:0] kind;
  // Limit the values for src
  constraint c {src > 10;
                src < 15;}
endclass

Packet p;
initial begin
  p = new();// Create a packet
  if (!p.randomize())
    $finish;
  transmit(p);
end
```

This class has four random variables. The first three use the `rand` modifier, so that every time you randomize the class, the variables are assigned a value. Think of rolling dice where each roll could be a new value or repeat the current one. The `kind` variable is `randc`, which means random cyclic, so that the random solver does not repeat a random value until every possible value has been assigned. Think of dealing cards from a deck where you deal out every card in the deck in random order, then shuffle the deck, and deal out the cards in a different order. Note that the cyclic pattern is for a single variable. A `randc` array with five elements has five different patterns, like five decks of cards, dealt in parallel. Simulators are only required to implement `randc` variables up to 8 bits wide with 256 different values, but most support much larger ranges.

A constraint is just a set of relational expressions that must be true for the chosen value of the variables. In this example, the `src` variable must be greater than 10 and less than 15. Note that the constraint expression is grouped using curly braces: {}. This is because this code is declarative, not procedural, which uses `begin`...`end`.

The `randomize` function returns 0 if a problem is found with the constraints. The code checks the result and stops simulation with `$finish` if there is a problem. Alternatively, you might want to call a special routine to end simulation, after doing some housekeeping chores like printing a summary report. The rest of the book uses a macro instead of this extra code.

You should not randomize an object in the class constructor. Your test may need to turn constraints on or off, change weights, or even add new constraints before randomization. The constructor is for initializing the object's variables, and if you called `randomize` at this early stage, you might end up throwing away the results.

Variables in your classes should be random and public. This gives your test the most control over the DUT's stimulus and control. You can disable randomization of a variable, as shown in Section 6.11.2. If you forget to make a variable random, you must edit the environment, which you want to avoid. The exception is that configuration variables such as weights and limits should not be random in transaction classes as their values are chosen at the start of simulation and do not change.

## 6.3.2 *Checking the Result from Randomization*

The `randomize` function assigns random values to any variable in the class that has been labeled as `rand` or `randc`, and also makes sure that all active constraints are obeyed. Randomization can fail if your code has conflicting constraints (see next section), so you should always check the status. If you don't check, the variables may get unexpected values, causing your simulation to fail.

The remaining code samples in this book employ the macro in Sample 6.2 to check for the result of randomization. If you adopt this style, you can easily add code to give meaningful error messages and gracefully wind down simulation. The macro shows off several coding tricks, including wrapping the generated code in a `do…while` statement so it can be used like a normal statement terminated with a semicolon, including in an `if-else` statement, something that VMM log macros did right, but not OVM.

**Sample 6.2** Randomization check macro and example

```
‘define SV_RAND_CHECK(r) \
   do begin \
     if (!(r)) begin \
       $display("%s:%0d: Randomization failed \"%s\"", \
                `__FILE__, `__LINE__, `"r`"); \
       $finish; \
     end \
   end while (0)

initial begin
  Packet p = new();                 // Create a packet
  ‘SV_RAND_CHECK(p.randomize());    // Randomize it
end
```

### 6.3.3 *The Constraint Solver*

The process of solving constraint expressions is handled by the SystemVerilog constraint solver. The solver chooses values that satisfy the constraints. The values come from SystemVerilog's PRNG, that is started with an initial seed. If you give a SystemVerilog simulator the same seed and the same testbench, it should always produce the same results. Note that changing the tool version or switches such as debug level can change results. See the exercises at the end of this chapter to see how to specify the initial seed.

The solver is specific to the simulation vendor, and a constrained-random test may not give the same results when run on different simulators, or even on different versions of the same tool. The SystemVerilog standard specifies the meaning of the expressions, and the legal values that are created, but does not detail the precise order in which the solver should operate. See Section 6.16 for more details on random number generators.

### 6.3.4 *What can be Randomized?*

SystemVerilog allows you to randomize integral variables, that is, variables that contain a simple set of bits. This includes 2-state and 4-state types, though randomization only generates 2-state values. You can have integers, bit vectors, etc. You cannot have a random string, or refer to a handle in a constraint. Randomizing `real` variables is not yet defined in the LRM.

## 6.4 Constraint Details

Useful stimulus is more than just random values — there are relationships between the variables. Otherwise, it may take too long to generate interesting stimulus values, or the stimulus might contain illegal values. You define these interactions in SystemVerilog using constraint blocks that contain one or more constraint expressions. SystemVerilog chooses random values so that the expressions are true.

At least one variable in each expression should be random, either `rand` or `randc`. The following class fails when randomized, unless `age` happens to be in the right range. The solution is to add the modifier `rand` or `randc` before `age`.

**Sample 6.3** Constraint without random variables

```
class Child;
  bit [7:0] age;  // Error - should be rand or randc
  constraint c_teenager {age > 12;
                         age < 20;}
endclass
```

The `randomize` function tries to assign new values to random variables and to make sure all constraints are satisfied. In Sample 6.3, since there are no random variables, `randomize` just checks the value of `age` to see if it is in the bounds specified by the constraint `c_teenager`. Unless the variable falls in the range of 13:19, `randomize` fails. While you can use a constraint to check that a non-random variable has a valid value, use an `assert` or `if`-statement instead. It is much easier to debug your procedural checker code than read through an error message from the random solver.

### 6.4.1 Constraint Introduction

Sample 6.4 shows a simple class with random variables and constraints. The specific constructs are explained in the following sections. Notice that in constraint blocks, you use curly braces, `{ }`, to group together multiple expressions. The `begin...end` keywords are for procedural code.

**Sample 6.4** Constrained-random class

```
class Stim;
  const bit [31:0] CONGEST_ADDR = 42;
  typedef enum {READ, WRITE, CONTROL} stim_e;
  randc stim_e kind;   // Enumerated var
  rand bit [31:0] len, src, dst;
  rand bit congestion_test;

  constraint c_stim {
    len < 1000;
    len > 0;
    if (congestion_test) {
      dst inside {[CONGEST_ADDR-10:CONGEST_ADDR+10]};
      src == CONGEST_ADDR;
    }
    else
      src inside {0, [2:10], [100:107]};
  }
endclass
```

### 6.4.2 Simple Expressions

Sample 6.4 showed a constraint block with several expressions. The first two control the values for the `len` variable. As you can see, a variable can be used in multiple expressions.

There should be a maximum of only one operator in an expression, such as <, <=, ==, >=, or >. Sample 6.5 shows a SystemVerilog gotcha in that it incorrectly tries to generate three variables in a fixed order.

**Sample 6.5** Bad ordering constraint

```
class Order_bad;
  rand bit [7:0] lo, med, hi;
  constraint bad  {lo < med < hi;} // Gotcha!
endclass
```

**Sample 6.6** Result from incorrect ordering constraint

```
lo =  20, med = 224, hi = 164
lo = 114, med =  39, hi = 189
lo = 186, med = 148, hi = 161
lo = 214, med = 223, hi = 201
```

Sample 6.6 shows the results, which are not what was intended. The constraint `bad` in Sample 6.5 is broken down into multiple binary relational expressions, going from left to right: `((lo < med) < hi)`. First, the expression `(lo < med)` is evaluated, which gives 0 or 1. Then `hi` is constrained to be greater than the result. The variables `lo` and `med` are randomized but not constrained. The correct constraint is shown in Sample 6.7. For more examples, see Sutherland (2007).

**Sample 6.7** Constrain variables to be in a fixed order

```
class Order_good;
  rand bit [7:0] lo, med, hi;
  constraint good {lo < med;   // Only use binary constraints
                   med < hi;}
endclass
```

### 6.4.3 *Equivalence Expressions*

The most common mistake with constraints is trying to make an assignment in a constraint block, which can only contain expressions. Instead, use the equivalence operator to set a random variable to a value, e.g., `len==42`. You can build complex relationships between one or more random variables: `len == (header.addr_mode * 4 + payload.size())`.

### 6.4.4 *Weighted Distributions*

A bug in the DUT may be found with constrained random stimulus if you apply enough patterns. However, it may take a long time for a particular corner case to be generated. When reviewing functional coverage result, see if corner cases are being generated. If not, you can use a weighted distribution to skew the stimulus in a particular direction, and thus accelerate finding bugs. The `dist` operator allows you to create weighted distributions so that some values are chosen more often than others.

The `dist` operator takes a list of values and weights, separated by the `:=` or the `:/` operator. The values and weights can be constants or variables. The values can be a single value or a range such as `[lo:hi]`. The weights are not percentages and do not have to add up to 100. The `:=` operator specifies that the weight is the same for every specified value in the range, whereas the `:/` operator specifies that the weight is to be equally divided between all the values.

**Sample 6.8** Weighted random distribution with `dist`

```
class Transaction;
  rand bit [1:0] src, dst;
  constraint c_dist {
     src dist {0:=40, [1:3]:=60};
     // src = 0, weight = 40/220
     // src = 1, weight = 60/220
     // src = 2, weight = 60/220
     // src = 3, weight = 60/220

     dst dist {0:/40, [1:3]:/60};
     // dst = 0, weight = 40/100
     // dst = 1, weight = 20/100
     // dst = 2, weight = 20/100
     // dst = 3, weight = 20/100
  }
endclass
```

In Sample 6.8, `src` gets the value 0, 1, 2, or 3. The weight of 0 is 40, whereas, 1, 2, and 3 each have the weight of 60, for a total of 220. The probability of choosing 0 is 40/220, and the probability of choosing 1, 2, or 3 is 60/220 each.

Next, `dst` gets the value 0, 1, 2, or 3. The weight of 0 is 40, whereas 1, 2, and 3 share a total weight of 60, for a total of 100. The probability of choosing 0 is 40/100, and the probability of choosing 1, 2, or 3 is only 20/100 each.

Once again, the values and weights can be constants or variables. You can use variable weights to change distributions on the fly or even to eliminate choices by setting the weight to zero, as shown in Sample 6.9.

**Sample 6.9** Dynamically changing distribution weights

```
// Bus operation, byte, word, or longword
class BusOp;
  // Operand length
  typedef enum {BYTE, WORD, LWRD } length_e;
  rand length_e len;

  // Weights for dist constraint
  bit [31:0] w_byte=1, w_word=3, w_lwrd=5;

  constraint c_len {
    len dist {BYTE := w_byte,      // Choose a random
              WORD := w_word,      // length using
              LWRD := w_lwrd};     // variable weights
  }
endclass
```

In Sample 6.9, the `len` enumerated variable has three values. With the default weighting values, longword lengths are chosen more often, as `w_lwrd` has the largest value. Don't worry, you can change the weights on the fly during simulation to get a different distribution.

### *6.4.5 Set Membership and the Inside Operator*

You can create sets of values with the `inside` operator. The SystemVerilog solver chooses between the values in the set with equal probability, unless you have other constraints on the variable. As always, you can use variables in the sets.

**Sample 6.10** Random sets of values

```
class Ranges;
  rand bit [31:0] c;      // Random variable
  bit [31:0] lo, hi;      // Non-random variables used as limits
  constraint c_range {
    c inside {[lo:hi]};   // lo <= c && c <= hi
  }
endclass
```

In Sample 6.10, SystemVerilog uses the values for `lo` and `hi` to determine the range of possible values. You can use the variables as parameters for your constraints so that the testbench can alter the behavior of the stimulus generator without rewriting the constraints. Note that if `lo` > `hi`, an empty set is formed, and the constraint fails.

If you want any value, as long as it is not inside a set, invert the constraint with the NOT operator: `!` as shown in Sample 6.11.

**Sample 6.11** Inverted random set constraint

```
constraint c_range {
  !(c inside {[lo:hi]});  // c < lo or c > hi
}
```

## 6.4.6 *Using an Array in a Set*

Sample 6.12 shows how you can choose from a set of values by storing them in an array.

**Sample 6.12** Random set constraint for an array

```
class Fib;
  rand bit [7:0] f;
  bit [7:0] vals[] = '{1,2,3,5,8};
  constraint c_fibonacci {
    f inside vals;
  }
endclass
```

This is expanded into the constraints in Sample 6.13.

**Sample 6.13** Equivalent set of constraints

```
constraint c_fibonacci {
  (f == vals[0]) ||   // f==1
  (f == vals[1]) ||   // f==2
  (f == vals[2]) ||   // f==3
  (f == vals[3]) ||   // f==5
  (f == vals[4]);     // f==8
}
```

Likewise, you can use the NOT operator to tell SystemVerilog to choose any value except those in an array as shown in Sample 6.14.

**Sample 6.14** Choose any value except those in an array

```
class NotFib;
  rand bit [7:0] notf;
  bit [7:0] vals[] = '{1,2,3,5,8};
  constraint c_fibonacci {
    !(notf inside vals);
  }
endclass
```

Always make sure your constraints work as you expect. You could create functional coverage groups and generate reports, or print a histogram of values with the code in Sample 6.15, with the output in Sample 6.16.

**Sample 6.15** Printing a histogram

```
initial begin
  Fib fib;
  int count[9], maxx[$];

  fib = new();
  repeat (20_000) begin
    `SV_RAND_CHECK(fib.randomize());
    count[fib.f]++;         // Count the number of hits
  end
  maxx = count.max();       // Get largest value in count

  // Print histogram of count
  foreach(count[i])
    if (count[i]) begin
      $write("count[%0d]=%5d ", i, count[i]);
      repeat (count[i]*40/maxx[0]) $write("*");
      $display;
    end
end
```

**Sample 6.16** Histogram for inside constraint

```
count[1]= 3980 **************************************
count[2]= 3924 *************************************
count[3]= 3922 *************************************
count[5]= 4175 ****************************************
count[8]= 3999 **************************************
```

Samples 6.17 and 6.18 choose a day of the week from a list of enumerated values. You can change the list of choices on the fly. If you make `choice` a `randc` variable, the simulator tries every possible value before repeating.

**Sample 6.17** Class to choose from an array of possible values

```
class Days;
  typedef enum {SUN, MON, TUE, WED,
                THU, FRI, SAT} days_e;
  days_e choices[$];
  rand days_e choice;
  constraint cday {choice inside choices;}
endclass
```

**Sample 6.18** Choosing from an array of values

```
initial begin
  Days days;
  days = new();

  days.choices = {Days::SUN, Days::SAT};
  `SV_RAND_CHECK(days.randomize());
  $display("Random weekend day %s\n", days.choice.name());

  days.choices = {Days::MON, Days::TUE, Days::WED,
                  Days::THU, Days::FRI};
  `SV_RAND_CHECK(days.randomize());
  $display("Random week day %s", days.choice.name());
end
```

The `name` function returns a string with the name of an enumerated value.

If you want to dynamically add or remove values from a set, think twice before using the `inside` operator because of its performance. Perhaps you have a set of values that you want to choose just once. You could use `inside` to choose values from a queue, and delete them to slowly shrink the queue. This requires the solver to solve N constraints, where N is the number of elements left in the queue. Instead, use a `randc` variable that is an index into an array of choices as shown in Samples 6.19 and 6.20. Choosing a `randc` value takes a short, constant time, whereas solving a large number of constraints is more expensive, especially if your array has more than a few dozen elements.

**Sample 6.19** Using randc to choose array values in random order

```
class RandcInside;
  int array[];                 // Values to choose
  randc bit [15:0] index;      // Index into array

  function new(input int a[]); // Construct & initialize
    array = a;
  endfunction

  function int pick();         // Return most recent pick
    return array[index];
  endfunction

  constraint c_size {index < array.size();}
endclass
```

**Sample 6.20** Testbench for randc choosing array values in random order

```
initial begin
  RandcInside ri;

  ri = new('{1,3,5,7,9,11,13});
  repeat (ri.array.size()) begin
    `SV_RAND_CHECK(ri.randomize());
    $display("Picked %2d [%0d]", ri.pick(), ri.index);
  end
end
```

Note that constraints and routines can be mixed in any order.

### *6.4.7 Bidirectional Constraints*

By now you may have realized that constraint blocks are not procedural code, executing from top to bottom. They are declarative code, all active at the same time. If you constrain a variable with the `inside` operator with the set `[10:50]` and have another expression that constrains the variable to be greater than 20, SystemVerilog solves both constraints simultaneously and only chooses values between 21 and 50.

SystemVerilog constraints are solved bidirectionally, which means that the constraints on all random variables are solved concurrently. Adding or removing a constraint on any one variable affects the value chosen for all variables that are related directly or indirectly. Consider the constraint in Sample 6.21.

**Sample 6.21** Bidirectional constraints

```
class Bidir;
  rand bit [15:0] r, s, t;
  constraint c_bidir {       // All are solved in parallel
    r < t;                   // A value for r affects s, t
    s == r;
    t < 10;
    s >  5;
  }
endclass
```

The SystemVerilog solver looks at all four constraints simultaneously. The variable `r` has to be less than `t`, which has to be less than 10. However, `r` is also constrained to be equal to `s`, which is greater than 5. Even though there is no direct constraint on the lower value of t, the constraint on s restricts the choices. Table 6.1 shows the possible values for these three variables.

**Table 6.1** Solutions for bidirectional constraint

| *Solution* | *r* | *s* | *t* |
|---|---|---|---|
| A | 6 | 6 | 7 |
| B | 6 | 6 | 8 |
| C | 6 | 6 | 9 |
| D | 7 | 7 | 8 |
| E | 7 | 7 | 9 |
| F | 8 | 8 | 9 |

### *6.4.8 Implication Constraints*

Normally, all constraint expressions are active in a block. What if you want to have an expression active only some of the time? Set the highest address, but only for IO space mode. SystemVerilog supports two implication operators, -> and if.

**Sample 6.22** Constraint block with implication operator

```
class BusOp;
  rand bit [31:0] addr;
  rand bit io_space_mode;
  constraint c_io {
    io_space_mode ->
      addr[31] == 1'b1;
  }
```

The expression A->B is equivalent to the expression (!A || B). When the implication operator appears in a constraint, the solver picks values for A and B so the expression is true. Truth Table 6.2 shows the value of the expression for the logical values of A and B.

**Table 6.2** Implication operator truth table

| *A->B* | *B=false* | *B=true* |
|---|---|---|
| **A=false** | true | true |
| **A=true** | false | true |

When A is true, B must be true, but when A is false, B can be true or false. Note that this is a partly bidirectional constraint, but that A->B does not imply that B->A. The two expressions produce different results.

In Sample 6.23, when d==1, the variable e must be 1, but when e==1, d can be 0 or 1.

**Sample 6.23** Implication operator

```
class LogImp;
  rand bit d, e;
  constraint c {
    (d==1) -> (e==1);
  }
endclass
```

If you add the constraint {e==0;}, the variable d must be 0; But if you add a constraint {e==1;} the values of d are not constrained, it can still be 0 or 1.

Sample 6.24 shows how Sample 6.22 could be written with an if implication constraint.

**Sample 6.24** Constraint block with if implication operator

```
class BusOp;
  rand bit [31:0] addr;
  rand bit io_space_mode;
  constraint c_io {
    if (io_space_mode)
      addr[31] == 1'b1;
  }
```

The if-else operator is a great way to choose between multiple expressions. For example, the bus defined in Sample 6.9 might support byte, word, and longword reads, but only longword writes if written like Sample 6.25.

**Sample 6.25** Constraint block with if-else operator

```
class BusOp;
  rand operand_e op;
  rand length_e len;

  constraint c_len_rw {
    if (op == READ) {
      len inside {[BYTE:LWRD]};
    }
    else {
      len == LWRD;
    }
  }
```

The constraint `if (A) B else C;` is equivalent to the two constraints (`A && B`); and (`!A && C`);. Sample 6.26 shows how you can chain together multiple choices.

**Sample 6.26** Constraint block with multiple if-else operator

```
class BusOp;
  ...
  constraint c_addr_space {
    if (addr_space == MEM)
      addr inside {[0:32'h0FFF_FFFF]};
    else if (addr_space == IO)
      addr inside {[32'1000_0000:32'h7FFF_FFFF]};
    else
      addr inside {[32'h8000_0000:32'hFFFF_FFFF]};
  }
```

### 6.4.9 *Equivalence Operator*

The equivalence operator `<->` is bidirectional. `A<->B` is defined as `((A->B) && (B->A))`. Table 6.3 is the truth table for the logical values of `A` and `B` as constrained in Sample 6.27.

**Table 6.3** Equivalence operator truth table

| *A<->B* | *B=false* | *B=true* |
|---|---|---|
| A=false | true | false |
| A=true | false | true |

**Sample 6.27** Equivalence constraint

```
rand bit d, e;
constraint c { (d==1) <-> (e==1); }
```

When `d` is true, `e` must also be true, and when `d` is false, `e` must also be false. So this operator is the same as a logical XNOR. If you start with the constraint `d<->e`, and add a constraint such as `d==1`, `e` is set to 1 by the solver. The constraint `d<->e` and `e==0` cause `d` to be set to 0 by the solver. If your class has all three of the constraints, `d<->e`, `d==1`, and `e==0`, the solver will not be able to choose values for `d` and `e`.

## 6.5 Solution Probabilities

Whenever you deal with random values, you need to understand the probability of the outcome. SystemVerilog does not guarantee the exact solution found by the random constraint solver, but you can influence the distribution. Any time you work with random numbers, you have to look at thousands or millions of values to average out the noise. Some simulators, such as Synopsys VCS, have multiple solvers to allow you to trade memory usage vs. performance. The distributions will vary between different simulators. The tables were generated with Synopsys VCS 2011.03.

## *6.5.1 Unconstrained*

Start with two random variables in a class with no constraints as shown in Sample 6.28.

**Sample 6.28** Class `Unconstrained`

```
class Unconstrained;
  rand bit x;          // 0 or 1
  rand bit [1:0] y;    // 0, 1, 2, or 3
endclass
```

Table 6.4 shows the eight possible solutions. Since there are no constraints, each has the same probability. You have to run thousands of randomizations to see the actual results approach the listed probabilities.

**Table 6.4** Solutions for `Unconstrained class`

| *Solution* | *x* | *y* | *Probability* |
|---|---|---|---|
| A | 0 | 0 | 1/8 |
| B | 0 | 1 | 1/8 |
| C | 0 | 2 | 1/8 |
| D | 0 | 3 | 1/8 |
| E | 1 | 0 | 1/8 |
| F | 1 | 1 | 1/8 |
| G | 1 | 2 | 1/8 |
| H | 1 | 3 | 1/8 |

## *6.5.2 Implication*

In Sample 6.29, the value of `y` depends on the value of `x`. This is indicated with the implication operator in the following constraint. This example and the rest in this section also behave in the way same with the `if` implication operator.

**Sample 6.29** Class with implication constraint

```
class Imp1;
  rand bit x;          // 0 or 1
  rand bit [1:0] y;    // 0, 1, 2, or 3
  constraint c_xy {
    (x==0) -> (y==0);
  }
endclass
```

Table 6.5 shows the possible solutions and probability. You can see that the random solver recognizes that there are eight combinations of `x` and `y`, but all the solutions where `x==0` (solutions A–D) have been merged together.

**Table 6.5** Solutions for `Imp1` class

| *Solution* | *x* | *y* | *Probability* |
|---|---|---|---|
| A | 0 | 0 | 1/2 |
| B | 0 | 1 | 0 |
| C | 0 | 2 | 0 |
| D | 0 | 3 | 0 |
| E | 1 | 0 | 1/8 |
| F | 1 | 1 | 1/8 |
| G | 1 | 2 | 1/8 |
| H | 1 | 3 | 1/8 |

## 6.5.3 *Implication and Bidirectional Constraints*

Note that the implication operator says that when `x==0, y` is forced to 0, but when `y==0`, there is no constraint on `x`. However, implication is bidirectional in that if `y` were forced to a nonzero value, `x` would have to be 1. Sample 6.30 has the constraint `y>0`, so `x` can never be 0 and Table 6.6 shows the solutions.

**Sample 6.30** Class with implication constraint and additional constraint

```
class Imp2;
  rand bit x;           // 0 or 1
  rand bit [1:0] y;     // 0, 1, 2, or 3
  constraint c_xy {
     y > 0;             // Force y = 1, 2, or 3
    (x==0) -> (y==0);
  }
endclass
```

**Table 6.6** Solutions for `Imp2` class

| *Solution* | *x* | *y* | *Probability* |
|---|---|---|---|
| A | 0 | 0 | 0 |
| B | 0 | 1 | 0 |
| C | 0 | 2 | 0 |
| D | 0 | 3 | 0 |
| E | 1 | 0 | 0 |
| F | 1 | 1 | 1/3 |
| G | 1 | 2 | 1/3 |
| H | 1 | 3 | 1/3 |

### 6.5.4 *Guiding Distribution with* `Solve...Before`

You can guide the SystemVerilog solver using the "`solve...before`" constraint as seen in Sample 6.31.

**Sample 6.31** Class with implication and `solve...before`

```
class SolveXBeforeY;
  rand bit x;           // 0 or 1
  rand bit [1:0] y;     // 0, 1, 2, or 3
  constraint c_xy {
    (x==0) -> y==0;
    solve x before y;
  }
endclass
```

The `solve...before` constraint does not change the solution space, just the probability of the results. The solver chooses values of `x` (0, 1) with equal probability. In 1000 calls to `randomize`, `x` is 0 about 500 times, and 1 about 500 times. When `x` is 0, `y` must be 0. When `x` is 1, `y` can be 0, 1, 2, or 3 with equal probability as shown in Table 6.7.

**Table 6.7** Solutions for `solve x before y` constraint

| *Solution* | *x* | *y* | *Probability* |
|---|---|---|---|
| A | 0 | 0 | 1/2 |
| B | 0 | 1 | 0 |
| C | 0 | 2 | 0 |
| D | 0 | 3 | 0 |
| E | 1 | 0 | 1/8 |
| F | 1 | 1 | 1/8 |
| G | 1 | 2 | 1/8 |
| H | 1 | 3 | 1/8 |

If you use the constraint `solve y before x`, you get a very different distribution as shown in Table 6.8.

**Table 6.8** Solutions for `solve y before x` constraint

| *Solution* | *x* | *y* | *Probability* |
|---|---|---|---|
| A | 0 | 0 | 1/8 |
| B | 0 | 1 | 0 |
| C | 0 | 2 | 0 |
| D | 0 | 3 | 0 |
| E | 1 | 0 | 1/8 |
| F | 1 | 1 | 1/4 |
| G | 1 | 2 | 1/4 |
| H | 1 | 3 | 1/4 |

Only use `solve...before` if you are dissatisfied with how often some values occur. Excessive use can slow the constraint solver and make your constraints difficult for others to understand.

For the simple class in Sample 6.31, the equivalence operator, `<->`, gives the same solution as the implication operator `->`. Try adding additional constraints and plot the results for your favorite simulator.

## 6.6 Controlling Multiple Constraint Blocks

A class can contain multiple constraint blocks. One block might ensure you have a valid transaction, as described in Section 6.7, but you might need to disable this when testing the DUT's error handling. Or you might want to have a separate constraint for each test. Perhaps one constraint would restrict the data length to create small transactions (great for testing congestion), whereas another would make long transactions.

You can turn constraints on and off with the `constraint_mode` function. You can control a single constraint with `*handle.constraint.*constraint_mode*(arg).*` To control all constraints in an object, use `*handle.*constraint_mode*(arg)*`, as shown in Sample 6.32. When the argument for `constraint_mode` is 0, the constraint is turned off, and when it is 1, the constraint is turned on.

**Sample 6.32** Using `constraint_mode`

```
class Packet;
  rand bit [31:0] length;
  constraint c_short {length inside {[1:32]}; }
  constraint c_long  {length inside {[1000:1023]}; }
endclass

Packet p;
initial begin
  p = new();

  // Create a long packet by disabling short constraint
  p.c_short.constraint_mode(0);
  `SV_RAND_CHECK(p.randomize());

  transmit(p);

  // Create a short packet by disabling all constraints
  // then enabling only the short constraint
  p.constraint_mode(0);
  p.c_short.constraint_mode(1);
  `SV_RAND_CHECK(p.randomize());
  transmit(p);
end
```

While many small constraints may give you more flexibility, the process of turning them on and off is more complex. For example, when you turn off all constraints that create data, you are also disabling all the ones that check the data's validity.

If you just want to make a random variable non-random, use `rand_mode` as described in Section 6.11.2.

## 6.7 Valid Constraints

A good randomization technique is to create several constraints to ensure the correctness of your random stimulus, known as “valid constraints.” In Sample 6.33, a bus read-modify-write command is only allowed for a longword data length.

**Sample 6.33** Checking write length with a valid constraint

```
class Transaction;
  typedef enum {BYTE, WORD, LWRD, QWRD} length_e;
  typedef enum {READ, WRITE, RMW, INTR} access_e;
  rand length_e length;
  rand access_e access;

  constraint valid_RMW_LWRD {
    (access == RMW) -> (length == LWRD);
  }
endclass
```

Now you know the bus transaction obeys the rule. Later, if you want to violate the rule, use `constraint_mode` to turn off this one constraint. You can turn these off with `constraint_mode` when you want to generate errors. For example, what if a packet has a zero-length payload? You should have a naming convention to make these constraints stand out, such as using the prefix `valid` as shown above.

## 6.8 In-Line Constraints

As you write more tests, you can end up with many constraints. They can interact with each other in unexpected ways, and the extra code to enable and disable them adds to the test complexity. Additionally, constantly adding and editing constraints to a class could cause problems in a team environment.

Many tests only randomize objects at one place in the code. SystemVerilog allows you to add an extra constraint using `randomize with`. This is equivalent to adding an extra constraint to any existing ones in effect. Sample 6.34 shows a base class with constraints, then two `randomize with` statements.

**Sample 6.34** The `randomize() with` statement

```
class Transaction;
  rand bit [31:0] addr, data;
  constraint c1 {addr inside{[0:100],[1000:2000]};}
endclass

initial begin
  Transaction t;
  t = new();

  // addr is 50-100, 1000-1500, data < 10
  `SV_RAND_CHECK(t.randomize() with {addr >= 50; addr <= 1500;
                                     data < 10;});

  driveBus(t);

  // force addr to a specific value, data > 10
  `SV_RAND_CHECK(t.randomize() with {addr == 2000; data > 10;});

  driveBus(t);
end
```

The extra constraints are added to the existing ones in effect. Use `constraint_mode` if you need to disable a conflicting constraint. Note that inside the `with{}` statement, SystemVerilog uses the scope of the class. That is why Sample 6.34 used just `addr`, not `t.addr`.

A common mistake is to surround your in-line constraints with parenthesis instead of curly braces `{}`. Just remember that constraint blocks use curly braces, so your in-line constraint must use them too. Braces are for declarative code.

## 6.9 The `pre_randomize` and `post_randomize` Functions

Sometimes you need to perform an action immediately before every `randomize` call or immediately afterwards. For example, you may want to set some nonrandom class variables (such as limits or weights) before randomization starts, or you may need to calculate the error correction bits for random data. SystemVerilog lets you do this with two functions, `pre_randomize` and `post_randomize` that are created automatically in any class with random variables.

### *6.9.1 Building a Bathtub Distribution*

For some applications, you want a nonlinear random distribution. For instance, small and large packets are more likely to find a design bug such as buffer overflow than medium-sized packets. So you want a bathtub shaped distribution; high on both ends, and low in the middle. You could build an elaborate `dist` constraint, but it might require lots of tweaking to get the shape you want. Verilog has several functions for nonlinear distribution, such as `$dist_exponential`, but none for a bathtub. The graph in Fig. 6.1 shows how you can combine two exponential curves to make a bathtub curve. The `pre_randomize` method in Sample 6.35 calculates a point on an exponential curve, then randomly chooses to put this on the left curve, or right. As you pick points on either the left and right curves, you gradually build a distribution of the combined values.

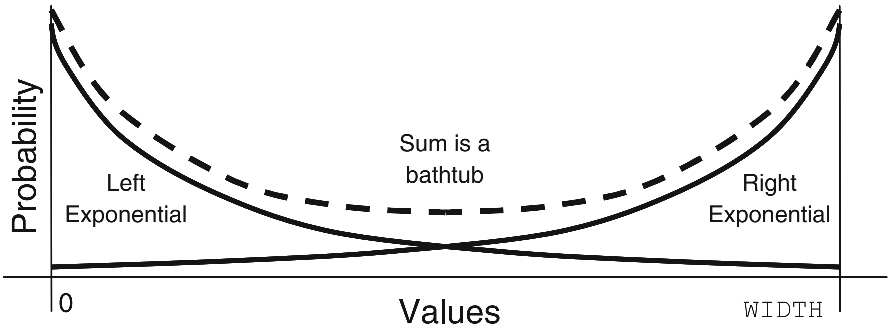


**Fig. 6.1** Building a bathtub distribution

**Sample 6.35** Building a bathtub distribution

```
class Bathtub;
  int value;  // Random variable with bathtub dist
  int WIDTH = 50, DEPTH=6, seed=1;

  function void pre_randomize();
    // Calculate an exponental curve
    value = $dist_exponential(seed, DEPTH);
    if (value > WIDTH) value = WIDTH;

    // Randomly put this point on the left or right curve
    if ($urandom_range(1))      // Random 0 or 1
      value = WIDTH - value;
  endfunction

endclass
```

Every time this object is randomized, the variable `value` gets updated. Across many randomizations, you will see the desired nonlinear distribution. Since the variable is calculated procedurally, not through the random constraint solver, it does not need the `rand` modifier.

See Sample 6.64 for another example of `post_randomize`.

### 6.9.2 *Note on Void Functions*

The functions `pre_randomize` and `post_randomize` can only call other functions, not tasks that could possibly consume time. You cannot have a delay in the middle of a call to `randomize`. When you are debugging a randomization problem, you can call your display routines if you planned ahead and made them void functions.

Chapter 8 describes advanced OOP concepts including extended classes and virtual methods. The `pre_randomize` and `post_randomize` functions are not virtual and so they are called based on the type of the handle, not the object. Additionally, if your extended class's `pre_randomize or post_randomize` need functionality in the base class's `pre_randomize` and `post_randomize` functions, they should call these methods using the super prefix, as in `super.pre_randomize`.

## 6.10 Random Number Functions

You can use all the Verilog-1995 distribution functions, plus several that are new for SystemVerilog. Consult a statistics book for more details on the "dist" functions. Some of the useful functions include the following.

- `$random` — Flat distribution, returning signed 32-bit random
- `$urandom` — Flat distribution, returning unsigned 32-bit random
- `$urandom_range` — Flat distribution over a range
- `$dist_exponential` — Exponential decay, as shown in Fig. 6.1
- `$dist_normal` — Bell-shaped distribution
- `$dist_poisson` — Bell-shaped distribution
- `$dist_uniform` — Flat distribution

The `$urandom_range` function takes two arguments, an optional low value, and a high value as shown in Sample 6.36.

**Sample 6.36** $urandom range usage

```
a = $urandom_range(3, 10); // Pick a value from 3 to 10
a = $urandom_range(10, 3); // Pick a value from 3 to 10
b = $urandom_range(5);     // Pick a value from 0 to 5
```

## 6.11 Constraints Tips and Techniques

How can you create constrained-random tests that can be easily modified? There are several tricks you can use. The most general technique is to use OOP to extend the original class as described in sections 6.11.8 and 8.2.4 but this requires more planning. So first learn some simple techniques, but keep your mind open to other ways.

### *6.11.1 Constraints with Variables*

Most constraint examples in this book use constants to make them more readable. In Sample 6.37, `length` is randomized over a range that uses a variable for the upper bound.

**Sample 6.37** Constraint with a variable bound

```
class Packet;
  rand bit [31:0] length;
  bit [31:0] max_length = 100;  // Configuration, not rand
  constraint c_length {
    length inside {[1:max_length]};
  }
endclass
```

By default, this class creates random lengths between 1 and 100, but by changing the variable `max_length`, you can vary the upper limit.

You can use variables in the `dist` constraint to turn on and off values and ranges. In Sample 6.38, each bus command has a different weight variable.

**Sample 6.38** `dist` constraint with variable weights

```
typedef enum {READ8, READ16, READ32} read_e;
class ReadCommands;
  rand read_e read_cmd;
  int read8_wt=1, read16_wt=1, read32_wt=1;
  constraint c_read {
    read_cmd dist {READ8  := read8_wt,
                   READ16 := read16_wt,
                   READ32 := read32_wt};
    }
endclass
```

By default, this constraint produces each command with equal probability. If you want to have a greater number of `READ8` commands, increase the `read8_wt` weight variable. Most importantly, you can turn off generation of a command by dropping its weight to 0.

### *6.11.2 Using Nonrandom Values*

If you have a set of constraints that produces stimulus that is almost what you want, but not quite, you could call `randomize`, and then set a variable to the value you want — you don't have to use the random value. However, your stimulus values may not be correct according to the constraints you created to check validity.

If there are just a few random variables that you want to override, use the `rand_mode` function to make them nonrandom. When you call this method with the argument 0 for a random variable, the `rand` or `randc` qualifier is disabled and the variable's value is no longer changed by the random solver, but the value is still checked in if it appears in a constraint. Setting the random mode to 1 turns the qualifier back on so the variable can changed by the solver.

**Sample 6.39** `rand_mode` disables randomization of variables

```
// Packet with variable length payload
class Packet;
  rand bit [7:0] length, payload[];
  constraint c_valid {length > 0;
                      payload.size() == length;}

  function void display(input string msg);
    $display("\n%s:", msg);
    $write("\tPacket len=%0d, bytes = ", length);
    for(int i=0; (i<4 && i<payload.size()); i++)
      $write(" %0d", payload[i]);
    $display;
  endfunction
endclass

Packet p;
initial begin
  p = new();
  `SV_RAND_CHECK(p.randomize());  // Randomize all variables
  p.display("Simple randomize");

  p.length.rand_mode(0);          // Make length nonrandom,
  p.length = 42;                  // set it to a constant value
  `SV_RAND_CHECK(p.randomize());  // then randomize the payload
  p.display("Randomize with rand_mode");
end
```

In Sample 6.39, the packet size is stored in the random variable `length`. The first half of the test randomizes both the `length` variable and the contents of the `payload` dynamic array. The second half calls `rand_mode` to make `length` a nonrandom variable, sets it to 42, then calls `randomize`. The constraint sets the `payload` size at the constant 42, but the array is still filled with random values.

### 6.11.3 *Checking Values Using Constraints*

If you randomize an object and then modify some variables, you can check that the object is still valid by checking if all constraints are still obeyed. Call `handle.randomize(null)` and SystemVerilog treats all variables as nonrandom ("state variables") and just ensures that all constraints are satisfied, i.e all expressions are true. If any constraints are not satisfied, the `randomize` function returns 0.

### 6.11.4 *Randomizing Individual Variables*

Suppose you want to randomize a few variables inside a class. You can call `randomize` with the subset of variables. Only those variables passed in the argument list will be randomized; the rest will be treated as state variables and not randomized. All constraints remain in effect. In Sample 6.40, the first call to randomize only changes the values of two `rand` variables `med` and `hi`. The second call only changes the value of `med`, whereas `hi` retains its previous value. Surprisingly, you can pass a non-random variable, as shown in the last call, and `low` is given a random value, as long as it obeys the constraint.

**Sample 6.40** Randomizing a subset of variables in a class

```
class Rising;
  bit [7:0] low;            // Not random
  rand bit [7:0] med, hi;   // Random variable
  constraint up
    { low < med; med < hi; } // See Section 6.4.2
endclass

initial begin
  Rising r;
  r = new();
  r.randomize();      // Randomize med, hi; low untouched
  r.randomize(med);   // Randomize only med
  r.randomize(low);  // Randomize only low, even though not rand
end
```

This trick of only randomizing a subset of the variables is not commonly used in real testbenches as you are restricting the randomness of your stimulus. You want your testbench to explore the full range of legal values, not just a few corners.

### 6.11.5 *Turn Constraints Off and On*

Sections 6.6 and 6.7 discuss valid constraints and `constraint_mode`. Turning off individual constraints is fine for error generation, but should be used in moderation.

### 6.11.6 *Specifying a Constraint in a Test Using In-Line Constraints*

If you keep adding constraints to a class, it becomes hard to manage and control. Soon, everyone is checking out the same file from your source control system. Many times a constraint is only used by a single test, so why have it visible to every test? One way to localize the effects of a constraint is to use in-line constraints, `randomize with`, shown in Section 6.8. This works well if your new constraint is additive to the default constraints. If you follow the recommendations in Section 6.7 to create "valid constraints", you can quickly constrain valid sequences. For error injection, you can disable any constraint that conflicts with what you are trying to do. A test that injects a particular flavor of corrupted data would first turn off the particular validity constraint that checks for that error.

There are several tradeoffs with using in-line constraints. The first is that now your constraints are in multiple locations which can make it more difficult to understand all the active constraints. If you add a new constraint to the original class, it may conflict with the in-line constraint. The second is that it can be very hard for you to reuse an in-line constraint across multiple tests. By definition, an in-line constraint only exists in one piece of code. You could put it in a routine in a separate file and then call it as needed. At that point it has become nearly the same as an external constraint.

### 6.11.7 *Specifying a Constraint in a Test with External Constraints*

The body of a constraint does not have to be defined within the class, just as a routine body can be defined externally, as shown in Section 5.10 . Your data class could be defined in one file, with one empty constraint. Then each test could define its own version of this constraint to generate its own flavors of stimulus as shown in Samples 6.41 and 6.42.

**Sample 6.41** Class with an external constraint

```
// packet.sv
class Packet;
  rand bit [7:0] length;
  rand bit [7:0] payload[];
  constraint c_valid {length > 0;
                      payload.size() == length;}
  constraint c_external;
endclass
```

**Sample 6.42** Program defining an external constraint

```
// test.sv
program automatic test;
`include "packet.sv"
  constraint Packet::c_external {length == 1;}
  ...
endprogram
```

External constraints have several advantages over in-line constraints. They can be put in a file and thus reused between tests. An external constraint applies to all instances of the class, whereas an in-line constraint only affects the single call to `randomize`. Consequently, an external constraint provides a primitive way to change a class without having to learn advanced OOP techniques. Keep in mind that with this technique, you can only add constraints, not alter existing ones, and you need to define the external constraint prototype in the original class.

Like in-line constraints, external constraints can cause problems, as the constraints are spread across multiple files. The LRM requires external constraints to be defined in the same scope as the original class. A class defined in a package must have its external constraint also defined in the same package, limiting its usefulness. That is why Sample 6.42 includes the class definition rather than using a package.

A final consideration is what happens when the body for an external constraint is never defined. The SystemVerilog LRM does not currently specify what should happen in this case. Before you build a testbench with many external constraints, find out how your simulator handles missing definitions. Is this an error that prevents simulation, just a warning, or no message at all?

### *6.11.8 Extending a Class*

In Chapter 8, you will learn how to extend a class. With this technique, you can take a testbench that uses a given class, and swap in an extended class that has additional or redefined constraints, routines, and variables. See Sample 8.10 for a typical testbench. Note that if you define a constraint in an extended class with the same name as one in the base class, the extended constraint replaces the base one.

Learning OOP techniques requires a little more study, but the flexibility of this new approach repays with great rewards.

## 6.12 Common Randomization Problems

You may be comfortable with procedural code, but writing constraints and understanding random distributions requires a new way of thinking. Here are some issues you may encounter when trying to create random stimulus.

### *6.12.1 Use Signed Variables with Care*

When creating a testbench, you may be tempted to use the `int`, `byte`, or other signed types for counters and other simple variables. Don't use them in random constraints unless you really want signed values. What values are produced when the class in Sample 6.43 is randomized? It has two random variables and wants to make the sum of them 64.

**Sample 6.43** Signed variables cause randomization problems

```
class SignedVars;
  rand byte pkt1_len, pkt2_len;
  constraint total_len {
    pkt1_len + pkt2_len == 64;
  }
endclass
```

Obviously, you could get pairs of values such as (32, 32) and (2, 62). Additionally, you could see (–63, 127), as this is a legitimate solution of the equation, even though it may not be what you wanted. To avoid meaningless values such as negative lengths, use only unsigned random variables, as shown in Sample 6.44.

**Sample 6.44** Randomizing unsigned 32-bit variables

```
class Vars32;
  rand bit [31:0] pkt1_len, pkt2_len;  // unsigned type
  constraint total_len {
    pkt1_len + pkt2_len == 64;
  }
endclass
```

Even this version causes problems, as large values of `pkt1_len` and `pkt2_len`, such as `32'h80000040` and `32'h80000000`, wrap around when added together and give `32'd64` or `32'h40`. You might think of adding another pair of constraints to restrict the values of these two variables, but the best approach is to make them only as wide as needed, and to avoid using 32-bit variables in constraints. In Sample 6.45, the sum of two 8-bit variables is compared to a 9-bit value.

**Sample 6.45** Randomizing unsigned 8-bit variables

```
class Vars8;
  rand bit [7:0] pkt1_len, pkt2_len;   // 8-bits wide
  constraint total_len {
    pkt1_len + pkt2_len == 9'd64;       // 9-bit sum
  }
endclass
```

### 6.12.2 Solver Performance Tips

Each constraint solver has its strengths and weaknesses but there are some guidelines that you can follow to improve the speed of your simulations with constrained random variables. Tools are always being improved, so check with your vendor for more specific information.

If you just need to fill an array with raw data, don't use the solver as it has some overhead choosing values, even for a variable that has no constraints. Don't declare these arrays as `rand`, instead calculate the values in `pre_randomize` with `$urandom` or `$urandom_range`. These functions calculate a value up to 100 times faster than the solver, which is important when you need a 1000 values quickly. Generally, the larger the array, the less important are the individual values, and the less likely that there is a need to use a solver. Even if you need a non-uniform range of values, or there is a simple relationship between values, you might be able to employ an `if` statement.

### 6.12.3 Choose the Right Arithmetic Operator to Boost Efficiency

Simple arithmetic operators such as addition and subtraction, bit extracts, and shifts are handled very efficiently by the solver in a constraint. However, multiplication, division, and modulo are very expensive with 32-bit values. Remember that any constant without an explicit size, such as `42`, is treated as a 32-bit value, `32'd42`.

If you want to generate random addresses that are near a page boundary, where a page is 4096 bytes, you could write the following code, but the solver may take a long time to find suitable values for `addr` if you use the constraint in Sample 6.46.

**Sample 6.46** Expensive constraint with mod and unsized variable

```
rand bit [31:0] addr;
constraint slow_near_page_boundary {
  addr % 4096 inside {[0:20], [4075:4095]};
}
```

Many constants in hardware are powers of 2, so take advantage of this with a bit extraction rather than division and modulo. Only constrain the bits that matter, not the upper bits. Likewise, multiplication by a power of two can be replaced by a shift. Note that some constraint solvers make these optimizations automatically Sample 6.47 replaces the MOD operator with a bit extract.

**Sample 6.47** Efficient constraint with bit extract

```
rand bit [31:0] addr;
constraint near_page_boundry {
  addr[11:0] inside {[0:20], [4075:4095]};
}
```

## 6.13 Iterative and Array Constraints

The constraints presented so far allow you to specify limits on single variables. What if you want to randomize an array? The `foreach` constraint and several array functions let you shape the distribution of the values.

Using the `foreach` constraint creates many constraints that can slow down simulation. A good solver can quickly solve hundreds of constraints but may slow down with thousands. Especially slow are nested `foreach` constraints, as they produce N$^2$ constraints for an array of size N. See Section 6.13.5 for an algorithm that used `randc` variables instead of nested `foreach`.

### *6.13.1 Array Size*

The easiest array constraint to understand is the `size` function. In Sample 6.48, you are specifying the number of elements in a dynamic array or queue.

**Sample 6.48** Constraining dynamic array size

```
class dyn_size;
  rand bit [31:0] d[];
  constraint d_size {d.size() inside {[1:10]}; }
endclass
```

Using the `inside` constraint lets you set a lower and upper boundary on the array size. In many cases you may not want an empty array, that is, `size==0`. Remember to specify an upper limit; otherwise, you can end up with thousands or millions of elements, which can cause the random solver to take an excessive amount of time.

### *6.13.2 Sum of Elements*

You can send a random array of data into a design, but you can also use it to control the flow. Perhaps you have an interface that has to transfer four data words. The words can be sent consecutively or over many cycles. A strobe signal tells when the data signal is valid. Figure 6.2 shows some legal strobe patterns, sending four values over ten cycles.

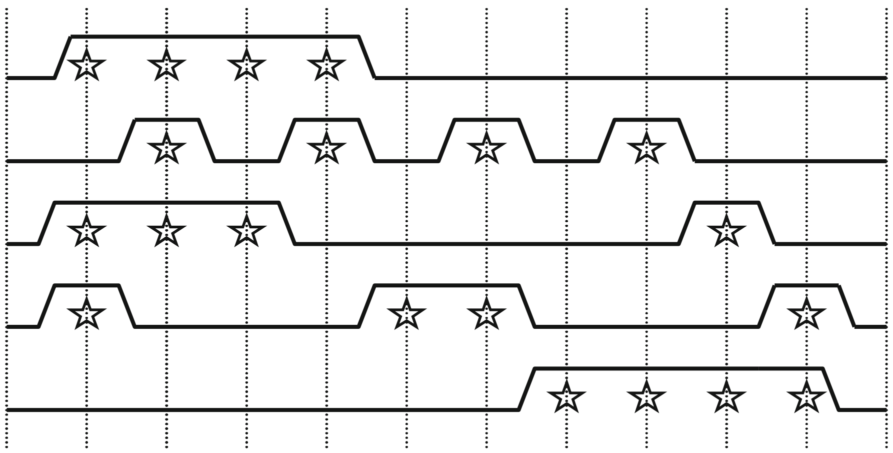


**Fig. 6.2** Random strobe waveforms

You can create these patterns using a random array as shown in Sample 6.49. Constrain it to have four bits enabled out of the entire range using the `sum` function.

**Sample 6.49** Random strobe pattern class

```
class StrobePat;
  rand bit strobe[10];
  constraint c_set_four { strobe.sum() == 4'h4; }
endclass

initial begin
  StrobePat sp;
  int count = 0;        // Index into data array

  sp = new();
  `SV_RAND_CHECK(sp.randomize());

  foreach (sp.strobe[i]) begin
    ##1 bus.cb.strobe <= sp.strobe[i];
    // If strobe is enabled, drive out next data word
    if (sp.strobe[i])
      bus.cb.data <= data[count++];
  end
end
```

As you remember from Chapter 2, the sum of an array of single-bit elements would normally be a single bit, e.g., 0 or 1. Sample 6.49 compares `strobe.sum` to a 4-bit value (`4'h4`), so the sum is calculated with 4-bit precision. The example uses 4-bit precision to store the maximum number of elements, which is 10.

### 6.13.3 *Issues with Array Constraints*

The `sum` function looks simple but can cause several problems because of Verilog's arithmetic rules. The following is a simple problem that one of the authors experienced creating constrained random stimulus. You want to generate from one to eight transactions, such that the total length of all of them is less than 1024 bytes. Sample 6.50 shows a first attempt, 6.51 has the test program, and 6.52 shows the output. The `len` field is a byte in the original transaction.

**Sample 6.50** First attempt at sum constraint: `bad_sum1`

```
class bad_sum1;
  rand byte len[];
  constraint c_len {len.sum() < 1024;
                    len.size() inside {[1:8]};}

  function void display();
    $write("sum=%4d, val=", len.sum());
    foreach(len[i]) $write("%4d ", len[i]);
    $display;
  endfunction
endclass
```

**Sample 6.51** Program to try constraint with array sum

```
program automatic test;
  bad_sum1 c;
  initial begin
    c = new();
    repeat (5) begin
      `SV_RAND_CHECK(c.randomize());
      c.display();
    end
  end
endprogram
```

**Sample 6.52** Output from `bad_sum1`

```
sum=  81, val=  62 -20  39
sum=  39, val= -27  67   1  76 -97 -58  77
sum=  38, val=  60 -22
sum=  72, val=-120  29 123 102 -41 -21
sum= -53, val= -58 -85-115 112-101 -62
```

This generates some smaller lengths, but the sum is sometimes negative and is always less than 127 — definitely not what you wanted! Sample 6.53 shows another attempt, but this time replace the `byte` data type with an unsigned field. The `display` function is unchanged. Sample 6.54 shows the output.

**Sample 6.53** Second attempt at sum constraint: `bad_sum2`

```
class bad_sum2;
  rand bit [7:0] len[];  // 8 bits unsigned, not byte
  constraint c_len {len.sum() < 1024;
                    len.size() inside {[1:8]};}
endclass
```

**Sample 6.54** Output from `bad_sum2`

```
sum=  79, val=  88 100 246   2  14 228 169
sum= 120, val=  74  75 141  86
sum=  39, val=  39
sum= 193, val=  31 156 172  33  57
sum= 173, val=  59 150  25 101 138 212
```

Sample 6.53 has a subtle problem. The sum of all transaction lengths is always less than 256, even though you constrained the array sum to be less than 1024. The problem here is that in Verilog, the sum of many 8-bit values is computed using an 8-bit result. Sample 6.55 bumps the `len` field up to 32 bits using the `uint` type from Section 2.8 .

**Sample 6.55** Third attempt at sum constraint: `bad_sum3`

```
class bad_sum3;
  rand uint len[];  // 32 bits
  constraint c_len {len.sum() < 1024;
                    len.size() inside {[1:8]};}
endclass
```

**Sample 6.56** Output from `bad_sum3`

```
sum= 245, val=1348956995 3748256598 985546882 2507174362
sum= 600, val=2072193829 315191491 484497976 3050698208
 2300168220 3988671456 3998079060 970369544
sum=  17, val=1924767007 3550820640 4149215303 3260098955
sum= 440, val=3192781444 624830067 1300652226 4072252356
 3694386235
sum= 864, val=3561488468 733479692
```

Wow – what happened here in Sample 6.56? This is similar to the signed problem in Section 6.12.1, in that the sum of two very large numbers can wrap around to a small number. You need to limit the size based on the comparison in the constraint. Samples 6.57 and 6.58 show the next attempt and result.

**Sample 6.57** Fourth attempt at sum constraint: `bad_sum4`

```
class bad_sum4;
  rand bit [9:0] len[];  // 10 bits, unsigned
  constraint c_len {len.sum() < 1024;
                    len.size() inside {[1:8]};}
endclass
```

**Sample 6.58** Output from `bad_sum4`

```
sum= 989, val= 787 202
sum=1021, val= 564  76 132 235   0   8   6
sum= 872, val= 624 101 136  11
sum= 978, val= 890  88
sum= 905, val= 663 242
```

This does not work either as the individual `len` fields are more than 8 bits, so the `len` values are often greater than 255. You need to specify that each `len` field is between 1 and 255, but use a 10-bit field so they sum correctly. This requires constraining every element of the array, as shown in the following section.

### *6.13.4 Constraining Individual Array and Queue Elements*

SystemVerilog lets you constrain individual elements of an array using `foreach`. While you might be able to write constraints for a fixed-size array by listing every element, the `foreach` style is more compact. The only practical way to constrain a dynamic array or queue is with `foreach` as shown in Samples 6.59 and 6.60.

**Sample 6.59** Simple foreach constraint: `good_sum5`

```
class good_sum5;
  rand uint len[];
  constraint c_len {foreach (len[i])
                      len[i] inside {[1:255]};
                    len.sum() < 1024;
                    len.size() inside {[1:8]};}
endclass
```

**Sample 6.60** Output from `good_sum5`

```
sum=1011, val=  83 249 197 187 152  95  40   8
sum=1012, val= 213 252 213  44 196  20  20  54
sum= 370, val= 118  76 176
sum= 976, val= 233 187  44 157 201  81  73
sum= 412, val= 172 167  73
```

The addition of the constraint for individual elements fixed the example. Note that the `len` array can be 10 or more bits wide, but must be unsigned.

You can specify constraints between array elements as long as you are careful about the endpoints. The class in Sample 6.61 creates an ascending list of values by comparing each element to the previous, except for the first.

**Sample 6.61** Creating ascending array values with `foreach`

```
class Ascend;
  rand uint d[10];
  constraint c {
    foreach (d[i])      // For every element
      if (i>0)          // except the first
        d[i] > d[i-1]; // compare with previous element
  }
endclass
```

How complex can these constraints become? Constraints have been written to solve Einstein's problem (a logic puzzle with five people, each with five separate attributes), the Eight Queens problem (place eight queens on a chess board so that none can capture each other), and even Sudoku.

### 6.13.5 Generating an Array of Unique Values

How can you create an array of random unique values? If your array has N elements, and the element values range from 0..N-1, you can simply use the array `shuffle` function as described in Section 2.6.3 .

What if the range of values is greater than the number of array elements? If you try to make a `randc` array, each array element will be randomized independently, so you are almost certain to get repeated values.

You may be tempted to use a constraint solver to compare every element with every other with nested `foreach` loops as shown in Sample 6.62. This creates over 4000 individual constraints, which could slow down simulation.

**Sample 6.62** Creating unique array values with `foreach`

```
class UniqueSlow;          // Bad code, do not use
  rand bit [7:0] ua[64];
  constraint c {
    foreach (ua[i])        // For every element,
      foreach (ua[j])
        if (i != j)        //   except the diagonals,
          ua[i] != ua[j];  //   compare to other elements
  }
endclass
```

Instead, you should use procedural code as shown in Sample 6.63 with a helper class containing a `randc` variable so that you can randomize the same variable over and over.

**Sample 6.63** Creating unique array values with a randc helper class

```
class randc8;
  randc bit [7:0] val;
endclass

class LittleUniqueArray;
  bit [7:0] ua [64];      // Array of unique values

  function void pre_randomize();
    randc8 rc8;
    rc8 = new();
    foreach (ua[i]) begin
      `SV_RAND_CHECK(rc8.randomize());
      ua[i] = rc8.val;
    end
  endfunction
endclass
```

Samples 6.64 and 6.65 give a more general solution. For example, you may need to assign ID numbers to N bus drivers, which are in the range of 0 to MAX-1 where MAX >=N.

**Sample 6.64** Unique value generator

```
// Create unique random values in a range 0:max-1
class RandcRange;
  randc bit [15:0] value;
  int max_value;  // Maximum possible value

  function new(input int max_value = 10);
    this.max_value = max_value;
  endfunction

  constraint c_max_value {value < max_value;}
endclass
```

**Sample 6.65** Class to generate a random array of unique values

```
class UniqueArray;
  int max_array_size, max_value;
  rand bit [15:0] ua[];       // Array of unique values
  constraint c_size {ua.size() inside {[1:max_array_size]};}

  function new(input int max_array_size=2, max_value=2);
    this.max_array_size = max_array_size;
    // If max_value is smaller than max array size,
    // array could have duplicates, so adjust max_value
    if (max_value < max_array_size)
      this.max_value = max_array_size;
    else
      this.max_value = max_value;
  endfunction

  // Array a[] allocated in randomize(), fill w/unique vals
  function void post_randomize();
    RandcRange rr;
    rr = new(max_value);
    foreach (ua[i]) begin
      `SV_RAND_CHECK(rr.randomize());
      ua[i] = rr.value;
    end
  endfunction

  function void display();
    $write("Size: %3d:", ua.size());
    foreach (ua[i]) $write("%4d", ua[i]);
    $display;
  endfunction
endclass
```

Sample 6.66 has a program. Here is a program that uses the `UniqueArray` class.

**Sample 6.66** Using the `UniqueArray` class

```
program automatic test;
  UniqueArray ua;
  initial begin
    ua = new(50);                         // Max array size = 50

    repeat (10) begin
      `SV_RAND_CHECK(ua.randomize());  // Create random array
      ua.display();                       // Display values
    end
  end
endprogram
```

### 6.13.6 *Randomizing an Array of Handles*

If you need to create multiple random objects, you might create a random array of handles. Unlike an array of integers, you need to allocate all the elements before randomization as the random solver never constructs objects. If you have a dynamic array, allocate the maximum number of elements you may need, and then use a constraint to resize the array as shown in Sample 6.67. A dynamic array of handles can remain the same size or shrink during randomization, but it can never increase in size.

**Sample 6.67** Constructing elements in a random array class

```
parameter MAX_SIZE = 10;

class RandStuff;
  rand bit [31:0] value;
endclass

class RandArray;
  rand RandStuff array[];    // Don't forget rand!

  constraint c {array.size() inside {[1:MAX_SIZE]}; }

  function new();
    array = new[MAX_SIZE];   // Allocate maximum size
    foreach (array[i])
      array[i] = new();
  endfunction;
endclass

RandArray ra;
initial begin
  ra = new();                // Construct array and all objects
  `SV_RAND_CHECK(ra.randomize()); // Randomize array
  foreach (ra.array[i])
    $display(ra.array[i].value);
end
```

The above code works well for a single array randomization. If you need to repeatedly randomize the same array over and over, allocate the array and construct the elements in `pre_randomize`. See Section 5.14.4 for more on arrays of handles.

## 6.14 Atomic Stimulus Generation vs. Scenario Generation

Up until now, you have seen atomic random transactions. You have learned how to make a single random bus transaction, a single network packet, or a single processor instruction. This is a good start, however your job is to verify that the design works with real-world stimuli. A bus may have long sequences of transactions such as DMA transfers or cache fills. Network traffic consists of extended sequences of packets as you simultaneously read e-mail, browse a web page, and download music from the net, all in parallel. Processors have deep pipelines that are filled with the code for routine calls, `for` loops, and interrupt handlers. Generating transactions one at a time is unlikely to mimic any of these scenarios.

### *6.14.1 An Atomic Generator with History*

The easiest way to create a stream of related transactions is to have an atomic generator base some of its random values on ones from previous transactions. The class might constrain a bus transaction to repeat the previous command, such as a write, 80% of the time, and also use the previous destination address plus an increment. You can use the `post_randomize` function to make a copy of the generated transaction for use by the next call to `randomize`.

This scheme works well for smaller cases but gets into trouble when you need information about the entire sequence ahead of time. A DUT may need to know the length of a sequence of network transactions before it starts.

### *6.14.2 Random Array of Objects*

If you want to generate stimulus for a complex, multi-level protocol, you could build up a combination of code and arrays of random objects. The UVM and VMM both allow you to generate random sequences through a sophisticated set of classes and macros. This section shows a simplified random sequence.

One way to generate random sequences is to randomize an entire array of objects. You can create constraints that refer to the previous and next objects in the array, and the SystemVerilog solver solves all constraints simultaneously. Since the entire sequence is generated at once, you can then extract information such as the total number of transactions or a checksum of all data values before the first transaction is sent. Alternatively, you can build a sequence for a DMA transfer that is constrained to be exactly 1024 bytes, and let the solver pick the right number of transactions to reach that goal.

Sample 6.68 shows a simple sequence of transactions, each one with a destination address that is greater than the one before. It builds on the array constraint shown in Sample 6.61.

**Sample 6.68** Simple random sequence with ascending values

```
class Transaction;            // Simple transaction
  rand bit [3:0] src, dst;
endclass

class Transaction_seq;
  rand Transaction items[10]; // Array of transaction handles

  function new();             // Construct the sequence items
    foreach (items[i])
      items[i] = new();
  endfunction // new

  constraint c_ascend         // Each dst addr is greater than
    { foreach (items[i])      // the one before it
        if (i>0)
           items[i].dst > items[i-1].dst;
}
endclass // Transaction_seq

Transaction_seq seq;

initial begin
  seq = new();                // Construct the sequence
  `SV_RAND_CHECK(seq.randomize());  // Randomize it
  foreach (seq.items[i])
    $display("item[%0d] = %0d", i, seq.items[i].dst);
end
```

### *6.14.3 Combining Sequences*

You can combine multiple sequences together to make a more realistic flow of transactions. For example, for a network device, you could make one sequence that resembles downloading e-mail, a second that is viewing a web page, and a third that is entering single characters into web-based form.The techniques to combine these flows is beyond the scope of this book, but you can learn more from the VMM, as described in Bergeron, et al. (2005).

### *6.14.4 Randsequence*

You may find it challenging to write random constraints as they don't execute sequentially like procedural statements. An alternative way to create random sequences is to describe the grammar of a protocol with a declarative style using a syntax similar to BNF (Backus-Naur Form) and random weighted case statements.

SystemVerilog's randsequence construct resembles the algorithmic code that you have traditionally used but can still be challenging.

Sample 6.69 generates a sequence called stream. A stream can be either cfg_read, io_read, or mem_read. The random sequence engine randomly picks one. The cfg_read label has a weight of 1, io_read has twice the weight and so is twice as likely to be chosen as cfg_read. The label mem_read is most likely to be chosen, with a weight of 5.

**Sample 6.69** Command generator using randsequence

```
initial begin
  for (int i=0; i<15; i++) begin
    randsequence (stream)
      stream :  cfg_read := 1 |
                io_read  := 2 |
                mem_read := 5;
      cfg_read : { cfg_read_task(); } |
                 { cfg_read_task(); } cfg_read;
      mem_read : { mem_read_task(); } |
                 { mem_read_task(); } mem_read;
      io_read  : { io_read_task(); } |
                 { io_read_task(); } io_read;
    endsequence
  end // for
end

task cfg_read_task();
  ...
endtask
```

A cfg_read can be either a single call to cfg_read_task, or a call to the task followed by another cfg_read. As a result, the task is always called at least once, and possibly many times.

One big advantage of randsequence is that it is procedural code and you can debug it by stepping though the execution, or adding $display statements. When you call randomize for an object, it either all works or all fails, but you can't see the steps taken to get to a result.

There are several problems with using randsequence. The code to generate the sequence is separate and a very different style from the classes with data and constraints used by the sequence. So if you use both randomize and randsequence, you have to master two different forms of randomization. More seriously, if you want to modify a sequence, perhaps to add a new branch or action, you have to modify the original sequence code. You can't just make an extension. As you will see in Chapter 8, you can extend a class to add new code, data, and constraints without having to edit the original class.

## 6.15 Random Control

At this point you may be thinking that this process is a great way to create long streams of random input into your design. Or you may think that this is a lot of work if all you want to do is occasionally to make a random decision in your code. You may prefer a set of procedural statements that you can step through using a debugger.

### *6.15.1 Introduction to `randcase`*

You can use `randcase` to make a weighted choice between several actions, without having to create a class and instance. Sample 6.70 chooses one of the three branches based on the weight. SystemVerilog adds up the weights (1+8+1 = 10), chooses a value in this range, and then picks the appropriate branch. The branches are not order dependent, the weights can be variables, and they do not have to add up to 100%. The function `$urandom_range` is described in Section 6.10.

**Sample 6.70** Random control with `randcase` and `$urandom_range`

```
initial begin
  bit [15:0] len;
  randcase
    1: len = $urandom_range(0, 2);  // 10%: 0, 1, or 2
    8: len = $urandom_range(3, 5);  // 80%: 3, 4, or 5
    1: len = $urandom_range(6, 7);  // 10%: 6 or 7
  endcase
  $display("len=%0d", len);
  end
```

You can write Sample 6.70 using a class and the `randomize` function. For this small case, the OOP version in Sample 6.71 is a little larger. However, if this were part of a larger class, the constraint would be more compact than the equivalent `randcase` statement.

**Sample 6.71** Equivalent constrained class

```
class LenDist;
  rand bit [15:0] len;
  constraint c {len dist {[0:2] := 1, [3:5] := 8, [6:7] := 1}; }
endclass

initial begin
  LenDist lenD;
  lenD = new();
  `SV_RAND_CHECK(lenD.randomize());
  $display("len=%0d", lenD.len);
end
```

Code using `randcase` is more difficult to override and modify than random constraints. The only way to modify the random results is to rewrite the code or use variable weights.

Be careful using `randcase`, as it does not leave any tracks behind. For example, you could use it to decide whether or not to inject an error in a transaction. The problem is that the downstream transactors and scoreboard need to know of this choice. The best way to inform them would be to use a variable in the transaction or environment. However, if you are going to create a variable that is part of these classes, you could have made it a random variable and used constraints to change its behavior in different tests.

### *6.15.2 Building a Decision Tree with* `randcase`

You can use the `randcase` statement to create a decision tree. Sample 6.72 has just two levels of procedural code, but you can see how it can be extended to use more.

**Sample 6.72** Creating a decision tree with `randcase`

```
initial begin
  // Level 1
  randcase
    one_write_wt: do_one_write();
    one_read_wt:  do_one_read();
    seq_write_wt: do_seq_write();
    seq_read_wt:  do_seq_read();
  endcase
  end

// Level 2
task do_one_write();
  randcase
    mem_write_wt: do_mem_write();
    io_write_wt:  do_io_write();
    cfg_write_wt: do_cfg_write();
  endcase
endtask

task do_one_read();
  randcase
    mem_read_wt: do_mem_read();
    io_read_wt:  do_io_read();
    cfg_read_wt: do_cfg_read();
  endcase
endtask
```

## 6.16 Random Number Generators

How random is SystemVerilog? On the one hand, your testbench depends on an uncorrelated stream of random values to create stimulus patterns that go beyond any directed test. On the other hand, you need to repeat the patterns over and over during debug of a particular test, even if the design and testbench make minor changes.

### *6.16.1 Pseudorandom Number Generators*

Verilog uses a simple PRNG that you could access with the `$random` function. The generator has an internal state that you can set by providing a seed to `$random`. All IEEE-1364-compliant Verilog simulators use the same algorithm to calculate values.

Sample 6.73 shows a simple PRNG, not the one used by SystemVerilog. The PRNG has a 32-bit state. To calculate the next random value, square the state to produce a 64-bit value, take the middle 32 bits, then add the original value.

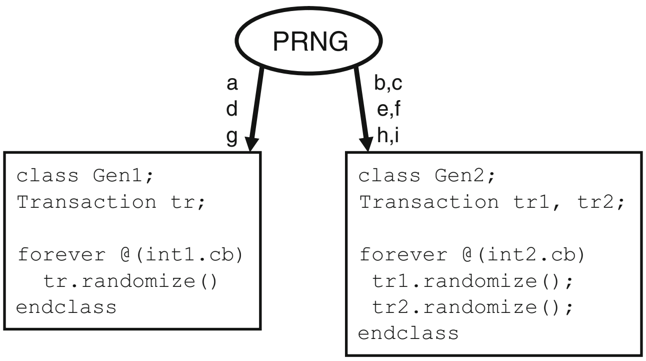
**Sample 6.73** Simple pseudorandom number generator

```
bit [31:0] state = 32'h12345678;
function bit [31:0] my_random();
  bit [63:0] s64;
  s64 = state * state;
  state = (s64 >> 16) + state;
  return state;
endfunction
```

You can see how this simple code produces a stream of values that seem random, but can be repeated by using the same seed value. SystemVerilog calls its own PRNG to generate a new value for `randomize` and `randcase`.

### *6.16.2 Random Stability — Multiple Generators*

Verilog has a single PRNG that is used for the entire simulation. What would happen if SystemVerilog kept this approach? Testbenches often have several stimulus generators running in parallel, creating data for the design under test. If two streams share the same PRNG, they each get a subset of the random values.



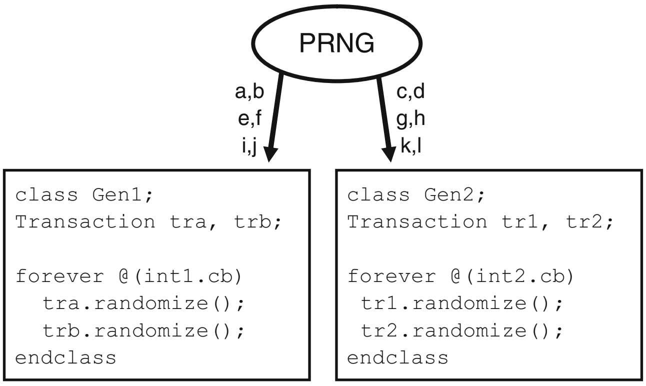
```
class Gen1;
Transaction tr;

forever @(int1.cb)
  tr.randomize()
endclass
```

```
class Gen2;
Transaction tr1, tr2;

forever @(int2.cb)
 tr1.randomize();
 tr2.randomize();
endclass
```

**Fig. 6.3** Sharing a single random generator

In Fig. 6.3, there are two stimulus generators and a single PRNG producing values `a`, `b`, `c`, etc. `Gen2` has two random objects, so during every cycle, it uses twice as many random values as `Gen1`.

A problem can occur when one of the classes changes as shown in Fig. 6.4. `Gen1` gets an additional random variable, and so consumes two random values every time it is called. This approach changes the values used not only by `Gen1`, but also by `Gen2`.

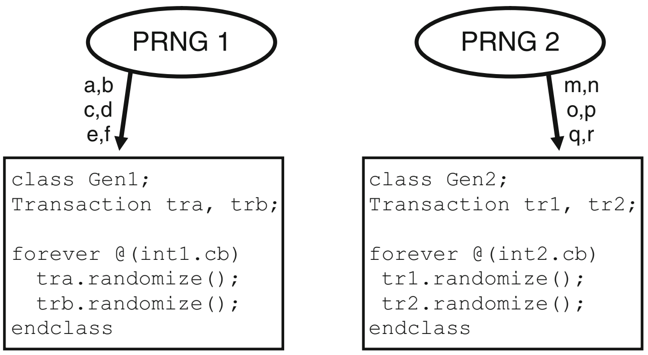


```
class Gen1;
Transaction tra, trb;

forever @(int1.cb)
  tra.randomize();
  trb.randomize();
endclass
```

```
class Gen2;
Transaction tr1, tr2;

forever @(int2.cb)
 tr1.randomize();
 tr2.randomize();
endclass
```

**Fig. 6.4** First generator uses additional values

In SystemVerilog, there is a separate PRNG for every object and thread. Figure 6.5 shows how changes to one object don't affect the random values seen by others.



```
class Gen1;
Transaction tra, trb;

forever @(int1.cb)
  tra.randomize();
  trb.randomize();
endclass
```

```
class Gen2;
Transaction tr1, tr2;

forever @(int2.cb)
 tr1.randomize();
 tr2.randomize();
endclass
```

**Fig. 6.5** Separate random generators per object

### *6.16.3 Random Stability and Hierarchical Seeding*

In SystemVerilog, every object and thread has its own PRNG and unique seed. When a new object or thread is started, its PRNG is seeded from its parent's PRNG. Thus a single seed specified at the start of simulation can create many streams of random stimulus, each distinct.

When you are debugging a testbench, you add, delete, and move code. Even with random stability, your changes may cause the testbench to generate different random values. This can be very frustrating if you are in the middle of debugging a DUT failure, and the testbench no longer creates the same stimulus. You can minimize the effect of code modifications by adding any new objects or threads after existing ones. Sample 6.74 shows a routine from testbench that constructs objects, and runs them in parallel threads.

**Sample 6.74** Test code before modification

```
function void build();
  pci_gen gen0, gen1;
  gen0 = new();
  gen1 = new();
  fork
    gen0.run();
    gen1.run();
  join
endfunction : build
```

Sample 6.75 adds a new generator, and runs it in a new thread. The new object is constructed after the existing ones, and the new thread is spawned after the old ones.

**Sample 6.75** Test code after modification

```
function void build();
  pci_gen gen0, gen1;
  atm_gen new_gen;       // New ATM generator
  gen0 = new();
  gen1 = new();
  new_gen = new();       // Construct new object after old ones

  fork
    gen0.run();
    gen1.run();
    new_gen.run();       // Spawn new thread after old ones
  join
endfunction : build
```

As new code is added, you may not be able to keep the random streams the same as the old ones, but you might be able to postpone any side effects from these changes.

## 6.17 Random Device Configuration

An important part of your DUT to test is the configuration of both the internal DUT settings and the system that surrounds it. As described in Section 6.2.1, your tests should randomize the environment so that you can be confident it has been tested in as many modes as possible.

Sample 6.76 shows a random testbench configuration that can be modified as needed at the test level. The `EthCfg` class describes the configuration for a 4-port Ethernet switch. It is instantiated in an environment class, which in turn is used in the test. The test overrides one of the configuration values, enabling all 4 ports.

**Sample 6.76** Ethernet switch configuration class

```
class EthCfg;

  rand bit [ 3:0] in_use;       // Ports used in test:3,2,1,0
  rand bit [47:0] mac_addr[4];  // MAC addresses
  rand bit [ 3:0] is_100;       // 100mb mode for ports 3,2,1,0
  rand uint run_for_n_frames;   // # frames in test

  // Force some addr bits when running in unicast mode
  constraint local_unicast {
    foreach (mac_addr[i])
      mac_addr[i][41:40] == 2'b00;
  }

  constraint reasonable {       // Limit test length
    run_for_n_frames inside {[1:100]};
  }

endclass
```

The configuration class is used in the `Environment` class during several phases. The configuration is constructed in the `Environment` constructor, but not randomized until the `gen_cfg` phase as shown in Sample 6.77. This allows you to turn constraints on and off before `randomize` is called. Afterwards, you can override the generated values before the `build` phase creates the virtual components around the DUT. (The classes such as `EthGen` and `EthMii` are not shown).

**Sample 6.77** Building environment with random configuration

```
class Environment;

  EthCfg cfg;
  EthGen gen[4];
  EthMii drv[4];

  function new();
    cfg = new();                      // Construct the cfg
  endfunction

  // Use random configuration to build the environment
  function void build();
    foreach (gen[i]) begin
      gen[i] = new();
      drv[i] = new();
      if (cfg.is_100[i])
         drv[i].set_speed(100);
    end
  endfunction
  function void gen_cfg();
    `SV_RAND_CHECK(cfg.randomize());  // Randomize the cfg
  endfunction

  task run();
    foreach (gen[i])
      if (cfg.in_use[i]) begin
        // Only start the testbench transactors that are in-use
        gen[i].run();
        ...
    end
  endtask

  task wrap_up();
    // Not currently used
  endtask
endclass : Environment
```

Now you have all the components to build a test, which is described in a program block. The test in Sample 6.78 instantiates the environment class and then runs each step.

**Sample 6.78** Simple test using random configuration

```
program automatic test;

  Environment env;

  initial begin
    env = new();     // Construct environment
    env.gen_cfg();   // Create random configuration
    env.build();     // Build the testbench environment
    env.run();       // Run the test
    env.wrap_up();   // Clean up after test & report
  end

endprogram
```

You may want to override the random configuration, perhaps to reach a corner case. The test in Sample 6.79 randomizes the configuration class and then enables all the ports.

**Sample 6.79** Simple test that overrides random configuration

```
program automatic test;

  Environment env;

  initial begin
    env = new();     // Construct environment
    env.gen_cfg();   // Create random configuration

    // Override random in_use – turn all 4 ports on
    env.cfg.in_use = '1;

    env.build();     // Build the testbench environment
    env.run();       // Run the test
    env.wrap_up();   // Clean up after test & report
  end

endprogram
```

Notice how in Sample 6.77 all generators were constructed, but only a few were run, depending on the random configuration. If you only constructed the generators that are in-use, you would have to surround any reference to `gen [i]` with a test of `in_use [i]`, otherwise your testbench would crash when it tried to refer to the non-existent generator. The extra memory taken up by these generators that are not used is a small price to pay for a more stable testbench.

## 6.18 Conclusion

Constrained-random tests are the only practical way to generate the stimulus needed to verify a complex design. SystemVerilog offers many ways to create a random stimulus and this chapter presents many of the alternatives.

A test needs to be flexible, allowing you either to use the values generated by default or to constrain or override the values so that you can reach your goals. Always plan ahead when creating your testbench by leaving sufficient “hooks” so that you can steer the testbench from the test without modifying existing code.

## 6.19 Exercises

1. Write the SystemVerilog code for the following items.
   a. Create a class `Exercise1` containing two random variables, 8-bit `data` and 4-bit `address`. Create a constraint block that keeps `address` to 3 or 4.
   b. In an `initial` block, construct an `Exercise1` object and randomize it. Check the status from randomization.
2. Modify the solution for Exercise 1 to create a new class `Exercise2` so that:
   a. `data` is always equal to 5
   b. The probability of `address==0` is 10%
   c. The probability of `address` being between [1:14] is 80%
   d. The probability of `address==15` is 10%
3. Using the solution to either Exercise 1 or 2, demonstrate its usage by generating 20 new `data` and `address` values and check for success from the constraint solver.
4. Create a testbench that randomizes the `Exercise2` class 1000 times.
   a. Count the number of times each `address` value occurs and print the results in a histogram. Do you see an exact 10% / 80% / 10% distribution? Why or why not?
   b. Run the simulation with 3 different random seeds, creating histograms, and then comment on the results. Here is how to run a simulation with the seed 42.

      VCS: `> simv +ntb_random_seed=42`
      IUS: `> irun exercise4.sv -svseed 42`
      Questa: `> vsim -sv_seed 42`
5. For the code in Sample 6.4, describe the constraints on the `len, dst`, and `src` variables.
6. Complete Table 6.9 below for the following constraints.

```
class MemTrans;
  rand bit x;
  rand bit [1:0] y;
  constraint c_xy {
    y inside{[x:3]};
    solve x before y;
  }
endclass
```

**Table 6.9** Solution probabilities

| *Solution* | *x* | *y* | *Probability* |
|---|---|---|---|
| A | 0 | 0 | |
| B | 0 | 1 | |
| C | 0 | 2 | |
| D | 0 | 3 | |
| E | 1 | 0 | |
| F | 1 | 1 | |
| G | 1 | 2 | |
| H | 1 | 3 | |

7. For the following class, create:

   a. A constraint that limits read transaction addresses to the range 0 to 7, inclusive.
   b. Write behavioral code to turn off the above constraint. Construct and randomize a `MemTrans` object with an in-line constraint that limits read transaction addresses to the range 0 to 8, inclusive. Test that the in-line constraint is working.

```
class MemTrans;
  rand bit rw; // read if rw=0, write if rw=1
  rand bit [7:0] data_in;
  rand bit [3:0] address;
endclass // MemTrans
```

8. Create a class for a graphics image that is 10x10 pixels. The value for each pixel can be randomized to black or white. Randomly generate an image that is, on average, 20% white. Print the image and report the number of pixels of each type.

9. Create a class, `StimData`, containing an array of integer samples. Randomize the size and contents of the array, constraining the size to be between 1 and 1000. Test the constraint by generating 20 transactions and reporting the size.

10. Expand the `Transaction` class below so back-to-back transactions of the same type do not have the same address. Test the constraint by generating 20 transactions.

```
package my_package;

  typedef enum {READ, WRITE} rw_e;

  class Transaction;
    rw_e old_rw;
    rand rw_e rw;
    rand bit [31:0] addr, data;
    constraint rw_c{if (old_rw == WRITE) rw != WRITE;};

    function void post_randomize;
      old_rw = rw;
    endfunction

    function void print_all;
      $display("addr = %d, data = %d, rw = %s",
                addr, data, rw);
    endfunction

  endclass

endpackage
```

11. Expand the `RandTransaction` class below so back-to-back transactions of the same type do not have the same address. Test the constraint by generating 20 transactions.

```
class Transaction;
  rand rw_e rw;
  rand bit [31:0] addr, data;
endclass


class RandTransaction;

  rand Transaction trans_array[];

  constraint  rw_c {foreach (trans_array[i])
    if ((i>0) && (trans_array[i-1].rw == WRITE))
      trans_array[i].rw != WRITE;}

  function new();
    trans_array = new[TESTS];
    foreach (trans_array[i])
      trans_array[i] = new();
  endfunction

endclass
```