SystemVerilog Testbench Constructs

VCS®/VCSi™Version X-2005.06 LCA
August 2005

The SystemVerilog features of the Native Testbench technology in VCS documented here are currently available to customers as a part of an Limited Access program. Using these features requires additional LCA license features. Please contact you local Synopsys AC for more details.

Comments?
E-mail your comments about this manual to
vcs_support@synopsys.com.

1

SystemVerilog Testbench Constructs

The new version of VCS has implemented some of the SystemVerilog testbench constructs. As testbench constructs they must be in a `program` block (see “Program Blocks” on page 1-2).

VCS has implemented these as LCA (Limited Customer Availability) features. Contact your local Synopsys AC for licensing information.

Enabling The Use of SystemVerilog Testbench Contructs

You enable the use of SystemVerilog testbench constructs with the `-sverilog` compile-time option. Descriptions and examples of these constructs follow this section.

Program Blocks

A program block is intended to contain the testbench for a design. In the current implementation of SystemVerilog testbench constructs, all these constructs must be in one program block.

Requiring these constructs in a program block help to distinguish between the code that is the testbench and the code that is the design.

Program blocks begin with the keyword `program`, followed by a name for the program, followed by an optional port connection list, followed by a semi colon (`;`). Program blocks end with the keyword `endprogram`, for example:

```
program prog (input clk,output logic [31:0] data, output
logic ctrl);
logic dynamic_array [];
logic assoc_array[*];
int intqueue [$] = {1,2,3};

class classA;
function void vfunc(input in1, output out1);
⋮
endfunction
⋮
endclass

semaphore sem1 =new (2);
mailbox mbx1 = new();

reg [7:0] reg1;
covergroup cg1 @(posedge clk);
cp1: coverpoint reg1;
```

```
.
.
.
endgroup

endprogram


bit clk = 0;
logic [31:0] data;
logic ctrl;

module clkmod;
.
.
.
prog prog1 (clk,data,ctrl); // instance of the program
.
.
.
endmodule
```

In many ways a program definition resembles a module definition and a program instance is similar to a leaf module instance but with special execution semantics.

A program block can contain the following:

- data type declarations including initial values. Dynamic arrays, associative arrays, and queues are implemented for program blocks.
- user-defined tasks and functions
- initial blocks for procedural code (but not always blocks)
- class definitions
- semaphores
- mailboxes
- coverage groups

When VCS executes all the statements in the initial blocks in a program, simulation comes to and end.

The constructs that follow this section must be in a program block.

Arrays

Dynamic Arrays

Dynamic arrays are unpacked arrays with a size that can be set or changed during simulation. The syntax for a dynamic array is as follows:

```
data_type name [];
```

The empty brackets specify a dynamic array.

The currently supported data types for dynamic arrays are as follows:

`bit`	`logic`	`reg`	`byte`
`int`	`longint`	`shortint`	`integer`
`time`	`string`	`class`	`enum`

The official description of dynamic arrays begins on page 35 of the SystemVerilog 3.1a LRM.

The new[ ] Built-In Function

The `new[]` built-in function is for specifying a new size for a dynamic array and optionally specifying another array whose values are assigned the dynamic array. Its syntax is as follows:

```
array_identifier = new[size] (source_array);
```

The optional `(source_array)` argument specifies another array (dynamic or fixed-size) whose values VCS assigns to the dynamic array. If you don't specify the `(source_array)` argument, VCS initializes the elements of the newly allocated array to their default value.
The optional `(source_array)` argument must have the same data type as the array on the left-hand side, but it need not have the same size. If the size of `(source_array)` is less than the size of the new array, VCS initializes the extra elements to their default values. If the size of `(source_array)` is greater than the size of the new array, VCS ignores the additional elements.

```
program prog;
⋮
bit bitDA1 [];
bit bitDA2 [];
bit bitDA3 [];
bit bitSA1 [100];
logic logicDA [];
⋮
initial
begin
bitDA1 = new[100];
bitDA2 = new[100] (bitDA2);
bitDA3 = new[100] (bitSA1);
logicDA = new[100];
end
⋮
endprogram
```

The official description of the `new[]` built-in function begins on page 35 of the SystemVerilog 3.1a LRM.

The size() Method

The `size` method returns the current size of a dynamic array. You can use this method with the `new[]` built-in function, for example:

```
bitDA3 = new[bitDA1.size] (bitDA1);
```

The official description of the `size()` method begins on page 36 of the SystemVerilog 3.1a LRM.

The delete() Method

The delete method sets a dynamic array's size to 0 (zero). It clears out the dynamic array.

```
bitDA1 = new[3];
$display("bitDA1 after sizing, now size = %0d",bitDA1.size);
bitDA1.delete;
$display("bitDA1 after sizing, now size = %0d",bitDA1.size);
```

VCS displays from this code:

```
bitDA1 after sizing, now size = 3
bitDA1 after sizing, now size = 0
```

The official description of the `delete()` method begins on page 36 of the SystemVerilog 3.1a LRM.

Assignments to and from Dynamic Arrays

You can assign a dynamic array to and from a fixed-size array, queue, or another dynamic array, provided they are of equivalent data types, for example:

```
logic lFA1[2];
logic lDA1[];
initial
begin
$display("lDA1 size = %0d",lDA1.size);
lFA1[1]=1;
lFA1[0]=0;
lDA1=lFA1;
$display("lDA1[1] = %0d", lDA1[1]);
$display("lDA1[0] = %0d", lDA1[0]);
$display("lDA1 size = %0d",lDA1.size);
end
endprogram
```

VCS displays:

```
lDA1 size = 0
lDA1[1] = 1
lDA1[0] = 0
lDA1 size = 2
```

When you assign a fixed-size array to a dynamic array, the dynamic array’s size changes to the size of the fixed-size array. This is also true when you assign a dynamic array with a specified size to another dynamic array, for example:

```
logic lDA1[];
logic lDA2[];
initial
begin
lDA1=new[2];
$display("lDA2 size = %0d",lDA2.size);
lDA1[1]=1;
lDA1[0]=0;
lDA2=lDA1;
$display("lDA2[1] = %0d", lDA2[1]);
$display("lDA2[0] = %0d", lDA2[0]);
$display("lDA2 size = %0d",lDA2.size);
end
endprogram
```

This code displays the following:

```
lDA2 size = 0
lDA2[1] = 1
lDA2[0] = 0
lDA2 size = 2
```

You can assign a dynamic array to a fixed-size array, provided that they are of equivalent data type and that the current size of the dynamic array matches the size of the fixed-size array.

The official description of assignments to dynamic arrays begins on page 37 of the SystemVerilog 3.1a LRM.

Associative Arrays

An associative array has a lookup table for the elements of its declared data type. Its index is a data type which serves as the lookup key for the table. This index data type also establishes an order for the elements.

The syntax for declaring an associative array is as follows:

```
data_type array_id [index_type];
data_type array_id [* | string];
```

Where:

data_type
: Is the data type of the associative array.

array_id
: Is the name of the associative array.

index_type

Specifies the type of index. Only two types are currently implemented. They are as follows:

`*`

Specifies a wildcard index.

`string`

Specifies a string index.

The official description of associative arrays begins on page 39 of the SystemVerilog 3.1a LRM.

Wildcard Indexes

You can enter the wildcard character as the index.

```
data_type array_id [*];
```

Using the wildcard character permits entering any integral data type as the index. Integral data types represent an integer (`shortint`, `int`, `longint`, `byte`, `bit`, `logic`, `reg`, `integer`, and also packed structs, packed unions, and `enum`.

```
program m;
bit [2:0] AA1[*];
int int1;
logic [7:0] log1;

initial begin
    int1 = 27;
    log1 = 42;
    AA1[456] = 3'b101;
    AA1[int1] = 3'b000;   // index is 27
    AA1[log1] = 3'b111;   // index is 42
end
endprogram
```

The official description of associative array wildcard indexes begins on page 39 of the SystemVerilog 3.1a LRM.

String Indexes

A string index specifies that you can index the array with a string. You specify a string index with the keyword `string`.

```
program p;

logic [7:0] a[string];
string string_variable;

initial begin
        a["sa"] = 8;
        a["bb"] = 15;
        a["ec"] = 29;
        a["d"] = 32;
        a["e"] = 45;
        a[string_variable] = 1;

end
endprogram
```

The official description of associative array string indexes begins on page 40 of the SystemVerilog 3.1a LRM.

Associative Array Assignments and Arguments

You can only assign an associative array to another associative array with a equivalent data type. Similarly, you can only pass an associative array as an argument to another associative array with a equivalent data type.

The official description of associative array assignments and arguments begins on page 44 of the SystemVerilog 3.1a LRM.

Associative Array Methods

There are methods for analyzing and manipulating associative arrays.

`num`
: Returns the number of entries in the array.

`delete`
: Removes all entries from an array. If you specify an index, this method removes the entry specified by the index.

`exists`
: Returns a 1 if the specified entry exists.

`first`
: Assigns the value of the smallest or alphabetically first entry in the array. Returns 0 if the array is empty and returns 1 if the array contains a value.

`last`
: Assigns the value of the largest or alphabetically last entry in the array. Returns 0 if the array is empty and returns 1 if the array contains a value.

`next`
: Finds the entry whose index is greater than the given index. If there is a next entry, the index variable is assigned the index of the next entry, and the function returns 1. Otherwise, variable is unchanged, and the function returns 0

`prev`
: Finds the entry whose index is smaller than the given index. If there is a previous entry, the index variable is assigned the index of the previous entry, and the function returns 1. Otherwise, variable is unchanged, and the function returns 0.

The following example shows how to use these methods.

```
program p;
logic [7:0] a[string];
string s_index;
initial begin
        a["sa"] = 8;
        a["bb"] = 15;
        a["ec"] = 29;
        a["d"] = 32;
        a["e"] = 45;
        $display("number of entries = %0d",a.num);
        if(a.exists("sa"))
           $display("string \"sa\" is in a");
        if(a.first(s_index))
           begin
            $display("the first entry is \"%s\"",s_index);
             do
               $display("%s : %0d",s_index,a[s_index]);
             while (a.next(s_index));
           end
        if(a.last(s_index))
           begin
            $display("the last entry is \"%s\"",s_index);
             do
               $display("%s : %0d",s_index,a[s_index]);
             while (a.prev(s_index));
           end
        a.delete;
        $display("number of entries = %0d",a.num);
        end
endprogram
```

VCS displays the following:

```
number of entries = 5
string "sa" is in a
the first entry is "bb"
bb : 15
d : 32
e : 45
```

```
ec : 29
sa : 8
the last entry is "sa"
sa : 8
ec : 29
e : 45
d : 32
bb : 15
number of entries = 0
```

The official description of associative array methods begins on page 41 of the SystemVerilog 3.1a LRM.

Queues

A queue is an ordered collection of variables with the same data type. The length of the queue changes during simulation. You can read any variable in the queue, and insert a value anywhere in the queue.

The variables in the queue are its elements. Each element in the queue has a number: 0 is the number of the first, you can specify the last element with the `$` (dollar sign) symbol. The following are some examples of queue declarations:

`logic logque [$];`
: This is a queue of elements with the `logic` data type.

`int intque [$] = {1,2,3};`
: This is a queue of elements with the `int` data type. These elements are initialized 1, 2, and 3.

`string strque [$] = {"first","second","third","fourth"};`
: This is a queue of elements with the `string` data type. These elements are initialized `"first"`, `"second"`, `"third"`, and `"fourth"`.

You assign the elements to a variable using the element number, for example:

```
string s1, s2, s3, s4;
initial
begin
s1=strque[0];
s2=strque[1];
s3=strque[2];
s4=strque[3];
$display("s1=%s s2=%s s3=%s s4=%s",s1,s2,s3,s4);
⋮
end
```

The `$display` system task displays:

```
s1=first s2=second s3=third s4=fourth
```

You also assign values to the elements using the element number, for example:

```
int intque [$]v = {1,2,3};
initial
begin
intque[0]=4;
intque[1]=5;
intque[2]=6;
$display("intque[0]=%0d intque[1]=%0d intque[2]=%0d",
intque[0],intque[1],intque[2]);
⋮
end
```

The `$display` system task displays:

```
intque[0]=4 intque[1]=5 intque[2]=6
```

Concatenation operations, for adding elements, are not yet supported, for example:

```
intque = {0,intque};
intque = {intque, 4};
```

Removing elements from a queue are not yet supported, for example:

```
strque = strque [1:$];
intque = intque[0:$-1];
```

The official description of queues begins on page 45 of the SystemVerilog 3.1a LRM.

Queue Methods

There are the following built-in methods for queues:

`size`

Returns the size of a queue.

```
program prog;
int intque [$] = {1,2,3};

initial
begin
for (int i = 0; i < intque.size; i++)
     $display(intque[i]);
end
endprogram
```

`insert`

Inserts new elements into the queue. This method takes two arguments: the first is the number of the element, the second is the new value.

```
program prog;
string strque [$] = {"first","second","third","forth"};
```

```
initial
begin
for (int i = 0; i < strque.size; i++)
     $write(strque[i]," ");
$display(" ");
strque.insert(1,"next");
strque.insert(2,"somewhere");
for (int i = 0; i < strque.size; i++)
     $write(strque[i]," ");
$display(" ");
end
endprogram
```

The `$display` system tasks display the following:

```
first second third forth
first next somewhere second third forth
```

`delete`

Removes an element from the queue, specified by element number. If you don’t specify an element number, this method deletes all elements in the queue.

```
string strque [$] = {"first","second","third"};

initial
begin
for (int i =0; i<strque.size; i++)
     $write(strque[i]," ");
$display(" ");
strque.delete(1);
for (int i =0; i<strque.size; i++)
     $write(strque[i]," ");
end
```

The system tasks display the following:

```
first second third
first third
```

`pop_front`

Removes and returns the first element of the queue.

```
string strque [$] = {"first","second","third"};
string s1;
initial
begin
$write("the elements of strque are ");
for (int i =0; i<strque.size; i++)
     $write(strque[i]," ");
$display("\ns1 before pop contains %s ",s1);
s1 = strque.pop_front;
$display("s1 after pop contains %s ",s1);
$write("the elements of strque are ");
for (int i =0; i<strque.size; i++)
     $write(strque[i]," ");
end
```

The system tasks display the following:

```
the elements of strque are first second third
s1 before pop contains
s1 after pop contains first
the elements of strque are second third
```

```
string strque [$] = {"first","second","third"};
string s1;
initial
begin
for (int i =0; i<strque.size; i++)
     $write(strque[i]," ");
$display(" ");
s1 = strque.pop_front;
for (int i =0; i<strque.size; i++)
     $write(strque[i]," ");
end
```

The system tasks display the following:

```
first second third
second third
```

`pop_back`

Removes and returns the last element in the queue.

```
program prog;
string strque [$] = {"first","second","third"};
string s1;
initial
begin
$write("the elements of strque are ");
for (int i =0; i<strque.size; i++)
     $write(strque[i]," ");
$display("\ns1 before pop contains %s ",s1);
s1 = strque.pop_back;
$display("s1 after pop contains %s ",s1);
$write("the elements of strque are ");
for (int i =0; i<strque.size; i++)
     $write(strque[i]," ");
end
endprogram
```

The system tasks display the following:

```
the elements of strque are first second third
s1 before pop contains
s1 after pop contains third
the elements of strque are first second
```

`push_front` and `push_back`

Add elements to the front and back of the queue.

```
int intque [$] = {1,2,3};

initial
begin
for( int i = 0; i < intque.size; i++)
     $write(intque[i]," ");
intque.push_front(0);
intque.push_back(4);
$write(" \n");
for( int i = 0; i < intque.size; i++)
```

```
        $write(intque[i]," ");
    end
```

The system tasks display the following:

```
1    2    3
0    1    2    3    4
```

The official description of queue methods begins on page 46 of the SystemVerilog 3.1a LRM.

Classes

The user-defined data type, class, is composed of data members of valid SystemVerilog data types (known as properties) and tasks or functions (known as methods) for manipulating the data members. The properties and methods, taken together, define the contents and capabilities of a class instance (also referred to as an "object").

Note: unpacked structures and unpacked unions inside classes are not yet supported.

Use the `class` and `endclass` keywords to declare a class.

```
class  B;
     int q = 3;
     function int send (int a);
          send = a * 2;
     endfunction

     task show();
          $display("q = %0d", q);
     endtask
endclass
```

In the above example, the members of class "B" are the property, `q`, and the methods send() and show().

Note:
: See “Class Packet Example” on page 1-42 for a more complex example of a class declaration. The name of the class is "Packet."

Note:
: See section 11, "Classes" of the SystemVerilog LRM v3.1a for a formal definition of class.

Creating an Instance (object) of a Class

A class declaration is the template from which objects are created. When a class is constructed the object is built using all the properties and methods from the class declaration.

To create an object (that is, an *instance)* of a declared class, there are two steps. First, declare a handle to the class (a handle is a reference to the class instance, or object):

```
class_name handle_name;
```

Then, call the new() class method:

```
handle_name = new();
```

The following example expands on the above example to show how to create an instance of class "B."

```
program P;
     class  B;
          int q = 3;
          function int send (int a);
               send = a * 2;
          endfunction

          task show();
               $display("q = %0d", q);
          endtask
     endclass

     initial begin
          B b1; //declare handle, b1
          b1 = new;  //create an object by calling new
     end
endprogram
```

The above two steps can be merged into one for instantiating a class at the time of declaration:

```
class_name  handle_name = new();
```

For example:

```
B b1 = new;
```

The new() method of the class is a method which is part of every class. It has a default implementation which simply allocates memory for the object and returns a handle to the object.

Constructors

You can declare your own new() method. In such a case, the prototype that you must follow is:

```
function new([arguments]);
     // body of method
endfunction
```

Use this syntax when using the user-defined constructor:

```
handle_name = new([arguments]);
```

Arguments are optional.

Below is an example of a user-defined constructor:

```
program P;

class B; //declare class
     integer command;
     function new(integer inCommand = 1);
         command = inCommand;
         $display("command = %d", command);
     endfunction
endclass
endprogram
```

When called, new() will print the value passed to it as the value of `command`.

When a class property has been initialized in its declaration, the user-defined new() constructor can be used to override the initialized value. The following example is a variant of the first example in this section.

```
program P;
     class  A;
         int q = 3;  //q is initialized to 3
         function new();
                q = 4; //constructor overrides and assigns 4
         endfunction
     endclass
         A a1;
         initial
          begin
           a1 = new;
           $display("The value of q is %0d.", a1.q);
         end
endprogram
```

The output of this program is:

```
The value of q is 4.
```

If a property is not initialized by the constructor, it is implicitly initialized, just like any other variable, with the default value of its data type (see section 5.2, Table 5-1 in the SystemVerilog LRM v3.1a for details).

Assignment, Re-naming and Copying

Consider the following:

```
class Packet;
     ...
endclass

Packet p1;
```

`Packet p1`; creates a variable, `p1`, that can hold the handle of an object of class `Packet`. The initial default value of `p1` is null. The object does not yet exist, and `p1` does not contain an actual handle, until an instance of type Packet is created as shown below:

```
p1 = new(); //if no arguments are being passed, () can be
                //omitted. e.g., p1=new;
```

Another variable of type `Packet` can be declared and assigned the handle, `p1` to it as shown below:

```
Packet p2;
p2 = p1;
```

In this case, there is still only *one* object. This single object can be referred to with either variable, `p1` or `p2`.

Now consider the following, assuming `p1` and `p2` have been declared and p1 has been instantiated:

```
p2 = new p1;
```

The handle `p2` now points to a *shallow* copy of the object referenced by `p1`. Shallow copying creates a new object consisting of all properties from the source object. Objects are not copied, only their handles. That is, a shallow copy is a duplication of members, including handles, of an object´s first level constituents in the hierarchy. However, an object referenced by a copied handle is not, itself, copied. That is, an object contained within an object is not copied, only the referential handle.

See section 11.11 "Assignment, re-naming and copying," in the SystemVerilog LRM v3.1a for details.

Static Properties

The `static` keyword is used to identify a class property that is shared with all instances of the class. A static property is not unique to a single object (that is, all objects of the same type share the property). A static variable is initialized once.

Syntax

```
static data_type variable;
```

Example

```
program test;

     class Parcel;
          static int count = 2;
          function new();
               count++;
          endfunction
     endclass

     initial begin
          Parcel pk1 = new;
          Parcel pk2 = new;
          Parcel pk3 = new;

          $display("%d Parcel",  pk3.count);
          $display("%d Parcel",  pk2.count);
          $display("%d Parcel",  pk1.count);
     end
endprogram
```

Output

```
     5 Parcel
     5 Parcel
     5 Parcel
```

In the above example, every time an object of type `Parcel` is created, the new() function is invoked and the static variable `count` is incremented. The final value of `count` is "5." If `count` is declared as non-static, the output of the above program is:

```
     3 Parcel
     3 Parcel
     3 Parcel
```

Note:

See section 11.8, "Static class properties," in the SystemVerilog LRM v3.1.a.

Global Constant Class Properties

The `const` keyword is used to designate a class property as read-only. Class objects are dynamic objects, therefore either global or instance properties can be designated "`const`." At present, only global constants are supported.

When declared, a global constant class property includes an initial value. Consider the following example: `//global_const.sv`

```
program test;
     class Xvdata_Class;
          const int pi = 31412;//const variable
     endclass

     Xvdata_Class data;
          initial begin
               data = new();
               data.pi = 42; //illegal and will generate
                             //compile time error
          end
endprogram
```

The line, `data.pi=42`, is not valid and yields the following compile time error message:

```
Error-[IUCV] Invalid use of 'const' variable
      Variable 'pi' declared as 'const' cannot be used
    in this context
    "global_const.sv", 17: data.pi = 42;
```

Class Extensions

Subclasses and Inheritance

SystemVerilog’s OOP implementation provides the capability of inheriting all the properties and methods from a base class, and extending its capabilities within a subclass. This concept is called inheritance. Additional properties and methods can be added to this new subclass.

For example, suppose we want to create a linked list for a class "Packet." One solution would be to extend Packet, creating a new subclass that inherits the members of the parent class (see Class Packet Example on page 42 for class "Packet" declaration).

```
class LinkedPacket extends Packet;
     LinkedPacket next;
     function LinkedPacket get_next();
              get_next = next;
     endfunction
endclass
```

Now, all of the methods and properties of Packet are part of LinkedPacket - as if they were defined in LinkedPacket - and LinkedPacket has additional properties and methods. We can also override the parent’s methods, changing their definitions.

Abstract classes

A group of classes can be created that are all "derived" from a common abstract base class. For example, we might start with a common base class of type BasePacket that sets out the structure of packets but is incomplete; it would never be instantiated. It is "abstract." From this base class, a number of useful subclasses can be derived: Ethernet packets, token ring packets, GPSS packets, satellite packets. Each of these packets might look very similar, all needing the same set of methods, but they would vary significantly in terms of their internal details.We start by creating the base class that sets out the prototype for these subclasses. Since it will not be instantiated, the base class is made abstract by declaring the class as virtual:

```
virtual class BasePacket;
```

By themselves, abstract classes are not tremendously interesting, but, these classes can also have *virtual* methods. Virtual methods provide prototypes for subroutines, all of the information generally found on the first line of a method declaration: the encapsulation criteria, the type and number of arguments, and the return type if it is needed. When a virtual method is overridden, the prototype must be followed exactly:

```
virtual class BasePacket;
    virtual function integer send(bit[31:0] data);
    endfunction
endclass

class EtherPacket extends BasePacket;
    function integer send(bit[31:0] data)
        // body of the function
        ...
    endfunction
endclass
```

`EtherPacket` is now a class that can be instantiated.

If a subclass which is extended from an abstract class is to be instantiated, then all virtual methods defined in the abstract class must have bodies either derived from the abstract class, or provided in the subclass. For example:

```
program P;
virtual class Base;
    virtual task print(string s);
        $display("Base::print() called from %s", s);
    endtask

    extern virtual function string className(); endclass

class Derived extends Base;
    function string className();
        return "Derived";
    endfunction
endclass
Derived d = new;
initial #1 d.print();
endprogram
```

Methods of non-abstract classes can also be declared virtual. In this case, the method must have a body, since the class, and its subclasses, must be able to be instantiated. It should be noted that once a method is declared as "virtual," it is forever virtual. Therefore, the keyword does not need to be used in the redefinition of a virtual method in a subclass.

Polymorphism

The ability to call a variety of functions using the exact same interface is called *polymorphism*.

Suppose we have a class called Packet. Packet has a task called display(). All the derived classes of Packet will also have a display() task, but the base version of display() does not satisfy the needs of the derived classes. In such a case we want the derived class to implement its own version of the display() task to be called.

To achieve polymorphism the base class must be abstract (defined using the `virtual` identifier) and the method(s) of the class must be defined as virtual. The abstract class (see page 28 for definition of abstract class) serves as a template for the construction of derived classes.

```
virtual class Packet;
     extern virtual task display();
endclass
```

The above code snippet defines an abstract class, Packet. Within Packet the virtual task, display(), has been defined. display() serves as a prototype for all classes derived from Packet. Any class which derives from Packet() must implement the display() task. Note that the prototype is enforced for each derived task (that is, must keep the same arguments, etc.).

```
class MyPacket extends Packet
     task display();
          $display("This is the display within MyPacket");
     endtask
endclass

class YourPacket extends Packet
     task display()
          $display("This is the display within YourPacket");
     endtask
endclass
```

This example illustrates how the two classes derived from `Packet` implement their own specific version of the display() method.

All derived classes can be referenced by a base class object. When a derived class is referenced by the base class, the base class can access the virtual methods within the derived class through the handle of the base class.

```
program polymorphism;
     //include class declarations of Packet, MyPacket
      //and YourPacket here

     MyPacket mp;
     YourPacket yp;
     Packet p;  //abstract base class

     initial
     begin
          mp = new;
          yp = new;
          p = mp;  // mp referenced by p
          p.display();// calls display in MyPacket
          p = yp;
          p.display();// calls display in YourPacket
     end
endprogram
```

Output:

```
This is the display within MyPacket
This is the display within YourPacket
```

This is a typical, albeit small, example of polymorphism at work.

See page 123 of the SystemVerilog LRM 3.1a for examples of using a variable in the base class to hold references to subclass objects.

super keyword

The `super` keyword is used from within a derived class to refer to properties of the parent class. It is necessary to use `super` when the property of the derived class has been overridden, and cannot be accessed directly.

```
program sample;
    class Base;
         integer p;
        virtual task display();
            $display("\nBase: p=%0d", p);
        endtask
     endclass

     class Extended extends Base;
        integer q;
        task display();
            super.display(); //refer to Base "display()"
            $display("\nExtended: q=%0d\n", q);
         endtask
    endclass

        Base b1;
        Extended e1;

    initial begin
        b1 = new; // b1 points to instantiated object of Base
        e1 = new; // e1 points to object of Extended
        b1.p = 1; //property "p" of Base initialized to 1
        b1.display(); //will print out "Base: p=1"
        e1.p = 2;  //"p" of Base is now 2
        e1.q = 3;  //"q" of Extended initialized to 3
        e1.display(); //prints "Base: p=2 Extended: q=3"
    end
endprogram
```

Output of the above program is:

```
Base: p=1

Base: p=2

Extended: q=3
```

In the above example, the method, display() defined in `Extended` calls display(), which is defined in the super class, `Base`. This is achieved by using the `super` keyword.

The property may be a member declared a level up or a member inherited by the class one level up. Only one level of super is supported.

When using the `super` keyword within the constructor, `new`, it must be the first statement executed in the constructor. This is to ensure the initialization of the superclass prior to the initialization of the subclass.

Casting

$cast enables you to assign values to variables that might not ordinarily be valid because of differing data type. For full details see section 3.15 of the SystemVerilog 3.1a LRM.

The following example involves casting a base class handle ("b2"), which actually holds an extended class-handle ("e1"), at runtime back into an extended class-handle ("e2").

```
program sample;
class Base;
     integer p;
     virtual task display();
          $write("\nBase: p=%0d\n", p);
     endtask
endclass

class Extended extends Base;
     integer q;
     virtual task display();
          super.display();
          $write("Extended: q=%0d\n", q);
     endtask
endclass

Base b1 = new(), b2;
Extended e1 = new(), e2;
```

```
initial begin
     b1.p = 1;
     $write("Shows the original base property p\n");
     b1.display(); // Just shows base property
     e1.p = 2;
     e1.q = 3;
     $write("Shows the new value of the base property, p,
      and value of the extended property, q\n");

     e1.display(); // Shows base and extended properties
// Have the base handle b2 point to the extended object
                b2 = e1;
     $write("The base handle b2 is pointing to the Extended
     object\n");
     b2.display();  // Calls Extended.display

// Try to assign extended object in b2 to extended handle e2
     $write("Using $cast to assign b2 to e2\n");
          if ($cast(e2, b2))
               e2.display();  // Calls Extended.display
          else
               $write("cast_assign of b2 to e2 failed\n");
          end
endprogram
```

Output of the above program is:

```
Shows the base property p, which is originally 1
Base: p=1

Shows new value of the base property, p, and value of the
extended property, q
Base: p=2
Extended: q=3

The base handle b2 is pointing to the Extended object
Base: p=2
Extended: q=3

Using $cast to assign b2 to e2
```

```
Base: p=2
Extended: q=3
```

In the current implementation, there is a restriction related to the casting of enumerated types. $cast() into an enum-type destination issues a NYI error in case the type of the source does not match that of the destination.

```
$cast(enum1, int1)  // NYI, destination = enum, source = int
```

The following case is supported as long as enum1 and enum2 are of same base enum-type:

```
$cast(enum1, enum2)
```

Enumerated types can, however, be cast into any other non-enum types (that is, they can be used as the source argument in $cast() without any restrictions:

```
$cast(int1, enum1)  // supported, destination non-enum (int)
```

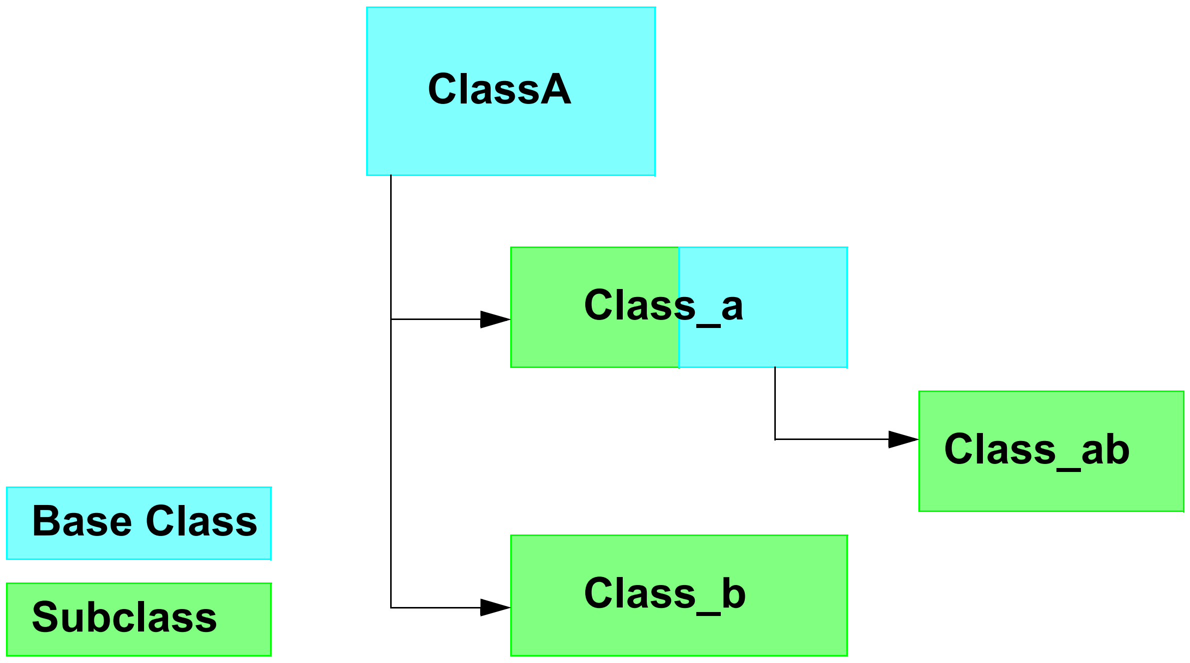
Chaining Constructors

To understand the relational use of the terms "base" and "super" as employed to refer to classes, consider the following code fragment:

```
class ClassA;
     ...
endclass

class Class_a extends ClassA;
     ...
endclass

class class_ab extends Class_a;
     ...
endclass

class Class_b extends ClassA;
     ...
endclass
```

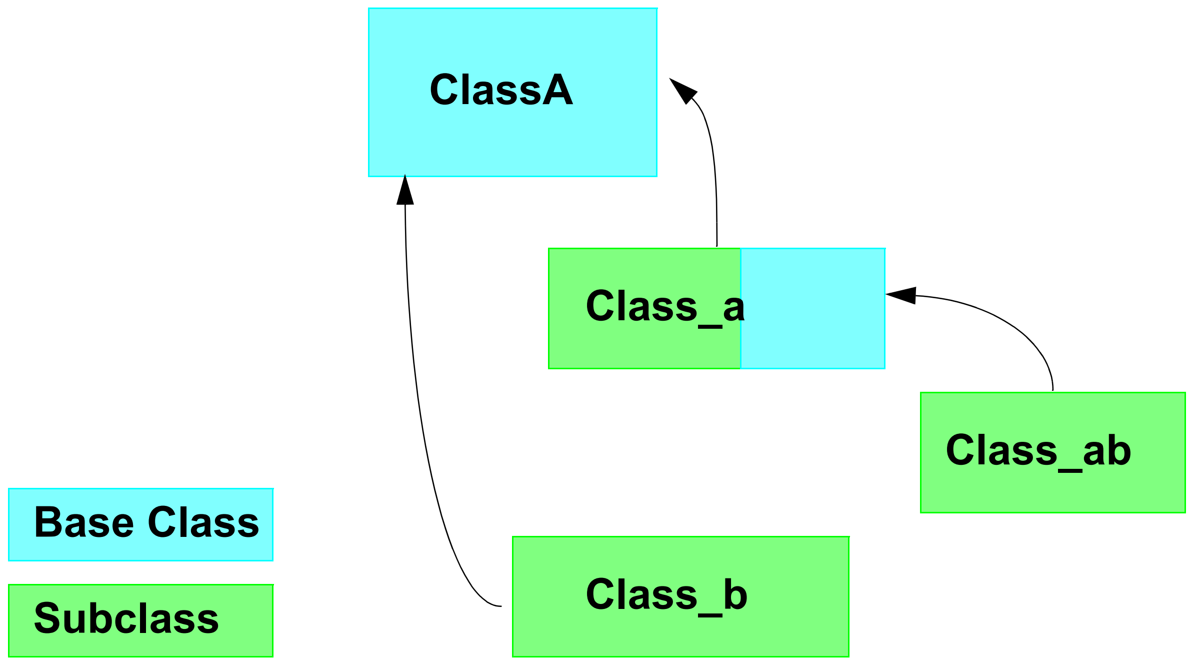
Both Class_a and Class_b are extended from ClassA using the `extends` keyword, making ClassA the base class for these two subclasses. Class_ab extends from Class_a, making Class_a the base class for the subclass, Class_ab. Both ClassA and Class_a are super classes since they each have subclasses extended from them.

Figure 1-1   Base and Sub Classes



When a subclass is instantiated, the constructor of the extended super class is implicitly called. If that class, in turn, has a super class, it will implicitly call the constructor of that class. This will continue up the hierarchy until no further super classes exist. In Figure 1-2,

Figure 1-2   Chaining Constructors



when Class_ab is instantiated, the “new” constructor for Class_a is called. In turn, the “new” constructor for ClassA is called. Since ClassA is the last super class in the chain, an object of Class_ab is then created. When Class_b is instantiated, the “new” constructor for ClassA is called and an object of Class_b is created.

This process is called “chaining constructors.”

If the initialization method of the super-class requires arguments, you have two choices. If you want to always supply the same arguments, you can specify them at the time you extend the class:

```
class EtherPacket extends Packet(5);
```

This will pass 5 to the new() routine associated with “Packet”.

A more general approach is to use the `super` keyword to call the superclass constructor.

```
function new();
     super.new(5);
endfunction
```

If included, the call to super() must be the first executable statement (that is, not a declaration) in the constructor.

Accessing Class Members

Properties

A property declaration may be preceded by one of these keywords:

- local
- public
- protected

“public” is the default protection level for class members Using public when declaring a property allows global access to that member via *`class_name.member`*.

In contrast, a member designated as `local` is one that is only visible from within the class itself.

A `protected` class property (or method) has all of the characteristics of a local member, except that it can be inherited; it is visible to subclasses.

Accessing Properties

A property of an object can be accessed using the dot operator (.). The handle name for the object precedes the dot, followed by the qualifying property name (e.g., address, command).

```
instance_name.property_name
```

Methods

Tasks or functions, known as “methods,” can be designated as `local`, `public`, or `protected`. They are public by default.

Accessing Object Methods

An object´s methods can be accessed using the dot operator (.). The handle for the object precedes the dot, followed by the method.

```
program access_object_method;
     class  B;
          int q = 3;
          function int send (int a);
               send = a * 2;
          endfunction

          task show();
               $display("q = %0d", q);
          endtask
     endclass

     initial begin
          B b1;       //declare handle
          b1 = new;  // instantiate
          b1.show(); //access show() of object b.1
          $display("value returned by b1.send()  = %0d",
               b1.send(4));//access send()of object b.1
     end
endprogram
```

Output of program

```
q = 3
value returned by b1.send() = 8
```

“this” keyword

The `this` keyword is used to unambiguously refer to properties or methods of the current instance. For example, the following declaration is a common way to write an initialization task:

```
program P;
     class Demo;
          integer x;
          task new (integer x);
               this.x = x;
          endtask
     endclass
endprogram
```

The `x` is now both a property of the class and an argument to the task new(). In the task new(), an unqualified reference to `x` will be resolved by looking at the innermost scope, in this case the subroutine argument declaration. To access the instance property, we qualify it with `this` to refer to the current instance.

The following is another, complete, example.

```
program P;
     class Demo;
          integer x=4;
          function integer send(integer x);
               this.x = x*3;
          endfunction
          task show();
               $display("The value of x in the object of type
                         Demo is = %d", send(this.x));
          endtask
     endclass
     intial begin
          integer x=5;
          Demo D =new;

          D.show();
          $display("The value of the global variable x is
                    %0d", x);
     end
endprogram
```

The output of this program is:

```
The value of x in the object of type Demo is = 12
The value of the global variable x is =5
```

Class Packet Example

```
class Packet;
     bit [3:0] command;  //property declarations
     bit [40:0] address;
     bit [4:0] master_id;
     integer time_requested;
     integer time_issued;
     integer status;

     function new(); // initialization
```

```
            command = 0;
            address = 41’b0;
            master_id = 5’bx;
        endfunction

        task clean(); //method declaration
            command = 0;
            address = 0;
            master_id = 5’bx;
        endtask

        task issue_request(int delay); //method declaration
            // send request to bus
        endtask

        function integer current_status(); //method declaration
            current_status = status;
        endfunction
    endclass
```

Random Constraints

SystemVerilog has constructs for the random testing of a design and a means of constraining the random testing to find hard to reach corner cases.

The official description of random constraints begins on page 128 of the SystemVerilog 3.1a LRM.

Random Variables

You need variables to which VCS assigns random values. There are two types of random variables and two different keywords to begin their declarations:

`rand`
: Specifies a standard random variable

`randc`
: Specifies a random-cyclic variable.

The following is an example of a standard random variable declaration:

```
rand bit [7:0] randbit1;
```

Variable `randbit1` has eight bits so its possible values range from 0 to 255. The chances that VCS assigns any of these values is precisely 1/256.

The following is an example of a random-cyclic variable declaration:

```
randc bit [1:0] randc1;
```

Variable `randc1` has two bits so its possible values range from 3 to 0. VCS derives a random order, or permutation, of all the possible values, and assigns the values in the permutation in a sequence.

If VCS assigns a value of 3, it won't assign 3 again until it first assigns 2, 1, and 0 (in no particular order), and after it assigns the permutation, it derives the next permutation, beginning with any of the possible values.

If this were a standard random variable, after VCS assigns the 3 value, there is a 1/4 chance that VCS assigns the 3 again, but because it is a random-cyclic variable, after VCS assigns the 3, there is no chance that 3 will also be the next assigned value.

Random variables can only be declared in a class.

```
class Bus;
   rand bit [15:0] addr;
   rand bit [15:0] data;
   ⋮
endclass
```

The official description of random variables is on pages 131-132 of the SystemVerilog 3.1a LRM.

Constraint Blocks

Constraints are specified in a constraint block. A constraint block can be declared in the same class in which the random variables are declared as well as in a class extended from the base class in which the random variable is defined (see page 36 for definition of "base class") A constraint block begins with the keyword `constraint`.

```
program prog;

class myclass;
   rand logic [1:0] log1,log2,log3;
   randc logic [1:0] log4;
   constraint c1 {log1 > 0;}
   constraint c2 {log2 < 3;}
endclass

myclass mc = new;

initial
repeat (10)
   if(mc.randomize()==1)
      $display("log1 = %0d log2=%0d log3=%0d log4=%0d",
                mc.log1, mc.log2, mc.log3, mc.log4);

endprogram
```

In this program block class myclass contains the following:

- Declarations for standard random variables, with the `logic` data type and two bits, named log1, log2, and log3.
- A declaration for cyclic-random variable, with the `logic` data type and two bits, named log4.
- Constraint c1 which says that the random values of log1 must be greater than 0.
- Constraint c2 which says that the random values of log2 must be less than 3.

The `randomize()` method is described in "Randomize Methods" on page 1-57 in this chapter.

The `$display` system task displays the following:

```
log1 = 1 log2=1 log3=1 log4=2
log1 = 1 log2=2 log3=0 log4=0
log1 = 2 log2=0 log3=1 log4=1
log1 = 1 log2=1 log3=1 log4=3
log1 = 3 log2=2 log3=1 log4=0
log1 = 3 log2=2 log3=3 log4=3
log1 = 3 log2=1 log3=0 log4=2
log1 = 1 log2=1 log3=1 log4=1
log1 = 2 log2=0 log3=2 log4=3
log1 = 1 log2=0 log3=2 log4=0
```

The possible values of all the random variables range from 3 to 0, but the values of log1 are never 0, and the values of log2 are never greater than 3. VCS can assign the same value to log3 over again, but never assigns the same value to log4 over again until it has cycled through all the other legal values.

By means of inline constraints, a constraint block can be declared for random variables declared in a class while calling the randomize() method on that class' object. (See section entitled "In-line Constraints" on page 66).

```
program test;
     class base;
       rand int r_a;
     endclass
     base b = new;

     initial begin
          int ret;
          ret = b.randomize with {r_a > 0; r_a <= 10;} ;
     // where my declaration {r_a >0 ; r_a <= 10; }
     // is now a constraint block
     if(ret == 1)
          $display("Randomization success");
     else
          $display("Randomization Failed");
     end
endprogram
```

The official description of constraint blocks begins on page 132 of the SystemVerilog 3.1a LRM.

Inheritance

Constraints follow the same rules of inheritance as class variables, tasks, and functions.

```
 class myclass;
   rand logic [1:0] log1,log2,log3;
   randc logic [1:0] log4;
   constraint c1 {log1 > 0;}
   constraint c2 {log2 < 3;}
endclass

class myclass2 extends myclass;
   constraint c2 {log2 < 2;}
endclass

myclass2 mc = new;
```

The keyword `extends` specifies that class myclass2 inherits from class myclass and then can change constraint c2, specifying that it must be less than 2, instead of less than 3.

The official description of inheritance is on page 134 of the SystemVerilog 3.1a LRM.

Set Membership

You can use the `inside` operator to specify a set of possible values that VCS randomly applies to the random variable.

```
program prog;

class myclass;
 rand int int1;
 constraint c1 {int1 inside {1, [5:7], [105:107]};}
endclass

myclass mc = new;

initial
repeat (10)
   if(mc.randomize()==1)
      $display("int1=%0d",mc.int1);

endprogram
```

Constraint c1 specifies that the possible values for int1 are as follows:

- 1
- a range beginning with 5 and ending with 7
- a range beginning with 105 and ending with 107

The `$display` system task displays the following:

```
int1=1
int1=7
int1=5
int1=107
int1=106
int1=5
int1=5
int1=6
int1=107
int1=1
```

All these values are equally probable.

The official description of set membership is on page 134 of the SystemVerilog 3.1a LRM.

Weighted Distribution

You can use the `dist` operator to specify a set of values or ranges of values, where you give each value or range of values a weight, and the higher the weight, the more likely VCS is to assign that value to the random variable.

If we modify the previous example as follows:

```
program prog;

class myclass;
   rand int int1;
  constraint c1 {int1 dist {[1:2] := 1, [6:7] :/ 5, 9 :=10};}
endclass

myclass mc = new;

int stats [1:9];

int i;

initial begin
   for(i = 1; i < 10; i++) stats[i] = 0;
   repeat (1700)
     if(mc.randomize()==1) begin
        stats[mc.int1]++;
        //$display("int1=%0d",mc.int1);
     end
   for(i = 1; i < 10; i++)
       $display("stats[%d] = %d", i, stats[i]);
   end

endprogram
```

constraint c1 in class myclass is now:

```
constraint c1 {int1 dist {[1:2] := 1, [6:7] :/ 5, 9 :=10};}
```

Constraint C1 now specifies the possible values of int1 are as follows:

- A range of 1 to 2, each value in this range has a weight of 1. The `:=` operator, when applied to a range, specifies applying the weight to each value in the range.
- A range of 6 to 7 with a weight of 5. The `:/` operator specifies applying the weight to the range as a whole.
- 9 with a weight of 10, which is twice as much as the weight for 6.

The repeat loop repeats 1700 times so that we have a large enough sample. The following table shows the various values of int1 during simulation:

value of int	**number of times int has this value**	**fraction of the simulation time**
1	106	1/17
2	98	1/17
3	0	
4	0	
5	0	
6	248	2.5/17
7	251	2.5/17
8	0	
9	998	10/17

The official description of weighted distribution is on pages 133-136 of the SystemVerilog 3.1a LRM.

Implications

An implication is an expression that must be true before VCS applies the constraint.

```
program prog;

class Bus;
   randc bit randcvar;
   bit norandvar;

   constraint c1 { (norandvar == 1) -> (randcvar == 0);}
endclass

Bus bus = new;

initial
begin
   bus.norandvar = 0;
#5 bus.norandvar = 1;
#5 bus.norandvar = 0;
#5 $finish;
end

initial
repeat (15)
  #1 if (bus.randomize() ==1)
      $display("randcvar = %0b norandvar = %0b at %0t",
                bus.randcvar, bus.norandvar,$time);

endprogram
```

In constraint c1, when variable norandvar has a 1 value (the implication expression), VCS constrains random variable randcvar to 0. The `->` token is the implication operator.

The `$display` system task displays the following:

```
randcvar = 0 norandvar = 0 at 1
randcvar = 1 norandvar = 0 at 2
randcvar = 0 norandvar = 0 at 3
randcvar = 1 norandvar = 0 at 4
randcvar = 0 norandvar = 1 at 5
randcvar = 0 norandvar = 1 at 6
randcvar = 0 norandvar = 1 at 7
randcvar = 0 norandvar = 1 at 8
randcvar = 0 norandvar = 1 at 9
randcvar = 1 norandvar = 0 at 10
randcvar = 1 norandvar = 0 at 11
randcvar = 0 norandvar = 0 at 12
randcvar = 0 norandvar = 0 at 13
randcvar = 1 norandvar = 0 at 14
```

When variable norandvar is 0, random-cyclic variable randcvar is either 0 or 1. When variable norandvar is 1, random-cyclic variable randcvar is always 0.

The official description of implications is on page 136 of the SystemVerilog 3.1a LRM.

if else Constraints

An alternative to an implication constraint is the `if else` constraint. The `if else` constraint allows a constraint or constraint set on the `else` condition.

```
program prog;

class Bus;
   randc bit [2:0] randcvar;
   bit norandvar;

   constraint c1 { if (norandvar == 1)
                     randcvar == 0;
                   else
                     randcvar inside {[1:3]};}
endclass
```

```
Bus bus = new;

initial
begin
   bus.norandvar = 0;
#5 bus.norandvar = 1;
#5 bus.norandvar = 0;
#5 $finish;
end

initial
repeat (15)
  #1 if (bus.randomize() ==1)
      $display("randcvar = %0d norandvar = %0b at %0t",
bus.randcvar, bus.norandvar,$time);

endprogram

   constraint c1 { if (norandvar == 1)
                     randcvar == 0;
                   else
                     randcvar inside {[1:3]};}
```

Constraint c1 specifies that when variable norandvar has the 1 value, VCS constrains random-cyclic variable randcvar to 0, and when norandvar doesn't have the 1 value, in other words when it has the 0 value, VCS constrains random-cyclic variable randcvar to 1, 2, or 3.

The `$display` system task displays the following:

```
randcvar = 1 norandvar = 0 at 1
randcvar = 2 norandvar = 0 at 2
randcvar = 3 norandvar = 0 at 3
randcvar = 3 norandvar = 0 at 4
randcvar = 0 norandvar = 1 at 5
randcvar = 0 norandvar = 1 at 6
randcvar = 0 norandvar = 1 at 7
randcvar = 0 norandvar = 1 at 8
randcvar = 0 norandvar = 1 at 9
randcvar = 1 norandvar = 0 at 10
randcvar = 2 norandvar = 0 at 11
randcvar = 3 norandvar = 0 at 12
randcvar = 3 norandvar = 0 at 13
randcvar = 2 norandvar = 0 at 14
```

When norandvar is 1, VCS always assigns 0 to randcvar.

The official description of if else constraints is on pages 136-137 of the SystemVerilog 3.1a LRM.

Global Constraints

Global constraints are described in detail on page 139 of the SystemVerilog 3.1a LRM.

Constraint expressions involving random variables from other objects are called global constraints.

```
program prog;

class myclass;
 rand bit [7:0] randvar;
endclass

class myextclass extends myclass;
 rand myclass left = new;
 rand myclass right = new;
 constraint c1 {left.randvar <= randvar; right.randvar <=
randvar;}
endclass
endprogram
```

In this example, class myextclass extends class myclass. In class myextclass are two randomized objects (or instances) of class myclass. They are named left and right and are randomized because they are declared with the `rand` keyword.

Constraint C1 specifies that the version of randvar in the left object must be less than or equal to randvar in myclass. Similarly the version of randvar in the right object must be less than or equal to randvar in myclass.

Constraint c1 is a global constraint because it refers to a variable in myclass.

The official description of global constraints is on page 139 of the SystemVerilog 3.1a LRM.

Variable Ordering

In an implication like the following:

```
constraint c1 { select -> data == 0;}
```

The select variable “implies” that data is 0, in this case, when select is 1, data is 0.

The default behavior is for the constraint solver to solve for both select and data at the same time. Sometimes you can assist the constraint solver by telling it to solve for one variable first. You can do this with another constraint block that specifies the order in which the constraint solver solves the variables:

```
constraint c1 { select -> data == 0;}
constraint c2 ( solve select before data;}
```

Note:
 Ordering also changes the distribution of values.

The official description of variable ordering is on pages 139-140 of the SystemVerilog 3.1a LRM.

Static Constraint Blocks

The keyword `static` preceding the keyword constraint, makes a constraint block static, and calls to the `constraint_mode()` method occur to all instances of the constraint in all instances of the constraint's class.

```
static constraint c1 { data1 < data2;}
```

The official description of static constrain blocks is on page 141 of the SystemVerilog 3.1a LRM.

Randomize Methods

randomize()

Variables in an object are randomized using the randomize() class method. Every class has a built-in randomize() method.

`randomize()`
: Generates random values for active random variables, subject to active constraints, in a specified instance of a class. This method returns a 1 if VCS generates these random values, otherwise it returns 0. Examples of the use of this method appear frequently in previous code examples.

The official description of random constraint methods begins on page 145 of the SystemVerilog 3.1a LRM.

pre_randomize() and post_randomize()

Every class contains built-in pre_randomize() and post_randomize() tasks, that are automatically called by randomize() before and after it computes new random values (See section 12.5.2 of the SystemVerilog 3.1a LRM for syntax and details.)

You may override the pre_randomize() method in any class to perform initialization and set pre-conditions before the object is randomized. Also, you may override the post_randomize() method in any class to perform cleanup, print diagnostics, and check post-conditions after the object is randomized.

The following example involves a case where the pre_randomize() method is used to instantiate and initialize the rand dynamic array, `payload[]`, before randomize() is called. The post_randomize() method is used to display the size of the payload[] after randomize() is called. `program test;`

```
class size;
     rand bit [7:0] s; //random variable s
     constraint con_size {
          s > 0;
          s < 50;
     }
endclass

class Pkt;
     integer header[7];
     rand bit [7:0] payload[];
     size sz;
     function void pre_randomize();
     /* Randomize the sz.s variable and instantiate
     the dynamic array. */
          integer res;
          sz = new();
          res = sz.randomize();  /* calling randomize() on
               object sz of class size. Randomizes rand
               variable s (constrained to >0 and <50 */
          if(res==0)
             $display("Randomization Failed");
          else
               begin
               if((sz.s>0) &&  (sz.s<50))/* check for size
                                 between 0 and 50*/
               payload = new[sz.s];/* the new[] operator
                        allocates storage and initializes the
                        rand variable s in sz*/
               else
                 $display("Failed to generate proper size");
          end     //end of outer else block
     endfunction

function void post_randomize();
/* display size of payload dynamic array using the size()
built-in method*/
     $display("payload.size = %d", payload.size);

endfunction
endclass

Pkt pkt;
integer success,i;

initial begin
     pkt = new(); //instantiate pkt

     for (int j = 0; j < 5; j++)
     begin
          success = pkt.randomize(); /*calling randomize
               on object of class packet*/
          if(success==0)
               $display("Randomization Failed");
          else
               $display("Randomization Succeeded");
     end // end of for loop
end  // end of initial block
endprogram
```

The output of the program is:

```
payload.size =         12
Randomization Succeeded
payload.size =         17
Randomization Succeeded
payload.size =          8
Randomization Succeeded
payload.size =         40
Randomization Succeeded
payload.size =         10
Randomization Succeeded
```

Controlling Constraints

The predefined constraint_mode() method can be used either as a task or a function. The constraint_mode() task controls whether a constraint is active or inactive. The predefined constraint_mode() function reports the current ON/OFF value for the specified variable. All constraints are initially active.

Syntax:

```
task object[.constraint_identifier]::constraint_mode
    (bit ON | OFF);
```

or

```
function int
    object.constraint_identifier::constraint_mode();
```

For detailed discussion, see section 12.8 of the SystemVerilog 3.1a LRM.

In the following example, there are two constraint blocks defined in a class, bus. The constraint_mode() method, when used as a task, turns on and off the word_align and addr_range constraints. constraint_mode() is also being used as a function to report the ON/ OFF value of the two constraints.

```
‘define N 5
program test;
class bus;
     rand bit [15:0] addr;
     constraint word_align {addr[0] == 1'b0;}
     constraint addr_range{addr >= 0 && addr <= 15;}
endclass

task generateRandomAddresses(integer how_many);
     integer i;
     for(i = 1; i <= how_many; i++) begin
          bb.randomize();
          $display("bb.addr = %d",bb.addr);
      end
endtask

bus bb = new;
initial begin

// By default all constraints are ON
     if (bb.word_align.constraint_mode() &&
                         bb.addr_range.constraint_mode())
          begin
           $display("========both constraints ON ======");
           end
         else
            $display("Error with constraint_mode");

     generateRandomAddresses(`N);

     // turn OFF "word_align" constraint in "bb"
     bb.word_align.constraint_mode(0);
     $display("=========one constraint ON =======");
     generateRandomAddresses(`N);
     // turn OFF all constraints in "bb"
     bb.constraint_mode(0);
     $display("=========both constraints OFF =======");
     generateRandomAddresses(`N);

     // turn ON "addr_range" constraint in "bb"
     bb.addr_range.constraint_mode(1);
     $display("=========one constraint ON =======");
     generateRandomAddresses(`N);

     // turn ON all constraint in "bb"
     bb.constraint_mode(1);
     $display("=========both constraints ON =======");
     generateRandomAddresses(`N);
        end
endprogram
```

Output of the above program:

```
=========both constraints ON =======
bb.addr =    14
bb.addr =     2
bb.addr =     4
bb.addr =     8
bb.addr =     2
=========one constraint ON =======
bb.addr =     2
bb.addr =     9
bb.addr =     9
bb.addr =     9
bb.addr =     3
=========both constraints OFF =======
bb.addr = 44051
bb.addr = 36593
bb.addr = 19491
bb.addr =  6853
bb.addr = 48017
=========one constraint ON =======
bb.addr =    11
bb.addr =     8
bb.addr =    15
bb.addr =     7
bb.addr =     4
=========both constraints ON =======
bb.addr =     2
bb.addr =    10
bb.addr =    12
bb.addr =    10
bb.addr =     8
```

Disabling Random Variables

SystemVerilog provides the predefined rand_mode() method to control whether a random variable is active or inactive. Initially, all random variables are active.

The rand_mode() method can be used either as a task or a function.

The rand_mode() task specifies whether or not the random variable is ON or OFF.

```
task object[.randvar_identifier]::rand_mode(bit ON | OFF);
```

where *ON* is "1" and *OFF* is "0."

As a function, rand_mode() reports the current value (ON or OFF) of the specified variable.

Syntax:

```
function int object.randvar_identifier::rand_mode();
```

rand_mode() returns -1 if the specified variable does not exist within the class hierarchy or it exists but is not declared as rand or randc.

For detailed discussion, see section 12.7 of the SystemVerilog 3.1a LRM.

Example:

In the following example, there are two random variables defined in a class, `bus`. The rand_mode() method, when used as a task, turns on and off the random variables, `addr` and `data`. rand_mode() is also being used as a function to report on the value (ON or OFF) of the two random variables.

```
`define N 5
program test;
    class bus;
        rand bit [15:0] addr;
        rand bit [31:0] data;
      constraint CC { data > 0; addr > 0; addr < 255; data
        < 512;}
    endclass

    task generateRandomAddresses(integer how_many);
        integer i;
            for(i = 1; i <= how_many; i++) begin
                bb.randomize();
                $display("bb.addr = %d,, bb.data =
                    %d",bb.addr,bb.data);
            end
    endtask

    bus bb = new;

    initial begin
        // By default all random variables are ON
        if (bb.addr.rand_mode() && bb.data.rand_mode())
            begin
                $display("======both random variables ON
                ======");
            end
        else
            $display("Error with rand_mode");
        generateRandomAddresses(`N);

        // turn OFF "data" random variable in "bb"
        bb.data.rand_mode(0);
        $display("======one random variable ON ======");
        generateRandomAddresses(`N);

         // turn OFF all random variables in "bb"
         bb.rand_mode(0);
        $display("======both random variables OFF======");
        generateRandomAddresses(`N);

        // turn ON "data" constraint in "bb"
        bb.data.rand_mode(1);
        $display("======one random variable ON========");
        generateRandomAddresses(`N);

        // turn ON all random variable in "bb"
        bb.rand_mode(1);
        $display("======both random variables ON ======");
        generateRandomAddresses(`N);
    end
endprogram
```

Output of the above program:

```
======both random variables ON =======
bb.addr =    88,, bb.data =        12
bb.addr =    49,, bb.data =       381
bb.addr =   185,, bb.data =         5
bb.addr =   212,, bb.data =       486
bb.addr =    49,, bb.data =       219
=======one random variable ON ========
bb.addr =     3,, bb.data =       219
bb.addr =   130,, bb.data =       219
bb.addr =    26,, bb.data =       219
bb.addr =    90,, bb.data =       219
bb.addr =   121,, bb.data =       219
```

```
======both random variables OFF=======
bb.addr =   121,, bb.data =        219
bb.addr =   121,, bb.data =        219
bb.addr =   121,, bb.data =        219
bb.addr =   121,, bb.data =        219
bb.addr =   121,, bb.data =        219
========one random variable ON========
bb.addr =   121,, bb.data =        456
bb.addr =   121,, bb.data =        511
bb.addr =   121,, bb.data =         67
bb.addr =   121,, bb.data =        316
bb.addr =   121,, bb.data =        405
======both random variables ON =======
bb.addr =    18,, bb.data =        491
bb.addr =   231,, bb.data =        113
bb.addr =   118,, bb.data =        230
bb.addr =    46,, bb.data =         96
bb.addr =   155,, bb.data =        298
```

In-line Constraints

You can use the `randomize()` method and the `with` construct to declare in-line constraints outside of the class for the random variables, at the point where you call the `randomize()` method.`program prog;`

```
class Bus;
   rand bit [2:0]  bitrand1;
endclass

Bus bus = new;

task inline (Bus bus);
int int1;
int1 = bus.randomize() with {bitrand1[1:0] == 2'b00;};
repeat (10)
   if (bus.randomize() ==1)
      $display("bitrand1 = %0b", bus.bitrand1);
endtask
```

```
initial
inline(bus);

endprogram
```

The `$display` system task displays the following

```
bitrand1 = 100
bitrand1 = 100
bitrand1 = 0
bitrand1 = 100
bitrand1 = 100
bitrand1 = 0
bitrand1 = 100
bitrand1 = 100
bitrand1 = 0
bitrand1 = 100
```

The official description of in-lined random constraints begins on page 147 of the SystemVerilog 3.1a LRM.

Seeding for Randomization

The srandom() method initializes the current Random Number Generator (RNG) of objects or threads using the value of the seed.

For the method prototype, see section 12.12.3 of the SystemVerilog 3.1a LRM.

The following example involves using srandom() to initialize the RNG for an object using a real variable as the seed.

```
program test;

class A;
        rand logic [7:0] x;
```

```
endclass

real r = 1;

A a;
int d1,d2,d3,d4;
initial begin
     a = new;
     a.srandom(r);//the r is the seed for RNG of a
     d1 = a.randomize();
     if(d1 == 1) //if randomize() is successful
     d2 = a.x;  //assign value of the variable x in a to d2
     a.srandom(r+1);
     d1 = a.randomize();
     if(d1 == 1)
     d3 = a.x;
     a.srandom(r);
     d1 = a.randomize();
     if(d1 == 1)
          d4 = a.x;
          if((d2 == d4) && (d2 != d3))
               $display("test passed");
     else
          $display("test failed");
end
endprogram
```

Output of the above program is:

```
test passed
```

Interprocess Synchronization and Communication

Semaphores

SystemVerilog semaphores are not signal devices. They are buckets that contain keys, where competing resources, such as different initial blocks in a program, require keys from the bucket to continue processing.

```
program prog;

semaphore sem1 = new(2);

initial
begin:initial1
   #1 sem1.get(1);
      $display("initial1 takes 1 key at %0t", $time);
   #6 sem1.put(1);
      $display("initial1 returns 1 key at %0t",$time);
   #1 sem1.get(1);
      $display("initial1 takes 1 key at %0t", $time);

end

initial
begin:initial2
   #5 sem1.get(2);
    $display("          inital2 takes 2 keys at %0t",$time);
   #5 sem1.put(1);
    $display("          inital2 returns 1 key at %0t",$time);

end
endprogram
```

In this program there are two initial blocks, labeled by the label on their begin-end blocks, initial1 and intital2.

The program has a semaphore named sem1 that starts with two keys, as specified with the `semaphore` keyword and `new()` method.

If it were not for initial2, initial1 would do the following:

1. Take a key at simulation time 1 (using the get `method`).
2. Return a key at time 7 (using the `put` method).
3. Take a key again at time 8 (using the get `method`).

If it were not for initial1, initial2 would do the following:

1. Take two keys at simulation time 5 (using the get `method`).
2. Return one key at time 10 (using the `put` method).

However both initial blocks contend for a limited number of keys that they need in order to finish executing, in taking keys that the other needs, they interrupt each other's processing. The `$display` system tasks display the following:

```
initial1 takes 1 key at 1
initial1 returns 1 key at 7
                 inital2 takes 2 keys at 7
                 inital2 returns 1 key at 12
initial1 takes 1 key at 12
```

The initial block initial2 could be rewritten to use the `try_get` method to see if a certain number of keys are available, for example:

```
initial
begin:initial2
   #5 if(sem1.try_get(2))
       begin
         sem1.get(2);
         $display("inital2 takes 2 keys at %0t",$time);
       end
   #5 sem1.put(1);
```

```
      $display("inital2 returns 1 key at %0t",$time);

end
endprogram
```

In the revised initial2, at simulation time 5, the `try_get` method checks to see if there are two keys in sem1. There aren't, because initial1 took one. At time 10 the put method "returns" a key to sem1. Actually the `get` and `put` methods only decrement and increment a counter for the keys, there are no keys themselves, so initial2 can increment the key count without having previously decrementing this count.

The `$display` system tasks display the following:

```
initial1 takes 1 key at 1
initial1 returns 1 key at 7
initial1 takes 1 key at 8
inital2 returns 1 key at 10
```

The official description of semaphores begins on page 166 of the SystemVerilog 3.1a LRM.

Semaphore Methods

Semaphores have the following built-in methods:

`new (`*`number_of_keys`*`)`
: You use this method with the `semaphore` keyword. It specifies the initial number of keys in the semaphore.

`put(`*`number_of_keys`*`)`
: Increments the number of keys in the semaphore.

`get(`*`number_of_keys`*`)`

Decrements the number of keys in the semaphore. If there aren't the specified number of keys in the semaphore, VCS halts simulation of the process (initial block, task, etc.) until there the `put` method in another process increments the number of keys to the sufficient number.

`try_get (`*`number_of_keys`*`)`

Decrements the number of keys in the semaphore. If there aren't the specified number of keys in the semaphore, this method returns a 0. If the semaphore has the specified number of keys, this method returns 1. After returning the value, VCS executes the next statement.

The official description of semaphore methods begins on page 166 of the SystemVerilog 3.1a LRM.

Mailboxes

Mailboxes are FIFO containers for messages that are expressions. The SystemVerilog 3.1a LRM specifies that you can specify a maximum number of messages that a mailbox can hold, but this feature isn't implemented yet.

```
program prog;

mailbox mbx = new ();
int i,j;
int k = 10;

initial
begin
repeat(3)
  begin
   #5 mbx.put(k);
      i = mbx.num();
      $display("No. of msgs in mbx = %0d at %0t",i,$time);
      k = k + 1;
  end
i = mbx.num();
repeat (3)
  begin
   #5 $display("No. of msgs in mbx = %0d j = %0d at %0t",i,j,$time);
      mbx.get(j);
      i = mbx.num();
  end
end
endprogram
```

This program declares a mailbox named mbx with the `mailbox` keyword and the `new()` method.

The initial block does the following:

1. Executes a `repeat` loop three times which does the following:
   a. Puts the value of k in the mailbox.
   b. Assigns the number of messages in the mailbox to i.
   c. Increments the value of k.
2. Execute another `repeat` loop three times that does the following:
   a. Displays the number of messages in the mailbox, the value of j, and the simulation time.
   b. Assigns the first expression in the mailbox to j.
   c. Assigns the number of messages to i.

The `$display` system tasks display the following:

```
No. of msgs in mbx = 1 at 5
No. of msgs in mbx = 2 at 10
No. of msgs in mbx = 3 at 15
No. of msgs in mbx = 3 j = 0 at 20
No. of msgs in mbx = 2 j = 10 at 25
No. of msgs in mbx = 1 j = 11 at 30
```

The official description of mailboxes begins on page 167 of the SystemVerilog 3.1a LRM.

Mailbox Methods

Mailboxes use the following methods:

`new()`
: Along with the `mailbox` keyword, declares a new mailbox. You cannot yet specify the maximum number of messages with this method.

`num()`
: Returns the number of messages in the mailbox.

`put(`*`expression`*`)`
: Puts another message in the mailbox.

`get(`*`variable`*`)`
: Assigns the value of the first message to the variable. VCS removes the first message so that the next message becomes the first method. If the mailbox is empty, VCS suspends simulation of the process (initial block, task, etc.) until a `put` method put a message in the mailbox.

`try_get(`*`variable`*`)`
: Assigns the value of the first message to the variable. If the mailbox is empty, this method returns the 0 value. If the message is available, this method returns a non-zero value. After returning the value, VCS executes the next statement.

`peek(`*`variable`*`)`

Assigns the value of the first message to the variable without removing the message. If the mailbox is empty, VCS suspends simulation of the process (initial block, task, etc.) until a `put` method put a message in the mailbox.

`try_peek(`*`variable`*`)`

Assigns the value of the first message to the variable without removing the message. If the mailbox is empty, this method returns the 0 value. If the message is available, this method returns a non-zero value. After returning the value, VCS executes the next statement.

The official description of mailbox methods begins on page 168 of the SystemVerilog 3.1a LRM.

Events

SystemVerilog has a number of extensions to named events. These extensions are as follows:

Waiting for an Event

Persistent Trigger

Merging Events

Reclaiming Named Events

Event Comparison

Waiting for an Event

You can enter a hierarchical name for a named event in an event control.

```
`timescale 1ns/1ns
program prog;

task t1;
event evt1;
#5 -> evt1;
endtask

initial
t1;

initial
@(t1.evt1) $display("t1.evt1 happened at %0t",$time);

endprogram
```

The `$display` system task displays the following:

```
t1.evt1 happened at 5
```

The official description of waiting for an event begins on page 171 of the SystemVerilog 3.1a LRM.

Persistent Trigger

The `triggered` property persists on a named event throughout the time step when it is triggered, preventing a race condition, for example, when a named event is triggered and is evaluated in an event control during the same time step.

```
program prog;

event evt1,evt2;

initial
-> evt1;

initial
```

```
begin
wait (evt1.triggered);
$display("evt1 triggered");
end

initial
fork
   -> evt2;
   begin
      wait (evt2.triggered);
      $display("evt2 occurred");
   end
join

endprogram
```

The `$display` system tasks display the following:

```
evt1 triggered
evt2 occurred
```

The official description of the `triggered` property of named events is on page 172 of the SystemVerilog 3.1a LRM.

Merging Events

You can assign a SystemVerilog named event to another named event. When you do, they alias each other and when VCS executes a line calling for the triggering of one of these events, VCS triggers both named events.

```
program prog;

event evt1, evt2, evt3;

initial
begin
evt2 = evt3;  // this is an alias
```

```
evt1 = evt3;  // this is an alias
#2 -> evt1;
end

initial
#1 @ (evt1) $display("evt1 triggerred");

initial
#1 @ (evt2) $display("evt2 triggerred");

initial
#1 @ (evt3) $display("evt3 triggerred");

endprogram
```

The `$display` system tasks display the following:

```
evt1 triggerred
evt2 triggerred
evt3 triggerred
```

IMPORTANT:
When you merge events, the merger takes effect only in subsequent event controls or `wait` statements.

In this example, the merging occurred at time 0, the event controls at time 1, and the triggering of the events at time 2.

The official description of merging named events is on page 173-174 of the SystemVerilog 3.1a LRM.

Reclaiming Named Events

When you assign the `null` keyword to a named event, that named event no longer can synchronize anything. In an event control it might block forever or not at all. In a wait statement, it is as if the named event were undefined, and triggering the named event causes no simulation events.

```
program prog;
event evt1;

initial
begin
evt1 = null;
#5 -> evt1;
end

initial
#1 @(evt1) $display("evt1 triggered");

initial
begin
#5 wait (evt1.triggered);
$display("evt1 occurred");
end
endprogram
```

The `$display` system tasks do not display anything.

The official description of reclaiming named events is on page 174 of the SystemVerilog 3.1a LRM.

Event Comparison

You can use the equality and inequality operators to see if named events are aliased to each other or have been assigned the null value, for example:

```
 program prog;

event evt1, evt2, evt3;

initial
begin
evt1 = evt2;
if (evt1 == evt2)
    $display("evt1 == evt2");
if (evt1 === evt2)
    $display("evt1 === evt2");
if (evt1 != evt3)
    $display("evt1 != evt3");
if (evt3 != null)
    $display("evt3 != null");
end
endprogram
```

The `$display` system tasks display the following:

```
evt1 == evt2
evt1 === evt2
evt1 != evt3
evt3 != null
```

In comparing named events, the case equality operator `===` works the same as the equality operator `==`, and the case inequality operator `!==` works the same as the inequality operator `!=`.

Clocking Blocks

A clocking block is a group of signals that are synchronous to a designated clock. This block makes the timing of these signals explicit. Consequently, timing and synchronization details for these signals is separate from the structural, functional, and procedural elements of the testbench. This enables synchronous events, input sampling, and synchronous drives to be written without explicitly using clocks or specifying timing.

Clocking blocks can be declared inside a program block or inside an interface.

Clocking Block Declaration

The syntax for declaring a clocking block is:

```
clocking clocking_identifier @clocking_event;
     [default clocking_dir clocking_skew [clocking_dir
          clocking_skew];]
     {clocking_dir [clocking_skew][clocking_dir
          [clocking_skew]]signal_identifier [=
          hierarchical_identifier]
          {,signal_identifier [= hierarchical_identfier]};}
endclocking[: clocking_identifier]
```

clocking_identifier

specifies the name of the clocking block being declared.

clocking_event

specifies an event that acts as the clock for the clocking block (e.g., posedge, negedge of a clocking signal):

```
@(posedge clk)
```

or

```
@(clk)
```

Note: Program signals *cannot* be used inside a clocking event expression.

clocking_dir

is the direction of the signal: input, output or inout. If specifying more than one *clocking_dir*, they must be in the order input...output:

```
input clocking_skew output clocking_skew
```

inout cannot be used in the declaration of a default skew. Also, if the *clocking_dir* of a clocking block signal is inout, you cannot specify a *clocking_skew.* For example:

```
inout #1 d; //results in a syntax error
inout d;    //is fine
```

clocking_skew

determines how long before the synchronized edge the signal is sampled, or how long after the synchronized edge the signal is driven. A *clocking_skew* can consist of an edge identifier and a delay control, just an edge identifier, or just the delay control. The edge identifiers are posedge and negedge. The edge can be specified *only* if the clocking event is a singular clock(that is, a simple edge of a single signal like @(posedge clk), @(clk), @(negedge top.clk), etc.).The delay control is introduced by "#" followed by the delay value. The following are examples of legal *clocking_skews*:

```
input #0 i1;
output negedge #2 i2;
input #1 output #2;
```

Note:
Time literals (e.g., #10ns and #2ns) are yet not supported in this release.

The skew for an input signal is implicitly negative (that is, sampling occurs before the clock event). The skew for an output signal is implicitly positive (that is, the signal is driven after the clock event).

Note: #1step is the default input skew unless otherwise specified. However, an explicit #1step skew is not yet supported.

signal_identifier

is the name of a signal in the clocking block and identifies a signal in the scope enclosing the clocking block declaration (and unless a *hierarchical_identifier* is specified). For example:

```
input #1 i1;
```

where `i1` is the *signal_identifier.*

Note: A clocking block signal can only be connected to a scalar, vector, packed array, integer or real variable. Program signals are not allowed in clocking block signal declarations.

hierarchical_identifier

specifies the hierarchical path to the signal being assigned to the *signal_identifier*. For example:

```
input negedge #2 i  = top.i2;
```

where `i2` is defined in a module "top."

Note: See page 182 of the System Verilog LRM 3.1a for formal definition of the syntax for declaring the clocking block.

Note: Slices and concatenations are not yet implemented

As indicated in the syntax above, a single skew can be declared for the entire clocking block:

```
  default input #10;
```

Default skews can be overridden when a signal is declared.

The following example includes a clocking block embedded in a program:

```
`timescale 1ns/1ns

module top;

reg out3;
reg out1;
reg clk = 0;

p1 p(out3,clk,out1);

assign out1 = out3;

initial forever begin
        clk = 0;
        #10;
        clk = 1;
        #10;
        end
endmodule

program p1(output reg out3,input logic clk,input reg in );


 clocking cb @(posedge clk);
     output #3 out2 = out3;  //CB output signal
     input #0 out1 = in;
 endclocking

 initial
  #200 $finish;
 initial begin
         $display($time,,,cb.out1);
        cb.out2 <= 0;          //driving output at "0" time
            @(cb.out1);          //sampling input for change
         $display($time,,,cb.out1);
         #100;
         $display($time,,,cb.out1);

           cb.out2 <= 1;      //driving o/p at posedge of clk

             @(cb.out1);

         $display($time,,,cb.out1);
         end

endprogram
```

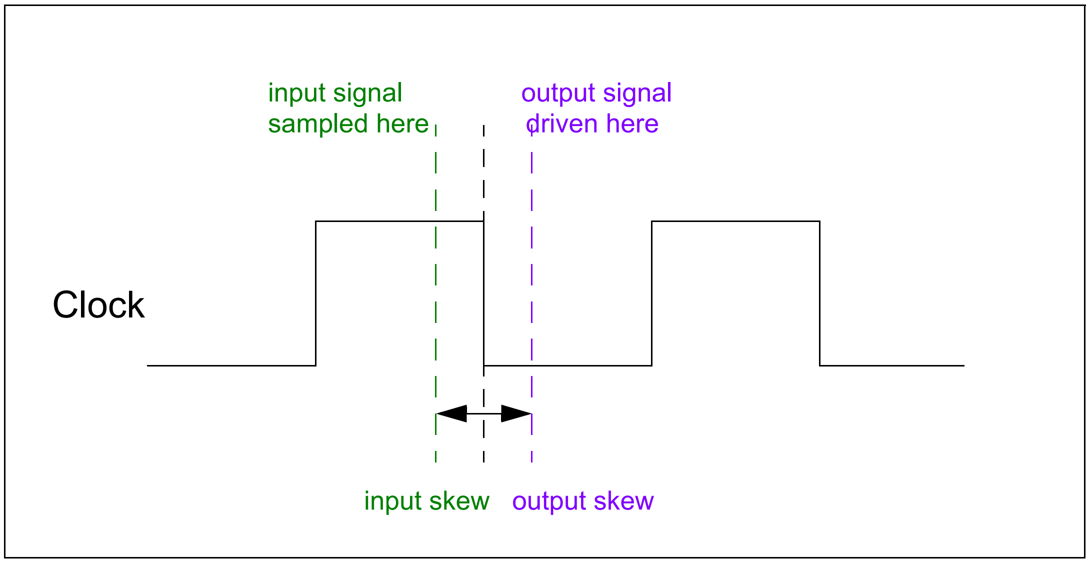
The output of the above program is:

```
            0   x
           30   0
          130   0
         150   1
```

Input and output skews

The skew for input and inout signals determines how long before *clocking_event* the signal is sampled. The skew for output and inout signals determines how long after the *clock_event* the signal is driven.

Figure 1-3 Driving and sampling on the negative edge of the clock



For more details see section 15.3 of the SystemVerilog LRM 3.1a.

Hierarchical expressions

Every signal in a clocking block is associated with a program port or a cross module reference.

As described when defining *hierarchical_identifier*, the hierarchical path is assigned to the *signal_identifier* defined in the clocking block.

```
clocking cb2 @ (negedge clk);
    input #0 b = top.q;
endclocking
```

Below is an example of the *hierarchical_identifier* as a program port:

```
program p1(output reg out3,input logic clk,input reg in );
    clocking cb @(posedge clk);
        output #3 out2 = out3;//out3 and in = program ports
        input #0 out1 = in;
    endclocking
endprogram
```

Clocking block events

The *clocking_identifier* can be used to refer to the *clocking_event* of a clocking block. For example:

```
clocking cb1 @(posedge clk);
    input #0 i1;
    input negedge #2 address;
endclocking
```

The clocking event of the `cb1` clocking block can be used to wait for that particular event:

```
    @(cb1);
```

Therefore, `@(cb1)` is equivalent to `@(posedge clk)`.

Input sampling

All inputs and inouts of a clocking block are sampled at the *clocking_event* for that block. The following is skew related behavior involving regions:

- When the skew is #0, the signal value in the Observed region corresponds to the value sampled.
- When the skew is *not* #0, then the signal value at the Postponed region of the timestep skew time-units prior to the clocking event corresponds to the value sampled.
- When the skew is #1step, the signal value in the Preponed region corresponds to the value sampled.

The last sampled value of signal replaces the signal when the signal appears in an expression.

Note:

See section 14.3 of the SystemVerilog LRM 3.1a for definitions of Observed, Postponed and Preponed regions.

Synchronous events

The event control operator, @, is used for explicit synchronization. This operator causes a process to wait for a particular event (that is, signal value change, or a clocking event) to occur.

Syntax

```
@ (expression);
```

expression

denotes clocking block input or inout, or a slice, which may include dynamic indices. The dynamic indices are evaluated when `@(expression)` executes.

For examples, see pages 189-190 of the SystemVerilog LRM3.1a

Synchronous drives

The output (or inout) signals defined in a clocking block are used to drive values onto their corresponding signals in the DUT at a specified time. That is, the corresponding signal changes value at the indicated clocking event as indicated by the output skew.

Note: For the syntax for specifying a synchronous drive, see section 15.14 of the SystemVerilog LRM 3.1a.

Consider the following clocking block and synchronous drives:

```
clocking cb1 @(posedge clk);
     default output #2;
     input #2 output #0 a = a1;
     output b = b1;
endclocking

initial
     begin
          @ (cb1); //synchronising with clocking event
          cb1.a <= 0;  //drive at first posedge
          cb1.b <= 0;   //drive after skew on first posedge
          ##2  cb1.a <= 1;
          ##1  cb1.b <= 1; //drive after 3 clock cycles
     end
```

The expression `cb1.a` (and `cb1.b`) is referred to as the *clockvar_expression* in the SytemVerilog LRM 3.1a (see page 190).

Note:Synchronous drives with a blocking cycle delay is supported. However, a synchronous drive with an intra cycle delay is not yet supported.

Drive Value Resolution

When the same net is an output from multiple clocking blocks, then the net is driven to its resolved signal value. When the same variable is an output from multiple clocking blocks, then the last drive determines the value of the variable.

SystemVerilog Assertions Expect Statements

SystemVerilog assertions expect statements differ from assert and cover statements (VCS has not implemented the assume statement) in the following ways:

- Expect statements must appear in a SystemVerilog program block, whereas assert and cover statements appear in module definitions.
- You can declare sequences and properties and use them as building blocks for assert and cover statements, but this is not true for expect statements. Expect statements don't have sequences, neither explicitly or implicitly. Expect statements have properties, but properties are not explicitly declared with the `property` keyword.

In an expect statement you specify a clock signal and have the option of specifying an edge for clocking events and delays, just like assert and cover statements, but these are not followed by a sequence, instead there is just a clock delay and an expression. There are action blocks that execute based on the truth or falsity of the expression. The clock delay can be a range of clocking events, and VCS evaluates the expression throughout that range. You can specify that the clock delay and evaluation of the expression must repeat a number of times (you can't both have a range of clocking events and also use repetition).

The following is an example of an expect statement:

```
e1: expect (@(posedge clk) ##1 in1 && in2)
         begin
           .        // statements VCS executes
           .        // if in1 && in2 is true
           .
         end
      else
         begin
           .       // Statements VCS executes
           .       // if in1 && in2 is false
           .
         end
```

Where:

`e1:`
: Is an instance name for the expect statement. You can use any unique name you want, followed by a colon (`:`).

`expect`
: The `expect` keyword.

`(@(posedge clk) ##1 in1 && in2)`
: Is the property of the expect statement. Such properties are enclosed in parentheses. This property is the following:

  `@(posedge clk)`
  : the clock signal is clk, the clocking event is a rising edge (`posedge`) on clk. Using the `posedge` keyword means that it, with the clock signal, are an expression and so are also enclosed in parentheses.

  `##1`
  : Is a clock delay. It specifies waiting for one clocking event, then evaluating the expression.

  `in1 && in2`
  : Is an expression. If true, VCS executes the first action block called the success block. if false VCS executes the second action blockafter the keyword `else`, called the failure block.

Here is another example of an expect statement. This one calls for evaluating the expression after a range of clocking events.

```
e2: expect (@(posedge clk) ##[1:9] in1 && in2)
```

```
         begin
           .        // statements VCS executes
           .        // if in1 && in2 is true
           .
         end
      else
         begin
           .       // Statements VCS executes
           .       // if in1 && in2 is false
           .
         end
```

This expression calls for evaluation the expression after 1, 2, 3, 4, 5, 6, 7, 8, and 9 clocking events, a range of clocking events from 1 to 9.

Here is another example of an expect statement. This one calls for evaluating the expression to be true a number of times after the clock delay.

```
e3: expect (@(posedge clk) ##1 in1 && in2 [*5])
         begin
           .        // statements VCS executes
           .        // if in1 && in2 is true
           .
         end
      else
         begin
           .       // Statements VCS executes
           .       // if in1 && in2 is false
           .
         end
```

`[* ]` is the consecutive repeat operator. This expect statement calls for waiting a clock delay and then seeing if the expression is true, and doing both of these things five times in a row.

Note:

You can use the `[* ]` consecutive repeat operator when you specify a range of clocking events such as `##[1:9]`.

The following is a code example that uses expect statements:

```
module test;
logic log1,log2,clk;

initial
begin
log1=0;
log2=0;
clk=0;
#33 log1=1;
#27 log2=1;
#120 $finish;
end

always
#5 clk=~clk;

tbpb tbpb1(log1, log2, clk);
endmodule

program tbpb (input in1, input in2, input clk);
bit bit1;

initial
begin

e1: expect (@(posedge clk) ##1 in1 && in2)
         begin
           bit1=1;
           $display("success at %0t in %m\n",$time);
         end
      else
         begin
           bit1=0;
           $display("failure at %0t in %m\n",$time);
         end
```

```
e2: expect (@(posedge clk) ##[1:9] in1 && in2)
         begin
           bit1=1;
           $display("success at %0t in %m\n",$time);
         end
      else
         begin
           bit1=0;
           $display("failure at %0t in %m\n",$time);
         end
e3: expect (@(posedge clk) ##1 in1 && in2 [*5])
         begin
           bit1=1;
           $display("success at %0t in %m\n",$time);
         end
      else
         begin
           bit1=0;
           $display("failure at %0t in %m\n",$time);
         end

end
endprogram
```

The program block includes an elementary clocking block, specifying a clocking event on the rising edge of clk, and no skew for signals in1 and in2.

The `$display` system tasks in the failure and success action blocks display the following:

```
failure at 15 in test.tbpb1.e1

success at 65 in test.tbpb1.e2

success at 125 in test.tbpb1.e3
```

Virtual Interfaces

A Virtual Interface (VI) allows a variable to be a *reference* to an actual instance of an interface. VCS classifies such a variable with the Virtual Interface data type. Here is an example:

```
interface SBus;
    logic req, grant;
endinterface

module m;
   SBus sbus1();
   SBus sbus2();
   ⋮
endmodule

program P;
virtual SBus bus;

initial
begin
   bus = m.sbus1;  // setting the reference to a real
                   // instance of Sbus
   $display(bus.grant); // displaying m.sbus1.grant
   bus.req <= 1; // setting m.sbus1.req to 1
   #1 bus =  m.sbus2;
   bus.req <= 1;
end
endprogram
```

Scope of Support

VCS supports virtual interface declarations in the following locations:

- program blocks and classes (including any named sub-scope)

- tasks or functions inside program blocks or classes (including any named sub-scope).

Variables with the virtual interface data type can be either of the following:

- SystemVerilog class members.
- Program, task, or function arguments.

You cannot declare a virtual interface in a module or interface definition.

Virtual Interface Modports

If only a subset of Interface data members is bundled into a modport, a variable can be declared as "virtual Inreface_name.modport_name":

Example.

```
interface SBus;
    logic req, grant;
    modport REQ(input req);
endinterface
program P;
virtual Sbus.REQ sb;
```

The semantic meaning is the same as in the example above with the difference that sb is now a reference only to a portion of Sbus and writing assignments are subject to modport direction enforcement so that, for example, "sb.req = 1" would become illegal now (violates input direction of the modport REQ).

Clocking Block

Similarly to modport a clocking block inside interface instance can be referred to by a virtual variable:

```
interface SyncBus(input bit clk);
   wire w;
   clocking cb @(posedge clk);
       output w;
   endclocking
endinterface
....
program P;
virtual SyncBus vi;
...
initial vi.cb.w <= 1;
...
endprogram
```

In this case the assignment executes in accordance with the clocking block semantics.

Event Expression/Structure

Consider SyncBus as defined in the section “Clocking Block” .

```
task wait_on_expr(virtual SyncBus vi1, virtual SyncBus vi2);
   @(posedge (vi1.a & vi2.a))
      $display(vi1.b, vi2.b);
endtask
```

There is a principal difference between Vera and SV in that the "@" operator can have an operand that is a complex expression. We support all event expression that involve virtual interface variables.

Structures inside an interface can also be referred to by means of a virtual interface.

Null Comparison

We support:

- Comparison of vi variable with NULL.
- Runtime error if uninitialized virtual interface is used

```
begin
  virtual I vi;
  vi.data <= 1;
end
```

- NULL assignment

```
virtual vi = 0;
```

Not Yet Implemented

- Named type that involves virtual interface
  - typedef struct { reg rl; virtual I ii} T;
  - typedef virtual I T;
- Arrays of virtual interfaces

```
virtual I vi[10];
```

We suggest using array of class object when each object has a virtual interface variable as immediate workaround;

```
class C;
   virtual I vi;
```

```
endclass
C c[10];
 ...
c[2].vi = top.il;
```

- Comparison of vi variables

  By definition, vi1 == vi2 iff they refer to the same instance of an interface (or both NULL).

- VI variables defined in the design.

Coverage

The VCS implementation of SystemVerilog supports the `covergroup` construct. Covergroups are specified by the user. They allow the system to monitor values and transitions for variables and signals. They also enable cross coverage between variables and signals.

VCS collects all the coverage data during simulation and generates a database. VCS provides a tool to read the database and generate text or html reports (see page-118) .

The covergroup Construct

The `covergroup` construct specifies the set of cover points of interest, crosses of these cover points and the clocking event that tells VCS when to sample these cover points during simulation.

```
program prog;
```

```
bit clk = 0;

enum {red, blue, yellow} colors;
colors my_color;

covergroup cg1 @(posedge clk);
         cp1 : coverpoint my_color;
endgroup

cg1 cg1_1 = new;

initial
repeat (15)
     #5 clk = ~clk;

initial
begin
   #40 my_color = blue;
   #23 my_color = yellow;
end

endprogram
```

This program contains the following:

- The enumerated data type colors, with members named red, blue, and yellow (whose default values are 0, 1, and 2).
- A variable of type colors called my_color
- A covergroup named cg1 that specifies the following:
  - the clocking event that is the rising edge on signal clk.
  - the coverage point that is to monitor the values of the variable my_color. The identifier of the coverage point, for hierarchical name purposes, is cp1.
- an instantiation of covergroup cg1 using the `new` method.

A covergroup can be defined inside a class. Furthermore there can be multple covergroups in a class.

The following is an example of declaring a covergroup inside a class.

```
program P;
     class MyClass;
          int m_a;
          covergroup Cov @(posedge clk);
               coverpoint m_a;
          endgroup

          function new();
               Cov = new;
          endfunction
     endclass
endprogram
```

The official description of the `covergroups` begins on page 306 of the SystemVerilog 3.1a LRM.

Defining a Coverage Point

In a coverage point definition you can specify the following:

- bins for value ranges
- bins for value transitions
- bins for illegal coverage point values

Bins for Value Ranges

You use the curly braces `{ }` and the `bins` keyword to specify the bins for a coverage point, for example:

```
covergroup cg1 @ (posedge clk);
coverpoint data
{
bins some_name [] = {[1:20]};
}
endgroup
```

In coverage point data:

- The keyword `bins` specifies one or more bins for coverage data.
- The name `some_name` is an identifier for all the bins. It is the root bin name.
- The empty square brackets `[]` specifies there will be a separate bin for each value in the specified range.
- The range of value follows the equal sign `=` and is in the nested set of curly braces `{ }`. This range is 1 through 20. The range is always specified as *`lowest_value`*`:`*`highest_value`*.

Coverage point data will have 20 bins, the first named some_name_1 and the last named some_name_20.

You can specify different bins for different value ranges, for example:

```
coverpoint data
{
bins first_ten = {[1:10]};
bins second_ten = {[11:20]};
}
```

Here the coverage information about when the coverage point data has the values 1 to 10 is in the bin named first_ten, and the information about when data has the values from 11 to 20 is in the bin named second_ten.

You can specify a default bin with the `default` keyword, for example:

```
coverpoint data
{
bins bin1 = {[1:5]};
bins bin2 = {[6:10]};
bins bin3 = default;
}
```

In this example coverage information about when data has the values 1-10 is in bins bin1 and bin2, information about all other values is in bin3.

You can specify a list of value ranges for example:

```
coverpoint data
{
bins bin1 = {[0:3],5,7,[9:10]};
bins bin2 = {4,6,8};
bins bin3 = default;
}
```

Here the information about when data is 0, 1, 2, 3, 5, 7, 9, and 10 is in bin1, the information about when data is 4, 6, and 8 is in bin2, and the information about when data has any other value is in bin3.

When you instantiate the covergroup, you can make the covergroup a generic covergroup and then pass integers to specify value ranges in the `new` method, for example:

```
covergroup cg1 (int low, int high) @ (posedge clk);
coverpoint data
{
bins bin1 = {[low:high]};
}
endgroup

cg1 cg1_1 = new(0,10);  // 0 is the low value
```

```
                               // 10 is the high value
                               // of the range
```

Bins for Value Transitions

In a transition bin you can specify a list of sequences for the coverpoint. Each sequence is a set of value transitions, for example:

```
coverpoint data
{
bins from0 = (0=>1),(0=>2),(0=>3);
bins tran1234 = (1=>2=>3=>4);
bins bindef = default;
}
```

In this example, coverage information for the sequences 0 to 1, 0 to 2, or 0 to 3 is in the bin named from0. Coverage information about when data transitioned from 1 to 2 and then 3 and then 4 is in bin tran1234.

You can use range lists to specify more than one starting value and more than one ending value, for example:

```
bins from1and5to6and7 = (1,5=>6,7);
```

Is the equivalent of:

```
bins from1and5to6and7 = (1=>6, 1=>7, 5=>6, 5=>7);
```

You can use the repetition `[* ]` operator, for example:

```
bin fivetimes3 = (3 [*5]);
```

Is the equivalent of:

```
bin fivetimes3 = (3=>3=>3=>3=>3);
```

You can specify a range of repetitions, for example:

```
bin fivetimes4 = (4 [*3:5]);
```

Is the equivalent of:

```
bin threetofivetimes4 = (4=>4=>4,4=>4=>4=>4,4=>4=>4=>4=>4);
```

Specifying Illegal Coverage Point Values

Instead of specifying a bin with the `bins` keyword, use the `illegal_bins` keyword to specify values or transitions that are illegal.

```
coverpoint data
{
illegal_bins badvals = {7,11,13};
illegal_bins badtrans = (5=>6,6=>5);
bins bindef = default;
}
```

VCS displays an error message when the coverage point reaches these values or makes these transitions.

Defining Cross Coverage

Cross coverage is when there are two coverage points that you want VCS to compare to see if all the possible combinations of the possible values of the two coverage points occurred during simulation. Consider the following example:

```
program prog;
bit clk;
bit [1:0] bit1,bit2;
```

```
covergroup cg1 @(posedge clk);
  bit1: coverpoint bit1;
  bit2: coverpoint bit2;
  bit1Xbit2: cross bit1, bit2;
endgroup

cg1 cg1_1 = new;

initial
begin
  clk = 0;
  repeat (200)
   begin
       bit1 = $random();
       bit2 = $random();
    #5 clk = ~clk;
    #5 clk = ~clk;
  end
end

endprogram
```

In covergroup cg1 there are two coverpoints labled bit1 and bit2. In addition,there is the following:

1. the `bit1Xbit2` identifier for the cross.
2. The `cross` keyword, specifying the coverpoints to be crossed.

Both coverpoints are two-bit signals. The four possible values of each are 0 through 3. There are 16 possible combinations of values.

The prog.txt file for this code contains the following:

```
 Automatically Generated Cross Bins

 bit1           bit2           # hits         at least
=========================================================
 auto[0]        auto[0]        10             1
```

```
    auto[0]        auto[1]        13             1
    auto[0]        auto[2]        12             1
    auto[0]        auto[3]        5              1
    auto[1]        auto[0]        12             1
    auto[1]        auto[1]        18             1
    auto[1]        auto[2]        10             1
    auto[1]        auto[3]        13             1
    auto[2]        auto[0]        19             1
    auto[2]        auto[1]        16             1
    auto[2]        auto[2]        17             1
    auto[2]        auto[3]        6              1
    auto[3]        auto[0]        6              1
    auto[3]        auto[1]        15             1
    auto[3]        auto[2]        16             1
    auto[3]        auto[3]        12             1
   =======================================================
```

There are 16 cross coverage bins, one for each possible combination.

Defining Cross Coverage Bins

```
covergroup cg1 @(posedge clk);
  cp1 : coverpoint bit1
  {
    bins lowcp1vals = {[0:7]};
    bins hicp1vals = {[8:15]};
  }
  cp2 : coverpoint  bit2
  {
     bins lowcp2vals = {[0:7]};
    bins hicp2vals = {[8:15]};
  }
  cp1Xcp2 : cross cp1, cp2
  {
    bins bin1 = binsof(cp1) intersect {[0:7]};
    bins bin2 = binsof(cp1.hicp1vals) ||
         binsof(cp2.hicp2vals);
    bins bin3 = binsof(cp1) intersect {[0:1]} &&
         binsof(cp2) intersect {[0:3]};
  }
endgroup
```

In this example, the respective cross coverage bins, `bin1`, `bin2`, and `bin3` get a hit (receive data) whenever the corresponding RHS binsof expressions are satisfied. For example, `bin1` gets a hit when any bin of `cp1` whose value range overlaps with the range [0:7] gets a hit. In this case `bin1` gets a hit whenever bin `lowcp1vals` of coverpoint `cp1` gets a hit.

Similarly, cross bin `bin2` gets a hit whenever either bin `hicp1vals` of coverpoint `cp1` gets a hit, or bin `hicp2vals` of cover point `cp2` gets a hit. Cross bin `bin3` gets a hit if any bin of cover point `cp1` whose value range overlaps with the range [0:1] gets a hit and, for the same sample event occurrence, any bin of cover point `cp2` whose value range overlaps with the range [0:3] also gets a hit. In this example, cross bin `bin3` gets a hit when bin `lowcp1vals` of cover point `cp1` gets a hit and bin `lowcp2vals` of cover point `cp2` also gets a hit.

In case none of the user defined cross bins are matched, then VCS automatically creates an auto cross bin to store the hit count for each unique combination of the cover point bins.

Coverage Options

You can specify options for the coverage of a covergroup with the `type_option.`*`option=argument`* keyword and argument for specifying options, for example:

```
covergroup cg2 @(negedge clk);
type_option.weight = 3;
type_option.goal = 99;
type_option.comment = "Comment for cg2";
  cp3 : coverpoint bit3;
  cp4 : coverpoint bit4;
```

```
endgroup
```

These options specify the following:

`type_option.weight =` *`integer`*`;`
: Specifies the weight of the covergroup when calculating the overall coverage. Specify an unsigned integer. The default weight value is 1.

`type_option.goal = integer;`
: Specifies the target goal of the covergroup. Specify an integer between 0 and 100. The default goal is 90.

Note: "Coverage number" is a percentage. If all bins of a covergroup are covered, then the coverage number for that covergroup is 100%.

`type_option.comment = "`*`string`*`";`
: A comment in the report on the covergroup.

You can also apply these option to coverage points, for example:

```
covergroup cg1 @(posedge clk);
  cp1 : coverpoint bit1
        {
        type_option.weight = 333;
        type_option.goal = 50;
        type_option.comment = "Comment for bit1";
        }
  cp2 : coverpoint  bit2;
endgroup
```

You can also apply these options to instances using the *`instance_name`*`.option.`*`option_name=argument`* keyword and argument, for example:

```
covergroup cg1 @(posedge clk);
  cp1 : coverpoint bit1;
  cp2 : coverpoint  bit2;
endgroup

cg1 cg1_1 = new;

initial
begin
  cg1_1.option.weight = 10;
  cg1_1.option.goal = 75;
⋮
end
```

Instance specific options are procedural statements in an initial block.

There are additional options that are just for instances:

`*instance_name*.option.at_least=*integer*`
: Specifies the minimum number of hits in a bin for VCS to consider the bin covered.

`*instance_name*.option.auto_bin_max=*integer*`
: Specifies the maximum number of bins for a coverage point when you don't define the bins for a coverage point.

`*instance_name*.option.detect_overlap=*boolean*`
: The *boolean* argument is 1 or 0. When *boolean* is 1, VCS displays a warning message when there is an overlap between the range list or transitions list of two bins for the coverage point.

`*instance_name*.option.name[=*string*]`

: This option is used to specify a name for the covergroup instance. If a name is not specified, a name is automatically generated.

`*instance_name*.option.per_instance=*boolean*`
: The *boolean* argument is 1 or 0. When *boolean* is 1, VCS keeps track of coverage data for the instance.

Predefined Coverage Methods

SystemVerilog provides a set of predefined covergroup methods described in this section. These predefined methods can be invoked on an instance of a covergroup. They follow the same syntax as invoking class functions and tasks on an object.

Predefined Coverage Group Functions

The predefined methods supported at this time are:

- get_coverage()
- get_inst_coverage()
- set_inst_name(string)
- sample()
- stop()
- start()

get_coverage()

Calculates the coverage number for the covergroup type (see page 109 for definition). Return type: real.

Below is an example of using `get_coverage()` to calculate the coverage number of a covergroup:

```
program test();
reg clk = 0;
reg [2:0] var = 3'b001;
class A;
covergroup covType  @(clk); //covergroup, covType, defined
                            //in class A,
    cp1: coverpoint var {
        bins s0 = {[ 0 : 2]} ;
        bins s1 = { 3 };
        bins s2 = { 4 };
        bins s3 = { 5 };
        bins s4 = { 6 };
        bins s5 = { 7 };
    }
endgroup

function new;
    covType = new(); //instantiate the embedded covergroup
endfunction

endclass

A A_inst;

initial begin
    repeat (10) begin
        #5 clk = ~clk;
        var = var + 1;
/* get_coverage() calculates the number of the embedded
covergroup covType as a whole */
        $display("var=%b coverage=%f\n", var,
             A_inst.covType.get_coverage());
    end
end

initial
     A_inst = new();

endprogram
```

Output of program:

```
var=010 coverage=0.000000

var=011 coverage=16.666666

var=100 coverage=33.333332

var=101 coverage=50.000000
```

```
var=110 coverage=66.666664

var=111 coverage=83.333336

var=000 coverage=100.000000

var=001 coverage=100.000000

var=010 coverage=100.000000

var=011 coverage=100.000000
```

See the get_coverage(), stop(), start() example on page 116 for another example of using the get_coverage() function .

get_inst_coverage()

Calculates the coverage number for coverage information related to the covergroup instance. Return type: real.

```
program test();
reg clk = 0;
reg [2:0] var = 3'b001;

covergroup covType (input integer param1) @(clk);
    cp1: coverpoint var {
        bins s0 = { [ 0 : param1]  } ;
        bins s1 = { 3 };
        bins s2 = { 4 };
        bins s3 = { 5 };
        }
endgroup

covType cov1;

initial begin
    repeat (5) begin
        #5 clk = ~clk;
        var = var + 1;
        $display("var=%b coverage=%f\n", var,
cov1.get_inst_coverage());
    end
end

initial
   cov1 = new(2);

endprogram
```

The output of the program is:

```
var=010 coverage=-1.000000

var=011 coverage=-1.000000

var=100 coverage=-1.000000

var=101 coverage=-1.000000

var=110 coverage=-1.000000
```

set_inst_name(*string*)

The instance name is set to *string*. Return type: void.

In the example below, `cov1` is the name of an instance that is of type `covType`, declared as such (`covType cov1; cov1 = new(2);`). The name is changed to `new_cov1` after calling `set_inst_name("new_cov1")`.

```
program test();
reg clk = 0;
reg [2:0] var = 3'b001;

covergroup covType (input integer param1) @(clk);
    cp1: coverpoint var {
        bins s0 = { [ 0 : param1]  } ;
        bins s1 = { 3 };
    }
endgroup
```

```
covType cov1;

initial
begin
     cov1 = new(2);
     $display("Original instance name is %s\n",
          cov1.option.name);
     cov1.set_inst_name("new_cov1");//change instance name
    $display("Instance name after calling set_inst_name()
          is %s\n", cov1.option.name);
end

endprogram
```

Output of the program is:

```
Original instance name is cov1

Instance name after calling set_inst_name() is new_cov1
```

sample()

sample() triggers sampling of the covergroup instance. Return void.

```
program test();
reg clk = 0;
reg [2:0] var = 3'b001;

covergroup covType ();
    cp1: coverpoint var {
        bins s0 = { 0 };
        bins s1 = { 1 };
        bins s2 = { 2 };
        bins s3 = { 3 };
        bins s4 = { 4 };
        bins s5 = { 5 };
        bins s6 = { 6 };
        bins s7 = { 7 };
    }
endgroup
```

```
covType cov1;

initial
   cov1 = new();

initial begin
    repeat (10) begin
        #10 clk = ~clk;
        var = var + 1;
    end
end
initial begin
    repeat (40) begin
        #3  cov1.sample();
    end
end


endprogram
```

stop()

When called, collecting of coverage information is stopped for that instance. Return type: void. See the get_coverage(), stop(), start() example on page 116.

start()

When called, collecting of coverage information resumes for that instance. Return type: void. See the get_coverage(), stop(), start() example on page 116.

get_coverage(), stop(), start() example

```
program test();
reg clk = 0;
reg [2:0] var = 3'b001;

covergroup covType (input integer param1) @(clk);
```

```
    cp1: coverpoint var {
        bins s0 = { [ 0 : param1]  } ;
        bins s1 = { 3 };
        bins s2 = { 4 };
        bins s3 = { 5 };
        bins s4 = { 6 };
        bins s5 = { 7 };
    }
endgroup

covType cov1;

initial begin
    repeat (10) begin
        #5 clk = ~clk;
        var = var + 1;
        $display("var=%b covergae=%f\n", var,
cov1.get_coverage());
    end
end

initial
begin
   cov1 = new(2);
   cov1.stop();
   cov1.option.weight = 5;
   #30 cov1.start();
end

endprogram
```

OUTPUT:

```
var=010 covergae=0.000000

var=011 covergae=0.000000

var=100 covergae=0.000000

var=101 covergae=0.000000
```

```
var=110 covergae=0.000000

var=111 covergae=0.000000

var=000 covergae=16.666666


var=001 covergae=33.333332

var=010 covergae=33.333332

var=011 covergae=33.333332
```

The Coverage Report

During simulation VCS writes a .db file named after the program. For the code example above, VCS writes the prog.db file in the current directory. This file is the coverage database file, it contains all the information VCS needs to write a coverage report. There are two types of coverage reports:

- an ASCII text file
- an HTML file

The ASCII Text File

The command to instruct VCS to generate a coverage report in ASCII format is :

```
vcs -cov_text_report prog.db
```

The `-cov_text_report` option instructs VCS to generate, in the current directory, a coverage report in a file called *program_name*.txt. The ASCII coverage report for the previous code example is as follows:

```
Functional Coverage Report

Coverage Summary
    Number of coverage types: 1
    Number of coverage Instances: 1
    Cumulative coverage: 100.00
    Instance coverage: 0.00

Coverage Groups Report

Coverage group                          # inst    weight    cum cov.
==============================================================================
cg1                                     1         1         100.00
=============================================================================


==============================================================================
= Cumulative report for cg1
==============================================================================

Summary:
    Coverage: 100.00
    Goal: 90

Coverpoint                           Coverage      Goal          Weight
==============================================================================
cp1                                  100.00         90            1
==============================================================================


Coverpoint Coverage report
CoverageGroup: cg1
    Coverpoint: cp1
Summary
    Coverage: 100.00
    Goal: 90
    Number of User Defined Bins: 0
    Number of Automatically Generated Bins: 3
    Number of User Defined Transitions: 0

    User Defined Bins
    Bin                                     # hits    at least
    ============================================================
    auto_blue                               2         1
    auto_red                                4         1
    auto_yellow                             2         1
    ============================================================


Database Files Included in This Report:

    prog.db


Test Bench Runs Included in This Report:

Test Bench           Simulation Time      Random Seed          Random48 Seed
============================================================================
prog.db::prog        75                   0                    0
============================================================================
```

The report begins with a summary of the coverage for the covergroup cg1. There is 100% coverage, meaning that VCS saw all possible values for the coverage point(s) in the covergroup.

For coverage point cp1, the report says the following:

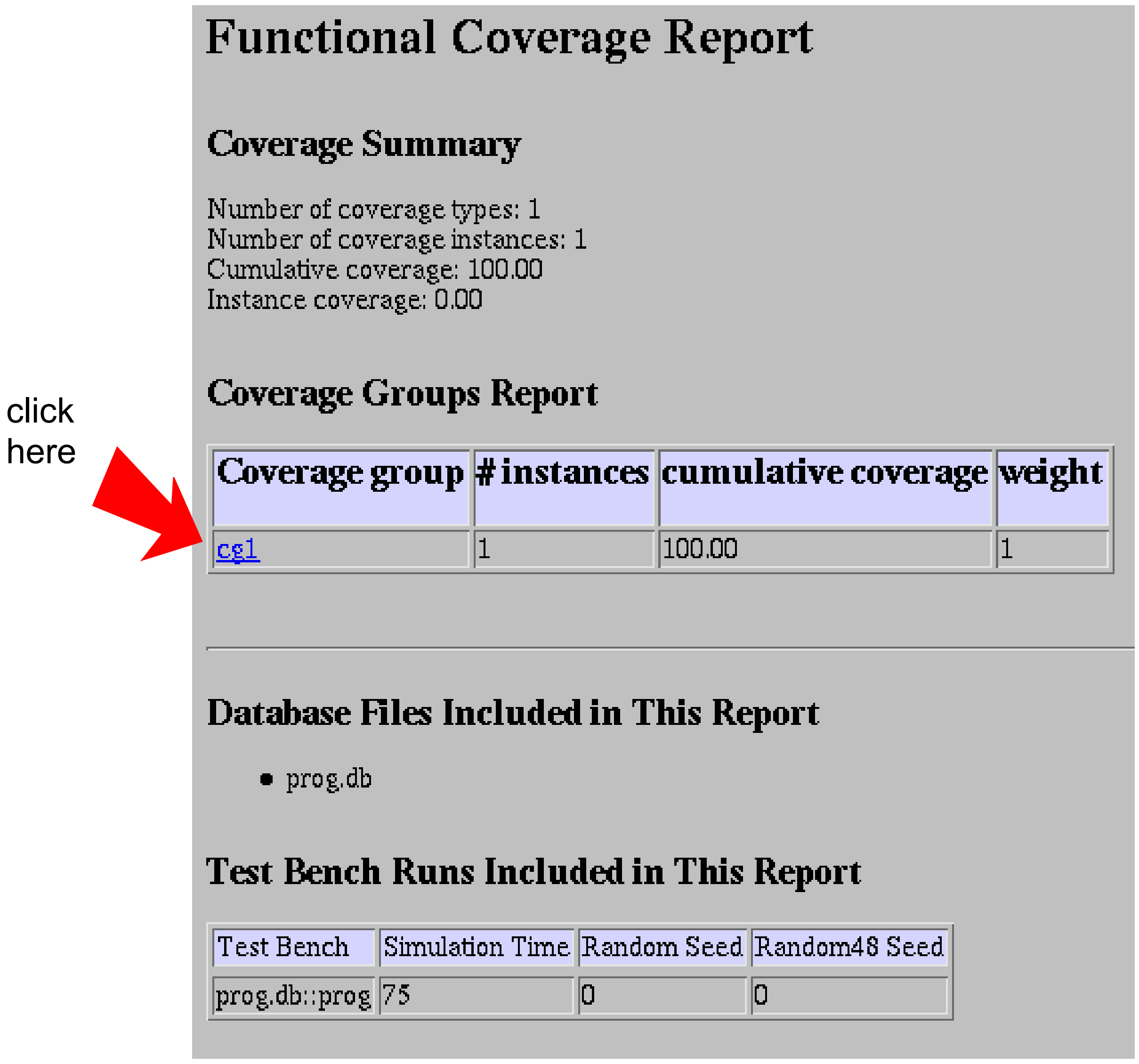
- The coverage point, in this case variable my_color, reached all its possible values.
- There were no user defined bins.
- VCS created bins for the coverage point named auto_blue, auto_red, and auto_yellow, all named after the members of the enumerated type colors, blue, red, and yellow.
- There were four hits for bin auto_red, this means that during simulation, there were four clocking events (rising edge on signal clk) where VCS detected that the value of the variable my_color was red..
- There were two hits for bins auto_blue and auto_yellow.

The HTML File

The command to instruct VCS to generate a coverage report in HTML format is :

```
vcs -cov_report prog.db
```

The `-cov_report` option instructs VCS to generate, in the current directory, a coverage report in a file called *program_name*.index.html. The HTML coverage report for the previous code example is as follows:



Functional Coverage Report

Coverage Summary

Number of coverage types: 1
Number of coverage instances: 1
Cumulative coverage: 100.00
Instance coverage: 0.00

Coverage Groups Report

Coverage group	# instances	cumulative coverage	weight
cg1	1	100.00	1

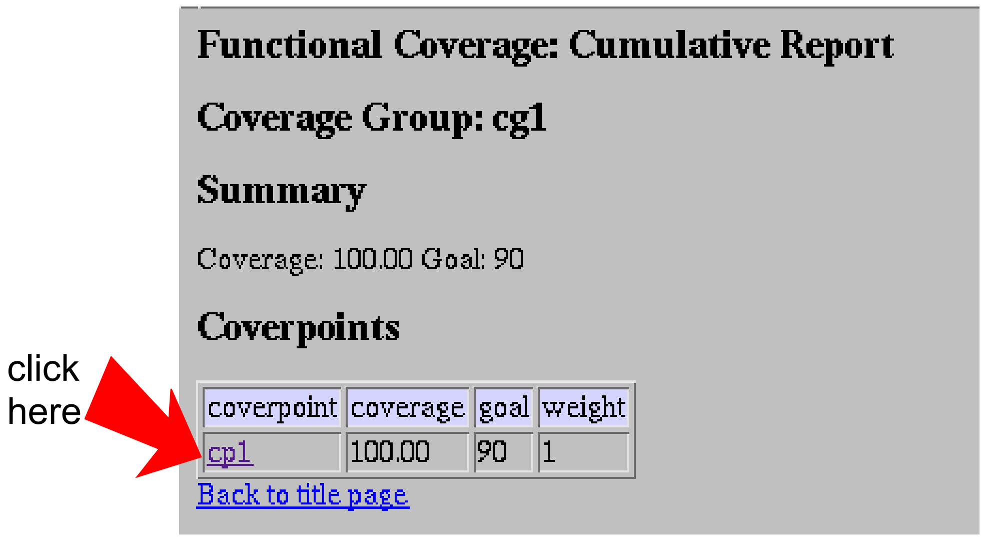
Database Files Included in This Report

- prog.db

Test Bench Runs Included in This Report

Test Bench	Simulation Time	Random Seed	Random48 Seed
prog.db::prog	75	0	0

The report shows one covergroup cg1. Click on cg1 to see information about covergroup cg1.



Functional Coverage: Cumulative Report

Coverage Group: cg1

Summary

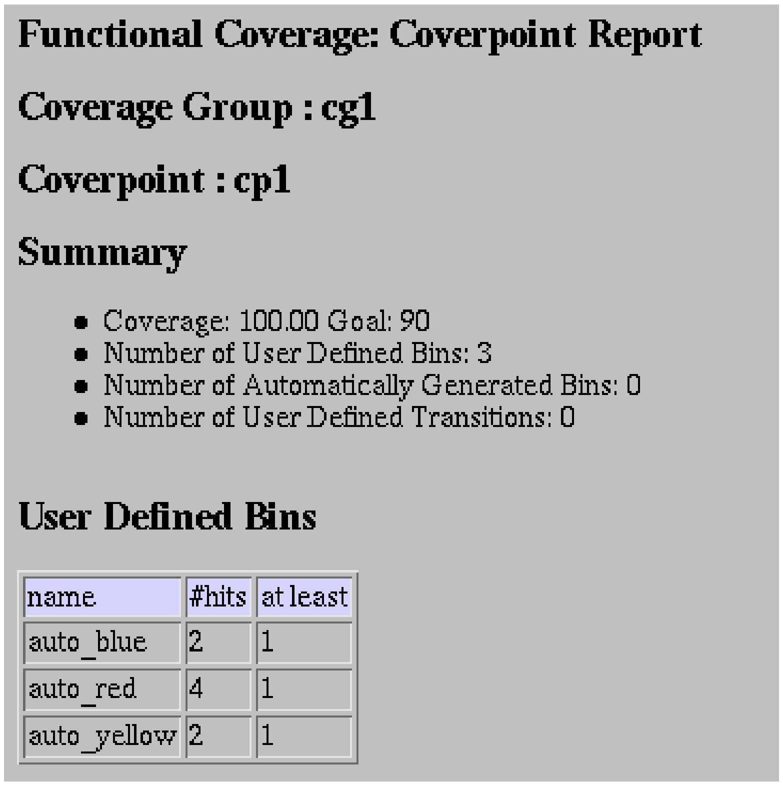
Coverage: 100.00 Goal: 90

Coverpoints

coverpoint	coverage	goal	weight
cp1	100.00	90	1

Back to title page

The report shows one coverage point, cp1. Click on cp1 to see information about coverage point cp1.



Functional Coverage: Coverpoint Report

Coverage Group : cg1

Coverpoint : cp1

Summary

- Coverage: 100.00 Goal: 90
- Number of User Defined Bins: 3
- Number of Automatically Generated Bins: 0
- Number of User Defined Transitions: 0

User Defined Bins

name	#hits	at least
auto_blue	2	1
auto_red	4	1
auto_yellow	2	1