Here are my comments on chapter 12.

Section 12.1:
Similar to my comments in chapter 11, I would like to replace the object oriented framework with an object oriented class type system. The reason for this is "framework" is very vague and kind of foreign to Verilog. Instead describing it as a class type system may be more illustrative to users.
The random constraints are built on top of an object oriented framework that models the data to be randomized as objects that contain random variables and user-defined constraints.

DWS: Done in LRM-221

In the following sentence, perhaps the term object is too general and should be qualified with class or replaced with class. In Verilog, users may think of objects as registers and variables. Either we replace object with class objects, or we define the term object at the beginning of this chapter and in the class chapter. (should also appear in the glossary).

Objects are ideal for representing complex aggregate data types and protocols such as Ethernet packets.

DWS: Object is defined in Section 11 and used consistently here. Since it is defined in Section 11 I do not think we need to put it in the glossary.

Similarly in the next sentence I would replace object-based by class-based:
Section 12.2 provides an overview of object-based randomization and constraint programming.

DWS: The term object is correct here. It is the instance that is important and not the definition.

Section 12.2:
page 101:
The following sentence is repeated twice:
Using inheritance to build layered constraint systems enables the development of general purpose
models that can be constrained to perform application-specific functions.

DWS: Fixed in LRM-222

Q: is it allowed to call .randomize method if the class has no random constraint variables? Does this result in a runtime error?

DWS: I would expect that if you call it nothing happens. If you wish to make it an implementation error or warning that might be useful but it is well defined.

In the following example, there is no pre_randomized or post_randomize methods in the class XYZ. What is then the effect of calling super.pre_randomize?
Is it a legal example?
The "endtask" in the example should be removed.

```
class XYPair;
rand integer x, y;
endclass
class MyYXPair  extends XYPair
function void pre_randomize();
super.pre_randomize();
$display("Before randomize x=%0d, y=%0d", x, y);
endfunction
function void post_randomize();
super.post_randomize();

$display("After randomize x=%0d, y=%0d\n", x, y);
endtask
endfunction
```

DWS: LRM_96 takes care of the syntax errors. The text describes pre_randomize and post_randomize as being built-in methods that can be overloaded. This means they exist (and do nothing). The example is correct since the implementation of MyYXPair needs to call the parent ones so that if they are overloaded in the future they behave as required. See the paragraph after the example.

Section 12.3 random Variables:
— The solver can randomize scalar singular variables of any integral type such as integer, enumerated types, and packed array variables of any size.

Q: what about packed structs, associative arrays, dynamic arrays, If these are included in integral types, we should remove the such as...

DWS: Remove “such as clause” in LRM-223.

Note that both singular and integral should appear in the glossary.

DWS: For later since they are defined in the LRM in other places.

I don't understand the following sentence:
If the array elements are object handles, all of the array
elements must be non-null. Individual array elements can be constrained, in which case the index expression must be a literal constant.

Q: Why are all array elements of an associative array required to be non null?
Q: is the last sentence (the index expression must be a literal constant) addressing both associative and dynamic arrays?

Q: if you have an array of packed struct S, and you randomize the array:
myarr.randomize (does it randomize the individual struct elements (S.a, S.b) even if each struct element is not declared as rand?

An object handle can be declared **rand** in which case all of that object's variables and constraints
are solved concurrently with the variables and constraints of the object that contains the handle.
Objects cannot be declared **randc**.
Q: does this also applies to packed structs

Q: How do you randomize a packed union? do you provide random values for each one of the union members? or you have to specify a single union member?

Section 12.4.1 external constraint blocks
Constraint block bodies can be declared outside a class declaration, just like external task and function bodies:

Q: Please define exactly in which declarative context can a constraint block appear.
Q: Does the class it constrains need to be visible?
Q: Can it constrains a class which is referred to by an XMR such as:
constraint top.mod.`XYPair::c { x < y; }`

DWS: I have sent these questions to Arturo and Mehdi for responses. These questions are surprising since we have had two reviews in the EC and they did not come up. Good questions but a little late.

Section 12.4.3:
value_range_list is a comma-separated list of integral expressions and ranges. Ranges are specified with a low
and high bound, enclosed by square braces **[ ]**, and separated by a colon (**:**), as in
`[low_bound:high_bound]`. Ranges

Verilog also ranges for part select expressions specified as [0+:3] and [3-:3] , why isn't it allowed here?

DWS: Did not come up in committee. Have to bring up in next version of SystemVerilog since this is a semantic change.

I think we should add that the expressions or value ranges must be compile time determinable (not dynamic).

The title of this section talks about the set membership operator but it is not clear what is this operator? Is it inside?
A formal definition of what is the set membership operator is needed.

DWS: The BNF in 12.4.2 clearly show the operator as inside as does the operator table in 7.9. The definition is actually quite formal. What suggestions or changes do you have (too late now for 3.1 but for later…).

Section 12.4.4 page 107
Replace: The distribution operator **dist** evaluates to true if the expression is contained in the set; otherwise it evaluates to false.
WIth:
The distribution operator dist evaluates to true of the value of the expression is contained in the set, otherwise it evaluates to false.

Because the expression is not contained in the value set, but the value of the expression should be.

DWS: Fixed in LRM-224

Optionally, each term in the list can have a weight, which is specified using the **:=** or **:/** operators. If no weight is specified, the default weight is 1. The weight can be any integral SystemVerilog expression.

Q: What occurs if the same value has 2 different weights; is it allowed?

DWS: The answer is for example x dist {100 := 1, 100 := 2, 300 := 5} would result in 100 be 3 out of 8 and 300 being 5 out of 8. No problem. It is allowed.

Section 12.4.7: Global constraints:

Q: Can the solver determine syntactically that this is a global constraint?

DWS: Sure. The example seems clear.

Section 12.5.3:
I don't think that these are notes, they are fairly important.
They appear to be like rules.
I suggest that each of these rules is inserted in the appropriate section and that section 12.5.3 is entirely deleted.

DWS: This was change to this as a result of EC-CH116 which has already been voted on. No change.

Section 12.6

I read the following multiple times and still don't understand the meaning:
The scope for variable names in a constraint block, from inner to outer, is:
randomize()...**with** object class, automatic and local variables, task and function

parameters, class variables, variables in the enclosing scope. The **randomize()**...**with** class is brought into scope at the innermost nesting level.

DWS: What is confusing? It defines the nesting of the scope for variable names. randomize()..with provides a scope, automatic and local variables in a subroutine are in the subroutine scope, parameters to a subroutine are in a scope, class variables are in the enclosing class scope, and the scope that the class is defined within is the outer scope. The innermost scope is randomize()..with. This has a constraint block (which is the scope).

Page 114:
Is the meaning of :
In the example, below, the **randomize()**...**with** class is `Foo`.

The class to which the "randomize..with" function applies is the class Foo?

DWS: Yes. The randomize..with is really f.randomize...with where f is of type class Foo.

In the following sentence, replace parameter by formal:
In the `f.randomize() with` constraint block, `x` is a member of class `Foo`, and hides the `x` in class `Bar`. It
also hides the `x` parameter in the `doit()` task. `y` is a member of `Bar`. `z` is a local parameter.

DWS: Fixed in LRM-226 using argument (too be consistent with LRM-1).

Q: Is the rand_mode applied recursively?
random_variable is the name of the random variable to which the operation is applied. If it is not specified
(only allowed when called as a task), the action is applied to all random variables within the specified object.

DWS: Yep to all variables within the class and the nested classes.

Q: in the following sentence, is it really a compile error or should it be a runtime error which occurs when the statement rand_mode is executed?
A compiler error shall be issued if the specified variable does not exist within the class hierarchy or it exists but is not declared as **rand** or **randc**.

DWS: Yep. That is what we voted on.

Q: In the following, it is assumed that singular variables do not include packed arrays and packed struct unions? However they are singular types.
The function form of **$rand_mode()** only accepts singular variables, thus, if the specified variable is an array, a single element must be selected via its index.

DWS: Change to unpacked array (since packed arrays are singular) in LRM-225

Section 12.8: Controlling constraints with constraint_mode

Q: shouldn't it be a runtime error instead?
A compiler error shall be issued if the specified constraint block does not exist within the class hierarchy.

DWS: Nope. That is what we voted on.

Section 12.9: Dynamic constraint modifications
This section should appear much earlier as an introduction (instead of a summary) for all types of dynamic modifications. This chapter 12 could be reorganized.

DWS: But it works here as well. We had lots of chances to reorganize and decided on this structure.

Section 12.10
An introduction to this section would be very helpful to understand how srandom function ties to rand variables and constraints.
In fact it in only after you have finished reading the last part of the chapter that you understand how RNG are initialized and why.
I suggest to reorganize 12.10, 12.11 and 12.12

DWS: I think the order is fine. srandom does NOT have to be used to set the seed for urandom. There have been another clarification (LRM-102) that makes 12.10.1 clearer. Leave as is for now.

Especially we should introduce here the notion of a RNG for an object class and for a thread (I still don't know what is a thread). Then how RNG are initialized, and what are the properties of values generated
by srandom, randomize methods.

DWS: Some overview would be helpful. Do not have it.

Section 12.10.2
If one argument is omitted, we should still have the, to be consistent with how functions and tasks can have default arguments.

DWS: What?

Replace:
If *minval* is omitted, the function shall returns a value in the range *maxval* .. 0.
Example: `val = $urandom_range(7);`

With: val = $urandom_range(7,);

DWS: The , is NOT required as defined for optional arguments. It would be silly to add it.

The following is silly: I am against it; arguments should be matched by position or name.

If *maxval* is less than *minval*, the arguments are automatically reversed so that the first argument is larger than the second argument.
Example: `val = $urandom_range(0,7);`

DWS: This is as we voted on.

Section 12.10.3
Q: I don't understand the following sentence. Is the user require to call srandom(1) in each
program block when he wants to use constraint randomization?
The **$srandom()** system task initializes the local random number generator using the value of the given seed.
The optional object argument is used to seed an object instead of the current process (thread).
The top level randomizer of each program is initialized with `$srandom(1)` prior to any randomization calls.

DWS: No, he can if he wants to control the seeding. Otherwise it is done for him.

Q: what is the top level randomizer?

DWS: The randomizer in the top level context.

Q: what is a process thread?

DWS: The thread of the process containing the code.

Q: what is the local RNG?

DWS: The Random Number Generator local to the class.

Section 12.11 Random stability

The Random Number Generator (RNG) is localized to threads and objects. Because the stream of random values returned by a thread or object is independent of the RNG in other threads or objects, this property is called *random stability*.

Q: what does the first sentence means? that we have an implicit RNG per object and per thread?

DWS: No. It means the sequence of random numbers generated is different for each object and thread.

Q: do we have a RNG for an object class if there is no rand variables in that class?

DWS: Maybe but what difference does it make since it does nothing.

Q: What is a thread?

DWS: Term is used many places in the LRM to refer to threads of execution of concurrent processes.

Q: how can a class object return a stream of random values? (see sentence above)

DWS: Sequence would be better than stream. Fixed in LRM_227.

Q: what does this paragraph tries to say? that the srandom, randomize are stable random methods which are independent of each other call?

DWS: Something like that.

Section 12.11.1 Random stability properties
This section is quite obscure.

— Thread stability
Each thread has an independent RNG for all randomization system calls invoked from that thread. When a new thread is created, its RNG is seeded with the next random value from its parent thread. This property is called “hierarchical seeding.”

Q: how do you create a thread? what is a parent Thread?

DWS: Each program block creates a new thread, for example, forks are another. Parent process.

Object stability is guaranteed as long as object and thread creation, as well as random number generation, are done in the same order as before. In order to maintain random number stability, new objects, threads and random numbers can be created after existing objects are created.

Q: what is meant by "in the same order as before"

DWS: Order of the sequence.

Q: what is object stability?

DWS: Stability of the sequence generation with respect to the object (as opposed to the thread).

— Manual seeding

Q: does manual seeding guarantee random stability?

DWS: Nope. It is under the user's control.

Section 12.11.2 Thread stability

I don't understand what is the parent thread in the example.

DWS: The thread the fork was invoked from.

Hierarchical seeding. When a thread is created, its random state is initialized using the next random value from the parent thread as a seed. The three forked threads are all seeded from the parent thread.
Q: where is the parent thread. I don't see any srandom or urandom before the fork.

DWS: See above.

Each thread is seeded with a unique value, determined solely by its parent. The root of a thread execution subtree determines the random seeding of its children. This allows entire subtrees to be moved, and preserve their behavior by manually seeding their root thread.
Q: how are entire sub-trees moved? where/ How do you preserve the behaviour?

DWS: Someone changing how threads are invoked by changing code.

Do you mean the random number generated are always the same for multiple runs of the design?
Where is the root thread?

DWS: The root of the tree of thread instances.

Section 12.11.3 Object stability
Rewrite :(grammar incorrect)
This is the property that calls to **randomize()** in one instance are independent of calls to **randomize()** in other instances, and independent of calls to other randomize functions.
To:
This is the property which means that calls to ...

DWS: I think the grammar is actually correct.

Q: which other randomize functions?

DWS: Ones not in the instance of the class or other instances of the class.

Each instance has a unique source of random values that can be seeded independently.

That random seed is taken from the parent thread when the instance is created.
Q: what is the parent thread. Do you mean that there will be a srandom call in the program block which instantiated the object?

DWS: srandom will be in some hieararchy of threads of concurrent processes. It may be a program block is the parent.

Section 12.12:
An example of seeding the RNG externally is:
```
Packet p = new(200); // Create p with seed 200.
$srandom(300, p); // Re-seed p with seed 300.
```

Q: Is this external seeding also called hierarchical seeding?

DWS: No since it was not seeded from the parent threed but from a manual seed.