Here are my review comments for the class chapter.

Main general comment:
This entire chapter looks more like a user manual or tutorial guiding the user to the use of classes rather than a Language Reference standard. (may be because it was extracted from Vera).The chapter needs to be rewritten in a more formal language so that it does not stands out from the rest of the document.

DWS: This has been a continuing comment. There is nothing to do about it at this point. Changes have been made, incrementally, to improve this.

Section 11.1:
SystemVerilog introduces the object-oriented **class** framework.

Q: The notion of framework is kind of foreign to Verilog.
Why not replacing this with:
SystemVerilog introduces an object-oriented class type system.
since classes are just a new type.

DWS: Done in LRM-201

Classes allow objects to be dynamically created and deleted, to be assigned, and to be accessed via handles, which provide a safe pointer-like mechanism to the language.

The sentence above is grammatically incorrect and should be split into 2 sentences.

DWS: Done in LRM-201

I would replace the sentence below (removing the framework concept):
With inheritance and abstract classes, this framework brings the advantages of C function pointers with none of the type-safety problems, thus, bringing true polymorphism into Verilog.
with:
Classes offer inheritance and abstract type modeling which brings the advantages....

DWS: Done in LRM-201

Section 11.3:
Everywhere in Verilog we talk about tasks and functions, we should be consistent with the terms and not use different ones such as subroutines.
A **class** is a type that includes data and subroutines that operate on that data.
I suggest to replace subroutines with functions, unless we describe them as methods of the class. (method is also used in the data type chapter to refer to methods of the type string or enum types). IN fact the terms are used later, why not introducing them

immediately in the first section.

The term object is used without being defined.
Should add something like:
an object is an instance of a class. The class properties and methods are available to the object.

Page 85 (example)
The data and methods portions should be delimited.
Sometimes the comment is on the first line, sometimes it is below.

```
class Packet ;
bit [3:0] command; // data portion
bit [40:0] address;
bit [4:0] master_id;
integer time_requested;
integer time_issued;
integer status;
function new(); // initialization
command = IDLE;
address = 41'b0;
master_id = 5'bx;
endfunction
task clean();
command = 0; address = 0; master_id = 5'bx;
endtask
// public access entry points
task issue_request( int delay );
// send request to bus
endtask
```

Q: Are functions defined in a class following the same syntax as regular SystemVerilog functions?
   If that is true, function new(); is not legal Verilog it should be function new;
Q: What is the return data type of function new? Shouldn't it be explicit?

Q: Can the task or functions inside a class be specified as automatic?
Q: if the mode is not specified, is it INPUT by default?
Q: can you have no ANSI C formal arguments? (old style funcitons and tasks)?
In other words are funcitons and tasks declared in classes following the same rules as tasks
and functions in Verilog?

Q: are there any restrictions on functions and tasks declared in a class?
Q: how do you specify the return value of a function?
    IN Verilog you either assign the return value to the name of the function or you use a return
statement. Does this rule apply here?
Q: is there any semantic restrictions on what the task/function can do?

Section 11.4
The first paragraph needs to be rewritten in a standard LRM form
Replace:

The previous section only provided the definition of a class *Packet*. That is a new, complex data type, but one
can't do anything with the class itself. First, one needs to create an instance of the class, a single `Packet`
object. The first step is to create a variable that can hold an object handle:
`Packet p;`
Nothing has been created yet. The declaration of *p* is simply a variable that can hold a handle of a *Packet*
object. For *p* to refer to something, an instance of the class must be created using the **new** function.
`Packet p;`
`p = `**`new`**`;`

With:
A class defines a data type, an instance of that class is called an object and can be created by first declaring a variable of that class type and then allocating an object of that class and assigning it
to the variable.
Packet p; // declare a variable of class Packet

p = new ; // initialize the variable to an new allocated object of the class Packet.

The variable p is said to hold a handle to an object of class Packet.

DWS: Done in LRM-204 (with some rewording)

Add a statement that Variables of class type are not static but dynamic.
Add a statement indicating which kind of Verilog things can be of a class type.
(regs, vars, parameters?....)

DWS: TBD

The sentence:
Accessing non-static members or virtual methods via a null object handle is illegal. The result of an illegal
access via a null object is indeterminate, and implementations can issue an error.
EC-CH104
introduces for the first time "non-static members and virtual methods". These terms need to
be defined and described in the previous section describing the class data type.

DWS: The LRM does many forward references. This is just another one.

The paragraph and table 11.2 below should be informative (may not be embedded here
and may be inserted at the end of this chapter) At least that way I think it would
not break the class chapter flow description, which I feel it does disrupt right now.

SystemVerilog objects are referenced using an "object handle". There are some differences between a C
pointer and a SystemVerilog object handle. C pointers give programmers a lot of latitude in how a pointer may
be used. The rules governing the usage of SystemVerilog object handles are much more restrictive. A C
pointer can be incremented for example, but a SystemVerilog object handle cannot. In addition to
object handles, Section 3.7 introduces the **chandle** data type for use with the DPI Direct
Programming Interface (see Section 26).

DWS: I will leave it where it is at this point. It is at the end of the section. It does
document the difference between object handles and chandle (by forward reference
again).

The last column of the table 11-1 and 11-2 should be logically their first column (to help reading the table).

DWS: I cannot find Table 11-1. I will reorder the table. Done in LRM-205

Section 11.6:
The whole paragraph should be removed:

Note that we did not say:

```
status = current_status(p);
```

The focus in object-oriented programming is the object, in this case the packet, not the function call. Also,
objects are self-contained, with their own methods for manipulating their own properties. So the object doesn't have to be passed as an argument to `current_status()`. A class' properties are freely and broadly available to the methods of the class, but each method only accesses the properties associated with its object, i.e., its instance.

The concept of a method working on a data type has already been introduced (strings, enums). Or if we want to leave something of this effect, we should introduce it earlier as a characteristic to a data type and forward refer the class chapter.

DWS: In the discussion in the SV-EC committee about this we decided to leave it in as help to reader's that might not be completely familiar with object oriented methodology. I will choose to leave it. It reminds me of much of the 1364-2001 LRM where examples and discussions seem prevalent (unlike 1076.1 which is concise and terse).

Page 88:
In the course of creating this instance, the **new** function is invoked,
in which any specialized initialization required may be done. The new task is also called the class constructor.

Is new a task or a function?

DWS: Fixed in LRM-206

The **new** operation is defined as a **function** with no return type, thus, it must be nonblocking. Even though
new does not specify a return type, the left-hand side of the assignment determines the return type.

In Verilog a function cannot block, the "it must be nonblocking" should be removed.

DWS: Why remove it? It is not incorrect just highlighting a "truth".

Q: isn't the new function always creating an object of the embedding class type? Realistically
it has a return type even if it is not specified.

DWS: True but there is no return type (nor should there be) specified in the declaration. Constructors are unique in C++ and SystemVerilog in that they are like functions but have special semantics and behavior.

At the bottom of 11.7:
The conventions for arguments are the same as for procedural subroutine calls, including

the use of default
arguments.

Replace procedural subroutine with function/task.
Q: does it also include passing by name and position?

DWS: Changed the phrase from "including" to "such as" since the statement says it uses the same conventions as for other procedural subroutine (as defined earlier) calls. The term procedural is used to distinguish it from class subroutines.

Section 11.8
A formal definition of what can be a class property needs to be provided.
Which type are allowed?
Can you declare event type properties, reg properties, wires?

DWS: The BNF clearly defines the list of items as data_declaration which does not include net_declaration. The discussion in 11.8 refers to variables (which do not include nets). The BNF when added to 11.2 will include the list of legal items. I agree a more formal description here would be helpful but the information does exist in the Section and the BNF.

If possible, I suggest changing the terminology from class property to class data members, because it is strange to denote an event variable as a property.

DWS: Good suggestion for future. A little too late for this kind of change in this version. Class properties are also a viable alternative (more in line with object oriented usage).

I would designate the class task and functions as class behavioral members. (class methods is also okay).

DWS: Class methods is how it is actually defined (with the term class being optional when the context is obvious).

Section 11.8.1
This section should not be under class properties. It should be a parallel section or a section under class methods.

DWS: Agreed. This was supposed to be 11.9. Oops. Fixed in LRM-208

Remove the use of subroutine.

DWS: The term subroutine is appropriate here as defined above.

End of the section:
A static method is different from a method with static lifetime. The former refers to the lifetime of the method

within the class, while the latter refers to the lifetime of the arguments and variables within the task.

```
class TwoTasks;
static task foo(); ... endtask // static class method with
// automatic variable lifetime
task static bar(); ... endtask // non-static class method with
// static variable lifetime
endclass
```

 The bnf for tasks and functions or class_item does NOT allow the static keyword before the task and function keyword.

DWS: Then the BNF is wrong. It needs to be supported both before and after with different meaning. Opened LRM-216

Section 11.9: a more formal definition for "this" should be there, rather than an informal description.
For example:
"this" is a reserved identifier which denote the current instantiated object.

DWS: I would agree. I have updated the description to give a better definition. More could still be done. This is done in LRM-209.

Q: is "this" a reserved word inside the class context?

DWS: Yes and no. It is a predefined class property (of a special type) similar to new being a predefined class method (of a special type).

Section 11.10:
A more formal definition would be nice.

DWS: Yes it would ☺.

Section 11.11
The title refers to inheritance and sub-classes but the text does not define which syntax specifies inheritance and which one specifies sub-classes.
A more clear definition of how to create a sub-class of a class is required.
A clear description on how to specify inheritance is required.
I suggest to rewrite this section and divide it into 2 sub-sections: sub-classes, inheritance.
And add the informative note that both provide ways to specialize classes into more specific
data types.

DWS: This does not make sense. The sentence "… extend the class creating a new subclass that inherits the members …" indicates there is only one action here. Extending a class. The terms inheritance and sub-classes are both related to this, as indicated in the text.

This section should define the terms of derived and parent classes, and specialized classes as these terms are used later but not defined.

DWS: These terms are standard object oriented methodology defined in many places. Do we need to define all English language terms used in the LRM? I know these are more specific than generic English language but so are many other terms in the language. I will live this for a future effort since the meanings are well defined.

Section 11.13 Super

The **super** keyword is used from within a derived class to refer to properties of the parent class. It is necessary
to use **super** to access properties of a parent class when those properties are overridden by the derived class.

Should say "properties and function and tasks members"
"Derived class" is not a defined term.

DWS: Derived class is well defined in the art. LRM_192 changed properties to members.

Replace:
The property may be a member declared a level up or a member inherited by the class one level up.

With:
The property or task or function referred to with the super keyword must be a a member declared a level up or a member inherited by the class one level up.

DWS: The proper wording is "The member may be declared a level up or be inherited by the class one level up". This is changed in LRM-210.

Add to the following paragraph that subclasses are also called derived classes.
Super classes are also called parent classes.

Subclasses are classes that are extensions of the current class. Whereas superclasses are classes that the *current* class is extended from, beginning with the original base class.

DWS: Modified the previous paragraph to define derived, base, and sub classes in LRM_211.

Section 11.14: casting
However, it may be legal to place the contents of the superclass handle in a subclass variable.

Q: is it legal? or not? What does "may" means?

Section 11.15:
Replace:
When a subclass is instantiated, one of the system's first actions is to invoke the class method **new().** The first implicit action **new()** takes is to invoke the **new()** method of its superclass, and so on up the inheritance hierarchy.
With:
When a subclass is instantiated it first invoke the class method new(). The first implicit or explicit statement in the method new should be to invoke the new method of its superclass and so ...

In the following sentence, the term base class is used but not defined.
Thus, all the constructors are called, in the proper order, beginning with the base class and ending with the current class.

The reminder of the section needs to be rewritten in a more formal way. It is also possible to move the initialization of the new arguments to the section which introduce class inheritance. The section which describes extends class_type should also mention that arguments can be passed to the new constructor through the extend syntax.

Section 11.16 hiding and encapsulation
In SystemVerilog, unlabeled properties and methods are public, available to anyone who has access to the
object's name.

A label has a specific meaning in Verilog, do not use "unlabeled" to mean that the property/method
has no specific scope. I would suggest to change unlabled to unqualified.

Section 11.17: Constant properties
First paragraph:
Global constants and Instance constants.
Q: Why capitalizing G and I?

What is the meaning of the following paragraph ? Can I write static const int size = 0;

Note that the BNF does not allow it.

Typically, global constants are also declared **static** since they are the same for all instances of the class.
However, an instance constant cannot be declared **static**, since that would disallow all assignments in the
constructor.

DWS: Clearly this is what we voted to support. I have asked Stefen to fix the BNF. LRM_215

Section 11.18:

The information in this section is valuable but does not look like a standard. This section needs to be formalized and complemented by an example rather than walking the reader through a tutorial. A formal definition of a virtual class and virtual method is needed.

DWS: Agree but not possible today. It is accurate and useful as is. The BNF provides the syntax.

remove "subroutine"

DWS: See earlier discussion on subroutine.

Q: Can a method declared as virtual have a body? If so, then why is it declared as virtual? This is not clear from this section.

DWS: The following indicates that only the prototype is specified and not the body. The BNF also clarifies.
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.
This means that a virtual cannot have a body. This should show up in the BNF as well.

Different size font used for specifying it can be made *abstract* by specifying,

DWS: Fixed in LRM-217

Section 11.19
Replace: (it sounds like superclass is a keyword)
Polymorphism allows one to use superclass to hold subclass objects, and to reference the methods of those
subclasses directly from the superclass variable.

With:
Polymorphism allows one to use a superclass instance to hold subclass objects. Methods of those

sub-classes are directly referenced from the superclass variable. The compiler performs runtime dynamic lookup.

DWS: Fixed in LRM-218

Section 11.20
Q: is a nested class the same as a sub-class?

DWS: No. A nested class is defined within a class while a sub-class is derived.

Q: when the embedding class is instantiated, are all the nested classes instantiated?

DWS: Clarification from Arturo requested.

Section 11.21
any *attributes* (**local**, **protected**, **public**, or **virtual**)

I think that these were called hiding encapsulation qualifiers? Do not use the word attribute as it has a special meaning in Verilog.

DWS: Done ion LRM-210

This section needs to be rewritten in a more formal way.

DWS: Formal is good but non-formal is ok.

Section 11.23
typedef class

Q: Is it required to have the keyword class in order to refer to a class type?
Section 3.10 does not say anything about it while this section says it is optional.
(see end of section 11.23)

in 3.10:

typedef foo; // forward type declaration
foo f = 1;
typedef int foo;

in section 11.23:
This is resolved using **typedef** to provide a forward declaration for the second class:
typedef class C2; // C2 is declared to be of type class
class C1
C2 c;
endclass
Note that the class keyword in the statement typedef class C2; is not necessary, and is used only for documentation purposes. The statement typedef C2; is equivalent and will work the same way.

DWS: In Section 3.10 the following is given which is consistent with 11.23 and the BNF

```
typedef type_declaration_identifier;
```

Section 11.24:
Bullet 2
2) SystemVerilog structs are type compatible so long as their bit sizes are the same, thus copying structs
of different composition but equal sizes is allowed. In contrast, SystemVerilog objects are strictly
strongly-typed. Copying an object of one type onto an object of another is not allowed.

What about polymorphism? it allows to assign a derived class to a parent class.

DWS: A parent and derived class are not different types as implied in the last sentence above. They are related which allows for copying. Standard object oriented terminology.

Replace framework with type system.

DWS: Done in LRM-220