Chapter 11, SV 3.1 draft2 notes (v2)

Section 11

a) (Brad)
This section might read better if it were rewritten without use of the pronoun 'one'.

This must be covered in each individual section of chapter 11. Suggestion is to include the sentences/paragraphs in text and provide alternative sentences for each sections.

Section 11.1

a)(Neil)
Delete the first two sentences of this chapter, starting with
"System Verilog 3.0..."

noted in CH-104

Section 11.3

a)(Brad)
"A class is a collection" ???
Isn't it more a category/type of object?

noted in CH-104

b) "A class's data is referred to as properties" ???
A class defines the common properties of a category of object.

The original text is much cleaner.

Section 11.4

a) (Chris Spear)
What happens if a user tries to access an object's members but the handle is null? Should the simulator crash like a C program with a bad pointer? Should a read of a variable using a null handle return a default value? What if a method is called using a null pointer? Anything other than a segfault is going to cost performance. But the user needs a way to debug this very common problem.

noted in CH-104

Two issues: Language definition and Debugging capability.
The first is covered by CH-104 and the latter is not part of the language

b) (Brad)
"The last section" --> "The previous section"

noted in CH-106

Section 11.7

a) (Chris Spear)

The example at the bottom of 75 has a variable called time which is a keyword.

noted in CH-104

Section 11.8

a) (Chris Spear)
Does a class have to be instantiated before its static properties can be accessed? The example seems to show this. In general, will SystemVerilog provide a way to access static properties outside of objects, like C++? Are we going to support Class::myStaticVar ?

Two items discussed above:
1) Accessing static properties
2) Support for C++-like name scope operator to access members.

1) The donation has no syntax to access members in any way other than via an object. Thus, the only way to access members (static or otherwise) is via an object. Note that a null object can be used to access static members.
2) The donation does not provide support for the name scope operator. It is used only to declare out of body methods.

NOTE: Enhancing classes to support the '::' operator in this manner does not represent any technical problems and would be a fine addition that would allow for access to static members and enable declaration of static methods that could be called in this same fashion (like C++).

AI-56 relates to this as well: write a proposal to include :: in SV.

b) (Neil)
The examples and the text appear to be inconsistent in their usage of variable names. There is a mixture of the use of "fileId" and "semId". I assume that all 3 should be fileId.

noted in CH-104

Section 11.9

a) (Neil)
In the example: endfunction and endclass should both be in bold.

noted in CH-104

Section 11.10

a) (Neil)
Page 77, 3rd paragraph that begins with "This statement has new executing twice, thus creating two objects". The example being referred to is

```
p2 = new p1;
```

This doesn't sound right. Doesn't this just create the p2 handle, which is then initialized to a shallow copy of what the p1 handle refers to?

noted in CH-104

The text is correct. Calling **new** a second time creates a new object that is initialized with a 'shallow copy' of p1's contents. In C++ parlance:
The syntax 'obj = new;' is the constructor.
The syntax `obj = new obj;' is the copy constructor.

b) (Francoise)
- inconsistency or error in the use of new:
p2 = new p1
why not
p2 = new(p1);
The notation above will match the dynamic allocation of dynamic arrays: arr = new(src_array);
"new" is a function and should always require the ().

The constructor **new** can take optional arguments, and those are specified in parenthesis (). The copy constructor accepts no arguments, only an object of the same type as the left-hand side, and it is different from the dynamic array optional argument. The syntax is different for all three. See comments above.

c) (Brad)
"re-naming" --> "renaming"

noted in CH-104

d) (Brad)
There seems to be a contradiction between the comments in the example task declaration of "test" and the claim that "This statement has new executing twice, thus creating two objects, p1 and p2."

noted in CH-104

Section 11.12

a) (Brad)
I would omit the final sentence. Most C++ programmers avoid multiple inheritance, except maybe of abstract base classes (in the style of Java interfaces).

C++ programmers may or may not use multiple inheritance, which is irrelevant to this LRM.

b) (Kevin)
Still see no need for the final sentence.

noted in CH-106

Section 11.13

a) (Neil)
The first paragraph of this section reads:
[This is also related to AI-58, see change in red below.]
"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."

Is the property of the parent class that is now hidden. The following re-write of the second sentence is suggested instead:

"It is necessary to use super to access the parent class properties when the property of the derived class has been overridden."

AI-58: Change the second sentence of first paragraph to, noted in CH-106.

It is necessary to use super to access properties of a parent class when those properties are overridden by the derived class.

b) (Brad)
"super.super.count is not allowed" ??? Why not?

It is not allowed because an object should not rely on the implementation or existence of any object other than its immediate parent class. This avoids problems associated with changes to the parent implementation in which super.super may no longer exist. Members of a grandparent class can always be accessed using a cast to the corresponding super-class.

c) (Kevin)
The "note:" about "super.new" points out a flaw in the syntax. C++ syntax puts parent constructor calls in the constructor declaration:

```
#include <stdio.h>
        class foo              { public: int a;   foo() {a = 2;}};
        class bar : public foo { public: bar() : foo () { a *= a;}};
           main() {bar b; printf("%d\n",b.a);} // prints 4
```

I would suggest using the C++ syntax instead - which makes "super" unnecessary as a keyword (if you use '::' as well).

There is no flaw in the syntax. It's just that the two languages use different mechanisms. Note that SystemVerilog has the same restriction with regards to data declarations and statements: all declarations must precede the statements. A restriction not shared by C++.

d) (Brad)
"super-class" --> "superclass"

change as indicated in the second to last paragraph, the last sentence

Section 11.14

a) (Neil)
First sentence of first paragraph. Add the word "a" as shown below:

noted in CH-104

b) (Neil)
The word "scalar" is used in several places. Based on the discussions for section 3.14 these should all be changed to the word "singular".

noted in CH-104

c) (Neil)
There are task and function forms of this subroutine. The last two paragraphs on page 79 refer to both of them as functions. The text of both paragraphs needs to be cleaned up to not imply that there are two forms of the same function.

noted in CH-106.

d) (Neil)
Second to last paragraph on page 79, states that a fatal error will occur. See the new text from EC-CH28 that should go here.

noted in CH-104

e) (Neil)
First paragraph, page 80, last sentence says, "Otherwise, it sets the destination handle to null and returns 0.". The text from EC-CH29 goes here instead.

noted in CH-104

f) (Brad)
"assign subclass" --> "assign a subclass"
In "handle to null", 'null' should be bold.

noted in CH-104

g) (Kevin)
$cast - What's this for? Isn't static checking sufficient, and it doesn't look user-replaceable to me.

This is equivalent to **dynamic_cast** in C++. One big difference is that in C++ an illegal cast always results in an exception being raised, whereas the **$cast** function form allows checking if the cast will succeed. Below is a copy of an older message that explains the need for dynamic casting.

A derived-class can be assigned directly to any of its super-classes. However, a super-class can only be assigned to a derived class if and only if the sub-class is actually of that type. In general, this can only be resolved at run-time, thus the need for a dynamic cast. For example:

```
class Animal { ... }
class Mammal extends Animal { ... }
class Dog extends Mammal { ... }
class Cat extends Mammal { ... }

Mammal m = new;
Animal a = m;           // correct, mammal is an animal
Dog d = m;              // error: m is not a dog
Cat c = d;              // error: d is a dog not a cat
$cast( d, m );          // allowed: triggers a run-time error
if( ! $cast ( d, m ) )  // user can check at run-time
  d = new;
```

Section 11.15

a) (Kevin)
See my comment in 11.13.

Same answer as above.

Section 11.16

a) (Neil)
The summary at the end of this section should be completely removed. This type of stuff belongs in a tutorial.

noted in CH-104

b) (Brad)
"However, for most data (and subroutines) one wants to hide them from the outside world."
This sentence is oddly phrased.

noted in CH-104

c) (Brad)
"other.i" (twice) and "this.i" should be in typewriter font

The text is correct: **this** is a keyword, other is not.

d) (Brad)
Why are the defaults of the language the converse of the advice in the summary? Why isn't 'local' or 'protected'the default?
Why don't we need a special label to make properties and methods public?

The default is public. No change is needed.

e) (Francoise)
local is the keyword used to mean private data. Why not stay with C++ and use private?

This is a possibility. It should be put to a vote.

f) (Kevin)
The statement that local variables don't get inherited appears incorrect if you can access them through superclass methods.

The text is correct: local variables are not inherited. They are local (i.e., private) to the parent class, hence, they can be accessed by methods of the parent class.

Section 11.17

a) (Chris Spear)
This needs to be a little more clear. Specify that only one assignment can be done at run time. The current wording would seem to forbid:

```
const int size;
function new();
      if (flag) size = 1; // First assignment
      else      size = 2; // second assignment
```

endfunction

The text is correct. Only one assignment in the constructor is allowed.
The example above is correct SystemVerilog code since only one assignment will take place. This can be detected with simple data flow or with a runtime check, the language doesn't specify which.
Note that the example could be easily re-written as
size = flag ? 1 : 2;
and, that makes it clear that one assignment takes place.

b) (Kevin)
It's unnecessarily difficult to check the "only written once" rule for instance constants, it should be relaxed to "can only be written to in the constructor(s)".

See above description. If the rule is relaxed as suggested then the member is not really a constant for a particular object. The objective is not to make it easier for the compiler writers, but to make it easier and safer for the users.

Section 11.18

a) (Brad)
"it can be declared to be abstract by declaring the class to be virtual" ??? Why not declare it 'abstract' then?

That requires another keyword, instead we reuse 'virtual'.

b) (Brad)
Can an abstract class have nonvirtual methods? If not, why do we need to declare the methods to be virtual? If so, what does it mean for an abstract class to have a real method?

An abstract class can have non-virtual methods.

noted in CH-106

c) (Brad)
"Methods of normal classes can also be declared virtual."
Yet there's nothing 'virtual' about them.
Unlike the virtual methods in abstract classes, they are not just prototypes.

A normal (non-abstract) class can also contain virtual methods, but it must provide an implementation for those methods. Note that an abstract class can be derived by another abstract class.

d) (Chris Spear)
Page 82, the third paragraph, second sentence should be changed as follows. The new wording is in quotes: In general, if an abstract class has "any" virtual methods, all of the methods must be overridden for

the subclass to be instantiated. If all of the "virtual" methods are not overridden, the subclass needs to be abstract.

noted in CH-104

e) (Neil)
First paragraph, page 82, second sentence reads: "Since the base class doesn't need to instantiate the base class, it can be declared to be abstract...".

This needs to be re-worded. Something like the following is suggested:

> "Since the base class is not intended to be instantiated, it can be made abstract by specifying the class to be virtual."

noted in CH-104

f) (Brad)
"super-class" (twice) --> "superclass"

change as indicated

> The word “superclass” will be used, however the sentence is omitted per g) below. This will be added as an item in section 13.13 since the hyphenated item appears there.

g) (Brad)
The final sentence could be omitted.

noted in CH-104

g) (Kevin)
It's not clear to me what you gain by having a "virtual class", C++ lets you define virtual methods as being null for either a superclass or a subclass, it's only an error if you try to use such a method, that allows subclasses to implement subsets of functionality without having to create a bunch of dummy methods (maybe with assertions in them). I'd suggest not using "virtual class" and allowing '= null' as an alternative to the method body.

> That statement is only partially correct. C++ does allow methods to be defined as null, but the statement about it only being “an error if you try to use such a method” is incorrect. In C++ it is an error to attempt to create an object of an abstract class (same as for SystemVerilog). The C++ syntax that specifies each abstract method using “= 0” is neither informative nor convenient. Being able to specify the class as abstract is simpler and clearer.
> A historical note: Stroustrup’s original intent was to add the “abstract class …” syntax to C++ . But, since he couldn’t get the additional “abstract” keyword through the ANSI/ISO committee, he settled for the inferior work-around of adding an “= 0” to each unspecified method. Let’s not make the same mistake as that committee, especially since we are not adding new keywords.

Section 11.19

a) (Kevin)
This appears to be only informative - is it necessary?

Yes, it's informative. But, it is important to readers that have not been exposed to other object-oriented languages.

Section 11.20

a) (Chris Spear)
The last line should be changed to the less ambiguous: "The out of block method declaration must match the actual method's declaration statement."

noted in CH-104

b) (Chris Spear)
Even this is not perfect. Can we state this in terms of BNF ?

No. BNF is not adequate for comparing syntax.
This will have to be a semantic check.

c) (Chris Spear)
Second, do the argument names have to match? Is the following legal?

```
class Packet;
      int a;
      extern function new(int b);
endclass
function Packet::new(int c);
      a = c;
endfunction
```

That is illegal. The arguments must match exactly.

noted in CH-104 : now included verbiage which states so explicitly.

d) (Neil)
Delete the first paragraph.

noted in CH-104

f) (Neil)
First sentence of second paragraph reads: "To make this practical, it is best to move long method definitions.." The following is suggested in place of this: "It is convenient to be able to move method definitions..."

noted in CH-104

g) (Neil)
The keyword public is shown. Do we really need this keyword? I suggest that we get rid of the keyword public. None of the examples in the document use it. The default is for class properties to be "public" do we really need to reserve this keyword so that we can explicitly state that we really want something to be public?

It's a good point. We can put this to a vote.

h)(Francoise)
The syntax class_name:: function_name to describe a function outside of its class is not very verilog-like. Why not keeping the out of module syntax with the . like it is currently used in Verilog to refer to an out of scope reference. class_name.function_name. The function name being defined as an extern in the class definition.

This is done to explicitly disambiguate class scopes from hierarchical (module) scopes, and differentiate between an XMR (or OUMR) from the out-of-body class declaration, provide consistency with C++, and to disallow out-of-module function declarations. If the class scope operator "::" is added to allow accessing static members and calling static methods then it will be very useful to not overload the '.' for yet another purpose.

i) (Kevin)
First paragraph seems inappropriate for an LRM. It also introduces the '::' operator - which I find preferable to "super".

noted in CH-104

Section 11.21

a) (Brad)
"called a specialization (or variant)" --> "called a specialization"

The text is correct.

b) (Kevin)
Can you use name binding for class template parameters?

Yes. The second example in page 83 shows this:
vector #(.size(2)) vtwo;

Section 11.22

a) (Neil)
The first line of the example at the bottom of page 84 has a font problem.

noted in CH-104

b) (Kevin)
Seems odd syntax - a by-product of using class...endclass rather than class..{..} where you can just skip the {...} for the forward declaration. Too many touch-typers :-)

It is not related to using class…endclass instead of class …{…}. It simply extends the existing SystemVerilog mechanism to provide a forward reference.

Section 11.23

a) (Neil)
Delete the first sentence of the first paragraph.

noted in CH-104

b) (Neil)
Throughout this section there are references to System Verilog 3.1, can't we just say System Verilog and drop all the 3.1 references?

noted in CH-104

c)(Brad)
"When an object is not needed anymore" --> "When an object is no longer needed"

noted in CH-104

d)(Brad)
"The automatic memory management system ..."
This and the rest of the section are advocacy and don't belong in an LRM.

> Maybe it can be reworded, but, I disagree. It belongs in the LRM since it is an integral part of the language. This decision facilitates some things and precludes others (such as sharing pointers across C).

e) (Kevin)
Still think a class without methods should be the same as a struct with syntax and semantics to match, doing anything else will be expensive to fix later (I await Jay's comments). I will strongly disagree with statement (4) until structs, classes, modules and interfaces are capable of cross inheritance.

> This has been discussed. A class is not a struct, module, or interface. Currently, struct's are part of the synthesizable subset of the language and designers like that feature. Classes are not (and are not likely to be) synthesizable.

Section 11.24

a) (Neil)
First example uses fork/join, the "join none" should be "join_none".

noted in CH-89

b) (Kevin)
"shoot themselves in the foot" seems inappropriate (if accurate). The section appears to be informative rather than something required in the LRM, maybe a description of the implementation requirements would be better - when do objects get destroyed exactly? If I pass a reference to an object inside another object does the system keep a copy of the outer object or just the inner object until the reference disappears?

> The implementation specifics are not part of the LRM.
> Automatic memory managers are well covered in many texts.
> The system keeps track of all live objects, that is all objects that are accessible by user code. When objects are no longer accessible they become candidates for reclaiming, which happens at an unspecified implementation-dependent time.

As for the question, if the outer object is no longer accessible but the inner object is still accessible (via the reference passed) then the outer object becomes a candidate for reclamation.

Handles

a) (Neil)
There needs to be a more thorough description of what a handle is. Section 11.23 has the best description but it is inadequate for an LRM. I have written up such a description for an internal Sun document. I have included that write-up here for your consideration. Feel free to correct this write-up if it is in error. This is the level of detail that I believe should be provided.

System Verilog objects are referenced using a "handle". There are some differences between a C pointer and a System Verilog handle. C pointers give programmers a lot of latitude in how a pointer may be used. The rules governing the usage of System Verilog handles are much more restrictive. A C pointer may be incremented for example, but a System Verilog handle may not.

C pointer	SV handle	Operation
Allowed	Not allowed	Arithmetic operations (such as incrementing)
Allowed	Not allowed	For arbitrary data types
Error	Not allowed	Dereference when null
Allowed	Limited	Casting
Allowed	Not allowed	Assignment to an address of a data type
No	Yes	Unreferenced objects are garbage collected
Undefined	null	Default value
(C++)	Allowed	For classes

noted in CH-104