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

Arturo: I believe that change is incorrect. Classes are object-oriented, but they are neither a type system, nor a class type system; they are part of the type system. Classes are an object-oriented framework upon which users can build other abstractions. If there's an objection to "framework", then I suggest "data abstraction": SystemVerilog introduces an object-oriented **class** data abstraction.

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.

Arturo: OK.

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....

Arturo: OK.

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.

DWS: This was discussed in EC. The term subroutine is being used in the generic sense of "routines" that incorporate both tasks and functions. Replacing all uses of subroutines with functions would be inappropriate. I would prefer to add (tasks and functions) in parenthesis after the first use of subroutine to clarify. I would not be adverse to using the term methods in other places where subroutines have been used but it would cause problems in this introduction.
DWS: Done in LRM-202

Arturo: I agree with DWS. Adding (tasks and functions) is appropriate (done in draft 5).

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.

DWS: In 11.3 the term object is not just an instance of a class. It is a conceptually model. It turns out that, with classes, it is an instance of a class. I guess I feel that this term is widely enough defined in the literature to not feel compelled to define it here. The definition you are looking for is actually given in Section 11.4.

FM:I don't see a formal definition of the term object in section 11.4. There is a comparison with SV handles and C pointers.
I agree that a formal definition of what is an object in SV should be in section 11.4
I proposed later to rewrite the section 11.4 in a more formal way, which will define the term object. You have done this; thus, this satisfies this issue as well.

Arturo: I agree with DWS. All that's needed is an advisory sentence in section 11.4, which does introduce the object term, to make it clearer (done in draft 5):

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
```

DWS: Done in LRM-203

Arturo: OK.

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?

DWS: It does not need to be explicit since it is based on the class defined within. This is the same as for C++.

Arturo: Functions defined in classes do follow the same syntax as regular SystemVerilog functions. Methods support additional qualifiers (local, protected, static,…) and extern for out-of-body declarations, but other than that they use the same syntax.
The **new()** function is a special case and doesn't specify a return type. The return type is explicit --- it returns a handle to an object of that class. That is why **new** is a keyword.

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 functions and tasks)?
In other words, are functions and tasks declared in classes following the same rules as tasks and functions in Verilog?

DWS: I believe that the functions used within Classes were not meant to support the non-ANSI-C style prototypes. That is what we discussed in committee.

Arturo:
1. Methods can be specified as static or automatic. The default is automatic:
- This needs to be explicitly specified in the LRM. Add at the end of section 11.9.

2. If the mode is not specified then it is **input** by default (like the rest of SystemVerilog).
3. DWS is correct. Classes should support only the newer ANSI formal argument declaration. Note that out-of-body methods need a complete ANSI prototype in the task declaration, which must match the actual function/task declaration.

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?

DWS: Sent request to Arturo for responses to the above (all of them even the ones I attempted to answer).

Arturo:
1. There are no restrictions on class methods (functions and tasks).
2. A class function follows Verilog rules. The return value can be specified either by an assignment to the function name, or by a return statement.
3. Methods have the same restrictions as regular SystemVerilog tasks and functions. A function may not contain blocking statements.

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)

Arturo: OK. Done in draft 5.

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

Arturo: Sections 11.18 and 11.25 state that classes are dynamic. I don't think additional statements are needed. As for what kind of Verilog "things" a class type may contain, a class may include any data-type, except wires, which are incompatible with dynamically allocated objects (this is described by the BNF). Class parameters are described in section 11.22 (draft 4). This should be stated explicitly in section 11.5 (Object properties). Add the sentence: Any data-type may be declared as a class property, except net types, which are incompatible with dynamically allocated data.

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.

FM: okay, but add these terms in the glossary/index so their forward references can be searched and found easily.

Arturo: Section 11.8 should be renamed – Static properties (not Class properties). Then, the LRM can include a reference in section 11.4: Accessing non-static members (Section 11.8) or virtual methods (Section 11.19) … [And, then both "static properties" and "virtual methods" appear in the index]

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).

Arturo: Even if this appears elsewhere, it is useful as part of the class discussion, so I propose we leave it where it is.

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

Arturo: I agree (done in draft 5).

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).

Arturo: I agree with DWS. The point being made by this paragraph (albeit a bit verbose) is that it's not necessary to pass the object handle as an argument to a class method, and that is different from the other built-in methods discussed before.

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

Arturo: It is a function (typo fixed in draft 5).

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".

FM: The way it is said, I could interpret it as we have non blocking and blocking functions, which is wrong.

I would rewrite as:
The new operation is a non blocking operation defined as a function with no return type.

Arturo: OK. To remove that ambiguity I'd rewrite it as:
The **new** operation is defined as a **function** with no return type, and, like any other function, it must be nonblocking.

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.

Arturo: I agree with DWS. The new method is special and unique, unlike any other regular function (like in C++). There's no need to add the return type.

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.

FM: this denomination of procedural subroutines to only mean task and functions calls but not class methods is NEW to me.
Where is it defined earlier?

Arturo: Add "any other" the sentence:
The conventions for arguments are the same as for any other procedural subroutine calls, such as the use of default arguments.

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.

Arturo: Taken care of in section 11.5 (see my earlier comment).

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).

FM: I really don't like class property applied to Verilog. I think we should change to globally change property to class data members in this version of the standard. It would be too difficult to change it later.

Arturo: "class property" is synonymous with "class data member", only more succinct. The term is defined at the start of the section and it's use is correct, even for Verilog. I don't think this need to change.

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).

Arturo: I agree with DWS.

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

Arturo: OK.

Remove the use of subroutine.

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

Arturo: I agree with DWS.

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

Arturo: DWS is right. BNF needs fixing

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.

Arturo: draft-5 (LRM-209) is good but still contains a small error: **this** is not a class property (it is in the general sense of a property). I propose the following change:

The `this` keyword is used to unambiguously refer to properties or methods of the current instance. The `this` keyword denotes a predefined object handle that refers to the object that was used to invoke the subroutine that `this` is used within. The term **this** may only be used within class methods, otherwise an error shall be issued.

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).

FM: The formal definition of this should say that it is a predefined class data member which refer to the current object instance.

Arturo: **this** is a keyword.
It is not a class property (i.e. a data member), but a built-in object handle that refers to the current object (see definition above) inside a method.
It is an error for **this** to be used outside a class method (or in a static method).

Section 11.10:
A more formal definition would be nice.

DWS: Yes it would ☺.

Arturo: I agree ☺.

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.

FM: I understand what you mean but I was confused by the title and the fact that this section provides 2 ways to creating a subclass: specialization (first example) and sub-class creation through the extend keyword. I was confused as I thought that the first ways provided sub-class creations while the second way was inheritance.

I would rephrase my request as:

I am looking for more formal sentences defining:
. what is specialization (is it only the first example)
. what is a sub-class
. what is inheritance.

For example we could have a definition such as:

"The following statement
class sub-packet extends packet;
creates a sub-class called sub-packet which inherits from
class packet. The inheritance relationship between classes is specified with the extend keyword. It is not allowed to have a sub-class inheriting from multiple parent classes: SV only supports single inheritance."

Arturo: I agree with DWS. There's no need to divide this into more sections.
Here are some useful definitions. Should these be added to the LRM?
A subclass is an extended class (or a derived class). A superclass is the class from which a class is extended.
Inheritance is the object-oriented nomenclature for the act of extending a class. Thus, an extended class inherits its superclass' properties and methods.
In general, a specialization is a special case of a class, such as the ones created by a parameterized class.

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.

FM: I think we need some definitions of the terms and consistency in using them.
There are various object oriented terms used here and some mean the same some don't.
Example:
Is a derived class <=> sub-class?

At least these terms should appear in the glossary and we should choose 1 term only to refer to one thing.

Arturo: Although in general I agree, these terms tend to creep in other places. We might be tempted to change every instance of “subclass” with “extended class”, but that terminology was already defined (it is a section title) and the change may make the text less readable.

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.

FM: Thanks for changing property 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.

Arturo: OK. Fixed by draft 5.

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.

Arturo: OK. Fixed by draft 5.

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?

DWS: Clearly it depends on the contents of the subclass definition. In C++ one of the constraints is due to multiple inheritance. We do not have that here. I have asked Arturo for clarification.

FM: we need the clarification.

Arturo: It is legal to assign a class handle to one of its subclass variable if the object handle refers to an object that IS of that type. Otherwise it is illegal.

I propose the following change to that sentence:

It is always legal to assign a subclass variable to a variable of a class higher in the inheritance tree. It is never legal to directly assign a superclass variable to a variable of one of its subclasses. However, it is legal to assign a superclass handle to a subclass variable if the superclass handle refers to an object of the given subclass.

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 ...

DWS: Reworded in LRM-212

Arturo: OK. Fixed in draft 5.

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.

DWS: Defined in Section 11.13

Arturo: Change that sentence to:

Thus, all the constructors are called, in the proper order, beginning with the root 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.

DWS: No change in this version.

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.

DWS: Done in LRM-213

Arturo: OK. Fixed in draft 5.

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

DWS: Fixed in LRM-214

Arturo: OK. Fixed in draft 5.

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

Arturo: OK. Fixed in draft 5.

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.

Arturo: I agree with DWS.

remove "subroutine"

DWS: See earlier discussion on subroutine.

Arturo: I agree with DWS.

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.

Arturo: I agree with DWS.

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

DWS: Fixed in LRM-217

Arturo: OK.

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

Arturo: OK. Fixed in draft 5.

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.

FM: Thanks for the answer but this is not clear from the draft.
The nested class is introduced in section 11.20 but the title does not say nested classes.
We should change the title from:
Class scope resolution operator ::
to
Class scope resolution operator :: and nested classes

or create a new paragraph for nested classes.

Arturo: No change needed. A class is only one if the types that can be declared inside a class, but not the only one (enums, struct, unions, and other typedefs are also allowed. Nested classes are a special case, but the only one.

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

DWS: Clarification from Arturo requested.

Arturo: No. A nested class is strictly a scoped type definition. To create any objects of the nested classes, their constructors must be called.

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

Arturo: Still need to change “hiding encapsulation qualifiers” to “qualifiers”

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

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

Arturo: I agree with DWS. This section is OK as is.

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 section11.23)

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;
```

Arturo: I agree with DWS. No change needed.

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.

Arturo: I agree with DWS. No change needed.

Replace framework with type system.

DWS: Done in LRM-220

Arturo: Again. “type system” should be replaced with “data abstraction”.