Within Section 1 change (as shown in red):

— Dynamic processes for modeling pipelines
— Definition and support of a compilation unit
— Packages containing declarations such as data, types, classes, tasks and functions
— Separate compilation support
— A compilation-unit scope visible only within a compilation unit
— Removal of the $root global declaration space from SystemVerilog 3.1 ~~A $root top-level hierarchy which can have global definitions~~

Within Section 10.6 change (as shown in red):

To access functions defined in any other scope~~, including `$root`,~~ the foreign code shall have to change DPIcontext appropriately.

Within Section 15.6 change (as shown in red):

Clocking domains cannot be nested. They cannot be declared inside functions, ~~or~~ tasks, packages, or outside all declarations in a compilation unit ~~at the global (`$root`) level~~. Clocking domains can only be declared inside a module, interface or program (see Section 16).

Within Section 16.3 change (as shown in red):

The degree of communication can be controlled by choosing to share data using nested blocks, packages, or hierarchical references ~~(including **`$root`**)~~, or making the data private by declaring it inside the corresponding program block.

Within Section 17.6 change (as shown in red):

A **`sequence`** can be declared in
— a module as a *module_or_generate_item*
— an interface as an *interface_or_generate_item*
— a program as a *non_port_program_item*
— a clocking domain as a *clocking_item*
— a package
— a compilation-unit scope
~~— $root~~

Within Section 17.10 change (as shown in red):

A property can be declared in
— a module as a *module_or_generate_item*
— an interface as an *interface_or_generate_item*
— a program as a *non_port_program_item*
— a clocking domain as a *clocking_item*
— a package
— a compilation-unit scope
~~— $root~~

Within Section 17.12 change (as shown in red):

A concurrent assertion statement can be specified in:
— an always block or initial block as a statement, wherever these blocks can appear\
— a module as a *module_or_generate_item*
— an interface as an *interface_or_generate_item*

— a program as a *non_port_program_item*
— ~~$root~~

Within Section 17.14 change (as shown in red):

The **`bind`** directive can be specified in
— a module as a *module_or_generate_item*
— outside of all other declarations in a compilation unit
— ~~$root~~

Within Section 18.1 change (as shown in red):

— Packages containing declarations such as data, types, classes, tasks and functions
— Separate compilation support
— A compilation-unit scope visible only within a compilation unit ~~A global declaration space, visible to all modules at all levels of hierarchy~~
— Removal of the $root global declaration space from SystemVerilog 3.1

Replace Section 18.2 with the new sections 18.2, 18.3, and 18,4 (renumber succeeding sections and syntax boxes):

18.2 Packages

SystemVerilog packages provide an additional mechanism for sharing parameters, data, type, task, function, sequence, and property declarations amongst multiple SystemVerilog modules, interfaces and programs. Packages are explicitly named scopes appearing at the outermost level of the source text (at the same level as top-level modules and primitives). Types, variables, tasks, functions, sequences, and properties may be declared within a package. Such declarations may be referenced within modules, macromodules, interfaces, programs, and other packages by either import or fully resolved name.

```
package_declaration ::=                 // A.1.3
          { attribute_instance } package package_identifier
          [ timeunits_declaration ] { { attribute_instance  } package_item }
          endpackage [ : package_identifier ]

package_item ::=                        // New A.1.9
           package_or_generate_item_declaration
         | specparam_declaration
         | concurrent_assertion_item_declaration
         | anonymous_program

package_or_generate_item_declaration ::=
            net_declaration
          | data_declaration
          | task_declaration
          | function_declaration
          | dpi_import_export
          | extern_constraint_declaration
          | extern_method_declaration
          | class_declaration
          | parameter_declaration ;
          | local_parameter_declaration

anonymous_program ::= program ; { anonymous_program_item } endprogram

anonymous_program_item ::=
           task_declaration
         | function_declaration
         | class_declaration
```

Syntax 18-1--Package syntax (excerpt from Annex A)

The package declaration creates a scope that contains declarations intended to be shared among one or more compilation units, modules, macromodules, interfaces, or programs. Items within packages are generally type definitions, tasks, and functions. Items within packages cannot have hierarchical references. It is also possible to populate packages with parameters, variables and nets. This may be useful for global items that aren't conveniently passed down through the hierarchy. Variable declaration assignments within the package shall occur before any **`initial`** or **`always`** blocks are started, in the same way as variables declared in a compilation unit or module.

The following is an example of a package:

```
package ComplexPkg;
      typedef struct {
            float i, r;
      } Complex;

      function Complex add(Complex a, b)
            add.r = a.r + b.r;
            add.i = a.i + b.i;
      endfunction

      function Complex mul(Complex a, b)
            mul.r = (a.r * b.r) + (a.i * b.i);
```

```
            mul.i = (a.r * b.i) + (a.i * b.r);
        endfunction
    endpackage : ComplexPkg
```

18.2.1 Referencing data in packages

Packages must exist in order for the items they define to be recognized by the scopes in which they are imported.

One way to use declarations made in a package is to reference them using the scope resolution operator "**::**".

```
ComplexPkg::Complex cout = ComplexPkg::mul(a, b);
```

An alternate method for utilizing package declarations is via the import statement.

```
data_declaration ::=                                    // A.2.3
        [lifetime] variable_declaration
      | constant_declaration
      | type_declaration
      | package_import_declaration

package_import_declaration ::=
      import package_import_item { , package_import_item } ;

package_import_item ::=
        package_identifier :: identifier
      | package_identifier :: *
```

Syntax 18-2--Import syntax (excerpt from Annex A)

The import statement provides direct visibility of identifiers within packages. It allows identifiers declared within packages to be visible within the current scope without a package name qualifier. Hierarchical references to imported identifiers are allowed as if they were defined in the importing scope. Two forms of the import statement are provided: explicit import, and wildcard import. Explicit import allows control over precisely which symbols are imported:

```
import ComplexPkg::Complex;
import ComplexPkg::add;
```

An explicit import is treated like a local declaration. An explicit import shall be illegal if the imported identifier is declared in the same scope or explicitly imported from another package. Importing an identifier from the same package multiple times is allowed.

A wildcard import allows all identifiers declared within a package to be imported provided the identifier is not otherwise defined in the importing scope:

```
import ComplexPkg::*;
```

A wildcard import makes each identifier within the package a candidate for import. Each such identifier is imported only when it is neither declared nor explicitly imported into the scope. Similarly, a wildcard import of an identifier is overridden by a subsequent declaration of the same identifier in the same scope. If the same identifier is wildcard imported into a scope from two different packages, the identifier shall be undefined within that scope and result in an error if the identifier is used.

18.2.2 Search order Rules

Table 18.1 describes the search order rules for the declarations imported from a package. For the purposes of the discussion below, consider the following package declarations:

```
package p;
      typedef enum { FALSE, TRUE } BOOL;
      const BOOL c = FALSE;
endpackage;

package q;
      const int c = 0;
endpackage;
```

Example	Description	Scope containing a local declaration of c	Scope not containing a local declaration of c	Scope contains a declaration of c imported using import q::c	Scope contains a declaration of c imported as import q::*
`u = p::c;` `y = p::TRUE;`	A qualified package identifier is visible in any scope (without the need for an import clause).	**OK**. Direct reference to c refers to the locally declared c. p::c refers to the c in package p.	**OK** Direct reference to c is illegal since it is undefined. p::c refers to the c in package p.	**OK**. Direct reference to c refers to the c imported from q. p::c refers to the c in package p.	**OK**. Direct reference to c refers to the c imported from q. p::c refers to the c in package p.

`import p::*;`
 	
`. . .`
 	
`y = FALSE;`
	All declarations inside package p become potentially directly visible in the importing scope:
	
	• c
	• BOOL
	• FALSE
	• TRUE
		OK.
		
		Direct reference to c refers to the locally declared c.
		
		Direct reference to other identifiers (e.g., FALSE) refer to those implicitly imported from package p.
			OK.
			
			Direct reference to c refers to the c imported from package p.
				OK.
				
				Direct reference to c refers to the c imported from package q.
					OK / ERROR
					
					c is undefined in the importing scope. Thus, a direct reference to c is illegal and results in an error.
					
					The import clause is otherwise allowed.

`import p::c;`
 	
`. . .`
 	
`if( ! c )`
`...`
	The imported identifiers become directly visible in the importing scope:
	
	• c
		ERROR.
		
		It shall be illegal to import an identifier defined in the importing scope.
			OK.
			
			Direct reference to c refers to the c imported from package p.
				ERROR.
				
				It shall be illegal to import an identifier defined in the importing scope.
					OK / ERROR
					
					It shall be illegal to reference c before the import of p::c.
					
					Otherwise, direct reference to c refers to the c imported from package p.

Table 18.1 Scoping Rules for Package Importation

When using a wildcard import, a reference to an undefined identifier that is declared within the package causes that identifier to be imported into the local scope. However, an error results if the same identifier is later declared or explicitly imported. This is shown in the following example:

```
module foo;
       import q::*;
       wire a=c;            // This statement forces the import of
       q::c;
       import p::c;         // The conflict with q::c and p::c
       creates an error.
endmodule;
```

18.3 Compilation Unit Support

SystemVerilog supports separate compilation using compiled units. The following terms and definitions are provided:

— compilation unit: a collection of one or more SystemVerilog source files compiled together
— compilation-unit scope: a scope that is local to the compilation unit. It contains all declarations that lie outside of any other scope
— $unit: the name used to explicitly access the identifiers in the compilation-unit scope

The exact mechanism for defining which files constitute a compilation unit is tool specific. Tools shall provide a mechanism to specify the files that make up a compilation unit. Two extreme cases are:

1. All files make a single compilation unit (in which case the declarations in the compilation-unit scope are accessible anywhere within the design)
2. Each file is a separate compilation unit (in which case the declarations in each compilation-unit scope are accessible only within its corresponding file)

The compilation-unit scope can contain any item that can be defined within a package. These items are in the *compilation-unit scope name space*.

The following items are visible in all compilation units: modules, macromodules, primitives, programs, interfaces, and packages. Items defined in the compilation-unit scope cannot be accessed by name from outside the compilation unit. Access to the items in a compilation-unit scope can be accessed using the PLI, which must provide an iterator to traverse all the compilation units.

In Verilog, compiler directives once seen by a tool apply to all forthcoming source text. This behavior shall be supported within a separately compiled unit; however, compiler directives from one separately compiled unit shall not affect other compilation units. This may result in a difference of behavior between compiling the units separately or as a single compilation unit containing the entire source.

When an identifier is referenced within a scope, SystemVerilog follows the Verilog name search rules:
— First, the nested scope is searched (1364-2001 12.6) (including nested module declarations)
— Next, the compilation-unit scope is searched
— Finally, the instance hierarchy is searched (1364-2001 12.5))

`$unit` is the name of the scope that encompasses a compilation unit. Its purpose is to allow the unambiguous reference to declarations at the outermost level of a compilation unit (i.e., those in the compilation-unit scope). This is done via the same scope resolution operator used to access package items.

For example:

```
bit b;
task foo;
       int b;
       b = 5 + $unit::b;         // $unit::b is the one outside.
endtask
```

The compilation-unit scope allows users to easily share declarations (e.g., types) across the unit of compilation, but without having to declare a package from which the declarations are subsequently imported. Thus, the compilation-unit scope is similar to an implicitly defined anonymous package. Because it has no name, the compilation-unit scope cannot be used with an import statement, and the identifiers declared within the scope are not accessible via hierarchical references. Within a particular compilation unit, however, the special name $unit can be used to explicitly access the declarations of its compilation-unit scope

18.4 Top-level instance

The name `$root` is added to unambiguously refer to a top level instance, or to an instance path starting from the root of the instantiation tree. `$root` is the root of the instantiation tree.

For example:

```
$root.A.B       // item B within top instance A
$root.A.B.C     // item C within instance B within instance A
```

$root allows explicit access to the top of the instantiation tree. This is useful to disambiguate a local path (which takes precedence) from the rooted path. In Verilog, a hierarchical path is ambiguous. For example, A.B.C can mean the local A.B.C or the top-level A.B.C (assuming there is an instance A that contains an instance B at both the top level and in the current module). Verilog addresses that ambiguity by giving priority to the local scope, thereby preventing access to the top level path. $root allows explicit access to the top level in those cases in which the name of the top level module is insufficient to uniquely identify the path.

Within Section 18.3 (current numbering) change (as shown in red):

18.3 Module declarations

SystemVerilog adds the capability to nest module declarations, ~~and to instantiate modules in the `$root` toplevel space, outside of other modules~~.

```
module m1(...); ... endmodule

module m2(...); ... endmodule

module m3(...);

      m1 i1(...); // instantiates the local m1 declared below
      m2 i4(...); // instantiates m2 - no local declaration
      module m1(...); ... endmodule // nested module declaration,
                                    // m1 module name is in m3's
                                 name space
endmodule

~~m1 i2(...);    // module instance in the $root space,~~
             ~~// instantiates the module m1 that is not nested in another module~~
```

Within Section 18.7 (current numbering) change (as shown in red):

There shall be at most one time unit and one time precision for any module, program, package or interface definition, or in ~~`$root`~~ any compilation-unit scope. This shall define a time scope. If specified, the **`timeunit`** and **`timeprecision`** declarations shall precede any other items in the current time scope. The **`timeunit`** and **`timeprecision`** declarations can be repeated as later items, but must match the previous declaration within the current time scope.

If a **`timeunit`** is not specified in the module, program, package or interface definition, then the time unit ~~is~~ shall be determined using the following rules of precedence:

1) If the module or interface definition is nested, then the time unit ~~is~~ shall be inherited from the enclosing module or interface (programs and packages cannot be nested).

2) Else, if a **`'timescale`** directive has been previously specified (within the compilation unit), then the time unit ~~is~~ shall be set to the units of the last **`'timescale`** directive.

3) Else, if the compilation-unit scope specifies ~~`$root` top level has~~ a time unit (outside all other declarations), then the time unit ~~is~~ shall be set to the time units of the compilation unit ~~root module~~.

4) Else, the default time unit ~~is~~ shall be used.

The time unit of the compilation-unit scope may only be set ~~`$root` shall only be determined~~ by a **`timeunit`** declaration, not a **`'timescale`** directive. If it is not specified then the default time unit shall be used.

If a **`timeprecision`** is not specified in the current time scope, then the time precision ~~is~~ shall be determined following the same precedence as with time units.

Within Section 18.10 (current numbering) change (as shown in red):

18.10 Name spaces

SystemVerilog has eight ~~five~~ name spaces for identifiers, two are global (*definitions name space* and *package name space*), two are global to the compilation unit (*compilation unit name space* and *text macro name space*) and four are local. ~~Verilog's global definitions name space collapses onto the module name space and exists as the top-level scope, `$root`. Module, primitive, package, program, and interface identifiers are local to the module name space where there are defined.~~ The eight ~~five~~ name spaces are described as follows:

1) The *definitions name space* unifies all the non-nested **module**, **macromodule**, **primitive**, **program**, and **interface** identifiers defined outside of all other declarations. Once a name is used to define a **module**, **macromodule**, **primitive**, **program**, or **interface** within one compilation unit the name shall not be used again (in any compilation unit) to declare another non-nested **module**, **macromodule**, **primitive**, **program**, or **interface** outside of all other declarations. This is compatible with the *definitions name space* as defined in of IEEE 1364-2001.

2) The *package name space* unifies all the package identifiers defined among all compilation units. Once a name is used to define a package within one compilation unit the name shall not be used again to declare another package within any compilation unit.

3) The *compilation-unit scope name space* exists outside the **`module`**, **`macromodule`**, **`interface`**, **`package`**, **`program`**, and **`primitive`** constructs. It unifies the definitions of the functions, tasks, parameters, named events, net declarations, variable declarations and user defined types within the compilation-unit scope.

4) The *text macro name space* is global within the compilation unit. Since text macro names are introduced and used with a leading ` character, they remain unambiguous with any other name space. The text macro names are defined in the linear order of appearance in the set of input files that make up the compilation unit ~~description of the design unit~~. Subsequent definitions of the same name override the previous definitions for the balance of the input files.

5) The *module name space* is introduced by the **`module`**, **`macromodule`**, **`interface`**, **`package`**, **`program`**, and **`primitive`** constructs. It unifies the definition of **module**, **macromodule**, **interface**, **program**, functions, tasks, named blocks, instance names, parameters, named events, net ~~type of~~ declarations, variable ~~type of~~ declarations, and user defined types within the enclosing construct.

6) The *block name space* is introduced by named or unnamed blocks, the **`specify`**, **`function`**, and **`task`** constructs. It unifies the definitions of the named blocks, functions, tasks, parameters, named events, variable type of declaration and user defined types within the enclosing construct.

7) The *port name space* is introduced by the **`module`**, **`macromodule`**, **`interface`**, **`primitive`**, and **`program`**~~**`, function, and task`**~~ constructs. It provides a means of structurally defining connections between two objects that are in two different name spaces. The connection can be unidirectional (either **`input`** or **`output`**) or bidirectional (**`inout`**). The port name space overlaps the module and the block name spaces. Essentially, the port name space specifies the type of connection between names in different name spaces. The port type of declarations includes **`input`**, **`output`**, and **`inout`**. A port name introduced in the port name space can be

reintroduced in the module name space by declaring a variable or a net with the same name as the port name.

8) The *attribute name space* ~~attribute name space~~ is enclosed by the `(*` and `*)` constructs attached to a language element (see Section 2.8). An attribute name can be defined and used only in the attribute name space. Any other type of name cannot be defined in this name space.

Within Section 18.11 (current numbering) change (as shown in red):

Hierarchical names are also called nested identifiers. They consist of instance names separated by periods, where an instance name can be an array element. The instance name $root refers to the top of the instantiated design and is used to unambiguously gain access to the top of the design.

```
$root.mymodule.u1 // absolute name
u1.struct1.field1 // u1 must be visible locally or above,
including globally
adder1[5].sum
```

Nested identifiers can be read (in expressions), written (in assignments or task/function calls) or triggered off (in event expressions). They can also be used as type, task or function names.

Within Section 21.2 change (as shown in red):

A library is a named collection of cells. A cell is a module, macromodule, primitive, interface, program, package, or configuration. A configuration is a specification of which source files bind to each instance in the design.

Within Section 21.3 change (as shown in red):

~~**21.3 Library map files**~~

~~Verilog 2001 specifies that library declarations, include statements, and config declarations are normally in a mapping file that is read first by a simulator or other software tool. SystemVerilog does not require a special library map file. Instead, the mapping information can be specified in the `$root` top level.~~

Within Section 26.3 change (as shown in red):

Thus the functions imported to and exported from SystemVerilog have their own global name space of linkage names, different from compilation-unit scope ~~$root~~ name space.

Within Section A.1.3 change (as shown in red):

source_text ::= [timeunits_declaration] { description }

description ::=
module_declaration
| udp_declaration
~~| module_root_item~~
~~| statement_or_null~~
| interface_declaration
| program_declaration
| package_declaration

| { attribute_instance } package_item
| { attribute_instance } bind_directive
| { attribute_instance } **;**

….

class_declaration ::=
~~{ attribute_instance }~~ [**virtual**] **class** [lifetime] class_identifier [parameter_port_list]
[**extends** class_identifier] **;** [timeunits_declaration] { class_item }
endclass [**:** class_identifier]

package_declaration ::=
{ attribute_instance } **package** package_identifier
[timeunits_declaration] { { attribute_instance } package_item }
endpackage [**:** package_identifier]

Within Section A.1.5 change (as shown in red):

module_common_item ::=
module_or_generate_item_declaration
| interface_instantiation
| program_instantiation
| concurrent_assertion_item
| bind_directive
| continuous_assign
| net_alias
| initial_construct
| final_construct
| always_construct
| combinational_construct
| latch_construct
| ff_construct
| **;**

module_item ::=
non_generic_port_declaration **;**
| non_port_module_item

module_or_generate_item ::=
{ attribute_instance } parameter_override
~~| { attribute_instance } continuous_assign~~
| { attribute_instance } gate_instantiation
| { attribute_instance } udp_instantiation
| { attribute_instance } module_instantiation
~~| { attribute_instance } initial_construct~~
~~| { attribute_instance } always_construct~~
~~| { attribute_instance } combinational_construct~~
~~| { attribute_instance } latch_construct~~
~~| { attribute_instance } ff_construct~~
~~| { attribute_instance } net_alias~~
~~| { attribute_instance } final_construct~~
| { attribute_instance } module_common_item
~~| { attribute_instance } ;~~

~~module_root_item ::=~~
~~{ attribute_instance } module_instantiation~~

~~| { attribute_instance } local_parameter_declaration~~
~~| interface_declaration~~
~~| program_declaration~~
~~| class_declaration~~
~~| module_common_item~~

module_or_generate_item_declaration ::=
  package_or_generate_item_declaration ~~net_declaration~~
~~| data_declaration~~
| genvar_declaration
~~| task_declaration~~
~~| function_declaration~~
~~| dpi_import_export~~
~~| extern_constraint_declaration~~
~~| extern_method_declaration~~
| clocking_decl
| **default clocking** clocking_identifier **;**

non_port_module_item ::=
  { attribute_instance } generated_module_instantiation
~~| { attribute_instance } local_parameter_declaration~~
| module_or_generate_item
~~| { attribute_instance } parameter_declaration ;~~
| { attribute_instance } specify_block
| { attribute_instance } specparam_declaration
| program_declaration
~~| class_declaration~~
| module_declaration

Within Section A.1.6 change (as shown in red):

interface_or_generate_item ::=
~~| { attribute_instance } continuous_assign~~
~~| { attribute_instance } initial_construct~~
~~| { attribute_instance } always_construct~~
~~| { attribute_instance } combinational_construct~~
~~| { attribute_instance } latch_construct~~
~~| { attribute_instance } ff_construct~~
~~| { attribute_instance } local_parameter_declaration~~
~~| { attribute_instance } parameter_declaration ;~~
| { attribute_instance } module_common_item
| { attribute_instance } modport_declaration
| { attribute_instance } extern_tf_declaration
~~| { attribute_instance } final_construct~~
~~| { attribute_instance } ;~~

non_port_interface_item ::=
  { attribute_instance } generated_interface_instantiation
| { attribute_instance } specparam_declaration
| interface_or_generate_item
| program_declaration
~~| class_declaration~~
| interface_declaration

Within Section A.1.7 change (as shown in red):

non_port_program_item ::=
{ attribute_instance } continuous_assign
| { attribute_instance } module_or_generate_item_declaration
| { attribute_instance } specparam_declaration
~~| { attribute_instance } local_parameter_declaration~~
~~| { attribute_instance } parameter_declaration ;~~
| { attribute_instance } initial_construct
| { attribute_instance } concurrent_assertion_item
| { attribute_instance } final_construct
~~| class_declaration~~

Add Section A.1.9 (as shown in red):

A.1.9 Package items

package_item ::=
package_or_generate_item_declaration
| specparam_declaration
| concurrent_assertion_item_declaration
| anonymous_program

package_or_generate_item_declaration ::=
net_declaration
| data_declaration
| task_declaration
| function_declaration
| dpi_import_export
| extern_constraint_declaration
| extern_method_declaration
| class_declaration
| parameter_declaration **;**
| local_parameter_declaration

anonymous_program ::= **program ;** { anonymous_program_item } **endprogram**

anonymous_program_item ::=
task_declaration
| function_declaration
| class_declaration

Add Section A.2.3 (as shown in red):

data_declaration ::=
[lifetime] variable_declaration
| constant_declaration
| type_declaration
| package_import_declaration

package_import_declaration ::=
import package_import_item { , package_import_item } **;**

package_import_item ::=
package_identifier **::** identifier
| package_identifier **::** *

Within Section A.2.6 change (as shown in red):

function_body_declaration ::=
[signing] [range_or_type]
[interface_identifier **.**] function_identifier **;**
{ function_item_declaration }
{ function_statement_or_null }
endfunction [**:** function_identifier]
| [signing] [range_or_type]
[interface_identifier **.**] function_identifier **(** tf_port_list**) ;**
{ data_declaration ~~block_item_declaration~~ }
{ function_statement_or_null }
endfunction [**:** function_identifier]

function_item_declaration ::=
data_declaration ~~block_item_declaration~~
| { attribute_instance } tf_input_declaration **;**
| { attribute_instance } tf_output_declaration **;**
| { attribute_instance } tf_inout_declaration **;**
| { attribute_instance } tf_ref_declaration **;**

Within Section A.2.7 change (as shown in red):

task_body_declaration ::=
[interface_identifier **.**] task_identifier **;**
{ task_item_declaration }
{ statement_or_null }
endtask [**:** task_identifier]
| [interface_identifier **.**] task_identifier **(** task_port_list **) ;**
{ data_declaration ~~block_item_declaration~~ }
{ statement_or_null }
endtask [**:** task_identifier]

task_declaration ::= **task** [lifetime] task_body_declaration

task_item_declaration ::=
data_declaration ~~block_item_declaration~~
| { attribute_instance } tf_input_declaration **;**
| { attribute_instance } tf_output_declaration **;**
| { attribute_instance } tf_inout_declaration **;**
| { attribute_instance } tf_ref_declaration **;**

Within Section A.2.8 change (as shown in red):

~~block_item_declaration ::=~~
~~{ attribute_instance } block_data_declaration data declaration~~
~~| { attribute_instance } local_parameter_declaration~~
~~| { attribute_instance } parameter_declaration ;~~

Within Section A.6.3 change (as shown in red):

function_seq_block ::=
begin [**:** block_identifier { data_declaration ~~block_item_declaration~~ }] {
function_statement_or_null }
end [**:** block_identifier]

seq_block ::=
begin [**:** block_identifier] { data_declaration ~~block_item_declaration~~ } {
statement_or_null }
end [**:** block_identifier]

par_block ::=
fork [**:** block_identifier] { data_declaration ~~block_item_declaration~~ } {
statement_or_null }