

Please note:

1. The information between here and the change for Section 17.12 is not part of the proposal. It is purely text describing the purpose and intent of the actual proposal.
2. The text in the document that is highlighted in blue was not voted upon by the working group. These changes were made after the vote and were based upon discussions that occurred among the working group members. These changes consist of:
 - a. corrections to the document
 - b. clarification of the search for an identifier name based on discussions between Randy Misustin, Matthew Maidment, and Dave Rich.
 - c. replacement of most uses of \$root with “local root” based on feedback from Randy Misustin, Matthew Maidment, and Francoise Martinolle. The feedback was not in the form of specific modifications but were along the line of confusion due to the use of \$root being use as synonymous with local root (which is not strictly true). There are two issues were \$root was still being used. The first was to access the instantiation path as in 3.1. The second was to access the declarations defined within the local root. This, it turns out, could be ambiguous and a different mechanism is suggested (which has to be reviewed, discussed, and approved).

Separate Compilation Discussion

The objectives of supporting separate compilation are:

- Support compilation of collections of SystemVerilog files. This is known as a unit of compilation.
- Support restricted global reference between units of compilation of SystemVerilog code.
- Support definition of items that can be shared within a unit of compilation.
- Support collecting declarative items into a restricted namespace that can be referenced globally and imported. This is the purpose of the package definition.

In order to support separate compilation and to simplify the current semantics of \$root within SystemVerilog the following changes are proposed:

- Change the definition of \$root within Section 18 (18.1, 18.2, 18.3, 18.6, 18.9) to [separate the items that can be placed in a local root and the mechanism to access instantiated items. This has resulted in limiting the items that can be placed in a local_root \\$root](#) to:
 - any item that can be defined in a package
 - an import of one or more items from a package
 - modules, macromodules, primitives, programs, interfaces, and packages
 - directives: bind, [timeunit](#), [timeprecision](#)
- This implies the removal of instances (module, interface, program) (Section 18), statements, genvar, and assert and cover statements (Section 17.12) from \$root.

- Rework relation between \$root and library statements (Section 21.3).
- Include packages and programs within libraries within Section 21.2.
- Include timeunit, timeprecision, property, and sequence declarations within the package (Section 18.4)
- Addition of two new sections on packages and separate compilation.

Within Section 1 change (as shown in red):

— A local root ~~\$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, ~~functions in the local root are not accessible 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, or at the local root ~~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 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
- local root
- ~~\$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
- local root
- ~~\$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*
 - local root
 - ~~\$root~~

Within Section 18.1 change (as shown in red):

- A global declaration space, visible to all modules, **interfaces, and programs** at all levels of hierarchy **within the unit of compilation** (see Section 18.3)
- **Separate compilation support**
- **A concept of packages to simplify sharing of items including data, types, classes, tasks and functions**

Within Section 18.2 change as shown in red.

18.2 The local root ~~\$root~~ top level

In SystemVerilog there is a top level called local root ~~\$root~~, which is the whole source text **within a compilation unit**. This allows declarations outside any named modules or interfaces, unlike Verilog.

~~SystemVerilog requires an elaboration phase. All modules and interfaces must be parsed before elaboration.~~

~~The order of elaboration shall be: First, look for explicit instantiations in \$root. If none, then look for implicit instantiations (i.e. uninstantiated modules). Next, traverse non-generate instantiations depth first, in source order. Finally, execute generate blocks depth first, in source order.~~

The source text can include the declaration **and use** of modules, **programs, packages,** and interfaces. Modules can include the declaration and use of other modules and interfaces. Interfaces can include the declaration and use of other interfaces. A module or interface need not be declared before it is used in text order.

~~A module can be explicitly instantiated in the \$root top level.~~ All uninstantiated modules become implicitly instantiated within the top level, which is compatible with Verilog.

The following paragraphs compare the local root ~~\$root~~ top level and modules.

The local root ~~\$root~~ top level:

- has a single occurrence **per compilation unit**
- can be distributed across any number of files
- **can contain any item that can be defined within a package and the definitions are in a namespace local to the compilation unit and can be accessed through the hierarchy within the unit of compilation**

- can contain declarations of modules, macromodules, primitives, programs, interfaces, and packages which are in a global namespace
- can contain one timeunit and timeprecision directive within the unit of compilation
- variable and net definitions are in a ~~local~~ **global** name space and can be accessed throughout the hierarchy **within the unit of compilation**
- task and function definitions are in a ~~local~~ **global** name space and can be accessed throughout the hierarchy **within the unit of compilation**
- ~~— shall not contain **initial** or **always** procedures~~
- ~~— can contain procedural statements, which shall be executed one time, as if in an **initial** procedure~~
- can contain ~~assertion declarations, assertion statements and~~ bind directives
- shall not contain instances, procedural statements, genvar declaration, or assert or cover statements

Modules:

- can have any number of module definitions
- can have any number of module instances, which create new levels of hierarchy
- can be distributed across any number of files, and can be defined in any order
- variable and net definitions are in the module instance name space and are local to that scope
- task and function definitions are in the module instance name space and are local to that scope
- can contain any number of **initial** and **always** procedures
- shall not contain procedural statements that are not within an **initial** procedure, **always** procedure, task, or function

When an identifier is referenced within a scope, SystemVerilog follows the Verilog name search rules (where first the nested scope is searched (1364-2001 12.6) and then the instance hierarchy is searched (1364-2001 12.5)). Nested module declarations and local root are searched as part of the nested scope search with the local root being the termination of the nested scope search. ~~and then searches in the \$root global local root name space. An identifier in the \$root global name space can be explicitly selected by pre-pending \$unit:: "\$root." to the identifier name. For example, a \$root global variable named system_reset can be explicitly referenced from any level of hierarchy using \$root.system_reset.~~

`$unit` is the name of scope that encompasses a unit of compilation. Its purpose is to allow the unambiguously reference to declarations at the outermost level of a unit of compilation. This uses the scope resolution operator, as if it were an implicit package.

For example:

```
bit b;
```

```

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

```

`$root` is the root of the instantiation tree. Its only purpose is to allow users to unambiguously refer to a top level instance, or to an instance path starting from 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

```

~~The `$root` space can be used to model abstract functionality without modules. The following example illustrates using the `$root` space with just declarations, statements and functions.~~

```

typedef int myint;

function void main();
    myint i,j,k;
    $display("entering main...");
    left(k);
    right(i,j,k);
    $display("ending... i=%0d, j=%0d, k=%0d", i, j, k);
endfunction

function void left(output myint k);
    k = 34;
    $display("entering left");
endfunction

function void right(output myint i, j, input myint k);
    $display("entering right");
    i = k/2;
    j = k+i;
endfunction

main();

```

ADD new sections 18.3 and 18.4 after Section 18.2 (renumber succeeding sections and syntax boxes):

18.3 Separate Compilation Support

SystemVerilog supports separate compilation through the idea of a separately compiled unit. The unit of compilation is defined as a collection of one or more files. This collection of files share a local root ~~and `$root` refers to the~~ that contains definitions ~~local to within~~ the unit of compilation. The exact mechanism for defining which files go into each individual compilation units is tool specific. The requirement is that a tool provides a mechanism to define the list of files which make up a compilation unit. Two extreme cases are:

1. all files make a single compilation unit (in which case the items in ~~the local root `$root`~~ are accessible anywhere within the design)
2. each file is a separate compilation unit (in which case the items in ~~the local root `$root`~~ are accessible only to the items defined within the file)

The items that can be shared between units of compilation are modules, macromodules, primitives, programs, interfaces, and packages. All other items which are defined within the local root cannot be accessed by name outside the unit of compilation. Access to the items in `local root` ~~`$root`~~ that are not shared can be accessed using the PLI which must provide an iterator to iterate through all of the units of compilation. Collision of items which are shared obey the standard rules as if the items were loaded together as a single text stream.

~~References within a unit of compilation can access objects within the local root name `$root` space either by the object's name (`<object_name>`) or by `$.root.<object_name>`. In resolving a reference, SystemVerilog follows the standard name search rules within the current separate compilation unit, and then searches the `$root` for the current separate compilation unit, and then searches the scope containing modules, macromodules, primitives, programs, interfaces.~~

Verilog supports the idea that compiler directives once seen by a tool will apply to all forthcoming modules or files in the design. This behavior shall be supported within a separately compiled unit; however, compiler directives from one separately compiled unit shall not impact the behavior of another separately compiled unit.

18.4 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 outer level of the source text (at the same level as modules, primitives, interfaces, etc.). 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 hierarchical 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 top-level region to contain declarations intended to be shared among one or more 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 occasionally be useful for globals that aren't conveniently passed down through the hierarchy. Any variables that have initial values defined for them will be initialized in the same way the initialization occurs for variables declared in [the local root \\$root](#).

The following is an example of a package:

```

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

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

    function Complex mul(input 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

```

18.4.1 Referencing data in packages

Packages must be defined before they are referenced to allow the types they define to be recognized by the modules that import from them.

One of the ways to utilize declarations made in a packages is to reference them using the namespace operator "::".

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

An alternate method for utilizing declarations is 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 symbols within Packages. It allows those symbols declared within Packages to be visible within the current scope by their declared simple name. Package hierarchical names to the imported symbols can be created as if the symbol 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;

```

Explicit imports are treated similarly to a declaration. An explicit import shall be illegal if another symbol by the same name has already been declared or imported into the same scope unless the symbol is from the same package. Similarly, after importing a symbol by a given name, it shall be illegal to then declare a symbol by that same name within the same scope.

Wildcard import allows symbols defined within a package to be imported provided the symbol is not otherwise defined in the importing scope:

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

All the symbols within a package implied by a wildcard import are candidates for import. They, in fact, become imported only if there are no other symbols by the same name declared or imported in the same scope. Similarly, their visibility may be limited by a subsequent declaration of the

same name in the same scope. If the same symbol is defined in two wildcard imports in the same scope, the symbol shall be undefined within that scope.

18.4.2 Search order Rules

Table 8.x describes the search order rules for the declarations imported from a package. To understand the table, consider the following package declarations:

```

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

package q;
    const c = 0;
endpackage;

```

Syntax	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::*
<pre> p::c; p::TRUE; e.g.: u = p::c; y = p::TRUE; </pre>	<p>A qualified package identifier is visible in any scope (without the need for an import clause).</p>	<p>OK.</p> <p>Direct reference to c refers to the locally declared c.</p> <p>p::c refers to the c in package p.</p>	<p>OK</p> <p>Direct reference to c is illegal since it is undefined.</p> <p>p::c refers to the c in package p.</p>	<p>OK.</p> <p>Direct reference to c refers to the c imported from q.</p> <p>p::c refers to the c in package p.</p>	<p>OK.</p> <p>Direct reference to c refers to the c imported from q.</p> <p>p::c refers to the c in package p.</p>
<pre> import p::*; . . . y = FALSE; </pre>	<p>All declarations inside package p become potentially directly visible in the importing scope:</p> <ul style="list-style-type: none"> • c • BOOL • FALSE • TRUE 	<p>OK.</p> <p>Direct reference to c refers to the locally declared c.</p> <p>Direct reference to other identifiers (e.g., FALSE) refer to those implicitly imported from package p.</p>	<p>OK.</p> <p>Direct reference to c refers to the c imported from package p.</p>	<p>OK.</p> <p>Direct reference to c refers to the c imported from package q.</p>	<p>OK / ERROR</p> <p>c is undefined in the importing scope. Thus, a direct reference to c is illegal and results in an error.</p> <p>The import clause is otherwise allowed.</p>

<pre>import p::c; . . . if(! c) ...</pre>	<p>The imported identifiers become directly visible in the importing scope:</p> <ul style="list-style-type: none"> • c 	<p>ERROR.</p> <p>It shall be illegal to import an identifier defined in the importing scope.</p>	<p>OK.</p> <p>Direct reference to c refers to the c imported from package p.</p>	<p>ERROR.</p> <p>It shall be illegal to import an identifier defined in the importing scope.</p>	<p>OK / ERROR</p> <p>It shall be illegal to reference c before the import of p::c.</p> <p>Otherwise, direct reference to c refers to the c imported from package p.</p>
---	--	---	---	---	--

Table 8.x Scoping Rules for Package Importation

When using the `import q::* p::c` form of importation, the use of a variable forces the import of that variable into the local scope, thus creating an error if another package with the same variable name is imported later. 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;
```

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 18.6 (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 ~~the local root of a compilation unit~~. 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 (**program and packages cannot be nested**).
- 2) Else, if a `'timescale` directive has been previously specified, then the time unit **is** shall be set to the units of the last `'timescale` directive.
- 3) Else, if the **local root of a unit of compilation** ~~the local root of a unit of compilation~~ `$root` ~~top level~~ has a time unit, then the time unit **is** shall be set to the time units of the **unit of compilation root module**.
- 4) Else, the default time unit is shall be used.

The time unit of **the local root of a unit of compilation** ~~the local root of a unit of compilation~~ `$root` shall only be determined by a `timeunit` declaration, not a `'timescale` directive.

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 18.9 (current numbering) change (as shown in red):

18.9 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 unit of compilation** (*local root 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 module, macromodule, primitive, program, and interface identifiers defined within the **local root** ~~the local root~~ `$root` among all compilation units. Once a name is used to define a module, macromodule, primitive, program, or interface within the **local root** ~~the local root~~ `$root` within one compilation unit the name shall not be used again to declare another module, macromodule, primitive, program, or interface within **local root** ~~the local root~~ `$root` in any compilation unit
- 2) The *package name space* unifies all the package identifiers defined within **local root** ~~the local root~~ `$root` among all compilation units. Once a name is used to define a package within **the local root** ~~the local root~~ `$root` within one compilation unit the name shall not be used again to declare another package within a **local root** ~~the local root~~ `$root` in any compilation unit.

- 3) The *local root 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 unit of compilation.
- 4) The *text macro name space* is global **within the unit of compilation**. 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 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 18.10 (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 design instantiated 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 ~~the local root \$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 }

```