All references are based on P1800-2008 draft 4

In 25.3

CHANGE:

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 referenced in the importing scope and 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 an error results if the identifier is used.

TO:

A wildcard import is of the following form:

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

For the purpose of describing how names resolve to wildcard imported declarations we define what is a locally visible and potentially locally visible name.

Locally visible names are the names of locally visible declarations. It shall be an error if there is more than one locally visible declaration of the same name.

A name is *locally visible* at some point, *P*, in a scope, *S*, if one of the following holds:

1. the name denotes a scope within *S*
2. the name is declared within *S* before point *P*
3. the name is visible from an import select occurring within S before point *P*

A name is *potentially locally visible* at some point, P, in a scope, *S*, if :

there exist a wildcard import of a package before point P within scope *S* and the package contains a declaration of that name.

Additionally a potentially locally visible symbol from a wild card import may become locally visible if

no locally visible symbol is found and the resolution of an identifier causes the potentially locally visible name

to become locally visible.

The following paragraph describes how identifiers can be resolved to wildcard imported declarations.

A wildcard import makes all identifiers declared within a package potential locally visible symbols in the importing scope, *S.*

A potentially locally visible symbol may become locally visible during name resolution of an identifier appearing

in a scope *S*, in accordance to the following rules:

If the identifier is not a function or task call,

the following search algorithm is repeated for each scope *S in the nested lexical scopes* until a declaration is found which matches the identifier name or there is no more upper lexical scopes. The compilation unit scope being the final scope searched.

A.1 - the locally visible declarations in current scope *S* are first searched for a match, if a match is found the identifier is bound to that declaration

A.2- else if no locally visible symbol match, then the potentially locally visible symbols in the scope *S,* ***lexically preceding*** the identifier name are searched for a match. If a match is found, the symbol from the package is imported in scope *S* and becomes a locally visible declaration in scope *S,* that declaration is bound to the identifier.

A.3- else current scope *S* is set to its upper scope

If there are no more lexical scopes, and the identifier cannot be bound, an undeclared symbol error shall be issued.

If the identifier is a function or task call, the rules are slightly different and follow upwards hierarchical name resolution (section 22.7).

B.1- First the **entire** scope *S* (all the declarations of locally visible names up to the end of the scope) is searched for a matching declaration, if a match, the identifier is resolved to that declaration,

B.2- else if no match is found, the potentially locally visible symbols from wildcard imports in current scope *S,* ***lexically preceding*** the identifier name are searched for a match,

if a match is found, the potential visible symbol becomes locally visible, is imported in current scope *S* and bound to that identifier,

B.3- else if no match is found, current scope *S* is set to upper scope and steps B.1, B.2, B.3 are repeated including when *S* is the compilation unit scope.

B.4. else if no match, steps b and c of the upwards name resolution are executed (the function and task name are searched up the instance hierarchy).

If the same name is potentially locally visible from more than one wildcard imported package in scope *S*, and those wildcard imports are **lexically preceding** an identifier of that name, neither symbols from the packages is made locally visible and the identifier is not resolved to any of the wildcard imported symbols.

If a wildcard imported symbol is made locally visible in a scope *S*, any later locally visible declaration of the same name in scope *S* shall be illegal.

Example 1:

```
package p;
    int x;
endpackage

module top;
    import p::*; // line 1

     if (1) begin : b
         initial x = 1; // line 2
         int x; // line 3
```

```
            initial x = 1; // line 4
        end
        int x; // line 5
endmodule
```

The reference in line 2 causes the potentially locally visible x from wildcard import p::* (`p::x`) to become locally visible in scope top and line 2 initializes `p::x`. Line 4 initializes `top.b.x`. Line 5 is illegal since it is a local declaration in scope `top` which conflicts with the name `x` imported from `p` which had already become a locally visible declaration.

Example 2:

```
package p;
      int x;
endpackage

package p2;
      int x;
endpackage

module top;
      import p::*; // line 1
      if (1) begin : b
          initial x = 1; // line 2
          import p2::*; // line 3
      end
endmodule
```

Line 2 causes the import of `p::x` in scope "top" because the wildcard import `p::*` is in the outer scope "top" and precedes the occurrence of `x`. The declaration x from package p becomes locally visible in scope "top".

Example 3:

```
package p;
    function int f();
        return 1;
    endfunction
endpackage

module top;
    int x;
    if (1) begin : b
        initial x = f(); // line 2
        import p::*; // line 3

    end

    function int f();
        return 1;
        endfunction
    endmodule
```

"f()" on line 2 binds to `top.f` and not to `p::f` since the import is after the function call reference.

Example 4:

```
    package p;
        function int f();
            return 1;
        endfunction
    endpackage

    package p2;
        function int f();
            return 1;
        endfunction
    endpackage

    module top;
        import p::*;
        int x;
        if (1) begin : b
            initial x = f(); // line 1
        end
        import p2::*;
    endmodule
```

Since "`f`" is not found in scope `b`, the rules require inspection of all wildcard imports in the parent scope. There are two wildcard imports, but only the wildcard import p::* that is lexically preceding the occurrence of `f()` is considered. In this case "f" binds to `p::f`.

In 3.10.1:

CHANGE:

a) All files on a given compilation command line make a single compilation unit (in which case the declarations within those files are accessible anywhere else within the constructs defined within those files).

TO:

a) All files on a given compilation command line make a single compilation unit (in which case the declarations within those files are accessible following normal visibility rules throughout the entire set of files ~~anywhere else within the constructs defined within those files~~).

CHANGE:

When an identifier is referenced within a scope:

— First, the nested scope is searched (see 22.7) (including nested module declarations), including any identifiers made available through package import declarations.

— Next, the compilation-unit scope is searched (including any identifiers made available through package import declarations).

— Finally, the instance hierarchy is searched (see 22.6).

TO:

When an identifier is referenced within a scope:

— First, the nested scope is searched (see 22.7) (including nested module declarations), including any identifiers made available through package import declarations.

— Next, the portion of the compilation-unit scope defined prior to the reference is searched (including any identifiers made available through package import declarations).

— Finally, if the identifier follows hierarchical name resolution rules, the instance hierarchy is searched (see 22.6 and 22.7).

CHANGE:

For example:

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

TO:

For example:

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

Other than for task and function names (see 22.7.1), references shall only be made to names already defined in the compilation unit. The use of an explicit `$unit::` prefix only provides for name disambiguation and does not add the ability to refer to later compilation unit items.

For example:

```
task t;

     int x;

     x = 5 + b;            // illegal – "b" is defined
later

     x = 5 + $unit::b;     // illegal – $unit adds no
special forward referencing

endtask

bit b;
```

Add the following to the end of Clause 13:

13.7 Task and Function names

Task and function names are resolved following slightly different rules than other references. Even when used as a simple name, a task or function name follows a modified form of the upwards hierarchical name resolution rules. This means that "forward" references to a task or function defined later in the same or an enclosing scope can be resolved. See Section 22.7.1 for the rules that govern task and function name resolution.

Add the following to the end of Section 22.7:

22.7.1 Task and Function name resolution

Task and function names are resolved following slightly different rules than other references. Task and function name resolution follows the rules for upwards hierarchical name resolution as described in 22.7, step (a). Then, before proceeding with step (b), an implementation shall look in the complete compilation unit of the reference. If a task or function with a matching name is found there, the name resolves to that task or function. Only then does the resolution proceed with step (b) and iterate as normal. The special matching within the compilation unit shall only take place the first time through the iteration through steps (a)-(c); a task or function name shall never match a task or function in a compilation unit other than the compilation unit enclosing the reference.

Example:

```
task t;
     int x;
     x = f(1);             // valid reference to
function f in $unit scope
endtask
function int f(int y);
     return y+1;
endfunction
```

Example:

```
package p;
  function void f();
    $display("p::f");
  endfunction
```

```
endpackage


module top;

  import p::*;


  if (1) begin : b          // generate block

     initial f();           // reference to “f”

     function void f();

       $display("top.b.f");

     endfunction

  end

endmodule
```

The resolution of the name `f` follows the hierarchical rules and therefore is resolved to the function `top.b.f`. The output of the example would be the output of the string “top.b.f”.