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

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

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

An identifier is *potentially locally visible* at some point within a scope if there is a wildcard import of a package before that point within the current scope and the package contains a declaration of that identifier.

An identifier is *locally visible* at some point within a scope if that identifier:

1. denotes a nested scope within the current scope, or
2. is declared as an identifier prior to that point within the current scope
3. is visible from an explicit import prior to that point within the current scope

A potentially locally visible identifier from a wildcard import may become locally visible if the resolution of a reference to an identifier causes finds no other matching locally visible identifiers.

For a reference to an identifier other than function or task call, the locally visible identifiers defined at the point of the reference in the current scope shall be searched. If the reference is a function or task call, all of the locally visible identifiers to the end of the current scope shall be searched. If a match is found the reference shall be bound to that locally visible identifier.

If no locally visible identifiers match, then the potentially locally visible identifiers defined prior to the point of the reference in the current scope shall be searched. If a match is found, that identifier from the package shall be imported into the current scope, becoming a locally visible identifier within the current scope, and the reference shall be bound to that identifier.

If the reference is not bound within the current scope, the next outer lexical scope shall be searched; first from the locally visible identifiers in that scope and then the potentially locally visible identifiers defined prior to the point of the reference. If a match is found among the potentially locally visible identifiers, that identifier from the package shall be imported into the outer scope, becoming a locally visible identifier within the outer scope.

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

The search algorithm shall be repeated for each *outer lexical scope* until an identifier is found that matches the reference or there are no more outer lexical scopes, the compilation unit scope being the final scope searched. For a reference to an identifier other than function or task call, it shall be illegal if no identifier can be found that matches the reference. If the reference is a function or task call, the search continues using upwards hierarchical identifier resolution (See 22.7.1).

It shall be illegal if the wildcard import of more than one package within the same scope defines the same potentially locally visible identifier and a search for a reference matches that identifier.

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 22.7 and 22.8).

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”.