SV-EC/BC Name resolution meeting

9/25/2007

Gord Vreugdenhil
Mentor Graphics

Purpose and Goals

- To discuss issues related to name resolution and to reach consensus on principles and directions to be used as the foundation for specific proposals.
- To determine the amount of work needed to resolve issues for P1800-2008 and a sense of the seriousness of those issues.
- This is a non-voting meeting in terms of any specific proposal. It is likely worthwhile to try to garner the “sense of the group” in terms of level of support for specific suggestions.

Agenda

- What is the goal
- The basic problem of opaque types
- Issues independent of opaque types
- Discussion on managing opaque types

Unless otherwise noted “the LRM” shall mean P1800-2008 Draft 3a

LRM Deficiencies

- LRM, Sec. 22.8 is “Scope Rules”. Clearly those rules are incomplete and contradicted by other parts of the LRM. In particularly, $unit and class issues are not addressed in 22.8 and conflict with the specific rules there.
- Issues related to overloaded syntax (dotted names and “::” syntax are simple examples).
- Special rules in some cases interact poorly with other general rules

Opaque types

- The term *opaque type* is not in the LRM but for this meeting we'll use that term to mean type names that can denote types that require elaboration to be known
  - There are two basic kinds of opaque types – type parameters and a typedef of an interface type. i.e. type declarations of the form:

```
typedef intf.T myT;
parameter type T = int;
```

Opaque types (continued)

- An opaque type does not permit an implementation to have knowledge about the nature of the type prior to elaboration. This definitely complicates the management of classes and the handling of identifiers imported from packages.
- The set of rules for how to deal with name resolution in the presence of opaque types is at the core of some of the deeper disagreements. This will be covered in depth later.

Other Resolution Issues

- There are numerous issues which don’t directly interact with the decisions regarding opaque types.

(A) Declaration before use

- The LRM says (6.5):

  > Data must be declared before they are used, apart from implicit nets…

- Even this isn't really true – certainly hierarchical references violate this rule. Even the concept of "declaration" is a bit fuzzy in various cases.
- Types are not directly addressed.
- The exact "point of declaration" is not specified.

(A) Declaration before use (continued)

- Example 1:

```
int A = A;
```

- Example 2:

```
module top;
    int x = top.y;
    int y;
endmodule
```

(A) Declaration before use (continued)

- Suggestions:
  - We should clarify that "hierarchical" resolution is not dependent on *order* of declaration, just on the *existence* of a declaration. All non-hierarchical resolution *is* order of declaration dependent. We need to be very careful to distinguish when "hierarchical" rules apply.
  - We need to more carefully state requirements regarding type knowledge – type references must either be *known* to be type names or, in the context of a typedef, be *asserted* to be a type. The latter is a way to think about typedefs to interface types:

```
        typedef intf.T T;
```

(A) Declaration before use (continued)

- Suggestions:
  - For a data object’s initial value, we should disallow references to the value of a data object being declared. We could allow references to the type but could restrict that as well.
  - For a typedef, we should disallow references to the type name within the default type.
  - Examples:

```
int A = A;                          // illegal
int A = type(A)’(7);                // legal?
parameter type T = T[];             // illegal
```

(B) Visibility into $unit

- The LRM says (3.10):
  - 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).
- The second bullet is unclear in terms of whether “declaration before use” is expected.
- Example:

```
module top;
    int x = y;
endmodule
int y;
```

(B) Visibility into $unit (continued)

- Also in 3.10, the LRM says:

  > $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 …

  and later:

  > Within a particular compilation unit, however, the special name $unit can be used to explicitly access the declarations of its compilation-unit scope.

- The first part implies that “$unit::” is only intended for disambiguation. That is less obvious from the latter statement.

(B) Visibility into $unit (continued)

- If “$unit::” is only for disambiguation of names, is it true that in the absence of a name ambiguity, that a “$unit::” reference is valid if and only if the equivalent non-prefixed name is valid?
- Example:

```
module top;
     int x = $unit::y;
endmodule
int y;
```

(C) How “package like” is $unit

- The LRM says:
  - Packages must not contain any processes. Therefore, net declarations with implicit continuous assignments are not allowed. (25.2)
  - Items within packages cannot have hierarchical references. (25.2)
  - Packages must exist in order for the items they define to be recognized by the scopes in which they are imported. (25.3)
- Do these rules apply to $unit?
- If so, does the third bullet disallow forward function references into $unit?

(C) How "package like" is $unit (continued)

- Example 1:

```
wire w = 1;
module top;
    assign w2 = w;
endmodule
```

- Example 2:

```
module top;
    int x = f(1);
endmodule
function int f(int a);
    return a;
endfunction
```

(C) How “package like” is $unit (continued)

- Example 3:

```
module top;
    $unit::T x;
endmodule
typedef int T;
```

(C) How “package like” is $unit (continued)

- Should $unit references be equivalent to an import?

```
typedef int T;
module top;
      T x;        // like “import $unit::T” ?
      int T;      // then this is illegal
endmodule
```

- Equivalent issues exist even in 1364-2001 so making $unit references behave like imports would be both inconsistent and incompatible.

```
module top;
      integer x;
      generate if (1) begin
            integer y = x;    // means top.x
            integer x;
      end endgenerate
endmodule
```

(C) How “package like” is $unit (continued)

- Can one reference hierarchically into a package (or $unit) item?

```
package p;
     task t;
          int x;
     endtask
endpackage
module top;
     int y = p::t.x;

     import p::t;
     int z = t.x;
endmodule
```

(D) How static are "::" references?

- The LRM (and Mantis 227) are clear that references into packages can only be made after compilation of the package.
- This means that "::" can be considered a "static resolution operator" in the sense that "existence before use" applies.
- Does the same apply to the class resolution operator?

(D) How static are "::" references?(continued)

- Example 1:

```
module top;
    typedef C;
    C::T x;
    class C;
        typdef int T;
    endclass
endmodule
```

(D) How static are "::" references?(continued)

- Example 2:

```
module top;
    typedef C;
    typedef T2;
    T2 x;
    class C;
        typdef T2 T;
    endclass
    typedef C::T T2;
endmodule
```

(E) When is a dotted name hierarchical?

- For now, consider a “dotted name” to be the names derivable from *hierarchical_identifier* in the BNF (ignoring package prefixing, etc).
- Neither in the context of structs or classes does the LRM talk about “.” as a field/property select operation. From a narrow LRM perspective, any dotted name is hierarchical.
- Clearly a purely “hierarchical” view of dotted names is nonsense since initial terms in a hierarchical name are required to be scopes.
- The rules that govern hierarchical name resolution are different in that they are dependent on the order of declaration within a scope.
- In 1364 (all versions), there was no ambiguity when one saw a dotted name – the name was hierarchical.
- In some circumstances we need a dotted name to be treated as a field/member select and not as a hierarchical name.
- Names can also be mixed – a combination of hierarchical and selected.

(E) When is a dotted name hierarchical? (continued)

- Suggestion: A dotted name shall be considered to be composed of a possibly empty hierarchical prefix followed by a possibly empty selected name.
- Intent: Once we start with a “selected” name, we never revert to a hierarchical resolution.
- Example:

```
module top;
     int x;
     child c();
endmodule
module child;
     struct { int y; } top;
     int z= top.x;  // should fail
endmodule
```

(E) When is a dotted name hierarchical? (continued)

- This approach makes the question of when one “commits” to a non-hierarchical resolution critical.
- Example 1 (example 7 in discussion examples)

```
int x;
generate if (1) begin : b
      int z;
      initial z = x;
      int x;
end endgenerate
```

- Example 2 (example 8 in discussion examples)

```
struct { int y; } x;
generate if (1) begin : b
      int z;
      initial z = x.y;
      struct { int y; } x;
end endgenerate
```

(E) When is a dotted name hierarchical? (continued)

- Suggestion: Name resolution should be biased towards a non-hierarchical resolution for a dotted name.
- Essentially this boils down to saying that one resolves the first item of a dotted name as though it were a normal “simple” reference. If it resolves to a non-scope identifier, then you are committed to resolving the full reference from that “anchor” point. If the name resolves to a scope identifier, then (for compatibility) you treat the name as hierarchical.

(E) When is a dotted name hierarchical? (continued)

- Next implied issue – what forms the hierarchical “prefix” during dotted name resolution?
- The current algorithm in 22.7 is simplistic, even for 1364 compliant code. It only directly deals with dotted names of the form:

  ```
          scope_name.item_name
  ```

  The assumption in the LRM (and in implementations) is that the “item_name” can itself be a hierarchical name that is resolved in a downwards manner.
- Clarifying this part to allow a downwards “dotted name” is trivial and one could argue that this may be implied by the combined LRM since “item_name” isn’t discussed.

(E) When is a dotted name hierarchical? (continued)

- Suggestion: if during a downwards resolution of “item_name”, a component of “item_name” permits a dotted select, resolution commits to that resolution path and no futher upwards resolution can occur. This is the same bias as suggested earlier.
- Example:

```
    module top;
          task mid;
                struct { int x; } s;
          endtask
          mid m();
    endmodule
    module mid;
          struct { int y; } s;
          child c();
    endmodule
    module child;
          int z1 = mid.s.x;       // fails
          int z2 = s.x;           // fails
    endmodule
```

(E) When is a dotted name hierarchical? (continued)

- The following assumes that anchored hierarchical references into a package item are permitted.
- Example:

```
    package p;
         task t;
              int x;
         endtask
    endmodule
…
    module child;
         import p::*;
         int z1 = p::t.y;     // fails?
         int z2 = t.y;        // what to do?
    endmodule
```

(E) When is a dotted name hierarchical? (continued)

- Suggestion: a package or class prefixed dotted name shall always be treated in a downward manner.
- Suggestion: a reference to a visible scope name from a package shall cause that name to be imported. A dotted name without a "::" prefix shall become hierarchical (with possible upwards resolution) if the first name denotes a scope name imported from a package.

(E) When is a dotted name hierarchical? (continued)

- In 6.21, the LRM restricts hierarchical references to be to static variables. It is not specified whether references to non-static variables cause errors or cause resolution to continue in an upwards manner.
- Example:

```
module top;
      task mid;
            struct { int x; } s;
      endtask
      mid m();
endmodule
module mid;
      task automatic mid;
            struct { int x; } s;
      endtask
      child c();
endmodule
module child;
      int z1 = mid.s.x;
            // fails since top.m.mid is is automatic?
            // or resolves to top.mid.s.x?
endmodule
```

(E) When is a dotted name hierarchical? (continued)

- Suggestion: if a downwards resolution succeeds, it shall be an error if the resolved name is not a static declaration.
- This would permit a resolution to succeed into an automatic context if the target declaration is explicitly static.

(F) What to consider during the upwards phase

- The algorithm for upwards hierarchical name resolution in 22.7 has, as step (b), the following:
    - b) Look in the parent module’s outermost scope for a scope named scope_name. If found, the item name shall be resolved from that scope.
- This doesn’t cover issues related to having a parent that is a generate scope. That is a minor change that should have been made in 1364-2001. The upwards search should move upwards through the instance tree starting in the scope of the instantiation rather than the “outermost scope” of the parent.

(F) What to consider.. (continued)

- The algorithm currently requires that the first component of the resolution be a scope name. This implies that upwards resolution should not even consider variables that admit dotted selects.
- Example:

```
module top;
        task s;
                int x;
        endtask
        mid m();
endmodule
module mid;
        struct { int y; } s;
        child c();
endmodule
module child;
        int z2 = s.x;           // Ok by current rules
endmodule
```

- Suggestion: leave this alone and continue to require a “scope” as the first upwards name.

(G) Packages and upwards resolution

- The LRM says:

  Items defined in the compilation-unit scope cannot be accessed by name from outside.

- This should be clarified to also explain that a hierarchical name does not consider any $unit declarations during upwards resolution. The implication is that although hierarchical references can initially consider a scope name visible within the local $unit, once the upwards resolution begins, the compilation units of ancestors will not be considered.
- Example (assumes separate compilation unit model):

```
// file1.sv
task t;
      int x;
endtask
module top;
      int y = t.x;         // Ok
      child c();
endmodule

// file2.sv
module child;
      int z2 = t.x;        // Should fail
endmodule
```

(G) Packages and upwards resolution (continued)

- Similarly for packages, imported names should not be considered.
- Example:

```
package p;
     task t;
          int x;
     endtask
endpackage
module top;
     import p::*;
     child c();
endmodule
module child;
     int z2 = t.x;  // Should fail
endmodule
```

(H) Name resolution for bind instances

- What names are considered when handling a bind instance?
- The LRM talks both about referencing items “declared in” the target instance and *also* that the bound instance is treated “as though” it was instantiated at the end of the module.
- The assumed intent of the “at the end” is that the addition of an extra module cannot cause name resolution differences.
- Unfortunately that isn’t true in the presence of imports and scope nesting.

(H) Name resolution for bind instances (continued)

- Example:

```
package p;
      typedef int t;
endpackage

import p::*;
module top;
      // child c(t);  -- post-bind form
endmodule

int t;
module child(input int a);
endmodule

bind top child c(t);
```

- If the intent is to insulate names in the target from the impact of the "bind", this needs to be stated explicitly. If not addressed, this is likely to become more problematic with any future extensions for bind targets of generate scopes, nested modules, etc.

(I) Resolution of clocking block names

- In 22.8 (Scope rules), clocking blocks are not listed as a scope. In addition, the LRM general refers to the "clocking block construct" and doesn't talk about it as a scope.
- However, clocking blocks clearly admit internal declarations.
- If clocking blocks are scopes, they are candidates for a match during upwards resolution. Is that the expected behavior?

(I) Resolution of clocking block names (continued)

- Example:

```
module m(output wire w, input reg clk);
      clocking cb(@clk);
            output w;
      endclocking
      child c();
endmodule
module child;
      initial cb.w <= 1;
endmodule
```

- Suggestion: Add clocking blocks to the list of scopes.

(J) Modport issues

- This area is incomplete due to lack of time in formulating all of the issues clearly.
- Some basic questions (not covering all issues):
  - How authoritative are modport directions?
  - How do modport directions impact procedural and continuous assignment determination? What about driver uniqueness for always_comb and similar?
  - Is an interface port (or modport) considered a "scope" for the purposes of upwards hierarchical name resolution? If so, if the search in the port fails, what is the next parent? The parent of the instance or the parent of the module containing the interface port or modport?
  - Can one see non-modport items when hierarchically resolving through a modport?
  - Is a modport name a "scope" containing indirections back into the interface when resolving into an interface? (i.e. is intf.mport.name valid?)

(K) Forward references to class properties

- This area is incomplete due to lack of time in formulating all of the issues clearly.
- The question has been raised about whether P1800 should allow forward references to class properties.
- Example

```
class C;
     function int f();
          return x;
     endfunction
     int x;
endclass
```

(K) Forward references to class properties (continued)

- Although this would conceptually bring SV closer to C++ class resolution, there are numerous interactions that would have to be very carefully defined.
- Issues:
  - Legality of a change in the “kind” of a reference.
  - Legality of forward references to types, parameters, etc. within the class body (not the methods).
  - Interactions with opaque types, package imports, etc.
- Although we believe that this would be tractable, the changes are non-trivial and there would likely be incompatibilities with existing SV code or inconsistencies between method and non-method referencing of class properites.

Opaque types

- Reminder: For this presentation, opaque types are those formed from a typedef of an interface type or a type parameter.

```
typedef intf.T myT;
parameter type T = int;
```

- The key tradeoff is between a more dynamic name referencing or a more local determination of what a name means in the context of the reference.

- Mentor's assertion – It is crucial for long term correctness of designs to maximize one's ability to reason about the correctness of design units locally. Dynamic resolution of simple names leads to surprising and unintended effects that interfere with correctness and local reasoning about designs.

Opaque types (continued)

- The problems occur in two contexts, both related to classes.
- Situation 1: Extension of a type parameter. Issues arise since one no longer knows the set of basic names that are being inherited.
- Example:

```
module m #(parameter type T = int);
      int x;
      class C extends T;
            function int get_from_env();
                  return x;
            endfunction
      endclass
endmodule
```

Opaque types (continued)

- Situation 2: Inline constraints. Issues arise since there are special rules regarding resolution of a name into the object context within the constraint and one cannot know the universe of names within the object.
- Example:

```
module m #(parameter type T = int);
      T c = new;
      int x;
      initial c.randomize with { y > x; };
endmodule
```

Opaque types (continued)

- In both of the scenarios, allowing dynamic resolution of names can result in late errors or non-obvious “capture” of a name.
- On the other side of the issue, the argument is that not adopting dynamic resolution creates a more limiting set of rules than what applies in the context of non-opaque types.

Opaque types (continued)

- Mentor’s suggestion is to regularize the handling of all scenarios by requiring explicit specifications (or assertions) from the designer regarding the requirements on the opaque type.
- We believe that this is consistent with the spirit of Verilog. In Verilog a designer could not be surprised by a local name reference being hijacked. Non-local (hierarchical) references were always clear – they were dotted names that could be determined by local inspection. This meant that local naming correctness couldn’t be compromised. Global dependencies were explicit by way of “escaping” names. We believe that retaining locally reliable reasoning is very important for the long term success of the language.

Opaque types (continued)

- There are both long and short term directions that such a regularization can take. In the short term, minor restrictions or syntactic enhancements can be used to express intent.
- In the longer term, there are many avenues that could be considered to make such compositional invariants explicit.

Basic Algorithm Difference

- The approach taken in the Mentor algorithm (as outlined by Gord Vreugdenhil) treats an opaque type as a boundary. One never looks into an opaque type when resolving a simple name. Resolution of the name ignores the opaque context and either resolves or does not resolve. Elaboration type binding cannot impact the decision for that name.

Basic Algorithm Difference (continued)

- The approach taken in the Synopsys algorithm (as outlined by Mark Hartoog) is more dynamic. When encountering an opaque type boundary, one defers the final binding. Information regarding an alternative binding or possible error condition is collected but not reported. The final decision about the binding (or error reporting) is deferred until elaboration when the nature of the final type is known.

Basic Algorithm Difference (continued)

- Although perhaps not obvious from discussion on the reflector, we believe that the intent of both algorithms is the same for non-opaque types and that results will always match in such scenarios. That is certainly Mentor’s intent.

Examples of opaque types and discussion

- Example 1:

```
module child #(type T = int);
        int x;
        class C extends T;
                int y = x;      // (1)
        endclass
endmodule
```

- Mentor would resolve the reference to "x" in (1) to the locally visible "x". In existing syntax, users could explicitly say "super.x" to effectively assert that "T" must have a member "x" and that is the desired binding. There are various models one could use to extend the current language to couple this assertion to the declaration of "T" and eliminate the need for the "super." prefix.
- Synopsys would conditionally resolve the "x" in (1) to either "super.x" or "child.x" depending on whether type T had a member named "x".

Examples of opaque types and discussion (continued)

- Example 2:

```
package p;
        int x;
endpackage
module child #(type T = int);
        import p::*;
        class C extends T;
                int y = x;     // (1)
        endclass
        int x;
endmodule
```

- Mentor would call this an error. The reference to “x” in (1) binds “eagerly” to the p::x reference. The import then conflicts with the declaration of “int x;” in “child”.
- Synopsys would conditionally resolve the “x” in (1) to either “super.x” or “child.x” depending on whether type T had a member named “x”.
  If T does not have a member “x” then an elaboration time error would be reported due to the conflict with child.x. If T has a member named “x”, no error occurs.
- In either case approach, explicit use of either “child.x” or “super.x” yields consistent behavior.

Examples of opaque types and discussion (continued)

- Example 3:

```
package p;
      int x;
endpackage
module child #(type T = int);
      import p::*;
      class C extends T;
            function int f();
                  return x;   // (1)
            endfunction
      endclass
endmodule
```

- Mentor would resolve “x” to p::x.
- Synopsys would conditionally resolve the “x” in (1) to either “super.x” or “p::x” depending on whether type T had a member named “x”.

Opaque types (continued)

- Example 4:

```
module child #(type T = int);
      int T2;
      class C extends T;
            T2 x;
      endclass
endmodule
```

- Mentor would call this an error. In order to permit references to a type inherited from an opaque base using (essentially) existing forms, Mentor would require either:

```
typedef super.T2 T2;
typedef T::T2 T2;
```

  Either form makes the assumed type invariant regarding the base type explicit. This closely resembles the form for being able to use a type from an interface.

Opaque types (continued)

- It is not clear whether Synopsys would call the original form an error.
- The suggestion on the reflector would be to use:
  ```
      type(T2)
  ```
  in place of each reference to T2. Since the type operator can be used with both types and variables, if T2 did not exist in the base class, the type of child.T2 would be used instead.
- It is not clear whether there is any way to handle inherited types in a dynamic approach without introducing some form of a more restrictive rule or the use of the type operator as suggested or by adopting some more explicit form as suggested by Mentor.

Opaque types (continued)

- The type issue can also occur in the context of inline constraints.
- Example 5:

```
module child #(type T = int);
        int T2;
        T c = new;
        initial c.randomize with { x < T2’(y); }
endmodule
```

Summary of Mentor's position on Opaque Types

- Mentor believes that a more explicit approach to dealing with resolution through opaque types leads to more clear designs with simpler to reason about invariants. There are clear paths along various fronts for creating extensible, general syntactic forms to make design invariants explicit and composable. We believe that this will lead to a fundamentally stronger language.