Virtual Interface and Interface Port Mapping to Interfaces with Multiple Logical Libraries

Chapter 33 of 1800-2009 discusses design configuration and introduces the concept of logical libraries. Organizing cells into logical libraries allows you to have cells of the same name in multiple logical libraries. Configurations are a way of controlling which cell gets selected for every instance in the design.

It is not clear, however, how interface ports and virtual interface variables get bound to the cells in logical libraries. Consider this example:

```
interface intf1;
endinterface

interface intf2;
endinterface

module m(intf1 i);
endmodule

module top;
   intf1 i();
  m m(i);
  virtual intf1 vi = i;
endmodule

config topcfg;
   design work.top;
   default liblist work;
   instance top.i use work.intf2;
endconfig
```

Without the configuration this design has an instance of interface ‘intf1’, the module ‘m’ has an interface port of type ‘intf1’ and there is a virtual interface variable of type ‘intf1’. The configuration assumes that all these cells have been placed in a logical library called ‘work’. The configuration changes the instance top.i to use the interface ‘intf2’ instead of ‘intf1’.

There is currently no way for the configuration to control the type of the interface port or the type of the virtual interface variable. With the configuration this design would become illegal, since the interface port hookup and virtual interface variable assignment would be illegal.

In this example I have used interfaces with different names in the same library for clarity, but this could easily be interfaces of the same name in different libraries. You would have the same problem. You can control which library the interface instance instantiates, but it is very unclear how the interface port or virtual interface variables are mapped to the cells in the libraries.

Possible solutions

1) Interface ports are bound to the interface from library of whatever interface instance is connected.

One proposal for an interface port is to bind the interface port to whatever interface instance gets connected to the port at elaboration time. The check that the connection was legal would only be that the name of the interface was the same. The example above would still be illegal, since the names of the interfaces were different in that example, but in the case where the names were the same, it would be legal.

You cannot, however, do the same thing for virtual interface variables. In the above example, there is an initial assignment to the virtual interface variable from the interface instance, but in the general case this is not true. The virtual interface variable may be a task or function port. You would have to analyze the call graph for the entire design to figure out what instance is passed to that port. This kind of static call graph analysis is not possible in a language with virtual functions.

This solves the problem of interfaces of same name in multiple libraries for interface ports, but not virtual interfaces.

2) Use configuration 'cell ... use ...' clause for virtual interface and interface ports

Configuration can contain 'cell ... use ...' statements. In the earlier example instead of:

```
  instance top.i use work.intf2;
```

We could have written:

```
  cell intf1 use work.intf2;
```

The current LRM does not specify whether these 'cell' rules apply to virtual interfaces or interface ports. We could add language to specify this. By using different configuration files for different parts of the design, one could get the same interface name to map to different libraries in different parts of the design, but not in the same module.

The big drawback of this is configurations do not apply to packages. Package names are bound to packages at parse time (but the LRM does not actually say how this happens). Packages are not instances in the design that can have their own configuration. Customers are currently using virtual interfaces in packages.

3) Virtual interface binding specified at parse time

All other data types have to be at least forward declared at parse time. One could say that the virtual interface types are bound to libraries at parse time.

There are several problems with this. First the LRM does not say anything about how binding happens at parse time. Packages are bound at parse time. Given the library map file mechanism discussed in section 33.3, it is possible to put two packages of the same name in different files and assign them to different logical libraries. The LRM does not say how a reference to a package name would be bound at parse time.

All real tools that support logical libraries probably have a mechanism for resolving package references at parse time, but they are probably different. We could say that virtual interface types are bound to libraries at parsing time without specifying exactly how this happens, and leave it to the vendors, as we have apparently done with packages.

4) Enhance configurations to allow binding of virtual interface types.

We could add some new syntax to configurations to allow binding of virtual interface types. Virtual interface types do not always have names, so it would be difficult to provide something like the instance name rules. One possibility would be to allow instance name like rules only to apply to named typedefs. Note that even variable names would be confusing because associative arrays can have multiple virtual interface types as key types.

The biggest problem with this is that we would have to extend these virtual interface rules to packages, which is highly undesirable. Packages were intended to encapsulate types and that packages could be precompiled. If the configuration specified at elaboration time can reconfigure a package, this makes separate package compilation impossible.

Conclusion

If we are going to solve the virtual interface binding problem without introducing new data types like “pure interfaces” or “stand-alone modports”, then we need to specify some reasonably flexible way of binding them.

Of the ideas discussed here, I like the idea of specifying the binding of virtual interfaces at parsing time and allowing interface ports to connect to any interface of the same name regardless of library. This will allow packages to be separately pre-compiled on their own and without a configuration.

There probably are existing designs where no binding is available at parsing time. Even if the interface has not been parsed yet, tools could provide some mechanism for specifying how to bind virtual interfaces at parsing time, although the actual binding would take place when the code was compiled.

Since the current LRM does not discuss how binding of packages happens at parse time, we would either have to leave this to the vendors or attempt to define some minimum capability for parse time binding.

This would mean that a user could not parse system verilog containing virtual interface types and then configure the virtual interface types at elaboration time to selected different interfaces in different compilations. System Verilog already has type parameters which can be used on modules and classes in packages to reconfigure the code at elaboration time. It is not clear that we need to provide more than this for virtual interfaces.