Response to the Cadence response to SystemVerilog.

To TCC-Chair Vassillios Gerousis, and SystemVerilog committee members.

Cadence presented a technical response to SystemVerilog on December 4th in San Jose, CA (attached as pdf file). We, as key contributors to the SystemVerilog language, have provided below a technical clarification response to the questions and issues raised.

SystemVerilog is the work of many excellent contributors, and these clarifications we hope will help participating members understand that SystemVerilog is building a solid extension to the Verilog standard. Users will benefit tremendously with the advancement, completeness and standardization of this language. We hope that Cadence will join us constructively in this effort.

Phil Moorby, Synopsys Scientist.
December, 2002

Slide 4 - Criteria – Retain Style Of Verilog 1364CV

Implicit declaration is not a general Verilog style, only scalar wires can be optionally implicitly declared.

Forward declarations of structural types was a de-facto standard from other HDL's.

Verilog does do type checking and has been improved without becoming cumbersome, and maintains obvious and useful implicit casting.

5 - Criteria – Scalability/InteroperabilityCS

To support this goal, SystemVerilog introduces the clock domain construct, properties, expects and sampling semantics common to both assertions and test bench, using identical syntax and semantics.

10 - Data Types - Logic

Back annotation is done at the implementation netlist level. SystemVerilog gets synthesized to a regular Verilog netlist. There is no need for SDF back annotation on abstract data. Attributes can be used as a clean way to provide additional information in the synthesis netlist output.

11 - Data Types - Composite Structures

Implementation moving from interfaces to RTL is a misleading comment. The major benefit of interfaces is that they can be used as containers of low-level wires, with RTL code, at the synthesizable implementation level of port connections.

12 - Data Types - References/Pointers

References/pointers can also have a massive potential positive impact on simulation performance. The challenge for simulation performance is to do with that of copying and being sensitive to large encapsulated data structures, and pass-by-reference rather than pass-by-value helps to solve this.

References are valuable and they are an integral part of SystemVerilog 3.1. However, they have been carefully crafted to avoid the pitfalls of C/C++. References are strongly typed, and memory is automatically managed. These two conscious design choices help shield users from common problems like dangling references and premature de-allocation. In a concurrent programming environment, it is impossible for any user (no matter how sophisticated) to know for certain when other processes no longer access a particular block of memory.

13 - Data Types – Higher-level Structures

The Goal should always be to make the use of the language as easy as possible for the end users of the language. Verilog has proven that building in agreed upon key constructs into the language is a more efficient and effective way to go.

The donation includes Mailboxes, Semaphores, and Lists. All three are implemented as classes. Mailbox and Semaphore are built-in while Lists are provided as a standard package written in the language itself.
Semaphore IS a fundamental synchronization primitive and should be built into the language, and any textbook on Operating Systems or Concurrent Programming includes a description of semaphore as a fundamental synchronization object.

Mailbox is a fundamental building block for all message-passing systems (whether an OS or a piece of hardware). It is also the most common communication mechanism used by test benches, and therefore deserves to be standard, unique, and built-in. There are multiple advantages to building these objects into the language:

1- Uniformity: Same behavior everywhere.
2- Ease of use and debug. All tools work the same.
3- Users WANT these particular objects built-in.

14 - New Features - $root

The concept and need for $root has nothing to do with the problems of compiler directives. There has always been a hidden concept of $root in Verilog, and it has now been made explicit.

Potential for non-determinism is inherent in all HDL's and this has nothing to do with the $root concept. Allowing flexibility in what can be declared in $root enables simple things to be simple. Large development teams should always enforce good coding rules with advanced lint-like utilities.

Adding 'use' or 'with' contributes to the name collision problem in large development teams, and provides little value.

15 - New Features - Interfaces

First off, it is a matter of opinion that these three views of the utility of interfaces are unrelated. It is our (and our customers') experience that interfaces allow the communication between blocks to be modeled effectively, in an analogous manner to how modules are used to describe functionality. If one only considers the communication between blocks to be a bundle of signals, then using ports and structs would be sufficient. However, as communications become more complex and functionality becomes important, simple structs are not sufficient. Rather than try to add additional features to structs, which are used also to model things other than interconnect, we realized it made more sense to create the interface construct.

If one considers the question of design-by-refinement, a popular next step above signal-level interfaces is transaction-level interfaces, which is why the interface construct needs to be more than just a bundle of wires, and another reason why simple structs are not sufficient. As you point out, tasks are an existing way for transactions to be modeled, and the management of where tasks are defined and used is an important feature of interfaces. For system-level modeling, the question of signal resolution and multiple drivers is a low-level detail that should not have to be addressed, so the issue you raise does not diminish from the usefulness of interfaces at this level.

The power of interfaces allows for the refinement of the "slave" side of the interface (as an example), without changing the "master" side of the interface at all.

The most useful overall practical feature of interfaces when considering design-by-refinement is that, when two modules are connected via an interface, one module may be refined to a different level of abstraction without affecting the module it is connected to, nor the parent module that instantiates both (other than perhaps changing the names of what gets instantiated, but that can be controlled via configurations). This is a useful feature that simply cannot be done

either in Verilog today nor without creating a new construct that is more than just a struct.

Our customers tell us also that there is benefit in distinguishing between pure block-level functionality and interconnect, which is why adding modport-type information to modules and allowing modules to be passed through ports is not an effective solution to the problem. Adding this functionality to modules would make them more complex and would make it much more difficult to determine easily whether one is modeling a block or the communication between blocks.

16 - New Features - Redundant Additions

The always_{comb,ff,latch} keywords do add important new semantics. For instance, always @(*) does not model the initialization of combinational logic correctly.

Iff has already been proven by users to be a very useful new feature.

17 - New Features – Functions/Tasks

System functions have always been allowed to have inout arguments, and it is desirable to make Verilog coded functions compatible with them.

The essential distinction between tasks and functions is that functions are not allowed to have timing controls, and thus never suspend.

18 - New Features - Verilog 2001 Conflicts

Using an always statement to implement synthesizable combinational logic does not separate the semantics, but is an integral whole.

20 - C-Interfaces - Coverage

Coverage is a well accepted methodology, and has been for several years. Standardization of coverage access has been demanded by users. It is time to standardize and make it easy for users and other EDA tools.

The C-level access functionality is not a methodology specific API, but a general mechanism to enables users to bring their own (or legacy) C code into Verilog in a standard and much easier way than currently done by VPI.

21 - C-Interfaces – Assertion API

The assertion API, which is being proposed as an extension to VPI, does not rely on any specific construct. It provides features such as control to start/stop an assertion and the information on property or sequence.

22 - VeraLite Donation

There is no new different language, SystemVerilog will always be an extension to Verilog in the spirit of Verilog with backwards compatibility.

Constraints and assertions have been unified.

The random constraints document is Synopsys' response to a public request by a sub-committee – a request to which no other vendor, including Cadence, chose to respond. As with all aspects of SystemVerilog, if Cadence has technology to donate, the open exchange of ideas will be welcomed.

25 - Recommendations – Data Types

We hope readers will recognize that these purely abstract and vague discussions imply resetting years of hard work by many good people and committees.

26 - Recommendations – Language Fundamentals

SystemVerilog does not change the fundamentals of Verilog!

28 - Recommendations – Summary

SystemVerilog does maintain the style of Verilog.

The problem of introducing new keywords is always a matter of careful judgment and opinion. A simple front-end parsing switch of keyword tables can easily provide perfect backwards compatibility.

SystemVerilog is backward compatible to Verilog by design. Testbench and Assertions constructs are extremely useful additions to System Verilog language for verification users. Many sources of expert efforts have gone into making assertions and testbench features an integral part of SystemVerilog. SystemVerilog is based on proven and efficiently implemented languages.