SystemVerilog Chairs and Champions Document

Version 3
15 May 2003

SystemVerilog Committee Chairs

Committee	Role	Name
SV-AC (assertions)	Chairperson	Faisal Haque
	Co-chairperson	Stephen Meier
SV-BC (basic – 3.0 review)	Chairperson	Johny Srouji
	Co-chairperson	Karen Pieper
SV-CC (C APIs)	Chairperson	Swapnajit Mittra
	Co-chairperson	Ghassan Khoory
SV-EC (Testbench/Extensions)	Chairperson	David Smith
	Co-chairperson	Stefen Boyd

SystemVerilog Champions

Committee	Name
SV-AC (assertions)	Surrendra Dudani
SV-BC (basic – 3.0 review)	Dave Rich
SV-CC (C APIs)	Joao Geada
SV-EC (Testbench/Extensions)	Arturo Salz
SV BNF	Stefen Boyd

SystemVerilog General Chair

Vassilios Gerousis

Table of Contents

Summary

This document contains both a summary of the process response to the Cadence negative ballot and a detailed technical response.

Cadence was a participant during the 3.0 and the 3.1 development and review process. Many of the comments that are raised in their technical response were never raised during the development or review process. Those issues, from the review document, that were raised either resulted in a committee vote that voted as the LRM stated or their was no solution proposed or developed by the committee (due to lack of support or time).

It is instructive to note that many of the issues raised are related to the already approved 3.0 version of the LRM (that Cadence also participated in). The ratio is 42% from 3.0 and 58% from 3.1.

Process Comments

a- Cadence members were active in all discussions, made numerous positive contributions, and had many chances to propose their views (and used those chances).

b- All issues were discussed in open and productive discussions and documented in the meeting minutes (available on the committee websites).

c- Cadence's votes on the vast majority of the issues were in line with the agreed-upon decisions. The number of times they voted against a given decision was much less (we have the data).

d- Voting process for most issues was not by company but by individual contributor and based on the technical assessment of each member. There were several cases were members of the same company voted differently and in line with members from another company (where technical opinions were similar).

e- All voting followed guidelines established within the committees and approved by the committees.

f- Most issues which were raised by a committee member, were backed up by a written proposal which was submitted to the reflector before the meetings. Committee members had the chance to read, comment, object or raise a counter proposal through email and during the meeting.

g- In cases where committee members felt that a more thorough investigation discussion was required, a proper action item was taken including tabling the issue to post LRM3.1. One example is SV-BC-21-1, where the committee suggested to table the issue, even though it has previously passed by the committee where cadence members objected to this issue. Their technical opinion was clearly respected.

Technical Comments

Quotes from the Cadence document are in "blue".
Quotes from the SystemVerilog LRM are in "purple"

Each item is labeled as either **(3.0)** or **(3.1)** depending on the SystemVerilog version addressed by the item. This is an important distinction because the 3.0 standard was approved by the technical committee (including Cadence) in the 3.0 process and sent to the board where it was ratified by Accellera one year ago. Nonetheless, almost half of the technical issues raised by Cadence are strictly 3.0 items:

3.0 items: 40 (42%)
3.1 items: 54 (58%)

3.2 Section 2 – Literal Values (3.1)

"Note there is no mention of how literals relate to the new class types at all."

The only new literal for classes is **null**. It should be mentioned in this chapter, but it is described elsewhere, so this is only an editorial issue.

3.2.1 Array and Structural Literal Syntax (3.0)

"…there is no BNF given for the syntax of an array or structural literal."
The array and structure literal syntax is given in section A8.1.

"The examples given use {} as in 'C' which creates a syntactic ambiguity with the existing 1364 concatenation operator."
There is indeed a syntactical ambiguity. However, the rules to disambiguate it semantically are clearly specified in the text.

3.3 Section 3 - Data Types (3.0)

"Allows data types on both variables and nets"
Verilog does not have the same data types for variables and nets. For example, real and integer have no corresponding net types. SystemVerilog follows in this tradition. Nets are only available with built-in resolution functions. It appears that what is proposed here is the VHDL style, and that is a proposal that was not raised during the 3.0 discussions. Uniform data types for both variables and nets was raised as a concept during the 3.1 discussions, an actual proposal to define them was never made.

3.3.1 Data Type Syntax (3.0)

"Instead, we would prefer to see these necessary types defined as standard-defined types using the same extension mechanism as is available to users."
Some new data types cannot be made from existing types, for example **bit, string, shortreal** and **void**. The notion of standard-defined types not part of the language is alien to Verilog. Leaving the actual types unspecified simply brings a seeming generality to the language with no real benefits to the user. Certainly, vendors are free to provide standard-types beyond those defined by the language; the current set is designed to facilitate mapping C code and algorithms into SystemVerilog.

3.3.2 Integer Data Types (3.0)

"It also has the same failing the 'C' types do in the presence of different width data types on different compilers/operating systems."
Unlike C/C++, but like Java, the size of **int** is defined as 32 bits to avoid portability problems with SystemVerilog, at least when not using the DPI. The same is true for the **shortint** and **longint** data types. The DPI does define rules specifying how SystemVerilog types are represented on the C side.

3.3.3 Real Data Types (3.0)

'An excellent paper was presented at DVCon 2003 on this topic and should be considered as a mechanism for extending the floating point number system."
The addition of **shortreal** allows numerical algorithms expressed in C to be written in SystemVerilog without a change of behavior, as well as to facilitate data exchange between C and SystemVerilog. While arbitrary precision fixed and floating point may be useful enhancements to SystemVerilog, such a proposal was never officially raised in the technical committees.

3.3.4 Void Data Type (3.0)

"Cadence believes that the SystemVerilog extensions to functions to be used in this way are not a good extension of Verilog"
The objections expressed here were not raised in the 3.0 committees; instead it was approved by the technical committee in the 3.0 process and sent to the board.

"If these extensions are not made, then the void type is unnecessary and its removal eliminates yet another keyword."
In 3.1, the distinction between a task that consumes time and a function that does not consume time is necessary for the DPI. A "void function" is needed to specify a C function that returns no value and has no timing.

3.3.5 String Data Type (3.1)

"If strings are being added as a fundamental data type then operators or system tasks should be defined to operate on them".
Operators are indeed defined for the string data type, and they are clearly listed in table 3.2.
The relevant system tasks in which strings can be used are defined and the LRM contains multiple examples of them (i.e., section 4.10, 7.11, etc). The method notation was chosen to extend the language without polluting the keyword namespace and without the problem of system functions being redefined by PLI (an undesirable feature of functions on built-in types).

"strings should be defined as a new standard-defined class".
Strings are useful as a built-in data type in many other languages, including Vera and Superlog. Defining them as classes would preclude their usage with operators (as defined by table 3.2) and limit their usability.

"…the operators on strings should be defined as system functions".
The method notation on non-classes is equivalent to VHDL built-in attributes.

There is also not sufficient integration of the string type into the direct programming interface. The relationship to a 'C' char * is not sufficiently specified. For instance is a SystemVerilog string terminated by a '\0' character?

There are two different questions here. The first is what the representation is within SystemVerilog. This is implementation dependent and not visible to the user. The second is what access is provided to SystemVerilog strings from the DPI. Clearly table D-1 indicates that this is handled as a C "char *" which has to be NULL terminated to be useful within C. This seems clearly stated in the LRM.

3.3.6 Event Data Type (3.1)

"These handles are a hybrid between the dynamic behavior now defined in classes and the statically allocated behavior of all other data types."
There was discussion about distinguishing the static and the dynamic events in SystemVerilog, for instance by using "**ref event**". Eventually the committee concluded that the balloted use would be easiest for the user. Events are special, anyway, even in Verilog: they are not synthesizable, nor are they recognized by the PLI, and they cannot be used in any system tasks.

3.3.7 Enumerated Types (3.0)

"… creating any guarantee that the opposite assignment will always be legally in the range of an enumeration will create an implementation burden to guarantee that non-consecutive encodings of an enumeration value are checked on assignment".
Because there are static and dynamic casts, the burden incurred by checking a cast from an integral to an enumeration is now (in 3.1) under user control.

"methods … should be adopted for all data types and retrofitted into existing Verilog types".
Generalizing the method notation to other data types may be a useful future enhancement that does not invalidate its use now.

3.3.8 Structures and Unions (3.0)

"If the original value contained no 'X' or 'Z' values then you should get the same value out. This special case poses an extreme implementation burden and just does not make sense"
The implementation burden mentioned for unions is only for simulation, and it is in fact very small:
Writing to a 2-state member should clear the *bval* values; reading from a 2-state member should 'and' the *aval* with the complement of the *bval*. An additional machine-level instruction is hardly extreme!

"… the lack of a method of dynamically allocating structure objects"
It is indeed useful to be able to dynamically allocate a structure. However, it can be easily done by putting it in a class or a dynamic array. Dynamic allocation of structures requires the addition of pointers to the language, as well as reference (address of) and de-reference operators, and manual de-allocation. These additions would complicate the language for the users, affect performance by

precluding other optimizations, and increase memory use by limiting the ability of the automatic memory manager to re-use memory.

"complete description for all complex data types should be added in the 'C' API section"
The compatibility with C should indeed be included in the DPI section, which is a minor editorial issue.

3.3.9 Classes (3.1)

"We believe that if an allocation mechanism for structs is created and a semantic for static classes is defined, then the two types can be merged into a single construct "
This issue was debated extensively in the technical committee, and it was balloted as defined. The main reasons for keeping classes as only dynamic, and structs as only static or automatic, is addressed in 3.3.8 above: ease-of-use, no need for explicit pointers, reference/de-reference operators, manual memory de-allocation, and no aliasing problems that will preclude existing optimizations, and finally facilitate effective automatic memory management.
The proposed paradigm of pointers to static or automatic objects is like C and needs user control of memory, which is a source of many bugs. In addition, the committee's choice leaves **struct** as part of the synthesizable subset (a very useful feature), whereas **class** is not intended to be synthesizable but useful for testbench and system level modeling.

3.4.1 Longest Static Prefix (3.0)

"The final resolution was that this concept was left unspecified; this will create ambiguous interpretations in different implementations which will hinder vendor interoperability."
A definition was proposed, which is available at http://www.eda.org/sv-bc/hm/0580.html. However, Cadence opposed its inclusion into the LRM.

3.4.2 Dynamic Arrays (3.1)

"If both dynamic and open array terminology stay in the standard and use the same syntax, users will be terribly confused."
The 3.1 LRM clearly states:

> "An open array is like a multi-dimensional dynamic array formal in both packed and unpacked dimensions, and is thus denoted using the same syntax as dynamic arrays, using [] to denote an open dimension."

They are the same thing when crossing the C/SystemVerilog boundary, hence, no confusion.

3.4.3 Associative Arrays (3.1)

"… content-addressable memory where the items being stored may be any arbitrary type would be extremely helpful"
An example of a content addressable memory that does not contain one of the permitted index types would be useful. The authors could not find one.

"The traversal order of iterating across all the items in the array is specified as "deterministic but arbitrary". … We believe that this will be a major problem for customers utilizing more than one vendor's simulation tools"
Leaving the order of class indexes undefined reduces the implementation burden, and no alternative that addresses all the issues was ever proposed. The balloted definition makes separate objects unique, regardless of their content, and that is useful.

3.5.1 Data Declaration Syntax (3.0)

"the use of **static** … It is always possible to simply declare the variable outside the function/task in a module and then reference it in the function/task.
Eliminating the **static** keyword would inhibit encapsulation of data, which is considered a good programming technique. Moreover, the need to declare static class data members makes this keyword necessary (a need that was not foreseen by the IEEE-1364 2001 committee).

3.5.2 Constants (3.0)

"The specification explicitly allows the use of hierarchical names when specifying the value of a constant but does not properly guard against circular references between constant initializations."
The LRM explicitly states:

"The parameters, local parameters or constant functions can have hierarchical names."

The inclusion of hierarchical names therefore only applies to parameter, local parameters, and constant functions, but not to other constants or genvars. Hence, their inclusion can never result in circular references. Circular references are an existing issue in Verilog 2001.

3.5.3 Variable Initialization (3.0)

"The problem with this change of initialization is that in the Verilog-2001 method an event is generated. In the SystemVerilog method, no event is generated. This difference, as explicitly given in the LRM above, has a severe impact on gate-level models and the behavior of continuous assignments, not procedural contexts as argued."
A previous response by Accellera has already addressed this issue. The latest example provided by Cadence seems to imply that in order for the evaluation of a continuous assignment, an event on the RHS must be generated, including at time 0. However, a continuous assignment does not require an event on its RHS. For example:

```
module init;
        wire w;
        assign w = 1;
        initial  #1 $display( "wire is %b", w );
endmodule
```

The code above must also show '1' as the value of wire w, and no event is generated by the constant 1. At least two separate implementations of the SystemVerilog initialization semantics exist and both are 100% backward compatible with respect to continuous assignments.

3.5.4 Automatic Variables (3.0)

"Having automatic variables in any other static context just does not make sense."
Having automatic variables in a static context makes a lot of sense, and is clearly useful. For example:

```
always begin
        bit [7:0] arr;
        for( int j = 0; j < N; j++ ) arr[j] = j;
        … // do something with arr
end
```

In the code snippet above, the variable arr is static, but to be meaningful and useful, the 'j' variable in the for loop must be automatic; otherwise a static variable declaration would be initialized only once, thereby executing the loop only once.

Note that for the purpose of storage allocation, an automatic variable declared inside a static context can be treated as a static variable. The statement seems to be confusing storage allocation and initialization.

3.5.5 Variables in Unnamed Blocks (3.0)

" …the behavior of Verilog such as VCD dumping, $display (%m) etc. is not described with respect to these variables."
This may be a minor omission that was nonetheless approved by the technical committee (including Cadence) in the 3.0 process and sent to the board over 1 year ago.
An implementation will surely create synthetic names for these blocks, and perhaps the committee felt that standardizing such a naming convention was unwarranted.

" …the original intent for allowing variables declared in unnamed blocks was to be able to hide data declarations from scopes above."

The intent for this enhancement was to not require users to invent artificial labels when all they want is to declare a local variable.

"…classes which have the possibility of declaring public, protected or private data declarations. These two methods of declaring private data should be reconciled."
Classes provide for data encapsulation via type declarations, which is very different from the lexical data encapsulation provided by unnamed blocks. There is no need to confuse the two.

3.6 Section 6 – Attributes (3.0)

"… this section is probably completely unnecessary as attributes should just be defined in the syntax for interfaces and modports"
This section introduces a new feature: the default attribute data type, which needs to be documented. It should not be explained via BNF alone.

3.7.1 Assignments in expressions (3.0)

"Cadence believes this sort of shorthand is borrowing some of the worst C features of 'C'."
This is a matter of opinion. The shorthand was approved by the technical committee (including Cadence) in the 3.0 process and sent to the board.
What is certain is that these features are commonly used.

3.7.2 Increment (++) and Decrement (--) in expressions (3.0)

"… inclusion in expressions creates syntax that is easily interpreted ambiguously."
A proposal was made to the committee that would remove this ambiguity, but that proposal was rejected by Cadence and other committee members.

3.7.3 Built-in methods (3.1)

"A unique namespace for the methods defined by the standard should be created so that they will not conflict with user-defined methods."
Built-in methods apply only to data types, not classes. Thus, there are no user-defined methods that could conflict with them.

"The standard should prefix all standard-defined methods with a unique character such as '$', or '_'."
While this may be a matter of opinion, no such proposal was made to the committee. The consensus within the committee was to leave identifiers prefixed with '$' as special PLI-enabled identifiers.

3.7.4 Literal expressions (3.0)

"No BNF syntax description for these expression types is given in the LRM."
The array and structure literal syntax are given in A8.1.

"We expect that when BNF for these expressions is provided that there will be a syntactic ambiguity between these three classes of initializers and the existing Verilog concatenation and replication operators."
There is a syntactical ambiguity, but the LRM provides rules to disambiguate it, hence, this is a non-issue. In fact, some of those rules are mentioned in the Cadence document, for example:
"Only by examining the left-hand side of the assignment can the context be determined."

3.8.1 Unique/Priority Keywords (3.0)

"This is a case where keywords are being overused."
This is a case where keywords are being used to modify the simulation and synthesis semantics, therefore, a keyword is an appropriate mechanism.

"Verilog already contains a mechanism whereby information can be associated with a given statement. That mechanism is attributes."
It is generally accepted that an attribute should not modify the simulation semantics. This proposal, which was never made during the 3.0 standardization process, would violate that principle.

"The **unique** and **priority** keywords are the first examples thus far that add lint-like capability into Verilog in the form of keywords."
This indicates a misunderstanding of these keywords. These are not lint-like capabilities; they modify the simulation behavior.

3.8.2 Performance of Unique (3.0)

"A second concern with the **unique** keyword is that it can have a potentially severe impact on simulation performance."
The use of the **unique** keyword with variable labels can indeed incur some run-time overhead. However, this is an extremely valuable feature since it can trap violations of the implied functionality. Without this feature, users must code these check themselves, but they may not have access to the optimization techniques available to the tool.

3.8.3 Final blocks (3.1)

"Cadence believes that the specification of these blocks is incomplete. … Final blocks should be defined at a minimum as not allowing assignments to any object to which any other construct is currently sensitive."
This may be a worthwhile clarification that should have been discussed in the technical committee.

3.9.1 Ordering requirement on final blocks (3.1)

"We believe that if the language can not define an ordering, then no mention of the ordering should be made at all."
This is a sensible editorial issue. The sentence that defines an ordering should be removed from the LRM.

3.9.2 always_comb, always_latch, always_ff as semantic checks (3.0)

"… using a keyword for this purpose is inappropriate. It adds unnecessary keywords"
The use of these as keywords is appropriate because **always_comb** and **always_latch** modify the simulation and synthesis semantics of an always block. The use of a keyword for **always_ff** preserves the regularity of these constructs.

"Once again, we would return to the existing attribute mechanism to associate additional information with a statement. That is exactly what attributes are for …"
It is generally accepted that an attribute should not modify the simulation semantics. Cadence's proposal, which was never made during the 3.0 standardization process, would violate that principle.

3.9.3 Always_comb sensitivity (3.0)

"Cadence believes that the implementation complexity of this additional sensitivity check is not justified by the expressive power added to the language."
This additional sensitivity was added at the specific request of a user. It was extensively debated in the 3.0 standardization technical committee, and, in the end, it was approved by the technical committee (including Cadence) in the 3.0 process and sent to the board.

We would prefer to simply produce a warning when we detect functions with side-effects and if a user calls a function with side-effects from an always_comb they will have a potential source of error.
This lint-like check is a feature that vendors can provide. It need not be part of the language.

3.9.4 Always_comb overlap with @(*) (3.0)

"We believe that the sensitivity should be expressed in the sensitivity list as is done with the @(*) syntax in 1364 and that the lint-like capability should be added as an attribute as defined above."
As stated above, the **always_comb** construct does affect the simulator semantics; it is not a lint-like capability.

3.10 Section 10 - Tasks and Functions (3.0)

"SystemVerilog makes major changes to functions by bringing a significant amount of 'C'-like content into Verilog. 'C' is a language that only has functions, there is no concept of a task. Simply modifying functions semantics significantly "because 'C' allows it", is not sufficient justification."
These capabilities were not added capriciously, "just because C allows it". They allow direct interaction between SystemVerilog and C.

3.10.1 Function output and inout argument modes (3.0)

"Functions execute in zero time and, except for hierarchical references, can not have side-effects; their only effect is through their return value."
This is untrue. Functions may exhibit side-effects without using hierarchical references.

"This change blurs the line between tasks and functions in Verilog and adds significant opportunity to have function side-effects."
As noted above, functions already allow side effects. This enhancement does not change that fact.

"If a subprogram is supposed to modify multiple arguments then it should be written as a task or an automatic task."
Unfortunately, this coding guideline will not work with the DPI as specified in 3.1.

3.10.2 Functions as statements (3.0)

"A function is a mechanism in Verilog to create an operand that can exist in expressions. … There is simply no need for this change other than to be more 'C'-like."
Some C functions return status that may be irrelevant to SystemVerilog (e.g., fprintf). SystemVerilog thus provides a convenient way to discard such return values when calling C functions via the DPI.

3.10.3 Void functions (3.0)

"Void functions are only necessary if functions have output arguments and therefore act like tasks."
This is untrue. A void function that displays a message is a perfectly good example of a function with only input arguments and no timing.

3.10.4 Implicit task and function lifetime (3.1)

"The lifetime of a task is a property purely of the task and the syntax should be limited specifically to the task, not inherited from elsewhere."
In general, testbench programs require that the lifetime of their tasks and functions be automatic. Cadence requested that the default always be static and hence the automatic keyword was needed for programs and for regularity it was added to modules as well. This item was balloted and Cadence supported it.

3.10.5 Lack of shared library support in DPI naming (3.1)

"The specification of the name for a function imported from 'C' to SystemVerilog should include a component that represents a shared library name."
This may be a fine enhancement to the existing DPI. However, no such proposal has been made to the committee.

3.11.1 Additional common keywords (3.1)

"The definition of classes adds the keywords **this**, **new**, **super**, and **class**. These are all common, short English words that will most likely conflict with identifiers in existing designs."
The keywords **this, new,** and **class** were added to C by C++, and the added functionality was more important than the conflicts that arose from these additional keywords.

"A careful unification with records and substituting operators for some of these could eliminate most if not all of these keywords."
Using the same keywords as used by C++/Java/Vera significantly adds to the acceptance and ease of use of the language. Moreover, no operators that could substitute these keywords and conveyed the same concepts as clearly were ever proposed in the committee.

3.11.2 Parameterized class types (3.1)

"Classes allow the parameterization of the type of objects declared within a class. This is similar to C++ templates and Ada generic packages. Although this does add significant modeling power, it also adds extremely high complexity in implementation."
This should be no more complex than parameterized modules. At any rate, the additional modeling functionality gained by users is well worth the complexity in the implementation.

3.11.3 Differences in struct and class (3.1)

"Unification could make both be static and/or dynamic, essentially unifying them as a single language construct."
This same argument was raised before in sections 3.3.8 and 3.3.9. The reasons for having made this distinction are covered in those sections.

"A struct would simply be a class with no inheritance or methods specified."
If that were the case then users and tools would be unable to distinguish synthesizable from non-synthesizable classes, and that will lead to confusion. The current paradigm makes these distinctions explicit and clear. Note that even C++ did not initially attempt to merge class and struct in this manner.

"The second is partly untrue. Only packed structs have the capability of being assigned based solely on width and this is a feature of a packed struct."
That is incorrect. A union may contain both packed and unpacked members. The LRM states:

> "A packed union shall contain members that must be packed structures, or packed arrays or integer data types all of the same size (in contrast to an unpacked union, where the members can be different sizes)."

Thus, assignment to the packed member can indirectly assign to the unpacked members.

"A class whose data elements are packed in the same way would be a powerful modeling capability for vector-like objects with unique field names."
Indeed it is a powerful modeling capability, which is why classes are allowed to contain unions.

"Cadence views the fact that classes are strongly typed in this way as a problem, not a benefit."
As covered earlier, classes are strongly typed to maintain the integrity of the memory. Strongly typed classes shelters users from the memory problems that plague many C programs. An alternative that addresses those issues was not proposed by Cadence.

"For unpacked data, structs are not assignment compatible based purely on width, but neither are structs assignable to classes even if the data members are identical; each member must be individually assigned."
This is easily done by placing the struct inside the class, same as in C.

"Dynamic allocation of other SystemVerilog types could follow this exact same paradigm and have exactly the same level of safety."
This is untrue. As explained in sections 3.3.8 and 3.3.9, the existing definition does provide the level of safety required by SystemVerilog. An alternate scheme that would provide similar ease-of-use and memory protection was not proposed by any other member, including Cadence.

3.12.1 Limitation to randomizing classes (3.1)

"In SystemVerilog, constraints and randomization can only be tied to objects of a class type. Cadence believes that it should be possible to constrain and randomize any variable in a SystemVerilog design."
There are many reasons for tying randomization to class types, among them:

1. If arbitrary data is constrained then how do users indicate that a particular group of constraints is to be solved and data modified? No other proposal that addresses this issue was made.
2. Classes allow constraints to be layered, using the inheritance mechanism.
3. Classes allow data and constraints in one object to be tied to the data and constraints in other objects.
4. Declarative constraints are best specified in a type declaration (class).

A proposal to extend constraints beyond classes was not provided.

"Note that limited randomization is heavily used today on non-class variables through $random(), adding the ability to constrain these in conjunction with $random() would add signification power."
A proposal to tie constraints and procedural randomization calls was never proposed to the technical committees. If such a mechanism exists and is deemed useful to users, it should be considered as a future enhancement to the existing, proven mechanism.

3.12.2 Rand, randc in a class declaration (3.1)

"When a class and the members of that class are declared in SystemVerilog, the individual members that are to be randomized must be declared with the keyword **rand** or **randc**. We believe that statically associating the ability to randomize with the declaration of the type is a mistake."
The random nature of variables declared **rand** or **randc** can be turned on or off dynamically.
Conversely, removing this information from the type declaration would force users to use a procedural mechanism to enable all the random variables, at least once for each instance of a class. As currently defined, users specify this information only once (in the declaration).

"We would prefer to have the ability to randomize a particular variable dynamically as is done with constraints."
Both variables and constraints can be dynamically turned on or off using very similar mechanisms: constraint_mode and rand_mode method calls.

"When this was discussed in committee the rationalization was that there are compile-time optimizations that can be applied by knowing in advance that these particular members would be randomized. Similar information could be derived from that fact that members were actually constrained or passed to system tasks that perform the randomization."
It is not at all clear that such information can be derived even with very sophisticated data-flow algorithms. A specific proposal was never provided to the committees.

"Another alternative would be to place a standard-defined attribute on the object to inform the compiler that it may be randomized. This would limit the keyword pollution and convey the same information."
Since the attribute modifies the simulation behavior, an attribute would not be appropriate. Furthermore, it would be unwise to tie procedural methods, such as rand_mode(), where the appropriateness or availability of the method would depend on a variable whose type is only defined via an attribute.

3.12.3 Constrain, inside, dist, extends, with, solve, before (3.1)

"All of these are examples of operators that have been added as keywords. They are also short English words that have a significant chance of overlapping with identifiers already used in designs."
These keywords clearly conveys their meaning (by the way, the **extends** keyword is not related to random constraints, but to class inheritance in general).
No alternative operators or keywords were proposed to the committee for their consideration.

3.12.4 Implementation of $urandom(), $urandom_range, $srandom() (3.1)

"The LRM does not however provide the specific algorithm or 'C' code for the generator. This missing portion of the LRM will cause the generators to not be deterministic across multiple vendors making it extremely difficult for a given customer to use more than one simulator."
This statement is misleading. Several other factors aside from the random number generator prevent regressions from being deterministic across multiple vendors. Some of these factors are:

1. The specific event scheduling order.
2. The exact random seeds.
3. The order in which processes are created, started, end executed.
4. The exact nature of the constraint solver, if used.

"Synopsys has been requested to provide the same level of detail by the SystemVerilog committee and has refused to do so."
Synopsys's response to the SV-EC committee request included a reference to "Maximally Equidistributed Combined Tausworthe Generators" by Pierre L'Ecuyer Mathematics of Computation, 65, 213 (1996), 203—213 defining the implementation that they have used. This paper includes not only the exact C code but also a more detailed analysis of the method than what is included in the IEEE

1364-2001 LRM. Synopsys indicated that they could not donate this technology on the basis that it is not their property to donate; it is in the public domain.

A lack of clarification in this area has been shown to lead to tests that are not portable across multiple vendors. Synopsys claims in responses to the sv-ec that this $random style is no longer prevalent. We disagree.
This opinion may simply indicate a difference due to a lack of experience using both random constraints and $random. Synopsys claims that their users have moved away from the $random style after being exposed to the benefits of random constraints.

3.13.1 Semaphores (3.1)

"Cadence concurs that the semaphore semantics must be defined as a primitive synchronizer because it can not be otherwise expressed in Verilog ... However, we believe that these should not be defined as new language keywords."
The word **semaphore** is not a keyword. It is defined as a built-in class, precisely as Cadence suggests.

"These should be defined by providing the class definition including prototypes for the methods in the class in SystemVerilog source form. The definition of the behavior of the methods can be given in prose form in the LRM since it is not expressible in SystemVerilog."
The LRM defines the behavior of the methods in prose form, as suggested by Cadence.
The semaphore class prototype should be included, which is a minor editorial issue.

3.13.2 Mailboxes (3.1)

"The mailbox is a second built-in class. The behavior of mailboxes is completely expressible in SystemVerilog."
This is incorrect. A non-parameterized mailbox uses a *dynamic* type that can not be expressed in SystemVerilog. It needs to be built-in.

"The standard should specify that this is a reference definition of the class and that vendors can provide a different implementation as long as the semantics remain unchanged. This would allow highly optimized implementations of mailboxes without introducing new keywords but with increased semantic specificity."
That is the existing definition. The LRM does not preclude optimizations as long as the semantics remain the same. Note again that **mailbox** is not a keyword, it is a built-in class.

"Note that this would also provide the opportunity for other users or vendors to provide alternative mailbox implementations that may be annotated with value-added capabilities such as statistics gathering and coverage analysis."
Because mailboxes are classes, users themselves can add these capabilities by extending the base class.

3.13.3 Named Events (3.1)

"This dynamic memory behavior is exactly the same behavior as objects of a class type, so rather than modifying the existing static named event mechanism, a new class should have been brought into existence with the specified functionality."
The committee discussed distinguishing the static and dynamic events in SystemVerilog, for instance by using "**ref event**". Eventually, the committee concluded that the balloted use would be easiest for the user.
Note that events are already a type in Verilog. Furthermore, events are special, even in Verilog. For example, they are not synthesizable, they are not recognized by the PLI, they cannot be used in system tasks, and there is one Verilog operator that applies only to events (->).

"The new **triggered** method could then be declared as a method in that class. The functionality of **triggered** would still need to be predefined because there is no way to express it in native SystemVerilog."
There are a few less-than-optimal ways to express the **triggered** functionality in native SystemVerilog (e.g., storing $time at the time of triggering). However, the balloted definition is easier for users and does not require a new keyword or class. The committee could not find a more appropriate name for this new *class* than event, so it was decided to fold the new functionality into the existing type.

3.14.1 Property Evaluation (3.1)

"In the presence of this sampling it is not necessary to delay evaluation of properties to the observe region. Since the values have been sampled, the execution of the property can happen at any time and the result will be exactly the same. This delayed execution places an undue burden on the implementation to conform to an over-constrained reference algorithm."
The LRM specifies the semantics of the evaluation, and does not dictate how an implementation should be structured. By defining these semantics, all simulators should produce the same property results, including values obtained via VPI calls. An implementation may internally optimize its evaluation, as long as it abides by the defined semantics. This is generally true for all language constructs.

3.14.2 Delaying of Pass/Fail Code (3.1)

"An assertion statement can have pass/fail code associated with the assertion. The scheduling semantics say that this code is scheduled in the reactive region of the simulation cycle. Cadence believes this code should be executed whenever the property is evaluated to ensure that its execution matches the simulation state precisely."
Nevertheless, the assertions committee did not agree with Cadence. Instead, the committee felt that it was more important to isolate the execution of the pass/fail code (which is deemed to be testbench code) than to attempt to match the execution state precisely. Furthermore, different implementations of the language may use sampling and other optimizations, thus, it may not be possible to "match the simulation state", only the property state.

"In this partial example … When the assertion executes it will use these sampled values; however, when the $display statement is executed in the reactive region, the current values, not the sampled values, will be displayed."
If coded as shown in the Cadence document, the $display statement would use the incorrect values. Nonetheless, users can work around this limitation by displaying the sampled values (using a clocking domain). A future extension to make the sampled values available to the pass/fail code will address this issue.

Assertion pass/fail code must execute whenever the property is evaluated in order to come even close to accurately reflecting the state of the simulation. Even with this, the fact that the assertion uses implicitly sampled data will make this difficult.
What is really important is that the pass/fail accurately reflect the state as of the time of the property evaluation, regardless of when the pass/fail code executes. And, the language includes mechanisms to do precisely that. As the technology matures, future enhancements that affect ease-of-use can be incorporated into the existing framework.

3.15 Section 15 - Clocking Domains (3.1)

"This concept … concentrates on their use in testbenches. Cadence believes that this emphasis should be removed as they add a very powerful general modeling capability".
Concrete proposals should have been made as part of the many reviews. Surely, this request means that additional examples should be added in the future, and not the removal of something.

3.15.1 Verbosity of declarations (3.1)

"Some form of implicit declaration or namespace inheritance should be included to make this less repetitious (something similar to ".*" and "@*")."
These suggestions may be good enhancements. However, this particular issue was discussed and was discarded since it was deemed comparable to a hierarchical reference to a non-existent signal; since the latter does not implicitly create the signal in the specified hierarchical location, implicit clocking domain signals should not either.

3.15.2 #1step will create non-deterministic IP (3.1)

"Since, **step** is a new general time literal, it can be used anywhere in a description, for instance in a blocking assignment. The value of this delay will actually change depending on the design in which this module is instantiated."
The fact that the delay may change depending on the design does not mean that the IP will be non-deterministic. Users do not need to use it in delay expressions unless they have a specific need for that,

in which case they presumably know what they are doing. This situation is no different to an IP model that specifies a delay in **ms** and then some other model increases the resolution to **ps**. Although the physical units are absolute, the relative event ordering might be different, hence, non-deterministic.

3.15.3 #0 semantics are misleading (3.1)

"A sampling input skew of #0 would intuitively be interpreted as a sampling at the beginning of the simulation cycle; however this is really the semantic of the #1step skew."
The LRM explicitly deals with that issue to avoid such confusion:

> "Skews are declarative constructs, thus, they are semantically very different from the syntactically similar procedural delay statement. In particular, a #0 skew, does not suspend any process nor does it execute or sample values in the Inactive region."

Thus, #0 just means at the same simulation time, regardless of whether it specifies a delay or a skew.

"We believe it will be a common error for users to utilize #0 where they intend #1step and get difficult to debug simulation errors."
The default skew is defined a **#1step**. The default was chosen judiciously so that most users never need to set any skew explicitly.

"We also can not think of an example where observe region sampling is actually useful, therefore #0 should be defined as the beginning of this time and a special case should be created for the unusual semantics of sampling during the observe region."
That reasoning is incorrect. Input skews are specified not as a delay, but as the amount of time before the clock edge (i.e., in the past). Thus **#2ns** means sample two nanoseconds before the clock edge. In this vein, **#1step** is easily interpreted as "immediately before the clock edge". The special case is indeed the one using #0 since it does not mean at the same time as the clock edge (0 time before), but after the clock edge.

3.16.1 Functional overlap with module (3.1)

"The only difference from a module is that the only behavioral constructs they can contain are initial blocks and tasks/functions, and that they have delayed simulation semantics … if removed, will make modules and program blocks identical."
The delayed simulation semantics are not removed, thus, they are not identical.

3.16.2 Modeling restrictions (3.1)

"A test environment, when expressed in native Verilog, is expressed as a system-level model surrounding the device under test. This relationship is naturally expressed by using hierarchy in the testbench itself as well as in the model."
This relationship is expressed at the boundary between the testbench and the device under test. At that level, users can utilize modules or interfaces to model these relationships. It is only the program block itself, which models a single unit of execution that is thus limited.

"Restricting program blocks by not allowing hierarchy in a program block will make this impossible."
Programs can be contained by modules or interfaces, which may incorporate hierarchies. This impossibility simply does not exist as claimed by Cadence.

Again, this restriction is just a legacy from Vera and adds no expressive power; it merely limits the user's flexibility.
The program block is intended to resemble a C program with well-defined entry and exit points. These simple semantics allow users to write testbenches that consist of several independent programs that do not need to be aware of one another.

3.16.3 Reactive semantics (3.1)

"… functionality relative to a process (always block, initial block, or task), not hierarchy. Creating a mechanism that allows a specific process to be reactive is the more natural place to add this concept, not by replicating and restricting the entire concept of a module to get the behavior."
The new mechanism to which Cadence alludes exists and is now called the program block. The program enables users to place all related code that is to execute with Reactive semantics under one construct instead of bringing different variations of always, initial, task, function etc… (Note how in

section 3.9.2 Cadence objects to this type of mechanism for the always_comb, always_latch, always_ff constructs).

"This reactive behavior is also introduced in an attempt to let testbenches "run last" in the simulation cycle… Specifying this special status for program blocks is completely artificial and we believe it can actually create verification problems not solutions."
The Reactive execution behavior enables SystemVerilog code to execute in the exact same manner as is currently available only to C code that uses the Read/Write Synchronize callback. This statement is not only misleading but contradicts the previous sentence:

> "This concept is similar to the Read/Write Synchronize callback available from PLI. Although we think having access to this region from Verilog code is a reasonable extension,…"

The Reactive region execution allows code to react to assertions in an orderly and predictable manner, and thus avoids creating additional races in the testbench.

"If the testbench is scheduled with special semantics, then it is not exactly emulating a device stimulating this object. When the device under test is embedded in another model it will not be stimulated by objects with this special semantic therefore it has not been accurately verified."
This is untrue. The testbench must stimulate the device under test in the exact same manner as the hardware being emulated. However, the important part is the interaction between the testbench and the device under test at the interface, and not the internals of the model.
If that statement were true then a device under test could be accurately verified only by a hardware model that uses the same simulation semantics. Clearly, that is not the case for most high level models, or formal tools. The Reactive execution is a very useful concept and that is why every single testbench solution (including Cadence's) provides such a capability.

3.16.4 Termination of all simulation through $exit() (3.1)

"The addition of $exit gives a single initial block an extreme form of global influence. By indicating that this one process is complete it can terminate the entire simulation."
That statement is inaccurate. The $exit call only terminates its enclosing program block plus any background processes started by that program.
The issue is not the $exit task, but what should be the behavior of a SystemVerilog simulation that uses multiple programs. The balloted definition enables users to include an arbitrary number of independent programs without any one of them having to know about the others. Instead, this global information is maintained by the tool.

"If this initial block has enough global state information to know that this can safely be done, then that initial block should call $finish and terminate simulation. If it does not have that global state then it should only have influence over its own environment, not the entire simulation."
This is precisely the point. An independent program typically never has enough global information to determine whether to terminate the simulation or not. This capability is extremely useful for users of verification-IP models, which cannot (and should not) have the global information as to whether to terminate the simulation. Instead, each model is only responsible for its own termination and the tool determines when to finish the simulation.

3.17.1 Complexity (3.1)

"The definition is extremely complex, tied to an underlying notion of how synthesis should be performed, and tied to a new simulation model that has added significant complexity to the language."
Assertions significantly simplify verification by adopting a model that is well-known to work with simulation, synthesis, and formal tools. They are defined using the SystemVerilog scheduling semantics that were devised jointly with Cadence.

"The rules for clock inference are complicated and provide many opportunities for errors in what should be a simpler, more abstract specification of behavior."
The rules for clock inference are not new; they are familiar to most hardware designers that use synthesis. No alternative proposal on how to infer clocks was provided to the committee.

"Overall, System Verilog assertions appear to be much more difficult to use than plain Verilog (as in OVL), with little or no additional benefit."

It is meaningless to compare Verilog constructs with assertion constructs. Assertions provide abstractions of behavior that Verilog does not have, such as sequences, repetition of sequences and other operations on sequences. The concepts and semantics of assertions are adopted from proven languages donated to Accellera: Sugar, OVA, ForSpec, 'e', and CBV.

Indeed, assertions provide a powerful capability to abstract hardware using a model that is proven and familiar. OVL is not a language that expresses such abstractions, but a library of simple but useful checkers. Therefore, a comparison with OVL is baseless.

That statement by Cadence is also surprising since Cadence has been involved in the language working group (DWG) and had concurred in deciding the underlying model of SystemVerilog Assertions, and its features.

3.17.2 Timing Alignment (3.1)

"If the hardware design responds to a clock in a given way, then the assertion needs to respond in the same way. Otherwise the assertions and the hardware are out-of-phase with respect to each other, and the assertion cannot function as an abstraction of the hardware."
The sampled values that Assertions use correspond to the same values seen by actual hardware. Nothing else will suffice to guarantee that the verification provided by simulation will not only hold from simulator to simulator, but also hold after synthesis. There is no out-of-phase behavior.

"This approach is user-controllable, affects only the clock signals, does not require an expensive and complex data sampling semantics, and works within current Verilog."
The semantics defined for sampling of values fits within the scheduling semantics of SystemVerilog. The efficiency issues related to the scheduling semantics were discussed, resolved and unanimously voted by all committee members.

"PSL assertions are defined in a manner that is consistent with the execution of Verilog, VHDL, and other event-driven hardware description languages."
PSL does not define how traces are obtained from simulation values, or what simulation values are to be considered for evaluation. That is why the comparison with PSL is not appropriate. As far as semantics of clocks and sequence evaluation, SystemVerilog assertions and PSL are consistent and both use the notion of tick points in the trace. There is no discrepancy as suggested above.

"The attempt to avoid race conditions by using sampled values also raises the possibility of false positives – the assertion, looking at sampled values, may fail to catch a race condition that will actually affect the hardware."
The sampled values prevent races, which may be introduced by the simulation algorithm, in the assertion evaluation, but not in the actual hardware. In fact, the approach proposed by Cadence would do the reverse, it would open up the assertion evaluation to the same sort of racy behavior extant in the simulator. The committee had extensively discussed these issues, and decided to use sampled values as defined.

3.17.3 Clocks (3.1)

"So given the concurrent assertion
"never clk1 && clk2"
to attempt to say that two clocks are mutually exclusive, there must also be a global clock that controls 'sampling' of these two specific clocks, and the assertion will only be checked at ticks of that global clock."
Checking of two simultaneous edges is not possible in Verilog as written above. This was also not a requirement of SystemVerilog assertions. Nevertheless, that particular condition can be captured by the following SystemVerilog immediate assertion:

```
always @(*) assert ( !(clk1 && clk2) );
```

Note that this assertion works because it does not attempt to check the simultaneity of the two edges.

"Instead, it clearly states that concurrent assertions are evaluated only at clock ticks:"
SystemVerilog assertions explicitly disallow defaulting to simulation time to prevent user errors. One can always model a clock in Verilog that is based on simulation time. As stated previously, there is no fundamental discrepancy between PSL and SystemVerilog assertions semantics. PSL does not define

how to obtain traces for assertion evaluation from simulation. However, once a trace is obtained, each trace point represents a tick point of the lowest granularity, which is same for SystemVerilog assertions. In practice, PSL uses a clock of the lowest granularity for the evaluation of un-clocked properties.

"In particular, the mapping from the (level-sensitive) boolean clock conditions in the formal semantics to the (edge-sensitive) event controls used to specify clocks in the LRM is not specified. The fact that this mapping is required restricts the formal semantic definition, which is more general than the LRM language."
The formal semantics of SystemVerilog assertions are meant to be more abstract and were intentionally left without including the restrictions. The objective was to keep the semantics simple to understand. The restrictions on the clock and elsewhere are explained in the LRM.

3.17.4 Syntax (3.1)

"The decision to use '##n' as the separator between elements of a sequence is needlessly verbose, and in fact it makes sequences difficult to read."
The ASWG group was formed to align the syntax of assertions with the rest of SystemVerilog. Cadence was a major participant in this group. The decision to use "##" as a delay notation was made to be consistent with the use of "##" as the cycle delay expression in procedural code, which is itself a simple extension to the Verilog delay expression "#". There was no disagreement to use "##" as a cycle delay operator in the group, including Cadence.

"This leads to non-intuitive semantics resulting from associativity of ## determining which of the two is executed first."
Using the same operator for overlapping and non-overlapping delays is consistent with Verilog #0 and #n semantics. In addition, it simplifies applications where the delay value needs to be parameterized. The semantics of ##0 and ##n do not require different associativity rules.

"… syntax for property_spec appears to require 'not' in conjunction with a multi_clock_property_expr."
This is an error in the BNF that was noted after draft6 was released.
The correct property_spec BNF should read:

property_spec ::=
[clocking_event] [**disable iff (** expression **)]** [**not**] property_expr
| [**disable iff (** expression **)**] [**not**] multi_clock_property_expr

The **not** is optional; it is not required.

3.17.5 Documentation (3.1)

"The LRM only vaguely defines the terminology used to describe the semantics of assertions. While a formal semantics has been defined by the semantics group, it is not part of the LRM, and the connection between the LRM and the formal semantics is not at all clear."
The formal semantics document will be added as an appendix to the LRM. The primary objective of the LRM is to convey the intuitive meaning of the concepts, whereas the formal semantics document is written to precisely define the constructs in mathematical form.

"The LRM intermixes references to 'evaluating an expression at a clock tick' and an expression 'being true at the nth sample'. Such inconsistency clouds the intent and confuses readers."
There have been numerous reviews with Cadence as participant. Unlike many other participants, Cadence did not provide any suggestions for correction or improvements as part of the review of draft4 or draft5. It is again surprising that the criticism of the documentation has become part of the negative ballot comment, rather than part of the reviews.

3.18 Section 18 - Hierarchy (3.0)

"These include $root and nested modules. These are yet again examples of layering rather than integration as explained below."
This philosophical discussion on "layering versus integration" is largely a matter of opinion. What is clear is that Cadence changed its mind since reviewing and ratifying the 3.0 LRM.

3.18.1 Does not address program blocks (3.1)

"SystemVerilog now adds program blocks and they must be included in each of these places. Rather than repeat this in all places, Cadence would prefer if both interfaces and program blocks were merged with modules, but as long as they are not, these references should be fixed."
This is a useful editorial comment. Wherever suitable, references to program should be added.

3.18.2 $root (3.0)

"The $root scope creates a single top-level scope in Verilog. We believe this is a disastrous addition to Verilog."
Verilog has always had the concept of a hidden $root; SystemVerilog makes it explicit. The hidden $root includes such things as timescales, and primitive and module definitions.
In order to re-use a task or function, Verilog forces users to include a copy of the task or function definition in each module, typically using a 'include directive, or by creating a single root-module in which all shared objects are explicitly added. In SystemVerilog, $root allows direct sharing of such code.

3.18.2.1 Separate compilation (3.0)

"The primary capability of Verilog that allows this methodology is that all objects are contained in modules. The addition of $root breaks this paradigm."
This is incorrect. Verilog allows cross-module hierarchical references that are not contained within the module.

"If objects or statements are declared in the $root scope, then it becomes extremely difficult to allow separate compilation"
This is no more difficult that hierarchical references. A proposal that uses extern definitions to facilitate separate compilation was made and accepted by the technical committee.

"The solution for any of these issues is to take the objects in $root and put them in a module so that they have a name and can be explicitly brought into a hierarchy."
This is not a solution; from the user's perspective it is an impediment. SystemVerilog simplifies the user's problem by having the compiler do this automatically.

3.18.2.2 Global name conflicts/visibility (3.0)

"The addition of $root creates a global declarative region for objects. ... if the same name has been used they will conflict and one of the designs will need to be modified."
This is not due to $root. The problem of module-names collisions already exists in Verilog. To avoid such problems, development teams have adopted coding guidelines and methodologies. Since SystemVerilog extends the same issue to $root declarations, there is every reason to believe that developers will adapt their methodologies appropriately.

"Declaring objects in a module and then referencing through this name means that only the module names need to be kept unique, not every single object."
As said above, module-name collisions is an existing Verilog issue. The above methodology simply moves the problem from the shared declaration to the name of the module containing the declaration. Thus, it does not solve the problem, it merely creates more work for the user.

3.18.3 Namespaces (3.0)

"The entire content of this section is baffling to Cadence. Verilog 2001 defines 7 name spaces, SystemVerilog defines only 5. There is no explanation of what the difference between these namespaces is or why this portion of Verilog was modified at all."
The Verilog 2001 definition is incorrect; there are only 5 namespaces. Verilog 2001 allowed specparam and module parameters to be used interchangeably, thus, the specify block becomes just a local scope and not a separate namespace. However, the namespace section was not updated to reflect this change.
In Verilog 2001 module and primitive definitions are said to occupy their own separate namespace, but in reality they are just a top-level local scope and not a different namespace.

"The term name space is ill-defined in Verilog."

This is not a problem introduced by SystemVerilog. In fact, SystemVerilog attempts to clarify many of these concepts (see above point).

"It was suggested by the committees that instead of addressing it in SystemVerilog we should instead take this to the IEEE Errata Task Force. We still believe that this chapter should just be removed from the LRM as it adds no value and just conflicts with 1364."
This was not *suggested* by the committees; it was balloted and the decision of the committee was to send this issue to IEEE Errata Task Force. SystemVerilog should adopt whatever resolution is implemented by the IEEE.

3.19.1 Overlap with modules (3.0)

"The basic capabilities of an interface declare ports, parameters, tasks, functions, and always/initial blocks to describe communication. This is all identical to the content of a module."
An interface models interconnect and can therefore be an argument to a module. Modules cannot be used in this way, thus, they are not the identical. In fact, an interface is closer to a type declaration since their definition is in-lined in the instantiating scope.

"The more advanced capability of passing interfaces instances through ports, defining multiple port lists (modports), and importing/exporting task and function definitions would all be excellent additions to the general definition of modules and do not require a new top-level language construct."
Keeping interfaces separate from modules allows users to correctly express their intent. This helps users in documenting their intent and, since they are structurally different, allows synthesis and debugging tools to treat them differently.

3.19.2 Overlap with classes (3.1)

"First of all, this is not new in Verilog. For years, users have defined modules with tasks that represent a devices behavior and then interact with the module by calling these tasks. It is simply made more convenient by being able to develop the hierarchical path through a port."
That is an important distinction, but it is only one of the features that distinguish interfaces from modules.

"In SystemVerilog 3.0, there was no class mechanism so discussing use of interfaces with a class-like metaphor made sense. In SystemVerilog 3.1 there is now a class mechanism that satisfies the need for object oriented extensions. This further motivates merging interfaces and modules to emphasize their structural nature and relegating object oriented programming to true classes."
Interfaces are purely a structural container, therefore synthesizable. Classes are not synthesizable. One may think that it is a simple matter to make classes synthesizable, but the reality is that there is no known technology that would allow a construct to contain wires (as an interface) and to also provide functional hierarchy and polymorphism as well as dynamic data allocation (as a class).

3.19.3 Lack of decomposition (3.0)

"Interfaces can only contain procedural constructs such as always blocks and initial blocks. They can not contain gate-level models or hierarchy underneath them. So despite the introduction to this section, they do not support hierarchical refinement well."
Interfaces model interconnect, thus, they require no gate level primitives. However, this could be a future enhancement without any loss of generality.

"Interfaces provide an excellent mechanism for encapsulating communication and allowing multiple devices to communicate."
The technical committees tend to agree.

3.20 Section 20 - Parameters (3.0)

"This section should only contain the extensions and not reiterate the content from 1364."
This entire section is under two pages, and other than in the introduction it relates to new features. This editorial comment should have been made during the 3.0 process.

3.20.1 Parameterized types (3.0)

"While Cadence understands the expressive power of parameterized types we believe it is an unwise extension of Verilog. This is an extremely difficult and complex thing to implement for functionality that can be gained in other ways."
What this statement really says is that regardless of how useful a feature may be to users, it should be excluded from SystemVerilog because Cadence does not know how to implement it.

"Modeling styles where macros are used for types can provide similar capability, or the addition of a generic handle type to allow modeling of externally linked data structures would both be possible alternatives."
These alternatives were never proposed during the 3.0 standardization process.

"An example of the difficulty of this can be found in the C++ world where type templates existed in the standard for years before any compiler vendors supported them."
Unlike C++ templates, there now exist two different SystemVerilog simulators that implement type parameters.

3.21 Section 21 - Configuration libraries (3.0)

"f this information is set globally in $root, then it would be visible for all configurations. During integration of IP from different sources this would be yet another form of global conflict created by $root."
As stated before, this from of global conflict already exists in Verilog and is not created by $root.

3.22 Section 22 - System Tasks and System Functions (3.0)

"This section documents new system tasks and functions that have been added to SystemVerilog. In general there is very little specific feedback on this section other than reiterating the previous comments that in many places system tasks, operators and now methods have been used in arbitrary places without any particular rationale for when one was used over the others."
The reasons for each use of a method, operator, and system task were debated in the committees, and re-iterated in this response. The rationale was discussed, various proposals were considered, and balloted. The LRM is the result of that process. It is hardly arbitrary.

3.22.1 $asserton, $assertoff, $assertkill (3.1)

"These functions are used to enable or disable assertion execution during simulation. These functions are extremely dangerous in many cases."
The same can be said of many other Verilog constructs. For example, **disable, force, release**, etc.. all manipulate the execution of simulation code from without, and can certainly be considered dangerous. Nevertheless, all of these exist because they satisfy a very real user need. It is incumbent upon the users of these features to modify the simulation carefully in a way that meets their objectives. To remove such functionality on the basis that "users will not know what they are doing" is simply arrogant.

"We would prefer to see these functions only control the reporting of assertion failures or coverage data, but they would continue to track the state of the simulation accurately."
These tasks only affect assertions, i.e., assert and cover statements. They do not disable sequences or other internal state that is required for future evaluations. Effectively, they only disable reporting and VPI callbacks, while improving the run-time speed by turning off unnecessary evaluations.

3.23 Section 23 - VCD Data (3.0)

This section points out that VCD data has not been extended to deal with all the new SystemVerilog data types. We believe this is a major shortcoming of the standard that should be fixed before this standard is approved.
The technical committees agree that this work should be finished once the language has been approved. There is no point in defining a format for a type that is not yet part of the language.
It worth noting that IEEE 1364-2001, which Cadence supports, introduced the notion of automatic variables but explicitly stated that no VCD support for such variables would be provided.

3.25 Section 25 - Features Under Consideration for Removal (3.0)

"Cadence believes that given this concern about deleting content from an existing standard that it is hypocritical to propose any deletions of an existing IEEE standard under the guise of Accellera work. Deprecation of functionality should solely be the work of the 1364 task force."
The 3.0 committee only proposed that these items be considered for removal by the appropriate body, which is the IEEE 1364 task force. It's worth noting that this section was approved by the technical committee (including Cadence) in the 3.0 process and sent to the board.

3.26 Section 26 - Direct Programming Interface (3.1)

"… there is not really a true equivalence between SystemVerilog 'C'-like data types which are defined in fixed size manner and 'C' native data types which size depends on the 'C' compiler and platform."
This is not correct. Sections 26.1.2, 26.4.6, 26.5.1.1 and D.5 precisely define how SystemVerilog types and C types are matched at the interface boundary. This is still transparent to the user.

"Note that section 26 and annexes D, E and F deal with the Direct Programming Interface. The comments in this section cover all these areas."
These negative comments are surprising since Cadence was an active member of the sv-cc committee, and with respect to the C side of the DPI, its two representatives voted in favor of the proposal (see http://www.eda.org/sv-cc/hm/0681.html and http://www.eda.org/sv-cc/hm/0685.html).

3.26.1 Mix of direct and abstract interface (3.1)

"Cadence believes that the SystemVerilog DPI abstract interface is unnecessary; a handle-based abstract interface already exists in VPI. DPI should only focus on providing a canonical representation and provide direct access to simulation object values without handles."
The abstract interface was proposed, discussed and accepted in the committee as a response to specific user requests. Users are not required to use the abstract interface; it is simply provided to enable access to data types that do not directly correspond to C types.

3.26.2 Two possible representations for packed (vector) types (3.1)

"We believe that a standard should not provide ways to promote non portable user code but rather should focus on defining a minimal and common portable approach acceptable for all vendor implementations."
This statement is attempting to redefine what "portable code" means. In general, code is considered to be portable if it can be recompiled to run on a different platform with no change. That is the general use of the term "portable". For example, C code is portable if the same code can be compiled to run on Solaris and Linux without change. In addition, DPI defines a more stringent binary-level compatibility layer that permits applications compiled once on a particular platform to interoperate with any simulator on that platform.

3.26.3 Source and binary portability (3.1)

"… the DPI interface proposes two header files, one which contains the public functions and canonical data structures and another which will contain vendor dependent internal data structures. If the C code uses packed SystemVerilog data types (for which 2 representations are possible), the C code written will not be source or binary compatible depending if the internal representation or the canonical representation was chosen."
That is misleading. The user of DPI is the one making this choice, and this choice is independent of the data being transferred across the DPI interface.

We believe that it is not the purpose of a standard to provide vendor specific header files. A standard should only specify a portable and common method. We believe that this will be strongly opposed when presented to the IEEE standardization entity.
That is a matter of opinion. If an API is valuable to the users then having a standard specification in the LRM is clearly better than many (potentially incompatible or nonexistent) vendor-provided APIs.

3.26.4 Overlap and redundant functionality with VPI and PLI (3.1)

"… believe that a new standard interface should not overlap with an existing Verilog standard, namely the VPI or PLI interface. We believe such an overlap in scope will not be accepted by the IEEE.

Functions such as svGetScope, svGetScopeByName, svGetUserData etc. are exact duplicates of existing VPI functions. VPI or PLI functions should be used instead."
That is incorrect. As covered in section D.8, there is no VPI equivalent for the capability provided by these DPI interfaces. Specifically, VPI only provides for a means to store a single pointer associated with a specific instance. This means that models from multiple sources that need to store instance associated data must do so on their own (outside VPI) or risk overwriting each others' data.

"Further these DPI functions return opaque handle which are not compatible with VPI handles. We are afraid that the DPI interface functionality will be extended to duplicate even more of the VPI functionality since the VPI and DPI handles are not compatible."
The same could be said for PLI and VPI handles. DPI and VPI are two separate interfaces serving different needs and users, just like VPI and PLI.

3.26.5 Many library access functions (3.1)

"We believe that it would be more performance efficient and less memory error prone if the memory allocation was done by the simulator and a copy of the value would be passed to the argument of the DPI function; the C code would directly access or modify that value. … If the object does not have any fanout, a reference to the internal canonical representation can be passed to C."
This presumes that there exists a canonical representation for all data types in SystemVerilog that all vendors must strictly agree to follow. For types with such canonical forms no DPI functions are required. DPI permits access to SystemVerilog data even when no canonical form is defined. It does this by providing functions that map from a C data type to the simulator's representation. This avoids large scale copying of data-structures (thus maximizing efficiency) without removing capability.

3.26.6 C data type mapping (3.1)

"The DPI interface maps many of the SystemVerilog data types to a C data type (SystemVerilog int to a C int, SystemVerilog byte to a C char), however this mapping assumes a particular implementation of a C compiler where int would be a 32 bit signed integer and char is an 8 bit signed integer. These SystemVerilog C types do not truly match a C int or a C char: size of these types is implementation defined in C."
This issue has been mentioned twice earlier. The DPI interface does define how values will be promoted and coerced to the appropriate types across the interface. As covered in sections 26.5.1.1 and D.5, users are in full control of what type will be seen on both the SystemVerilog and C sides of the DPI interface.

The current DPI interface does not support all SV data types: classes, events, associative arrays, semaphores, and structs/unions of these types.
As noted in an earlier response, this type of work always lags the actual language definition. That was the case for 1364-2001. Furthermore, passing some these types of objects across to C may not even be possible nor desirable.

3.26.7 Open array arguments (3.1)

"Cadence strongly believes that if one part of the language is enhanced, all dependent features of the language such as (VCD, SDF, or VPI) need to be enhanced in parallel to preserve consistency and integrity in the language. Failure to do so will result in not only an incomplete specification but also catastrophic flow breakage in our customer's methodology."
This is specious. Open arrays are defined in the context of DPI and only for use by C code (as C code cannot be parameterized to deal with differing argument widths as can SystemVerilog code). No Verilog interface exists to dump/annotate or otherwise manipulate C variables, even in the existing VPI standard and this has not led to any "catastrophic breakage" of our customer's methodology.

3.26.8 SystemVerilog context and pure qualifiers (3.1)

"We believe that there are better alternatives to scope setting which would avoid runtime SystemVerilog export function look up. For example, solutions such as function instance specific export or combining a function export declaration export with a C name denoting the hierarchical name of a specific function instance."

These suggestions were discussed in the committee, they were balloted, and rejected because they were deemed not scalable. Instance names are absolute. Should a user instantiate 2 identical modules each exporting the same function, users would be forced to modify either the SystemVerilog code for the module or the corresponding C code (or both). For this reason, this approach was rejected by the committee, and in favor of the approach defined in the LRM.

"In any cases, adding qualifiers such as pure, context or string such as "DPI" to import function declaration is yet another way to express a property of a function. Verilog attributes could have been defined and used for the same purpose, eliminating the need for short English words as new keywords."
An attribute is only appropriate if the execution semantics are unaffected. That is not the case here, since the simulation may hang or crash if the wrong attribute is specified.

3.26.9 DPI object code inclusion (3.1)

"Cadence believes that this annex is a good attempt to standardize on a foreign code delivery and linking mechanism. However we think that there are some problems with the approach presented in this Annex."
This is again surprising given that Cadence was an active participant in this committee and voted to approve this section of the LRM (ref: http://www.eda.org/sv-cc/hm/0881.html)

"Command line switch names should be avoided in a standard LRM. It should not be mandatory for a tool to provide command line switches; for example a GUI driven tool does not have a command line. This annex will have to be completely rewritten to avoid mention of any switch name when SystemVerilog is folded in the Verilog 1364 as there is no mention of any switch in the Verilog standard."
This issue was much debated in the technical committees, and the consensus is to make these switches informative rather than normative. However, this issue was raised and strongly championed by the user representatives in the committee, who often need to integrate multiple tools from multiple vendors into a single environment. These users felt that having some common switches for common functionality would simplify their task. Since the switches are informative rather than normative, the GUI argument is specious.

"… DPI function names have to be global and unique across all foreign object code. We believe that this severely constrains object code inclusion and can lead to errors when linking if the same name is defined in multiple libraries."
This is not true. The link name of extern and export functions must be unique, thus, this is checked by the SystemVerilog tool prior to the link stage. This same issue is also covered in section 26.3.

3.27.1 Static information model of assertions (3.1)

"The information model was restricted to the minimum. The current assertion API is mostly a runtime API allowing an application to interact with the assertion evaluator. We believe that a more detailed property/assertion/cover static information model should also be provided."
The Assertion VPI extensions are primarily a dynamic information model. In general, as previously pointed out, VPI extensions lag behind the language definition. For the Assertion VPI extensions the committee felt that the dynamic portions of the model would be least affected by the actual structure of the language and could be defined in parallel with the evolution of the assertion syntax, structure and semantics. To this end a specific set of requirements (ref Section 27.1) were targeted, namely support of debuggers, assertion dumping tools, coverage tools and user's C code reacting and interacting with assertions. Thus, the committee defined the VPI extensions required to satisfy those requirements. The committee felt this was the correct approach, including Cadence whose representatives voted in favor (ref: http://www.eda.org/sv-cc/hm/0665.html, http://www.eda.org/sv-cc/hm/0666.html).

3.27.2 Callbacks (3.1)

"A new callback registration function was introduced to place an assertion callback rather than using the existing mechanism vpi_resgister_cb. … We believe that another better alternative would have been to use the same registration function and provide another method from the assertion handle which would return the assertion attempt info. The assertion current status would be available through this method but only at the time of the callback. That way the user is not confused in which callback functions to use."

Note that IEEE 1364-2001 does not define a VPI *get* function that returns an arbitrary pointer as the result of obtaining a property from some handle. Thus, this suggestion is not applicable without also introducing a new type of VPI function. Since the reasons and data associated with assertion callbacks are disjoint from those associated with other VPI callbacks, the committee felt that it was better to introduce a new callback function, rather than merging two very dissimilar capabilities under the IEEE 1364-2001 vpi_register_cb function.

3.27.3 Assertion Control (3.1)

"We believe that this capability is dangerous in certain cases for the same reasons given in section 3.22.1."

This is just a rehash of section 3.22.1. The procedural interface VPI provides a means for Verilog users to access and modify simulation and assertion data. It is incumbent upon the users of VPI to modify the simulation carefully in a way that meets their objectives. Following these guidelines, VPI provides additional control over the evaluation of assertions than what is available via system control tasks. vpiAssertionDisable and vpiAssertionKill are identical to $assertoff and $assertkill, respectively. vpiAssertionReset extends the functionality of $assertkill by resetting all the states related to the assertion to their initial state. These calls, enables users to build customized applications. The risks and benefits of providing explicit control over assertion evaluation are no different than for VPI calls that provide control over other simulation data.

3.28 Section 28 - SystemVerilog Coverage API (3.1)

"Cadence believes that the entire coverage API is ill-conceived. A language interface can only provide information about constructs that are explicitly specified in the language. Without adding an explicit model for coverage points and the kinds of coverage to be measured into the Verilog language itself, then a generic coverage API is inappropriate."

Definitions of all coverage points are provided in section 28.1.3

3.28.1 Pragma usage (3.1)

"Cadence believes that standard coverage attributes should have been defined instead of specifying these items in comments. A proposal was made but was postponed to SV 3.2."

That proposal was made too late to be considered for inclusion in the current LRM.