Cadence Design Systems Negative Ballot Comment on Accellera SystemVerilog 3.1

Cadence Design Systems, Inc.

Contributors: Jay Lawrence, Francoise Martinolle, Steven Sharp, Erich Marschner

4/24/03

Table of Contents

Executive Summary

Cadence believes that Verilog needs to be extended in order to support hardware design and verification within the Verilog environment. We have been actively involved in the SystemVerilog Accellera process for the last year with representatives on all relevant committees. We believe extensions in the areas of data types, constraints and randomization, direct interfaces, and assertions are important to the productivity of the industry.

Many of the concepts included in SystemVerilog 3.1 move Verilog in this direction, but Cadence believes that the draft 5 LRM is not coherent and complete enough to be considered by the Board of Directors as a proposed Accellera standard. The remainder of this document provides detail on the issues Cadence has with the draft 5 LRM. Cadence is providing this feedback because we are absolutely committed to enhancing Verilog in a manner that will provide the capabilities the industry needs, while preserving the users’ and vendors’ extensive investment in IEEE 1364 Verilog.

1 Overview

This document is being provided by Cadence Design Systems to all Accellera SystemVerilog technical committees and the Accellera Board of Directors as the Negative Ballot Comment on SystemVerilog 3.1[1]. Although Accellera bylaws do not strictly call for such a ballot comment, Cadence believes that it is an important part of the IEEE balloting process and that the justification for any negative ballot should be provided.

Throughout the SystemVerilog 3.1 standardization process, Cadence has participated at the Board of Directors level and in all technical subcommittees. We have repeatedly raised both procedural and technical objections related to the Accellera compliance to its own bylaws, the relationship of SystemVerilog to the IEEE Verilog 1364 standard, and the technical content of SystemVerilog. This document focuses on the technical content of SystemVerilog chapter by chapter.

It has been noted in email communication to the Board from the Accellera Technical Chairperson that because these objections were heard, voted on, and defeated, that Cadence should now support SystemVerilog 3.1 and vote in the affirmative. We respectfully disagree. There are major problems with the Accellera SystemVerilog technical content that we feel it is our responsibility as founding members of Accellera to point out. Accordingly, we have voted in the negative. We also do not feel that such a negative ballot violates our responsibility as a member to cooperatively and proactively promote the purpose of Accellera. We do not believe that promotion includes an affirmative vote on all issues.

The IEEE balloting process would require a formal response from the technical committees to a negative ballot comment of this sort that

[1] All references to the SystemVerilog LRM refer to the content of the Draft 5 LRM.

contains both specific and unspecific comments on the draft standard[2]. Since this is not a part of the Accellera bylaws we do not expect any such formal response, however, we would hope that other Board members and technical contributors consider these objections when placing their own vote on SystemVerilog 3.1.

2 General Comments

This section documents some general comments on the overall process and actions that were taken that we believe have led to an unacceptable standard at this time. These are brought out explicitly here because they will be referred to in the comments on individual chapters in the following sections.

2.1 Lack of Specific Commitment for IEEE 1364 Coordination

The goal of Accellera is to accelerate the specification and adoption of standards for language-based electronic design. As such, we and many other committee members participate in the SystemVerilog development process with the expressed belief that the effort would be tightly coordinated with IEEE 1364 and that a clear roadmap would be followed to donate to 1364. Only through this planned, clear communication between the groups will the definitive 1364 standard be accelerated and a divergent Accellera standard will be avoided. This document lays out numerous issues that we believe exist in SystemVerilog today that will be modified when the IEEE process is undertaken.

Cadence believes there have been three specific opportunities where this commitment could have been shown by the Board of Directors, and such a commitment was not made.

Upon completion and approval of the SystemVerilog 3.0 donation in June 2002 the standard should have been moved to the IEEE. The lack of this donation serves only to delay the IEEE process. If SystemVerilog 3.0 was sufficiently unstable that it was unfit for donation to the IEEE then it should not have been approved as an Accellera.

Again on January 16 2003, Cadence attempted to get a clarification of intent from the Accellera Board of Directors. A motion for a SystemVerilog 3.0 donation immediately was defeated 6 negative, 4 positive, 1 abstain by the 11 member board.

At that same Board meeting, a motion was passed committing only to a donation at an unspecified future time to an unspecified IEEE committee leaving too much of a lack of commitment to accelerating the 1364 standard.

2.2 Content Has Suffered from Immovable Deadlines

The Accellera technical chairperson has laid out milestones for completing various activities that are required to happen before a new standard is produced. Such milestones are absolutely necessary,

[2] The IEEE process for a negative ballot response is explained in The IEEE Standards Companion - Annex B.

especially when working with an organization composed primarily of volunteers. Without them, standards work would always take second place to other activities and progress would be brutally slow. However, we believe that the milestones should be treated as checkpoints at which one can measure the progress, not immovable dates that must be met at the expense of the quality of the standard.

The deadlines for the SystemVerilog 3.1 Draft 4 LRM were constructed to coincide with the DVCon conference. The motivation for this was to be able to announce completion of the draft standard in a very public way. Despite diligent effort by many contributors this deadline was not met. This should have been an indication that the timeline for the final standard was in jeopardy and the dates should be re-examined. However, this did not happen.

The deadlines for the final draft that will be voted on by the Board of Directors are set such that the vote will occur before DAC 2003. Again, this is motivated by being able to make a marketing statement at DAC about the acceptance of the standard (assuming it is approved). In the rush to meet this milestone the final LRM review period between the Draft 4 and Draft 5 LRM was shortened to 2 weeks. During this short period over 258 recorded issues were raised by reviewers. In the one week review period between Draft 5 and Draft 6 an additional 88 issues were identified. Each of these should have been individually addressed by the committees. Instead a quick triage of the incoming issues was performed by a single engineer, and any non-trivial issues were forwarded to a single “champion” in each of the committees.

This triage and champion process was a complete breakdown of the established procedures of presenting issues, coming to consensus and having a committee vote on the content. The specific individuals involved did a phenomenal job of attempting to end up with reasonable content given this rate of input, but the process was certainly out of control with the sole justification being completion of the standards process prior to DAC. No member company would accept such a complete breakdown of the established decision making processes. We believe Accellera should not accept it either.

2.3 Lack of General Extension Philosophy and Requirements

Extending a ubiquitous language like Verilog must be done extremely carefully with backward-compatibility being the foremost concern. We believe that the first step in extending Verilog should be consensus on the techniques that should be used, and possibly fundamental changes to the language to permit future extension.

During the SystemVerilog review process every time a new functional operation was required the debate began on whether it should be a keyword, an operator, a system task, a method, or something else. This debate was undertaken for every single change individually and as a result these concepts were applied inconsistently in different places. Instead a fundamental discussion of when each of these techniques is appropriate and whether additional new techniques could be used should have been the first and only discussion on this topic. The rules established by such a process for classifying when operators, system tasks, methods, or keywords are appropriate could then just have been followed, not revisited on every feature.

In the section 3 of this analysis many examples of this lack of basic agreement on philosophy and requirements are found surrounding not only operators but keywords, attributes, and pragmas.

2.4 Layering vs. Integration; Implementation vs. Design

Much of the content of SystemVerilog 3.0 and 3.1 is derived from the donation to Accellera of portions of SuperLog and Vera. These languages were certainly technological progress in system-level modeling and verification arenas. However, the mantra throughout the committee work processing these donations has been “It was implemented that way in Superlog (Vera) and therefore it is good”. This has resulted in a language where much of the ongoing activity has been to layer the content of these languages on top of what already existed in Verilog rather than to integrate the content into the language. Often very similar concepts had different keywords in the languages (sometimes in all three of Verilog, Superlog, and Vera) and instead of agreeing that one of them would survive and be extended to satisfy the capabilities of the other languages, all of them were added to the language creating duplicate functionality, increased complexity and unnecessary keywords.

2.5 Missing Functionality

In the process of extending Verilog one must be careful to extend all relevant portions of the language. Although SystemVerilog has extended the syntax of the language, it neglected to update SDF, VPI, or VCD for the extended language features.

2.5.1 No SDF Specification

System-level models are written (among other things) to verify functionality and system-level throughput. A critical component of system-level throughput is delays in and between components. SDF is capable of expressing these delays with arbitrary precisions. Allowing SDF annotation of system-level models would allow these performance tuning parameters to be modified simply by changing SDF and not the design, exactly as it is done for gate-level models.

Defining SDF for delays through interfaces, clocking domains, and program blocks would allow this sort of use of SDF. However no discussion of how SDF applies to these constructs is given in the SystemVerilog LRM. While expanding this would be one solution, the remedy preferred by Cadence is to collapse modules, interfaces, clocking domains, and program blocks into the existing module construct and simply using the existing SDF specification (and eliminate more keywords along the way).

2.5.2 No VPI interfaces

One of the top reasons for the success of Verilog as an HDL is the existence of PLI and VPI. It allows users to write their own simulation extensions, design rule checkers, and internal tools. It also promotes a strong breeding ground for EDA startups to add significant value to the end-user by piggy-backing interesting new functionality on top of standard simulation interfaces.

Some claim that it is standard practice for the VPI to lag behind the specification of the language extensions. This is indeed the case while IEEE committee work is ongoing. First new language additions are discussed by the Behavioral Task Force, and then the PLI Task Force follows along with a VPI data model for it. However, the 1364 standard was not deemed to be ready for a ballot until the VPI had caught up and was complete.

Some committee members also claim that the new Direct Programming Interface (DPI) will minimize the need for VPI. This is simply not true. VPI provides for walking the entire elaborated model. DPI only provides a facility calling back and forth to ‘C’ at simulation runtime. A Direct Programming Interface is useful but does not in any way supplant the need for VPI for every language construct.

3 Technical Comments

This section contains a subsection for each chapter of the SystemVerilog LRM. Although we do not have comments on each chapter this organization will make it simpler for readers to cross-reference to the actual draft LRM. Only the top section corresponds to the specification chapter, not each sub-section.

Many of the points brought out in these sections have already been discussed in detail in Accellera committee work and on the email reflectors, therefore in many cases detailed descriptions of many of these issues are not provided. These sections simply point out the areas where Cadence believes there are unresolved issues with a brief description of the problem and/or our position.

3.1 Section 1 - Overview

This section provides only informative content that describes the content of SystemVerilog 3.1; therefore there is no specific technical feedback on this section. However, this section does emphasize that SystemVerilog 3.1 is an extension of the IEEE Verilog 1364 standard. Cadence continues to believe that this very process of Accellera extending an existing IEEE standard outside of the IEEE working groups is an unwise decision. Accellera’s work as an incubator for standards where no industry standard exists is important work, but the creation of committees separate from the IEEE committees for existing IEEE standards can only serve to create divergence and confusion.

Accellera does provide an important role in the support of IEEE standardization work through the funding of editing, conferences, web space, and publicity activities. This is a very important role to promote industry wide standards adoption that Cadence whole-heartedly supports.

3.2 Section 2 - Literal Values

The section on literal values extends literals in the language to be able to express the new data types in SystemVerilog. Note there is no mention of how literals relate to the new class types at all.

3.2.1 Array and Structural Literal Syntax

References: 2.7, 2.8

As of the Draft 5 LRM available when committee members are being asked to vote, there is no BNF given for the syntax of an array or structural literal

Both the array and structure literals borrow initialization syntax from ‘C’. The examples given use {} as in ‘C’ which creates a syntactic ambiguity with the existing 1364 concatenation operator. This is an example of where the integration of Verilog content from another language is being layered rather than integrated.

3.3 Section 3 - Data Types

One of the most significant contributions of SystemVerilog is the extension of data types. Cadence believes that data types in Verilog should certainly be extended, but that this should occur through a careful specification of a type system that:

- Allows data types on both variables and nets
- Defines a small number of basic types
- Defines mechanisms for creating aggregate types from the basic types
- Defines standard-defined types composed of the basic types using the same mechanisms that are utilized for user-defined types

SystemVerilog data type’s extensions do none of these things. Content was taken from ‘C’, Superlog, and Vera and layered onto the 1364 language without begin integrated. This has created numerous technical and stylistic problems for which descriptions are given below.

3.3.1 Data type syntax

References: 3.2

The syntax for data types has been extended in two different ways. The first is to define a mechanism for creating composite or aggregate data types. These include structures, unions, enumerations, arrays, and classes. Although we have some specific comments below on how this could be accomplished with fewer keywords and better integration between these type extensions, there is clearly a need for more complete composite type support in Verilog and SystemVerilog makes progress in this area.

The second mechanism is to define numerous new predefined types which are defined as new keywords. These include: **byte**, **shortint**, **int**, **longint**, **bit**, **logic**, **void**, **shortreal**, **string** and **mailbox**. This addition of new data types as keywords has numerous problems.

The syntax defines these data types as specifying both the kind of object (a variable), and the value the object can hold (an 8-bit 2-state value for instance). This is consistent with how Verilog-1364 defines the basic **reg** and **integer** types, but as the use of types increase in Verilog it is entirely too restrictive. In particular it will quickly be necessary to allow these types to occur on nets as well as variables.

Nets provide two important pieces of functionality. They allow resolution of multiple drivers, and they allow values to be scheduled with delay. We believe that it is beyond the current extensions to Verilog to define resolution of composite values (a la VHDL), but it is not beyond scope to allow composite values to occur on nets for scheduling purposes. It will be extremely natural to create a net of a composite value that represents interconnect between two system-level models. In particular the ability to schedule a delay for this value to propagate will be extremely useful for performance and throughput analysis.

Following the extension techniques in SystemVerilog today, if data types were to be added to nets, then entirely new names for the types will have to be created as was done by Verilog-AMS for the **wreal** type. This will only further compound the problem of exploding number of keywords in the language.

By defining these as new keywords a non-sustainable language extension mechanism is being utilized. Instead, we would prefer to see these necessary types defined as standard-defined types using the same extension mechanism as is available to users. For instance:

```
typedef logic [7:0]  byte;
```

Along with extensions to `include discussed later, these types could then be `included in modules where they are needed and made visible. In modules where they are not necessary they are not `included and therefore do not pollute the identifier namespace or create backward-compatibility issues.

3.3.2 Integer data types

*References:	3.3*

The integer data types (byte, shortint, int, longint, etc) in SystemVerilog are motivated by the desire to add ‘C’-like integer data to Verilog. We agree with the need for this addition, but the mechanism of adding new keywords as discussed in the previous section is not maintainable. It also has the same failing the ‘C’ types do in the presence of different width data types on different compilers/operating systems. Verilog integer data, in general, will be of a known maximum width. Allowing the compiler or OS to change this width will lead to the same portability issues that ‘C’ programs have across 32 and 64-bit platforms.

The need to pass these data types from Verilog to ‘C’ is well-understood, but simply using the same identifier for them does not solve the problem. An interface that defines the actual representation of these objects in Verilog so that they can be manipulated meaningfully in 'C' is necessary. More comments on this concept are given in the section on DPI (see 3.26) below.

This section goes on to state that 2-state types are at least in part motivated for the desire for faster simulation. Cadence absolutely supports the addition of a basic 2-state data type, but cautions against setting the expectation that they will simulate significantly faster than 4-state data types. The overwhelming factor in simulation performance is the actual loading of a data value. On today's computer architectures, if a value is available in cache then the load is very fast, if not then it can take a significant number of clock cycles. Arithmetic computations on data once they are loaded into local caches or registers are extremely fast on today's processors; the arithmetic operators come almost for free. The widespread use of 2-state data may decrease the total memory footprint, having an effect on the caching of data, and therefore an impact on capacity and performance, but the performance from doing less arithmetic calculations on a 2-state value will be minimal.

3.3.3 Real data types

References: 3.5

The new keywords for new real data types suffer from exactly the same problems created by new integer data types. Instead a system for defining floating point and fixed point representations should be constructed so that users can define the real numbers they need. 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 work in this area can be found at http://www.eda.org/fphdl.

3.3.4 Void data type

References: 3.6

The void data type is motivated by the changes to the definition of Verilog functions. In particular if a function returns no value or if a function is used as a statement and the returned value is to be discarded (i.e. cast to void) then this type is necessary. Cadence believes that the SystemVerilog extensions to functions to be used in this way are not a good extension of Verilog as explained below (see 3.10). If these extensions are not made, then the void type is unnecessary and its removal eliminates yet another keyword.

3.3.5 String data type

References: 3.8

One of the general issues discussed above was the lack of a consistent application of extension technology. Strings and the methods defined on them are a classic example. If strings are being added as a fundamental data type then operators or system tasks should be defined to operate on them as is done for all other Verilog data types. The extension of method syntax to objects that are not defined as classes creates a situation where it is difficult for a user to remember how the extension was made. This makes the language harder to write and learn. Either strings should be defined as a new standard-defined class and that class would contain these methods, or the operators on strings should be defined as system functions.

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?

3.3.6 Event data type

References: 3.9

SystemVerilog extends named events by promoting them to a full-blown data type including assignment and comparison operators. Allowing an event as a variable is a fine extension of Verilog and is required to allow events to be embedded in other data types such as a struct. However, SystemVerilog defines events as a “handle to a synchronization object”. The new events are in effect a new dynamically allocated object to which the user is given a handle. These handles are a hybrid between the dynamic behavior now defined in classes and the statically allocated behavior of all other data types.

This is a further example of where an idea, in this case dynamic allocation, is only included in SystemVerilog in limited contexts (classes, strings, events). Instead we would prefer to see the specification of how all types can be both statically and dynamically allocated and the interaction of these techniques cleanly defined. The piecemeal addition in limited contexts makes the definition very difficult and will require revisiting the issue later to define dynamic allocation for all other types.

3.3.7 Enumeration types

References: 3.11

Enumerations in SystemVerilog have been dealt with in two separate committees. The sv-bc has been attempting to clarify the meaning in SystemVerilog 3.0, while the sv-ec has been extending enumeration types to correspond with how they were dealt with in the Vera donation. This has led to endless debate about the intent of enumeration types. The LRM still contains language like “SystemVerilog enumerated types are strongly typed”. However, they are also allowed to be assigned into unions, and casts can assign out-of-range values into an enumeration.

Cadence believes that Verilog should not contain any “strongly-typed” integral values. Enumerations should simply be treated as a set of named constants that make designs easier to read and allow explicit encodings of pneumonic identifiers. It will be incredibly common for values specified as enumerations to be assigned into other integer objects, registers, or assigned onto nets. Therefore, 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. As a debug or lint check, this sort of runtime checking may or may not be enabled by a vendor, but it is not in the spirit of Verilog to do this on every assignment and could have significant performance impact.

Enumerations are another case where new operators were defined as methods on objects that are not classes. Either this should be adopted for all data types and retrofitted into existing Verilog types or limited only to classes.

3.3.8 Structures and Unions

References: 3.12

The addition of a **struct** type is absolutely a required addition to Verilog and Cadence supports its addition. We do however have some objections to some of the specific content.

Special considerations are given for 2-state and 4-state vectors in unions. If an object contains two overlapping union members, one of which is a 2-state object and one of which is a 4-state value, then they are specified as being interchangeable. For instance you can write into the 4-state value and read it out as a 2-state value. 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 purpose of 2-state is, at least in part, to allow a more compact representation. This special case would cause a vendor to have two separate implementations of 2-state based on how the object overlaid 4-state objects.

A second consideration, related to the next section on classes, is the lack of a method of dynamically allocating structure objects (or any type for that matter). As soon as a **struct** type is added to the language, then you immediately have the power to construct complex data types that require some form of dynamic allocation. During the SystemVerilog development process different versions of “handles”, “references”, and “pointers” were discussed. Regardless of the terminology used, the creation of dynamic data structures should be a fundamental extension of structs. The best solution for this, in our opinion, is to unify the **struct** and **class** into a single extension mechanism as discussed in the next section.

Finally, the specification of struct implies compatibility with ‘C’.

> ”An unpacked structure has an implementation-dependent packing, normally matching the C compiler.”

The sv-cc committee prepared a more detailed description which was never discussed in the sv-bc committee. No resolution has been made to date therefore this leaves this section poorly defined: what does “normally” mean? The sentence about unpacked structs should either be removed or a complete description for all complex data types should be added in the ‘C’ API section where it matters.

3.3.9 Classes

References: 3.13

Classes form a fundamental extension to the type system by allowing dynamic memory allocation, single inheritance and data specific methods. The success of languages such as ‘e’ and Vera has shown how these concepts are extremely useful in verification environments. The fundamental issue Cadence has with classes today is that they are restricted to being dynamically allocated. All other data types in Verilog are statically allocated. 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. Structs are simply classes that do not inherit from any other class, and do not define any methods.

This unification will be most useful when you consider system-level models instead of just testbenches. Today with classes and structs as separate items the user must choose the language construct based on the type of memory allocation they want. If they begin by modeling with classes to get dynamic memory for a specific data structure (perhaps an infinite length queue) at some point they will have to change to a struct type as they refine the design to a fixed length, statically allocated piece of memory. If classes were allowed to be statically declared, and structs dynamically allocated, then the user would instead make the more intuitive decision on whether they wanted to utilize object-oriented programming techniques and then could stick with that decision as they refined from infinite to fixed length.

3.4 Section 4 - Arrays

References: 4

For the most part, the content of the chapter on arrays is non-controversial and extends existing 1364 in a reasonable way. There are a few outstanding issues that the committees have discussed that we believe need further resolution.

3.4.1 Longest Static Prefix

References: 4

The concept of the sensitivity of a process when an array is a part of the sensitivity list can have both performance and complexity of implementation considerations. There was significant debate in the sv-bc committee around what the rules for determining the longest static prefix of an array and the effect on expressive power and implementation complexity/performance. The final resolution was that this concept was left unspecified; this will create ambiguous interpretations in different implementations which will hinder vendor interoperability.

3.4.2 Dynamic Arrays

References: 4.6

In SystemVerilog 3.1, dynamic arrays were introduced through Vera and were restricted to be one dimensional arrays. The array size is allowed to be set or changed at runtime. They can be declared anywhere and allocated with the **new** method. The LRM also says that:

> "A subroutine that accepts a dynamic array can be passed a dynamic array of a compatible type or a one-dimensional fixed-size array of a compatible type."

The DPI interface adds the concept of "open arrays" for denoting unconstrained array types which can *only* be the type of DPI function formal arguments. This allows one to write general purpose ‘C’ functions which can handle arrays of any size. Open arrays have the same syntax as dynamic arrays ([]) but allow open sizes for any and multiple array dimensions. The size of the open dimension is determined at runtime by the actual arguments to the function call.

This issue of open versus dynamic arrays was discussed at length in the sv-cc committee. The sv-cc committee felt that the current restrictions on dynamic arrays should be removed and felt that the name dynamic array was a misnomer, when the dynamic array syntax [] appeared as a task/function formal argument. Removing the restriction on the number of dimensions and changing the name of dynamic to open arrays was discussed.

This illustrates the lack of synchronization between the sub-technical committees resulting in inconsistency in the LRM. If both dynamic and open array terminology stay in the standard and use the same syntax, users will be terribly confused. In one case, such an array will represent a dynamically allocated one-dimensional array, in the other (with the same syntax) it will refer to an unconstrained formal array type.

Note that Cadence does not necessarily believe that it should be the purpose of a direct C interface to deal with generic open arrays (see section 3.26.7 for details). Such open arrays have to be accessed in an abstract way through DPI query and access functions which causes overlap with existing VPI functionality.

3.4.3 Associative Arrays

References: 4.9

Associative arrays are defined with limitations on the data types that can be used to index the array. The allowed types are integral, string, class, packed arrays, and packed structs. This is another example of limiting what could have been defined as a generic language extension. Defining associative arrays such that any type can be used as the index would actually have been a simpler process than restricting it and would be extremely useful for system-level models. In particular implementing content-addressable memory where the items being stored may be any arbitrary type would be extremely helpful. There are some implementation concerns but in a sufficiently abstract model the expressive benefit and simplicity outweigh the implementation complexity.

Associative arrays indexed by class also have an issue that we believe must be addressed. The traversal order of iterating across all the items in the array is specified as “deterministic but arbitrary”. In committee discussions this was deemed to mean that the same simulator will do the traversal the same way every time, but different simulators might yield different orders. We believe that this will be a major problem for customers utilizing more than one vendor’s simulation tools. Just like $random in the 1364 standard, customers will expect a way to get deterministic results across all vendors.

Finally, Associative arrays are also another location where methods are defined on non-class data.

3.5 Section 5 - Data Declarations

3.5.1 Data Declaration Syntax

References: 5.2, 10.2

SystemVerilog 3.1 has allowed the use of **static** as a lifetime declaration in automatic tasks and functions. This exact functionality was considered and rejected by the IEEE 1364 working group. Note that this is not a case where 1364 rejected the functionality because there was just no good definition of what it would do, or there was insufficient time to consider the proposal. The exact same content was proposed, voted on, and rejected. The construct in question allows a variable to be declared as static in an automatic function or task. It is always possible to simply declare the variable outside the function/task in a module and then reference it in the function/task. This places the static object where it belongs (i.e. in the module) and the automatic data where it belongs (i.e. in the task/function).

This is the kind of extension that Cadence expects will be revisited by the IEEE and that their original decision on this issue will stand. Until this issue is vetted by the 1364 group a user adopting this style is in danger of having their code not work in the future 1364 standard.

3.5.2 Constants

References: 5.3

A new form of constant that can exist in tasks and functions has been added to SystemVerilog. This in general is a harmless addition and does add some expressive power; however the specification is incomplete when talking about initialization of constants. 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. Due to defparams, Verilog already suffers from this ambiguity in definition of constants so this adds no additional ambiguity; however, it would have been nice to avoid this known problem by making hierarchical circularity illegal in this case.

3.5.3 Variable Initialization

References: 5.4

The specification of variable initialization in SystemVerilog is:

> “In Verilog-2001, an initialization value specified as part of the declaration is executed as if the assignment were made from an initial block, after simulation is started. Therefore, the initialization may cause an event on that variable at simulation time zero.

> In SystemVerilog, setting the initial value of a static variable as part of the variable declaration shall occur before any **initial** or **always** blocks are started, and so does not generate an event.”

The argument made in favor of this change is that it simply makes the use of variable initialization in a procedural context deterministic. This argument has nothing to do with why we believe this is a non-backward compatible change. 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.

Consider the following example:

```
module init;
        integer var_i = 1;    // A variable with an initial value
        wire [31:0] wire_i;  //  A wire

       assign wire_i = var_i;  // Continuously assign the wire the variable’s value

        initial #1 $display("wire_i is %d\n", wire_i); // display the wire
endmodule
```

In Verilog 1364, the initial value on *var_i* is guaranteed to produce an event. This event is critical because it causes the continuous assignment to the wire ***wire_i*** to execute. Without this event, the continuous assignment does not execute at time 0 and therefore the initial value of the variable would not propagate to the wire, leaving the wire at the default value of 32’bz. The exact same problem would occur if a gate were substituted for the continuous assignment above.

In Verilog 1364 the code snippet above would produce a 1 on the wire_i, in SystemVerilog a 32’bz would be produced.

This is not a trivial problem. The vast majority of Verilog modules have this style of code wherein an internal value is calculated and stored in a register and then the value is propagated either through a continuous assignment, buffer, or port onto a wire. Any of these forms of interconnect would not propagate the initial value in SystemVerilog. This would cause most devices to propagate the default value of ‘z’ on a wire instead leading to catastrophic simulation failures.

3.5.4 Automatic Variables

References: 5.5, 10.2

In addition to reintroducing the previously rejected **static** keyword in tasks and functions, SystemVerilog also allows explicit declaration of some data elements as **automatic**. Cadence sees absolutely no need for this extension. Variables in automatic tasks and functions are already automatic so no explicit indication is necessary. Having automatic variables in any other static context just does not make sense. What does it mean to call a static task and have some of the data in it allocated automatically? The only purposes we can possibly imagine are to make the data not available as a hierarchical reference, or to ensure that the item is initialized each time the function/task is called. If a way to create data that cannot be referenced hierarchically is desired then we should extend Verilog to have public and private data. This has already begun in the section on classes. Initialization is also easily handled by just initializing the data procedurally in the task or function.

Note that the **automatic** declaration is even allowed in initial and always blocks. We have no idea why these purely static contexts need to have automatic data.

3.5.5 Variables in Unnamed Blocks

References: 5.5

SystemVerilog adds the capability to declare objects in unnamed blocks as well as in named blocks. This data is visible to the unnamed block and any nested blocks below it. Hierarchical references cannot be used to access this data by name.

We have several objections to this. First, the behavior of Verilog such as VCD dumping, $display (%m) etc. is not described with respect to these variables. Secondly, the original intent for allowing variables declared in unnamed blocks was to be able to hide data declarations from scopes above. The same concept is brought into the language through classes which have the possibility of declaring public, protected or private data declarations. These two methods of declaring private data should be reconciled.

3.6 Section 6 - Attributes

References: 6

The purpose of this section is to define attributes on interfaces and modports. Cadence has no issues with the content of this section. We do believe this section is probably completely unnecessary as attributes should just be defined in the syntax for interfaces and modports.

3.7 Section 7 - Operators and Expressions

SystemVerilog borrows many operator and expression concepts from ‘C’. While some of these concepts are useful and allow for more succinct code, others actually introduce coding styles that can lead to misunderstandings or even ambiguous behavior between different implementations. Specific examples are given below.

3.7.1 Assignments in expressions

References: 7.3

SystemVerilog now allows assignments in expressions. The specific example of given in the LRM for this construct is:

```
If ( (a = b))  b = (a += 1);
```

Note that this single line assigns two values to ‘a’ and tests it. Cadence believes this sort of shorthand is borrowing some of the worst features of ‘C’. Most ‘C’ coding styles guidelines explicitly disallow this form of assignment in ‘C’ and adding it to SystemVerilog adds no expressive power.

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

References: 7.3

Also brought from ‘C’ are the increment and decrement operators. In general, Cadence believes these are a reasonable addition as statements in the language, but there inclusion in expressions creates syntax that is easily interpreted ambiguously. The SystemVerilog LRM says

> “The ordering of assignment operations relative to any other operation within an expression is undefined. An implementation may warn whenever a variable is both written and read-or-written within an integral expression or in other contexts where an implementation cannot guarantee order of evaluation.”

We believe that if this new construct is so ill-defined that it must have such a caveat explicitly attached to it in the LRM, then it should not be added in the language. Auto-increment and auto-decrement in expressions should be disallowed.

3.7.3 Built-in methods

References: 7.10

The concept of built-in methods deserves recognition for creating a construct that extends the language without creating conflicts with the existing operators; however there are some issues with built-in method definitions.

The first issue is that the existing hierarchical ‘.’ operator is used to separate an object from methods called on the object. This follows the conventions in ‘C++’ and Java, but the pervasive nature of hierarchical references in Verilog make it more confusing in Verilog. This problem is compounded by the fact that methods which take no arguments have been allowed to have the parenthesis be optional when the method is called. An expression such as:

```
r = a.b.c
```

Could either be a hierarchical reference or a call of a method on the object a.b. In the most extreme case, this could be a call to a method on the object a, followed by either a hierarchical reference to ‘c’ or another method call. These options could be written as:

```
      r = a.b.c()
or
      r = a.b().c
or
      r = a.b().c()
```

This ambiguity could have been clarified either by making the method operator something other than ‘.’, or by requiring the () on null argument lists.

The second issue with built-in methods is that they are not differentiated in any way from user-defined methods. In future Verilog revisions new methods will certainly be added by the standard. A unique namespace for the methods defined by the standard should be created so that they will not conflict with user-defined methods. The standard should prefix all standard-defined methods with a unique character such as ‘$’, or ‘_’. User-defined methods should be precluded from using these identifiers for methods thereby ensuring that conflicts in the future will be completely avoided.

3.7.4 Literal expressions

References: 7.12, 7.13, 7.14

There are three sections that define how unpacked array expressions, structure expressions, and aggregate expressions are dealt with. All of these expression forms use the {} notation to form the expression. No BNF syntax description for these expression types is given in the LRM. This is essentially the same omission as was mentioned previously by not providing the BNF for structure or array initialization expressions. 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.

A statement such as

```
x = {1, 0, 1};
```

could either be a concatenation being assigned to any packed type, an assignment to an unpacked struct, or an assignment to an unpacked array. Only by examining the left-hand side of the assignment can the context be determined. This problem is even worse when a literal is used as an argument to a task.

```
T ( {1, 0, 1} );
```

In this case, only after elaboration when the specific task being called is resolved and its arguments can be examined can the meaning of the literal be determined. This will lead to error messages very late in the compile flow and could impair optimizations because such decisions must be deferred.

3.8 Section 8 - Procedural Statements and Control Flow

References: 8

SystemVerilog adds some of the procedural statements that exist in ‘C’ but that are not available in Verilog. Some of these such as ‘continue’, and ‘return’ are simple changes supported by Cadence; however the behavioral extensions go far beyond these simple extensions and create keywords where they are unnecessary, and have possibly very significant negative performance impact.

3.8.1 Unique/Priority Keywords

References: 8.4

SystemVerilog adds the keywords **unique** and **priority** in front of **if** and **case** statements to indicate that the individual branches should have specific relationships. The **unique** relationship indicates that the branches do not overlap, and the **priority** relationship indicates that they should be evaluated in priority order. These are roughly equivalent to the commonly used parallel and full case pragmas.

This is a case where keywords are being overused. Verilog already contains a mechanism whereby information can be associated with a given statement. That mechanism is attributes. A reasonable way of adding lint-like semantic checks to any statement is to add standard-defined attributes that must be interpreted by tools. For instance if a completely new namespace came into existence for standard-defined attributes, it might begin with %. So if a **%unique** attribute was defined for SystemVerilog then instead of:

```
unique if (XXX)
```

the syntax

```
(* %unique *) if (XXX)
```

could be used. Since this alternative adds no keywords it is a significantly better enhancement to the language while capturing all the same information. Similar to extending standard-defined methods, the establishment of standard-defined attributes allows them to be created in a new private namespace so that they will never conflict with user-defined attributes or identifiers of any kind.

The **unique** and **priority** keywords are the first examples thus far that add lint-like capability into Verilog in the form of keywords. Cadence believes that this is inappropriate due to the impact on backward compatibility, and that standard-defined attributes or pragmas could be used equally as effectively with many fewer conflicts.

3.8.2 Performance of Unique

References: 8.4

A second concern with the **unique** keyword is that it can have a potentially severe impact on simulation performance. In the worst case, in order to detect conflicts between each and every branch of an if/case statement, every condition in every branch would have to be evaluated every time the statement is encountered. Sometimes static analysis of the branches could be used to limit this impact but the complexity of this optimization is substantial and often all branches would need to be evaluated.

3.8.3 Final blocks

References: 8.7

SystemVerilog adds an interesting feature, the final block. The final block executes at the end of simulation and is provided to allow a user or application to summarize behavior or coverage prior to exiting simulation. This is certainly a worthwhile addition to the language. Cadence believes that the specification of these blocks is incomplete.

Final blocks are not prohibited from assigning to registers or wires in any way and if they do perform these assignments, the behavior is undefined. If final blocks are allowed to do assignments (even in zero-delay) then they may actual trigger additional simulation cycles such that the summary information gathered may be inaccurate.

Final blocks should be defined at a minimum as not allowing assignments to any object to which any other construct is currently sensitive.

3.9 Section 9 - Processes

SystemVerilog defines new forms of processes that extend the functionality available in **always** and **initial** blocks. We believe that most of these are improperly formed extensions that could have been accomplished by adding information to the existing constructs rather than inventing entirely new forms. The details are provided below.

3.9.1 Ordering requirement on final blocks

References: 9.1

SystemVerilog contains the following statement:

> “SystemVerilog does not specify the ordering [of final blocks], but implementations should define rules that will preserve the ordering between runs.”

Cadence is unclear on exactly what we are being required to implement here. Between which runs are we required to preserve the ordering of final blocks: multiple runs of the same design with no changes; runs where only behavior changes are made; runs where structural changes are made; runs where one previously simulated block is completely subsumed in another block? We believe that if the language can not define an ordering, then no mention of the ordering should be made at all.

3.9.2 Always_comb, always_latch, always_ff as semantic checks

References: 9.2, 9.3, 9.4

SystemVerilog adds three new forms of always block by introducing new keywords. These new forms imply a coding style that the block is combinatorial logic, a latch, or a flip-flop. Identifying the intent of an always block in this way does allow static and dynamic checks to be performed to ensure that a design criteria is being met, and may allow both simulation and synthesis tools to optimize the block and achieve better quality of results. However, using a keyword for this purpose is inappropriate. It adds unnecessary keywords, and is not a long-term sustainable method of extending the language.

Once again, we would return to the existing attribute mechanism to associate additional information with a statement. That is exactly what attributes are for; Verilog should begin to use them as standard-defined attributes. In this case something like:

```
(* %comb *) always
(* %latch *) always
(* %ff *) always
```

could be used.

We call the keyword method a non-sustainable method for extension because it does not allow multiple characteristics to be associated with the same block. For instance, imagine a future extension that would allow an always block to run in a specific region of a simulation cycle. Instead of introducing a new keyword (such as **always_observe**) to do this, an attribute on the block could be used.

```
      (* %observe *) always
```

Then if you had a block which was both scheduled and combinatorial, both attributes could be associated with it.

```
      (* %observe, %comb *) always
```

In the keyword method you would have to further expand the keyword space with something silly like **always_observe_comb**.

3.9.3 Always_comb sensitivity

References: 9.2

Similar to the @(*) functionality of 1364, the always_comb block is implicitly sensitive to all of the objects on the right-hand side of any statements in the block. However, SystemVerilog takes this one step further and requires an analysis of any functions called from the block and a tool must create sensitivity to any objects referenced globally by those functions. Note that this process is recursive so that any functions called by those functions must always be considered as well, ad infinitum.

The only case where a function can add to the sensitivity of the always_comb is if it references an object as a hierarchical reference. This is a dangerous coding style that adds side-effects to functions. Because function calls themselves can be hierarchical references that may not resolved until after elaboration, a tool has no idea where a given function definition is used when that definition is parsed. In order to add this check to always_comb, every function would be required to keep a list of any objects that are referenced hierarchically. Then at code generation time, all possible call chains would need to be analyzed recursively to detect if a function with a side-effect is called and that hierarchical reference would need to be added to the sensitivity list.

Cadence believes that the implementation complexity of this additional sensitivity check is not justified by the expressive power added to the language. 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.

3.9.4 Always_comb overlap with @(*)

References: 9.2

The new always_comb statement is extremely similar to the existing @(*) sensitivity in Verilog 2001. It has similar sensitivity semantics as discussed in the previous section. However, always_comb adds a lint-like capability to ensure that the contents of the always block are in fact combinatorial logic. 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.

In our opinion, the IEEE will almost certainly reconcile these two different forms into one.

3.10 Section 10 - Tasks and Functions

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. The sections below enumerate specific objections.

3.10.1 Function output and inout argument modes

References: 10.3

Functions execute in zero time and, except for hierarchical references, can not have side-effects; their only effect is through their return value. SystemVerilog allows the modes output and inout on function arguments. This change blurs the line between tasks and functions in Verilog and adds significant opportunity to have function side-effects. If a subprogram is supposed to modify multiple arguments then it should be written as a task or an automatic task.

3.10.2 Functions as statements

References: 10.3.1

A function is a mechanism in Verilog to create an operand that can exist in expressions. With the addition of automatic tasks in Verilog 2001, the same scope and lifetime can easily be achieved. There is simply no need for this change other than to be more 'C'-like.

3.10.3 Void functions

References: 10.3.1, 10.3.2

Void functions are only necessary if functions have output arguments and therefore act like tasks. Since we object to adding output and inout arguments to functions, this change is simply unnecessary.

3.10.4 Implicit task and function lifetime

References: 10.4

SystemVerilog allows the lifetime specifier **automatic** and **static** to be placed on a module, interface, and program block declaration in order to indicate that all tasks declared in that module will have the specified lifetime. Placing this specifier on a module may be viewed as a convenient shortcut, but it creates a non-local syntax that has severe effects on the behavior of tasks declared in that module. If for instance, a module is declared in this fashion and then a utility task made visible in the module through a `include, then the task becomes automatic even though it was originally intent was static.

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.

3.10.5 Lack of shared library support in DPI naming

References: 10.6

The specification of the name for a function imported from ‘C’ to SystemVerilog should include a component that represents a shared library name. Applications will commonly want to provide ‘C’ functions in shared libraries as this is a common distribution mechanism in ‘C’. This will also serve to minimize name conflicts between multiple applications all running in the same simulation.

3.11 Section 11 - Classes

Classes are a solid extension of Verilog. Languages such as *e* and Vera have demonstrated the power that they can bring to verification environments. The only issue Cadence has with classes as they exist in SystemVerilog is a lack of integration with the overall language as explained below and in previous sections.

3.11.1 Additional common keywords

References: 11.2

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. A careful unification with records and substituting operators for some of these could eliminate most if not all of these keywords.

3.11.2 Parameterized class types

References: 11.22

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. Cadence believes that this tradeoff was not considered strongly enough in the addition of classes of a parameterized type. This construct will have a potentially serious negative impact on compile time and simulation runtime.

3.11.3 Differences in struct and class

References: 11.24

The SystemVerilog LRM dedicates a section to explaining the differences between structs and classes. They are listed as (edited for brevity):

1) SystemVerilog struct are strictly static objects … SystemVerilog objects (i.e. class instances) are exclusively dynamic …

2) SystemVerilog structs are type compatible so long as their bit sizes are the same, thus copy structs of different composition but equal size is allowed. In constrast, SystemVerilog objects are strictly strongly-typed…
3) SystemVerilog objects are implemented using handles, thereby providing C-like pointer functionality. But, SystemVerilog disallows casting handles onto other data types, thus, unlike C, SystemVerilog handles are guaranteed to be safe.
4) SystemVerilog objects form the basis of an Object-Oriented framework that provides true polymorphism. Class inheritance, abstract classes, and dynamic casting are powerful mechanisms that go way beyond the mere encapsulation mechanism provided by structs

Taking each one of these points individually, the first is simply an artifact of the limited way in which struct and class are defined. Unification could make both be static and/or dynamic, essentially unifying them as a single language construct. A struct would simply be a class with no inheritance or methods specified.

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. 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. Cadence views the fact that classes are strongly typed in this way as a problem, not a benefit. 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 makes refining from a class-level specification to a struct-level realization in hardware very difficult.

The third is not a difference between structs and classes at all. It is simply a statement that it is possible to have relatively safe memory allocation compared to ‘C’. Dynamic allocation of other SystemVerilog types could follow this exact same paradigm and have exactly the same level of safety.

The last is really just a statement that specifies that no attempt was made to integrate classes into the type framework that exists in SystemVerilog, rather a new kind of type was invented that is assignment incompatible with all other object types.

3.12 Section 12 - Random Constraints

In support of directed random testing techniques, SystemVerilog has added capabilities for constraints and randomization. These are certainly necessary extensions which Cadence supports. We do however have a few issues with the details of the specification.

3.12.1 Limitation to randomizing classes

References: 12

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. It should not be required to adopt classes and inheritance in order to get constraints and randomization. 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.

3.12.2 Rand, randc in a class declaration

References: 12.3

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. In a given simulation it may be desirable to randomize some members of a class and then randomize other members in a different simulation. It may also be that given two objects of the same type you want to randomize different members in the two different objects.

In order to randomizing different members in different simulation (or different members in the same simulation) you would either need to redeclare the class specifying different **rand**/**randc** members, or declare all the members that might possibly be randomized as **rand/randc** and then use $rand_mode() to turn off randomization of the specific members you do not want randomized.

We would prefer to have the ability to randomize a particular variable dynamically as is done with constraints. In that way it is not necessary to redeclare a class or over-specify randomization in order to have a flexible specification.

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.

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.

3.12.3 Constrain, inside, dist, extends, with, solve, before

References: 12

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.

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

References: 12.10

SystemVerilog adds three new random number generator system functions. These provide thread and object stability for random number generation.

The LRM states that these are deterministic in that the same seed will produce the same values in each simulation. This is an important characteristic for regression testing. 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.

Note that this problem originally existed in IEEE 1364 and the $random code was donated by Cadence so that all vendors could utilize the exact same function and thereby guarantee stability even across vendors. Synopsys has been requested to provide the same level of detail by the SystemVerilog committee and has refused to do so. 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.

3.13 Section 13 - Inter-Process Synchronization

This section adds to the synchronization primitives of Verilog. The capabilities added are semaphore, mailbox, and changes to named events.

3.13.1 Semaphores

References: 13.2

The addition of semaphores as a built-in class begins to slide down the slippery slope of how far built-in data types can go. Cadence concurs that the semaphore semantics must be defined as a primitive synchronizer because it can not be otherwise expressed in Verilog. It also adds required functionality. However, we believe that these should not be defined as new language keywords. 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 SystemVerilog 3.1 standard includes the new specification of a `include mechanism that references portions of the standard defined in header files. This mechanism was introduced to support the List class defined in Annex C. The prototypes for the Semaphore class should be similarly defined in a new header file (such as Semaphore.vh) and models which require semaphores should `include this header. This avoids the addition of the keyword and allows for future extension of Verilog using this extension mechanism.

3.13.2 Mailboxes

References: 13.3, 13.4

The mailbox is a second built-in class. The behavior of mailboxes is completely expressible in SystemVerilog. The LRM includes 4 pages of prose that attempt to specify the behavior a compliant simulator must have for mailboxes. Cadence believes that this entire class definition and the implementation of the methods should just be included as a standard library element available using the new `include mechanism. This would completely disambiguate the implementation of a mailbox. 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.

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.

3.13.3 Named Events

References: 13.5, 13.7

Verilog named events have been extended to include a persistent state that is testable throughout a simulation time. The assignment and comparison operators are also now defined. This extension has confused many reviewers who where not a part of the SystemVerilog Enhancement Committee. The LRM states “SystemVerilog events act as handles to synchronization queues”. These are defined as dynamically allocated queues that come into existence explicitly through the declaration of an event, and can be deallocated implicitly when the event is no longer referenced. 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 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.

3.14 Section 14 - SV 3.1 Scheduling Semantics

The SystemVerilog LRM provides a much better specification of the Verilog simulation semantics. This new definition provides for a partial ordering of execution regions into which each language construct can be scheduled. This flexible specification model allows for future extension of the language semantics by adding new regions relative to these regions within the partial order. Within any given region, the order of execution of statements is not specified, reflecting the existing non-determinism in Verilog. In general, this section is exactly how we would like to see the simulation semantics specified. We do have a few specific comments below.

3.14.1 Property Evaluation

References: 14.3

The LRM says:

> “The *observed* region is for the evaluation of the property expressions when they are triggered. It is essential that the signals feeding and producing all the clocks to the property expressions have stabilized, so that the next state of the property expressions can be calculated deterministically.”

This is over-specified with respect to the content of Section 17 Assertions. In Section 17, it is specified that all variables and nets referenced in a property are sampled implicitly at the beginning of a simulation cycle, thereby guaranteeing that properties always have a stable value for all inputs. Cadence objects to this implicit sampling of all inputs as given later (see References:), but it is a part of the language as it stands today. 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.

3.14.2 Delaying of Pass/Fail Code

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. Possible mismatches can occur through this delayed execution. Assertions are executed in the observe region. Between this region and the reactive region, multiple simulation cycles can occur and therefore the values of simulation objects can change. When combined with implicit sampling, this can lead to actually seeing at least three potentially different values for the same variable: the value implicitly sampled, the value when the property executes, and the value when the pass/fail code wakes up in the reactive region. For instance:

```
typedef enum (S_UNKNOWN, S_ACTIVE, S_DONE, S_ERROR) state_e;
state_e state;
state_e next_state;
reg [31:0] data;

always @(posedge clk)
begin
    A1 : next_state = UNKNOWN;    // Initialize next_state

    case (state)                  // compute next_state
    S_UNKNOWN:
        begin
            next_state = S_ACTIVE;
            assert  property ( |data !== 1'bx )    // assert the data has no Xs
                else $display(“state: %d, next_state %d, data %d\n”,
                                state, next_state, data);
        end
    ….
    endcase

    state = next_state;           // assign next_state to state
end
```

In this partial example, the values referenced in the assertion are ‘data’ and (through inference) ‘state’. These variables will be implicitly sampled prior to this simulation cycle and the value stored. 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. Between the execution of the assertion and the $display, the variable ‘state’ will be reassigned due to the last assignment in this always block, so ‘state’ will always equal ‘next_state’, and some other process may have assigned a different value to ‘data’.

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.

3.15 Section 15 - Clocking Domains

Clocking domains add a powerful mechanism for interacting with a design in a cycle accurate manner. This concept originally was donated as a part of Vera and therefore this chapter concentrates on their use in testbenches. Cadence believes that this emphasis should be removed as they add a very powerful general modeling capability; testbenches are just one example of how this could be used. In general, we are highly supportive of the capabilities provided but have some comments below on specific content.

3.15.1 Verbosity of declarations

References: 15

The clocking domain requires that all variables and nets to be referenced in the clocking domain are explicitly redeclared in the clocking domain. This will quickly lead to cases, in a synchronous coding style, where every variable or net is declared both inside and outside the clocking domain. Some form of implicit declaration or namespace inheritance should be included to make this less repetitious (something similar to “.*” and “@*”).

3.15.2 #1step will create non-deterministic IP

References: 15.3

A new time literal **step** was introduced to handle sampling in clocking domains. A **#1step** sample is defined by: “An input skew of 1**step** indicates that the signal is to be sampled at the end of the previous time step.” This is a necessary semantic to have in clocking domains. The problem comes from the fact that the #1step literal itself is defined as: “The step time unit is equal to the global time precision.” 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.

3.15.3 #0 semantics are misleading

References: 15.3

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. A #0 skew waits until the observe region of the current simulation cycle and then samples the values. This occurs after non-blocking assignments have executed for this simulation time. We believe it will be a common error for users to utilize #0 where they intend #1step and get difficult to debug simulation errors. 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.

3.16 Section 16 - Program Block

The program block is a prime example of layering rather than integrating functionality. The program block was the construct equivalent to module in Vera. It provides a declarative scope for shared objects and initial blocks to encapsulate the testbench. It also implies a simulation semantic where all objects declared in the program block are delayed in their execution until the reactive region of simulation. This entire construct should have just been subsumed into the existing module construct for the reasons given in the sections below.

3.16.1 Functional overlap with module

References: 16.1

As with the ‘interface’ discussed earlier, program blocks overlap almost entirely with modules. They have ports, parameters, create a declarative region, and can contain executable code. This is exactly what modules do. 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. The sections which follow show Cadence’s objections to these restrictions in program blocks which, if removed, will make modules and program blocks identical.

3.16.2 Modeling restrictions

References: 16.2

Program blocks are limited to containing initial blocks and task/function declarations to express their functionality. This concept comes from Vera where all programs were dynamic objects and had to be because they were integrated during simulation by executing a PLI task at runtime. When integrating this into Verilog, this restriction is not necessary; they will be elaborated and can be brought into existence statically if that is the user’s desire.

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. Restricting program blocks by not allowing hierarchy in a program block will make this impossible. This restriction is just a legacy from Vera and adds no expressive power; it just limits the user’s flexibility.

Similarly, always blocks are the natural way to express a static process that waits for something to happen. In Vera, all processes had to be dynamically brought into existence, therefore a single initial block which spawned multiple threads was natural. However, when this is integrated into native Verilog, sometimes a static model created by an always block is more convenient. Again, this restriction is just a legacy from Vera and adds no expressive power; it merely limits the user’s flexibility.

3.16.3 Reactive semantics

References: 16.4

Any initial blocks or tasks defined in a program block execute during a special simulation region named the ‘reactive’ region. This executes when all other events are complete for this simulation cycle, but can generate new events. 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 scheduling behavior is 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.

This reactive behavior is also introduced in an attempt to let testbenches “run last” in the simulation cycle. Over the years, many applications have wanted this functionality. A problem always occurs however because no one construct can assume it is running last when more than one is allowed to make this request. Specifying this special status for program blocks is completely artificial and we believe it can actually create verification problems not solutions. A testbench should be coded to act exactly like a system-level model stimulating a device under test. 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.

3.16.4 Termination of all simulation through $exit()

References: 16.6.1

An initial block in a program block can signal it is done executing by calling the new system task $exit(). When all program blocks present have called $exit(), then this simulation terminates. We believe this will cause some simulations to prematurely exit unless users are very careful. Imagine a situation where a user has an existing Verilog test environment, if they adopt SystemVerilog and begin writing program blocks they may have only a single program block. When this one program block is done the user may call $exit(). Since it is the only program block the simulation will exit at this point even if the existing Verilog part of their test has not completed.

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. 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.

Once again, this is an artifact of Vera where the user utilized only Vera for the testbench environment and this may have been a natural way of specifying completeness. This however becomes very dangerous when integrated into native Verilog.

3.17 Section 17 - Assertions

References: 17

The sections that follow contain Cadence comments on SystemVerilog assertions. They do not follow the convention of section by section comment because our issues are more about the interoperability of the different assertion content than about the specific sections.

3.17.1 Complexity

Assertions are abstractions of behavior. They are intended to provide a way to model behavior more simply, with less concern for the detailed implementation. This allows the user to specify intent more clearly, and gives a reference model for behavior against which the actual, detailed implementation can be compared. But System Verilog assertions have lost this notion of simplicity. 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. The rules for clock inference are complicated and provide many opportunities for errors in what should be a simpler, more abstract specification of behavior. Overall, System Verilog assertions appear to be much more difficult to use than plain Verilog (as in OVL), with little or no additional benefit.

3.17.2 Timing Alignment

Assertions are an abstraction of hardware. They should act like hardware, but more abstractly. 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 sampling semantics for concurrent assertions causes these assertions to be out-of-phase with the actual hardware block they are abstracting. What's worse is the fact that this out-of-phaseness is mandated by the language, and it is difficult to undo the effect. Furthermore, a much simpler way of achieving this out-of-phaseness (if it is actually desired by the user) is available - simply delay slightly the clock used hardware design with repect to the clock used in the assertion, rather than pre-fetching all the signals in the assertion. 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 fact that SVA concurrent assertion semantics are defined so that they are required to be out-of-phase with respect to the hardware virtually guarantees that semantic alignment of PSL and SVA will not be possible. PSL assertions are defined in a manner that is consistent with the execution of Verilog, VHDL, and other event-driven hardware description languages. This enables abstract specification of behavior independent of the language used to express the behavior, which will in turn enable assertion-based specification and design. SystemVerilog’s out-of-phase definition will inhibit it's use for abstract specification and design, and limit it to simple checking logic embedded in System Verilog code.

The adaptation of assertion semantics to deal with sampled values represents a forced intermingling of functional and timing concerns. For a language that purports to be useful for system-level design, this failure to abstract away timing is odd. The more usual approach is to separate timing and functional verification, to enable more efficient verification at more abstract levels of design.

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. This will be the case unless the assertion and the hardware block the assertion represents both look at the same signals at the same time. If the designer wants to make use of System Verilog clock domains to cause the HDL code to sample certain signals before the clock edge, then the designer should write assertions that sample those signals in the same manner, for consistency. If assertions were allowed in clock domains, this consistency would be accomplished trivially.

3.17.3 Clocks

There is a major issue with the definition of clocks. System Verilog concurrent assertions are only defined with respect to a clock edge. Even a combinational invariant concurrent assertion only has meaning at the edges of the relevant clock. 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. The fact that System Verilog concurrent assertions are not defined for the base case of an unclocked system means that invariants such as this do not really express what we think they mean. This is a fundamental flaw.

Note that the formal semantics document does define rewrite rules that express the semantics of clocked concurrent assertions in terms of equivalent unclocked concurrent assertions, but the LRM does not allow users to write unclocked concurrent assertions. Instead, it clearly states that concurrent assertions are evaluated only at clock ticks:

(in Section 17.4)

> "Concurrent assertions describe behavior that spans over time. The evaluation model is based on a clock such that a concurrent assertion is evaluated only at the occurrence of a clock tick. The values of variables used in the evaluation are the sampled values. ..."

(and later in the same section)

> “An expression is always tied to a clock definition. The sampled values are used to evaluate value change expressions

> or boolean sub-expressions that are required to determine a match with respect to a sequence expression. ..."

In general, the mapping from the formal semantics document to the LRM semantics description is left undefined. 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.

3.17.4 Syntax

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 decision to use the same operator (##) with different delay values for overlapping concatenation and non-overlapping concatenation (##0, ##1) means that both operations must necessarily have the same precedence. This leads to non-intuitive semantics resulting from associativity of ## determining which of the two is executed first.

The syntax for property_spec appears to require 'not' in conjunction with a multi_clock_property_expr. This would mean that the following is not legal:

```
property P;
   @(a) |=> @(b);
endproperty;
```

3.17.5 Documentation

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.

Consider the following text (in section 17.5, Sequences):

> "A sequence is a list of SystemVerilog boolean expressions in a linear order of increasing time. These boolean expressions must be true at those specific points in time for the sequence to be true over time. A boolean expression at a point in time is a simple case of a sequence with time length of one unit. To determine a match of a sequence, the boolean expressions are evaluated at each successive sample point to satisfy the sequence. If all expressions are true, then a match of the sequence occurs."

In these two paragraphs, the following terminology is used:

- true (applied to a boolean)

- true (applied to a sequence)
- match (applied to a sequence)
- satisfy (applied to a sequence)

This appears to define three different terms relating to the evaluation of a sequence – true, match, and satisfy – all apparently meaning the same thing. Yet none of the definitions is complete, since none of the definitions consider the meaning of the repetition, counting, and nth_event operators.

Later, in 17.7.3, an additional term is introduced:

> "The two operands of and are sequence expressions. The requirement for the success of the and operation is that both the operand expressions must succeed."

'Success' here seems to mean the same thing as 'matched'. Or does it?

The following paragraph is either unclear or circular:

> “The context in which a sequence occurs determines when the sequence is evaluated. The first element in a sequence is checked at the first occurrence of the clock at or after the element that triggered evaluation of the sequence. Each successive element (if any) in the sequence is checked at the next subsequent occurrence of the clock."

What is "the element that triggered evaluation of" a sequence that is the beginning of a concurrent assertion? For that matter, what is an "element"? This term is undefined.

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.

3.18 Section 18 - Hierarchy

SystemVerilog 3.0 introduced new hierarchy concepts which came in through the introduction of Superlog content. These include $root and nested modules. These are yet again examples of layering rather than integration as explained below.

3.18.1 Does not address program blocks

References: 18

A general comment on this section is that it often refers to both modules and interfaces. Statements such as “All modules and interfaces must be parsed before elaboration” are common throughout the chapter. 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.

3.18.2 $root

References: 18.2

The $root scope creates a single top-level scope in Verilog. We believe this is a disastrous addition to Verilog.

3.18.2.1 Separate compilation

Many simulation vendors and all synthesis vendors are now supporting separate compilation of Verilog source files. Using this technique, only portions of a Verilog design are provided to the compiler at any one time. These pre-compiled units can then be combined later during elaboration into a single hierarchy. The primary capability of Verilog that allows this methodology is that all objects are contained in modules. The addition of $root breaks this paradigm. If objects or statements are declared in the $root scope, then it becomes extremely difficult to allow separate compilation. Examples of the issues are:

- Where in a library system should this content be stored?
- When elaborating a hierarchy, what $root content should be included? All information every precompiled into $root?

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.

3.18.2.2 Global name conflicts/visibility

The addition of $root creates a global declarative region for objects. When a design is assembled, many pieces are brought together from different designers and even companies. As this assembly process takes place if the same name has been used they will conflict and one of the designs will need to be modified. This situation is actually rather common in other languages such as ‘C’ that allow this sort of global scope and should be avoided by design in extending Verilog. 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.

3.18.3 Namespaces

References: 18.9

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 term name space is ill-defined in Verilog. In most languages, there is a difference between a namespace and a declarative region. A namespace is a set of identifiers. For instance, macros are distinct from system tasks because macros begin with the back tick character (`) and system tasks begin with the dollar sign ($). A declarative region is a lexical region where identifiers are declared such as modules, ports, and attributes. Verilog makes no such distinction and this section attempts to address both concepts.

In discussing this chapter in committee, Cadence suggested that instead of creating content that conflicts with the IEEE definition and continues to overload the meaning of the term namespace, we should instead define the namespaces and the declarative regions to clean up the definitions. 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. We will address these issues during the upcoming 1364 revision process.

3.19 Section 19 - Interfaces

References: 19

Interfaces provide a powerful means of passing an entire description hierarchically through a design. In the introduction to this section in the LRM the major benefits are:

- to encapsulate the communication between blocks, allowing a smooth migration from abstract system-level design through successive refinement down to lower-level register-transfer and structural views of the design.
- an interface is a named bundle of nets or variables. The interface is instantiated in a design and can be passed through a port as a single item, and the component nets or variables referenced where needed.
- Additional power of the interface comes from its ability to encapsulate functionality as well as connectivity, making an interface, at its highest level, more like a class template.
- In addition to task/function methods, an interface can also contain processes (i.e. **initial** or **always** blocks)and continuous assignments, which are useful for system-level modeling and test bench testbench applications.

3.19.1 Overlap with modules

References: 19

Cadence believes that the interface defines a modeling style that is enforced by defining a new language construct. This continues the theme of layering on capability rather than integrating it. 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. 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.

The unique benefit of interfaces is the ability to pass them hierarchically through a design. This allows a hierarchical name for the interface to be written that can deterministically refer to the interface even when this block is moved around a design or embedded at a different hierarchical path. This would be very useful for modules in general.

Other pieces of functionality described in this section as unique benefits are really derived from this fundamental concept.

3.19.2 Overlap with classes

Interfaces are described as allowing an abstract interface to a device where the objects in the interface can be viewed as class members and the tasks and functions in the interface can be viewed as class methods.

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 convenent by being able to develop the hierarchical path through a port.

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.

3.19.3 Lack of decomposition

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.

The argument commonly made for restricting interfaces is that they model pure communication, not hardware. There are many places where this modeling style is very useful in system-level models where the communication is abstract, or not realized in hardware. However, the examples given for interfaces all demonstrate modeling communication on a bus. Busses when implemented are hardware, and the behavior of the bus is created by hardware devices connected to the bus such as pads and/or muxes.

If a designer begins to model his interconnect abstractly using a SystemVerilog interface, then he creates the interconnect in a very stylized fashion where the interface itself is passed through the hierarchy. This is an excellent extension to Verilog (and we believe it should be expanded to modules as well). The problem comes when the designer now wants to refine the communication on his bus. This refinement should be accomplished only by changing the interface representing the bus, not by changing every single device which uses the bus. However, since interfaces have been arbitrarily restricted to not contain gates you can never completely refine the bus to hardware without changing every single device connected to the bus. Furthermore, all devices connected to the bus must be simultaneously changed so no incremental refinement is possible; all devices driving the bus must be modeled at the same level of abstraction.

Interfaces provide an excellent mechanism for encapsulating communication and allowing multiple devices to communicate. They also have an ingenious forkjoin task and import/export mechanism which places an obligation on a device using the interface to provide its own slave functionality. All of this works beautifully with a behavioral description. Allowing interfaces to contain any Verilog modeling construct and having a similar import/export mechanism for portions of actual hardware (as opposed to just tasks), would allow them to be refined without remodeling the communicating devices.

Once interfaces are extended in this necessary fashion, then there is so little difference between an interface and a module that the two constructs should just be merged into one by adding modports and the other extensions to all modules.

3.20 Section 20 - Parameters

The SystemVerilog LRM includes an entire section on parameters that mostly just explains the functionality of parameters in Verilog 2001. This section should only contain the extensions and not reiterate the content from 1364.

3.20.1 Parameterized types

References: 20.2

The major extension to parameters is allowing parameterized types to a module. 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. 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. 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.

3.21 Section 21 - Configuration libraries

References: 21

This section is extremely brief and simply specifies that library map information can be specified in $root instead of in a library map file. Since we object to the addition of $root at all, we obviously do not support this addition. If 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.

3.22 Section 22 - System Tasks and System Functions

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.

3.22.1 $asserton, $assertoff, $assertkill

References: 22.6

These functions are used to enable or disable assertion execution during simulation. These functions are extremely dangerous in many cases. Assertions that contain enabling conditions or sequences must maintain state information and match multiple sequences simultaneously. If these sequences are enabled and disabled by the user this state may be tracked inaccurately and result in either false firings, or more likely, missed firings of an assertion. 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.

3.23 Section 23 - VCD Data

References: *23*

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. VCD forms a tool interchange medium that has been extremely important in the interoperability of Verilog tools in the past and it should be kept in sync with the standard.

3.24 Section 24 - Compiler Directives

References: *24*

This section makes minor modifications to a few compiler directives that are in general harmless. We would note that this contains an important extension of the `include mechanism that now allows the standard to define parts of the language through language-defined header files. This mechanism should be used for defining many more of the language extensions for improved backward compatibility.

3.25 Section 25 - Features Under Consideration for Removal

References: *25*

The SystemVerilog Basic Committee was chartered with fixing or clarifying the SystemVerilog 3.0 standard during the creation of SystemVerilog 3.1. The charter for this group was explicitly that no content of 3.0 could be deleted. 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.

3.26 Section 26 - Direct Programming Interface

References: *26, Annex D*

Cadence completely supports the idea of providing a direct foreign language interface which will allow Verilog object values to be directly read and updated in ‘C’ code and ‘C’ functions to be called directly in the Verilog code. The original intent of DPI was to simplify the life of a user who wants to use some ‘C’ functions he wrote, in his Verilog design by allowing directly recognition of ‘C’ function symbols (without complex registration like in VPI). Another motivation was also to provide better performance. Better performance is expected if the ‘C’ code can deal directly with simulation object values. Also some optimizations may be allowed if it is known that the ‘C’ functions used by the Verilog design only may do read and write to their arguments. Finally, SystemVerilog with the addition of new ‘C’ compatible data types should allow the natural definition of a mapping between any SV data types and ‘C’ data types and hence the transparent ability to directly read and update in ‘C’ the value of a Verilog object. This should help in facilitating writing mixed systemC/’C’/C++/systemVerilog designs. However, as demonstrated in the following sections, the technical specification either deviated from the original requirements or does not strictly fulfill them : for example a mix of abstract/direct access is proposed in the Direct Programming Interface, some functionality other than reading and updating object values was introduced, and 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.

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.

3.26.1 Mix of direct and abstract interface

References: D.1, include files Section D.4.1, E.1

DPI provides direct access to simulation objects of ‘C’ compatible types such as SV int type, unpacked struct and array types, etc., but provides abstract access through library functions to packed types, and open arrays. In fact, DPI has not less than 60 interface functions defined to:

- o read and update values of whole vectors, part select of vectors and bit selects.
- o query size, dimensions, left and right bounds of open arrays.

Open arrays (which denote unconstrained array type formal arguments of a DPI function) are accessed by handles and query functions. 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.

3.26.2 Two possible representations for packed (vector) types

References: Ref: Section D.6.3, D.6.4, D.6.7

DPI provides either a canonical representation for packed types or the simulator internal representation. Either can be used by the ‘C’ writer. Better performance is claimed to be achievable if the internal simulator representation is used but the ‘C’ code will not be portable and must be recompiled with each proprietary vendor specific header file.

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.

3.26.3 Source and binary portability

References: Ref: section D.3

As a consequence of the previous section, 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.

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.

3.26.4 Overlap and redundant functionality with VPI and PLI

References: D.8.3, D.8.5

The current DPI specification provides additional functionality in addition to reading and writing of values. This includes saving C user data in a Verilog specific instance, and getting Verilog module instance handles.

Cadence truly supports the idea of a pure direct read/write programming interface but strongly 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.

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 following is an extract from the section in the SystemVerilog LRM which cautions the user about VPI and DPI incompatibility.

> "Programmers must make no assumptions about how DPI and the other interfaces interact. In particular, note that a vpiHandle is not equivalent to an svOpenArrayHandle, and the two must not be interchanged and passed between functions defined in two different interface standards. If a user wants to call VPI or PLI functions from within an imported function, the imported function must be flagged with the context qualifier."

3.26.5 Many library access functions

D.9.1.4, D.10.3.1, D.10.3.2

DPI has library functions to manipulate the values and transform them between the native SystemVerilog representation and the canonical representation. DPI provides library functions to read/update the entire value, a bit select or a partselect of that value. There are 4 variants of the same function with one, two, three and a variable number of arguments depending on the number of dimensions of the array to deal with. For SystemVerilog types which are C compatible (like unpacked arrays and structures), direct access is available.

Cadence believes that a library of functions goes against the original focus of DPI (providing direct access) and against performance as there is inherent overhead in functions calls.

All the memory allocation for the canonical representations to hold the value to read or write must be done by the C code. 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. This would avoid user memory leaks. This value copy may be necessary in order to determine if the value was changed by the C code and to wake up the appropriate fan out. If the object does not have any fanout, a reference to the internal canonical representation can be passed to C.

Furthermore, the library functions proposed are sometimes limited in functionality. For example, DPI access to part selects is limited to reading and writing part selects of less than 32 bits. Instead with a direct access and user manipulation of the defined canonical representation, there would not have been any restriction.

3.26.6 C data type mapping

References: D.6.3, D.6.4

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. Therefore when running a simulation on a 64 bit platform, a SystemVerilog int object would be 32 bit wide while on certain C compiler implementations the int size may be 32 or 64 bits. We believe that the Verilog C-like data type sizes should instead be parameterized and customized to a given C compiler implementation to truly provide equivalent types and thus direct access.

The current DPI interface does not support all SV data types: classes, events, associative arrays, semaphores, and structs/unions of these types.

3.26.7 Open array arguments

References: D.7.6, D.11

DPI allows one to write a DPI C function which takes open (unconstrained) formal array type arguments and provides library functions to query the actual argument low and high bounds of the ranges, the dimensions, the address of the value of entire array or the address of the value of an array element. Formal arguments declared in SystemVerilog as open arrays are passed by a handle (svOpenArrayHandle), and are accessible via DPI library functions. Open array arguments allow one to write in C a general function that may handle SystemVerilog arrays of different sizes at the price of some performance overhead and at the price of a couple of dozen of library functions.

Cadence believes that if one wants to write some general code which is not susceptible to array dimensions and ranges, the VPI interface is already available and has all the array traversal navigation methods available. Furthermore, the number of VPI methods/properties to do the equivalent of the SystemVerilog open arrays functions is much smaller than the one proposed in DPI. Though, VPI for SystemVerilog needs to be extended to support traversal and access to arbitrary arrays, structure and unions. This task has been completely put on the side by the SystemVerilog CC committee because of the time constraints set by an aggressive schedule and an already overloaded charter. The result is that VPI has not been enhanced to support SystemVerilog. This will cause serious problems to users or 3rd party tools which have or want their VPI application to work in a SystemVerilog design.

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.

3.26.8 SystemVerilog context and pure qualifiers

References: 26*.4.1.3, 26.4.2, 26.4.3*

DPI allows SystemVerilog to invoke DPI C imported functions. DPI also allows SystemVerilog functions to be exported and be callable from the C code including from within a DPI function. An import DPI function (which is implemented in C) must be qualified with the **context** word if the DPI function is context sensitive. The function is context sensitive if the DPI function may call an exported SV function or the C function calls VPI or PLI and therefore needs knowledge of the scope where the function is either defined or called. If not qualified with the context keyword, the DPI specification states that calls to VPI and PLI functions may crash and context DPI utility functions will not work.

This particular model requires that an internal variable be set prior to the call to a DPI imported context sensitive function. All DPI exported functions require that the context of their call is known. This is needed because SystemVerilog function declarations always occur in instantiatable scopes, hence allowing a multiplicity of unique function instances. A call to a DPI exported function requires that the scope of definition of the exported function instance be set prior to the call, or it inherits the current default set scope. Therefore the context of the export function call must be determined dynamically by the tool.

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.

DPI also provides functions to associate and retrieve C user data from their context, but there is no provision for these models to be saved and restarted. In fact this issue was brought up but its specification was postponed to 3.2. We believe that this particular functionality is incompletely specified.

The pure qualifier is certainly advisable and may help optimizing code if it was known that the function should only be called when its input changes; note that the Verilog compiler cannot validate that the C extern function is really pure and we would be relying on the writer. The string "DPI" qualifier may also be useful (to the Verilog compiler) to qualify the import or export declaration to be a DPI user function.

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.

3.26.9 DPI object code inclusion

References: Annex F

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.

First, throughout this annex, switch names are provided. Even if these switch names are provided as informative and non normative as pointed out by the note:

> “NOTE—This annex defines a set of switch names to be used for a particular functionality. This is of informative nature; the actual naming of switches is not part of this standard. It might further not be possible to use certain character configurations in all operating systems or shells. Therefore any switch name defined within this document is a recommendation how to name a switch, but not a requirement of the language.”

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.

Secondly, since the DPI function names do not include the shared library where the symbol may be defined, 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.

3.27 Section 27 - SystemVerilog Assertion API

3.27.1 Static information model of assertions

References: 27.3, 27.3.1

The static VPI information model of property/assertion access is very limited. Only iteration on assertions is provided, more detailed access to the contents of the assertion is not available. Property declarations, cover statements, sequence declarations, concurrent assertions, and immediate assertions are all represented by the vpiAssertion type. 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.

3.27.2 Callbacks

References: 27.4.2

A new callback registration function was introduced to place an assertion callback rather than using the existing mechanism vpi_resgister_cb. The user callback function itself has a different prototype than other regular callback functions. The reason given was that the callback function needed to return information other than the assertion handle, the reason of the callback and the user_data; There was not an adequate field in the vpis_cb_data structure to store the vpi_attempt_info (basically information of the matched expressions, failed expression and their source line information).

A new callback function was invented, vpi_register_assertion_cb(), to place an assertion callback; the prototype is:

```
vpiHandle vpi_register_assertion_cb(
        vpiHandle,                /* handle to assertion */
        PLI_INT32 event,          /* event for which callbacks needed */
        PLI_INT32 (*cb_rtn)()      /* callback function */
        PLI_INT32 event,
        vpiHandle assertion,
        p_vpi_attempt_info info,
        PLI_BYTE8 *userData),
        PLI_BYTE8 *user_data  /* user data to be supplied to cb */
);

typedef struct t_vpi_assertion_step_info {
        PLI_INT32 matched_expression_count;
        vpiHandle *matched_exprs;                /* array of expressions */
        p_vpi_source_info *exprs_source_info;    /* array of source info */
        PLI_INT32 stateFrom, stateTo;            /* identify transition */
} s_vpi_assertion_step_info, *p_vpi_assertion_step_info;

typedef struct t_vpi_attempt_info {
        union {
                vpiHandle failExpr;
                p_vpi_assertion_step_info step;
        } detail;
        s_vpi_time attemptTime,
} s_vpi_attempt_info, *p_vpi_attempt_info;
```

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.

3.27.3 Assertion Control

References: 27.5

There has been capability added to control through VPI extensions the assertion system: stop all assertion evaluations, restart the assertion evaluation, etc… We believe that this capability is dangerous in certain cases for the same reasons given in section 3.22.1.

3.28 Section 28 - SystemVerilog Coverage API

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.

The current coverage API reflects one vendor’s interpretation of what coverage information is implicitly recognizable in a simulation run. The inference from a general language structure to this information is not specified at all.

3.28.1 Pragma usage

References: 28.3

FSM coverage uses pragmas to specify the current FSM state vector, the next state and the set of values for a state.

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.

4 Conclusions

In conclusion, Cadence believes that Verilog needs to be extended in order to support hardware design and verification within the Verilog environment. We believe extensions in the areas of data types, constraints and randomization, direct interfaces, and assertions are important to the productivity of the industry. However, as detailed in this document, there are many reasons why Cadence believes that the current SystemVerilog specification should not be forwarded to the Board of Directors for approval as an Accellera standard.