Easier SystemVerilog with UVM: Taming the Beast

John Aynsley
Doulos
Church Hatch, 22 Market Place
Ringwood, United Kingdom
+44 1425 471223
john.aynsley@doulos.com

ABSTRACT

SystemVerilog has been widely adopted as a language for hardware design and verification. At the same time, SystemVerilog is a very large and complex language which can be daunting to learn and use, and differences still remain between implementations. SystemVerilog adoption has been given a new impetus in recent years with the introduction of UVM, the Universal Verification Methodology for SystemVerilog. The UVM codebase has provided a convergence point for SystemVerilog implementations and applications by creating a de facto SystemVerilog subset that all implementations must support. UVM uses a compact set of object-oriented programming features which are very general and expressive, and which are well-supported by the major implementations. When combined with other SystemVerilog features to express constraints, functional coverage, and to abstract the interface between the design-under-test and the class-based verification environment, the resultant set of language features is robust and sufficient for hardware verification.

Keywords

SystemVerilog, Verilog, UVM, functional verification, C, programming language

1. INTRODUCTION

SystemVerilog is now established as a successful language for hardware design and verification, and at the time of writing SystemVerilog has become the language-of-choice for many companies, particular those adopting constrained random verification for the first time. One of the main reasons for SystemVerilog's commercial success has been support from all the major tool vendors: at this time, SystemVerilog has come to dominate other single-vendor solutions. However, the adoption of SystemVerilog has been hampered over the years by it being a very large and complex language specified by a somewhat ambiguous language reference manual. These factors have made it a challenge for simulator vendors to create complete and mutually consistent implementations of the SystemVerilog standard, and differences between implementations still exist to this day. It has proven uneconomic to refine the SystemVerilog standard to the point where we have a set of complete, consistent implementations: tool vendors simply have other commercial priorities.

The designers of SystemVerilog did not appear to subscribe to the principles of simplicity, orthogonality, and consistency amongst language features. SystemVerilog is sometimes described as being the union of several languages. As well as being a superset of Verilog it incorporates features taken directly from Superlog and OpenVera as well as being inspired by features from C, C++, Java, VHDL, and PSL. The result is a set of language features with many complex and unexpected interactions, which is a burden on implementers and users alike. Further evidence of SystemVerilog's size and complexity is the length of its BNF formal syntax definition, which covers 43 pages of the standard language reference manual, and is 70-80% larger than that of VHDL.

The plurality of approaches offered by SystemVerilog makes learning the language a particular challenge. Whereas an industrial training class teaching the main features of Verilog or VHDL would be typically 4 or 5 days in length, a hypothetical class teaching the whole of SystemVerilog would be a lot longer than 5 days. In practice there is little demand for such extended training classes, with project teams preferring to focus on a more prescriptive approach, that is, how to use SystemVerilog to perform particular tasks. This typically means SystemVerilog for RTL design, SystemVerilog Assertions, and SystemVerilog for constrained random verification, also known as SystemVerilog Test Bench. In practice, each of these training classes would select a limited set of SystemVerilog features that represent best practice for the task at hand and that have relatively robust implementations.

UVM, the Universal Verification Methodology for SystemVerilog, represents the latest member of a family of methodologies (and their associated *base class libraries*) for using SystemVerilog for constrained random verification. SystemVerilog methodologies have played a valuable role in capturing best practice and avoiding the need for each user to reinvent the mechanisms needed to use SystemVerilog classes to build verification environments. UVM is the first standard verification methodology to be actively supported and championed by all the major tool vendors, and has added a significant impetus to SystemVerilog adoption. Potential SystemVerilog users now have increased confidence to move forward with the adoption of a new language knowing that they have the support of the major tool vendors and a whole ecosystem of IP and service providers. Having a single standard methodology encourages the development of a market for verification IP re-use, which promises to be highly significant.

UVM has achieved even more than capturing best practice and enabling verification IP reuse. It has also provided a commercial imperative for simulator vendors to complete the work of creating mutually consistent SystemVerilog implementations, at least for the parts of the languages used by UVM. Indeed, this is exactly what has happened.

2. SYSTEMVERILOG CODING GUIDELINES

Most SystemVerilog simulators now support all the major areas of the SystemVerilog language, and areas of inconsistency between the major implementations are by-and-large restricted to corner-case interactions between the language features. A simulator that supports feature A in most contexts and feature B in most contexts may have

compatibility issues where feature A is used in the context of feature B.

Two significant issues remain for the user:

- How to avoid the remaining pitfalls of inconsistent language support across the tools
- How to use a coding style in keeping with best practice across the industry

The number of tool issues is now so small that it would be feasible to address the first point above by building a *black list* of language features to avoid. However, a prescriptive approach is more practical for addressing the second point. There are still some areas of SystemVerilog that are relatively unexplored, particularly the interactions between some of the newer language features, so it is best to keep with known coding idioms. We are all creatures of habit, and in practice we tend to use only a subset of any given language and to do so in a repetitive way. In the case of SystemVerilog, using only known good coding idioms can be the best way to avoid pitfalls.

So, we propose the following approach:

- Start from the Verilog subset of SystemVerilog, which is well-defined and stable due to its legacy
- Add the concise RTL features of SystemVerilog for hardware synthesis, if required (outside the scope of this paper)
- Add the object-oriented programming and C-inspired features of SystemVerilog from the UVM base class library
- Add further features necessary for constrained random verification, in particular constraints, covergroups, and assertions
- Add features for interfacing between the class-based verification environment and the module-based design-under-test, in particular interfaces and clocking blocks
- Create a black list of language features to avoid based on experience in your company and publicly available information

Creating a feature black list can be problematic because, understandably, tool vendors do not advertise the shortcoming of their wares and because any such list will go out-of-date as tools improve. Nevertheless, it is possible to give some general guidance on which language features to avoid for portability. Indeed, certain issues can be inferred from studying the UVM source code itself and other publicly available information.

The UVM base class library and the SystemVerilog codebase in many companies are littered with conditional compilation directives in order to avoid tool compatibility pitfalls. Fortunately such directives tend to become redundant over time. We have seen the number of corner-case issues reducing year-by-year, but some still remain.

3. THE UVM BASE CLASS LIBARY

UVM uses a fairly compact subset of the SystemVerilog language. Unsurprisingly, given that UVM is a *base class library*, UVM makes heavy use of SystemVerilog classes, data types, and procedural statements, but not much else. This could generally be described as the object-oriented subset of SystemVerilog. The UVM base class library looks much like a class library in any other object-oriented programming language, apart from the obvious language differences and the absence of features such as function and operator overloading, multiple inheritance, and template meta-programming.

In the wider software context, the object-oriented programming paradigm has proved itself to be very expressive, very powerful, and very durable. The realization of OOP within SystemVerilog is similarly expressive: using a combination of a relatively small number of language features, class-based SystemVerilog is able to address a wide range of problems with a small volume of code and a small set of coding idioms. In fact, the UVM codebase makes use of most of the OOP features from SystemVerilog, and exercises those features very heavily and in many combinations. This is a sign of a well-designed language and should help to ensure a robust implementation (although a few pitfalls remain).

The UVM base class library uses the following list of language features:

class and **virtual class** within **package**

class parameterized with a type

class ... **extends**

new

task and **function** methods

virtual and **pure virtual** methods

extern methods

local and **protected** members and methods

static members and methods

const and **const static** members

Inline member initialization

this and **super**

input, **output**, **inout**, and **ref** arguments to methods

Default values for method arguments

Class scope resolution **::** to access methods, types, and enums

Handles as members, arguments, and block-scope variables

null, and dynamic cast

Variable declaration and initialization within methods and blocks

static variables within methods and blocks

typedef in package and in class

Forward **typedef**

bit, **byte**, **int**, **integer**, **time**, **logic**

enum, cast to enum

struct in class

event as member

string

Queue, associative array, dynamic array

foreach used with the above

if, **case**, **for**, **while**, **do while**, **break**, **continue**, **return**

@ -> # wait

begin end, fork join, join_any, disable fork

process, process::self, kill

package containing variable, **parameter**, **typedef**, **task**, **function**

$sformatf

The above list amounts to the all of the OOP features of SystemVerilog with very few exceptions (described below), together with the procedural statements and variables from Verilog and their C-inspired enhancements from SystemVerilog. The OOP features

include the class-based data types, that is, the string, queue, associative array, and dynamic array. The C-inspired features include the data types **bit**, **byte**, **int**, **enum**, and **struct**, and control constructs **foreach**, **do while**, **break**, **continue**, and **return**.

There is not much more that needs saying about the above list because by-and-large the features mentioned just work reliably together as described in the LRM. Some notable coding idioms used in the UVM BCL (Base Class Library) are listed below. The significance of the items in the list below is that each of them could very easily *not* have worked. They are non-trivial to implement, and later in this paper we list many examples of similar or lesser complexity that are not implemented consistently. The claim made in this paper is that it is UVM (and VMM and OVM before it) which has provided the commercial impetus for vendors to offer robust and consistent implementation of these parts of the SystemVerilog language.

- Having a type parameter to a class and passing that parameter to a superclass, as in

```
class C #(type T = int) extends BASE #(T).
```

- The use of the class scope resolution operator :: to access static methods, type names, and enum literals within other classes.

```
classname::typename::method();
```

- Inline instantiation of parameterized classes, as in

```
classname #(typename)::method();
```

- Performing a polymorphic dynamic cast on handles, as in

```
if ($cast(to_handle, from_handle))
```

- Testing for the null value of a handle, as in

```
if (handle == null)
```

- Declaring and initializing variables, including handles, at class scope and block scope (within methods), as in

```
begin static string blank = "”; ... end
begin classname q[$]; ... end
```

- Using a void cast to throw away the value returned from a function, as in

```
void’(obj.method());
```

- Enum types, including base types and initializers, as in

```
typedef enum bit { lit1 = 0, lit2 = 1 } name;
```

- Using strings, including the test for an empty string, the **substr** and **len** methods, and string concatenation, as in

```
string S;
if (S == "”)
  S = {"pre”, S.substr(expr1, expr2)};
```

- Associative arrays of arbitrary type, including arrays-of-handles and arrays-of-events, as in

```
-> assoc_array[index].named_event;
```

- Default argument values, ref arguments, handles and queues as arguments to methods, as in

```
function void f (
      ref   uvm_component comps[$],
      input uvm_component comp = null,
      string arg = "");
```

- C-style **for** loops, as in

```
for (int i = 0; i < n; i++)
```

- Use of **foreach** to iterate over arrays, associative arrays and queues, as in

```
foreach (array[i])
```

4. APPLICATIONS THAT USE UVM

UVM applications can use a coding style similar to that of the base class library, but must inevitably bring in other SystemVerilog language features in order to create UVM tests and to interface to the design-under-test, which will be module-based. There are many reliable SystemVerilog features that are absent from the code of the UVM BCL itself.

The SystemVerilog language features given in the following list have proved useful and robust when using in conjunction with UVM (in addition to the features listed above for the UVM BCL itself):

interface
Variable, **task**, **function** within interface
Port on interface
clocking block within interface
Clocking drive, sense, and synchronize
modport
virtual interface
import
Handles in module scope

rand members
randomize with
rand_mode and **constraint_mode**
pre_randomize and **post_randomize**
constraint
covergroup and **coverpoint**
assert and $error

Array manipulation methods
$bits
All of Verilog!

Interfaces, virtual interfaces, and class handles in module scope permit communication between the module-based and classed-based parts of SystemVerilog. This issue is discussed further toward the end of this paper.

Issues surrounding the use of clocking blocks and the SystemVerilog scheduler regions have been well-documented elsewhere [7][8]. Suffice it to say that we would recommend (but not mandate) the use of clocking blocks with **#1step** sampling to help isolate the verification environment from the design-under-test with respect to SystemVerilog scheduler issues, but would not recommend the SystemVerilog **program** in general.

The use of randomization, constraints, covergroups, and assertions is key to constrained random verification, so these features are essential. There are just a small number of tool compatibility pitfalls to avoid (described below).

The array manipulation methods are useful in the context of arrays, associative arrays, and queues, and their implementations now seem to be robust.

5. THINGS TO AVOID

The good news is that there are now great swathes of the SystemVerilog language that are relatively robust and portable. The bad news is that the set of problematic corner cases cannot be reduced to any compact description or simple formula. So rather than writing in general terms about problematic areas of the SystemVerilog language, some of the corner cases are spelled out one-by-one below.

In a sense, this list of corner cases is remarkably short considering the size of the language, and it a tribute to the simulator vendors who have worked to close the gap between their implementations since the introduction of the SystemVerilog standard. In another sense this list is quite alarming, since it highlights areas of the language where simulator vendors have evidently not felt the pressure to converge on a single interpretation of the standard.

The UVM source code identifies certain features as problematic with respect to tool compatibility by the use of conditional compilation directives. Other features are noticeable by their absence from the UVM codebase. Several of the issues raised below are indicative of the existence of complex or little-used language features, and as such the avoidance of these features may be regarded as good practice anyway regardless of tool compatibility issues.

The following list is not meant to be complete or definitive, and will inevitably (and hopefully) go out-of-date very quickly. Nevertheless, it is offered in the hope that it will be of some practical use in the short term.

5.1. Separation of classes from modules

The UVM BCL is exclusively class-based. There are several tool compatibility pitfalls at the intersection of the module-based and class-based parts of SystemVerilog, so this needs to be coded carefully. While the new SystemVerilog data types fit seamlessly into class-based code, they do not always play so well with existing Verilog features.

There are tool compatibility pitfalls around using user-defined types for nets, including enum nets, for example:

```
typedef enum logic [1:0] {e1, e2, e3} et;
wire et w;  // ???
```

Types defined using **typedef** and enums should be confined to variables.

There are tool compatibility pitfalls around using class handles as ports and making continuous assignments to class handles, for example:

```
class C;
  ...
endclass

module top;
  C handle1 = new;
  C handle2;
  assign handle2 = handle1; // ???
```

Class handles and virtual interfaces can be declared in module scope or within a procedure, and methods can be called using those handles. Indeed, these features are necessary in order to instantiate and run a class-based verification environment. However, there are tool compatibility pitfalls around making hierarchical references to handles (including dynamic arrays) declared in other modules or interfaces, and making hierarchical references to virtual interfaces. For example,

```
class C;
  ...
endclass

module top;
  modu inst ();
  initial
    inst.handle = new; // ???
endmodule

module modu;
  C handle;
endmodule
```

So references to handles and to virtual interfaces should not be made through hierarchical names at present.

5.2. $unit

$unit is a mechanism to explicitly access identifiers declared at compilation unit scope, that is, names (such as variables and types) declared at the top level outside of any module [12]. Although such declarations are permitted by the SystemVerilog language, it is preferable to restrict declarations at compilation unit scope to modules, interfaces, and packages, and to move all other declarations into packages, thus avoiding any need to use $unit. UVM does not use $unit.

5.3. Nested modules

The ability to nest modules within modules has not been widely implemented, nor has the ability to nest interfaces within modules, interfaces within interfaces, programs within programs, and so forth. Hence nested modules, programs, and interfaces should be avoided.

5.4. Unpacked struct, union, and array

UVM confines the use of structs to those containing simple types and strings. These are mostly unpacked structs defined within packages and classes, although it is worth noting that simulator support for the **packed struct** and **packed union** is robust in the context of RTL coding. Handles, dynamic arrays, queues, and associative arrays should be avoided as members of a **struct**, although they are robust as members of a **class**. For example,

```
struct {
  byte a[];  // ???
} s1, s2;
```

In general, packed structs are more robust than unpacked structs, and unpacked unions should be avoided altogether. There are tool compatibility issues around the use of unpacked structs and unpacked arrays as nets and as ports of net type (variables and variable ports are okay). There are tool compatibility issues around the use of bit-stream casting between unpacked structs and unpacked arrays, for example:

```
typedef struct    // Unpacked struct
{
  bit a;
  byte b;
} T1;

typedef struct
{
  byte c;
  bit d;
} T2;

module top;
  T1 s1;
  T2 s2;

  initial
  begin
    s1 = '{1, 1};
    s2 = T2'(s1);  // ???
```

There are tool compatibility issues around the use of unpacked structs, unpacked arrays, and queues as arguments to $display and related calls.

With respect to unpacked arrays, there are tool compatibility issues around taking part selects of an unpacked dimension and calling $unpacked_dimensions.

In general, it would seem safest to avoid unpacked structs and unpacked unions altogether and restrict the use of unpacked arrays to a Verilog-like subset, that is, Verilog memories.

5.5. Assignment patterns and %p

Assignment patterns provide a language mechanism to write values of unpacked types, in particular allowing in-line initialization of unpacked struct and array variables, for example:

```
struct { int a, b, c; } s = '{1, 2, 3};  // Okay
```

UVM makes very little use of assignment patterns, although a few trivial examples are starting to creep into the register layer. The implementation of the simpler cases of assignment patters seems to be robust, although there have been tool incompatibility pitfalls associated with assignment patterns, specifically with member tags and the %p formatter, so these features are probably best avoided for the present. In any case, it is simple to manipulate array-like objects without using assignment patterns.

5.6. Type parameter substitution in classes

The UVM BCL makes limited use of type parameters to classes. It would seem that implementers have probed this area of the language to the depth necessary for UVM but no further. The implementation of basic type parameter substitution appears robust, but cases deeper than those appearing in the UVM BCL reveal many inconsistencies between implementations.

One area of inconsistency involves accessing type parameters through typedefs, as in

```
class C #(type T = int);
  typedef T::some_type T2;
  ...
```

Another area of inconsistency involves calculating static constants that depend on type parameter substitution, such as

```
class C #(type T = int);
  static const int c = T::c;
  ...

class D;
  typedef C #(C1) T;
  static const int c = T::c;
  ...
```

5.7. Protected constructor

Protected constructors (**protected new**) are useful to help enforce the singleton pattern whereby the instantiation of a class is restricted to just a single object. The singleton pattern is used in UVM to construct objects for the factory and for the built-in phases. However, UVM contains a conditional compilation directive to exclude protected constructors, so it may be wise to avoid using protected constructors at present as a work-around for tool compatibility pitfalls.

5.8. Array-to-queue assignment

UVM contains a conditional compilation directive to exclude direct assignment from a dynamic array to a queue, so this operation should be avoided for the present. For example,

```
int a[];
int q[$];
a = new[4];
q = a;     // ???
```

5.9. Iterator index query

By-and-large the implementation of the array manipulation methods (that is, array locator methods, ordering methods, and reduction methods) is now very robust and consistent across implementations, and there are a couple of examples in the UVM BCL, namely **find_index** and **sort**. However, the use of the iterator index query method **(index)** within the **with** clause of the array manipulation methods still has some tool compatibility pitfalls, for example:

```
int a[];
int q[$];
a = '{0, 3, 2, 1, 4, 5, 7, 8};
q = a.find with ( item == item.index ); // ???
```

5.10. Block identifiers

The SystemVerilog standard permits block identifiers (labels) before procedural statement, but this feature is not widely supported. For example,

```
blk: begin          // ???
  loop: repeat(8) ; // ???
end: blk
```

5.11. Assignment as a side-effect

Although permitted by the standard, there are tool compatibility issues around the use of the assignment operators within an expression, where the evaluation of the expressions executes an assignment as a side-effect. For example

```
int a, b = 1;
if ((a = b))    // ???
```

5.12. final

The SystemVerilog standard allows multiple **final** procedures which can execute in any order and does not specify the precise circumstances under which final is executed (it merely says "at the end of simulation time"). **final** procedures are intended for the display of statistical information, but aside from the ambiguity in their definition, they are inadequate as a mechanism to execute actions at the end of a test and have not been consistently implemented or widely adopted. It is best to use the UVM phasing mechanism rather than SystemVerilog **final** procedure.

5.13. wait fork

There are tool compatibility pitfalls concerning the interpretation of the **wait fork** statement, which according to the SystemVerilog standard blocks the calling process until its *immediate* child subprocesses have completed. Some implementations interpret the word *immediate* as excluding subprocesses created by nested fork-joins, others do not. For example,

```
begin
  fork
    #44;
    fork
      #125;
      #14;
    join_none
    #2;
  join_none

  wait fork;

  $display($time); // 44 or 125 ?
end
```

Nevertheless, **wait fork** is used in a couple of places within the UVM BCL, and aside from the ambiguity concerning nested fork-joins, appears robust.

5.14. Method Prototype in Modport

While all implementations permit tasks and functions to be exported through a **modport**, there are tool compatibility issues around the use of task/function prototypes within a **modport** declaration. For example,

```
modport mp (import function void f()); // ???
modport mp (import f);                 // Okay
```

5.15. Modport expressions

There are tool compatibility issues around the use of **modport** expressions.

5.16. Functional coverage

SystemVerilog provides language features for functional coverage collection, but the implementation of a coverage database in support of those features is tool-specific. Unsurprisingly, there are some significant differences of interpretation.

The **$get_coverage** system task, which is intended to return the overall coverage of all **covergroup** types, is not consistently implemented.

There are tool compatibility pitfalls around the use of the **get_coverage** method for individual coverpoints.

The interpretation of **option.weight** and **type_option.weight** for **covergroup** and **coverpoint** is inconsistent across tools. **option.at_least** seems to be a more reliable reference point.

5.17. Minor syntax sensitivities

All implementations are occasionally sensitive to minor nuances of the SystemVerilog syntax. For example, depending on context, only some implementations disallow the **std::** prefix before the keyword **process** and only some implementations require the **std::** prefix before the keyword **randomize**, for example:

```
begin
  byte unsigned a, b, c;
  assert( randomize(a, b, c) );      // ???
  assert( std::randomize(a, b, c) ); // Okay
```

Only some implementations permit parentheses to be omitted after calls to the **num** method of the **mailbox**, only some implementations permit parentheses to be omitted after calls to **randomize**, and only some implementations permit empty braces {} in the definition of a **coverpoint**, for example,

```
rand longint data;
covergroup cg;
  coverpoint data {}  // ???
endgroup
```

6. SYSTEMVERILOG AS A BETTER VERILOG

Verilog was used for gate-level simulation, cell library modeling, RTL design, and directed test benches, amongst other things. The topic of SystemVerilog as a language for RTL design is largely outside the scope of this paper on the impact of UVM. However, it is worth remarking that many of the features of SystemVerilog for RTL design have been robustly and consistently implemented, and have been widely adopted by users. These include:

- Consistent ANSI-style syntax for parameters and ports (first introduced with Verilog 1995), extended in SystemVerilog to include task and function arguments.
- Shorthand port connection syntax, that is, **.name** and **.***
- Relaxed rules for using variables in contexts that previously demanded the use of a net type.
- The synthesis-aware procedural constructs, namely **always_comb**, **always_ff**, **always_latch**, **unique**, and **priority**.
- **timeunit** and **timeprecision** for defining time units without the need for compilation directives

These features combine to make SystemVerilog a more elegant and user-friendly language for RTL design when compared to the original Verilog language.

7. COMMON GROUND BETWEEN SYSTEMVERILOG AND C

SystemVerilog has inherited several data types and control constructs from the C language, adapted where necessary to fit with the syntax of SystemVerilog. Even though the minor syntactic differences between some features in C and SystemVerilog may be a cause for annoyance, the commonality is a very positive thing from the point of view of making SystemVerilog easy to learn and natural to use.

The common features introduced into SystemVerilog include:

- The 2-state types such as **int**, **shortint**, **longint**
- **enum**, **struct**, **union**, and **typedef**
- **do-while**, **break**, **continue**, and **return**
- Operators ++, --, and the assignment operators

In general, SystemVerilog users have been able to adopt these new constructs with ease because they appear familiar and do not bring too many surprises or quirky semantics. For example, it is now possible to use well-known C coding idioms natively in SystemVerilog, such as

```
for (int i = 0; i < 4; i++)
```

or even

```
for (int i = 0, j = 8; i < 4; i++, j--)
```

These features are heavily used in the UVM BCL, as identified above. With a few exceptions as described earlier in this paper, namely unpacked **struct** and **union** and assignments as side-effects of expression evaluation, these features have been robustly and consistently implemented.

8. THE INTERFACE BETWEEN SYSTEMVERILOG AND UVM

Interfacing between the class-based UVM verification environment and the module-based design-under-test (DUT) is typically accomplished by having a virtual interface in the UVM verification environment contain a reference to the SystemVerilog **interface** instance. This is the approach recommended by the UVM User Guide. Until recently, this approach was blighted by inconsistent implementation support for access through a **virtual interface** to certain kinds of declaration in the corresponding **interface**. In particular, there were tool compatibility issues around access to named events and handles through virtual interfaces. Fortunately most of these issues now seem to have been resolved.

Several authors have suggested (see the paper [4] by Rich and Bromley) the alternative approach of having an abstract class whose methods are called from the verification environment and then having a concrete instance of that class declared within the scope of an interface or module and hence having access by hierarchical reference to the variables and wires used to connect to the DUT. This approach offers the advantage of further decoupling the verification environment from the DUT environment in the sense that the verification environment does not need to name a specific SystemVerilog **interface**.

This latter *polymorphic interface* approach also helps address fundamental language issues in SystemVerilog concerning access through virtual interfaces to parameterized interfaces: SystemVerilog requires the type of any virtual interface to reflect the parameterization of the interface to which it refers. When using virtual interfaces, this ties the verification environment to a specific parameterization of the SystemVerilog interface, which could be a significant obstacle to code reuse (see the paper [6] by Shashi Bhutada). By using an abstract base class in the verification environment, it is possible to re-use the same verification environment unchanged with several different SystemVerilog interfaces: all that is necessary is that each SystemVerilog **interface** must contain a concrete implementation of the abstract interface class.

On the face of it this *polymorphic interface* approach runs counter to the advice given above regarding the separation of classes from modules. It is arguable that in general it is better to keep modules and classes distinct, as exemplified by the **virtual interface** approach commonly used with UVM, rather than to embed classes within modules or interfaces. However, the SystemVerilog language features necessary to create abstract classes and to embed the concrete implementations of those classes within SystemVerilog interfaces and modules have been robustly and consistently implemented, so both the virtual interface and the polymorphic interface approaches are viable.

9. CONCLUSION

Despite much progress by tool vendors since the first introduction of the SystemVerilog language standard, there remain some significant inconsistencies between simulator implementations. There are areas of the SystemVerilog language that have been implemented comprehensively and consistently by tool vendors, and other areas of the language that have not. It is evident that pressure from the user community is one of the factors that sets the implementation priorities for tool vendors, and this is particularly so for the class-based verification methodologies. UVM seems to have had a very positive influence in driving tool vendors to implement the core set of features needed for class-based verification in a consistent manner.

Aside from the class-based subset of SystemVerilog used by the UVM BCL and features such as constraints and functional coverage that are necessary for use alongside UVM, there are other areas of SystemVerilog that have proved robust and natural to use, and hence have become popular. These include the features that improve Verilog as a language for RTL design and the features inspired by the C programming language. With a few exceptions, many of the

latter are used within the UVM BCL. While outside the scope of this paper, SystemVerilog Assertions are generally robust and widely used too.

What of the areas where inconsistencies remain? If any language area has more than its fair share of inconsistencies, this could be taken to suggest that tool vendors have not come under pressure from users to solidify that area of the language. The many inconsistencies at the intersection of classes with structural Verilog might suggest that users are happy to keep class-based and module-based code separate. The many issues surrounding the use of unpacked types might suggest that the inclusion of such features within a hardware design and verification language was a little misjudged in the first place. However, there are still a few inconsistencies where tool vendors just need to do better.

10. REFERENCES

[1] IEEE Std 1800-2009 "IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language", http://dx.doi.org/10.1109/IEEESTD.2009.5354441

[2] Universal Verification Methodology (UVM) 1.1 Class Reference, May 25 2011

[3] Universal Verification Methodology (UVM) 1.1 User's Guide, May 18, 2011

[4] Rich D., Bromley, J. Abstract BFMs Outshine Virtual Interfaces for SystemVerilog Testbenches. DVCon, 2008.

[5] Baird M, Coverage Driven Verification of an Unmodified DUT within an OVM Testbench. DVCon, 2010.

[6] Shashi Bhutada, Polymorphic Interfaces: An Alternative for SystemVerilog Interfaces, Mentor Graphics Verification Horizons, November 2011

[7] Rich D, Are SystemVerilog Program Blocks Needed, http://www10.edacafe.com/blogs/daverich/2009/06/04/hello-world/

[8] Cummings CE, Salz A, SystemVerilog Event Regions, Race Avoidance & Guidelines, http://www.sunburst-design.com/papers/CummingsSNUG2006Boston_SystemVerilog_Events.pdf

[9] On-line resources from http://www.accellera.org/activities/vip

[10] On-line resources from http://www.uvmworld.org/

[11] On-line resources from http://www.doulos.com/knowhow/sysverilog/uvm/

[12] Rich D, The finer details of $unit versus $root http://blogs.mentor.com/nosimulation/