Of Camels and Committees

Standards Should Enable Innovation, Not Strangle It

Tom Fitzpatrick
Verification Evangelist
Mentor Graphics Corp.
Groton, MA
tom_fitzpatrick@mentor.com

Dave Rich
Verification Architect
Mentor Graphics Corp.
Fremont, CA
dave_rich@mentor.com

***Abstract*— When developed and used wisely, standards such as SystemVerilog and UVM can have a profound impact on both users and tool developers. Users benefit from having a common code base that they can use across multiple tools, and tool developers benefit from (ideally) only having to support one language or library, freeing them up to develop tools and technologies to improve the lives of their users. However, keeping in mind the old maxim that "a camel is a horse designed by committee," the downside of standards development is that all standards eventually reach a point of diminishing returns, where the standard is able to deliver on the needs of the vast majority of the user community but development in committee continues to be driven by a few atypical users who continue to try and include more "pet features" that they believe will make the standard "better." Unfortunately, these features often make the standard bloated, more complex, inefficient, and many times incompatible with previous versions of the standard, taking away from the primary goal of providing a stable platform to the user community.**

This paper takes the position that SystemVerilog and UVM have both reached this stage and that, rather than continuing to add features driven by a small percentage of the user community, a "cooling off" period should be adopted to allow tool developers a stable platform on which to innovate.

NOTE: This paper assumes a working knowledge of SystemVerilog and UVM insofar as we will be discussing specific language and library features and their pros and cons, without providing a tutorial on basic UVM functionality.

Keywords—SystemVerilog, UVM, Functional Verification, Standards, Abstract Stimulus

I. INTRODUCTION

The ideal of standards for EDA was concisely captured by the current president of the IEEE Standards Association, Karen Bartleson[1], whose first commandment for effective standards is to "cooperate on standards, compete on products." The evolution of SystemVerilog and the Universal Verification Methodology (UVM) shows the value of this ideal. Before these standards came along, users were limited either in having to use a hardware description language (HDL) like Verilog or VHDL to create their testbenches, or to use a proprietary hardware verification language (HVL), which locked them into a particular vendor since the HVLs were tool-specific. The existence of HVLs proved that HDLs by themselves were incapable of effectively modeling environments to adequately verify the increasingly complex designs being developed.

There were two essential features of HVLs that drove their adoption. The first was their ability to describe objects that were allocated at run-time allowing greater flexibility in describing testbenches, as opposed to HDLs where all components were instantiated statically at elaboration time[1]. The second was their ability to specify constraints to allow a solver to choose randomly from a predefined solution space. Combining these two capabilities meant that verification environments could be created algorithmically that permitted randomization of the environment's structure as well as the stimulus data that flowed through it. For example, based on the size and structure of the statically-elaborated design under test, a testbench could be created algorithmically to match the DUT layout, as well as to randomly attach additional components to the bus to generate additional traffic.

In the development of SystemVerilog, the decision was made to codify the constraint syntax and semantics, but the committee wisely decided not to standardize on the solver algorithm(s) to be deployed to reach a solution. This led to a great deal of competition between tools based on their solver's ability to achieve maximum coverage of a solution space in the shortest amount of time. Indeed, the authors recall the early 2000's when the Einstein Puzzle[2] was often used as a differentiator in marketing presentations arguing the superiority of one tool or another. Such competition allowed for innovation in this important area of tool development, which has had a profound impact on user productivity.

HVLs were developed, in part, due to the recognition that SoC functional verification is really a software problem, not a hardware problem. Where HDLs were adequate to describe the static hardware in a chip, the inclusion of randomization, objects and other HVL features allowed the languages to describe more dynamic behaviors required to describe the "real world" in which chips are expected to operate. Users quickly realized that these environments could be quite complicated and reinventing new environments from scratch for each project was not a viable strategy. Thus was born the concept of verification *methodologies* to facilitate *reuse* of verification IP.

---

[1] Both Verilog and VHDL support a "generate" statement, but their ability to specify algorithmically the layout of an environment is limited by simplistic constructs.

The first verification methodologies were language-specific and were tool-specific since the early HVLs were themselves tool-specific. Once a user had chosen a language and the corresponding simulator, it was exceedingly difficult to change to another vendor, since it would have meant having to rewrite the testbench code in another language. Even with the advent of SystemVerilog, there were still competing methodologies, some of which relied on proprietary language extensions or underlying PLI code and therefore still locked users into a particular tool and vendor, even as multiple vendors claimed to support the SystemVerilog standard. Even when users did stick with the common subset of SystemVerilog supported by all vendors, their private methodologies prevented exchanging verification IP written by others with an incompatible methodology.

These particular problems were solved by the standardization of the UVM, which is implemented in SystemVerilog and is today supported by all major simulator vendors. There are, however, many lessons that the UVM committee can learn from the SystemVerilog standardization effort. Unfortunately, it is too late to apply some of these lessons to the UVM, even retroactively, but they should give the committee some "food for thought" in considering future UVM development.

II. SystemVerilog as a Cautionary Tale

SystemVerilog has its roots in a language called SUPERLOG, which began development in 1999 by some of the original creators of Verilog. It had the lofty goal of being the next generation HDL/HVL by replacing Verilog with a more modern language. However users were unwilling to start from scratch in order to get the language into production. They had large databases of Verilog design and testbench code and would only accept an evolutionary approach, not a revolutionary approach. So SUPERLOG was re-designed to be 100% backward compatible with Verilog.

Eventually, SUPERLOG was donated to Accellera and became the basis for the first version of SystemVerilog 3.0 in 2002. SystemVerilog started out as a project to consolidate the proliferation of a number of different languages including Verilog, Specman, and Vera into a unified design and verification language with the hope that it would reduce the overall knowledge required when developing a complete verification environment (e.g., the syntax and semantics of defining a function is always the same). Over the last decade, SystemVerilog has become a dumping ground for many great features, as well as a few not so great. What began as a 140 page set of enhancements to Verilog is now a 1400 page tome.

Unfortunately, the people picking up that tome, they do not have the historical perspective behind many of these features, nor would you expect them to. It is extremely difficult to remove any feature from the SystemVerilog LRM unless you can prove that no one has implemented it; or if it has been that no one has been able to use it successfully. A few examples of this problem are always_comb versus always @*, associative arrays with wildcard indexes, and program blocks.

```
always @*
   C = A & B;

always_comb
   C = A & B;
```

Verilog 1364-2001 added the @* construct to automatically generate a sensitivity list that enables execution of the block based on when any signal used in an expression inside the block (in the example, A or B) changes. SUPERLOG, whose development came before the release of the Verilog-2001 standard, created a similar construct to solve the same auto-sensitivity problem, but went a bit further solving a few other problems. What if A and B turn out to be constants? Then there is no change, and out remains un-initialized. The SystemVerilog `always_comb` solved that by guaranteeing execution at least once at time 0. Now there are two constructs in SystemVerilog that do *almost* the exact same thing. Unless you read it here or in some other guideline, you would not know to only use `always_comb`.

```
int aa1[*]; // wildcard index

aa1["a string <128 bits"] = 1;

int aa2[string]; //string index
```

Vera has a much simpler type system than SystemVerilog's parameterized, multi-dimensional aggregate type system. An associative array in Vera could only be declared with a wildcard index type. A wildcard is simply an unsized integral type. SystemVerilog added the ability to declare the array index with almost any user-specified type. SystemVerilog also added a `foreach` looping statement that can iterate over elements of an associative array. However, the foreach loop will not work if the associative array index was declared as a wildcard type. So there is no longer any reason to use the wildcard type in the first place.

The **program** block also came from the *Vera* verification language, which was donated to SystemVerilog. In *Vera*, a **program** was a single procedure that represented the "test". Your test was started at time 0 and when the test terminated, the program terminated the simulation. If you needed multiple test threads, you either had to use the fork statement to start it, or use multiple programs. When the last program terminated, the simulation terminated. The UVM methodology completely replaces the need for any of that test control from a `program` block.

Program blocks also try to mimic the scheduling semantics that a PLI application has interacting with a Verilog simulator and tries to eliminate races conditions between the testbench and the DUT. But another SystemVerilog construct – the `clocking` block – takes care of the races just as well. However, there is nothing in program block semantics to prevent races within your testbench, and to prevent races within a design the verification engineer needs to understand

the same fundamental Verilog scheduling semantics that a designer needs to. Those semantics would be easier to learn without the complication of program block semantics.[3]

III. UVM History

The standardization of the UVM a direct result of users driving the process in Accellera to reach consensus on a single methodology to support the development of "modular, scalable, and reusable generic verification environment[s]."[4] At the time, the two competing SystemVerilog-based methodologies were the Open Verification Methodology (OVM), jointly developed by Mentor Graphics Corp. and Cadence Design Systems, Inc., and the Verification Methodology Manual (VMM), developed by Synopsys, Inc. After developing an interoperability library[2] to allow the OVM and VMM base class libraries to be used together, the committee set its sights on developing the UVM.

There was no debate about using SystemVerilog as the HVL in which to implement the UVM base class library. According to a double-blind [3] study conducted by Wilson Research Group in 2012[6], 65% of respondents were using SystemVerilog for their verification while other proprietary HVLs were used by only about 15% of respondents. Also driven by the end-users on the committee, consensus was quickly reached to use the OVM as the basis for the UVM. However, in attempting to incorporate ideas from several precursor methodologies, the committee unfortunately fell into some of the same problems that have plagued SystemVerilog.

The UVM committee was breaking new ground in actually developing code as a "reference implementation" of the standard, which served as the documentation. Due to a combination of history, accepted practice and legal constraints, it is the *documented API* to the UVM that is officially considered "the standard." The reference implementation is technically just a "proof of concept" and anyone is allowed to develop their own implementation of the UVM library, as long as it conforms to the documented API. This conformance to the API is what allows users to write their own code that makes use of the UVM and compile and run it with a UVM implementation, whether it is run with the approved reference implementation from Accellera or a vendor-supplied implementation which might include some non-standard extensions.

The committee released an "early adopter" version of the UVM 1.0EA in May of 2010, followed by the UVM 1.0 in February of 2011. Since then, the committee has continued development and is, at the time of this writing, working on UVM1.2. During this time, the UVM library has continued to grow in size and complexity, as shown in TABLE I.

[2] Approved in 2009

[3] The respondents did not know which company had commissioned the study, and the data evaluators did not know the affiliation of individual respondents. For a more in-depth analysis of the study parameters, see [5].

TABLE I. Size and Complexity of UVM Releases

	Classes	Files	Lines
UVM1.0-p1	288	125	65534
UVM1.1	311	131	66660
UVM1.1a	324	135	67307
UVM1.1b	317	134	67724
UVM1.1c	317	135	67969
UVM1.1d	316	133	67967
UVM1.2[a]	351	144	75070

[a] Based on the latest merge as of this writing. Numbers may change.

The primary reason for the increase in size between 1.0p1 and 1.1 was the inclusion of "phasing" in 1.1. This was an attempt to provide a set of predefined methods to subdivide the run_phase, which is the only task-based method executed by all components in a UVM testbench (see Fig. 1).

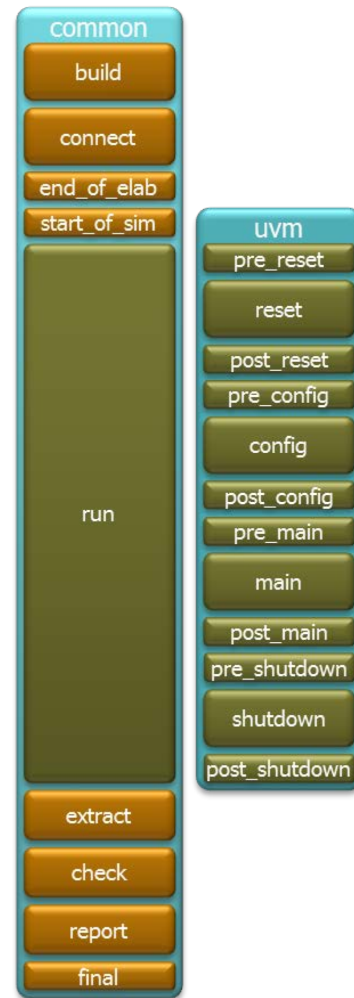

Fig. 1. Phasing in UVM

The original intention of the runtime phasing mechanism was to allow users to arrange a set of tasks that would be called automatically in a predefined order so that users could group similar functionality across multiple components into the same

phase. However, the devil, as always, was in the details. When the committee attempted to document exactly what operations should be done in each of the phases, it was not always possible to agree, leaving open the possibility that components developed by different teams or companies – while each could be in compliance with the standard – might do specific actions in different phases.

A further requirement was to allow users to define their own phases and either create a separate "schedule" of phases to be run in parallel with the original set of phases, or to insert their own phases within the predefined schedule. But the requirement that ultimately doomed the usability of runtime phasing was the ability to jump backward and forward between phases. As with any complex system, there were a number of "moving parts" that proved difficult, if not impossible, to mesh together. Because of the inability to account for sequences and other possibly spawned behavior in a given phase being adequately controlled when a phase was prematurely terminated via a phase-jump, the committee recommended that the run-time phasing methods only be used from a test to start sequences, and that components such as drivers and monitors not implement any of them and just implement the `run_phase()` phase. Jumping, especially backward, between phases is discouraged. Of course, if these methods are only used to execute sequences, then there is really no need to use them instead of virtual sequences, which provide greater control and flexibility in orchestrating the execution of sequences from a test.

It is interesting to note that the inclusion of phasing was considered a "top priority" in the UVM from the beginning, although over 4 years later, we still do not have a fully-accepted solution for run-time phasing, nor does one appear imminent. However, the lack of this particular feature does not appear to affect the usefulness nor adoption of the UVM, as the 2012 study[6] shows that UVM usage has grown 486% since 2010 and is now used by a plurality (41%) of the industry, followed by OVM – which doesn't include run-time subphases either – at 34%, followed by VMM at 20%. Unfortunately, having included run-time phasing in UVM1.1, there is now a portion of the user community who are deploying at least a portion of this functionality, making it extremely difficult to remove it from the standard, even though the implemented functionality doesn't meet the original requirements. This highlights the danger in releasing complex functionality in a standard before it has a chance to be properly vetted by the user community.

The larger jump in complexity proposed for UVM1.2 is due primarily to the proposed inclusion of enhancements to the messaging and transaction recording capabilities of the UVM that, in the authors' opinion, are the epitome of "niche" features being driven by a vocal minority of the committee that will not have a substantial positive benefit for the vast majority of UVM users. As eloquently expressed by Bjarne Stroustrup, the developer of C++, in an October, 2013 interview[7], "Only by providing something almost universally agreed to be genuinely useful can we make progress. A standard committee is no place for single-issue fanatics."

In fact, since these proposed enhancements break backward compatibility, they could actually have a negative impact. It is therefore important to understand when a standard may, or should, be considered "done." We will explore the impact of these and other proposed changes in UVM1.2 later in this paper, but it is our experience with the SystemVerilog standard itself that has taught us to be wary of the impact of such late additions to established standards.

IV. Specific Issues with UVM

A. *UVM Messaging*

Up through UVM 1.1d, UVM messaging has been handled by a set of macros that allow the user to issue a message with an ID associated with it, as well as a particular severity. These messages are delegated to a core component called the uvm_report_handler that can be configured to control whether certain messages are printed to stdout, a log file, or are suppressed altogether. The report handler also allows the user to customize the appearance and layout of the messages, and the macros themselves allow for the inclusion of the file and line number where the message is originally created. Command line options and UVM methods allow the setting of verbosity on a per-component or per-hierarchy basis to suppress the reporting of certain messages. In addition, specific actions may be defined for messages based on the message ID and/or severity. This facility has served UVM users well for years.

One of the proposed enhancements for UVM1.2 is to allow the inclusion of multiple fields in a message, requiring multiple macros be employed for each message:

```
`uvm_info_begin(…)
…
`uvm_info_end(…)
```

It also includes the option of specifying a "context" (i.e., the `uvm_report_object` in which the message is to be printed). Since messages can be of type INFO, WARNING, ERROR and FATAL, this is sixteen new macros, plus another four macros to add fields to the message objects. These "add" macros allow you to add integers, strings, string/string pairs (called a "tag"), and uvm_objects as fields of the message objects.

The proposal adds additional complexity for limited additional functionality. The value of the added functionality is questionable as well. In 1.1, actions could be assigned to an entire message. By adding additional fields to a message in 1.2, it is now possible to apply different actions to different parts of a message, so part of a message might be displayed while another might not be displayed but could raise an error condition. This is another example of complex functionality being released without a) field-testing it with a substantial group of users, or b) fully understanding the usability aspects of the feature.

Our measurements with the proposed code at the time of this writing show a 20% performance penalty when using the proposed extensions to print messages as opposed to 1.1d. When a message is issued via a `` `uvm_info(…) `` macro call,

for example, the UVM uses the VERBOSITY (for INFO messages only) and ID field of the message to check if the report is enabled for that particular setting. Then it processes any callbacks associated with the message via the report_catcher, and finally it processes and outputs the message. The 20% performance penalty occurs without adding additional tags to the messages – it is just additional overhead based on the additional processing required to handle the case where there might be additional fields associated with the messaging.

B. *UVM Transaction Recording*

In the UVM, transaction recording allows information about specific objects to be recorded into a simulation debug database. Since each simulator has its own format, a generic API was chosen for which each simulation vendor could provide an implementation to cause the appropriate recording to occur. In the reference implementation provided by Accellera along with the standard, this recording takes the form of ASCII output to a file.

The proposed changes to the transaction recording API are backward-incompatible with 1.1d and require simulation vendors to provide two mechanisms to support transaction recording, depending on which version the user is deploying. In addition to this added support burden on vendors, our performance measurements indicate a 65% slowdown *relative to 1.1d*, when recording transactions in a straightforward example, discounting the amount of time required for actual file i/o. For the same example, in 1.1d, the transaction recording overhead was 76%, while for 1.2, it was 126%. It is our opinion that no additional functionality is worth that much of a performance degradation.

V. The Changing Verification Landscape

It is not the intention of the authors to dissect every proposed change to the UVM between 1.1d and 1.2. Rather, we believe that the issues raised above indicate that there may be a fundamental issue with the committee's approach, and that "more cowbell"[8] is probably not the right solution. Rather, it is important to realize that SoC verification has changed in the years since UVM standardization began, and certainly it has changed since the precursor languages and methodologies were developed.

The Wilson Research Group study[6] shows that SoC designs have become considerably more complex over the years, evidenced by the following data points:

- The average number of gates of logic and datapath, excluding memories, has risen from 400K gates in 2004 to 11.1M gates in 2012,
- The average number of embedded processors has more than doubled from 1.06 in 2004 to 2.25 in 2012,
- In 2004, 52% of designs had one or more embedded processors. In 2012, the number is 79% (with 29% having 3 or more),

This increase in complexity has led to two new requirements for verification that weren't really accounted for when the UVM was initially developed. Clearly, the advent of embedded processors has led to the realization that software is now a critical piece of the system, and therefore a large part of the functionality to be verified is specified in the software.

If one considers that the ultimate goal of a verification team is to ensure that the chip/system will work when deployed in the field, the ultimate goal of functional verification must be to accurately model the real-world in which the chip/system will operate, and ensure that both the hardware and software that specify the functionality will react correctly, subject to the full range of possibilities described by the verification environment.

We used to have the luxury of considering "hardware-software verification" to simply be making sure that the embedded processor could successfully talk to the hardware blocks to configure them and perhaps initiate data transfers. Such behaviors were relatively easy to mimic in the UVM by replacing the processor model with a UVM verification component that could initiate the desired operations as a bus master through the use of UVM sequences.

Once the blocks were properly configured, the UVM's use of SystemVerilog's constrained random data generation capability would be deployed on the external stimulus to the design to try and reach certain corner cases of functionality based on the interrelations of the SoC blocks connected to those interfaces. Obviously, that approach is much more likely to be successful with a design of 400K gates than one with 11M gates. The problem with constrained-random solvers, as their semantics are defined in SystemVerilog, is that it is possible and often likely that you will repeat random values before you exhaustively cover your desired functionality.

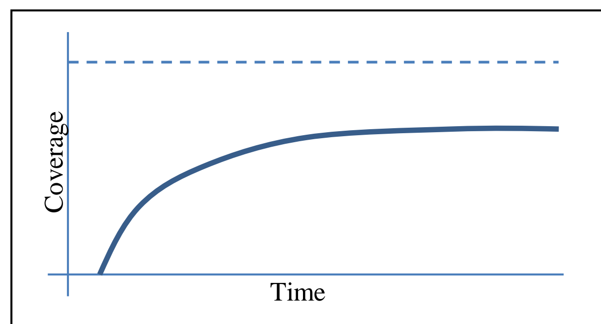

Fig. 2. Typical Functional Coverage Curve

When relying on random interactions of stimuli across multiple interfaces to expose system-level corner cases, such "leveling off" of coverage is to be expected. When this happens, the only recourse is to modify the testcase to alter the constraints and hope for better results. Of course, in large systems, the overhead associated with solving such a complex constraint equation also negatively impacts verification productivity. In keeping with the spirit of innovation shown in SystemVerilog by not standardizing the solver algorithm, the UVM expands the possibilities for innovation so that this particular problem may be addressed.

Another problem with the UVM-sequence-based approach to hardware-software verification is that, while it was somewhat useful in ironing out basic problems on the bus,

there was no easy way to replicate the behavior of the UVM sequences running on the processor stand-in verification component in software running on the actual embedded processor model, or even an ISS model of the processor. Therefore, in migrating from block-level to system-level verification, a discontinuity was invariably introduced that required additional development and resources to ensure that the verification scenarios, when running actual software, were still valid. And of course, with the introduction of multiple processors, the problems became more pronounced since there was more software that must be verified, including interactions between software threads on multiple processors via shared memory, caches and other complexity.

In fact, with complex protocols like ARM's ACE (AXI Coherency Extensions), the bus fabric itself has become a subsystem that must be verified. No longer is the "interconnect" on an SoC simply a set of individual communication paths between components on the bus. Instead, the interconnect now integrates caches, memory controllers and other functionality and is, in many cases, its own network used to connect many components to each other. It is exceedingly difficult if not impossible to be able to write constrained-random sequences to be run on multiple UVM components connected to such a fabric and rely on constrained-random results to generate all of the scenarios required to verify the cache coherence of the system.

VI. Actual UVM Requirements

In just this brief exploration of multiprocessor SoC verification requirements, we can see that the constrained-random capabilities of SystemVerilog, on which much of the UVM is based, are no longer sufficient to meet the productivity requirements of today's SoC designs. Rather, what is needed is a new way of building on top of a UVM environment the ability to specify stimulus and coverage at a higher level of abstraction that can:

- be reused as both UVM sequences and as software,
- better cover the desired state space without repetition,
- allow the composition of multiple state spaces into a system-level space with no loss of effectiveness

When one looks at these requirements, it is reasonable to make the argument that UVM, as currently constituted, provides an ideal platform for users to specify verification *infrastructure* while allowing tool developers the freedom to innovate to attempt to solve these problems through further *abstraction* and *automation*. If one were to propose a solution to these problems, it is easy to see that the obstacles that the proposed extensions in UVM 1.2 are intended to address *no longer need to be solved in the UVM itself*. Rather, the UVM's ability to specify a set of verification components that can be arranged and configured algorithmically (and randomly, if desired), paired with the ability to use UVM sequences that could provide an API for higher-level stimulus specification to drive UVM components, means that it is not necessary to continue to expand the core functionality of the UVM. Features like transaction recording can now be done as part of the higher-level tool on top of UVM, so things like transaction linking and advanced debug are taken out of the realm of the UVM.

Consider a system in which a bus master must be able to drive the bus in all possible modes. In a UVM environment, the master would be represented as an agent in which the driver would get transactions from a sequence and use mode information embedded in the transaction object to control the interface signals to execute the transaction on the bus. Consider a hypothetical bus protocol:

```
class myxfer extends uvm_sequence_item;
  `uvm_object_utils(myxfer)

  rand bit[31:0] addr;
  rand bit[31:0] data;
  rand bit[3:0]  size;
  rand bit       rw;
  rand bit[3:0]  waits;
  rand bit[7:0]  xmit_delay;
  …
  {constraint c_size {
    size inside {1,2,4,8}; }
  …
endclass
```

Assuming 4 possible values for `size`, 2 possible values for `rw`, 16 possible values for `waits` and 100 possible values for `xmit_delay`, this protocol would involve

$$4 \times 2 \times 16 \times 100 = 12{,}800$$

possible variations of transaction types. Typical coverage models also require that each of these transaction types be performed to different areas of memory, which increases the number of unique bus transactions required considerably. To expect to be able to reach all of these coverage points via constrained-random generation is simply unrealistic, and the problem becomes exponentially worse when multiple masters are included. If sequential behaviors are required (burst write followed by 0-wait single-byte write, for example), it becomes extremely difficult even to write the constraint expressions to make that happen, so the engineer is reduced to writing a coverage point (which is no simple task) and hoping that the random transaction stream eventually hits the desired combination.

Instead of separately specifying the constraints and the coverage goals which, in SystemVerilog, use very different syntax, it would be beneficial to raise the level of abstraction to a single specification that could guarantee that the set of transactions generated would cover the desired state space without repeating any scenarios. This abstract specification could thus be used both to generate the transactions and tool automation could record interesting information about them to assist in coverage analysis and debug.

In starting with an abstract specification, the verification team is presented with options on how actually to map the specification into the verification environment. In a UVM environment, stimuli are modeled via sequences, so either sequence code could be generated from the specification, or a

UVM sequence could be built with automation hooks in it to link to another generation engine that would tell the sequence what transaction to generate next. In either case, the user would simply specify via the UVM test to run the automated sequence. The modularity of the UVM allows the rest of the UVM environment, including the agents, scoreboards and other components, including other constrained-random sequences, if desired, to continue to be reused without modification. The UVM is the ideal platform to allow such automation, which is quickly becoming a requirement, to be included in the verification process.

Another advantage of the abstract specification is that it allows for other options beside just interacting with a UVM sequence. As mentioned, the ultimate goal is to ensure that the software running on the processor successfully interacts with the environment. The abstract specification would allow the transaction stream generation to be accomplished as software running on the processor model as well. Other than replacing the bus master UVM agent with the processor model, the rest of the verification environment wouldn't need to change, demonstrating again the value of reuse in the UVM.

However, if we could link the abstract specification for the software to an abstract specification for the external (i.e. "real world") stimuli, automation would then be able to generate scenarios where the software would react to specific behaviors that the final design is required to handle. Using the same analysis, optimization and generation capabilities previously discussed, all of these scenarios could be achieved automatically, guaranteeing that we reach our coverage goals.

VII. CONCLUSION

The standardization of SystemVerilog and the UVM has been of substantial benefit to our industry. By having a single language that combines hardware description, verification, assertions and the DPI, SystemVerilog provides a base on which additional functionality can be layered. The growth and success of the SystemVerilog Assertion (SVA) library is one such result. Nevertheless, one must admit that SystemVerilog, while incredibly useful and successful, is not really the ideal language. If a small group of unbiased dedicated language designers had set out from the beginning to create a single language that addressed the concerns of the original Accellera committee, there is no doubt that, while much of the language might look the same, there are parts of the current standard that would not be there, having been adopted in the committee through political, historical or other reasons.

Similarly, the UVM has undoubtedly been a success, but because of the nature of standards committees, particularly the unique case of the UVM committee attempting to develop software along with the standard, the library includes many overlapping, redundant, unnecessary and, in some cases, contradictory features. Over the course of its development, the requirements for SoC verification have changed, introducing the need for higher levels of abstraction and offering the opportunity for innovative tools to take advantage of the UVM's ability to build a modular verification environment to provide additional capabilities that are not available in SystemVerilog.

Since successful SoC verification will require the use of such innovative tools above and beyond what the UVM offers, or what is proposed in UVM1.2, it makes little sense to continue building in additional functionality that slows down the core of the UVM, thus making it less useful as a platform for innovation. The purpose of standards is to enable innovation, so standards must be bounded to provide an efficient, stable platform so this new generation of tools can focus on providing value and not being compatible with multiple versions of the underlying standard. Unfortunately, continued tinkering with the UVM and SystemVerilog threaten to make each standard less effective rather than more.

REFERENCES

[1] Bartleson, Karen, and Rick Jamison. *The Ten Commandments for Effective Standards: Practical Insights for Creating Technical Standards*. Mountain View, CA: Synopsys, 2010.

[2] "Einstein's Logic Puzzle," http://www.begent.org/einstein.htm

[3] Rich, Dave, "Are Program Blocks Necessary?", http://go.mentor.com/programblocks

[4] "Accellera Systems Initiative UVM Working Group Charter," http://www.accellera.org/apps/org/workgroup/uvm/description.php

[5] Foster, Harry, "Epilogue: The 2012 Wilson Research Group Functional Verification Study," http://blogs.mentor.com/verificationhorizons/blog/2013/11/28/epilogue-the-2012-wilson-research-group-functional-verification-study/

[6] Wilson Research Group, 2012 Functional Verification Study

[7] Stroustrup, Bjarne, and Wong, William, "Interview: Bharne Stroustrup Discusses C++," http://electronicdesign.com/dev-tools/interview-bjarne-stroustrup-discusses-c

[8] "More Cowbell," http://en.wikipedia.org/wiki/More_cowbell