If SystemVerilog is so good, why do we need the UVM? Sharing responsibilities between libraries and the core language

Conference Paper · January 2013

CITATIONS
21

READS
3,962

1 author:

Jonathan Bromley
Verilab Ltd
7 PUBLICATIONS **108** CITATIONS

SEE PROFILE

If SystemVerilog Is So Good, Why Do We Need the UVM?

Sharing Responsibilities between Libraries and the Core Language

Jonathan Bromley
Verilab Ltd
Edinburgh, Scotland
jonathan.bromley@verilab.com

***Abstract*— Probably the most effective catalyst for widespread adoption of advanced SystemVerilog features has been availability of the Universal Verification Methodology (UVM). In addition to a rich base class library, it provides a reference best-practice verification methodology. Fully supported by major tool vendors, and maintained by an industry-recognized body (Accellera), UVM has enjoyed widespread adoption by a diverse user base. For many users, "verification using SystemVerilog" is synonymous with "verification using the UVM".**

After outlining the UVM's value to users, and discussing how it has built on field-proven techniques from other languages and methodologies, this paper goes on to explore the relationship between the UVM and its host language SystemVerilog. In particular, it considers whether the addition of some features to SystemVerilog could have made the UVM smaller, easier to use, or even completely redundant. After examining some selected technical challenges and their solutions, the paper concludes that the needs of users are best served by a combination of an expressive base language and a comprehensive reference methodology.

Keywords—SystemVerilog; Design Verification; Universal Verification Methodology;

I. INTRODUCTION

The SystemVerilog hardware design and verification language was developed in response to user needs for a more modern and flexible version of the popular Verilog HDL™. It was originally promoted by Accellera Systems Initiative, and later appeared as an IEEE standard that has been revised several times, most recently at [1].

SystemVerilog's aim is to be a single language that is sufficiently expressive to model digital systems at various levels of abstraction from untimed functional models all the way through to netlist level. To support the diverse needs of verification and modelling, it also provides general-purpose object-oriented (OO) programming capabilities.

II. SYSTEMVERILOG FOR VERIFICATION

SystemVerilog aims to provide a complete object-oriented programming language that includes domain-specific features to support digital hardware verification. Notable among such features are constrained random generation, temporal assertion, and functional coverage constructs. An overview of SystemVerilog's special features can be found in the companion paper [2]. Detailed tutorial guidance on the verification subset is available in many textbooks such as [3].

Despite the richness of the base SystemVerilog language it became clear very early in its evolution that the language alone was insufficient to enable widespread adoption of the best-practice verification techniques that inspired its development. Although the language provides all that a skilled practitioner needs to implement complex verification environments, numerous cultural and practical challenges emerged that limited its usefulness and encouraged provision of additional libraries, toolkits and methodology guidance. The background to some of these issues is described in this section. Section III describes in general terms how these extensions to SystemVerilog have developed since the language's introduction. Section IV examines in detail a leading example of such extensions.

A. Implementation support for the language's features

The SystemVerilog language is large (newcomers are often surprised to find that its language reference manual runs to over 1300 pages and defines more than 200 reserved words), and it continues to grow and evolve as the language standards committees work on clarifications and enhancements. No single tool vendor was able to provide a truly complete implementation of the language when it was first made available, and each major vendor's implementation provided a somewhat different subset of the language. Consequently there was (and, to a much smaller extent, there remains to this day) a risk that software developed using one vendor's tools would be unusable with a different vendor's implementation. This presented a major hurdle for users who wish to transfer their code base from one vendor's tools to another for any reason, commercial or technical.

B. Dissemination of best practice

Creating a working verification environment in SystemVerilog can be fairly straightforward thanks to the language's powerful domain-centric features. However, best practice requires more than just a working environment. Reuse of code across projects and even among users is highly desirable. It is also important for the industry as a whole, and for the career prospects of individual engineers, that there be a

shared understanding of best practice that engineers can take with them from one workplace to another. The language alone cannot provide that.

C. An open market for reusable verification IP

Typical digital designs connect to their environment largely through well-defined standard protocols including parallel bus schemes such as AMBA®[4], memory interface protocols such as DDR3, and communication interfaces such as Ethernet. To create high quality verification infrastructure (stimulus generation, protocol checking, transaction detection, etc.) for even one such interconnect protocol is a challenging effort. Consequently there is an active marketplace for *verification intellectual property* (VIP) in which third party suppliers (or, occasionally, specialist internal groups within a user organization) develop and maintain reusable blocks of verification software, usually known as *verification components* (VCs). A VC is usually designed to implement the verification functionality required for a given protocol. A typical user obtains licenses for the VCs they require, and assembles those VCs, along with the user's problem-specific code, to create a complete verification environment. For such a marketplace to operate successfully, it is important that VCs from different vendors should be interoperable in a user's environment. This requires not only that a common programming language be used, but – much more important – that VCs conform to a shared set of conventions and have a consistent look-and-feel, regardless of which vendor offers them.

D. A ready-to-use toolkit for every user

Verification, like every other domain of software development, has a recurring set of problems that need to be solved for almost every project. It is obviously desirable that as many of these problems as possible should have off-the-shelf solutions. Typical general-purpose programming languages, if they are to be successful, provide a rich set of library facilities providing users with ready-to-use implementations of common requirements such as math functions, I/O operations, data structure manipulation and so on. By analogy it seems reasonable to expect that a verification-focused library should be available to every verification engineer.

III. Evolution of Industry-Standard Methodologies

Mid-2005 saw a landmark in the SystemVerilog world with the publication of the Verification Methodology Manual [5], a user manual for a comprehensive verification methodology and class library. This publication was based on an earlier reference methodology, RVM [6], that had been developed for use with the dedicated verification language Vera [7]. The Verification Methodology Manual (VMM) mandated an architecture for testbenches using OO SystemVerilog, and provided a comprehensive base class library supporting that architecture. The VMM's base class library was at first a proprietary one, developed by a single vendor and provided in encrypted form to users of that vendor's SystemVerilog simulator, but the source code was later released under a very permissive open-source licence [8]. At roughly the same time, another tool vendor published their own reference methodology for OO testbench architecture [9], which was open-source from the outset.

These methodology specifications, with their associated base class libraries (BCLs), influenced the world of SystemVerilog in a number of important ways.

- They provided a common set of architecture and interconnect standards that allowed users and third parties to develop VCs that could reasonably be expected to be interoperable with VCs implementing other protocols, encouraging the development of a VC marketplace and the reuse of VCs and even larger testbench subsystems from one project to another.
- By mandating a set of architecture conventions and by documenting some important aspects of best practice, they enabled engineers to gain domain-specific skills that were portable from one project to another and even from one workplace to another.
- Widely promoted in the industry, they catalyzed the rapid sharing of expertise and proven techniques among the user base. In particular, they provided a set of shared assumptions that could be used as a starting point for more advanced discussion of interesting techniques.
- Each BCL provided an existence proof that currently available SystemVerilog implementations had the power needed to solve real verification problems. Similarly, they provided a testable set of minimum requirements that had to be met by any implementation that claimed to be of serious practical utility.

The factors outlined above contributed to the widespread adoption of SystemVerilog for digital design verification that began around 2006 and continues today. The rapid maturing of commercially available implementations, and publication of the first IEEE standard for SystemVerilog[10], no doubt also influenced users' choices around that time.

Unfortunately, at that early stage in the maturation of SystemVerilog, a user's choice of verification methodology was strongly coupled to their choice of tool vendor. To achieve vendor independence in effect required a user to create their own methodology and then test it on all major vendors' implementations. Clearly this situation was very unsatisfactory. After a lengthy process (during which yet another methodology was announced and promoted, this time by two tool vendors working together) it was eventually resolved when Accellera formed a technical subcommittee tasked with creating an open standard methodology that could be used with all major vendors' tools. This collaborative effort to develop a methodology and BCL provided tool vendors with an opportunity to work together to ensure interoperability. The result, first available early in 2010, was known as the Universal Verification Methodology (UVM) [11]. It seems likely that the UVM will become a dominant methodology approach for constructing HDL testbenches for some time to come, and the remainder of this paper considers the UVM in detail and pays little attention to other published approaches.

IV. THE UNIVERSAL VERIFICATION METHODOLOGY (UVM)

The UVM follows the pattern already outlined for industry standard verification methodology. It provides a comprehensive BCL supporting the construction and deployment of VCs and testbenches, dramatically reducing users' coding effort and automatically enforcing certain aspects of interoperability. For example, all data objects flowing around a testbench are modelled using classes derived from a single base class `uvm_sequence_item`, allowing generic infrastructure to be written that can store and manipulate any such data object.

Furthermore, the BCL enforces a set of conventions concerning the life-cycle of a testbench (known as *phases*). As each phase is launched by the UVM's infrastructure, a corresponding *phase callback* virtual method is automatically invoked in each and every verification component object. Users do not need to be much concerned with this mechanism, but merely override the phase callbacks in their own derived classes to provide custom functionality so that their verification component objects behave as required.

The remainder of this section outlines some key concepts of the UVM.

A. Single class hierarchy

The UVM's BCL establishes a class hierarchy, rooted in `uvm_object`. This hierarchy makes it possible for the infrastructure to implement key services, such as printing and various resource pools, that are useful to many different derived classes.

B. Components and data

Although there are a few exceptions, classes in the UVM hierarchy largely fall into two distinct categories: *data* and *components*. The data class hierarchy derives from `uvm_sequence_item` and component classes are derived from `uvm_component`. Both these base classes share `uvm_object` as a common ancestor.

Component objects are created (constructed) at simulation time zero, after the HDL instance hierarchy has been elaborated and static variables have been initialized, but before the RTL simulation model begins to consume time. Users and VC developers are expected to override the component base class's `build_phase` method in their derived component class. This phase callback method should construct all the component's children, so that each component takes responsibility for constructing the whole of the component tree below it. Once a component's `build_phase` method has finished, the `build_phase` method of each child component it constructed will in due course be called by the BCL. Component objects are assembled into a tree structure whose root is a singleton object, of class `uvm_root`, constructed automatically by the BCL. Components are intended to model permanent, structural parts of the testbench such as monitors and drivers.

Data objects, by contrast, may be created at any time. They are intended to model stimulus, observed transactions, and other data flowing around the testbench. Unlike the tree of component instances, data objects are not automatically organized into any predetermined structure.

C. Object factory

Using OO techniques to build a testbench yields several major benefits. One that should already be apparent is the ability to derive most or all of the testbench's classes from some common ancestor class, so that the BCL's infrastructure can manipulate user-defined objects in a consistent and generic way. Another key benefit is the user's freedom to create new versions of an existing class simply by deriving from it. A traditional (though not necessarily typical) example of this is creating a derived data class that supports additional verification-oriented features such as error injection.

To make good use of this flexibility, it is necessary to replace instances (objects) of a given class type with objects of some user-specified derived type. Decisions about which derived type to use typically must be deferred until run-time for maximum flexibility, and it is therefore inappropriate for user code to invoke class constructors directly. Instead they should appeal to a *factory object*. This factory accepts requests to construct an object of a given class, but can – at run-time – be configured to construct instead an object of some derived class type. The factory pattern in general OO programming practice is discussed in [12] and other texts.

A factory mechanism of this kind is implemented in UVM and in many other SystemVerilog verification methodology libraries. It is central to the flexibility of the approach, allowing development of a testbench to be largely independent of the development of tests that use the testbench.

The factory can be used to make run-time decisions about which derived class to use for *any* object in a UVM testbench, including components. This makes it possible for a test to install, for example, a specialized version of a protocol driver component at the lowest level of the component hierarchy, without needing to change any other part of the testbench code. It is merely necessary to configure the factory, before the component tree is constructed, so that it performs the appropriate class override.

D. Configuration or resource database

Among the facilities provided by the UVM BCL, one of the most important and useful is the resource database. This structure allows configuration values, of any data type including user-defined types, to be stored in a globally accessible table and later retrieved using a string name key. Because components' locations in the instance tree are also known by a string name, it is natural (though not essential) to store configuration values destined for a certain component using a key that is based on the target component's name in the instance tree. Wildcard (glob-pattern) and regular expression pattern matching are supported on the keys.

This resource database can be used for any purpose the user chooses. However, it has one especially important application: to configure the testbench as it is constructed. This is possible because the resource database, being global, is created before construction of the testbench begins. Start-up code in the user's test class (which is always the root of the instance tree of user-

defined components) can populate the resource database, which can then be interrogated by each component just before building its own children. In this way, each component can take responsibility for building its children in accordance with test-specific configuration data.

E. Interconnection of components

Although the component instance tree provides a powerful mechanism for structuring the testbench, it is not sufficient. Data must be passed from component to component, typically to model the corresponding flow of data through the DUT or to pass stimulus or observed transactions from a source to a sink component. The connection among components is inevitably application-specific and cannot be known to a VC's original author. UVM handles this by means of a SystemVerilog implementation of transaction-level modelling (TLM) as described in, for example, [13]. The indirection provided by TLM allows VCs to be written to pass data through their TLM ports and exports, without regard for the details of other VCs that may be connected to them. Connection of ports to exports is deferred until the BCL's `connect_phase`, which is called in a bottom-up fashion on all components in the tree after every component's `build_phase` method has executed and therefore the component tree is complete. Users and VC developers are expected to write an overridden `connect_phase` method in each component they write. This method should make all necessary port-to-export connections among the component's children or deeper descendants.

F. Sequences for stimulus generation

With the testbench component hierarchy complete, verification effort requires that stimulus be driven into appropriate ports of the DUT. The SystemVerilog language has constrained randomization features intended to support current best-practice random verification methodology. To take full advantage of constrained-random stimulus generation it is important not only that each transaction be appropriately randomized, but that coordinated sequences of activity take place both on individual ports and across multiple ports of the DUT. Furthermore, the randomization provided by VC developers in their data objects may not be entirely appropriate for any specific verification task, and it is likely that users will need the flexibility to add application-specific randomization constraints case-by-case.

The SystemVerilog base language, UVM's sequences mechanism, and the object factory together provide for all these requirements. UVM sequences bear strong similarities to the sequences mechanism of the eRM verification methodology [14]. There is a conventional arrangement of components within a VC, known as an *agent*, that supports the use of sequences. A VC that fully conforms to these and other conventions is known as a *uVC* (UVM Verification Component).

G. Automated code generation using macros

There are many routine coding tasks that must be undertaken when creating a new class derived from one of the UVM base classes. Some of these are laborious and error-prone, and lend themselves to macro automation. The most important of them are outlined in the following subsections.

1) Macros for factory registration

Proper operation of the factory, as described in section IV.C, requires the creation of a so-called *object wrapper* class for each user-defined derived class. The factory constructs a singleton instance of every object wrapper class, and can use these instances to create instances of user classes on demand. In particular, the object wrapper for some user class `C` must contain a method that creates, and returns, an instance of `C` as shown in the following simplified sketch:

```
function uvm_object create_object();
  C c = new;
  return c;
endfunction
```

Note that although most of the code in this method is completely generic, the specific class `C` that is to be created must appear explicitly. This can only be done by writing appropriate code with the class name inserted. It is unreasonable to expect users to implement the entire object wrapper class manually, so the UVM provides some automation by means of macros. Consequently a typical user-provided UVM class declaration might look like this:

```
class user_monitor
    extends uvm_monitor;
  `uvm_component_utils(user_monitor)
  ...
```

The macro invocation `` `uvm_component_utils `` constructs a significant amount of boiler-plate code, including the object wrapper class as outlined above. Passing the class name to a macro also makes it possible for the class's name to appear in literal strings in the macro expansion, which is helpful for automatic creation of printing methods and string-based class type lookup for the factory.

2) Macros to automate the creation of utility methods

Users are expected to implement a number of utility methods for each derived UVM class they create. As an example we will consider the `print` method; similar principles apply to various other utility methods.

To display the contents of an object it is necessary to know the name and data type of each data member of interest in the object's class. This information is of course specific to each new user-coded derived class, and cannot be known in advance by the BCL. Consequently the `print` method must be customized manually for each new class.

Automating this code generation requires that each data member's name and data type be known to a macro that can then construct the required code. The UVM BCL provides so-called *field automation* macros to implement this requirement. The field automation macros have been criticized on the grounds that they can be clumsy to use and can generate inefficient code, and many users choose to avoid them and instead implement the required methods manually.

V. THE RELATIONSHIP BETWEEN UVM AND SYSTEMVERILOG

For many SystemVerilog verification users today, "verification with SystemVerilog" and "verification with the UVM" (or some comparable methodology) are broadly synonymous. Indeed, the UVM is becoming so pervasive in verification practice that tool vendors are already beginning to offer precompiled versions of the UVM's BCL, and built-in UVM-specific debug capability, as integral parts of their SystemVerilog simulation tools. It is clear that to apply SystemVerilog effectively to verification problems, users need (or, at least, value highly) the support provided by the UVM's prescriptive guidelines and its BCL. It seems natural, then, to ask whether the facilities provided by the UVM should perhaps be integrated into the SystemVerilog core language and its IEEE-standardized language reference manual (LRM). SystemVerilog already offers a selection of verification-specific constructs – notably temporal assertions, coverage, and constrained random generation. Why does it not also offer some of the base classes, phasing infrastructure, resource database and other features of the UVM?

There is at least one precedent for this kind of integration of methodology and toolkit into the core of a verification-oriented language. The IEEE standard for the ***e*** programming language [15] includes specification of the eRM verification methodology, which is widely applied by users of that language and is bundled with the language implementation.

A. Does SystemVerilog have what it takes?

Comparing the relationship between ***e*** and eRM with that between SystemVerilog and the UVM leads to some interesting challenges for SystemVerilog. In this section we outline those challenges. Later we go on to discuss whether existing solutions are satisfactory, and to consider other possible solutions.

1) Metaprogramming

Some parts of the eRM define new language features that do not appear in the base ***e*** language, the `sequence` construct being a notable example. The ***e*** language offers metaprogramming facilities (known as macros) that allow user-written source code to define completely new language constructs. In this way it was possible to develop and test eRM as a standalone source code package; indeed, it was deployed in that way for many years. Now, though, those eRM facilities are available, unaltered from a user's point of view, in the core language implementation. SystemVerilog cannot offer this seamless upgrade path. Its macros provide the means for expert users or tool vendors to provide significant language extensions, but those extensions will always have, at their point of use, the characteristic and syntactically somewhat clumsy appearance of a SystemVerilog macro invocation.

2) Reflection and introspection

In section IV.G we described two ways in which the UVM uses SystemVerilog macros to automate the generation of code to meet certain common requirements. In both cases, macros are used to solve a usability problem that could perhaps be better handled using language-native introspection or reflection capabilities, sometimes known as *run-time type identification* (RTTI). Such capabilities provide a running program with access to the compiler's knowledge of user-created data structures, such as the names of classes and the names and types of their data members. If such information were available to a running SystemVerilog program, many of the UVM's factory registration and field automation macros could be replaced with fully automatic library code that was entirely transparent to users.

SystemVerilog has an alternative mechanism for reflection of this kind: the Verilog Procedural Interface (VPI). This is a library of routines that gives access, at run-time, to the elaborated data structures of the running simulation model, and to information about the source code that would otherwise be known only to the compiler. It is mandated by the language standard [1] and therefore should be available in any conforming implementation, and it appears sufficiently rich to provide the required facilities. No VPI-based implementation of these facilities yet exists as far as the author is aware, but it seems likely that such an implementation could in principle be created.

3) Code patching

The ***e*** language is well known for its aspect oriented features, and in particular its ability to add patches (extensions and advice) to any part of a code base from a problem-specific extension module, typically a test case. A useful overview of these interesting features, and tutorial guidance on their use, can be found in [16]. The flexibility this brings has been criticized because of the risk that ill-disciplined extensions applied to a large code base can lead to unmaintainable "spaghetti code", but there is no doubt that in the hands of skilled practitioners it can offer good productivity for the typical challenges encountered when applying an existing verification environment to the often fluid requirements of digital design verification.

SystemVerilog has no direct counterpart for this approach. Instead it relies on OO programming, in which existing base classes are extended to add new functionality. This class extensibility is made useful by the factory mechanism described in section IV.C, allowing user extensions to be designed very late in the testbench development process and then to be deployed in an existing verification environment without any need for the original developers to have made specific provision for it.

B. Could the UVM be smaller if SystemVerilog were richer?

There seems little doubt that the factory automation and utility method macros of the UVM would be unnecessary, or at least very much reduced, if SystemVerilog had better introspection, reflection and metaprogramming capabilities. Other parts of the UVM, however, make good use of SystemVerilog's language features and appear to fit very naturally on top of it, providing domain-specific functionality that would not be appropriate in the core language.

C. Could the UVM and SystemVerilog be merged?

The division of responsibilities between SystemVerilog and the UVM may not be perfect, but it would be hard to argue that the two should be merged into a common mandatory standard. It is especially important to note that the UVM, despite its

considerable complexity, is nothing more than SystemVerilog source code. There is no fundamental obstacle to replacing it with a different methodology and BCL, and indeed many users have chosen to use their own in preference to the UVM.

VI. Dividing Responsibility Between Language and Libraries

Verification engineers face difficult problems daily, and it is not surprising that sometimes they express frustration that available tools do not provide all the support they need. Concerns noted anecdotally by the author in discussion with colleagues, clients and training class students include:

- Why are certain UVM features so clumsy? Surely it would be preferable if SystemVerilog had built-in provision for such features, so that complicated UVM macros or function calls were not needed.
- SystemVerilog is excessively large. It is unreasonably difficult for a working engineer to become familiar with all its features, difficulties and pitfalls. Surely it would be preferable if the language were smaller, and the features added in the form of library support?
- The language offers no feature to support my specific requirement.
- The language's features, being built-in, cannot be extended by user code to meet my specific requirements.

It is clear that some of these concerns are mutually contradictory. Nevertheless, tools designed for a specific problem domain must be careful to address the needs of their user base, and therefore such concerns expressed by users should be taken seriously.

A. *Some features are best provided in the core language*

SystemVerilog is already a large language, with many core features that – at first glance – appear to be better removed and replaced with library functionality. However, many domain-specific requirements cannot comfortably be implemented as library functions.

Constrained randomization is probably the best example of a feature whose usability is sure to be compromised if implemented as a library. Randomization constraints describe relationships among arbitrary groups of variables, sometimes using specialized syntax that is inappropriate in any other context. It would be very difficult to capture this flexibility in the form of library functions. As an example it is interesting to compare the limited expressiveness and somewhat clumsy API of the SystemC Verification Library's randomization features [17] with the native features of SystemVerilog.

In a similar way, the functional coverage and temporal assertion features of SystemVerilog would not be easy to replace with a bolt-on library.

Finally, a large part of the set of SystemVerilog features and keywords deals with gate- and switch-level modelling concerns that were part of the original Verilog HDL and must continue to be supported for backward compatibility.

B. *A language should not provide features that can equally well be provided as a library*

If a programming language is powerful enough to allow for the expression of some functionality as an add-on library, it seems unnecessary to add that functionality to the core language. It could be argued that SystemVerilog has not always followed this guidance. It has some built-in operations and functions that could instead have been provided as library functions without significant loss of usability (for example, some built-in string and array manipulations). Furthermore, it is the author's opinion that there is now an expectation among the user base that *any* useful feature should be provided in the language itself, and this seems to have held back or discouraged the creation of comprehensive libraries for such purposes as I/O and string manipulation.

Despite these concerns, the author believes that the balance between SystemVerilog and the UVM is broadly right. Key methodology concepts such as phasing and the major base classes should not be built in to the language, but instead should be provided in a library in exactly the way the UVM has done. In this way, the UVM (and other similar toolkits) will be able to grow and take advantage of improvements in methodology, without the disruption that would ensue if those changes were introduced to the core language. Vendors' implementations of SystemVerilog can remain stable across such future changes, because the changes can be made in pure SystemVerilog source code that can easily be distributed to users.

VII. Conclusions

At first glance it can seem that the UVM is making amends for inadequacies in the SystemVerilog base language. Closer consideration shows that, overwhelmingly, this is not the case. SystemVerilog is sufficiently expressive to allow the entire UVM BCL to be expressed in pure SystemVerilog source code. In this light we can see SystemVerilog and the UVM as heirs to the long tradition of programming language and associated standard libraries that are perceived by a majority of users to be a seamless whole. Just as many working C programmers informally think of `malloc()` and `printf()` as part of the language itself, to be learnt and used alongside `for` and `switch`, there are working verification engineers who have no reason to differentiate between features in the core of SystemVerilog and those that are provided by the UVM's BCL. The entire set of facilities is available in the tools they use, and they are free to choose the most appropriate language construct or library feature in response to each problem encountered.

The design of a programming language and its libraries will inevitably involve some compromise. The sometimes conflicting requirements of backward compatibility, ease of use, expressiveness, completeness and simplicity must all be considered. After extensive experience not only with SystemVerilog and the UVM but also competing solutions, the author has reached the following conclusions:

- It would be inappropriate and unhelpful to integrate UVM functionality into the core language (this in no

way diminishes the value of integrated UVM support in SystemVerilog simulation tools, however).

- The fact that the UVM BCL is coded entirely in SystemVerilog source code suggests strongly that the language is sufficiently expressive to meet likely user needs.
- Although there are a few SystemVerilog language features that could instead have been provided as library functions, the balance between library and core is broadly satisfactory.
- SystemVerilog lacks built-in reflection and metaprogramming facilities, and has only limited macro capabilities. There is little doubt that some parts of the UVM, and indeed some user code too, could be more elegantly implemented using such facilities. However, the lack of such facilities has not been a major obstacle to the use of SystemVerilog in practical situations.

Other users may, of course, take a different position. Nevertheless, it is clear that the combination of SystemVerilog and the UVM provides users with a powerful toolkit that can be applied effectively to a wide range of problems in the domain of functional verification of digital hardware designs.

ACKNOWLEDGMENTS

The author wishes to thank the following for their contributions:

- Kaiming Ho of Fraunhofer Institute for stimulating the writing of this paper;
- the paper's FDL reviewers for their constructive suggestions;
- colleagues at Verilab for their consistently exciting, challenging and expert discussion of all matters verification (and many other matters too);
- the IEEE SystemVerilog and Accellera VIP-TSC committees for their sustained work in bringing the language and methodology to their current state.

REFERENCES

[1] IEEE, Standard 1800-2012 for SystemVerilog Hardware Design and Verification Language, New York, NJ: IEEE, 2012.

[2] P. Flake, "Why SystemVerilog?," in *FDL*, Paris, 2013.

[3] C. Spear and G. Tumbush, SystemVerilog for Verification, 3rd ed., New York: Springer, 2012.

[4] ARM Ltd, "AMBA AXI and ACE Protocol Specification issue E," 22 February 2013. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0022e/index.html.

[5] J. Bergeron, E. Cerny, A. Hunter and A. Nightingale, Verification Methodology Manual for SystemVerilog, Springer, 2006.

[6] Synopsys, Inc, Reference Verification Methodology, 2002.

[7] J. Michelson, F. Haque and K. Khan, The Art of Verification with VERA, Verification Central, 2001.

[8] Apache Software Foundation, "Apache License, version 2.0," 2004. [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0.

[9] Mentor Graphics, Inc, Advanced Verification Methodology, 2005.

[10] IEEE, IEEE Std.1800-2005: IEEE Standard for SystemVerilog, New York: IEEE, 2005.

[11] Accellera Systems Initiative, "UVM (Universal Verification Methodology)," [Online]. Available: http://www.accellera.org/downloads/standards/uvm.

[12] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: elements of reusable object-oriented software, Addison-Wesley, 1994.

[13] F. Ghenassia, Transaction Level Modeling with SystemC, Springer, 2005.

[14] Cadence Design Systems, Inc., e Reuse Methodology, 2004-2013.

[15] IEEE, IEEE-1647: Standard for the Functional Verification Language 'e', New York: IEEE, 2008.

[16] D. Robinson, Aspect-oriented Programming with the e Verification Language, Burlington, MA: Morgan Kaufmann, 2007.

[17] Accellera Systems Initiative, "SystemC Verification Working Group," [Online]. Available: http://www.accellera.org/activities/committees/systemc-verification/.