# Vlang
# A System Level Verification Perspective

Puneet Goel,
Coverify Systems Technology
Gurgaon, India
Email: puneet@coverify.com

***Abstract*—Memory Wall and Power Wall are redefining the hardware and software design paradigms. The impact on Hardware Verification is two fold. System Level Verification and Coverification of Hardware and Software are gradually becoming mainstream. On the other hand changing profile of servers that run simulations makes it imperative that we take a re-look at the contemporary simulation and verification tools that were essentially designed with single server core and with RTL in mind. In this paper we take a look at how opensource verification language Vlang intends to fill the void.**

## I. Introduction

With the ever increasing verification gap (Fig 1), the focus of functional verification is gradually shifting to System Level Verification. Most System-on-Chip (SoC) designs have one or more processor cores integrated on the device. Often, when the chip comes back from the foundry, more effort is required in bootstrapping the integrated processor cores and loading the software drivers than in analyzing the hardware waveforms. Obviously, pre-silicon coverification of software along with a model of the hardware would infuse more confidence in the team responsible for verifying the SoC design as well as the team developing and testing the systems software.

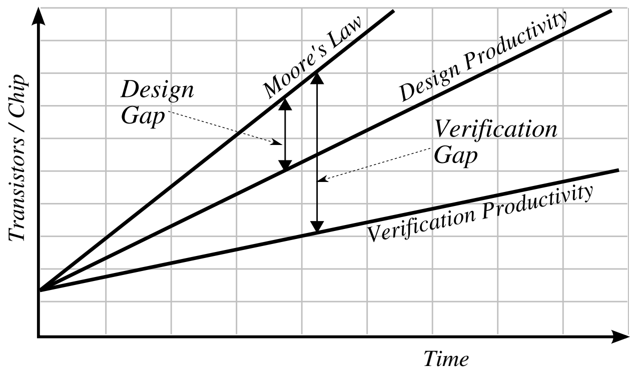

Fig. 1. Verification Gap (adapted from [1])

There are other factors as well that need our attention. After over four decades of relentless run, the Moore's Law (Fig 2) has hit bottlenecks, that are widely known as *Power Wall* and the *Memory Wall* [2]. As the clock frequency of the processors quadrupled every three years, the power consumed by the chips increased at an alarming rate. Over the last one decade, practical constraints imposed by heat dissipation of the computing engines has put a hard limit on the frequency these processors can operate at.

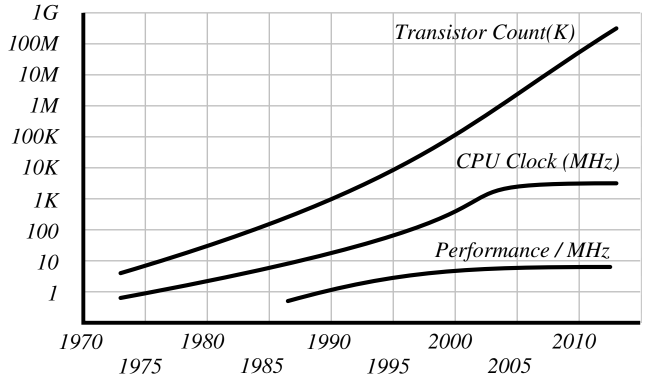

Fig. 2. Server CPU Trends (adapted from [3])

The *Memory Wall* has its implications too. Over the years, the memory systems failed to keep up with the rise of clock frequency of processor cores. During the last decade, as processor companies pack more and more cores on the same processor, the problem has compounded as these processor cores get to share the memory system's bandwidth [4].

### A. The Path Ahead

A generational shift is underway in the way software and hardware are designed. Constraints on the CPU frequency has led the designers to package multiple processor cores on the same device. Software design paradigms are also undergoing a transformation. Researchers are looking at ways to limit access to memory by reducing the number of bits required to perform a computation, even if it requires slightly more compute cycles [5].

In the past, frequency scaling of newer generation processors helped software to handle increasing need for high performance computing.

Data Processing on FPGA [2] is another trend to help compensate for the performance deacceleration imposed by *Power Wall*. In future, application software would be able to take advantage of application specific computation accelerators in form of highly configurable FPGAs packaged along with the processors [6].

### *B. The Future of Design Verification*

The profile of the systems being designed is changing. Fig 3 illustrates how Ethernet bandwidth is increasing over time. In the past network communication stack were mostly handled by software running on network processors, with the role of hardware remaining limited to Layer 2 and below. With network bandwidths exceeding the compute capacity of the processors by an order of magnitude, today, hardware accellerators are often deployed to handle network protocols even at Layer 3 and 4. As the hardware penetrates domains that were earlier exclusively handled by software, the stimulus generation for hardware verification becomes correspondingly more complex. The intricate interaction between hardware and software also needs to be tested, making hardware software coverification absolutely essential.

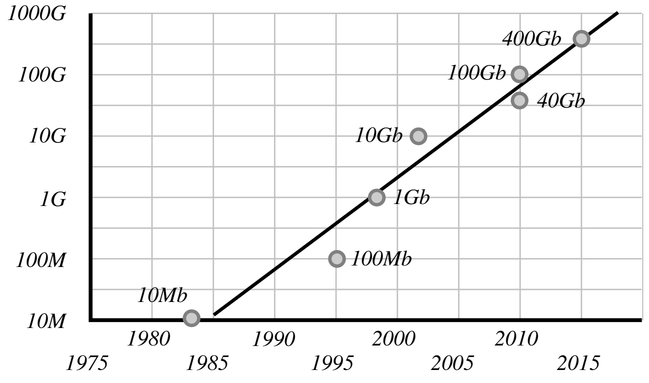

Fig. 3. Ethernet Timeline and Growth of network bandwidth (Diagram Source [7])

There is bound to be a significant impact on the Hardware Verification tools as well. Over the past decade, SystemVerilog Hardware Verification Language (HVL) [8] has become the de-facto standard for functional verification of chips. A majority of FPGA and chip designers use Verilog language to code RTL and a SystemVerilog based testbench integrates well into the simulator without any runtime overhead. On the other hand, any foreign language or tool would incur significant runtime penalty due to inefficacy of integrating the tool via Verilog Programming Language Interface (PLI). Ironically, the inventors of Verilog had functional verification in mind when they first introduced PLI.

Until a decade back, the increase in complexity of verification due to rising number of transistors was partly offset by faster server machines running simulations. But with the advent of multicore servers, the verification tools have failed to keep up with the shift in compute server technology. The loss is not obvious when pertaining to the RTL verification domain. But as we shall see in the next section, it becomes a severe impediment in the System Level Verification arena.

## II. Multicore Testbenches

Multicore parallelism is exceptionally useful when verifying system level designs. At RTL level, more often the discrete event simulation of the RTL design itself consumes more server resources compared to the resources utilized by the testbench. An RTL simulation runs very slow (often processing just about 1-10 transactions in a second). Making the testbench faster would not gain much for RTL simulations. In contrast, a system level simulation, whether it runs on a virtual platform or on an emulation platform, typically processes over a hundred times transactions when compared to RTL simulations. Using multiple threads to generate transactions parallelly therefor makes much more sense when running system level simulations.

Fig 4 illustrates the impact of testbench execution time on a simulation that runs on a simulator that uses a single thread. Since only a single thread is available to the simulator, when the *Design Under Test* (DUT) seeks a transaction from the testbench, the testbench takes a significant period of execution time to prepare the transaction and make it available to the DUT.

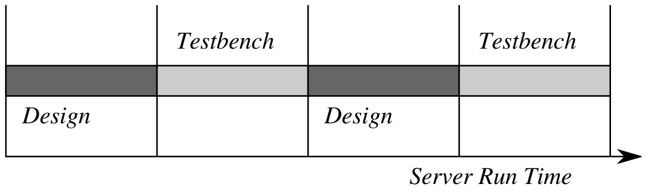

Fig. 4. A non-multicore testbench driven design simulation running on a single thread

Fig 5 illustrates the same testbench running on a simulator that has multicore testbenching capabilities. As the design simulates for a transaction cycle, the testbench simultaneously kicks in. When the design seeks a transaction on the testbench, it simply has to pull it from the testbench and moves on. Parallelly, on a set of concurrently running threads, the testbench too moves on to build the next set of transactions for the next cycle.

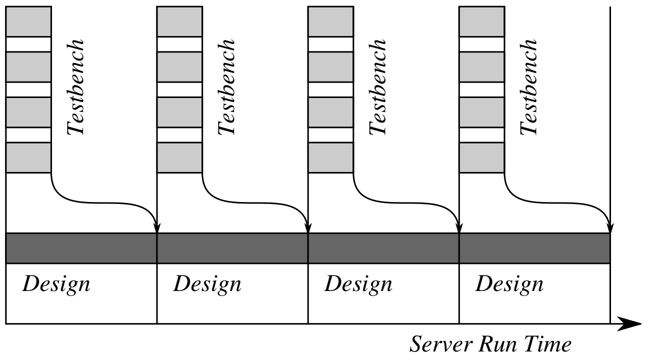

Fig. 5. A multicore enabled testbench driven design simulation running on multiple threads

A common abuse of multicore parallelism is apparent in running multiple simulations together on a processor as is widely done in the verification regression farms. This practice is a direct affront to the *Memory Wall*. Each of the running regression job has its own memory footprint. All the cores of the processors share the same access to the main memory of the server and in many cases, the cores share the Level 3 cache as well. When multiple single threaded processes are executed on the same processor, together the jobs result in that much bigger memory footprint, and that much more constraint on the external memory access and level 3 cache. In contrast, a multithreaded concurrent simulation job would harness the power of multiple processor cores and since all the threads operate on the shared memory map, it does not translate to any extra load on the memory subsystem of the server.

## A. Multicore Testbench Enablers

At the core of a Vlang Multicore testbench is a discrete event simulator capable of running multiple simulation tasks in parallel. There are a number of multicore programming strategies. Of these, Vlang chose the *shared memory* approach for its wide acceptability especially in the systems programming domain.

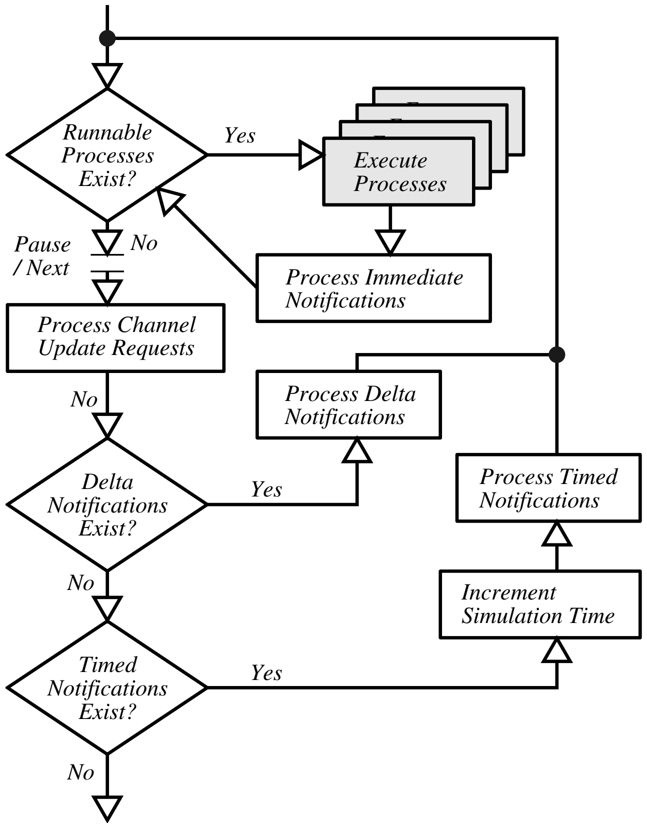
Runnable Processes Exist?
Yes
Execute Processes
Pause / Next
No
Process Immediate Notifications
Process Channel Update Requests
No
Process Delta Notifications
Delta Notifications Exist?
Yes
Process Timed Notifications
No
Increment Simulation Time
Timed Notifications Exist?
Yes
No

Fig. 6. Vlang's multicore simulator executes multiple tasks parallelly on multiple threads

Normally, a discrete event simulator would list the tasks that are triggered by a particular event or on a given time. The tasks are then activated one by one using cooperative threading. Cooperative threading in a normal simulator would execute all the tasks on one CPU thread. A multicore enabled simulator, would run the tasks to be executed, on a number of CPU threads. When multiple threads are used in such a way, a number of issues related to parallel programming crop up and need to be addressed. A good multicore programming approach is to address these issues at the library level and provide a normal interface to the end user.

Vlang's approach is to group all the tasks that might share data into a separate group and run this group entirely on the same thread. As illustrated in Fig 7, in a UVM based verification environment there is very little shared data between various `uvm_agents` (a `uvm_component` that generally corresponds to the abstraction of a Verification IP). Vlang automatically forms a group of tasks (UVM component level `run_phases`) that belong to a given UVM agent. A user defined attribute is available to the end-user to customize the task grouping, when required.

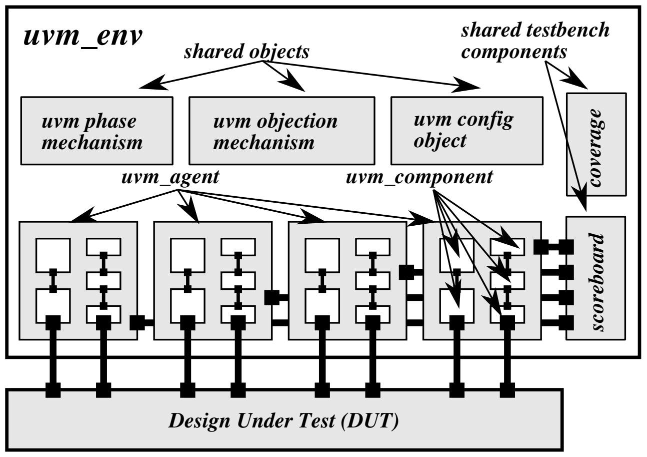

Fig. 7. Vlang's multicore simulator executes multiple tasks parallelly on multiple threads

When multiple threads are run concurrently, care has to be taken to maintain *random stability*. If a common *random generator* is used by all the threads, it would lead to a race condition where concurrently running threads could invoke the random generator in an undetermined order. This would lead to generation of different random stimulus in a repeat simulation run. Vlang takes care of this issue by seeding each thread with its own random generator. The seeding is done at the time of creation of threads. The threads are created by the scheduler, which executes on a single thread, and therefor in a determinitic fashion, thus maintaining *random stability*.

## III. Hardware Software Coverification

As discussed in Section I, an embedded software engineer is often the first level user of an SoC. It is also a fact that due to *Power Wall*, processor frequency is no longer increasing with passage of time as it used to a decade back. As a result hardware is being used more often in systems to accelerate processing of transactions. The integration of hardware and software in a system has therefor become more intricate than ever before. It has become difficult to test software and verify hardware in isolation.

In this section we take a look at essential enablers of coverification.

### A. C ABI Compatibility

Inspite of *system* prefix, SystemVerilog has failed to provide any features that enable hardware software coverification. In Section I-B, we saw how SystemVerilog benefitted from provisioning the same compiler for RTL design and for verification. In contrast, stand-alone verification languages have to depend on PLI layer to communicate with the simulator.

When it comes to System Level Verification, things turn upside down. System level design platforms often involve virtual platforms and emulation platforms, which almost always provide API in C language. A language like SystemVerilog would require either PLI or DPI interface to integrate with system level platforms, thus incurring avoidable runtime overhead.

UVM connect [9] is a popular package created for the purpose of integrating SystemVerilog based verification with system level designs coded in SystemC. Under the hood, when a transaction generated in SystemVerilog is passed on to SystemC, UVM connect packs the transaction into a byte array and passes the array to SystemC (essentially C++). Any transaction based response from the SystemC world also goes through a similar transformation. On top of packing and unpacking, some runtime overhead due to DPI interface is also incurred.

Bitfields and packed structures are often used by C/C++ programmers to map array buffers directly into various fields of a transaction. Packed structures and arrays are also possible in SystemVerilog, but lack of low level pointer operations and casting capabilities in SystemVerilog make such packed constructs inefficient, thus forfeiting the purpose of their existence.

Vlang is build on top of D language, which maintains a complete ABI compatibility with C programming language. D programming language also maintains almost complete back compatibility with C. Any function written in C can be directly called in the D language code without incurring runtime or memory overhead.

### B. Native Compilation

Vlang (and the underlying C Language) compiles to native assembly code and therefor it can be accessed from any environment that runs natively on the server. Native compilation is a useful property for maximizing the runtime efficiency of the testbench. D programming language also allows you to integrate assembly language code into your application to fix any run-time bottlenecks.

### C. Systems Programming

Wikipedia defines a systems programming language as…

> A system programming language usually refers to a programming language used for system programming; such languages are designed for writing system software, which usually requires different development approaches when compared with application software.

It is imperative that a *systems programming language* would be used for developing software at system level. Any language that supports hardware software coverification has to intrinsically integrate with the software. It should also be possible that certain chunks of code otherwise belonging to systems software are required to be coded in the verification language.

## IV. A Coverification Usecase

In this section we take a look at a quick use-case of coverification.

One of the oft-used platform for embedded software development is QEMU. The tool lets software developer compile and run software for the target processor in a caged emulation setup at a reasonably fast speed.

While some hardware modules are still under development, it is often desired that the hardware simulation is tested along with the software that is being concurrently developed. In many cases the software's interaction with the hardware module is limited to the Hardware Abstraction Layer (HAL). A Coverification setup in such cases boosts the confidence of the software team responsible for developing the software driver code. In some other cases, the software is part of the data plane and could be sending/receiving transactions to a hardware module. In such cases, both hardware and software designers tend to gain confidence by successfully simulating and verifying hardware and software together.

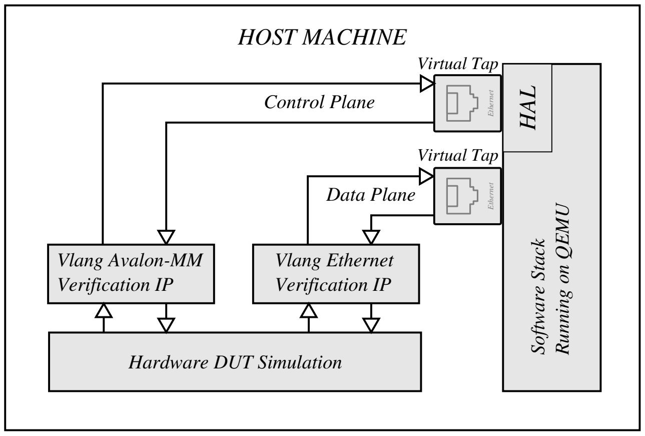

Fig. 8. Coverification setup integrating QEMU based SDK with Vlang verification platform

For our coverification setup (Fig 8), we assume that the software is running in a QEMU based caged environment on a host machine that also runs simulation for one or more hardware modules. The system software running on the QEMU based virtual machine generates transactions and needs to communicate these transactions to the hardware simulation. The hardware model being simulated, is required to send a response after processing the transaction.

One of the fastest channels of communication between QEMU and the host machine can be setup using a *Virtual Ethernet Tap*. Unlike a wired Ethernet port, a virtual tap can be configured to enable read/write access to a particular linux user or group. Thus, except at the time of creation of a virtual tap, no superuser permissions are required to access the tap port from a simulation environment. It is trivial to create a set of tap ports as part of an init script on the servers commissioned for running simulations.

The Vlang Verification IP for Ethernet comes packaged with ability to read/write packed Ethernet frames from a virtual tap. The verification IP forks out a linux posix thread for listening on the port. When it sniffs a packet, it pushes the packet into a UVM fifo specifically designed to interface with Vlang simulator on one end and asynchronous software on the other. Note that the posix thread spawned by Vlang for sniffing on the virtual tap, is quite unlike normal Vlang tasks. A normal Vlang task uses cooperative threading and is controlled by Vlang's scheduler. The posix thread is like any other independent posix thread controlled by the Linux Kernel scheduler.

Vlang simulator's interface with regular asynchronous software is defined by a pair of Fifo architectures specifically sculpted for this purpose. The Fifo designed for receiving transactions from the software has been nomenclated as `uvm_tlm_fifo_ingress`. Its counterpart for transferring data to the software world has been named as `uvm_tlm_fifo_egress`. A normal tlm fifo in UVM uses simulator events to block read/write methods. These Vlang Fifo's created specifically for interfacing with software use events on UVM end and software semaphores to block on the software end.

The rest of the Vlang Ethernet VIP works like any other Ethernet VIP architected in UVM. A virtual sequencer is used to route the transactions sniffed on the virtual tap to the driver. When configured as a stand-alone tool, the VIP is capable of generating constrained randomized Ethernet packets that can be used to cover the corner cases.

## V. Why D? (Or Why Not C++)?

As mentioned above, SystemVerilog fails to meet the challenging requirements of System Level verification. In author's view, any new functional verification language, that must support system level testing, should fulfil the following criteria:

1) Must be a modern System Programming Language. This is an obvious criterion given the need of suitable solutions in the hardware/software co-verification space.
2) Must be Open Source, and available under a license that allows commercial use.
3) Must allow language extension at library level.
4) Must provide ABI compatibility with C/C++. There is a very vast amount of C/C++ code in the wild and SystemVerilog DPI experience [10] shows that data conversion while passing function parameters becomes a runtime bottleneck.

Fortunately, the very first of the above mentioned criteria, brings down the choice to only four contemporary programming languages [11]: C++, D [12], Go [13] and Rust [14]. Of these, Rust is in nascent state of development and Go does not meet criteria 3 and 4. Thus our options get limited to only C++ and D.

While C++ has the obvious advantage of a large user base, it is difficult to build a DSL on top of it without having to overly depend on the C pre-processor. In Comparison, D, as we shall see in Section V-A, provides a multitude of features that make extending the language a lot convenient. Additionally, in that section we shall also see that D also provides features that make it more suitable for hardware modelling and verification.

### A. Motivation for Selecting D as Base Language

The D Programming Language is an evolution of C++. D has multiple features that make the language more suitable for building a Design Specific Language on top of it. These include:

*1) Reflections:* D allows a programmer to introspect the structure of code and make changes to its runtime behaviour. Vlang uses reflections to generate UVM util functions and to give out compile time error, when the end-user fails to provide necessary attributes. Reflections in D are compile-time and therefore do not have any undesired effect on runtime performance or memory footprint of the application.

*2) User Defined Property (UDP):* A D user can add UDPs to any declaration in the code. These UDPs are then made visible at compile time. This is a convenient feature that allows modification in the code behaviour on basis of presence or absence of certain attributes. Vlang uses this feature to provide `@rand` attribute that tags class elements that are required to exhibit randomization behavior. Note that UDP is a compile-time feature and does not add to runtime application memory footprint.

*3) Compile Time Function Evaluation (CTFE):* CTFE in D is very powerful. There are very few D constructs that are not allowed to be evaluated at compile-time. Vlang uses CTFE to implement a parser for constraint blocks.

*4) Mixins:* A mixin enables change in behaviour of a class by allowing addition of code at compile time. D allows string as well as template mixins. Vlang's constraint engine converts the parsed constraint into BDD equations at compile time and uses string mixins to insert the BDD equations.

Additionally, the D Programming Language has multiple features that make it more attractive to hardware verification engineers:

*5) Automatic Garbage Collection:* An automatic GC takes away the pain of memory management away from the end-user. [15] notes that modern garbage collectors are not a source of inefficiency. Also, when required, D allows a user to take control of memory management by allowing him to shut down the GC on certain portions of the code.

*6) First Class Arrays:* Unlike C/C++, a D array object is a fat pointer that stores both the address and the length of an array. D also has support for dynamic arrays, slices and array operators that make vector operations in D very convenient.

*7) Associative Arrays:* D supports associative arrays as a language feature. This basically means that the user does not have to rely on a library and that D enables a convenient/readable syntax for associative arrays.

*8) Class Objects are References:* Like Java, class objects in D are references by default. The keyword `struct` is still available for creating *value* type objects and *plain old data* type objects that are compatible with C/C++.

*9) Unittest:* The `unittest` construct in D makes it convenient to add localized test blocks to D code. These tests can be used to verify the test-bench. Unit level test support is not native to SystemVerilog and verification engineers have to rely on non-native library support such as [16].

*10) A Pointer-less Programming Experience:* Most of D code (thanks to first class arrays and automatic garbage collection), is devoid of pointers. Pointers are still available to enable low level memory and IO access.

*11) Application Binary Interface (ABI) Compatibility and C/C++ interface:* D allows native calls to any C/C++ global and namespace scoped functions. D also allows the user to directly call a virtual member function of a C++ class object. Unlike SystemVerilog DPI, there is no runtime overhead while calling C/C++ functions from inside D or vice versa.

*12) Generic Programming:* D has extensive support for generic programming and meta-programming. It ships with a powerful library of data structures, algorithms and other utility modules.

## VI. Future Work

The future and future work of Vlang is very interesting and we want to place a brief update on that.

### A. Software Engineering Based Methodology

One of the very important reason for D to be the base language of Vlang is that D makes Vlang a SW domain language which provides a clean and extremely pleasurable coding experience to user. Every methodology in verification is a direct import of test concepts in SW engineering domain. Hence, work towards Vlang methodology will be directly attributed towards exporting SW engineering concepts on Vlang level directly so that Vlang can be a playground of verification concepts.

### B. Support for Emulation Platform

As a verification language with full and native support for systems programming language Vlang is the most suitable initiator in emulation platform development and hence a library can be developed on the top of Vlang to support emulation.

### C. Verification IPs

We are in the process of creating a set of standard VIPs that will also integrate seamlessly with software development tools.

Vlang as an open source verification language provides an immediate working platform to academia and enthusiast and expects itself to be exploited in academic and industrial research environment and result of these activities will greatly decide the future direction Vlang development.

## VII. Acknowledgements

I wish to thank *Sumit Adhikari* of *NXP Semiconductors, Hamburg*. The idea of creating verification language Vlang was conceived during a friendly discussion with him. He has also been a constant source of encouragement and has provided valuable feedback in the process of development of Vlang.

## References

[1] A. Molina and O. Cadenas, "Functional verification: approaches and challenges," 2007. [Online]. Available: http://www.scielo.org.ar/scielo.php?pid=S0327-07932007000100013&script=sci_arttext

[2] J. Teubner and L. Woods, *Data Processing on FPGAs*. Morgan & Claypool Publishers, 2013.

[3] H. Sutter. (2005) The free lunch is over: A fundamental turn toward concurrency in software. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[4] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2013.

[5] J. L. Gustafson, *The End of Error: Unum Computing*. Chapman & Hall, 2015.

[6] K. Paranjape, S. Hebert, and B. Masson, "Heterogenous computing in the cloud," 2012. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/xeon-phi-life-sciences-computing-paper.pdf

[7] IEEE. The next generation of ethernet, an overview. [Online]. Available: http://www.ieee802.org/3/hssg/public/sept07/tutorial_01_0907.pdf

[8] C. Spear, *SystemVerilog for Verification*. Springer, 2006.

[9] A. Erickson. Introducing uvm connect. [Online]. Available: http://verificationhorizons.verificationacademy.com/volume-8_issue-1/articles/stream/introducing-uvm-connect-verification-horizons-article.pdf

[10] A. S. Parag Goel and H. V. Balisetty, ""c" you on the faster side: Accelerating sv dpi based co-simulation," *Proceedings of Design Verification Conference, San Jose*, 2014.

[11] Wikipedia. System programming language. [Online]. Available: en.wikipedia.org/wiki/System_programming_language

[12] D programming language. [Online]. Available: dlang.org

[13] The go programming language. [Online]. Available: golang.org

[14] The rust programming language. [Online]. Available: rust-lang.org

[15] P. Vladimir, "Memory management in the d programming language," Ph.D. dissertation, Technical University of Moldova, 2009.

[16] B. Morris, "Svunit: Bringing agile methods into functional verification," *SNUG, San Jose*, 2009.