Abstract—IP/SoC verification is a fundamental problem in design cycle which needs support from a suitable and powerful language which must include the latest findings in computer science. State-of-the-art verification languages are decade old, closed source, not software domain, single paradigm, type unsafe, advocate code boilerplate, single core and not supportive to generic programming. This forces verification engineer to use decade old language concepts, using a HW domain language to build a software called test-bench using a single incomplete programming paradigm called object orientation. Furthermore, incomplete support for object orientation added with no support for generative programming forces the user to write redundancy which could be avoided. In this article we solve this age old verification problem by introducing a novel open source verification language called Vlang [1]. Vlang is built on top of D Programming Language [2], and consequently it inherits support for parallel, generic, generative, and functional programming paradigms in addition to Object Oriented Programming. Vlang is ABI compatible with C/C++, creates single executable with no macros as not evil by mentioning that they are unavoidable and integral part of any software. According to authors of this article, macros are abuse of modern computer science and this malpractice must not be encouraged in a language which has been developed over past decade. The article [17] presents a cost-benefit analysis on favour of using macros! We believe that modern programming language features like reflections and generative programming can be used in combination to avoid macros and the related gotchas altogether. Articles [18]–[22] and many more encourage SV to be used in RTL design, which brings monolithicity in testing, a strict negative in modern testing (see Section I-B). Without availability of proper research articles, authors of this article had to get into deeper language and methodological aspects of SV on their own. Following are the observations:

A. Language Related Issues

As a language, SV is an extremely bulky standard with over 248 keywords [11]. SV encourages boilerplate code generation using Verilog Pre-Processor (VPP) macro expansions, a practice that leads to a plethora of hard to discover bugs [17]. Integral (and numeric) type discipline of Verilog is weak [23], [24]. Consequently SV allows implicit typecasting when assigning numeric variables often resulting in unintentional loss of information. Integral type-safety is not guaranteed and is particularly dangerous for verification in automotive, medical, aeronautic and other safety critical domains. Integral overflow has been found to be one of the most common source of software bugs [25]. Although SV has been projected as an Object Oriented Programming (OOP) language, serious flaws are there in OOP capabilities of SV. SV encapsulation model does not scale. SV provides access specifiers: local, protected and public. Even though this is very much like C++ and Java, this encapsulation model is not sufficient since often it may be desired to provide access to other classes or functions. C++ allows that via friend specifier [26] and

I. INTRODUCTION

Modern System on Chips (SoCs) are complex, heterogeneous [6], [7] and heavily programmable which makes the task of functional verification [8] extremely complex owing to several domains addressing, register combination and the effect validation. Furthermore, modern SoCs are verified over HW/SW co-simulation platforms which escalates the complexity even further. This requires the verification language used, to be multi-paradigm [9], highly abstracted, generic programmable software domain language [10]. On the other hand the availability of multicore computational platforms requires the verification language to be multicore concurrent in order to extract maximum efficiency from the available computational resources and accomplish verification faster. Compile time features are essential in order to generate lean code for faster computation and to avoid unnecessary memory access.

The world of digital verification is dominated by System Verilog (SV) [11] and hence SV has been considered as the state of art verification language in this article. Credible peer reviewed literatures on SV as a language is barely available and SV as a language has not created interest in serious language experts community in research institutes and academia. Article [12] believes that the generic part of the programming language is served reasonably well and the hardware orientation of the language is beneficial. The only problem mentioned in the article pertains to support for various language constructs across different simulators. Articles [13]–[15] say that the language is weakly typed and full of gotchas and still the same articles encourage SV as a test & verification language!. Article [16] expounds upon standard gotchas subtleties in the SV standard itself which every engineer must know and yet encourages SV to be a preferred verification language. Article [17] clearly addresses macros as not evil by mentioning that they are unavoidable and integral part of any software. According to authors of this article, macros are abuse of modern computer science and this malpractice must not be encouraged in a language which has been developed over past decade. The article [17] presents a cost-benefit analysis on favour of using macros! We believe that modern programming language features like reflections and generative programming can be used in combination to avoid macros and the related gotchas altogether. Articles [18]–[22] and many more encourage SV to be used in RTL design, which brings monolithicity in testing, a strict negative in modern testing (see Section I-B). Without availability of proper research articles, authors of this article had to get into deeper language and methodological aspects of SV on their own. Following are the observations:

A. Language Related Issues

As a language, SV is an extremely bulky standard with over 248 keywords [11]. SV encourages boilerplate code generation using Verilog Pre-Processor (VPP) macro expansions, a practice that leads to a plethora of hard to discover bugs [17]. Integral (and numeric) type discipline of Verilog is weak [23], [24]. Consequently SV allows implicit typecasting when assigning numeric variables often resulting in unintentional loss of information. Integral type-safety is not guaranteed and is particularly dangerous for verification in automotive, medical, aeronautic and other safety critical domains. Integral overflow has been found to be one of the most common source of software bugs [25]. Although SV has been projected as an Object Oriented Programming (OOP) language, serious flaws are there in OOP capabilities of SV. SV encapsulation model does not scale. SV provides access specifiers: local, protected and public. Even though this is very much like C++ and Java, this encapsulation model is not sufficient since often it may be desired to provide access to other classes or functions. C++ allows that via friend specifier [26] and

Puneet Goel and Sumit Adhikari
Coverify Systems Technology, India and NXP Semiconductors, Germany
email: puneet@coverify.com and sumit.adhikari@nxp.com
Java extends the encapsulation to packages, but SV does not support either (SV does have a package construct, but it does not provide an access specifier that works across the package scope). Article [27] notes the absence of a standard library for SV. [28] notes that lack of data structure compatibilities comes in the way while using Direct Programming Interface (DPI) to fill the gap created by absence of a generic library. SV lacks function overloading. Function overloading is an essential feature to enable generic programming. Though SV supports parameterized classes, it lacks support for template functions. Class parameterization too is incomplete since SV provides no support for template specialization. Although SV provides const specifier, it does not provide a way to make a class value constant [29]. Consequently, SV does not allow specifying class methods as const. SV does not enable any multicore support. With the advent of multicore processors, lack of parallelism may become a bottleneck [30].

B. Methodology Related Issues

SV maintains complete backward compatibility with Verilog which is essential to support legacy code. But this advantage, thought to be very important, is lost because of monolithicity thus introduced. Monolithic testbenches have been considered evil [31]. Following are the problems associated with monolithicity that SV introduces in testbenches:

1) Propagation of Gotchas: When the design and the reference model are both coded in the same language, there is a greater probability of having similar gotchas [32] and pitfalls. This gives rise to situations where the verification environment fails to catch a bug in the design because both the verification and the design are effected by the same language specific gotchas.

2) Exposure to Malpractice: A monolith testbench makes it easy for the verification engineer to copy code from the design to the reference model used for verification. How is that we keep our designer and verification engineers in separate buildings and yet make it possible for them to reuse code across design and verification?

3) Limited Reuse: A monolithic test-bench is tightly integrated with the design and hence makes reuse of the verification environment more difficult.

4) Compilation and Elaboration: A small incremental change in the verification environment coded in SV forces the simulator to elaborate the whole test-bench again. And in case the Design Under Test (DUT) is big, elaboration of test-bench takes hours. On the other hand we see that most contemporary modern languages like Go [33] and D [2] have set themselves a goal to reduce compile time from minutes to a few seconds. A long compile (and elaboration) time becomes an impediment where frequent compile cycles are required. As a result, developers tend to avoid compilation for long development periods and loose on the value provided by such compilation cycle.

The system level features of SV has been described as oxymoron in article [34]. In fact SV does not enable any aspect of systems programming [35]. This makes SV to be extremely unfriendly for it to be used as HW/SW co-verification or emulation. It may be argued that DPI is an enabler for systems programming in SV. But SV DPI has its own drawbacks. Though more efficient compared to Verilog Programming Language Interface (PLI), DPI still has a runtime overhead which becomes a performance bottleneck [30].

To conclude, design and verification both are two completely different tasks. For design, Verilog and VHDL serve the purpose finely, but SV is an insufficient language for many aspects of verification. In fact any software domain language with notion of time and bit-vector support can serve the purpose. Constraint solving can be easily achieved by using a Binary Decision Diagram (BDD) [36]. This article addresses these issues by introducing a novel, open source verification language called Vlang which is an extension of a software domain programming language known as D [37], [38].

II. INTRODUCTION TO VLANG

As mentioned above, SV fails to meet the challenging requirements of System Level verification. In authors 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 Application Binary Interface (ABI) compatibility with C/C++. There is a very vast amount of C/C++ code in the wild and SV DPI experience [30] 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 [35]: C++, D [2], Go [33] and Rust [39]. 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 Domain Specific Language (DSL) on top of it without having to overly depend on the C pre-processor. In Comparison, D, as we shall see in Section II-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 Universal Verification Methodology (UVM) util functions (ref section II-D) 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):** Compile Time Function Evaluation (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 (ref II-B2).

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 Garbage Collector (GC) takes away the pain of memory management away from the end-user. [40] 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 SV and verification engineers have to rely on non-native library support such as [41].

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) **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 SV 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.

**B. Introduction to Vlang Features and Semantics**

Vlang is a DSL built on top of the D Programming Language [38]. D is a mainstream software language and Vlang adds hardware modelling and verification features to D. This is in contrast to SV which adds software and verification features to the Hardware Description Language, Verilog.

![Fig. 1. Vlang vs System Verilog approach](image)

1) **Multicore Simulations:** Vlang allows its users to create threads that run parallelly on multiple cores of a processor. It also enables the user to have a fine grained control on how the thread gets executed. The user is also allowed to disable parallelism on any module that he deems unfit for concurrent execution.

Vlang supports both user-level and kernel-level threads. User-level threads (called Tasks in Vlang) are executed as a sequence on a pool of threads pre-designated only for executing tasks. By default, all the tasks in an entity instance are grouped together and get associated with the same thread in the pool. This is done to minimize the tasks that share data – as tasks belonging to the same entity instance normally would – from having to refresh processor caches. The user is allowed to fine tune this behavior by associating a given task with another entity.

Vlang tags an ID to every entity instance. This ID is used by Vlang to automatically determine the thread in the thread-pool to associate the entity with. Thus, each server thread in the thread-pool handles list of entities, and each entity has a group of tasks that need to be executed. In certain cases, if the default association of entities with threads in the thread-pool, is resulting in an unbalanced distribution of tasks on the threads, Vlang allows the user to tweak the task distribution by explicitly tagging certain entities with a specified thread. The user can generate a report on the task load distribution over various threads in the pool by specifying a UDP tag on the simulator instance he wishes to generate a report for.
Currently, Vlang does not have an automatic load balancer. An automatic load balancer could relocate a group of tasks to another thread which may have already executed all the tasks assigned to it in the current simulation cycle. But relocating a task might result in a cache update cycle. Thus, frequent relocation of tasks could be counter-productive. Any useful load balancer should observe the prevalent load over a number of simulation cycles before deciding to move tasks between the threads.

In certain cases, a user might want to dedicate a kernel thread for certain tasks. Vlang implement a Worker process for this purpose. Unlike a user-level thread, a worker thread is visible to the underlying operating system kernel. Since Operating System (OS) calls are costly, creating unnecessary worker threads might have a bearing on the simulation performance.

Like SystemC, Vlang simulation engine mainly constitutes of three distinct phases: schedule phase, execute phase and update phase. During execute phase, tasks are executed. Once the simulator has executed all the tasks, it updates the channels, and then triggers the events that have been notified during the update phase or during execute phase.

```c
class Foo: Randomizable {
    mixin(randomization());
    @rand byte[] foo;
    @rand ubyte baz = 12;
    void display();
}

class Bar: Foo {
    mixin(randomization());
    @rand ubyte[8] bar;
    override void display() {
        writeln("foo: ", foo);
        writeln("bar: ", bar);
        writeln("baz: ", baz);
    }
}

Constraint! q{ foo.length > 2;
    baz < 16;
} cstFooLength;

Constraint! q{
    foreach(i, f; bar) f <= i;
    foreach(i, f; foo) {
        f < 64 && f > 16;
    }
} cstFoo;

void main() {
    Foo randObj = new Bar();
    for (size_t i=0; i!=10; ++i) {
        randObj.randomize();
        randObj.display();
    }
}
```

Fig. 2. Constrained Randomization in Vlang

2) Constrained Randomization Engine: Like SV, Vlang constraints and randomisation are based on class semantics. As illustrated in Figure 2, the method randomize follows inheritance and has a polymorphic character. The user is allowed to declare a dynamic array as @rand. When adding @rand attribute to a dynamic array, the user must provide the maximum allowed size of the array as an argument. This forces an upper size constraint on all the randomizable dynamic array elements to avoid unconstrained sizing of arrays during randomization. if-else and foreach constraints blocks are also supported and can be freely nested.

The constraint solver supports enabling and disabling constraints and overriding them in a derived class. Like randomize, another method randomizeWith is provided to allow the user specify external constraints on a randomized object.

Under the hood, all constraint blocks are parsed at compile time. The identifiers used in constraint expressions are resolved and mapped to class members or other variables visible in the scope. Since the constraints are parsed at compile time, any unresolved identifiers and any syntactic errors in constraint expressions are reported during compilation itself.

Once all the identifiers are resolved, the constraint parser transforms the constraints into BDD equations that are inserted into the code via mixin mechanism. Vlang uses Buddy [42] BDD package for solving BDD equations. For this purpose, Buddy has been ported to D language. The ported version of Buddy also has an object wrapper over buddy instance to enable multiple instantiations of the constraint solver engines. For every randomized class object, Vlang instantiates an object specific constraint solver. In a real world verification platform supporting multicore processing (e.g. the Vlang port of UVM), there are multiple sequence generators running over a multitude of processor threads. These sequence generators each have their own individual constraint solver running parallelly with other constraint solvers serving other sequence generators.

C. Discussion of Simulation Kernel and Integration

The core part of Vlang constitutes of a lightweight Discrete Event Simulator. The Simulator is completely encapsulated in a class and its instance is automatically attached to a RootEntity. A RootEntity in Vlang is a base class that may be extended by the user to create the top level hierarchy of his simulation model. One or more entity (another base class) objects may be instantiated inside a RootEntity, to form the hierarchical structure of the simulation model. Before starting the simulation, the user has to create an instance of the RootEntity and elaborate it. Both elaboration and simulation are handled on a RootThread associated with the simulator. The elaboration process consists of multiple phases, namely: build, config and connect. Each simulation cycle too consists of multiple phases as illustrated in Figure 3.

As already mentioned, Vlang simulation engine is completely encapsulated inside a class object. This enables the user to create multiple simulators. Since each simulator has its own RootThread, the simulations can be run in parallel or if required in a sequence.
1) SystemC Co-simulation: Any Co-simulation environment involving two or more simulators requires synchronization of simulation time and efficient exchange of data. On both these aspects Vlang enables a better interface to SystemC compared to SV.

There are two aspects of any co-simulation environment involving two or more simulators. The first aspect deals with how the simulators synchronize with each other with respect to simulation time. The second aspect deals with synchronization of data to be shared/passed between the simulators.

When Vlang is launched in multicore mode, SystemC can be integrated with it on a parallely running thread. It is easy to interface Vlang with SystemC using 

```
sc_time_to_pending_activity
```

and

```
sc_pending_activity
```

Application Programming Interface (API) calls provided by SystemC. Just like SystemC, Vlang provides APIs for incremental execution of simulation, thus enabling Vlang to interface with SystemC in both master and slave mode.

SV provides Direct Programming Interface (DPI) to integrate a SV simulation with SystemC. Though DPI is a lot more efficient compared to traditional Verilog PLI interface, being non-native, it still has a significant overhead.

Since even the Vlang scheduler runs on a separate thread, Vlang can execute in parallel with SystemC.

D. UVM Implementation

Vlang has a complete, multicore enabled implementation of UVM-1.1d release. Reflections and Compile Time Function Evaluation help Vlang in enabling a more bug free and productive experience to the end-user, as illustrated by the code snippet in Figure 2.

1) UVM Utility Methods: SV UVM provides convenient macros that help automatic generation of utility methods such as create, pack, unpack, copy, clone, compare, sprint and record. These macros `uvm_object_utils` and `uvm_component_utils` are also responsible for registering uvm objects and components with UVM factory. Vlang implementation uses D reflection and mixin mechanism, thus completely avoiding all the disadvantages of macros pointed out in [17]. Additionally, Vlang UVM implementation takes advantage of function overloading and template mechanism of D and the utility methods thus created do not incur runtime inefficiency.

2) Multicore UVM Architecture: Figure 7 illustrates generic architecture of a UVM based test-bench. The object oriented architecture of the UVM library and the phasing mechanism provide a handy structure for introducing parallelism into UVM. From data sharing and encapsulation perspective, a `uvm_agent` provides the right abstraction to provide a parallelism context, and that Vlang does by default. The
import uvm;
import esdl;

enum bus_op_t: int {READ, WRITE, NOP} @UVM_DEFAULT // tag all fields

class bus_trans: uvm_sequence_item {
    @rand bvec!12 addr;
    @rand!32 byte data[];
    @rand bus_op_t op;
    // Look ma no macros
    mixin uvm_object_utils;
    Constraint! q{
        if(op == bus_op_t.NOP)
            data.length == 0;
        else
            data.length != 0;
        foreach(d; data) d < 32;
    }
    // Constructor
    this(string name="") {
        super(name);
    }
}

class my_driver(REQ, RSP):
    uvm_driver(REQ, RSP) {
        mixin uvm_component_utils;
        private int data_array[512];
        // Constructor
        this(string name, uvm_component parent) {
            super(name, parent);
        }
        override void run_phase(uvm_phase phase) {
            while(true) {
                assert(seq_item_port !is null);
                seq_item_port.get(req);
                auto rsp = new RSP();
                rsp.set_id_info(req);
                if(req.op == bus_op_t.READ) {
                    rsp.addr = req.addr[0..9];
                    rsp.data = cast(bvec!8) data_array[rsp.addr].toBitVec;
                    uvm_info("sending", rsp.to!string, UVM_MEDIUM);
                }
                else {
                    data_array[req.addr] = req.data;
                    uvm_info("sending", req.to!string, UVM_MEDIUM);
                }
                seq_item_port.put(rsp);
            }
        }
    }
}

Fig. 6. UVM code snippet in Vlang

end user has the freedom to move that context to another UVM component though. Working with multiple parallel threads, generally involves two concerns: Data Sharing and Synchronization. For any complex System Level Design Under Test (DUT), there would be multiple hardware interfaces. As a thumb rule, there would be a separate uvm_agent for every hardware interface. Since most hardware interfaces have an independent character, in general there would be no data sharing between two uvm_agents, and therefore it is perfectly safe to run each uvm_agent on a separate thread. At uvm_env and uvm_root abstraction level, UVM implements control mechanisms including phases, objections and configuration. The synchronization required for each of these mechanisms is handled under the hood as part of the uvm base class libraries and the end user is not exposed to multicores synchronization intricacies. Coverage and Scoreboarding do require sharing of data at uvm_env abstraction level. But since blocking Transaction Level Modeling (TLM) port fetches are used for getting transactions for scoreboard as well as coverage, data synchronization would never be an issue. This is because under the hood, every transaction exchange via blocking methods involves locking and unlocking of mutex locks.

3) Support for Multiple UVM Roots: The Vlang port of UVM provides a completely encapsulated platform, leading to the possibility of creating multiple UVM root instances, which can execute in parallel or in sequence. Multiple UVM root instances open up exciting possibilities in System Level Verification and bring in new opportunities for better processor utilization on modern multicore servers.

E. Multicore UVM Performance

As per Amdahl’s Law, the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program [43]. As illustrated in Figure 3, only the tasks are run in parallel in a Vlang simulation. The rest of the simulator, consisting of scheduler steps, is executed sequentially. At TLM abstraction level, the number events, and consequently, the scheduler part of the simulation, are both minimalized.

For a UVM testbench in Vlang, only the run_phase tasks are executed parallelly. In any simulation these tasks constitute almost the whole of the simulation runtime. Further, any resources used by the parallel tasks in a shared manner, also result in performance depletion. In UVM these shared
resources could be the objection mechanism and the configuration database. Of the two, the configuration database is generally not accessed so much in the run phase. Additionally some hardware resources such as the terminal on which the uvm_report messages are dumped, as well as the level2 and level3 caches are also shared. In case of hyperthreading processor cores, much of instruction fetch and decode logic is also shared between the logical processors.

Since Vlang lacks an automatic load balancer, the performance of a parallel simulation could also take a hit if the various CPU cores available are assigned the tasks unevenly. In such a scenario, some CPU cores would sit idle for the time when other CPU cores are working hard to finish the tasks assigned to them. Often, the runPhase tasks would be dominated by the activities pertaining to constrained randomization part of stimulus generation.

Table I and Table II tabulate the results of some example testcases that were run on a multicore processor having 4 cores. While Table I testcases generated transaction packets of randomly varying sizes, Table II data corresponds to fixed size transactions.

### TABLE I

**MULTICORE PERFORMANCE GAIN FOR VARIABLE SIZED TRANSACTIONS**

<table>
<thead>
<tr>
<th>Agents Count</th>
<th>Transaction Count</th>
<th>Multicore Runtime</th>
<th>Single Core Runtime</th>
<th>Performance Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1250</td>
<td>81</td>
<td>195</td>
<td>2.41</td>
</tr>
<tr>
<td>32</td>
<td>2500</td>
<td>78</td>
<td>193</td>
<td>2.47</td>
</tr>
<tr>
<td>16</td>
<td>5000</td>
<td>74</td>
<td>186</td>
<td>2.51</td>
</tr>
<tr>
<td>8</td>
<td>10000</td>
<td>84</td>
<td>180</td>
<td>2.14</td>
</tr>
<tr>
<td>4</td>
<td>20000</td>
<td>91</td>
<td>177</td>
<td>1.95</td>
</tr>
</tbody>
</table>

The variation in the performance gain can be attributed to:

1) Multicore performance gain is better for fixed size transaction generation. This is expected since the dynamic spread of task activities across the various cores would be more evenly distributed in case of fixed size transactions.

2) For small number of UVM agents, the multicore performance gain is less. This variation pertains to larger percentage of scheduler activity in such scenarios.

3) For very large number of UVM agents too, multicore performance gain takes a slight hit. This may be due to requirement of more memory to solve constraints as the number of transactions to be generated in a simulation cycle would increase with increase in number of UVM agents. Since level 2 and level 3 caches are often shared between multiple cores, requirement of more memory from multiple active agents (as would be the case with multicore simulation) would result in more cache misses and hence the slight dip in performance gain.

It is imperative to note here that the present version of Vlang is not optimized for multicore performance. Since this is the first release of a multicore enabled UVM, fixing race conditions and deadlocks had much more priority compared to fine tuning the multicore performance. The first release of SMP Linux Kernel too had similar development priorities [44].

### III. CONCLUSION

### TABLE III

**PERFORMANCE ENABLER COMPARISON TABLE**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Vlang</th>
<th>SV</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent Threads</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Multiple Conc. Simulators</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Generic Lib Support</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### TABLE IV

**CODING PRODUCTIVITY COMPARISON TABLE**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Vlang</th>
<th>SV</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile Time</td>
<td>Fastest</td>
<td>Slow</td>
<td>Fast</td>
</tr>
<tr>
<td>Incremental Compile</td>
<td>Yes</td>
<td>Partial</td>
<td>Yes</td>
</tr>
<tr>
<td>Automatic GC</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>User-friendly Containers</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

### TABLE V

**RUN TIME SAFETY COMPARISON TABLE**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Vlang</th>
<th>SV</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Bound Check</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Support for Unittests</td>
<td>Builtin</td>
<td>Library</td>
<td>Library</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Contract Programming</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### TABLE VI

**SYSTEMS PROGRAMMING FEATURES COMPARISON TABLE**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Vlang</th>
<th>SV</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW/device access</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Custom memory allocation</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Efficient File IO</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Parsing tools/libraries</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Embedded Assembly Code</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

In this article we have discussed the language issues associated with state of art verification language namely SV. To solve the problems we have introduced a novel open source verification language called Vlang. We discussed the language overview and internal workings of Vlang in brief. A
TABLE VII
REFLECTIONS AND GENERATIVE PROGRAMMING COMPARISON TABLE

<table>
<thead>
<tr>
<th>Feature</th>
<th>Vlang</th>
<th>SV</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Introspection and Reflections</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Generative and Metaprogram.</td>
<td>Yes</td>
<td>No</td>
<td>Limited</td>
</tr>
</tbody>
</table>

TABLE VIII
UVM COMPARISON TABLE

<table>
<thead>
<tr>
<th>Feature</th>
<th>Vlang</th>
<th>SV</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Class Libraries</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for Sequences</td>
<td>Yes</td>
<td>Yes</td>
<td>Limited</td>
</tr>
<tr>
<td>Register Abstraction Layer (RAL)</td>
<td>No*</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>TLM1 Support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TLM2 Support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*RAL support for Vlang is being actively ported

TABLE IX
VERIFICATION FEATURE COMPARISON TABLE

<table>
<thead>
<tr>
<th>Feature</th>
<th>Vlang</th>
<th>SV</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Randomization</td>
<td>Yes</td>
<td>Yes</td>
<td>Limited</td>
</tr>
<tr>
<td>Sequence Randomization</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Coverage Support</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

comparison of performance among Vlang, SV and SystemC has been summarized in Table III. As it can be seen that currently the only language which supports multicore concurrency is Vlang. Both Vlang and SystemC (SC) are supported by a strong generic standard library. Table IV summarizes the language performance comparison, where Vlang has the fastest compile time, supports incremental compilation, pointer-less programming, has automatic garbage collector and has user friendly containers. The run time safety has been summarised in Table V where Vlang is at par with SC and also adds additional feature of contract programming which is extremely useful for self checking test benches and early capture of testbench bugs. Systems programming features comparison has been summarized in Table VI, where it can be seen that Vlang is at par with SC and hence it is an extremely suitable language for developing emulation platforms in comparison to SV. Comparison Table VII shows Vlang is the most powerful in terms of generic programming which authors of this article considers most important modelling feature. Vlang-UVM is multicore concurrent and multiple uvm_root can be instantiated. Table IX shows that Vlang does not support functional coverage currently.

From the above discussion, we can see that Vlang excels on all aspects (except for some features like coverage which can easily be implemented) making it potentially the most modern and powerful verification language. Also, by the virtue of D which is the base language of Vlang, Vlang is extremely modern and easy to learn especially when the user wishes to use powerful features. Vlang is fastest (multicore, compile time for lean code generation), supports heavy generic programming which allows user to spend less time is repetitive writing. Developing library on the top of Vlang is very easy as base language D supports it fundamentally. Hence, we conclude that Vlang is the most suitable language for modern functional verification.

IV. 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. Support for Embedded Systems (HW/SW) Verification

The next big thing is verification/testing of SW is a virtual prototyped environment and with the virtue of D, Vlang is utmost suitable to perform the SW verification along with HW.

D. Standard Template Models

In future, Vlang will add a standard template library as a part of language which will provide a significant power to user.

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.

V. ACKNOWLEDGEMENTS

We thank Martin Barnasconi of NXP Semiconductors and Torsten Maehne of UMPC Paris for providing us encouragement, continuous feedback and criticism of what we did at every step of our development.

VI. ACRONYMS

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABI</td>
<td>Application Binary Interface</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BDD</td>
<td>Binary Decision Diagram</td>
</tr>
<tr>
<td>CTFE</td>
<td>Compile Time Function Evaluation</td>
</tr>
<tr>
<td>DPI</td>
<td>Direct Programming Interface</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>DUT</td>
<td>Design Under Test</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collector</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property (Core)</td>
</tr>
</tbody>
</table>