The Emperor Has No Clothes!
Issues with SystemVerilog and UVM that You Must Know

Puneet Goel
Coverify Systems Technology

October 2013
Setting the Agenda

- A few years back Industry Guide and SystemVerilog Guru Stuart Sutherland (along with Don Mills) published a book titled “Verilog and SystemVerilog Gotchas”
- While the book is certainly worthy of a thorough read, this presentation is not about Gotchas
- We will cover stuff at much more fundamental level
- SystemVerilog as a language has been often celebrated and seldom critically analyzed
Setting the Agenda

- A few years back Industry Guide and SystemVerilog Guru Stuart Sutherland (along with Don Mills) published a book titled “Verilog and SystemVerilog Gotchas”
- While the book is certainly worthy of a thorough read, this presentation is not about Gotchas
- We will cover stuff at much more fundamental level
- SystemVerilog as a language has been often celebrated and seldom critically analyzed
Setting the Agenda

- A few years back Industry Guide and SystemVerilog Guru Stuart Sutherland (along with Don Mills) published a book titled “Verilog and SystemVerilog Gotchas”
- While the book is certainly worthy of a thorough read, this presentation is not about Gotchas
- **We will cover stuff at much more fundamental level**
- SystemVerilog as a language has been often celebrated and seldom critically analyzed
Setting the Agenda

- A few years back Industry Guide and SystemVerilog Guru Stuart Sutherland (along with Don Mills) published a book titled “Verilog and SystemVerilog Gotchas”
- While the book is certainly worthy of a thorough read, this presentation is not about Gotchas
- We will cover stuff at much more fundamental level
- SystemVerilog as a language has been often celebrated and seldom critically analyzed
The Agenda

Not necessarily in this order, but I will be covering:

- SystemVerilog Language Design Flaws and much desired features
- Verification Futures – some hardware/software perspectives
  - Multicore
  - The widening ESL/RTL gap
- Some Alternatives
The Agenda

Not necessarily in this order, but I will be covering:

- SystemVerilog Language Design Flaws and much desired features
- *Verification Futures* – some hardware/software perspectives
  - Multicore
  - The widening ESL/RTL gap
- Some Alternatives
The Agenda

Not necessarily in this order, but I will be covering:

▶ SystemVerilog Language Design Flaws and much desired features
▶ Verification Futures – some hardware/software perspectives
  ▶ Multicore
  ▶ The widening ESL/RTL gap
▶ Some Alternatives
The Agenda

Not necessarily in this order, but I will be covering:

- SystemVerilog Language Design Flaws and much desired features
- *Verification Futures* – some hardware/software perspectives
  - Multicore
  - The widening ESL/RTL gap
- Some Alternatives
The Agenda

Not necessarily in this order, but I will be covering:

- SystemVerilog Language Design Flaws and much desired features
- Verification Futures – some hardware/software perspectives
  - Multicore
  - The widening ESL/RTL gap
- Some Alternatives
A Little Perspective

long long ago...

- Designers used to verify designs
- People used Verilog for verifying their designs
  - Or they used C/C++ and/or some scripting language
- RTL has more or less remained static for the past 20 years now
  - Except for some syntactic sugar getting added in SystemVerilog
  - Behavioural synthesis has failed to get mainstream
A Little Perspective

long long ago...

- Designers used to verify designs
- People used Verilog for verifying their designs
  - Or they used C/C++ and/or some scripting language
- RTL has more or less remained static for the past 20 years now
  - Except for some syntactic sugar getting added in SystemVerilog
  - Behavioural synthesis has failed to get mainstream
long long ago . . .

- Designers used to verify designs
- People used Verilog for verifying their designs
  - Or they used C/C++ and/or some scripting language
- RTL has more or less remained static for the past 20 years now
  - Except for some syntactic sugar getting added in SystemVerilog
- Behavioural synthesis has failed to get mainstream
long long ago...

- Designers used to verify designs
- People used Verilog for verifying their designs
  - Or they used C/C++ and/or some scripting language
- RTL has more or less remained static for the past 20 years now
  - Except for some syntactic sugar getting added in SystemVerilog
  - Behavioural synthesis has failed to get mainstream
A Little Perspective

long long ago...

- Designers used to verify designs
- People used Verilog for verifying their designs
  - Or they used C/C++ and/or some scripting language
- RTL has more or less remained static for the past 20 years now
  - Except for some syntactic sugar getting added in SystemVerilog
  - Behavioural synthesis has failed to get mainstream
A Little Perspective

long long ago...

- Designers used to verify designs
- People used Verilog for verifying their designs
  - Or they used C/C++ and/or some scripting language
- RTL has more or less remained static for the past 20 years now
  - Except for some syntactic sugar getting added in SystemVerilog
  - Behavioural synthesis has failed to get mainstream
The RTL part is so static that IEEE merged the Verilog standard into SystemVerilog

SystemVerilog adds HVL features:
- Constrained Randomization
- Assertions
- Functional Coverage

And Java-like software features:
- Object Oriented Programming
- Dynamic Scoping
- Garbage Collection

The Emperor Has No Clothes!
SystemVerilog: a Bottom Up Language

- The RTL part is so static that IEEE merged the Verilog standard into SystemVerilog
- SystemVerilog adds HVL features:
  - Constrained Randomization
  - Assertions
  - Functional Coverage
- And Java-like software features:
  - Object Oriented Programming
  - Dynamic Scoping
  - Garbage Collection
SystemVerilog: a Bottom Up Language

- The RTL part is so static that IEEE merged the Verilog standard into SystemVerilog.
- SystemVerilog adds HVL features:
  - Constrained Randomization
  - Assertions
  - Functional Coverage
- And Java-like software features:
  - Object Oriented Programming
  - Dynamic Scoping
  - Garbage Collection
SystemVerilog: a Bottom Up Language

- The RTL part is so static that IEEE merged the Verilog standard into SystemVerilog
- SystemVerilog adds HVL features:
  - Constrained Randomization
  - Assertions
  - Functional Coverage
- And Java-like software features:
  - Object Oriented Programming
  - Dynamic Scoping
  - Garbage Collection

The Emperor Has No Clothes!
The RTL part is so static that IEEE merged the Verilog standard into SystemVerilog.

- SystemVerilog adds HVL features:
  - Constrained Randomization
  - Assertions
  - Functional Coverage

- And Java-like software features:
  - Object Oriented Programming
  - Dynamic Scoping
  - Garbage Collection
The RTL part is so static that IEEE merged the Verilog standard into SystemVerilog

SystemVerilog adds HVL features:
- Constrained Randomization
- Assertions
- Functional Coverage

And Java-like software features:
- Object Oriented Programming
- Dynamic Scoping
- Garbage Collection
The RTL part is so static that IEEE merged the Verilog standard into SystemVerilog.

SystemVerilog adds HVL features:
- Constrained Randomization
- Assertions
- Functional Coverage

And Java-like software features:
- Object Oriented Programming
- Dynamic Scoping
- Garbage Collection
The RTL part is so static that IEEE merged the Verilog standard into SystemVerilog.

- SystemVerilog adds HVL features:
  - Constrained Randomization
  - Assertions
  - Functional Coverage

- And Java-like software features:
  - Object Oriented Programming
  - Dynamic Scoping
  - Garbage Collection
SystemVerilog: a Bottom Up Language

- The RTL part is so static that IEEE merged the Verilog standard into SystemVerilog
- SystemVerilog adds HVL features:
  - Constrained Randomization
  - Assertions
  - Functional Coverage
- And Java-like software features:
  - Object Oriented Programming
  - Dynamic Scoping
  - Garbage Collection
SystemVerilog: the HDVL

- Unified Language for Design and Verification

Coding reference model in the same language as the language in which your design has been coded is often a bad idea.

- Share the language features and the Corresponding Gotchas

- Working with a common language leads to more opportunities of more interactions between the designer and the verification engineer

- If your designer is a Punjabi, get a Tamil to verify the design.

The Emperor Has No Clothes!
SystemVerilog: the HDVL

- Unified Language for Design and Verification
- Coding reference model in the same language as the language in which your design has been coded is often a bad idea

- Share the language features and the Corresponding Gotchas
- Working with a common language leads to more opportunities of more interactions between the designer and the verification engineer

- If your designer is a Punjabi, get a Tamil to verify the design
SystemVerilog: the HDVL

- Unified Language for Design and Verification
- Coding reference model in the same language as the language in which your design has been coded is often a bad idea

- Share the language features and the Corresponding Gotchas
- Working with a common language leads to more opportunities of more iterations between the designer and the verification engineer
- If your designer is a Punjabi, get a Tamil to verify the design
SystemVerilog: the HDVL

- Unified Language for Design and Verification
- Coding reference model in the same language as the language in which your design has been coded is often a bad idea

- Share the language features and the Corresponding Gotchas
- Working with a common language leads to more opportunities of more interactions between the designer and the verification engineer
  - If your designer is a Punjabi, get a Tamil to verify the design

The Emperor Has No Clothes!
SystemVerilog: the HDVL

- Unified Language for Design and Verification
- Coding reference model in the same language as the language in which your design has been coded is often a bad idea

- Share the language features and the Corresponding Gotchas
- Working with a common language leads to more opportunities of more interactions between the designer and the verification engineer
  - If your designer is a Punjabi, get a Tamil to verify the design
Verification is Software

- **SystemVerilog (and UVM) is 80% software**
- With the ever increasing gap between the specifications and RTL, there would be more and more software in the future
Verification is Software

- SystemVerilog (and UVM) is 80% software
- With the ever increasing gap between the specifications and RTL, there would be more and more software in the future
Verification is Software

- SystemVerilog (and UVM) is 80% software
- With the ever increasing gap between the specifications and RTL, there would be more and more software in the future
For high-level applications programming to be effective and convenient, we need libraries. Using just the bare language features makes almost all programming quite painful. That’s true for every general-purpose language. Conversely, given suitable libraries just about any programming task can be pleasant...
SystemVerilog Containers

What does SystemVerilog have for Containers?

- Well, we have the queue construct – is that all?
- No we also have dynamic arrays – Ok, but that is all broken construct
  - Does not grow by itself
  - More like a fancy malloc
- Ok, we also had Linked Lists in IEEE-1800 2005 standard – But it was so broken that it was purged from the later standards
- If you need a container in SV (other than the queue), you have to re-invent the wheel
SystemVerilog Containers

What does SystemVerilog have for Containers?

- Well, we have the queue construct – is that all?
- No we also have dynamic arrays – Ok, but that is all broken construct
  - Does not grow by itself
  - More like a fancy malloc
- Ok, we also had Linked Lists in IEEE-1800 2005 standard – But it was so broken that it was purged from the later standards
- If you need a container in SV (other than the queue), you have to re-invent the wheel
SystemVerilog Containers

What does SystemVerilog have for Containers?

- Well, we have the queue construct – is that all?
- No, we also have dynamic arrays – Ok, but that is all broken construct
  - Does not grow by itself
  - More like a fancy malloc
- Ok, we also had Linked Lists in IEEE-1800 2005 standard – But it was so broken that it was purged from the later standards
- If you need a container in SV (other than the queue), you have to re-invent the wheel
SystemVerilog Containers

What does SystemVerilog have for Containers?

▷ Well, we have the queue construct – is that all?
▷ No we also have dynamic arrays – Ok, but that is all broken construct
  ▷ Does not grow by itself
  ▷ More like a fancy malloc

▷ Ok, we also had Linked Lists in IEEE-1800 2005 standard – But it was so broken that it was purged from the later standards

▷ If you need a container in SV (other than the queue), you have to re-invent the wheel
What does SystemVerilog have for Containers?

- Well, we have the queue construct – is that all?
- No we also have dynamic arrays – Ok, but that is all broken construct
  - Does not grow by itself
  - More like a fancy malloc
- Ok, we also had Linked Lists in IEEE-1800 2005 standard – But it was so broken that it was purged from the later standards
- If you need a container in SV (other than the queue), you have to re-invent the wheel
SystemVerilog Containers

What does SystemVerilog have for Containers?

- Well, we have the queue construct – is that all?
- No we also have dynamic arrays – Ok, but that is all broken construct
  - Does not grow by itself
  - More like a fancy malloc
- Ok, we also had Linked Lists in IEEE-1800 2005 standard – But it was so broken that it was purged from the later standards
- If you need a container in SV (other than the queue), you have to re-invent the wheel
Who Killed the Linked List?

- SystemVerilog inherits `include pre-processor construct from Verilog
- The package construct is semantically close to namespace in C++
- package and `include do not like each other. Consider:

```systemverilog
file foo.sv
1 package Foo;
2 `include "linked-list.sv"
3 endpackage

file bar.sv
1 package Bar;
2 `include "linked-list.sv"
3 endpackage
```

- Nobody #includes C/C++ header files like this
- Since SystemVerilog does not support header files (no separate compiles), such use of `include is prevalent

The end-user killed the Linked List. Ironically he has to code it himself now.
SystemVerilog supports parameterized classes, but still Generic Programming is not possible in SV since:

- No support for parameterized functions/tasks
- No support for function overloading
- No support for operator overloading
- But how does all this affect us? Hang on!
SystemVerilog supports parameterized classes, but still Generic Programming is not possible in SV since:

- No support for parameterized functions/tasks
- **No support for function overloading**
- No support for operator overloading
- But how does all this affect us? Hang on!
SystemVerilog supports parameterized classes, but still Generic Programming is not possible in SV since:

- No support for parameterized functions/tasks
- No support for function overloading
- No support for operator overloading
- But how does all this affect us? Hang on!
SystemVerilog supports parameterized classes, but still Generic Programming is not possible in SV since:

- No support for parameterized functions/tasks
- No support for function overloading
- No support for operator overloading
- But how does all this affect us? Hang on!
If you have programmed using UVM, you would have come across `uvm_object_utils`

UVM Object/Component Util macros automagically define utility functions, including:
- bit/byte packing/unpacking
- copy/cloning
- pretty printing

Since the fields of a transaction item or a component could be of various types (there are a few integral types as well), how does SystemVerilog/UVM handle these operations under the hood?
- All integral types are converted to 4096 bits wide integers
- Even if your field was a mere 8 bit wide, it gets copied into 4096 bits wide parameter of a function

Code your own copy/print/packer functions if you bother about efficiency!
If you have programmed using UVM, you would have come across `uvm_object_utils`

**UVM Object/Component Util macros automagically define utility functions, including:**

- bit/byte packing/unpacking
- copy/cloning
- pretty printing

Since the fields of a transaction item or a component could be of various types (there are a few integral types as well), how does SystemVerilog/UVM handle these operations under the hood?

- All integral types are converted to 4096 bits wide integers
- Even if your field was a mere 8 bit wide, it gets copied into 4096 bits wide parameter of a function

Code your own copy/print/packer functions if you bother about efficiency!
If you have programmed using UVM, you would have come across `uvm_object_utils`.

UVM Object/Component Util macros automagically define utility functions, including:
- bit/byte packing/unpacking
- copy/cloning
- pretty printing

Since the fields of a transaction item or a component could be of various types (there are a few integral types as well), how does SystemVerilog/UVM handle these operations under the hood?
- All integral types are converted to 4096 bits wide integers.
- Even if your field was a mere 8 bit wide, it gets copied into 4096 bits wide parameter of a function.

Code your own copy/print/packer functions if you bother about efficiency!
If you have programmed using UVM, you would have come across `uvm_object_utils`

UVM Object/Component Util macros automagically define utility functions, including:

- bit/byte packing/unpacking
- copy/cloning
- pretty printing

Since the fields of a transaction item or a component could be of various types (there are a few integral types as well), how does SystemVerilog/UVM handle these operations under the hood?

- All integral types are converted to 4096 bits wide integers
- Even if your field was a mere 8 bit wide, it gets copied into 4096 bits wide parameter of a function

Code your own copy/print/packer functions if you bother about efficiency!
If you have programmed using UVM, you would have come across `uvm_object_utils`

**UVM Object/Component Util macros automagically define utility functions, including:**

- bit/byte packing/unpacking
- copy/cloning
- pretty printing

Since the fields of a transaction item or a component could be of various types (there are a few integral types as well), how does SystemVerilog/UVM handle these operations under the hood?

- All integral types are converted to 4096 bits wide integers
- Even if your field was a mere 8 bit wide, it gets copied into 4096 bits wide parameter of a function

Code your own copy/print/packer functions if you bother about efficiency!
If you have programmed using UVM, you would have come across `uvm_object_utils`.

UVM Object/Component Util macros automagically define utility functions, including:
- bit/byte packing/unpacking
- copy/cloning
- pretty printing

Since the fields of a transaction item or a component could be of various types (there are a few integral types as well), how does SystemVerilog/UVM handle these operations under the hood?
- All integral types are converted to 4096 bits wide integers
- Even if your field was a mere 8 bit wide, it gets copied into 4096 bits wide parameter of a function

Code your own copy/print/packer functions if you bother about efficiency!
If you have programmed using UVM, you would have come across uvm_object_utils

UVM Object/Component Util macros automagically define utility functions, including:
- bit/byte packing/unpacking
- copy/cloning
- pretty printing

Since the fields of a transaction item or a component could be of various types (there are a few integral types as well), how does SystemVerilog/UVM handle these operations under the hood?
- All integral types are converted to 4096 bits wide integers
- Even if your field was a mere 8 bit wide, it gets copied into 4096 bits wide parameter of a function

Code your own copy/print/packer functions if you bother about efficiency!
If you have programmed using UVM, you would have come across `uvm_object_utils`

**UVM Object/Component Util macros automagically define utility functions, including:**
- bit/byte packing/unpacking
- copy/cloning
- pretty printing

Since the fields of a transaction item or a component could be of various types (there are a few integral types as well), how does SystemVerilog/UVM handle these operations under the hood?
- All integral types are converted to 4096 bits wide integers
- Even if your field was a mere 8 bit wide, it gets copied into 4096 bits wide parameter of a function

Code your own copy/print/packer functions if you bother about efficiency!
If you have programmed using UVM, you would have come across `uvm_object_utils`.

UVM Object/Component Util macros automagically define utility functions, including:

- bit/byte packing/unpacking
- copy/cloning
- pretty printing

Since the fields of a transaction item or a component could be of various types (there are a few integral types as well), how does SystemVerilog/UVM handle these operations under the hood?

- All integral types are converted to 4096 bits wide integers
- Even if your field was a mere 8 bit wide, it gets copied into 4096 bits wide parameter of a function

Code your own copy/print/packer functions if you bother about efficiency!
Despite the *System* prefix, SystemVerilog does not have features to support System Level verification:

- No low level control over memory or IO resources
- Naive File IO
- Missing library for basic operations like RegExp
- DPI supports interface with C (no support for C++ interface)
Despite the *System* prefix, SystemVerilog does not have features to support System Level verification:

- No low level control over memory or IO resources
- **Naive File IO**
- Missing library for basic operations like RegExp
- DPI supports interface with C (no support for C++ interface)
Despite the *System* prefix, SystemVerilog does not have features to support System Level verification:

- No low level control over memory or IO resources
- Naive File IO
- Missing library for basic operations like RegExp
- DPI supports interface with C (no support for C++ interface)
Despite the _System_ prefix, SystemVerilog does not have features to support System Level verification:

- No low level control over memory or IO resources
- Naive File IO
- Missing library for basic operations like RegExp
- DPI supports interface with C (no support for C++ interface)
SV: Other Missing Features

This list is indicative. Modern programming languages have all these features and more.

- **No Exception Handling**
- **No support for Weak References**
  - For the uninitialized, these are essential to plug some memory leaks
- **No support for data introspection (reflections)**
- **No generative programming (meta-programming)**
- **Compile Time**
This list is indicative. Modern programming languages have all these features and more.

- No Exception Handling
- No support for Weak References
  - For the uninitialized, these are essential to plug some memory leaks
- No support for data introspection (reflections)
- No generative programming (meta-programming)
- Compile Time
This list is indicative. Modern programming languages have all these features and more.

- No Exception Handling
- No support for Weak References
  - For the uninitialized, these are essential to plug some memory leaks
- No support for data introspection (reflections)
- No generative programming (meta-programming)
- Compile Time
SV: Other Missing Features

This list is indicative. Modern programming languages have all these features and more.

- No Exception Handling
- No support for Weak References
  - For the uninitialized, these are essential to plug some memory leaks
- No support for data introspection (reflections)
- No generative programming (meta-programming)
- Compile Time
This list is indicative. Modern programming languages have all these features and more.

- No Exception Handling
- No support for Weak References
  - For the uninitialized, these are essential to plug some memory leaks
- No support for data introspection (reflections)
- No generative programming (meta-programming)
- Compile Time
This list is indicative. Modern programming languages have all these features and more.

- No Exception Handling
- No support for Weak References
  - For the uninitialized, these are essential to plug some memory leaks
- No support for data introspection (reflections)
- No generative programming (meta-programming)
- Compile Time
In a late breaking news, it was discovered that light travels only $3 \times 10^8$ meters in a second.

At 3GHz, that would mean only 10cm in a clock cycle.

CPU frequency is no longer doubling over time.

The Free Lunch Is Over – Herb Sutter

Your EDA vendor might tell you that your favorite simulator is multicore enabled.

But SystemVerilog has little support for concurrent programming.

Nor is UVM designed to work in a concurrent fashion.

With processors scaling to up to 16 cores, there is a lot of CPU power lying wasted.
In a late breaking news, it was discovered that light travels only $3 \times 10^8$ meters in a second.

At $3 \text{GHz}$, that would mean only $10 \text{cm}$ in a clock cycle.

CPU frequency is no longer doubling over time.

*The Free Lunch Is Over* – Herb Sutter

Your EDA vendor might tell you that your favorite simulator is multicore enabled.

But SystemVerilog has little support for concurrent programming.

Nor is UVM designed to work in a concurrent fashion.

With processors scaling to up to 16 cores, there is a lot of CPU power lying wasted.
In a late breaking news, it was discovered that light travels only $3 \times 10^8$ meters in a second.

At 3GHz, that would mean only 10cm in a clock cycle.

CPU frequency is no longer doubling over time.

The Free Lunch Is Over – Herb Sutter

Your EDA vendor might tell you that your favorite simulator is multicore enabled.

But SystemVerilog has little support for concurrent programming.

Nor is UVM designed to work in a concurrent fashion.

With processors scaling to up to 16 cores, there is a lot of CPU power lying wasted.
In a late breaking news, it was discovered that light travels only $3 \times 10^8$ meters in a second.

At 3GHz, that would mean only 10cm in a clock cycle.

CPU frequency is no longer doubling over time.

*The Free Lunch Is Over* – Herb Sutter

Your EDA vendor might tell you that your favorite simulator is multicore enabled.

- But SystemVerilog has little support for concurrent programming.
- Nor is UVM designed to work in a concurrent fashion.
- With processors scaling to up to 16 cores, there is a lot of CPU power lying wasted.
In a late breaking news, it was discovered that light travels only $3 \times 10^8$ meters in a second.

At 3GHz, that would mean only 10cm in a clock cycle.

CPU frequency is no longer doubling over time.

*The Free Lunch Is Over* – Herb Sutter

Your EDA vendor might tell you that your favorite simulator is multicore enabled.

- But SystemVerilog has little support for concurrent programming.
- Nor is UVM designed to work in a concurrent fashion.
- With processors scaling to up to 16 cores, there is a lot of CPU power lying wasted.
In a late breaking news, it was discovered that light travels only $3 \times 10^8$ meters in a second.

At $3\,GHz$, that would mean only 10 cm in a clock cycle.

CPU frequency is no longer doubling over time.

*The Free Lunch Is Over* – Herb Sutter

Your EDA vendor might tell you that your favorite simulator is multicore enabled.

- But SystemVerilog has little support for concurrent programming
- Nor is UVM designed to work in a concurrent fashion
- With processors scaling to up to 16 cores, there is a lot of CPU power lying wasted
In a late breaking news, it was discovered that light travels only $3 \times 10^8$ meters in a second.

At 3GHz, that would mean only 10cm in a clock cycle.

CPU frequency is no longer doubling over time.

*The Free Lunch Is Over* – Herb Sutter

Your EDA vendor might tell you that your favorite simulator is multicore enabled.

- But SystemVerilog has little support for concurrent programming.
- Nor is UVM designed to work in a concurrent fashion.
- With processors scaling to upto 16 cores, there is a lot of CPU power lying wasted.

The Emperor Has No Clothes!
In a late breaking news, it was discovered that light travels only $3 \times 10^8$ meters in a second.

At 3GHz, that would mean only 10cm in a clock cycle.

CPU frequency is no longer doubling over time.

*The Free Lunch Is Over* – Herb Sutter

Your EDA vendor might tell you that your favorite simulator is multicore enabled.

- But SystemVerilog has little support for concurrent programming
- Nor is UVM designed to work in a concurrent fashion
- With processors scaling to up to 16 cores, there is a lot of CPU power lying wasted
Given the limitations of SystemVerilog, there is a lot of demand for Multilingual UVM

- Multiple proposals have been discussed
- But ultimately the VIP Task Committee decided to can it
- An alternative being offered is SystemVerilog API to connect with SystemC
Given the limitations of SystemVerilog, there is a lot of demand for Multilingual UVM

- Multiple proposals have been discussed
- But ultimately the VIP Task Committee decided to can it
- An alternative being offered is SystemVerilog API to connect with SystemC
Given the limitations of SystemVerilog, there is a lot of demand for Multilingual UVM
Multiple proposals have been discussed
But ultimately the VIP Task Committee decided to can it
An alternative being offered is SystemVerilog API to connect with SystemC
Given the limitations of SystemVerilog, there is a lot of demand for Multilingual UVM

Multiple proposals have been discussed

But ultimately the VIP Task Committee decided to can it

An alternative being offered is SystemVerilog API to connect with SystemC
The OpenSource Revolution

EDA | CMOS | Verilog | SystemVerilog
---|---|---|---
C | C++ | GNU | Java | DSLs multicore
4004 | 8086 | i486 | OpenSource Hardware
8085 | i386 | pentium | Atom/ARM

OpenSource Verification?
Introducing VLang

System Level
- DPI
- UVM
- SystemVerilog
- Verilog

System Level
- UVM
- VLang
- PLI
- Verilog

The Emperor Has No Clothes!
Why VLang

- Standards are good, but...
- Concurrency
- Function and operator overloading
- C++ Interface (please note that SystemVerilog provides for DPI which provides for only C interface)
- Reflections (code introspection) and metaprogramming (code generation) capabilities
- A sophisticated Unit Test approach (self checking testbenches)
- A sophisticated generic algorithmic library
- Based on D – a powerful evolution of C++
- OpenSource – Ideal for SystemC Verification
Why VLang

- Standards are good, but . . .
- Concurrency
  - Function and operator overloading
  - C++ Interface (please note that SystemVerilog provides for DPI which provides for only C interface)
  - Reflections (code introspection) and metaprogramming (code generation) capabilities
  - A sophisticated Unit Test approach (self checking testbenches)
  - A sophisticated generic algorithmic library
  - Based on D – a powerful evolution of C++
  - OpenSource – Ideal for SystemC Verification

The Emperor Has No Clothes!
Why VLang

- Standards are good, but...
- Concurrency
- Function and operator overloading
- C++ Interface (please note that SystemVerilog provides for DPI which provides for only C interface)
- Reflections (code introspection) and metaprogramming (code generation) capabilities
- A sophisticated Unit Test approach (self checking testbenches)
- A sophisticated generic algorithmic library
- Based on D – a powerful evolution of C++
- OpenSource – Ideal for SystemC Verification
Why VLang

- Standards are good, but…
- Concurrency
- Function and operator overloading
- C++ Interface (please note that SystemVerilog provides for DPI which provides for only C interface)
- Reflections (code introspection) and metaprogramming (code generation) capabilities
- A sophisticated Unit Test approach (self checking testbenches)
- A sophisticated generic algorithmic library
- Based on D – a powerful evolution of C++
- OpenSource – Ideal for SystemC Verification
Why VLang

- Standards are good, but...
- Concurrency
- Function and operator overloading
- C++ Interface (please note that SystemVerilog provides for DPI which provides for only C interface)
- Reflections (code introspection) and metaprogramming (code generation) capabilities
- A sophisticated Unit Test approach (self checking testbenches)
- A sophisticated generic algorithmic library
- Based on D – a powerful evolution of C++
- OpenSource – Ideal for SystemC Verification
Why VLang

- Standards are good, but...
- Concurrency
- Function and operator overloading
- C++ Interface (please note that SystemVerilog provides for DPI which provides for only C interface)
- Reflections (code introspection) and metaprogramming (code generation) capabilities
- A sophisticated Unit Test approach (self checking testbenches)
- A sophisticated generic algorithmic library
- Based on D – a powerful evolution of C++
- OpenSource – Ideal for SystemC Verification

The Emperor Has No Clothes!
Why VLang

- Standards are good, but...
- Concurrency
- Function and operator overloading
- C++ Interface (please note that SystemVerilog provides for DPI which provides for only C interface)
- Reflections (code introspection) and metaprogramming (code generation) capabilities
- A sophisticated Unit Test approach (self checking testbenches)
- A sophisticated generic algorithmic library
- Based on D – a powerful evolution of C++
- OpenSource – Ideal for SystemC Verification
Why VLang

- Standards are good, but . . .
- Concurrency
- Function and operator overloading
- C++ Interface (please note that SystemVerilog provides for DPI which provides for only C interface)
- Reflections (code introspection) and metaprogramming (code generation) capabilities
- A sophisticated Unit Test approach (self checking testbenches)
- A sophisticated generic algorithmic library
- Based on D – a powerful evolution of C++
- OpenSource – Ideal for SystemC Verification
Why VLang

- Standards are good, but...
- Concurrency
- Function and operator overloading
- C++ Interface (please note that SystemVerilog provides for DPI which provides for only C interface)
- Reflections (code introspection) and metaprogramming (code generation) capabilities
- A sophisticated Unit Test approach (self checking testbenches)
- A sophisticated generic algorithmic library
- Based on D – a powerful evolution of C++
- OpenSource – Ideal for SystemC Verification
VLang Status

- Basic Constraints
  - Complete port of UVM 1.2
  - Testing in progress – Objection/Phases/Sequencer work
  - Development release pending
    - Waiting for version 2.064 release of D – the base language
    - Contact me if you want to have sneak peek
VLang Status

- Basic Constraints
- Complete port of UVM 1.2
- Testing in progress – Objection/Phases/Sequencer work
- Development release pending
  - Waiting for version 2.064 release of D – the base language
  - Contact me if you want to have sneak peek
VLang Status

- Basic Constraints
- Complete port of UVM 1.2
- Testing in progress – Objection/Phases/Sequencer work
- Development release pending
  - Waiting for version 2.064 release of D – the base language
  - Contact me if you want to have sneak peek
VLang Status

- Basic Constraints
- Complete port of UVM 1.2
- Testing in progress – Objection/Phases/Sequencer work
- Development release pending
  - Waiting for version 2.064 release of D – the base language
  - Contact me if you want to have sneak peek
VLang Status

- Basic Constraints
- Complete port of UVM 1.2
- Testing in progress – Objection/Phases/Sequencer work
- Development release pending
  - Waiting for version 2.064 release of D – the base language
  - Contact me if you want to have sneak peek
VLang Status

- Basic Constraints
- Complete port of UVM 1.2
- Testing in progress – Objection/Phases/Sequencer work
- Development release pending
  - Waiting for version 2.064 release of D – the base language
  - Contact me if you want to have sneak peek
Thank You!