DVCon 2012
Design & Verification Conference & Exhibition

Easier SystemVerilog with UVM: Taming the Beast

John Aynsley, Doulos

DOULOS

SystemVerilog

The language of choice

because it's a ~~standard~~

supported by all tool vendors

SystemVerilog

Large and complex

SystemVerilog

Not simple,
orthogonal,
or consistent

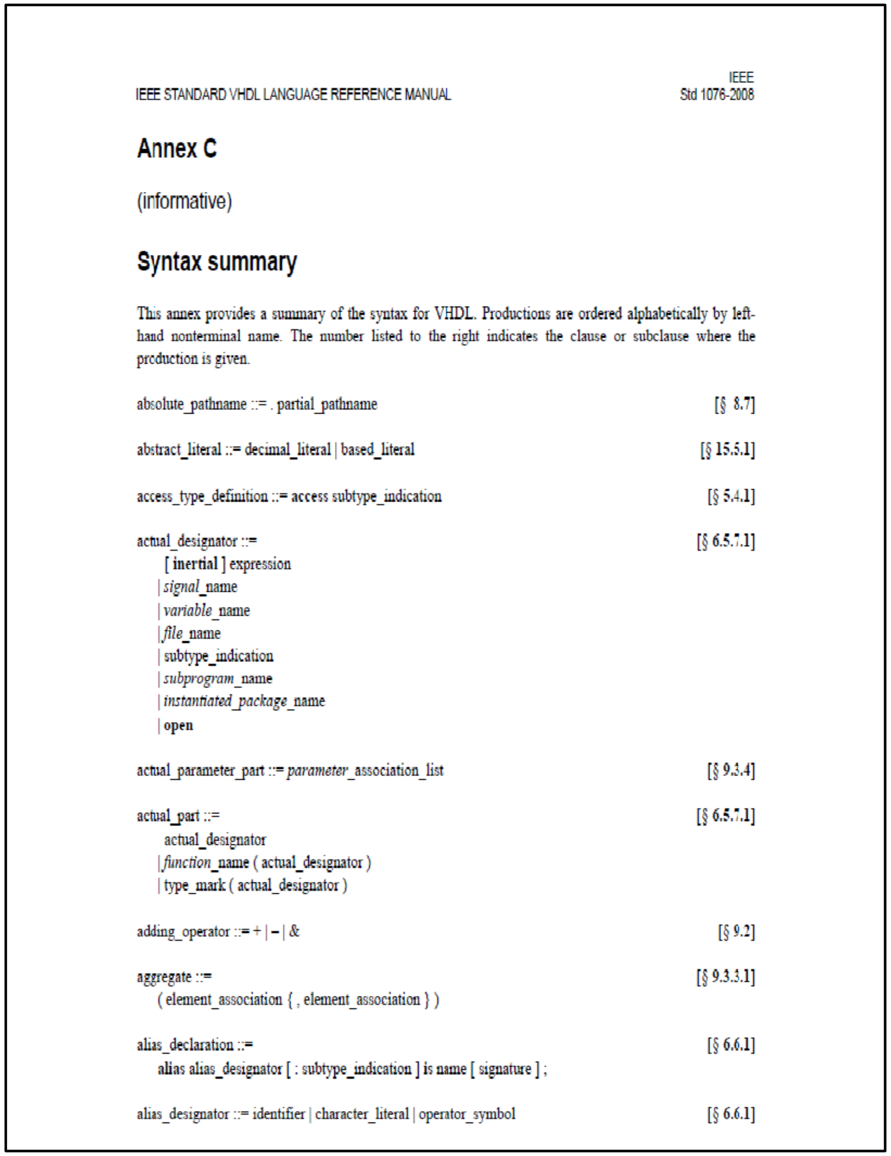
Several Languages in One?

- Includes features from
  - Verilog
  - VHDL
  - PSL
  - Superlog
  - OpenVera
  - C
  - C++
  - Java

Differences between implementations

and don't blame the vendors

Syntax Definition - VHDL



IEEE STANDARD VHDL LANGUAGE REFERENCE MANUAL — IEEE Std 1076-2008

Annex C

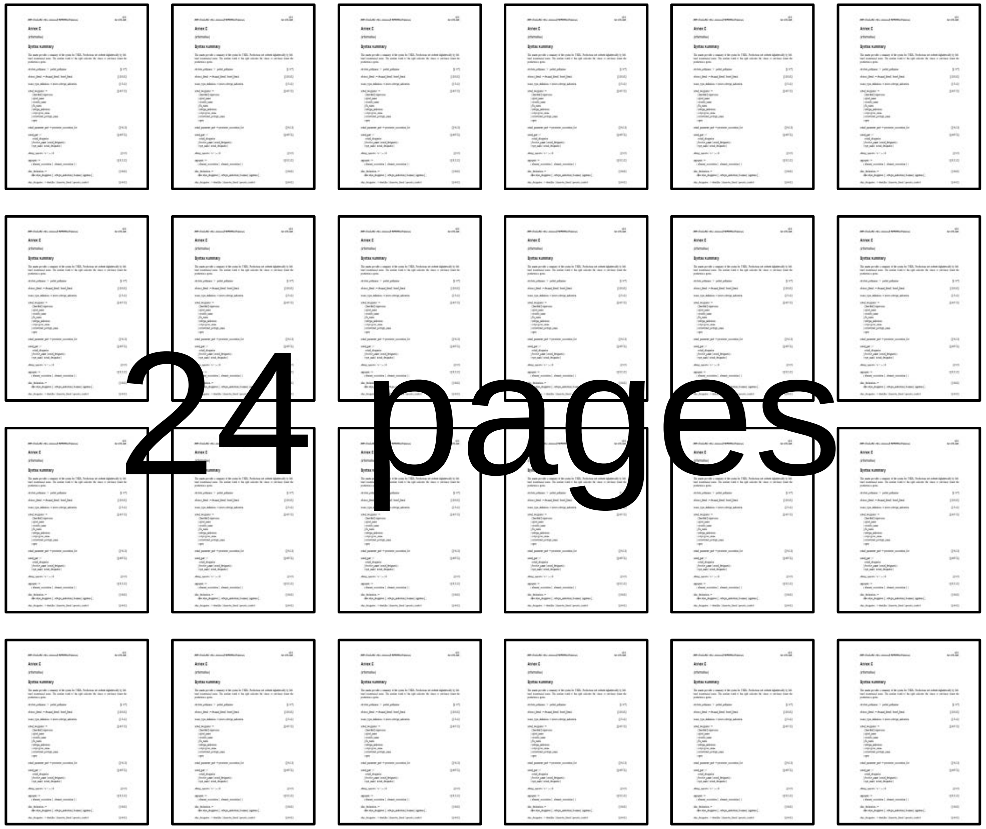
(informative)

Syntax summary

This annex provides a summary of the syntax for VHDL. Productions are ordered alphabetically by left-hand nonterminal name. The number listed to the right indicates the clause or subclause where the production is given.

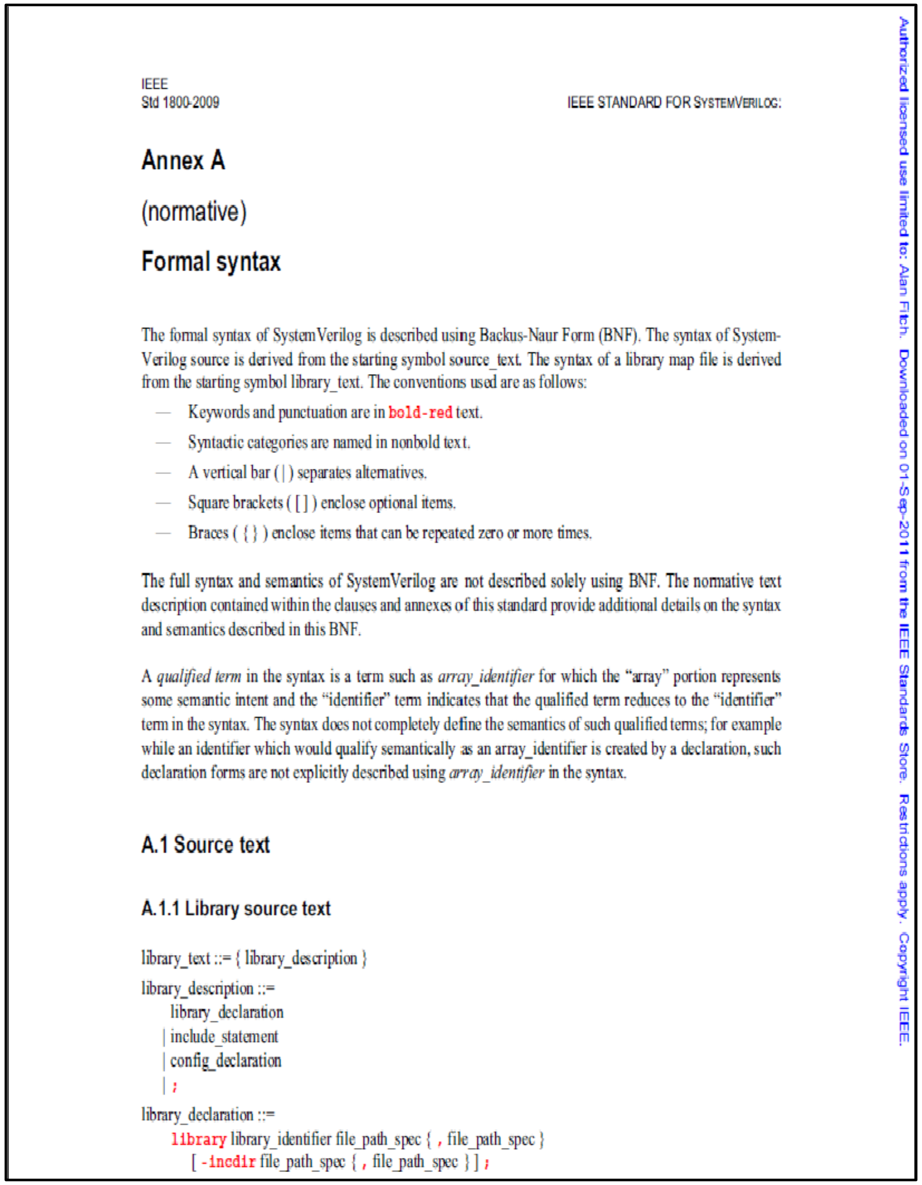
```
absolute_pathname ::= . partial_pathname                              [§ 8.7]

abstract_literal ::= decimal_literal | based_literal                  [§ 15.5.1]

access_type_definition ::= access subtype_indication                  [§ 5.4.1]

actual_designator ::=                                                 [§ 6.5.7.1]
      [ inertial ] expression
    | signal_name
    | variable_name
    | file_name
    | subtype_indication
    | subprogram_name
    | instantiated_package_name
    | open

actual_parameter_part ::= parameter_association_list                  [§ 9.3.4]

actual_part ::=                                                       [§ 6.5.7.1]
      actual_designator
    | function_name ( actual_designator )
    | type_mark ( actual_designator )

adding_operator ::= + | – | &                                         [§ 9.2]

aggregate ::=                                                         [§ 9.3.3.1]
    ( element_association { , element_association } )

alias_declaration ::=                                                 [§ 6.6.1]
    alias alias_designator [ : subtype_indication ] is name [ signature ] ;

alias_designator ::= identifier | character_literal | operator_symbol [§ 6.6.1]
```

Syntax Definition - VHDL



24 pages

Syntax Definition - SystemVerilog



IEEE
Std 1800-2009 IEEE STANDARD FOR SYSTEMVERILOG:

Annex A

(normative)

Formal syntax

The formal syntax of SystemVerilog is described using Backus-Naur Form (BNF). The syntax of SystemVerilog source is derived from the starting symbol source_text. The syntax of a library map file is derived from the starting symbol library_text. The conventions used are as follows:

- Keywords and punctuation are in **bold-red** text.
- Syntactic categories are named in nonbold text.
- A vertical bar (|) separates alternatives.
- Square brackets ([ ]) enclose optional items.
- Braces ({ }) enclose items that can be repeated zero or more times.

The full syntax and semantics of SystemVerilog are not described solely using BNF. The normative text description contained within the clauses and annexes of this standard provide additional details on the syntax and semantics described in this BNF.

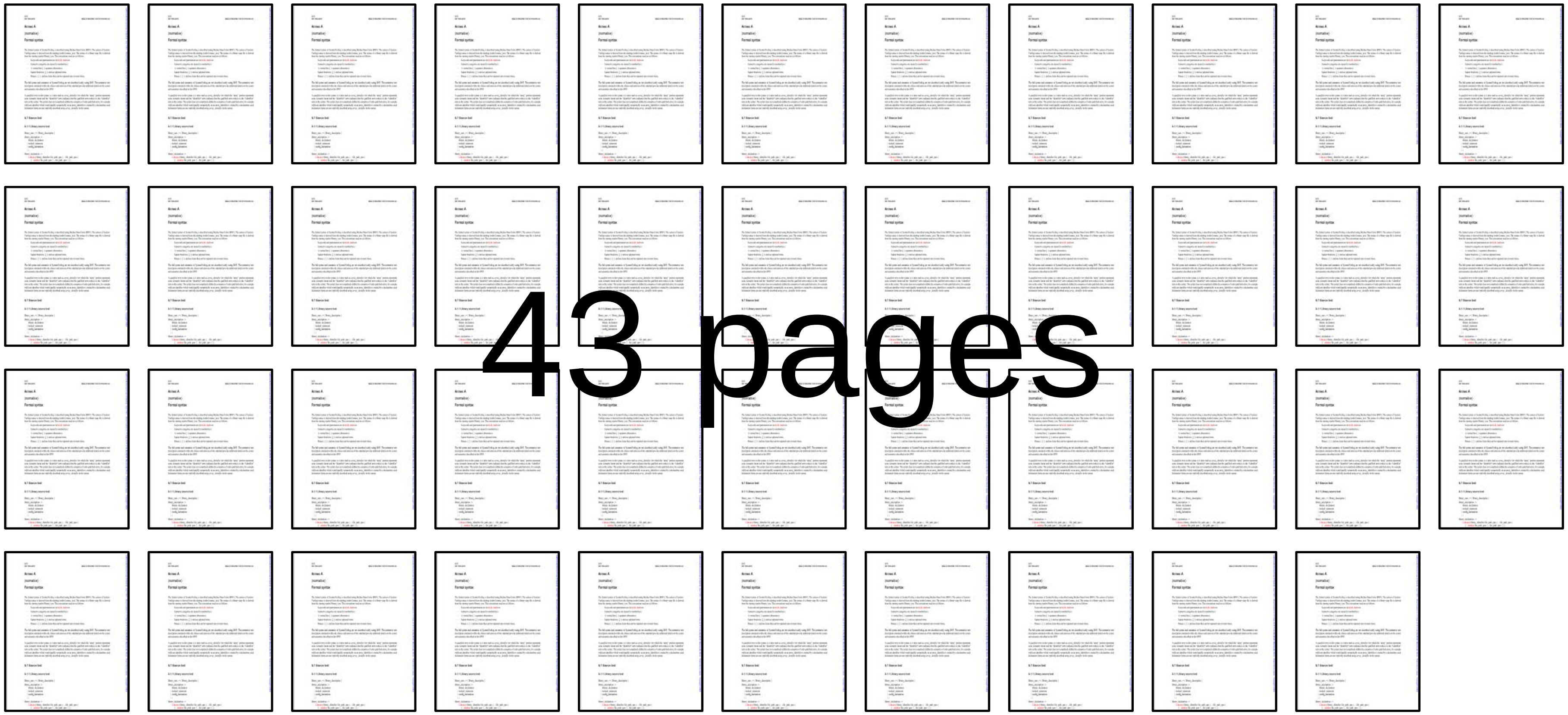
A *qualified term* in the syntax is a term such as *array_identifier* for which the "array" portion represents some semantic intent and the "identifier" term indicates that the qualified term reduces to the "identifier" term in the syntax. The syntax does not completely define the semantics of such qualified terms; for example while an identifier which would qualify semantically as an array_identifier is created by a declaration, such declaration forms are not explicitly described using *array_identifier* in the syntax.

A.1 Source text

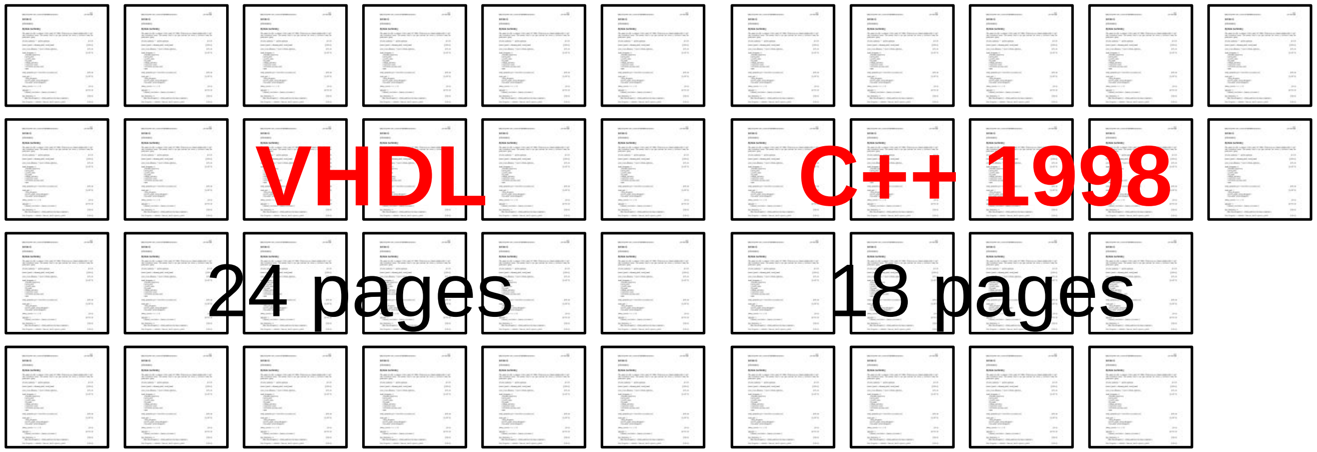
A.1.1 Library source text

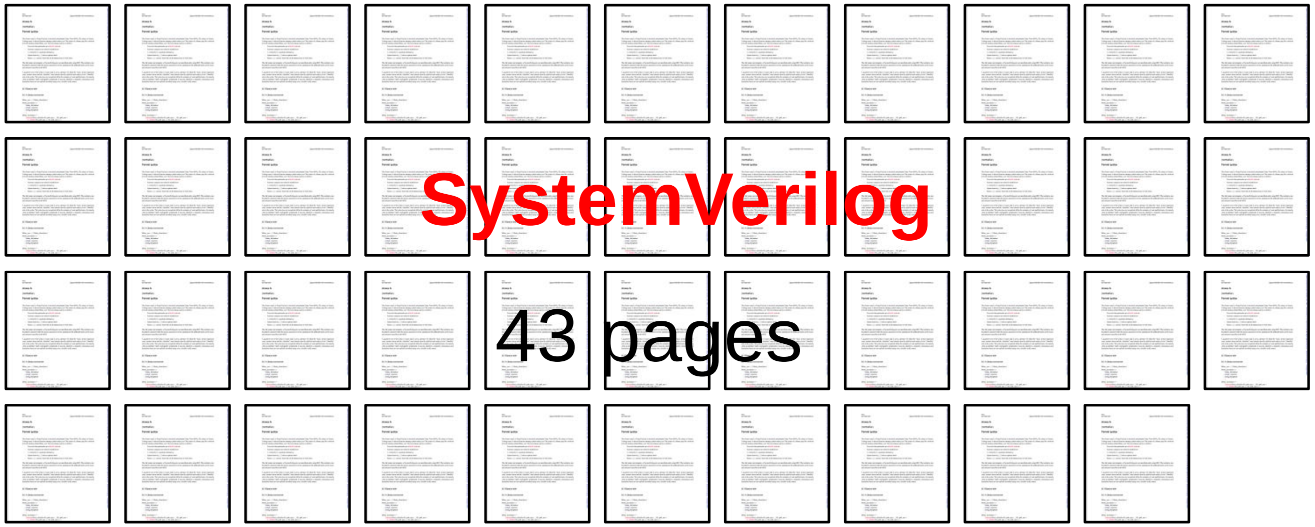
```
library_text ::= { library_description }
library_description ::=
    library_declaration
  | include_statement
  | config_declaration
  | ;
library_declaration ::=
    library library_identifier file_path_spec { , file_path_spec }
      [ -incdir file_path_spec { , file_path_spec } ] ;
```

Syntax Definition - SystemVerilog

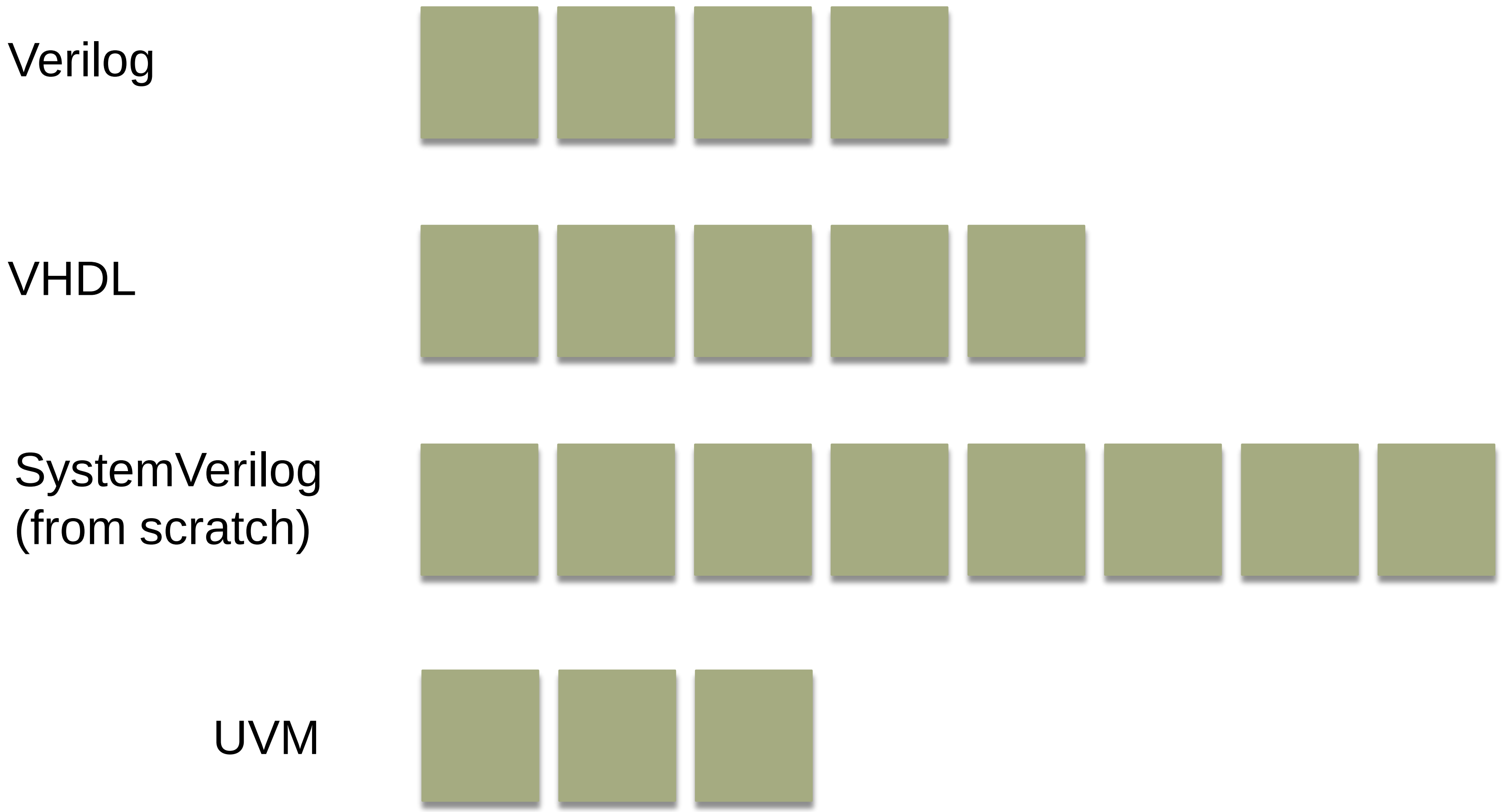


Syntax Definition





Hard to Learn



Enter UVM

Taming
the Beast

UVM

- Enables verification IP reuse & captures best practice
- Supported by all major vendors
- Increased confidence to adopt SystemVerilog

- Implementations now mutually consistent
- Still need to avoid remaining pitfalls...

DVCon 2012
Design & Verification Conference & Exhibition

Easier SystemVerilog Approach

- Do use
  - Verilog
  - Concise RTL, for hardware synthesis
  - Features used by UVM – OOP and C-like
  - Features for constrained random verification
  - Features for interfacing test bench to DUT
- Don't use
  - Features which are not portable

Used by the UVM BCL

- Packages
- All the class syntax (almost)
- typedef, 2-state types, enums, (some) structs *in classes*
- Strings, queues, associative and dynamic arrays
- C-like procedural statements *in methods*
- fork-join *in methods*

Idioms from the UVM BCL

```
class C #(type T = int) extends BASE #(T);
```

Idioms from the UVM BCL

```
begin

  classname::typename::method();

  classname #(typename)::method();

end
```

Design & Verification Conference & Exhibition

Idioms from the UVM BCL

```
if ($cast(to_handle, from_handle))
  ...
```

Idioms from the UVM BCL

```
begin
  static string blank = "";
  ...
end


begin
  classname q[$];
  ...
end
```

Design & Verification Conference & Exhibition

Idioms from the UVM BCL

```
void'( obj.method() );
```

Idioms from the UVM BCL

```
typedef enum bit { lit1 = 0, lit2 = 1 } name;
```

Idioms from the UVM BCL

```
string S;

if (S == "")

  S = {"pre", S.substr(expr1, expr2)};
```

Idioms from the UVM BCL

```
for (int i = 0; i < n; i++)
   ...
```

Idioms from the UVM BCL

```
foreach (array[i])
   ...
```

Design & Verification Conference & Exhibition

Idioms from the UVM BCL

```
-> assoc_array[index].named_event;
```

Idioms from the UVM BCL

```
function void f (
      ref   uvm_component comps[$],
      input uvm_component comp = null,
      string arg = "");
```

Used by UVM Applications

- interface
- clocking
- modport
- virtual interface
- Handle in module scope

Module-class communication

- assert
- covergroup
- rand member
- randomize() with { ... }
- constraint

Constrained random

- Array manipulation methods
- Verilog!

Pitfalls

Pitfalls

- Many features robust and portable
- Remaining pitfalls cannot be simply described
- UVM BCL side-steps some pitfalls
- List of pitfalls is now short and getting shorter!
- Areas where the vendors have not converged?

Continuous Assignment to Handle

```
class C;
  ...
endclass


module top;

C handle1 = new;
C handle2;

assign handle2 = handle1;
```

Handles as Ports

```
class C;
  ...
endclass

module child1 (input C p, output C q);
...

module child2 (ref C p,   ref C q);
```

Hierarchical Reference to Handle

```
class C;
  ...
endclass

module top;

  modu inst ();

  initial
    inst.handle = new;

endmodule

module modu;
  C handle;
endmodule
```

Hierarchical Reference to Parameter

```
interface iface;
  parameter int p = 8;
endinterface

module top;
  iface iinst();

  bit [iinst.p-1:0] vec;
```

Design & Verification Conference & Exhibition

Objects as Struct Members

```
struct {
  byte a[];
} s1, s2;
```

```
initial
begin
  s1.a = '{1, 2, 3, 4};

  s2 = s1;

  s1.a[0] = 5;

  $display(s2.a[0]);
end
```

1 or 5?

Unpacked Unions

```
typedef struct
{
  bit a;
  byte b;
} T1;

typedef struct
{
  byte c;
  bit d;
} T2;
```

```
typedef union
{
  T1 p;
  T2 q;
} U;
```

Bitstream Casting

```
typedef struct
{
  bit a;
  byte b;
} T1;

typedef struct
{
  byte c;
  bit d;
} T2;
```

```
module top;

T1 s1;
T2 s2;

initial
begin
  s1 = '{1, 1};

  s2 = T2'(s1);
```

Type Parameter Substitution

```
class C #(type T = ...);
  typedef T::T2 T3;

  static const int d = T::c;
endclass


class D;
  typedef C #(C1) T;
  static const int e = T::d;
```

Protected Constructor

```
class C;

  protected function new;
    ...
  endfunction
```

Array-to-Queue Assignment

```
begin
  int da[];
  int q[$];
  da = '{1, 2, 3, 4};
  q = da;
```

```
`define UVM_DA_TO_QUEUE(Q,DA)\
   foreach (DA[idx]) Q.push_back(DA[idx]);
```

Iterator Index Query

```
int a[];
int q[$];
a = '{0, 3, 2, 1, 4, 5, 7, 8};

q = a.find with ( item == item.index );
```

Statement Labels

```
initial

blk: begin

  loop: repeat (8);

end: blk
```

Assignment as Side Effect

```
int i;

if ((i = 99))
  $display("i is 99");
```

final

```
final
  $display("The End at ", $time);
```

wait fork

```
begin
  fork
    #44;
    fork
      #125;
      #14;
    join_none
    #2;
  join_none

  wait fork;

  $display($time);
end
```

44 or 125?

$get_coverage

```
initial
  $display( $get_coverage );
```

Empty Coverpoint

```
rand longint data;


covergroup cg;

  coverpoint data {}

endgroup
```

std::randomize

```
begin

  byte unsigned a, b, c;

  assert( randomize(a, b, c) );


  assert( std::randomize(a, b, c) );
```

Prototype in modport

```
modport mp (import function void hello());
```

```
modport mp (import hello);
```

Themes?

- Keep modules and classes separate?
- No need for unpacked structs, unions, or arrays

  (other than Verilog memories)?

Conclusions

- Do use
  - Verilog!
  - Synthesis-friendly RTL features
  - Classes (from UVM)
  - C-like features: data types and statements (from UVM)
  - Interfaces, virtual interfaces, clocking blocks
  - Assertions, coverage, constraints

UVM has tamed the beast!