Is Your Testing N-wise or Unwise?

Pairwise and N-wise Patterns in SystemVerilog for Efficient Test Configuration and Stimulus

Kevin Johnston
Verilab, Inc

September 18th, 2015
SNUG Austin

Agenda

Introduction to pairwise/N-wise test generation

Using our SystemVerilog N-wise implementation

Performance concerns, additional features

Application suggestions

Wrap-up

Introduction to Pairwise and N-wise

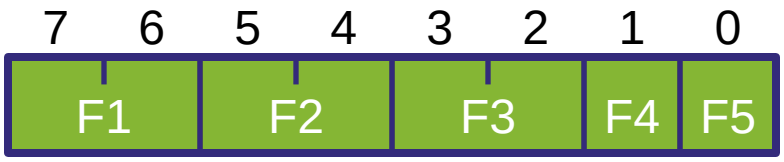
- Testing can never be exhaustive
  - too much state, too many combinations
- Software testing has used *pairwise* for many years
  - an attempt to focus our finite testing effort as effectively as possible
  - complements constrained-random – does not replace it!
- Basic principle: don't try to test all possible combinations of all parameters, but instead

> for every *pair* of parameters,
> test every combination of that pair

- Picks up any bug that is caused when two parameters have specific values
  - usually, with less testing effort than constrained-random

Configurations Must Be Tested

- 8-bit configuration register for a simple DUT:



- F3 has only 3 legal values
- **192** values to be tested!

- Each configuration should be tested thoroughly
  - ideally, full coverage on each
- Not enough project time!
- In practice, only a few important (lead customer?) configurations will be tested fully

Major concern for IP authors/vendors

Why Pairwise?

- Testing every parameter value is easy but not so useful
  - only 4 tests cover every value of each parameter in isolation

F1	F2	F3	F4	F5
00	00	00	0	0
01	01	01	1	1
10	10	10	0	0
11	11	00	1	1

Observation: Bugs are often triggered by interaction of *two* values

- Test **every** combination of **pairs**

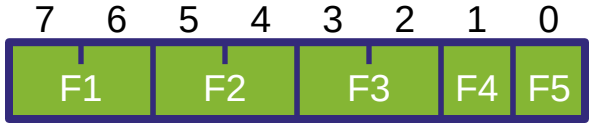
F1×F2	**16**						
F1×F3	**12**	F2×F3	**12**				
F1×F4	**8**	F2×F4	**8**	F3×F4	**6**		
F1×F5	**8**	F2×F5	**8**	F3×F5	**6**	F4×F5	**4**

parameter pair F2,F3 has 12 possible values

parameter pair F4,F5 has 4 possible values

Total of **88** value-pairs

Getting the Problem Under Control



192 possible configurations

Pairwise: thorough testing with only **16** configurations

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F1	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
F2	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
F3	0	1	2	1	2	0	1	0	1	2	0	2	0	1	2	1
F4	0	1	1	0	0	1	0	1	1	0	0	0	0	1	0	1
F5	0	1	0	0	1	0	1	1	0	1	1	0	0	1	0	1

- Probably the best we can do with that number of tests
- Widely applicable:
  - configuration
  - parameterization
  - stimulus scenarios

Complete coverage of F2xF3

Complete covera

Larger example

- 20 parameters, each with 10 values
- 10^{20} possible configurations
- Pairwise covered by **230** tests
- Generator runtime ~20 seconds

Using our N-wise Implementation

- Pure SystemVerilog
  - fits easily into typical verification flow
  - compatible with UVM and other methodologies
  - uses SV constraint language for flexibility
- Minimal restriction on your coding style
- Small overhead if you add the code but don't use it

- Development continues, but the current implementation is already useful
  - freely available from www.verilab.com

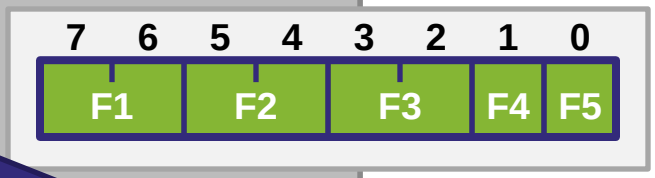
Code for Register Example

- Minimal coding overhead in user's configuration object

```
`include "nwise.svh"

typedef enum {P_NONE, P_ODD, P_EVEN} parity_e;

class Config extends
      Nwise_base#(uvm_sequence_item);
  ...
  `NWISE_BEGIN(Config)
  `NWISE_VAR_INT(  int,      F1, {[0:3]} )
  `NWISE_VAR_INT(  bit[1:0], F2 )
  `NWISE_VAR_ENUM( parity_e, F3 )
  `NWISE_VAR_INT(  bit,      F4 )
  `NWISE_VAR_INT(  bit,      F5 )

  string name;
  function new(...); ...
endclass
```

Mixin supports UVM base classes if required

Value-set required for wide types




```
rand int F1;
constraint c_NWISE_F1 {
  F1 inside {[0:3]};
  F1 == __nwise_value_proxy[...];
}
```

Macro-generated code (simplified)

Other user code OK

Using the N-wise Enabled Class

- Call methods of class's generator object to get patterns

```
class Config extends Nwise_base#
  ...
  `NWISE_BEGIN(Config)
  `N
  `N
```

```
Config cfg = new(...);
int num_patterns = cfg.nwise.generate_patterns(2);
for (int p = 0; p < num_patterns; p++) begin
  cfg.nwise.render_pattern(p);
  $display("cfg %2d = %b,%b,%s,%b,%b",
           p, cfg.F1, cfg.F2, cfg.F3.name, cfg.F4, cfg.F5);
end
```

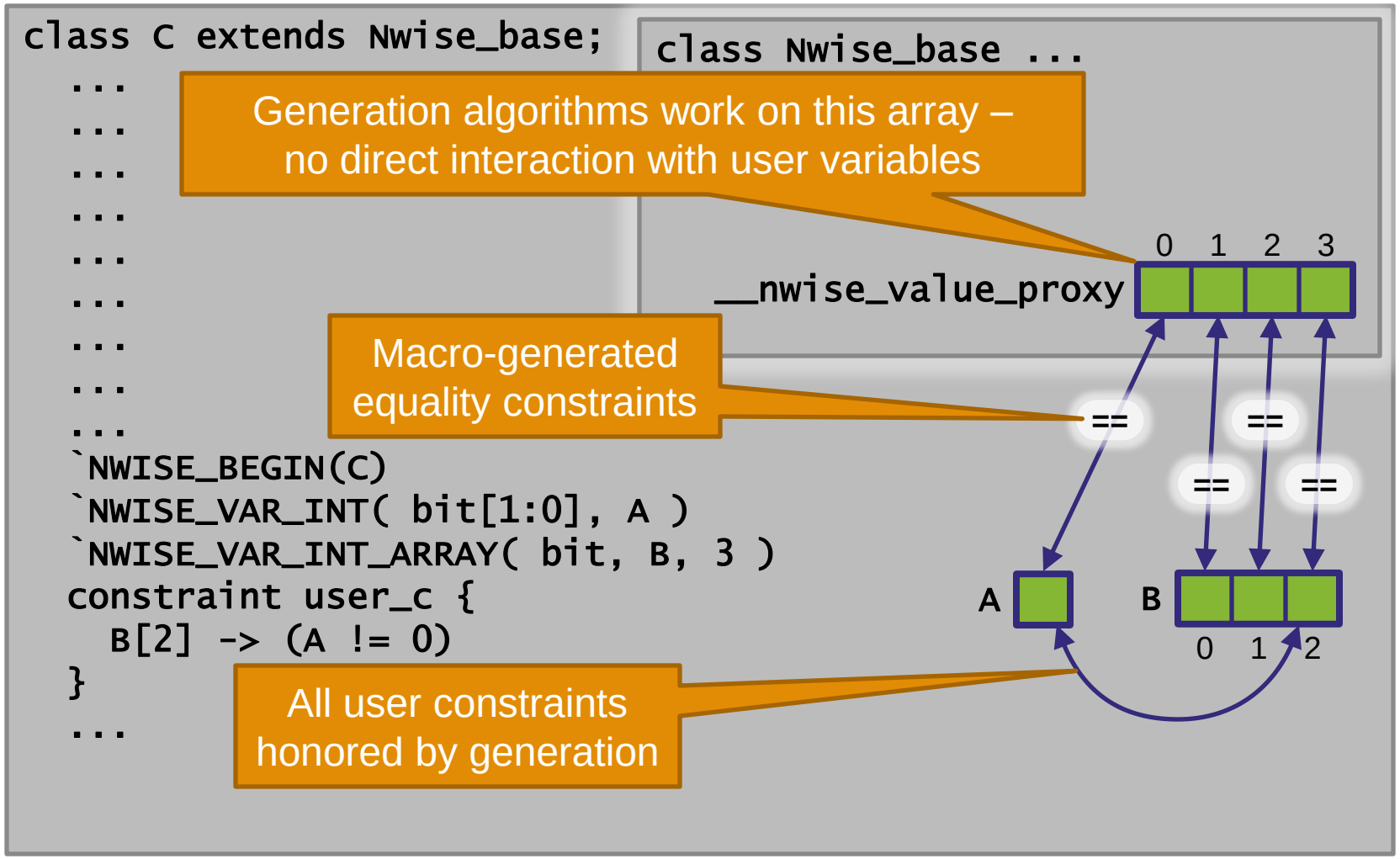
Generator object

Generation order:
2 for pairwise

```
cfg  0 = 00,00,P_NONE,0,0
cfg  1 = 01,00,P_ODD,1,1
cfg  2 = 10,00,P_EVEN,1,0
```

- **`render_pattern(p)`**:
  Change this object's contents to match **`pattern[p]`**
- **`render_pattern_new(p)`**:
  Make a **new** object with contents matching **`pattern[p]`**

Under the Hood



Performance

- Pairwise efficiency (test compression)
  - currently within ~15%-30% of best available tools, ~10% of PICT

Problem space		Number of patterns for pairwise coverage		
cardinality	*params*	***our tool***	*PICT*	*best available*
2	100	16	15	10
3	[illegible]	20	18	15
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

data from ref [2] in our paper

Runtime on VCS H-2013.06
(64-bit Linux, modest x86 server)

- insignificant on small problems
- for N-wise, scales as $cardinality^{2N} \times params^{4}$
- ~7 seconds for 2^{100} example

Additional Features, Future Work

- API improvements
  - specify required patterns to include in the set
    - e.g. key early customer configuration
  - higher-order interactions for selected parameters
    - e.g. pairwise, with 3-wise for some set of critical parameters
  - incremental pattern generation for fast startup

Details will be influenced by real-world experience

- Ongoing work on performance, capacity
- Support variable-size arrays

Application Suggestions

- Alternative to randomization for UVM configuration objects
  - choice of TB options, selection of tests, register settings...
- Provide successive values for a DUT control register
  - especially useful when each register setup needs extensive testing
- Choose build parameters for configurable IP
  - always requires a 2-pass approach
    (module parameters must be static)
- Configuration of embedded software to run on DUT

Time to Play!

- Get the code and user notes
  - Apache open-source license, just like the UVM code

www.verilab.com/resources

- Check out other available implementations
  - see the references in our paper
  - compare
    - usability
    - convenience
    - performance
    - compatibility with your existing flow

verilab

snug
Synopsys Users Group

Thank You