

Is Your Testing N-wise or Unwise?

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

Jonathan Bromley, Kevin Johnston

verilab



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

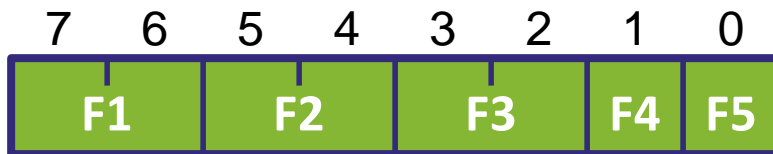
- Testing can never be exhaustive
 - too much state, too many combinations
- Software testing has used pairwise for many years
 - an attempt to focus our finite testing effort as effectively as possible
 - complements constrained-random – does not replace it!
- Basic principle: don't try to test all 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 configurations tested fully
 - choose important configs (lead customer?)

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×F1	Total of 88 value-pairs			F4×F4
F1×F2 16	F2×F2	F3×F2	F4×F2	F5×F2
F1×F3 12	F2×F3 12	F3×F3	F4×F3	F5×F3
F1×F4 8	F2×F4 8	F3×F4 6	F4×F4	F5×F4
F1×F5 8	F2×F5 8	F3×F5 6	F4×F5 4	F5×F5

parameter pair F2,F3:
12 possible values

parameter pair F4,F5:
4 possible values

Getting the Problem Under Control



192 possible configurations

Pairwise: thorough testing with only 16 configurations

1	0	0	0	0	0
2	1	0	1	1	1
3	2	0	2	1	0
4	3	0	1	0	0
5	0	1	2	0	1
6	1	1	0	1	0
7	2	1	1	0	1
8	3	1	0	1	1
9	0	2	1	1	0
10	1	2	2	0	1
11	2	2	0	0	1
12	3	2	2	0	0
13	0	3	0	0	0
14	1	3	1	1	1
15	2	3	2	0	0
16	3	3	1	1	1

• Probably the best we can do with that number of tests

complete coverage of F4xF5

• Widely applicable:

complete coverage of F2xF3

Larger example

- 20 parameters, each with 10 values
- 10^{20} possible configurations
- Pairwise covered by 230 tests
- Generator runtime ~10 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 it's already useful
 - freely available from www.verilab.com

Code for Register Example

- Minimal coding overhead in user's configuration

`#include "nwise.svh"`

```
typedef enum {P_NONE, P_ODD, P_EVEN} parity;

class Config extends Nwise_base#(uvm_sequence_item);
```

```
...
`NWISE_BEGIN(Config)
`NWISE_VAR_INT(int, F1, {[0:3]})
`NWISE_VAR_INT(int[1:0], F2)
`NWISE_VAR_ENUM(parity, F3)
`NWISE_VAR_INT(bit, F4)
`NWISE_VAR_INT(bit,
```

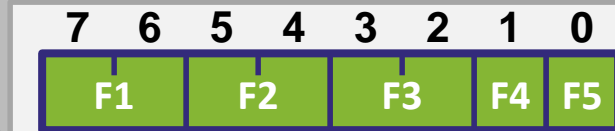
```
string name;
function new(...);
```

Other user code OK

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

Mixin supports UVM base classes if required

Value-set required for wide types



Macro-generated code (simplified)

Using the N-wise Enabled Class

- Call methods of Nwise_base to get patterns

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

Generation order:
2 for pairwise

```
...  
`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
```

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

- **Nwise_render_pattern(p):**
Change this object's contents to match **pattern[p]**

Under the Hood

```
class C extends Nwise_base; class Nwise_base ...
```

Generation algorithms work on this array –
no direct interaction with user variables

__nwise_value_proxy

0	1	2	3
█	█	█	█

Macro-generated
equality constraints

==

==

==

==

```
`NWISE_BEGIN(C)  
`NWISE_VAR_INT( bit[1:0], A )  
`NWISE_VAR_INT_ARRAY( bit, B, 3 )  
constraint user_c {  
  B[2] -> (A != 0)  
}  
...
```

A

B

0 1 2

All user constraints
honored by generation

Performance

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

Problem space		Number of patterns for pairwise coverage		
<i>cardinality</i>	<i>params</i>	<i>our tool</i>	<i>PICT</i>	<i>best available</i>
2	100	16	15	10
3	13	20	18	15
10		230	210	180

Runtime on 64-bit Linux, modest x86 server:

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

data from ref [2] in our paper

Additional Features, Future Work

- API improvements
 - include required patterns in the set
 - e.g. key early customer configuration
 - higher-order Nwise for selected parameters
 - e.g. pairwise, but 3-wise for some critical params
 - incremental pattern generation for fast startup
- Ongoing work on performance, capacity
- Support variable-size arrays

Details will be influenced by real-world experience

Application Suggestions

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

Time to Play!

- Get code and user notes

www.verilab.com/resources

- Apache open-source license,
just like the UVM code

- Check out other available implementations

- see references in our paper

- compare:

- usability
- convenience
- performance
- compatibility with your existing flow

Thank You!

Any questions?

