Is Your Testing N-wise or Unwise?
Pairwise and N-wise Patterns in SystemVerilog for Efficient Test Configuration and Stimulus

Jonathan Bromley, Kevin Johnston
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:
  
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>F4</td>
<td>F5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  • 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

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

- Test every combination of pairs

<table>
<thead>
<tr>
<th>F1×F1</th>
<th>F1×F2</th>
<th>F1×F3</th>
<th>F1×F4</th>
<th>F1×F5</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>12</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

**Total of 88 value-pairs**

<table>
<thead>
<tr>
<th>F2×F1</th>
<th>F2×F2</th>
<th>F2×F3</th>
<th>F2×F4</th>
<th>F2×F5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F3×F1</th>
<th>F3×F2</th>
<th>F3×F3</th>
<th>F3×F4</th>
<th>F3×F5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F4×F1</th>
<th>F4×F2</th>
<th>F4×F3</th>
<th>F4×F4</th>
<th>F4×F5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F5×F1</th>
<th>F5×F2</th>
<th>F5×F3</th>
<th>F5×F4</th>
<th>F5×F5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Parameter pair F2,F3:** 12 possible values

**Parameter pair F4,F5:** 4 possible values

© Accellera Systems Initiative
Getting the Problem Under Control

- Pairwise: thorough testing with only 16 configurations
- Probably the best we can do with that number of tests
- Widely applicable:
  - complete coverage of F4xF5
  - 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

© Accellera Systems Initiative
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
typedef enum {P_NONE, P_ODD, P_EVEN} parity_e;

class Config extends uvm_sequence_item;
...
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)
Other user code OK

© Accellera Systems Initiative
Using the N-wise Enabled Class

• Call methods of Nwise_base to get patterns

```java
class Config extends Nwise_base{

  \NWISE_BEGIN(Config)
  \NWISE_VAR_INT( int, F1, {0:3} )
  \NWISE_VAR_INT( bit[1:0], F2 )

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
```

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

Generation order: 2 for pairwise

© Accellera Systems Initiative
Under the Hood

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

Macro-generated equality constraints

All user constraints honored by generation
Performance

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

<table>
<thead>
<tr>
<th>Problem space</th>
<th>Number of patterns for pairwise coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>our tool</td>
</tr>
<tr>
<td>cardinality</td>
<td>params</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Runtime on 64-bit Linux, modest x86 server:
• insignificant on small problems
• for N-wise, scales as \( \text{cardinality}^{2N} \times \text{params}^4 \)
• ~7 seconds for \( 2^{100} \) example
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
  – 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?