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
Today we're going to discuss a technique for managing test complexity that has been popular in the software testing world for many years, but – perhaps surprisingly – has received little attention in the digital verification community.

There are many existing tools available for pairwise and N-wise testing, some of them very competent indeed, but none of the readily available tools fit comfortably into a hardware verification flow. For example, one of the best freely available tools, PICT from Microsoft, can only be run from a Windows command box. It also requires a non-standard plain text input format, causing wasted work if – as is very likely for us - the problem description is already available in the data members and constraints of a SystemVerilog class.

To ease this usability problem we have created a SystemVerilog package, along with some macros, that allow us to add N-wise capability to an almost unaltered SystemVerilog data object (class). We will take a look at how to use this package on a simple example, and suggest how it could be used in practice.
It's well-known and obvious that simulation-based testing can never be complete. In hardware verification we use a combination of constrained-random stimulus and functional coverage to get the best possible verification confidence from the resources available in our project. The N-wise technique we're looking at today doesn't alter that basic problem, but it can provide a powerful additional way to focus our efforts when combinations of a large number of parameters must be considered.

IMPORTANT: I'm using "parameter" here NOT to mean a SystemVerilog parameter, but in the sense that it's used in the N-wise testing literature: any factor that can be adjusted, and that has an effect on the DUT. It could be a real SV module parameter, or perhaps a configuration setting, a sample-on-reset pin-strap, or the value of a field in a control register.

We all know that the nasty, hard-to-find bugs occur when some unhappy combination of parameters (in this broader sense of the word) excites a corner case that hasn't been properly specified or implemented. Pairwise testing guarantees to test all such combinations of any two parameters. It turns out that we can build a complete pairwise test set using many fewer test cases than you might expect – and, in almost every case, with many fewer test cases than constrained random would need to achieve the same coverage.
To introduce the principles of pairwise, and the use of our package, we're going to look at a very simple example. A DUT (perhaps it's a UART or something of the sort) has an 8-bit configuration register. You would imagine that has 256 values needing to be tested, but it's not quite that bad because field F3 has only 3 legal values.

For a really high quality verification project, we need to run extensive tests (with various kinds of traffic, some errors, and so on) for every control register setting! That leads to an explosion of our testing problem: suddenly we need 192 times as many tests.
Of course, our verification plan will require us to test every possible setting of each individual control field. It is easy to choose four configurations that will achieve this – the green boxes show one possible way to do it. However, that gives us very poor coverage indeed of the configuration space. Our four tests have never tested what happens if field F1 is 00 and field F2 is 11, for example. Surely, at the very least we should check every possible interaction of every pair of configuration parameters.

The table at the bottom lists all ten parameter pairings, with the cardinality (number of possible values) of each pairing. That still looks like a rather large test set, but in fact we can compress it spectacularly, because any given test case will provide a value-pair for each of those ten pairings.
The table on this slide is taken directly from our paper, and shows how only 16 test cases can cover every possible pairwise combination. You can go through the table by hand checking that every pair is covered, if you like; we’ve highlighted how the pairwise pattern set covers all twelve combinations of F2:F3, and the four combinations of F4:F5, to give you an idea of how it works.

Pairwise offers no guarantee of completeness or perfect bug-finding, but we think this approaches the best that can be achieved with that number of tests. Suppose you had a set of 16 tests that did not achieve pairwise coverage... surely that would not be as valuable, because you’ve definitely missed something interesting.

In the lower blue box you can see how pairwise testing is even more powerful and effective when applied to larger problems. The problem space (20 variables each having 10 possible values) is huge – pretty much the same as the number of values that you can represent in a 62-bit number, and far too large even to count through, never mind to test. But our SystemVerilog package yielded only 230 test cases for complete pairwise coverage (some other tools did even better). The generator's runtime may appear to be quite long, but if you bear in mind that each of those 230 test cases will likely be used to configure a simulation that will run for many minutes, it's an insignificant overhead.

The table on this slide is taken directly from our paper, and shows how only 16 test cases can cover every possible pairwise combination. You can go through the table by hand checking that every pair is covered, if you like; we’ve highlighted how the pairwise pattern set covers all twelve combinations of F2:F3, and the four combinations of F4:F5, to give you an idea of how it works.

Pairwise offers no guarantee of completeness or perfect bug-finding, but we think this approaches the best that can be achieved with that number of tests. Suppose you had a set of 16 tests that did not achieve pairwise coverage... surely that would not be as valuable, because you’ve definitely missed something interesting.

In the lower blue box you can see how pairwise testing is even more powerful and effective when applied to larger problems. The problem space (20 variables each having 10 possible values) is huge – pretty much the same as the number of values that you can represent in a 62-bit number, and far too large even to count through, never mind to test. But our SystemVerilog package yielded only 230 test cases for complete pairwise coverage (some other tools did even better). The generator's runtime may appear to be quite long, but if you bear in mind that each of those 230 test cases will likely be used to configure a simulation that will run for many minutes, it's an insignificant overhead.
Over the next few slides we will show how to implement that same 8-bit example using our SystemVerilog package. Of course, we only have time to scratch the surface today, and much more detail and documentation is available along with the code package which you can download from our website.
First we include the Nwise code. You can put the include at the top of any source file that needs it, because the included file has guards so it won't be compiled twice.

The enum typedef is used here to illustrate how the package handles enum variables. It describes the 3-value field F3.

The real work happens in the user-defined class Config. Normally this would just contain some rand variables for the five control fields, along with any necessary constraints. To enable N-wise testing, nothing changes except that we declare those variables using the NWISE_VAR_xxx macros. We also add the NWISE_BEGIN macro to set up some infrastructure. Its argument is the name of the class (Config).

Take a closer look at the first NWISE_VAR_INT macro invocation. It declares a rand variable named F1, of type "int". ANY integral type is OK here. However, for data types wider than 8 bits, you MUST also specify a third macro argument which represents the set of possible values of the variable. That argument is used to construct an "inside" constraint, as shown in the blue callout.

Enum variables are similarly declared using NWISE_VAR_ENUM; they don't have a third value-set argument, because the value-set is already defined as part of the enum type declaration. The value-set argument is optional for NWISE_VAR_INT where the data type is 8 bits or less.

The type-parameterization of Nwise_base is of particular interest. We regarded it as a priority that users should be able to inherit from anything they want – typically, from a UVM base class – when creating their Nwise-enabled classes. But we also needed to inherit from our Nwise_base class to provide some essential infrastructure. SystemVerilog lacks multiple inheritance, so we have used the mixin coding pattern: Nwise_base is parameterized for a type, and extends that type. Consequently, anything that extends Nwise_base#(uvm_object) inherits from uvm_object but also gets all the features of Nwise_base. If you don't want to inherit from uvm_object, that's OK too – simply leave out the parameter and accept the default.
Now we need to use the N-wise enabled class to generate test cases. There is an N-wise generator object instantiated in our Nwise_base class, but it is not directly accessible to users. In this way we can arrange for it to be created on demand, so that a user object that never calls any Nwise methods will not carry the burden of this generator object. Various "Nwise_xxx" methods use that object:
- construct a pattern set using Nwise_generate_patterns;
- "render" a chosen pattern from the set, setting up the object so its Nwise data members reflect the values in that pattern
- and various debug methods.

Our example generates a set of pairwise patterns and then iterates through them, rendering each pattern into the user's configuration object – there's also the option to create a completely new rendered object. Once you have that rendered object, you can use it in any way you choose as a test configuration object, as a stimulus data item, or whatever.

NOTE: We are also considering a modified API that yields test cases one by one. This would have an important benefit in very large examples, because it would avoid the long startup delay caused when the generate_patterns method is asked to generate the full set of test cases for a large problem.
The NWISE_VAR macros allow us to decouple our generation algorithm from the details of the user's variable names and possible constraints among those variables. The key trick is that the macros generate equality constraints, locking each user variable (including each element of any user array) to an element of the dynamic array of int, __nwise_value_proxy. It's this array that is manipulated by the generation algorithm. Because its elements are constrained to be equal to user variables, any application of randomize() on the object will cause the user's additional constraints to be applied both to the user's variables, and to the generator's proxy array. In this way, we can perform generation that fully honors any constraints on the generated variables.

Of course, there's quite a lot more to it than that. In particular, we had quite a tough time getting the code to work correctly when a user extends an existing Nwise-enabled class, adding further Nwise variables in the new class. And we went to a lot of trouble to ensure that the Nwise data structures don't impose any burden on the user's object if they are not used. For example, the __nwise_value_proxy array is a dynamic array, and it is brought into existence only if it's needed for Nwise generation.

Note that this example simply extends Nwise_base without any parameter. The default parameter is type Nwise_void, which provides a stub to allow the Nwise generation to work when your class is not derived from uvm_object or some other UVM base class.
This slide shows how our SystemVerilog implementation stacks up against other available pairwise tools. Our algorithm closely follows the PICT algorithm described in one of the references in our paper, but does not deliver quite such compact test sets as the PICT tool that's based on it (Microsoft Inc, freely downloadable Windows command-line executable). We continue to investigate the reasons for this discrepancy, but we regard the current performance of our algorithm as - at least - useful. Its runtime speed on commercial SystemVerilog simulators is a little disappointing, being at least an order of magnitude slower than the PICT tool. We initially suspected that this was mainly due to the cost of SystemVerilog randomization calls, but subsequent measurements have shown that randomization accounts for only a very small fraction (less than 5%) of runtime. Again we are continuing to investigate the reasons for this poor performance. Nevertheless, the time spent generating each testcase, even in the worst cases, is dramatically smaller than the time likely to be spent in simulating that testcase. We conclude that the tool remains useful despite its disappointing runtime performance.
There are many interesting possible features that we have not had time to implement, and improvements that have become clear with experience. One important facility, which is readily supported by our algorithm but is not yet available with the current API, is to provide a set of required test patterns – typically, those matching configurations required for a specific application. The algorithm will check that those patterns conform to the specified constraints, and then add further test patterns to achieve complete N-wise coverage.

A further useful enhancement is the ability to specify higher-order N-wise coverage for some subset of the problem's parameters. For example, a verification plan may call for pairwise testing of some set of parameters, but also require that every possible combination of values of three especially important parameters be tested. Once again this is straightforward within our algorithm, but requires some API enhancements to make it accessible.

Incremental pattern generation allows the algorithm's runtime to be spread across many test cases, rather than requiring that all patterns be generated up-front.
This slide suggests a few potential applications for the Nwise technique.

One obvious application area is in choosing module-parameter values when verifying highly configurable design IP. Inevitably, this requires a two-pass technique because module parameters must be known at elaboration time – it's not possible for a SystemVerilog simulation to take run-time actions to generate its own parameters.

The other examples shown here, though, could potentially be configured and executed in the same simulation run.
Encourage audience to get the code, experiment with it, and compare it with other available tools.
At this point the presenter will give a brief summary of content and invite Q&A.