Is Your Testing N-wise or Unwise?

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

Jonathan Bromley
Verilab Ltd, Edinburgh, Scotland
jonathan.bromley@verilab.com

Kevin Johnston
Verilab Inc, Austin, Texas
kevin.johnston@verilab.com

ABSTRACT

Pairwise, and more generally N-wise, pattern generation has long been known as an efficient and effective way to construct test stimulus and configurations for software testing. It is also highly applicable to digital design verification, where it can dramatically reduce the number and length of tests that need to be run in order to exercise a design under test adequately. Unfortunately, readily available tools for N-wise pattern generation do not fit conveniently into a standard hardware verification flow. This paper reviews the background to N-wise testing, and presents a new open-source SystemVerilog package that leverages the language's constrained randomization features to offer flexible and convenient N-wise generation in a pure SystemVerilog environment.

Table of Contents

Table of Figures

Table of Tables

Table of Code Examples

1 Introduction

In this paper we claim that the technique known as *pairwise testing* (or, more generally, *N-wise testing*) is applicable to design verification, where it can significantly reduce the amount of testing effort required to achieve confidence that a design has been fully verified. We describe a new open-source package that supports N-wise testing within any class-based SystemVerilog verification environment, including those based on the UVM [1] or similar methodologies. Finally we report on the features and performance of our package relative to other available N-wise tools.

Section 2 outlines the background to the problems that can be addressed by N-wise testing. Section 3 is a brief tutorial review of the fundamentals of N-wise testing and generation.

Readers who are already familiar with these concepts may wish to skip those two sections and go directly to section 4, which shows by example how our SystemVerilog N-wise package can be applied in practice. Section 5 points out some possible drawbacks of the approach. Section 6 gives details of the features and internal implementation of our package, and section 7 reports our results and indicates our future plans.

Finally, section 8 summarizes our conclusions, and section 9 provides references and further reading.

2 Background

2.1 Exhaustive Testing is Impossible

Design verification (DV), like any other form of software testing, can never hope to be exhaustive. Each single-bit register in a design doubles the number of possible states of the design, and any non-trivial device under test (DUT) has so much state that there is no way to test it exhaustively in any realistic period of time using dynamic (simulation) methods.

Current best practice in verification relies on a number of well-established techniques for exploring those parts of the DUT's state space, and the space of possible stimulus, that are most important for real applications and for discovering possible design errors. Constrained random stimulus generation, combined with carefully planned and measured functional coverage, helps to ensure that all practically useful operational scenarios are exercised, creating specific stimulus cases with maximum automation and minimum engineering effort.

2.2 An Alternative Approach

In this paper we argue that the technique known as *N-wise testing* is worthy of consideration alongside these established strategies, especially in handling the range of possible configuration and initial setup of a given DUT. We review existing practice in the field, and present an open-source SystemVerilog package that supports N-wise testing in a convenient and flexible way that fits easily into existing widely-used verification methodologies.

2.3 The Problem of Configurability

Almost every DV project requires that testing be performed on a varied set of configurations of the DUT. For typical design IP projects, the structure of the DUT (number of ports, presence or absence of features, etc) is highly configurable by its users, and the IP vendor must take great care to confirm that all legal configurations will work as expected and that illegal or meaningless configurations are correctly trapped by assertions or other checks. Other designs may have a fixed structural configuration, but are programmable through registers or pin-strapping so that the design can operate in various modes, with the modes being altered relatively infrequently. In

both cases it is a major verification challenge to ensure that the DUT's configuration space has been thoroughly and realistically exercised. It is not unusual to find that the space of possible configurations is itself large enough to make exhaustive testing impractical, even before any run-time activity has been exercised.

For example, consider an IP block with between 2 and 8 bus interfaces, each of which can support 4 different protocols. Additionally, each port has a buffer whose depth can be chosen to be between 1 and 8 transactions. Already, with only this modest configurability, we have more than 1000 plausible configurations that a user might set up. In a high-quality verification project, any single configuration will be the subject of many hundreds of individual test cases in order to achieve good coverage of the verification plan. Configurability has dramatically enlarged the verification problem.

2.4 Pragmatic Management of the Configurability Problem

In practice, users typically choose a modest number of realistic configurations (including, as a minimum, those specified by key early customers) and run their complete test suite on those, targeting full coverage for each. Additionally, a randomized selection of configurations may be tested, but they are likely to be so numerous that it is impractical to apply the complete battery of tests to all of them, and so a limited subset of tests is used. This leads to a worrying dilemma. If a sufficiently large set of configurations is used, so that the team can claim good coverage of the configuration space, then it is likely that many of those configurations will have been inadequately tested. On the other hand, if each new configuration is tested to the point of full coverage according to the verification plan, this will be sufficiently time-consuming that there is little chance of covering the desired range of configurations.

3 Pairwise and N-wise Testing: Blast Containment for the State Explosion

For many years, software testers - faced with configuration space challenges very similar to those described above - have turned to *pairwise testing* to bring the problem within reasonable bounds. The key idea behind pairwise testing is that difficult bugs are most commonly triggered by the interaction, or coincidence, of two parameters *and not more*. The reasoning goes like this (and is justified in much more detail in the comprehensive reading list found in reference [2]):

- Although it is common to find bugs that are triggered when a single parameter has some special value, those bugs are almost always found quite early. It is not difficult to scan each individual parameter over its full range of values, testing at each different value of that parameter.
- More difficult bugs require *more than one* parameter to take a specific value. For example, it may be that if buffer A has its minimum size, *and* buffer B has its maximum size, then there is a deadlock. It is clearly important to seek out such situations in an exhaustive manner.
- Bugs that occur only when *three or more* parameters have specific values are extremely rare. It is neither practical nor useful to cover all such situations. 3-wise and higher order pattern generation is possible, and is supported by the tools and techniques described in this paper. If there is sufficient time available in the verification project, higher order N-wise patterns can offer reduced risk of missing subtle corner-case bugs.

Consequently, pairwise testing proposes that for any given set of parameters (or register values, or any other such adjustable values) we should construct a set of tests that exhaustively exercises every possible combination of any pair of parameters. In many cases this can be done

with a surprisingly compact test set. By considering a rather small example we will show how effective pairwise test construction can be. First, however, it is useful to establish some terminology.

3.1 Terminology used throughout this paper

3.1.1 Parameters, value-sets and cardinality

Each individual variable (or module parameter, or register value, or other adjustable value) that participates in the test configuration is known as a *parameter*[1]. Each parameter has a discrete collection of possible values that we call its *value-set*. For example, a Boolean parameter has a value-set containing the two values TRUE, FALSE. The number of values in a parameter's value-set is the parameter's *cardinality*. A Boolean parameter has cardinality 2; an eight-bit parameter has cardinality 256.

3.1.2 Patterns

In any given situation we have a collection of parameters, usually with differing cardinalities. This collection of parameters defines the problem space. A *pattern* in that problem space is a complete set of specific values, one value for each parameter. Clearly, a pattern represents a single test case or a single configuration to be tested. The total number of possible patterns is easily calculated: it is simply the product of the cardinalities of all parameters.

3.1.3 N-wise Groupings

To perform pairwise testing, we must identify every possible pair of parameters. For example, if we have three parameters P1, P2 and P3, then we want to try every combination of values of parameters P1 and P2, every combination of parameters P2 and P3, and every combination of parameters P1 and P3. Each of these pairs (P1-P2, P2-P3, P1-P3) is a *grouping*[2]. More generally, we can imagine *N-wise* testing in which we consider all possible values of every grouping of N parameters. There is also a degenerate (but nevertheless interesting) case of 1-wise testing, in which our groupings each contain only one parameter, and our aim is simply to test that every value of each parameter has been tested at least once (*i.e.* it has appeared in at least one pattern).

3.1.4 Combinations and pattern coverage

For each grouping, we identify the complete set of all possible values of the group's N parameters. Each member of such a set is called a *combination*. A combination denotes specific values for each of the N parameters in its group, but also marks all remaining parameters as "don't care". If the parameters in a given N-wise grouping have cardinalities $C_1, C_2, \ldots, C_N$ then the number of combinations in the grouping is the product of its parameters' cardinalities: $C_1 \times C_2 \times \ldots \times C_N$.

If a pattern has the same values of a group's N parameters as does some combination, then we say that the pattern *covers* the combination. Clearly, a pattern can cover only one of a grouping's combinations. However, such a pattern usually covers combinations of other groupings too, and it is this multiple coverage that gives N-wise testing its power. The sole exception occurs when N is the same as the number of parameters; this is equivalent to demanding a pattern set containing every possible combination of every parameter.

---

[1] The word "parameter" has a specific meaning in SystemVerilog, and we considered using *factor* to denote an adjustable term in an N-wise set. However, use of "parameter" is so widespread in the literature of N-wise testing that we felt it would be confusing to use a different term here. Consequently we have accepted this new use of "parameter", and we use the phrase *module parameter* if necessary to denote the SystemVerilog language feature.

[2] Reference [2] uses the term *parameter interaction* where we use *grouping*.

3.2 An Example of Pairwise Test Generation

We now present a simple but plausible example to illustrate the value of pairwise testing in reducing the number or length of tests that must be run. As already mentioned, pairwise (2-wise) is the most common and often the most useful form of N-wise testing.

Consider a small DUT having an 8-bit configuration register. Although this register can be modified at any time, it affects a communications protocol and therefore in practice it will be set up and then left unaltered for an extended period during a test. Some tests must, of course, modify this register on the fly, but the majority of functional testing uses it in a static setup that is configured just once at the start of a test.

This 8-bit register has 256 possible values, and therefore it would seem that at least 256 tests must be run to exercise all possible operating modes. However, on closer examination we find that the register is naturally split into several fields, each controlling one aspect of device operation.

- field F1 (2 bits, 4 values)
- field F2 (2 bits, 4 values)
- field F3 (2 bits, 3 valid values, one illegal value)
- field F4 (1 bit, 2 values)
- field F5 (1 bit, 2 values)

The restriction of F3 to only three values means that there are 192 rather than 256 meaningful patterns. Using pairwise testing, however, we need to exercise many fewer combinations. There are:

- 16 possible combinations of F1 and F2,
- 12 possible combinations of F1 and F3,
- …
- 4 possible combinations of F4 and F5.

Using the SystemVerilog implementation of N-wise generation described in this paper, pairwise coverage was achieved using only 18 test patterns, as shown in the following table. Inspection of the table shows that every possible combination of pairs of parameters has been covered. As an example, coverage of the 12 combinations of parameters F2 and F3 has been shaded, and coverage of the 4 combinations of F4 and F5 has been hatched.

		Test pattern																	
		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*
Parameter	*F1*	0	0	3	2	2	1	1	1	2	3	1	3	3	3	2	0	0	0
	F2	2	3	0	1	2	1	3	2	0	1	0	2	0	3	3	1	2	0
	F3	0	1	2	1	0	0	2	1	2	2	1	1	0	0	2	2	2	1
	F4	0	1	1	0	1	1	0	0	0	0	1	1	0	0	1	1	1	0
	F5	1	0	1	0	0	1	0	1	1	0	0	1	0	1	1	1	0	0

Table 1: Pairwise coverage of the 5-field example

In this small example, pairwise test generation has reduced the number of setup configurations from 192 to only 18, a 10× improvement. Larger examples show much greater relative improvement. To illustrate this, consider a setup having 10 parameters each with 4 possible values. (Reference [2] would denote this as a "4^10" configuration.) Complete coverage of all possible combinations of these ten parameters requires more than a million test patterns. Our

prototype SystemVerilog implementation achieved pairwise coverage in only 42 patterns. Some other available tools did even better, finding pairwise coverage in as few as 32 patterns. Clearly this is a massive improvement of at least 20,000×.

3.3 Effectiveness relative to random generation

Although our pairwise generation makes use of some randomization internally, it is deterministic in the sense that it is guaranteed to yield a set of patterns that fully cover all pairwise combinations. An alternative approach might be to use a pure random generator to construct test patterns. To compare the two approaches, we set up a simple covergroup in the class describing our five-parameter example. The covergroup included a cross of each pair of parameters. For example, cross coverpoint `f3Xf4` covered all possible combinations of the grouping of parameters F3 and F4. We then repeatedly randomized the object, sampling coverage after each randomization. After 18 randomizations (the number of patterns constructed by our pairwise generator), random generation had covered 100% of individual parameter values, but had been much less successful in covering pairwise combinations, as shown in the left-hand coverage report below. Only 3 of the 10 pairwise combinations were fully covered. Even after *twice as many* random tests (right-hand report), one of the 10 combinations was not yet fully covered. Only after **47** randomizations did we finally achieve 100% coverage of all pairwise combinations. By contrast, using pairwise testing would cover all the combinations with only 18 tests, a significant reduction in required test runtime. As shown in section 4 below, coding the pairwise generation imposes only a trivial overhead beyond the effort required in any case to create a configuration data class suitable for randomization.

CROSS	EXPECTED	UNCOVERED	COVERED	PERCENT
f1Xf2	16	4	12	75.00
f1Xf3	12	1	11	91.67
f1Xf4	8	0	8	100.00
f1Xf5	8	0	8	100.00
f2Xf3	12	2	10	83.33
f2Xf4	8	1	7	87.50
f2Xf5	8	2	6	75.00
f3Xf4	6	1	5	83.33
f3Xf5	6	1	5	83.33
f4Xf5	4	0	4	100.00

After 18 randomizations

CROSS	EXPECTED	UNCOVERED	COVERED	PERCENT
f1Xf2	16	0	16	100.00
f1Xf3	12	0	12	100.00
f1Xf4	8	0	8	100.00
f1Xf5	8	0	8	100.00
f2Xf3	12	1	11	91.67
f2Xf4	8	0	8	100.00
f2Xf5	8	0	8	100.00
f3Xf4	6	0	6	100.00
f3Xf5	6	0	6	100.00
f4Xf5	4	0	4	100.00

After 36 randomizations

Figure 1: Pairwise coverage only partly achieved by randomization

From these results it is clear that in situations where pairwise coverage is an effective stimulus generation strategy (*i.e.* is effective in finding bugs that might be revealed in real use) then deterministic pairwise pattern generation achieves it much more efficiently than does randomized pattern generation. It is certainly true that randomized pattern generation left some pairwise cases uncovered even after quite extensive randomization. If any of those pairwise cases ("corner cases") were important, then either a bug could be missed, or needless testing effort might be spent in hitting the corner by randomization.

4 Using the SystemVerilog Package

Our goal was to make it as easy as possible for users to adopt pairwise testing in their SystemVerilog verification environments, using data members of an ordinary SystemVerilog class to represent the parameters. In particular, we wanted users to be able to use familiar SystemVerilog data types for their parameters, with no restriction on variable names. Internally, though, the package needs to manipulate these parameters not as individual variables, but as an array of integers. Keeping these two representations in lockstep requires a significant amount of additional code injected into the user's class, which is most conveniently done using macros. To set up N-wise generation features, a user's class must meet two requirements:

- it must be derived from the package-provided base class `Nwise_base`;
- variables representing N-wise parameters must be declared using package-provided macros `NWISE_VAR…`, described in section 4.1.

The configuration register example from section 3.2 above can be represented as in Code Example 4-1 below. Note that an additional user constraint `c_no_parity_if_F1_zero` has been added to the example for illustration purposes.

```
`include "nwise_pkg.svh"        Compile the package if necessary. Make macros available.

package config_info_pkg;        User's package (could also be a module or program)
  import nwise_pkg::*;

  typedef enum {PARITY_NONE, PARITY_ODD, PARITY_EVEN} parity_e;

  class Config extends Nwise_base;     User's Nwise-enabled class
    `NWISE_VAR_INT(  int,      F1, {[0:3]})     Equivalent to rand int F1;
    `NWISE_VAR_INT(  int,      F2, {[0:3]})     Value-set [0:3] must be specified
    `NWISE_VAR_ENUM( parity_e, F3)              No value-set specification for enums
    `NWISE_VAR_INT(  bit,      F4)          Value-set is optional for variables 8 bits or less
    `NWISE_VAR_INT(  bit,      F5)
    constraint c_no_parity_if_F1_zero {   Additional user constraint limits the Nwise set
      (F1==0) -> (F3==PARITY_NONE);
    }
    string name;                          User methods and data members – no restrictions
    function new(string _nm); name = _nm; endfunction
    function void print();
      $display("Config(%s) has parity=%s", name, F3.name);
    endfunction
  endclass

endpackage
```

Code Example 4-1: Using `nwise_pkg` to implement the example of section 3.2

4.1 The NWISE_VAR… macros

Each `NWISE_VAR…` macro invocation is, in effect, the declaration of a `rand` data member. The first argument is the variable's data type. The second argument is its variable name.

- **`NWISE_VAR_INT`** declarations can be of any integral data type, but it is also necessary to specify a value-set for the variable. This follows the same syntax as for an `inside` constraint: braces containing a comma-separated list of values and ranges. The current implementation throws a fatal runtime error if the value-set contains more than 256

different values. If the data type is no more than 8 bits wide, the value-set argument can be omitted and is inferred as the full range of possible 2-state values of that data type.

- **`NWISE_VAR_ENUM`** declarations specify an enumeration type and a variable name. The value-set is inferred from the enum type's declaration.
- Each of the above macros also has an **`_ARRAY`** variant allowing the user to specify a fixed-size array of parameters all having the same data type and value-set.

4.2 Constraints over N-wise parameters

As illustrated by constraint `c_no_parity_if_F1_zero`, constraints can be imposed on the relationships among the N-wise parameters. The constraint illustrated here (which, for simplicity, was not applied in our earlier example) limits the value of parity parameter `F3` if parameter `F1` is zero. The generation algorithm honors all such constraints when generating an N-wise test pattern set.

4.3 Additional properties

As illustrated by the additional methods and data member `name`, there is no restriction on what properties may be added to such a user class, nor on how the N-wise variables are used in other methods. However, it is very strongly recommended that the class have no other `rand` data members, as this could disturb the generation algorithm in potentially undesirable ways.

4.4 Generating and rendering an N-wise set

The user should now create an object, and call methods of its `Nwise_base` base class to perform generation and rendering of test patterns. The first step, generation, is usually performed only once by a single call to the `generate_patterns` method, specifying the N-wise order of generation (defaulted to 2 for pairwise). Generation builds an internal table of parameter patterns, and returns the number of generated patterns. User code can then call the `render` method to set up the object's data members to match any one of these patterns. The following example simply displays a table of generated values, using the built-in `pattern_debug_string` method. It assumes that the code in Code Example 4-1 has already been compiled.

```
module ConfigTest;
  import config_info_pkg::*;
  initial begin
    Config cfg;
    int num_patterns;
    cfg = new("DemoConfig");
    num_patterns = cfg.generate_patterns(2);          make  pairwise patterns
    for (int p = 0; p < num_patterns; p++) begin      scan over all generated patterns
      cfg.render_pattern(p);                  set up values in cfg to match the chosen pattern
      $display("pattern %2d: %s", p, cfg.pattern_debug_string());
    end
  end
endmodule
```

Code Example 4-2: Using the class from Code Example 4-1

Although this example does nothing more than print the patterns on the console, normal usage would be to make use of each new pattern in turn to perform some testing. An alternate method `render_pattern_new` is capable of creating a completely new object with the desired contents, rather than (as in the example) modifying the existing generator object. The output from this

example is shown below. Note that it differs somewhat from the results in Table 1 because of the additional user constraint that appears in Code Example 4-1. For patterns with `F1==0`, the constraint insists that `F3==PARITY_NONE`.

```
pattern  0: { F1:1, F2:2, F3:PARITY_ODD , F4:0, F5:0 }
pattern  1: { F1:3, F2:1, F3:PARITY_NONE, F4:0, F5:1 }
pattern  2: { F1:2, F2:3, F3:PARITY_ODD , F4:1, F5:1 }
pattern  3: { F1:3, F2:3, F3:PARITY_EVEN, F4:1, F5:0 }
pattern  4: { F1:1, F2:1, F3:PARITY_ODD , F4:1, F5:0 }
pattern  5: { F1:0, F2:0, F3:PARITY_NONE, F4:0, F5:0 }
pattern  6: { F1:1, F2:2, F3:PARITY_NONE, F4:1, F5:1 }
pattern  7: { F1:2, F2:1, F3:PARITY_EVEN, F4:0, F5:1 }
pattern  8: { F1:0, F2:2, F3:PARITY_NONE, F4:1, F5:0 }
pattern  9: { F1:1, F2:0, F3:PARITY_EVEN, F4:1, F5:1 }
pattern 10: { F1:3, F2:0, F3:PARITY_ODD , F4:0, F5:1 }
pattern 11: { F1:2, F2:3, F3:PARITY_NONE, F4:0, F5:0 }
pattern 12: { F1:3, F2:2, F3:PARITY_EVEN, F4:1, F5:1 }
pattern 13: { F1:0, F2:1, F3:PARITY_NONE, F4:1, F5:1 }
pattern 14: { F1:1, F2:3, F3:PARITY_EVEN, F4:0, F5:0 }
pattern 15: { F1:2, F2:0, F3:PARITY_ODD , F4:1, F5:0 }
pattern 16: { F1:0, F2:3, F3:PARITY_NONE, F4:0, F5:1 }
pattern 17: { F1:2, F2:2, F3:PARITY_ODD , F4:0, F5:1 }
```

Table 2 : Output from Code Example 4-2

5 Possible Drawbacks of N-wise Testing

As with any stimulus generation technique, N-wise has some disadvantages – some rather evident, others fairly well hidden.

5.1 Generation Performance

The most obvious problem is related to generation runtime. Small examples run so quickly that the generation runtime is entirely insignificant. As an example of a more realistic problem, the $4^{15},3^{17},2^{29}$ example used as a benchmark in reference [2] was solved, on a modest Linux workstation running VCS™ version, in about 150 seconds using our "home-grown" algorithm, and slightly longer using our IPO implementation (see section 7.2). These figures are at least an order of magnitude slower than other implementations tabulated in [2]; we believe that this is mainly because of overhead related to SystemVerilog randomization, and we are currently making considerable efforts to improve our toolkit's runtime performance. Nevertheless, in this example it represents only about 3 seconds of generation time for each test pattern. Since a typical simulation run occupies at least some tens of seconds, this seems to be a reasonable price to pay for a potentially large reduction in test set size.

There is no doubt, however, that the runtime of our current algorithms scales quite badly with increasing problem size, and we have been shown realistic examples that are simply not tractable by our current toolkit. We continue to investigate this area with interest.

5.2 False Verification Confidence

Bach and Schroder's paper [3] on possible methodological pitfalls of pairwise testing has much important and useful guidance, and raises some important questions. However, the authors do not find its arguments against N-wise testing totally compelling, and remain persuaded that N-wise can be a valuable technique even though, like any other technique, it must be used with care and awareness of its limitations.

6 Implementation Details

Our implementation is organized as a single SystemVerilog package, `nwise_pkg`, and a collection of macros for automated code generation. As indicated in the code examples of section 4, each user variable that will participate in N-wise generation must be a data member of a class derived from class `Nwise_base` provided by the package. All such data members must be declared using a macro invocation whose arguments specify the variable's data type and name. These macros arrange for elements of a value-proxy array of `int` to be kept in lockstep with the values of the user's declared variables. In this way, construction of the N-wise patterns can proceed without knowledge of the names and data types of the user's variables, but the user's constraints on relationships among those variables are nevertheless honored.

6.1 Automation Macros

The macros (one of the `NWISE_VAR...` family) not only declare the data member as a `rand` variable, but also inject extensive automation code supporting the N-wise generation. Code Example 6-1 indicates the most important parts of the code generated by a sample invocation of macro `NWISE_VAR_INT`.

```
`NWISE_VAR_INT( shortint, MyVar, {5, [10:19]} )
expands to:
  class __NWISE_VAR_MyVar_PROXY
        extends nwise_pkg::IntSetVarUtils#(shortint);
    randc int x;
    constraint c {x inside {5, [10:19]};}
  endclass

  rand shortint MyVar;
  const int __value__proxy__MyVar__idx =
        __value_proxy_next_idx(__NWISE_VAR_MyVar_PROXY::create("MyVar"));
  constraint __c__value_proxy__MyVar__ {
    MyVar inside {5, [10:19]};
    value_proxy[__value__proxy__MyVar__idx] == MyVar;
  }
```

Code Example 6-1: Expansion of `NWISE_VAR_INT` macro

Code Example 6-1 shows that the macro expands into two major sections. The first is the declaration of a utilities class derived from `IntSetVarUtils`. The second section is a declaration of the named member as a `rand` variable, with its chosen data type and with a constraint limiting it to have one of the specified set of values. This section also contains an initialized declaration of a `const` variable that is set up to contain the index number (in array `value_proxy`) of an array element that will represent the variable within the N-wise generation code. The `create` method that's called by this initialization serves many purposes. It constructs an object of the utilities class, establishes its index in the proxy array, and passes on the variable's name as a string so that it can be reported properly in debug messages.

The utility class contains a constraint limiting the value of its randc variable to the same set of values as specified for the variable of interest. This is necessary to permit the construction of a list of the complete set of possible values, using a technique described in section 6.2 below.

6.2 Using a `randc` constraint to build a list of the variable's value-set

Although the `inside` constraint syntax is convenient for limiting the set of possible values of a variable, it does not directly permit calculation of the complete list of legal values. This list is needed for some N-wise generation algorithms. We were able to achieve this calculation by exploiting the cyclic randomization of SystemVerilog's `randc` variables, as follows. A `randc` variable of `int` type, which is the only random variable in the class, is constrained to be inside the value set. Repeated randomization of this variable cycles through the set of possible values. As each new value is drawn, it is recorded in an associative array indexed by the drawn value. After a certain number of drawings, the set of values is exhausted and a new randomized set is provided for future drawings. At this point, new drawings will be duplicates of those already in the associative array, and they are tallied. As soon as any value is seen for the *third* time, we know that we must have seen at least one complete cycle of drawings and the associative array is certain to be complete. We can then extract all its keys to construct our full list of possible values. Code Example 6-2 shows the fragment of code that achieves this.

```
protected randc int x;
protected int value_list[$};

protected function void populate_value_list();
  int tally[int];
  forever begin
    void'(this.randomize());
    if (!tally.exists(x)) begin
      tally[x] = 1;
      ast_int_cardinality_limit:
        assert (tally.num <= (`NWISE_CARDINALITY_MAX)) else
          $fatal(1, $sformatf(
            "%s: value-set of \"%s\" exceeds maximum cardinality %0d",
            get_file_line(),  name,          `NWISE_CARDINALITY_MAX));
    end
    else begin
      tally[x]++;
      if (tally[x] == 3) begin
        break;  // we got them all!
      end
    end
  end
  foreach (tally[v]) begin
    value_list.push_back(v);
  end
endfunction
```

Code Example 6-2: Constructing the complete list of values of an `inside` set

6.3 Obtaining the code and documentation

Full source code and details of the implementation, including documentation of its API, can be found at the resources section of the authors' website www.verilab.com where it is available under the liberal Apache open source license [4].

7 Results

7.1 Status of the current SystemVerilog implementation

N-wise generation is a nontrivial mathematical problem, but many good solutions have been described in the literature (see, for example, [5], [6], and many others listed in reference [2]). Rather than addressing that mathematical (algorithmic) concern, our work focused on the challenge of integrating the generation smoothly into a typical SystemVerilog verification flow. In particular, we regarded it as a priority that users should be able to write SystemVerilog classes for configuration and stimulus that not only support N-wise generation but also fit smoothly into the users' standard flow, including that of the Universal Verification Methodology (UVM) [1]. We believe we have successfully addressed this concern.

At first we chose to create a *tabula rasa* implementation of an N-wise generator, to help reinforce our own learning experience. Although our "home-grown" generation algorithm is functionally correct, on realistic problems it typically yields test pattern sets that are between 10% and 100% larger than those from the best available tools, and its execution runtime does not scale particularly well with increasing problem size. However, our algorithm is isolated as a distinct subsystem within our SystemVerilog N-wise package. This made it straightforward for us or others to integrate more powerful generation algorithms, taken from the published literature. We have already implemented the *IPO* algorithm of Lei and Tai [6], and expect to have implemented the *PICT* algorithm of Czerwonka [5] by the time this paper is presented. Section 5.1 gives some performance measurements on a sample problem, showing that our current SystemVerilog implementation is somewhat slower than other widely used tools, but can still provide useful overall improvements in runtime by its ability to shrink the number of testcases that must be run. As already mentioned, the algorithm we first implemented – although it was useful for our internal development and has some interesting features – typically generated somewhat larger test sets than the best available tools. We continue to work on these issues.

7.2 Future Enhancements

As suggested in reference [5], our implementation is structured so that the following additional features can readily be added in future by making small additions to the API.

- The ability to include pre-determined test patterns – typically from a previous run for consistency, or from known important configurations that must be tested as part of the verification plan (e.g. configurations needed for a key customer). The generation algorithm will add to these only enough patterns to complete the requested N-wise coverage.
- Smooth integration with the UVM. This will require a different base class (extended from `uvm_object` or `uvm_sequence_item`) but otherwise will have almost no impact. The `NWISE_VAR_...` macros can readily coexist with UVM factory and field automation macros.
- The ability to modify parameter groupings to give more aggressive testing across certain groups of parameters. For example, it may be interesting to do mainly pairwise testing, but combine some critical parameter with others in a 3-wise manner.

We are currently experimenting with these extensions to try to understand what user-facing functionality will be most useful in practice.

8 Conclusions

We have described a new SystemVerilog implementation of N-wise test generation that we hope may be useful to the design verification community. We believe that pairwise and N-wise testing can be valuable, especially for covering a configuration space. The implementation presented here is particularly flexible thanks to its use of SystemVerilog constraint language to describe additional relationships among the parameters.

Our implementation imposes only minimal restrictions on the other contents of a data class written by users to work with the N-wise generator package, reducing the difficulty of adopting this technique. It also integrates easily with existing methodologies and flows, and can easily be adapted to work with data classes in the UVM.

Much work remains to be done to improve performance and capacity of the implementation, and to add further useful API features, but the package has utility in its present form.

9 Bibliography

[1] Accellera Systems Initiative, "UVM (Standard Universal Verification Methodology)," June 2014. [Online]. Available: http://www.accellera.org/downloads/standards/uvm. [Accessed April 2015].

[2] J. Czerwonka, "Pairwise Testing: Combinatorial Test Case Generation," January 2015. [Online]. Available: http://www.pairwise.org/. [Accessed April 2015].

[3] J. Bach and P. Shroeder, "Pairwise Testing - a Best Practice that Isn't," in *22nd Pacific Northwest Software Quality Conference*, 2004.

[4] Apache Software Foundation, "Apache License, version 2.0," 2004. [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0.

[5] J. Czerwonka, "Pairwise Testing in Real World," in *Proceedings, 24th Pacific Northwest Software Quality Conference*, 2006.

[6] K.-C. Tai and Y. Lei, "A Test Generation Strategy for Pairwise Testing," *IEEE Transactions on Software Engineering,* vol. 28, no. 1, 2002.

[7] IEEE, Standard 1800-2012 for SystemVerilog Hardware Design and Verification Language, New York, NJ: IEEE, 2012.

[8] IEEE, Standard 1800-2009 for SystemVerilog Hardware Design and Verification Language, Piscataway, NJ: IEEE, 2009.