SystemVerilog Constraints: Appreciating What You Forgot in School to Get Better Results

Dave Rich
Mentor, A Siemens Business
Dave_Rich@mentor.com

Abstract: Constrained Random Verification (CRV) addresses the time-consuming task of writing individual directed tests for complex systems. We sometimes say that CRV automates writing tests for quickly producing the test cases you can think of, or hitting the corner cases you didn't.

But the reality is, like with any computer programming language, your code executes *exactly* the way it is written, and has no concern for what you were thinking. In particular when coding constraints, this manifests as results that satisfy the constraints, but may not match what you intend. Crashes or conflicting constraint failures are usually easier to resolve because of their abrupt termination. However, without an abrupt termination, you may not notice anything wrong with the results until much later in the process; perhaps after you check your functional coverage reports.

This paper looks at two of the most common issues when constraint solver results do not match your intent: 1) not understanding how Verilog expression evaluation rules apply to interpret the rules of basic algebra, and 2) not understanding the affect probability has on choosing solution values. These are subjects you may have learned (or slept through) in school long ago and need refreshing. This paper presents a background defining how SystemVerilog constraints work, and how these issues play into getting unwanted results. Also, it offers a few coding recommendations for improving your code to get better results along the way.

Introduction

In its simplest form, a constraint is nothing more than a Boolean expression with random variables where the solver is asked to find values that make the expression true. One simplistic way of thinking how a constraint solver works is that it repeatedly tries different values for the random variables until finding values making the expression true. Then those values are left as the result for the random variables. It's actually much more complicated because we want the solver to quickly converge on a solution or tell us no solution is possible without wasting a lot of CPU resources. To understand this better, it helps to look at how one might go about solving constraints without a constraint solver. Assume we have two random variables X and Y that need to have the constraint X < Y. The code might look like what we have in Figure 1.

```
bit [1:0] X; // 0, 1, 2, 3
bit [1:0] Y;
do begin
   X = $urandom;
   Y = $urandom;
end while (! (X<Y) );
```

Figure 1 – Brute force randomization

This loop iterates an average of $\frac{16}{6}$ times before finding a set of values for the X and Y variables that satisfy the constraint. That is 16 possible value combinations for X and Y with only 6 valid solutions We could break this down into individual dependent steps shown in Figure 2

```
begin
  X = $urandom_range(0,2);
  Y = $urandom_range(X+1,3);
end
```

Figure 2 – Constraint dependencies

But this rapidly gets complex with the addition of other constraints, like X[0] != Y[0]. Constraint solvers take all constraint expressions at once, figure out all the dependencies, and come up with solution sets for all random variables at once. In SystemVerilog, we would write this as a class, and the constraint solver would formally deduce that there are six possible solutions as listed in Figure 3.

```
class A;
  rand bit [1:0] X;
  rand bit [1:0] Y;
  constraint C1 { X < Y;}
endclass : A
```

X	Y
0	1
0	2
0	3
1	2
1	3
2	3

Figure 3 – SystemVerilog Class with random variables

Verilog Expressions

A fundamental principle that drove SystemVerilog's development was the unification of language semantics so that rules for expression evaluation were identical across all facets of the language. This meant all of the old idiosyncrasies from Verilog's weak type and expression evaluation rules got absorbed into SystemVerilog's new constraint expressions.

One of the unique things that distinguishes the Verilog Hardware Description Language (HDL) from most other software programming languages is its ability to declare variables of different bit-widths and have expressions of mixed bit-width operands, whereas most other programming languages only consider of byte or word sized operands. But most hardware engineers are not trained in software development and tend to favor many implicit rules and weak type systems. These rules mean many expressions silently truncate or pads to different bit widths without any notice of overflow or underflow. This type system radically different from another rival language, VHSIC-HDL (VHDL) which has an extremely strong type system.

Since Verilog is used for synthesis into hardware, it does need precise definitions for generation of hardware without ambiguity or creating unnecessary extra logic. In addition to syntax, there are many transformational steps Verilog needs to follow when evaluating an expression. Suppose we had the code in Figure 4.

```
bit [2:0] A = 4;  // 3 bits
bit [3:0] B = 14; // 4 bits
int C = 1;
bit [3:0] D = 8; // 4 bits
if( A + B >> C < D )...
```

Figure 4–Steps in evaluating an expression

The first step when evaluating an expression is figuring out the order of operators in the form of precedence rules. When we see

$$(A + B \gg C < D)$$

precedence rules group the operators as

$$\big((A + B) \gg C\big) < D)$$

The next step is context determination. This tells us which operands within an expression influence the implicit casting of other operands. Expressions are either *self-determined*, meaning they do not have outside influences, or they are *context-determined* when some outside part of the expression has influence over it. Context affects the resulting types when mixing different bit-widths, signed and unsigned types, and real and integer types. The Language Reference Manual (LRM) specifies these steps in a number of tables which have been consolidated into one in Figure 5 for this example.

Operators i op j	i op j result length	Context	Precedence
+ -	Max(Length(i),Length(j))	operands get sized to Max before operation	Higher
<< >>	Length(i)	j is self-determined	
< <= > >=	1 bit	operands get sized to Max before operation	
{}	L(i)+L(J)	All operands are self-determined	Lower

Figure 5 – Context and bit-length determination

According to these rules, we recognize that operands A, B, and D from Figure 4 get sized to the maximum length, 4 bits—so A gets padded with one 0 bit. The size of C is self-determined and has no influence on the rest of the expression. The result of 4'd4 + 4'd14 is 4'd2 because of the overflow truncation. It is not 5'd18 as most would expect. That makes the overall result of 4'd2 < 4'd8 to be true, 1'b1. To make the expression deal with the overflow, some operand or sub-expression must be cast to 5 bits (e.g. 5'(A + B) >> C < D).

Constraint Expressions

Now let's take a look at the same expression used in a random constraint expression. For simplicity of the code in Figure 6, only the variables A and B are declared **rand**. Variables C and D are converted into literal constants.

```
class tx;
  rand bit [2:0] A;
  rand bit [3:0] B;
  constraint C1{ A + B >> 32'd1 < 4'd8; }
endclass : tx
```

Figure 6 –Class constraint

As we saw in the previous section, 4 and 14 are perfectly valid values for A and B that make the expression result true, satisfying the constraint. The constraint must be written to deal with overflow properly by casting to the appropriate width to handle the overflow. Any one of the casts shown in Figure 7 handle the overflow properly giving us the solutions we expect.

```
constraint c { A + B        >> 32'd1 < 5'd8; }
constraint c { int'(A + B) >> 32'd1 < 4'd8; }
constraint c { 5'(A) + B   >> 32'd1 < 4'd8; }
```

Figure 7 – Possible corrections

Another common mistake arises when using the array reduction methods, like `sum()`. These methods get expanded to iterate over each element in an unpacked array. But the problem is that by default the result is the same bit-width as each element type. If you have

```
rand bit bits[6];
constraint c {bits.sum() == 3;}
```

The built-in sum method gets expanded to

```
{ (bits[0]+bits[1]+bits[2]+bits[3]+bits[4]+bits[5]) } == 3
```

The concatenation operator serves to show the result is self-determined (review the table in Figure 5), and only 1-bit wide since each array element is only 1-bit wide. The reduction methods have an in-line extension capability using the `with()` clause allowing you to substitute a different expression for each element. By casting each element to a larger width, the result no longer overflows.

```
constraint c {bits.sum() with (int'(item)) == 3; }
```

Signed Expressions

A signed operand only affects how that operand gets padded to a larger width (sign extension) or its magnitude for the comparison of relational operators. It has no effect on how the value is stored or truncated. One of the most confusing rules when mixing signed and unsigned operands is that all operands in the context must be signed for the result to be signed, and that introduction of an unsigned type halts propagation of sign extension.

```
bit signed [2:0] A = -1;
bit [3:0] B = 14;
A + B < 14
```

Figure 8 – Mixing signed and unsigned operands

In Figure 8, even though `A` and `B` both get extended to 32 bits before the addition (From the LRM, literals (14) are implicitly a 32-bit signed operands), `A` does not get sign extended. This is because `B` is unsigned, and the rest of the expression is treated as unsigned. `A` becomes `32'd7` and the result of the expression becomes `32'd21 < 32'd14`, which is a false `1'b0`. The need for mixing signed and unsigned operands is rare. Try to use all one type or the other, or make sure you cast the unsigned operands to signed (`signed'(B)`).

But the most common mistakes when using signed operands is forgetting that they could have negative values and that comparing signed to unsigned operands becomes all unsigned. In Figure 9, -1 is a valid value that could be chosen for `max_address`.

```
rand int max_address;
rand bit [11:0] address;
constraint c { max_address < 10;
               address < max_address; }
```

Figure 9 – Allows negative values

But in the relational comparison to the unsigned `address` that value would become an unsigned positive 32'hFFFFFFFF resulting in unexpected large values for `address`. It is always safer to use an explicit range using the `inside` operator as shown in Figure 10.

```
rand int max_address;
rand bit [11:0] address;
constraint c { max_address inside {[0:10]};
               address < max_address; }
```

Figure 10 – Bounded range constraint

Probabilities and Statistics

Going back to Figure 3 with the constraint X<Y, we saw that there are 6 solutions the constraint solver can choose from. Although each solution has an even probability of being chosen, the probability of a particular value being chosen for a random variable is uneven. The table in Figure 11 shows the probabilities of the three possible values available to be chosen for Y and Y.

Y==3	X==2 or 1 or 0	3/6 = 50%
Y==2	X==1 or 0	2/6 = 33%
Y==1	X==0	1/6 = 16%
X==0	Y==3 or 2 or 1	3/6 = 50%
X==1	Y==3 or 2	2/6 = 33%
X==2	Y==3	1/6 = 16%

Figure 11 –Probability of selecting a value

We can add a construct that changes the approach to how the solver chooses values. Figure 12 instructs the solver to

```
class A;
  rand bit [1:0] X;
  rand bit [1:0] Y;
  constraint C1 { X < Y; solve X before Y;}
endclass : A
```

Figure 12 – Choose variable value selection order

choose a value for X evenly over all the possible values for X in the solution set. It does not change the total number of solutions. Now all three value choices for X will each have a 33% chance of being selected. Realize this affects the probabilities for the three value choices for Y, which now becomes slightly harder to calculate. The probably of choosing Y==1 is

$$\frac{1}{3} * \frac{1}{3} = 11\%$$

and choosing Y==3 is

$$\frac{1}{3} * \frac{1}{3} + \frac{1}{3} * \frac{1}{2} + \frac{1}{3} * \frac{1}{1} = 61\%$$

It is not possible with most constraints to get uniform value distributions for all random variables unless their solution spaces are independent of each other. For a small solution set like the one generated for Figure 12 this might not be an issue, but for larger solution sets, getting particular values becomes much harder. The seemingly simple example

```
class A;
    rand bit ctrl;
    rand bit [31:0] data;
    constraint C0 {ctrl -> data == 0;}
endclass : A
```

Figure 13 – Unobtainable solutions

shown in Figure 13 has $2^{32}+1$ (~4 billion) possible solutions. But only 1 in 2^{32} solutions allows `ctrl` to have the value 1—practically never! Adding a `solve ctrl before data` construct lets `ctrl` have a uniform distribution, but the probability of `data`==0 is now 50%. If we do not want that to happen, we can add a distribution constraint as Figure 14 shows.

```
class A;
    rand bit ctrl;
    rand bit [31:0] data;
    constraint C0 {ctrl -> data == 0;
    ctrl dist {1:=10, 0:=90;}
endclass : A
```

Figure 14 – Correcting distribution

Many people expect distributions to come out exactly the way they specify. But the probability of getting exactly the specified distribution is low until you approach an infinite number of randomizations. If you flip a coin 10 times, there is only a 25% chance of getting exactly 5 heads and 5 tails, $\binom{10}{5}/2^{10}$.

If you do need a specific distribution in a determinate number of randomizations, the `randc` construct comes in handy because it randomly cycles through all possible values evenly before repeating a value. The example in Figure 15 sets up an exact 50/50 distribution over the course of 100 randomizations.

```
typedef enum {HEADS,TAILS} coin_t;
class C;
    rand coin_t toss;
    randc int scale;
    constraint c {
       scale inside {[0:99]};
      (scale < 50) -> toss == HEADS;
      (scale >=50) -> toss == TAILS; }
endclass : C
```

Figure 15– Fixed distribution

Conclusion

To summarize, I've shown some simple examples of how Verilog expression evaluation rules affect the solution space of your SystemVerilog constraints, sometimes giving you unexpected, but valid results. I've also shown examples of getting valid results, but how probabilities work preventing you from seeing *all* the expected valid results. Analyzing these situations requires that you go back to review your Verilog, Algebra and Statistics textbooks to appreciate all the factors involved in producing solution sets from your constraints.

References

1. The Top Most Common SystemVerilog Constrained Random Gotchas, Ahmed Yehia, DVCon-Europe 2014.
2. IEEE Standard for SystemVerilog, Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800-2017. https://ieeexplore.ieee.org/browse/standards/get-program/page/series?id=80
3. UVM Random Stability: Don't leave it to chance, Avidan Efody, DVCon 2012.
4. Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them, Stuart Sutherland and Don Mills, Springer.