Reverse Gear:
Re-imagining Randomization using the VCS Constraint Solver

Paul Marriott[1]
Jonathan Bromley[2]

[1]Verilab Canada Inc. - Montreal, Canada
[2]Verilab UK Ltd. - Oxford, UK

www.verilab.com

ABSTRACT

Randomization is usually used to create stimulus as well as interesting device-under-test configurations. With functional coverage, confidence that the interesting state space of the DUT has been hit can be achieved.

Since the constraint solver is so powerful, it is interesting to use it in reverse gear. Due to a SystemVerilog feature, it is possible to use it as a versatile checking engine. E.g. after monitoring a transaction, you can use the existing constraints to determine the transaction kind, if it was a valid transaction or which errors have been observed.

This paper will describe techniques to use the VCS constraint solver, SystemVerilog class constraints, together with judicious use of state variables to build a powerful declarative engine for verification environments in order to answer questions of the form "which kind of configuration could have given me these values", as well as solve an equation for one of its variables.

Table of Contents

Table of Code Examples

1. Introduction

In this paper we will first describe the typical usage of SystemVerilog's randomization and constraints as well as recommended methods for handling the failure of the constraint solver to find a solution. We will also discuss various techniques to randomize specific variables, rather than all of the random properties of a class. We then discuss how to use the constraint solving engine to re-create meta-data from a physical packet which leads into a discussion of using constraints as a concise mechanism with which to write a checker. We then expand on these concepts to describe how to use constraints and partial-randomization to explore the configuration space of a device. Finally, we present a technique for using the constraint solver to solve a set of equations from a given starting point.

A good set of references to the analysis of constraints used in a verification environment context can be found in Große et al [1].

2. Introduction to Constraints

In their simplest form, constraints can be considered to be boolean expressions which *must* be maintained *true* by the solver. They differ from procedural code in that their order of declaration is not important - all active constraints are taken into account simultaneously (though there is a caveat here; see *Global (scope) randomization using* `std::randomize` in section 4).

Constraints can be defined as class members, and must be given a name. To illustrate various concepts in this paper consider this imaginary data packet class. We have not shown all the necessary constraints - you can easily work out for yourself what else would be needed. We have also made the packet format very simple. Of course, in reality, it would have checksums and other framing information, but we have omitted these here to save space and to keep the example small and clear.

```
typedef bit [7:0] ubyte;
class Packet extends some_useful_base_class;
  // HEADER
  rand ubyte control; // payload byte count 0:127, or
  // control msg type 128:255 (no payload)
  rand ubyte addr[4]; // destination IPv4 address
  // PAYLOAD
  rand ubyte payload[];  // contains 'length' bytes
  rand enum {BROADCAST, LOCAL, WAN} addr_kind;
  // CONTROL INFO FOR TESTBENCH USE - not part of the packet
  rand bit is_ctrl_msg;
  constraint c_payload_length {
     if (is_ctrl_msg) {
        payload.size() == 0; control >= 128;
        } else {
        payload.size() == control; control <= 127;
    }
  }
  constraint c_address_kind {
     (addr_kind==BROADCAST) == (addr[0] == 255);
     (addr_kind==LOCAL) ==
       (   addr[0]==192 && addr[1]==168
        || addr[0]==10  && addr[1]==0
       );
    }
endclass
```

Code Example 2-1: Example packet class with constraints, used throughout this paper

In the above example, there are two constraint members, named `c_payload_length` and `c_address_kind`. Both constraints establish relationships between the physically meaningful data members of the class (`addr, payload, control`) and "control knob" data members `is_ctrl_msg` and `addr_kind`.

If-and-only-if constraints

It may be of interest that we have avoided using implication constraints, which are of the form `expr1 -> expr2`, in the `c_address_kind` constraint. Instead we have insisted that the truth values of two comparison operations should be the same, using an "if and only if" constraint of the form `expr1 == expr2`. To understand why this might be valuable, consider the implication constraint

```
  addr_kind==BROADCAST) -> (addr[0] == 255);
```

This constraint specifies that if `addr_kind` is `BROADCAST`, then `addr[0]` must be equal to 255. However, it specifies nothing about the value of `addr[0]` when `addr_kind` is *not* `BROADCAST`. It would therefore be possible to get a non-broadcast packet having `addr[0]==255`, which was not our intent. By contrast, the if-and-only-if formulation insists that `addr[0]` being equal to 255 is exactly the same condition as `addr_kind` being `BROADCAST`.

Enabling and Disabling Constraints

By default constraints are active (enabled), but there are methods available to enable or disable a particular constraint or to enquire about its current enabled/disabled state. Every constraint has a method[1] named `constraint_mode()`. When called with no argument, `constraint_mode()` returns 1 or 0 to indicate if the constraint is currently enabled or disabled; when called with a single truth-value (1 or 0) argument, it enables or disables the constraint.

For example, assume we have produced an instance of `Packet` called `P`. We can do the following:

```
if (P.c_payload_length.constraint_mode() )
  $display("Payload length constraint is active");
else
   begin
     $display("Enabling Payload length constraint ");
     P.c_payload_length.constraint_mode(1);
   end
```

Code Example 2-2: Checking and setting `constraint_mode()`

Constraints can also be declared in-line using the optional `with` clause in a call to a class's `randomize()` method. Such inline constraints apply *in addition* to all other active constraint members.

3. Normal Constraint Usage

Constraints are used to drive the solver to produce a consistent set of values which satisfy all of the constraints simultaneously. Depending on the complexity of the constraints, there is usually more than one solution that can be generated and this fact is the basis of producing interesting stimulus in a constrained-random methodology. If no solution can be found then the randomization fails and returns FALSE (zero) from the call to `randomize()`. If appropriate the programmer can flag this as an error in the test environment.

Handling Randomization Failures

If you do not need to check for randomization failure, it's easy to throw away the success/failure result:

```
void'( my_obj.randomize() );
```

Code Example 3-1: Voiding the pass/fail result of randomization

Whether you check the result or not, randomization failure will leave all random variables unchanged from their original (pre-randomization) values.

[1] Strictly speaking, a constraint is not a class or object and therefore it cannot have *methods* in the usual sense. However, `constraint_mode` is invoked on the constraint using method-like syntax, and it is customary to describe it as being a method of the constraint. Some other languages describe such method-like operations as *pseudo-methods*, but that term is not commonly used in the SystemVerilog community.

Here's a method that we have often seen suggested, but we discourage:

```
assert ( my_obj.randomize() ) else …
```

Code Example 3-2: Using an assertion to check randomization – NOT RECOMMENDED

We strongly advise against this approach, because it can have a very surprising effect. If this immediate assertion is disabled with any of the simulator's assertion control commands, or any of the `$assertoff` family of system tasks, then the assertion is **not** processed at all, and the randomization does not take place! A safer way to check for randomization failure is to test the result explicitly:

```
if ( my_obj.randomize() ) …
```

Code Example 3-3: Explicitly testing for randomization success

The structured error message provided by an assertion is still useful, of course, so it may be appropriate to handle the error like this:

```
int ok;
ok = my_obj.randomize();
assert (ok) else $error("The randomization failed horribly");
```

Code Example 3-4: Proper use of assertion to check randomization success

In this case, disabling the assertion will remove any error indication, but the randomization will nevertheless take place. We think the approach shown in code example 3-4 makes a lot of sense.

4. Randomizing Individual Variables

It is often useful to randomize just one or two data members of an object. There are several ways to do this, and it is very important not to confuse them, as their results can very different.

Section 5 describes using the global randomization function `std::randomize`. This function allows any variable to be randomized, but it does not honour constraints on `rand` variables of a class, and so we generally do not recommend its use with classes. Furthermore, some tools (including VCS) do not permit this function to be applied to a class member except within a method of the class itself, further limiting its usefulness in the context of classes.

Section 6 describes how to control an object's `randomize` method so that it affects only a subset of the class's `rand` data members. This is a very flexible technique, and later in the paper we show how to apply it in various ways to achieve useful results.

Finally, it is possible to use the `rand_mode` method of individual data members of an object to exclude them temporarily from the effects of `randomize`. We discourage this as a way to randomize subsets of the object's data members, because it is easy to forget to re-enable `rand_mode` for the data members in question; if a data member is inadvertently left with its `rand_mode` disabled, future randomization attempts can fail or yield unexpected results in a manner that is very hard to diagnose.

5. Global (scope) randomization using `std::randomize`

If code that is not part of a class method uses the built-in `randomize()` function, then a slightly different randomization mechanism is used. The `randomize()` function used in this case is a member of the `std` package that is built-in to SystemVerilog. There is no need to import it, but beware: if this code is in a module, it must be compiled in SystemVerilog mode, which might not be the default, as the `std` package is not present in standard Verilog.

You may pass one or more variables as arguments to `std::randomize()`, along with an optional constraint block, and the randomization is done using those variables alone, and with no constraints other than those specified in your constraint block. It is important to note that this is true even if the variable is a `rand` member of a class!

The example below shows the difference between the two forms of `randomize()`. Note that we invoke `std::randomize()` from within the class, because VCS forbids its use on a data member of a class if called from outside the class.

```
module revrand_ex;

  class multiple_of_4;
    rand bit [7:0] value;
    constraint mul4 { (value % 4) == 0; }
    function bit non_class_rand();
      bit ok = std::randomize(value);
      return ok;
    endfunction
  endclass

  initial begin

    multiple_of_4 x;
    x = new;

    $write("Class randomization honors constraint :");
    repeat (5) begin
      x.randomize();
      $write(" 'h%h", x.value);
    end
    $display();

    $write("Scope randomization ignores constraint:");
    repeat (5) begin
      x.non_class_rand();
      $write(" 'h%h", x.value);
    end
    $display();

  end

endmodule
```

Code Example 5-1: Illustrating the difference between scope and class randomization

We ran this example in VCS and got the following results:

```
Class randomization honors constraint : 'h7c 'h78 'h00 'h44 'hdc
Scope randomization ignores constraint: 'h3c 'h7d 'he2 'h0b 'hdf
```

Note how the class randomization correctly honors the class constraints and yields only values that are a multiple of 4, whereas scope randomization ignores the constraint and generates unconstrained values.

To get the effect of constraints that you have written as part of a class, you must use the class's own built-in `randomize()` method, either by calling it from a method of the class itself, or by invoking it through an object of the appropriate class type, as shown above.

However, as with so many things in SystemVerilog, it ain't that simple! Just as with `std::randomize()`, you can supply variables as arguments in the call to a class's `randomize()` method (although, in this case, the variables must be data members of the same class). And you can provide additional constraints in a `with` block, but, in this case, the `with`-constraints are imposed in *addition* to all the other constraints that already exist in the class. There are some very interesting tricks that you can play with this, and we look at some of them in later sections of this paper.

6. Randomizing a Subset of a Class's Variables

It can be useful to apply randomization to some, rather than all, of the `rand` data members of a class. For example, suppose you have a class representing a data packet, and you want to generate several packets with identical headers but different payloads - rather than writing complicated `with`-constraints to keep the header fields invariant across several randomizations, it is probably easier simply to tell the randomization that it should not touch those fields, but instead, should randomize only the remaining data members of the class.

Referring back to our Packet class example introduced in section 2:

```
class Packet extends some_useful_base_class;
…
```

Now we can of course construct such an object and randomize it:

```
bit ok;
Packet p = new();
ok = p.randomize() with {addr_kind==LOCAL; !is_ctrl_msg;};
```

Code Example 6-1: Construct and randomize an object with problem-specific constraints

This will give us a packet with IP address either 192.168.x.x or 10.0.x.x, with a control byte in the range 0..127, and a random payload of that length. But suppose we now wish to create several more packets with exactly the same address and length, but with different data. Repeating the same constraints is not sufficient, because although every transaction's IP address will meet the constraints, they will all be given different random addresses.

A far better way is to disable randomization of the address and length fields. We have two distinct ways to do this:

- We can disable randomization of a particular variable by setting its `rand_mode` property to zero.
- We can pass only the variables we want altered into the `randomize()` method, and all other variables will be left untouched.

Both methods work well, and both correctly use the class's constraints as we require. However, the second method is usually more convenient. Using `rand_mode()` is a little tricky because you must remember to switch randomization back on again when you have finished; it's very easy to get this wrong, especially in a complex piece of code. Often, using selective `randomize()` is a better alternative, as outlined in the following example.

```
bit ok;
Packet p = new();
// First attempt: randomize all the object's rand variables
ok = p.randomize() with {addr_kind==LOCAL; !is_ctrl_msg;};
<use that packet>
// Second attempt: randomize only data, leaving header  alone
ok = p.randomize(payload);
<use the second packet>
// Third attempt: re-randomize everything
ok = p.randomize(); // No need to flip a rand_mode control.
```

Code Example 6-2: Randomization of selected properties of an object

Note how this approach differs from using `std::randomize()`. In our example, all of the class's constraints continue to apply when `p.randomize(data)` is called. By contrast, if you tried to use `std::randomize(p.payload)`, the class's constraints would be completely ignored - not good.

Capturing randomization in class methods

The approach we outlined in the previous section works very well, but it requires the user to know exactly which data members to randomize and which to leave alone. For requirements such as our "same header, different data" example that you can expect to be useful on many occasions, it is a good idea to add specialized methods to your class so that a user can invoke the method rather than needing to remember exactly which data members to randomize. Our class method might look like this:

```
<within class Packet>
  function int rand_payload_same_header();
    return this.randomize(payload);
  endfunction
```

Code Example 6-3: Class method to capture re-usable selective randomization

Note that it's not really necessary to use **`this.`** as a prefix to the `randomize()` call, because we are already inside a method of the same class. However, in our own work we find it useful as a reminder that we are not making a call to `std::randomize()`.

7. Using the Constraint Solver to Reverse-Engineer a Packet

An especially interesting and useful application of the constraint solver is to reconstruct meta-data from the physical data captured off a DUT interface by a monitor. Suppose we have a monitor component that observes our DUT output and captures data into one of our `Packet` objects so that it can be passed on to other parts of a testbench. At this stage, of course, we have the physical fields `control`, `addr` and `payload` picked up directly from DUT signals, but we do not yet have the meta-data `addr_kind` and `is_ctrl_msg`. It would be very useful to reconstruct these fields based on the observed data, so that other parts of the system don't need to duplicate this effort, and our `Packet` object has all the same information that you would expect from an object created by your testbench's stimulus generator.

Of course, we could write a class method to do this. It would need to examine the various physical fields, compare them against the various legal message formats, and reconstruct the appropriate meta-data values. Not only would this be tedious, but it would also represent unpleasant duplication of information, because *the desired relationships are already fully described by our randomization constraints*. And we can recruit those constraints to our service and get the constraint solver to find a solution! All we need to do is to take the physical data packet and randomize *only its meta-data fields*:

```
get packet p with physical data already collected
ok = p.randomize(addr_kind, is_ctrl_msg);
if (!ok)
  $display("Could not analyze data packet");
else
  $display("packet kind = %s, is_ctrl_msg = %b",
                    addr_kind.name, is_ctrl_msg);
```

Code Example 7-1: Reverse-engineering an object's meta-data from its physical data

This gives us excellent re-use because we had to describe the conditions for legality of the packet only once, in the constraints. It gives us checking, because if the packet was in some way ill-formed then the constraint solver would not be able to find a solution and would return `FALSE`. And, perhaps best of all, it is amazingly neat, compact and easy to code by comparison with manual (procedural) analysis of the packet format.

8. Constraints as Checkers

In the previous section we mentioned that using the constraint solver to infer meta-data also provides us with some checking for legality, because the `randomize()` method will return `FALSE` if no solution can be found. SystemVerilog takes this idea a step further, allowing you to create a legality checker by using the return status of the `randomize()` method call when called with an explicit null argument:

```
if ( p.randomize(null) ) …
```

Code Example 8-1: Constraint solver as checker: `randomize(null)`

The call to `randomize(null)` will return `TRUE` (nonzero) if all the current values of the random members satisfy the currently active set of constraints. If the set of values of the random

members does not satisfy the constraints, then the call to `randomize(null)` will return `FALSE` (zero) just as it does whenever the solver cannot find a solution. This allows us to create a set of constraints defining the conditions for legality of the random variables, and then use the constraint solver to determine if the conditions are satisfied or not. For example, in a networking application, the range of valid destination addresses could be defined as a constraint:

```
addr[0] inside {[1:127]};
```

Code Example 8-2: Problem-specific rule written as a constraint fragment

On receiving a `Packet P` from the DUT we randomize it with `null` as an argument, using an additional `with` clause in order to apply our specialized constraint as a check:

```
Packet P;
bit ok;
...
// receive P from the DUT
...
//then check it for legality, with additional restrictions
ok = P.randomize(null) with {addr[0] inside {[1:127]};};
assert (ok) else
  $error("Received packet was not in the restricted range");
```

Code Example 8-3: Constraint solver as checker with additional problem-specific rule

On receiving a packet from the DUT, the validity of the packet's next destination can thus be checked. A more complex example, shown in the presentation slides, would be to define the valid relationships between, say, source and destination ports and check to see if these hold.

9. Applications of Declarative Programming

Once we have learnt to think of the constraint solver as just that - a machine for finding some solution to a set of simultaneous equations and inequalities, the constraints - it's not difficult to find other creative uses for that idea.

Inventing Testbench Configurations

Creating interesting partially-randomized configuration objects for a testbench is a fine and useful example. Typically, the testbench configuration includes a large collection of interrelated parameters such as the number of ports on a networking DUT, the width of address and data busses, presence or absence of specific DUT features, and information about what values need to be placed in DUT control registers to set-up a given test. Exploring the resulting huge configuration space is very time-consuming and in practice it is very difficult to try out more than a handful of configurations in full detail during the course of a typical verification project.

Requirements specifications often demand a variety of specific configuration features - for example, with or without cache memory; 64-bit or 32-bit data; with or without edge-sensitive interrupts. Often these features interact so, for example, it might be that a 64-bit system cannot use a cache smaller than 512KiB whereas 32-bit systems can only use 256KiB or 512KiB caches.

Writing these relationships as constraints over the data members of a configuration object, and using the constraint solver to construct a configuration object that meets all requirements of the constraints, allows us to explore the configuration space in a controlled way. For example, we could set the cache size to 512KiB and then randomize everything else. This can easily be done using `randomize() with{...}` in addition to the kinds of techniques we have already described. One interesting possibility is to allow the configuration object to be semi-automatic. To do this requires that each randomizable field should have a default value that is illegal - for example, zero for the cache size. The object's `pre_randomize()` method can then inspect all the data fields, and set `rand_mode(0)` for any that have already been given a legal value. Now we have an object that you can set up by writing manually chosen values to any subset of its data members, and then using a simple `randomize()` call to provide legal values to all the remaining data members.

With a little more work, we can even write constraints that are not just relationships between the various configuration items, but drill deeper into the desired DUT behaviour. For example, consider a configuration object that is then used to determine the values to be written to a set of DUT control registers. If we write constraints to describe how the control register values are related to the configuration, we can use those constraints to meet testing requirements such as:

- find configurations that require us to write the same value (whatever it may be) to each of two different control registers
- find configuration that can be set up by writing only a chosen subset of the registers
- find as many configurations as possible that require us to write the value `16'hDEAD` to a given register, because in some previous test we found a bug when that value was used

Solve from Any Starting Point

Another useful way to look at the constraint solver is as a way to solve a set of equations or inequalities from any chosen starting point.

Suppose, for example, we have a packet-based communications system in which messages are typically much larger than a packet, and so must be spread across multiple packets. We specify that messages with N packets are constructed so that the first N-1 packets all have the same length, and the last (Nth) packet has whatever length is needed to complete the message.

In a verification environment, we might want to construct such a message in many different ways:

- We want a message of exactly 10 packets, and we want each packet to contain an even number of bytes. We don't care about the exact message size or the exact length of each packet.
- We want a message to carry between 2000 and 3000 bytes, and each packet (except the last) should contain exactly 120 bytes. We don't care exactly how many packets are used to carry the message.
- We want a wide variety of different message sizes, but we want to try using exactly 127-byte packets in every case.

Each of these is a completely reasonable request, and of course you could write procedural code to handle each one - but it would be *completely different* procedural code in every case! By contrast, we can write a set of constraints describing the relationships among packet length, message length and number of packets - and then, by adding just one or two additional constraints, we can use exactly the same randomize call to solve each one of these and many other problems.

Here's an example of how this might work. We'll create a new object to represent how the message is packetized, with `rand` variables indicating the message length, number of packets and packet lengths:

```
class Message extends some_useful_base_class;
  parameter LONGEST_MESSAGE = 3000;
  parameter LONGEST_PACKET = 127;

  // The number of bytes in the message
  rand int unsigned messageLength;
  // The number of data packets that make up the message
  rand int unsigned numPackets;
  // Lengths of the various packets
  rand int unsigned lastPacketLength;
  rand int unsigned otherPacketLength;

  constraint c_maxLength {
    messageLength inside {[1:LONGEST_MESSAGE]};
  }

  constraint c_somePackets {
    numPackets > 0;
    numPackets <= messageLength; // see text for details
  }

  constraint lastPacketLegal {
    lastPacketLength inside {[0:LONGEST_PACKET]};
    otherPacketLength inside {[1:LONGEST_PACKET]};
  }
  constraint c_totalOK {
    otherPacketLength * (numPackets-1) + lastPacketLength
     ==  messageLength;
    numPackets == 1 -> otherPacketLength == 0;
  }
  constraint c_avoidVeryShortPackets {
    if (messageLength < LONGEST_PACKET) {
      soft otherPacketLength > messageLength/4;
      } else {
      soft otherPacketLength > LONGEST_PACKET/4;
     }
  }

  virtual function void print();
    $display("Message has %0d bytes over %0d packets",
    messageLength, numPackets  );
    if (lastPacketLength == otherPacketLength)
    $display("   All packets have length %0d", lastPacketLength);
    else
      $display("   %0d packets of %0d bytes, one packet of %0d bytes",
               numPackets-1, otherPacketLength, lastPacketLength );
  endfunction

endclass
```

Code Example 9-1: Class representing mapping of a long message to its component packets

Given the set of legality rules captured in this object, we can now write amazingly simple solutions for the three requirements we outlined:

```
Message msg = new;
bit ok;

$display("\n=== UNCONSTRAINED ===");
ok = m.randomize();
m.print();

$display("\n=== EXAMPLE 1: 10 packets, all even length ===");
ok = m.randomize() with {
  numPackets == 10;
  (otherPacketLength & 1) == 0;
  (lastPacketLength & 1) == 0;
};
m.print();

$display("\n=== EXAMPLE 2: 2000..3000 bytes,",
      "all packets 120 bytes except last ===");
ok = m.randomize() with {
  otherPacketLength == 120;
  messageLength inside {[2000:3000]};
};
m.print();

$display("\n=== EXAMPLE 3: all packets 127 bytes ===");
ok = m.randomize() with {
  lastPacketLength==127;
  otherPacketLength==127;
};
m.print();
```

Code Example 9-2: Randomizing the message class to meet various requirements

In each case we only needed to write a few simple lines of constraints to describe our requirements, and all the other rules for creating a legal message are handled automatically for us by the constraint solver and the baseline rules that we captured in constraints of the Message class. Here are the results from VCS:

```
=== UNCONSTRAINED ===
Message has 1497 bytes over 24 packets
    23 packets of 65 bytes, one packet of 2 bytes

=== EXAMPLE 1: 10 packets, all even length ===
Message has 436 bytes over 10 packets
    9 packets of 36 bytes, one packet of 112 bytes

=== EXAMPLE 2: 2000..3000 bytes, all packets 120 bytes except the last ===
Message has 2291 bytes over 20 packets
    19 packets of 120 bytes, one packet of 11 bytes

=== EXAMPLE 3: all packets 127 bytes ===
Message has 2540 bytes over 20 packets
    All packets have length 127
```

Points to note about the constraints in the example above

The example in the previous section has some features in its constraints that are not obvious at first glance, but the authors found were essential to make things work well.

Additional constraints required to avoid integer overflow

The constraint `numPackets<=messageLength` appears to be redundant, because constraint `c_totalOK` is designed to control the relationship between number of packets, packet size and message length. However, without the first constraint we got completely ridiculous values. This odd behaviour was caused by integer overflow. The constraint solver could choose absurdly large values for packet size and number of packets, and thanks to arithmetic overflow the condition in constraint `c_totalOK` would still be satisfied. Here is an example result:

```
Message has 2268 bytes over 3249236149 packets
   3249236148 packets of 115 bytes, one packet of 0 bytes
```

This kind of arithmetic overflow is a major pitfall when writing non-trivial constraints and the authors have encountered it on many occasions. It is usually quite easy to work around by adding some simple "ultimate limit" constraints, as we have done here.

Avoid constraining both an array's size and the sum of its elements

Initially we tried to represent the packets as a dynamic array of packet lengths (one element per packet), with a constraint forcing the total length to be correct:

```
rand int unsigned packetLengths[ ];
constraint c_noCrazyOverflow { packetLengths.size() <= messageLength; }
constraint c_correctTotal { packetLengths.sum() == messageLength; }
```

Code Example 9-3: Problematic constraints controlling both an array's size and its contents

However, this gave some surprising results in VCS and in other simulators. When a dynamic array is randomized, its size will be changed (re-randomized) *only if there is an explicit constraint on its size* ([2] clause 18.4, fifth bullet point). We were aware of this, and provided constraint `c_noCrazyOverflow` partly to deal with the overflow problem mentioned above,

and partly to ensure randomization of the array's size. However, this approach proved unworkable because the array's size is established very early in the randomization process, without concern for the exact contents of the array. The element values are randomized later, after the size has been fixed. This makes it very difficult to establish constraints that control the relationship between the number of elements in the array and properties of its elements (such as their sum). Consequently we found that this approach led either to constraint contradictions or to unrealistic distributions of values, and we were unable to find a really satisfactory solution. Instead we adopted the simpler approach shown in the example, with a common size for all but the last packet.

Use of soft constraints

Constraint `c_avoidVeryShortPackets` makes use of the new *soft* constraint feature, introduced in SystemVerilog 2012 (although a similar feature had been available for some time in VCS as a proprietary language extension). This allows us to write constraints that enforce typical behaviours in most situations, but can be silently overridden by other constraints if required. Without this constraint, we found the constraint solver would sometimes provide solutions with very large numbers of very small packets. We added the soft constraint to avoid this unrealistic behaviour. An additional constraint that enforces rather short packets will contradict this constraint, but because it is soft the contradiction is accepted without error and we get the desired result. Adding this code to the test case:

```
$display("\n=== VERY SHORT PACKETS ===");
ok = m.randomize() with {otherPacketLength < 10;};
m.print();
```

Code Example 9-4: Additional constraints that may supersede a soft constraint

yielded this result without error:

```
=== VERY SHORT PACKETS ===
Message has 11 bytes over 4 packets
   3 packets of 3 bytes, one packet of 2 bytes
```

10. Conclusions

In this paper we have demonstrated some useful techniques for using the power of the constraint solver in order to greatly reduce the amount of code required to create some complex checkers and as a means to describe relationships between properties of an object as well as create interesting sequences of objects such as packetizing messages. The recent introduction of soft constraints into the SystemVerilog standard was found to be particularly useful in these examples.

11. References

[1] Contradiction Analysis for Constraint Random-based Simulation. Große et al http://www.informatik.uni-bremen.de/agra/doc/konf/08_fdl_overconstr_analysis.pdf

[2] IEEE Std.1800-2012