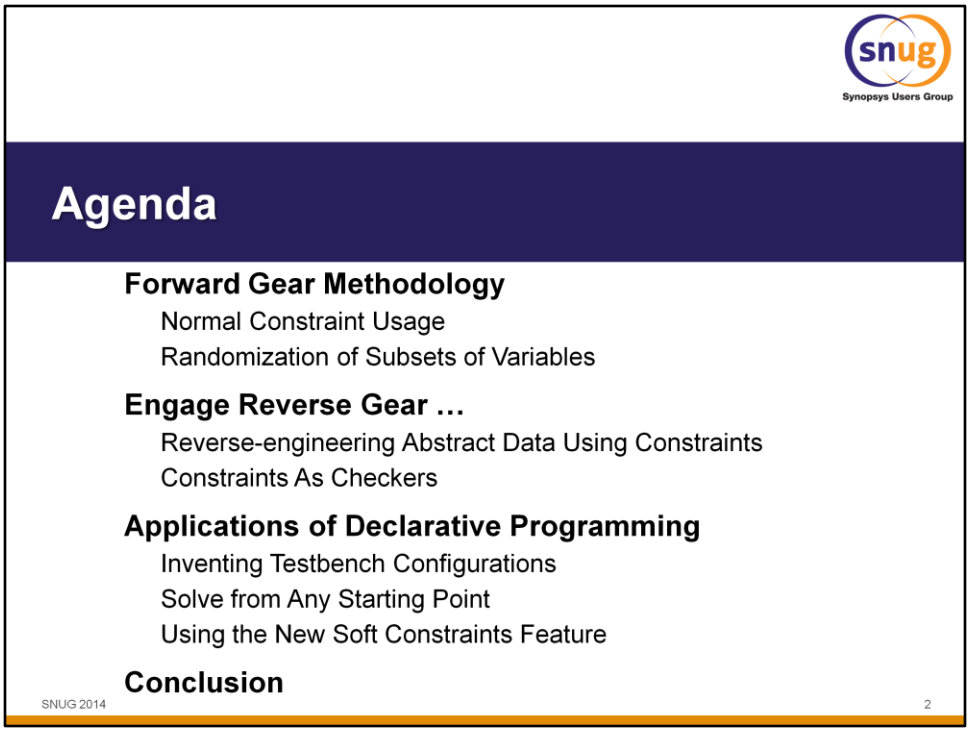
Reverse Gear:

Re-imagining Randomization Using the VCS Constraint Solver

Paul Marriott – Verilab Canada Inc., Montreal
Jonathan Bromley – Verilab Ltd, Oxford, UK

May 2014
SNUG Europe



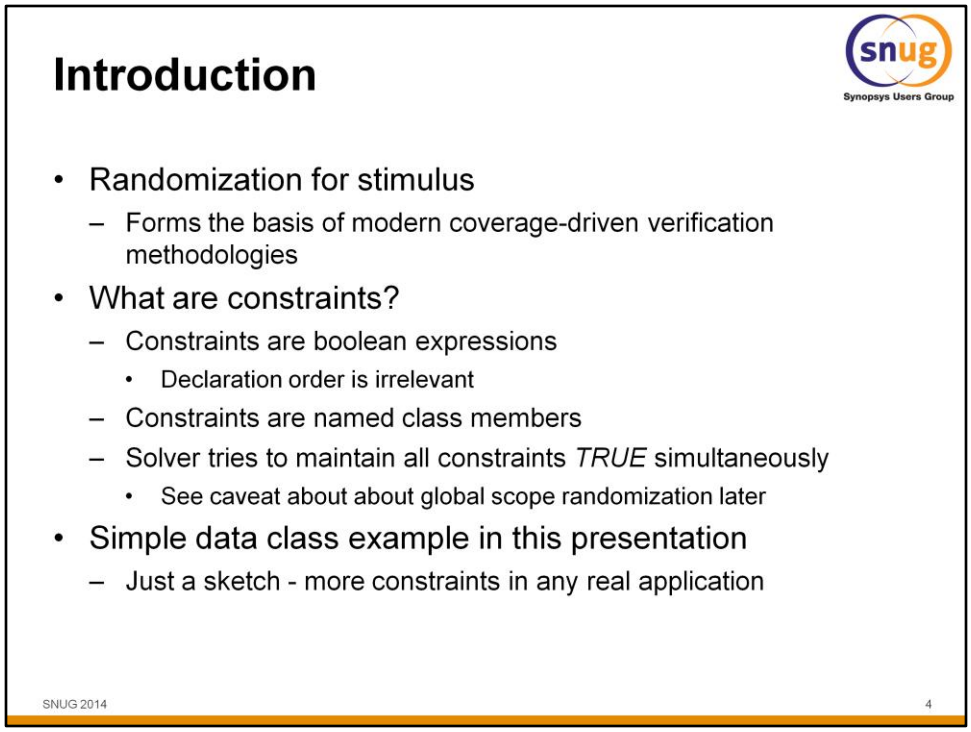
First we will take a brief look at the conventional use of randomization to generate constrained stimulus. That raises an interesting question about how to randomize only a subset of a class's variables, something that can occasionally be rather useful.

We will then move on to see how the constraint solver can be used to "reverse engineer" some unknown parts of an object, given the known values of some other parts. We will also examine how the solver can be used to check an object for legality, using constraints to express validity rules.

Finally we'll take a look at other unusual applications of the constraint solver, and introduce the useful new "soft constraints" feature that was added in SystemVerilog-2012 and is already fully supported by VCS.

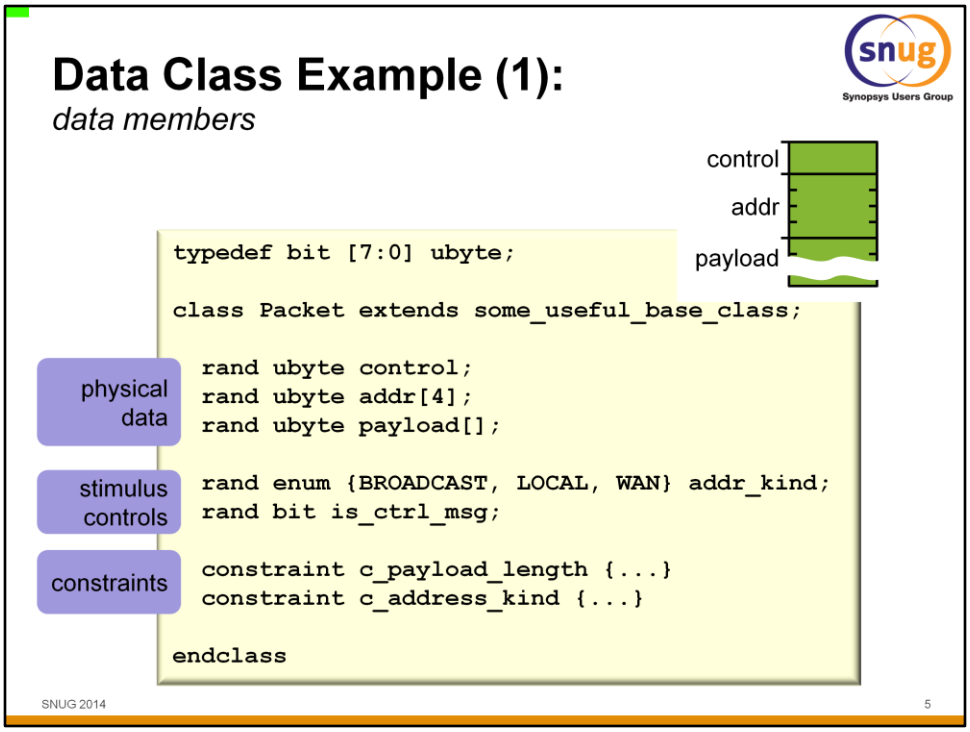
Forward Gear Randomization

- *Typical Constraint Usage*
- *Randomizing Individual Variables*
- *Randomizing a Subset of Class Properties*



For sure, everyone at SNUG is aware that constrained random stimulus generation is one of the cornerstones of modern coverage-driven verification methodology. We use *constraints* to express restrictions, relationships, and statistical distributions that we wish to enforce on the data members of a class (randomization and constraints generally work best in the context of classes, although you can use them on non-class variables too). Generally the constraint solver will generate a set of values (a *solution*) that satisfies all your constraints, although there's a gotcha if you try to use std::randomize (scope randomization) instead of class-based object randomization.

For the first part of this presentation we'll use a simple data class, with lots of features missing to save time and space!

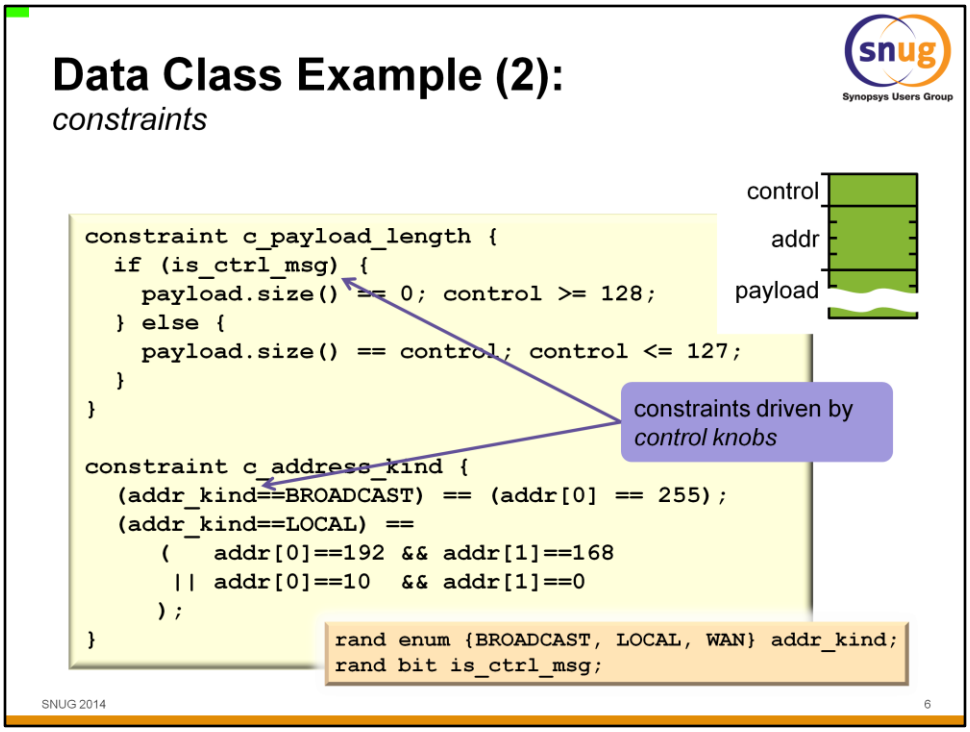


Here's our example data class. Like everyone else, we use a network message packet example - with *lots* of features absent! The class will probably be extended from some base class such as *uvm_sequence_item*, but that doesn't really concern us here.

click
Here's the real, physical packet data that will appear on the DUT pins or the network data stream. We have a control byte that specifies the payload length, a 4-byte address (think IPv4), and a variable-length payload that may even have zero bytes in it. All these data members are declared **rand** so that they can be randomized if we so wish.

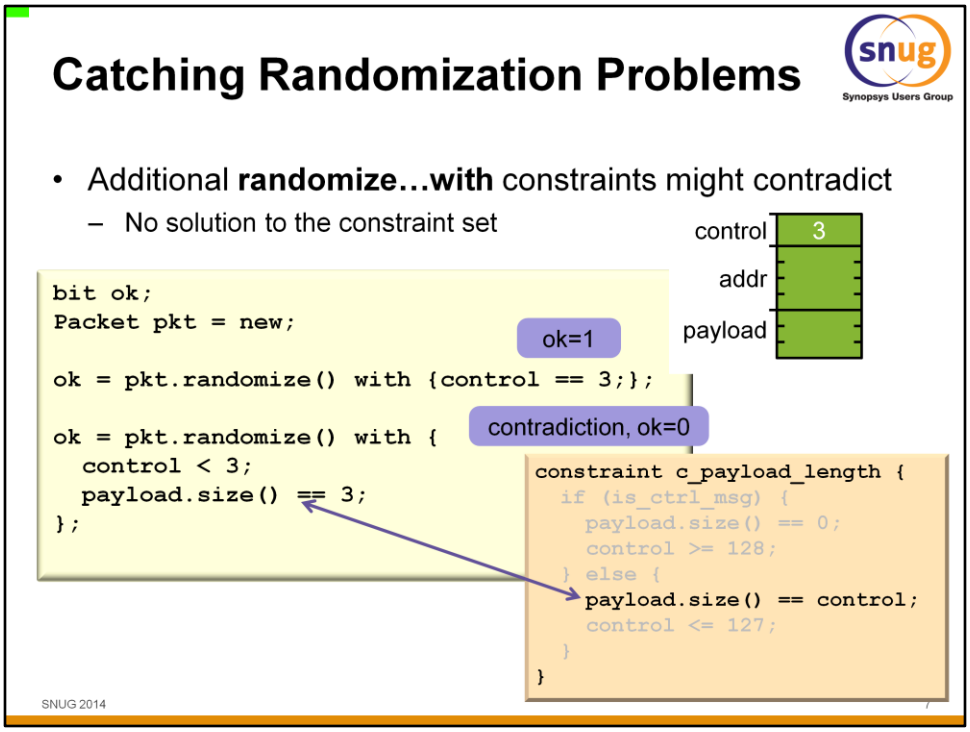
click
Our class, being part of a verification environment, also has some non-physical information (often called *metadata*) that can be used in various ways by the testbench to reflect interesting properties of the object. It does not appear on the physical DUT pins or data stream, but is closely linked to the real data. These values, like the physical values, are declared **rand** so that we can manipulate them using the constraint solver.

click
Finally, we need some *constraints* to control randomization. Constraints have names; we've used the common (but of course optional) conventions of using a **c_** prefix to highlight their purpose and avoid collisions with variable names.

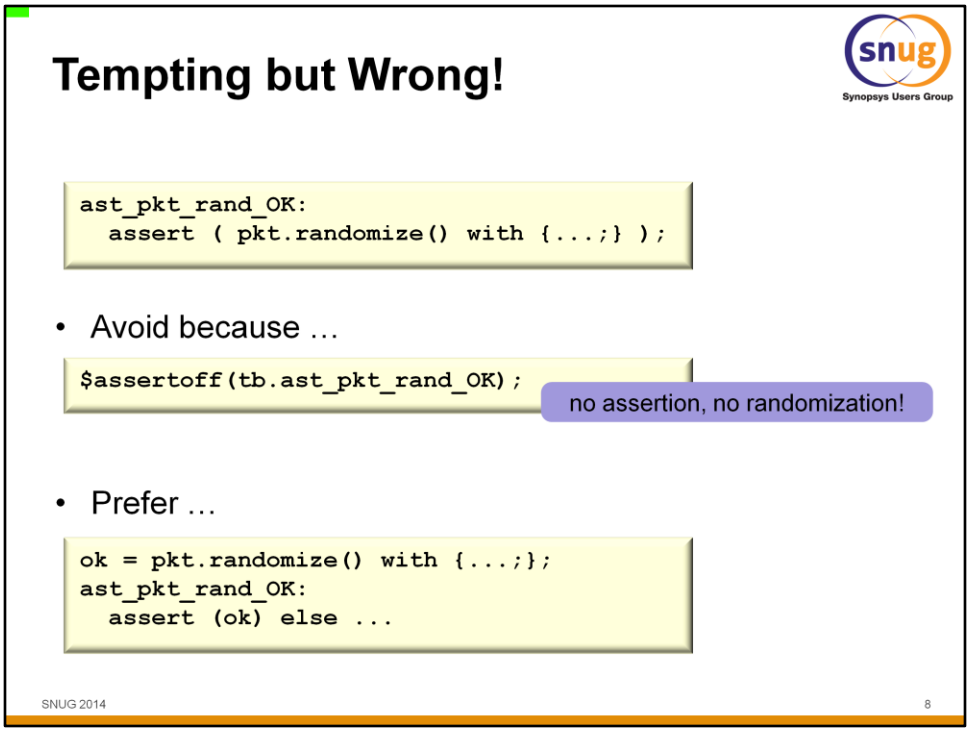


In the above example, there are two constraint members, named c_payload_length and c_address_kind.

Note that we have avoided using implication constraints (which are of the form expr1 -> expr2) in the c_address_kind constraint because addr[0]==255 would be a valid solution even if addr_kind!=BROADCAST
Instead, we use an "if and only if" constraint, expr1 == expr2.



When you use this object, you are free to add further constraints in a *with* block. Here we attempt to use this to create an illegal packet that violates the rule relating the payload length to the control value. However, the constraint solver won't allow us to do this! Because there is a *contradiction* between the two constraints, the solver refuses to update *any* of the rand data members, and returns zero. We can capture this error code in a variable, as shown here.



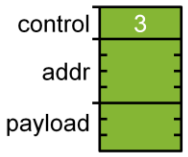
We've often seen it suggested that you should *assert* that randomization succeeds, as shown here. I'm ashamed to say that I have even recommended it myself in the past. But this is a bad idea!

click
Assertions can be completely disabled by tool command options or by system calls such as **$assertoff**. If you do this to an immediate assertion, the expression it tests is never evaluated so *the randomization doesn't happen at all*!

click
Instead, capture the randomization success code in a variable (bit, reg, int - anything will do!). You can then use an assertion, if you wish, to throw the error message - it's a great way to get a clear, comprehensive diagnostic.

Subverting Class Constraints?

- std::randomize() ignores class constraints
- cannot use on data member via object handle




```
bit ok;
Packet pkt = new;
ok = pkt.randomize() with { control == 3; };

ok = std::randomize(pkt.control) with { control < 3; };
```

not supported by VCS

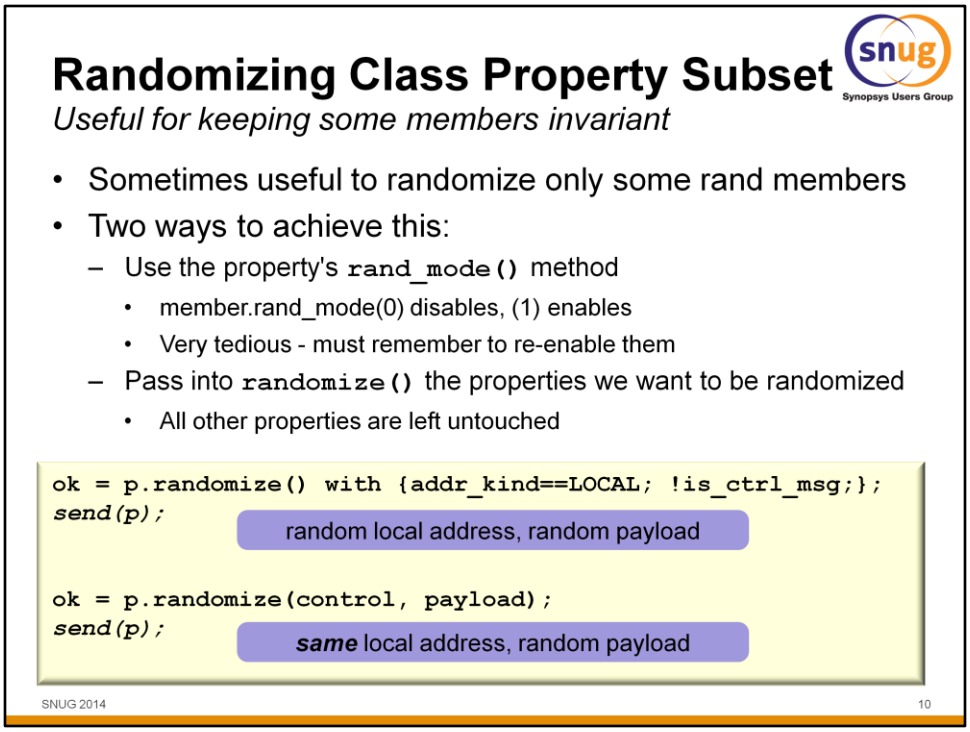
```
ok = pkt.randomize(control) with { control < 3; };
```

contradiction!

- class constraints are **always** respected by *obj*.randomize()
- disable constraints with constraint_mode() ?

Let's return to the problem of how to subvert the existing constraints to create an illegal set of values.

The first randomize call creates a packet with control==3 and 3 bytes of payload. Now let's try to mess with the control value alone. We can do this by using std::randomize instead of object randomization - it ignores all your class's constraints, so we can do anything we like by applying our own constraints. However, this doesn't work because VCS doesn't support calling std::randomize() on a class member. An alternative approach might be to supply the name of the single variable you want to randomize - *control* - as an argument to class randomize; we'll be saying much more about that later. But this doesn't work, because all the constraints are still active. You can disable some or all constraints using **constraint_mode**, but that's quite unpleasant - it's far too easy to leave them disabled by mistake, and with a complex constraint set it can be very hard to know exactly which constraints to switch off and which to leave active.



The ability to specify a subset of a class's variables, as arguments to the object's randomize call, is very useful for other reasons.

The first randomization 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, but with a different payload. 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.

Instead, we can ask the solver to randomize only the control and payload values. It will still respect all the constraints, but it leaves the address value unaffected.

Encapsulate Specialized Randomization

- Awkward to remember ***what*** to randomize
- Consider encapsulating as a class method

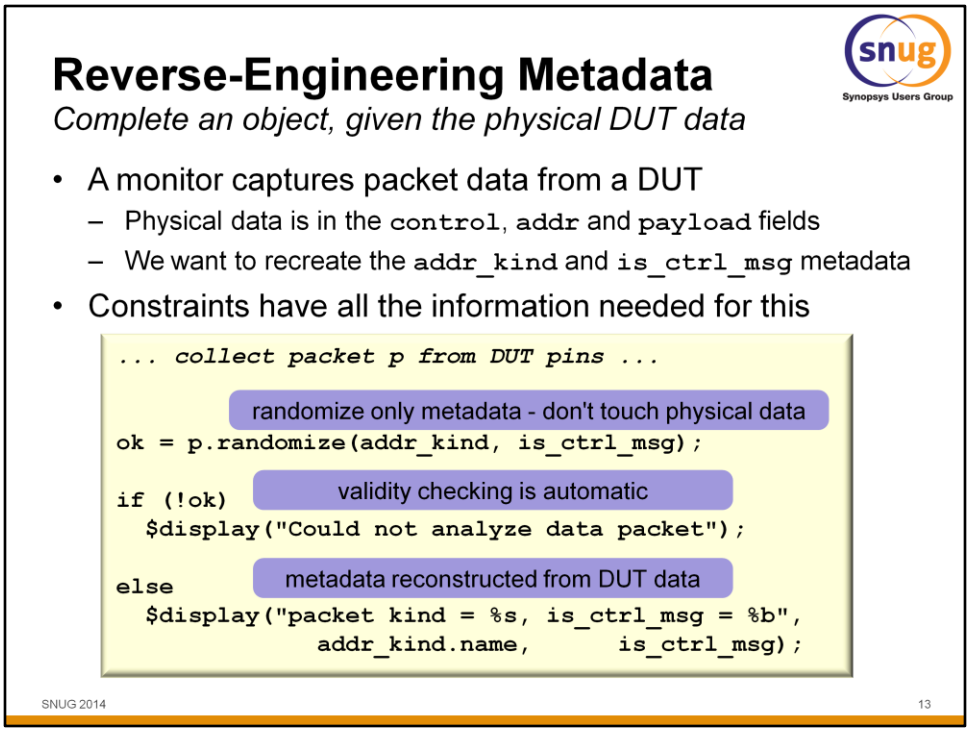
```
class Packet extends some_useful_base_class;
  ...

  function bit randomize_payload_only();
    return this.randomize(control, payload);
  endfunction

endclass
```

+ Neat encapsulation

- Cannot add with-constraints

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. Unfortunately, this encapsulation makes it impossible to add custom *with* constraints on any call to the specialized function.

Now that we have a good sense of how the solver operates when asked to randomize only a subset of a class's variables, we can explore how to apply that idea to some interesting problems.



In this example, we consider how the packet data class could be used in monitoring code. First, the verification environment collects physical data (control, address, payload) from the DUT pins. There's no randomization here, of course - we simply write to the data members with ordinary assignment.

That's fine, but now our packet is incomplete; the metadata fields **is_ctrl_msg** and **addr_kind** contain garbage with no relation to the real data. This is inconvenient and inconsistent. However, reconstructing the metadata isn't trivial - just think of the long chain of if-tests that would be needed - and it feels nasty that we have already described the relationships in our constraints!

Here's how we do it with the help of the solver. We just randomize the metadata fields – all the others are held constant, and the constraints force the solver into finding metadata values that are consistent with the physical values. We get checking as well – if randomization fails, it means the received data was somehow ill-formed and there is no set of metadata values that satisfy all the constraints.

Constraints as Checkers

Validity Specified by Set of Active Constraints

- Check values against constraints:

```
if ( p.randomize(null) ) …
```

no values affected

- Additional constraints for test-specific limits:

```
// receive p from the DUT
…
ok = p.randomize(null) with {
  addr[0] inside {[1:127]};
};
```

will this work?
metadata?

There's an extreme version of checking, in which we ask the constraint solver to change *nothing*, but instead merely to check that all the data members in the object collectively satisfy the active constraints. I suppose, if you were desperate, you could use constraint_mode(0) to judiciously turn off the constraints we're not interested in…

I hope you found the idea of reconstructing metadata to be interesting and novel. However, it's not the only creative way to use the solver. Once you've recognized that the solver is just that - a tool for finding a set of values that meet the requirements specified by your constraints - you can easily adapt it for use as a rather general way to find solutions to various numerical problems.

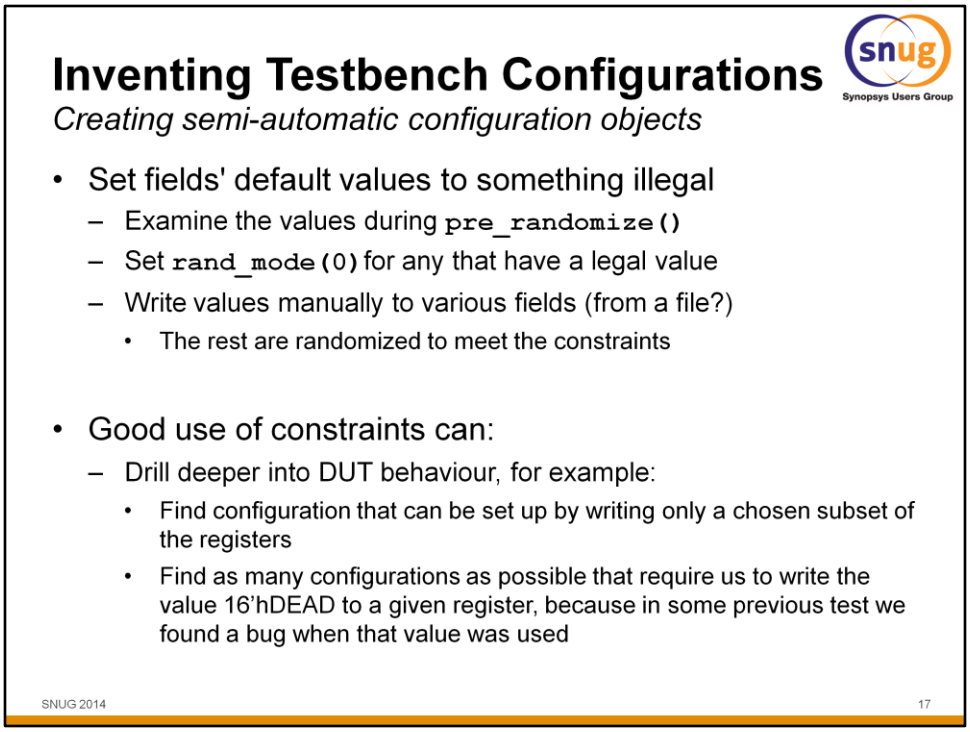
Inventing Testbench Configurations

Creating interesting partially-randomized configuration objects

- Requirements often demand a variety of features
  - e.g with/without cache
  - 64b/32b data
- The features are often interrelated
  - Cache >= 512KiB on 64b systems, only 256 or 512KiB for 32b
  - Ideal application for constraints
- Can use `randomize() with{ }` to fix certain values
  - Using the techniques described so far for randomizing subsets
    - The base constraints apply as well as the ones specified in the `with{ }` clause
    - Constraint solver produces valid values for all other fields

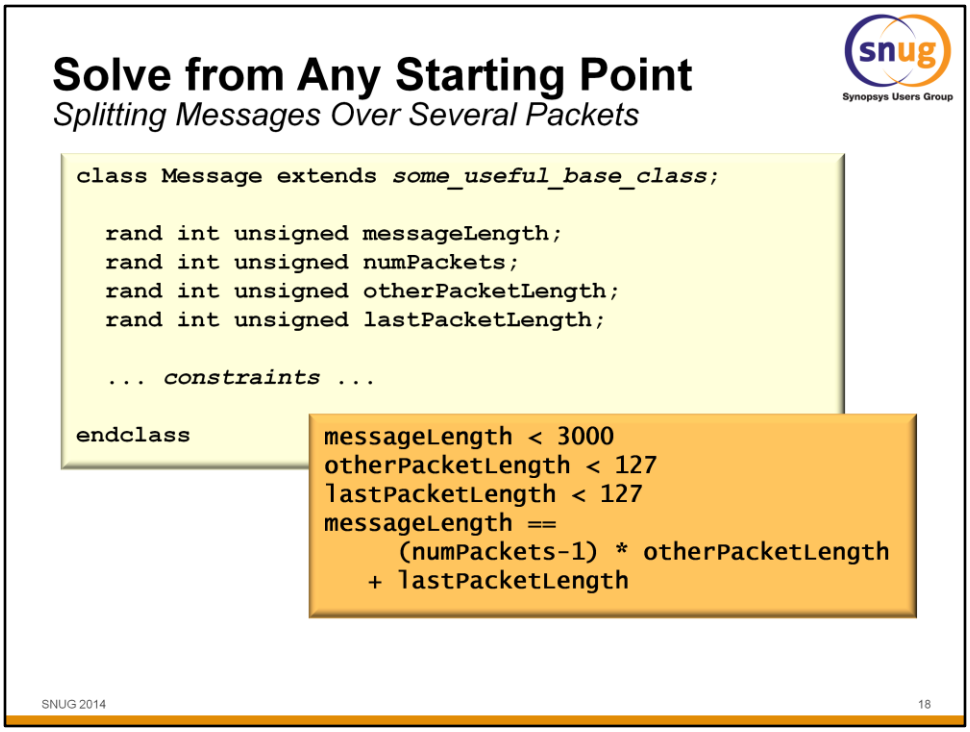
One of our favourite applications is to generate interesting testbench configurations. Your testbench probably has one or more configuration objects that control the operation of the whole verification environment. It's great to randomize these configurations. Sometimes, of course, the configuration reflects static features of the DUT structure such as number of ports, size of memory, absence or presence of certain features... but randomization can still be useful: run SV once to generate an interesting random configuration, and from that configuration generate a small piece of SV source code specifying the DUT parameter values - you can then recompile the DUT with these parameters in place.

The configuration will naturally have constraints or relationships among its data members, as hinted on the slide. These are of course a good candidate for constraints, and we can use additional **with** constraints to drive the solver to a configuration of interest.



Here's an idea that you may find interesting. Sometimes, configuration values can be picked up from a file. How do we reconcile that with the need to randomize? You can't create constraints from a file in any simple way. One possibility is to pre-set all configuration values to some illegal value (for example, most config values are positive or zero; any negative value is obviously illegal). Then, in the pre_randomize method, set rand_mode(0) for any data member that *doesn't* have an illegal value (because it's been set already by reading from a file). Randomization will then touch only the un-set values, while honouring all constraints.

Constraints can also allow us to work backwards. Suppose we have configuration set from register values. Our constraints describe how to determine the right register values from the configuration. If we now constrain one or more of the register values, we will only find configurations that require the specific register values! More generally, we can constrain the *outputs*of the solver, just as easily as the *inputs.*



Suppose we have to construct a message from multiple packets. There are of course many ways to break up the messages into packets, but a few rules describe the relationships (we've written them here as simple expressions rather than as constraints, simply to save space).

Sometimes we may want to start with a specific number of packets, and compute the packet sizes. Other times, we wish to specify the packet length, and compute how many packets are required. If we were to do those things procedurally we would need two separate pieces of code, both capturing the same set of relationships.

But the solver can do it with only *one* set of relationships, the constraints!

Solve-from Any Starting Point

Different Scenarios

- Scenario requirements:
  1. Message of exactly 10 packets all with even number of bytes
  2. Message with between 2000 and 3000 bytes
     - All packets except the last should be 120 bytes
  3. A variety of message sizes, but all packets must be 127 bytes

- We could write procedural code for all three scenarios
  - The code would be *different* for each case!
- We can instead write constraints for the relationships
  - Now we can just call randomize with the additional constraints for each scenario
  - Perfect code reuse for the message and packet generation

Here are three different requirements illustrating the point from the previous slide.

Solve-from Any Starting Point

Splitting Messages Over Several Packets

- Paper gives details of suitable constraints
  - Pitfalls from arithmetic overflow - needs "sanity" constraints
  - Avoid large numbers of short packets using soft constraints

```
Message msg = new;
bit ok;

$display("=== UNCONSTRAINED ===");
ok = msg.randomize();
msg.print();
```

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


The written paper describes our solution to this problem in detail, together with some not-so-obvious "gotchas" that you need to be aware of. Here we'll just look at a few examples of the idea in use.

This code gives us full randomization that nevertheless obeys the basic rules.

Solve-from Any Starting Point

Splitting Messages: Scenario 1

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

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

In this example we insist that we have exactly ten packets, and also that all packets have an even number of bytes. The solver finds a solution that still conforms to the basic rules; we didn't need to write any code at all except that required to specify our special requirements. Perfect code reuse.

Solve-from Any Starting Point

Splitting Messages: Scenario 2

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

```
=== 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
```

And another example, again with no coding effort.

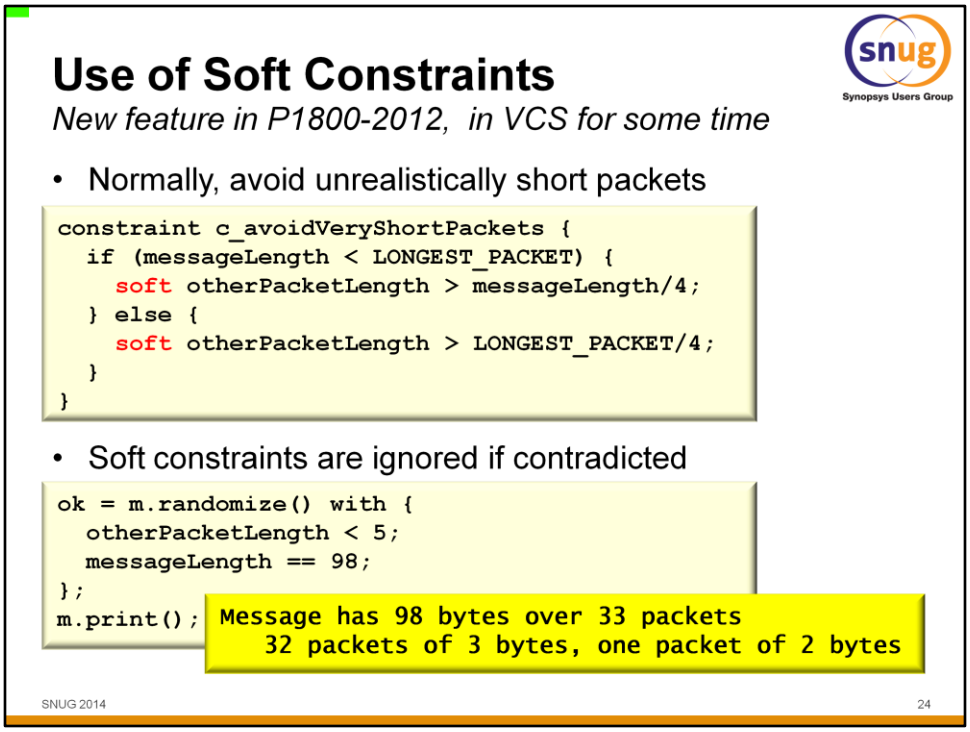
Solve-from Any Starting Point

Splitting Messages: Scenario 3

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

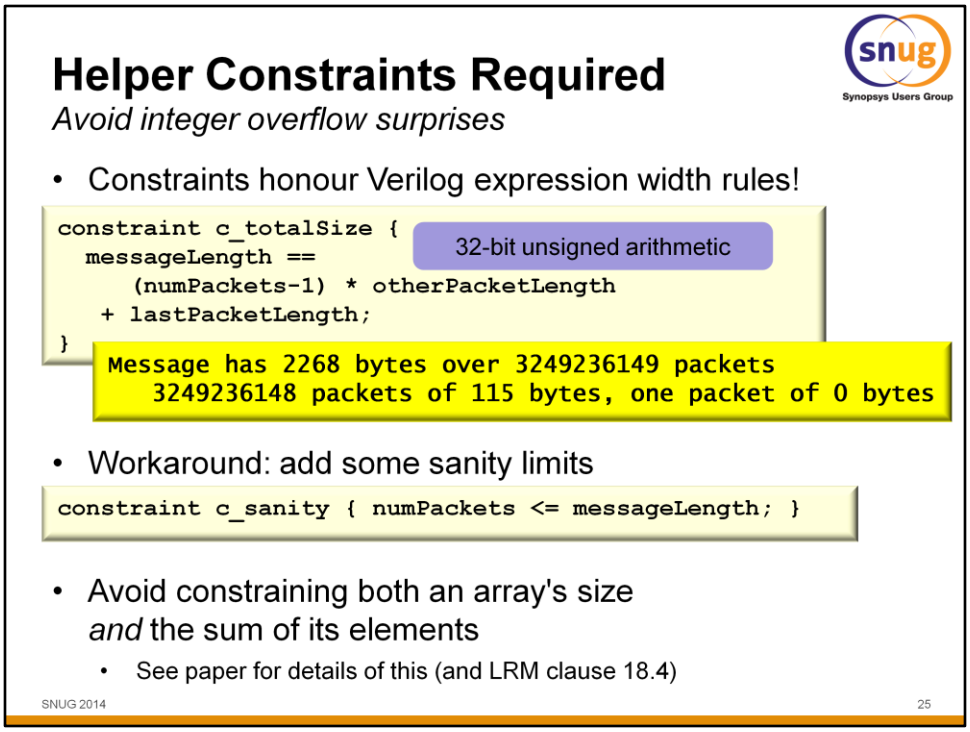
```
=== EXAMPLE 3: all packets 127 bytes ===
Message has 2540 bytes over 20 packets
    All packets have length 127
```

And yet another possible requirement, again with no coding effort.



When we were developing the example for this last section of the presentation, we found a little problem: the solver would occasionally give us a solution that, while legal according to the basic rules, contained some unreasonably short packets. We don't want to *exclude* the possibility of short packets, but we would prefer not to get them in the default case. It's quite hard to do this with regular constraints, but SV-2012 introduced *soft constraints* for precisely this purpose. A soft constraint works exactly like a regular constraint, *unless it's contradicted* by some other constraints that you apply later.

VCS had soft constraints (in almost exactly the form mandated in SV-2012) for several years, and now it fully supports the new features.



Here is one of the most common constraint gotchas. Take a look at our "total size" constraint - nothing wrong there, surely? It just captures the obvious arithmetical relationship between packet sizes, packet count and message length. But *it doesn't work*!!! The reason is *arithmetic overflow*. The constraint has 32-bit operands everywhere, so it's evaluated using 32-bit arithmetic according to the standard Verilog rules. In that world, 3249236148x115=2268, because

```
3249236148 x 115 = 373662157135 = 'h57_0000_08DC
```

and of course the bottom 32 bits of this value is `32'h0000_08DC = 2268`.

There's also a slightly unpleasant problem about constraints relating the size of an array to the value of its elements. Because SV specifies that the solver must establish the size of an array once and for all, *before* attempting to randomize the values of its elements, the solver may create array sizes that lead to not very useful data values, or even cause the remaining constraints to be insoluble. The paper gives more information on how to handle this.

Conclusions

- Creative use of constraints and the solver can save a *lot* of manual work
  - Creating reusable checkers
  - Reconstructing control knob values
  - Generating testbench configurations
  - Creating interesting scenarios without complex coding

- Not so much *assigning random values* to variables …
- … instead, **enforcing a set of rules** over the data


verilab
snug
Synopsys Users Group
Thank You
Any Questions?