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

Agenda

Forward Gear Methodology
- Normal Constraint Usage
- Randomization of Subsets of Variables

Engage Reverse Gear …
- Reverse-engineering Abstract Data Using Constraints
- Constraints As Checkers

Applications of Declarative Programming
- Inventing Testbench Configurations
- Solve from Any Starting Point
- Using the New Soft Constraints Feature

Conclusion

Forward Gear Randomization

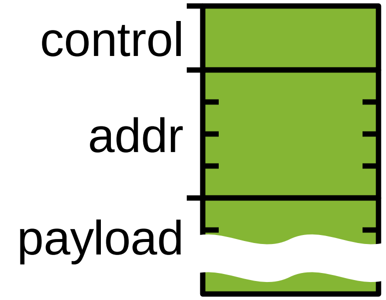
- *Typical Constraint Usage*
- *Randomizing Individual Variables*
- *Randomizing a Subset of Class Properties*

Introduction

- Randomization for stimulus
  - Forms the basis of modern coverage-driven verification methodologies
- What are constraints?
  - Constraints are boolean expressions
    - Declaration order is irrelevant
  - Constraints are named class members
  - Solver tries to maintain all constraints *TRUE* simultaneously
    - See caveat about about global scope randomization later
- Simple data class example in this presentation
  - Just a sketch - more constraints in any real application

Data Class Example (1):

data members



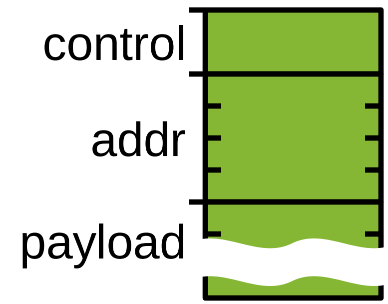

```
typedef bit [7:0] ubyte;

class Packet extends some_useful_base_class;

  rand ubyte control;
  rand ubyte addr[4];
  rand ubyte payload[];

  rand enum {BROADCAST, LOCAL, WAN} addr_kind;
  rand bit is_ctrl_msg;

  constraint c_payload_length {...}
  constraint c_address_kind {...}

endclass
```

physical data

stimulus controls

constraints

Data Class Example (2):
constraints



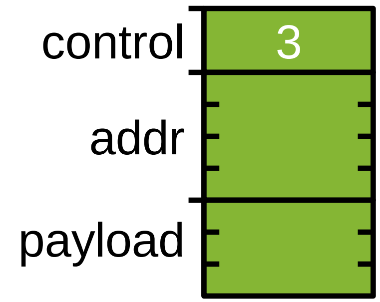

```
constraint c_payload_length {
  if (is_ctrl_msg) {
    payload.size() == 0; control >= 128;
  } else {
    payload.size() == control; control <= 127;
  }
}

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
     );
}
```

constraints driven by *control knobs*

```
rand enum {BROADCAST, LOCAL, WAN} addr_kind;
rand bit is_ctrl_msg;
```

Catching Randomization Problems

- Additional **randomize…with** constraints might contradict
  - No solution to the constraint set




```
bit ok;
Packet pkt = new;

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

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

ok=1

contradiction, ok=0

```
constraint c_payload_length {
  if (is_ctrl_msg) {
    payload.size() == 0;
    control >= 128;
  } else {
    payload.size() == control;
    control <= 127;
  }
}
```

Tempting but Wrong!

```
ast_pkt_rand_OK:
  assert ( pkt.randomize() with {...;} );
```

- Avoid because …

```
$assertoff(tb.ast_pkt_rand_OK);
```

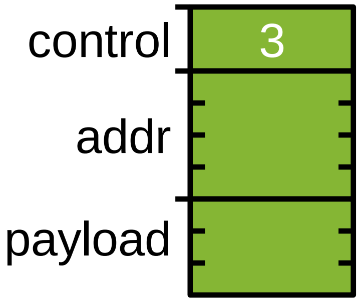
no assertion, no randomization!

- Prefer …

```
ok = pkt.randomize() with {...;};
ast_pkt_rand_OK:
  assert (ok) else ...
```

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() ?

Randomizing Class Property Subset

Useful for keeping some members invariant

- Sometimes useful to randomize only some rand members
- Two ways to achieve this:
  - Use the property's **`rand_mode()`** method
    - member.rand_mode(0) disables, (1) enables
    - Very tedious - must remember to re-enable them
  - Pass into **`randomize()`** the properties we want to be randomized
    - All other properties are left untouched

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

random local address, random payload

```
ok = p.randomize(control, payload);
send(p);
```

same local address, random payload

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

Engage Reverse Gear …

- *Reverse engineering Metadata*
- *Constraints as Checkers*

Reverse-Engineering Metadata

Complete an object, given the physical DUT data

- A monitor captures packet data from a DUT
  - Physical data is in the **`control`**, **`addr`** and **`payload`** fields
  - We want to recreate the **`addr_kind`** and **`is_ctrl_msg`** metadata
- Constraints have all the information needed for this

```
... collect packet p from DUT pins ...

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);
```

randomize only metadata - don't touch physical data

validity checking is automatic

metadata reconstructed from DUT data

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?

Applications of Constraint Programming

- *Inventing testbench configurations*
- *Solve from any starting point*

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

Inventing Testbench Configurations

Creating semi-automatic configuration objects

- Set fields' default values to something illegal
  - Examine the values during **`pre_randomize()`**
  - Set **`rand_mode(0)`** for any that have a legal value
  - Write values manually to various fields (from a file?)
    - The rest are randomized to meet the constraints

- Good use of constraints can:
  - Drill deeper into DUT behaviour, for example:
    - 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

Splitting Messages Over Several Packets

```
class Message extends some_useful_base_class;

  rand int unsigned messageLength;
  rand int unsigned numPackets;
  rand int unsigned otherPacketLength;
  rand int unsigned lastPacketLength;

  ... constraints ...

endclass
```

```
messageLength < 3000
otherPacketLength < 127
lastPacketLength < 127
messageLength ==
     (numPackets-1) * otherPacketLength
   + lastPacketLength
```

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

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
```

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();
```

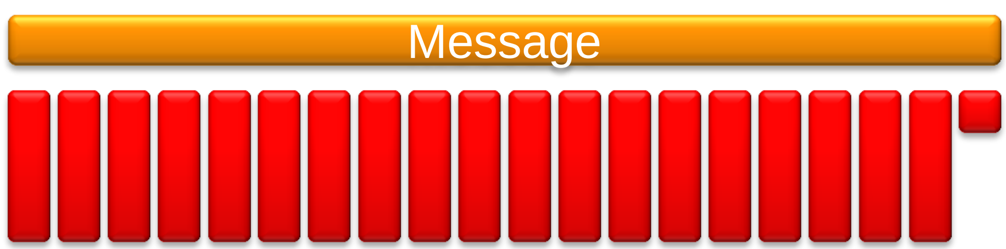
```
=== EXAMPLE 1: 10 packets, all even length ===
Message has 436 bytes over 10 packets
    9 packets of 36 bytes, one packet of 112 bytes
```

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
```



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
```

Use of Soft Constraints

New feature in P1800-2012, in VCS for some time

- Normally, avoid unrealistically short packets

```
constraint c_avoidVeryShortPackets {
  if (messageLength < LONGEST_PACKET) {
    soft otherPacketLength > messageLength/4;
  } else {
    soft otherPacketLength > LONGEST_PACKET/4;
  }
}
```

- Soft constraints are ignored if contradicted

```
ok = m.randomize() with {
  otherPacketLength < 5;
  messageLength == 98;
};
m.print();
```

```
Message has 98 bytes over 33 packets
   32 packets of 3 bytes, one packet of 2 bytes
```

Helper Constraints Required

Avoid integer overflow surprises

- Constraints honour Verilog expression width rules!

```
constraint c_totalSize {
  messageLength ==
      (numPackets-1) * otherPacketLength
   + lastPacketLength;
}
```

32-bit unsigned arithmetic

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

- Workaround: add some sanity limits

```
constraint c_sanity { numPackets <= messageLength; }
```

- Avoid constraining both an array's size *and* the sum of its elements
  - See paper for details of this (and LRM clause 18.4)

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

Thank You

Any Questions?