Accellera VHDL-TC Extensions-SC

Randomization

Jim Lewis, SynthWorks

jim@synthworks.com

Version 1.0 Draft, 24 Sept 2008

Abstract

This paper explores ideas for implementation of randomization in VHDL.

Revision History

Version 1.1 24-Sept-08, Jim Lewis. Revised.

Version 1.0 26-Mar-07, Jim Lewis. Initial work in progress draft.

1. Motivation

The rationale for bringing randomization and other verification constructs into VHDL is to remove the language barrier between verification and design engineers. While verification and design techniques may be different, familiarity with the language will help bridge the gap. While language constructs are being added to VHDL, it is important to realize there are several existing verification languages and offering similar constructs and capability is important so that similar and/or common methodologies can be utilized.

Many designs contain numerous configurable features. Testing features individually in an isolated manner is typically straightforward. However, testing how these features interact can be a large verification space – one that may not be able to be simulated completely. It is also may be difficult to predict all of the corner cases. Randomization has been used to sequence a test in a non-deterministic way to get reasonably good coverage of this verification space.

Each randomization covers a set of values such as the values used in a transaction or the mix of transactions to be used on an interface. As a result, relationships need to be expressed between different items. These relationships can be expressed either procedurally with code or declaratively. Existing verification languages use both approaches.

Procedural based approach can use the same process structure and transaction based procedures used by a directed test approach. Basic randomization capability is used to randomize each item separately. Items may be transaction types (controlling the sequencing), transaction values (including error injection), delays, and other items.

In a declarative approach, typically a class is used to bundle items together such that relationships between them can be expressed. In addition, a class offers the benefits being OO extensible and including methods with the objects. Sequencing and transaction value generation become part of the class. Except when the class value need to be overridden, the process generating the test sequence becomes very generic – one has to focus on what is in the class to know what the test is doing.

Procedural approach gives a great degree of readability in a familiar environment for a verification team currently using a directed approach. In a procedural approach, randomization order (determined by order of code) and relative weights (frequency of one item being picked vs another) can be used to accomplish many useful test cases. A declarative approach allows constraints on multiple items can be randomized simultaneously and, hence, gives a uniform distribution across an entire transaction. Using OO in a declarative approach allows one to either build on or remove existing constraints – hence, new tests can be generated with minimal changes.

2. Design of Randomization Language Features

In order for VHDL randomization features to be compatible with existing methodologies as much as possible will be leveraged from other verification languages including SystemVerilog. As we do this, features can be either a direct translation with minimal changes, or they can be adapted some to better suit VHDL. It is my intent to consider both options.

Since there are candidates for the syntax to be considered, examples will be provided. Syntax productions will be added once we agree on syntax.

3. Basic Randomization

The intent of basic randomization is to provide the occasional single random value. While it can be used for randomizing transactions, using class-based randomization is more effective at expressing constraint relationships between multiple items.

3.1. Baseline

Randomization uses a seed as the basis for generating the next value. To avoid maintaining the seed externally, the seed and randomization functions can be encapsulated in a class (note don't confuse this with class-based randomization which is presented later).

```
Package RandomPkg is
  type SeedType is record
    seed1 : positive ;
    seed2 : positive ;
  end record SeedType ;

  type RandClass is class
    variable Seed : SeedType := (7, 1) ;
    procedure seed_random (A : SeedType ) ;
    impure function  seed_random return SeedType ;

    impure function RandReal return real ;
    impure function RandReal(Max: Real) return real ;  -- 0.0 to Max
    impure function RandReal(Min : Real; Max: Real) return real ;
    impure function RandInt (Max: Integer) return integer ; -- 0 to Max
    impure function RandInt (Min : Integer; Max: Integer) return integer ;
    impure function RandSlv (Len: integer) -- 0 to 2**Len - 1
      return std_logic_vector ;
    impure function RandSlv (Max: Natural; Len: integer) -- 0 to Max
      return std_logic_vector ;
    impure function RandSlv (Min : Natural; Max: Natural; Len: integer)
      return std_logic_vector ;
    impure function RandUnsigned (Len: integer) -- 0 to 2**Len - 1
      return Unsigned ;
    impure function RandUnsigned (Max: Natural; Len: integer) -- 0 to Max
      return Unsigned ;
    impure function RandUnsigned (Min : Natural; Max: Natural; Len: integer)
      return Unsigned ;
    impure function RandSigned (Len: integer) -- -2**(Len-1) to 2**(Len-1) - 1
      return Signed ;
    impure function RandSigned (Max: Integer; Len: integer) -- 0 to Max
      return Signed ;
    impure function RandSigned(Min : Integer; Max: Integer; Len: integer)
      return Signed ;
   end class RandClass;
end package RandPkg ;
```

Value based distributions can also be handled in a similar fashion, however, in section Procedural Randomization, we will see that it can be more effectively handled with syntax.

All randomization functions call RandReal for basic randomization. RandReal returns a value between 0.0 and 1.0 (same as ieee.math_real.uniform) and the other functions scale this value to the appropriate range. By extending RandClass and overloading RandReal, different distributions can be generated for all types.

```
  type ExtRandClass extends RandClass is class body
    impure function RandReal return real is
    begin
      return poisson(super.RandReal) ;
    end function RandReal ;
  end class body ExtRandClass;
```

3.2. Tradeoff: Randomize

Name all random functions randomize instead of RandReal, RandInt, RandSlv, RandUnsigned, and RandSigned.

3.3. Tradeoff: Types ufixed, sfixed, and float

How do we handle types ufixed, sfixed, and float? Although they are generic types, the generic values associated with them do not impact the value itself. Instead they impact things when operators are used (such as rounding, …).

3.4. Tradeoff: Visability

SystemVerilog provides $urandom and $urandom_range to randomize integer values. These functions have access to a seed and are thread stable, hence, they are somewhat of an implicit class. The class is associated with a thread and its methods are accessed with function call notation rather than class notation.

In VHDL a variable of type RandClass would need to be created and then its methods would need to be accessed using class method call notation. To be like SystemVerilog, each process would need to implicitly create a variable of type RandClass and calls to method of RandClass would need to be permitted without referencing the variable name. SystemVerilog uses this type of notion already for Procedural Randomization, VHDL will need to do something similar.

4. Class Based Randomization

4.1. Specifying Random Variables to Randomize

4.1.1. Baseline

A class is used to bundle items together such that relationships between them can be expressed. SystemVerilog uses public class variables and marks them with either randc or rand if they are to be randomized by default. Translating this into VHDL:

```
Type CpuOpType is (CpuRead, CpuWrite, CpuIntAck) ;
Type CpuOpGen is class
  Rand Variable CpuOp : CpuOpType ;
  Rand Variable Addr  : std_logic_vector(21 downto 0);
  Rand Variable Data  : std_logic_vector(15 downto 0) ;
End class CpuOpGen ;
```

4.1.2. Tradeoff: Randc and Rand

Some how class state variables (ones not part of randomization) need to be differentiated from random class variables. Randc and rand are as good as a methodology as any. Alternately, these could be done solely with attributes.

```
Type CpuOpType is (CpuRead, CpuWrite, CpuIntAck) ;
Type CpuOpGen is class
  Variable CpuOp : CpuOpType ;
  Attribute Rand of CpuOp is true ;
  Variable Addr  : std_logic_vector(21 downto 0) ;
  Attribute Rand of Addr is true ;
  Variable Data  : std_logic_vector(15 downto 0) ;
  Attribute Rand of Data is true ;
End class CpuOpGen ;
```

To meet other needs of randomization, it would help if the attribute value assigned in a class could be overridden by a specification/assignment outside of the class. Also a shorter notation or a default specification in the variable declaration would be helpful. Perhaps that is how the "Rand Variable CpuOp : CpuOpType;" should be viewed?

4.2. Types to Randomize

4.2.1. Baseline

SystemVerilog requires random class variables to integral types. In VHDL, minimally this should include the types (user defined enumerated types, boolean, bit, bit_vector, std_ulogic, std_logic, std_logic_vector, std_ulogic_vector, unsigned, signed, and integer). For bit types (std_ulogic and std_logic) and arrays of bit types, values generated shall be limited to '0' and '1' and constraints based on values other than '0' and '1 are illegal and shall be an error. For ordering purposes, array types without a numeric representation (bit_vector, std_logic_vector, and std_ulogic_vector) shall be treated as if they are unsigned.

4.2.2. Enhancement: Randomizing Real

The procedure ieee.math_real.uniform already supports randomization of real numbers. There does not seem to be any reason to limit usage to integral numbers – other than perhaps loosing the notion of how many bits are in the representation.

Given the extension to real, it seems logical to also support types ieee.<generic_fixed_pkg_instance>.sfixed, ieee.<generic_fixed_pkg_instance>.ufixed, and ieee.<generic_float_pkg_instance>.float. As already pointed out, the generics do not affect the value represented by the type.

4.3. Randomizing Class Objects

4.3.1. Baseline

In SystemVerilog, objects are randomized using the built-in randomize class pseudo-method. It uses method call syntax (a function returning integer), but has arguments that are beyond what is representable in SV and is not overloadable. This Randomize also has extended syntax that allows a single additional constraint to be specified. The same format can be used for VHDL with the method returning false when it fails:

```
<class_identifier>.randomize [([<variable_list>|null])] with constraint_block ;
```

If the variable_list is specified, only the variables specified in it are randomized (including ones not marked randc or rand). The constraint_block allows additional constraints to be specified. To work around randomize not being overloadable, overloadable methods (procedures) for pre_randomize and post_randomize are provided. The randomization is seeded with the srandom method. External to the class, each variable and constraint may be turned off in mass or individually using methods rand_mode and constraint_mode respectively.

```
randomize <class_identifier> [([<variable_list>|null])] ;

randomize <class_identifier> [([<variable_list>|null])] with
  constraint block ;
end randomize ;
```

4.3.2. Tradeoff: Randomize Return Success/Failure

Returning a success/failure value complicates the use model. Each call to randomize must handle the return value. Instead of having a return value, the predefined method, named error_handler, shall be called when an error in randomization occurs. Error_handler is overloadable and has a default implementation that prints randomization failed and the name of the class variable for which it failed.

4.3.3. Tradeoff: Enabling Randomization and Constraints

The implication of rand_mode and constraint_mode is that for each random variable and for each constraint there is an implied variable that controls whether it is enabled or disabled. VHDL uses attributes to address situations like this.

```
Attribute Rand of class_variable is true ;
Class_variable'rand is true ;
```

4.3.4. Tradeoff: Srandom is ill Named.

It is not obvious that srandom is the seed method. Based on names it is difficult to distinguish srandom from urandom. It would be better if the seed initialization method for randomize included the word seed., for example, seed_random or init_seed.

4.3.5. Tradeoff: Randomize Is Not Overloadable

If randomize were overloadable, methods pre_randomize and post_randomize would not be needed. To do this, some how the current variable list and constraint block would need to be replaced by a subprogram generic or parameter. The object could be either a constraint block or an anonymous in-line extension of the class. If the object is a constraint block, the statements that enable/disable variable randomization would need to be expressible in the constraint block.

Expressed in an overloadable form:

```
<class_identifier>.randomize (anonymous class extension) ;
<class_identifier>.randomize (constraint_block) ;
```

Neither of these seem quite right – it might be better to stick with the pseudo-method model.

4.3.6. Tradeoff: Randomize Does Not Return a Randomized Value

Randomize does not return a randomized value. Instead an application accesses the class variables that have been randomized to gain access. If we view a class as a structure + methods, then this seems reasonable as then the entire class itself can be passed as a transaction value.

```
Shared Variable CpuOpTrans : CpuOpGen ;
. . .
CpuOpTrans.randomize ;
Scoreboard.put(CpuOpTrans) ;
TransactionIF.put(CpuOpTrans) ;
```

4.3.7. Conclusions

While there are merits to doing an exact syntax sugaring of SystemVerilog, it is also important to keep consistent with the nature of VHDL.

If we don't do a direct VHDL'ization of SystemVerilog, I would at least address the following issues:

- Make Randomize a pseudo-procedure and provide an overloadable error_handler method.
- Enable randomization and constraints with attributes, but only if we also allow attributes to be specified as a statement as well as a declaration.
- Put "seed" in the name of the method that specifies the seed value (seed_random, init_seed, …)

4.4. Format of Class Based Constraints

4.4.1. Baseline

Constraints are specified in both the class and the randomize pseudo-call. The following shows constraints specified in a class:

```
Type CpuOpType is (CpuRead, CpuWrite, CpuIntAck) ;
Type CpuOpGen is class
  Rand Variable CpuOp : CpuOpType ;
  Rand Variable Addr : std_logic_vector(21 downto 0);
```

```
    Rand Variable Data : std_logic_vector(15 downto 0) ;

    Constraint ChipRegs is (
       Addr(21 downto 16) = "000000" ;
       Data >= 0 ;
       Data <= 255
    ) ;
  End class CpuOpGen ;
```

The following shows a constraint specified in the pseudo-call to randomize:

```
CpuOpTrans.randomize with (Addr(1 downto 0) = "00"; Data <= 127 ) ;
```

Questions:

- Should '{}' be used instead of '()'. I was thinking we should save '{}' for a PSL SERE which may be interesting to consider adding as an individual constraint.
- ';' is the natural terminator for a constraint, however, it does not seem right within the call to randomize.
- Should ';' be a separator or a terminator or either?
- Use ',' rather than ';'? Will not work with selected constraints.

4.4.2. Why not use "end"

I did not consider the usage of "end" as it seems in appropriate to use in the pseudo-call to randomize:

```
CpuOpTrans.randomize with Addr(1 downto 0) = "00"; Data <= 127 end randomize ;
```

Perhaps if when adding constraints they were considered a block of constraints in a similar manner to an if then statement having a block of code, then we end up with:

```
CpuOpTrans.randomize with
  Addr(1 downto 0) = "00";
  Data <= 127 ;
end randomize ;
```

Perhaps evolving this to something like:

```
randomize CpuOpTrans with
  Addr(1 downto 0) = "00";
  Data <= 127 ;
end randomize ;
```

4.5. Specifying Constraints

Beyond simple boolean expressions, SystemVerilog gives constraints an entire language of their own. The section that follows gives a terse translation of how these features could look in VHDL. As we consider additional capability, it would be good to consider how we can leverage the syntax it introduces in other places in the language.

4.5.1. Issues

Need some way to use integers as constraints for types unsigned and signed. Is there a way to leverage the visible "=" operator?

4.5.2. Set Membership

4.5.2.1. Baseline – PSL uses in and {}

```
Constraint C is ( x in {0 to 15, 31 to 63, 127} ) ;
Constraint D is ( x + y in {0 to 15, 31 to 63, 127} ) ;
```

-- Separate section --

Using this same notation, the for loop could be expanded to:

```
For I in {0 to 15, 31 to 63, 127} loop
```

It would be nice to extend this syntax to array aggregates. The parentheses around the index specification below would be required:

```
Y(0 to 127) <= ( {0 to 15, 31 to 63, 127} => '0', others => '1') ;
```

4.5.2.2. Tradeoff: Use '|' instead of ','

```
Constraint C is ( x in (0 to 15 | 31 to 63 | 127) ) ;
Constraint D is ( x + y in (0 to 15 | 31 to 63 | 127) ) ;
```

Using this same notation, the for loop could be expanded to:

```
For I in 0 to 15 | 31 to 63 | 127 loop  -- perhaps requires parentheses?
```

'|' used as separator simplifies the notation in an array aggregates:

```
Y(0 to 127) <= (0 to 15 | 31 to 63 | 127 => '0', others => '1') ;
```

4.5.2.3. – No PSL does "in". Tradeoff: Use "inside" like SV instead of "in"

SystemVerilog uses the keyword "inside". Since VHDL already has "in" and this is somewhat consistent with its usage, it seems appropriate to use "in" instead of "inside".

4.5.3. Distributions -- replace inner)(with }{

4.5.3.1. Baseline

By default, each item in set membership has equal weight. Distributions add a relative weight to each set element. The following gives X a value that is either 0, 2, or 4 with a relative weight of 1, 2, and 2 respectively.

```
Constraint C is ( x in (1 => 0, 2 => 2, 2 => 4) ) ;
```

Distributions can be applied to a range. When specified with "=>" each element of the range has that weight. When specified with "/=>", each element has an equal piece of the specified weight divided among them. Hence, the following gives X a value of 0, 1, 2, 6, 7, or 8 with a relative weight of 1, 1, 1, 2, 2, and 2 respectively.

```
Constraint D is ( x in (1 => 0 to 2, 6 /=> 6 to 8 ) ) ;
Constraint E is (x in (1 => 0, 1 => 1, 1 => 2, 2 => 6, 2 => 7, 2 => 8) ) ;
```

Expressions are also permitted to express the distribution weights:

```
Constraint F is ( x in (I0 => 0 to 2, I1 /=> 6 to 8 ) ) ;
```

The baseline uses a different syntax from SystemVerilog so as to leverage the similarity to aggregate notation. Hence, the weight is in the position of the formal and the value or range is in the position of the actual. This gives it some consistency with the randcase and the select statements.

4.5.3.2. Tradeoff: Use ":=" like SystemVerilog

The following gives X a value that is either 0, 2, or 4 with a relative weight of 1, 2, and 2 respectively.

```
Constraint C is ( x in (0 := 1, 2 := 2, 4 := 2) ) ;
```

Distributions can be applied to a range. When specified with ":=" each element of the range has that weight. When specified with ":/", each element has an equal piece of the specified weight divided among them. Hence, the following gives X a value of 0, 1, 2, 6, 7, or 8 with a relative weight of 1, 1, 1, 2, 2, and 2 respectively.

```
Constraint D is ( x in (0 to 2 := 1, 6 to 8 :/ 6) ) ;
```

4.5.3.3. – PSL uses ',' … Tradeoff: Use '|' instead of ','

```
Constraint C is ( x in (1 => 0 | 2 => 2 | 2 => 4) ) ;
Constraint D is ( x in (1 => 0 to 2  | 6 /=> 6 to 8 ) ) ;
```

4.5.3.4. Tradeoff: Use separate keyword dist like SystemVerilog

SystemVerilog uses a separate keyword named dist. When looking at the syntax, I noted that the primary difference between the inside operation (in) and distribution operation (dist) is the specification of weights. Unless there is a compelling reason for an additional keyword, I am recommending only using "in" like in the baseline.

4.5.4. Conditional Constraints – return to this

Syntax is borrowed from conditional assignment and supports both when and when – else forms as shown below:

```
Constraint ChipRegs is (
   Data(15 downto 5) = 0 when Addr = 0 ;
   (Data(15 downto 7) = 0; Data(0) = '0') when Addr = 1 else
   Data(16 downto 8) = 0
) ;
```

SystemVerilog goes further and supports a PSL like implication operator (which reverses the order of the expression). Note the compelling reason to support this is consistency with PSL – something that would become more interesting if we integrate more of the PSL syntax here. Note there is not "else" to this form.

```
Constraint ChipRegs is (
 Addr = 0 -> Data(7 downto 5) = 0 ;
) ;
```

4.5.5. Selected Constraints -- return to this

Syntax is borrowed from selected assignment (SV does not do this).

```
Constraint ChipRegs is (
     With Addr select
       Data(15 downto 5) = 0 when 0,
       (Data(15 downto 7) = 0; Data(0) = '0') when 1,
       Data(16 downto 8) = 0  when others ;
  ) ;
```

When multiple constraints are specified in a single condition, they must be enclosed in parentheses.

4.5.6. Loop Constraints -- PSL has something like this

Used in SV to iterate across a dynamically sized array:

```
Constraint D is ( foreach (A[I]) A[I] > 2 * I ) ;
```

While this can be translated to VHDL, just how much flexability can the solver handle? Does the loop have to be able to be unrolled like for synthesis? If we need general sequential code, then we should consider how to include a sequential block of code here. Functions can be called here, however, SV puts restraints on their order of solving.

If we do implement this, we could consider a direct translation that leverages VHDL's attributes such as:

```
Constraint D is ( foreach I in A'range (A[I] > 2 * I ) ) ;
Constraint E is ( for I in A'range (A[I] > 2 * I ) ) ;  -- keyword for ok?
```

Alternately we should also look at PSL and other languages (APL?).

PSL does something like this with a conjunction (and) and disjunction (or) capability – for constraints, only conjunction makes sense.

4.5.7. Functions In Constraints

Same as SV.

4.5.8. Constraint Guards

Same as SV.

4.5.9. Solve First Constraints

As much as possible, the randomization solver tries to give a uniform value distribution over all values possible in the constrained set. To accomplish this, constraints are solved simultaneously. With the following constraints:

```
Type Example1 is class
  Rand Variable B : Natural ;
  Rand Variable C : Natural ;
  Constraint X is ( B in (1 to 4) ; C < B ) ;
End class Example1 ;
```

For example1 class, all solutions for (B, C) are equally likely: (1,0), (2,1), (2,0), (3,2), (3,1), (3,0), (4,3), (4,2), (4,1), and (4,0), hence, each happens 10% of the time. The "Solve First" constraint gives the user some control of this.

```
Type Example2 is class
  Rand Variable B : Natural ;
  Rand Variable C : Natural ;
  Constraint X is ( solve B before C ; B in (1 to 4) ; C < B ) ;
End class Example2 ;
```

For example2 class, each value of B is equally likely. B = 1, 2, 3, 4 each happen 25 % of the time. Next C is solved. So the solution probabilities for (B, C) are: (1,0) = 25 %, (2,1) and (2,0) = 12.5%, … (4,0) = 6.25%.

4.5.10. Constraint Solution Ordering

Same as SV.

5. Procedural Randomization

5.1. RandCase

-- get better example

```
I0 := 1;  I1 := 1;  I2 := 1;
for i in 1 to 3 loop
  RandCase is
    with I0 =>
      CpuWrite(CpuRec, DMA_WORD_COUNT, DmaWcIn);
      I0 := 0 ;  -- modify weight
    with A-B =>
      CpuWrite(CpuRec, DMA_ADDR_HI, DmaAddrHiIn);
      I1 := 0 ;  -- modify weight
    with (I2) =>
      CpuWrite(CpuRec, DMA_ADDR_LO, DmaAddrLoIn);
      I2 := 0 ;  -- modify weight
  end case ;
end loop ;
CpuWrite(CpuRec, DMA_CTRL, START_DMA or DmaCycle);
```

5.2. Select / RandSelect – {} -- multiset – class definition

RandCase is for sequential statements. Select is for expressions. Syntax is similar to an "in" constraint.

```
Y := select(0 to 15, 31 to 63, 127) ;
X := select(1 => 0 to 2, 6 /=> 6 to 8) ;  -- equivalent to Z
Z := select(1 => 0, 1 => 1, 1 => 2, 2 => 6, 2 => 7, 2 => 8) ;
W := select(I0 => 0 to 2, I1 /=> 6 to 8 ) ;
```

Is it possible for select to return a correct value for type unsigned, or is a conversion needed:

```
X_unsigned := to_unsigned(select(1 => 0 to 2, 6 /=> 6 to 8), 8) ;
```

5.3. Sequence / RandSequence

Is there a way to make this more similar/more compatible to PSL?

Sequences to constraints on method sequencing – concurrent path expressions

Specifying protocol of allowed method invocation on an interface

How is recursion done?

Where did sequence come from?

5.3.1. Baseline

Sequence specifies BNF productions to procedurally sequence of code.

```
sequence (main) is
  Main     =>  first ; second ; last ;
  First    =>  select (add, dec) ;
  Second   =>  select (pop, push) ;
  Done     =>  write(output, "done" & LF) ;
  Add      =>  write(output, "add" & LF) ;
  Dec      =>  write(output, "dec" & LF) ;
  Pop      =>  write(output, "pop" & LF) ;
  Push     =>  write(output, "push" & LF) ;
end sequence ;
```

Is there a problem since this re-uses the keyword sequence?

5.3.1.1. Randomizing Alternatives

Extend select to allow it to choose between production alternatives:

```
  First    =>  select (add,  dec) ;
  Second   =>  select (2 => pop, 5 => push) ;
```

5.3.1.2. Sequential Code

All code that follows a non-terminal is sequential code. So there is no need for if-else, case, or loop productions.

5.3.1.3. Interleaving Productions: RandJoin

```
sequence (TOP) is
  TOP      =>  RandJoin (S1, S2) ;
  S1       =>  A ; B ;
  S2       =>  C ; D ;
  A        =>  write(output, "A" & LF) ;
  B        =>  write(output, "B" & LF) ;
  C        =>  write(output, "C" & LF) ;
  D        =>  write(output, "D`" & LF) ;
end sequence ;
```

5.3.1.4. Aborting Productions: Exit and Return

The exit statement terminates the sequence generation. The return statement terminates a production and sequence generation continues with the next production.

5.3.1.5. Value passing between productions

Each production can accept parameters in the same manner of a subprogram (both function and procedure).

More work is needed here to specify this. For now, see the SV specification.

5.3.2. Repeat --?? Similar to inFact – more similar to PSL.

5.3.3. Tradeoffs

A sequence can be emulated using multiple subprogram calls and the use of randcase. The features unique to sequence are RandJoin and exit (effectively returning from multiple layers of subprogram calls).

-- exit could be added by throwing an exception

-- RandJoin ??

Concurrency control / inheritance abnormality be controlled with constraints of this nature

Alternately, it seems that PSL syntax should be leveraged here.

6. Random Stability

Random stability refers to a program, such as a testbench, producing the same randomization results for executions on the same software (simulator). As much as possible, a program should remain stable when small changes are made.

A separate random number generator is created for each thread (a process or concurrent statement) and each object. This makes the sequence of random values generated by a thread or object independent from sequence random values generated by other threads or objects. This applies to randomization using:

- Procedural Randomization statements (thread)
- Basic Randomization (thread)
- Class Randomization (object)

Additional control over seed can be accomplished through manual seed generation.

--

-- Seeding of threads must be independent of the seeding of objects, otherwise, the addition of an object to the declarative region of a design will cause a change in the randomization to the threads.

-- elaboration based seeding

Seeding stable/independent of added designs, added processes, added objects

Each instance has an initialization generator.

Does SV hierarchically elaborate? -> this may cause each component to have a deterministic value

Can we manually seed an entity?

6.1. Random stability properties

Same as SV

6.2. Thread Stability

Same as SV

6.3. Object Stability

Same as SV

7. Manual Seeding

The seed for threads and objects can be seeded with the method, seed_random. Manual seeding will override any of the default seeding done at elaboration of a design.

Is there a place in an architecture/thread at which you can call seed_random that will effect the generated seeds within the object?

Implies information must be available to the elaborator.

Objects within objects? What is the name/thing that can be referred to?

In SV objects are created dynamically with new.

Attribute an object within declaration.

Short hand notation for attribute on a declaration.

How to get at things in general that is an implicit part of a construct?

7.1. Random stability properties

Random stability encompasses the following properties:
— *Initialization RNG*. Each module instance, interface instance, program instance, and package has an initialization RNG. Each initialization RNG is seeded with the default seed. The default seed is an implementation-dependent value. An initialization RNG shall be used in the creation of static threads and static initializers (see the following bullets).
— *Thread stability.* Each thread has an independent RNG for all randomization system calls invoked from that thread. When a new dynamic thread is created, its RNG is seeded with the next random value from its parent thread. This property is called *hierarchical seeding*. When a static thread is created, its RNG is seeded with the next value from the initialization RNG of the module instance, interface instance, program instance, or package containing the thread declaration.
Program and thread stability is guaranteed as long as thread creation and random number generation are done in the same order as before. When adding new threads to an existing test, they can be added at the end of a code block in order to maintain random number stability of previously created work.
— *Object stability.* Each class instance (object) has an independent RNG for all randomization methods in the class. When an object is created using **`new`**, its RNG is seeded with the next random value from the thread that creates the object. When a class object is created by a static declaration initializer, there is no active thread; thus, the RNG of the created object is seeded with the next random value of the initialization RNG of the module instance, interface instance, program instance, or package in which the declaration occurred.
Object stability is guaranteed as long as object and thread creation and random number generation are done in the same order as before. In order to maintain random number stability, new objects, threads, and random numbers can be created after existing objects are created.
— *Manual seeding.* All noninitialization RNGs can be manually seeded. Combined with hierarchical seeding, this facility allows users to define the operation of a subsystem (hierarchy subtree) completely with a single seed at the root thread of the subsystem.

7.2. Thread stability

Random values returned from the `$urandom` system call are independent of thread execution order. For example:

```
integer x, y, z;
fork //set a seed at the start of a thread
begin process::self.srandom(100); x = $urandom; end
//set a seed during a thread
begin y = $urandom; process::self.srandom(200); end
// draw 2 values from the thread RNG
begin z = $urandom + $urandom ; end
join
```

The above program fragment illustrates several properties:
— *Thread locality.* The values returned for `x`, `y`, and `z` are independent of the order of thread execution. This is an important property because it allows development of subsystems that are independent, controllable, and predictable.
— *Hierarchical seeding.* When a thread is created, its random state is initialized using the next random value from the parent thread as a seed. The three forked threads are all seeded from the parent thread.
Each thread is seeded with a unique value, determined solely by its parent. The root of a thread execution subtree determines the random seeding of its children. This allows entire subtrees to be moved and preserves their behavior by manually seeding their root thread.

7.3. Object stability

The `randomize()` method built into every class exhibits object stability. This is the property that calls to `randomize()` in one instance are independent of calls to `randomize()` in other instances and are independent of calls to other randomize functions.
For example:

```
class Foo;
rand integer x;
endclass
class Bar;
rand integer y;
endclass
initial begin
Foo foo = new();
Bar bar = new();
integer z;
void'(foo.randomize());
// z = $random;
void'(bar.randomize());
end
```

— The values returned for `foo.x` and `bar.y` are independent of each other.
— The calls to `randomize()` are independent of the `$random` system call. If one uncomments the line `z = $random` above, there is no change in the values assigned to `foo.x` and `bar.y`.
— Each instance has a unique source of random values that can be seeded independently. That random seed is taken from the parent thread when the instance is created.
— Objects can be seeded at any time using the `srandom()` method.

```
class Foo;
function new (integer seed);
//set a new seed for this instance
this.srandom(seed);
endfunction
endclass
```

Once an object is created, there is no guarantee that the creating thread can change the object's random state before another thread accesses the object. Therefore, it is best that objects self-seed within their **`new`** method rather than externally.
An object's seed can be set from any thread. However, a thread's seed can only be set from within the thread itself.

7.4. Manually seeding randomize

Each object maintains its own internal RNG, which is used exclusively by its `randomize()` method. This allows objects to be randomized independent of each other and of calls to other system randomization functions. When an object is created, its RNG is seeded using the next value from the RNG of the thread that creates the object. This process is called *hierarchical object seeding*.
Sometimes it is desirable to manually seed an object's RNG using the `srandom()` method. This can be done either in a class method or external to the class definition:
An example of seeding the RNG internally, as a class method, is as follows:

```
class Packet;
rand bit[15:0] header;
...
function new (int seed);
this.srandom(seed);
...
endfunction
endclass
```

An example of seeding the RNG externally is as follows:

```
Packet p = new(200); // Create p with seed 200.
p.srandom(300); // Re-seed p with seed 300.
```

Calling `srandom()` in an object's **`new()`** function assures the object's RNG is set with the new seed before any class member values are randomized.

8. End Notes

8.1. Important Considerations

Stability

How do we turn things on and off.

8.2. Stability

DVCon Meeting notes (Ray Ryan)

Solvers only do 2 state values.

Seeding algorithm: Root Seed created during elab. Note that order of elab not defined by language

Need randomization stability such that turning randomization on/off for any given object/process does not effect any other obj/proc as long as no obj/proc added lexically before an already defined obj/proc.

Every obj/proc has a seed whether it uses it or not. Seed for each obj/proc is settable.

Stability is vendor dependent and hence there is no stability between vendors

Irony: 2 different instances of same model have same basic RVs

8.3. TF & JB from Generating Stimulus

Tom F & Janick B from "Generating Stimulus" in The Functional Verification of Electronic Systems edited by Brian Bailiey

Good example on generating constraints using set operations:

```
Class Xaction ;
  Rand bit [1:0] sel ;
  Rand bit [4:0] addr ;
  Rand bit [7:0] data ;
  Rand enum kind {READ, WRITE} ;
  Constraint rd_wr {
    Kind dist {READ := 1 ; WRITE := 3; }
    Sel inside {[0:3]} ;
    If (sel == 0 && kind = READ)
      Addr inside {[0:15], 27, 31} ;
    Else
      Addr inside {[0:31]} ;
    If (kind == WRITE)
      Data inside {8'h00, 8'h55, 8'hAA, 8'hFF} ;
  } ;
endclass

class XactionNarrow extends Xaction;
  constraint narrow {
    kind inside {READ, WRITE};
    addr == 0 ;
    if (kind == WRITE) data == 8'hFF;
  } ;
endclass
```