Synopsys Users Group
SILICON VALLEY 2013

Random Stability In SystemVerilog

Doug Smith
Doulos

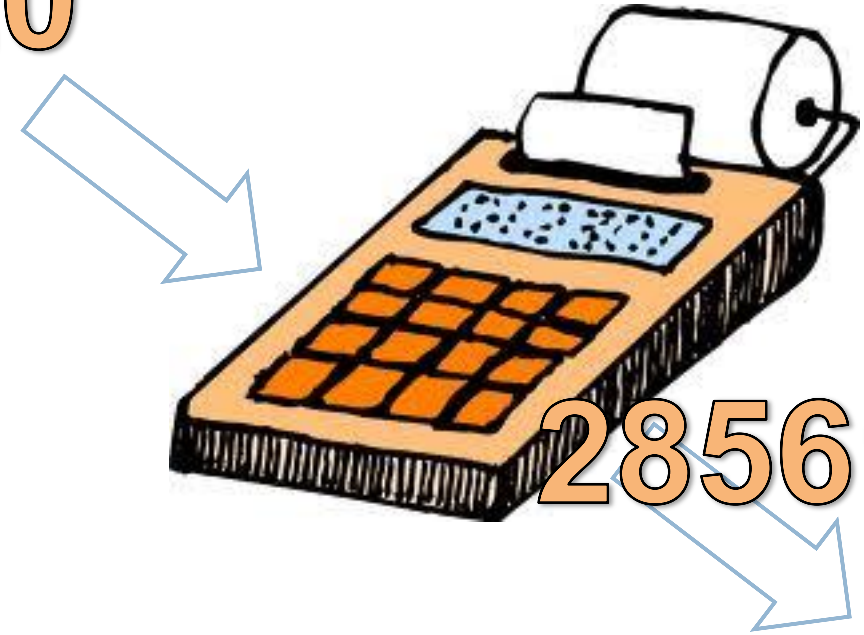
Random stability

- ***Random Number Generators***
- ***Random Seeds***
- ***SystemVerilog Hierarchical Seeding***
- ***Random Stability in VMM and UVM***
- ***Summary***

Random number generators

Seed

20



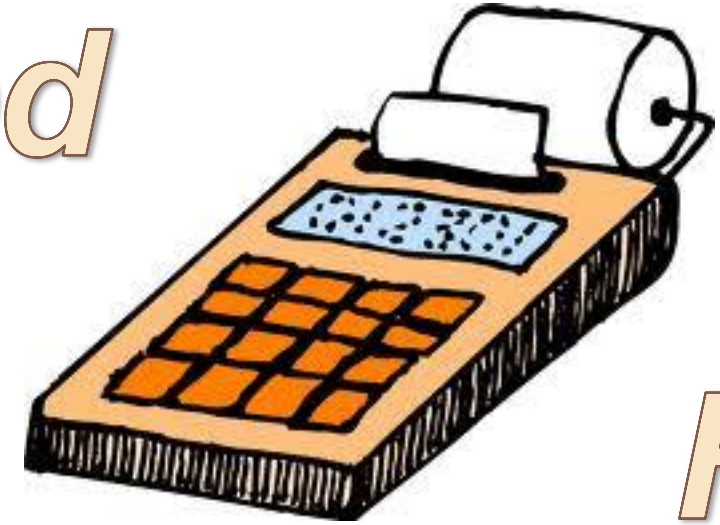
2856130560

Random Number

Random number quality

Number quality depends on:

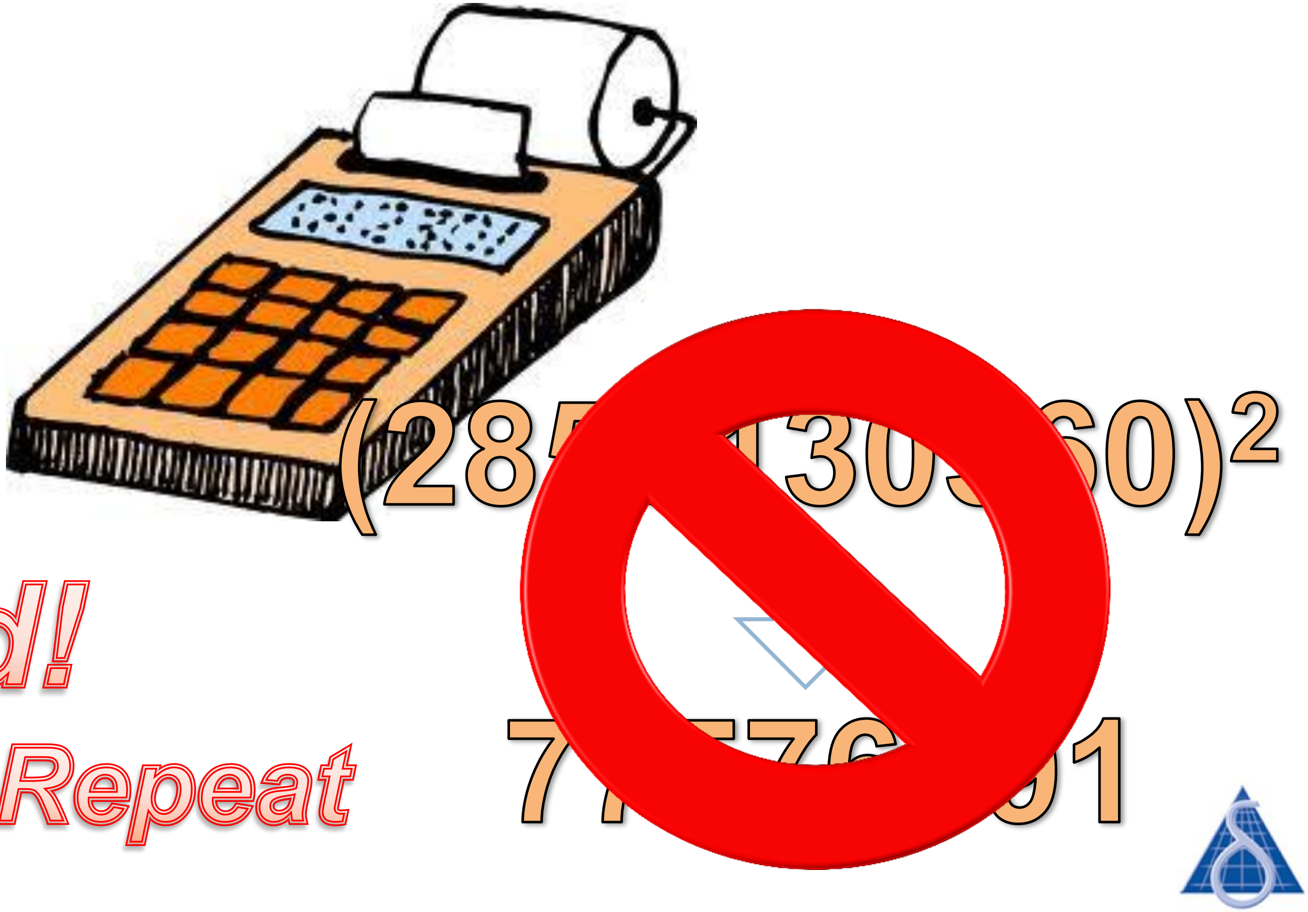
Seed



RNG
Algorithm

Random number generators (RNG)

Middle-square Method



Bad!

Numbers Repeat

Random number generators (RNG)

Linear Congruential Generator (LCG)

X_0 = seed

a = prime number

c = increment

m = range

$$X_{n+1} = (a\,X_n + c) \bmod m$$

Better, but still repeats!

Random number generators (RNG)

LFSR RNG

Mersenne Twister

Others

Best (currently)

SystemVerilog RNG?

Verilog:

- $random
- $dist_uniform
- $dist_*

→ ~ LCG

SystemVerilog:

- $urandom
- $urandom_range
- randomize

? ←

Random stability

- *Random Number Generators*
- ***Random Seeds***
- *SystemVerilog Hierarchical Seeding*
- *Random Stability in VMM and UVM*
- *Summary*

Seeds to avoid

- Zero – not good for most LCG
- Limited range values
  - Small number of bits – 1 byte == 256 values!
  - Process id (pid) – usually only 32767 values
  - Time and date – limited variations
  - Shell $RANDOM – only 15-bit result
  - $random to seed $random – strong correlation between values

What makes a good seed?

- Generated from a random source
- Not from the same RNG
- Not limited to a small subset of values
- Source that relies on physical randomness
- As many bits as possible

Recommendation

- In Unix, use /dev/urandom
  - Uses entropy pools inside the kernel
  - Injects random kernel jitter measurements

```
% head -4 /dev/urandom | od -N 4 -D -A n | awk '{print $1}’
```

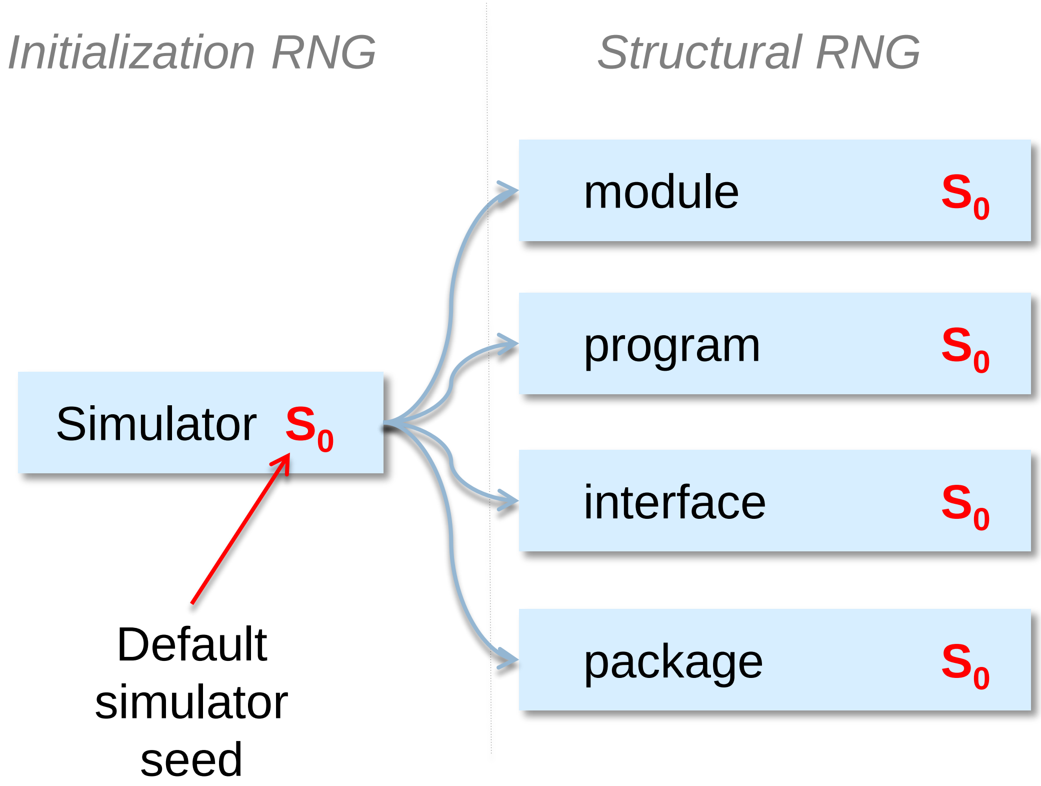
Random stability

- *Random Number Generators*
- *Random Seeds*
- ***SystemVerilog Hierarchical Seeding***
- *Random Stability in VMM and UVM*
- *Summary*

SV Hierarchical Seeding

- Thread and object locality
  - Each thread and object has its own RNG
  - RNG calls in different threads/objects are independent
- Hierarchically seeded
  - Initialization RNG – default seed
  - Testbench is seeded from initial RNG

Structural elements



Initialization RNG

Structural RNG

Simulator S_0

Default simulator seed

module S_0

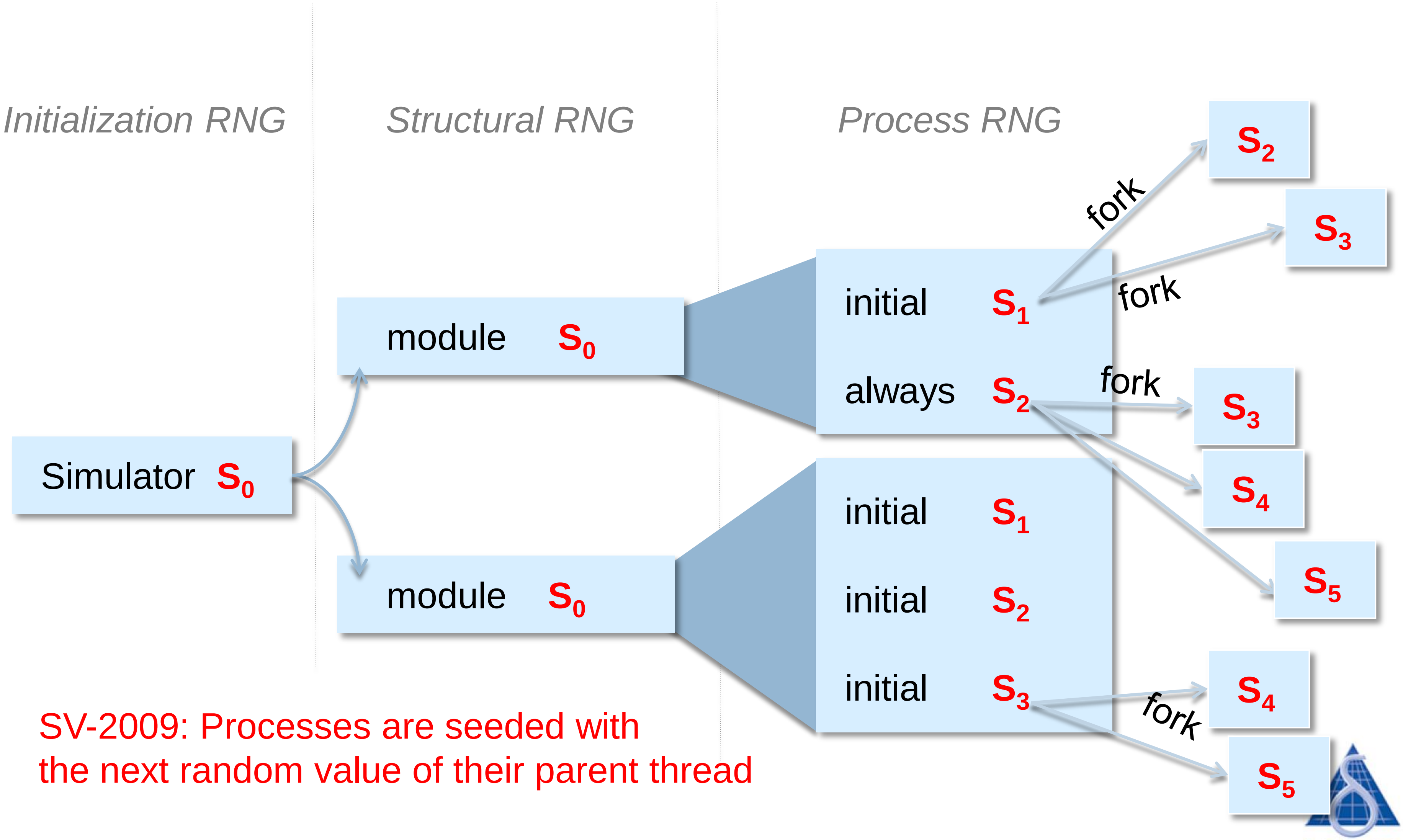
program S_0

interface S_0

package S_0

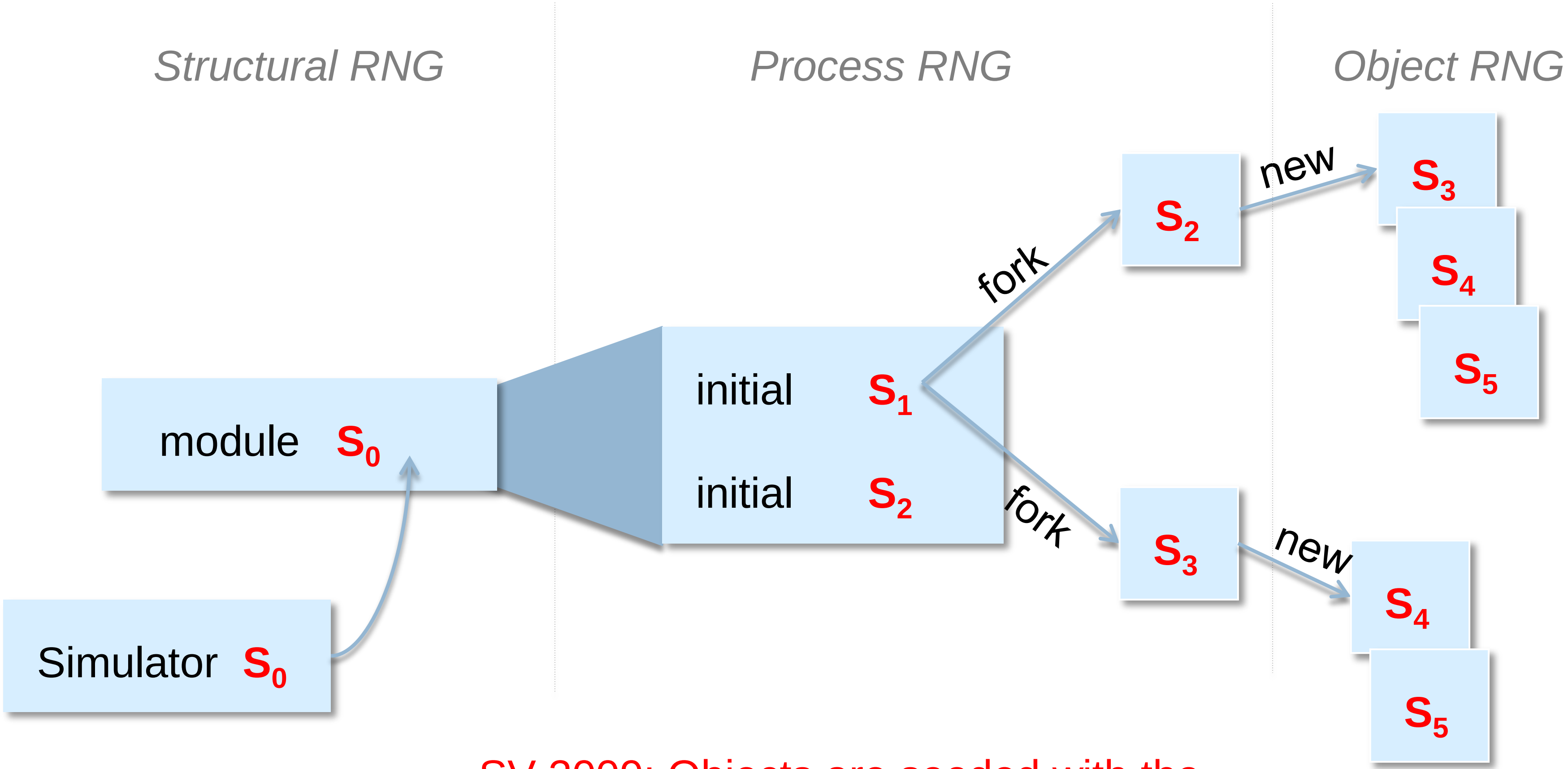
SV-2009:
Structural elements seeded with the initial seed of the simulator

Processes



SV-2009: Processes are seeded with the next random value of their parent thread

Objects



Initialization RNG

SV-2009: Objects are seeded with the next random value of their parent thread

Simulator differences

- SV-2009 standard is unclear on processes:
  - *“An initialization RNG shall be used in the creation of* ***static processes*** *and static initializers…”*
  - *“Each initialization RNG is seeded with the* ***default seed****.” [S_0]*
  - *Static processes are seeded with the “****next value*** *[S_1] from the initialization RNG of the module instance, interface instance, program instance, or package containing the thread declaration.”*
- Is it the default seed (S_0) or next value (S_1)?
- Does *next value == next random number*?

Simulator differences (cont’d)

```
module test;
    process m = process::self();

    initial begin
        process p;
        p = process::self();

        $display(“Module randstate = ”, m.get_randstate());
        $display(“Module randstate = ”,  p.get_randstate());
    end
endmodule
```

- Statically declared processes and objects:
  - VCS: (object) same seed instead of the next random value
  - IUS: (process) different random values
  - Questa: (process) same initial value instead of the next random value

Random instability

- So where does random instability come from?
- Test stimulus:
  - The order of random calls changes
- Testbench:
  - A new process is inserted before an existing one
  - The order of the creation of forks changes
  - The order of the creation of objects changes

This we can control

Locking down the seed

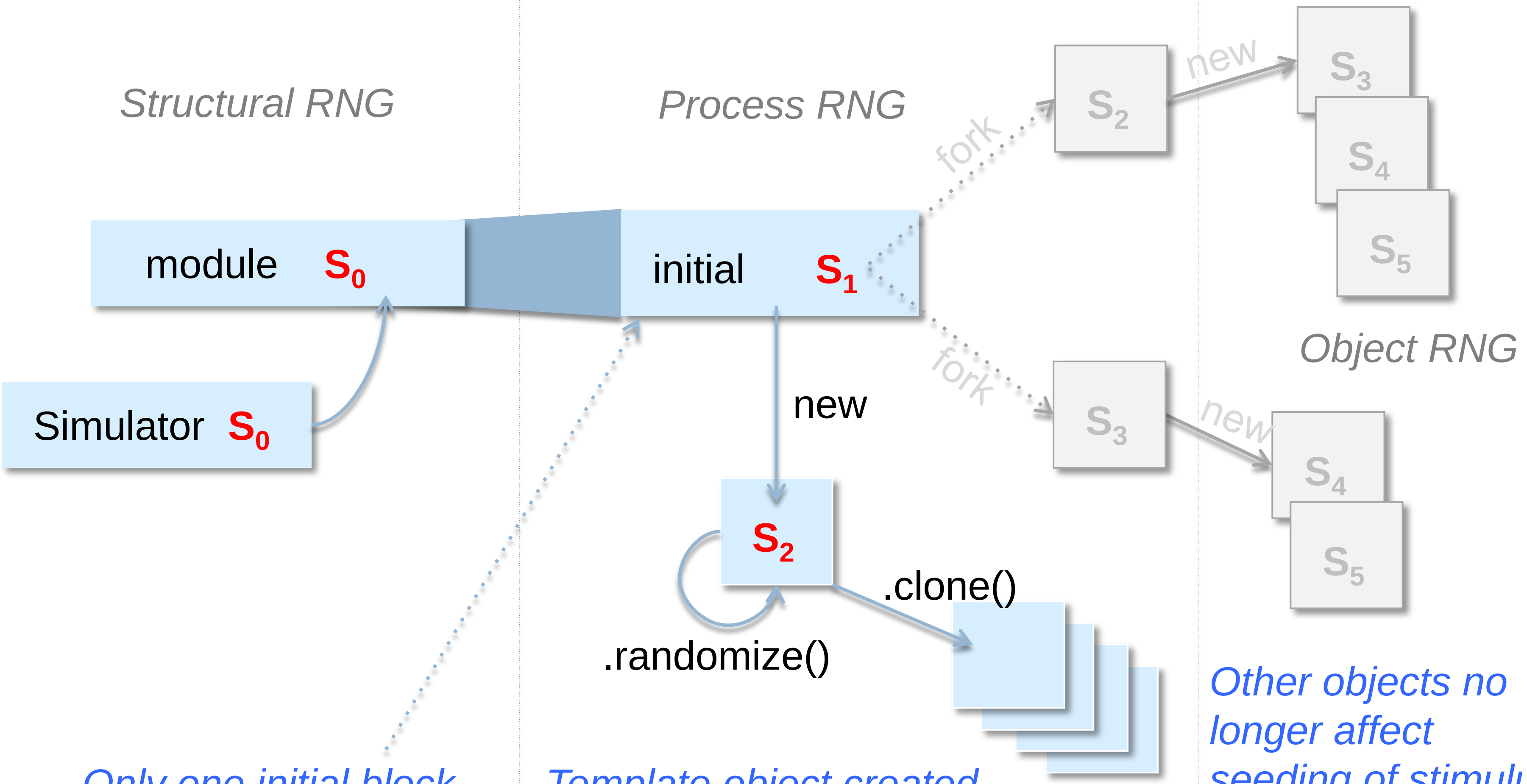
- Ways to lock down the seed

1. ~~Control order of creation~~

Hard to manage

2. Use a template generator and seed it
3. Manually seed each testbench component

Template object



Only one initial block so always the same initial seed

Template object created first thing at time zero so always the same seed

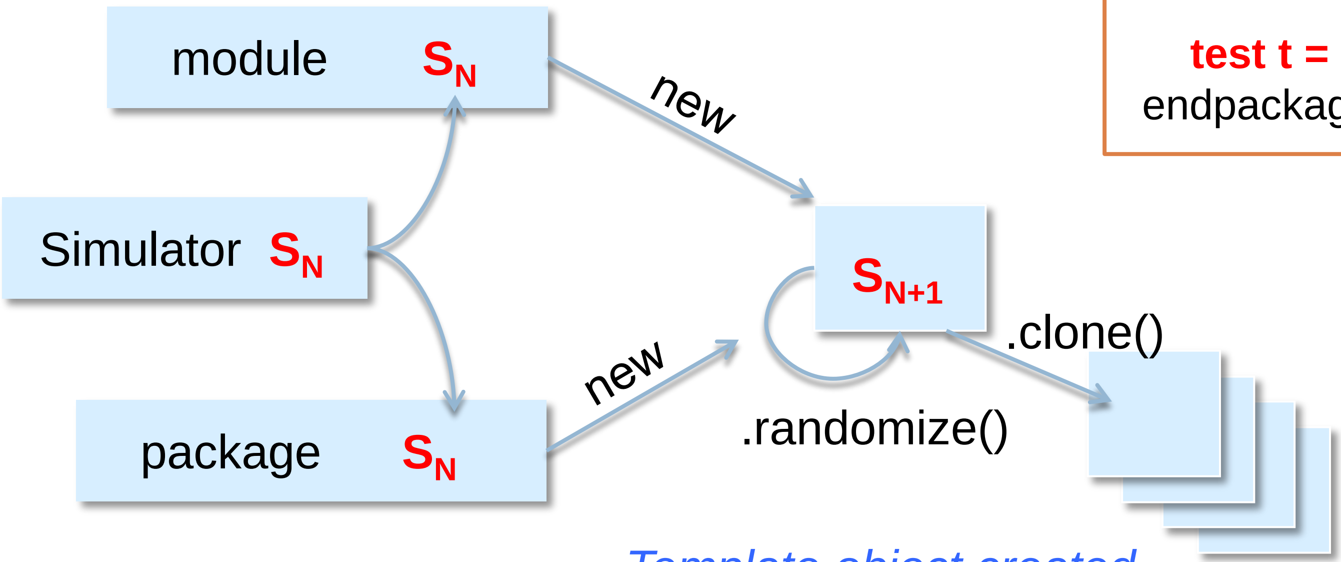
Other objects no longer affect seeding of stimulus

Statically declared initializer

Static declaration initializer (variable initialized with new) in module or package

```
package my_pkg;
  class test;
     …
  endclass

    test t = new();
endpackage
```



Template object created at time zero with next seed

Randomize and copy only the template to preserve the random sequence

Manual seeding *(recommended)*

- srandom()

```
initial begin
   process::self.srandom( 1234 );            // initial block seed = 1234

   fork
       begin
            process::self.srandom( 1 );        // fork process seed = 1
            …
       end
       …
   join
```

```
initial
begin
    my_test t = new;
    t.srandom( 6879 );                        // object t seed = 6879
end
```

Manual seeding *(not recommended)*

- set_randstate()

```
string state;

initial begin
   process p;
   p = process::self();
   state = p.get_randstate();                    // Grab the current RNG state
   …

   p.set_randstate( state );                     // Restore RNG state

end
```

- Randstate string examples from VCS:

```
00Z1ZZZ1Z0ZZ11XZX11Z1ZX1XZ1Z0X01XZZZZZXZXZXZZZZXXXXZZXZZXXZZXXZZ
```

```
0000000000000000000000000000000000110101010001011001101111110100
```

Random stability

- *Random Number Generators*
- *Random Seeds*
- *SystemVerilog Hierarchical Seeding*
- ***Random Stability in VMM** and UVM*
- *Summary*

VMM strengths

- Uses get_randstate() to save state
- Uses set_randstate() to restore state when simulation reset
- Favors the use of a data factory (i.e., template generator)

Random stability in VMM

- Does not manually seed or control seeding
- get_/set_randstate() works *within* a simulation, not *between* simulations
  - Used in xactors and env, but not vmm_data!

VMM Guideline: Manually seed all objects

Suggested VMM enhancements

- Manually seed with srandom!!
  - Use a good seeding algorithm
- Provide a +VMM_SEED command line argument for portability

Random stability

- *Random Number Generators*
- *Random Seeds*
- *SystemVerilog Hierarchical Seeding*
- ***Random Stability in** VMM and **UVM***
- *Summary*

UVM strengths

- UVM manually seeds all objects ☺

```
int unsigned uvm_global_random_seed = $urandom;
```

- UVM avoids duplicates seeds

```
// Function- uvm_oneway_hash
//
// A one-way hash function that is useful for creating srandom seeds. An
// unsigned int value is generated from the string input. An initial seed can
// be used to seed the hash, if not supplied the uvm_global_random_seed
// value is used. Uses a CRC like functionality to minimize collisions.
//
```

Random stability in UVM

- Environmental changes still break seeding

```
seed_map.seed_table[type_id] = uvm_oneway_hash ({type_id,"::",inst_id},
                                                         uvm_global_random_seed);
```

- No portable way of setting a seed across simulators (+UVM_SEED?)

UVM Guideline: Avoid changing instance names once an environment is created

Suggested UVM enhancements

- Provide portable mechanism to set the seed (+UVM_SEED)

```
int unsigned uvm_global_random_seed = $urandom(UVM_SEED);
```

- Print to log/console the UVM_SEED
- Separate seeding from instance name *(is this possible?)*
- Provide additional DPI 64-bit RNG functions

```
int myRandomInteger;
int randomData = open("/dev/urandom", O_RDONLY);
read(randomData, &myRandomInteger, sizeof(myRandomInt));
close(randomData);
return (myRandomInteger);
```

Random stability

- *Random Number Generators*
- *Random Seeds*
- *SystemVerilog Hierarchical Seeding*
- *Random Stability in VMM and UVM*
- ***Summary***

Summary (1)

- Use high quality seeds (/dev/urandom)
- Ensure random stability:
    - Control order of creation
    - Use a stimulus template generator
    - Manually seed
- Ask vendors to standardized on SV hierarchical seeding

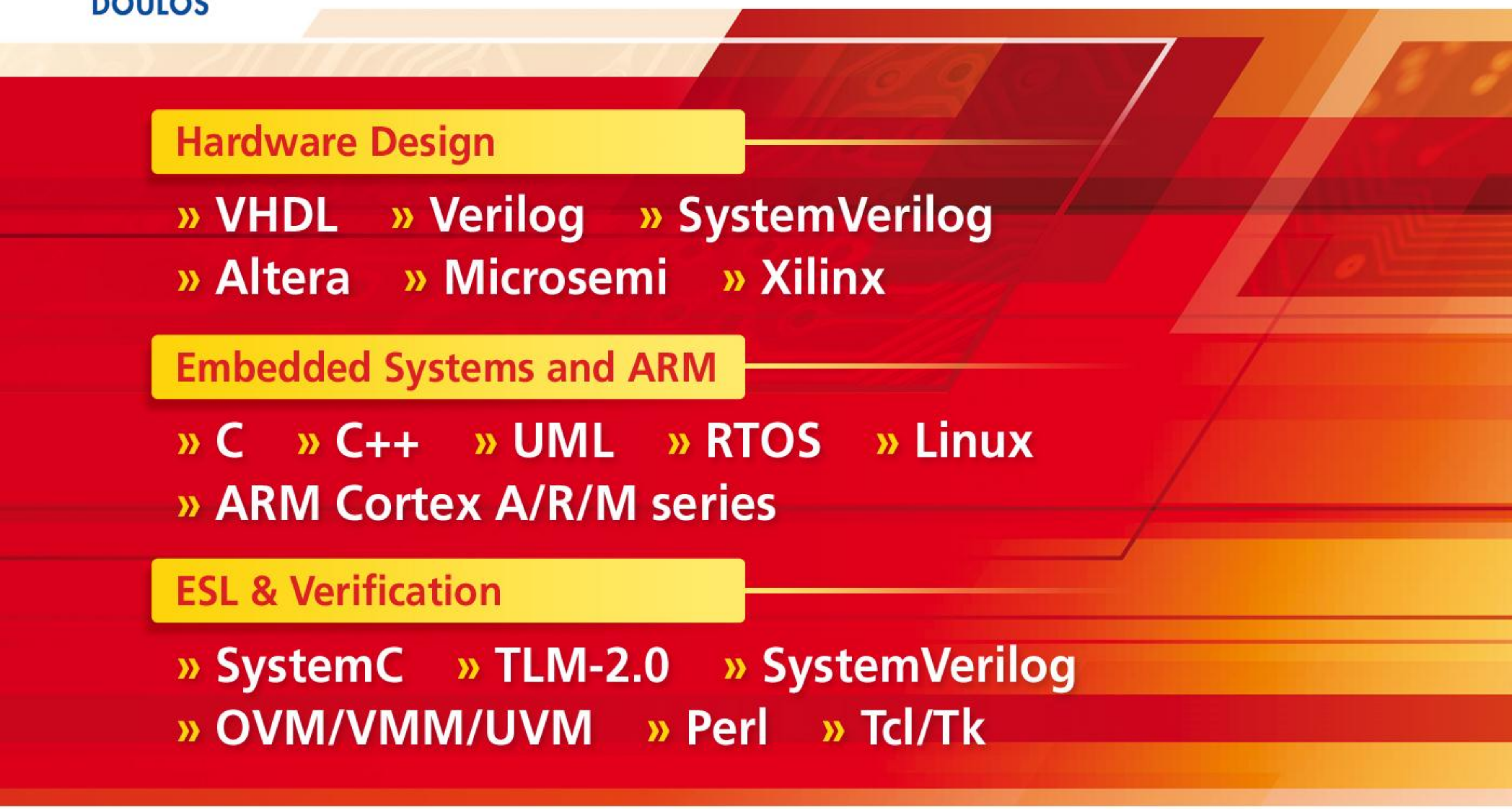
Summary (2)

- Manually seed your VMM environments
- Use UVM for random stable environments
- Suggested enhancements for UVM:
  - +UVM_SEED=
  - Use /dev/urandom if no seed provided
  - Possibly provide 64-bit DPI RNG functions

Questions?

DOULOS

Delivering Know-How www.doulos.com



Hardware Design

» VHDL » Verilog » SystemVerilog
» Altera » Microsemi » Xilinx

Embedded Systems and ARM

» C » C++ » UML » RTOS » Linux
» ARM Cortex A/R/M series

ESL & Verification

» SystemC » TLM-2.0 » SystemVerilog
» OVM/VMM/UVM » Perl » Tcl/Tk

All trademarks acknowledged