```
class Packet;
  bit [3:0] command;
  bit [39:0] address;
  bit [4:0] master_id;
  integer time_requested;
  integer time_issued;
  integer status;

  function new();
    command = 4'hA;
    address = 40'hFE;
    master_id = 5'b0;
  endfunction

  task clean();
    command = 4'h0; address = 40'h0;
    master_id = 5'b0;
  endtask
endclass
```

```
class Bus;
  rand bit[15:0] addr;
  rand bit[31:0] data;
  constraint word_align { addr[1:0] == 2'b0; }
endclass

//Generate 50 random data values with quad-aligned addresses
Bus bus = new;
repeat(50)
  begin
    int result = bus.randomize();
  end
```

```
sequence request_check;
  request ##[1:3] grant ##1 !request ##1 !grant;
endsequence

always @(posedge clock)
  if (State == FETCH)
    assert request_check;
```

```
interface simple_bus; // Define the interface
logic req, gnt;
logic [7:0] addr, data;
logic [1:0] mode;
logic start, rdy;
endinterface: simple_bus

module memMod(simple_bus a, // Use the
    simple_bus interface
input bit clk);
logic avail;
// a.req is the req signal in the 'simple_bus'
    interface
always @(posedge clk) a.gnt <= a.req & avail;
endmodule
```

```
module cpuMod(simple_bus b,
    input bit clk);
...
endmodule


module top;
logic clk = 0;
simple_bus sb_intf; // Instantiate
    the interface
memMod mem(sb_intf, clk);
cpuMod cpu(.b(sb_intf), .clk(clk));
endmodule
```

```
module Design
( input  logic [7:0] d
, output logic [7:0] q
);
    ...
endmodule: Design
```

```
SC_MODULE(Design) {
    sc_in <sc_lv<8> > d;
    sc_out<sc_lv<8> > q;
    ...
};
```

```
if (EXPR) STMT1
else      STMT2

case (EXPR)
  EXPR: STATEMENT
  default: STATEMENT
endcase
```

```
if (EXPR) STMT1
else      STMT2

switch (EXPR) {
 case CONST: STATEMENT; break;
 default: STATEMENT;
}
```

```
input       clock;
input  int d;
output int q;

always_ff @(posedge clock)
begin :REGS
  q <= d;
end
```

```
sc_in<bool> clock;
sc_in<int>  d;
sc_out<int> q;

SC_METHOD(REGS);
sensitive << clock.pos();
...
void REGS(void) {
    q->write(d);
}
```

```
`include "disciplines.vams"

module rc_block(in, out, gnd);
  input in, gnd;
  output out;
  electrical in, out, gnd;
  parameter real R0 = 100.0;
  parameter real C0 = 0.0001;
analog begin
    I(in, out) <+ V(in, out) / R0;
    I(out, gnd) <+ ddt(V(out, gnd)) * C0;
  end
endmodule
```

I don't have pre-existing modules for loose comp

ddt(operand, [abstol|nature])	Time derivative
idt(operand, [ic], [assert], [abstol|nature])	Time integral
transition(operand, delay, trise, [tfall])	Transition
slew(operand, [rising_sr], [falling_sr])	Slew
absdelay(operand, delay, [max_delay])	Delay
laplace_zp(operand, [zeta], [rho], [epsilon])	Laplace, zero-pole form
laplace_nd(operand, [n], [d], [epsilon])	Laplace, numerator-denominator form
last_crossing(operand, [direction])	Last crossing
limexp(operand)	Limited exponential

These functions allow us to join and mix analog and digital simulations

Move values from analog part to digital, and viceversa