Oroboro

tom.sheffler@sbcglobal.net

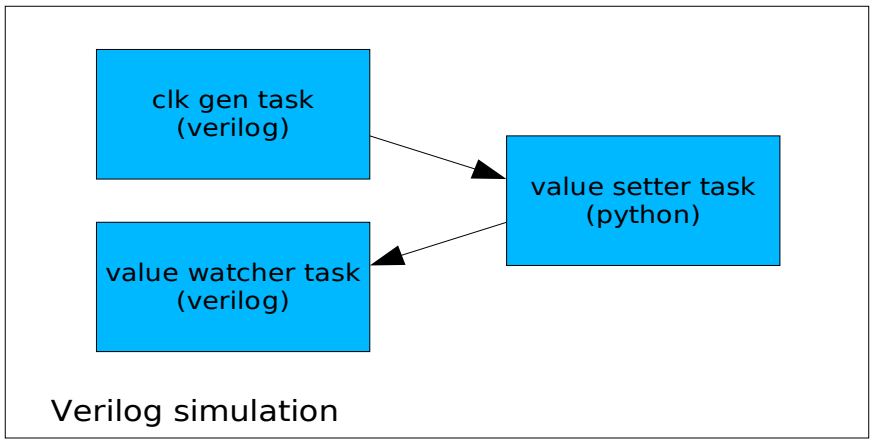
Oroboro is an approach to integrating Python with Verilog. Python generator functions are used to model simulation tasks in a cooperative multitasking style. An Oroboro task suspends its execution with a `yield` statement listing the reasons the task should be resumed. Typical reasons map directly to VPI callback reasons. Oroboro does not implement a scheduler of its own, rather, it relies on Verilog for all event scheduling through the use of callback functions. Oroboro is a simple, Python-based Verification Language for Verilog.

Because the Python interpreter is embdedded in the Verilog simulator using the VPI interface, testbench programs and simulation models written using Oroboro run with full access to the Verilog simulation netlist database and can access and change signal values. The resulting system is essentially a single simulator, with the Verilog scheduler scheduling both Verilog and Python tasks. Synchronization modeling issues are simplified because a single scheduling algorithm unifies both languages.

1 An Example

Key to Verification and testbench languages is the ability to create tasks that interact with concurrently executing Verilog simulation tasks. In the following section, we will show a very simple example in which an Oroboro Python task interacts with Verilog. In the example, there is a clock generator written in Verilog. A Python task synchronizes with the positive edge of the clock and then sets the value of a Verilog register to a new value.

A Verilog "watcher" task notices the value change and prints the new value of the register each time it changes. The structure of the intertask synchronization is illustrated in the figure below.



The Verilog code is very simple. It is presented below. The clock generator task is implemented in the "initial" block that toggles register "clk." The watcher task is the "always" block. Each of these become

independent tasks in a Verilog simulation.

```
module top();

    reg     clk;    initial clk = 0;
    integer value;  initial value = 0;
    integer i;
    integer res;

    initial
      res = $apvm("oro", "oroboro", "oroboro");

    /* clock generator task */
    initial
      for (i = 0; i < 10; i=i+1) begin
        #10 clk = 1;
        #10 clk = 0;
      end

    /* watcher task */
    always @(value) begin
      $display("VL: Change detected at time %t.  New value is %d",
               $time, value);
    end

endmodule
```

The Python task, "testtask," is presented below. Its first two lines create Oroboro signal objects attached to the HDL objects defined in Verilog. The main body of the task is an infinite loop that waits for a positive edge on the clock and then sets the HDL object "top.value" to a new value.

```
def testtask():
  clk_sig = signal("top.clk")
  val_sig = signal("top.value")
  i = 10

  while 1:
    yield posedge(clk_sig)          # wait until posedge
    print "PY: Setting 'value' to %d" % i
    val_sig.set(BV(i))              # set new Verilog value
    i = i + 10
```

When this example is run, we can see how the Verilog and Python tasks interact. The output of the run is the following. (You can run this example yourself. It is part of the distribution and is called "cosim" in

the examples subdirectory.)

```
PY: Setting 'value' to 10
VL: Change detected at time                  10.  New value is 10
PY: Setting 'value' to 20
VL: Change detected at time                  30.  New value is 20
PY: Setting 'value' to 30
VL: Change detected at time                  50.  New value is 30
PY: Setting 'value' to 40
VL: Change detected at time                  70.  New value is 40
PY: Setting 'value' to 50
VL: Change detected at time                  90.  New value is 50
PY: Setting 'value' to 60
VL: Change detected at time                 110.  New value is 60
PY: Setting 'value' to 70
VL: Change detected at time                 130.  New value is 70
PY: Setting 'value' to 80
VL: Change detected at time                 150.  New value is 80
PY: Setting 'value' to 90
VL: Change detected at time                 170.  New value is 90
PY: Setting 'value' to 100
VL: Change detected at time                 190.  New value is 100
```

Whereas Verilog tasks are static, Python tasks can be dynamically created as simulation proceeds. It is the dynamic nature of Python that makes Oroboro a powerful system for creating Verilog testbenches.

2 Overview

The Oroboro system introduces only a few new classes and functions. Because it is built as an APVM application, all of VPI is available as well, but a knowledge of VPI is not necessary to use Oroboro. This section gives a very brief overview of the major components of Oroboro. A good approach to using the information in this section is to examine the examples in the distributrion and then read the information presented here.

2.1 Simulation Time

In Oroboro, all simulation times are long integer values. These are converted directly into 64-bit `vpiSimTime` structures for Verilog as needed.

2.2 Bitvector

The BV class is imported directly from AVPM and provides a four-valued bitvector. Operations on bitvectors include typical bitwise AND and OR operations, as well as shifts, concatenation and other operations. These functions are bound to overloaded operators &, |, >>, <<, etc. The BV class also provides a variety of methods for creating random bitvectors and comparing “expect” values in simulation.

Bitvectors can be created using Verilog syntax. The following are some examples of using the BV class.

```
v1 = BV(“12'b000100010001”)
v2 = BV(“16'h34aX”)
```

```
v3 = BV("4'b0001") << 1          # the result is 4'b0010
```

2.3 Signal

An Oroboro signal is a wrapper object for a VPI handle that can have a vector value. Typically, a signal is attached to a Verilog wire or reg. Oroboro can watch for values changes on a signal. A signal also has two main methods: `get` and `set`.

The `get` method retrieves the current value of the Verilog object and returns it as a four-valued bitvector instance of class `BV`. The `set` method takes a bitvector value and assigns it to the Verilog object at the current simulation time. The `set` method also optionally takes a second parameter, a simulation time delay, which will cause the assignment to be scheduled at a future simulation time. By default, these scheduled assignments use the `vpiPureTransportDelay` method.

2.4 Event

Tasks may signal each other through event objects. An event is a simple object that has one method: `post`. A task may yield with the reason `waitevent(e)` to suspend its execution until another task performs a `post` operation on the event. Multiple tasks may wait on the posting of an event. The `post` method accepts an optional value, which is attached to the event at the time of posting. A waking task can examine the `val` field of an event if it is expecting a value to be attached.

An example of two tasks synchronizing by using an event is shown later.

2.5 Task

The task is the main worker class in Oroboro. A task is initialized with a reference to a generator, and the generator must yield with one or more instances of the `reason` class. A task is most naturally viewed as a thread of execution in the Verilog scheduler.

The arguments to the task constructor are the generator function or method and the arguments that are to be passed to the generator when the task starts. A small example is shown below in which a task is started that prints a message after a simulation time delay of 10 ticks.

```
def genfn(msg, delaytime):

  # Print the message after a delay
  yield timeout(delaytime)
  print "This is the message:", msg

new_task = task(genfn, "Hello World!", 10)
```

A task may yield with one or more reasons. If there are multiple reasons, then each signifies an *alternative* reason that the task may be resumed. For example, a task that suspends until *either* a signal changes or a delay time expires might have yield clause that looks like the following.

```
yield timeout(delaytime), sigchange(sig)
```

When a task yields, the function's execution is suspended and a VPI callback is scheduled for each of its yield reasons. Whichever callback occurs first wakes the suspended task and causes it to resumed. Other callbacks – even if they eventually occur – are simply ignored by the task.

A task supports only one publicly usable method: `kill`. A running task can be prematurely terminated by another task using its `kill()` method.

A task also has a current status, which is available in its `status` instance variable. Those states currently

implemented are: BORN , RUNNING, WAITING, EXITED and KILLED. A BORN task is one that has been initialized but has not started executing yet because its first callback has not yet occurred. A task is RUNNING if it has resumed because of a callback. Only one task is ever in the RUNNING state. A WAITING task is one that has yielded (suspended) and is waiting for a callback. An EXITED task has exited by executing a return statement, and a KILLED task is one whose `kill()` method has been called.

2.5.1 A Brief Description of Python Generator Functions

This section is a detour into a very short explanation of Python generator functions for those who have not encountered them before. Python generators are a very limited form of coroutine. This form of coroutine only returns execution to its caller and is sometimes called a "semicoroutine" for this reason. They are most often used to produce sequences of values, without creating a list of all the values at once.

The example below shows how to write a generator function that produces the Fibonacci sequence, one value at a time. The fib function is written as an infinite loop that produces one value of the sequence each time it resumes execution. The yield statement causes the state of the generator function (fib) to be saved and returns the value to the caller.

Example 2

```
def fib():
  a = 0
  b = 1
  while 1:
    yield b
    a = b
    b = a+1

# example of getting the first 10 values
gen = fib()
for i in range(10):
  val = gen.next()
  print val
```

Generator functions have many applications. They are useful in lexical analysis, to produce the tokens of an input stream only as needed. They are also a convenient way to write functions that represent state machines. Oroboro uses them as a very simple and lightweight simulation task.

In Oroboro, the ultimate caller of all tasks is the Verilog scheduler itself - layered through a series of C and Python function calls. When an Oroboro task invokes the "yield" statement, it suspends its execution at at the yield statement, and its state is saved in the Python heap. Because resumption "reasons" are due to the actions of the Verilog scheduler, a task resumes as the "callee" of the scheduler.

This approach is simple, low-overhead (generator functions have almost no additional overhead in Python), and elegant. The fact that the Verilog scheduler is the "caller" of all tasks, means that once the programmer understands the Verilog scheduler, he or she understands *all* of the scheduling interactions that may occur in an Oroboro program. Other Verilog verification lanagues often interpose an additional scheduler, and the interactions between the two schedulers can lead to confusion.

2.6 Reason

When a task wishes to relinquish control of execution back to the Oroboro task dispatcher, it executes a “yield” statement with a list of reason objects describing the conditions under which the task should be

resumed. Reasons are so named because most map directly to VPI callback reasons: `cbValueChange`, `cbAfterDelay`, etc, and are the "reason" a task should resume execution. The class “reason” is an abstract base class and should not be used directly. Objects of the derived classes described are appropriate for use with the “yield” statemet.

The family of reasons forms a very simple, shallow class hierarchy, as shown in the figure below. A description of each class follows.

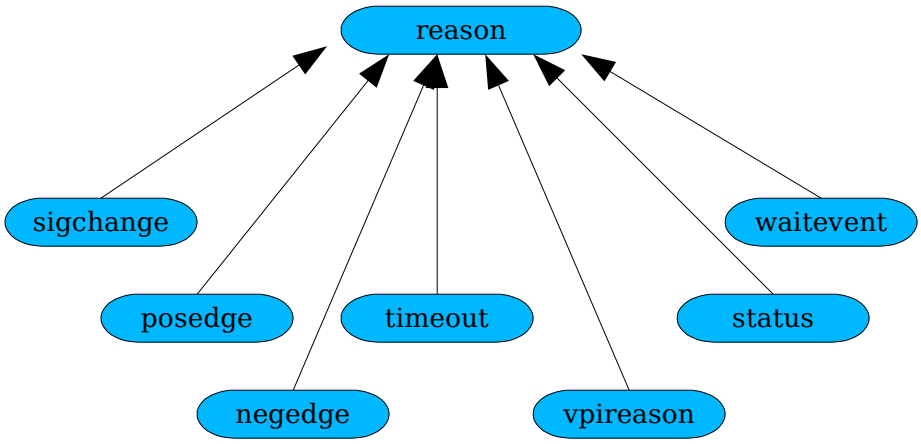


Figure 1:The Reason Class Hierarchy

2.6.1 sigchange(sig)

The `sigchange` reason is used to indicate that the task wishes to be resumed upon any signal change to the signal object, sig. A `sigchange` reason causes a task to resume in a `cbReadWriteSynch` region (see a description of the Verilog scheduling algorithm description for details).

An example Oroboro task that waits for any change in the Verilog net “top.foo” is shown below. Once started, this task loops indefinitely, printing each new value of the net “top.foo.”

```
def foo_task():
  sig = signal(“top.foo”)        # attach to Verilog net

  while 1:                       # infinite loop
    yield sigchange(sig)         # like @(top.foo) in Verilog
    print “New value is”, sig.get() # print new value
```

2.6.2 posedge(sig)

The `posedge` reason is used to indicate that the task wishes to remain suspended until the value of the signal transitions from a 0 to a 1. It is a special case of the `sigchange` reason.

An example Oroboro task that waits for positive edges on a Verilog clock named “top.clock” is shown below. On each positive edge of the clock, a counter is updated and its new value is printed.

```
def pos_edge_counter():
```

```
  clk = signal("top.clock")      # attach to Verilog net
  i = 0                          # integer counter

  while 1:                       # inifinite loop
    yield posedge(clk))          # wait for pos clock edge
    i = i + 1
    print "Clock edge number", i, "seen"
```

2.6.3 negedge(sig)

The `negedge` reason is used to indicate that the task wishes to remain suspended until the value of the signal transitions from a 1 to a 0. It is also a special case of the `sigchange` reason.

An example Oroboro task that waits for negative edges is shown below. This task is a simple stimulus generator that drives Verilog from Python. The task loops indefinitely waiting for a negative edge on Verilog net "top.clock." On each negative edge, the task increments its counter, "i", and drives its new value onto the Verilog register "top.drv."

```
def neg_edge_driver():
  clk = signal("top.clock")      # attach to Verilog net
  drv = signal("top.drv")        # attach to Verilog driver reg
  i = 0                          # integer counter

  while 1:                       # inifinite loop
    yield negedge(clk))          # wait for pos clock edge
    i = i + 1
    print "Putting value", i, "on Verilog driver"
    drv.set(BV(i))               # drive Verilog reg
```

2.6.4 timeout(t)

The `timeout` reason is used to indicate that the task wishes to be resumed after the time, t, has elapsed. It maps directly to a VPI `cbAfterDelay` callback. Note that this callback will resume the task at the beginning of a Verilog timestep, before other events have been processed. This behavior is appropriate for tasks that initiate signal changes immediately following a timeout, like the clock generator example below.

The following Oroboro task example alternately drives the Verilog reg "top.clock_drv" to 0 and 1, producing a clock. The task loops indefinately. It first suspends itself for half of the clock cycle and then drives the net to "1". It then suspends itself again for the rest of the clock cycle and then drives the net to "0".

```
def clockgen_example():
  clk_d = signal("top.clock_drv")

  while 1:                        # infinite loop
    yield timeout(cycle_time/2)   # delay until rise edge time
    clk_d.set(BV("1'b1"))
    yield timeout(cycle_time - (cycle_time/2))  # delay again
    clk_d.set(BV("1'b0"))
```

2.6.5 waitevent(e)

The `waitevent` reason is used with an event, e, to indicate that the task should be resumed when another task performs a `post` operation on the event e. The `post` action causes the task to be immediately resumed. The example below shows how an event could be used to synchronize two free-running tasks.

```
def task1(e):
    yield timeout(10)           # delay 10 Verilog ticks
    print "Posting the event in task1."
    e.post()                    # wake up task 2

def task2(e)
    yield waitevent(e)          # sleep until task 1 wakes us
    print "Event seen in task2."

# The parent creates a new event and starts both tasks.
e = event()                     # create shared event
t1 = task(task1, e)             # start task1
t2 = task(task2, e)             # start task2
```

The user may also attach a value to the event for passing values between the posting and the waiting task.

2.6.6 status(task)

The `status` reason is used with a task to indicate that the task should be resumed when the argument task undergoes a status change. Currently, the only status change implemented is when a task exits. Its status transitions to EXITED. The pattern for starting a task and waiting for it to exit is the following.

```
def taskparent():
  ...
  t = task(taskchild, args ...) # start a child task
  yield status(t)               # wait for child to complete
```

2.6.7 vpireason(cbreason)

Arbitrary VPI callbacks may be scheduled using `vpireason`. Typical callback reasons are `cbNextSimTime`, or `cbReadOnlySynch`. This class is intended for users with some knowledge of the VPI interface. A simple example using this type of reason is to construct a "strobe"-type synchronization point. When dumping signals, it is important that the dumping function gets the final-final value of the net it wants to dump. To do this, the task must first wait for a value change, and then must wait until *all* updates to that signal have been processes. The following shows how to do this in Oroboro.

```
def dumper(sig):
  print "Dumping signal %s" % sig
  while 1:

    # wait until 'sig' changes
    yield sigchange(sig)

    # wait again for ReadOnlySynch region
    yield vpireason(apvm.cbReadOnlySynch)
```

```
print “Value: “, sig.get()
```

Of cousre, a more useful dumping function would print the signal name and the current simulation time, but these details have been omitted here.

2.7 Task Functions

A running task can query the environment to obtain information about itself or the reason that was triggered in a yield statement. The following module-level functions are available for this purpose.

2.7.1 currentreasonindex()

The reasons in a yield clause are assigned identifying numbers beginning at 0 when the yield statement is executed. When a task resumes, this function can be used to get the index of the current reason in the list of reasons given to the “yield” statement.

Below is an example task that waits for one of two things to happen first by yielding with two reasons. The task goes to sleep until either the signal “s” changes value or until 10 simulation time ticks have elapsed. When it resumes it uses the “currentreasonindex” function to determine which of the two events occurred first, printing a message. The task then exits.

```
def task_ex(s):
  ...
  # resume if the signal changes, or after 10 time units
  yield sigchange(s), timeout(10)    # assigned index [0, 1]

  if currentreasonindex() == 0:
    print “The signal changed value before the timeout elapsed.”

  if currentreasonindex() == 1:
    print “The timeout elapsed before the signal changed.”

return
```

2.7.2 currentreason()

When a task yields, it passes a list of reasons to the Oroboro dispatcher. When the task resumes, the particular reason from the list of reasons is given back to the task through the “currentreason” function.

The example below repeats the previous example using the “currentreason” function instead of the “currentreasonindex” function. The standard Python function “isinstance” is used to determine the type of the object, x, to determine if it is a sigchange or a timeout. The achieved effect is the same as the previous example.

```
def task_ex(s):
  ...
  # resume if the signal changes, or after 10 time units
  yield sigchange(s), timeout(10)

  # when the task resumes, x is one of the reason objects above
  x = currentreason()

  if isinstance(x, sigchange):
    print “The signal changed value before the timeout elapsed.”
```

```
  if isinstance(x, timeout):
    print "The timeout elapsed before the signal changed."

return
```

In the example of the previous section, the sigchange reason is assigned an index of 0 in the yield clause, and the timeout reason is assigned an index of 1.

2.7.3 currenttask()

This returns the task object corresponding to the currently executing task. Instance variables of interest to programmers are the id of a task, its __name__,and its parent task. Converting a task to a string produces a printable reprentation of the task that uniquely identifies it.

```
def task_ex(x):
  yield sigchange(x)
  me = currenttask()   # get my task instance
  id = me.id           # get task id, an integer
  name = me.__name__
  parent = me.parent   # my parent task
  pid = parent.id
  print str(me)        # print representation of me
```

2.7.4 currenttime()

This returns the current Verilog simulation time as a long integer. The time is obtained from the VPI callback responsible for resuming execution. A notable thing about simulation times in Oroboro is that they are represented as Python long integers, which are infinite length integers. Computations on times are simplified for the user because all computations on times are coerced into infinite length arthmetic. The user need not worry about arthmetic overflow.

An example task using the "currenttime" function is shown below. This very simple task prints the current time, sleeps for 10 simulation ticks and then prints the new time is shown below. It also computes the difference between the two times.

```
def time_task():
  t1 = currenttime()
  print "The current time is", t1
  yield timeout(10)
  t2 = currenttime()
  print "The new time is", t2
  diff = t2 - t1
  print "The time difference is", diff
```

2.7.5 currentsystf()

This returns the APVM `systf` object corresponding to the currently executing Oroboro application. (Though not typical, there may be more than one Oroboro application running simultaneously.) It is through this object that configuration information may be obtained, for example. See the APVM documentation for a description of the methods of the `systf` class.

A typical use for this function, would be to access configuration parameters for the Oroboro instance of

which this task is part.

```
def task_ex():
  stf = currentsystf()     # which apvm.systf instance
  param = stf.my_config_param_string("userparam")
  ...
```

It is typical to see code such in the main startup task of an Oroboro application

2.8 Other Functions

A few other functions are available to structure message output and error reporting.

2.8.1 taskmsg(s)

This function prints a message to the log file (using apvm.vpi_print) with the current time and task id in a nice format.

2.8.2 error(s)

This is the APVM error function. It is used to indicate a simulation error. The global APVM error counter is incremented and the message, s, is printed to the log file. If APVM has been configured to link this error count to a Verilog variable, then the count is synchronized with the global Verilog counter.

2.8.3 warning(s)

This is the APVM warning function. It is used to indicate a simulation warning. The global APVM warning counter is incremented and the message, s, is printed to the log file. If APVM has been configured to link this error count to a Verilog variable, then the count is synchronized with the global Verilog counter.

2.8.4 vpi_print(s)

This is the Python interface to the raw VPI print function which normally prints to both the screen and the logfile. This function does not prepend the simulation time or a task identifier.

2.8.5 traceon()/traceoff()

Verbose tracing of the event processing performed by Oroboro may be turned on and off with these functions. Viewing event tracing is useful for debugging and for understanding implementation details of Oroboro. What you may notice by running with tracing on is that the Oroboro examples create an enormous number of tasks on the fly for even the simplest of programs.

2.8.6 simstop()

Call the simulator's $stop function. Normally, this returns the user to an interactive prompt.

2.8.7 simfinish()

Call the simulator's $finish function. Normally, this ends the simulation. Because Oroboro has registered an end-of-simulation callback, the Oroboro error and warning report also get printed.

2.8.8 simreset()

Call the simulator's $reset function. While this restarts the simulation at time 0, it does not reset Oroboro.

3 Brief User's Guide

This section gives a few pointers for using Oroboro. The examples here discuss some more interesting aspects of the package.

Because Oroboro is an APVM application, the first step is to build APVM and make sure it is operating properly by running a few APVM examples.

Oroboro is most typically integrated with Verilog in the following way. In the top level, add the AVPM call that creates an Oroboro instance and give it the name "oro."

```
initial begin
  reg [31:0] res;
  res = $apvm("oro", "oroboro", "oroboro");
end
```

Oroboro requires two configuration parameters to name the Python module and function of the main user task. These are most typically given on the Verilog command line as plusargs, but could also be configuration file parameters. A typical example follows.

```
verilog ... +oro:module=mymodule +oro:task=mymain
```

The function `mymain` ***must*** be a generator function, and this leads to a strange requirement. The main function must have a "yield" statement in it even if it serves no purpose. The main function is passed a single argument: the APVM `systf` instance corresponding to the root Oroboro instance. The user can use methods of this instance to retrieve configuration parameters, among other things.

For example, if your application requires a command line parameter, you can get it in your main function as shown in the following example.

```
def mymain(systf):

  fooval = systf.my_config_param_string("foo")
  t = task(main_worker_task, fooval)
  yield timeout(0)
  return
```

And on the command line, the foo parameter is given as a plusarg.

```
verilog ... +oro:foo=value ...
```

As a convenience, Oroboro allows you to set the global Python random number generator from the command line as well. If you desire multiple random number generators, you're in luck. Python provides a number of random number generation services - and their use has nothing to do with Verification, specifically, so they aren't described here.

```
verilog ... +oro:seed=N
```

3.1 Building the top-level

Following a few suggestions helps simplify the top-level simulation harness for your Verilog module. At this point in time, Oroboro offers no automatic support for creating Verilog drivers or wires, but adopting this straightforward methodology makes it simple.

3.1.1 For each output port

It is sufficient to define a wire and attach it to the output port. An Oroboro signal can attach to this wire to read values using the `get` method.

3.1.2 For each input port

It is sufficient to define a reg and attach it to the input port. An Oroboro signal can attach to this reg to drive values using the `set` method.

3.1.3 For each inout port

It is best to create a wire and a reg. For a port called foo, create wire 'foo' and reg 'foo_d' (driver). In Verilog, use an assign statement like this.

```
wire foo;
reg foo_d; initial foo_d = 1'bz;
assign foo = foo_d;
```

Attach the wire to the module under test. Then, in your Oroboro code, create a read-only signal for 'foo' and a write-only signal for 'foo_d'.

```
s = signal("top.foo")
s_d = signal("top.foo_d")
```

When you do not want to drive values, set the driver value to all Zs.

```
s_d.set(BV("1'bz"))
```

3.1.4 For internal nodes

Consider these the same as other "ports" and define top-level entities for them, following the conventions above. For instance, if there is a node called "node" with verilog path "`NODEPATH" buried in the design that you wish to observe and drive, create this harness.

```
`define NODEPATH top.dut.x.y.node
wire node;
reg  node_d;  initial node_d = 1'bz;
assign node = NODEPATH;
assign NODEPATH = node_d;
```

The use of the `define slightly simplifies the code and makes it possible for people familiar with Verilog (but not Python) to keep these paths up to date. Oroboro code then needs to only refer to the invariant paths "top.node" and "top.node_p" when it creates signals.

3.2 Connecting to Verilog Nets

Two methods are available for connecting to Verilog objects. The first method is to simply use the full pathname of the verilog object as the initializer to a `signal` object.

```
s0 = signal("top.sig0")
s1 = signal("top.sig1")
s2 = signal("top.sig2_d")
```

A second method is to use the `apvm.systf` object passed to the main task to access verilog objects that are passed as arguments in the `$apvm` call that spawned Oroboro.

```
module top;
  ...
  res = $apvm("oro", "oroboro", "oroboro", sig0, sig1, sig2_d)
  ...
endmodule
```

In the Python main task, the VPI handles for these objects can be accessed through the `vobjs` list in the `apvm.systf` object. These handles may also be used as initializers for the `signal` class.

```
main(systf):
  s0 = signal(systf.vobjs[0])
  s1 = signal(systf.vobjs[1])
  s2 = signal(systf.vobjs[2])
  ...
```

The first method is the more flexible of the two and should be used if simplicity and flexibility are key. The second method has advantages in terms of modularity and possibly simulation performance. If the Oroboro application is internal to a module that is instantiated multiple times, then this approach may simplify connecting the right signals to the right Oroboro instance.

```
module top;
  ...
  res1 = $apvm("oro1", "oroboro", "oroboro", sig0, sig1, sig2_d)
  res2 = $apvm("oro2", "oroboro", "oroboro", sig3, sig4, sig5_d)
  ...
endmodule
```

Most importantly, some Verilog compilers perform optimizations based on knowledge of which nets are accessed by the Verilog code and PLI applications. In the first approach, global access to all Verilog objects must be enabled (since any object is accessible through its pathname) and this may disable code optimizations. If the second approach is used consistently, the Verilog compiler has complete knowledge of the nets accessed and can apply full optimization.

3.3 Tasks and Typical Patterns

A task is a Python generator function that yields with a reason, tuple of reasons or list of reasons. Such a generator function becomes a running simulation thread when it is used as an initializer for a new task instance. When a new task is spawned, it becomes a free-running thread. The child thread does not begin execution until after its parent yields.

To wait for completion of a child task, the parent must yield with a `status` reason. In typical use, a parent that wishes to spawn a child and wait for its completion uses the following pattern.

```
child = task(childtask, args, ...)
yield status(child)
```

This pattern can be shortened to the following if a reference to the task object is not needed except as an argument to `status`.

```
yield status(task(childtask, args, ...)
```

A task may suspend execution and resume again at the same simulation time by using the following construct.

```
yield timeout(0)
```

This is equivalent to introducing a #0 delay in a task in Verilog. The task will suspend its execution, but the Verilog scheduler will resume it again in the same time step.

3.4 Composing Tasks to Form Transactions

Tasks and the `status` reason are used together to compose sequences of tasks, or to compose tasks into larger transactions. The following example shows the definition of some low-level tasks and how they might be built up to form a transaction.

Example 3

```
def cycle(clk):
    yield posedge(clk)
    yield negedge(clk)

def cmdpacket(clk, cmd_d, opcode):
    cmd_d.set(opcode)
    for i in range(2):
        yield status(task(cycle, clk))
    cmd_d.set(BV("5'bzzzzz"))

def datapacket(data_d, data):
    ...

def transaction(clk, cmd_d, opcode, data_d, data):
    yield status(task(cmdpacket, clk, cmd_d, opcode))
    for i in range(4):
        yield status(task(cycle, clk))
    yield status(task(datapacket, data_d, data))
    for i in range(4):
        yield status(task(cycle, clk))
```

The task “cycle” is a simple task that waits for two edges of a clock and exits aligned to the “negedge” of the clock. This is significant, because a waiting task following this one will awake synchronized with that edge. The task “cmdpacket” places an opcode on a signal driver “cmd_d” for two cycles and then ceases driving the driver by placing Zs on it. The definition of “datapacket” is omitted, but its function is to drive data on a signal “data_d” for some number of cycles. The top level “transaction” task drives a command packet, waits for four cycles, drives a data packet, and then waits another four cycles.

3.5 Using Classes to Group Tasks

Classes can be used to group together tasks and transactions that apply to a particular set of signals. This is a good way to structure BFMs. For each port group, a BFM object is initialized with the signals that comprise the port. Because Python generator functions may be method bodies, the BFM class can contain both simple methods and task methods.

Example 4

```
class reg_bfm:

  def __init__(self, cyc_c, rw_d, addr_d, data, data_d):
    self.cyc_d = cyc_d         # register cycle driver
    self.wr_d = wr_d           # write/read select
    self.addr_d = addr_d       # address drive
    self.data = data           # data sample
    self.data_d = data_d       # data drive

  def write(self, waddr, wdata):
     yield timeout(10)
     self.cyc_d.set(BV(“1'b1”))
     self.wr_d.set(BV(“1'b1”))
     self.addr_d.set(waddr)
     self.data_d.set(wdata)
     ... other details of write BFM omitted

  def read(self, raddr, rdatamut):
     yield timeout(10)
     ... details of read BFM omitted
     rdatamut[0] = self.data.get()   # rdatamut is a mutable object

def main(systf):
  # create one register interface over one port group
  reg0 = reg_bfm(top.cyc0, ....)
  # create another register interface over another port group
  reg1 = reg_bfm(top.cyc1, ....)

  # example of calling one of the operations
  yield status(task(reg0.write, addr, data))
```

In this example, the register interface BFM supports two methods: write and read. The class initializer

connects the instance to a particular set of register interface signals. Multiple register interfaces may be instantiated over different ports with no ambiguity. The write task accepts an address and data and applies them to the ports. The read task accepts an address and a list in which to place the return data. When the read task samples the read data, it places it in this mutable object. At that point in time the caller can examine the read data.

4 Advanced Topics

Python is a dynamic interpreted language. With Oroboro, tasks and reasons are simply objects that may be generated on the fly. The dynamic nature of Python allows for very flexible code. This section briefly touches on some aspects of this topic.

4.1 VPI Underneath

Because nearly all of VPI is available through APVM, an Oroboro application may query the Verilog netlist database to obtain information about simulation objects. For example, a general-purpose testbench could be written that sizes itself to the width of a particular port.

```
from oroboro import *
from apvm import *

s = signal("top.port")
width = vpi_get(vpiSize, s.vpih)
s.set(BV("1'bx") * width)  # set to all X
...
```

Using VPI, one can access arbitrary HDL objects, traverse the HDL hierarchy and read from and write to memories, among other things.

4.2 Generating Tasks Dynamically

A task is spawned by initializing a new task object. Consider the watcher task shown earlier. Given a list of signals, `ss`, a list of watcher tasks could be spawned as follows. This example uses Python list comprehension syntax.

```
tt = [ task(watcher, x) for x in ss ]
```

This construct would start a new watcher for each signal in the list.

4.3 Generating Reasons Dynamically

The argument to the yield statement in a task must be a reason, tuple of reasons or list of reasons. For readability, reasons are often written in a comma-separated list. (This comma-separated list is actually just a literal Python tuple.) However, the list of reasons could be generated dynamically.

For example, to yield waiting on a status change of any of the tasks spawned above, we could do the following.

```
reasons = [ status(x) for x in tt ]
yield reasons
```

This construct gives the same result as writing out

```
yield status(tt[0]), status(tt[1]), status(tt[2]), ...
```

4.4 Synthesizing Synchronization Primitives

The dynamic capabilities of Python and Oroboro create some interesting possibilities for creating new synchronizataion primitives. Most multitasking simulation languages offer some sort of fork/join primitive. In Oroboro, it is not difficult to synthesize our own.

In the example below, the argument to the join task is a list of tasks. The join task loops until all of the tasks in the list have exited and then it exits itself.

```
def join(tasks):

  reasons = [ status(x) for x in tasks ]
  while (len(reasons) != 0):
    yield reasons
    tasks.remove(currentreason().t)
    reasons = [ status(x) for x in tasks ]
  return
```

Given the list of tasks created above, a task would use `join` in the following way.

```
yield status(task(join, tt))
```

This yield statement spawns a new join task whose arguments are the list of running tasks, `tt`. The yield statement resumes after the join task exits.

5 Conclusion

This first implementation of Oroboro is an experiment in using Python generators as a simple mechanism for representing Verilog simulation tasks. In contrast to the capabilities of more general threading packages, a generator is a a very restrictive kind of thread that can only suspend its execution when its stack is empty. While limiting, the construct is shown to be useful in practice. Whether this is useful enough for serious work remains to be seen.

Future work will add features of more developed "verification languages" to Oroboro through class libraries. Constrained random generation is usually considered one such feature. Python already has a sophisticated package of random number generators, and the most straightforward way of adding constrained random constructs for Verification is through a class library. Instrumentation that aids in tracing and viewing complex sequences of tasks (ala transactions) would be helpful, and so would a class library that provides constructs for collecting run-time coverage data. Ideally, coverage would implemented as a Python wrapper around an existing C-based coverage library. Temporal expressions for implementing checkers might also be defined as a class library at some point. Because of the abilities to create tasks dynamically and to store execution state in generators, temporal expressions seem to be an interesting next step. Class libraries can sometimes be cumbersome, however, and the languages that support these features as first-class language constructs may prove to be more convenient to use.

At this stage of its development, Oroboro is a robust base platform for the development of more advanced features. It provides the basic mechanisms needed to interact with a running Verilog simulation. With the addition of the right collection of class libraries, it could become a powerful Verification environment.

6 References

The following have been useful, stimulating or are somehow relevant.

http://www-106.ibm.com/developerworks/linux/library/l-pythrd.html
"Charming Python: Implementing "weightless threads" with Python generators"
This article gives a good description of using generators as ultra-lightweight threads and makes a case for why they have almost no overhead. Understanding this article was the "ah-ha" I needed to understand how to use Python generators with the Verilog scheduler.

IEEE P1364-2005 Draft 3
This is the proposed 2005 Verilog standard. Besides the usual language reference manual, this document includes a *detailed description of the Verilog scheduling algorithm* and a complete reference for VPI.

http://www.sutherland-hdl.com/publications.shtml
Stuart Sutherland's 806 page book "The Verilog PLI Handbook: A Tutorial and Reference Manual on the Verilog Programming Language Interface,Second Edition" is the best reference on VPI I have found that includes detailed examples.

http://www.pragmatic-c.com/gpl-cver
GPLCVER is a version of the CVER verilog simulator that is freely distributable. GPLCVER is an interpreted Verilog simulator. Both APVM and Oroboro work well with it.

http://www.icarus.com/eda/verilog
Icarus is an open source verilog compiler and runtime system. It first compiles Verilog to bytecodes and then runs the bytecode result. It has a very good VPI implementation and is compatible with APVM and Oroboro.

http://www.open-vera.com
Vera is a testbench and simulation language that can be used with Verilog. Vera code is normally compiled and then run in a PLI run time system that is linked with a Verilog simulator.

http://www.testbuilder.net
TestBuilder is a very sophisticated testbench system written in C++. It can be used with both Verilog and VHDL. TestBuilder uses either pthreads or QuickThreads as its underlying mechanism for representing simulation tasks.

http://www.ieee1647.org
The Specman "e" language is now an IEEE standard, P1647. This site explains some of the features of e that make it an excellent verification language.

http://apvm.sourceforge.net
APVM embeds the Python interpreter in Verilog using the VPI interface. Oroboro is distributed with

APVM as an example application.

http://jandecaluwe.com/Tools/MyHDL/Overview.html
MyHDL is an interesting project that advocates using Python as a Hardware Description Language. While the focus is on creating synthesizable Python, it can also be be used as a hardware verification language. MyHDL presents a use of generators and the yield statement very similar to the one presented here.

http://simpy.sourceforge.net
SimPy is an event driven simulation system written in Python. It too represents "tasks" using Python generators.

http://kamaelia.sourceforge.net/
Kamaelis is an experimental testbench for implementing network protocols suited to media delivery systems. It uses Python generators to implement a light-weight event-driven programming system.

Table of Contents