A Python VPI Module

tom.sheffler@sbcglobal.net

1 Introduction

APVM (A Python VPI Module) is a framework for writing Verilog PLI applications in Python. PLI applications are foreign C routines that are linked into a Verilog simulator to add new functions ("system functions") to a Verilog program. Simple PLI applications can be as trivial as one that adds a single C math function ("cos" for example) to Verilog. More complex PLI applications add entire verification languages or waveform analysys to Verilog. APVM links the Python interpreter as a PLI application and exposes the VPI interface to the Python PLI Application writer.

The C-based VPI interface is very general, but complicated enough that it can be a challenge to implement even the simplist PLI Application. APVM does much to simplify the process by introducing an object-oriented framework over the basic PLI interface. In APVM, each Verilog instance of a call to the $apvm system function is mapped to an instance of the "avpm.systf" class in Python. The base class defines empty methods corresponding to simulation events that result in function callbacks. Most applications are written by simply overriding the base class definitions in an extension class. Most commonly, the “calltf” method is overridden. This method is called whenever the $apvm instance is called by the Verilog code.

Here is an example of a very simple APVM application.

`helloworld.v`

```
module top;
initial
  $apvm("example", "helloworld", "hw");
endmodule
```

`helloworld.py`

```
from apvm import *

class hw(systf):
  def calltf():
    print "Hello World!\n"
```

The Python module "helloworld" defines a class "hw" whose base class is APVM's systf. The calltf method is over-ridden to print the now famous string. In Verilog, we call $apvm with a unique identifying name ("example" here), the name of the module ("helloworld") and the name of the class of which to create an instance ("hw"). APVM initializes the instance of the Python hw class and does all of the work of arranging for its calltf method to be called whenever Verilog execution reaches the $apvm call.

APVM does more than just map VPI callbacks to method calls. It exposes the entire VPI programming interface to Python. VPI object pointers and pointers to VPI structs are mapped to opaque

data types in Python. VPI functions for traversing the Verilog hierarchy or accessing and setting values are available, as are functions for creating simulation callback events. All VPI #define values are mapped to constants in the apvm module. As an application framework, APVM also provides a 4-valued bit-vector data type, a predefined mechanism for error and warning handling, and a structured method to configure $apvm instances from configuration files and Verilog +plusargs. Thus, it is more than a simple Python wrapper for VPI.

Because of the simplicity that APVM brings to writing a PLI application, it opens up interesting possibilities for projects that might be considered difficult if starting from scratch in C. Here are a few ideas.

- Rapid prototyping of a simplified reference model may be accomplished in Python early in the design of a Verification testbench so that testbench development can proceed before RTL is available.
- Highly mathematical algorithms are more easily explored in languages other than Verilog. Before committing to RTL, using Python's infinite length integers or add-on numerical libraries may prove to be a more agile environment for exploring algorithmic tradeoffs.
- Verilog is not necessarily a good language for implementing job reporting, manipulating files and formatting text. These things are easy in Python. One can even imagine a sophisticated reporting system that logs all simulation data to a relational database on another machine using a database connection. Python's database connection facility makes this easy.
- Custom GUI applications for displaying simulation progress become attractive when you can code them in a few hours. Python's Tkinter interface to Tk/Tcl is appropriate for this.
- Sometimes it is desirable to collect information about the system a job is running on in a large distributed simulation farm and adding it to the log file. Such a task is trivial in Python but not easy in Verilog.

1.1 Why Python?

Python is an object-oriented interpreted language with a rich set of built in datatypes. It also comes with a large library of useful modules for doing things like compressing files, producing HTML, serving data over pipes or opening up database connections. It can integrate with Tk/TCL for graphical applications. Contributed libraries are available from an enormous number of sources. At this stage of its development, Python is a stable computing platform with a huge community of users. It has been in development for over 10 years.

While the Python programming model is not necessarily compatible with any of the features of Verilog, as a replacement for C for orchestrating VPI calls it presents a very friendly environment. Python offers a great prototyping environment, and modern programming constructs like exceptions help users write clear code. Being byte-compiled it offers good performance for many applications.

1.2 Related Projects

There have been only a few projects attempting to integrate Python and Verilog. One is called ScriptSim by Dave Nelson [http://nelsim.com]. This approach runs Python as a separate process and communicates with Verilog via a pipe. The communication protocol allows object values to be transferred back and forth, and allows Python to synchronize with the simulator and schedule callbacks. ScriptSim has a nice Tk/TCL interface as well.

Another effort is the ScriptEDA project at Berkeley by Pinhong Chen [http://www-cad.eecs.berkeley.edu/~pinhong/scriptEDA]. This effort explored using SWIG (a powerful package for mapping and linking C libraries into interpreters) [http://swig.org] to map the VPI interface into Python.

The project did not add much structure or value beyond this translation.

Another related project is "Pivot", a commercial product that integrates Perl with Verilog through the PLI 1.0 interface [http://www.greenl.com/]. Pivot has a software layer that seems similar to APVM but also provides many higher-level constructs on top of that layer to allow the design of complex testbenches in Perl.

EDA industry efforts to link programming languages into Verilog for test generation and modeling include Vera, Specman and System C. All of these efforts are motivated by the fact that descrbing anything other than hardware is difficult in Verilog, whereas most testbenches are large software efforts. These languages all add features for describing "tasks" and synchronization primitives in a meaningful way, and hide the underlying PLI interface. In constrast, APVM opens up the VPI interface to the Python programmer. Oroboro is the higher-level software layer that presents a simplified abstraction of tasks and synchronization.

1.3 Structure of this Document

The next section presents a slightly more full-featured example. The example is a “memory” model implemented using a Python hash table. The example shows a fairly useful application implemented in just a page of code. A similar application coded in C would be much more complicated. Understanding this example is a good place to start to get familiar with APVM.

Section 3 discusses the implementation of APVM. This may be of interest to those familiar with implementing VPI applications, or those wondering how to embed Python into a simulator. This section may be too low-level for those just wanting to know the basics of APVM and wanting to trying out some examples.

Section 4 discusses the “Framework” facilities provided by APVM. While most of the examples just show how simple applications are constructed, in a full-featured PLI application, there is a need for uniform ways to obtain configuration information and present error and warning messages. APVM provides the basics.

Section 5 summarizes this article and presents some directions for future work. Finally, Section 6 presents some miscellaneous notes about the APVM/Oroboro project.

2 Examples

The following example demonstrates a sparse memory implemented using a hash table. The application would be called from Verilog with three Verilog register arguments indicating a command (write or read), an address and the data. When called, the application attempts to turn the address value into a long integer. The long integer is used as the key to the memory hash table, "mydata." If the conversion to a long fails because there are x's or z's in the address, the call returns 0 and fails.

The rest of the application is straightforward. If a write command is requested, the data value is retrieved as a four-valued binary string and stored in the hash table. If a read is requested, the memory location is looked up and applied to the data register using "put_value."

```
verilog code
```

```
reg mem;

on (posedge clk)
  $apvm(mem, "memex", "memex", cmd, addr, data);
```

memex.py

```
class memex(systf):

def soscb(self):
  self.cmd  = self.vobjs[0]
  self.addr = self.vobjs[1]
  self.data = self.vobjs[2]

  self.mydata = { }  # empty hash table

  self.fmt = pack_s_vpi_value(vpiBinStringval, "01x")


def calltf(self)

  addr_t = get_value_like(self.addr, self.fmt)
  addr_s = addr_t[1]
  cmd_t  = get_value_like(self.cmd,  self.fmt)
  cmd_s = cmd_t[1]

  # parse as two-valued logic, return error is Xs or Zs
  try:
    addr_l = long(addr_s, 2)
  except:
    return 0

  if cmd_s == "1":  # write
    data_s = get_value_like(self.data, self.fmt)
    self.mydata[addr_l] = data_s

  elif cmd_s == "0":  # read
    try:
      data_s = self.mydata[addr_l]
    except:
      return 0
    del = pack_s_vpi_time(vpiSimTime, 0, 0, 0.0)
    put_value(self.data, data_s, del, vpiTransportDelay)
    return 1

  else:
    print "Unknown command\n"
    return 0
```

2.1 Other Examples

APVM is distributed with a number of example applications that illustrate the basic capabilities of the module and how to interact with the VPI interface through Python. These are described very briefly below. These examples are provided with the Python code, a small Verilog top-level and a shell script that shows how to run the example with Icarus Verilog, Cver and NC-Verilog. Icarus and Cver are

available for free.

apvm_mem - a sparse memory package, slightly different than the one shown above.

apvm_delay - a delay element showing how to use callbacks chedule events.

apvm_sr - a "stimulus/response" package. A stimulus/response file lists event times, stimuli to apply to nets and responses to check. This illustrates complex interactions with the Verilog scheduler. An extension of this example could form the basis for a test generation package.

plusargs - how to parse and use plusargs.

shownets - show all of the nets in a given module.

checker - a task that looks for a sequence of integers over time. This example uses a Python "generator" function as a lightweight process so that the checker is properly reentrant.

tkserver - demonstration of using Python's Tkinter interface to Tk/Tcl to build a simple X window-based message facility for Verilog programs. This might be useful in a compute server farm to allow simulation runs to send messages to a user's workstation.

rusage – print the execution hostname and resource statistics (CPU, IO, Swap) in the log file.

These examples show that APVM can implement fairly complex behavior with not a lot of code. In fact, this is one of the main strenghts of Python. With its full-featured built-in datatypes, many applications can be written without needing to defind new classes. If you are new to VPI or APVM, studying these examples should give a good understanding of the basics of both.

3 Considerations in Implementing $apvm

The VPI interface is very powerful, but presents a number of challenges to implenting PLI applications. This section discusses a few of those challenges and describes how APVM handles them. For those not familiar with VPI, a good reference is [http://www.verilog.com/1364-2005_D3.pdf].

3.1 Memory Allocation and Instance Workareas

Most Verilog implementations separate compile, elaborate and run phases of Verilog execution. When writing a PLI application, the user is able to specify a compile-time function to be called for the application and a run-time (calltf) function to be called for the application. The user is severely limited as to the operations that can be performed at compile time because compilation may be implemented as a separate Unix process by some Verilog implementations. Thus, even implementing memory on behalf of the runtime application is not permitted.

One operation that is allowed is to request that the simulator execute a callback on behalf of the application. APVM registers a start-of-simulation callback that is guaranteed to run just before time 0 in the run phase of execution. It is in this callback that memory is allocated for each instance of an $apvm call. The requirement for a callback at start-of-simulation seems so basic that it is surprising that VPI does not provide this directly.

Each $apvm instance requires a separate workarea to maintain its instance-specific information. The most recent VPI specification describes two functions for this purpose: vpi_put_userdata and vpi_get_userdata. Most versions of Verilog now have this support, but some do not. For those versions of Verilog that do not yet have these functions, APVM implements a simple closed growable hashtable to map system function handles to the workareas. The option is selectable at compile time by setting a few #define variables.

3.2 Mapping VPI structures to Python Data Structures

The VPI interface defines a few C structures that must be manipulated by the user to perform operations in VPI. These are limited to just a few types and describe only simple structures. The s_vpi_time struct is a good example. It is shown here.

```
struct s_vpi_time {
  unsigned int type, hi, lo;
  double simtime;
};
```

When using the VPI interface in C, a programmer typically loads these structures with values to pass to VPI functions. To obtain values, the programmer passes a pointer to one of these structures, and VPI fills it with values that can then be examined by the programmer. This use pattern does not map well to Python.

Instead, we implemented opaque pointers to these structures and provide a "pack" function in Python that constructs a new s_vpi_time struct from a Python tuple and an "unpack" function that takes an opaque pointer and returns a tuple. From Python, an s_vpi_time object is an immutable object. This notion fits much better with Python's garbage collection.

The use pattern is pretty simple. When a Python VPI function requires an s_vpi_time type, the user creates one on the fly with a function that packs base types into a newly allocated struct.

```
t = pack_s_vpi_time(vpiSimTime, 10, 0, 0.0)
put_value(vpi_h, value, t, vpiTransportDelay)
```

Such structs are also used as templates for result containes. This other use of these structs is mapped into Python in a slightly curious way. The s_vpi_time struct is a good example. The first field is a type tag that may take on either the value vpiSimTime or vpiScaledRealTime. If it is vpiSimTime then the low and high fields are used to describe the 64-bit simulation time. If the type field is vpiScaledRealTime, then the "real" field describes time as a double.

To get the current time, in C the programmer sets the type field of an s_vpi_time struct and passes it to get_time.

```
s_vpi_time t;
t.type = vpiSimTime
t.low = 0;
t.high = 0;
t.real = 0.0;
vpi_get_time(vpiHandle, &t)
/* use the return value somehow */
newtime = t.low + ...
```

Rather than adopt this directly and allow the VPI interface to modify allocated structures, in Python we chose to do something a little different. The user first creates a "template" s_vpi_time object and then passes it to a function that uses the template but returns a new struct as the value.

```
t1 = pack_s_vpi_time(vpiSimTime, 0, 0, 0.0)
t2 = get_time_like(vpiHandle, t1)
(typ, lo, hi, real) = unpack_s_vpi_time(t2)
```

The value returned is a *new* s_vpi_time object that can be "unpacked" to query the return value. This use pattern is sufficient to perform all required tasks, and is simple enough to learn quickly. Its drawback is a greater impact on the memory allocation and collection system.

Some VPI structs have union fields with variant field names for each of a number of fixed number of different types: integer, double, string. When packing such a struct, APVM infers the type field from the type of the Python argument. When unpacking the struct, APVM uses the type tag to infer the variant field to read. This mechanism is very simple but has been sufficient in practice.

3.3 Callbacks in Python

The implementation of callbacks in APVM deserves a discussion. APVM provides a system-function instance-specific callback, while VPI associates a callback with a particular C function, but not a Verilog system-function instance.

VPI does however, allow a user to attach an identifying "char*" to each callback, and it is this mechanism that we chose to use for our callback dispatch mechanism. The VPI standard is not exactly clear whether the payload can be an arbitrary pointer or that it must be a string pointer. We chose to be very conservative and assume only that it is a string pointer and create unique strings to identify callback instances.

The APVM systf class defines two methods for arbitrary callbacks: method "callmeback" schedules a callback for the instance, and arranges for the "callback" method to be called.

```
class systf:
  def callmeback(self, reason, object, time, value, userdata):
    ...

  def callback(self, cbdata, userdata):
    ...
```

When a user calls the callmeback method, APVM constructs a unique string (like a gensym in Lisp) for the call, s. The string s is put in a dispatch table (represented as a Python dictionary) mapped to the calling instance object with copies of the callback arguments. APVM then calls the VPI vpi_register_callback function with a pointer to the static APVM callback function, and the userdata field set to the string, s. When the simulator calls the static APVM C callback function with string s, APVM looks up the calling instance and data from the dispatch table and calls the "callback" method.

This implementation provides an object-oriented callback mechanism on top of the VPI calls. It seems simple when explained here, but was not simple to arrive at. One source of confusion is that the VPI interface allows each callback request to specify an "obj" (object) field that the callback is associated with. In most simulators this field is relevant only if the callback is associated with a structure like a wire or register and not an arbitrary system function instance. The dispatch table implementation with uniques strings is an efficient and simple approach to the problem.

One drawback of this approach is that the size of the dispatch table could grow as the simulation proceeds, since it stores information for every callback performed by APVM. This occurs because APVM cannot automatically determine which callbacks occur only once and which may persist throughout the simulation. To address this problem, we require users to specify when a callback is "persistent", meaning that the callback may be executed many times. For calls that are not identified as "persistent", we remove their entry from the dispatch table after they are executed. This simple policy helps keep memory from growing unacceptably.

3.4 New Callback Facility in Version 0.11

APVM Version 0.11 provides the existing callmeback/callback methods for each systf instance, but also provides a more general mechanism that is not tied to a particular systf instance. This streamlined interface is modeled after more traditional callback facilities and is much more flexible in that an instance of *any* callable Python type may be be the target of a callback.

The APVM module function `schedule_cb` schedules a new VPI callback and returns a handle. An example of the use of `schedule_cb` follows.

```
def mycbfn(reason, object, time, value, userdata):
    print "I am being called back for reason: %d\n" % reason

....
notime = pack_s_vpi_time(vpiSimTime, 0, 0, 0.0)
noval = pack_s_vpi_value(vpiIntVal, 0)

h = schedule_cb(mycbfn, cbNextSimTime, vpih, notime, novalue, "")
```

In the example above, we defined a callback function named "mycbfn." All Python VPI callback functions must have the formal parameter list shown above. These arguments mirror those required by native C callback functions. When scheduling the callback with the APVM function `schedule_cb`, the user gives the VPI reason, the object (a VPI handle, if any, None if the C pointer should be NULL), a simulation time object, a VPI value object and any optional user data. When Verilog schedules the callback, it calls the user function with the arguments shown.

If the callback has not yet transpired, it can be cancelled as shown below. This function both removes the entry from the dispatch table, and calls the VPI remove_cb and free_object functions on the corresponding C-level callback object.

```
cancel_cb(h)
```

Note that any callable Python object may be used as the callback function. Bound method calls are especially useful in this role. A short example follows. The interesting part is highlighted in bold.

```
class myclass:
  def cb_example(self, reason, object, time, value, userdata):
    print "I am a bound method callback for: ", name

inst = myclass()
...

h = schedule_cb(inst.cb_example, cbAfterDelay, vpih, t2, value, "")
```

This callback mechanism subsumes and extends the previous mechanism based on the callmeback and callack methods. Any method of the same instance can be the target of a callback. The very short example below shows how a method of one class can schedule another method as the target of a callback. Following this design pattern ensures that any number of instances of a class can be created and the callbacks of each is restricted to the members of the same instance.

```
class myclass:
  def meth1(self):
    ...
```

```
    self.h = schedule_cb(self.meth2, .... )

  def meth2(self, reason, object, time, value, userdata):
    pass
```

4 Framework Facilities

APVM offers facilities beyond that of simple interaction with the VPI interface. This section discusses some of those features.

4.1 Configuration

The first argument to the $apvm call is the "name" of the instance. Users of APVM should ensure that each name in the system is unique. This name may be a string or a Verilog object (usually a register). If it is a string, then the string is the name of the call. If the argument is a Verilog object, then its pathname becomes the name of the call.

Using a Verilog object makes it possible to write Verilog modules containing $apvm calls in a way that each instance will have a unique name. Consider the following example.

```
module apvm_module(i, o);
  in  [0:0]  i;
  out [0:0]  o;
  reg [0:0]  o_drv;
  reg        id;
  assign o = o_drv;
  initial
    $apvm(id, "foo", "bar", i, o_drv);
endmodule
```

The path name of the id register will become the name of each $apvm instance.

The instance name is used for such mundane things as printing a welcome banner and occasional debugging messages. However, the unique name becomes important for APVM applications that require (or provide) configuration.

APVM uses Python's native ConfigParser class to look for configuration files in standard places (current directory, home directory, system install directory). Python configuration files have sections with named parameters and resemble ".ini" files that are found on some systems. APVM adopts the convention that an instance looks for its configuration parameters in the section with its instance name. Consider the example below for an instance that was given the unique name "top.config_example.id." The application can look up its configuration parameter "myval" using a standard method.

```
file.py
```

```
from apvm import *
def config_example(systf):

  def soscb():
     self.myval = self.get_config("myval")
```

apvm.cfg

```
[top.config_example.id]
myval:   45
```

APVM also allows instance parameters to be specified on the Verilog command line with plus-args. APVM adopts the convention that a plusarg of the form "name:param" is mapped to the instance with the given name. If using the example above, a user could override the configuration file default with a different value.

```
% verilog +top.config_example.id:myval=46 ...
```

This simple combination of unique instance names and structured configuration mechanisms provides a useful feature for writing reusable applications.

4.2 Bit Vectors

As a convenience, APVM provides a bit vector class and functions for getting the value of a Verilog variable as a bit vector and for setting the value of a Verilog variable. The bit vector class overrides standard Python operators (+, *, <<, >>, subscripting, etc) to make operations on bit vectors clear and convenient. Bit vectors can be initialized from integers, or four-valued strings in Verilog syntax.

```
x = BV("4'b01xz")
```

The example below shows an APVM application that computes a function on two Verilog input variables and then schedules the result value to be applied two time units in the future. The list "self.vobjs" is provided by APVM for all instances: it is the list of VPI handles to the objects in the $apvm call following the class name. Functions bv_from_vpi and bv_to_vpi get bit vectors from and send bit vectors to objects referenced by VPI handles.

bitvect.v

```
$apvm("bvex", "bitvect", "bitvect_example", in0, in1, out)
```

bitvect.py

```
from apvm import *
class bitvect_example(systf):

  def calltf(self:
    val0 = bv_from_vpi(self.vobjs[0])
    val1 = bv_from_vpi(self.vobjs[1])
    delay = 2.0

    retval = (val0 << 1) & (val1 >> 1)
    bv_to_vpi(self.vobjs[2], retval, delay)
```

The bit vector class represents its values as simple four-valued strings. Thus, computations on bit vectors are straightforward but not necessarily as efficient as they could be. The convenience of using the class may outweigh the performance consideration. In the future, we may consider trying to optimize this class.

4.3 Save/Restore (Persistence)

Writing a PLI application is tough enough. Writing one that plays well with the Verilog Save/Restore mechanism is even harder. APVM provides a nice solution to the problem largely because it is based on Python. The "shelve" module is a standard part of a Python distribution defining a dictionary-like object that is mapped to a file. Users can save any type of Python structure (most nested and even recursive structures are supported) in the dictionary and when the shelve object is written to a file, the Python objects are "pickled" (serialized) so that they can be saved.

The standard APVM systf class defines empty Save/Restart simulation event methods, and two methods to help save and restore data from the shelve object. APVM calls the method "save_app_data" when Verilog does a system save, and calls method "restore_app_data" upon a restart. These methods default to empty function bodies. Implementing persistence is as simple as what is shown in the example below.

```
class myclass(systf):

    def save_app_data(self):
      self.save_data(self.mydata)

    def restore_app_data(self):
      self.mydata = self.restore_data()
```

The methods "save_data" and "restore_data" simply put data in the shelve object and get it out using the systf instance name as the dictionary key. Other parts of APVM handle opening and closing the shelve object and reloading Python on Restart.

While many aspects of state can be saved this way, other things require more complicated restart operations. One example might be a PLI application that has registered callback events on certain signal changes. These callbacks would have to be re-registered upon restart.

The name of the shelve file defaults to "apvm.gdbm", but can be set with configuration file or plusargs using the key "apvm:shelve_file".

5 Conclusions

This first release of APVM presents a framework for writing PLI applications in Python. The object oriented features of Python help to simplify the call back model of VPI and "systf" instances unify issues such as instance data handling, configuration and checkpoint/restart. APVM automates much of the tedium of writing a new PLI application and allows the programmer to focus on the interesting parts.

Many aspects of the VPI interface are unchanged in APVM. Traversing the Verilog database requires the same use of tags and iterators one would use in C. Creating time and value structures also requires knowing the corresponding C struct layout. Higher levels of abstraction layered on top of APVM might help simplify some of these issues one day, but all of the functionality required for full-featured applications is present in APVM now.

The examples provided here and in the release illustrate a variety of tasks that can be accomplished relatively easily using APVM. These include generating tests and interacting with the scheduler,

traversing the database, implementing a checker and instrumenting Verilog with a Tk/Tcl interface.

6 Addenda

These are some miscellaneous notes regarding this project.

6.1 Development Environment

APVM has been developed using Python, Icarus Verilog, GPL Cver, and Tkinter on Redhat 9. APVM has also been demonstrated using NC-Verilog, VCS and Modelsim. The examples included with the distribution are written so as to work properly in the subset of the VPI interface provide by Icarus [http://icarus.com/eda/verilog]. Another promising free Verilog simulator is GPLCVER [http://www.pragmatic-c.com/gpl-cver]. As of this writing, APVM works with GPLCVER1.10i. Universal compatibility with all Verilog simulators is the eventual goal.

6.2 Documentation Tool

I have adopted OpenOffice.org as my main documentation tool. The program seems to have anything one might want, but it is complicated. My mastery of this program is minimal, so I have difficulty with even the simplest things – like properly numbering sections. Getting to know this tool has been a great learning experience.

6.3 You Can Help

If you find this project or tool interesting, you might be able to help make it better. One area that needs improvement is the configure/build process, which is largely manual right now. The correct use of the Python "distutils" package should help. Some installations also require setting Python environment variables, and this can be very difficult. Developing a better way to get all of this right would be an enormous step forward.

Table of Contents