Verification Learns a New Language:  
PyUVM – An IEEE 1800.2 Implementation

Ray Salemi

Mentor, a Siemens Business

[ray\_salemi@mentor.com](mailto:ray_salemi@mentor.com)

Tom Fizpatrick

Mentor, a Siemens Business

[tom\_fitzpatrick@mentor.com](mailto:tom_fitzpatrick@mentor.com)

1. Why Consider Python?

While the Universal Verification Methodology (UVM) continues to dominate the industry for both ASIC and FPGA verification projects, achieving greater than 50% usage in both industry segments, there remains a substantial portion of the verification community for whom UVM is not a viable option.

There are many reasons for this situation, starting with a lack of knowledge of – and resistance to learning – SystemVerilog, for either design or for verification. Especially in the military/aerospace (mil/aero) segments, the pervasive use of VHDL makes it difficult for a SystemVerilog-based solution, such as UVM, to achieve any significant market penetration.

It could be argued (and in fact is has been) that it would be in their best interests to “bite the bullet” and move to SystemVerilog in order to take advantage of the unique capabilities of the language for verification, particularly constrained-random data generation and functional coverage. While there have been some attempts to mimic these capabilities through open-source VHDL libraries, the best that has been achieved is to approximate the structured component-based approach of UVM to improve modularity and reuse for VHDL users, but any attempt at constrained-random coverage-driven verification has been rudimentary at best.

Of course, this kind of institutional inertia has always existed. This is the very reason that SystemVerilog was created as a strict superset of Verilog, to at least make it seem like it was not a new language, even though it introduced many powerful programming features, most notably Object-Oriented Programming. Clearly, the success of SystemVerilog and UVM has proven the utility of this approach. But what comes next?

Mil/Aero companies are currently on a college hiring spree. For all the success that SystemVerilog and VHDL have had in the industry, there are precious few colleges where they are taught, meaning that today’s new engineers often do not enter “real life” with a working knowledge of the major languages used by our industry. Instead, this growing cohort prefers newer languages like Python and others.

As we know, the sheer volume of legacy design code and “back-end” tool flows based on SystemVerilog and VHDL means that these language will continue to dominate the design side of the equation for the foreseeable future. But there is evidence that new languages for verification could be viable, given the proper tool support in simulation and emulation. Efforts such as CocoTB prove that even basic attempts to write verification environments in Python can attract interest. But for a Python-based solution to really be viable, it would need to provide all of the functionality from SystemVerilog and UVM.

1. What’s Different About Python?

To see the difference between Python and SystemVerilog and VHDL one has to indulge in a bit of language history. The engineers who created the first programming languages were augmenting assembly language programming, and thus could not get away from the notion that different data types took a different number of bits.

They devised language types to ensure that programs allotted the correct number of bytes for the given type (thus we have int and longint) or that several data could be sent as a block of bits (thus we have struct)

Compilers checked that programmers were transferring data between like types with varying degrees of strictness with the ALGOL-based languages such as Ada, and VHDL taking a hard line, and the CPL based languages such as C and SystemVerilog allowing flexibility between types.

There are two assumptions that go into all these languages:

* Programmers are, at core, transferring bits between variables that have been correctly sized.
* Programmers must ask the language’s permission before transferring data between variables, otherwise bits could get overwritten.

Python is different in two ways:

* Python programmers are, at core, transferring handles to objects. The handles are all the same size and so they always transfer properly.
* Programmers ask forgiveness instead of permission. Programmers can read any member from an object, and if the member doesn’t exist Python raises a runtime exception.

We will see below how asking forgiveness instead of permission makes it easier to write and maintain testbenches.

1. Parameters: The Bane of Programming

One can think of languages as either manipulating bits (C, Verilog) or manipulating objects (Simula, Python). However one can also imagine a bit-manipulating language that wants to manipulate objects. For example, inspired by Simula Bjarne Stroustrup created C with Classes which became C++.[[1]](#footnote-1)

The problem here was that the classes were stored as bits and the compiler needed to keep track of the size of all the data members in a class. This created problems of reuse when you had a class, say a FIFO, that could be used to store int or shortint or char. How do you write one set of code for all FIFOS when you don’t know the size of the data being stored? You create typing parameters that provide the size of the data in the FIFO.

SystemVerilog ran into the same problem when classes were introduced to Verilog. A class, such as a uvm\_tlm\_fifo needs a parameter to provide the type being run through the FIFO, and each parameterized class becomes a different type. This makes for convoluted class diagrams and lots of syntax errors.

As we’ll see, life is much easier in Python since all variables hold instances of objects. This, combined with asking forgiveness instead of permission makes it much easier to write testbench code in Python.

1. Class Instances Everywhere

Everything in Python is an instance of an object. Consider the number 5. The type() method returns the type of an object and so we can do this at a python command line:

>>> type(5)

<class 'int'>

The example above shows that the number 5 is an instance of the class int. Yet we can also see that int is also an object.

>>> type(int)

<class 'type'>

So we see that int is of class type. The type class is the default root class for all classes in a Python program. Though, we’ll see below that we can change this for our benefit.

1. Just Enough Python

In this section we’ll cover just enough Python to be able to talk about how IEEE 1800.2 was implemented in Python. One of the advantages of Python is that is comes with an enormous ecosystem of training classes, websites, and books that delve deeply into the language.[[2]](#footnote-2)

1. Defining Classes

The class statement defines a new class, but unlike SystemVerilog or C, Python executes the class statement rather than compiling it. When we execute the class statement it creates a new class object and stores it in the script’s list of classes for later use.

Here is a simple example:

class Animal():

... def \_\_init\_\_(self, name):

... self.name = name

...

... def say\_name(self):

... print(self.name)

...

... def make\_sound(self):

... print("generic sound")

...

aa = Animal(4433)

aa.say\_name()

4433

aa.make\_sound()

generic sound

The above example demonstrates common elements of class declaration. The first thing we notice is the infamous Python indenting. Python uses indenting instead of begin/end or {/} to signify blocks. Whether one likes this is largely personal taste, but there it is.

The def \_\_init\_\_(self, name): overrides the \_\_init\_\_ method and demonstrates the double underscore convention for methods that exist in all classes. The \_\_init\_\_ method does the initialization one usually does in new() in SystemVerilog. There are many such methods including \_\_str\_\_ and \_\_eq\_\_ that server the UVM roles of convert2string() and compare().

The \_\_init\_\_ above requires that we provide a name for the animal. You can also see that we’re not doing any type checking on the name. In the cold and bureaucratic world of this program the animal stored in aa received only a number.

1. The self Variable

When we declare a class in SystemVerilog we declare class variables that SystemVerilog implicitly references as in C:

class point;

byte unsigned x;

byte unsigned y;

function new(byte unsigned X, byte unsigned Y);

x = X

this.y = Y

endfunction

endclass

In the above code the x = X line does the same thing as the this.y = Y code. They both set the instance’s variable to the constructor argument.

Python does not use the implicit assignment.

class Point:

def \_\_init\_\_(self, X, Y):

self.x = X

self.y = Y

Unlike the implicit this in SystemVerilog, Python requires that we explicitly supply the self variable as the first variable in an instance method. The calling mechanism hides this from us so we see when we instantiate a point:

make\_my\_point **=** Point(**10**,**3**)

The above causes Python to create an instance of the Point class and call \_\_init\_\_(self, X, Y), passing the newly created object as self.

Methods that don’t have self as the first argument must be either class methods (which receive a first argument of cls) or static methods (which have no required first argument)

1. Inheritance

Classes can inherit attributes from other classes and override methods from the base class. For example:

class Lion(Animal):

... def make\_sound(self):

... print("Lion roar")

...

ll = Lion('Stanley')

ll.make\_sound()

Lion roar

ll.say\_name()

Stanley

We see here that we’ve overridden Animal to create a Lion. We’ve only overridden the make\_sound() method, so we inherited \_\_init\_\_ and say\_name().

When we call make\_sound() Python looks for the make\_sound() method in the Lion class, finds it, and executes it. When we call say\_name() Python does not find the method in Lion and so it searches Animal. Finding the method there, it executes it.

1. Multiple Inheritance

Unlike SystemVerilog, Python provides multiple inheritance. This made it much easier to implement UVM in Python than SystemVerilog since SystemVerilog required us to create classes that mimicked multiple inheritance behavior. There are no \_imp classes in pyuvm.

Given that we have Animal, Lion, and Tiger we can create a Liger:

... class Liger(Lion, Tiger):

... ...

...

... ll = Liger("Bitey")

... ll.say\_name()

... ll.make\_sound()

Bitey

Lion roar

The Liger inherits from both Lion and Tiger. The ... is Python’s way of defining a class that inherits all its methods.

You’ll notice that we’ve created the dreaded Diamond of Death in that Lion and Tiger both inherit from Animal and Liger inherits from Lion and Tiger. In a compiled language this is a problem since one can’t tell which make\_sound() method to call.

But Python determines this dynamically. As above it looks for make\_sound() in Liger and, not finding it, it searches the parent classes in the order they appear in the declaration. That’s why it finds the make\_sound() in Lion.  
We now have enough class definition information to examine pyuvm.

1. Exceptions are the Rule

Asking forgiveness instead of permission is a key Python design philosophy. Languages such as C and SystemVerilog take the opposite approach. They use typing to ensure that a programmer cannot accidentally mix types and overwrite bits. Even those languages use the forgiveness philosophy when issuing runtime errors such as trying to access an array with an index beyond its range.

Python comes with built-in exceptions [[3]](#footnote-3) that extend the BaseException base class. It throws exceptions when we try to execute an illegal action such as trying to pull a value out of an associative array that doesn’t exist:

my\_array = {}

my\_array['one'] = 1

my\_array['two'] = 2

my\_array[3] = 3

print(my\_array['three'])

Traceback (most recent call last):

File "<input>", line 1, in <module>

KeyError: 'three'

In the above example we created an associative array (called a dict in Pythonese) and stored two values in it. Notice that the keys here can be of any type. Our bug was using 3 instead of three.

The KeyError class extends LookupError which extends Exception. The error class tree becomes important when we want to catch exceptions.

Consider a case where we’re implementing a uvm\_pool. The IEEE 1800.2 specification says that the pool get() method will return the value at the key, and if they key doesn’t exist, then initialize the location at key from the nonexistent Table 7-1. ++where is that table?++ We’ll use the Python universal object for emptiness None (not to be confused with SystemVerilog null, which is a null pointer. None is an actual object named None) to do the initialization.

from pyuvm import \*

class uvm\_pool(uvm\_object):

def \_\_init\_\_(self):

self.pool = {}

def get(self, key ):

try:

return self.pool[key]

except KeyError

self.pool[key] = None

The try/except block says to try the operation, and if the operation throws an exception of type KeyError we recover and set the pool’s location to None.

If any other kind of exception were thrown the exception would go up the call stack. If nothing caught the exception with an except block, then it would print to the screen and terminate the program.

1. The Joy of Duck Typing

Throughout this paper, we’ve often pointed out that Python allows us to implement the UVM without the complications created by constant type-checking and the parameterization it engenders.

We can write the code this way because of the Pythonic philosophy of duck typing. Duck typing says that, given an object, one says “If it walks like a duck and quacks like a duck, then it is a duck.” So we see the following:

class Canary(Animal):

... def make\_sound(self):

... print("tweet")

...

my\_duck = Canary('Phil')

try:

... my\_duck.migrate()

... except AttributeError:

... print ("Hey. That's not a duck")

...

Hey. That's not a duck

Rather than declare my\_duck to be of type Duck we say that any object that has the method migrate() must be a Duck. Given a Canary object we tried to make it migrate and found out that it could not.

This is not to say that you have to blindly try any object handed to you. You can check an object’s type so as to handle an error in a meaningful way:

class Duck(Animal):

... def make\_sound(self):

... print('quack')

... def migrate(self):

... print('Gone south.')

...

assert(isinstance(my\_duck, Duck)), "You must provide a Duck."

Traceback (most recent call last):

File "<input>", line 1, in <module>

AssertionError: You must provide a Duck.

The assert statement checks a condition (isinstance() in this case) and raises the AssertionError exception if the checked condition is false.

1. Implementing UVM in Python

SystemVerilog is the boiled frog of languages. The frog didn’t notice the slow heating of the water as bandaid after bandaid, kludge after kludge, and syntax after syntax were piled on top of what was originally a simple RTL solution to create something that prompts dismay in anyone who looks into the pot without proper warning.

Given that, creating the object-oriented UVM on top of SystemVerilog was a heroic exercise in ingenuity. The developers cobbled together macros, static class members, parameterization, and a judicious combination of inheritance and composition to create a powerful object-oriented verification methodology.

The result was a clearly-defined specification IEEE 1800.2, that lays out the steps needed to create the UVM in any object-oriented language. While it is true that we can ignore some elements of the specification such as the \*\_imp classes in a language with multiple inheritance, overall the spec gives us an excellent roadmap.

In this section we’ll examine the way Python has made it easier to implement the UVM and how we’ve structured the pyuvm project.

1. The pyuvm Package

The pyuvm package allows users to import all the UVM classes into a Python script:

from pyuvm import \*

**import** pytlm

class tinyalu\_test(uvm\_test):

The repository organizes the project by the sections in the IEEE 1800.2 specification. So pyuvm.py starts like this:

**from** enum **import** Enum, auto

# Support Modules

**from** error\_classes **import** **\***

**from** utility\_classes **import** **\***

# Section 5

**from** s05\_base\_classes **import** **\***

# Section 6

**from** s06\_reporting\_classes **import** **\***

# Section 7

**from** s07\_recording\_classes **import** **\***

Unlike the SystemVerilog import statement which reads from a compiled library unit, the Python import executes the code in the imported file. For the most part these files contain class statements whose execution adds another class object to the collection available classes.

The work consists primarily of going through the specification and implementing what we see there:

class uvm\_object(utility\_classes.uvm\_void):

"""

5.3.1

"""

def \_\_init\_\_(self, name=''):

"""

Implements behavior in new()

5.3.2

"""

# Private

**assert** (**isinstance**(name, **str**)), f"**{name}** is not a string it is a {type(name)}"

**self**.set\_name(name)

**self**.\_\_logger **=** logging.getLogger(name)

def get\_name(self):

"""

5.3.4.2

"""

**assert** (**self**.\_\_name **!=** **None**), f"Internal error. {str(self)} has no name"

**return** **self**.\_\_name

def set\_name(self, name):

"""

5.3.4.1

"""

**assert** (**isinstance**(name, **str**)), f"Must set the name to a string"

**self**.\_\_name **=** name

Notice above that the code honors the type definitions in the specification by checking name’s type using an assertion.

Notice also that the \_\_name variable denotes a protected variable as we are accustomed to in SystemVerilog. Python implements the protected status by mangling the variable name, changing \_\_name to Point\_\_name. So one could still access the protected variable directly, but only a monster would do that.

1. Docstrings

The strings in the triple quotes """ right after the function definition are docstrings. They appear in IDEs when you hover over the function call, or in automatically generated documentation. It could be argued that they deserve more information than the IEEE 1800.2 section number.

1. Python Properties

A Python-familiar reader may take offense to the existence of aget\_name and set\_name as Python has done away with the need for these sorts of accessors. More Pythonic code would look like this:

@property

def name(self):

**return** **self**.\_\_name

@name.setter

def name(self, name):

**self**.\_\_name **=** name

The @property string is a decorator that wraps these function calls in code that allows us to do this:

my\_object.name **=** "Foo"

**print**(my\_object.name)

# which results in Foo being printed.

This is, of course, much cleaner than the accessor functions needed in the SystemVerilog UVM, and one could argue that these accessors should have been implemented in a more Pythonic way. But, the goal here is to make pyuvm easy to use for existing UVM programmers, and changing basic elements of the specification would defeat that goal.

1. Key base classes, SV vs Python

Much of the work of writing the UVM in Python is, as we saw above, writing simple functions that implement the specification. However there are some base classes which can take more advantage of Python’s capabilities. This section shows how Python can make it easier to both write and use the UVM.

1. The Factory

The SystemVerilog UVM’s implementation of the factory pattern is a heroic act of engineering akin to the Gilligan’s Island professor making a Geiger counter out of coconuts. Still it imposes some work on the programmer.

First there is the need to remember the \`uvm\_\*\_utils macros.

class my\_component extends uvm\_component;

`uvm\_component\_utils(my\_component)

And then there is the creation incantation that allows a component to be overridden:

my\_comp\_h = my\_component::type\_id::create("my\_comp\_h",this);

This requires section 8.2.2 in the 1800.2’s Factory classes section which specifies a proxy type for all descendants of uvm\_object:

typedef my\_component type\_id

In addition there is a uvm\_component\_registry proxy class and other factory enabling tools. Here is how a user creates a component in pyuvm:

class my\_component(uvm\_component):

...

pyuvm automatically adds any descendent of uvm\_void to the factory. We create a new object like this:

my\_comp\_h **=** my\_component.create("my\_comp\_h", **self**)

One can also sidestep the factory with a simple instantiation.

my\_comp\_h **=** my\_component("my\_comp\_h, self)

One can implement overrides using the uvm\_factory singleton.

factory **=** uvm\_factory()

factory.set\_type\_override\_by\_type(my\_component, overriding\_component)

The factory also implements all the instance-based overrides.

1. Implementing the Factory in Python

The Python factory implementation takes advantage of the fact that the class statement is executed and not compiled. This gives us an opportunity to control what it means to create a class object.

As we saw above, most types in Python are objects of type type.

type(int)

<class 'type'>

type(type)

<class 'type'>

We see that even the type class is an object of class type But it is possible to make classes that are of a different type. These are called metatypes. The uvm\_void class is such a type:

type(uvm\_void)

<class 'utility\_classes.FactoryMeta'>

We specify this in its declaration:

class uvm\_void(metaclass=FactoryMeta):

"""

5.2

In pyuvm, we're using uvm\_void() as a meteaclass so that all UVM classes can be stored in a factory.

"""

This code means that the uvm\_void class object and all class objects descended from it are of type FactoryMeta. FactoryMeta registers all these classes with the factory:

class FactoryMeta(type):

"""

This is the metaclass that causes all uvm\_void classes to register themselves

"""

def \_\_init\_\_(cls, name, bases, clsdict):

FactoryData().classes[**cls**.**\_\_name\_\_**] **=** **cls**

**super**().**\_\_init\_\_**(name, bases, clsdict)

The code above says that when you execute a class statement to create a class object that extends uvm\_void that class object runs the above initialization code as is done with any other object. Notice though that we have cls as the first variable rather than self. This is to remind us that we’re being passed a class object. (The name is otherwise meaningless.)

We store the class object in the FactoryDatasingleton’s associative array (dict in Python parlance) named classes.

The FactoryMeta class extends the type class, so we call super().\_\_init\_\_ to ensure that all the work needed to set up a type gets done.

Now when you define a class that extends uvm\_void pyuvm automatically registers it with the factory.

1. Singletons

The UVM uses the Singleton Pattern in many places. The Singleton pattern describe a class that has only one instantiated object used throughout the testbench. We implement singletons in SystemVerilog using a static get() method.

class my\_singleton;

static my\_singleton common\_handle = null

static function get();

if (common\_handle == null) then

common\_handle = new();

return common\_handle;

endfunction

Then we get the handle like this:

single\_h = my\_singleton::get()

The get() method either returns the previously created handle or creates a new one, stores it in the static common\_handle location and returns the newly created handle. Regardless new() only gets called once.

Of course, this is susceptible to this bug:

my\_singleton bad\_h;

bad\_h = new()

And now bad\_h is a rogue instance of what is supposed to be a singleton.

Python allows you to avoid this by combining the get and new functionality in a single call. The Python code above looks like this:

single\_h **=** my\_singleton()

We cannot create the bad\_h

bad\_h **=** my\_singleton() # not so bad after all

1. Implementing the Singleton in Python

There are many ways to implement the Singleton patterning Python, but the pyuvm uses the metaclass approach as was done with the factory:

class Singleton(type):

\_instances **=** {}

def \_\_call\_\_(cls, \*args, \*\*kwargs):

**if** **cls** **not** **in** **cls**.\_instances:

**cls**.\_instances[**cls**] **=** **super**(Singleton, **cls**).**\_\_call\_\_**(**\***args, **\*\***kwargs)

**return** **cls**.\_instances[**cls**]

The above code demonstrates the built-in \_\_call\_\_ method. \_\_call\_\_ gets called whenever you put parentheses after any object.

Of course not all objects have a \_\_call\_\_ method, or they use the method to raise an error. For example the number 5 is an object, what happens if we call it? Python raises a TypeError exception:

5()

<input>:1: SyntaxWarning: 'int' object is not callable; perhaps you missed a comma?

Traceback (most recent call last):

File "<input>", line 1, in <module>

TypeError: 'int' object is not callable

But, if our object is of type class, then the parenthesis cause Python to call \_\_call\_\_ and eventually \_\_call\_\_ calls \_\_new\_\_ and ultimately \_\_init\_\_.

In the Singleton metaclass, \_\_call\_\_ receives the class object in the cls variable, and it creates an instance of that object (using super to call the Singleton’s parent constructor in type) and it stores that instance in an associative array using the cls object as an index. Now future calls to the class return the stored pointer.  
We define a singleton like this:

class my\_singleton (metaclass=Singleton):

And so the two examples of calling my\_singleton() above deliver the same handle.

pyuvm uses the Singleton metaclass for uvm\_root(), uvm\_factory() and uvm\_pool() among others.

1. Ports and Exports Without the Imps

Like SystemVerilog, Python has a Queue object that implements communication between processes. And like SystemVerilog the UVM needs us to use the Queue to implement the ports and exports that allow us to connect arbitrary components.  
Unlike SystemVerilog, Python has multiple inheritance.

1. IMPS and Multiple Inheritance

SystemVerilog does not implement an important object-oriented function: multiple inheritance. Multiple inheritance lets me do the following:

class Human:

def say\_hello:

**print**("hello")

class Racer:

def run:

**print**("running")

class BostonMarathoner(Human, Racer):

...

You use it like this:

my\_runner = BostonMarathoner()

my\_runner.run()

In the above we have a class for Human and another class for Racer. Since someone running the Boston Marathon must be human we can create the BostonMarathoner class using multiple inheritance.

The SystemVerilog UVM had no such luxury. It had to do the following:

class human;

imp role;

function new(imp rr);

role = rr;

endfunction

function void say\_hello();

$display("hello");

endfunction

function do\_role();

role.do\_it();

endfunction

class racer\_imp;

function void do\_it();

$display("running");

endfunction

And you use it like this:

racer\_imp racer;

human my\_runner;

racer = new();

my\_runner = new(racer);

my\_runner.do\_role()

Since the SystemVerilog doesn’t have multiple inheritance it needs to solve the problem of functional flexibility using composition and thus the SystemVerilog UVM needed to create \*\_imp classes that implemented behaviors such as blocking put, nonblocking get, etc.

The entire port/export structure is much easier to implement in Python since we don’t need the \*\_imp classes.

1. Implementing Ports and Exports

The UVM implements TLM behavior using a variety of ports and exports. These are divided into all the permutations of operations and TLM interfaces.

The operations consist of the following: put, get, peek, transport, master, and slave.

The TLM interfaces are blocking and nonblocking.

This gives us 6 \* 3 = 18 combinations of operations and interfaces.

1. Implementing Ports

First we have the uvm\_port\_base that provides common functions to all ports:

class uvm\_port\_base(uvm\_component):

def \_\_init\_\_(self, name, parent):

**super**().**\_\_init\_\_**(name, parent)

**self**.connected\_to**=**{}

**self**.export **=** **None**

def connect(self, export):

**try**:

**self**.export**=**export

**self**.connected\_to[export.get\_full\_name()]**=**export

export.provided\_to[**self**.get\_full\_name()]**=self**

**except**:

**raise** **UVMTLMConnectionError**(f"Error connecting {self.get\_name()} using **{export}**")

def check\_export(self, export, check\_class):

**if** **not** **isinstance**(export, check\_class):

**raise** **UVMTLMConnectionError**(f"**{export}** must be an instance of**\n{check\_class}** not**\n**{type(export)}")

We see that all ports provide a connect method and also the ability to check the an offered export is the right type. pyuvm raises UVMTLMConnectionError exceptions if there is problem.

The connect method sets the self.export variable and populates the connected\_to and the export’s provided\_to associative arrays.

Now we implement a uvm\_blocking\_put\_port:

class uvm\_blocking\_put\_port(uvm\_port\_base):

def connect(self, export):

**self**.check\_export(export, uvm\_blocking\_put\_export)

**super**().connect(export)

def put(self, data):

"""

12.2.4.2.1

A blocking put that calls the export.put

:param data:

:return: None

"""

**try**:

**self**.export.put(data)

except AttributeError:

**raise** **UVMTLMConnectionError**(f"Missing or wrong export in {self.get\_full\_name()}. Did you connect it?")

The uvm\_blocking\_put\_port overrides the connect method because this class knows which type of export it wants. It checks export against its needs for uvm\_blockinig\_put\_export and then calls super().connect(export) to make the connection.

The uvm\_blocking\_put\_port also provides the put() method and implements it using the export.put() method. Notice that here we ask permission rather than forgiveness. We assume that we have the right export and raise UVMTLMConnectionError if export does not have a put() method.

Implementing uvm\_nonblocking\_put\_port is similar:

class uvm\_nonblocking\_put\_port(uvm\_port\_base):

def connect(self, export):

**self**.check\_export(export, uvm\_nonblocking\_put\_export)

**super**().connect(export)

def try\_put(self, data):

# snipped

def can\_put(self):

# snipped

Now we can implement uvm\_put\_port using multiple inheritance giving us a one-line definition because a uvm\_put\_port is both a uvm\_blocking\_put\_port and uvm\_nonblocking\_put\_port:

class uvm\_put\_port(uvm\_blocking\_put\_port, uvm\_nonblocking\_put\_port): ...

1. Implementing Exports

Exports are classes that promise to implement the put, get, try\_put, etc functionality that the port expects. As we saw above the ports will assume that the exports have the right function calls.

This means that its easy to define exports:

class uvm\_export\_base(uvm\_component):

def \_\_init\_\_(self, name="", parent = None):

**super**().**\_\_init\_\_**(name, parent)

**self**.provided\_to **=** {}

# put

class uvm\_blocking\_put\_export(uvm\_export\_base): ...

class uvm\_nonblocking\_put\_export(uvm\_export\_base): ...

class uvm\_put\_export(uvm\_nonblocking\_put\_export, uvm\_blocking\_put\_export): ...

These classes have no methods as we use them only for the error checking as arguments to the check\_export() method in uvm\_port\_base.

This allows a wide variety of classes to support the TLM interfaces of their choice. For example, the uvm\_tlm\_fifo.

Implementing the FIFO

The uvm\_tlm\_fifo uses the Python Queue class to coordinate TLM communication between threads (each run\_phaseruns in its own thread.) This means that the exports in a uvm\_tlm\_fifo need to have a handle to the FIFO’s Queue. We implement this with the QueueAccessor class:

class QueueAccessor():

def \_\_init\_\_(self, name, parent, queue, ap):

**super**(QueueAccessor, **self**).**\_\_init\_\_**(name, parent)

**assert**(**isinstance**(queue, Queue)), "Tried to pass a non-Queue to export construtor"

**self**.queue **=** queue

**self**.ap **=** ap

The QueueAccessor assumes that it will be extended along with another class that needs the name and parent variables. It’s \_\_init\_\_ method has four arguments: name, parent, queue, and ap (the analysis port). It uses the first two arguments to create the uvm\_component that is an export, and it uses the second two arguments to store the queue and ap handles.

Now that we have the QueueAccessor we can implement the export in the uvm\_tlm\_fifo. For example BlockingPutExport:

class BlockingPutExport(QueueAccessor, uvm\_blocking\_put\_export):

def put(self,item):

**self**.queue.put(item)

**self**.ap.write(item)

And then we instantiate this class in the \_\_init\_\_()method to create blocking\_put\_export:

**self**.blocking\_put\_export**=self**.BlockingPutExport("blocking\_put\_export", **self**, **self**.queue, **self**.put\_ap)

Now we can connect this blocking\_put\_export to any blocking\_put\_port and pass the check\_export() method while also having Queue access.

1. Executing Phases

Python completely breaks with one of the basic assumptions behind SystemVerilog, the notion that classes cannot be modified at run time. In SystemVerilog the compilation step locks classes in place and syntax errors control whether one successfully makes a function call.

In Python everything is an object, including functions. A class can have a function object added to it at any time and thus gain new functionality over the course of a run.

Similarly one can use operations such as hasattr and getattr to inspect a class and get a handle to a function in it. We use this capability when executing phases.

The UVM defines a list of common phases that ship with the UVM and are expected to be supported in any uvm\_component. The list contains phases such as uvm\_build\_phase, uvm\_connect\_phase, and uvm\_run\_phase. We can see that all the common phases have the string uvm\_ at the beginning and if we strip that off we get the name of the phase function or task.

This gives the following pseudocode deep in the section 9: Phasing where we loop through a list of phase names and execute the associated function:

function exec\_func(comp, phase, state):

**for** phase\_name **in** ['uvm\_build', 'uvm\_connect', 'uvm\_extract']:

method**=**phase\_name[**4**:**0**] #remove uvm\_

**try**:

phase\_method **=** **getattr**(method)

phase\_method(phase)

except AttributeError:

**raise** **UVMError**(f"Missing component method **{method}**")

Once again we see ourselves asking forgiveness rather than permission. Given that we were passed a real component the phase methods should always exist, but if they don’t we’ll catch the error and raise an informative exception.

1. Using The Python UVM

If one takes as given that pyuvm works however it is implemented then one must ask how to use it. In this section we’ll see how to create a testbench with pyuvm.

We’ll use the TinyALU example from the UVM Primer. This is a simple ALU with ADD, AND, XOR, and MUL functions.

1. Defining the UVM Test

We can start at the top of the testbench defining a test to launch our test sequence:

class test\_top(uvm\_test):

def build\_phase(self,phase=None):

**self**.env **=** env("env",**self**)

def run\_phase(self,phase=None):

myseq **=** alu\_sequence("myseq")

**self**.env.agent.seqr.start(myseq)

This is, of course, a simplistic test. The hardcoded path to the sequencer in the agent limits reusability. However Python’s introspection capability would make it easy to write a function that finds all the available sequencers in an environment, thus enabling more reusability.

One advantage we see immediately is that an IDE such as PyCharm can help us write code more quickly:

<insert IDE image>

1. Defining The TinyALU Agent

Once we import pyuvm implementing an agent is remarkably similar to the same SystemVerilog code:

class tinyalu\_agent(uvm\_agent):

def build\_phase(self,phase):

**self**.cm\_h **=** command\_monitor("cm\_h",**self**)

**self**.dr\_h **=** tinyalu\_driver("dr\_h",**self**)

**self**.seqr **=** uvm\_sequencer("seqr", **self**)

# Fifos

**self**.cmd\_f **=** uvm\_tlm\_fifo("cmd\_f",**self**)

**self**.rslt\_f**=** uvm\_tlm\_fifo("rslt\_f",**self**)

# Make with the factory

**self**.rm\_h **=** result\_monitor.create("rm\_h",**self**)

**self**.sb **=** scoreboard.create("sb",**self**)

**self**.cmd\_mon\_ap **=** uvm\_analyis\_port("cmd\_mon\_ap", **self**)

**self**.result\_ap **=** uvm\_analyis\_port("result\_ap", **self**)

def connect\_phase(self, phase=None):

**self**.dr\_h.command\_port.connect(**self**.cmd\_f.get\_export)

**self**.rm\_h.ap.connect(**self**.cmd\_f.put\_export)

**self**.cm\_h.ap.connect(**self**.cmd\_mon\_ap)

**self**.rm\_h.ap.connect(**self**.result\_ap)

**self**.dr\_h.command\_port(**self**.cmd\_f.get\_export)

**self**.dr\_h.sequence\_item\_port(**self**.seqr.seq\_item\_export)

We now have an agent that provides score boarding, monitoring, and analysis ports for other parts of the testbench.

1. Transactions

The TinyALU transactions are similarly common to the SystemVerilog versions:

class command\_transaction(uvm\_sequence\_item):

def \_\_init\_\_(self, name, A, B, op):

**super**().**\_\_init\_\_**(name)

**self**.**A** **=** **A**

**self**.**B** **=** **B**

**self**.op **=** op

class result\_transaction(uvm\_transaction):

def \_\_init\_\_(self, name, r):

**super**().**\_\_init\_\_**()

**self**.result **=** r

1. The Dual-Top Testbench: The Proxy Approach

Accelerating a testbench on an emulator requires that we create a testbench with two parts. The HVL part (or Python part in this example) creates the stimulus, checks the results, and stores functional coverage. The HDL part contains the DUT and the synthesizable part of emulation-compatible VIP. (Cite [DVCon Europe 2015 Paper](https://dvcon-europe.org/sites/dvcon-europe.org/files/archive/2015/proceedings/DVCon_Europe_2015_P1_4_Paper.pdf))

Here we will see one mechanism for connecting a Python testbench to an HDL simulation using a suggested pytlm interface. The interface hides the ultimate connections to uvm\_connect that connect this testbench to a simulation or emulation of the HDL.

1. The TinyALU Driver

The pytlm uses a proxy object to connect the Python to a given BFM in the HDL side of the testbench. The proxy sends data to the HDL and blocks until the operation has finished. We use the in the tinyalu\_driver:

class tinyalu\_driver(uvm\_driver):

def build\_phase(self, phase = None):

**self**.bfm **=** pytlm.Proxy("xrtl\_top.tinyalu\_bfm")

**self**.command\_port **=** uvm\_get\_port("command\_port", **self**)

def run\_phase(self, phase = None):

**while** **True**:

command **=** **self**.command\_port.get()

**self**.bfm.send\_op(command.**A**, command.**B**, command.op)

**self**.bfm.wait\_for()

1. The TinyALU Monitors

Similarly the monitors use proxies to wait for their data. We have a command monitor and a result monitor. The agent uses the command monitor to write the command to an analysis port (for the scoreboard predictor to use) and the result monitor writes the result to an analysis port for the scoreboard to use for comparison:

class command\_monitor(uvm\_component):

def build\_phase(self, phase = None):

**self**.ap **=** uvm\_analyis\_port("ap")

**self**.monitor\_bfm **=** pytlm.MonitorProxy("xrtl\_top.tinyalu\_monitor", **self**)

def run\_phase(self, phase = None):

**while** **True**:

(**A**, **B**, op) **=** **self**.monitor\_bfm.wait\_for()

mon\_tr **=** command\_transaction(**A**, **B**, op)

**self**.ap.write(mon\_tr)

We can see above that the monitor uses Python’s ability to return arbitrary tuples from a function call. Rather than having to define a struct to return multiple values we can simply return them directly, in this case with (A, B, op) getting returned.

The result monitor similarly waits for its proxy to send it data:

class result\_monitor(uvm\_component):

def build\_phase(self,phase=None):

**self**.ap **=** uvm\_analyis\_port("ap")

**self**.bfm **=** pytlm.MonitorProxy("xrtl\_top.result\_monitor", **self**)

def run\_phase(self, phase):

**while** **True**:

result **=** **self**.bfm.wait\_for()

result\_t **=** result\_transaction(result)

**self**.ap.write(result\_t)

1. A TinyAlu Sequence

Having all the above in place allows us to create a uvm\_sequence:

class alu\_sequence(uvm\_sequence):

def body(self):

**A** **=** random.randint(**0**,**255**)

**B** **=** random.randint(**0**,**255**)

op **=** random.choice(ALUOps)

cmd\_tr **=** command\_transaction(**A**, **B**, op)

**self**.start\_item(cmd\_tr)

**self**.finish\_item(cmd\_tr)

We now have a working TinyALU testbench that is compatible with either a simulator or an emulator and that can leverage the entire Python ecosystem.

1. Conclusion

This approach literally provides the best of both worlds. Rather than reinventing the wheel, we build on all of the work that has gone in over the years to the development of the UVM, the most popular verification methodology in the industry, as well as existing constraint solvers and other capabilities provided by a simulator, but provide it to a new generation of engineers in a language with which they are already familiar.

1. [Wikipedia](https://en.wikipedia.org/wiki/C%2B%2B#History) [↑](#footnote-ref-1)
2. The >>> in the examples is the prompt from the Python interpreter. You see the interpreter when you type python on the command line. [↑](#footnote-ref-2)
3. <https://docs.python.org/3/library/exceptions.html#bltin-exceptions> [↑](#footnote-ref-3)