Verification Learns a New Language: – An IEEE 1800.2 Implementation

Ray Salemi
Siemens EDA
ray.salemi@siemens.com

Tom Fitzpatrick
Siemens EDA
tom.fitzpatrick@siemens.com

Abstract: UVM testbenches are powerful, reusable programs that generate transaction-level stimulus and analyze transaction-level results, leaving the signal-level control to bus functional models. One has to wonder why we are writing this complex software using a language intended to describe RTL in an event-driven simulator. Wouldn't we be better off using the most popular software development language[1], Python?

This paper introduces `pyuvm`, a Python implementation of IEEE Spec 1800.2. It discusses the Python tricks used to implement UVM features such as the factory, FIFOs, and `config_db`.

I. 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[2], 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. However, any attempt at constrained-random coverage-driven verification in VHDL 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.

[1] https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020
[2] Foster, H. (2020). *2020 Wilson Research Group Functional Verification Study: IC/ASIC Functional Verification Trend Report.* Siemens EDA.

As we know, the sheer volume of legacy design code and "back-end" tool flows based on SystemVerilog and VHDL means that these languages will continue to dominate the design side of the equation for the foreseeable future. However, 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.

II. 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.

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++.[3]

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.

[3] Wikipedia. (n.d.). C++. https://en.wikipedia.org/wiki/C%2B%2B

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.

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.

III. 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.[4]

A. 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:

[4] The >>> in the examples is the prompt from the Python interpreter. You see the interpreter when you type `python` on the command line. We will show Python output in *`italicized`* font.

```
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.

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)

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.

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()
Bitey
>>> ll.make_sound()
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`.

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 [5] 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 the key doesn't exist, then initialize the location at `key` from Table 6-7 in the SystemVerilog LRM (IEEE 1800-2017). 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.

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.

[5] https://docs.python.org/3/library/exceptions.html#bltin-exceptions

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.

IV. Implementing UVM in Python

It has been said that a camel is a horse designed by committee. Sadly, SystemVerilog is a similar beast. Band-Aid after Band-Aid, kludge after kludge, and syntax after syntax were piled on top of what was originally a simple RTL language to satisfy the languages many stakeholders, creating something that prompts dismay in anyone who tries to use it 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.

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 of 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`. One could still access the protected variable directly, but only a monster would do that.

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.

Python Properties

A Python-familiar reader may take offense to the existence of a `get_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)
Foo
```

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. Changing basic elements of the specification would defeat that goal.

V. 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.

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_component("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)
```

In addition to the UVM factory type overrides shown above, pyuvm also implements all the UVM factory instance-based overrides.

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'>
```

The `type` class is the default base class of all types and classes in Python. But we can create objects using base classes other than `type`. These are call *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 `FactoryData` singleton'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.

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
```

Implementing the Singleton in Python

There are many ways to implement the Singleton pattern in 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 parentheses 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.

Ports and Exports Without the Imps

Since 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. 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.

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.

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.provided_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 KeyError:
            raise UVMTLMConnectionError(\
                  f"Error connecting {self.get_name()} using {export}")

    @staticmethod
    def check_export(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 that 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_port)
        super().connect(export)


    def put(self, data, timeout=None):
        try:
            self.export.put(data)
        except AttributeError:

              raise UVMTLMConnectionError(f"Missing or wrong export in”
                         f”{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_port` and then calls `super().connect(export)` to make the connection. (The UVM allows a port to be connected to a port, and so connect checks for a port.)

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.

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):
...
```

Implementing Exports

Implementing uvm_nonblocking_put_export consists of nothing but extending uvm_blocking_put_port.: This means that its easy to define exports:

```
class uvm_blocking_put_export(uvm_blocking_put_port):
    ...

class uvm_nonblocking_put_export(uvm_port_base):
...

class uvm_put_export(uvm_nonblocking_put_port, uvm_blocking_put_port):
...
```

These empty classes aren't really necessary for the Python version of the UVM. All classes could be a ports with a final port class implementing the put(), get(), and other functions. But the TLM has a convention of naming. these classes exports and so we honor that naming convention here. When we implement the TLM methods we do it in an export class.

Implementing the FIFO

The uvm_tlm_fifo uses the Python Queue class to coordinate TLM communication between threads (each run_phase runs 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, UVMQueue)),
                "Tried to pass a non-UVMQueue to QueueAccessor constructor"
        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 turn itself into an export class (which is, in turn a uvm_component) and it uses the second two arguments to store the queue and ap handles.

One note about the super() keyword. Given a list of multiple parent classes, the argument tells super() to call the __init__ method in the class *after* the QueueAccessor class in the parent list. We are certain there is an interesting explanation for this strange behavior.

Now that we have defined the QueueAccessor we can implement the export in the uvm_tlm_fifo by defining a nested class, BlockingPutExport:

```
class BlockingPutExport(QueueAccessor, uvm_blocking_put_export):
    def put(self,item):
        self.queue.put(item)
        self.ap.write(item)
```

As stated above, the `super()` method calls the `__init__` after the `QueueAccessor`: uvm_blocking_put_export. The put method writes to `self.queue` and `self.ap`, following the expected UVM behavior for a put port.

We instantiate this class in the `uvm_tlm_fifo's __init__()`method to create the `blocking_put_export` that can later be passed to a `connect()` method.

```
class uvm_tlm_fifo(uvm_tlm_fifo_base):
def __init__(self, name, parent):
#snipped
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.

Extending the `Queue` Class

SystemVerilog is well acquainted with threads (which it calls *processes*) since every `always` or `initial` block acts as its own thread. Python also supports threads in the `threading` import library. The difference is that SystemVerilog has been comfortable with killing threads instantly ever since the `$finish()` system call appeared in Verilog-XL. When you finish a simulation all the threads in the testbench get surprised and killed.

Python doesn't allow such barbarous thread-killing. Instead, it requires that threads be allowed to put their affairs in order (closing files, for example) before they die. Python programmers must create a mechanism to tell threads to exit and let them do it on their terms.

This creates a problem for the blocking `put()` and `get()` methods in a `Queue` since a blocked thread has no way to respond to an exit request. Pyuvm solves this by extending `Queue` to create the `UVMQueue`. The `UVMQueue` overrides the blocking `put()` and `get()` methods to allow them to exit.

We do this by adding a timeout to the `put()` or `get()` method in a while loop. The while loop checks to see if we're at the end of the run phase, and if we are it exits:

```
def get(self, block=True, timeout=None):

   if not block or timeout is not None:
       try:
           return super().get(block, timeout)
       except queue.Empty:
           raise
   else: # create block that can die
       while not ObjectionHandler().run_phase_complete():
           try:
               datum = super().get(block=True,timeout=self.sleep_time)
               return datum
           except queue.Empty:
               pass
       exit() # Kill thread if it's time to die
```

The code first checks to make sure that you are doing a blocking get with no timeout. If you are then it creates a while loop that checks to see if the run phase has completed. If it has not then we call `get()` again, otherwise we exit.

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 implementation of the `uvm_phase.execute()` method, where we loop through a list of common phases, strip off the "`uvm_`" string to get the method name (`uvm_build_phase` becomes `build_phase`), use `getattr()` to find a handle of the method with that name, and finally, execute the method using the handle:

```
def execute(cls, comp):
    assert (issubclass(cls, common_phase)), "We only support phases whose
                                            "names start with uvm_"
    method_name = cls.__name__[4:] # strip off uvm_
    try:
        method = getattr(comp, method_name)
    except AttributeError:
        raise error_classes.UVMBadPhase(f"{comp.get_name()} is missing"
                                        f"{method_name} function")
    method() # call the phase 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.

Handling Objections to Completion

The `pyuvm` makes no attempt to implement the baroque phasing system of the SystemVerilog UVM. Instead it implements the set of functionality familiar to all users. This allowed for two simplifications:

1. The phase methods no longer take a `phase` argument as it is typically only used to raise and drop objections to ending the test.
2. The `uvm_component` now provides `raise_objection()` and `drop_objection()` convenience methods.

The `alu_test` class's `run_phase` now looks like this:

```
def run_phase(self):
    self.raise_objection() # Keeps the phase from advancing
#                            until the sequence is done.
    seq = alu_sequence("seq") # Here is the ten-item sequence
    seqr = self.config_db_get("SEQR") # The sequencer
#                                       stored itself in the config_db
    seq.start(seqr) # Start the sequence on the sequencer.
    time.sleep(1) # Give everything time to settle out.
    self.drop_objection() # Allow the testbench to go to the next phase
```

There is no longer a need to use the `phase` variable to raise and lower objections. The `ObjectionHandler()` singleton seen in the `UVMQueue` now creates a list of objecting objects and then deletes the objects as they drop their objections. When the list has a length of zero, the `run_phase` is over.

VI. 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*[6]. This is a simple ALU with ADD, AND, XOR, and MUL functions.

A. *Defining the UVM Test*

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

```
class alu_test(uvm_test):

    def build_phase(self):
        self.env = env("env", self)

    def run_phase(self):
        # shown above

    def final_phase(self):
        bfm = self.config_db_get("ALUDRIVERBFM")
        bfm.done.set() # Trigger the cocotb event to end the sim
```

Here you can see the pyuvm version of the `ConfigDB` convenience routines in action. They are now part of the `uvm_component`. Any component can raise or lower objections.

[6] Salemi, R. (2013). *The UVM primer: An introduction to the Universal Verification Methodology*. Boston, MA: Boston Light Press.

Defining The TinyALU Agent

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

```
class tinyalu_agent(uvm_agent):
    """ Provides the sequence and monitoring structure. """
    def build_phase(self):
        self.cm_h = command_monitor("cm_h",self) # Add the command monitor
        self.dr_h = driver("dr_h", self)         # the driver (note no parameter)
        self.seqr = uvm_sequencer("seqr", self)  # the sequencer (note no
        #                                          parameter here either)

        self.config_db_set(self.seqr, "SEQR", "*") # Sequencer to configDB

        # Factory Examples
        self.rm_h = self.create_component("result_monitor", "rm_h")
        self.sb_h = self.create_component("scoreboard", "sb_h")

        self.cmd_mon_ap = uvm_analysis_port("cmd_mon_ap", self)
        self.result_ap = uvm_analysis_port("result_ap", self)

    def connect_phase(self):
        self.cm_h.ap.connect(self.cmd_mon_ap)  # connect aports to aports
        self.rm_h.ap.connect(self.result_ap)

        # Connect driver to sequencer
        self.dr_h.seq_item_port.connect(self.seqr.seq_item_export)

        self.cm_h.ap.connect(self.sb_h.cmd_f.analysis_export # cmds to scoreboard
        self.rm_h.ap.connect(self.sb_h)  # Scoreboard is an ap.
```

We now have an agent that provides self-checking, monitoring, and analysis ports for other parts of the testbench.

Transactions

The TinyALU transactions are similarly common to the SystemVerilog versions. For example, here is the `command_transaction`:

```
class command_transaction(uvm_sequence_item):

    def __init__(self, name, A=0, B=0, op=ALUOps.ADD):
        super().__init__(name)
        self.A = A
        self.B = B
        self.op = ALUOps(op) #enums in Python

    def __str__(self):
        """The equivalent of the UVM convert2string()"""
        return f"A: {self.A} OP: {self.op} ({self.op.value}) B: {self.B}"
```

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[7].

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.

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 driver(uvm_driver):
    """
    A classic UVM driver.  It inherits a seq_item_port and uses it
    to get sequence items sent by the sequencer
    """
    def run_phase(self):
        self.bfm = self.config_db_get("ALUDRIVERBFM")

        while True:
            command = self.seq_item_port.get_next_item()
            self.bfm.send_op(command)
            self.seq_item_port.item_done()
```

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. Here is the `command_monitor` class:

```
class command_monitor(uvm_component):
        def build_phase(self, phase = None):
            self.ap = uvm_analysis_port("ap", self)
            self.monitor_bfm = self.config_db_get("ALUDRIVERBFM")


        def run_phase(self):
            while True:
                (A, B, op) = self.monitor_bfm.get_cmd()
                mon_tr = command_transaction("mon_tr", 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 the monitor bfm can simply return them directly, in this case with `(A, B, op)` getting returned. We use them to create a transaction and write the transaction to the analysis port. The result works similarly.

[7] van der Schoot, H. and Yehia, A., 2015. UVM and Emulation: How to Get Your Ultimate Testbench Acceleration Speed-up. In: DVCon Europe. [online] Available at: <https://dvcon-europe.org/sites/dvcon-europe.org/files/archive/2015/proceedings/DVCon_Europe_2015_P1_4_Paper.pdf>.

A TinyAlu Sequence

Having all the above in place allows us to create a `uvm_sequence`:

```
def body(self):
    cmd_tr = command_transaction("cmd_tr")
    for ii in range(10):
        self.start_item(cmd_tr) # UVM start_item gets the sequencer
        cmd_tr.A = random.randint(0,255) # use Python randomization
        cmd_tr.B = random.randint(0,255)
        cmd_tr.op = random.choice(list(ALUOps)) # Pick an operation
        self.finish_item(cmd_tr) # waits for the driver to call item_done
```

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.

VII. RUNNING THE PYUVM WITH COCOTB

The UVM code above has abstracted out the access to the DUT using an object it calls the `bfm`. The BFM could be implemented with DPI-C function calls, but in this example we use the coroutines in cocotb.

Coroutines are Python functions that run in the context of an event loop. They run until they await some other routine to complete. At that point they cede control of the processor and let other routines run. The open source cocotb project on GitHub is an open source interface between Python and supported simulators.

This paper looks at two elements of getting cocotb to work with pyuvm.

A. Creating a Driver BFM

The `RisingEdge` and `FallingEdge` triggers in cocotb allow us to create Python that acts a lot like an RTL BFM. We loop on the positive edge of the clock and run a little state machine to to response to commands.

```
async def driver_bfm(self):
   # snip initialization
    while True:
        await RisingEdge(self.dut.clk)
        if self.dut.start == 0 and self.dut.done == 0:
            try:
                cmd = self.driver_queue.get(timeout=0.1)
                self.dut.A = cmd.A
                self.dut.B = cmd.B
                self.dut.op = int(cmd.op.value)
                self.dut.start = 1
            except queue.Empty:
                pass
        elif self.dut.start == 1:
            if self.dut.done.value == 1:
                self.dut.start = 0
                self.dut.op = 0
```

Notice that the first line in the `try` block does a timed `get()` from a Queue. If there is a transaction in the Queue then it processes it, otherwise it loops around and waits for the next clock edge. This is similar to the way we code a BFM in RTL.

Creating a Monitor BFM

Similarly we create a monitor that fills an infinite Queue (what UVM calls an analysis FIFO) with results. We look for the synchronous rising edge of `done` and then get the result and send it to the result queue.:

```
async def result_mon_bfm(self):
    prev_done = 0
    while True:
        await FallingEdge(self.dut.clk)
        done = int(self.dut.done)
        if done == 1 and prev_done == 0:
            self.result_mon_queue.put_nowait(int(self.dut.result))
        prev_done = done
```

The queues allow the UVM threads to make blocking calls into the asynchronous coroutines without having to be part of the coroutine event loop.

Running the Test

Like all testbench frameworks, cocotb wants to own and run the tests. Therefore, we need to create a cocotb test and then launch the `uvm_root().run_test()` method in a thread. Then we wait on an event that gets triggered by the UVM test's `finish_phase()` method.

Here is the test

```
@cocotb.test()
async def test_alu(dut):
    ConfigDB().set(dut, "DUT", "*")
    clock = Clock(dut.clk, 2, units="us")
    cocotb.fork(clock.start())
    bfm = AluDriverBfm(dut, "ALUDRIVERBFM")
    await bfm.reset()
    cocotb.fork(bfm.start())
    await FallingEdge(dut.clk)
    test_thread = threading.Thread(target=run_uvm_test, args=("alu_test",),
                               name="run_test")
    test_thread.start()
    await bfm.done.wait()
```

The cocotb library launches this coroutine, forks off the clock, calls `reset`, and starts the bfms. Then it launches the pyuvm `run_test()` thread and awaits pyuvm telling it the test is complete.

VIII. 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.

The pyuvm is a Free Open Source Software project. As of the writing of this paper, Siemens is choosing an open-source Git platform for delivery. That repository will contain the current cocotb testbench as well as coding guidelines for the project.