Refresh your UVM Testbench with a Spritz of Python

Matthew Ballance

THE CHIPS TO SYSTEMS CONFERENCE

AMD

SPONSORED BY CEDA sigda

Testbench Languages

- Common test languages like SystemVerilog and PSS are domain-specific languages (DSLs)

- Designed to efficiently capture a specific problem domain
  - Provide language features tailored to the domain
    - Constrained randomization, assertions, system-level resource modeling
  - Library ecosystems focused around the problem domain
    - BFMs, protocol assertions, system-level traffic libraries

- General-purpose languages have different strengths
  - Designed to apply to wide set of algorithmic problems
  - Broad and extensive ecosystem of supporting libraries

- Using both together in a testbench is a win-win
  - Easily model domain specifics with tailored language features
  - Easily access general-purpose algorithms via library ecosystem

Benefits of Using Python in Testbenches

- Leverage rich ecosystem of libraries
  - Access data in specific file formats
  - Use reference algorithms and access to capabilities like AI
  - Explore new use cases and applications – analyze coverage data with AI

- Leverage domain expertise of engineers that don’t know our EDA DSLs
  - Firmware engineers may feel more comfortable writing in Python vs. SystemVerilog

- Leverage unique language characteristics
  - Python is an interpreted, dynamically-typed language
  - Slower execution than compiled languages, but enables much faster iterations

Integrating A General-Purpose Language

- Goal: Minimize integration cost
  - One-time integration effort
  - Repeated/incremental effort

- Goal: Maximize features
  - Manage object lifetimes across languages
  - Support interactions between threaded behavior

- Wrapper generators (e.g. SWIG) are insufficient
  - Don’t allow object-oriented Python to SystemVerilog calls
  - Don’t support threaded cross-language calls
  - Flatten object-oriented Python to a flat C API
  - Require per-application code generation and DPI library creation

Cost of Integration

- Add new generated code
- Add new DPI library
- Add new SV source
- Update Python Path

PyHDL-IF Library

- Python package that implements a SystemVerilog interface to Python
- Single application-independent DPI/VPI library
  - One-time integration vs. per-application/per-library integration
- Provides SystemVerilog APIs for calling Python from SystemVerilog
  - Reduces user code by abstracting above Python C API
- Provides Python and SystemVerilog APIs to call SystemVerilog from Python
  - Supports calling both functions and tasks
  - Adapts between Python and SystemVerilog threading models
- Code generator creates portable, reusable SystemVerilog interface classes
  - Resulting code can be checked in as primary source

https://github.com/fvutils/pyhdl-if

Calling Python from SystemVerilog

- Use Python as a dynamic language via the PyHDL-IF SystemVerilog class library
  - No need to generate any SystemVerilog interfaces
  - Lookup Python class and methods by name
  - Utility classes simplify calling and data conversion
- Example: accessing JSON data
  - Import ‘json’ package from Python
  - Read file data using the built-in Python methods
  - Parse and extract data using ‘json’ API
- Can create higher-level convenience APIs
  - Provide simpler higher-level API for targeted tasks
  - Hides implementation details

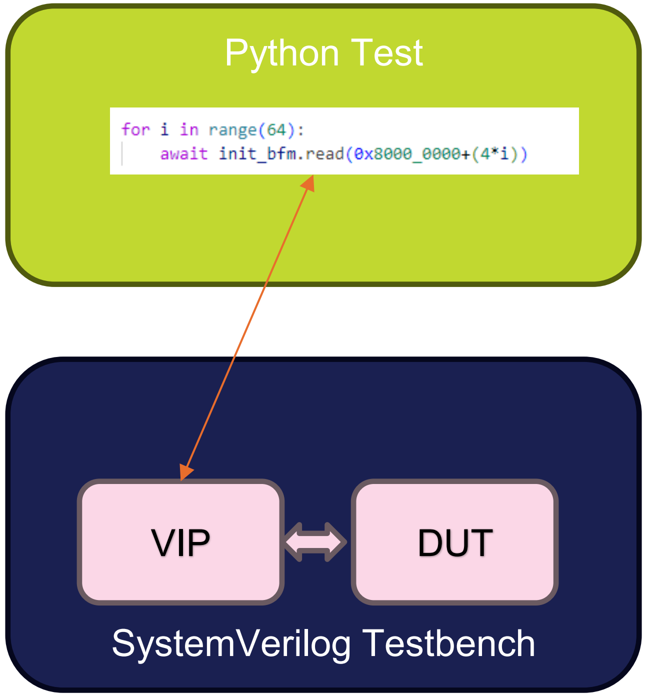
```
import pyhdl_if::*;

function void read_data(string datafile);
    automatic py_object json, data_fp, data_s;

    // Import Python's 'json' package
    json = py_import("json");

    // Open and read the specified data file
    data_fp = py_call_builtin("open", py_tuple::mk_init('{
        py_from_str(datafile),
        py_from_str("r")}));
    data_s = data_fp.call_attr("read");
    data_fp.call_attr("close");

    // Parse the data
    data = py_dict::mk(json.call_attr("loads", py_tuple::mk_init('{data_s})));

    // Get the list of keys
    keys = data.keys();

    // ...

endfunction
```

Calling SystemVerilog from Python

- Calling SystemVerilog from Python has different requirements
  - Must bridge between dynamic-typed Python and static-typed SystemVerilog
  - Must handle calls between threaded behaviors

- Example: Call SystemVerilog Verification IP from Python
  - VIP has a task-based SystemVerilog API
  - Want to call from a test implemented in Python

- PyHDL-IF bridges Python/SV concurrency differences
  - Connects SV processes and Python coroutines
  - Supports bi-directional calls that consume simulation time



Defining the Shared API

- Define the shared API in Python
  - Decorators mark call direction
    - imp: Python->SV
    - exp: SV->Python
  - Type annotations specify API types in SystemVerilog
    - Ensures a consistent SystemVerilog API
  - *async* Python methods connect to SystemVerilog *tasks*

- Code generator creates a reusable SystemVerilog class
  - Constructing a SV object creates a ‘peer’ Python object

- Implement ‘imp’ methods by extending the SystemVerilog class

```
import hdl_if as hif

@hif.api
class WishboneInitiator(object):

    @hif.imp
    async def write(self, addr : ct.c_uint32, data : ct.c_uint32):
        pass

    @hif.imp
    async def read(self, addr : ct.c_uint32) -> ct.c_uint32:
        pass
```

```
class WishboneInitiatorImpl extends WishboneInitiator;
    virtual task write(int unsigned addr, int unsigned data);
        bfm_write(addr, data);
    endtask

    virtual task read(
        output int unsigned retval, input int unsigned addr);
        bfm_read(retval, addr);
    endtask
endclass
```

Putting Everything Together

- Python test is implemented within a class
  - Async 'run' method
  - Calls SystemVerilog the VIP methods
- SystemVerilog testbench
  - Creates an instance of the Test class
    - Or, a Python class that inherits from *Test*
  - Obtains a handle to the VIP interface class
  - Passes the interface class to Python via the *run* method

```python
@hif.api
class Test(object):

    @hif.exp
    async def run(self, bfm : ct.py_object):
        for i in range(64):
            wr_val = (i+1)
            await bfm.write(0x8000_0000+(4*i), wr_val)
            rd = await bfm.read(0x8000_0000+(4*i))
            assert wr_val == rd
```

```systemverilog
task run_phase(uvm_phase phase);
    WishboneInitiatorImpl vip_if;
    Test test;

    // Obtain handle to the VIP interface
    // ...

    test = new();
    test.run(vip_if);

endtask
```

Python-Refreshed UVM Testbenches

- Python is complementary to existing UVM content
  - Enables access to a broad ecosystem of libraries
  - Shortens test-development iteration time
  - Leverage engineer's knowledge of Python
- PyHDL-IF library lowers barrier to entry
  - Removes the need to design a testbench-specific integration
  - Simplifies the task of calling Python with a SystemVerilog class library
  - Supports calling SystemVerilog tasks from Python
- PyHDL-IF library enables new possibilities
  - Development of pure-SV facades for using commonly-used Python libraries
  - Development of reusable APIs for Python to call SystemVerilog BFMs
- PyHDL-IF library provides a use-model pattern to follow
  - With other general-purpose languages like Typescript or Rust
  - With other EDA DSLs like PSS

https://github.com/fvutils/pyhdl-if

COPYRIGHT AND DISCLAIMER

AMD

62

THE CHIPS TO SYSTEMS CONFERENCE