2023
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
JAPAN

Pythonを用いたRTL検証

EE Tech Focus - 三橋明城男

accellera
SYSTEMS INITIATIVE

設計言語とテストベンチ言語の歴史

- RTLハードウェア設計言語
  - IEEE-1076 (VHDL) 初版1987年、IEEE-1364 (Verilog) 初版1995年
  - テストベンチ言語として設計されてはいなかった
- テストベンチ言語の出現
  - InSpec社（後のVerisity）による *e* 言語 - 1996年
  - Co-Design Automation社のSUPERLOG - 1999年
  - Accelleraは Vera、SUPERLOG、ForSpecを元にSystemVerilog標準化

SystemVerilogベースの検証メソドロジ

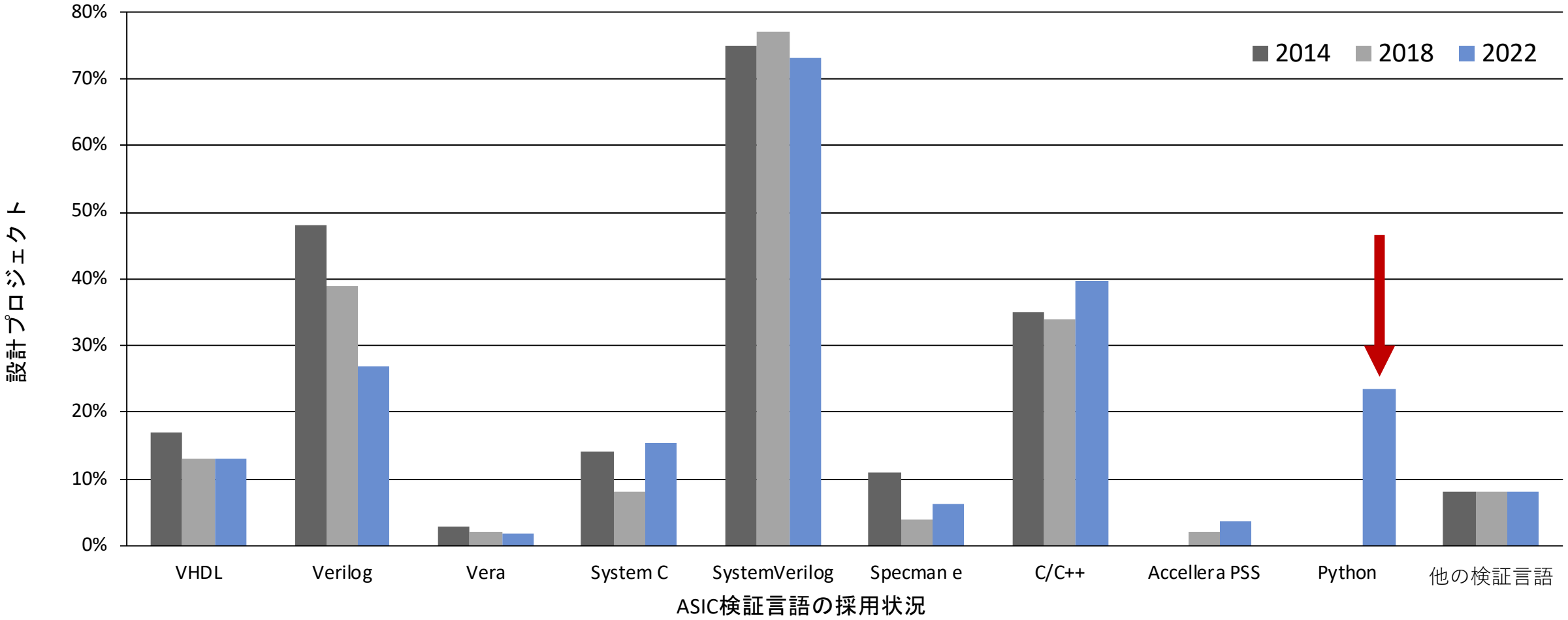
- 2005年～2006年は検証メソドロジ乱立時代
  - Cadence - Universal Verification Methodology
  - Synopsys - Verification Methodology Manual
  - Mentor Graphics - Advanced Verification Methodology
- 2007年 Cadence と Mentor Graphics は共同でOVM発表
  - 2010年、AccelleraはOVM 2.1.1をベースにUVM開発を発表
- 2011年、Accelleraが UVM 1.0をリリース
- 2017年、IEEE-1800.2としてUVMを標準化
  - IEEE 1800 : SystemVerilog
  - IEEE 1800.1 : SystemVerilog-AMS（予約）
  - IEEE 1800.2 : UVM

IEEE 1800.2ではUVMのAPIのみを標準化

- UVMのAPIとはユーザーが使用するAPI
  - AccelleraではIEEE 1800.2標準のAPIを SystemVerilogコードで実装し、“Reference Implementation” としてオープンソースで提供

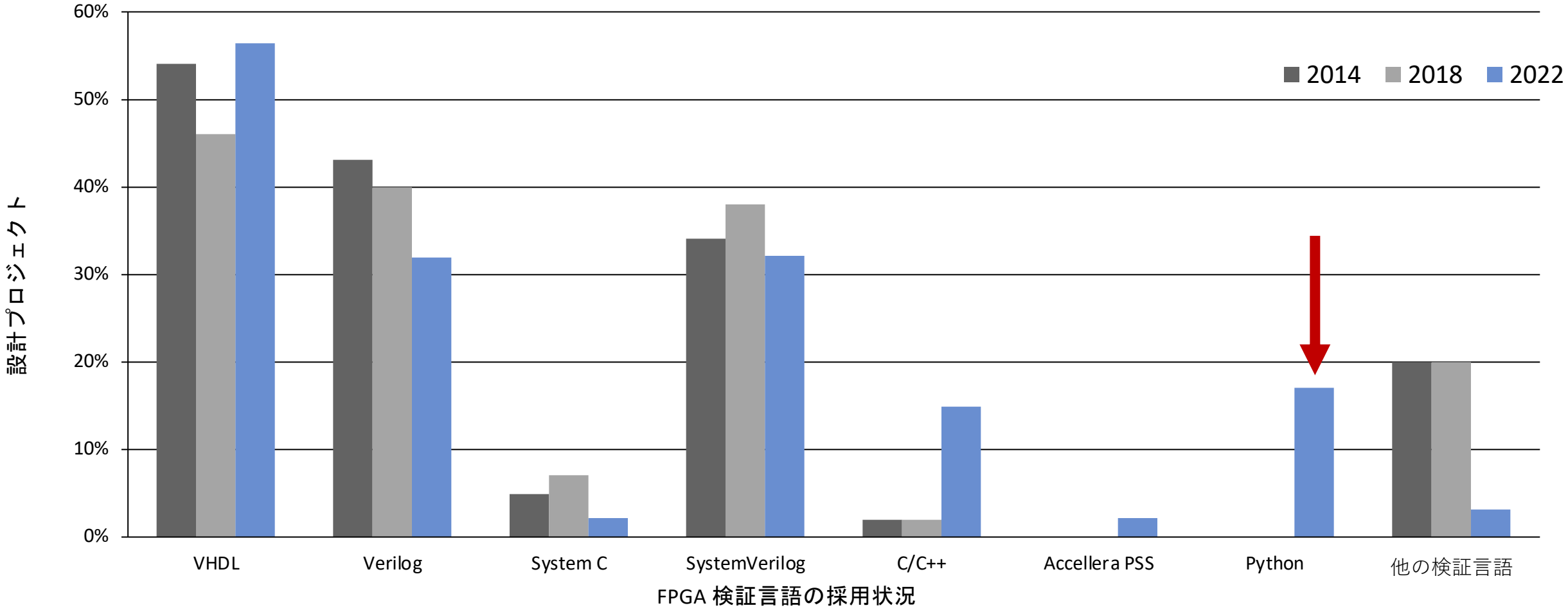
- SystemVerilog以外の言語で実装することも可能
  - AccelleraのSystemC WGでは、SystemCの実装を進めている
  - GitHub上ではPythonによる実装のプロジェクトがオープンソース形式で進んでいる (pyuvm)

ASICプロジェクトで使用される検証言語



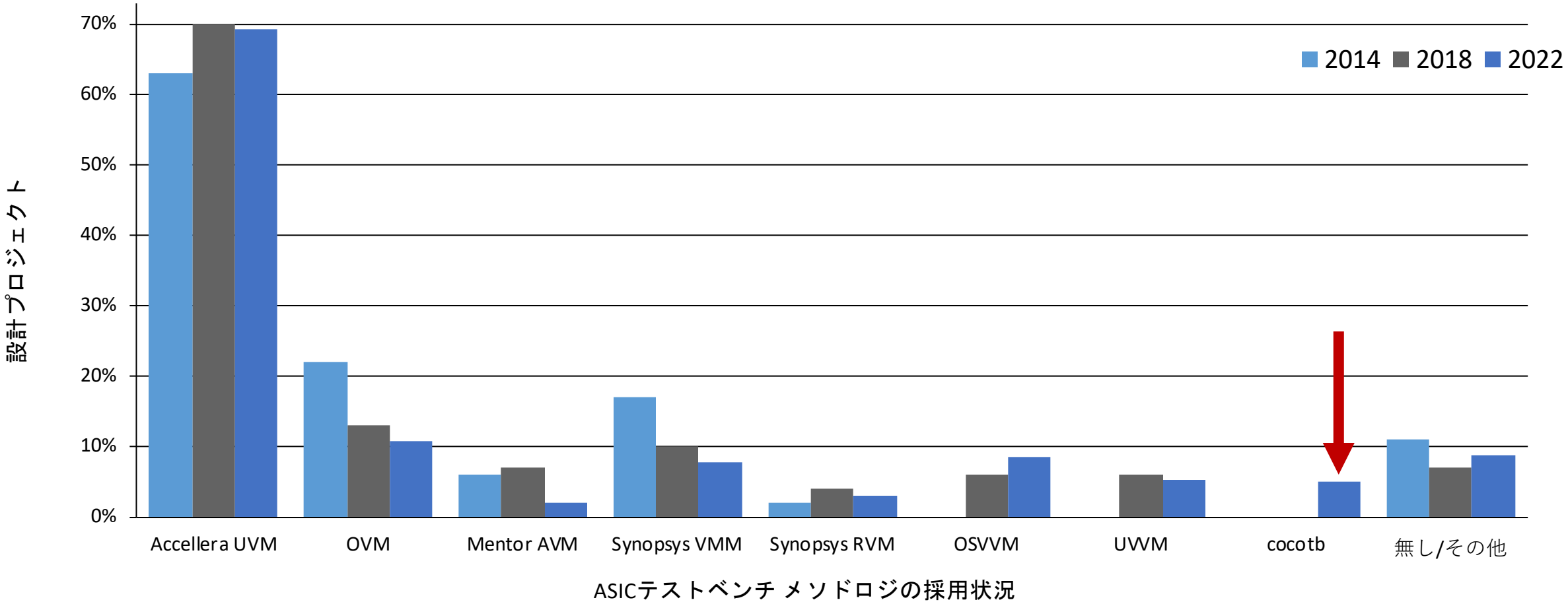
出典：Wilson Research Group “Functional Verification Study 2022”

FPGAプロジェクトで使用される検証言語



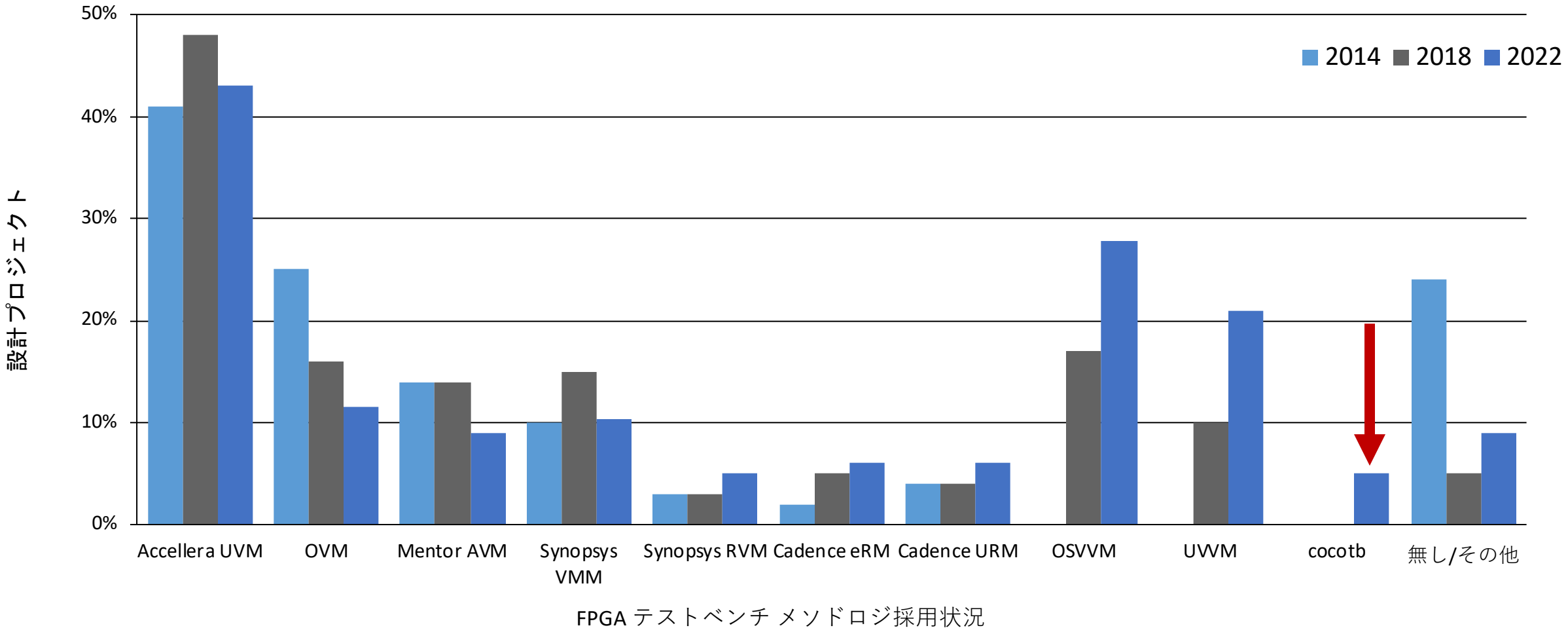
出典：Wilson Research Group “Functional Verification Study 2022”

ASICプロジェクトで使用される検証メソドロジ



出典：Wilson Research Group “Functional Verification Study 2022”

FPGAプロジェクトで使用される検証メソドロジ



出典：Wilson Research Group “Functional Verification Study 2022”

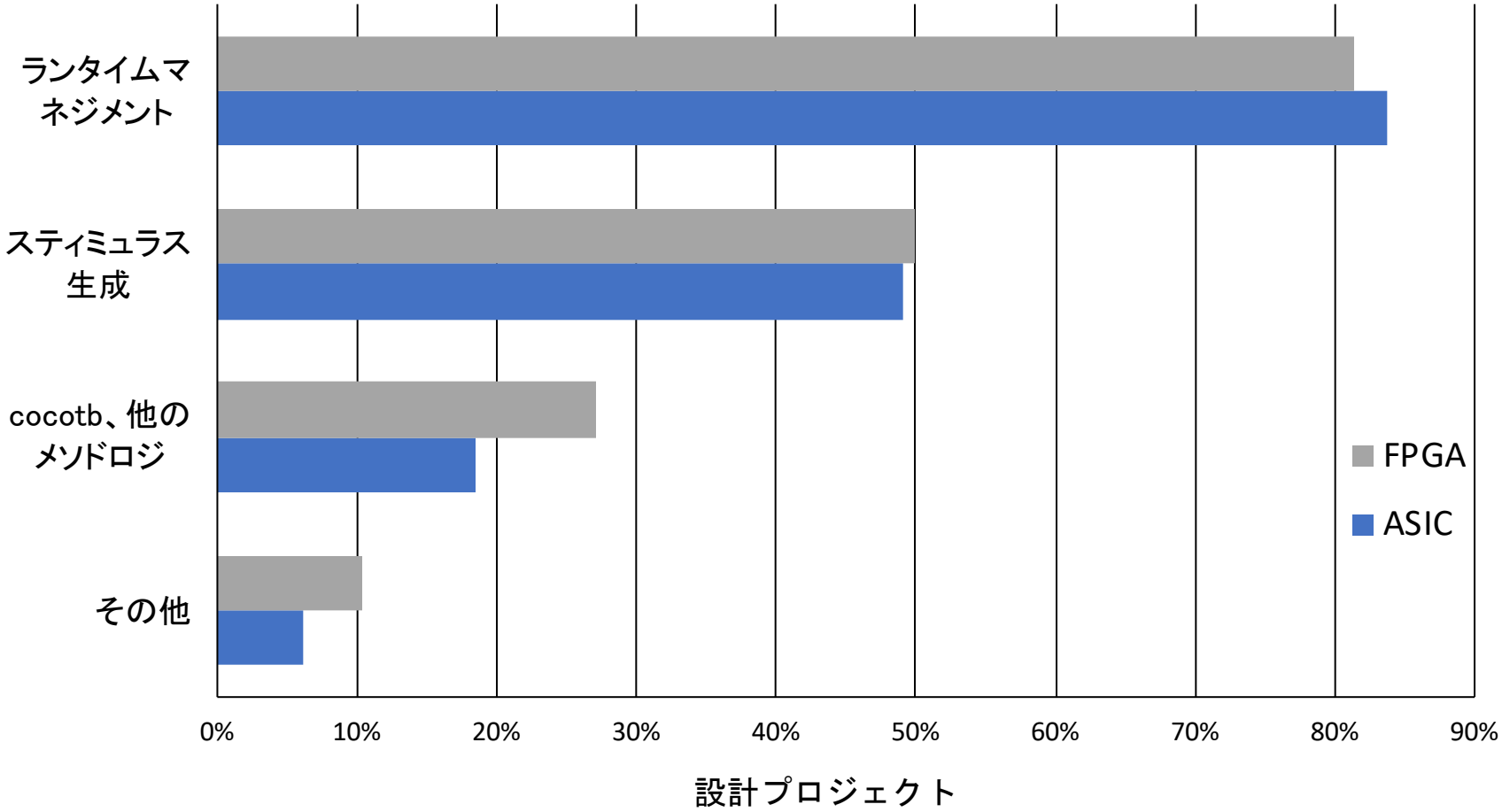
Pythonの使われ方

11.3%

全ASICプロジェクトでPythonを スティミュラス生成に使用する割合

8.5%

全FPGAプロジェクトでPythonを スティミュラス生成に使用する割合



出典：Wilson Research Group “Functional Verification Study 2022”

プログラミング言語 - Python

分類方法		
プログラミングパラダイム	オブジェクト指向言語	関数型言語
実行形式	インタープリタ言語	コンパイル言語
コード抽象度	高水準言語	低水準言語
変数の型定義	動的型付け言語	静的型付け言語

Python オブジェクト

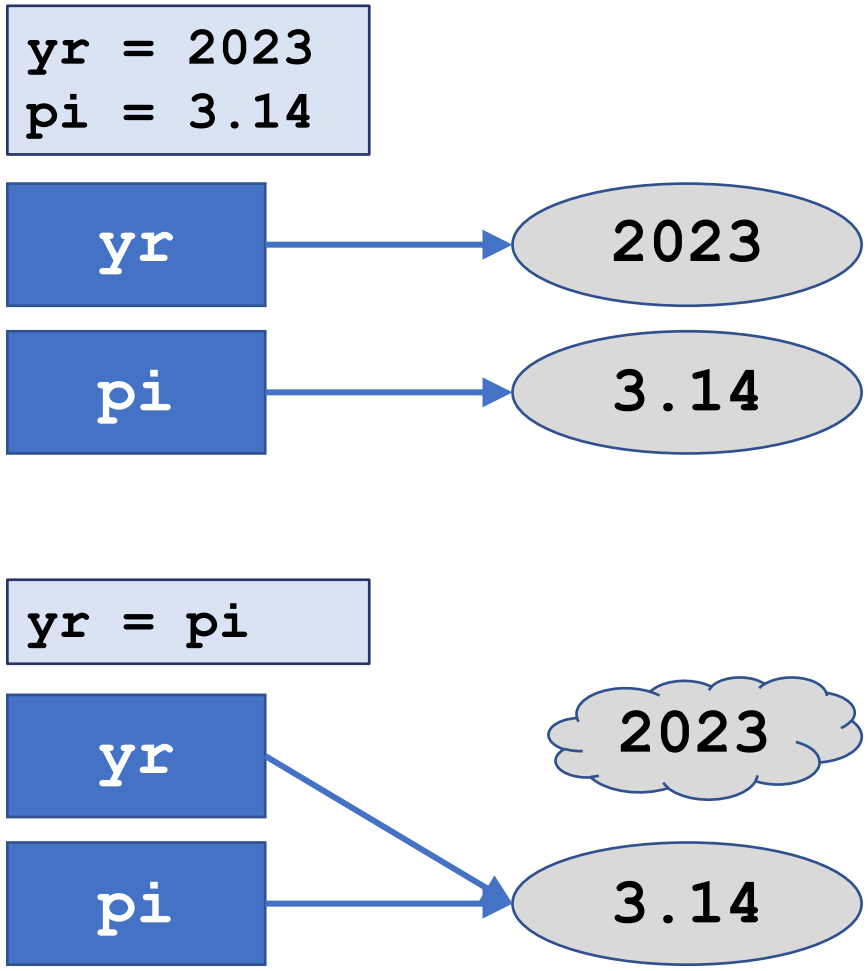
- Pythonで扱うものはすべてオブジェクト
  - オブジェクトはクラスをインスタンス化したもの

- C++などではタイプとクラスは分けられているがPythonでは同じ
  - type() ファンクションを使うとオブジェクトのクラスが分かる

```
print(type(2023))
--
<class 'int'>
```

Pythonではすべてがオブジェクトなので
タイプチェッキングやパラメタライズが不要

Pythonではデータでなくハンドルがコピーされる

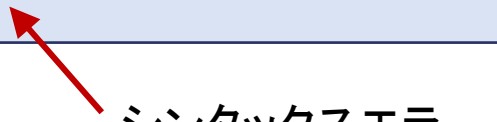



```
yr = 2023
pi = 3.14
print("Type of yr:", type(yr))
print("Type of pi:", type(pi))
yr = pi
print("yr is now", type(yr))
print("New type of yr", type(yr))

--

Type of yr: <class 'int'>
Type of pi: <class 'float'>
yr is now 3.14
New type of yr <class 'float'>
```

SystemVerilogと比べると・・・

SystemVerilog

```
uvm_put_port #(txn_a) ff;
ff = new("ff",this);
txn_b bb;
bb = new("bb")
ff.put(bb);
^^^^^^^^^^
```



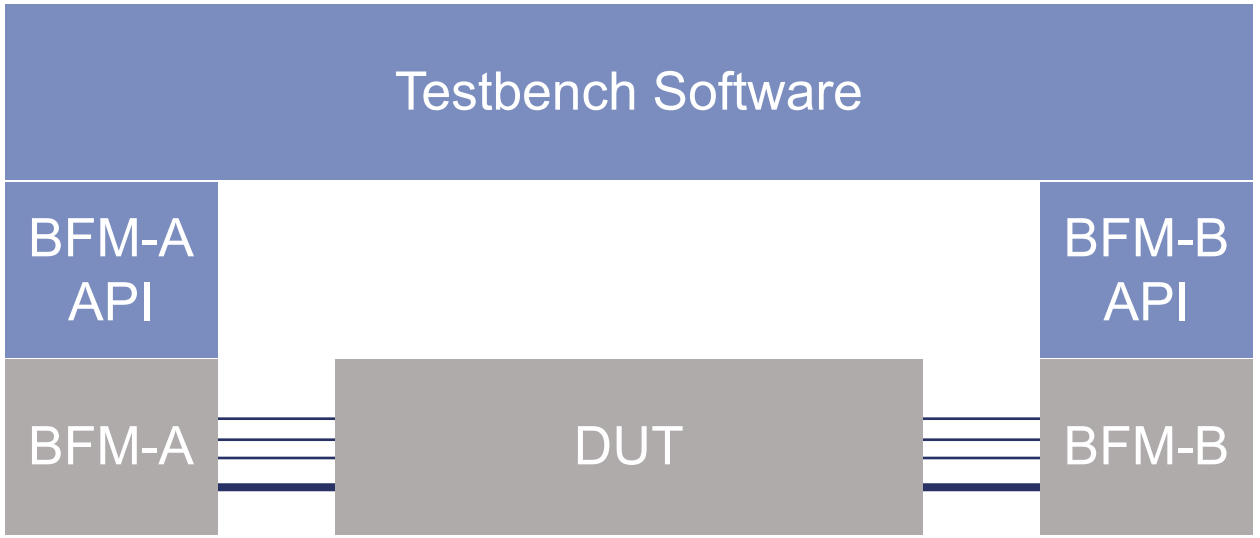
SystemVerilogでは、、、
タイプによるパラメタライズを
多用する傾向にある

Python

```
ff=uvm_put_port("ff",this);
bb = TxnB("bb")
ff.put(bb);
```

そもそもタイプ宣言がない

テストベンチ構成



テストベンチのソフトウェア
- スティミュラス生成
- 結果のチェック
- カバレッジのキャプチャ

ソフトウェアのファンクションコールを
ブロッキングするインタフェース

PythonではBFM APIは cocotb で実現

プロトコルコマンド⇔信号の変換

cocotb とは？

www.cocotb.org

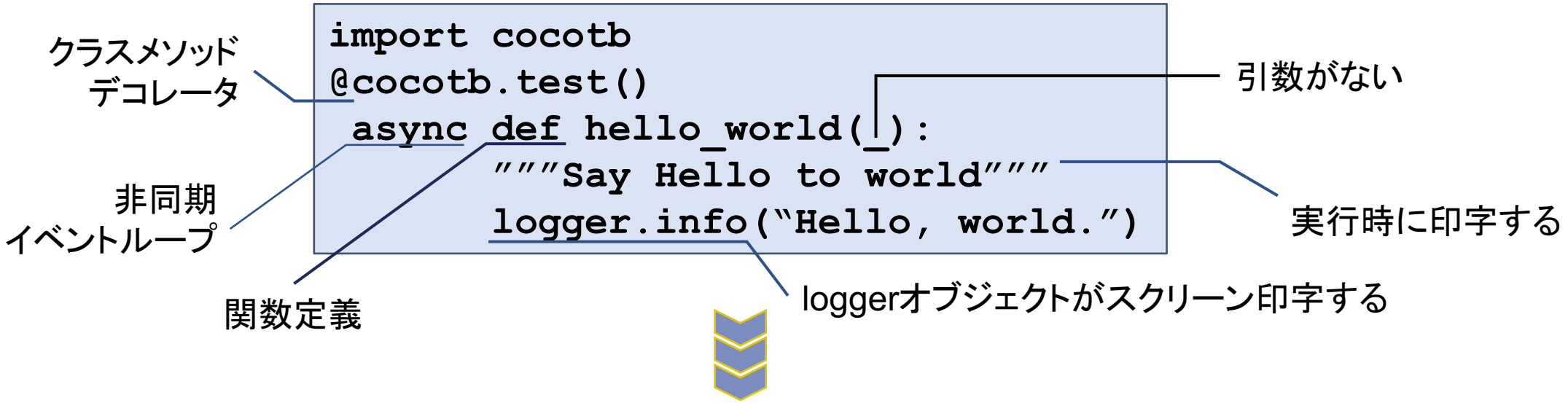
- **co**routine **co**simulation **t**est **b**ench
- coroutine
  - シュレーション用イベントドリブンの機能を提供するモジュールで、関数のように定義し呼び出して使用する
- cosimulation
  - RTLシミュレータからPythonを実行することで可能な協調シミュレーション
- testbench
  - Testbench Software ブロックを実装する Pythonプログラムを指す
  - Pythonのエコシステムで提供されている 44万以上のソフトウェアパッケージを活用してテストベンチ・ソフトウェアを実現できる

cocotb で行うこと

www.cocotb.org

- シミュレーションのイベント発生時に実行するcoroutineを定義する
- トップレベルのcoroutine - test を特定し、これによってシミュレーションを起動する

coroutineの定義と実行




```
import cocotb
@cocotb.test()
 async def hello_world(_):
        """Say Hello to world"""
        logger.info("Hello, world.")
```

```
0.0ns INFO     running hello_world
                 Say Hello to world
0.0ns INFO     Hello, world.
```

シミュレーション時間を待つ

- 時間を消費する関数
  - VHDLではprocess、SystemVerilogではtask、Pythonではcoroutine

VHDL

```
process is
begin
    wait for 2 ns;
    report “I’m done”;
    wait;
end process;
```

SystemVerilog

```
initial begin
    #2ns;
    $display( “I’m done”);
end
```

Python

```
import cocotb
from cocotb.triggers import Timer
@cocotb.test()
   async def wait_2ns(_):
     await Timer(2, units=“ns”)
     logger.info(“I’m done”)
```

Timer を使用したカウンタ定義例

```
import cocotb
from cocotb.triggers import Timer
import logging

   async def countup(name, delay, count):

     """Count up to the count in every delay"""

     for ii in range(1, count+1):
       await Timer(delay, units="ns")
       logger.info((f"{name} counts {ii}")
```

タスクの起動とタスク終了待ち

```
import cocotb
from cocotb.triggers import Timer, Combine
import logging

async def countup(name, delay, count);
   for ii in range(1, count + 1);
     await Timer(delay, units="ns")
     logger.info(f"{name} counts {ii}")

@cocotb.test()
   logger.info("START COUNTING")
   running_task_h = cocotb.start_soon(countup("DVConJP", 1, 3)
   await running_task_h
```

```
0.0ns INFO     START COUNTING
1.0ns INFO     DVConJP counts 1
2.0ns INFO     DVConJP counts 2
3.0ns INFO     DVConJP counts 3
```

- タスク起動は start_soon()で行う
- start_soon() が返すハンドルを await で指定して終了を待つ

実行タスクの無視

```
@cocotb.test()
async def do_not_wait(_):
    """Launch counter """
    logger.info("START COUNTING")
    cocotb.start_soon(countup("DVConJP", 1, 3)
    logger.info("ignored running_task")
```

```
INFO     START COUNTING
INFO     ignored running_task
```

- start_soon()でタスクは起動したが、ハンドルを使って await していない
- シミュレーションは起動の直後の終了する

- 無限に続く while ループなどがある状況で使用する（タスクはシミュレーション終了まで継続）

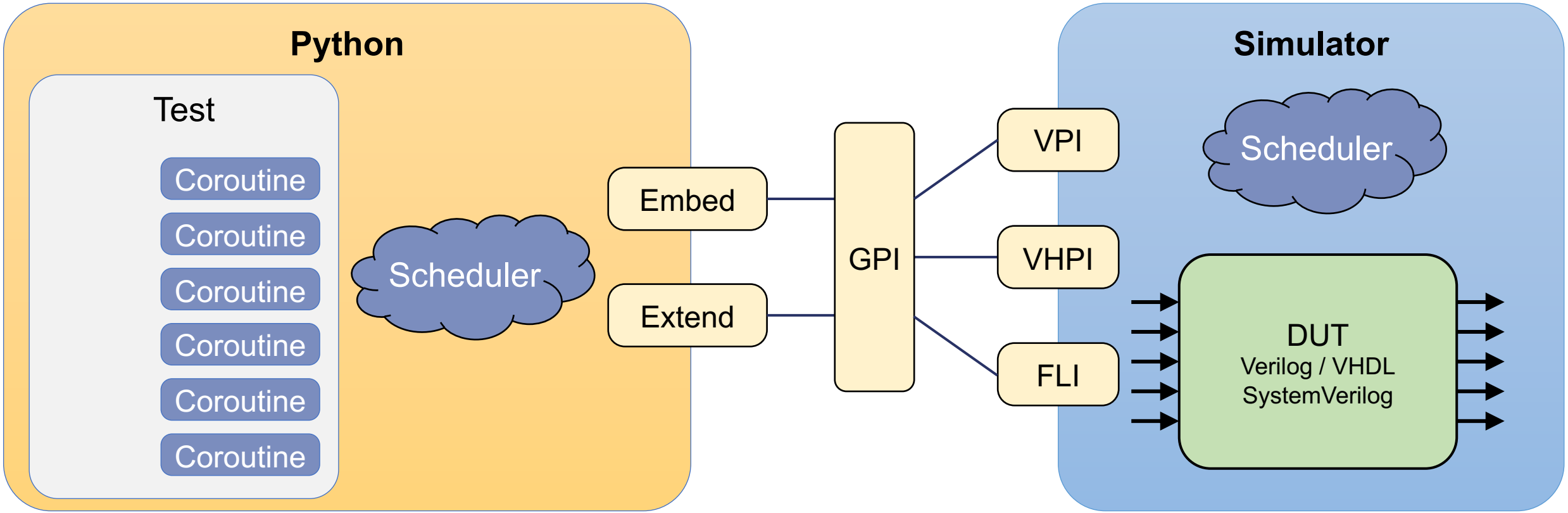
タスクの並列起動

```
@cocotb.test()
async def task_interleaver(_):
    logger.info("Count KEN to 5, PA to 3")
    KEN_h = cocotb.start_soon(countup("KEN", 1, 5)
    PA_h = cocotb.start_soon(countup("PA", 2, 3)
    await Combine(KEN_h, PA_h)
    logger.info("Task interleaver test done")
```

- await Combine() は両方のタスクが終わるのを待つ
  （SystemVerilog の fork~join に似ている）
- Combine 以外に First() もあり最初のタスク終了を待つ
  （SystemVerilog の fork~join_any に似ている）

```
2.00ns INFO    Count KEN to 5, PA to 3
3.00ns INFO    KEN
4.00ns INFO    PA
4.00ns INFO    KEN
5.00ns INFO    KEN
6.00ns INFO    PA
6.00ns INFO    KEN
7.00ns INFO    KEN
8.00ns INFO    PA
8.00ns INFO    Task interleaver test done
```

シミュレーションとの連携



シミュレーションとの信号の同期

- Trigger : シミュレータのタイミングに同期するための coroutine
  - Edge(signal)
    - 指定した信号の次のエッジ（立上り／立下り）を待つ
  - RisingEdge(signal)
    - 指定した信号の次の立上りエッジを待つ
  - FallingEdge(signal)
    - 指定した信号の次の立下りエッジを待つ
  - ClockCycles(signal, num_cycles, rising=True)
    - 指定した信号の num_cyclesで指定したサイクル数分の立上り／立下りエッジを待つ
    - 極性のデフォルトは立上り (rising=True)

counter とリセットテスト

```
//SystemVerilog counter.sv
`timescale 1ns/1ns
module counter(input bit clk,
               input bit reset_n,
               output byte unsigned count);
  always @(posedge clk)
       count <= reset_n ? count + 1 : `b0;
endmodule
```

- この記述以外にもリソースの import が必要

```
@cocotb.test()
async def reset(dut):
  cocotb.start_soon(Clock(dut.clk, 2, units="ns").start())
  dut.reset_n.value = 0
  await ClockCycles(dut.clk, 5)
  count = get_int(dut.count)
  logger.info(f"After 5 clocks count is {count}")
  assert count == 0
```

```
--
8.00ns INFO    After 5 cloks, count is 0
```

counter のカウントテスト

```
@cocotb.test()
async def count3(dut):
  cocotb.start_soon(Clock(dut.clk, 2, units="ns").start())
  dut.reset_n.value = 0
  await FallingEdge(dut.clk)
  dut.reset_n.value = 1
  await ClockCycles(dut.clk, 3, rising=False)
  count = get_int(dut.count)
  logger.info(f"After 3 clocks count is {count}")
  assert count == 3
```

```
--
16.00ns   INFO    After 3 cloks, count is 3
```

PythonベースのUVM

PythonベースのUVM - pyuvm

- IEEE 1800.2 の仕様
- 実装上のルール
  - 一般的に使われる機能を実装
  - ほぼ使われない機能は実装せず
  - Pythonの機能は再実装せず
    - copy.deepcopy() ⇔ do_copy()
    - __str__() ⇔ convert2string()

- インストールするには

```
% pip install pyuvm
```

- パッケージを使用するには

```
from pyuvm import *
```

```
class AluTest(uvm_test):
  def run_phase(self):
    self.raise_objection()
    seqr = ConfigDB().get(self, "", "SEQR")
    seq = AluSeq("seq")
    seq.start(seqr)
    time.sleep(1)
    self.drop_objection()
uvm_root().run_test("AluTest")
```

PythonによるUVM記述例

マクロは使用しない、自動的にFactoryにある

build_phaseの引数はなし

Factoryへのアクセスが簡単

```
class AluAgent(uvm_agent):
  def build_phase(self):
    super().build_phase()
      if self.active():
        self.driver = Driver.create("driver", self)
      try:
        self.is_monitor = ConfigDB().get(self, "", "is_monitor")
      except UVMConfigItemNotFound:
        self.is_monitor = True
      if self.is_monitor:
        self.cmd_mon = Monitor("cmd_mon", self, "get_cmd")
        self.result_mon = Monitor("result_mon", self, "get_result")
```

ConfigDBも単純化

例外処理の機能で記述の容易化

Introspectionによる柔軟なコーディングスタイル

実装されているセクション

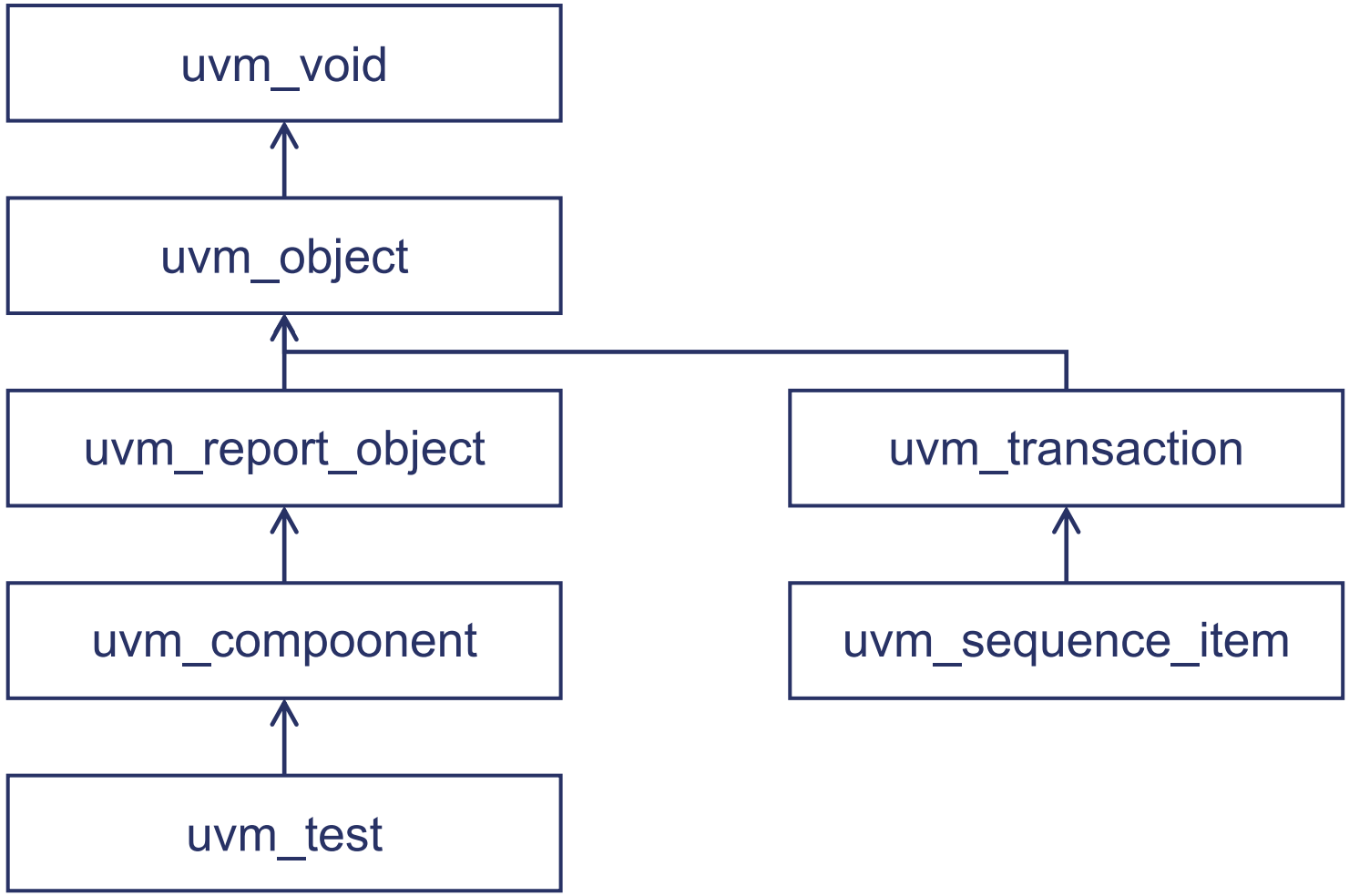
		Notes
5	Base Classes	uvm_object does not capture transaction timing information
6	Reporting Classes	Leverages logging, controlled using UVM hierarchy
8	Factory Classes	All uvm_void classes automatically registered
9	Phasing	Simplified to only common phases. Supports objection system
12	UVM TLM Interfaces	Fully implemented
13	Predefined Component Classes	Implements uvm_component with hierarchy, uvm_root singleton,run_test(),simplified ConfigDB, uvm_driver, etc
14 15	Sequences, sequencer, sequence_item	Refactored sequencer functionality leveraging Python language capabilities. Simpler and more direct implementation

実装されていないUVM機能

- カスタムのUVMフェージング
  - 既存UVMフェーズのみで充分
- uvm_resource_db
  - uvm_config_db で代用可能
- _imp classes
  - 不要な実装
- Report handler
  - Pythonのlogging で代用可能

- Synchronization class
- Container Class
- Recording class
- Transaction recording
- Policy class
- Register layer / model

pyuvm UML ダイアグラム



PythonでのFactoryパターン：metaclass

- class は実行されるステートメントで class タイプのオブジェクトを生成する
- cls というオブジェクトを辞書 - name インデックスの連想配列にストアする
- class を Factory登録するためのマクロ - ‘uvm_component_utils は不要
- Factory のSingletonメソッドが実装される

このコードは uvm_void クラスが定義される際に実行される
（クラスがインスタンス化される時ではない）

```
class FactoryMeta(type):
   def_init_(cls, name, bases, clsdict):
       FactoryData().classes[cls.__name__] = cls
       super().__init__(name, bases, clsdict)
```

```
class uvm_void(metaclass=FactoryMeta):
    . . .
```

```
class uvm_factory(metaclass=Singleton):
    . . .
```

```
class uvm_factory().set_type_override_by_type(Driver, NewDriver)
    . . .
```

Factoryの使い方が簡素化される

SystemVerilog UVM

```
function void build_phase(uvm_phase phase);
    driver_h     = driver::type_id::create("driver_h",this);
    coverage_h   = coverage::type_id::create("coverage_h",this);
    scoreboard_h = scoreboard::type_id::create("scoreboard_h",this);
```

Python UVM

```
def build_phase(self):
    self.driver     = Driver.create("driver", self)
    self.coverage   = Coverage.create("coverage", self)
    self.scoreboard = Scoreboard.create("scoreboard", self)
```

Python での ConfigDB

- Singleton 実装が簡素化され、タイプによるパラメタライズが不要
  - uvm_resource_db は使わず uvm_config_db インタフェースを実装

SystemVerilog UVM

```
config_db#(uvm_sequencer)::set(null, "*", "SEQR", seqr)
if(!uvm_config_db #(uvm_sequencer)::get(this, "", "SEQR", seqr))
  `uvm_fatal("Could not find sequencer")
```

Python UVM

```
ConfigDB().set(None, "*", "SEQR", self.seqr)
self.seqr = ConfigDB().get(self, "", "SEQR")
```

Python におけるスレッド

スレッドのKillは
- SystemVerilogでは何も警告なくKill

- Python ではスレッドが色々と片付けられる仕組みが必要

- ただしブロッキングのget() はExitしなくてはならない

```
class UVMQueue(queue.Queue):
#### In init ()
  self.end_while_predicate = ObjectionHandler().run_phase_complete
  self.sleep_time = 0.1
####
  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 self.end_while_predicate():
        try:
          datum = super().get(block=True,timeout=self.sleep_time)
          return datum
        except queue.Empty:
          pass
      sys.exit()    # Kill thread if it's time to die
```

UVMQueue を使ったTLM FIFO

```
class QueueAccessor:
    def __init__(self, name, parent, queue, ap):
        super(QueueAccessor, self).__init__(name, parent)
        self.queue = queue
        self.ap = ap
```

TLM通信するためにFIFOの
Queueへのハンドルを渡す

```
class NonBlockingGetExport(QueueAccessor,
                           uvm_nonblocking_get_export)
   def  can_get(self):
        return not self.queue.empty()

   def  try_get(self):
        try:
            item = self.queue.get_nowait()
            self.ap.write(item)
            return True, item   # return tuple
        except queue.Empty:
            return False, None  # return typle
```

```
self.nonblocking_get_export =
self.NonBlockingGetExport("nonblocking_get_export",
                          self, self.queue, self.put_ap)
```

uvm_tlm_fifo _init_

Phase をリファクタリング

- UVM Common Phaseを実装
  - Common Phase が何かはLRMで定義されている
- カスタムの Phase は実装せず簡素化を目指す
- トップダウン／ボトムアップはtraverse methodで区別

```
class uvm_topdown_phase(uvm_phase):

    @classmethod
    def traverse(cls, comp):
        ...

class uvm_build_phase(uvm_topdown_phase, common_phase):
      ...
```

```
uvm_common_phases = [
uvm_build_phase,
uvm_connect_phase,
uvm_end_of_elaboration_phase,
uvm_start_of_simulation_phase,
uvm_run_phase,
uvm_extract_phase,
uvm_check_phase,
uvm_report_phase,
uvm_final_phase]
```

Sequence をリファクタリング

- Sequence / Sequencer はLRMでは2つの章を費やしている
  - タイプ指定や下位互換性を維持するために実装が複雑になっている

- Python UVMのシーケンスでは以下の簡素化が行われている
  - Sequencer も Driver もパラメタライズされていない
  - uvm_sequence は uvm_sequence_item ではなく uvm_object を拡張
  - Sequencer が提供するアービトレーションは FIFOのみ
  - get_response() では Transaction ID のみを使用

UVMレポート機能は Logging で代用

- Logger はメッセージを生成
- Logging レベルでメッセージをフィルタリング
- Logging ハンドラで出力先を指定
  - スクリーン、ファイル、HTML、など
- Logging フォーマッタでメッセージを
  フォーマット化

pyuvmで追加 →

レベル	数値
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
FIFO_DEBUG	5
NOTSET	0

Logging の使用

全ての uvm_component が self.logger 機能を持つ

```
class LogComp(uvm_component):
    def run_phase(self):
        self.raise_objection()
        self.logger.debug("This is debug")
        self.logger.info("This is info")
        self.logger.warning("This is warning")
        self.logger.error("This is error")
        self.logger.critical("This is critical")
        self.logger.log(FIFO_DEBUG, "This is a FIFO message")
        self.drop_objection()
```

pyuvm のフォーマッタは UVMに類似 (message_idは無い)

```
DEBUG: <src_file>(6)[uvm_test_top.comp]: This is debug
INFO: <src_file>(7)[uvm_test_top.comp]: This is info
WARNING: <src_file>(8)[uvm_test_top.comp]: This is warning
ERROR: <src_file>(9)[uvm_test_top.comp]: This is error
CRITICAL: <src_file>(10)[uvm_test_top.comp]: This is critical
FIFO_DEBUG: <src_file>(11)[uvm_test_top.comp]: This is a FIFO message
```

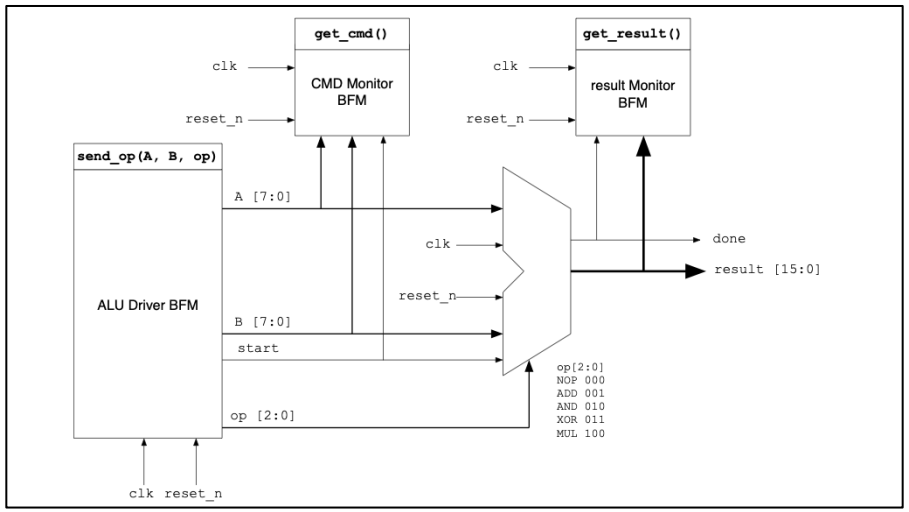
Logging のコントロール

- uvm_report_object
  - set_logging_level(logging_level)
  - add_logging_handler(handler)
    - StreamHandler - sys.stdout および sys.stderr にプリント
    - FileHandler - ファイルへの書込み
    - NullHandler - NOP
  - set_formatter_on_handlers(formatter)

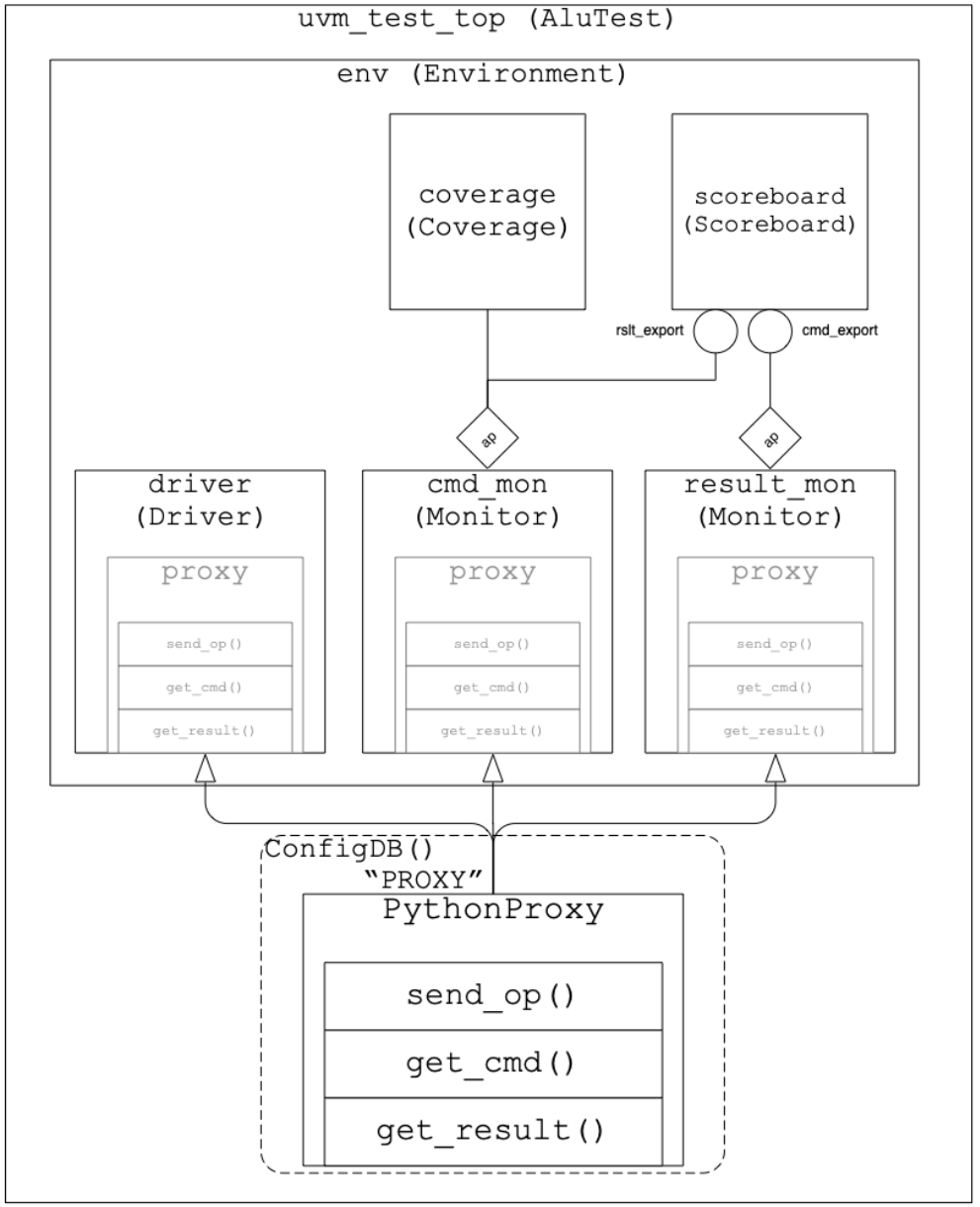
- uvm_component
  - set_logging_level_hier(logging_level)
  - add_logging_handler_hier(handler)
  - set_formatter_on_handlers_hier(formatter)

TinyALU 使用例

- シミュレータとは無関係に開発されたPythonテストベンチ
- cocotb を用いて開発されたシミュレータ用テストベンチ



cocotb proxy + BFM



Python / Simulation 共通のProxy Interface

```
proxy.send_op(aa, bb, op)
cmd = proxy.get_cmd()
result = proxy.get_result()
```

Blocking の API

```
class PythonProxy(uvm_component):
  def init (self, name, parent, label):
  super(). init (name, parent)
  ConfigDB().set(None, "*", label, self)
```

```
class CocotbProxy:
  def init (self, dut, label):
    self.dut = dut
    ConfigDB().set(None, "*", label, self)
    self.driver_queue = UVMQueue(maxsize=1)
    self.cmd_mon_queue = UVMQueue(maxsize=0)
```

UVMQueue の maxsize = 0 は無限のキュー

Monitor

```
class Monitor(uvm_component):
   def __init__(self, name, parent, method_name):
      super().__init__(name, parent)
      self.method_name = method_name
   def build_phase(self):
       self.ap = uvm_analysis_port("ap", self)
   def connect_phase(self):
       self.proxy = self.cdb_get("PROXY")
   def run_phase(self):
       while not ObjectionHandler().run_phase_complete():
          get_method = getattr(self.proxy, self.method_name)
          datum = get_method()
          self.ap.write(datum)
```

Pythonではモニター関数を
文字列で渡せてしまう

```
def build_phase(self):
   self.cmd_mon = Monitor("cmd_mon", self, "get_cmd")
   self.rslt_mon = Monitor("rslt_mon", self, "get_result")
```

cocotbによるモニター関数BFM

```
def get_result(self):
      return self.result_mon_queue.get()
```

Blocking の API

```
async def result_mon_bfm(self):  prev_done = 0
while True:
  await FallingEdge(self.dut.clk)
  try:
     done = int(self.dut.done)
  except ValueError:
     done = 0
  if done == 1 and prev_done == 0:

self.result_mon_queue.put_nowait(int(self.dut.result.value))
    prev_done = done
```

Functional Coverage

```
class Coverage(uvm_subscriber):
   def end_of_elaboration_phase(self):
      self.cvg = set()

   def write(self, cmd):
     self.cvg.add(cmd.op)

   def check_phase(self):
     if len(set(Ops) - self.cvg) > 0:
       self.logger.error(f"Functional coverage - Missed:
           {set(Ops)-self.cvg}")
```

Python には covergroup が無いため、すべてのopsと観測されたopsの差分を取り、抜けを見つける

Sequence Item

```
class AluSeqItem(uvm_sequence_item):
  <snip __init__>
  def __eq__(self, other):
    same = self.A == other.A and self.B == other.B and self.op == other.op return same
  def __str__(self):
    return f"{self.get_name()} : A: 0x{self.A:02x} OP: {self.op.name}"
                                 f"({self.op.value}) B: 0x{self.B:02x}"
  def randomize(self):
    self.A = random.randint(0,255)
    self.B = random.randint(0,255)
    self.op = random.choice(list(Ops))
```

do_compare() や convert2string() と等価な機能を Python 作法で実装している

ALU Test

- 環境 (env) のインスタンス化
- Logging Level を DEBUGにセット
- オブジェクション操作とシーケンス実行

```
class AluTest(uvm_test):
  def build_phase(self):
    self.env = AluEnv.create("env", self)

  def end_of_elaboration_phase(self):
    self.set_logging_level_hier(logging.DEBUG)

  def run_phase(self):
    self.raise_objection()
    seqr = ConfigDB().get(self, "", "SEQR")
    seq = AluSeq("seq")
    seq.start(seqr)
    time.sleep(1)
    self.drop_objection()
```

AluTest を拡張して Python/cocotb のTest

- Python の Proxy をインスタンス化
- “PROXY” にストア

```
class PythonAluTest(AluTest):
    def build_phase(self):
        _ = PythonProxy("model_proxy", self, "PROXY")
        super().build_phase()
```

- Test 終了を cocotb に伝えるのも Proxy を用いる

```
class CocotbAluTest(AluTest):
    def final_phase(self):
        cocotb_proxy = self.cdb_get("PROXY")
        cocotb_proxy.done.set()
```

```
% python tynyalu_uvm.py
```

```
if __name__ == "__main__":
    uvm_root().run_test("PythonAluTest")
```

Cocotb Test

cocotb の proxy をインスタンス化
自身が ConfigDB() にストアされる

```
@cocotb.test()
async def test_alu(dut):
    clock = Clock(dut.clk, 2, units="us")
    cocotb.start_soon(clock.start())
    proxy = CocotbProxy(dut, "PROXY")
    await proxy.reset()
    cocotb.start_soon(proxy.driver_bfm())
    cocotb.start_soon(proxy.cmd_mon_bfm())
    cocotb.start_soon(proxy.result_mon_bfm())
    await FallingEdge(dut.clk)
    test_thread = threading.Thread(target=run_uvm_test,
                args=("CocotbAluTest",), name="run_test")
    test_thread.start()
    await proxy.done.wait()
    await FallingEdge(dut.clk)
```

UVM test を起動

実際に試せます

- % pip install pyuvm

- Github上のオープンソース・プロジェクト
  - https://github.com/pyuvm/pyuvm

cocotb / pyuvm はバズるか

- UVM土俵で見ると
  - SystemVerilog よりすっきりしている / 必要なところだけを実装
  - VIP は SystemVerilog + UVM が圧倒的
- cocotb + その他プロジェクトの可能性
  - Coverage, Random, etc

- 誰が使うか
  - ソフトウェア設計者 or ハードウェア設計者
- 人口がどれくらいいるか
  - Python ⇔ SystemVerilog

質疑応答