VHDL generation from Python Synchronous Message Exchange Networks

Truls Asheim <truls@asheim.dk>

August 23, 2016

University of Copenhagen, Niels Bohr Institute

Outline

1. Introduction and motivation
2. Synchronous Message Exchange recap
3. Translating Python to VHDL
4. An example and test benches
5. Implementation
6. Sumary and future work

Introduction and Motivation

Motivation

Specialized hardware (FPGAs, ASICs) is more complicated to develop thant software.

Reduced power consumption, and parallel processing

Hardware development has a high barrier of entry and common Hardware Description Languages (HDLs) are hard to work with.

Particularly for test code

What we have

A transpiler (source-to-source compiler) capable of translating Python SME networks implemented using the PySME library to functionally equivalent VHDL code.

Automatic test bench generation! — Makes it easy to perpetually verify the correctness of the generated VHDL code.

Generated code can be simulated using VHDL simulators and/or synthesizers such as GHDL and Xilinx Vivado.

Proof of concept, but shows the potential of the SME model.

Synchronous Message Exchange Recap

SME is a globally synchronous message passing model, with an equivalence in CSP, mimicking signal propagation in hardware.

Single broadcasting channel type, called a bus by hardware analogy.

Conceived after an attempt to generate Vivado C and VHDL from PyCSP models showed that enforcing globally synchronous message propagation in pure CSP caused an explosion of complexity.

First presented at CPA 2014, with revised version at CPA 2015

It’s not High Level Synthesis (HLS)

HLS relies on auto-paralellizing sequential code.

- Efficiency of generated code can be an issue.
- Generated code difficult to understand
- Opaque translation process.
- Level of abstraction decreasing

Converting SME to VHDL is different

- SME makes it easy to program using hardware-like synchronous data propagation
- SME models already parallel
- Structural mapping to VHDL is trivial
- Level of abstraction mostly the same
- Close correlation between input and output

Why Python?

Using a general purpose programming language for hardware design means that:

- Increased accessibility for software developers.
- Nicer to work with than VHDL
- Easier testing/simulation:
  - Full ecosystem available
  - Existing code can be reused
  - Common and established libraries still available

Python is particularly well suited for rapid prototyping due to its high productivity nature.

Why not Python?

- Highly dynamic language, while hardware is inherently static.
- Only a subset of Python can be translated to VHDL.
- Type information required in VHDL — not provided by Python.

These are the main challenges of the translation.

Translating Python to VHDL

Translatable subset

Obviously, the complete Python language cannot simply be translated to VHDL. Only a restricted subset:

- Only conditionals and variables assignments (but almost full expressions)
- No loops (yet)
- No lists (yet). This is fairly limiting.

And some additional restrictions:

- SME process variables must be declared class-globally

Process Types

Two types of processes. Functions and Externals

Externals	Functions
Only used for simulation	Implements hardware targeted processes
Any Python code	Restricted (static) subset of Python
Only structure is translated	Translated completely

```
class Process(External):
  def setup(self):
    pass
  def clock(self):
    pass
```

```
class Process(Function):
  def setup(self):
    pass
  def clock(self):
    pass
```

`Function`s and `External`s are identical when simulating PySME — only different when translating to VHDL.

PySME to VHDL Overview

Mappings from PySME to VHDL

PySME	VHDL
Variable	VHDL variable or constant
Function parameter	Generic
Bus definition	VHDL ports and signals
External process	File containing skeleton translation
Function process	File containing complete translation
Network definition	File containing top-level entity

Types

Python is dynamically typed, while VHDL statically typed and require explicit type information.

Thus, we need to add type information

For variables, solved through a combination of “typing on first assign” (e.g. `self.n = 4` — `n` is a 32-bit signed integer) and optional annotations.

Annotations currently mandatory for bus channels.

Not a lot of types. Only signed and unsigned integers and booleans

But it not just types!

Number widths crucial for efficiency of implemented hardware since each bit of a number corresponds to a “wire” in the hardware implementation.

So we need to decide, not just types, but integer widths as well.

Annotations of signedness and bitwidths:

Variables `self.n = 0 # type: t.u12` The variable `n` is a 12-bit unsigned integer

Bus channels `Bus("ValueBus", [t.i24("val")])` The channel `val` of the bus `ValueBus` is a 24-bit signed integer

Better solution: Augment annotations with optimal width inference! (this is future work)

Constants

No constants in Python, but correct variable *constness* is important in VHDL!

So we designate variables that are never assigned to as constants in the VHDL code.

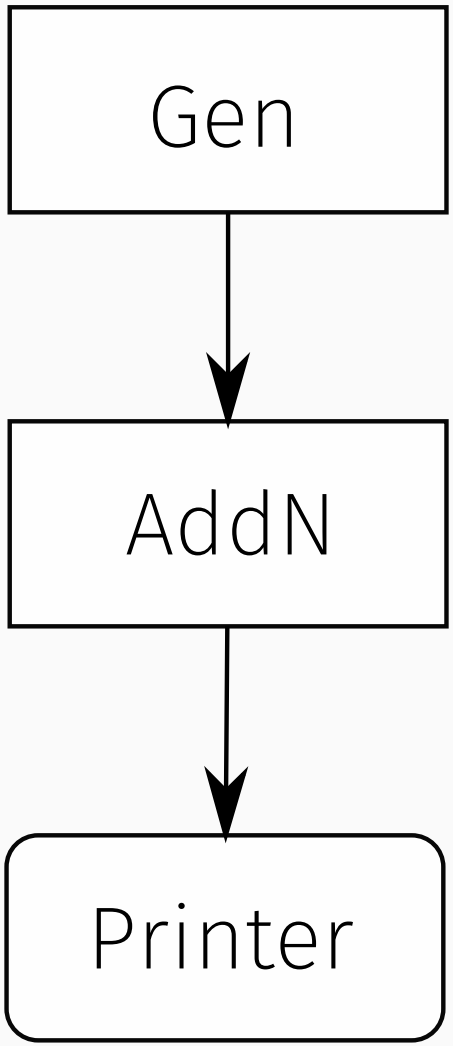
Names

The VHDL code we generate, should be easily recognizable and comprehensible by SME model implementer.

Preservation of process, variable, bus channel names important in ensuring this.

A Small Example

A small example (AddNNet)



The AddN network:

Three processes:

- Gen emits a parameter value
- AddN accumulates a value, added to value from Gen, a constant and a parameter value.
- Printer prints value from AddN

AddNNet Source Code (1/2)

```
from sme import Network, Function,
                External, Bus, SME,
                Types
t = Types()

class Gen(Function):
    def setup(self, ins, outs, n):
        self.map_outs(outs, "out")
        self.n = n  # type: t.u3

    def run(self):
        self.out["val"] = self.n

class AddN(Function):
    def setup(self, ins, outs, n):
        self.map_ins(ins, "num")
        self.map_outs(outs, "res")
        self.n = n
        self.c = 4  # type: t.u3
        self.accum = 0  # type: t.u10

    def run(self):
        self.accum += self.n + self.c +
                      self.num["val"]
        self.res["val"] = self.accum

class Printer(External):
    def setup(self, ins, outs):
        self.map_ins(ins, "res")

    def run(self):
        print(self.res["val"])
```

AddNNet Source Code (2/2)

```
33  class AddNNet(Network):
34      def wire(self):
35          bus1 = Bus("ValueBus",
36                     [t.u2("val")])
37          bus1["val"] = 0
38          self.tell(bus1)
39
40          bus2 = Bus("InputBus",
41                      [t.u10("val")])
42          bus2["val"] = 0
43          self.tell(bus2)
44
45          gen_param = 2
46          gen = Gen("Gen", [], [bus1],
47                    gen_param)
48          self.tell(gen)
49
50          addn_param = 4
51          addn = AddN("AddN", [bus1],
52                      [bus2], addn_param)
53          self.tell(addn)
54
55          p = Printer("Printer", [bus2], [])
56          self.tell(p)
57
58  def main():
59      sme = SME()
60      sme.network = AddNNet("AddNet")
61      sme.network.clock(100)
62
63  if __name__ == "__main__":
64      main()
```

AddN process

```
-- Library includes snipped
entity AddN is
  generic (n: integer);
  port (res_val: out u10_t;
        num_val: in u2_t;
        rst: in std_logic;
        clk: in std_logic
        );
end AddN;
architecture RTL of AddN is
begin
  process (clk, rst)
    constant c: u3_t := std_logic_vector(to_unsigned(4, u3_t'length));
    variable accum: u10_t := std_logic_vector(to_unsigned(0, u10_t'length));
  begin
    if rst = '1' then
      res_val <= std_logic_vector(to_unsigned(0, u10_t'length));
      accum := std_logic_vector(to_unsigned(0, u10_t'length));
    elsif rising_edge(clk) then
      accum := std_logic_vector(unsigned(accum) + to_unsigned(n, u10_t'length) +
               unsigned(c) + unsigned(num_val));
      res_val <= std_logic_vector(unsigned(accum));
    end if;
  end process;
end architecture;
```

Top-level entity

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4   use ieee.numeric_std.all;
5
6   library sme_types;
7   use work.sme_types.all;
8
9   entity AddNNet is
10    port (AddNNet_ValueBus_val:
11          inout u2_t;
12          AddNNet_InputBus_val:
13          inout u10_t;
14          rst: in std_logic;
15          clk: in std_logic
16          );
17  end AddNNet;
18  architecture RTL of AddNNet is
19    -- signals
20  begin
21    AddN: entity work.AddN
22    generic map (n => 4)
23    port map (num_val => AddNNet_ValueBus_val,
24              res_val => AddNNet_InputBus_val,
25              rst => rst,
26              clk => clk);
27    Gen: entity work.Gen
28    generic map (n => 2)
29    port map (out_val => AddNNet_ValueBus_val,
30              rst => rst,
31              clk => clk);
32    Printer: entity work.Printer
33    port map (res_val => AddNNet_InputBus_val,
34              rst => rst,
35              clk => clk);
36  end architecture;
```

Test Benches

Test Benches

A test bench is used for testing and verifying hardware descriptions.

- Test vectors generated by simulating a PySME model.
- Read by auto-generated VHDL test bench code.
- Values SME buses cycle-accurately mirrors the the values of the VHDL signals that they are transformed into.

Manual modifications of the generated VHDL code can be verified for correctness against the original Python implementation.

Trace CSV file

```
AddNNet_InputBus_val,AddNNet_ValueBus_val
0,0
8,2
18,2
28,2
38,2
48,2
58,2
68,2
78,2
88,2
98,2
108,2
[...]
```

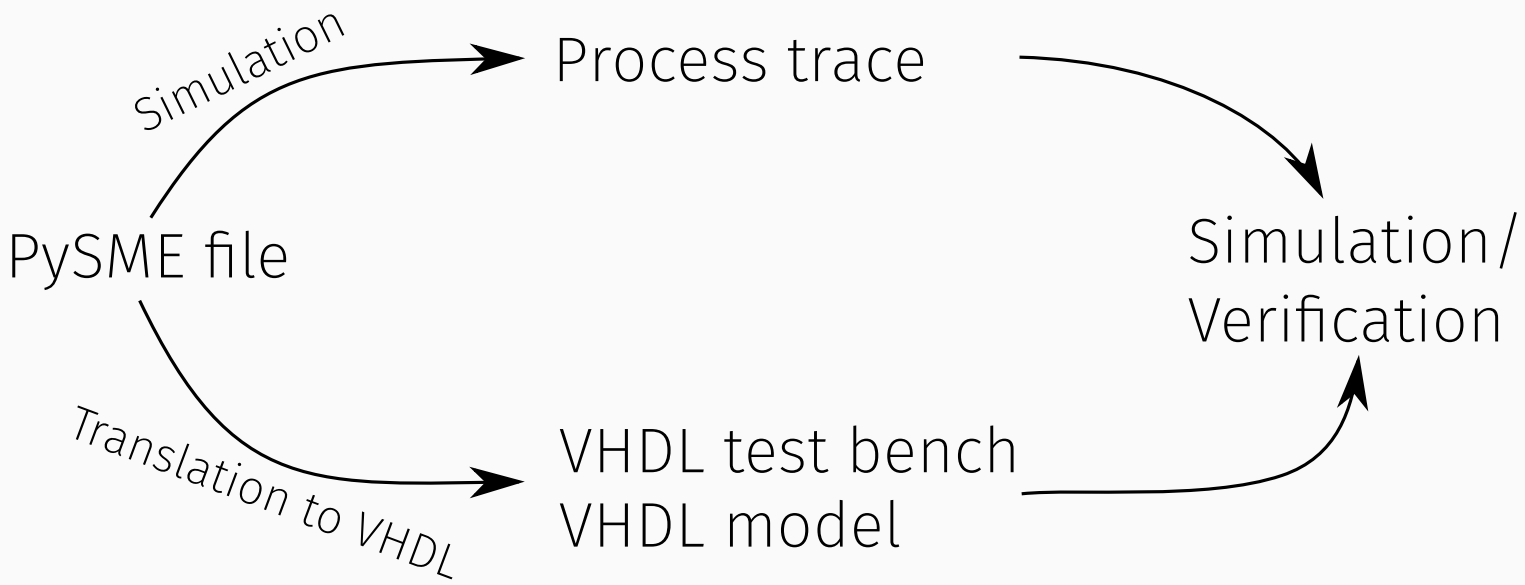
Test Bench Code

```
while not endfile(F) loop
  readline(F, L);
  wait until rising_edge(clock);
  fieldno := 0;
  read_csv_field(L, tmp);
  if not are_strings_equal(tmp, "U") then
    assert are_strings_equal(uint_image(AddNNet_InputBus_val), tmp)
      report "Unexpected value of AddNNet_InputBus_val in cycle " &
      integer'image(clockcycle) & ". Actual value was: " & uint_image(AddNNet_InputBus_val)
      & " but expected " & truncate(tmp) severity Error;
  end if;
  fieldno := fieldno + 1;

  read_csv_field(L, tmp);
  if not are_strings_equal(tmp, "U") then
    assert are_strings_equal(uint_image(AddNNet_ValueBus_val), tmp)
      report "Unexpected value of AddNNet_ValueBus_val in cycle " &
      integer'image(clockcycle) & ". Actual value was: " & uint_image(AddNNet_ValueBus_val)
      & " but expected " & truncate(tmp) severity Error;
  end if;
  fieldno := fieldno + 1;

  clockcycle := clockcycle + 1;
end loop;
```

Workflow overview



Running it

```
$ dist/build/almique/almique examples/addn.py
$ cd output
$ ls
AddN.vhdl  AddNNet.vhdl  AddNNet_tb.vhdl  Gen.vhdl
Printer.vhdl  csv_util.vhdl sme_types.vhdl
$ ghdl -a --ieee=synopsys --work=sme_types
  sme_types.vhdl Gen.vhdl AddN.vhdl Printer.vhdl
  csv_util.vhdl AddNNet.vhdl AddNNet_tb.vhdl
$ ghdl -e --ieee=synopsys --work=sme_types AddNNet_tb
$ python ../examples/addn.py -t trace.csv
$ ./addnnet_tb
AddNNet_tb.vhdl:91:5:@1us:(report note):
Completed after 100 clockcycles
$
```

Overall stats

63 lines of Python turns into 440 lines of VHDL (including test benches)

So a 270% increase in code size

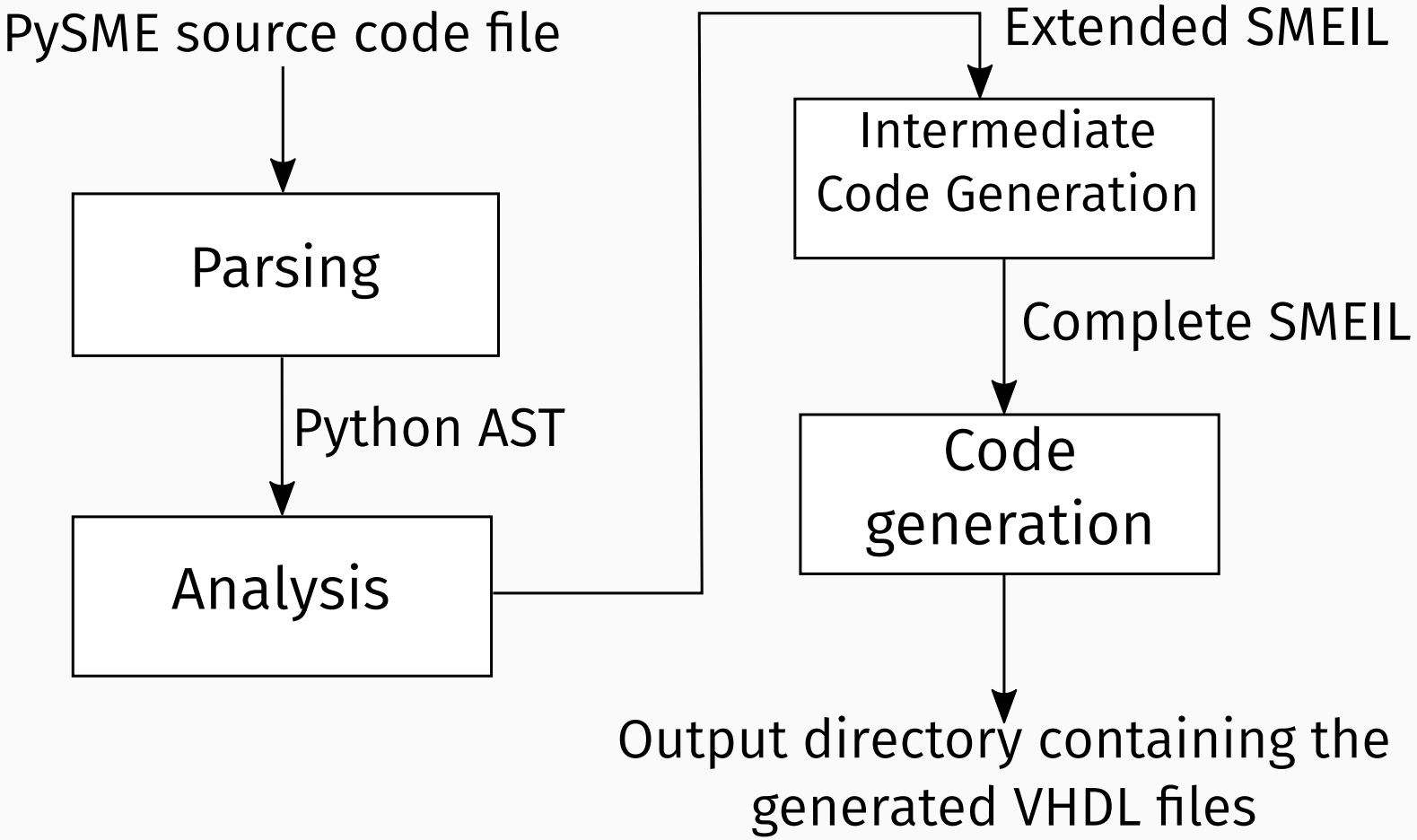
or 377 lines of code you didn’t have to write.

Implementation

The Transpiler

- Writen in Haskell
- 1883 SLOC (Including some inline VHDL)
- Python parsed using the `language-python` module
- VHDL generated using `Text.Pretty` pretty printing combinators
- Code transformation through “classic” compiler pipeline

Compilation pipeline



SMEIL is the SME Intermediate Language

Summary

We have a translation system which

- Translates Python SME programs to VHDL
- Produces functionally equivalent VHDL code with similar structure
- Close correlation between input and output code makes transformations transparent
- Automatic test bench generation allows for “lifecycle” verification of VHDL

Future work

So where do we go from here?

- Expanding supported Python subset (lists, loops, functions)
- Avoid annotations in the “standard case”
  - Optimal bitwidth inference
  - Improved type inference
- Standard library
- Floating point
- More dynamic and “Pythonic” translations enabled by improved abstract interpretation model

Thank you!

Complete transpiler source code:
`https://github.com/truls/almique`

PySME library source code:
`https://github.com/truls/pysme`

Questions?