Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016

VHDL Generation From Python Synchronous Message Exchange Networks

Truls ASHEIM [a], Kenneth SKOVHEDE [a] and Brian VINTER [a]
[a] *University of Copenhagen, Niels Bohr Institute*

Abstract. The Synchronous Message Exchange, SME, is a programming model that both resembles communication in hardware, and can be implemented as a CSP network. This paper extends on previous work for modeling hardware-like programs using SME in Python, with the addition of a source-to-source compiler that converts an SME network implemented in Python to an equivalent implementation in VHDL. We describe the challenges, constraints, and solutions involved in translating a highly dynamic language like Python into the hardware-like VHDL language. We also show how the approach can assist in further VHDL refinement by generating tedious test bench code, such that VHDL designs can be simulated and verified with vendor supplied simulation and synthesis tools.

Keywords. FPGA, VHDL, HLS, CSP, Synchronous Messaging, Haskell, Python, Transpiler

Introduction

Power consumption is a concern for many applications, ranging from tiny battery-operated devices used in Internet-of-Things, to super computer installations with megawatt sized computation units [1,2]. Both ends of the power-usage spectrum can benefit greatly by switching from using traditional CPUs to Application Specific Integrated Circuits (ASIC) or Field Programmable Gate Array (FPGA) solutions [3].

Unfortunately, writing applications for FPGAs and ASICs is significantly more complicated, due to the low-level nature of such hardware-like devices. MyHDL [4], CλaSH [5], Xilinx Vivado HLS [6] and Altera OpenCL [7] are existing approaches which aid in the building of high-level abstractions.

With Altera OpenCL, the programmer can use the OpenCL framework, which usually targets GPGPUs, and treat an FPGA device as an accelerator, similar to a GPGPU. The benefits of this model are the maturity and widespread use of OpenCL. The downside is that the FPGA device needs to communicate with a host at runtime, and thus it is unable to work stand alone, which would be needed for an Internet-of-Things device. Additionally, there is also some latency involved in memory transfers, as well as a lack of control for *how* the OpenCL code is implemented.

Xilinx Vivado HLS takes a more radical approach. Xilinx has defined a subset of the C programming language, and attempts to parallelize the code by transforming it into a finite state machine. Through extensive use of annotations, it is possible to tweak the program into generating even more parallel FPGA designs than the compiler can automatically deduce. One disadvantage of this approach is that C is not a high-productivity language. The use of annotations is a double-edged sword, in that it both makes it easier to change how the program is transformed, and makes it harder to guess and obtain the exact transformation desired.

The CλaSH and MyHDL solutions differ from the aforementioned approaches, as they are not languages themselves, but leverage existing programming languages, rather than defining new ones. The CλaSH library is closer to a *transpiler* (source-to-source compiler), in that it takes a subset of the Haskell language and transforms it into VHDL for *synthesis*[1] on hardware. MyHDL can also be described as a transpiler, in that it can transform a subset of Python code into VHDL, but it also contains a set of tools that allows simulation and testing entirely within the Python language.

Our work revolves around the use of Synchronous Message Exchange (SME) for designing hardware [8,9]. The SME logic is

based on CSP, but only contains a single broadcasting channel type, similar to how signals propagate in hardware. With the SME library, we are able to model hardware designs directly from within Python, using encapsulation and compositionality from CSP in the process.

However, the model built with SME is just that: a model; it cannot be used on FPGA devices or in ASIC designs. Instead, the SME model is designed to simulate hardware signals, just like the VHDL programming language. This makes it possible to transform a SME model into an equivalent VHDL design.

A common concern of High Level Synthesis (HLS) approaches, such as Vivado HLS mentioned above, is how the efficiency of their generated HDL code compares to hand-written implementations [10]. In our approach, this is less of an issue since we maintain a close correlation between the Python source code and the generated VHDL code. This is possible since a properly designed SME network is inherently parallel and has a structure which trivially maps to VHDL constructs. This means, that we do not need to rely on automatic parallelization of sequential code in order to infer the structure of the VHDL code we generate. This makes it easy for a PySME model developer to predict how his model will be translated to VHDL.

Contribution

In this work, we present a transpiler that works with the SME library to generate VHDL code directly from the SME model. The language design of Python, as well as the constructs used in SME, makes it attractive to write and test the model in SME, and then emit the verbose VHDL code, which can then be edited and used with standard FPGA and ASIC test- and synthesis tools.

A key advantage of using Python for writing SME models, is that the parts of the model that is not intended for hardware implementation can take advantage of the entire Python ecosystem. This includes modules and other preexisting code, such as a sequential reference implementation of an algorithm implemented as an SME network.

The resulting tool uses the SME network structure to generate a functionally equivalent VHDL project, where signal and process names are carried to the VHDL for easy identification. The transpiler is capable of transforming a subset of valid Python code into VHDL, such that the VHDL project utilizes VHDL constructs that perform the same operations as the Python code. Finally, the resulting tool generates test benches in VHDL, by capturing traces from executions with the SME model, such that the resulting designs can be verified as equivalent in a VHDL simulator.

We consider this approach to be more viable, as the VHDL output is human-readable and can easily be used as a starting point for a fully manual implementation, should the generated model be inefficient.

[1] Synthesis is the process of transforming a high-level hardware description into a gate-level on-chip implementation.

By modeling VHDL processes and signals, the generated network structure code that encapsulates and connects different components is likely optimal, such that only the inner VHDL code, which is transpiled from Python, needs to be considered. As the tool also outputs test benches, it is trivial to compare a handwritten version of a component with the generated version.

Although we are optimistic about the possibilities for using SME, we are well aware that the current implementation is in the proof-of-concept phase, when compared to the other, more mature, solutions mentioned previously.

1. Translating Python SME to VHDL

Numerous challenges arise when translating a highly dynamic language, like Python, to a static language such as VHDL. Most of these center around how to infer the static elements of a well-formed VHDL program from the Python source code which is destined for dynamic interpretation.

The PySME code supported and the VHDL code generated follows three guiding principles:

1. The generated VHDL code should be easily readable and recognizable in order to support subsequent manual modifications.
2. The Python SME network implementations should retain their "Pythonic" feel and the Python programmer should not be bound by more restrictions than necessary.
3. The Python programmer should be able to write his program as freely as possible, only respecting the restrictions that we impose, and the resulting VHDL code should retain the semantics and structure.

We refer to several VHDL language constructs in the remainder of this paper. For those unfamiliar with the language, we give their definitions in Table 3, included as an appendix.

1.1. Translatable Subset

As the hardware described by VHDL is inherently static, it is trivial to write a Python program that uses dynamic features, say a list, which is not translatable to VHDL. Furthermore, a portion of a Python program may contain code that is not intended for hardware implementation, such as a setup routine, a test bench, a verification suite, etc.

We leverage the distinction between `Function` and `External` processes introduced in [9] to allow the designer of the SME model to indicate if a process should be translated into VHDL or, conversely, if it is used purely for simulation.

`Externals` allow any Python code, while `Functions` are limited to a subset of Python that can be translated into VHDL, i.e., the translatable subset. Currently, this subset only contains variable assignments and conditionals. This is a very minimal subset, but as we additionally support most of the Python expression grammar, it is sufficient to allow implementation of most "hardware relevant" problems. However, lists are not currently included in the subset, and these could be useful in certain cases.

1.2. Structural mapping

VHDL is a highly hierarchical language consisting of a large number of different structural constructs. The mappings between structural elements in SME and VHDL are listed in Table 1.

Table 1. Mapping between structural elements in PySME and VHDL

PySME	VHDL
Class implementing a `Function` process	File containing a single `entity` definition and an `architecture` implementing the entity with a `process` containing the translated body of the SME process.
Class implementing an `External` process	File with only a single `entity` containing only the structural code needed to implement the functionality of the process
`Bus` definition	`Ports` and `signals` in `process` and `entity` definitions.
`Process` instantiation	Top-level `Entity` instantiation with `port maps` and `generic maps`

1.3. Variables

Python has only one kind of variable; a variable. VHDL on the other hand distinguishes between `variables`, `constants`, `ports` and `signals`. Another important difference between Python and VHDL is that Python allows definition of new variables anywhere while, VHDL, on the other hand, requires explicit declaration of all variables, constants, ports and signals. Listing 1 shows an example of how variable types are translated for a very simple SME process.

Ports and signals are relatively easy to handle since they correspond directly to the channels of SME buses. Variables are likewise not a problem since they have similar semantics in Python and VHDL. However, values of variables in VHDL are persistent across clock cycles and are thus similar to a variable defined in the dictionary of a Python class (e.g. `self.var = 0`) comprising a PySME process. For this reason, we currently enforce that all variables used in the clock functions of PySME processes must be defined class-globally in the setup function of the process.

Parameterized instantiation of processes is an essential method for reducing code duplication. In VHDL, this is achieved through `generics` which PySME process parameters are mapped to. Different from Python, however, is that generics in VHDL are constants and therefore we do not allow them to be modified.

Using correct variable *constness* in VHDL is more important than in CPU-targeted languages. Variables consume valuable FPGA space while constant values can simply be substituted upon synthesis. Therefore, we determine variable constness using a simple heuristic which designates Python variables that are never assigned to as constants in the VHDL code.

1.4. Types

The concept of types is dissimilar in hardware and software. Software uses a large number of simple and derived types to support a wide range of different values. In hardware, these are not representable, and thus the only "type" that exists is a wire, carrying an electrical signal, representing the value of a bit. By extension, integers can be thought of as a bundle of wires, each carrying a bit of the number. Consequently, using integers of appropriate widths (i.e. number of bits) is of crucial importance to the efficiency of the implemented hardware.

Handling the typing of PySME programs is a balancing act between complexity and practicality. If we require too much annotation, the complexity of the Python code will increase, reducing the productivity advantage it has over VHDL. If we, on the other hand, rely entirely on inferred typing, its usefulness for creating actual hardware designs decreases.

In an attempt to maintain this balance, we provide optional type annotations which can be used when one needs to manually define the types of variables. Thus, a hardware designer

```python
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
        self.accum = 0

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

```vhdl
[..]
entity AddN is
   generic (n: integer);
   port (res_val: out i32_t;
         num_val: in i32_t;
         rst: in std_logic;
         clk: in std_logic
         );
end AddN;
architecture RTL of AddN is
begin
   process (clk, rst)
   constant c: i32_t := [..]
   variable accum: i32_t := [..]
   begin
      if rst = '1' then
         res_val <= [..]
         accum := [..]
      elsif rising_edge(clk) then
         accum := [..]
         res_val <= [..]
      end if;
   end process;
end architecture;
```

Listing 1: Side-by-side comparison of a how a very simple SME process which adds three numbers together is translated to VHDL. The numbers added originate from a bus, a constant and a parameter respectively. This example shows how variables performing different roles in the python code are mapped to VHDL.

Table 2. Default python to VHDL type mappings

b**Python Value**	**VHDL Type**
Integer literal	`i32_t` (alias of `std_logic_vector(31 downto 0)`)
Bool literal	`std_logic`

can write PySME prototypes without type-related considerations and subsequently, when the model is ready for synthesis, annotate optimal number widths.

Unannotated variables are typed using a simple type inference scheme that is based on the requirement that all variables of translatable processes are assigned a literal (e.g. an integer value). This allows us to type variables using a set of predefined mappings listed in Table 2.

Type annotations are enabled through a dedicated PySME library module supporting type definitions. The module allows the definition of booleans and signed and unsigned integers of arbitrary widths. The type names follows a concise and intuitive format that is favored by several languages such as Rust and Nim, that has recently gained traction. Integer type names start with a single-letter prefix, `u` or `i`, meaning unsigned and signed respectively, followed by a number denoting the width of the number, e.g., `u13` is a 13-bit unsigned integer. Boolean types are denoted by the single letter `b`. For familiarity, we use similar type names in the generated VHDL code where they are defined as aliases of the `std_logic_vector` type.

To annotate Python variables (Listing 2) and function parameters, we use the syntax standardized by the Python Enhancement Proposal no. 484 [11]. Unfortunately, this syntax does not extend to express the typing of individual SME bus channels. Therefore, they are annotated using a syntax (Listing 3) reminiscent of object instantiations or function calls.

```
from sme import Types
t = Types()
[...]
class Controller(Function):
    def setup(self, ins, outs, rate, pixels):
        self.map_outs(outs, "controlbus")
        self.samplecnt = 0 # type: t.u8
        self.readcnt = 0 # type: t.u8
        self.readoutcnt = 0 # type: t.u8
[...]
```

```
architecture RTL of Controller is
begin
  process (clk, rst)
    variable samplecnt: u8_t := std_logic_vector(to_unsigned(0, u8_t'length));
    variable readoutcnt: u8_t := std_logic_vector(to_unsigned(0, u8_t'length));
    variable readcnt: u8_t := std_logic_vector(to_unsigned(0, u8_t'length));
```

Listing 2: PEP484-style type annotations of variables. All variables defined here are typed as 8-bit unsigned integers. First listing is the Python code, while the second shows the resulting variable definitions in the generated VHDL code. The code fragments are excerpts from the line-detector example discussed in Section 3.2

```
from sme import Types
t = Types()
[...]
class System(Network):
    def wire(self, pixels, buffer, rate, simdata, result):
        controlbus = Bus("Control", [t.b('readout'),
                                     t.u8('selector'),
                                     t.u8('data')])
[...]
```

```
entity Controller is
  generic (rate: integer;
           pixels: integer);
   port (controlbus_readout: out bool_t;
         controlbus_selector: out u8_t;
         controlbus_data: out u8_t;
         [..]);
```

Listing 3: Type annotations of bus definitions. The listing shows how type annotations of bus channels in the Python code (top) decides the types of ports in the VHDL code (bottom). Here, we define the bus "Control" with the channels "readout", "selector" and "data". The first channel is typed as a boolean, while the last two are typed as 8-bit unsigned integers.

1.5. Object naming

An important aspect of ensuring that the generated VHDL code remains recognizable and understandable by the implementer of the SME model is transforming names in a predictable and logical manner. The names of variables do not need to be changed since they have the same scope in Python and VHDL. Buses are more interesting since they a) transcend process boundaries and b) are considered discrete components in the SME model and thus have names themselves, but are bound to local names in PySME processes.

The names of bus channels are transformed with reference to their nearest bounding VHDL structure. For example, in the top level network description, the names of buses are transformed using the name of the network and the actual name of the bus. In Listing 4 we

```
1  [...]
2  class AddNNet(Network):
3      def wire(self):
4          bus1 = Bus("ValueBus",
5                     [t.u2("val")])
6          bus1["val"] = 0
7          self.tell(bus1)
8
9          bus2 = Bus("OutputBus",
10                    [t.u10("val")])
11         bus2["val"] = 0
12         self.tell(bus2)
13 [...]
14         addn_param = 4
15         addn = AddN("AddN", [bus1],
16                     [bus2],
17                     addn_param)
18         self.tell(addn)
19 [...]
```

```
1  [...]
2  entity AddNNet is
3    port (AddNNet_ValueBus_val:
4            inout u2_t;
5          AddNNet_InputBus_val:
6            inout u10_t;
7          rst: in std_logic;
8          clk: in std_logic
9          );
10 end AddNNet;
11 [...]
12   AddN: entity work.AddN
13   generic map (n => 4)
14   port map (num_val =>
15               AddNNet_ValueBus_val,
16             res_val =>
17               AddNNet_OutputBus_val,
18 [...]
```

Listing 4: An example showing the transformation of bus names in top-level entities

```
1  [...]
2  class AddN(Function):
3      def setup(self, ins, outs, n):
4          self.map_ins(ins, "num")
5          self.map_outs(outs, "res")
6  [...]
```

```
1  [...]
2  entity AddN is
3    generic (n: integer);
4    port (res_val: out u10_t;
5          num_val: in u2_t;
6  [...]
```

Listing 5: An example showing the names of buses bound to local names in processes are transformed. The `ins` and `outs` parameters of the `setup` function are set in lines 15-16 of the Python code in Listing 4

see how the bus named `ValueBus` defined in the network `AddNNet` with a single channel `val` gets the VHDL port name `AddNNet_ValueBus_val`.

Listing 5 we look at this mapping from the perspective of the `AddN` process. We see how `ValueBus` and `OutputBus` are mapped to the local names `num` and `res` respectively. The right side of the listing shows how those same local names are used in the generated VHDL code. This code should be understood in context of the code in Listing 4. Specifically, the values of the `ins` and `outs` parameters of the `setup` function are set to `[bus1]` and `[bus2]` respectively in lines 15–16 of the Python code in Listing 4.

1.6. Test Benches

Test Benches are used to verify the correctness of hardware implementations. By definition, they stimulate the inputs of a device and verify that its output values are as expected. Common hardware development workflows use VHDL for writing both the hardware specification and the test bench [12]. However, as cumbersome as VHDL is for writing the actual hardware, it is even less well suited for writing test code. We therefore consider automatic test bench generation one of the most appealing features of the SME model, yet it is uncomplicated, both in concept and in realization. The SME model facilitates this simplicity since the values of SME buses cycle-accurately mirror the values of the corresponding signals in the VHDL implementation.

We distinguish between *verifying test benches* and *full test benches*. A verifying test bench merely verifies, cycle-by-cycle, that the actual outputs of the simulated model correspond to the values in the trace file generated by executing the SME model. A full test bench, on the other hand, is self-stimulating in the sense that it both provides stimulus to the model being tested and verifies its output values.

Hence, verifying test benches can only be used with networks where input data is provided externally or the data generation processes is internal, i.e. translatable to VHDL. Full test benches, contrarily, can be used for the verification of any network since the test bench takes the role of the (usually unsynthesizable) data generation process and stimulates the network using values from the trace file.

Currently, we generate verifying test benches. However, a verifying test bench can be rewritten to a full one through a few simple and predictable modifications.

The source code of test benches is rather tedious, as one would imagine, and therefore, we will not show it here. We will, however, take a closer look at running VHDL model simulations using the generated test benches in Section 3.

```
clk: process
begin
  reset <= '1';
  wait for 5 ns;
  reset <= '0';

  while not stop_clock loop
    clock <= '1';
    wait for 5 ns;
    clock <= '0';
    wait for 5 ns;
  end loop;
  wait;
end process;
```

```
entity AddN is
  port ([...]; rst: in std_logic;
        clk: in std_logic);
end AddN;
architecture RTL of AddN is
begin
  process (clk,rst)
  begin
    if rst = '1' then
      -- Sets all output ports and
      -- variables to 0
    elsif rising_edge(clk) then
      -- Body of AddN function
    end if;
  end process; end architecture;
```

Listing 6: The process used in our automatically generated test benches to generate a clock signal (left) and the standard structure of a clocked process (right).

1.7. Clocking

One of the defining features of the SME model is its globally synchronous value propagation which mimics the clocked signal propagation found in hardware. While this gives the SME model an implicit clock, an explicit clock source needs to be present in hardware. In actual hardware, circuits are usually driven by an external clock source. By analogy, a clock source must also be connected to a VHDL hardware description before it can be simulated.

A typical VHDL clock source is a process, which use explicit timing to repeatedly raise and lower a clock signal generating a discrete-time signal that alternates between 0 and 1. VHDL processes must also follow a certain structure in order to be synchronously triggered. The clock process that we use in our test benches and the structure of clocked VHDL processes are listed in Listing 6. From the body of the process, shown on the right side of the listing, notice how the states of the `clk` and `rst` signals are used to control the process. Specifically, since the process would otherwise be triggered on every change of the `clk` signal, `rising_edge(clk)` is used to prevent the process body from being run multiple times per clock cycle and ensure that it is only triggered when the signal rises, i.e., transits from 0 to 1.

It is important to note that VHDL clock sources merely simulates a clock signal and will therefore not have any impact on the hardware implementation of a design.

2. Implementation details

We have implemented a transpiler in Haskell, which performs the actual translation of PySME code to VHDL that we have described in the previous sections. In this section we give

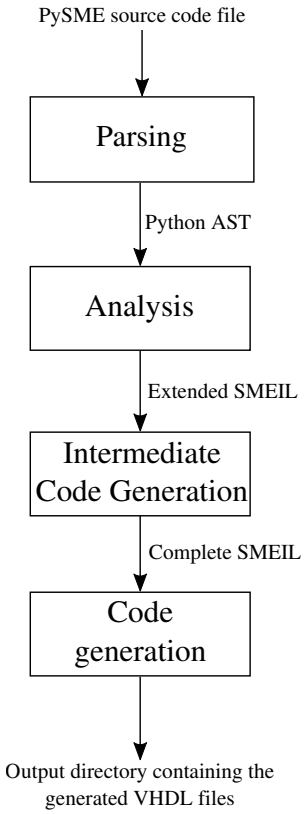


Figure 1. Compilation stages

an overview of its design and implementation before describing the intermediate language, SMEIL, used in the translation between Python and VHDL.

The complete source code for the transpiler and the examples that we present in Section 3 is available from the project GitHub repository [13].

2.1. Transpilation stages

The translation is performed in four stages. A high-level overview of the input and output of each stage is shown in Figure 1 with details provided below.

Parsing The Python code is parsed using the parser provided by the Haskell module `language-python` [14] which emits a Python AST that is passed on to the next stage.

Analysis The analysis stage identifies structures defined in the Python code and outputs what we refer to as Extended SMEIL (ESMEIL). ESMEIL is fragments of SMEIL syntax extended with contextual information containing the interpretation state when an object was instantiated. Such information includes, e.g., the bindings of the variables in scope.

Depending on which part of the Python code is being processed, different code processing strategies are employed. Python code that defines the structure of the SME network, such as bus instantiations, bus mappings and process instantiations, are processed using a *partial interpretation* strategy. Here, we are predominantly interested in understanding the meaning of the code, e.g., how processes are instantiated, which buses are defined and what is bound to which variables. The bodies of translatable processes are processed using a direct translation strategy which translates their Python code to semantically equivalent SMEIL code.

Intermediate code generation The ESMEIL is transformed into final SMEIL. In this process, we also assign types to our SMEIL program, decide variable constness and perform static consistency checks. Specifically, we check that no undefined variables are referenced

and that all referenced names are either buses, parameters or variables assigned to primitive values.

VHDL generation Based on the complete SMEIL tree, we generate the VHDL code. The code is spread across multiple files, one per VHDL entity, which are placed in a dedicated output directory, respecting VHDL best practices stating that a file should only contain a single entity. We format the code using a custom pretty printing library, which implements the subset of VHDL that we use. It is based on the pretty printing combinators in the `Text.Pretty` [15] Haskell module.

The separation between the analysis and SMEIL generation stages stems from our desire to support the translation of natural Python code. In combining the stages, we would need to impose additional restrictions on Python code in order to enable its translation. Specifically, we would need to enforce strict ordering requiring all classes, functions and variables to be defined before they are used.

2.2. SMEIL: The SME Intermediate Language

```
data Network =
  Network { netName :: Ident
          , functions ::
            [Function]
          , busses :: [Bus]
          , instances ::
            [Instance]
          }

data Instance =
  Instance { instName :: Ident
           , instFun :: Ident
           , inBusses :: [Ident]
           , outBusses :: [Ident]
           , instParams ::
             [(Ident, PrimVal)]
           }

data Bus =
  Bus { busName :: Ident
      , busPorts :: [(Ident, DType)]
      }

data Function =
  Function { funName :: Ident
           , funInports ::
             [(Ident, Ident)]
           , funOutports ::
             [(Ident, Ident)]
           , funParams :: [Decl]
           , locals :: [Decl]
           , funBody :: Stmts
           , funType :: FunType
           }

data Stmt =
  Assign Variable Expr
  | Cond [(Expr, Stmts)] Stmts
  | NopStmt

data Expr =
  BinOp { op :: BinOps
        , left :: Expr
        , right :: Expr
        }
  | UnOp { unOp :: UnOps
         , unOpVal :: Expr
         }
  | Prim PrimVal
  | Var Variable
  | Paren Expr
  | NopExpr
```

Listing 7: Excerpt of the SMEIL AST definition showing the supported statements and expression grammars

As part of the implementation, we have defined a Domain Specific Language (DSL) for describing SME networks. In our transpiler, it serves as the previously mentioned intermediate language between Python and VHDL. The language is basic, and only supports the constructs defined by the SME model. It operates at approximately the same level of abstraction as both Python and VHDL.

We have not defined a human-usable syntax for the language, as this would be of no benefit for the way we currently use it. However, the language is sufficiently expressive that it could easily serve as an independent DSL for implementing SME designs.

The entire SMEIL language is translatable to VHDL. Thus, any valid SMEIL program has an equivalent representation in VHDL. This is different from VHDL, where only a subset of the language is synthesizable to hardware and the remainder only usable for simulation.

The central elements of the abstract SMEIL syntax are listed in Listing 7 as Haskell datatype definitions. The ordering of the listing reflects the hierarchical nature of the language, where `Network` serves as the root element. For anyone familiar with the SME model, its building blocks should be easily recognizable in the SMEIL syntax.

Since the language is derived from the SME model, it inherits its central properties. In particular, the shared-nothing property of processes makes it easy to statically verify the consistency of the SMEIL code before generating the resulting VHDL code. Additionally, it contributes to modularizing the implementation by providing a clean, constant interface between the ends performing the Python interpretation and the VHDL generation. It also potentially enables retargeting to different source or destination languages. For instance, adding support for Verilog, another widespread Hardware Description Language, would be appreciated by many.

3. Examples

We revisit a couple of the examples from [9] and show how our transpiler handles them. The examples show that, even at this early stage, our transpiler is capable of realizing simple hardware design concepts.

The source code of the PySME networks and (especially) the generated VHDL code are too long to show here. For getting a more tangible idea of what the generated code looks like, we refer to the appendix where we have included an additional example, with source code.

3.1. Exponentially Weighted Moving Averages

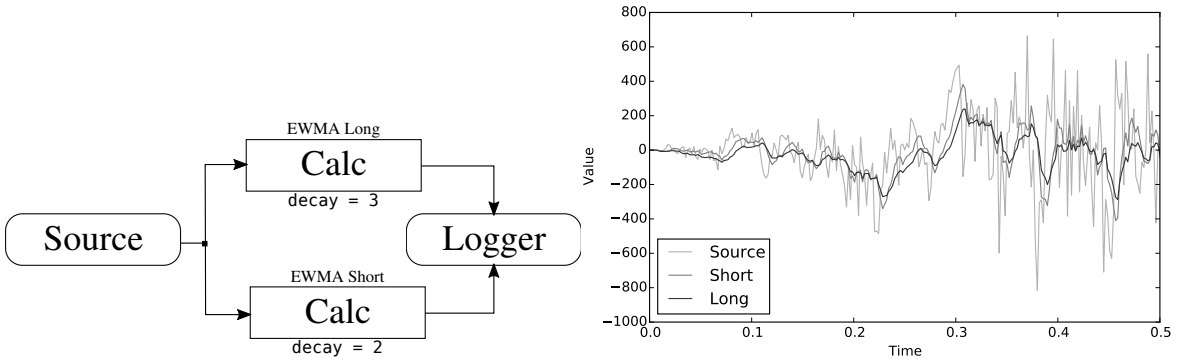
This example is taken from the design of a chip for performing High Frequency Trading (HFT) [16]. The Exponentially Weighted Moving Average (EWMA), is one of the metrics used to decide whether to buy or sell, a stock. In HFT, the decision window is only milliseconds long. Therefore, implementing the decision making entity as application-specific hardware directly connected to a data source (e.g. a network interface), eliminates the latency that a software implementation would otherwise induce.

The PySME implementation consists of three processes: The Source process reads input data and sends it on the bus, the Calc process performs the actual EWMA calculation and finally the Logger process receives and saves the results. Two instances of the EWMA calculator process are used. One is configured with a long decay time and the other with a short decay time. The structure of the network is depicted in Figure 2a When the Python implementation is executed, the results of the calculation are presented as a graph, similar to Figure 2b.

The test data used is generated by a Brownian bridge which is a stochastic process commonly used to simulate the development of the value of a financial asset over time [17].

The generated design can be imported into a synthesis tool, yielding schematics (Figure 3) similar to the original SME model; we have used Xilinx Vivado. Simulating the generated test bench verifies that the values produced by the VHDL implementation are consistent with the values produced by the SME model. The test bench will report any inconsistencies that are discovered (Figure 5), and the VHDL simulator will additionally generate a waveform (Figure 4) giving a trace of the values generated during simulation.

This described workflow demonstrates the main strength of our model. It shows how a SME network can be developed, simulated and verified in Python using familiar tools and libraries. After the design is converted to VHDL, the trace of values generated by simulating



(a) Network layout **(b)** Simulation result

Figure 2. Layout of the EWMA SME network and the result of simulating it. Boxes with rounded and square corners denotes External and Function processes respectively. The "Source" signal seen in the graph corresponds to the output of the Source process. It is not an actual output of the SME network, but is added to the graph by the non-translatable parts of the Python code to visualize the results of the EWMA calculations.

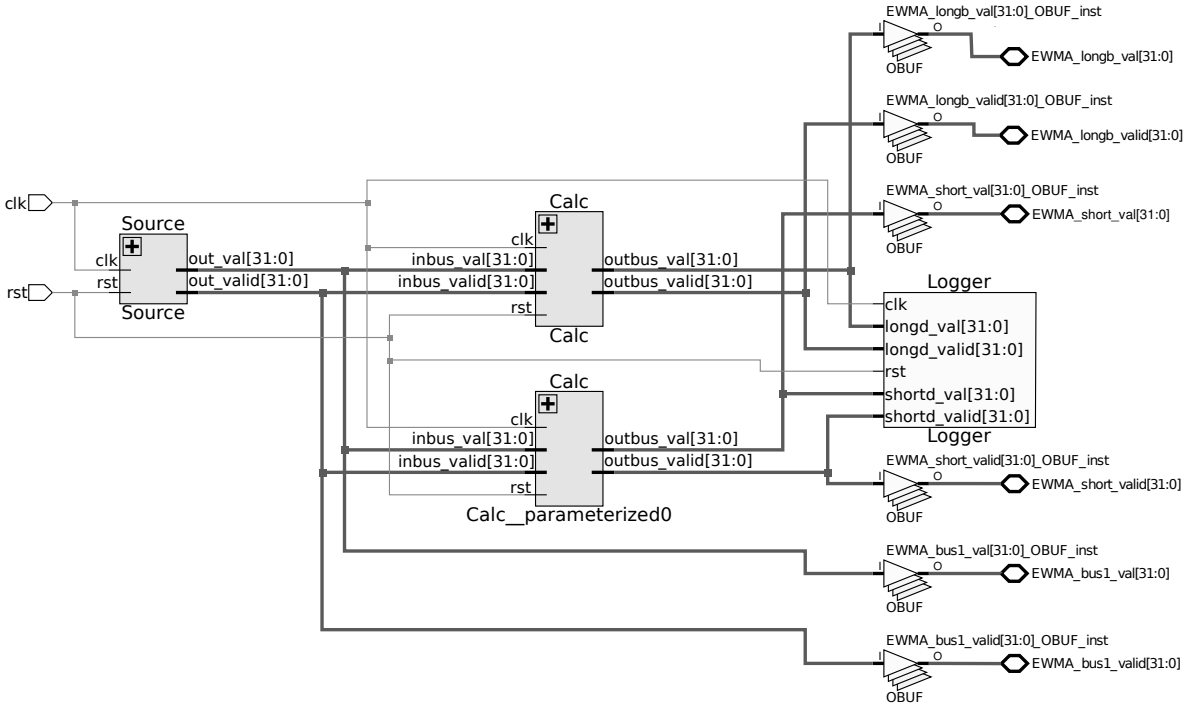


Figure 3. The EWMA calculation network elaborated from the generated VHDL code by Xilinx Vivado. The texts in the schematic have been enlarged for better legibility.

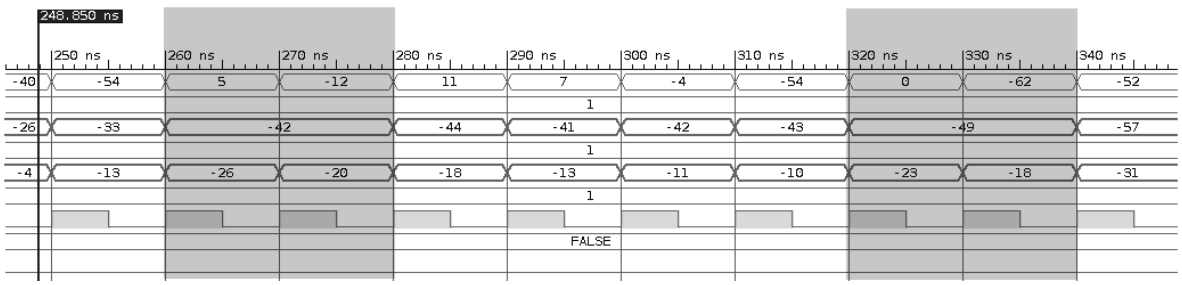


Figure 4. Trace of signal values as presented by Xilinx Vivado during simulation. Each vertical line denotes the end of one clock cycle and the start of the next. From the top, the signals shown are input data, input bus validity, EWMA long decay, EWMA long decay validity, EWMA short decay, EWMA short decay validity, clock and reset. The shaded areas highlights cycles where its obvius that, as expected, the EWMA with long decay times reacts slower to input signal changes.

```
System_tb.vhdl:113:3:@11590ns:(assertion error): Unexpected value of
  System_Control_selector in cycle 1158. Actual value was: 104 but expected 103
```

Figure 5. Example of errors reported by the automatically generated VHDL test bench when expected and actual values are inconsistent.

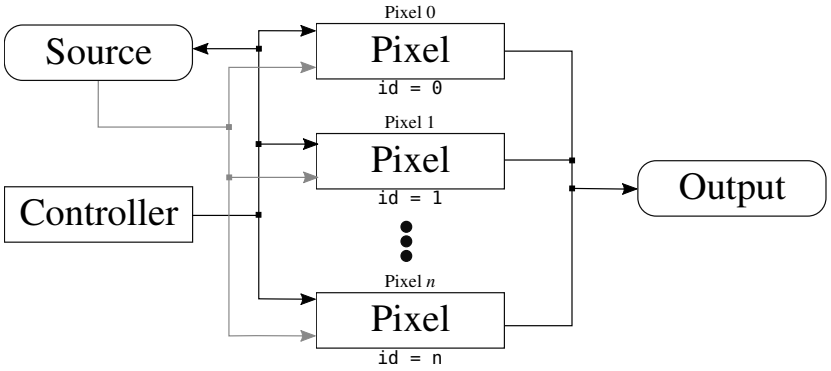


Figure 6. SME network of the line detector. Processes with rounded and square corners are `Externals` and `Functions` respectively.

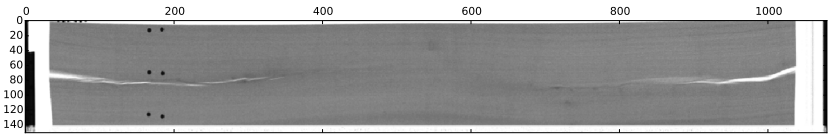


Figure 7. Image produced by simulating the line scanner in Python

the PySME model are then used to verify functional equivalence, through the automatically generated test benches, directly in the FPGA vendor's tooling.

3.2. Line-detector

Our second example is a scaled down version of an X-ray camera [18]. The camera consists of a 1 by n array of detector pixels that captures an image. In order for the image to be transferred, the value of the pixels must be read out, one by one, and serially transmitted on the output bus. The structure of the SME implementation is depicted in Figure 6 and the output of running the simulation in Python is shown in Figure 7. The image used for the simulation is an actual X-ray scan of a wooden plank.

The network has two modes of execution: the Pixel processes either receive a pixel value from the Source process or they write a pixel value to the Output process. Since the Pixel processes are connected to the Source and Output processes through shared buses, only a single Pixel process can send or receive a value during a clock cycle. The Controller controls this process and signals which Pixel process gets to read or write and when the Source process should write a value.

In the original PySME implementation of this example, described in [9], the `Pixel` processes directly accessed a NumPy array holding the input data. This design accurately reflects the system that the example is derived from, but since we are currently unable to automatically translate processes that use NumPy arrays (and `Pixel` processes contribute the bulk of the implementation), it would not do much to exhibit the capabilities of our transpiler. Therefore, the design has been modified so that the data is transmitted to the `Pixel` processes from a dedicated source process. This change makes the example somewhat contrived, but it does a better job at demonstrating the level of translation that we are able to do.

The generated VHDL code is transformed into the schematic depicted in Figure 8.

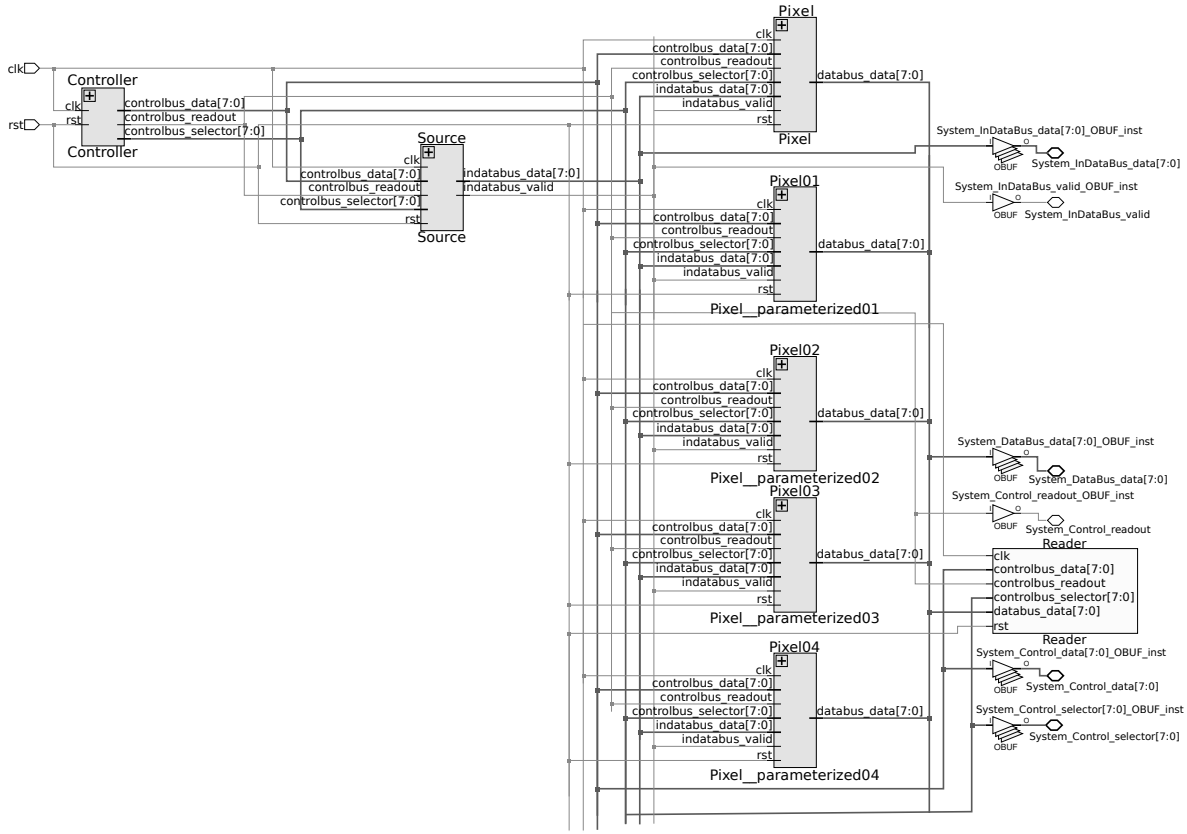


Figure 8. Schematics of the line-detector network generated by Xilinx Vivado. The drawing has been cropped to show only 5 of the 150 pixel processes and its texts has been enlarged to improve legibility.

Finally we note that in order for this example to be synthesizable, manual modification of the generated code is required. The design of the line detector requires that the output ports of pixel processes are connected to the input port of the output process through a common, shared wire. In hardware, this is problematic since multiple contradicting electrical signals driving the same wire will produce a non-deterministic output. Therefore, before a hardware implementation of the design is viable, we need to ensure that only a single connected output port can affect the value of a shared wire during a clock cycle. In VHDL, this is prevented by letting unused ports (i.e. ports that should not be assigned a value during a cycle) assume a high-impedance state (denoted by the special value "Z" in the `std_logic` type of VHDL).

The specific code modification required for this example is shown in Listing 8. The value of the `controlbus_selector` signal determines which pixel process gets to write on the output bus. Since the `id` value of every pixel process in the network is unique, only a single pixel process is active and writes a value on the output bus during a clock cycle. Thus, by adding the `else`-clause shown in the second part of the listing, we prevent any signal from leaving the output ports of the inactive processes.

In spite of being relatively simple, requiring such manual modifications of the generated VHDL code in a common case is undesirable. Therefore, we propose two different methods which could lessen or remove this requirement. The first, which is trivial to implement, is adding the VHDL special values to PySME allowing them to be set directly from the Python code (Listing 9). While this method remove the need for manually editing the generated VHDL code in this case, it violates the principles outlined in Section 1 by moving low-level hardware-related details into the Python code. Alternatively, it may be possible to implement an automatic method that use control flow analysis to identify code paths which leaves bus

```
1 if signed(controlbus_selector) = to_unsigned(id, 8) then
2   databus_data <= std_logic_vector(unsigned(data));
3 end if;
```

becomes

```
1 if signed(controlbus_selector) = to_unsigned(id, 8) then
2   databus_data <= std_logic_vector(unsigned(data));
3 else
4   databus_data <= "ZZZZZZZZ";
5 end if;
```

Listing 8: Manual addition to the VHDL code generated from the `Pixel` SME process setting a high-impedance default output value (lines 3 and 4 inserted).

```
1 if self.controlbus['selector'] == self.id:
2     self.databus['data'] = self.data
```

becomes

```
1 from sme import Special
2 [...]
3 if self.controlbus['selector'] == self.id:
4     self.databus['data'] = self.data
5 else:
6     self.databus['data'] = Special.Z
```

Listing 9: Example of how the need for modifying the generated VHDL code as shown in Listing 8 could be eliminated by adding support for hardware-specific special values to the PySME library.

channels unassigned. Modifications similar to those made in Listing 9 can then be performed as needed, on the intermediate code level, to ensure that any possible code path leaves all bus channels assigned.

4. Conclusion

We have described a translation system that utilizes the SME model, and the unique properties of SME networks, to automatically build synthesizable VHDL designs from a Python program. This enables hardware designers, and software developers as well, to quickly prototype ideas entirely within the Python programming environment. Once the model works as desired, the model can be automatically translated to VHDL. This translation process places additional constraints on the program, such as only using statically sized elements, and supports optional annotations to augment the type inference otherwise used. The generated VHDL design can then be used with standard vendor tools, such as simulators and synthesis tools. The design retains much of the structure and names from the original design, which enables further low-level or device specific optimizations with the VHDL code. Furthermore, this makes it easy to identify how the Python source code should be modified in order to fix errors discovered in the generated VHDL code. The generated test bench is produced as part of the translation, and can be used to verify that the generated code is working as expected in the translated version, and can further be used to verify that hand-modified VHDL still work as expected.

5. Future work

The most immediate focus for additional development efforts is on widening the supported subset of the Python language. We believe that there are still low-hanging fruits (features that we can easily support) that will improve the usefulness of the transpiler, enabling translation of simpler and more Python-like implementations.

Scaling our programming model to larger implementation projects requires PySME models to be split across multiple modules (i.e. files). We can leverage the Python module system for this, and thus only minimal modifications of the PySME library are required. We do, however, need to support translating this into VHDL. Additionally, supporting modules will allow us to provide an SME standard component library, of re-usable components, performing common tasks. These should be easily integrateable into PySME projects.

Another interesting subject of exploration is using static analysis to bound the range of numeric variables of SMEIL intermediate code [19]. Implementing this will diminish the current requirement to annotate the appropriate size of integers, as this could rather be inferred. Furthermore, since also the number of list indices is bound by this process, it would enable translation of a subset of unannotated Python lists to VHDL arrays.

Numerous scientific problems can only be implemented using floating point numbers and our current inability to support them is therefore somewhat limiting. Representing and manipulating these numbers in FPGAs, however, is currently problematic.The floating point arithmetic algorithms are complicated and take up a lot of space when implemented as hardware. In spite of this, we should provide a standard VHDL implementation of floating point numbers, enabling their use by programs that require them. On a related note, the dire situation of floating point support in FPGAs may improve in the future as more manufacturers add native support. Altera, an FPGA manufacturer, currently have one FPGA with integrated support for floating point calculations [20].

In some ways, the objectives of this project and the PyPy Python JIT compiler are enabled by the same method, namely, extracting a subset of Python code that is suitable for static analysis. The PyPy JIT compiler performs full abstract interpretation of Python code, generating a control flow graph which can be statically typed and used as the basis for JIT compilation [21]. We should investigate if employing a similar approach will enable us to support a wider subset of python.

References

[1] Martin C Herbordt, Tom VanCourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, and Doug DiSabello. Achieving high performance with fpga-based computing. *Computer*, 40(3):50, 2007.

[2] Harold Martin, Enrique San Millán, Pedro Peris-Lopez, and Juan E Tapiador. Efficient asic implementation and analysis of two epc-c1g2 rfid authentication protocols. *Sensors Journal, IEEE*, 13(10):3537–3547, 2013.

[3] Peter Bishop. A tradeoff between microcontroller, dsp, fpga and asic technologies. *EE Times design*, 2009.

[4] Jan Decaluwe. Myhdl: a python-based hardware description language. *Linux journal*, 2004(127):5, 2004.

[5] Rinse Wester. *A transformation-based approach to hardware design using higher-order functions*. PhD thesis, University of Twente, 2015.

[6] Xilinx Inc. Vivado hls overview. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`. [Online; accessed June 2016].

[7] Altera corporation. Altera sdk for opencl. `https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.highResolutionDisplay.html`. [Online; accessed June 2016].

[8] Brian Vinter and Kenneth Skovhede. Synchronous message exchange for hardware designs. *Proceedings of Communicating Process Architectures 2014*, 2014.

[9] Brian Vinter and Kenneth Skovhede. Bus centric synchronous message exchange for hardware designs. *Proceedings of Communicating Process Architectures 2015*, 2015.

[10] Ekawat Homsirikamol and Kris Gaj. Can high-level synthesis compete against a hand-written code in the cryptographic domain? a case study. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*. IEEE, 2014.

[11] Guido van Rossum, Jukka Lehtosalo, and ÅĄukasz Langa. Pep 484 – type hints. `https://www.python.org/dev/peps/pep-0484/`, 09 2014. [Online; accessed June 2016].

[12] Douglas L Perry. *VHDL: programming by example*, volume 4. McGraw-Hill, 2002.

[13] Truls Asheim. Almique source code. `https://github.com/truls/almique`. [Online; accessed June 2016].

[14] Bernard James Pope. language-python: Parsing and pretty printing of Python code. `http://hackage.haskell.org/package/language-python-0.5.3`, Jun 2015.

[15] John Hughes. The design of a pretty-printing library. In *International School on Advanced Functional Programming*, pages 53–96. Springer, 1995.

[16] Abdalla Kablan and Joseph Falzon. The use of dynamically optimised high frequency moving average strategies for intraday trading. *Word Academy of Science, Engineering and Technology*, 2012.

[17] Paul Glasserman. *Monte Carlo methods in financial engineering*, volume 53. Springer Science & Business Media, 2003.

[18] T Lohse, P Krüger, H Heuer, M Oppermann, H Torlee, and N Meyendorf. Counting x-ray line detector with monolithically integrated readout circuits. In *SPIE Microtechnologies*, pages 87632Q–87632Q. International Society for Optics and Photonics, 2013.

[19] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Dunod*, 1976.

[20] Altera corporation. The industryâĂŹs first floating-point fpga. `https://www.altera.com/en_US/pdfs/literature/po/bg-floating-point-fpga.pdf`. [Online; accessed June 2016].

[21] Armin Rigo and Samuele Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953. ACM, 2006.

[22] DOULOS. *The VHDL Golden Reference Guide*. DOULOS, 1995.

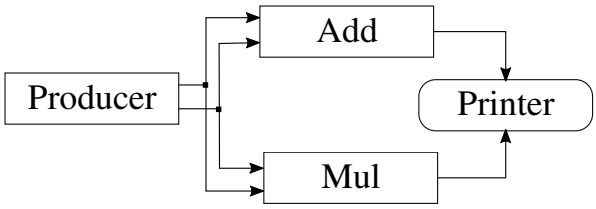


Figure 9. Structure of the SomeOps network

Definition of VHDL terms

Table 3. Definitions of VHDL concepts referred to in the paper.

VHDL term	Definition
`entity`	Entities define an interface to a component of a design. They are the top-level VHDL hierarchical element.
`architecture`	An architecture implements the functionality of an entity. Several architectures may be defined for single entity, all adhering to the common interface defined by the entity.
`process`	Processes are defined as sequential collections of statements running concurrently. This definition makes VHDL processes analogous to processes in SME or CSP.
`signal`	A signal represents a bus between two or more components. In usage, signals are similar to variables, except that values written to signals have a propagation time and are thus only visible in the subsequent clock cycle.
`port`	Entities can define ports which represents points where entities can be connected to each other forming a network.
`std_logic`	Is defined as a part of the IEEE standard libraries for hardware design. The `std_logic` type represents a wire in hardware carrying a bit. In addition to the values 0 and 1 it can, among others, take the values Z, X or -, meaning high-impedance, undecidable or don't care respectively
`std_logic_vector`	As the name suggests, it is a vector of `std_logic`s or a bundle of wires. It is used for representing values spanning over more than a single bit in VHDL.

We define the meanings of VHDL terms that may be alien to readers who are not familiar with VHDL in Table 3 [22].

SomeOps example with source code

In this example, with source code included, we look at the SomeOps network. It is a very simple network consisting of three processes: The `Producer` generates two numbers which are passed on to two processes, `Add` and `Mul`. They respectively perform arithmetic addition or multiplication of the values that they receive. The final process is `Printer`, which prints the results of the calculations. The structure of the network is depicted in Figure 9.

The network is a simplified version of the AllOps network described in [9] which only includes some of the operations of the original. Hence the name SomeOps.

By showing this example, we intend to give a more complete demonstration of the quality of the generated VHDL code. SomeOps is well suited for this purpose as it is quite simple, and thus it is easy to view comprehensively.

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

class Producer(Function):
    def setup(self, ins, outs):
        self.map_outs(outs, "out")
        self.v1 = 0  # type: t.u7
        self.v2 = 0  # type: t.u7

    def run(self):
        self.out["val1"] = self.v1
        self.out["val2"] = self.v2
        self.v1 += 1
        self.v2 += 1
        if self.v1 > 100:
            self.v1 = 0
            self.v2 = 0

class Mul(Function):
    def setup(self, ins, outs):
        self.map_ins(ins, "valbus")
        self.map_outs(outs, "mulbus")

    def run(self):
        self.mulbus["res"] =
          self.valbus["val1"] *
          self.valbus["val2"]

# The definition of the Add class is
# similar

class Printer(External):
    def setup(self, ins, outs):
        self.map_ins(ins, "addbus",
                     "mulbus")
    def run(self):
        print(self.addbus["res"],
              self.mulbus["res"])

class SomsOps(Network):
    def wire(self):
        valbus = Bus("ValueBus",
                     [t.u7("val1"),
                      t.u7("val2")])
        valbus["val1"] = 0
        valbus["val2"] = 0
        self.tell(valbus)

        addbus = Bus("AddBus", [t.u8("res")])
        addbus["res"] = 0
        self.tell(addbus)

        mulbus = Bus("MulBus", [t.u14("res")])
        mulbus["res"] = 0
        self.tell(mulbus)

        prod = Producer("Producer", [], [valbus])
        self.tell(prod)

        add = Add("Add", [valbus], [addbus])
        self.tell(add)

        mul = Mul("Mul", [valbus], [mulbus])
        self.tell(mul)

        printer = Printer("Printer",
                          [addbus, mulbus], [])
        self.tell(printer)

def main():
    sme = SME()
    sme.network = SomeOps("SomeOps")
    sme.network.clock(200)

if __name__ == "__main__":
    main()
```

Listing 10: Python source code defining the SomeOps model

The (almost) complete Python code of the SomeOps network is listed in Listing 10. The top level entity, listed in Listing 11, is generated from the `wire` function of the Python code. Here, the structure of the network is defined and connections between the processes are established.

In Listing 12, the translated source code of the `Mul` process can be seen. Notice how the port mapping between the top-level entity and the `Mul` process takes place. The `Add` process is similar to the `Mul` process and it is therefore not included here.

The final VHDL source code example that we show is the translated source code of the `Producer` process (Listing 13). It is a slightly more complicated process which sends values between 1 and 100 on its two output ports.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
```

```
5
6  library sme_types;
7  use work.sme_types.all;
8
9
10 entity SomeOps is
11   port (SomeOps_AddBus_res: inout u8_t;
12         SomeOps_MulBus_res: inout u14_t;
13         SomeOps_ValueBus_val1: inout u7_t;
14         SomeOps_ValueBus_val2: inout u7_t;
15         rst: in std_logic;
16         clk: in std_logic
17         );
18 end SomeOps;
19 architecture RTL of SomeOps is
20   -- signals
21 begin
22   Add: entity work.Add
23   port map (valbus_val1 => SomeOps_ValueBus_val1,
24             valbus_val2 => SomeOps_ValueBus_val2,
25             addbus_res => SomeOps_AddBus_res,
26             rst => rst,
27             clk => clk);
28   Mul: entity work.Mul
29   port map (valbus_val1 => SomeOps_ValueBus_val1,
30             valbus_val2 => SomeOps_ValueBus_val2,
31             mulbus_res => SomeOps_MulBus_res,
32             rst => rst,
33             clk => clk);
34   Printer: entity work.Printer
35   port map (addbus_res => SomeOps_AddBus_res,
36             mulbus_res => SomeOps_MulBus_res,
37             rst => rst,
38             clk => clk);
39   Producer: entity work.Producer
40   port map (out_val1 => SomeOps_ValueBus_val1,
41             out_val2 => SomeOps_ValueBus_val2,
42             rst => rst,
43             clk => clk);
44 end architecture;
```

Listing 11: The contents of the generated `SomeOps.vhdl` holding the top-level entity of the VHDL design.

```
1  -- Common library imports snipped
2  entity Mul is
3    port (mulbus_res: out u14_t;
4          valbus_val1: in u7_t;
5          valbus_val2: in u7_t;
6          rst: in std_logic;
7          clk: in std_logic
8          );
9  end Mul;
10 architecture RTL of Mul is
11 begin
12   process (clk, rst)
13   begin
14     if rst = '1' then
15       mulbus_res <= std_logic_vector(to_unsigned(0, u14_t'length));
16     elsif rising_edge(clk) then
```

```
      mulbus_res <= std_logic_vector(resize(unsigned(valbus_val1) *
                                            unsigned(valbus_val2), 14));
    end if;
  end process;
end architecture;
```

Listing 12: The contents of the `Mul.vhdl` as generated from the `Mul` PySME process.

```
-- Common library imports snipped
entity Producer is
  port (out_val1: out u7_t;
        out_val2: out u7_t;
        rst: in std_logic;
        clk: in std_logic
        );
end Producer;
architecture RTL of Producer is
begin
  process (clk, rst)
  variable v2: u7_t := std_logic_vector(to_unsigned(0, u7_t'length));
  variable v1: u7_t := std_logic_vector(to_unsigned(0, u7_t'length));
  begin
    if rst = '1' then
      out_val1 <= std_logic_vector(to_unsigned(0, u7_t'length));
      out_val2 <= std_logic_vector(to_unsigned(0, u7_t'length));
      v2 := std_logic_vector(to_unsigned(0, u7_t'length));
      v1 := std_logic_vector(to_unsigned(0, u7_t'length));
    elsif rising_edge(clk) then
      out_val1 <= std_logic_vector(unsigned(v1));
      out_val2 <= std_logic_vector(unsigned(v2));
      v1 := std_logic_vector(unsigned(v1) + to_unsigned(1, u7_t'length));
      v2 := std_logic_vector(unsigned(v2) + to_unsigned(1, u7_t'length));
      if unsigned(v1) > to_unsigned(100, u7_t'length) then
        v1 := std_logic_vector(to_unsigned(0, u7_t'length));
        v2 := std_logic_vector(to_unsigned(0, u7_t'length));
      end if;
    end if;
  end process;
end architecture;
```

Listing 13: Contents of the `Producer.vhdl` file generated from the PySME definition of the `Producer` function.