Enhancing Hardware Design Flows with MyHDL

Keerthan Jaic, Melissa C. Smith
Holcombe Department of of Electrical and Computer Engineering
Clemson University, Clemson, South Carolina, USA
{kjaic, smithmc}@clemson.edu

ABSTRACT

MyHDL is a Python based HDL that harnesses the power and versatility of Python for hardware development. MyHDL has excellent simulation capabilities and also allows for conversion to Verilog and VHDL, so developers can enter a conventional design flow as desired. Verilog and VHDL are used extensively, particularly because most synthesis tools only support these two languages. However, they are simply outdated; poor parameterization limits high level design and modern abstraction features such as classes are missing.

On the other hand, MyHDL has great support for parameterization. However, MyHDL did not have support for converting code that used attributes, so abstraction was limited. We extended MyHDL support to include attribute conversion. We explored methods for abstracting interfaces between components and hardware-software interfaces. The result is increased code reuse, simplified module declaration, and reduced boilerplate. These extensions result in streamlining between design, simulation, and a final synthesizable hardware, thus reducing limitations on high level development and making MyHDL an even more powerful design environment for rapid hardware prototyping.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*hardware description languages*

General Terms

Hardware, Design

Keywords

Hardware Description Language; Python; Interfaces;

1. INTRODUCTION

Technological advances over the last few decades have enabled us to build extremely powerful digital systems. However, Verilog and VHDL, which were developed in the 1980's, are still the dominant HDLs. These languages only have basic parameterization features. Since chips are getting bigger and FPGAs are becoming more commonly used, many projects need to resort to ad-hoc methods such as shell scripts to work around the parameterization limitations. Additionally, Verilog and VHDL lack modern abstraction features such as classes. This causes code bases to contain large amounts of repeated information, which reduces developer productivity and makes the code harder to maintain.

SystemVerilog[1] solves some of these issues by providing a better type system, parameterization facilities and advanced verification features. However, it has not been widely adopted for hardware design because of non-uniform tool support and uncertainty about which features are synthesizable.

Academic endeavors have examined this issue recently, following one of two basic approaches to the problem. Some projects use the functional units of existing language to call on lower level HDL constructs. Included in this category are JHDL, built on Java, Genesis2, built on Perl, and SysPy, built on Python. JHDL[5] emerged as one of the early attempts in this field to serve hardware developers, striving to combine ease of use and functionality. Genesis2[9] extends SystemVerilog capabilities for connecting HDL to hardware simulation. SysPy is a relatively new Python based HDL undertaking[7].

Other projects choose instead to build a new language for describing hardware. Most notably is the tool Chisel[4], developed out of the University of California, Berkeley[4]. Chisel is built off Scala, though it does not keep the existing design language, instead opting to introduce a new hardware construction language. Chisel has the benefits of data type inference and high parameterization. The predecessor to this, out of the Massachusetts Institute of Technology, is Bluespec[8]. Bluespec adds to the basic functional units by offering flexible parameterization and is also keen to the need for dynamic modification in hardware generation.

MyHDL[6] is another hardware development environment based in Python. Python is a widely used high-level dynamic programming language with a design philosophy that emphasizes concise and readable code. Additionally, Python has a large standard library. These factors make Python a popular choice for creating quick prototypes of algorithms and applications. MyHDL serves several functions within hardware development: modeling, simulation, and verification. We explored methods for abstracting interfaces between components and hardware-software interfaces.

In our model interfaces, we have developed a framework for logically grouping data in models and sharing data structures

FPGA'15, February 22–24, 2015, Monterey, California, USA.

http://dx.doi.org/10.1145/2684746.2689092 .

between hardware and software. Our interface abstraction revolves around newly introduced attribute support that allows for increased code reuse and a reduction in development time. By adding these modifications to MyHDL, rapid prototyping of FPGA applications using MyHDL is facilitated.

2. MYHDL

MyHDL is an open-source package for using Python as a hardware description and verification language. Under the hood, MyHDL uses generators to model concurrency. In Python, a generator is a function that returns an iterator. Generators look like normal functions except for the fact that they use `yield` statements instead of `return`. When a function uses `yield`, it is suspended. When it is resumed, it picks up where it left off. Generators allow MyHDL to efficiently model hardware in Python itself.

Additionally, MyHDL provides decorators to facilitate hardware description. Decorators are syntactic sugar that allow one to easily modify a callable object such as a function, method or a class. For example, this:

```
@decorator
def function():
  ...
```

is equivalent to:

```
function = decorator(function)
```

This allows MyHDL to cleanly isolate procedural code from event driven code. MyHDL provides a simulation framework to run multiple generators simultaneously, which can be used for simulating hardware or high level models.

2.1 Designing hardware

MyHDL supports RTL (Register Transfer Level) modeling by providing two decorators called `@always_comb` and `@always_seq`. The following code blocks, taken from the MyHDL documentation[2] show how one can design hardware using MyHDL. Listing 1 shows the definition of a combinational multiplexer. Note that the direction and width of the ports are not defined. Since Python is duck-typed, they are automatically inferred during elaboration.

```
def mux(z, a, b, sel):
  @always_comb
  def logic():
    if sel == 1:
      z.next = a
    else:
      z.next = b

  return logic
```

Listing 1: Combinational multiplexer

As shown in Listing 2, A design's behavior can be observed quickly by simulating an instance of it with a stimulus.

```
z, a, b, sel = [Signal(0) for i in range(4)]
mux_1 = Mux(z, a, b, sel)

def stimulus():
  print "z a b sel"
  for i in range(8):
      a.next = randrage(8)
      b.next = randrage(8)
      sel.next = randrage(2)
      yield delay(10)
      print "%s %s %s %s" % (z, a, b, sel)

stim = stimulus()
sim = Simulation(mux_1, stim)
sim.run()
```

Listing 2: Running a simulation

The second argument of the `@always_seq` decorator accepts an instance of a specialized subclass called `ResetSignal` It can be initialized as follows:

```
ResetSignal(1, active=0, async=True)
```

The `@always_seq` decorater automatically infers the reset structure based on the `ResetSignal` and the initial values of the signals inside the block.

```
def inc(count, enable, clock, reset):
  @always_seq(clock.posedge, reset=reset)
  def logic():
    if enable:
      count.next = (count + 1)

  return logic
```

Listing 3: Sequential incrementer with enable signal

2.2 Conversion and Cosimulation

MyHDL can convert designs to Verilog or VHDL using the `toVerilog` or `toVHDL` functions respectively. The first argument of the conversion function is the module to be converted, followed by the arguments to instantiate the module.

```
count = Signal(intbv(0)[8:])
enable = Signal(bool(0))
clock  = Signal(bool(0))
reset = ResetSignal(0, active=0, async=True)

toVerilog(Inc, count, enable, clock, reset)
```

Listing 4: Converting a design to Verilog

The conversion function creates a testbench that can be used to cosimulate the design with an external simulator such as iverilog or modelsim through the VPI interface. This involves compiling the converted file, testbench through the eternal simulator and using the Cosimulation class.

3. EXTENDING MYHDL

The aim of this work is to simplify the hardware design process by enabling higher abstractions, and reducing boilerplate code. In modern programming languages, classes are used to logically group data and algorithms. This allows programmers to use powerful features such as class inheritance

to facilitate code reuse. Current HDLs lack such abstraction capabilites.

MyHDL is a very powerful tool for concisely designing and testing hardware. However, it did not have support for converting code that used attributes. Therefore, it was not possible to use class instances to group together signals. Since this problem prevented using interfaces in any convertible code, we set out to fix it.

3.1 Attribute Conversion

When MyHDL receives a design to convert, it lets the Python interpreter handle the elaboration while using the Python profiler to infer design structure and name specs. It then extracts all relevant generators and their symbol dictionaries for further analysis and compilation. The converter analyzes the Abstract Syntax Trees (AST) of all these generators and builds the resulting HDL source code. We analyzed MyHDL's source code and discovered that the AST parsing logic assumed that all objects were either signals or integer bit-vectors. This made adding arbitrary attribute access support tricky. In order to prevent increasing the complexity of the conversion logic and introducing new bugs, we decided to apply a general transformation to the input code and leave the subsequent machinery untouched. We created an AST transformation function that smartly flattens attribute references and modified the MyHDL conversion source code to make all generators pass through our transformation function before proceeding to the main AST Parser. Additionally, we modified the hierarchy extraction code to find all generators that used attribute references and update the symbol dictionaries accordingly. Finally, the signal names are explanded by replacing '.' with '_' and ensuring that there are no name collisions. Name conflicts are automatically resolved by padding the expanded name with additional underscores. We also modified the conversion code to detect attribute accesses and create necessary ports in module declarations and instantiations.

This feature is similar to VHDL records and SystemVerilog interfaces. However, VHDL records are limited since all the signals in a record need to be of the same direction, and System Verilog interfaces are not uniformly supported by various synthesis tools. MyHDL conversion expands attribute accesses to plain names in the target HDL and maps them to the underlying signal. Therefore, classes can be used to create powerful abstractions without worrying about synthesis restrictions.

3.2 Utility Library

We have also developed a library to simplify using MyHDL for hardware design[1]. MyHDL's Cosimulation class is cumbersome to use because it requires you to supply the vpi file and map every signal to an object manually. Our library provides a uniform conversion and simulation API, which allows you to transparently use any simulation backend, including MyHDL itself. Additionally, it also provides a command line utility that automatically finds installed simulators downloads and compiles the corresponding Cosimulation VPIs. This is particularly useful for automated tests. The library also contains helper functions to simplify common tasks like clock generation.

[1] https://github.com/jck/uhdl

3.3 Interfaces

FPGA applications typically use highly parametrized interfaces to wire components together. This is especially true in the early stages of project development, when the requirements are less clear. One such example is the Avalon Streaming[3] (Avalon-ST) interface, which is widely used on Altera FPGAs. Avalon-ST is a low latency and high throughput bus designed for various use cases such as multiplexed streams, packets, and DSP data.

Adding attribute conversion support to MyHDL opens up a lot of opportunities for raising the abstraction level. Python classes can be used to create parametrized interfaces. Listing 5 demonstrates a parametrized interface with run time correctness checks and dynamic portwidths.

```
class AvalonST(object):
  def __init__(self, symbolwidth=8, symbolsPerBeat=1):
    if symbolwidth not in range(1, 513):
      raise ValueError("symbolwidth must be between 1-512")

    if symbols_per_beat not in range(1,33):
      raise ValueError("symbolsPerBeat must be between 1-32")

    self.valid = Sig(False)
    self.ready = Sig(False)
    self.data = Sig(symbolwidth*symbols_per_beat)
```

Listing 5: Simple parametrization example

Advanced features, such as class inheritance, can be used to create complex interfaces while reusing code, particularly since elaboration is pure Python. Listing 6 demonstrats how class inheritance can be used to extend existing interfaces and provide additional features.

```
class AvalonSTPkts(AvalonST):
  def __init__(self, symbolwidth=8, symbolsPerBeat=1):
    super(AvalonSTPkts, self).__init__(width, symbolwidth)
    self.startofpacket = Sig(False)
    self.endofpacket = Sig(False0)

    if symbols_per_beat > 1:
      self.empty = Sig(max=symbolsPerBeat)
```

Listing 6: Advanced parametrization using inheritance

Interfaces can be used as ports in MyHDL. This greatly reduces the amount of boilerplate required to declare a module.

```
def some_module(clk, rst, asi, aso, amm):
  ..
  @always_seq(clk, rst)
  def logic():
    ..
    aso.data.next = asi.data
    ..
```

Listing 7: Module declarations using interfaces

Similarly, instantiation of modules is also simplified. This is very useful when writing testbenches. Interface classes can also define test bench methods to simplify common tasks such as sending or receiving data from buses.

```
..
..
asi = AvalonST()
aso = AvalonST()
amm = AvalonMM()
some_inst = some_module(clk, rst, asi, aso
                                        , amm)
```

Listing 8: Module instantation using interfaces

As shown in listings 7 and 8, interface abstractions greatly reduce the amount of redundant information spread across various hardware modules and testbenches.

3.4 Models

HW/SW applications for FPGAs typically need to share data structures across hardware and software code. For example, register and bit field definitions need to be duplicated in the RTL models, driver and testbench code.

To facilitate sharing of data structures across hardware and software, we developed the Models system that uses Python Metaclasses to represent any structured data. Metaclasses are classes whose instances are classes. Our Models system was inspired by the Django web framework that provides a similar system to uniformly access database fields irrespective of the backend.

```
class SomeModel(Model):
  a = bits(4)
  b = bits(4)
```

Listing 9: Defining a Model

Listing 9 shows the definition Model named SomeModel with fields a and b. Models remember the order which that fields were defined in. This information is used by factory functions to transform the Model. Examples of useful factory functions include bitfields and ctypes. Bitfields can be used by logic to parse structured data from memory. Models converted to ctypes are useful in HW/SW systems, such as an FPGA connected to a host via PCIe, and there is a C api to communicate with the FPGA device. In such cases, the Host software could monitor status registers on the FPGA and write to the control registers. Additionally, the Model API can also be used to allow software programs to seamlessly interact with RTL simulations.

4. CONCLUSIONS

MyHDL is already a powerful language for hardware development. However, since MyHDL could not handle attribute conversion, the extent of abstraction was limited. We added attribute conversion and then explored the use of class inheritance and metaclasses for abstraction. We demonstrated that classes can be used to create reusable IP cores, and Models can be used to share data structures across hardware and software. These advanced abstraction features allow developers to write concise, maintainable code while reducing the development time.

5. ACKNOWLEDGEMENTS

We would like to thank Jan Decaluwe and Christopher Felton for providing valuable feedback and guidance during development, and the MyHDL community for testing and submitting bug reports.

6. REFERENCES

[1] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pages 1–1315, Feb 2013.

[2] The MyHDL Manual. `http://docs.myhdl.org/en/latest/manual/index.html`, 2014.

[3] Altera. Avalon interface specifications. `http://www.altera.com/literature/manual/mnl_avalon_spec.pdf`, 2014.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, DAC '12, page 1216, New York, New York, USA, 2012. ACM Press.

[5] P. Bellows and B. Hutchings. JHDL-an HDL for reconfigurable systems. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, pages 175–184. IEEE Comput. Soc, Apr. 1998.

[6] J. Decaluwe. MyHDL: A Python-based Hardware Description Language. *Linux journal*, 2004(127):5, Nov. 2004.

[7] E. Logaras and E. S. Manolakos. SysPy: using Python for processor-centric SoC design. In *2010 17th IEEE International Conference on Electronics, Circuits and Systems*, pages 762–765. IEEE, Dec. 2010.

[8] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, pages 69–70. IEEE, June 2004.

[9] O. Shacham, S. Richardson, S. Galal, S. Sankaranarayanan, M. Wachs, J. Brunhaver, A. Vassiliev, M. Horowitz, A. Danowitz, and W. Qadeer. Avoiding game over. In *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, DAC '12, page 623, New York, New York, USA, 2012. ACM Press.