Python Wraps Yosys for Rapid Open-Source EDA Application Development

Benedikt Tutzer*, Christian Krieg*, Clifford Wolf† and Axel Jantsch*
*Institute of Computer Technology, TU Wien, Vienna, Austria †SymbioticEDA, Vienna, Austria
christian.krieg@alumni.tuwien.ac.at, clifford@symbioticeda.com, {first.last}@tuwien.ac.at

***Abstract*—Yosys is an open-source synthesis and verification tool which natively supports Verilog-2005 and offers a variety of passes to allow a user to adapt the design and verification flow. A user can extend the functionality of Yosys with a plugin interface in C++. C++ is a powerful language, and it is easy to lose track in fixing low-level issues like memory allocation, instead of focusing on the original problem. This can significantly increase the length of the compile/debug cycle. Python, on the other hand, is a dynamically-typed, garbage-collected language which takes away low-level management from the user, who can focus on the original problem, and therefore keep the debug cycle short. Python is simple and easy to learn. We therefore propose *pyosys*, a script that generates *Boost.Python* wrappers around the C++ implementation of Yosys. These wrappers maintain seamless interoperability between C++ and Python. *pyosys* makes Python's success accessible to Yosys: rapid application development. With *pyosys*, a user can interactively use Yosys in a Python session, with direct access to the Yosys data structures (e.g., design, modules, cells, wires) and methods (e.g., run_pass, load_plugin, get_selection). In addition, the user can develop passes step-by-step directly in Python, with immediate feedback and debug capabilities. This is significant for scientists who can now test their algorithms using an easy-to-use and maintainable high-level language. The results of their experiments are stored in Python objects, and can be directly processed with data analysis frameworks like *scipy*, machine learning libraries such as *TensorFlow*, and data visualization frameworks like *matplotlib*.**

***Index Terms*—Electronic design automation, Yosys, hardware synthesis**

I. Introduction

Imagine you wish to show the output of the Yosys *stat* pass as a histogram of cell types. You may choose to look for a dedicated tool that generates such a histogram, but find yourself with the need of implementing an algorithm yourself.

Listing 1 shows a Yosys script to generate such a histogram by calling the *cell_stats* pass (Line 8). Its output is visualized in Figure 1.

Because the *cell_stats* pass from Line 8 in Listing 1 does not exist yet in Yosys, a user needs to implement a pass that provides the desired functionality. Yosys offers an infrastructure to implement custom passes with full access to data structures that represent the design in memory. These custom passes are built using the C++ programming language. C++ provides great flexibility, detailed control and high performance but comes with initial overhead and high design, debug and verification effort. Python is an alternative since it provides a different trade-off, reducing complexity and therefore enabling fast-paced pass development. Table I lists differences between C++ and Python.

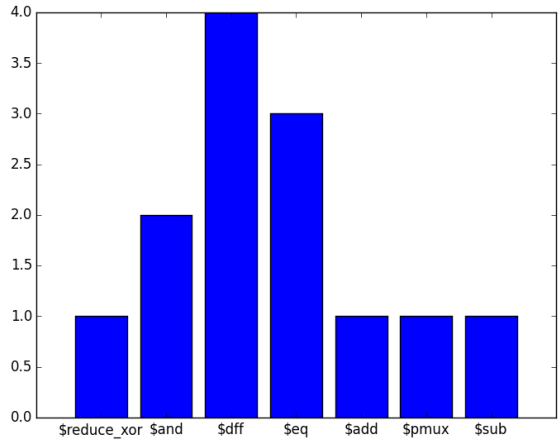

Fig. 1: Output of the *cell_stats* pass from Listing 1 (called in Line 8)

That is why we propose *pyosys*, Python wrappers for the Yosys library. *Pyosys* allows users of Yosys to implement custom passes using Python instead of C++, enabling fast-paced development. It wraps around existing Yosys structures and methods to expose most of the functionality of Yosys to Python. Users may also choose to use Python as their main interface to Yosys, using Python scripts instead of the Yosys command line.

The proposed Python library enables the user to access all of the data structures used by Yosys to represent hardware designs. The user may work on these structures using well-

Listing 1: Generate histogram using Yosys

```
1 # Read and process design
2 read_verilog fiedler-cooley.v
3 prep
4 opt -full
5
6 # Visualize numbers of cell types as
      histogram
7 plugin -i cell_stats
8 cell_stats
```


This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

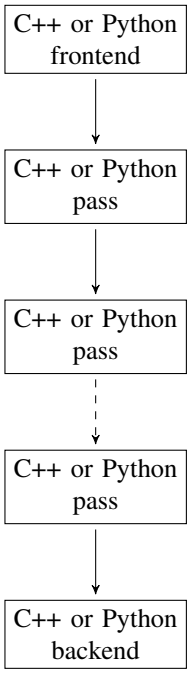

Fig. 2: Flow into which *pyosys* is embedded; Figure adapted from Figure 3.1 in [1]

known Python control flow structures such as loops, conditions, classes and functions, as well as built-in functionalities of Yosys. The user does not need to convert Yosys and Python data structures. This happens at library-level and all data is kept in sync with Yosys automatically.

Listings 4 and 5 show the simplicity of the usage of *pyosys*. Minimal effort is required to interact with hardware designs. An abstract design flow using Yosys with Python passes is visualized in Figure 2.

II. Implementation

This section discusses implementation details of *pyosys* library. The goal of *pyosys* is to keep the user away from C++ code, so that the resulting library can be used with Python. To accomplish this goal, the library captures all communication between Python and C++.

C++ and Python differ in most aspects that define a programming language, as seen in Table I. These differences make it difficult to interchange data between the two languages and make them interact with each other. The Boost C++ library [2]

TABLE I: C++ vs. Python

C++	Python
No garbage collection	Garbage collection
Compiled to machine code	Interpreted by virtual machine
Variable scopes limited to blocks they are declared in	Variables accessible outside of block they are declared in
Strictly typed	Loosely typing
Static typing	Dynamic typing

Listing 2: Including Python module in C++

```
1 PyInitialize();
2 PyObject *module_p =
3   PyImport_Import(filename_p);
```

offers the *Boost.Python* module that includes utilities to mitigate these issues.

The *Boost.Python* library includes a Python *virtual machine (VM)* that can be started by a C++ function. This VM can then execute Python code. This allows the integration of Python code inside C++. Listing 2 shows how a Python module is loaded by the Python module of the *Boost.Python* library. The module is then available from C++.

This functionality allows us to define Yosys passes in Python.

Python is an interpreted language, so this does not happen during compile-time, but during run-time. This means, that symbols defined in the Python module are not available to the C++ compiler and linker, and can therefore not be called the same way as regular C++ functions. One must either call functions by packing the functions name and parameters into a string and passing it to the virtual machine, or define interfaces in C++ and use them in Python.

If interfaces are given, it is much easier to pack functions, since it does not require to distinguish between C++ and Python methods, as long as they use the same interface. This is the case with Yosys passes. Their interface is well defined and easily exposed to Python. Python classes that use this interface can be used in the same way as C++ classes are used.

The *Boost.Python* module also allows the developer to expose C++ constructs to Python. The developer marks functions, structures and classes with their methods, static functions, member variables and operators using only C++ code. The linker then includes meta information about the marked constructs in the shared object file, which makes it possible for Python to import that shared object file as a Python module.

Since Yosys is under active development and is updated frequently [3], static Python wrappers are not considered to be feasible, since they would need to be updated every time new functions are introduced or their signature is changed. Therefore, we chose to generate the wrapper-file when building the Yosys C++ library. During the regular Yosys build, the build system reads the header files which are defined in a configuration file, and generates Python wrappers, provided that all types which appear in a function's interface are either built into Python by default or are wrapped as well. We introduce new targets, *%.pyh* and *$(PY_WRAPPER_FILE).cc*, in the Yosys Makefile to include the Python wrappers in the Yosys build. First, the compiler's preprocessor removes comments, resolves macros, and reformats the code of the header files. Second, a Python script analyzes the resulting code and generates the wrapper classes. The Makefile-switches to enable Python wrappers are given in Listing 3.

Line 1 triggers the build process to generate the Yosys

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

Listing 3: Makefile switches to install *pyosys*

```
1 ENABLE_LIBYOSYS := 1
2 ENABLE_PYOSYS := 1
3 PYTHON_VERSION := 3.5
4 PYTHON_DESTDIR := /usr/local/lib/Python$(
      PYTHON_VERSION)/dist-packages
```

shared library. We set *ENABLE_PYOSYS* to 1 to make *libyosys* to include the meta information that is required to be importable from Python (Line 2). *PYTHON_VERSION* needs to be set to the Python version for which *libyosys* is generated (Line 3). The build process then generates the file *libyosys.so* that can be either used from C++ or Python. It will only be importable by exactly this version of Python, so the user needs to make sure to choose the correct version. We thoroughly tested Python version 3.5. *PYTHON_DESTDIR* in Line 4 is the directory to which we install the Python module. This variable adapts to the chosen Python version and does not need to be changed on default Python installs.

After building, the user can load the *pyosys* module in Python. The module can be used to work with Yosys interactively in a Python shell, or to implement passes in Python and use them from the Yosys shell.

A typical use case would be to prototype some functionality in an interactive Python shell, and once the user is happy with the result, it can be moved to a Yosys pass. We demonstrate *pysosys* in Listing 4, which implements the generation of cell-type histograms as given in Figure 1 and Listing 1. In Line 1, the *pyosys* module is loaded. A new Design object is created in Line 5 and a Verilog file is parsed in Line 6. In Lines 7 to 8, the design is processed (high-level-synthesized and optimized). Afterwards, a Python *for* loop is used to iterate over all selected modules in the design (Line 12) and then another loop iterates over the selected cells of of every selected module (Line 13), and the types of the cells are counted (Lines 14 to 17). Lines 20 to 21 take advantage of the Python library *matplotlib* [4] to plot the resulting statistics.

Listing 5 implements the functionality of Listing 4 as a Yosys pass in Python. The pass is derived from the *Pass* class as in Line 4 and at least the constructor *__init__* and the methods *py_help* and *py_execute* need to be defined as in Line 6, Line 10 and Line 13 respectively. The interface is very similar to the C++ Pass interface, but all objects are the wrapped Python versions of the corresponding C++ objects.

Passes defined as Python classes can be loaded into Yosys just as a compiled C++ pass, using either the *load_plugin* function when using Yosys as a library, or the *-m* switch when invoking Yosys from the command line, or the *plugin* command when using the Yosys shell or script (as given in Line 8 of Listing 1).

Listing 4: Interactively use *pysoys* to create a histogram of cell types as shown in Figure 1

```
1 from pyosys import libyosys as ys
2 import matplotlib.pyplot as plt
3
4 # Read and process design
5 ys.design = Design()
6 ys.run_pass("read_verilog fiedler-cooley.v
      ", design)
7 ys.run_pass("prep", design)
8 ys.run_pass("opt -full", design)
9
10 # Calculate histogram of cell types
11 cell_stats = {}
12 for module in design.
       selected_whole_modules_warn():
13   for cell in module.selected_cells():
14     if cell.type.str() in cell_stats:
15       cell_stats[cell.type.str()] += 1
16     else:
17       cell_stats[cell.type.str()] = 1
18
19 # Visualize histogram
20 plt.bar(range(len(cell_stats)), height =
       list(cell_stats.values()),align='center
       ')
21 plt.xticks(range(len(cell_stats)), list(
       cell_stats.keys()))
22 plt.show()
```

III. Conclusion

pyosys allows to process hardware designs with Yosys in multiple ways:

1) Traditionally, using the Yosys command line, and C++ plugins
2) Interactively, using Python, C++ plugins, and Python plugins
3) Using the Yosys command line, C++, and Python plugins
4) Using Python modules to share functionality between Python plugins and scripts

Three new approaches enable fast-paced Yosys pass development. The user can use interactive Python to develop a pass, using all debug mechanisms Python has to offer, and later use the created Python code to define a pass which can then be used in Yosys just as a C++ pass. The flexibility of Python outweighs the drawbacks it has compared to C++, namely worse performance and less control.

IV. Related Work

We evaluate the importance of *pyosys* by studying the availability of Python interoperability in available *electronic design automation (EDA)* tools of the following vendors:

- Xilinx
- Cadence
- Altera
- Mentor
- Synopsys

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

Listing 5: Python pass that generates a histogram of cell types as shown in Figure 1

```
from pyosys import libyosys as ys
import matplotlib.pyplot as plt

class CellStatsPass(ys.Pass):

  def __init__(self):
    super().__init__("cell_stats",
        "Shows cell stats as plot")

  def py_help(self):
    log("This pass uses the matplotlib
        library to display cell stats\n")

  def py_execute(self, args, design):
    log_header(design, "Plotting cell
        stats\n")
    cell_stats = {}
    for module in design.
        selected_whole_modules_warn():
      for cell in module.selected_cells():
        if cell.type.str() in cell_stats:
          cell_stats[cell.type.str()] += 1
        else:
          cell_stats[cell.type.str()] = 1
    plt.bar(range(len(cell_stats)), height
        = list(cell_stats.values()),align=
        'center')
    plt.xticks(range(len(cell_stats)),
        list(cell_stats.keys()))
    plt.show()

  def py_clear_flags(self):
    log("Clear Flags - CellStatsPass\n")

p = CellStatsPass()
```

None of the above vendors provides support for Python interoperability on design level as *pyosys*. They each provide a *Tool Command Language (TCL)* interface, which is easy to implement using Python and pipes, but that interface is designed as an alternative to the respective *Graphical User Interface (GUI)*, but not to interact with the design on structural level. The TCL language, as it's name suggests, only supplies an interface on command level. It is used to automate synthesis jobs from the command line. *Pyosys* on the other hand allows the user to access the design on the structural level and on command level.

For Cadence, there is a third-party Python module [5] that allows communication with their SKILL core. SKILL is mostly used to control Cadence, but it can also interact with the design. All objects (wires, pins, labels, shapes, ...) are stored in a database which can be queried and edited using the SKILL language. The Python module uses pipes to communicate with SKILL, so it is completely string-based and does not allow for object oriented synthesis as *pyosys* does. This results in a tedious user experience.

RapidWright [6] is a Java framework that can perform netlist manipulation in a similar manner to *pyosys*. During the various synthesis steps of Vivado, the synthesis tool for Xilinx chips, e.g. between *synth_design* and *opt_design*, *Design Checkpoints (DCPs)* are generated which can then be read by the *RapidWright* framework. *RapidWright* is then used to manipulate the design and to write back a DCP that is then read by the next synthesis step. *RapidWright*'s main goal is to optimize digital designs towards more speed. In [6], Lavin et al. claim to have reached 50% *Quality of Result (QoR)* improvements and faster compilation time by using *RapidWright*. The *RapidWright* interface claims to be *Free Open Source Software (FOSS)* and is released under the Apache license, but it's main functionalities are hidden in compiled java archives whose source code is not accessible to the general public. Also, Vivado is necessary to utilize it, which is neither open source nor free, making it an invalid alternative to Yosys for FOSS enthusiasts.

References

[1] C. Wolf, *Yosys manual*, http://www.clifford.at/yosys/files/yosys_manual.pdf, [Online; accessed 2019-01-15], 2013.

[2] *Boost c++ libraries*, https://www.boost.org/, [Online; accessed 2018-11-27].

[3] *Code frequency for yosys*, https://github.com/YosysHQ/yosys/graphs/code-frequency, [Online; 2018-11-27].

[4] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55. eprint: https://aip.scitation.org/doi/pdf/10.1109/MCSE.2007.55. [Online]. Available: https://aip.scitation.org/doi/abs/10.1109/MCSE.2007.55.

[5] V. Borisov, *Cadenceskill-python*, https://github.com/unnir/CadenceSKILL-Python, 2016.

[6] R. Hale and B. Hutchings, "Enabling low impact, rapid debug for highly utilized fpga designs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2018, pp. 81–813. DOI: 10.1109/FPL.2018.00022.