APPENDIX 1: DESIGN AND IMPLEMENTATION OF THE SIMPHONY FRAMEWORK

Simphony source code can be found in a GitHub repository at github.com/BYUCamachoLab/simphony. The extended SiEPIC-Tools package for Klayout with Simphony integration can be found at github.com/BYUCamachoLab/SiEPIC-Tools. Additionally, Simphony is available through the Python Package Index (PyPI).

Simphony has three modules: 1) a core library containing the base classes for models and instances from which all user-defined photonic component models are derived; 2) a simulation library which handles operations related to sub-network growth; and 3) a device library of standard components with presimulated S-parameters and device configurations.

Simphony is implemented in Python, and aims to be very lightweight and fast to install. As such, it only requires dependencies needed to provide basic functionality. For example, our package doesn't provide built-in functions for plotting simulation results; instead, it returns the raw data as NumPy arrays, allowing the user to choose their own visualization package.

A. The Simphony Core

One of the main objectives of the Simphony package is to provide an open-source, concise, independent, and fast simulation tool. By concise, we mean that scripting a simulation does not require many lines of code, imports, or the instantiation of many objects. Independent means that it has very few dependencies and it doesn't force the user into a specific user interface or proprietary (and hence incompatible) code syntax in order to perform simulations. Fast means not recomputing what's already been simulated, which is accomplished by caching the calculations performed on each component.

The *core* library contains the base classes required for any simulation. The most basic building block of a Simphony simulation is an object called *ComponentModel*. It is a representation of a type of photonic component and not of a specific component within a circuit. The task of tracking specific components within a circuit or netlist is left to *ComponentInstance*.

1) Component Models: All photonic component models, whether scripted on-the-fly or provided by a device library, are derived from the *ComponentModel* class. We use the convention of naming each component model with a human-readable name prefixed with the name of the library. For example, Simphony includes a device library created by SiEPIC and the University of British Columbia (UBC) [**?**]. Each of the components in this device library are prefixed with *ebeam_*. Simphony also includes a library created in-house by our research group, and its components are prefixed with *sipann_*.

```
class ComponentModel:
    @classproperty
    def component_type(cls):
        # Returns class name

    ports = 0
    s_parameters = None
    cachable = False

    @classmethod
    def get_s_parameters(cls, **kwargs):
        # Implementation (return preloaded or
    calculated values)
```

Listing 1: The ComponentModel class.

The Simphony core requires models to be registered before being used within the program. This is done by adding the Python decorator *@register_component_model* before the class definition. This allows the program to perform some elementary error checking to verify that user-implemented classes contain all the required attributes and are not malformed. Because Python is interpreted and not compiled, errors are only encountered at runtime when you try to perform an illegal operation. By checking the models before they are used, we ensure a successful simulation later without deeply layered and confusing tracebacks.

Since model names are required to be registered and unique, we can use its name as an identifier at simulation time. In order to avoid repeating calculations, Simphony caches the calculated S-parameters for all the models found in a netlist at simulation time. It does this by keeping a dictionary of model names to S-parameters. Instead of generating a unique identifier for each model, we made the design choice to simply use the class name itself for unique identification.

```
@register_component_model
class YBranchCoupler(ComponentModel):
    ports = 3
    s_parameters = # Some preloaded tuple
    cachable = True
```

Listing 2: A cachable y-branch coupler model.

```
@register_component_model
class Waveguide(ComponentModel):
    ports = 2
    def s_parameters(self, length, width, thickness,
     **kwargs):
        # return some calculation based on
    parameters
    cachable = False
```

Listing 3: A non-cachable waveguide model.

In order to create your own model based on *ComponentModel*, there are some required attributes that need to be overridden. First, *ports* (default 0) must be assigned some positive integer value. The *s_parameters* attribute can either be a tuple of S-parameters of the form *([frequency_array], [S-parameter_matrix])* or a callable function that calculates and returns a tuple of the same form based on some arbitrary list of keyword arguments. The S-parameters could be hard-coded, loaded from a file, or calculated by a function. The way Simphony treats the *s_parameters* attribute depends on whether *cachable* is set to true. If cachable is true, it signifies that the component doesn't rely on any other parameters to calculate its S-parameters and the loaded values already define the component's response. If cachable is false, the program assumes that calculation of the S-parameters depends on other arguments which are stored at the instance level. For example, a waveguide's S-parameters are dependent on its length and path. In contrast, a y-branch coupler's S-parameters are considered unchanging since it doesn't have parameters to vary and its physical layout is fixed.

```
class ComponentInstance:
    def __init__(self, model: ComponentModel=None,
    nets: List=None, extras: Dict=None):
        # Store values internally

    def get_s_parameters(self):
        return self.model.get_s_parameters(**self.
    extras)
```

Listing 4: A cachable RingResonator model.

Just as a *ComponentModel* is the building block of a simulation, *ComponentInstance* objects, or "instances," are the building blocks of a circuit. Each instance must reference a ComponentModel and represents a physical manifestation of some photonic component in the circuit. For example, a waveguide has a single ComponentModel which specifies its attributes and S-parameters. However, a circuit may have many waveguides in it, each of differing length; these are therefore instances of the waveguide. Any number of instances for a certain photonic component will reference the same ComponentModel, thereby obtaining its S-parameters from a "single source of truth."

One major difference between models and instances is that to build a simulation, instance objects need to be instantiated whereas models are simply defined. Instances store three attributes. The first, *model*, stores a reference to a ComponentModel from which S-parameters can be obtained. The second, *nets*, can optionally be defined upon construction of an instance or it can be left for the netlist object (discussed later) to assign automatically. The functionality allowing the declaration of nets upon instantiation was built in to allow other circuit layout programs, which may already implement netlist generation routines, to be plugged in. The final attribute, *extras*, is an optional dictionary containing parameters that may need to be passed on to models where scattering parameter calculations depend on other variables (e.g. waveguide lengths and bend radii).

Simphony is layout-agnostic; it doesn't care where components are physically located. However, the "extras" attribute allows other programs to store within the instance itself useful key-value pairs, including (but not limited to) layout positioning information. This contributes to its flexibility when used in conjunction with other programs. For example, Lumerical's INTERCONNECT, a schematic-driven commercial simulation software, ignores layout and only requires components and connections when simulate a circuit. On the other hand, KLayout with SiEPIC-Tools, created by SiEPIC and UBC, implements a layout-driven approach to designing photonic circuits, exporting locations with the components when generating a netlist [**?**]. Because Simphony can optionally store any information with its components, including layout information, it can act in either capacity.

3) Netlist: Simphony implements a simple *Netlist* class. A netlist simply contains a list of component instances and has routines used to assign net numbers to connections between components. However, the netlist doesn't maintain the list of connections internally; instead, as it parses connections, it generates net numbers and stores those values within each instance's *nets* attribute. The netlist maintains a list of the components in a circuit, each of which independently maintain a list of their nets.

4) Defining circuit connections in Simphony: There are two methods for defining circuits when scripting simulations in Simphony. The two methods are actually very similar, and the method of choice is simply a matter of preference in how the connections are formatted or typed. The first method explicitly states which ports of each component are connected together and is represented as a list of lists. The second method stores components and ports in separate lists and then zips them together.

```
conn = []
conn.append([device_1, port_a0, device_2, port_b0])
conn.append([device_1, port_a1, device_2, port_b1])
```

Listing 5: Creating an MZI using a list of connections

```
c1 = [ device_1,  device_1]
p1 = [  port_a0,   port_a1]
c2 = [ device_2,  device_2]
p2 = [  port_b0,   port_b1]
conn = zip(c1, p1, c2, p2)
```

Listing 6: Creating an MZI using "zip"

The above two code listings are equivalent. The method of choice is simply the preference of the user or the most understandable way for the computer to algorithmically define connections within the circuit.

B. Simulation

The Simulation class performs the basic sub-network growth matrix operations required to simulate a photonic circuit. While Simphony's simulation module contains various different kinds of simulations, all simulations extend this base Simulation class. Simulation performs interpolation on the S-parameters for all cachable components and caches the results. Sub-network growth then cascades all of the instances' matrices into one matrix.

Simphony presently includes three different simulators. The first is a single-mode, single-input simulation that returns transmission parameters from any input port to any output port (including back reflections). The second is a single-mode, multi-input simulation that allows you to sweep the wavelength of multiple input ports simultaneously and see the response at any output port. Finally, a monte carlo simulator that randomly varies waveguide widths and thicknesses, calculating S-parameters by interpolating the effective index of waveguides, allows us to mimic fabrication variation. Other simulators can easily be written, extending the functionality of the base *Simulation* class.

C. DeviceLibrary

Simphony includes two basic device libraries. The first was created by SiEPIC and UBC for their Process Design Kit (PDK) used with KLayout. It correlates with the physical component model layouts found in SiEPIC-Tools for KLayout and the S-parameters are the result of FDTD simulations for ideal components. The second library includes a few components designed by our research group with S-parameters

generated using machine learning techniques. This allows for imperfections from manufacturing to be easily simulated or for new components to be quickly designed.

Beyond the default device libraries, it is very easy to define your own collection of component models. Each library, which is just a collection of class definitions, is formatted as a Python module with all the models it provides defined in *__init__.py*. Instead of hardcoding values, our cachable models simply load their S-parameters from a NumPy file (.npz) each time the library is imported. End-users can easily create their own Python modules to be used in simulation by subclassing base components provided by Simphony. This is another way in which Simphony's capabilities can be easily extended. It also allows simulation results, scripts, and circuits to easily be shared between collaborators or computers, since the entire system is cross-platform, non-proprietary, and the only libraries required are the modules implementing the components used in the circuit and the script defining the circuit.