Journal of Computational Science 4 (2013) 313–324

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Improving the design cycle for nanophotonic components

Martin Fiers*, Emmanuel Lambert, Shibnath Pathak, Bjorn Maes, Peter Bienstman, Wim Bogaerts, Pieter Dumon

Photonics Research Group (INTEC), Ghent University – IMEC, Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium

ARTICLE INFO

Article history:
Received 24 February 2011
Received in revised form 17 February 2013
Accepted 17 May 2013
Available online 5 June 2013

Keywords:
Nanophotonics
Designing and modeling optical components
Optical circuit design
Parametrized cell
Python

ABSTRACT

We present IPKISS, a software framework that greatly simplifies the design of nanophotonic components. In this approach, all steps in the workflow are based on a single high-level definition of the component, in a Python script. Because there is only one description, the design flow becomes less error prone due to incorrect definitions, and the overall reproducibility is greatly improved.

Furthermore it enables easy closed-loop modeling of components and circuits. Also, previous work can easily be built upon because lower level blocks can seamlessly be replaced by new blocks. While we illustrate the application in photonics, this software and the used design patterns can be extended to other domains such as RF design and to multidomain physics such as opto-electronics.

1. Introduction

In a typical research or design environment, fabrication of micro-and nanoscale devices is an expensive process with long turnaround times. Prior to submitting a design for fabrication, these devices are typically modeled and simulated in software. For example, in the field of nanophotonics, electromagnetic simulations are used to calculate how light propagates through such a device. Often it is also required to perform tolerance analysis on the design parameters as well as on effects of the fabrication process. One major difficulty that arises when designing these devices is that the different simulation tools have their own user interface and moreover have their own representation to define components. Defining these devices in different tools is a laborious job, and there is a considerable risk of introducing errors in the specification of the device in each tool.

The main characteristic of our approach is that a component is defined only once on a high level [1]. This component is available as a parametrized cell (PCell), a concept originating from the design of electronic circuits. Then, the necessary representations (e.g. a discretized matrix representing the component, a cross-section, a list of polygons, port positions) to drive the different tools (simulation, visualization, routing) are extracted from this definition. The transition to different simulation tools should only be written once in a generic way, which makes simulations much less error prone. It is also much easier to reproduce earlier results and to change sub-parts of the design. In this way, many variations can easily be compared to one another (e.g. a different simulation method, an improved component, or a modified design).

Python is our programming language of choice. The main reason for using this programming language is the flexibility which it offers: it can be used to make very complex software designs, yet it has a relatively low threshold for researchers without extensive programming skills. Our software toolset revolves around a central design framework called IPKISS [1], which can interface with different in-house and third-party simulation tools.

The paper is structured as follows: as the reader might not be familiar with photonics, we very briefly describe this specific research field in Section 2. In Section 3, we illustrate a typical workflow, i.e. the steps needed to design a nanophotonic component. We show which design problems typically arise and demonstrate how the software framework improves this flow. In Section 4, the technical design and implementation of the framework is described, and in the fifth section we illustrate how we use the software to efficiently design a complex optical component: An Arrayed Waveguide Grating. We conclude by providing license information. As previously noted, it is easy to extend this architecture beyond the horizon of photonics: electronic design, multidomain physics and so on. Throughout the paper, we use Python code to explain several core concepts. The code aims to be descriptive rather than to explain all details.

* Corresponding author. Tel.: +32 92643272.
E-mail addresses: martin.fiers@intec.ugent.be, mfiers@gmail.com (M. Fiers), emmanuel.lambert@intec.ugent.be (E. Lambert), shibnath.pathak@intec.ugent.be (S. Pathak), bjorn.maes@umons.ac.be (B. Maes), peter.bienstman@intec.ugent.be (P. Bienstman), wim.bogaerts@intec.ugent.be (W. Bogaerts), pieter.dumon@intec.ugent.be (P. Dumon).

1877-7503/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.jocs.2013.05.008

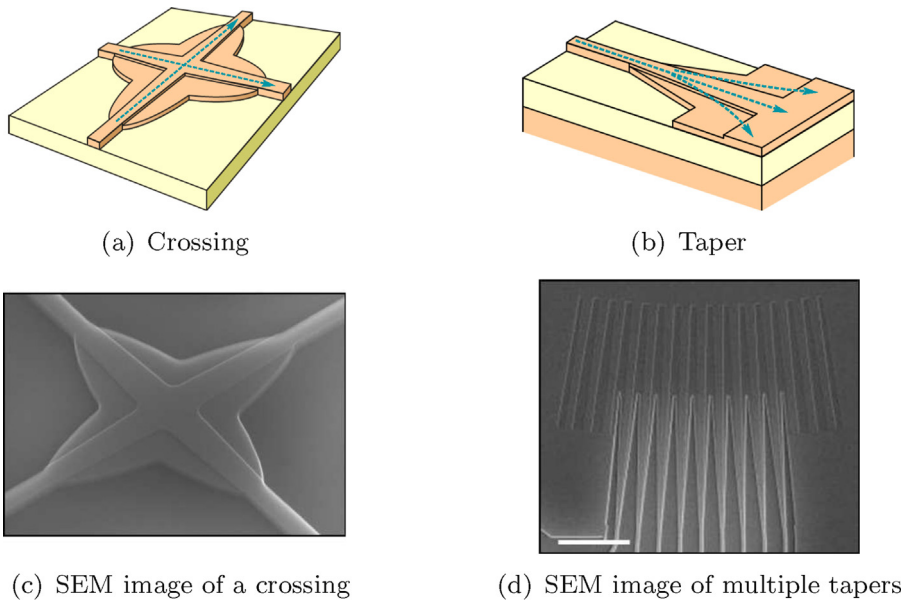


Fig. 1. Some examples of nanophotonic subcomponents, used for designing small integrated optical circuits. Because a nanophotonic circuit is planar, crossings (left) are sometimes needed. Tapers (right) are used to spread light from a narrow waveguide to a broad one. On the bottom, Scanning Electron Microscope (SEM) pictures of the fabricated devices are shown.

2. Photonics

Photonics is the field of manipulating, generating and detecting light (photons) by means of optical components. This is in contrast to electronics, in which electrons are the information carriers. Some examples of photonic devices are: lasers, optical receivers and transmitters, CD/DVD drives and LED lighting. A recent trend in photonics is the drive towards miniaturization of components, and integrating many of them on a single chip. These so-called (nano)photonic integrated circuits have a better performance, are more robust, and consume less power than bulk photonics, low-contrast integrated photonics and electronics. One excellent material for making such optical chips is silicon. Silicon has very low absorption losses in the wavelength range that is used for fibre-optic communications (1300 nm and 1550 nm). Fortunately, silicon is already widely used in electronic chip fabrication, so we can reuse standard Complementary Metal Oxide Semiconductor (CMOS) technology to manufacture photonic chips. In this technology, the silicon on insulator (SOI) wafer is patterned using deep UV lithography [2]. This opens the door to wafer-scale fabrication of nanophotonic chips, leading to devices that can be manufactured in large volumes at low cost.

A few subcomponents of a nanophotonic circuit are displayed in Fig. 1. The resulting device consists of submicrometer wide silicon lines on top of a thick glass layer. Because silicon has a very high index of refraction, the submicron line acts as a waveguide for light: electromagnetic waves with wavelengths between 1.3 μm and 1.55 μm can travel along the line (a so-called "photonic wire") without much loss.

By optimizing the geometry of the silicon, the light can be manipulated. Fig. 1(a, c) shows a crossing of two waveguides, where the geometry is engineered such that there is no *crosstalk* between the waveguides. In Fig. 1(b, d) we change the width of the silicon around the *core* of the waveguide and then stop the waveguide, so that light can diffract in the thin layer of silicon on the chip.

3. Workflow for designing a component

To illustrate the problems associated with a manual workflow (that is, before adopting the framework), as well as the innovation brought by our framework, we will discuss the workflow for designing a typical photonic integrated component: a multimode interferometer (MMI). Although we use an optical component to illustrate the workflow, readers from other research domains might identify the same or similar problems based on their own experience.

In Section 3.1 we briefly introduce this device and its typical design steps. We show that in the classical workflow (3.2) there are a lot of manual interactions, leading to a slow, and more importantly an error-prone workflow. A workflow based on our software (3.3) shows how one can circumvent these problems.

3.1. Example device: MMI

We will illustrate our workflow using a device that splits the light in a waveguide into two equal parts, an important building block in photonic IC design. It is called a multimode interferometer (MMI), and is depicted in Fig. 2. This example is representative to many photonic design problems and is practiced by most photonic designers today, irrespective of the specific tools they use in each step of the problem. The MMI consists of a sequence of waveguide elements of different widths and shapes. Each waveguide supports a number of electromagnetic waveguide modes, i.e. eigensolutions of the light distribution in the dielectric medium.

There are several aspects to modeling this device, which are illustrated in Fig. 2. When exciting the MMI with a mode in an input waveguide, one needs to know the shape (spatial distribution) of this mode, called the mode profile. We calculate this waveguide mode profile using an eigenmode solver (Fig. 2, top left). The mode has a gaussian-like profile, as shown in the figure. The mode profile is then entered as input for a full-wave time-domain simulation to calculate the light propagation in the device (Fig. 2, top right). As three-dimensional (3D) full-wave simulations are computationally very intensive, one will first run an approximate simulation in 2D using well-known approximation methods and only then run full 3D simulations.

In order to get a highly accurate representation of the device characteristics, a 3D simulation is then performed. From this simulation, the scatter matrix is extracted, leading to a high-level description of the building block. In a circuit simulation tool (Fig. 2, bottom right), several of these building blocks are combined, in

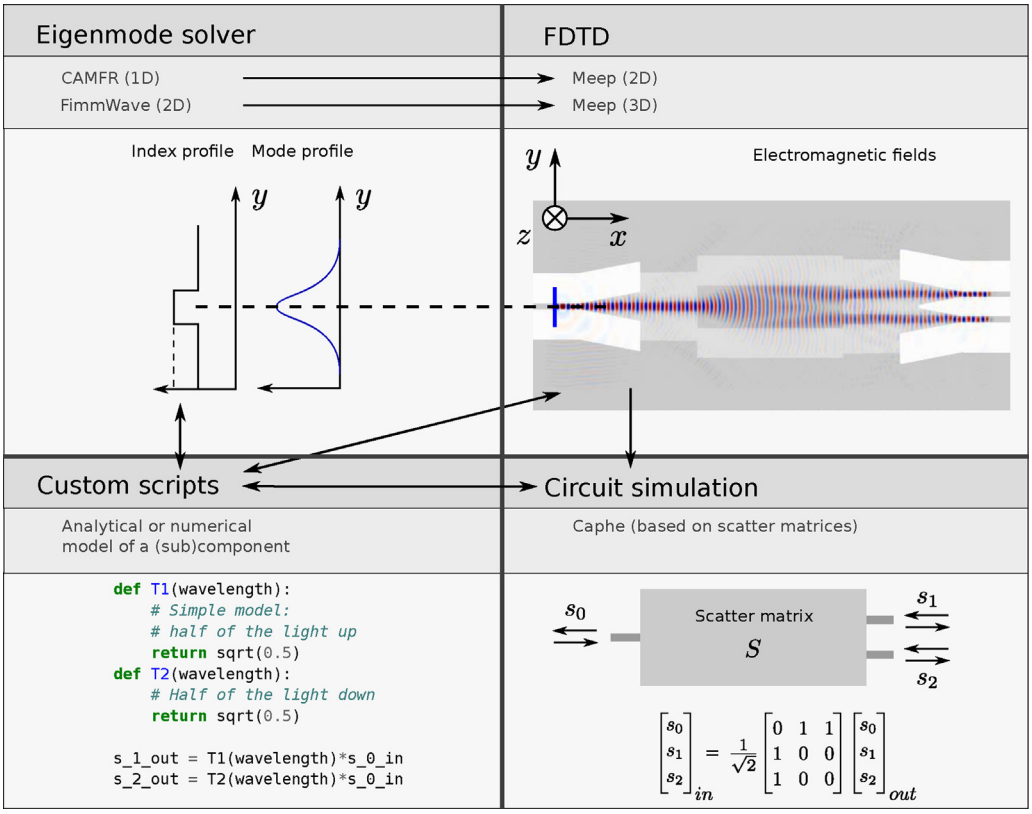


Fig. 2. Combining several simulation tools when designing a nanophotonic light splitter. Top left: an eigenmode solver (e.g. CAMFR [3]) which calculates the mode profile of a waveguide (a gaussian-like shape), which is used as input for a finite difference time domain (FDTD) simulation (e.g. Meep [5]) on the top right. The output of this simulation is then sent to a circuit simulation tool (e.g. Caphe [19], [20]), see bottom right. Also, users might want to perform a part of the simulation using their own code (bottom left) in Python.

order to create functional devices (for example optical filters). These simulations run much faster, and more attention is given to the phenomenological parameters, rather than the physical layout of the actual device.

In addition, most designers will want to integrate a piece of own-written code (for example a numerical algorithm or an analytical model) in the design flow as well (Fig. 2, bottom left).

Below we give a description of the specific simulation tools we have used:

3.1.1. Eigenmode solvers: CAMFR [3] and Fimmwave [6]

CAMFR is written in C++with a Python interface. It is developed at Ghent University and can be downloaded for free. The source is being distributed under a dual license scheme (GPL and proprietary) [4]. The software is used to calculate modes in a one-dimensional approximation of the geometry, for a fast simulation. For accurate simulations, we use the commercial software tool Fimmwave (Photon Design) to simulate the eigenmodes in the actual two-dimensional cross-section.

3.1.2. Full-wave time domain simulation: Meep FDTD (MIT) [5]

Meep is an open-source finite difference time domain simulator. With this tool, the fields are calculated at all positions and at all times. It is used either in 2D approximation or in full 3D. Making a geometry in Meep is usually done using the Scheme programming language. Although very powerful, experience learns that it takes some iterations before the device is represented correctly if it has to be defined manually. For devices with complex geometries, this can be a tedious job. Also, importing modes from an electromagnetic simulator is a laborious and error-prone job.

3.1.3. Circuit simulation: Caphe [19,20]

We use an in-house developed circuit simulator Caphe to simulate optical circuits. The extracted results of the full-wave simulations (for example a scatter matrix S) for each device are imported in Caphe, which can then calculate the time and frequency response of the full circuit.

3.2. Classical workflow

From the example above, it is clear that a lot of manual actions are required to successfully model an optical component, and even more to model a circuit. In the classical workflow, one manually communicates information from one tool to the other tool. One of the key issues here is that it is rather tedious as well as error-prone to exchange information between these tools: the output of different tools usually have different file formats and/or the way this data is loaded into the simulation is different. In some cases, one even has to write different scripts for one tool, for instance for running the 2D and 3D simulations in Meep. Also, data is gathered in different formats and post-processed in various external environments such as Excel and Matlab, which involves a lot of manual data conversion for each tool. Use of the different tools requires learning new graphical and scripting or programming interfaces over and over again.

3.3. New workflow

The new workflow that we developed in the past years and which we use for our own work, is much more automated than the workflow described previously which involved manual importing and exporting of results, and redefining components. It is based on

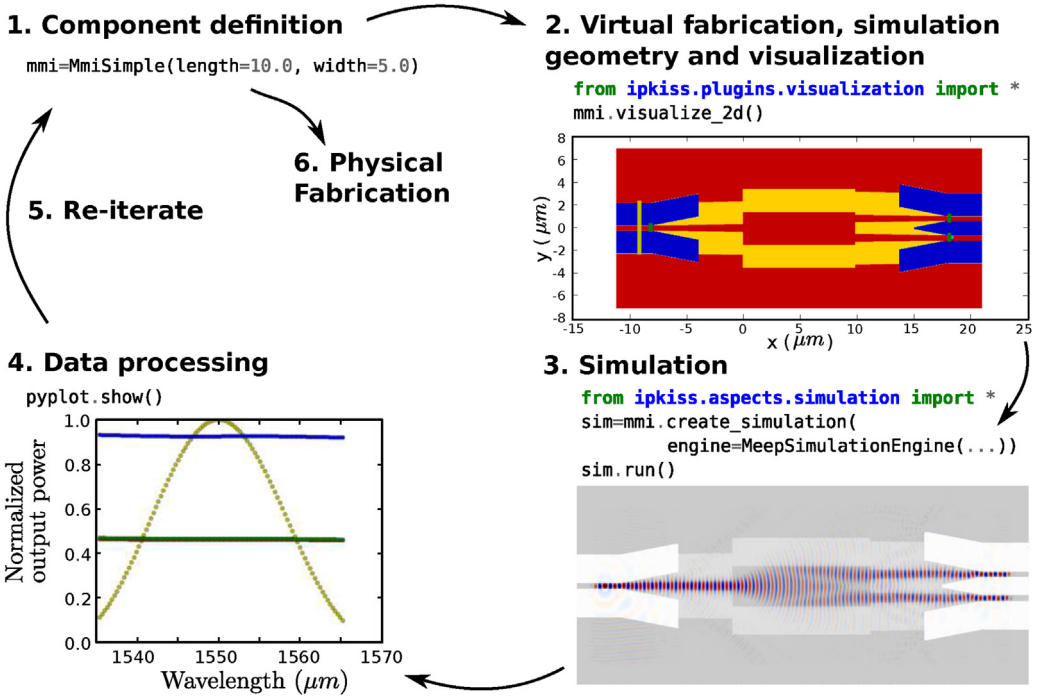


Fig. 3. Integrated workflow for designing a multimode interferometer (MMI). There is only one high-level description of the component, and all other steps extract information from this description. For example, the simulation is performed without additional programming work, so errors when describing a design for different simulation tools are eliminated. The reproducibility is greatly improved, and the designer can focus on iterating through several designs for optimization rather than describing his component manually in different simulation tools.

a single representation of the component in a high-level Python script. The workflow is depicted in Fig. 3 and incorporates the following steps:

1. Component definition
2. Virtual fabrication, generating the simulation geometry and visualization
3. Simulation
4. Data processing
5. Re-iterate previous steps where needed
6. Generate the final layout for fabrication

1. Component definition. Users can write their own library of components on top of IPKISS. The multimode interferometer from Fig. 2 is part of such a library and can be instantiated as following:

Listing 3.3.1. Creation of the component (see also Listing 4.1.1 for the class definition). Keyword arguments are required, to improve readability of the code.

```
mmi = MmiSimple(length=10.0, width=5.0)
```

2. Virtual fabrication, generating the simulation geometry and visualization. The physical description of the component is typically done through a geometric representation of the photomasks used for the fabrication of the devices: the devices are fabricated in semiconductor using a series of lithography and etching steps (see [2]), and at each lithography step a geometric mask is projected onto the material. At design time, the final geometry of the fabricated component is therefore represented as a list of polygons on each mask layer. This can be exported to a number of common image formats for quick inspection. The simulation geometry for electromagnetic simulations is automatically generated from the mask layout. This is done through a virtual fabrication routine which performs an approximation of the actual fabrication process, and operates on the different mask layers defined in the component layout. The resulting geometry is a distribution of materials in a 2D or 3D space. Basic design errors can already be corrected here.

In a second step, this internal representation is then converted to the geometric representation of a specific simulation tool. This is illustrated in detail in Section 4.3. Also, from the component description we can extract the positions of the input and output ports, which can be passed to the simulation tool for the correct excitation and for the correct interpretation.

3. Simulation. A next step involves simulation of the physical behavior of the device. As explained in the previous paragraph, the high-level description of the component is used to extract this geometry. The interfacing between the different tools is done automatically. For example, the mode profile is calculated in CAMFR and then passed on to Meep, without any additional programming work on the user side. As interfacing to different tools plays a major role in this framework, we further elaborate on how the actual interfacing is done in Section 4.3, using CAMFR as example.

Simulation is an important step in the workflow and usually requires a lot of resources. In the next code example we create a simulation object to perform a simulation with Meep. The high-level design can be persisted to a file (`persist`) and then executed (`run`) on a cluster without the user needing to worry about the specific details on how to run a simulation on a cluster. As explained before, the mode profile is calculated by CAMFR and then used as input (`sources`). Detectors are then added (`datacollectors`).

Listing 3.3.2. Defining a simulation for the multimode interferometer (MMI). Simulations can be persisted to a file and then executed

on a simulation cluster.

```
sources = [ModeProfileAtPort(center_wavelength=1550,
                             port=source_port,amplitude=0.1)]
datacollectors = [Fluxplane(port=input_port,
                            name="Flux at input port"),
                  Fluxplane(port=output_port,
                            name="Flux at output port"]
sim = mmi.create_simulation(
                  engine=MeepSimulationEngine(sources=sources,
                                datacollectors=datacollectors,
                                resolution=36))
sim.persist('mysimulation')   # Persist to a file
sim.run()                     # Or, simulate
```

4. Data processing. Using the scientific tools available in NumPy and SciPy, all kinds of post-processing such as curve fitting and parameter extraction can be done seamlessly, because simulation results (e.g. the transmission spectrum of a component) are readily available as Python numpy arrays.

5. Re-iterate. After interpreting the results of the data processing one can change the parameters (e.g. the length in Listing 3.3.1) and repeat the cycle until the desired behavior for the component is reached. In the old workflow, much more manual actions were required before the same set of operations in the workflow were repeated.

6. Physical fabrication. The framework contains a number of export routines to write this geometry to file formats commonly used in semiconductor processing, such as the GDSII format. From this GDSII file, a physical component is made as illustrated in Fig. 4. The Python scripting makes it very easy to incorporate different design variations on the same mask.

4. Design and implementation of the framework

The IPKISS software platform consists of four modules. The first module is the IPKISS engine, which is the core of the software. It is a parametric cell (PCell) engine, which means each component is described by several parameters. Second, using plugins one can interact with the PCell and extend the functionality of each component. In our framework, we added plugins for photonic design, such as the TECH object (explained later). The third module is a component library for photonic design, and the fourth module allows one to interface these PCells to simulation tools.

The proposed software framework is based on the programming language Python. This choice is based on several requirements: It should enable the core software developers to create a sustainable platform with the ability to make a complex, flexible architecture. But on the other hand, the researcher does not want to bother about all technical details of the implementation and wants a clean scripting environment. Furthermore, the ability to integrate different tools in the framework is very important: Some tools are written in C/C++and need to be executed from within the framework. In Python, there are several ways to interface to C/C++, for example using SWIG [7].

4.1. The IPKISS engine

The core of the framework is a parametrized cell (PCell) engine. PCell is a concept widely used in the automated design of electronic circuits. Basically it is a class which is used to represent physical entities such as a transistor, a resistor, an optical component and so on. In our software framework, we call this basic entity a `Structure`. Each structure has some `Properties` that describe the object. This section describes what a `Structure` is, explains how we use `Properties`, and show how we use mixins to add functionality to a PCell.

4.1.1. Structure

This is the PCell object, the basic class on which our framework is based. It allows to check for variable types and supports the mixing in of other classes, for example providing visualization and simulation interfaces to the class. Structure objects are stored in a library and have a unique identifier with which they can be retrieved. Using a caching mechanism, duplicate structures are avoided, which is necessary when a certain structure is repeated many times. Fig. 5 shows how the Structure is defined, and Listing 4.1.1 gives an illustration of how to define a new component based on this `Structure`.

4.1.2. Properties

In Python, variables do not have to be declared with a certain type. This has the drawback that type errors are not caught upon initialization, but much later, when these variables are actually used. E.g. when multiplying two strings, an exception is thrown and a stack trace is displayed. This stack trace can be very intimidating for a novice user. For that reason, we choose to give the user immediate and clear feedback on the validity of the arguments upon initialization of a Pcell.

To achieve this, we make use of Python descriptors [12]. This mechanism allows to define generic objects that control the setting and getting of attributes of other objects (i.e. validating attributes for consistency), with the added advantage that it requires much less code from the user to define the attributes of the class. This is similar to the concept of the `property` built-in in Python, and for clarity we also use the term Property in the IPKISS framework. We implement this as a set of classes which derive from `PropertyDescriptor`, with different behavior and built-in restrictions. The resulting user code is shown in Listing 4.1.1, where the properties require only one line each, including the restrictions we want to apply to them. In Listing 4.1.2 we show how we can override specific properties with new restrictions. Combining restrictions is possible, for example `RestrictType(list)` & `RestrictLength(0,5)` requires the variable to be a list, with a length between 0 and 5. Additionally, a property can be automatically calculated by adding a specific `define_` method to the class. For example, the variable `area` in Listing 4.2.1 is associated automatically with the method `define_area`.

Listing 4.1.1. Making a multimode interferometer (MMI) in IPKISS. `Properties` are used to define parameters of the component, to set restrictions, default values and so on. Furthermore they remove the need for a dedicated `__init__` function and they force the user to use keyword arguments to improve readability.

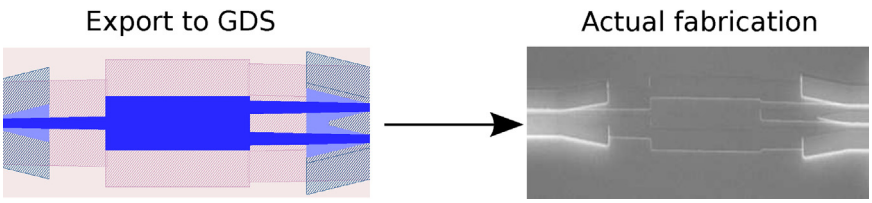


Fig. 4. Fabricating the multimode interferometer. Left: Exporting the design to the GDSII format (a format commonly used in semiconductor processing). Right: A Scanning Electron Microscope (SEM) image of the actual fabricated component.

```
class MMISimple(Structure):
    __name_prefix = "MMISimple"
    length=PositiveNumberProperty(default=10.0)
    width=PositiveNumberProperty(default=5.0)
    area=PositiveNumberProperty(doc="Area under the MMI")
    def define_area(self):
        return self.width*self.height
    def define_layout(self, layout):
        # Construct the MMI using rectangles and triangles
        layout+=Rectangle(center=(0.0,0.0),
                          box_size=(self.width, self.length))
        # add more rectangles, triangles, ...
        ...
        # Return the layout
        return layout
```

Listing 4.1.2. Extra restrictions on a `Property`. The newly created class now has a restriction on its length.

```
class MyMmiSimple(MmiSimple):
    length=RestrictedProperty(default=10.0,
                      restriction=RESTRICT_NUMBER & RestrictRange(3,15))
```

The class `StrongPropertyInitializer` manages these `Properties` (see also Fig. 5). It also causes values to be assigned automatically to the `Properties` of the PCell, effectively removing the need for a dedicated `__init__` function. Also, the consistency of the object can be validated at instantiation. Again, this removes a substantial amount of user code, improving readability.

In parallel to the development of our PCell class, other powerful libraries were developed that allow typing of Python variables, support delegation and initialization of variables. One of these is the Traits library, developed at Enthought [13]. The functionality of this library is very similar to the functionality we provide, and we are even considering of migrating to the Traits library in the future. The first thing to investigate is the scalability of both libraries, because IPKISS is relatively slow when instantiating a lot of different PCells.

4.1.3. Mixins

The PCell engine is enriched with new functionality by mixing additional classes into it. After mixing in a class, the PCell inherits from this class. This is used to add functionality such as generating a representation of the physical layout, visualizing, simulations, and interfacing to external tools, as shown in Fig. 6. There are several reasons to use mixins rather than to inherit all classes explicitly (multiple inheritance). First of all, it reduces the complexity of the PCell class. In this way you do not pollute the userspace with functionalities that will never be used. Second, new modules can simply be plugged in without changing the code base. Third, it is a way to protect intellectual property. Additional functionality can be part of

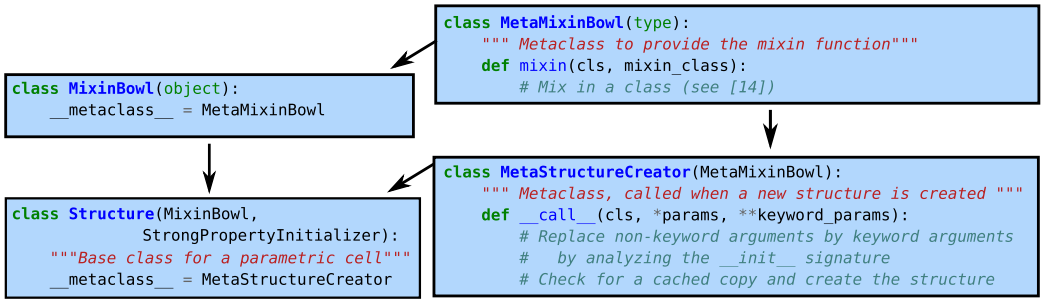


Fig. 5. Structure is the basic entity of the IPKISS framework. Other classes can be mixed in using the `mixin` function. Structure creation is modified using the `MetaStructureCreator`, which can analyze the `__init__` function and can check for a cached version of the created structure. `StrongPropertyInitializer` assigns Properties and checks the consistency of the created object.

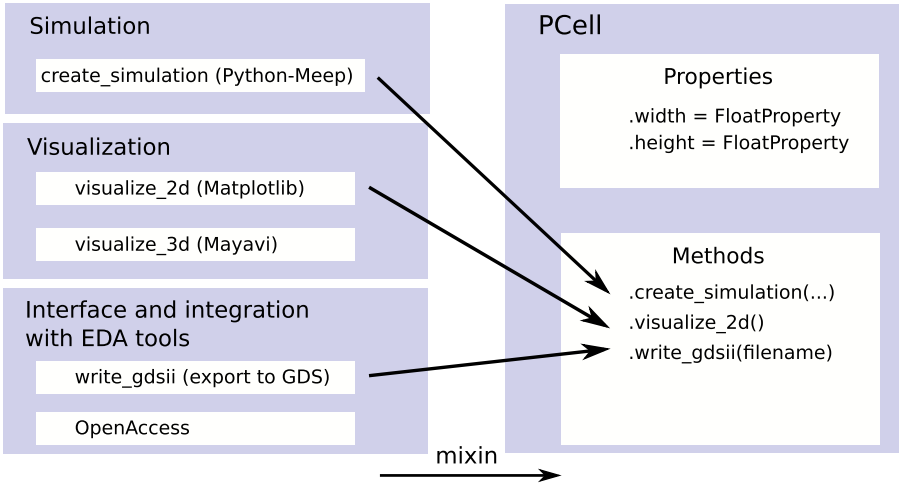


Fig. 6. Functionality can be dynamically mixed into the PCell. For instance, the `visualize_2d` method can be added to the PCell, which generates a 2D representation of the component (shown in Fig. 3), or a GDSII file can be generated with `write_gdsii` (shown in Fig. 4).

a proprietary module and can easily be plugged into the framework if allowed.

Mixins are realized in Python by changing the special attribute `__bases__`, which is a member of the class object. A good introduction to mixins can be found in [14]. While mixins are not a novel concept, they are not often used in software.

To facilitate the use of mixins, we provide a metaclass `MetaMixinBowl` that contains the function mixin, see also the inheritance diagram in Fig. 5.

4.2. Plugins for photonics

4.2.1. Technology

Many of the library components can be defined in a generic way, independent of the actual materials and technology processes used to fabricate it. For instance, the concept of a waveguide is universal, whether the material used is glass, III–V semiconductor or silicon. Therefore, the framework provides the concept of a Technology Tree: This is a collection of settings and predefined objects orthogonal to the component library. By loading the correct technology definition, a global object `TECH` is defined, which provides default arguments and settings for all parametric components. This `TECH` object should be loaded in the first import statement of the executable scripts: this way it will set all default values of function and class attributes automatically to match the technology used.

Listing 4.2.1. Definition of a physical material stack using the global `TECH` object as explained in Section 4.2.1.

```
# a 220 nm thick Silicon layer in IMEC [10] technology
TECH.MATERIAL_STACKS.MSTACK_SOI_SI_220nm = MaterialStack(
                name = "220nm Si",
                materials_heights = [(TECH.MATERIALS.SILICON_OXIDE,2.0),
                                     (TECH.MATERIALS.SILICON,0.220)],
                display_style = DisplayStyle(color = COLOR_RED))
```

The technology concepts allows the user to generate circuit and component designs for different fabrication processes. For example, the silicon photonics Multi-Project Wafer (MPW) service ePIXfab [8,9], which provides access to the fabrication facilities of IMEC [10] and CEA-LETI [11], provides technology trees for IPKISS, as well as a library with basic components.

4.3. Interfacing IPKISS to simulation tools

In order to interface to different simulation tools it was necessary to create several abstract classes in the core of our software framework. We illustrate the concepts with an example of interfacing IPKISS with the eigenmode solver CAMFR, and end this paragraph with a small word about netlists, a feature which allows PCells to be linked to each other to allow circuit simulation.

4.3.1. Interfacing IPKISS to a photonics simulation tool

The physical concepts. The link between the PCell object and the simulation tools is a generic geometric representation of the device, consisting of a distribution of materials in a 2D or 3D coordinate space. Each material has its own physical properties such as refractive index, a temperature coefficient, a stress and strain matrix and so on. Predefined materials are defined by the TECH object, as explained in Section 4.2.1, or the user can supply his own custom materials. Fig. 7 shows that the high level PCell is converted to a 2D distribution of 1D material stacks, which is an efficient way of describing devices made with planar process technologies (such as often used for silicon photonics). We can then first convert this 3D geometry (represented with 2D polygons and 1D stacks) into a flat 2D optical geometry, by compressing the refractive index distribution in the 1D stack into a single effective index (bottom right in Fig. 7). It is also possible to extract the distribution of any material property (such as the refractive index or the dielectric constant) on a cartesian grid, for simulation tools that require a discretized distribution (e.g. FDTD).

Abstract models for the different simulation types. In optics, there are different type of solvers, as explained in Section 3.2 and illustrated in Fig. 2. The representation of the field is different for a mode solver (sum of eigenmodes) than for a FDTD simulation (field at all times), and the abstract model needs to take care of the appropriate

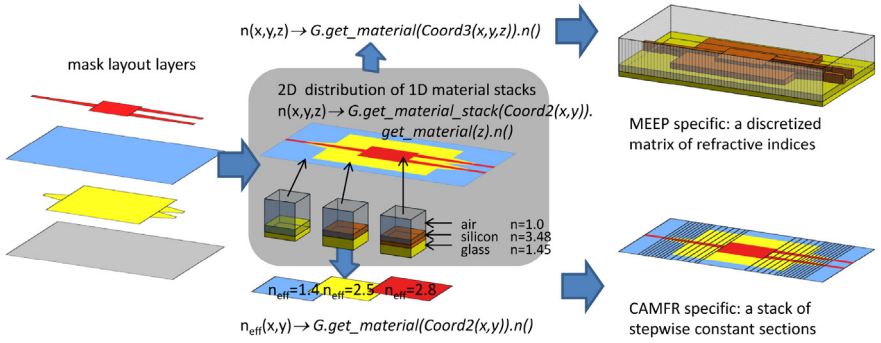


Fig. 7. A structure is defined as a 2D distribution of 1D material stacks. Through several layers of abstraction, the material properties such as the refractive index of these stacks are translated on to a 3D cartesian grid $n(x, y, z)$, or on a 2D grid of effective indices $n_{eff}(x, y)$.

conversions. Fig. 8 shows which classes were used to interface with the CAMFR simulation tool.

Implementations. From the abstract simulation models, concrete implementations are inherited for the specific tools used (for example in optics, we use Meep as FDTD solver, CAMFR and FimmWave as eigenmode solvers). Additional implementations can be developed for other research domains. This is an important investment in the framework before we started with the technical integration of different tools. The advantage is that all PCells can now flexibly switch between different tools.

Listing 4.3.1. Users can create a simulation (CAMFR in this case) of any component they design without additional programming. The CAMFR engine was constructed using several abstract classes (see Fig. 8).

```
engine = CamfrEngine()
my_geometry = StructureSimulationVolumeCartesianGrid2D(
                                          structure=mmi, resolution=5)
# The following method returns a CAMFR-native simulation object. This
# simulation object can be customized by the user or used to calculate
# the eigenmodes of the structure. CAMFR default settings are defined
# in the TECH tree, but can be overruled.
my_sim_obj = engine.get_camfr_object_for_geometry(geometry=my_geometry)
my_sim_obj.calc() # Call the calc function of CAMFR
my_sim_obj.plot() # Plot the modes (a built-in method of CAMFR)
```

It becomes clear that implementing an interface to an external tool demands a significant investment and technical expertise. It also requires good knowledge of the simulation tools because an interface should be based on best practice. An experienced researcher which is familiar with the scripting tools or user interface can easily set up a simulation. When implementing an interface to a tool (such as the CamfrEngine in Fig. 8), this know-how becomes part of the framework. After this, researchers do not have to care about learning a wide variety of scripting tools or user interfaces, saving time and making simulations less error prone. The know-how of fellow researchers can be easily leveraged in this way, so that researchers can focus on their core research activity.

Interfacing with different tools depends on how these tools are controlled. CAMFR, for example, is scripted in Python, so integrating this in the toolbox does not add technical difficulties, apart from converting the nanophotonic component to the necessary syntax as we explained in this section. For some tools, it is possible to communicate with them using sockets (for example: Fimmwave).

Another way to interface to a tool is by using the tool's API. If an API is provided, one can tightly integrate the tool with the framework. A very good example of this is Python-Meep, a thin wrapper

```
class __SimulationEngine__(StrongPropertyInitializer):
    """ Generic engine for a simulation """

class ModeSolverEngine(__SimulationEngine__):
    """ A mode solver calculates the eigenmodes of
        a cross-section of the device """

class CamfrEngine(ModeSolverEngine):
        """ Eigenmode solver based on CAMFR """
    def get_camfr_object_for_geometry(self, geometry):
        # Translate a generic Geometry into a CAMFR-
        # specific simulation object, based on the
        # geometry and CAMFR defaults defined in
        # the TECH tree. Result is camfr_object
        return camfr_object
```

```
class __StructureSimulationVolume__():
        """ Abstract class for a simulation volume """
    structure=RestrictedProperty(required=True,
                      restriction=RestrictType(Structure),
                      doc="The ipkiss structure")

class StructureSimulationVolumeCartesianGrid2D(
                                  _StructureSimulationVolume__):
        """ A regular 2D grid, used for example
            for FDTD and eigenmode solvers """
    resolution = IntProperty(default=5,
                      doc="Resolution for discretisation")
    def get_material_dataset(self, corner, width, height):
        # Calculate x_min,max and y_min, max
        # from corner,width, height
        material_array = self.geometry.get_material_array()
        return material_array[x_min:x_max+1,y_min:y_max+1]
```

Fig. 8. Abstract models for a simulation (left) and for a simulation volume (right). Using these conventions, new simulation tools can be integrated in the framework. Adding a new simulation tool can be rather involved and requires good programming skills and the knowledge of the specific tool, but once a new simulation class is added, it can be used to model all components that were previously designed in the framework.

around Meep, developed at Ghent University. SWIG [7] was used to make the bridge between the C++program Meep and Python. Technical details about interfacing C++and Python, and why we chose to use SWIG, can be found in [15]. In this way, scripts can directly be written in Python instead of using the C++API or the Scheme programming language. When interfacing directly is not possible, one can still interface through files.

Another mixin has been developed that makes the PCell available to OpenAccess compliant tools, such as Cadence, which is an EDA tool commonly used to design electronic systems.

4.3.2. Optical circuits: interfacing to circuit simulation tools

Circuit-level design is crucial in making integrated optical systems [16–19]. IPKISS allows the user to link PCells, allowing it to drive circuit simulation tools both in frequency domain (for example in optical filter design) and in time domain (for example for modeling advanced modulation schemes). The following piece of code shows how two Structures A and B can be linked together:

Listing 4.3.2. Defining a netlist to link components. This allows us to simulate circuits, and automatically trigger different simulation strategies for individual subcomponents.

```
class AB(Structure):
    def define_netlist(self, netlist):
        N = Net()   # A net can link an arbitrary amount of ports
        N += self.children.A.east_ports[0]
        N += self.children.B.west_ports[0]
        netlist += N
        return netlist
```

The `Structure` AB now contains an internal representation of its network. This can be used to route electrical and/or optical signals from one `Structure` to the other.

4.4. Component library

On top of our framework we have made a component library (a subset of it is distributed with IPKISS). In it there are a lot of designs for already fabricated and tested devices. Using little programming work, new components can be designed. The flexibility of Python allows to easily swap and redesign pieces of components, which we will illustrate in Section 5.

4.5. Python libraries

The rich ecosystem of Python greatly facilitates research activities. We list some of the employed libraries together with their use. We use Mayavi [21] for 3D visualization of the devices and Matplotlib [22] for 2D visualization. Shapely [23] is used to manipulate the geometry of the components with logical operations during the algorithm for virtual fabrication. h5py [24] is used to read data from simulations, and SciPy is used for data fitting. Next to these free libraries, we also interface with commercial tools, e.g. FimmWave [6]. Our philosophy is to include at least a free tool where possible to cover the basic functionality without an additional cost. Other libraries can be added in the future. For consistency within our group, and to facilitate installation, we use the Enthought Python distribution [13], which contains many of these libraries, and is free for academic use.

5. Advanced workflow for designing an Arrayed Waveguide Grating

In this paragraph, we demonstrate how we create a workflow to design and model an Arrayed Waveguide Grating (AWG). We demonstrate how we can easily swap components, and how different simulation models can be used for different subcomponents. The used concepts can be generalized to other domains, such as multiphysics simulations and electronic design.

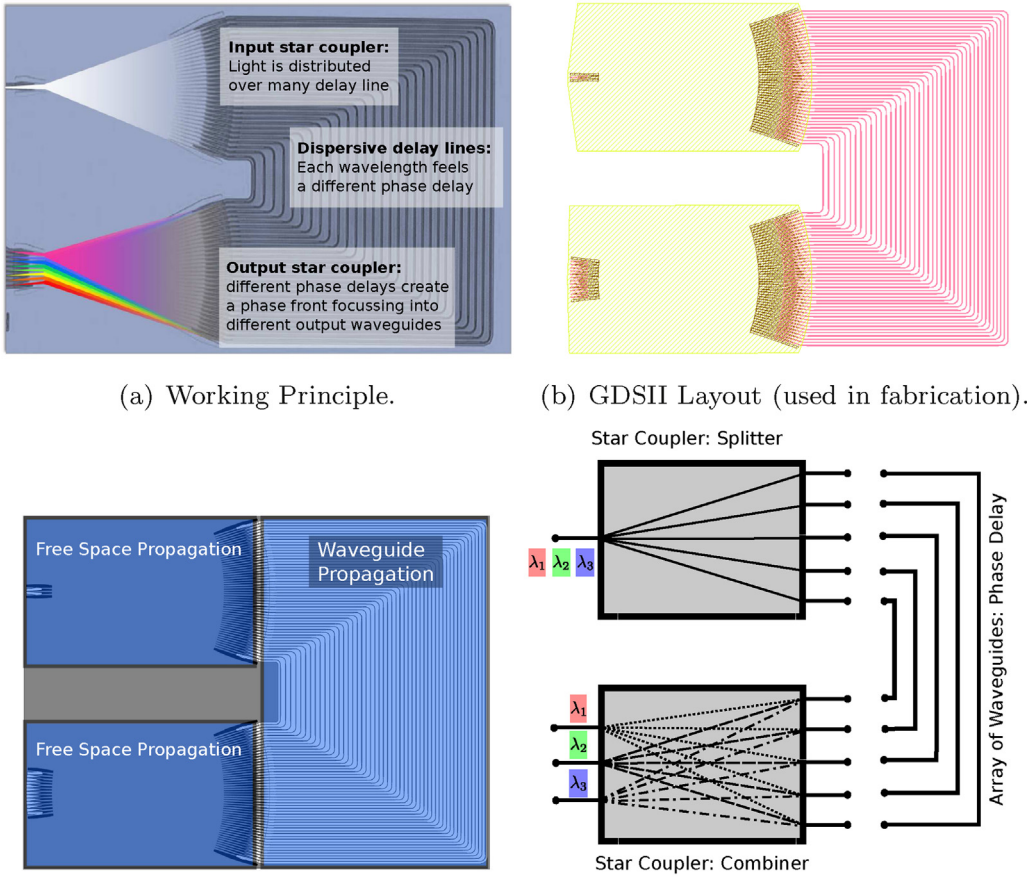
An Arrayed Waveguide Grating (AWG) is one of the vital components in Wavelength Division Multiplexing (WDM) systems. They are used to separate many wavelength channels into different waveguides (or vice versa, merge them). It consists of two star couplers and an array of waveguides with a linear increment of length. The principle is demonstrated in Fig. 9(a): A light beam enters the input star coupler and is distributed over the waveguide array. The different wavelengths reach the second star coupler with a different phase shift. Because of this, different wavelengths focus at different output positions.

Using a single simulation technique it is difficult to simulate these kinds of complex structures. We developed a hybrid model using our software framework to design and simulate the AWG [25], consisting of a well integrated combination of (semi-)analytical methods (in Python code) and numerical methods (using programs interfaced to IPKISS).

The AWG is divided into three parts: two star couplers and an array of waveguides. Fig. 9(c) shows the simulation domain. For simulation of the array of waveguides we use an analytical waveguide propagation model which can take some non-idealities into account. The length of the waveguides is automatically extracted from the building block and the waveguide propagation properties are described in our waveguide model. To simulate the transmission of the two star couplers, they are broken down further into several parts: the input and output waveguides and the free space propagation region in which the light beams expand. The input and output waveguides are simulated using CAMFR – as CAMFR is integrated into the framework, the extraction of the physical geometry from the higher level description and feeding that to CAMFR is automated. The free space propagation region is simulated using a semi-analytical method implemented in Python code, taking the electromagnetic modes of the input and output waveguides as returned by CAMFR. Positions and dimensions of the waveguides, dimensions of the free space propagation region are all automatically extracted from the higher-level description, and material properties are obtained from the different subcomponent models. In the end, the transmission of the full AWG is obtained by multiplying the calculated transmission matrices (T matrices) of the star couplers and the waveguide array.

To validate our simulations we compare these results with measurements for a 12×400 GHz MMI-AWG, shown in Fig. 10.

The spectral response of the individual channels match nicely with the experimental results as shown in Fig. 10. Note that there is a shift in wavelength between the simulations and experiments, and that in the experiments the extinction ratio is about 5 dB smaller. Based on these observations, we can make corrections in the models, such as slightly modifying the refractive indices, or incorporating additional losses in the waveguides. The flexibility of the framework allows us to plug in other subcomponents without any duplication of work, such as different input/output waveguide



(a) Working Principle.

(b) GDSII Layout (used in fabrication).

(c) Two simulation techniques used: free space propagation and waveguide propagation.

(d) Circuit Model: multiplication of the three matrices results in the total transmission.

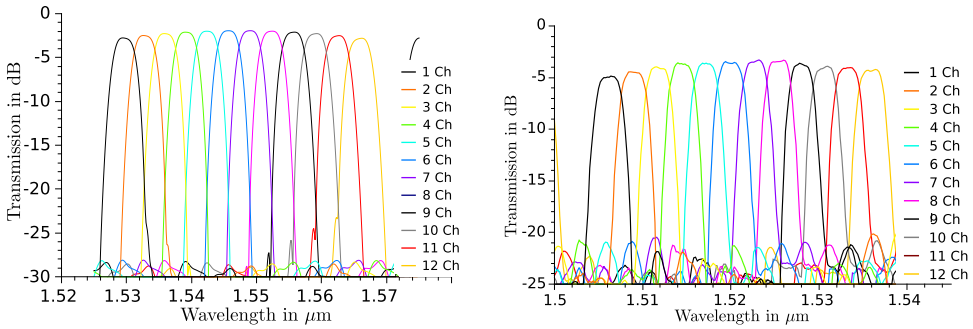
Fig. 9. Different models of the Arrayed Waveguide Grating. The GDSII file (b) and the simulation volume (c) are automatically generated by the framework.

apertures [25] and different types of waveguides and waveguide bends. This allows us to very quickly simulate and fabricate an AWG with functional specifications without fabrication errors and with good correspondence between simulation and measurement. This idea was introduced in [26]. Also, the influence of various parameter variations can easily be studied, as long as they are within the acceptable parameter range of the device and waveguide models.

Instead of writing separate independent scripts that do all this work, we were able to describe and solve the problem fully in the software framework. Python subclassing facilitates this: the users only need to override the part of the design in which they are interested. Other designers can now easily plug in their own apertures and waveguides, optimize this component using the proposed framework, and finally fabricate the component.

6. Availability and licensing

IPKISS is a multi-licensed open-source project. There are three available licenses, targeted at different users (see also [1]):



(a) Simulation Result

(b) Experimental Result

Fig. 10. Measurements (right) of the AWG match very good with the semi-analytical model which was constructed using IPKISS (left). Using the high-level description of the AWG, we can immediately simulate and fabricate an AWG with different parameters.

- Community license: A GPL2-licensed code base of IPKISS will allow access to the framework free of cost.
- Developer: A custom license that allows the user to distribute his own plug-ins and component libraries at his discretion, without the open-source requirements imposed by the GPL (for instance, under non-disclosure terms which would be incompatible with GPL2). The licensee does not have the right to distribute the IPKISS framework itself, ensuring that plugins remain compatible with the main code base.
- Custom licenses: Developers who require custom license terms or want to incorporate IPKISS into their own products can contact the authors.

Apart from the IPKISS framework itself, a subset of the component library is available as open-source.

7. Conclusion

The IPKISS software framework provides a powerful and generic environment for the design, simulation and fabrication of electronic and optical components and circuits. The software framework improves the workflow for designing components because all steps in the workflow are based on a single high-level description. The flexibility of the framework allows easy customization of components and workflow using the standard high-level language Python. Most commercial packages (for example, Phoenix software [27] and Lumerical [28]) do not allow this type of flexibility and do not feature mixing of different tools. IPKISS, on the other hand, is not tied to any specific vendor of simulation tools. The single description reduces errors in the design process and greatly simplifies the optimization workflow of a given component. By linking components using netlists, one can simulate circuits, and automatically trigger different simulation strategies for individual subcomponents.

The IPKISS framework has been in development at Ghent University and IMEC since 2002 and has proven its worth extensively for silicon photonic design and simulation. Recently, it has been made publicly available through an open-source licensing scheme [1].

Acknowledgements

This work is supported by the Interuniversity Attraction Pole (IAP) Photonics@be of the Belgian Science Policy Office and the ERC NaResCo Starting grant. M. Fiers acknowledges the Special Research Fund of Ghent University. We acknowledge Y. De Koninck for his useful comments.

References

[1] http://www.ipkiss.be
[2] S. Selvaraja, P. Jaenen, W. Bogaerts, P. Dumon, D.V. Thourhout, R. Baets, Fabrication of photonic wire and crystal circuits in silicon-on-insulator using 193 nm optical lithography, Journal of Lightwave Technology 27 (18) (2009) 4076–4083.
[3] P.R. Bienstman, Baets, Optical modelling of photonic crystals and VCSELs using eigenmode expansion and perfectly matched layers, Optical and Quantum Electronics 33 (2001) 327–341.
[4] http://camfr.sourceforge.net
[5] A.F. Oskooi, D. Roundy, M. Ibanescu, P. Bermel, J.D. Joannopoulos, S.G. Johnson, MEEP: a flexible free-software package for electromagnetic simulations by the FDTD method, Computer Physics Communications 181 (2010) 687–702.
[6] http://www.photond.com/products/fimmwave.htm
[7] http://www.swig.org
[8] http://www.ePIXfab.eu
[9] P. Dumon, W. Bogaerts, R. Baets, J.-M. Fedeli, L. Fulbert, Towards foundry approach for silicon photonics: silicon photonics platform ePIXfab, Electronics Letters 45 (2009) 581–582.
[10] http://www.imec.be
[11] http://www.leti.fr/en
[12] http://docs.python.org/reference/datamodel.html#descriptors
[13] http://www.enthought.com
[14] C. Esterbrook, http://www.linuxjournal.com/article/4540, Linux Journal (84), 2001.
[15] E. Lambert, M. Fiers, S. Nizamov, M. Tassaert, S. Johnson, P. Bienstman, W. Bogaerts, Python bindings for the open source electromagnetic simulator Meep, Computing in Science Engineering 3 (3) (2011) 53–65.
[16] http://www.aspicdesign.com
[17] http://www.lumerical.com/tcad-products/interconnect
[18] http://www.vpiphotonics.com/optical systems.php.
[19] M. Fiers, T.V. Vaerenbergh, K. Caluwaerts, D.V. Ginste, B. Schrauwen, J. Dambre, P. Bienstman, Time-domain and frequency-domain modeling of nonlinear optical components at the circuit-level using a node-based approach, Journal of the Optical Society of America B (2012).
[20] http://photonics.intec.ugent.be/research/topics.asp?ID=138
[21] http://mayavi.sourceforge.net
[22] http://matplotlib.sourceforge.net
[23] http://trac.gispython.org/lab/wiki/Shapely
[24] http://h5py.alfven.org
[25] S. Pathak, E. Lambert, P. Dumon, D.V. Thourhout, W. Bogaerts, Compact SOI-based AWG with flattened spectral response using an MMI, in: Proc. Group IV Photonics., 2011, pp. 45–47.
[26] W. Bogaerts, P. Bradt, L. Vanholme, P. Bienstman, R. Baets, Closed-loop modeling of silicon nanophotonics from design to fabrication and back again, Optical and Quantum Electronics 40 (2008) 801–811.
[27] http://www.phoenixbv.com
[28] http://www.lumerical.com

Martin Fiers completed his studies in engineering (electrical engineering) at Ghent University, Belgium in 2008 and joined the department of information technology (INTEC) at the same university. He is a Ph.D. student working on Photonic Reservoir Computing. His main interests are the modeling of nanophotonic components and reservoir computing.

Emmanuel Lambert received the master degree in engineering from K.U.Leuven University, Belgium in 1999. He joined the department of information technology (INTEC) in 2009, and is working on an integrated software framework of photonic design tools. His interest are the modeling of nanophotonic circuits, and the integration of different software tools.

Shibnath Pathak received his M.Sc. degree in Physics from Indian Institute of Technology, Madras (India) in June 2009. As part of M.Sc. degree he completed his master thesis on near field scanning microwave microscopy. In November 2009 he joined the department of information technology (INTEC) at Ghent University as a Ph.D. student in the photonics research group. His main research interests are simulation and design of a silicon AWG and of optical switches.

Bjorn Maes received the engineering degree in applied physics in 2001 from Ghent University, Belgium, and a Ph.D. from the same university in 2005. During 2005–2006 he spent one year as a postdoctoral associate at the Joannopoulos research group at MIT. In 2006–2010 he was a FWO postdoctoral fellow at the Photonics Research Group from Ghent University. He is working on the physics of photonic crystals, plasmonics, nonlinear photonics and solar cells. Bjorn Maes started a staff position at the University of Mons in September 2010.

Peter Bienstman received a degree in electrical engineering from Ghent University, Belgium, in 1997 and a Ph.D. from the same university in 2001, at the Department of Information Technology (INTEC), where he is currently an associate professor. During 2001–2002, he spent a year in the Joannopoulos research group at MIT. His research interests include several applications of nanophotonics (biosensors, photonic information processing, etc.) as well as nanophotonics modeling. He has published over 50 papers and holds several patents. He is a member of IEEE-LEOS.

Wim Bogaerts is professor in the Photonics Research Group at Ghent University, Belgium. He completed his studies in engineering (applied physics) at Ghent University in 1998 and joined the department of information technology (INTEC) at both Ghent University and the Interuniversity Microelectronics Center (IMEC) where he received his Ph.D. degree in 2004. In the photonics research group he specialized in the modeling, design and fabrication of nanophotonic components. Currently he coordinates the development of nanophotonic components in SOI in IMEC. He is a member of the IEEE Photonics Society, the Optical Society of America (OSA) and SPIE.

Pieter Dumon received the degree in electrical engineering from Ghent University, Belgium, in 2002, where he received a Ph.D. degree in electrical engineering in 2007 for his work in wavelength filters in silicon photonic wires. He currently coordinates ePIXfab, a initiative for multi-project wafer fabrication in photonics.