Summary

The integrated design environment described in deliverable WP3, D3.1 Report on design environment and building block specification is a key part in the definition of the co-operation between the different software vendors within EuroPIC. This co-operation will be handled via an information exchange protocol between the existing software tools, which should primarily support the sub-system simulation in the short term, while in the longer term physical layer simulations and mask layout will also be needed.

This application programming interface (API) will handle:

- convergence between frequency and time-domain numerical methods at the sub-system layer
- allow for co-simulation between sub-system and physical layer
- incorporate simulation, layout and process-info models in building blocks

It is intended that as the toolset matures access to the parts of the API will be given to expert designers.
Table of Contents

Table of Contents ................................................................................................................................. 2

1.1. Programme Objectives .................................................................................................................. 3

1.2. Background ..................................................................................................................................... 3

1.3. Motivation and Purpose .................................................................................................................. 5

1.4. Introduction to the Design Environment ....................................................................................... 6

1.5. Programming language(s) ............................................................................................................. 9

  Integration approach .......................................................................................................................... 10

1.6. Scope of the API ............................................................................................................................ 10

  Foundry definitions .......................................................................................................................... 11

  Building block definitions ................................................................................................................ 11

  Netlist definition .............................................................................................................................. 12

  Optical sub-system signal definition .............................................................................................. 12

  Access to physical layer solvers ...................................................................................................... 13

1.7. Conclusions .................................................................................................................................... 13

---

Dissemination level

| PU | Public | PU |
1.1. Programme Objectives

It is the objective of the project to bring the application of photonic integrated micro-systems in advanced products within reach for a broad class of SMEs by reducing their required investment costs by more than an order of magnitude. This will be done by developing a knowledge-based technology for production of Photonic Integrated Circuits (PICs) that will combine an increase in flexibility with a dramatic reduction of cost. It will lay the foundation for a breakthrough of PICs in a wide range of applications.

The methodology the consortium will apply to reach the project goals is:

1. To decompose the functionality of complex photonic micro-systems into a small set of basic functions.
2. To develop a Building Block for each of the basic functions, and to develop production processes that are capable of integrating the basic Building Blocks in any arbitrary combination and number.
3. To develop a dedicated design kit that contains an accurate model of the performance of the basic Building Blocks and that can simulate the response and the performance of complex circuits built from these building blocks.
4. To develop dedicated measurement tools and vehicles for testing the quality and performance of the basic Building Blocks, in such a way that testing the performance of complex circuits is reduced to proper testing of the basic Building Blocks.
5. To test the foundry concept with examples of high complexity Application Specific Photonic Integrated Circuits (ASPICs)
6. To develop a small set of generic packages by introducing standardization in the positions of optical and electrical connections, in chip dimensions and positioning of heat generating elements.

This methodology will result in a generic integration technology. This deliverable specifically addresses item 3.

1.2. Background

CMOS technology, wherein a huge number of functions is reduced to a few elementary electrical functions which are performed by elementary building blocks like transistors, diodes, resistors and capacitors fabricated in a generic process, supports integration of these elementary building blocks in large numbers and in different circuit topologies. Therefore CMOS is capable of realizing a wide variety of functions (chips) for a very broad range of
applications. The fabrication processes are made available in foundries to a large number of designers who use powerful design kits that allow fast and accurate design of the chips. By analogy in InP-based photonics, Figure 1 below shows how a number of very distinct functionalities can be realized with a small number of basic building blocks in InP such as waveguides, phase modulators and optical amplifiers. The Figure illustrates how a generic foundry approach based on the definition of elementary building blocks can be also feasible in photonics also.

*Figure 1 Building Blocks (top line) and their relationship to more complex photonic devices*

The new approach to a generic InP based technology matches well with a sub-system layer approach in which building blocks are connected on-screen and simulation work happens at a high level. The underlying reason for this is that building blocks need to be transferable from fab to fab and so need to be specified by their optical functionality rather than detailed geometric and materials data. This has as second impact, a shift from detailed underlying physics to more functional designs, which are therefore easier to maintain and upgrade when new platform capability is added. New users of photonics components can avoid the detailed physics, and the detailed knowledge of fab processes which goes with it, for longer, greatly simplifying the evaluation of optics in their applications.
1.3. Motivation and Purpose

The design environment, comprising of design tools built into an applications programming interface (API) is a central part of the overall foundry approach being taken in EuroPIC. As illustrated in the generic process flow illustrated below (Figure 2), the design toolset allows a transformation from a design concept in the mind of an applications engineer to a mask layout for a chip or ASPIC which can be fabricated in the generic fab using a foundry platform process. In essence it shields the applications engineer from the complexity of the foundries and provides the designers with access to a set of generic platform tools with which they can work. Definition of the API is an important step forward, and needed to create the inter-operability of the software tools from the different commercial vendors and design house internal libraries.

The purpose of this deliverable is to set out our view of how the design tool set should be structured, and provide an overall description of the proposed interface (API). In fact the typical designer will not see the API, which is a protocol for the interworking of software elements, but rather will work with high level tools. However, more advanced design houses may wish to use it to integrate their own components into the toolset.

Figure 2 Generic Manufacturing chain for Photonic Integrated Circuits

The physical layer which is traditionally used in photonics to obtain maximum performance in the design requires extensive physical models, of which a variety is available within the consortium. Although we expect that this can be avoided for relatively novice users with medium complexity design requirements, the high-end design houses with demanding applications require access to such detailed modelling as well. Finally, output to a mask layout suitable for the generic foundry needs to be supported by the design environment. Therefore the usage of tools relating to the physical layer will be supported at the sub-system level.
Rather than build such a design environment from scratch, we build on top of existing software tools and extend these with foundry-validated building block descriptions (Figure 2). The measurement and calibration of components or building blocks from the foundries is an expensive task, therefore it is important that the data obtained can be used directly in the design environment, rather than having a variety of tools with incompatible data structures, since this would require keeping a lot of data sets up to date and thus maintain a high cost level. Our goal is to provide the user with a single working interface, with automated plug-in factory specific data according to the target manufacturing route.

1.4. **Introduction to the Design Environment**

The design environment is built up from available commercial software tools, which are integrated to reduce cost of maintenance of the content contained within the calibrated building blocks. This modularised approach enables foundries, design houses and software developers to write and calibrate building blocks, material or cross-section information only once and then use them many times by having this information available to many software tools working in physical and/or subsystem layer. Such an approach enhances re-usability and thus increases reliability of the design toolset.

The integration of the various commercial photonic software tools will be realized using an application programming interface, API, which can also build on related background work undertaken in existing FP7 Call 2 projects Helios\(^1\) and Apache\(^2\). Within EuroPIC the focus is on the optical sub-system layer, for which two commercial software packages are available within the consortium; Aspic from Filarete, PicWave from Photon Design, and a third (academic) package, an optical element library built by COBRA on top of ADS (Advanced Design Suite – Agilent\(^3\)).

A variety of commercial and in-house developed software suites are available at different partners for describing the physical layer, and this pre-existing capability also needs to be considered.

---

\(^1\) HELIOS (FP7) is concerned with Silicon photonics and the integration of electronic design though tools from Mentor Graphics and Cadence.

\(^2\) APACHE (FP7) seeks to realise a 1TBit/s transmission system.

\(^3\) No longer a commercially supported approach.
The design environment, including the user interface (Aspic or PicWave) and software components linked through the API

The design environment is the overall combination of software tools and element libraries, but it is not a monolithic piece of software as is typically delivered by software companies. Therefore the API is needed to specify how the different software modules can work in partnership which each other. Figure 3 illustrates the concept of the API protocol. Tools such as Aspic or PicWave provide the user interface. The API provides the linkages by which stored, modular information about materials and processes inside the toolkit can be loaded for design use into C++ objects via the factory design pattern. To provide for flexibility and extensibility, loaders will be used to load all classes into memory. They will scan predefined directories and run entry functions. This will be organized using factory design patterns to ensure as little dependence as possible between codes.

This approach also allows integration of third party software such as physical layer tools.

Figure 4 shows how the core API links to the mask design software (PhoeniX) and offers the possibility of other plug-in extensions. Mask design is carried out by linking to existing PhoenixBV tools and libraries and their data files. This approach also allows for the integration of physical layer tools such as FimmWave and FimmProp.
The factory design pattern provides central storage of information about classes of a certain type, e.g. material or building block and allows us to create (instantiate) an object of such a type based on the name\(^4\). Therefore the "provider" side registers the types into the factory class, or more precisely the method to create an object. This is typically done with the combination of a simple name ("myBB") with a function ("createMyBB"). Using the C++ template class mechanisms this allows for easy definition.

The "user" side queries the names available and or directly calls the foundry to instantiate an object:

\[
pdaBB* \text{ obj=foundry.Create("myBB");}
\]

This is schematically shown in the Figure 5.

1.5. **Programming language(s)**

One of the key decisions to make concerns the choice of programming language within the design environment and so indirectly the API. The (commercial) partners are using tools written in C/C++, which makes this language the natural choice for defining the software interaction in the API. However, for the data formats for the content / element libraries a strong variety exists, so rather then developing many cross-combinations of reader (N) and writer code (M), we will not consider the data file formats to be part of the API, but part of the individual tools. This removes the N*M relation replacing it by an N+M relationship, but the approach may lead to limitations with respect to interoperability later on, because the different software tools do not directly share data files – a minor problem since direct access to such file formats can be built-in by the original software provider later if needed. Therefore, the usage of file formats needed to support the API will be made available to designers under the same conditions as the API itself, and preferably a reference implementation for accessing such files should be coded within the C++ code.

---

**Dissemination level**

| PU | Public | PU |
Integration approach

The approach of **dynamically loaded libraries** (DLL) will be used under the Windows operating system, while under Unix and Linux the **shared object** (SO) will be applied. This allows the use of plug-in libraries, which can be developed by parties outside EuroPIC or non-software partners, e.g. from design houses or universities. Such a DLL/SO will register its definitions during loading from the design environment, so no detailed interactions between software parties are needed. The DLL/SO may contain foundry definitions, material data or building block definitions. The classes, which are provided by the DLL/SO, are therefore only visible via the basic interfaces of the API. and since they are compiled by the provider no source code needs to be distributed. This approach also supports inter-process communication, since a single DLL/SO can register to many different parts, each of which can communicate with software running in parallel on the same computer or indeed elsewhere on the network.

This registering step is done via the factory design pattern as discussed earlier, which basically defines how to create an object when you know its name. Since a variety of objects is needed, a set of factories is defined by the design environment.

The main starting point for an application design is, therefore, a command to call the library loader to fill the factories. Examples will be provided to show the usage of the different parts of the API.

1.6. Scope of the API

The definition of optical building blocks, their parameters and connections in combination with the function which they perform on the optical signal is the key area for the API. In addition an initial link with the physical layer will be provided to enable wider parameter design ranges for building blocks without specific measurement content, for example the usage of a mode solver for obtaining a dispersion curve using a waveguide cross-section.

In essence the API thus contains:

- foundry definitions
- building block definitions
- netlist definition
- optical sub-system signal definition
- limited access to physical layer solvers

<table>
<thead>
<tr>
<th>Dissemination level</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
</tr>
<tr>
<td>Public</td>
</tr>
<tr>
<td>PU</td>
</tr>
</tbody>
</table>
In addition to these main topics, a set of basic C++ classes will be added for convenience. These are related to parameter definitions, loading of libraries etc and although important for the overall functionality of the API, a description of this low level functionality is considered outside the scope of the overview presented in this Deliverable.

In the following sections the main elements of the API will be addressed individually. Each of these briefly describes the class involved and its objective. All of these classes are new but will follow general design patterns and have libraries to support them if appropriate.

### Foundry definitions

The `pdaFoundry` class of the API defines the functions which need to be implemented by foundries (partners OCLR and FhG-HHI), although within EuroPIC this implementation task will be done by the software partners. It provides a factory to allow extensions via dll/so. In this factory "fab"s can be registered for future use within the application, with a library loader to provide the content from dynamically loaded libraries (*.dll, win32) or shared libraries (*.so, Unix) to fill this factory with implementations. This process is shown schematically in Figure 3. Each factory defines which building block it supports and with what parameter ranges. The same parameters are used for the same building block to ensure design transferability.

Each foundry class can provide for process flow or cross-section factories to support physical layer modelling, but this is not needed.

### Building block definitions

The building blocks derive from a base class `pdaBB` which handles connections and parameters. It also handles the library loading of building block definitions to allow plug-in building blocks from new parties, e.g. the design houses. The `photBB` is the "optics" class, which will be used primarily within EuroPIC.

The basic approach is therefore the use of a building-block factory.

The `pdaBB::Port` class defines the ports of an element, which is basically a relative location for mask layout and provides for simulation in the physical domain. In general, elements will have a few ports e.g. one or two optical inputs and outputs, two RF leads, maybe some DC power ports, but there is no upper limit. Elements like a star coupler may have > 1000 optical ports for example.
Using the ports of different elements permits them to be connected and this allows the combine them into a netlist. An extension to define optical ports is the `photBB::PortMode` which provides the optical properties on such a port.

An important aspect is the documentation part of the `pdaBB` class in which aspects like maturity and licensing can be specified, information which may assist the designer or their applications partner when qualifying the maturity of the design, making checks against IP infringement, or checking the need for (commercial) agreements.

**Netlist definition**

The `pdaNetlist` class stores the building blocks (`pdaBB`), their ports (`pdaBB::Port`) and the connections between all ports. A simple reference netlist implementation is provided, but because it is likely that more advanced netlist code will be developed, a factory approach is used here also in combination with a library loader. This allows developers to extend the capability or performance, without the need of introducing a lot of licensed code into the framework.

To support future extensions towards composite building blocks, a netlist can be a building block also and a template class `hierBB` to handle this is defined.

The use of a netlist with building blocks defined in DLL/SOs allows to browse through the netlist and apply different functions (e.g. create a layout, start a simulation) on each of the building blocks. Therefore very little information is needed to describe each building block, since the C++ API handles the rest.

**Optical sub-system signal definition**

The signal class `pdaSignal` defines the optical signal which can be used by either a frequency or time-domain solver. The current definition uses an S-matrix approach, which directly fits a frequency domain solver, however, conversion to and from the time-domain can be done using Fourier transforms. Dispersion effects also need to be considered within the API specification, but implementation lies outside the scope of EuroPIC.

To define the signal, a basic class is defined, `pdaSignal::Smatrix`, which allows a non-linear S-matrix to be defined. This non-linearity is needed to allow for example for cross-gain modulation in non-linear devices like amplifiers, where a series of signals at different wavelength passing through the amplifier will influence the response of each other, an effect which is exploited in a wavelength converter for example, but would compromise the

| Dissemination level | PU | Public | PU |
performance of a linear amplifier. A standard linear S-matrix is also provided as sub-class, which is primarily important for the passive building blocks.

### Access to physical layer solvers

Limited access to physical layer data for physics based solvers will be provided. Via the `pdaFoundry`, a `pdaCrossSection` cross-section class can be created for a certain width and radius to define a (passive) waveguide which should allow for mode solvers etc to be applied to it. Alternatively a `pdaProcessFlow` can be obtained from which such a cross-section can be obtained. Using the process flow which is more directly linked to clean room equipment allows more extensive tolerance and yield modelling, since process effects may influence each other and thus the resulting cross-section change is not necessarily orthogonal with respect to a geometric change.

Each foundry will standardise and fix is material layer specifications and processes so that as the generic foundry process matures, only a limited amount of cross-sectional information and/or process flow data are expected to be needed. For the API no limits are imposed in principle of course.

In addition to tolerance / yield analysis, the objective of using the `pdaProcessFlow` to provide the cross-section rather then using the `pdaCrossSection` directly is to support "virtual fab" approaches in which 3D structural modelling can be carried out (e.g. KOH etching). This will not be studied within EuroPIC, but is of interest for the future for addressing problems such as analysis of crosstalk effects in photonic circuits.

Interoperability of tools at this level needs more study since the different solvers available to the consortium require different types of input. Since EuroPIC's focus is not primarily on this layer, and the expected amount of cross-sectional data is expected to be low, we feel there is no need to focus on integration at this level for now, keeping the existing file formats for the physical layer solvers and only defining the optical output parts via the class `pdaModeModel`.

### 1.7. Conclusions

An application programming interface (API) has been defined and a basic implementation of the interface in C++ has been carried out in order to ensure correctness and completeness of the definition.

In addition, example files are being developed to demonstrate the use of the API and provide a test bed for API extensions and concepts.