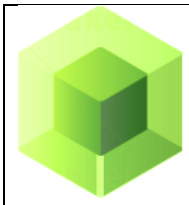


MMP

Deliverable D2.5

Deliverable:	D2.5 Report on developed application interfaces
Responsible Partner:	TNO
Contributors:	TNO, VTT, Philips
Dissemination Level:	Public
Distribution List:	Consortium members, European Commission, PTA
Due Date:	M22 (extended to M24)
Preparation Date:	2015-12-15
Report Status:	final

NMP.2013.1.4-1:	Multiscale Modelling Platform: Smart design of nano-enabled products in green technologies
Project nr:	604279, collaborative project
Project Duration:	01.01.2014 – 31.12.2016
Project website:	http://www. mmp-project.eu
Coordinator:	Dr. Jan-Paul Krugers, jan-paul.krugers@tno.nl Ir. Sjoukje Wiegersma, sjoukje.wiegersma@tno.nl
Project partners:	TNO, TU/e, CTU, Access, VTT, CelSian, Philips, Abengoa

Table of Contents

1 Introduction

Deliverable 2.5 is the report on the developed application interfaces (API): it provides details of the developed APIs of the individual software models, including data and command exchange according to Deliverable 1.1 (D1.1 Application interface specification).
The deliverable covers task 2.3 which concerns the 'development of the code that facilitates interfacing between the sub-models and the platform'. An API is to be developed for each model or software used within the simulation chain. These APIs will facilitate the data input and output required to run the simulation at hand. The interfaces will be implemented according to the requirements and specifications of task 2.1. Task 2.1 is described in Deliverable D2.1 and D2.2/Chapter 5. Furthermore, additional services for data exchange and conversion, according to task 2.1.3 (Deliverable 2.2/Chapter 5.3), should be developed and implemented.
Chapter 2 of this deliverable provides a summary of the simulation chain requirements and API requirements, followed by a description of the developed different APIs in Chapter 3. Additional services for data exchange and conversion are not described, because none were implemented. Task 2.1.3 describes platform requirements instead of required additional services. The additional services which were required up to this moment were just included within the APIs and are described within the API descriptions. Other services (like vtk visualization and variable storage) is provided through MuPIF (Multi-Physics Integration Framework). The deliverable is finished with conclusions and recommendations (Chapter 4).
The APIs described in this deliverable are developed by TNO and VTT. The MMP-TNOAPI was developed by TNO, and the MMP-TracerAPI and MMP-MieAPI were developed by VTT. However, the API development is performed in close collaboration with WP1 partner CTU.
It should be noted that the dissemination level of this report is public, because it holds important lessons learned on application programming interface development. This can be of great value for further API development within, but even more so, outside of the MMP consortium. Therefore, the authors tried to also describe the steps, options, arguments and decisions within the API development process.

1.1 Nomenclature

The developed APIs are named respectively:

APIs are named:

```
MMP-TNOAPI
MMP-MieAPI
MMP-TracerAPI
```

API (mupif:Application is the super class) classes:

```
MMPMie
MMPTracer
MMPAppTNO
```

Actual programs that the APIs run:

```
MMP-MieGenerator
MMP-Raytracer
Comsol + Matlab
```

The `MMP-MieAPI` and `MMP-TracerAPI` are also package as mmp_mie_api and mmp_tracer_api -python modules. These modules can then be shared at pypi (Python Package Index: https://pypi.python.org/pypi).

2 Summary of API specifications and requirements

In this chapter, we provide a short summary of the simulation chain and the API specifications and requirements that were described in the project's earlier, confidential deliverables, i.e., D1.1, D2.1, D2.2 and D2.3.

2.1 Introduction and motivation

As stated in D1.1, the reliable multiscale/multiphysics numerical modeling requires including all relevant physical phenomena along the process chain and across multiple scales. This requires the combination of knowledge from multiple fields. The development of a new multi-physics tool for a particular problem is extremely time and resource consuming. Therefore, a more viable approach lies in combining existing (usually single-physics tools) to build a customized multiphysics simulation chain for a particular problem. The most important advantage of this modular approach is the reusing of existing tools/models. In order to achieve a full potential of such a modular approach, an integration framework is needed. This framework provides the underlying infrastructure that enables to integrate individual applications and it also provides tools for application steering and mutual data exchange.

2.2 Integration framework: MuPIF

In the MMP project's WP1, an integration framework called MuPIF has been developed. MuPIF [D1.1] is based on an object-oriented approach, consisting in designing a system of interacting objects for the purpose of facilitating the interaction of scientific software when solving multiscale/multiphysics problems. The identification of individual objects and their mutual interaction has been based on expertise of MMP project partners, and later refined by analysis of simulation scenarios considered in the project's WP2 and WP3. The main advantage of this approach lies in independence from particular data format(s), as the exchanged data (fields, properties) are represented as abstract classes. Therefore, the focus on services is provided by objects (object interfaces) and not on underlying data itself.
MuPIF has been implemented in Python. Python is an interpreted, interactive, object-oriented programming language. It runs on many Unix/Linux platforms, on the Mac, and on PCs under MS-DOS, MS Windows, MS Windows NT, and OS/2. The Python language will be enriched by new objects/classes to describe and to represent complex simulation chains. Such approach allows profiting from the capabilities of established scripting environment, including numerical libraries, serialization/persistence support, and remote communication.
The proposed abstract classes, depicted in Figure 1 below, are designed to represent the entities in a model space, including simulation tools, fields, discretizations, properties, etc. The purpose of these abstract classes is to define a common interface that needs to be implemented by any derived class. Such interface concept allows using any derived class on a very abstract level, using common interface for services, without being concerned with the implementation details of an individual software component. Further details on the MuPIF and its classes can be found in D1.1 or in the MuPIF User Manual available in

http://sourceforge.net/p/mupif/wiki/Home/attachment/MuPIF-userman-public.pdf

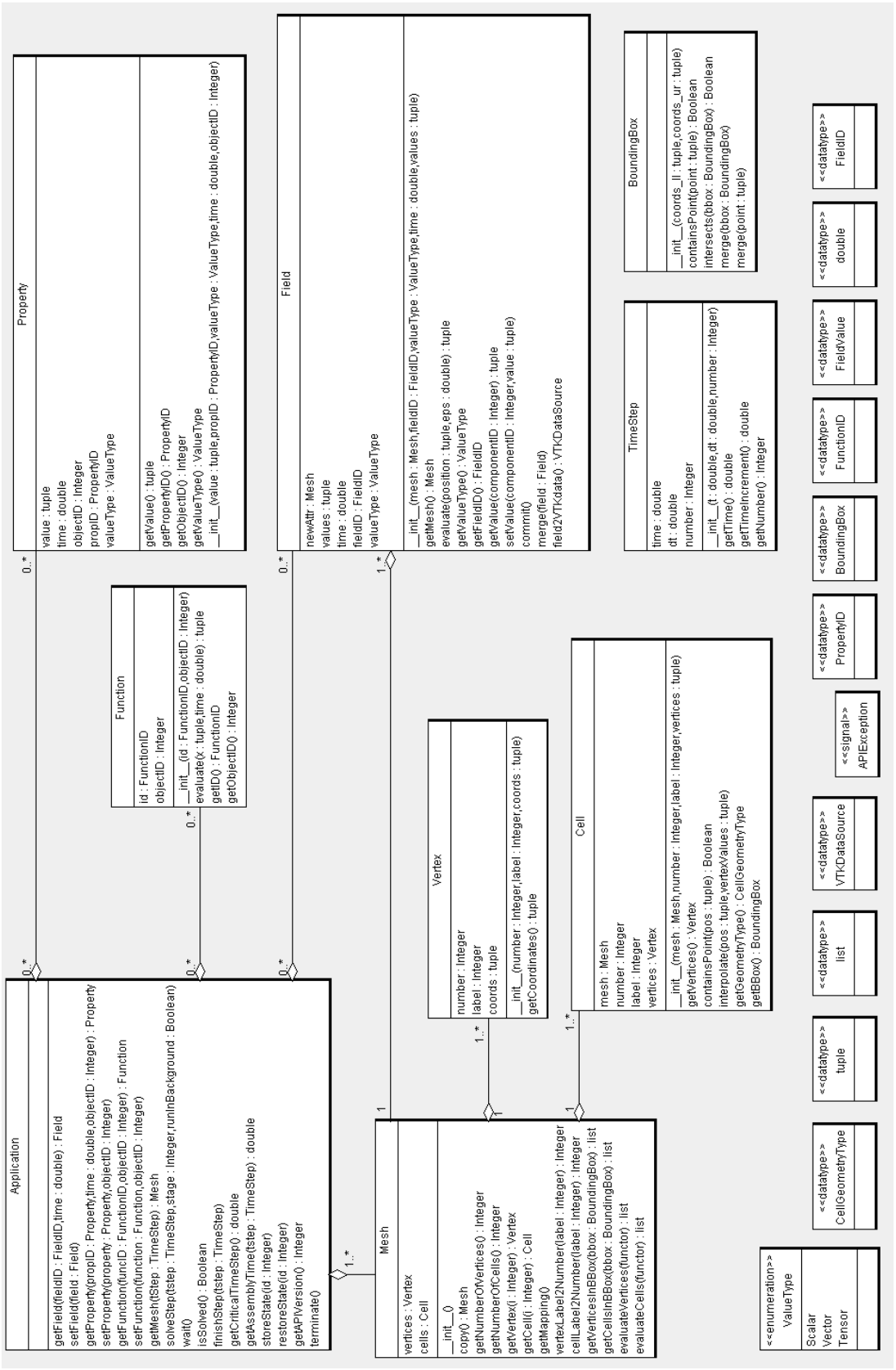


Figure 1: MuPIF UML class diagram (D1.1)

The following sections (2.3-2.6) describe requirements at different levels. These requirements were written down in earlier deliverables. How the current results meets these requirements is discussed in section 2.7.

2.3 Multiphysics simulation chain in WP2

The target of WP2 is to develop an opto-thermal multiscale modelling scheme to solve the design optimization problem using phosphors as light conversion material and computing the system level light output, taking heat dissipation into account. The optical model calculates the light absorption distribution inside the phosphor layer, blue die and the side walls of the molding component. This data is then transferred into the thermal model, where the absorption distribution is treated as an effective heat source. The thermal model calculates the stationary temperature distribution inside the entire LED and the transient temperature distribution during cooling down. These transient cooling curves can be used by the end user to derive structure functions (thermal resistance/capacitance map), which provide information about device reliability.
Originally, an optical-thermal coupling was envisaged, as excitation, absorption and emission spectrums of phosphor materials are temperature dependent. Therefore, the colour correlated temperature (CCT) of the LED's also changes with temperature. However, the required operation temperature of LEDs is within a small range. Within this temperature range, there is no relevant temperature dependency. Therefore, it was decided to omit the optical-thermal coupling. The Figure 2 shows the original (simplified) envisaged scheme in which the temperature distribution is still fed back into the optical model.

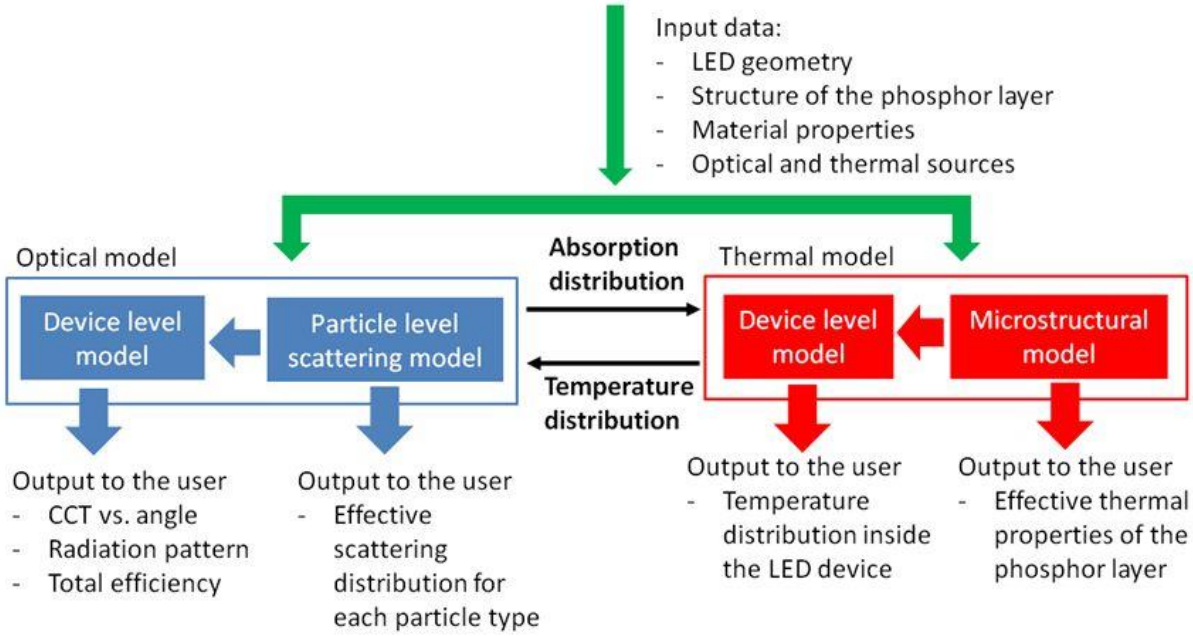


Figure 2: original model scheme (taken from D2.3)

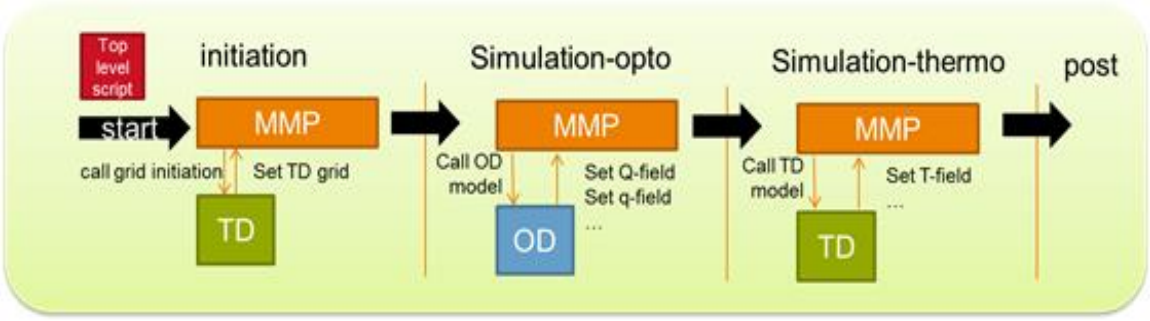
Inside the thermal model, the microstructural model and the device level model are coupled. The device level model does not include any phosphor particles as the phosphor layer is represented via effective thermal parameters calculated by the microstructural model.

2.4 Simulation chain in practice

Based on the original model scheme, a few possible 'generations' of simulation chains were envisaged: GEN1, GEN2 and GENX. This should be interpreted as a progression towards a final simulation chain. So, GEN1 was the most simple one and when the goals of GEN1 were more or less reached, GEN1 was improved towards GEN2 etc.

GEN1: Only device level models run at the platform level (Figure 3**)**

- The model focusses on running successive models, specifying the grid structure used by MMP in the phosphor/silicone-matrix domain, and on the absorbed heat transfer from the optical simulations to the thermal simulation by utilizing the MMP-platform capabilities.
- Material properties are all (locally) specified at device level model. This includes thermal and optical properties of the material used in the device.
- Optical modelling is done first, successively thermal-modelling is done. No thermal to opto feedback.



Optical Device (OD) model
Prepared TD grid
- Local scattering and emission data
- Local P_{blue}
- Local phosphor distribution
- Local reflectivity

Thermal Device (TD) model
Prepared TD grid
- Local geometry
- Local thermal properties
- Local boundary conditions
- Local P_{LED}

Figure 3: Schematic overview of the GEN1 simulation chain

GEN2: Successive thermal particle, optical and thermal device level simulations (Figure 4**).**

- End-user is enabled to set global parameters for design optimization.
- Thermal material properties are pre-calculated into a database using the particle level models.
- Optical and thermal device level models are run successively.

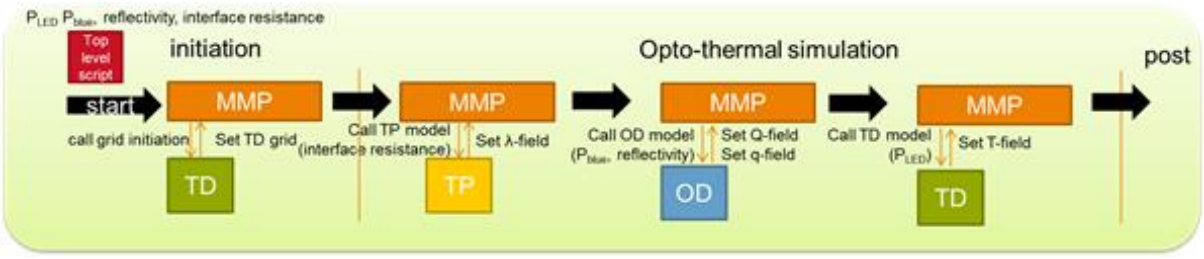


Figure 4: Schematic overview of the GEN2 simulation chain

GENx: Thermal microscale model and coupled device level models run at the platform level

- Optical and thermal device modelling are coupled, iteration between thermal and opto states required.
- More free parameters, more optimization freedom.
- The following input parameters were described as relevant regarding the LED design optimization:
- thermal conductivities of all LED components,
- the volume fraction of the phosphor and thermal conductivities of the silicone and the components of the phosphor mixture, and
- the layer thicknesses of the die attach and the epi layer.

2.5 Platform requirements

Basic requirements for the platform to run the opto-thermal simulation chain in the MMP platform were identified in Deliverable 2.2. Although the actual API requirements can be regarded as a subset of the total platform requirements, it is important to keep the complete picture in mind. The platform should:

- support steering of MATLAB, COMSOL, and the ray tracing software (C++ code),
- support data exchange (three dimensional temperature and absorption distributions are exchanged between the models),
- provide exchange of a spatial mesh/grid (nodes and elements),
- support device level description of modelling geometries,
- provide sufficient interpolation functions so that a field distribution can be mapped from a tetrahedral mesh to a rectangular mesh and vice versa,
- enable synchronous execution of different applications,
- provide persistent storage of results in all phases of the simulation chain,
- provide sufficient error handling and reporting,
- guarantee security when the simulation chain is run in distributed computing environments.

These requirements were communicated with WP1 and they were taken into account when the platform application interface specification was defined.

2.6 Additional requirements

The same input data should be fed into both models to ensure that both programs are modelling exactly the same problems with the same coordinate system. The origins of the coordinate systems in both models need to be exactly in the same place to guarantee that exchanged data is used in the right place of the LED geometry in the models. The input data defines (1) all parts in the LED geometry using primitive geometrical objects (such as sphere, block and cone), (2) structure of the phosphor layer (mixing ratios, particle types, density,

size distribution), (3) optical and thermal material properties (such as refractive index, thermal conductivity, heat capacity, density, and excitation and emission spectrum) and (4) optical and thermal sources (e.g. radiation pattern of the led die).
The optical model calculates the absorption distribution (W/m^3) in a uniformly sampled three dimensional rectangular mesh. The distribution is mapped to tetrahedral elements in the thermal model. In the mapping, it is important to ensure that the total amount of the absorbed power remains unchanged.

2.7 Discussion of the current result regarding the API and platform requirements

Section 2.4 ('simulation chain in practice') described 3 generations of envisaged simulation chains: GEN1, GEN2 and GENx. It should be noted that the simulation chain is already one step ahead of the development of API interfaces: the APIs should (only) allow for sufficient functionality within the simulation chain. Nevertheless, the best proof of API functionality is the fact that a fully distributed simulation was performed within the TNO network with the TNO and VTT APIs running on different computers. During this distributed simulation, the APIs provided all necessary functionality and accessibility to intermediate and final results.
Comparing the current simulation chain with earlier envisage chains, it can be concluded that the requirements of GEN1 are fulfilled and that those of GEN2 and GENx are largely fulfilled. The reason that GEN2 and GENx are not completely fulfilled is mainly caused by some project and model decisions which led to alterations in the simulation chain. Therefore, there is no database with thermal properties of the phosphor filled silicone (GEN2), no thermal feedback to the optical models (GEN2/GENx) and no free geometrical input parameters (GENx). There was no necessity to calculate and store the thermal properties of the phosphor filled silicone in advance, as the effective thermal conductivity can be calculated by the thermal circuit model instantly. Furthermore, the thermo-opto feedback was not implemented as the optical properties of the LED device hardly change within the allowed temperature range according to device specifications. Hence, both the database and the thermo-opto feedback had become irrelevant. Lastly, it was decided fix the coordinate system and the actual geometrical parameters, as significant geometrical changes (where the amount of domains/boundaries within COMSOL change) very likely results in a non-functional COMSOL model. There is not a problem in changing any of the other properties. Although several parameters are still locally set at this stage, freeing additional parameters (setting them globally) will be similar to freeing a single parameter.
Several of the platform requirements (section 2.5) were already mentioned and discussed. MATLAB, COMSOL and the ray tracing software consisting of MMP-MieGenerator and MMP-Raytracer can be steered through the APIs. The APIs include methods to set/extract the required parameters and results during the simulation. The data exchange (including the mesh) in between the APIs, and persistent storage, is supported through MuPIF. All necessary interpolations are implemented; the MMP-TracerAPI has some internal interpolation support functions, and MMP-TNOAPI also has support functions to interpolate the absorption as calculated within the MMP-Raytracer onto the COMSOL mesh and to interpolate the final transient result to user defined time points. The MMP-TracerAPI and MMP-MieAPI also work in combination with a MuPIF job manager. Hence, they allow for synchronous execution. The MMP-TNOAPI does not work in combination with a job

manager (yet). Although no fundamental issues are envisaged in combining also MMP-TNOAPI with a job manager, there are only a limited amount of MATLAB and COMSOL licenses available which also limits synchronous simulations. Error handling and supporting is implemented at different stages. Security aspects are addressed during the configuration of the simulation chain and therefore were not part of the API development.

3 Implementation of interfaces

In this section, the developed APIs for the thermal and optical modelling software are described.

3.1 Thermal modelling software

Table 1: MMP-TNOAPI summary

Name and model	MMP-TNOAPI
Owner	TNO
Type of implementation (native/indirect)	Indirect
Type of local OS	MS Windows or Linux
Security	Local or SSH-secured traffic
Error handling	Basic
Reporting	Logging-module enabled
Application classes	MMPAppTNO

3.1.1 API actions

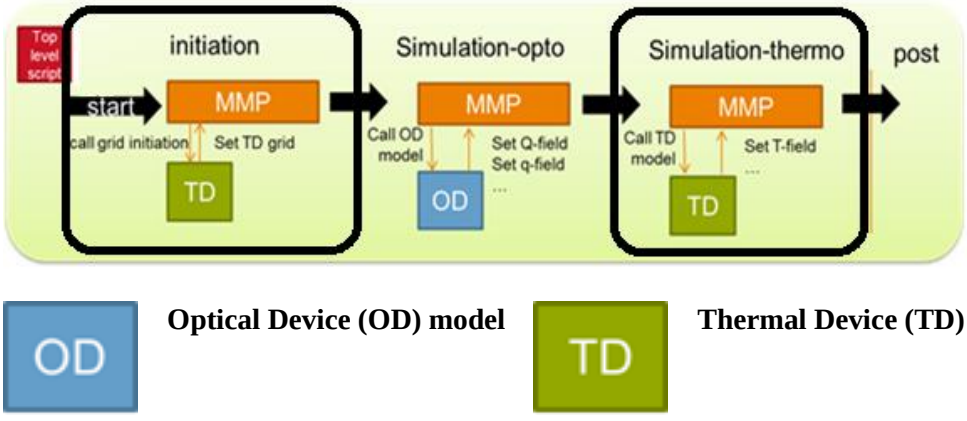


Figure 5: Simplified simulation chain with encircled MMP-TNOAPI parts

The API as developed by TNO (MMP-TNOAPI) covers the parts of the simulation chain where the thermal device model is engaged (encircled by the black boxes in Figure 5). The actions which are performed by MMP-TNOAPI are the following. During the initiation phase, the mesh of the thermal device model, silicone lens and lens reflector are requested and transferred to the API of the optical device model, i.e., MMP-TracerAPI described in section 3.2.2. After simulations with the optical device model are finished, MMP-TNOAPI receives two heat source fields. First, MMP-TNOAPI runs a model to calculate the effective thermal

conductivity of the phosphor filled silicone. Secondly, this effective thermal conductivity and the heat source fields are used to simulate the stationary and transient behaviour of a complete LED device. The main output is a time dependent temperature distribution of the LED.

3.1.2 Approach to COMSOL API

TNO developed two models: a microstructural MATLAB model for the calculation of the effective thermal conductivity of the phosphor filled silicone and a COMSOL model for the calculation of the thermal distribution within the LED device. Therefore also two APIs were developed initially: a MATLAB API for the microstructural thermal phosphor model and a COMSOL API for the LED device model. However, COMSOL has two APIs of itself; a JAVA API and a MATLAB API. These two APIs allow for complete control over the COMSOL model and simulation parameters. We decided to assess COMSOL through the COMSOL-MATLAB API as we were already experienced with that API. We had no experience with the COMSOL-JAVA API route.

3.1.3 Towards a single API (MMP-TNOAPI)

The COMSOL-MATLAB API starts the COMSOL server and MATLAB (GUI) which is connected to the COMSOL server. After connection, the MATLAB command line can be used to create, adapt and run COMSOL models. Furthermore, all available MATLAB routines and libraries can also be used, hence COMSOL and MATLAB code can be mixed. However, by using this route/option, MATLAB is always started together with COMSOL and it seems superfluous to create a separate API which only runs MATLAB (for the thermal phosphor model). The disadvantage is that the API does not get a 'single task' like only running the COMSOL LED device model or the MATLAB microstructural thermal phosphor model. Furthermore, the MATLAB microstructural thermal phosphor model can not be ran separately, unless a switch variable is included. However the advantage is that the data transfer is completely within MATLAB and does not require any additional measures. The thermal phosphor model is run first, followed by the COMSOL LED device model which uses the calculated effective thermal conductivity of the thermal phosphor model. Currently, only one value for the effective thermal conductivity is calculated (for the whole lens) as the differences in phosphor distribution over the silicone lens are not well known. Nor is the effective thermal conductivity very sensitive to the relative small differences in this distribution. However, the MATLAB LiveLink (which is different from the COMSOL-MATLAB API) in COMSOL allows for the connection of a MATLAB function to a COMSOL domain. In that case COMSOL runs the MATLAB function for every point in that domain separately, and the iteration process is taken care of by COMSOL (the MATLAB function runs within COMSOL). That option was not applied in the current case. However, in that case one would also create just a single API.

3.1.4 I/O between MMP-TNOAPI and MMP-TracerAPI

The former chapter explained the information which needs to be communicated between the MMP-TNOAPI and MMP-TracerAPI. During the grid initiation, MMP-TNOAPI determines the grid/mesh of the phosphor filled silicone lens and of the reflector and this information is

provided to MMP-TracerAPI. After the optical simulations launched via MMP-TracerAPI, MMP-TracerAPI provides a value for heat absorption per element for the silicone and reflector to MMP-TNOAPI. After the device level simulations with MMP-TNOAPI, the temperature dependent heat distribution for the whole LED device model is provided to the user. When the API development started, the data transfer between the MMP-TNOAPI and MMP-TracerAPI was set up using vtk/vtu files. This required additional scripts to transform output data to vtk/vtu and transform vtk/vtu to suitable API input. These vtk/vtu files had to be emailed or transferred by alternative file transfer. Although it is a very practical engineering approach and suitable for testing and local API development purposes, it is highly unpractical in case of many simulations. Furthermore, it also contradicts with the aim of MuPIF, which is intended to take care of the variable transfer. Hence, instead of using vtk/vtu files, the data transfer between the MMP-TNOAPI and MMP-TracerAPI was changed to MuPIF classes. This means that during the grid initiation, three empty fields are created by MMP-TNOAPI: one field for the whole mesh, one for the silicone mesh and one for the reflector mesh. The two last fields are transferred to MMP-TracerAPI and filled with volume and surface heat absorption values after the ray tracing simulations are finished. These updated fields are returned to MMP-TNOAPI and used as input in LED device model. After running the LED device model, the first empty field containing the whole mesh is filled with time dependent temperature distribution data. All these fields are available to the user and can be checked and plotted through MuPIF build-in vtk file generators.

3.1.5 API MuPIF compliancy

In order to establish an interface between the platform and an application, one has to implement the MuPIF Application class. Initially, some application dependent methods were added to interface with the application. However, then every application has its own typical methods and the user needs to know these. The idea of the Application class of MuPIF is that it defines a generic interface in terms of general purpose, problem independent methods that are designed to steer and communicate with the application. So, every application should have the same methods with the same function calls and input/output. However, the code within these methods is different for every interface and should be defined by the API developer. It is not required that all available methods are used, only the ones which are needed within the simulation chain. Therefore, the (initial) application dependent methods were redefined and reprogrammed in terms of MuPIF Application class methods.

3.1.6 Accessing the COMSOL/MATLAB workspace from Python

It is not possible to just connect Python with the MATLAB workspace and access the simulation variables. Although there is partly a one way connection, meaning variables can be provided as strings within the startup command of COMSOL/MATLAB. Nevertheless, this option is limited.

Two other possible options were distinguished to directly interface the MATLAB workspace with Python: pymatlab and the Python Engine for MATLAB. Pymatlab was tested and with some code adaptations/repairs it was possible to access the MATLAB workspace and transfer variables between Python and MATLAB, however it was not tested in combination with Pyro. We decided not to use pymatlab, at least not during this part of the project, as it has little support for MS Windows (it was developed for Unix/Linux), it connects to the 32-bit

MATLAB and is not further updated or revised since 2013. The Python Engine for MATLAB was released with MATLAB R2105a. That option was not tested as COMSOL 4.4, which is used for the LED device model, requires an older MATLAB version (R2103). Another issue for the current client-server implementation may be that it is stated by MATLAB that 'the engine (Python Engine for MATLAB) cannot start or connect to MATLAB on a remote machine'. Within this project, Python and MATLAB are both run on a remote machine, and therefore this limitation may not be problem.

3.1.7 I/O between Python and MMP-TNOAPI

As we decided not to use pymatlab or the Python Engine for MATLAB (R2015a), Python/MuPIF is not able to directly access the variables within the MATLAB workspace. To transfer data from MATLAB to Python, it was decided to use the write file/read file methods. Hence, the write file/read file methods are applied only locally within the API. In case of MMP-TNOAPI, there are a few of these write file/read file 'sessions'. During the grid initialization, the whole mesh and two submeshes are written to file, read again and stored in MuPIF fields. Before the LED device level simulation, the MuPIF fields from the MMP-TracerAPI are transferred to suitable COMSOL input files. COMSOL writes the time dependent temperature distribution to a file and MMP-TNOAPP reads and provides it as a MuPIF field when it is requested.

3.1.8 MMPAppTNO

MMPAppTNO extends MuPIF's Application class and the following functions of MuPIF's Application class have been re-implemented:

General:

- _init_

Getters:

- getProperty
- getMesh
- getField
- getAssemblyTime

Setters:

- setProperty
- setField

Simulation methods:

- solveStep

In addition to these functions, there are a few local Python support functions:

- IsNumber
- ReadComsolMesh
- ReadComsolValues
- ReadVTKMesh
- WriteMesh
- Field2Comsol
- UpdateLtime

There are also 3 MATLAB scripts:

- Write_vtkMesh
- create_mphtxt
- run_Comsol_model

3.1.9 Implementation details

Figure 6 provides a graphical overview of the implementation details. The grid initialization is performed in 'MMPAppTNO._init_', hence directly when the MMP-TNOAPP is started on the local machine. Three empty unstructured meshes are created first. COMSOL and MATLAB are started, the COMSOL model of the LED device model is loaded. The complete mesh is written to file (mphtxt format, create_mphtxt.m), together with two local meshes (lens and reflector) which are written to vtk format (write_vtkMesh.m). The complete mesh is read and stored by support.ReadComsolMesh. The vtk-files are read and stored by support.ReadVTKMesh. These two submeshes are used to create two empty fields for the thermal absorption.

The 'MMPAppTNO.setField' method stores the absorption fields from the MMP-TracerAPI as MMP-TNOAPI fields. It also writes two input files for COMSOL (support.Field2Comsol function); one for the lens, containing the volumetric thermal absorption, and one for the reflector, containing the surface thermal absorption. The files contain the center coordinate of the elements with a thermal absorption value.

The 'MMPAppTNO.solveStep' method starts with support.WriteMesh in which it writes a file with all the coordinates of the mesh. This file is required by COMSOL to export the temperature data in the correct order/sequence. COMSOL with MATLAB is started and the run_Comsol_model.m script is executed. This script runs the thermal circuit model (microstructure model: TRC.m) first, followed by the COMSOL LED device model (with the calculated effective conductivity of the thermal circuit model as input). The stationary simulation is automatically followed by the transient (cooling down) calculation. The time dependent temperature distribution is automatically written to file for every calculated time step.

'MMPAppTNO.getAssemblyTime' can be used to retrieve the exact time values at each step from the simulation.

'MMPAppTNO.getField' can then be used to retrieve the temperature distribution at the time points obtained by 'MMPAppTNO.getAssemblyTime', but also at interpolated times. With 'MMPAppTNO.getField(…).giveValue(…)', the temperature within a specific node/vertex can also be retrieved. The getField function uses support.ReadComsolValues to read the temperature values from file at that certain time point. The 'MMPAppTNO.getField' can also be used to retrieve the absorption fields. The time in getField function should then be set to 'None' and the appropriate FID should be used.

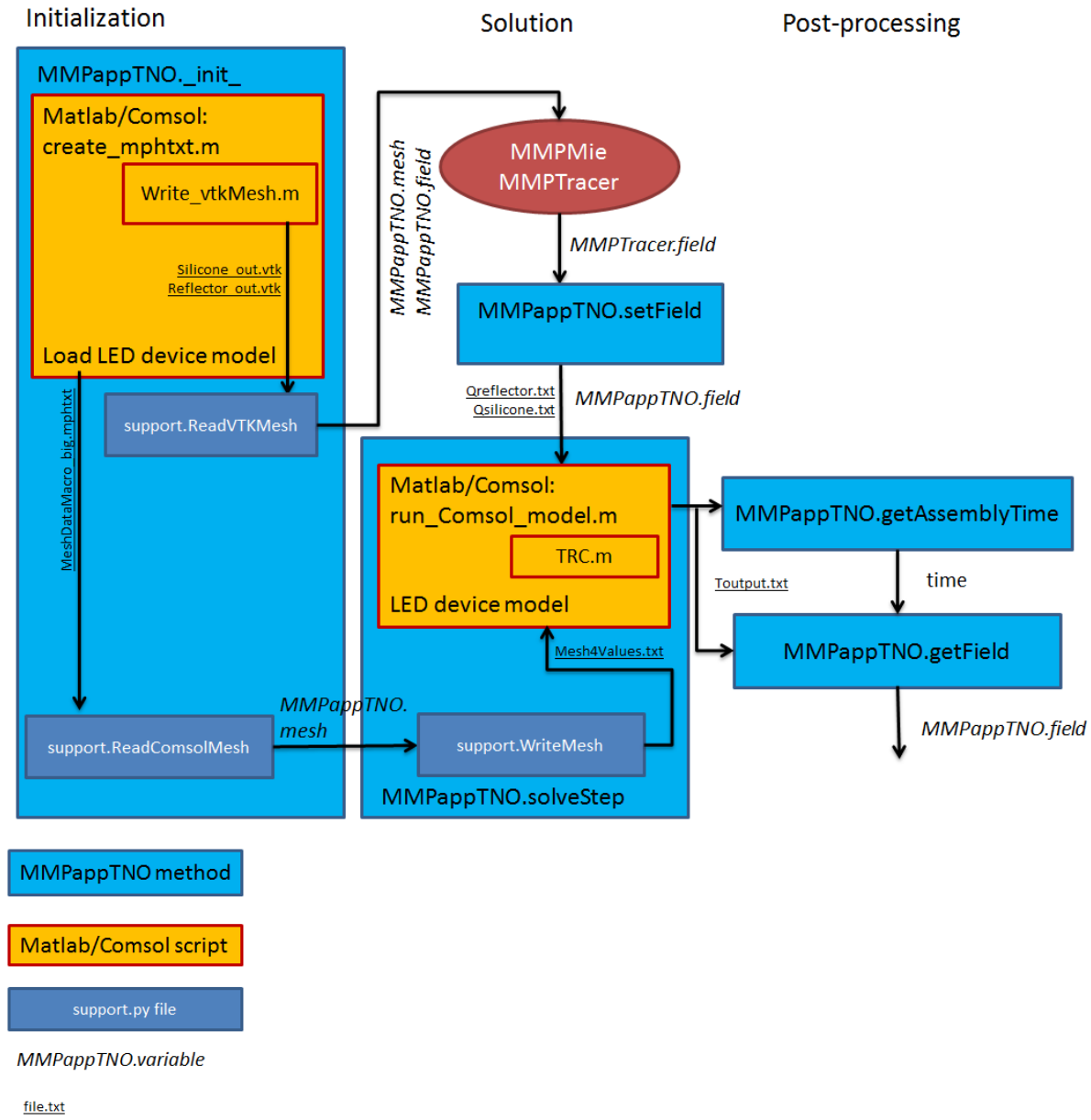


Figure 6: implementation overview

3.2 Optical modelling software

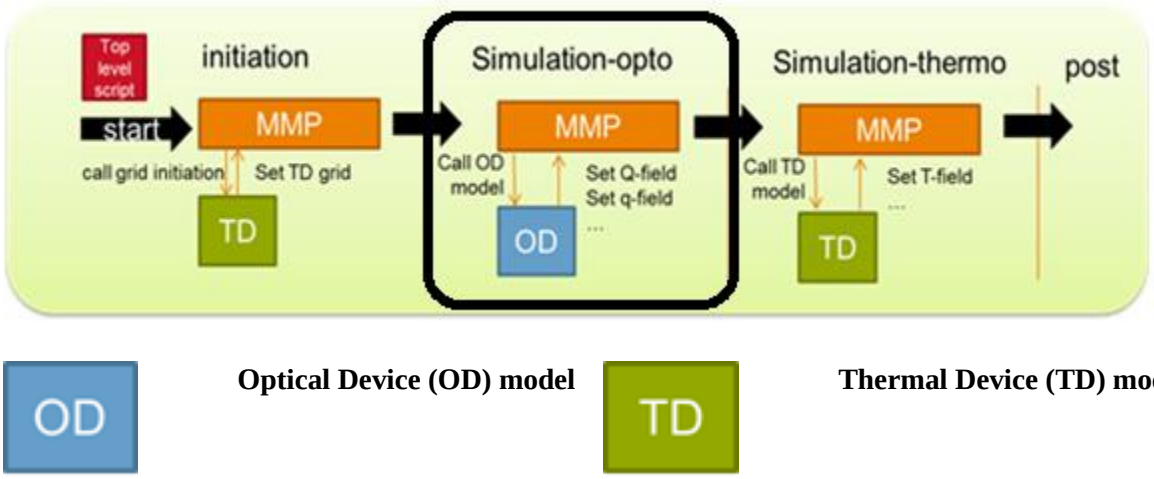


Figure 7: Optical simulations covers the middle part in the simulation chain

As depicted in Figure 7, the optical modelling software handles the middle part in the simulation chain. As inputs, it uses the same fields that are initialised in the thermal device model. As outputs, it gives two heat absorption fields that are then used in the thermal device model. The optical modelling software consists of micro-level MMP-MieGenerator and macro-level MMP-Raytracer models. The following subsections shortly summarize these two models and introduce the APIs for both of them: MMP-MieAPI and MMP-TracerAPI, respectively. The source codes of these APIs have been released in GitHub (https://github.com/ollitapa/MMP-TracerApi and /MMP-MieApi) as open source under Apache License, Version 2.0. Installation instructions, example scripts and user guide can be found in the same place.

3.2.1 MMP-MieAPI for the MMP-MieGenerator

Table 2: Table 2: MMP-MieAPI summary

Name and model	MMP-MieAPI
Owner	VTT
Type of implementation (native/indirect)	Native
Type of local OS	Linux
Security	Local or SSH-secured traffic
Error handling	Basic
Reporting	Logging-module enabled
Application classes	MMPMie

The MMP-MieGenerator model provides the effective scattering cross-section and the cumulative scattering probability distribution functions as a function of the wavelength for the defined particle size distribution. This data is used to model Mie scattering events in the macro-scale model (i.e., MMP-Raytracer, see the next subsection). The calculation of probability distributions requires information about the particles and the host material. As inputs, MMP-MieGenerator needs the wavelength range and number of points to linearly discretize the range, refractive index of particles, log-normal particle size distribution, and the refractive index of the host material.
As output, the model generates two HDF5-formatted files containing the effective scattering cross-section and the cumulative scattering probability distribution functions for given input wavelengths. The scattering cross-sections (μm^2) together with particles' densities are used in the optical device level model to determine the mean free path of light rays in particle mixtures, while the cumulative scattering distribution function data is used to calculate a new ray direction due to Mie scattering events. Each discrete wavelength defined by the input parameter "wavelength range" has a unique scattering function.

To support distributed simulations, MMP-MieGenerator was integrated with MuPIF by creating MMP-MieAPI software in Python. Figure 8 depicts the MMP-MieAPI package's Python files. The arrows illustrate import relations between the files. The `__init__.py` file is required to make Python treat the directory as containing packages. `MMPMie.py` contains the definition for the MMPMie class, which is described with more details later. `initialConfiguration.py` contains functions that check that all the required inputs (e.g., Properties) are present. `objID.py` contains definitions for objectIDs. `mie` is a subfolder containing `mieDatabase`, `mieGenerator`, `bhmie_herbert_kaiser_july2012`, and `scatteringTools` files. The data base defined in `mieDatabase.py` is intended for permanent storage for Mie data files, in order to avoid generating the same data files again and again with functions defined in `mieGenerator.py` and the Bohren and Huffman code for Mie theory in `bhmie_hderber_kaiser_july2012.py`. The Mie data files are saved in HDF5 format [https://www.hdfgroup.org/HDF5/], which was chosen for its speed and available read/write packages. `mieServer.py` contains functions to start the MMP-MieGenerator program, either a single instance on a local machine, or several distributed instances using JobManager from MuPIF. `killMieServer.py` is a script to kill all running MMP-MieGenerator programs from the local machine.

The main class of MMP-MieAPI, MMPMie, inherits from the MuPIF Application class. MMPMie has containers for Properties and Fields, as well as for outputs.

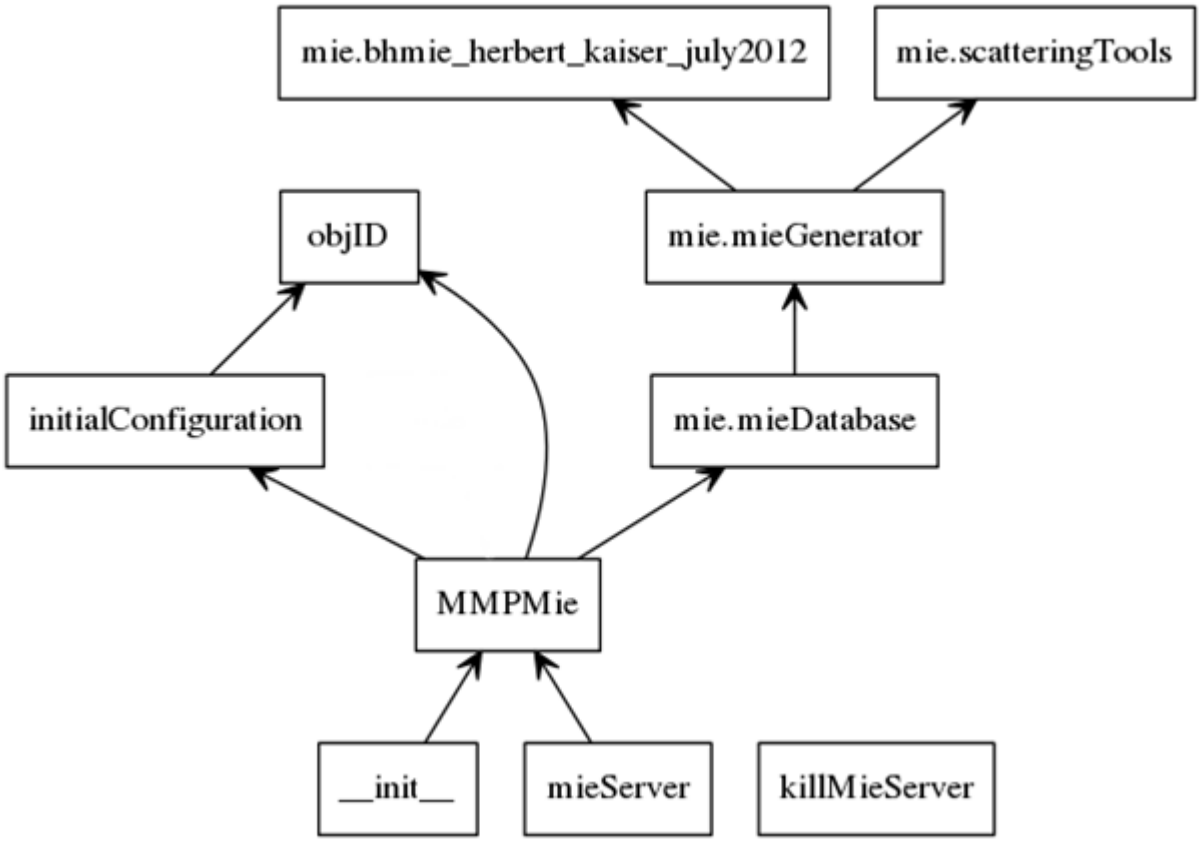


Figure 8: MMP-MieAPI package diagram

The following functions of the MuPIF Application class have been overwritten in MMPMie:

Getters:

- getField
- getMesh
- getCriticalTimeStep
- getAssemblyTime
- getApplicationSignature

Setters:

- setField
- setProperty

Simulation methods:

- solveStep
- isSolved
- terminate
- wait

3.2.1.1. Overview of implemented methods

We focus on a few key aspects on couple of functions.

setField and setProperty:
The fields and properties are stored in pandas series objects. Each property is stored as a copy for each timestep simulated. As new timesteps are simulated the previous solution is internally copied to current timestep. Pandas [http://pandas.pydata.org/] is a well-established package with fast data containers and versatile manipulation methods.

solveStep:
The `solveStep` of MMPMie application uses the well-known Bohren and Huffman code for Mie theory to calculate the cumulative phase functions and scattering cross sections. The calculations are done natively in Python with numpy package. Multiprocessing module is used for speeding up the process by running the code parallel on multiple processor cores. The calculated data is then packaged to the two properties: PID_InverseCumulativeDist and PID_ScatteringCrossSections, respectively.

3.2.2 MMP-TracerAPI for the MMP-Raytracer

Table 3: MMP-TracerAPI summary

Name and model	MMP-TracerAPI
Owner	VTT
Type of implementation (native/indirect)	Indirect
Type of local OS	Linux
Security	Local or SSH-secured traffic
Error handling	Basic
Reporting	Logging-module enabled
Application classes	MMPRaytracer

MMP-Raytracer is a simple engine for tracing rays with Monte Carlo method. This ray tracing software by VTT is implemented in C++ and it takes its input parameters in a JSON-formatted file. A snapshot of this JSON file is presented in Figure 9. JSON [http://www.json.org/] is a well-known and widely used machine readable data format. We chose it because of widely available read/write packages. For coupled opto-thermal simulations, MMP-Raytracer has been integrated with MuPIF by creating MMP-TracerAPI software in Python. Figure 10 presents the structure of the MMP-TracerAPI package. The arrows illustrate import relations between the files. The `__init__.py` file is required to make Python treat the directory as containing packages. `mmpraytracer.py` contains the definition for the MMPRaytracer class, which is described with more details later.

`initialConfiguration.py` contains functions for checking and loading the inputs (e.g., Properties). `objID.py` contains definitions for objectIDs. `hdfSupport.py` contains a function to write the Mie scattering data to a HDF5 format that is recognized by the MMP-Raytracer. `interpolationSupport.py` contains functions to interpolate the fields and properties between solved time steps. `mmpMeshSupport.py` includes mesh-related functions like converting point-data to mesh-based cell data. `vtkSupport.py` contains functions to read data from a vtk-formatted file. `generalSupport.py` contains some common helper functions. `tracerServer.py` contains functions to start the MMP-Raytracer program, either a single instance on a local machine, or several distributed instances using JobManager from MuPIF. `killTracerServer.py` is a script to kill all running MMP-Raytracer programs from the local machine.


```
{
    "general": {
        "maxIterations": 1000,
        "maxThreads": 10
    },
    "sources": [
        {
            "name": "Source1",
            "type": "IsotropicVolume",
            "rays": 100000,
            "insideOf": "ChipActive",
"wavelengths": [412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424,
"intensities": [  8.82221890e-05,   1.05760335e-04,   1.24095669e-04,
        }
    ],
    "geometry": [
        {
            "name": "Air",
            "location": [
                0.0015,
                0.0015,
                0.00019
            ],
            "type": "Sphere",
            "radius": 0.01,
            "material": "Air",
            "insideof": "None",
            "surfaceProperties": [],
            "detectors": [
                {
                    "type": "AllDetector",
                    "name": "LEDLayerNoSilicone"
                }
            ]
        },
        {
            "name": "Case",
            "type": "Cuboid",
            "wx": 0.003,
            "wy": 0.003,
```


Figure 9: Snapshot from MMP-Raytracer's input JSON file

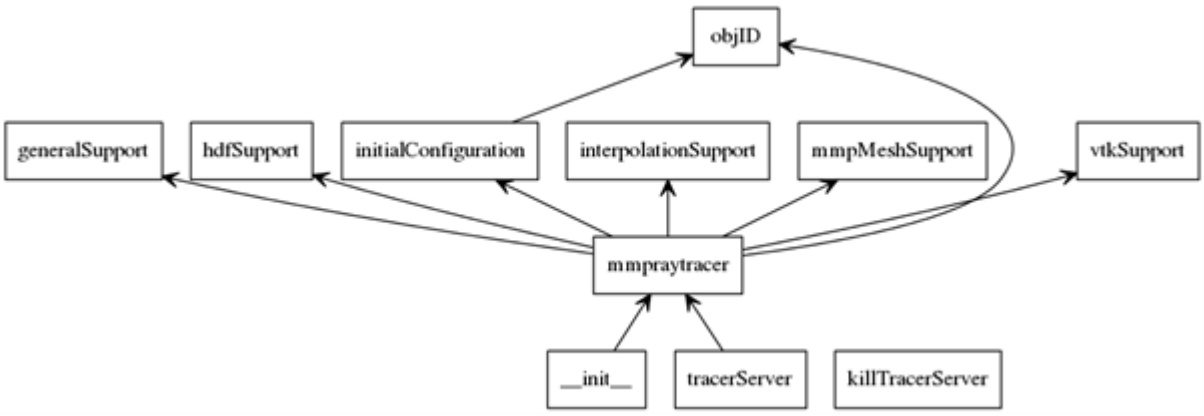


Figure 10: MMP-TracerAPI package diagram

The main class of MMP-TracerAPI, MMPRaytracer, inherits from the MuPIF Application class. It has containers for Properties and Fields. Properties and Fields can be set and get by the corresponding get/setProperty functions.
The following functions of the MuPIF Application class have been overwritten:

Getters:

- getField
- getMesh
- getCriticalTimeStep
- getAssemblyTime
- getApplicationSignature

Setters:

- setField
- setProperty

Simulation methods:

- solveStep
- isSolved
- terminate
- wait

MMP-Raytracer takes the following input parameters:

- Optical properties of the phosphor cup walls
  - Type: diffuse, mirror
  - Absorbance
- Optical properties of the chip substrate
  - Refractive index
- Optical properties of the chip active area
  - Refractive index
- Optical properties of the phosphor mix
  - Array of number of the particles in μm^3 of the particle types used.
  - Excitation spectrums of the particles
  - Emission spectrums of the particles
  - Refractive index of the host silicone
- Properties of the optical source
  - Spectrum of the source
  - Number of rays to trace
- Input coupling:
  - Scattering properties of the phosphor particles from MMP-MieGenerator.

The input parameters in MMP-TracerAPI are presented by MuPIF Properties and Fields. Each Property and Field has a certain ID. The following Table 4 presents the Property and Field IDs for MMP-TracerAPI, as well as their relation to the MMP-Raytracer's JSON input file.

Table 4: MMP-TracerAPI's Property and Field IDs

Property/Field ID	Description	JSON parameter index	ValueType
PID_RefractiveIndex	Refractive index of cone	['materials'][1]['refractiveIndex]	Scalar
PID_NumberOfRays	Number of rays to be traced	['sources']['rays']	Scalar
PID_LEDSpectrum	LED spectrum	['sources']['wavelengths'] and ['sources']['intensities']	
PID_ParticleNumberDensity	Particle number density	['materials'][3][''particleDensities']	Scalar
PID_ParticleRefractiveIndex	Particle refractive index	Used in MieApplication	Scalar
PID_EmissionSpectrum	Emission spectrum	['materials'][3]['cumulativeEmissionSpectrumFilenames']	
PID_ExcitationSpectrum	Excitation spectrum	['materials'] [3]["excitationSpectrumFilenames"]	
PID_AsorptionSpectrum	Absorption spectrum	['materials'] [3]["absorptionSpectrumFilenames"]	
PID_ScatteringCrossSections	from MMP-MieAPI	mieData.hdf5	
PID_InverseCumulativeDist	from MMP-MieAPI	mieData.hdf5	
FID_HeatSourceVol	HeatSourceVol		FT_cellBased
FID_HeatSourceSurf	HeatSourceSurf		FT_cellBased

`solveStep` function generates the input JSON file based on the Properties and Fields, and launches the MMP-Raytracer software. As output parameters, we get:

- Absorption distribution for phosphor volume
- Absorption distribution for phosphor cup side surfaces
- Ray set in VTK-file format (optional)
- Ray source file (optional).

3.2.2.1. Overview of implemented methods

We focus on a few key aspects on couple of functions.

setField and setProperty:

As in the MMP-MieAPI, the fields and properties are stored in pandas series objects. Each property is stored as a copy for each timestep simulated. As new timesteps are simulated the previous solution is internally copied to current timestep.

getField and getProperty:

The getter methods for fields and properties return the queried data from the pandas storage. MuPIF API description also requires interpolation between timesteps. The timesteps simulated are stored and the interpolation of data between them is provided by `interpolationSupport.py`. Currently we have implemented a linear interpolation only.

solveStep:

The solveStep of the MMP-TracerAPI is divided in a few steps illustrated in Figure 11. First we check that we have all the required fields, properties and functions set using internal functions `checkRequiredFields()`, `checkRequiredParameters()` and

`checkRequiredFunctions()` from `initialConfiguration.py`. Stored properties and fields from previous timestep are copied to current timestep by internal function `_copyPreviousSolution()`. After that a JSON input file is prepared for the MMP-Raytracer program. We map the properties from the API to the JSON format in `_writeInputJSON()` and write to disk. Data received from the MMP-MieAPI is transformed to format and files that the MMP-Raytracer accepts. This is shown in function `_getMieData()`.

The main tracing routine is started with Popen class from subprocess module. We use a background thread from the threading module to start the Popen process so that we can monitor its process in the background. If the `runInBackground` argument of the `solveStep` is set we return to the caller. The background thread will wait for the tracing process to complete and initialize the post processing functions to read the output of the MMP-Raytracer and convert it to MuPIF-compliant format.

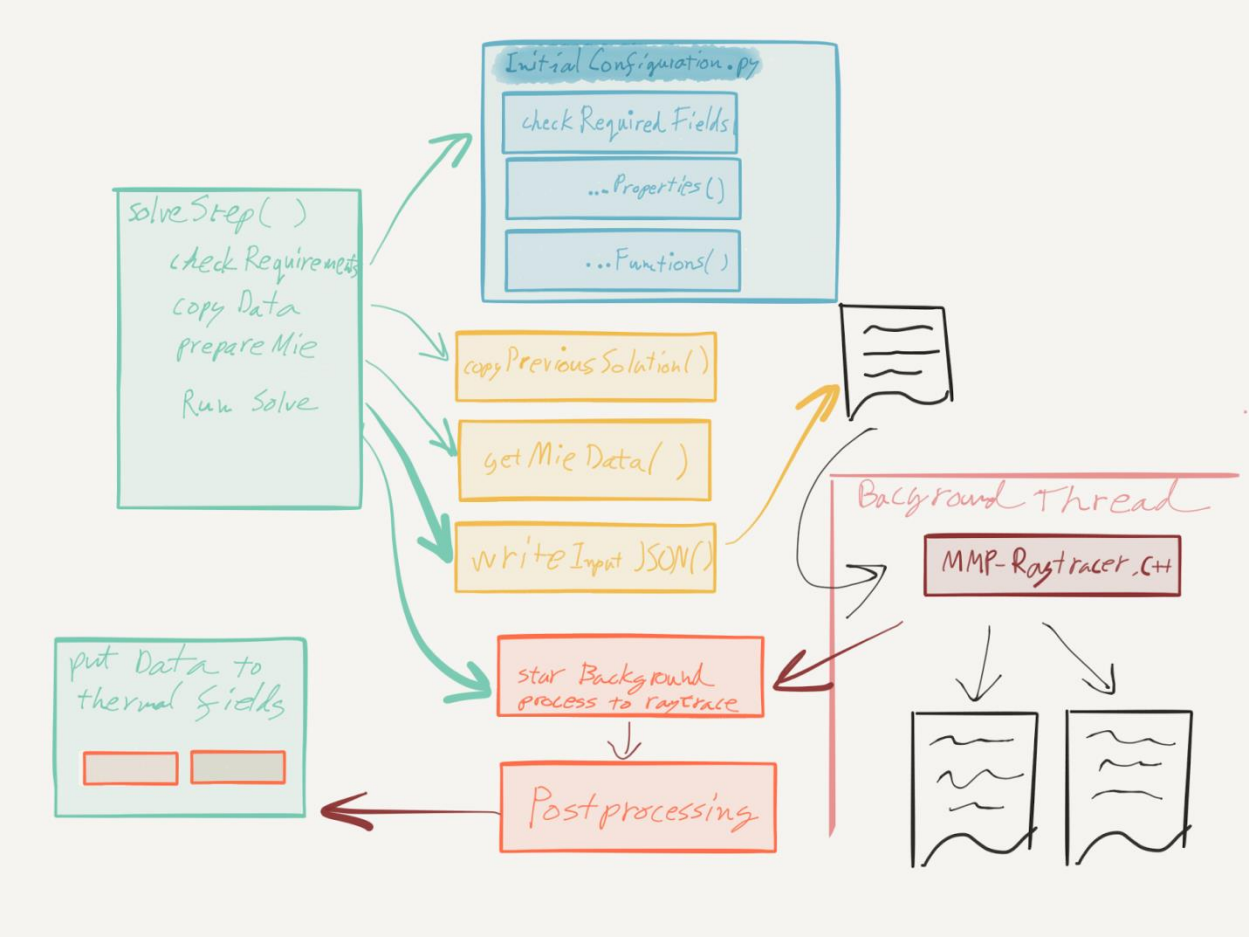


Figure 11: The internal steps of the MMP-TracerAPI's solveStep function

Post processing:

Post processing of the results from the MMP-Raytracer involve reading vtk-formatted data and localising point data to a mesh. The data reading is implemented in `vtkSupport.py` file with reading methods for a ray file and absorption point data.

3.3 Running distributed simulation chain

Figure 12 and Figure 13 below present an example top level simulation script illustrating the use of developed APIs distributed across two physically separated computers. The first step is to initialise the 3 instances of API classes, i.e., MMPMie, MMPRaytracer and MMPAppTNO.
The second step is to set the properties and fields. After that, the `solveStep()` functions of each application can be called. Shared fields are updated automatically in all applications if they are changed in any of the applications. In the end, the results are gathered in the top level script and plotted into vtk files.

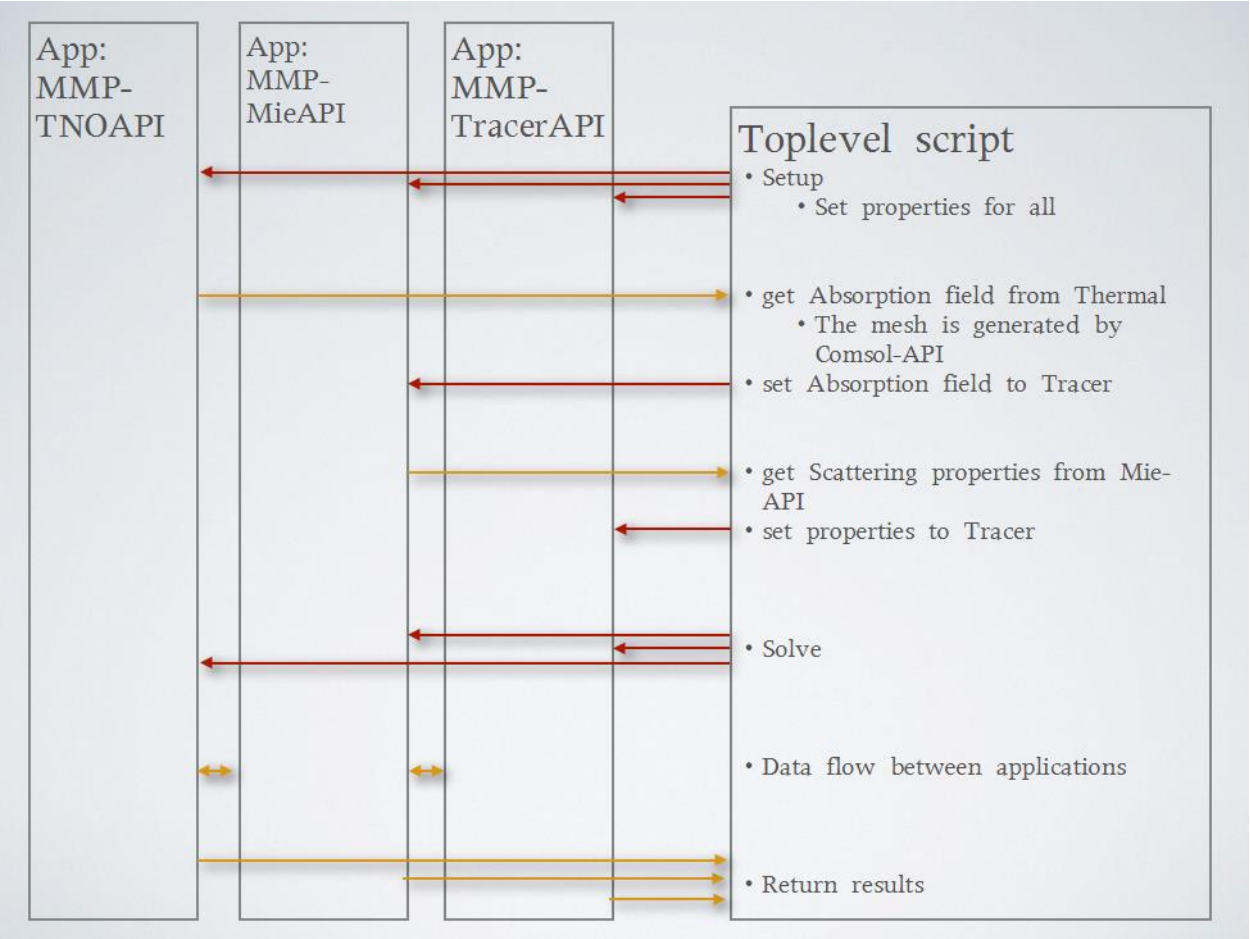


Figure 12: schematic top level script

The code used as an example top-level Python script is shown below:

```
import Pyro4
Pyro4.config.SERIALIZERS_ACCEPTED = ['pickle', 'serpent', 'json', 'marshal']
Pyro4.config.SERIALIZER = 'pickle'
from mmp_tracer_api import objID


import ex_em_import


from mupif import *
from vars import *
```

```
ns = Pyro4.locateNS(port=9091)
appTNO = Pyro4.Proxy(ns.lookup('TNO'))
mieApp = Pyro4.Proxy(ns.lookup('Mie'))
tracerApp = Pyro4.Proxy(ns.lookup('Tracer'))


fHeatSurf = appTNO.getField(FieldID.FID_HeatSourceSurf, 0)
fHeatVol = appTNO.getField(FieldID.FID_HeatSourceVol, 0)

pScat = mieApp.getProperty(PropertyID.PID_ScatteringCrossSections, 0,
                           objID.OBJ_PARTICLE_TYPE_1)

pPhase = mieApp.getProperty(PropertyID.PID_InverseCumulativeDist, 0,
                            objID.OBJ_PARTICLE_TYPE_1)

vDens = 0.00000003400
pDens = Property.Property(value=vDens,
                          propID=PropertyID.PID_ParticleNumberDensity,
                          valueType=ValueType.Scalar,
                          time=0.0,
                          units=None,
                          objectID=objID.OBJ_CONE)

pRays = Property.Property(value=100,
                          propID=PropertyID.PID_NumberOfRays,
                          valueType=ValueType.Scalar,
                          time=0.0,
                          units=None,
                          objectID=objID.OBJ_CONE)

aabs = Property.Property(value=ex_em_import.getAbs(),
                         propID=PropertyID.PID_EmissionSpectrum,
                         valueType=ValueType.Scalar,
                         time=0.0,
                         units=None,
                         objectID=objID.OBJ_CONE)

EffCond = 1.2
EffCondProp = Property.Property(EffCond,
                                PropertyID.PID_EffectiveConductivity,
                                ValueType.Scalar,
                                None, "[W/m*K]", 1)

tracerApp.setProperty(pScat, objID.OBJ_PARTICLE_TYPE_1)
tracerApp.setProperty(pPhase, objID.OBJ_PARTICLE_TYPE_1)
tracerApp.setProperty(pDens)
tracerApp.setProperty(pRays)
tracerApp.setProperty(aabs)
```

```
tracerApp.setField(fHeatSurf)
tracerApp.setField(fHeatVol)

appTNO.setProperty(EffCondProp, 1)

mieApp.solveStep(0)
tracerApp.solveStep(0, runInBackground=False)

fHvolnew = tracerApp.getField(FieldID.FID_HeatSourceVol, 1)
fHsurfnew = tracerApp.getField(FieldID.FID_HeatSourceSurf, 1)

appTNO.setField(fHvolnew)
appTNO.setField(fHsurfnew)

appTNO.solveStep(0)

# counter=1
# while(True):
#       try:
#               (t,xxx) = appTNO.getAssemblyTime(counter)
#               fTemp = appTNO.getField(None,t)
#               print t, fTemp.giveValue(8098)
#               tmp = 'ftemp{}.vtk'
#               fTemp.field2VTKData().tofile(tmp.format(counter))
#               counter = counter + 4
#       except:
#               break
```

Figure 13: actual top level script

4 Summary, conclusion and recommendations

This deliverable described the development of three different Application Interfaces (APIs) within the MuPIF framework. A simulation chain for a multiphysics and multiscale thermo-optical simulation of an LED device was already defined in earlier deliverables. The actual models were also already developed at an earlier stage during the project (D2.3 and D2.4). The current challenge was to interface with these different models, such that certain model properties could be set, simulations could be run and results could be extracted and transferred. Although this is only just a very coarse picture, detailed requirements for the APIs result from the definition of the simulation chain and these were described in D1.1 and D2.3. Methods, as inherited from the MuPIF application class, were programmed such that the top level user obtained all required handles. The ultimate proof of the functionality of the APIs was provided by a successful distributed run of the simulation chain (milestone 3), where the models were interfaced individually and where data and variables were transferred between the subsequent models/API.

Within this project, together with the APIs, also MuPIF itself was still under development. This meant that when the API development started, it was not easy for the API developers to understand how MuPIF would develop and how it exactly should be used. MuPIF has grown mature since, which is a great advantage for future API developers. We would recommend them two things to speed up their API development. First, make sure to get a very good understanding of what the possibilities of MuPIF are and how it should be used. The current project members and especially the developers of MuPIF (CTU) can help with that. Second, when sufficient understanding about MuPIF has been obtained, the developers of the different APIs within the envisaged chain should define very clearly which data should be transferred where and how this should be defined (which class). Hence, when transferring data from one API to the other, the ‘exported’ class of API 1 should be identical to the ‘imported’ class of API 2. As the API developers also have a thorough knowledge of how the models work and what they can steer, change or retrieve, they should be able to find a good solution and thereby set requirements for the APIs. It has to be kept in mind that it is not necessary to provide the top level user with all possible model handles; it is only necessary to provide the user with the ones he is intending to change. Finally, as we succeeded in setting up a distributed simulation, it may be an idea to start with setting up a real distributed simulation network with a name server and some ‘empty’ APIs. This will create API development interaction and makes it easier to perform testing.