	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	1/27

Euso Simulation and Analysis Framework

May 13, 2002 - Version 1.0.1

D. De Marco¹ and **M. Pallavicini**²


Istituto Nazionale di Fisica Nucleare & Università di Genova
Italy

Abstract

This document describes the goals, the design guidelines, and the implementation details of the Euso Simulation and Analysis Framework, ESAF. This framework aims to be the standard software environment for the Euso experiment. It is written in C++ and it is based on the root package.


¹e-mail: Daniel.DeMarco@ge.infn.it

²e-mail: Marco.Pallavicini@ge.infn.it

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	2/27

Contents

Introduction	3
1 Design	4
1.1 LightToEuso	7
1.2 EusoDetector	9
1.2.1 Detector Optics	9
1.2.2 Electronics	12
2 Implementation	13
2.1 Optics	13
2.1.1 YPOpticalSystem	13
2.1.2 FastFocalPlane	16
2.2 Electronics	16
2.3 Miscellaneous	18
2.3.1 Config files and EusoConfigurable	18
2.3.2 Root	20
2.3.3 Makefiles & Directory Structure	21
3 Installation	24
3.1 Getting the sources	24
3.1.1 Download from web	24
3.1.2 Download via CVS	24
3.2 Compilation	25
References	27

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	3/27

Introduction

ESAF stands for Euso Simulation and Analysis Framework. It is a software framework developed for the Extreme Universe Space Observatory, EUSO. In our intentions, when it will be completed, it will provide the Euso scientists with a consistent framework for the whole process of *data-simulations* and *data-analysis*, from the simulation of the primary particle interaction in atmosphere, to the transport of light to the euso optical pupil, to the detector response simulation and finally to the reconstruction and the physical analysis.


We designed ESAF so that each one of the above steps could be run individually and independently from the other ones. With this approach it is possible to run the same reconstruction and analysis code for the real data and for the simulated ones. Moreover, it is also possible to run single parts of this chain and check quantitatively the differences between different configurations of the detector or different approximations for the physical processes involved.

We started this project developing the general framework and application and implementing the section of the detector response simulation. At this stage, the simulation and analysis chain is foreseen and approximately designed, but not implemented yet. The only part that is rather complete, albeit with many rough approximations, is the detector response simulation.

This document is structured as follows: in the first section we present the general/abstract design of ESAF. In the following chapter we describe the implementation that we developed, dwelling upon the description of the simulation of the detector response. The last chapter is a sort of walk-through manual explaining how to get, compile and run ESAF.

Esaf is written in C++ language [1] and it is designed using Object Oriented (OO) technology [2]. We think that this is nowadays a natural choice and it is actually the only way to achieve the high degree of modularity and flexibility we aim to. In this document some OO programming concepts and terminology are used. The reader that is completely unfamiliar to them should refer to one of the many good books on this topic. Anyway, OO technicalities will be kept to the minimum.

ESAF is based on the root package [3], developed at CERN for high energy physics applications at the Large Hadron Collider. Section 2.3.2 describes the way ESAF and root interact.

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	4/27

1 Design

ESAF is built with an "onion-like" structure, following the general guidelines of OO programming. Within this frame the outer layers are more general and abstract than the inner ones and going deeper inside the structure you find that the things become step by step more and more *real* and specialised. This is accomplished with a massive use of abstract interfaces and *factories* [5]. In this way outer layers need to know only the interface of objects in the inner layers (what these objects do) without knowing what the actual implementation will be.

Esaf has been designed following the general ideas proposed by E. Plagnol and G. Dalì Staiti [4]. The conceptual scheme we started from is depicted in fig. 1.

Let's now start the description of ESAF. We will follow the onion structure starting from the topmost layer that is `EusoApplication` (see fig. 2). As you can see the application is divided into four subsystems:

- `LightToEuso` is the subsystem that encapsulates all the physical phenomena occurring outside EUSO that can produce photons on the entrance of the detector (extensive air showers, lightning, meteorites, lidar...)
- `EusoDetector` encapsulates the detector response simulation. As you can see from the picture it is internally subdivided in two more branches: `DetectorTransportManager` and `EusoElectronics`. These two represent respectively the tracking of photons from the entrance to the focal surface and electronics simulation;
- `Reconstruction` encloses algorithms for event reconstruction;
- `Analysis` encloses algorithms for the physics analysis.

Each one of the above subsystems knows nothing about the others, i.e. they are very loosely coupled. The interconnection is done via `EusoApplication` that controls them and forwards the data from one to the other. During the run of the simulation `EusoApplication` asks `LightToEuso` for a `PhotonsOnPupil` (that is a list of photons on the entrance of the detector). `LightToEuso` produces this object in some way: it can read the photons at the entrance stored in a file by a previous simulation, it can simulate some showers, generate the photons and propagate them up to the entrance of the detector or it can simulate some other light generating process (lighting, lidar, meteorites...). Once produced it delivers the `PhotonsOnPupil` to `EusoApplication`. The important thing in this process is that `EusoApplication`

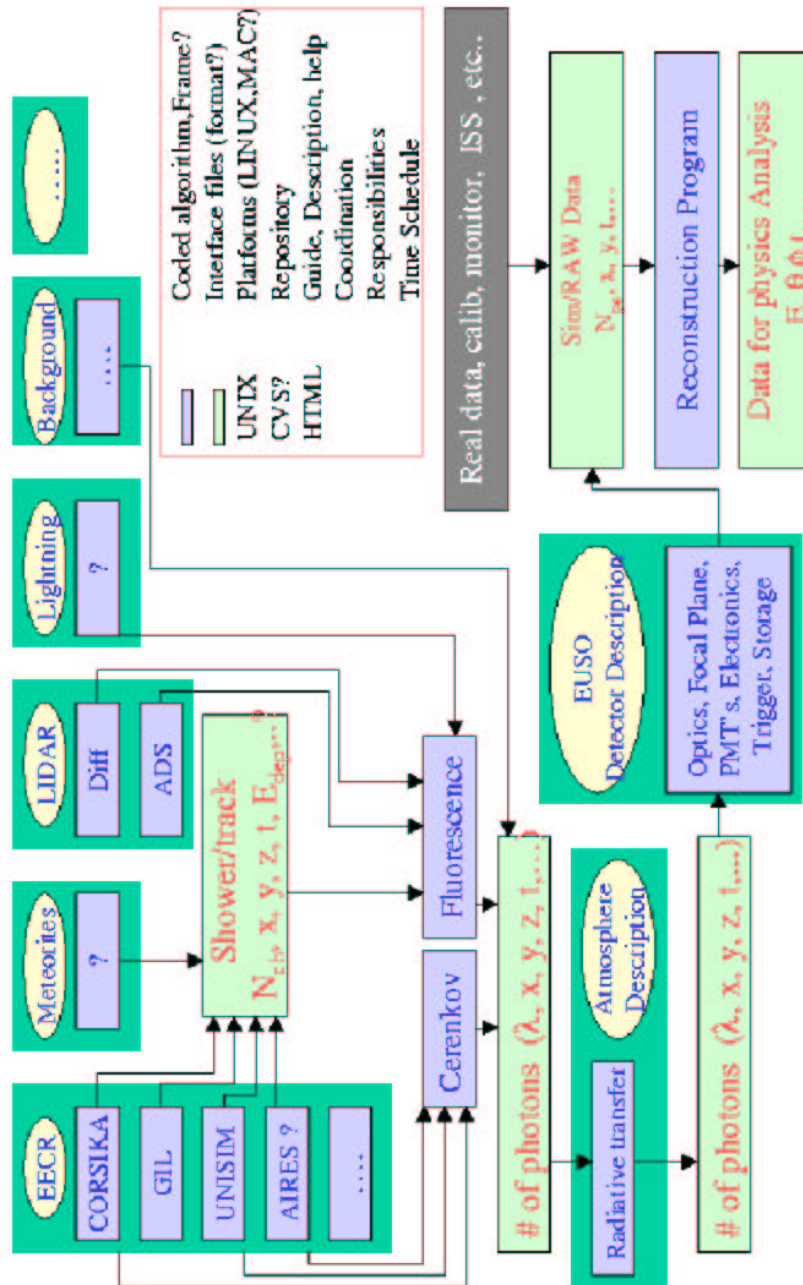


Figure 1: General conceptual scheme of the Euso Simulation software. This picture is not a class diagram. It is the logical scheme which ESAF is designed to implement.

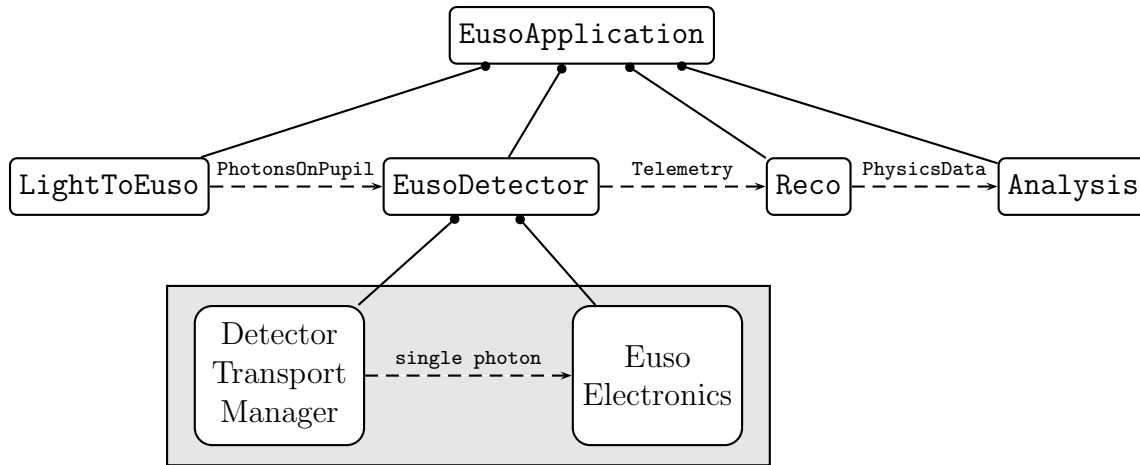


Figure 2: **EusoApplication**. $A \text{---}\bullet B$ means B knows A, that is B has a pointer, reference or something to A. $A \text{---}\rightarrow B$ means that A produces a data object that is handled to B (usually via a third object in the upper level).


does not know how the photons are produced and propagated, it only knows that exist an object of type **LightToEuso** that can give it the list of photons.

Once the photon list has arrived to **EusoDetector**, it handles it to the **DetectorTransportManager** that propagates each one of them through the detector (main optical system, focal surface mapping, optical adaptors) up to the photo-multiplier surface. Here the photons are taken by **EusoElectronics** that simulates the electronics chain and produces a **Telemetry** object. This object contains the complete detector response to an event. Eventually, there will be a **Telemetry** object with the same data format that will arrive to the ground station from the ISS.

At this point the process should continue and **Telemetry** should go to **Reconstruction** and then to **Analysis** but this part of the application has not yet been written.

All the subsystems (or modules) mentioned above are implemented as objects that are instances of C++ classes. In this way, thanks to the features of OO programming, it is easy to get many different versions of each module that differ from the point of view of detailed implementation while preserving their interface. For example, in ESAF it is possible to have many different ways of simulating the air showers, having the possibility to choose among the available options at run time.

Also we would like to stress that, thanks to the accurate analysis and design, you can begin and end your simulation in any point between or inside

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	7/27

the above subsystems saving your partial results that can be later on retaken for further investigation or to continue the simulation. You can rerun some part of the simulation chain with different parameters. You can even change the *kind* (that is implementation) of the subsystem you are using without the others noticing. These features open a wide range of possibilities. Just to mention some examples you can do a whole simulation from showers to reconstruction then change the type of optical adaptor you are using, rerun the simulation starting from the detector response simulation and cross-check the results. Otherwise you can generate some showers, produce the photons and then try different models for the light propagation in atmosphere. You can do all these analysis within the same application and this provides you with the same user interface, guarantees you that the parameters are coherent in the whole process, and allows comparison among options or modules or parameter value on event basis³

Now let's continue the description of ESAF going one level deep and analysing the internal design of the two already built subsystems.

1.1 LightToEuso

The `LightToEuso` subsystem encapsulates all the physical phenomena occurring outside EUSO that can produce photons on the entrance of the detector. As you can see from fig. 3 it is divided into three subsystem: `ShowerSource`, `LightSource` and `RadiativeTransfer`.

The first subsystem, `ShowerSource`, simulates extensive air showers. It is an abstract interface that provides `LightToEuso` with a `ShowerTracks` object. This object is created in a different way by each implementation:

- `CorsikaShowerSource` is a wrapper that, after reading some parameters from the user, calls the CORSIKA [6] executable and imports the data it produces providing a full, reliable and widely accepted simulation of extensive air showers;
- `UnisimShowerSource` is also a wrapper that runs the UNISIM⁴ executable and thus provides a faster way to create showers (moreover UNISIM can generate also neutrino induced showers);
- `FileShowerSource` provides the possibility to read previously generated shower tracks. It can read both extern generated files (aires,

³This is one of the main features of this design. The user can simulate an event with some configuration, then change some parameter or module, re-run the program on the same event and check for the differences in the simulation results.

⁴UNISIM is a shower simulation package developed in Florence [7].

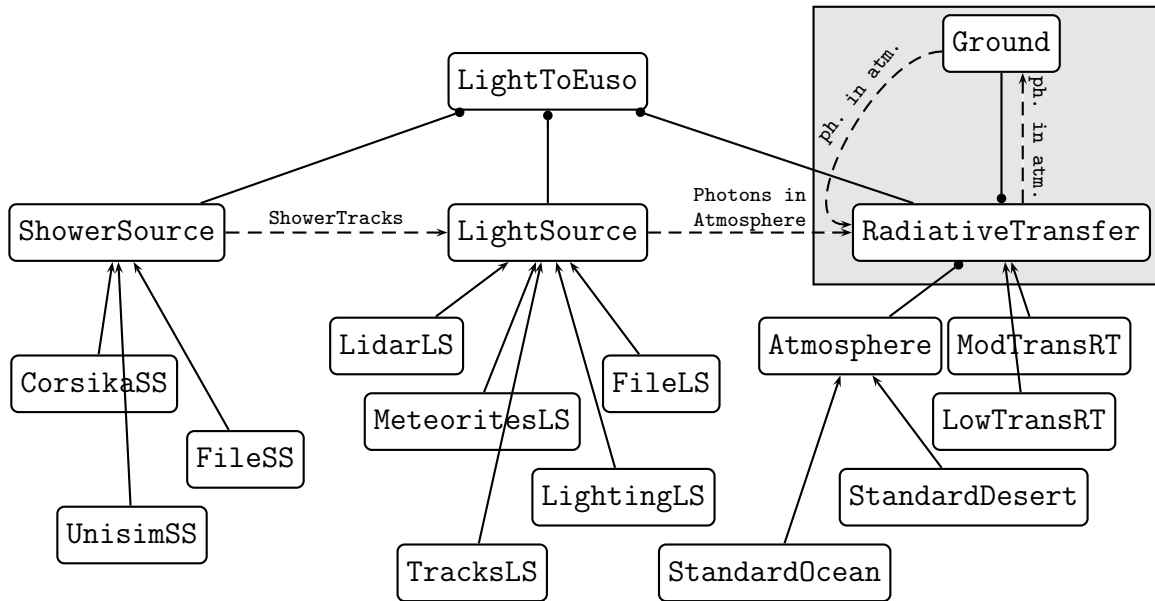



Figure 3: `LightToEuso`. In addition to the conventions used in the previous figure, here $A \longrightarrow B$ means that A inherits from B (this means A is a different implementation of the B interface).

CORSIKA, UNISIM...) and files generated by a previous session with ESAF and saved in its own format (see 2.3.2);

Once the tracks are generated, it is time to simulate the photon generation process in atmosphere. This is the task performed by `TracksLightSource`, that produces a list of the photons generated in the various points of the atmosphere along the shower trajectory. This object generates both the fluorescence photons and the Čerenkov ones.

`TracksLightSource` belongs to the family of objects inheriting from `LightSource`. All these objects generate a list of photons in atmosphere and represent the various possible light-generating processes in atmosphere, the names are self-explanatory: `LightningLightSource`, `MeteoritesLightSource`, `LidarLightSource`, `TracksLightSource` and `FileLightSource`. This last one provides the possibility to read a previously saved ESAF file containing the `PhotonsInAtmosphere` object.

The photons have been generated, it is time to propagate them taking into account the effects of the atmosphere. This is the task of the radiative transfer block. As you can see from fig. 3 this part is a bit more complex than the other ones because there are two additional objects used to encapsulate the models for the atmosphere (`Atmosphere`) and for the reflection on ground (`Ground`).

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE:	9/27

In the radiative transfer block the fluorescence photons are propagated from the point in which they were generated (by **LightSource**) to the detector entrance where a **PhotonsOnPupil** object is created and handled to **EusoDetector** via **LightToEuso** and **EusoApplication**.

Čerenkov photons are first propagated to the ground then handled to the **Ground** object for reflection and/or absorption and finally returned to **RadiativeTransfer** that propagates them to the EUSO entrance as in the fluorescence case.

1.2 EusoDetector


When the photons arrive at the entrance of the detector (see fig. 4) the simulation path has reached the **EusoDetector** object. Now the detector response simulation starts: this part of the application is the most developed one and in the next section (sec. 2) the choices and approximations done during the preliminary implementation of this part will be discussed. Let's now focus on the design-related issues.

1.2.1 Detector Optics

In fig. 4 is shown a schematic sketch of the EUSO detector. The optics simulation models the detector as a collection of objects that are able to transport photons. This means that each element capable of interaction with photons inherits from **DetectorPhotonTransporter** and has a method named **transport** that accepts in input a photon and gives in output the new photon after the interaction.

The family of **DetectorPhotonTransporters** objects is shown in fig. 5. As you can see there are five types of objects:

- **PhotonGenerator** that represents the photon arrival at the entrance of the detector;
- **OpticalSystem** that models the main optical system;
- **FocalSurface** that represents the focal surface and its subdivision in macrocells and sensors;
- **Wall** that represents the outer surface of EUSO;
- **OpticalAdaptor** that models the optical adaptors used to recover the dead areas of the photo-multipliers.

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	10/27

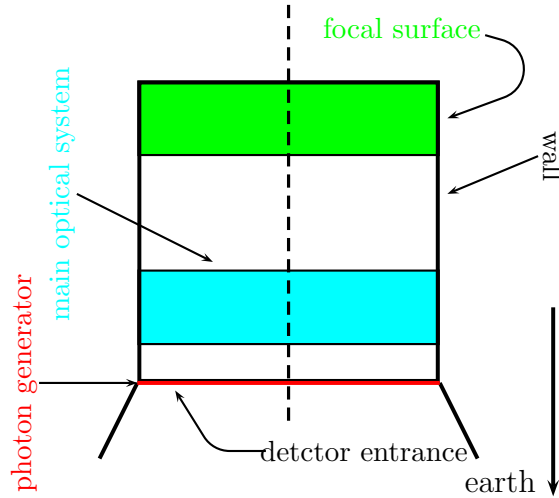


Figure 4: The EUSO Detector

The `DetectorTransportManager` treats all these transporters as black-boxes able to transport photons. The first three are modeled as cylinders concentric with EUSO and are shown in fig. 4.

The ray-tracing is managed by the `DetectorTransportManager` that propagates the photons rectilinearly between the various `DetectorPhotonTransporters` that simulate the interactions.

As an example let's follow a typical photon path inside the detector: a photon is obtained from the `PhotonGenerator` (that retrieves it from the `PhotonsOnPupil`), it is then propagated rectilinearly up to the next intersection with a detector surface and the interaction is simulated calling the `transport` method of the hit object. In this situation it can be the bottom face of the `OpticalSystem` or a `Wall`. In this latter case the photon is reflected or absorbed while in the first one it is propagated inside the main optics of the detector and it reappears on the upper side of the optics with its attributes changed appropriately. Here the `DetectorTransportManager` continues its job and propagates the photon up to the next intersection with a surface that now can be a `Wall` or the bottom surface of the `FocalSurface`. In this latter case the `FocalSurface` checks if the photon direction intersects a macrocell and if this happens it asks the hit macrocell what `OpticalAdaptor` is hit and then delivers the photon to that object. The `OpticalAdaptor` transports the photon to the sensor surface and checks if it is accepted, reflected (cathode reflectivity) or absorbed (quantum efficiency). In the first case it delivers the photon to the photo-multiplier (and from here the electronics simulation starts, see section 1.2.2), in the second case the photon is transported back to the optical adaptor surface and its path in the

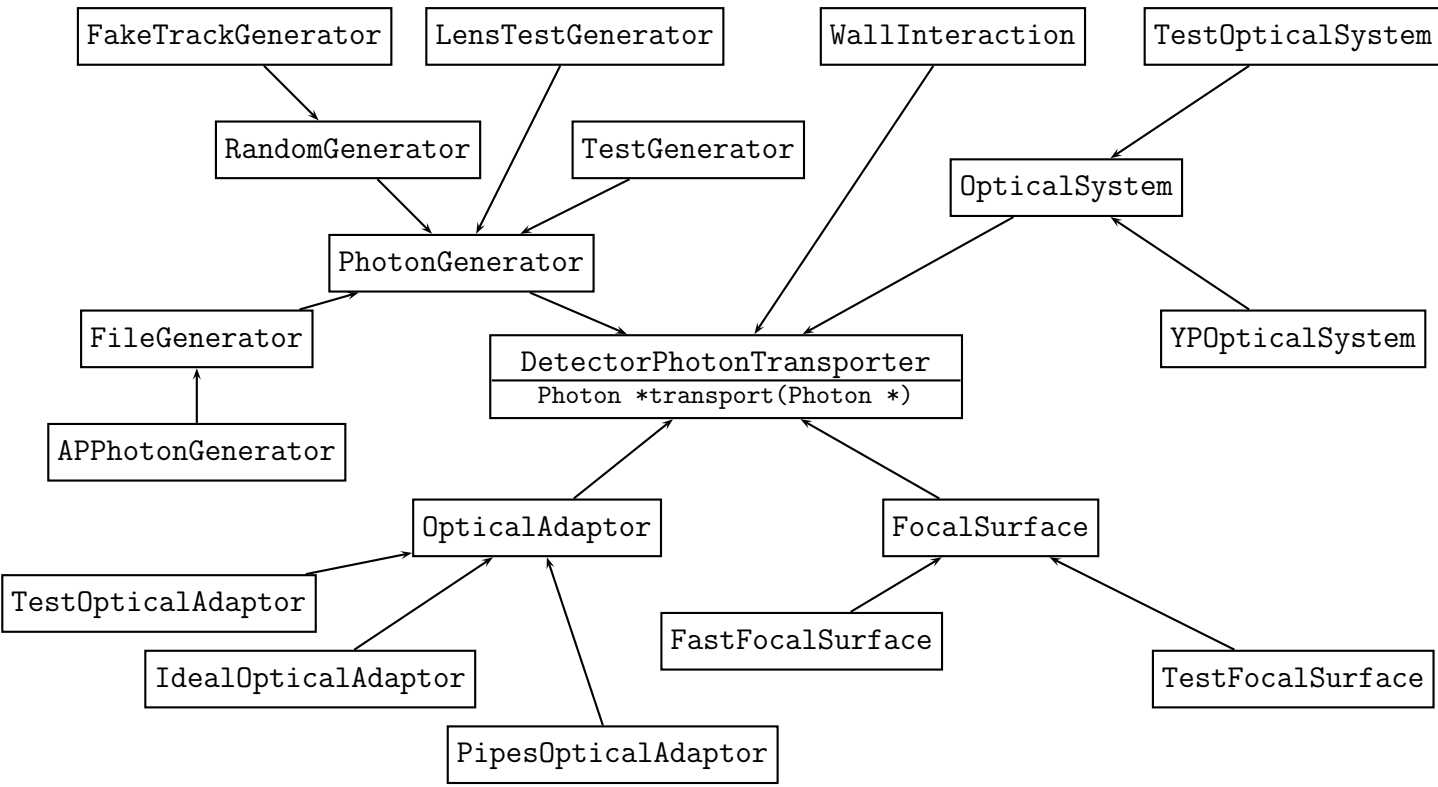



Figure 5: Detector Photon Transporters Classes.

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	12/27


detector continues. In the latter case the photon is destroyed.

In this part as well we used abstract interfaces and factories so that each one of the `DetectorPhotonTransporters` can have many different implementations without the need to change the `DetectorTransportManager`. It is so possible to switch at run-time between the different choices available for each object.

1.2.2 Electronics

The design of the electronic simulation in ESAF is done following the same guidelines described for the optics simulation. Each element of the electronics system is described by an object:

- **ElectronicsFactory.** This class is the Factory of each object in the electronics simulation. In this way it easily possible to switch from one description of a sub-element (like for example the Photodetector or the Front End Electronics or the Trigger scheme) to a different one. It is always the Factory that creates the right object according to the configuration.
- **EusoElectronics.** This class is the higher layer of the electronics simulation. It holds a pointer to all Macrocells and delegates most of job to them.
- **Photomultiplier.** This class is at the same time the base class for all kind of photodetectors and the actual implementation of a simple fast simulation of the MAPMT 16 or 64 channels. The class `Photomultiplier` does not contain any geometry related parameters. Only the electronics response to photons is modelled by the methods of this class. Geometry description is delegated to the class `PmtGeometry`. Each `Photomultiplier` holds a pointer to its own `PmtGeometry` object.
- **PmtGeometry.** This class is the geometrical description of a photodetector. It describes the photocathode as a matrix of any size. It also holds the absolute coordinates of the parent pmt and its orientation in space.
- **FrontEndChip.** This class simulates the behaviour of the front end electronics. It holds a pointer to the attached pmt. Right now only the DFEE part of the front end electronics is simulated. Eventually this class will evolve to perform the simulation of AFEE as well.

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	13/27

- **MacroCell.** This class simulates the Macrocell, i.e. the triggering unit of the Euso electronics. The `Macrocell`, in this implementation, must be rectangular of any size but it may evolve easily to any planar geometry. The Macrocell holds a pointer to all the `FrontEndChips` and `Pmts` connected to it.
- **ChipGtuData.** This class stores the `FrontEndChip` response in a single `GTU`.
- **MacroCellData.** This class stores the whole `MacroCell` response to an event.
- **Telemetry.** All Macrocells' response is stored here.

2 Implementation

ESAF is written in C++ using the framework provided by the `root` [3] libraries. These are used for the graphical user interface, the event display, histogramming and for intermediate files. Actually these files are saved in `root` format thus enabling the user to do, even for these intermediate results, checks and analysis not directly provided by ESAF. For more information on the `root` files see section 2.3.2.

In this section we will describe the implementation of the most important points of the above design.

2.1 Optics

2.1.1 YPOpticalSystem

The `YPOpticalSystem` implements an `OpticalSystem` that simulates the main optics of the detector according to the note circulated by A. Zuccaro and Y. Takahashi [8]. This is very rough and preliminary, based on the little available information.

Let's follow a photon path from the moment it is handled to `YPOpticalSystem` by the `DetectorTransportManager`. The code first checks that the angle θ (see fig. 6) between the photon direction and the detector axis is smaller than 30° to take into account the field of view (FoV). Photons outside FoV are destroyed. After that the code checks, against a probability θ -dependent, if the photon is absorbed or transmitted by the optics. The

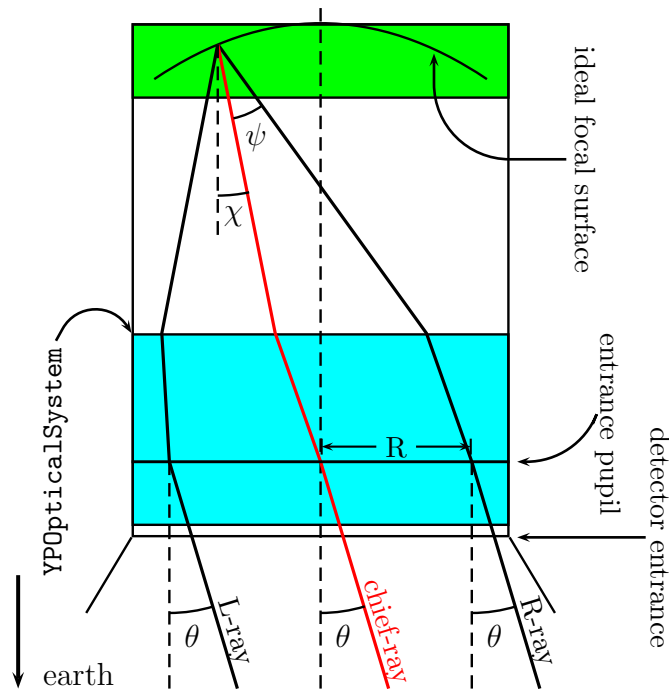


Figure 6: Photons going through YPOpticalSystem

surviving photons are then propagated to the center of the optical system. Here we calculate the impact point on the focal surface according to tab. 1 in [8]. This point is then moved in a random direction by a random amount⁵ to take into account the spot size at this θ .

Having found the impact point we have to calculate the arrival direction of the photon. To do this we first obtain (interpolating tab. 1 of [8]) the impact angle of the chief-ray of that point of the focal surface (χ in fig. 6). A chief-ray is a ray going through the center of the entrance pupil (the red ray in fig. 6). Photons crossing the entrance pupil of the optical system at a distance R from the center form approximately a cone around the chief-ray (see fig. 7). The aperture of this cone (ψ in fig. 6) is calculated interpolating the aperture given for marginal rays in tab. 1 of [8].

We are now ready to compute the arrival direction of our photon: we go in the reference frame in which the chief-ray direction is along z-axis where we fix the polar angle to the cone aperture calculated above and the azimuthal angle equal to the one of the vector connecting the arrival point on the focal surface with the photon position on the entrance pupil of the optical system⁶.

⁵The values of this amount have a gaussian distribution with mean equal to zero and sigma (θ -dependent) guessed from fig. 2 in [8].

⁶In [8] there are no informations about the azimuthal angle, as only meridional rays are described, so we made the above choice. We also tried with a random azimuthal angle.

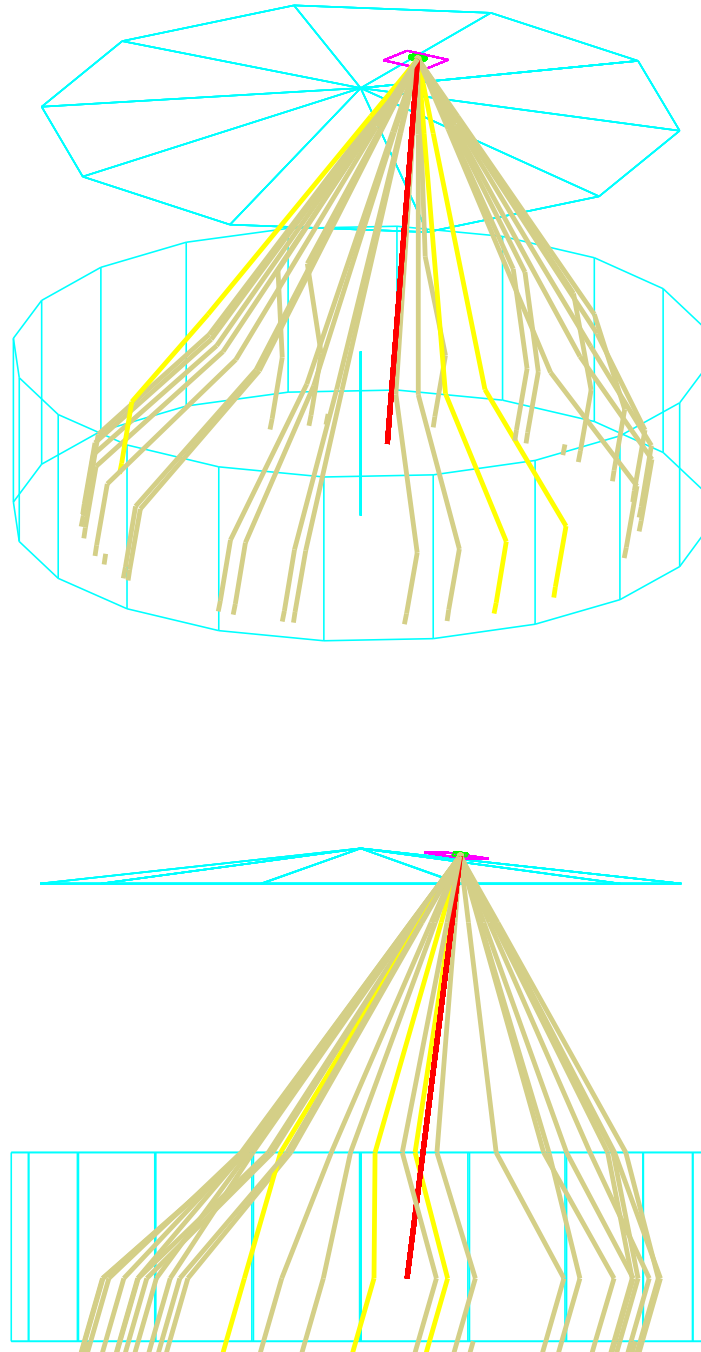



Figure 7: Two views (in the event-display) of the cone formed around the chief-ray (in red) by the non central photons in the YPOpticalSystem

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	16/27

This direction is finally transformed in the original reference frame.

Now we trace back the photon from the ideal impact point on the focal surface along the arrival direction up to the upper surface of the `YPOpticalSystem` and we find the exit position of the photon from the main optics.

The photon is then propagated from the entrance pupil of the optical system to to the upper side in the exit point and then is handled back to the `DetectorTransportManager`.

2.1.2 FastFocalPlane

In this paragraph we will describe the only implementation available, up to now, of `FocalSurface: FastFocalPlane`. This is an object able to simulate an arbitrary covering of the ideal focal surface with square modules (see fig. 8). The operation of this object is quite trivial: at the beginning it reads from a file a list of the coordinates of the modules (the macrocells) and then for every photon it scrolls the module list to find the hit macrocell, if any. To do this it calculates the intersection point between the photon direction and the plane containing the macrocell and then checks if this point is inside or outside the macrocell border. If there is no hit the photon is propagated up to the exit point of the `FastFocalPlane` and here treated appropriately (absorbed on the upper surface or handled back to `DetectorTransportManager` on the bottom surface).

If there is a hit macrocell the code asks it the list of photo-multipliers and (with a trick) queries each one of them to find the hit one. If the photon does not hit any photo-multiplier it is absorbed by the dead areas. Viceversa if there is a hit the photon is handled to the `OpticalAdaptor` attached to the hit photo-multiplier via the `transport` method. If the photon is returned by this procedure then it is a photon reflected by cathode, and is propagated to the next interaction. If there is no return value then the photon has been handled by the `OpticalAdaptor` to the electronics simulation starting from the photo-multiplier (see sec. 1.2.2).

2.2 Electronics

The classes enumerated in section 1.2.2 are at the same time base classes and simple implementations. The following list gives the most important implementation details of the current classes:

In each case the choice is arbitrary and neither one is better than the other at the present level of knowledge.

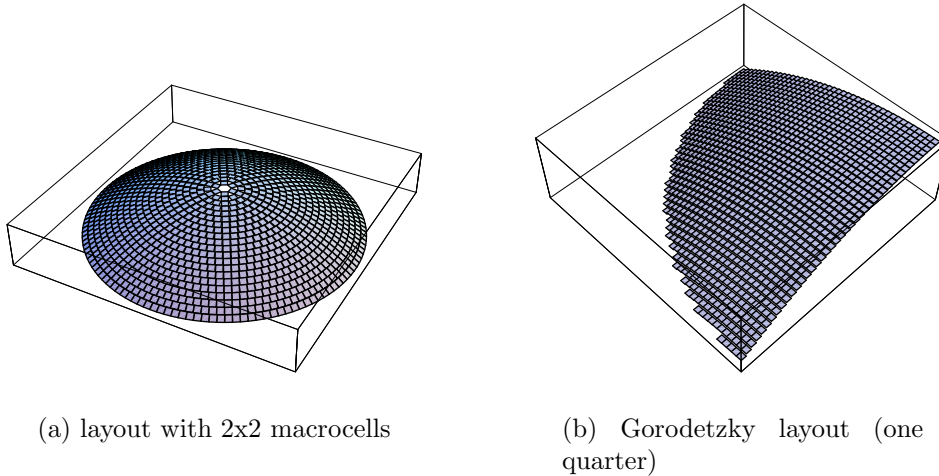



Figure 8: Two of the available implementations of the focal surface mapping [9].

- **ElectronicsFactory.** This class is the *factory* of each object in the electronics simulation. Each object is created according to what is selected in the configuration files.
- **EusoElectronics.** This class is the higher layer of the electronics simulation. It holds a pointer to all **Macrocells** and delegates most of job to them. At the beginning of each event the **Simulate()** member function of **EusoElectronics** is called. This function identifies the relevant time interval for this event, i.e. the time of the first and the last photon. Then computes a random phase to simulate the random position of the event with respect to the GTU clock and invoke the **Simulate()** member function of each **Macrocell**. This returns a pointer to a **MacroCellData** object that contains the whole **Macrocell** response to this event. Before this sequence, the member functions **CheckPmtState()** is also called for each **Macrocell**. This make sure that before the electronics is simulated, the function **Simulate()** for each photo detector was called.
- **Photomultiplier.** This class has 2 main functions. The first is the function **Add()** that allows to add a photon to the pmt. This is called by the optics module whenever a Photon hits the focal surface. It computes the pmt response to each photon as a set of **PmtSignals** objects. The pmt signals in this implementation is approximated by

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	18/27

a gaussian current peak, whose time width and height are external parameters. The gain is not necessarily uniform in the pmt but it can have a random percentual spread. The second important function is `Simulate()`; it must be called after all photons have reached the focal surface. This function transfers the `PmtSignals` list to the front end chip. At this level is also possible to include the night glow related background. The night glow rate is a configuration parameter. A poissonian distribution of photons is added to each channel. In this way it is possible to avoid tracking of a huge number of photons.


- **PmtGeometry**. This class is the geometrical description of a photodetector. All geometry related functions are here.
- **FrontEndChip**. This class simulates the behaviour of the front end electronics. It holds a pointer to the attached pmt. The main function is `Gtu()`, which simulates the DFEE behaviour in a GTU interval by returning a pointer to a `ChipGtuData` object. This objects stores the timing of the X, Y and Fast OR logic. The internal memory buffer data of the DFEE are not simulated yet.
- **MacroCell**. This class simulates the Macrocell, i.e. the triggering unit of the Euso electronics. The main function is `Simulate()`, which returns a `MacrocellData` object. The Macrocell divides the event time interval in the right number of gtus and invoke the `Gtu()` function of each `FrontEndChip`. Then ask the `MacroCellData` object to put together the signals from each chip and simulate the XY trigger logic. Higher level trigger (track finder) is not implemented yet.
- **Telemetry**. All Macrocells' response is stored here.

2.3 Miscellaneous

In this section we describe some *very* technical parts of the application not directly connected with the physics simulation, but of paramount importance for its functioning. It is so an interesting section for an aspirant developer, but a boring and probably incomprehensible one for an end user.

2.3.1 Config files and EusoConfigurable

All the parameters used in the simulation are stored in ASCII files (with extension `.cfg`) in subdirectories of the directory `config` in the ESAF principal directory (see fig. 10). Each class has its own configuration file. In this file

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	19/27

```
# config file for class YPOpticalSystem
# $Id: YPOpticalSystem.cfg,v 1.2 2002/04/03 07:51:38 ddm Exp $


YPOpticalSystem.DZup = 850 # mm
YPOpticalSystem.DZdown = 425 # mm
YPOpticalSystem.pos.Z = 500 # mm
YPOpticalSystem.FocalDistance = 2900 # mm
YPOpticalSystem.Radius = 1250 # mm
YPOpticalSystem.psf.filename = Optics/YPpsf
YPOpticalSystem.tr.filename = Optics/YPtr
YPOpticalSystem.ytheta.filename = Optics/YPytheta
YPOpticalSystem.chief.filename = Optics/YPchief
YPOpticalSystem.aperture.filename = Optics/YPaperture
```

Figure 9: Example of config file: YPOpticalSystem.cfg. As you can see the first five parameters are numbers while the last five are names of file to be used to get further data.

are stored all the parameters regarding the class and also the parameters regarding all its parents. With this technique is easier for the end-user to change a parameter and is much more simple to maintain consistency among parameters. As an example if the user wants to change a parameter of an object A (that inherits from B) he does not need to know that A inherits from B, or where the parameter is defined (in A or in B). He simply finds the configuration file for A and there he changes the chosen parameter. Moreover with this approach the config file of B is not changed with a parameter value that is convenient for A, but that might not be suitable for C (that also inherits from B).

As said above the parameters are stored in ASCII files in the form of `par_name=par_value` pairs (one per line). `par_name` is the parameter name while `par_value` is the value. `par_value` can be a number, a string or a filename (see fig. 9 as an example). Anything after a `#` is considered a comment and is discarded, whitespaces are ignored.

The configuration mechanism is implemented in the following way: each class that needs to access any type of config data should inherit from `EusoConfigurable` and call in its definition the macro `EusoConfigClass(type, name)`, where `type` and `name` are the type (the name of the subsystem to which the class belongs) and the name of the class being defined. `EusoConfigurable` is an abstract interface that defines three methods: `ClassName`, `ClassType` and `Conf`. The first two return the name and type defined above

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	20/27

while the third method returns a pointer to a `ConfigFileParser` object that contains all the parameter values for this object and provides access to them via some `Get` methods.

`ConfigFileParser` is an object that, basing on the type and name of the class, identifies the file containing the configuration parameters for the object considered (class type identifies the subdirectory (see fig. 10) while class name identifies the filename), parses it and stores the `par_name=par_value` pairs in maps. It then provides access to the parameters basically via two methods: `GetNum(par_name)` and `GetStr(par_name)` that return respectively the number or the string associated with the parameter-name `par_name`.


The `ConfigFileParser` objects are created by a *factory* [5]: `Config`. This object is a *singleton* [5] that has a method `GetCF` that returns the appropriate `ConfigFileParser` in each case. Having centralized the creation of the `ConfigFileParsers` into one object has the major advantage that this central class has the possibility to change the base directory for all the config-files. It is so possible to have in the `config` directory various subdirectories describing different configurations of the detector (as an example `layout1` and `layout2` in fig. 10). These configurations are provided with ESAF and inside them the data are correct, consistent and tested (guaranteed to function properly). The user is still able to try and test new configurations changing the files in the `user` directory.

We would like to point out that is not mandatory to search and edit the ASCII files to modify the configurations: a GUI interface is provided and permits even the new user the modification of the most important parameters of the simulation. Then when the user is no more new and want to fine tune all the parameters he should turn to the files, but now he should, hopefully, know where to act.

2.3.2 Root

ESAF uses the root classes to provide the user with a user friendly Graphical User Interface (GUI) and almost everywhere in the code. In particular they are used for:

- **Linear algebra** Vectors, matrices, rotations, coordinate transformations, are done everywhere using the linear algebra root classes and functions.
- **Random numbers** The class `TRandom3` is used as random number generator. In order to keep the random seed under ESAF's control, the class `EusoRandom` has been created; this class owns the only instance

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE:	21/27

of the class `TRandom3`. Each element of ESAF gets flat or Gauss or Poisson distributed random numbers from this class.

- **Interface and Application** The main Graphical User Interface, class `EusoMainFrame` is completely based on the root classes (menus, buttons and so on), as it is the main application program as well (class `EusoApplication`). Actually ESAF IS an interactive root program thanks to the inheritance of `EusoApplication` from class `TRint`.
- **Graphics and event display** A simple event display of the detector simulation is available, thanks to the root graphic tools. The detector optics and focal surface is drawn, together with the path done by each photon.
- **EEvent and ETree** The simulation output is stored in a root file in the form of a `ETree` of `EEvent` objects. The class `ETree` inherits from `TTree`. The class `EEvent` contains all the output data for each module of the simulation. The design of `EEvent` is only sketched right now.

2.3.3 Makefiles & Directory Structure

In fig. 10 is shown the directory structure created by ESAF. We now describe the purpose of the various directories:

- `config` contains the configuration files for the various possible layouts of the detector and a `user` subdirectory for the user to play with;
- `doc` contains this paper and other useful documents related to the application or to its development;
- `output` contains all the resulting files of a run;
- `lib` contains the libraries compiled during the making of ESAF. They are in a directory describing the architecture for which they are compiled;
- `bin` contains the executable of ESAF. It is in a directory describing the architecture for which it is compiled.
- `packages` contains all the source code. It is divided in many packages and subpackages. The files are ultimately divided in two more directories: `src` and `include`.

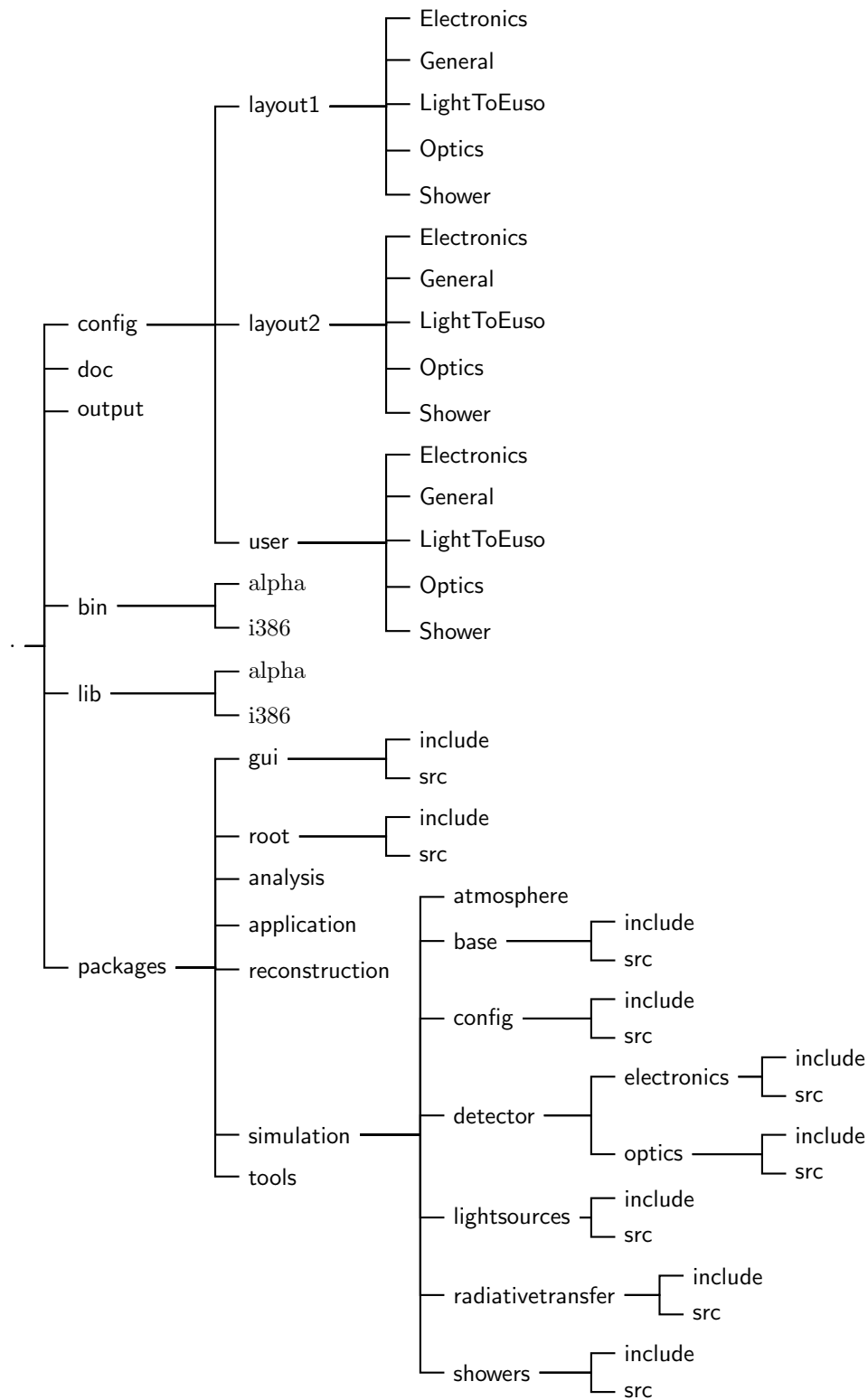



Figure 10: Directory Structure.

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	23/27

```

# Building options for esaf
# $Id: Rules,v 1.4 2002/04/12 13:55:40 demarco Exp $

# Enable debug (DEBUG=1) or optimization (DEBUG=0)
#DEBUG = 1
DEBUG = 0

# Temporary Directory
ESAFTMP = /tmp

# verbose compiling
VERBOSE_COMPILING = 0


```

Figure 11: Rules: this file contains some options for the compilation step.

Besides using the directories above, ESAF needs some scratch space for the objects and it tries to use the `/tmp` directory (this is configurable, see below).

The GNUmakefiles are modular and are spread across almost all these directories. They are very simple and based on common parts that are included dynamically during the compilation. In the principal directory of ESAF there is a file named `Rules` that contains some options for the compilation step (see fig. 11). The first one, `DEBUG`, is the most important. It specifies if we are compiling with the debugging options (slow, but with many aids for the debugging phase) or with the optimisations (much more fast, but debugging almost impossible). Other options allow to change the temporary directory used for objects and to toggle visualisation of each step of the compilation.

Each directory has a GNUmakefile that defers the compilation process to the GNUmakefiles in its subdirectories until the second-last that contains the `src` and `include` subdirectories. This last GNUmakefile is the one doing the compilation. It includes a global configuration makefile (`packages/config.gmk`) that defines the location of the various directories, checks that all these are present (and creates them if necessary) and makes sure that all the prerequisites are met. Then it includes a common makefile (`packages/common.gmk`) that makes the dependencies for all the files in the `src` subdirectory, checks which need to be compiled (or recompiled), compiles them and then creates the library for the considered package. All the libraries are at the end linked with the main program by the topmost GNUmakefile to create the ESAF executable.

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	24/27

3 Installation

3.1 Getting the sources

There are two ways you can get the ESAF source code: downloading the compressed archive from the web or getting the latest developers version from CVS.

3.1.1 Download from web

In the first case you download a `.tar.gz` archive from:

<http://www.ge.infn.it/euso/esaf/download/>

then you uncompress it with:

```
gunzip filename.tar.gz
tar xfv filename.tar
```

where `filename.tar.gz` is the name of the file you downloaded. At this point you should have a `esaf` directory that contains a structure like the one in fig. 10 and you can move to sec. 3.2.

3.1.2 Download via CVS

In the second case the things are a bit more complex, but you can access the latest version. With CVS there are two different possibilities of access: via ssh or via pserver.

Let's now describe the steps to be done in the first case: first of all you should assign to the environment variable `CVS_RSH` the value `ssh` doing:

```
export CVS_RSH=ssh      (if you use sh, bash...)
setenv CVS_RSH ssh      (if you use csh, tcsh...)
```


Next you set the repository location using:

```
export CVSROOT=user@host:esafdir  (sh, bash...)
setenv CVSROOT user@host:esafdir  (csh, tcsh...)
```

Next you get the sources using:

```
cvs checkout esaf
```

where `user`, `host` and `esafdir` are respectively your username to access

	Euso Simulation and Analysis Framework	DOC. REFERENCE:	
		ISSUE:	1.0
		REVISION:	1
		DATE:	May 13, 2002
		PAGE	25/27

the system, the name of the host containing the CVS repository and the home directory of the ESAF repository. At the moment the repository is in Genoa, but in the future it will be probably moved to Lyon. As an example to connect to the Genoa repository one of us can do:

```
export CVSROOT=demarco@g2farm3.ge.infn.it:/home/cvseuso
cvs checkout esaf
```

At this point you should have a `esaf` directory that contains a structure like the one in fig. 10 and you can move to sec. 3.2.

Now we describe the procedure used to access the sources via pserver. As before first of all is necessary to set the repository location:

```
export CVSROOT=:pserver:user@host:esafdir      (sh, bash...)
setenv CVSROOT :pserver:user@host:esafdir     (csh, tcsh...)
```

then you should login using:

```
cvs login
```

and then get the sources as before:

```
cvs checkout esaf
```

As an example to connect to the Genoa repository one of us can do:

```
export CVSROOT=:pserver:demarco@g2farm3:/home/cvseuso
cvs login
cvs checkout esaf
```


As in previous cases at this point you should have a `esaf` directory that contains a structure like the one in fig. 10 and you can move to the next section.

3.2 Compilation

The compilation phase is very simple. First of all you should specify where your root installation resides using:

```
export ROOTSYS=path      (sh, bash...)
setenv ROOTSYS path     (csh, tcsh...)
```

and where to find root libraries and binaries:

	Euso Simulation and Analysis Framework	DOC. REFERENCE: ISSUE: REVISION: DATE: PAGE	1.0 1 May 13, 2002 26/27
---	---	---	-----------------------------------

```

export PATH=$PATH:$ROOTSYS/bin      (sh, bash...)
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ROOTSYS/lib
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./lib/arch

setenv PATH $PATH:$ROOTSYS/bin      (csh, tcsh...)
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:$ROOTSYS/lib
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:./lib/arch

```

where arch is the name of the architecture of you computer (i386, i686, arch...).

Next you start the compilation with:

```
gmake
```

After a while, when the compilation finishes, you can try running ESAF doing:


```
bin/architecture/esaf
```

where architecture is the name of the architecture of you computer (i386, i686, arch...). As an example one of us do:

```

export ROOTSYS=/usr/local/cern/root
export PATH=$PATH:$ROOTSYS/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ROOTSYS/lib
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./lib/i686
gmake
bin/i686/esaf

```

	Euso Simulation and Analysis Framework	DOC. REFERENCE: ISSUE: REVISION: DATE: PAGE	1.0 1 May 13, 2002 27/27
---	---	---	-----------------------------------

References

- [1] Bjarne Stroustrup, *The C++ programming language*; Addison-Wesley, MA 1991.
- [2] Grady Booch, *Object Oriented Analysis and Design with applications*; Benjamin, Redwood City, CA 1994
- [3] <http://root.cern.ch>
- [4] Euso Meeting, March 2001, Genova (Italy);
1st Euso General Meeting, October 2001, Annecy (France).
- [5] E. Gamma et al., *Design patterns: elements of reusable Object-Oriented software* (Addison-Wesley, 1994)
- [6] D. Heck et al., *The CORSIKA Air Shower Simulation Program*, FZKA report-6019, ed. FZK, Karlsruhe, 1998
- [7] <http://hep.fi.infn.it/AIRWATCH/>
- [8] A. Zuccaro and Y. Takahashi, *Distribution of Incidence Angles of Rays Hitting the Image Surface for Euso: Analysis on a f/1.25 Configuration*, internal note circulated in Jan. 2002.
- [9] A. Petrolini, internal note.