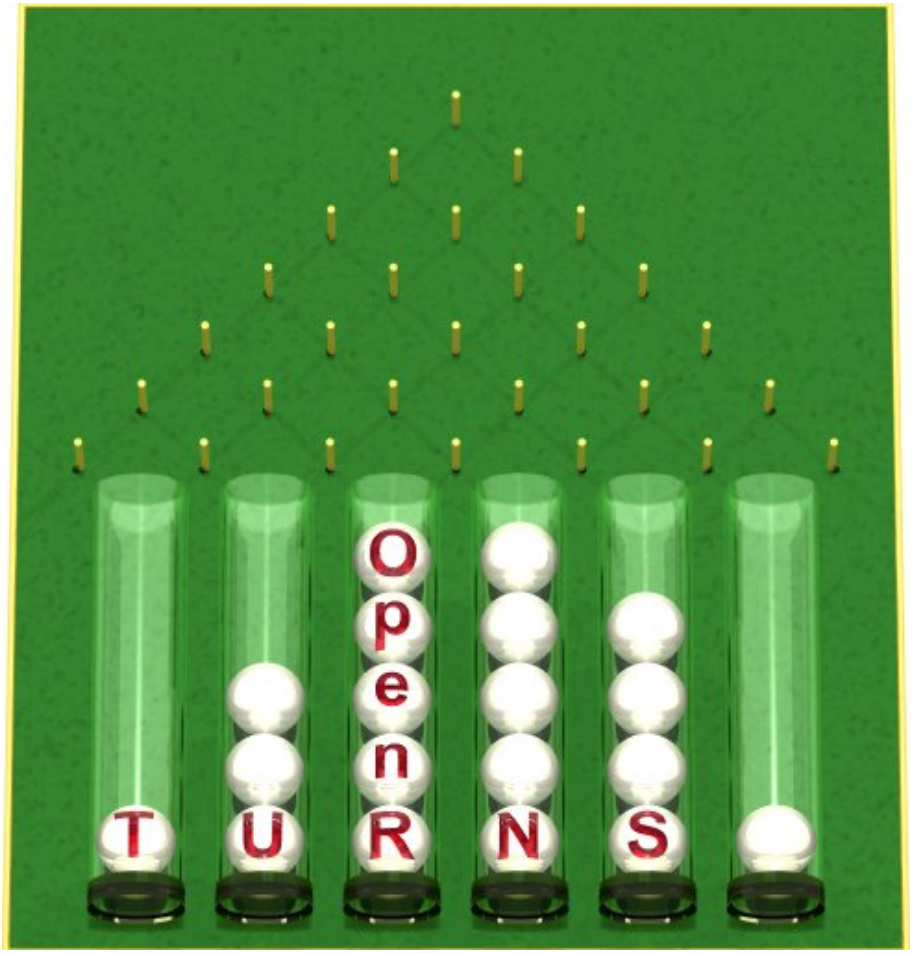
First Elements of the Architecture Guide

Open TURNS version 0.12.1

November 8, 2008



Open TURNS project: General architecture specifications

Abstract

The Open TURNS project is an open source development project. It aims at developing a computational platform designed to carry out industrial studies on uncertainty processing and risk analysis.
This platform is intended to be released as an Open Source contribution to a wide audience whose technical skills are very diverse. Another goal of the project is to make the community of users ultimately responsible for the platform and its evolution by contributing to its maintenance and developing new functions.
This architecture specifications document therefore serves two purposes:

- to provide the design principles that govern the platform, in order to guide the development teams in their development process;
- to inform external users about the platform's architecture and its design, in order to facilitate their first steps with the platform.

In the first section of this document, we will introduce the concepts that governed the construction of the platform. These concepts resulted from the requirements analysis carried out with the users and the developers, following the UML approach. The general functions of the platform allowed us to categorize the concepts by giving us a more synthetic and global view of its components.
The second section details the technical choices that were retained for the platform's development.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This document makes up the general specifications for the architecture of the Open TURNS platform. The architecture described here adresses the following goals:

- *building an open, upgradable and generic platform for the treatment of uncertainties*, relying on recognized and valid mathematical methods as well as on a methodological approach that was put forward and supported by the partners.

- *interfacing with any field-specific code.*

To address these questions, the Open TURNS platform needs to be:

- *portable*: the ability to build, execute and validate the application in different environments (operating system, hardware platform as well as software environment) based on a single set of source code files.

- *extensible*: the possibility to add new functions to the application with a minimal impact on the existing code.

- *upgradable*: the ability to control the impact of a replacement or a change on the technical architecture, following an updgrade of the technical infrastructure (such as the replacement of one tool by another or the use of a new storage format).

- *durable*: the technical choices must have a lifespan comparable to the application's while relying on standard and/or public domain (Open Source) solutions.

The computing architecture will be detailed in this document according to 3 different abstraction levels:

- *functional architecture*: describes the application's functions and the distribution of its components without addressing any actual programming issues.

- *software architecture*: defines the programming design of the modules identified in the functional architecture.

- *technical architecture*: lists the technical choices that were made for the development of each module. These technical choices are justified both by environmental constraints and by the requirements expressed in other archectural levels.

1.1 Document outline

The document is organized as follows:

- *Chapter 2* deals with the functional architecture of the Open TURNS platform (module distribution and description of each brick's function).
- *Chapter 3* defines the technical architecture of the Open TURNS platform (technical choices).
- The *bibliography* gives an alphabetical list of all references used in this document.

1.2 Bibliography

The following subsections give organize the bibliographical references used in this document according to the main fields at stake. For more details about each reference, please refer to the bibliography at the end of this document.

1.2.1 Modelling

[Mul] [GHJV]

1.2.2 C++ and STL

[ES] [Del] [Meya] [Meyc] [Ale] [Aus] [Meyb]

1.2.3 Multithreading

[LB]

1.2.4 Python

[LA] [Lut] [PY]

1.2.5 Qt

[Dal]

1.2.6 Subversion

[CSFP] [SVN]

1.2.7 Websites

[SWI] [BOO] [OT] [OTD] [UNI] [R]

Chapter 2

Analysis model and functional architecture

This section deals with the analysis of the requirements that guided the general implementation choices for the Open TURNS platform. The system is here to be seen from a rather generic, functional point of view; it mainly consists in blocks handling the main functions of the platform. We shall detail the fundamental concepts that arise from the requirements and their role within the system.
At this stage of modeling, technical considerations shall not (except in specific cases) be addressed yet. The platform is seen as a system made up of components interacting with each other, providing designated functions. In this section, we shall answer the question "WHAT does the system actually perform ¿'

2.1 Description of the analysis approach

The Open TURNS project aims at producing a risk analysis tool. When the project started, there was no equivalent tool available on the market, whether it be proprietary or Open Source software. Therefore, a choice was made to design an entirely new tool "from scratch", beginning with the requirements analysis, using the most effective methods and the most up-to-date scientific concepts, relying on tools widely recognized by the scientific community.
Considering the recent progress in computer science and software engineering, the platform will be implemented using an object-oriented language, which naturally leads us to use an object modeling approach to analyze and design the elements that will make up the code. The UML approach was therefore chosen for this preliminary stage of the project.
Readers interested in UML will find more detailed information in the guide written by Pierre-Alain Muller [Mul]. However, we will sum up the approach to present all the notions relevant to this document.
UML relies on the analysis of the requirements expressed by the future platform users, using a standard graphical formalism. Everything starts with the definition of use-cases that describe how the users will interact with the system. There can be several users (sometimes several thousands) but only their role regarding the system is taken into consideration. These users are supposed to describe the actions they want to perform and the responses they expect from the system. Therefore, at this stage of modeling, the actual realization ("how") of actions is not taken into account; only the functional need of the user ("what") is being considered.
First, these use cases are graphically modeled with ellipses, the users being represented as human figures (as shown in Figure 2.1).
All of the use cases triggered by the same user allow us to have an overview of the actions this user can carry out on the system.
Conversely, requiring all the users to describe the same use case allows us to study the case from multiple angles and to determine all the interactions that need to be taken into account between users.
Naturally, the definition of a use case is not limited to putting a designation within an ellipse. This description

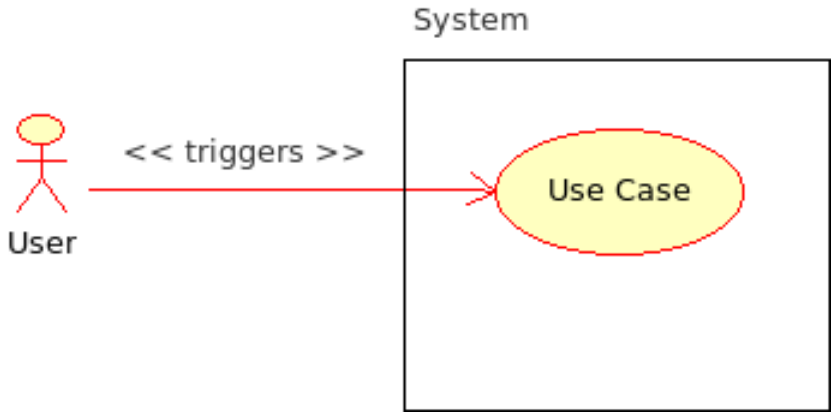


Figure 2.1: *Example of a use case triggered by a system user.*

is only used to provide an overview of the use case. The use case must be described in detail: most often, this stage uses a natural language (i.e. free format) description of the user's actions and the system's responses. If this description requires highly technical content (which is the case for the Open TURNS platform), it may be wiser to use a language close to either the user's requirements or the target system. The appendix provides an extensive description of the use cases (in algorithmic form) written for the Open TURNS platform.
Based on these detailed descriptions, we can now extract concepts that are fundamental for the system and link them so as to view their interactions, their functional proximity and the abstract ideas that can be derived from them. We refer to this set of concepts and links as an *analysis model*.
It is then possible, based on these concepts, to create bricks of variable size encompassing related notions, and thus to show the logical and functional view of the platform. This global view of the platform is essential to have a general understanding of the system as well as to consider its evolution capabilities.
The present chapter describes the results of this analysis, which was carried out with and for the platform users. In the next chapter, we will rely on this analysis to introduce the design model and the software architecture.

2.2 Requirements analysis and use cases

Interviews with the future users of the system have uncovered the following general requirements:

- the platform must allow the users to carry out uncertainty and risk analysis studies as well as statistically process data provided both internally and externally;
- the platform must be of ergonomic and easy use for novice users, as well as complete and precise for expert users. It must therefore follow the Methodological Reference [Dut] provided by EDF R&D;
- the platform must provide a graphical user interface (GUI) as well as a text user interface (TUI); it must also provide a means of external control by another code (*scripting* and *batch*);
- the platform must be efficient in order to handle several millions (or more) of computations, and must be able to use the distributed computing capabilities of an heterogenous network or a remote computer;
- the platform must run in a Unix-like environment, but future ports to other environments (Windows, etc.) will be considered;
- the platform is generic and must be able to interface with (almost) any field specific code;

- the platform is developed using Open Source software and will be distributed under the terms of an Open Source license;
- the platform includes efficient scientific methods that can be expanded or supplemented;
- the platform uses Open Source tools that are considered as references in the field of statistics and optimization;
- the platform will be primarily developed using object-oriented technologies;
- the platform must be able to run either as a *standalone* application;
- the platform must interface with an indexed storage system that allows the preservation of results coming from previous computations, so as to avoid having to compute them again;
- the platform must include an extensive online documentation.

These general requirements are described with more details in the use cases, which are grouped in packages according to the recommendations given in the Methodological Reference.

2.2.1 Definition of the actors involved with the Open TURNS platform

The analysis of the actors involved with the Open TURNS platform brings us to the following list:

- *User*: this actor carries out the uncertainty treatment studies with the platform. From a practical point of view, it is the main actor which we will focus on, since the platform is being developed with that role in mind. It is in most cases a human being that will be interacting with the system, but other systems can also play that role in automated running modes (*scripting* and *batch*).
- *Developer*: this actor adds new functions to the Open TURNS platform. Although the platform does not focus on this actor, it is implemented so as to make the development and integration tasks as easy as possible for them. It is always a human being.
- *Administrator*: this actor sets up the Open TURNS platform on the target network. Their role is to configure the platform so that it provides the expected services. It is always a human being.
- *External code*: this actor is the field specific code that will be called upon in an uncertainty study. From the system's point of view, it is a passive actor.
- *Storage system*: this actor is the system that allows the preservation of results coming from previous studies. From the system's point of view, it is also a passive actor.

2.2.2 Modeling package

This package gathers all the use cases involved in defining an uncertainty treatment study before any computation is actually carried out by the system. Figure 2.2 details these use cases.
The only actor involved in this package is the *user*.
For the detailed use cases, please refer to the appendix.

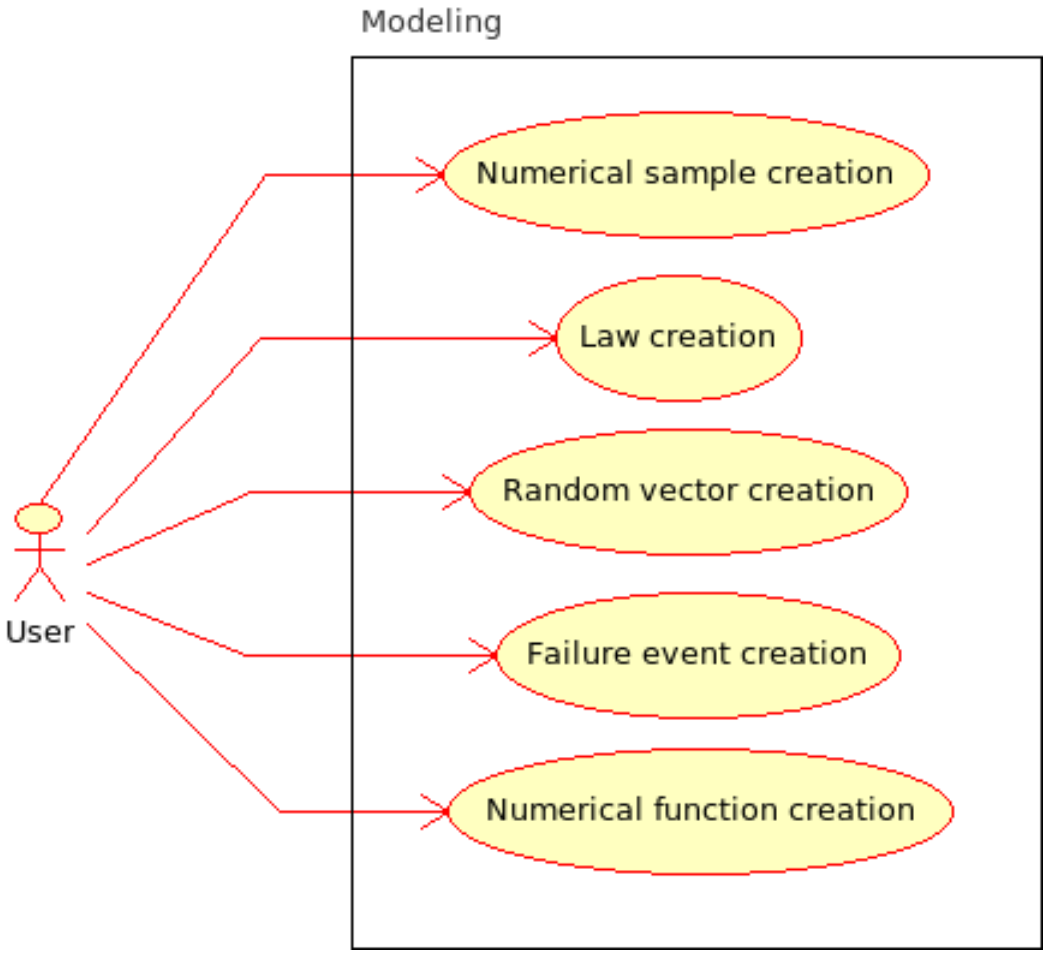


Figure 2.2: *Modeling use cases.*

Creating a numerical sample

The numerical sample is a key concept for the Open TURNS platform, particularly with regards to its statistical capabilities. The user can create a sample in the system in different ways:

- by importing a file containing the description of the sample;
- by manually setting the values of the sample through the interface;
- by using a random variable to produce the sample;
- by extracting a subsample from a given sample or expanding[1] a given sample into a larger one.

Creating a law

The law is probably the core concept of the Open TURNS platform. The uncertainty treatment model is, for the most part, based on the notion of law. A law can be created in different ways:

- by setting its parametric type and parameters through the interface;

[1] In the case of the non parametric bootstrap, elements of a sample are randomly drawn to create a larger sized sample.

- by automatically establishing its parameters based on a numerical sample and a hypothesis on its type (distribution inference);
- by combining other distributions (mixture law, assembly law);
- by extracting a sub-section of a given law;
- by creating the law from other laws and numerical functions.

Creating a random vector

The random vector represents the concept of random variable within the Open TURNS platform. It is the concept most easily manipulated by users in their studies. It can be created as follows:

- by setting its joint law;
- by combining several other random vectors;
- by extracting a sub-vector from a given random vector, or expanding a given random vector into a larger one;
- by creating the vector from other random vectors and numerical functions.

Creating a failure event

A failure event is used to identify the limit state of a numerical function. It can be created very easily with a random vector, a threshold (in the form of a point) and a comparison operator that will compare the value of the vector with the one of the threshold.

Creating a numerical function

The numerical function is, along with the law, another core concept of the Open TURNS platform. The numerical function represents the physical or theoretical model that is to be studied and through which the uncertainties on the input random vector will be propagated. This numerical function can be given in an analytical form or in a coded form (in which case it is defined within a external code).

2.2.3 Propagation Package

This package encompasses all the use cases that appear after modeling, when the user wants to propagate uncertainty through the external code. Figure 2.3 details these use cases.
In all use cases from this Propagation package, two new actors appear beside the user: the external code and the storage system. From the system's point of view, the external code behaves as a numerical function on which an uncertainty study is to be carried out. The storage system, which is important but optional, acts as a replacement for the external code during the (costly) evaluation of the numerical function.
The use cases are detailed in the appendix.

Generating a numerical sample

Generating a numerical sample means applying the numerical function on the input numerical sample. The generated sample is the numerical sample produced as an output of the numerical function. The samples can reach very large sizes (several million points) and therefore, the evaluation of the numerical function needs optimization. This can be achieved either by computing only the points that have not been computed yet,

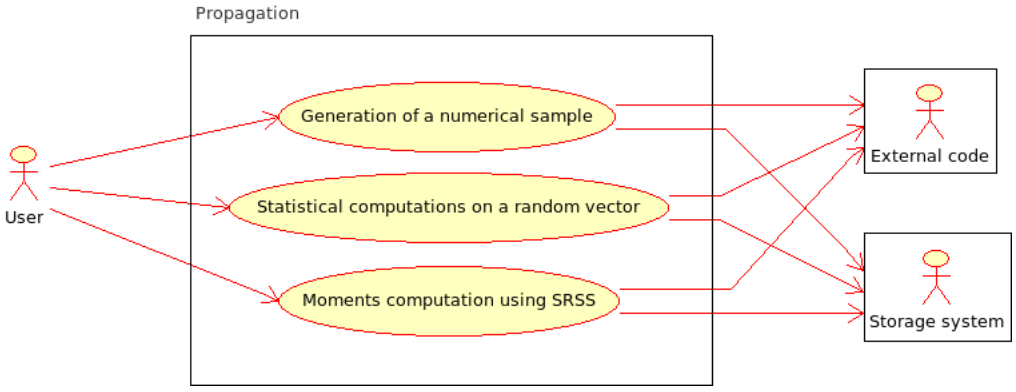


Figure 2.3: *Propagation use cases.*

which makes use of the storage system; or by distributing the external code's execution on a computer network. The propagation of uncertainty based on Monte-Carlo methods belongs to this application field.

The distributed computing system being integrated in the Open TURNS platform, it was decided not to represent it as an external actor.

Statistical computations on a random vector

The goal of an uncertainty processing study is to evaluate a given number of statistics on a random vector, which is generally defined as the output of a numerical function. The nature of these statistics may vary: mean value, standard deviation, threshold-crossing probability, and so on.

Depending on the statistics chosen, different algorithms can be implemented, which influences the number of calls on the numerical function and its derivative functions. As a direct corollary, the external code can be called a very large number of times, which results in an important computational overload that needs to be distributed throughout the network in order to reduce the waiting time for the user: an example of this would be the computation of the numerical function's gradient using the finite-difference method.

A storage system that is external to the platform and works as a cache can help reduce the number of computations by preserving the already computed results.

Computing the moments using SRSS

Computing moments using SRSS involves evaluating the numerical function, its gradient and its higher order derivatives. As for the computation of a threshold-crossing probability, it is sometimes necessary to evaluate the numerical function's derivatives using a finite difference algorithm, which requires the parallelization of the computations and the use of a storage system to preserve previous results.

2.2.4 Prioritization Package

The prioritization is the last stage in the uncertainty processing procedure, which evaluates the sensitivity of the output with respect to the input parameters. It comes after the propagation stage (see the previous section).

The use cases are detailed in the appendix.

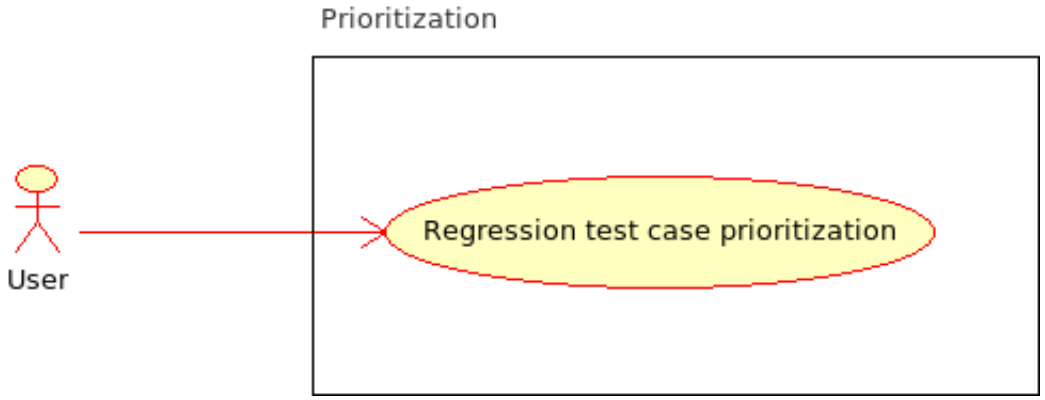


Figure 2.4: *Prioritization use cases.*

Regression test case prioritization

Prioritization uses a linear regression computation between the input and output parameters of the numerical function evaluated during the propagation stage, so as to assess the sensitivity of the output parameters regarding the input. This stage usually makes use of the platform's graphical abilities in order to visualize the results.

2.2.5 Contribution package

This package encompasses the use cases dealing with the development and integration of new functionalities into the Open TURNS platform.

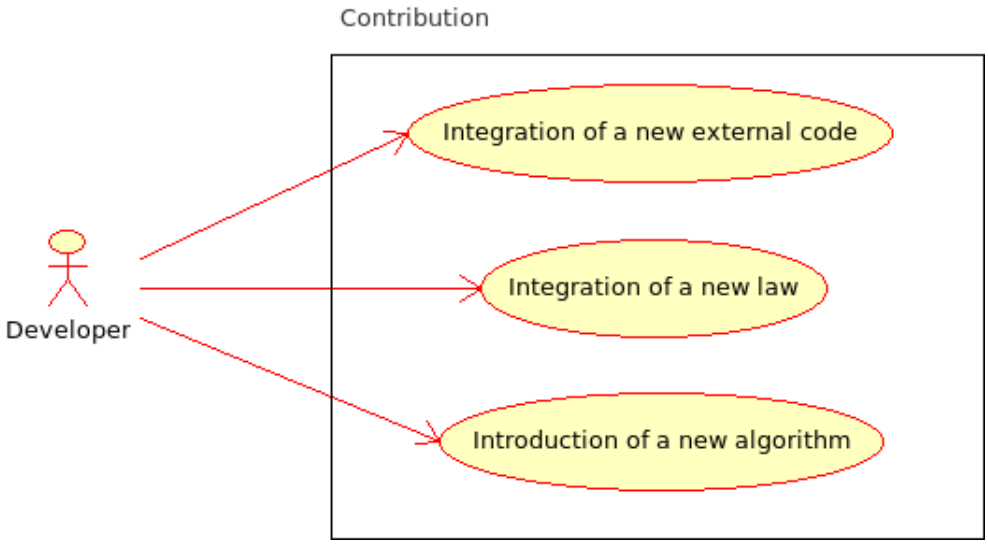


Figure 2.5: *Contribution use cases.*

As shown in Figure 2.5, a new actor, the developer, is involved and can be competent in one or several of the following fields:

- Computer science

- Numerical analysis
- Statistics/probabilities

Several persons may be needed to combine all of the above skills. Integrating new functions into the Open TURNS platform may require the collaboration of several people.
These use cases are not detailed in this document; they will however be introduced as documented examples provided with the platform distribution.

Integrating a new external code

The external code is the passive actor that supports the notion of numerical function; integrating new functions into the platform therefore requires interfacing with a field-specific code, using an API that the code must follow. If it is not natively the case, an interface layer between the API and the code needs to be developed. This layer is not the responsibility of the Open TURNS platform. The platform only defines the API that enables communication with the external code.

Integrating a new law

Integrating new laws in the platform enhances its modeling capabilities. The standard laws included in the platform are described in the project's Methodological Reference [OTmeth].

Integrating a new algorithm

Integrating new algorithms in the platform enhances its modeling and computing capabilities. The standard algorithms included in the platform are described in the project's Methodological Reference [OTmeth].

2.2.6 Configuration package

This package encompasses the use cases related to the platform's setup on a computer park for a field-specific use.

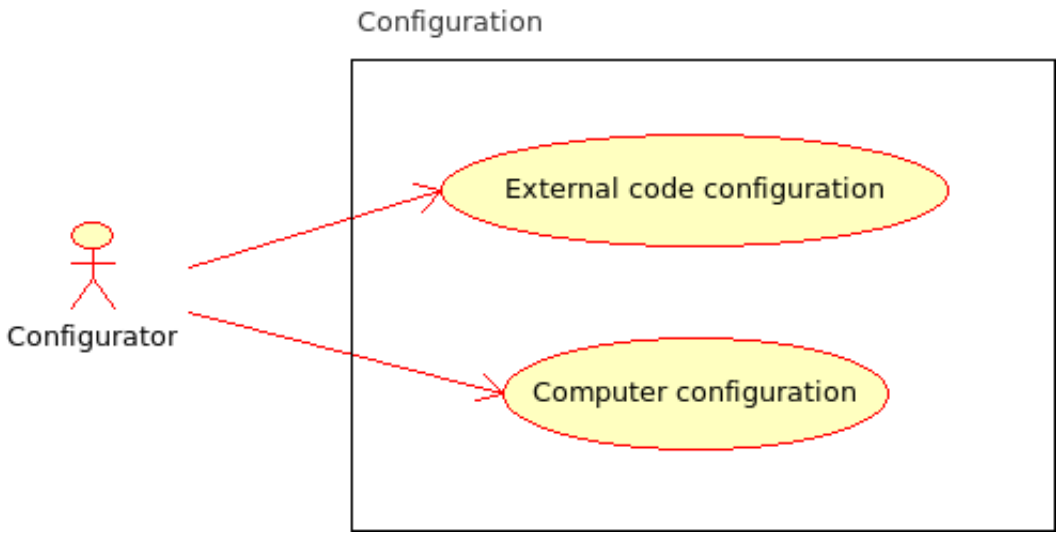


Figure 2.6: *Configuration use case.*

The configurator (shown in Figure 2.6) acts as the person responsible for the installation and administration of the platform. They set up the connection between the external codes and Open TURNS, and configure the platform's networking parameters. It is a static configuration: there is no automatic exploration of the computing environment.
These use cases are not detailed in this document; they will however be introduced as documented examples provided with the platform distribution.

External code configuration

Configuring external codes requires the configurator to define which data sets of the external code are related to the current uncertainty study, to state in the platform which external code is to be used and which of its parameters are considered uncertain.

Computer configuration

The computers' configuration is carried out by the configurator; a configuration states which computing resources are available for the execution of the platform and of its dependencies (external code, storage system, and so on), as well as the protocols, identifiers, priorities, et. related to these resources.

2.3 Description of the analysis concepts

Writing the use cases in a detailed form allows us to shed light on the concepts the user wishes to manipulate through the platform. These concepts are linked with one another by relationships such as "depends", "uses", "combines" ("aggregates", "composes"), "generalizes", "specializes", and so on. The following section shows the UML notational system (for more information, please refer to [Mul]) that was used, in order to make the graphs easier to understand.
In the analysis model, these concepts are abstract entities that do not necessarily have a direct projection as a programming entity (object, class,...). Their role is to clear the vision one can have of the system and of its internal mechanism. No technical considerations should normally appear at this stage of modeling. However, a technical consideration may exceptionally be taken into account at the analysis level if it should have nefarious consequences on the model.

2.3.1 Notations used in the analysis model

This section briefly describes the visual representation of the concepts and of the relationships that link them.

Concept

The concept is an entity manipulated by the system's user, who has an intellectual representation of the concept. Its symbol is a rectangle containing the name of the concept. Figure 2.7 shows six concepts A, B, C, D, E, and F, linked with one another.

Association

Association defines a relationship of reciprocal use between two concepts. In the example given in Figure 2.7, the concepts A and C are linked by an association. Associations are represented by full lines linking the concept boxes. Association is bidirectional: A knows C and C knows A.
Example: consider the concepts Vector and Law; we can say that a Vector may be associated to a Law, and vice versa.

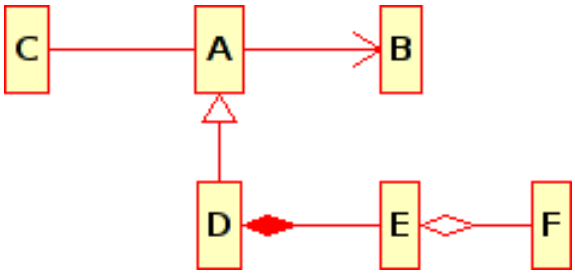


Figure 2.7: *Graphical representation of concepts from an analysis model and their relationships.*

Unidirectional association

This association type is a restriction of the more general association, which allows flow in only one direction. This is the case for concepts A and B of our example: A knows B but B does not know A.
Example: consider the concepts Vector and Sample; we can say that a Vector may be associated to a Sample (a Vector can produce a Sample) but a Sample cannot be associated to a Vector.

Generalization/Specialization

This relationship links two concepts, one of them (A) being more general than the other (D). We can also revert the reading of the link and say that one of them (D) is more specialized than the other (A). Its graphical representation is a hollow arrow pointing to the more general concept.
Example: a Law is more general than a UsualLaw, whereas a UsualLaw is more specialized than a Law.

Aggregation

Aggregation is the relationship that links a container concept to a contained concept. Its graphical representation is a clear diamond shape on the containing concept end of the relationship.
Example: a Sample is a collection or aggregation of Points.

Composition

Composition is a restriction of Aggregation which introduces a life cycle dependency: contained concepts exist only for the container and within the container's lifespan; conversely, the container exists only if the containees also exist. Its graphical representation is a black diamond shape on the container end of the relationship.
Example: a MixtureLaw is a composition of WeightedLaws, each characterized by its scalar weight.

Multiplicity of associations

Each end of the association may have a multiplicity indicating the number of instances of the concept (that is, the number of real objects belonging to this concept) simultaneously existing for each instance of the facing concept. This multiplicity can have one of the following values:

- *1*: one instance only;
- *n*: n instances (n positive integer);
- *m..n*: between m and n instances (m and n positive integers);
- ***: any number of instances (0 included);

- etc.

Example: A Sample is a collection made up of any number of Points (multiplicity * regarding the Point concept).

Role within an association

Along with the multiplicity, each end of the association can carry a name that designates the role played by the concept regarding the concept at the other end of the association.
Example: a Vector is associated to a Law. This Law (on its end of the association) is assigned the role of "joint law". Therefore the Law is a "joint law" for the Vector.

Association's name

Each association can have a name, generally taking the form of an action verb describing the association and the way it should be interpreted. If any disambiguation is required, it can be completed with an arrow indicating the direction for which the name makes sense.
Example: a Vector "creates" a Sample, therefore the association carries the name "creation".

2.3.2 Basic concepts

These concepts are the foundation bricks on which more advanced concepts will rely.

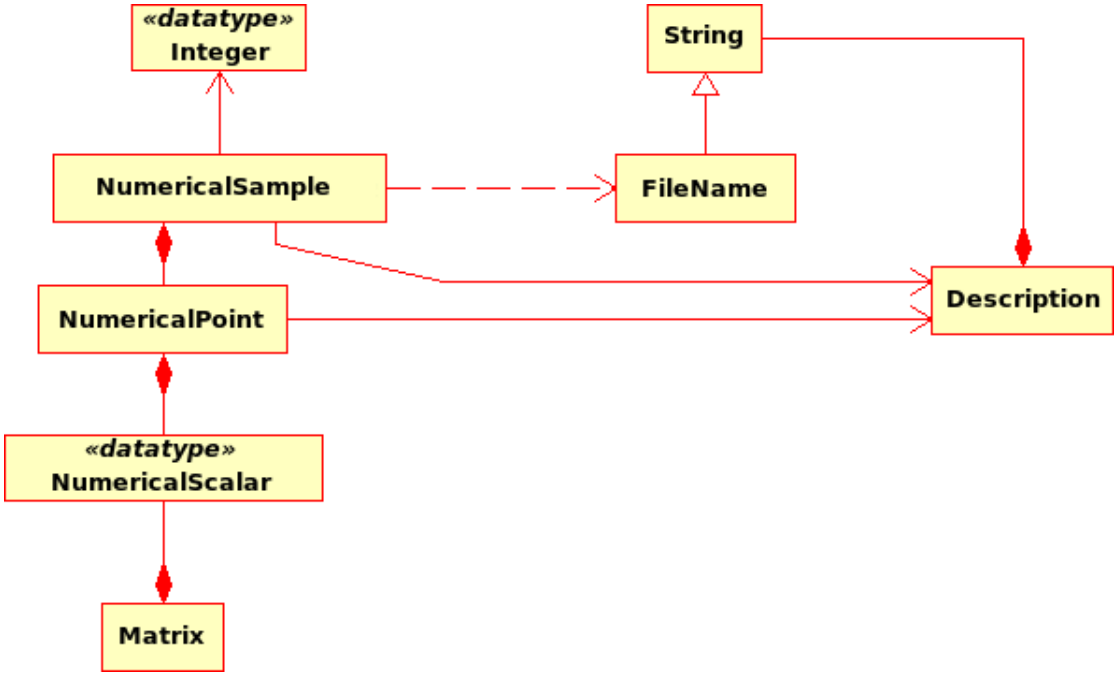


Figure 2.8: *Basic concepts.*

Integer

The integer is a basic type for the Open TURNS platform. It stores a whole positive or zero numerical data. Mathematically, it is a natural number. It may be used to describe the size of a sample, the dimension of a vector, the rank of a matrix, etc.

NumericalScalar

The NumericalScalar is another base type. It contains a numerical data that can be used in an uncertainty computation. Mathematically, it is a real number.

String

The String is a basic type that stores a text data of any length.

FileName

The FileName is a type derived from String specialized in the storage of filenames. Its syntax must correspond to a valid expression of a relative or absolute path to a file or directory on the disk.

NumericalPoint

The NumericalPoint is an elementary type made up of NumericalScalars. It covers the mathematical notion of a point in a multi-dimensional space, and once it is instantiated, its value remains constant. It can be used as a realization of a vector, mean value of a sample, etc. Its behavior is strictly deterministic.
The NumericalPoint gives access to its components. The NumericalPoint can be linked to a Description that will represent the names of its components.

Description

The Description is an elementary type made up of Strings. Its role is to provide a name or a description for an ordered set the same size as the Description. It can be associated with a NumericalPoint or a RandomVector to describe its components, with a NumericalSample to describe its points, etc.

NumericalSample

The NumericalSample is an elementary type made up of NumericalPoints. It is an homogeneous collection of points that have the same dimension. It covers the mathematical notion of a set of points. Given the multi-dimensional aspect of the points, the NumericalSample is also multi-dimensional. Its behavior is also deterministic. It can be used as a sample from a vector.
The NumericalSample gives access to its elements.

Matrix

The Matrix (see Figure 2.9) is an elementary type made up of NumericalScalars. It covers the mathematical notion of a matrix in its most general form (rectangular, asymmetric, etc.). The Matrix has a deterministic behavior. It can be specialized into more specific concepts such as square matrices, symmetric matrices, covariance matrices, etc.

2.3.3 Uncertainty concepts

RandomVector

The RandomVector (see Figure 2.10) is, from the user's point of view, the core concept. It covers both the mathematical notion of random vector and the programming concept of variable. It is essentially a multi-dimensional object, which means it always has a dimension (even if this dimension is 1). As a corollary, *there*

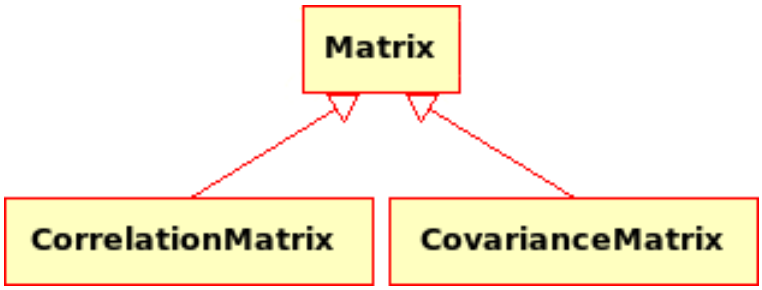


Figure 2.9: *Matrix concepts.*

is no notion of random scalar: within the platform, such an object is represented by a random vector whose dimension is 1.
The RandomVector is always associated with a Law called a joint law. This law models the vector's uncertain behavior. The RandomVector can be derived into a ConstantRandomVector whose realization is deterministically always the same NumericalPoint. The RandomVector can produce a NumericalPoint called realization and a NumericalSample called sample.
The RandomVector can be created from other RandomVectors; therefore there is a reflexive aggregation of the concept on itself, which bears the name *antecedent* on the thus created RandomVector end, so as to allow the created RandomVector to find its antecedents through the creation process. In other words, this mechanism allows a RandomVector $Y = f(X)$ (where X is the antecedent RandomVector of Y) to identify X and (if the RandomVector definitions are chained) all of the previous antecedents up to the first RandomVector.

FailureEvent

The FailureEvent (see Figure 2.11) is a concept that allows to define the limite state of a RandomVector "vector" with respect to a NumericalPoint "threshold", depending on a comparison operator "operator".

Law

From the modeling point of view, the Law is the core concept. It is the richest concept that encompasses most of the notions related to the treatment of uncertainties, as shown in Figure 2.12. The law is at the crossroads of the deterministic and uncertain domains. Like the other concepts of the platform, it is a multi-dimensional concept.
The previous section about the RandomVector already showed the relationship that links it with the Law: each RandomVector is defined by a joint Law. This Law can in turn be derived into UsualLaw, FunctionalLaw, MixtureLaw and AssemblyLaw.

UsualLaw

The UsualLaw is the basic brick for the Law concept. The concept of UsualLaw can in turn be derived into various laws: Exponential, Normal, LogNormal, Triangular, Uniform, GumbelMax, GumbelMin, Pearson, WeibullMax, WeibullMin, Beta, Spline, Gamma and Histogram. To each law corresponds an analytical formula describing the exact behavior of the said law. These laws are by definition multi-dimensional. However, in some cases, there is not always an analytical formulation for all of the dimensions. The implementation will therefore not be able to cover all possibilities.

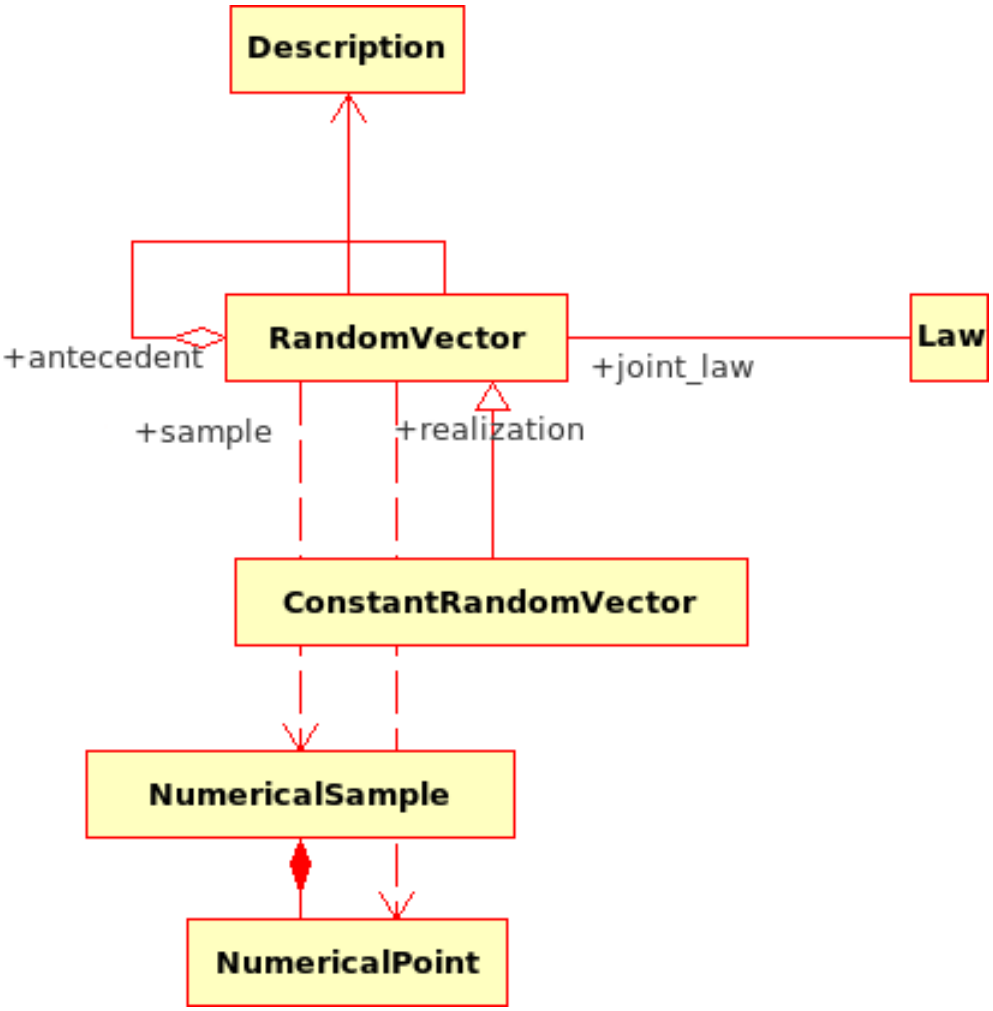


Figure 2.10: *The concept of random vector.*

AssemblyLaw

When a UsualLaw cannot be directly defined for a RandomVector of a given dimension, either because the law is unknown, or does not exist, or for any other reason, it is possible to combine Laws describing the marginal law of the RandomVector with the help of a Copula, which determines the dependency structure of this RandomVector. The Law assembly mechanism is supported by the concept of AssemblyLaw. The aggregation of Laws (called marginal_laws) is ordered and has the same size as the RandomVector.

WeightedLaw

The WeightedLaw is an arbitrary Law associated with a NumericalScalar that serves as a weight value in a linear combination of Laws. Each WeightedLaw is associated with a unique Law for which it defines a weight.

MixtureLaw

The MixtureLaw is a concept defining a linear combination of Laws. It is modeled as an aggregation of WeightedLaws, whose underlying relationship is the sum; each WeightedLaw brings the weight of the Law in

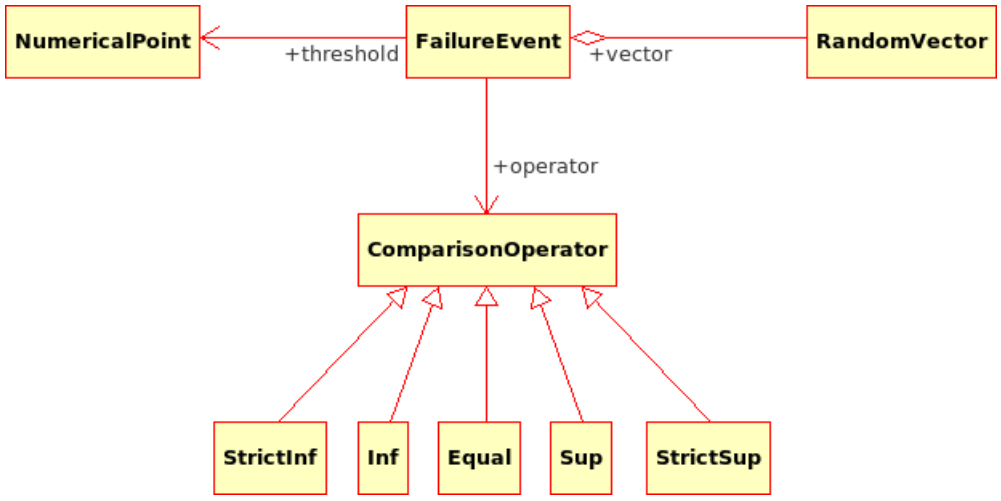


Figure 2.11: *The concept of failure event.*

the linear combination. The mathematical definition requires the *weights* to belong to the [0,1] interval and their sum to be 1.
Like all Laws, the MixtureLaw is a multi-dimensional concept and de facto requires the aggregation to include same-size Laws only.

FunctionalLaw

The FunctionalLaw is the Law resulting from a RandomVector being passed to a NumericalFunction. This sort of Law cannot be precisely defined in the general (non analytical) case, therefore the FunctionalLaw is determined by the input parameters that define it.
NB: it was decided to use the RandomVector rather than its joint law, because the RandomVector (contrary to the Law) can access all of its antecedents. This history of antecedents is also necessary to define the Law.

LawFactory

The LawFactory is a concept whose need arises during the design stage. It answers the need to designate a Law by its type when a Law-type object designates a specific instance (on this topic, refer to section **??**; for more details, see [GHJV]). The LawFactory therefore designates a family of Laws. It is an abstract concept that needs to be derived into concrete sub-concepts such as ExponentialFactory, NormalFactory, and so on, as many times as there are concrete instantiable classes.
Therefore, to each specialized UsualLaw corresponds a specialized LawFactory that can instantiate a specialized object: NormalFactory corresponds to Normal, ExponentialFactory to Exponential, and so on. NormalFactory can produce any Normal law, ExponentialFactory can produce any Exponential law, and so on for any other UsualLaw: LawFactory therefore covers the mathematical concept of a law family.
The LawFactory can produce Laws in different ways, therefore the concept is linked to the MomentsMethod and MaxLikelihoodMethod concepts. Both provide the algorithms needed to produce laws.

Kernel

On top of the law family notion supported by the concept of LawFactory, it is necessary to have a specific instance of each UsualLaw at one's disposal. In a way, this instance represents the generator for the law family.

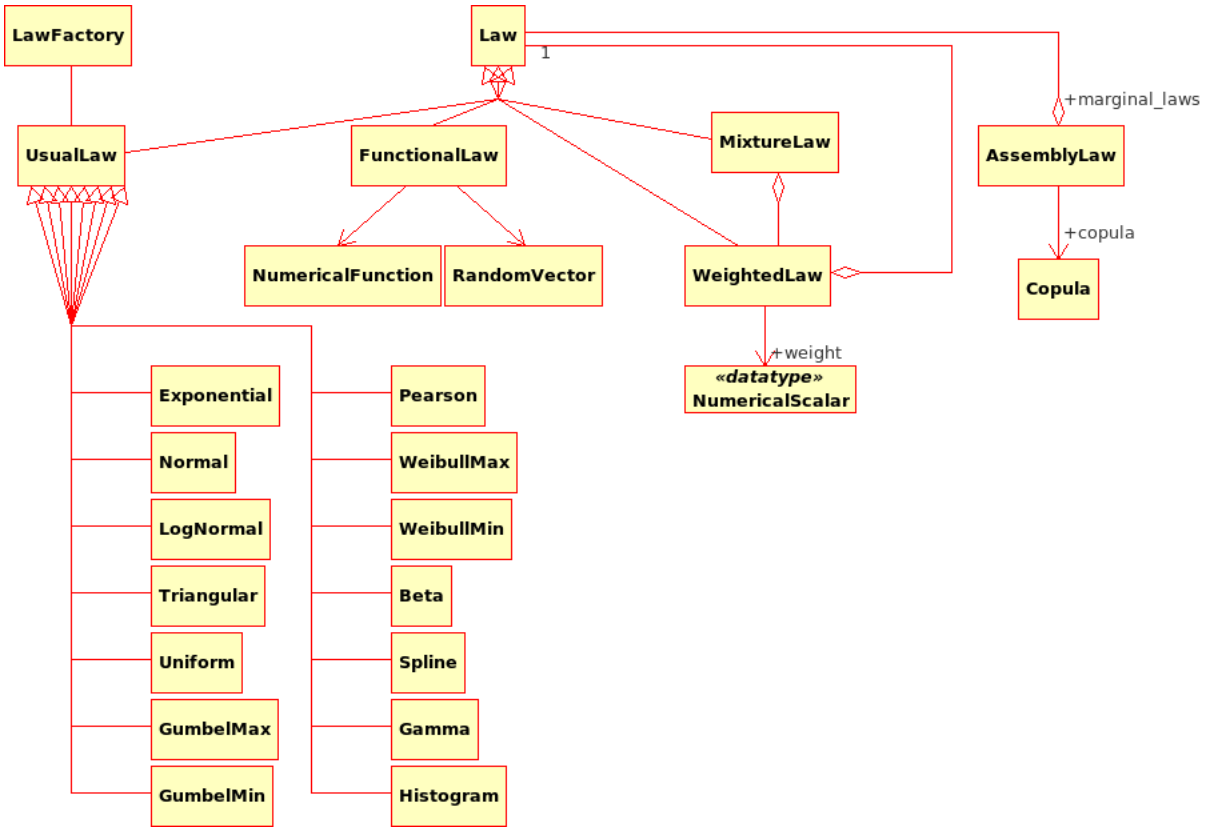


Figure 2.12: *The concept of uncertain law.*

This specific instance is described by the Kernel concept. Each UsualLaw can generate its Kernel: therefore a Normal generates NormalKernel, Exponential generates ExponentialKernel, and so on.

2.3.4 Function concepts

NumericalFunction

The NumericalFunction is the concept covering the notion of mathematical function. It is an entity that takes a NumericalPoint as input and produces a NumericalPoint as output. The dimensions of both NumericalPoint can differ.
The NumericalFunction can be derived into AnalyticalFunction, CodedFunction and TabulatedFunction. The analysis is not directly concerned with these derived concepts, but they indirectly show:

- the link with the external code that the CodedFunction will have to support;
- the need to have pre-wired functions within the platform (in the AnalyticalFunction);
- the need for user-defined functions (TabulatedFunction).

Copula

The copula is the entity that covers the notion of mathematical copula. Mathematically speaking, it is as much a numerical function as the NumericalFunction; however, these concepts have been made distinct because they

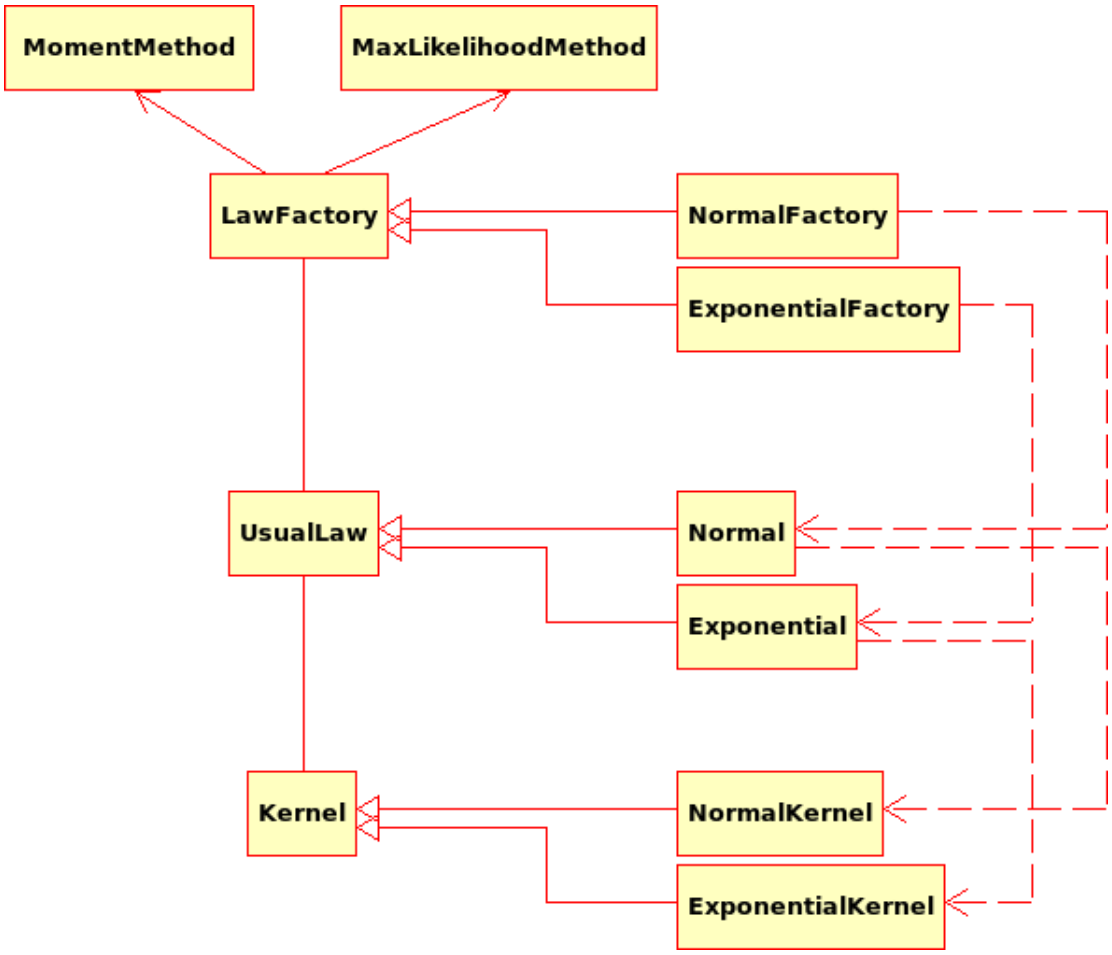


Figure 2.13: *The concept of law family.*

are used in very different contexts : the NumericalFunction is used in conjunction with RandomVectors and NumericalSamples, whereas the Copula is used only to define an AssemblyLaw. Moreover, as we will see in the design model, this separation is emphasized by the operations required from each concept.

2.3.5 Algorithm concepts

SimulationAlgorithm

The SimulationAlgorithm is the concept covering all of the uncertainty propagation methods in the platform. It is an abstract concept that derives into sub-concepts such as MonteCarlo, FORM, SORM, SRSS, and so on. A SimulationAlgorithm produces a SimulationResult as output, which stores all the information needed for the prioritization stage.

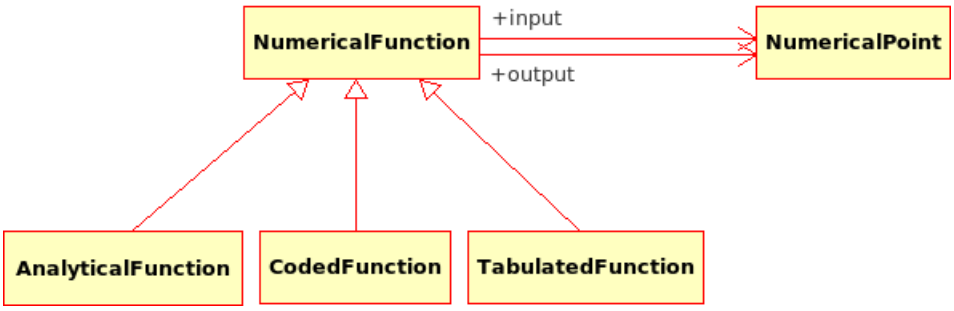


Figure 2.14: *The concept of numerical function.*

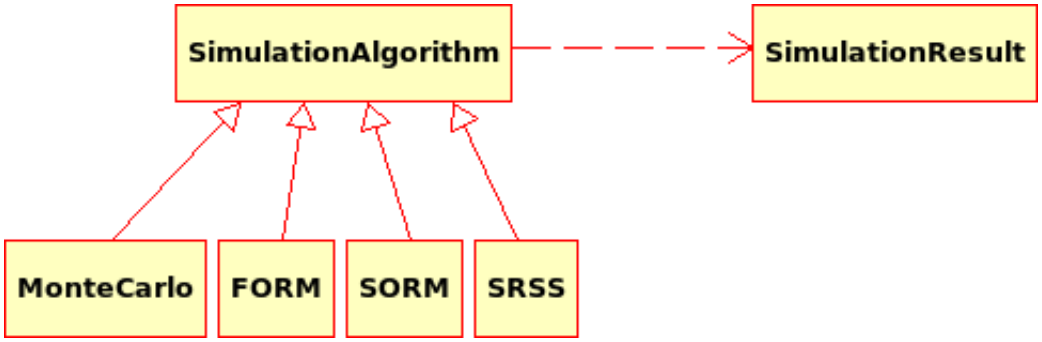


Figure 2.15: *The concept of simulation algorithm.*

2.4 Synopsis of a functional architecture

2.4.1 Packages

The tasks of modeling, analysis (described in this chapter) and design (described in next chapter) enable us to define concepts and objects closely related to one another. These will be grouped in the following packages:

- *GUI*: this package encompasses all objects directly linked to the graphical interface (e.g. Qt objects);
- *Graphical objects*: this package encompasses all of the objects whose role is to adapt field-specific and technological objects into GUI objects. It can be seen as the graphical interface for field-specific objects (e.g. histograms, pie charts, graphs of functions, and so on);
- *TUI*: this package encompasses all the objects interfacing with field-specific objets for a text-based use of the platform (e.g. SWIG objects);
- *Procedures*: this package encompasses any processing that is not (or cannot) be modeled with field-specific or technological objects (e.g. conditional random vector). It also contains sequences that represent typical action chains frequently used in uncertainty treatment studies;
- *Uncertain objects*: this package encompasses objects and concepts linked to the modeling of field-specific data with uncertain variables (e.g. random vectors, laws, etc.);
- *Simulation algorithms*: this package encompasses the various uncertainty propagation algorithms supported by the platform and the objects associated with the results (e.g. Monte-Carlo, FORM, SORM, etc.);

- *Statistical objects*: this package encompasses the objects and concepts associated with the statistical processing of field-specific data (e.g. sample);
- *Optimization functions*: this package encompasses the standard optimization methods needed for the simulation algorithms (e.g. SQP);
- *Basic numerical functions*: this package encompasses the mathematical and numerical functions standardly distributed with the platform (e.g. log, exp, sin, cos, etc.);
- *Basic classes*: this package encompasses the utility classes on which all of the higher level packages rely (e.g. String, FileName, Matrix, Tensor, etc.).

2.4.2 Layers

The previously introduced packages can also be grouped into software layers, each layer representing a different abstraction level of the Open TURNS platform.
As can be seen in Figure 2.16, the layers can be stacked as follows:

- *External services*: these are all the development prerequisites on which the platform relies, as well as the services offered by passive actors such as the external code or the storage system;
- *Basis brick*: it encompasses the packages offering core statistical and mathematical services for the platform;
- *Field-specific brick*: it is the essential layer that represents the core of the platform, in which are to be found packages responsible for the modeling and the propagation of uncertainties;
- *Sequences*: this layer assembles components from the field-specific brick to carry out complex uncertainty processing tasks;
- *User interface*: it is the layer responsible for the display and abstraction of field-specific objects into cognitively representable elements.

The perimeter of the Open TURNS platform is the solid border that surrounds the four top layers. The bottom layer representing the external services does not strictly belong to the platform: it is made up of prerequisites that are not included in the developments and in the modeling.
Cutting out the functional architecture into layers like this has direct implications on the platform's development process: the top layers will have to be developed on the basis of the bottom layers. Therefore, the development will be carried out chronologically, from the bottom layers to the top ones. This does not mean that one layer has to be entirely finished before the development of the next one starts. However, the packages on which one layer relies will have to be at least partially developed before actually beginning the development of the said layer's packages.

2.4.3 General diagram

Figure 2.16 shows the general diagram of the Open TURNS platform and its brick structure:

- *Interface brick*: used to provide communication with the operating system and the platform's prerequisites;
- *Basis brick*: basic components of the platform on which the functional, higher-level bricks rely;
- *Field-specific brick*: components used to model an uncertainty treatment study (quantization, propagation, prioritization, etc.);

- *UI brick*: components used to manipulate the platform and to visualize results.

The Storage service has the responsibility to store the computated data; the Distribution service's role is to distribute computations on a computer network. Both are considered non-critical: their development can be postponed in time in order to focus on the platform's core.
All of these elements make up the development perimeter within the Open TURNS project.
The Plugin component is in charge of the communication between the platform and the external code. The platform will standardly offer a version providing a mechanism for generic communication with external codes, but aside from this version, the Plugin component is outside of the project's scope: its implementation is the responsibility of the external code, which will have to follow the API provided by Open TURNS.
Figure 2.17 describes the architecture of the Open TURNS platform from a functional angle. It shows the packages resulting from the previously described concept groups, as well as other packages resulting from the design stage, the use cases or the general specifications of the platform. The layers are also represented as frames around the packages.

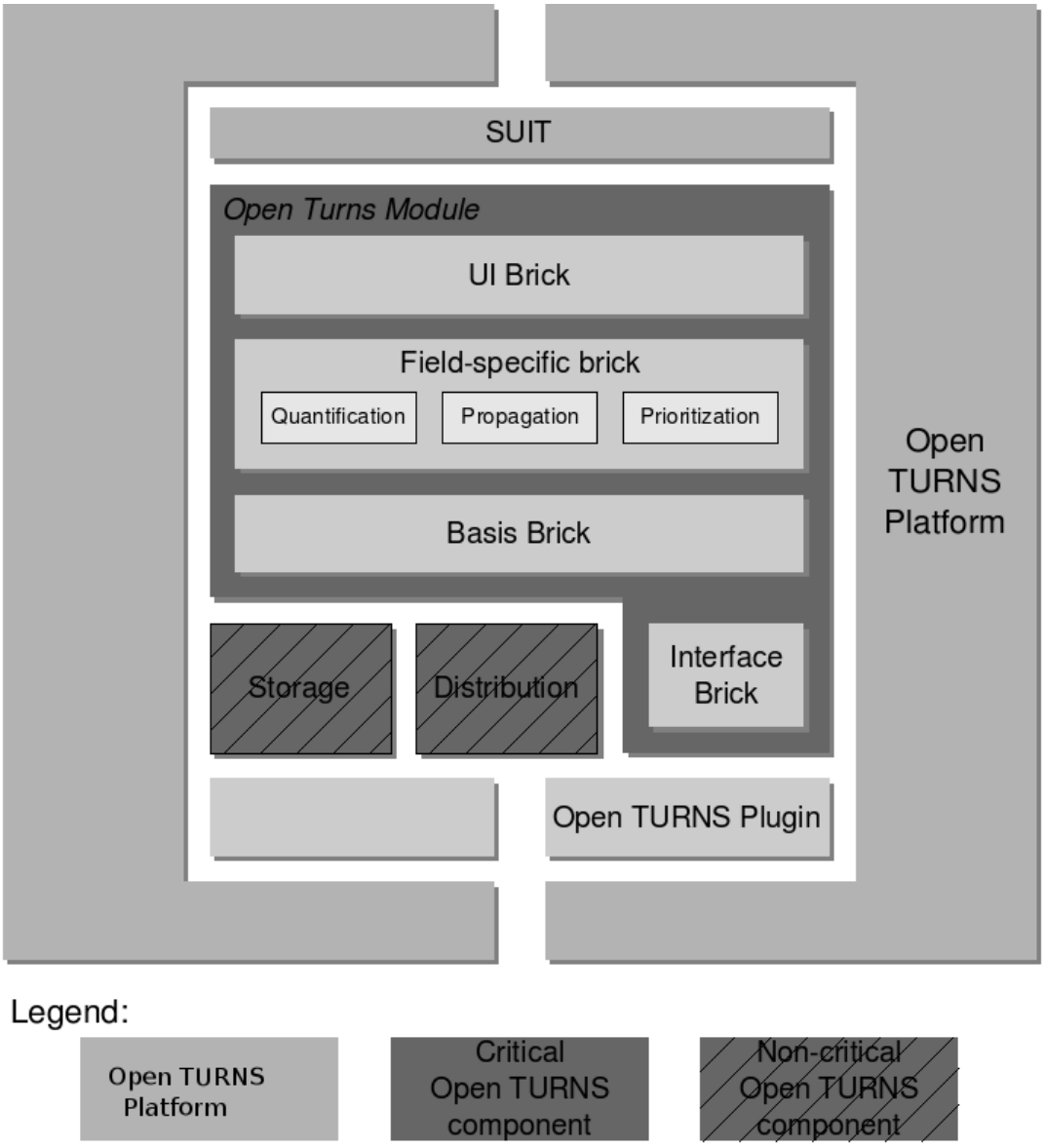


Figure 2.16: *General diagram.*

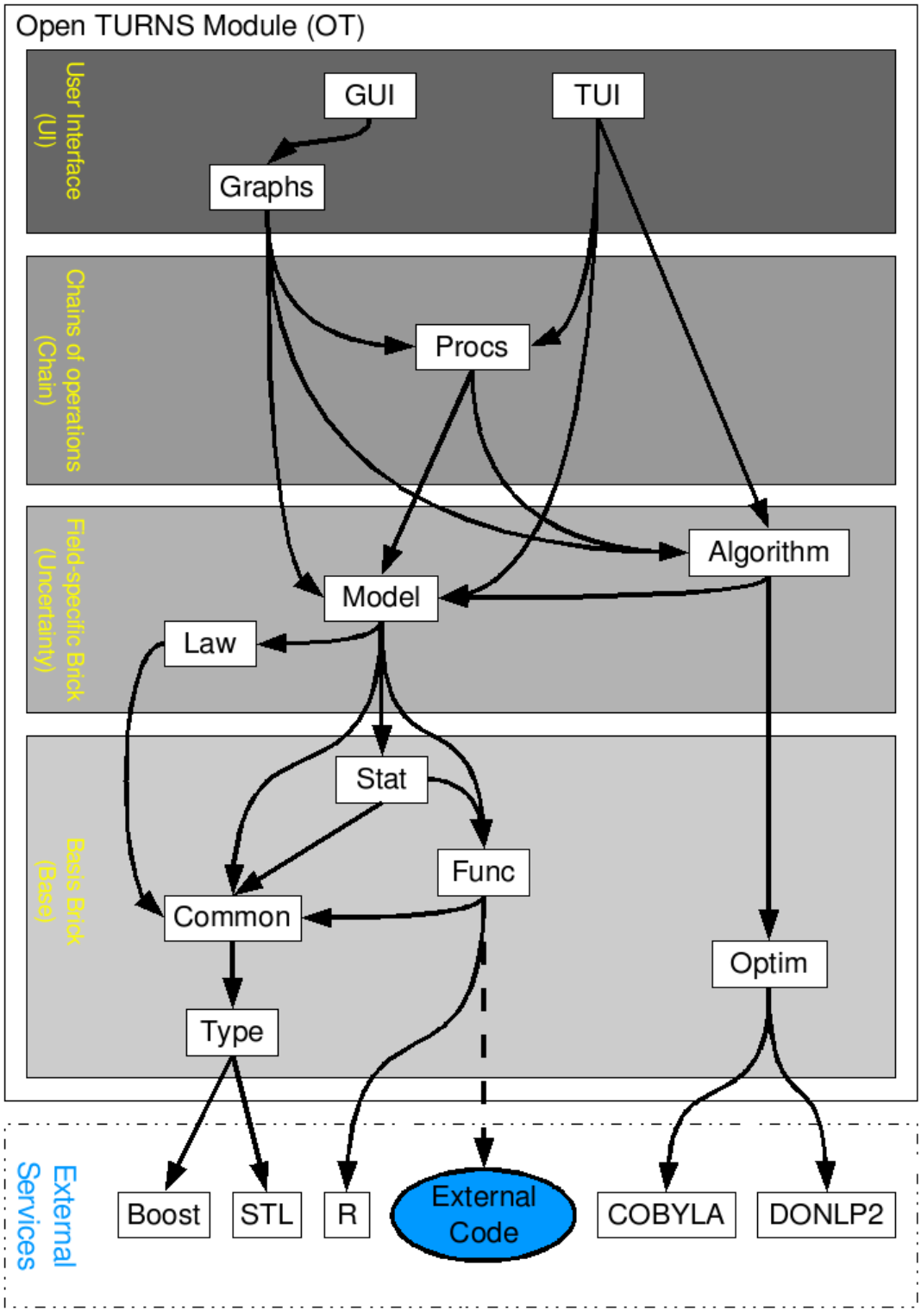


Figure 2.17: *Functional diagram.*

Chapter 3

Technical architecture

This chapter details the technical elements required by the Open TURNS platform, namely the system requirements, the tools and the development environment of the project.

3.1 Target platforms

The Open TURNS platform is meant to carry out uncertainty treatment studies in a scientific environment. Most of the scientific codes being available on Unix platforms, Open TURNS is naturally designed to run on this family of systems. Unix being a standard with multiple implementations, available on different architectures, this gives a wide choice of target platforms.
Linux is currently the most attractive Unix system for the Open TURNS project, it was chosen as the main target system for the project's development as well as for the delivery of the different versions.
The partners involved in the project have each chosen different Linux distributions, for technical and historical reasons. Therefore, it was decided to support several distributions, a choice that should not be seen as final or minimal. The distributions considered here include for example the list given in Table 3.1

Table 3.1: *Examples of Linux distributions supported by the project's parters*

Linux distribution	**Version**
Debian	Sarge
Mandriva	2005

However, there are also uncertainty treatment studies carried out in the proprietary Windows environment. While this system is not identified as a target for the project, developments should be carried out so as to facilitate the port to Windows.

3.2 Namespace

Both the modular structure of the Open TURNS platform, and the interactions it will be lead to have with other modules stemming from other projects, imply the possibility of name conflicts between the objects and the classes.
To deal with this issue, the project uses namespaces in order to isolate identifiers. Each package has its own namespace. The advantage of this system is twofold, since it also enables to classify objects in a named space and therefore helps structuring the project.

The namespaces are organized in a tree structure. The top-level namespace is designated as OpenTURNS and aliased as OT. It contains the namespaces Base, Uncertainty, Chain, and UserInterface (aliased as UI). Base contains the namespaces Common, Func, Stat, and Type. Uncertainty contains Algorithm, Distribution, and Model.
Developments have to abide by this naming system in order to prevent any conflict with other tools.

3.3 Internationalization

The Open TURNS platform is meant to be widely distributed within the scientific community revolving around probability and statistics, which is essentially an international community. Therefore, the platform should be designed so as to be adjustable to the users, particularly those who do not speak English[1].
This involves not using any messages directly in the source code of the platform, but rather to create a resource catalogue that can be loaded, according to the locale setting of the user, when the application is launched.
Another consequence of internationalization is the need for the Unicode extended character set (see [UNI]) to be used for all strings.

3.4 Accessibility

The Open TURNS platform shall be accessible to disabled users. This has implications on the ergonomy and the design of the User Interface, particularly the GUI which should offer keyboard shortcuts for any available function as well as keyboard-based (rather than mouse-based) mechanisms to handle and select objects.

3.5 Tools

3.5.1 Tool evolution policy

The tools chosen for the development of the platform are listed in Table 3.2

The versions given here are only meant as indications and other versions may be used. However, in case of compatibility issues arising from the use of other packages than those suggested here, support may not be the responsibility of the project.

3.5.2 Programming conventions

The present document does not deal with the the programming conventions. These are described in a separate document cited in the bibliography under the reference [**?**].

3.5.3 Version control

The present document does not deal with the version control policy, which is described in a separated document cited in the bibliography under the reference [**?**]

[1] English has been chosen as the native language for the Open TURNS platform.

Table 3.2: *Software development tools*

Category	Name	Version
Configuration	Autoconf	Current version. 2.59 or later
Compilation	Automake	Current version. 1.9 or later
Library support	Libtool	Current version. 1.5.6 or later
C++ compiler	Gcc	3.3.5 or later
Python language support	Python	2.3.5 or later
C++/Python wrapper	SWIG	1.3.24 or later
Graphic library	Qt	3.3.3 or later
Statistics library	R	2.0.1 or later
Version control	Subversion	1.1 or later
	flex	2.5.33 or later
	python-imaging (PIL)	1.1.6 or later
	xerces-c	2.7
	boost	1.30 or later
	BLAS	3.0 or later
	LAPACK	3.0 or later

Bibliography

[Ale] Andrei Alexandrescu. *Modern C++ design, Generic programming and design patterns applied.* Addison Wesley.

[Aus] Matthew H. Austern. *Generic programming and the STL, Using and extending the C++ Standard Template Library.* Addison Wesley.

[BOO] BOOST website. `http://www.boost.org/`.

[CSFP] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Micheal Pilato. *Version Control with Subversion, for Subversion 1.1.* Book compiled from revision 1337.

[Dal] Matthias Kalle Dalheimer. *Programming with Qt.* O'Reilly.

[Del] Claude Delannoy. *Programmer en langage C++.* Eyrolles.

[Dut] Anne Dutfoy. Partenariat EDF-EADS-Phimeca, Contenu méthodologique d'Open TURNS version standard. Note EDF R&D HT-52/05/021/A.

[ES] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual, ANSI base document.* Addison Wesley.

[GHJV] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns.* Addison Wesley.

[LA] Mark Lutz and David Ascher. *Learning Python.* O'Reilly.

[LB] Bil Lewis and Daniel J. Berg. *Threads primer, A guide to multithreaded programming.* Prentice Hall.

[Lut] Mark Lutz. *Programming Python.* O'Reilly.

[Meya] Scott Meyers. *Effective C++.* Addison Wesley.

[Meyb] Scott Meyers. *Effective STL, 50 specific ways to improve your use of the Standard Template Library.* Addison Wesley.

[Meyc] Scott Meyers. *More effective C++, 35 new ways to improve your programs and designs.* Addison Wesley.

[Mul] Pierre-Alain Muller. *Modélisation objet avec UML.* Eyrolles.

[OT] Open TURNS project public website. `http://www.openturns.org/`.

[OTD] Open TURNS project public website. `https://81.80.78.196/Incertitude`.

[PY] Python website. `http://www.python.org/`.

[R] R project website. `http://www.r-project.org/`.

[SAL] SALOME project web site. `http://www.salome-platform.org/`.

[SVN] Subversion website. `http://subversion.tigris.org/`.

[SWI] SWIG website. `http://www.swig.org/`.

[UNI] Unicode website. `http://www.unicode.org/`.