



**HAL**  
open science

## Open-PEOPLE : Ergonomics

Jonathan Ponroy, Kévin Roussel, Olivier Zendra, Dominique Blouin

► **To cite this version:**

Jonathan Ponroy, Kévin Roussel, Olivier Zendra, Dominique Blouin. Open-PEOPLE : Ergonomics. [Technical Report] 2011, pp.37. inria-00624000

**HAL Id: inria-00624000**

**<https://inria.hal.science/inria-00624000v1>**

Submitted on 16 Sep 2011

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



## Open-PEOPLE

*Open Power and Energy Optimization Platform and Estimator*

---

### Internal Deliverable 2.8

*Ergonomics*

	<b>Document Manager</b>	<b>Contributors</b>	<b>Checked by</b>
<b>Name</b>	Jonathan PONROY	Kévin ROUSSEL INRIA NGE	Kévin ROUSSEL Dominique BLOUIN Olivier ZENDRA
<b>Contact</b>	Jonathan.Ponroy@inria.fr		
<b>Date</b>	8-Jul-2011( <b>version 0.5</b> )		
<b>Summary</b>	This document describes the study and choices made to offer a good ergonomic for the Open-PEOPLE software platform.		

## Table of contents

1.Preface.....	3
1.1.Versions.....	3
1.2.Table of references and applicable documents.....	4
1.3.Acronyms and glossary.....	4
2.Executive summary.....	7
3.Scope of the document.....	7
4.Analyses.....	8
4.1.User profiles.....	8
4.2.Researcher profile and tasks model.....	8
4.3.Industrial engineer profile and tasks model.....	10
4.4.Technical constraints.....	11
4.4.1.Eclipse platform constraints.....	11
4.4.2.Eclipse GMP constraints.....	13
4.4.3.Meta model constraints.....	16
4.4.4.Modular constraints.....	17
5.Style guide recommendation.....	18
5.1.Lookup table creation (importation) and edition.....	18
5.1.1.Importing OPHWP measurements into a new LUT.....	18
5.1.2.Using the LUT editor.....	21
5.2.Quantity kind and unit creation.....	24
5.2.1.Ergonomic evaluation.....	28
5.3.Math law creation.....	29
5.4.PCMD creation.....	31
5.5.PCAO: weaving model.....	35

# 1. Preface

## 1.1. Versions

<i>Version</i>	<i>Date</i>	<i>Description &amp; rationale of modifications</i>	<i>Sections mainly modified</i>
0.1	8 November 2010	First version	–
0.2	5 April 2011	Added constraints and UI recommendations	4.4 and 5
0.3	30 June 2011	Added the LUT section	06/01/11
0.4	6 July 2011	Made some corrections to the text	2,3,4,5, and 6
0.5	8 July 2011	Move use case chapter in D2.1	

## 1.2. Table of references and applicable documents

<i>Reference</i>	<i>Title &amp; edition</i>	<i>Author</i>	<i>Date</i>
Open-PEOPLE project: deliverable D2.1	Specification for the software platform definition (V1)	Sophie ALEXANDRE	18 May 2010
Open-PEOPLE project: deliverable D2.2	Tools Integration Protocol	Kévin ROUSSEL	10 September 2010
Open-People project: deliverable D4.1	Basic Components Model Homogenizations	Dominique BLOUIN	15 July 2010
Open-People project: deliverable D4.2	Generic models, interoperability and interchangeability	Dominique BLOUIN	N.D.

## 1.3. Acronyms and glossary

<i>Term</i>	<i>Definition</i>
OP	<b>Open-PEOPLE:</b> the name of the project we're talking about, and by extension, the name of the platform(s) developed within this project.
OPSWP	<b>Open-PEOPLE's software platform:</b> the central piece of code, around which resolve all of the software developments undertaken within the Open-PEOPLE project.
Java	<b>The Java programming language, and the platform on which it executes,</b> once compiled into <i>bytecode</i> (Java virtual machine). Both created by <i>Sun Microsystems</i> (now <i>Oracle</i> ), and defined respectively in: “ <i>The Java Programming Language</i> ” and “ <i>The Java Virtual Machine Specification</i> ” books.
JVM	<b>Java Virtual Machine:</b> main component of the Java platform, providing the latter with its portability and security (among others). The OPSWP is designed to be executed by the JVM.
Bytecode	<b>JVM bytecode:</b> machine code for the JVM, that is: program code directly executable by the JVM. The OPSWP and its plug-ins will ultimately take the form of executables coded in this format.
JCP	<b>Java Community Process,</b> by which <i>Sun Microsystems/Oracle</i> designs the evolution of Java (both the language and the platform) in cooperation with its users.
JSR	<b>Java Specification Request:</b> official proposals of evolution for Java (both the language and the platform), made within the JCP.

<b>Term</b>	<b>Definition</b>
OSGi	Formerly meaning “ <b>Open Services Gateway initiative</b> ”, it is a framework designed by the <i>OSGi Alliance</i> , in order to provide dynamic modularization and <i>service-oriented architecture</i> (SOA) for Java-based applications.
CRI-NGE	<b>Centre de Recherche INRIA – Nancy Grand-Est:</b> the place where the development of OPSWP is managed and—for a major part—realized.
PCMD	<b>Power Consumption Models Development:</b> the set of features offered by the OP platform (hardware & software) which allow to develop and validate new power consumption models.
PCAO	<b>Power Consumption Analysis and Optimization:</b> the set of features offered by the OP platform which allow embedded system designers to estimate and optimize the energy consumption of the systems they create, using predefined consumption models.
GUI	<b>Graphical User Interface:</b> OPSWP will offer a GUI to allow for easy yet efficient and productive use.
RCP	<b>Rich Client Platform:</b> a framework (comprising widgets, desktop/workbench, extensible architecture, update management, and other paradigms) on which one can build coherent, robust, standardized, and feature-rich desktop stand-alone and client applications (thus named “rich client” applications). The OPSWP is to be built on Eclipse project's RCP.
IDE	<b>Integrated Development Environment:</b> a graphical application/framework gathering tools in order to help developers and programmers to perform more easily and efficiently their work; one of the most advanced, best known and most used IDE for the Java platform is the Eclipse JDT, built upon the RCP produced by the Eclipse project. A variant of this JDT is precisely specialized in development of applications based on Eclipse's RCP.
COP	<b>Component Oriented Programming :</b> Programming method which consist to use modular approach to make software architecture using pre-existing components.
JAAS	<b>Java Authentication and Authorization Service :</b> is a <a href="#">Java</a> security framework for user-centric security to augment the Java code-based security. Since <a href="#">Java Runtime Environment</a> 1.4 JAAS has been integrated with the JRE - previously JAAS was supplied as an extension library by Sun.
EMF	Eclipse Modeling Framework : is a modeling framework and code generation facility for building tools and other applications based on a structured data model.
GMF	Graphical Modeling Framework : provides a model-driven approach to generating graphical editors in Eclipse.

---

<i>Term</i>	<i>Definition</i>
GMP	Graphical Modeling Project : provides a set of generative components and runtime infrastructures for developing graphical editors based on <a href="#">EMF</a> and <a href="#">GEF</a>
OPHWP	<b>Open-PEOPLE's hardware platform</b> : the place where hardware measures are realized to be computed by OPSWP.

## 2. Executive summary

Our software platform is composed of several tools implemented as a set of Eclipse plugins using the Eclipse RCP standard “extension point” mechanism. This is the lightweight software integration model we decided to use during the conception of the OPSWP (see deliverable D2.2 for details), whose graphical consistency is guaranteed *a minima* thanks to the basic UI mechanisms (SWT widgets, workspace, etc.) of the Eclipse RCP.

Each one of the integrated software tools is part of the model-driven workflow used by our platform to implement its features; however, these tools are not basically conceived to work together, thus undermining the coherency and usability of the OPSWP. Since the usability of our future software platform is a major concern for us, we are putting much effort into the harmonization of the various software elements to offer the best possible user experience to the system architects who will be our main users.

The purpose of this document is to describe how we intend to provide good ergonomics to the OPSWP: we will especially explain the choices we made about the GUI conception. Ergonomics are only a question of usability, which depends on the good organization of GUI, on good use of visual components, forms, and so on. Ergonomics don't define neither the graphic charter nor the visual identity of our software: these “artistic” considerations are part of the project's task 6 (dissemination).

The present study is divided into several consecutive parts: we first begin by analyzing user profiles and their related tasks, then we review the technical and architectural constraints that are imposed to us. From all of these, we eventually build a style guide, that defines the usability requirements that shall be followed as guidelines to implement the graphical user interfaces (GUIs) of the OPSWP software elements with the expected consistency.

This document – not planned at the beginning – is part of task 2 of the OP project. This task is under the responsibility of the INRIA NGE team, and its main goal is the specification, conception and implementation of the OPSWP.

## 3. Scope of the document

The goal of this document is to present the background work realized on ergonomics, so as to make the OPSWP an integrated software toolset, instead of a mere concatenation of various tools.

It is supposed that the reader of the present document knows what OP is, and what goals it is supposed to achieve. This document won't talk about these subjects: this information can be found in the D1.1 to D1.4 deliverables of the project.

Moreover, this document focuses itself exclusively on the **software platform** (OPSWP). All data concerning the project's hardware platform is located in other deliverables (deliverables from task 3; plus D2.3 for the remote control of hardware platform).



## 4. Analyses

### 4.1. User profiles

The first step of our analysis work consists in defining the correct profiles for each possible kind of users our platform will have, each profile corresponding to a set of desired / employed features. As far as the OPSWP is concerned, we identified the following user profiles:

- **academic researchers**, doing fundamental research on electronic power consumption;
- **industrial systems engineers**, building new embedded electronic systems, and trying to optimize / minimize their power consumption.

These two populations don't have the same goals nor the same methods. Understanding how they work is the first essential step to best suit the OPSWP to their needs.

*A première vu, 2 types d'utilisateurs : l'employé d'une entreprise privée et le chercheur d'un centre de recherche. Les deux non a priori pas les mêmes besoin ni les mêmes but ce qui n'implique pas forcément que leurs tâches soient fondamentalement différents.*

*Démarche :*

*Relecture des précédents livrables pour essayer de trouver début de réponse à la question du but, du souhait mais également les tâches prévues → on a déjà 2 pseudo tâches PCAO et PCMD.*

*Interview avec un panel représentatif pour compléter ces 2 questions. Pour vérifier et compléter les tâches prévues, voir ce qu'il font avec leur logiciel et non notre plateforme concaténée (cat pour lorient, un autre logiciel pour inpixal, un autre pour thales) → quel but, comment et avec quelles informations. Si possible dans leur environnement de travail.*

### 4.2. Researcher profile and tasks model

We've conducted interviews with three different academic researchers and research engineers. These three interviews allowed us to determine the tasks model (flow) described in figures 1, 2 and 3 hereafter.

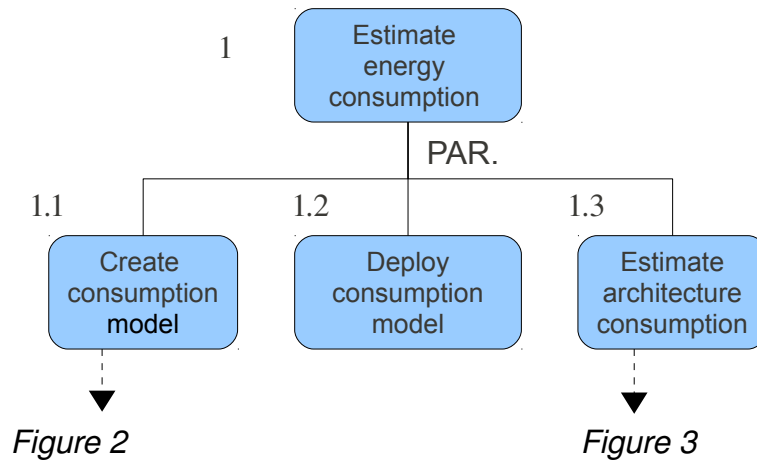


Figure 1: Researcher profile's main task flow diagram (PAR. == parallelizable, potentially simultaneous steps)

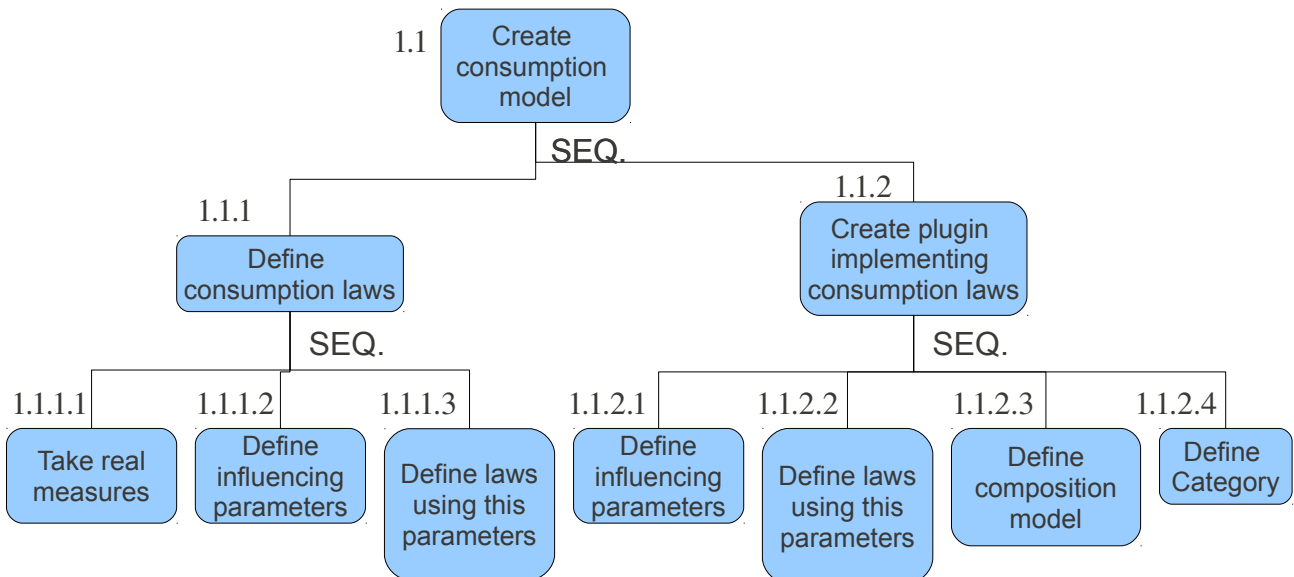


Figure 2: Consumption models creation task flow (SEQ. == sequential actions that need to be accomplished one after another)

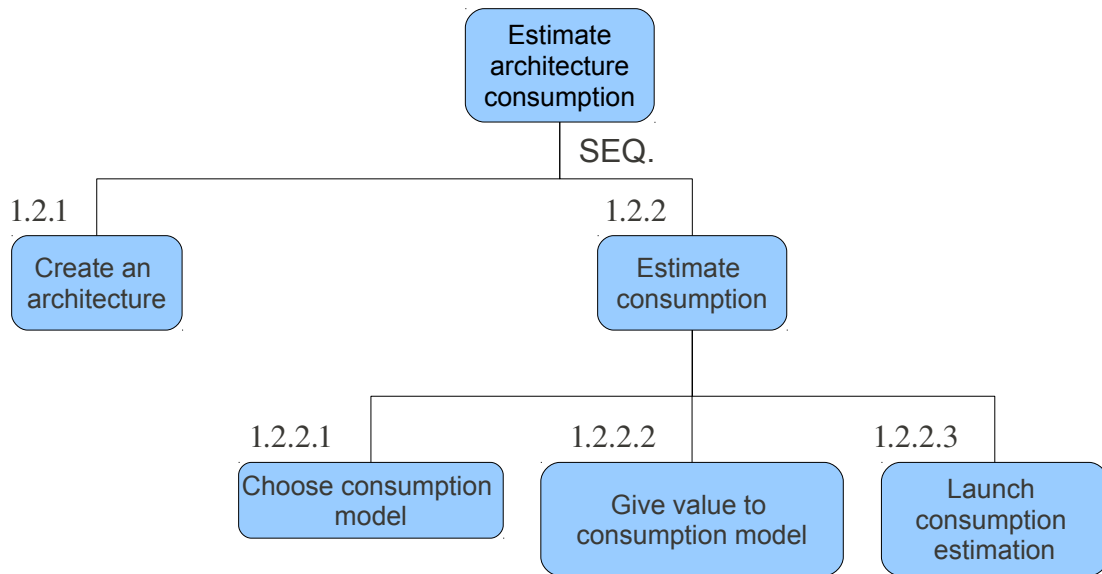


Figure 3: Consumption estimation task flow  
(SEQ. == sequential actions, that need to be accomplished one after another)

### 4.3. Industrial engineer profile and tasks model

We've had interviews with two industrial engineers. These interviews learned us that industrial engineers don't create consumption models: they only want to use them.

Currently, they have no tool allowing them to predict the power consumption of electronic devices *a priori*, : system components are chosen according to their intuition and experience ("board sizing"), then the final system consumption and performance are estimated from hardware models built from evaluation boards and devices. When this first step is complete, they write "data sheets" that they can then use to refine the final versions of their systems.

Thus, the tasks flow for industrial users is quite simple, and can be summarized by figure 4 hereafter:

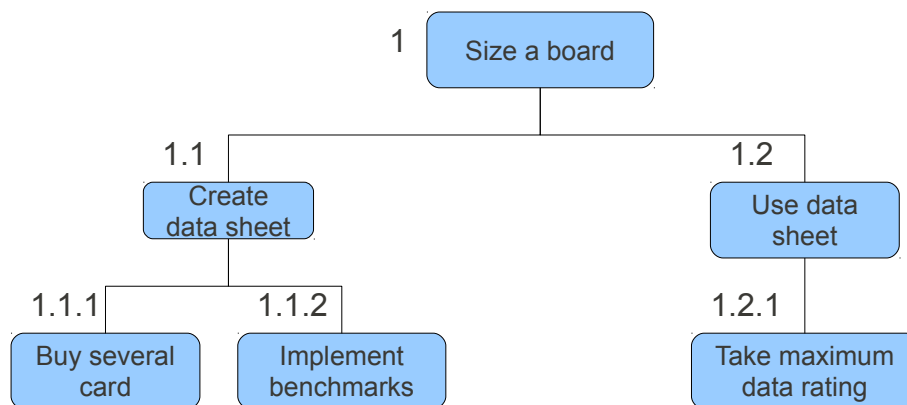


Figure 4: Industrial engineer profile's task flow

## 4.4. Technical constraints

The architectural and technical decisions we made during the OPSWP conception impose several constraints to us.

- The very first of these constraints is the use of the Eclipse rich-client platform (RCP) as the basis for our software developments. Consequently, we have to use the standard mechanisms of this RCP to build our software, that will be implemented under the form of Eclipse plugins (see D2.2 for more details).
- The second constraint is the use of model-driven development, which compels us to employ the GMP toolset (that includes the well-known EMF and GEF frameworks).
- The third constraint comes from the definition of the metamodel we use. This metamodel provides some necessary features and tasks that complete the researcher and industrial engineer profiles' task flows.
- The fourth and last constraint is the modularity of the OPSWP: the platform is composed of various tools. Some of these tools are developed by us and our partners – and can thus be developed and modified according to our needs and wishes – while on the contrary, we don't have such flexibility for other third-party tools.

### 4.4.1. Eclipse platform constraints

The Eclipse RCP framework defines some paradigms that every plugin has to follow in order to contribute UI elements to the RCP “workbench”. Consequently, one has to know and abide by these rules to provide GUI-based add-ons for the OPSWP. A large section of deliverable D2.2 (“Tools Integration Protocol”) is devoted to the Eclipse GUI elements, where one can find much details on this subject. Thus, we will only briefly summarize here the most important user interface elements that will be used in the OPSWP.

The editors and views are the two main kind of parts that compose the vast majority of the Eclipse workbench window, as we can see on figure 5 below. They are the main ways to interact with the “objects” that will be used in the OPSWP. While views are usually simple UI elements (i.e. containing only one list, one table or one text area), focused on a precise domain (e.g., they are the containers of choice to hold a file list, or a properties' table), the editors are quite complex elements, allowing for the complete edition of a given element (usually a file, but they can also be built to edit anything). Editors can be subdivided into many sub-editors contained in tabs (see for example the Eclipse `plugin.xml` editor). Both of these UI elements are arranged by Eclipse-specific layouts called perspectives that allow to describe what parts (views and editors) are open and how they are arranged on the workbench window.

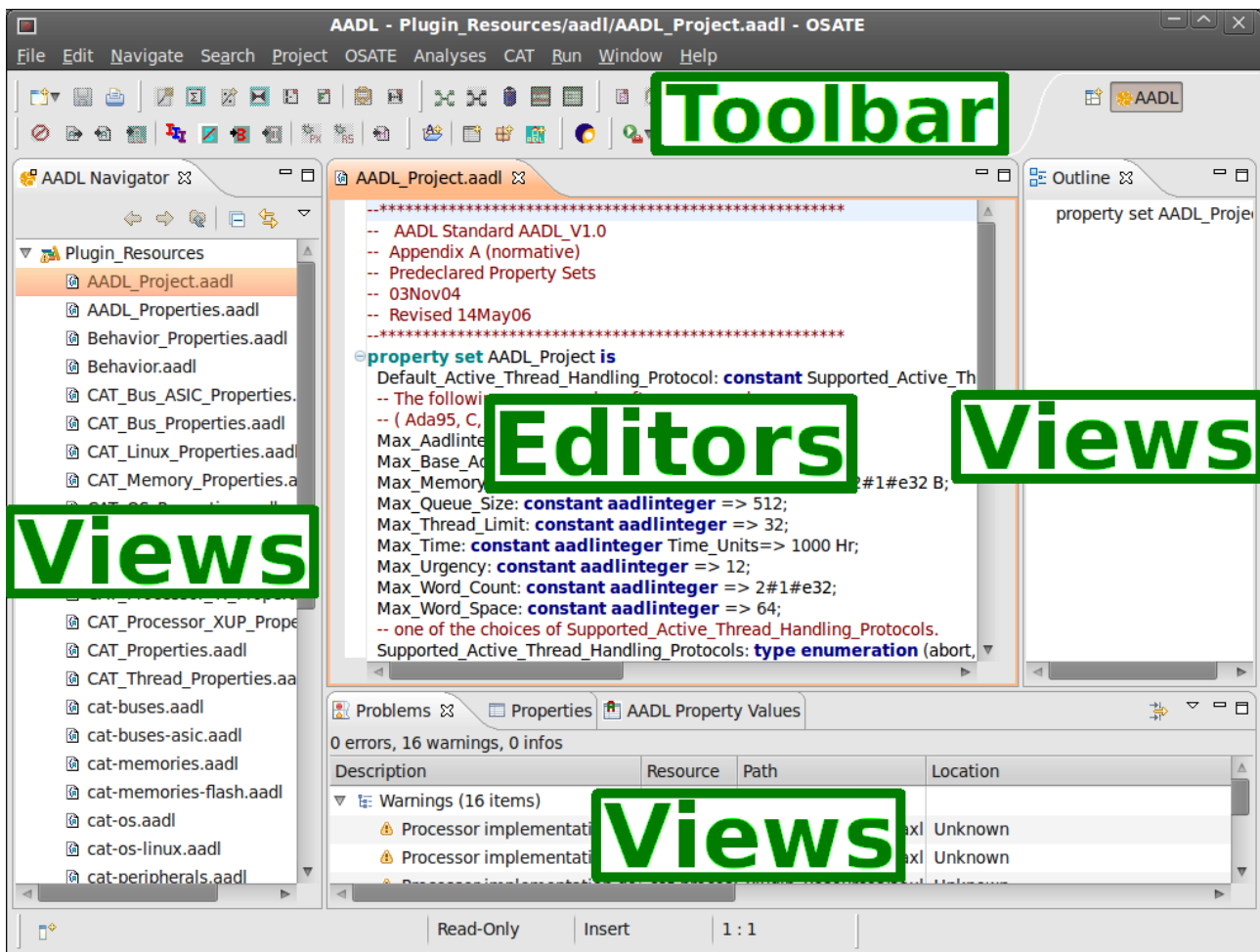


Figure 5: The different parts of the Eclipse workbench UI.

Wizards are also important UI elements which will be used in the OSPWP. They form an advanced kind of dialog boxes, helping the user to accomplish complex tasks step-by-step. Thus, these dialogs comprise several pages, that are shown one after another during the progression of the procedure to accomplish.

Both of these paradigms (Editors/views and wizards) are specific to Eclipse, and impose a visual organization that the OPSWP will have to respect. There are also some well-defined procedures that are to be followed in order to create these. But others, more classical UI elements – like menus, dialogs, and so on – are also available to build the OPSWP graphical interfaces.

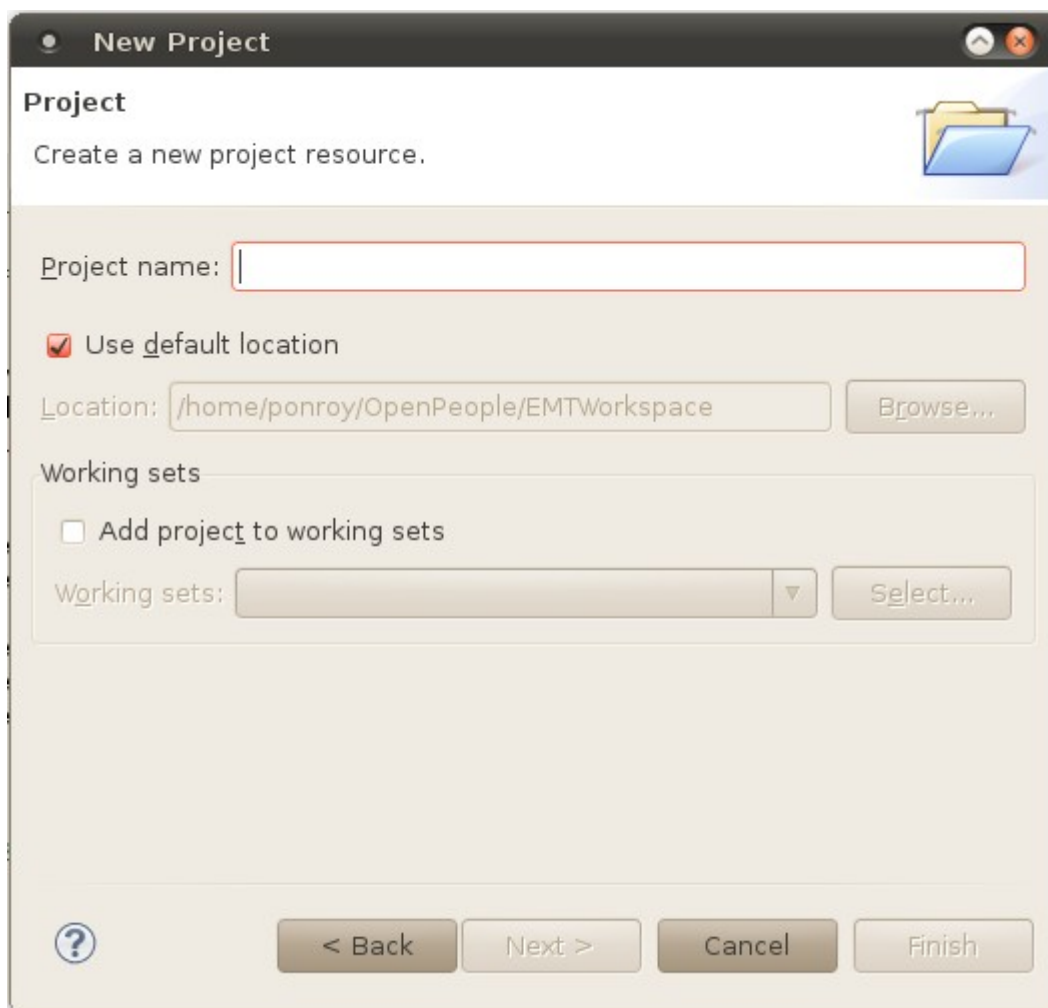


Figure 6: A wizard UI in the Eclipse platform.

#### 4.4.2. Eclipse GMP constraints

*Comment fonctionne emf; les différents modèle, les éditeurs automatiques, etc*

GMP (Graphical Modeling Project) is a project whose goal is to provide a set of automatic generation tools and runtime infrastructures for graphical editors development, based on the EMF and GEF frameworks as well as meta-models. In practical terms, GMP allows, starting from a meta-model definition, to automatically generate interactive editors or code to programmatically edit models (that is: instances of the meta-model). Figure 7 hereafter explains how GMP works:

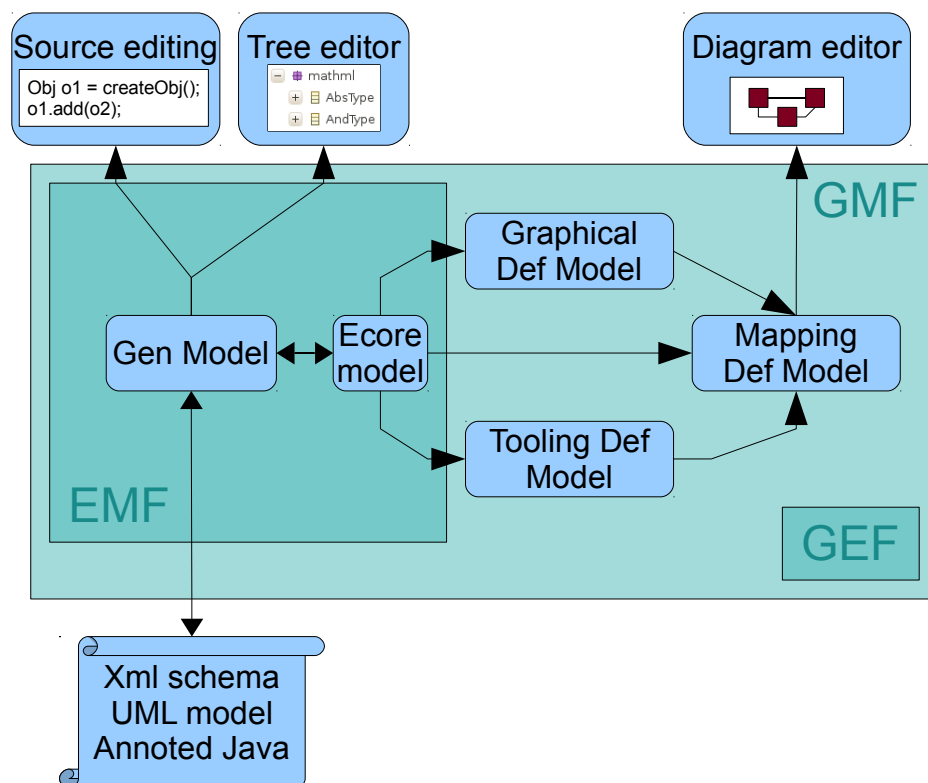


Figure 7: Functional blocks of GMP.

As one can see, interactive editors (i.e. Tree editor and Diagram editor) are generated automatically from the meta-model. They can be adapted and customized thanks to parameters or intermediate configuration models. So we can, for example, define an icon for an object of a meta-model, modify the default representation of an object in a diagram editor, and so on. But even if we can customize some properties, the global representation in the tree editor will remain a tree, as well as the global representation in the diagram editor will remain a diagram constituted of “boxes” linked with relations.

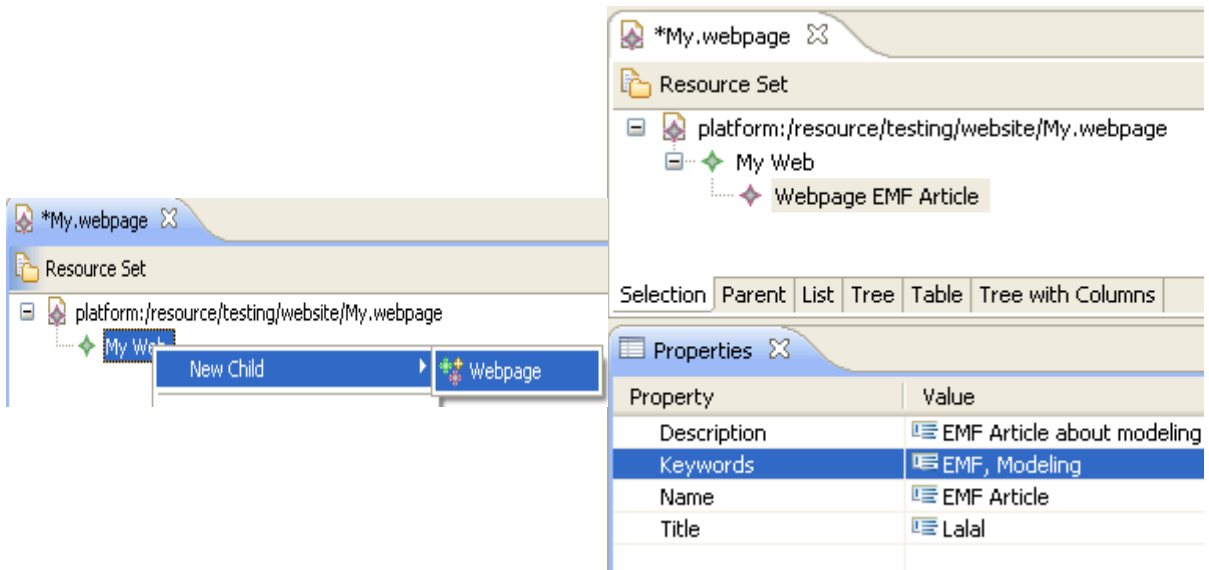


Figure 8: Global representation of a tree editor.

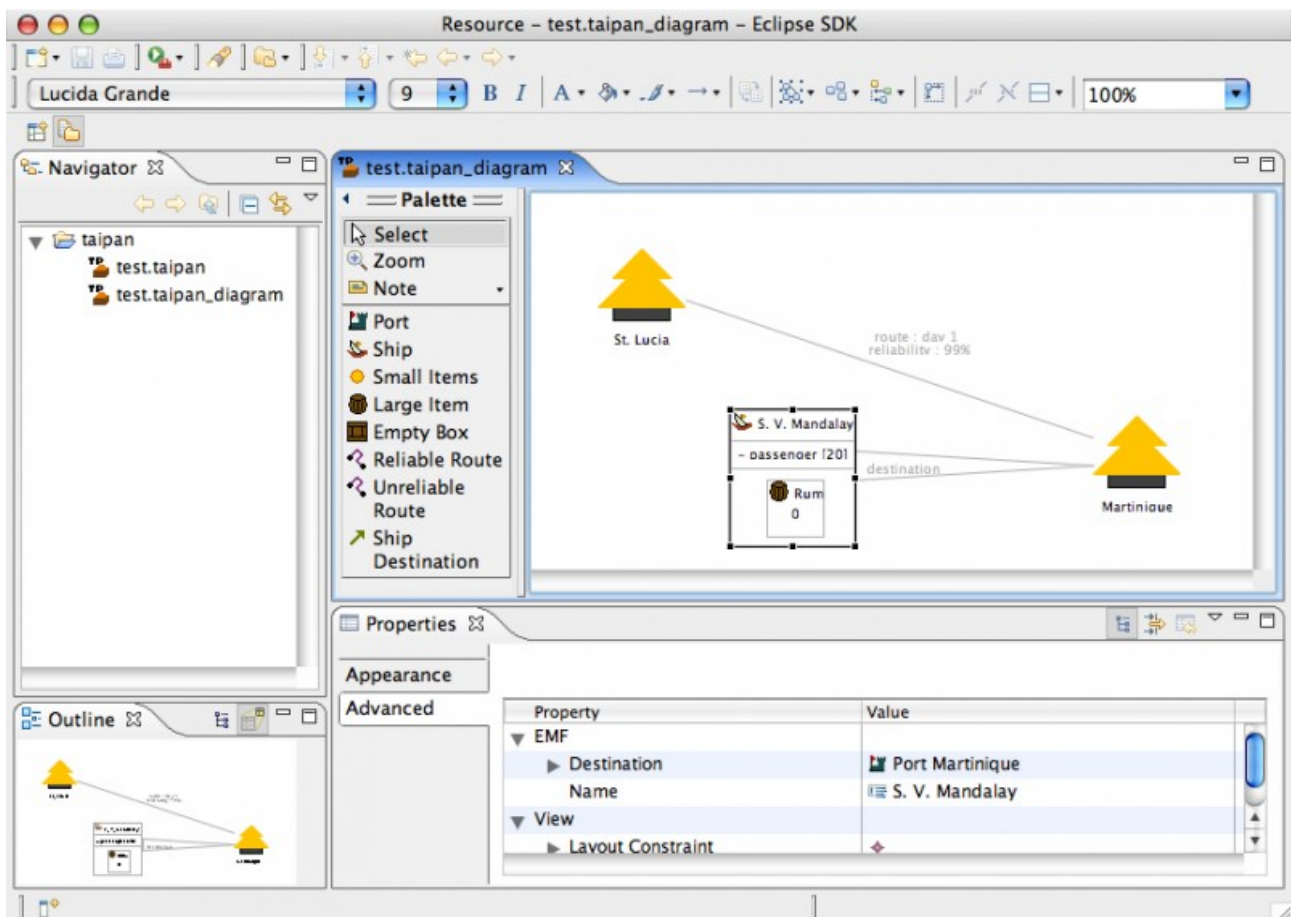


Figure 9: Global representation of a diagram editor.



That's why we have three kinds of available UI to edit a model :

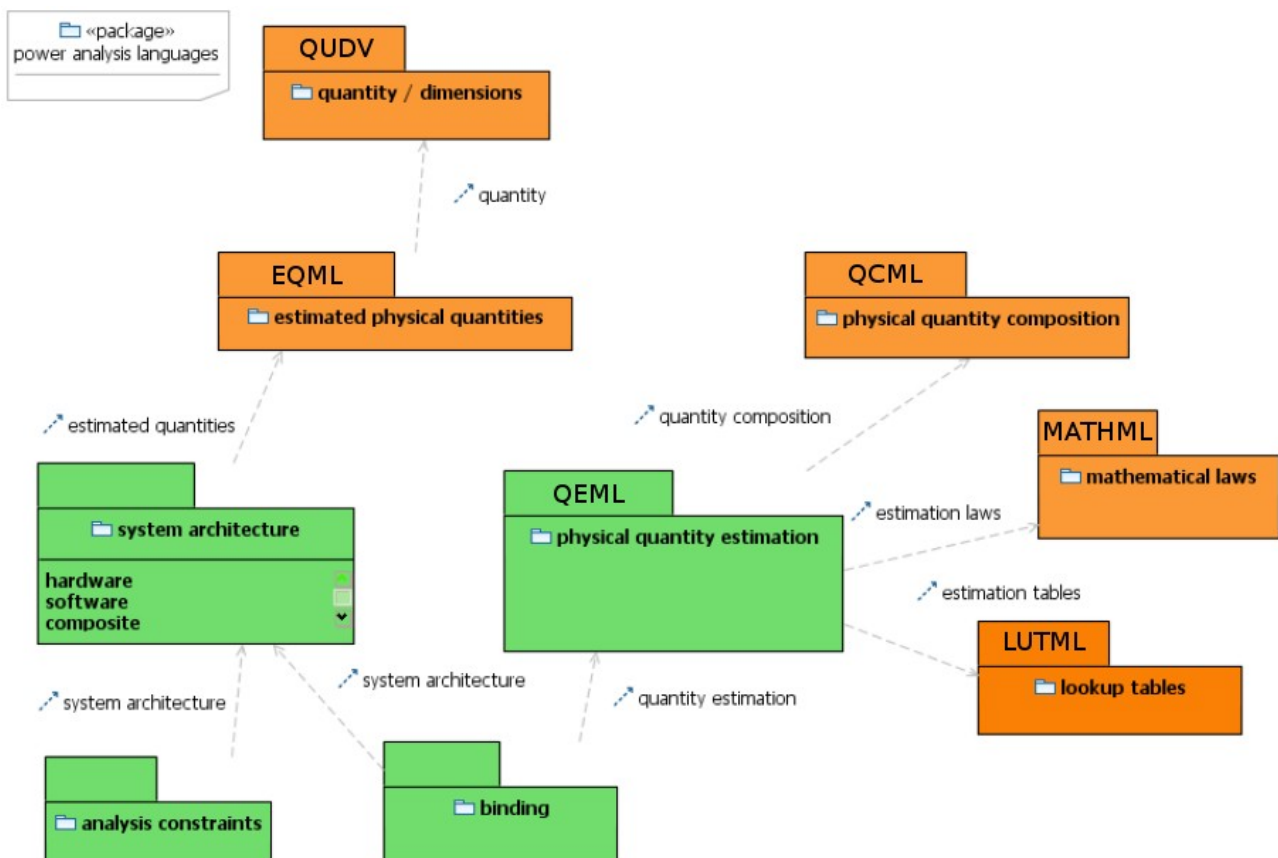
- a standard diagram editor with linked boxes;
- a standard tree editor;
- a customized editor, that we have to build from scratch using Eclipse common user interface elements (Editors, views, wizards, and so on).

#### 4.4.3. Meta model constraints

*Expliquer en quelques paragraphes les différents modèles.*

*Parler de la contrainte technique liée au MathML*

To support the features of OPSWP – which can be grouped into consumption model creation features (PCMD) and consumption evaluation features (PCAO) – we decided to use a multi-DSL based architecture (see deliverable D4.1 “Basic Components Model Homogenization” for all the details). The global view of the multi-DSL architecture is presented in the following diagram:



To create a consumption model, we then need to use these DSLs:

- MathML (Mathematical Modeling Language) : this model allows to define all kinds of math formula. Contrarily to the other DSLs used in OP (that we specifically created and tailored to our purposes), it is an international, widely-used, W3C-backed norm.

- QUDV (Quantities Units Dimensions Values): this model (Domain Specific Language: DSL) allows to define quantities (e.g.: length and mass) and units (e.g. respectivly metre and kilogram).
- EQML (Estimated Quantity Modeling Language): this model allows to define uncertainties for a quantity (e.g.:  $\pm 10\%$ , as well as more complex functions).
- QCML (Quantity Composition Modeling Language): this model allows to define compound quantities.
- LUTML (LookUp Table Modelling Language): this model allows to define a lookup table, i.e.: an associative array of one or more ordered variable(s) and values.
- QEML (Quantity Estimation Model Concepts): this model allows to define a consumption model; it is built by composing all of the previously cited DSLs (models).

All these meta-models – “models” being a misuse of language – have a goal: they allow to represent something. To build ergonomical software, we must understand during its building how the user thinks when using it. The more the user interface respects the user's mental representation, the better the user interface will be accepted by this user. But the building scheme of users will often not match directly the building scheme used to create data models (or meta-models in our case). So to match them, we must create adapters between both these building schemes. If such adapters are not enough – as it is often the case – we must either adapt the data models or adapt the user interfaces. In our case, we can only modify user interfaces. That's why it is important to understand how to create instances of meta-models in order to correctly define adapters and, only if necessary, modify adequately user interfaces.

Concerning diagram editors or tree editors, both these kind of editors already include general-purpose adapters. We can modify a part of these adapters as well as user interfaces. But if user interface goes too far from the user's mental representation of the related task, then it won't be acceptable to choose one of these kind of editors.

For example, if a meta-model representing a file system is pretty well represented by a tree editor, a meta-model representing a math formula is not to be represented a tree editor: nobody naturally writes a math formula in the form of tree!

Moreover, meta-models are developed to provide functionalities to the OPSWP. These functionalities are resumed below.

PEUT ETRE METTRE QUE LES META MODEL COMPLETE LE MODELE DE TÂCHES

#### 4.4.4. Modular constraints

As mentionned in 4.4., the OPSWP is modular, since it is composed by several tools. Amongst these tools, some of them are developed by partners of the OP project, and as such, their development and evolution can be adapted to our needs. For the other, third party tools, such flexibility is not possible: we can (partially) modify their user interface in the software platform, but we have no way to change their inner architecture and behaviour.

## 5. Style guide recommendation

*Ce chapitre évoque des maquettes dont certaines seront encore soumises à modifications liées à des contraintes non prévisibles.*

### 5.1. Lookup table creation (importation) and edition

As stated in the first part of this document, the Power Consumption Models Development (PCMD) features of the OP SoftWare Platform (OPSWP) use as their base, raw material the measurements provided by the OP HardWare Platform (OPHWP). The latter provides its results in the form of OPTR archives.

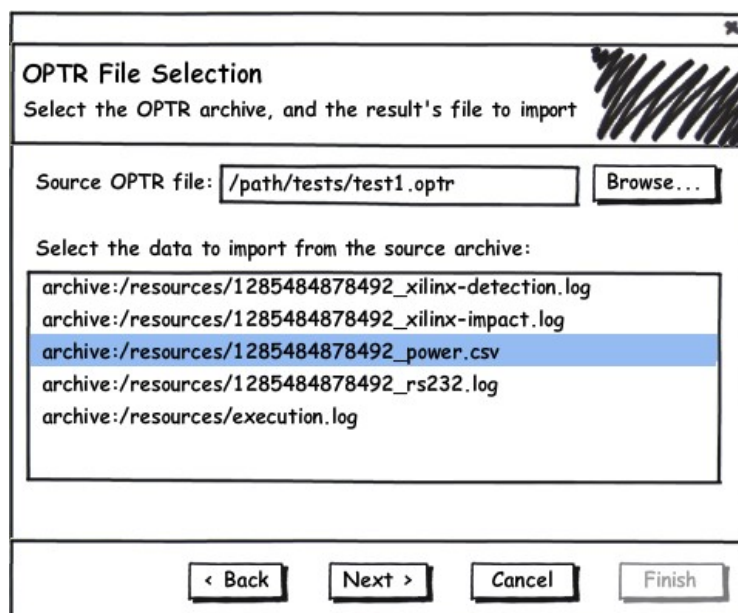
To be usable, these power consumption measurements have to be imported into a directly usable data structure: the LookUp Table (LUT). Our software platform offers an integrated, convenient mechanism to import the OPTR-embedded data into LUTs.

Once imported, the LUTs can be explored and edited via a dedicated (Eclipse-based) editor. That editor shall also soon offer advanced statistical analysis features, allowing thorough studies of the LUT's data, and determination of any underlying mathematical function(s) – if such functions exist. These LUTs as well as their derived mathematical laws will then be available to build new consumption models.

#### 5.1.1. Importing OPHWP measurements into a new LUT

The OPTR importation feature is provided to the user via a dedicated import wizard. Since this importation function needs many parameters, and especially a quite complex configuration matching source data with destination columns in the new LUT, this wizard is divided into several steps, represented by as many wizard pages.

The *first wizard page* is dedicated to the **selection of the source file** containing the data that will be imported. Its interface is represented by the following diagram:

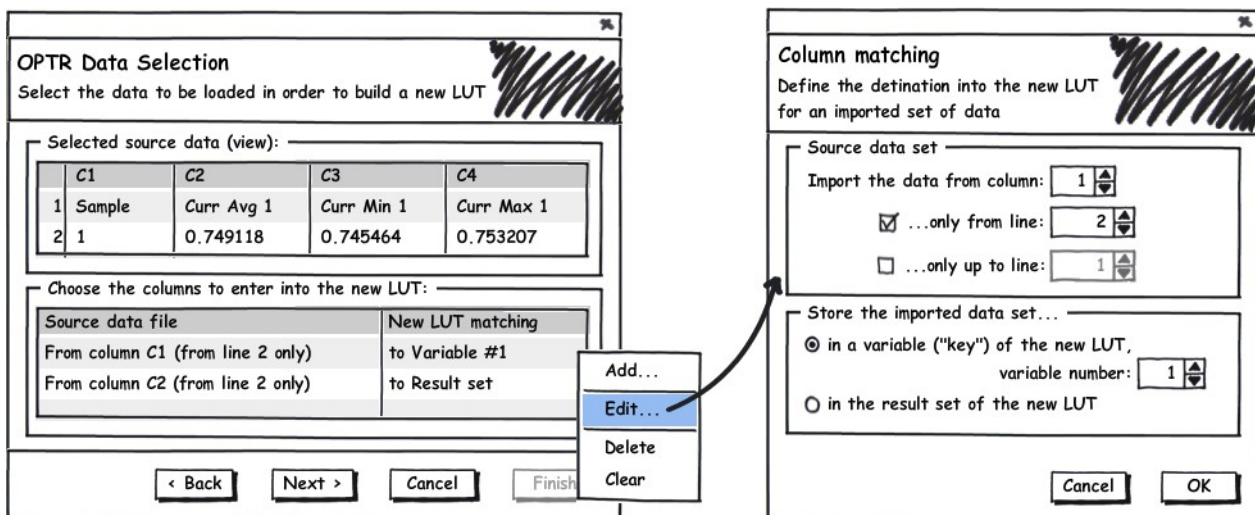


This page is vertically divided into two zones, to be filled sequentially during the import procedure:

- The first text field (on the top) is where the name of the source OPTR archive is to be typed in, the neighbouring “Browse” button allowing to choose graphically the OPTR file (via the system's standard file dialog box).
- Once the source OPTR is designated, the user still has to designate the raw data file. Since an OPTR file is actually a ZIP archive containing all of the output produced by a test executed on the OPHWP, the bottom field lists the files embedded into the archive, so that the user can choose the file where the power measurements data are gathered; it is normally a CSV file.

When a data file is selected in the list, the “Next” button activates, allowing the user to proceed to the second wizard page.

This *second wizard page* is dedicated to the **selection of the data** to be imported into the LUT. Its interface is represented by the following diagram:



Because the CSV files generated by the OPHWP can be huge, and often comprise a lot of data that may not all be worth importing into a single LUT, that page allows the user to perform fine-grained selection amongst the data, allowing only some columns and lines to be imported from the source CSV file into the new LUT.

To that aim, the page (on the left in the previous figure) is vertically divided into two tables:

- The top table presents the raw data contained in the source file selected in the previous page; this table is read-only (no modification can be made to its contents): its role is to show the user how the source data is organized, so that (s)he can adequately choose what data to import into the LUT, and how<sup>1</sup>;

<sup>1</sup> Note: it also gives the user the opportunity to realize that (s)he selected the wrong source file: (s)he can then click on the “Back” button to return to the first wizard page and correct his/her selection.

- The bottom table is the “active part” of the page, where the user will express its choices about the data importation; the importation process (as the LUTs themselves) are “column-centric”, that is: all the data they contain are organized into **homogenous sets**, each of these datasets correspond to one **column**; thus, defining the data importation actually means choosing which column of the source CSV file will be imported into which LUT column. We choosed to call each of these import relations a “**column matching**”. Thus, each line in the bottom table corresponds to a column matching that will constitute the import process. The sum of these column matchings represents the integrality of the current OPTR-to-LUT import process.

Lines of the bottom table can be manipulated (added, deleted...) thanks to the table's popup menu (right-click on the table). Editing a line of the bottom table – by calling the “Edit...” command of the popup menu, or by double-clicking on the line – calls forth a dialog box specifically made for editing column matchings. We will study this dialog box below.

The wizard pages ensure that there are at least enough valid columns matchings to populate a variable and the result set of the new LUT. Once these requirements are met, the “Next” button activates, and the user can proceed to the third and last wizard page.

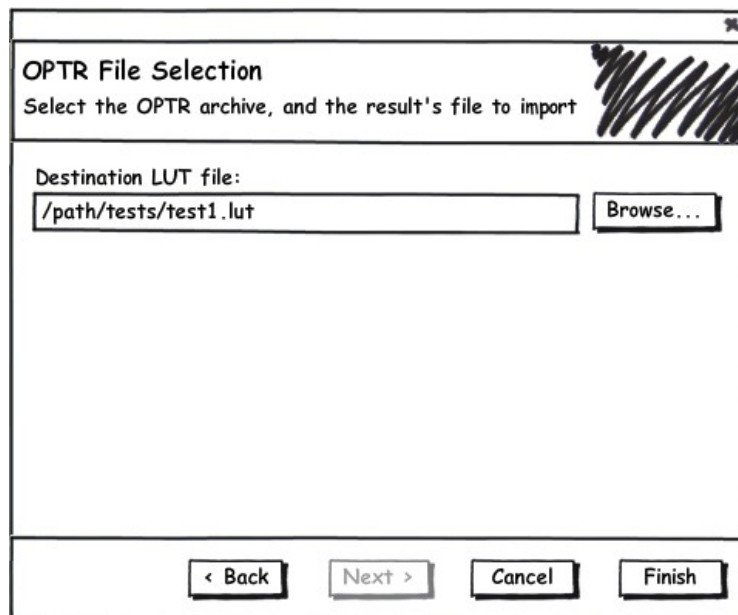
The column-matching detail dialog is shown on the right of the previous diagram. Its role is to allow the user to put into relation a column of the source CSV file, with a column of the new LUT that the import process will create. Thus, it is logically divided into two zones: the top zone is dedicated to the definition of the source data set, while the bottom zone indicates where in the destination LUT the imported dataset will be stored.

- The top, source-dedicated zone serves mainly to decide what column of the source file will constitute the source dataset: it is the purpose of the first (top) “number spinner” to select that source column's number. Moreover, that zone also offers the opportunity to restrict the imported dataset to a specific range of lines from the chosen column – since the latter may contain unwanted elements that are not to be imported. Thus, there are two checkboxes (coupled with two “number spinners”) that allow to define respectively a first and a last line, restricting to the adequate range the importation of the data from the source column.
- The bottom, destination-dedicated zone serves to indicate what place the imported dataset will occupy into the destination LUT. A given dataset can be used to fill either a “key” (variable) of the LUT, or its result set; the two radio buttons of the bottom zone allow the user to choose between these two kinds of role. If the destination is a variable, the “number spinner” coupled with the first radio button will indicate the index number of that variable into the new LUT – since our LUTs can hold many variables<sup>2</sup>.

---

2 As a reminder, an OPSWP LUT has one or more variable(s), but always has one and only one result set; thus, the simplest form of LUTs corresponds to two-columns tables – a variable plus a result set. Currently, we allow LUTs to have up to 9 variables (i.e.: LUTs can have up to 10 columns). It is important to note that the order of the columns matter, that is: the compound function that the LUT implements is **not** commutative! **Multivaried LUTs are actually tree structures.**

The third wizard page is dedicated to the selection of the file where the destination LUT's file will be stored. Its interface is represented by the following diagram:



OPTR File Selection  
Select the OPTR archive, and the result's file to import

Destination LUT file:  
/path/tests/test1.lut Browse...

< Back Next > Cancel Finish

This is the last and simplest stage of the importation procedure: the user just has to set the file where the resulting new LUT will be saved.

There is thus only one zone on the top of the page. It comprises a text field where the path and name of the destination LUT file is to be typed in, the neighbouring “Browse” button allowing to provide these information graphically (via the system's standard file dialog box).

### 5.1.2. Using the LUT editor

Since LUTs can be quite huge and complex – especially if they comprise more than one variable – the dedicated LUT editor we built is divided into several pages:

- The first page is dedicated to the management of the LUT's “metadata”, that is: the definition of its columns – variable(s) and result set;
- The second page is dedicated to the browsing and the edition of the LUT's data: to that aim, we specifically chose to take advantage from relatively little-used technique that can be applied to tree structures;
- The third page is dedicated to statistical analysis of the LUTs data: it is still a work-in-progress. We eventually want to integrate into this page sophisticated tools that will allow the users to find easily, efficiently and reliably any mathematical function/law that could underlies any LUT's data.





**LUT Variable**  
Define the characteristics of the given LUT variable.

Variable #1

General

Name:

Unit:

Unknown Values Determination Policies

- Accept and interpolate unknown values in the definition domain
- Accept and compute unknown values under definition domain
- Accept and compute unknown values over definition domain

The user interface of this dialog is easy to understand: its top part is dedicated to the edition of the main characteristics of the variable – its name and its unit – whereas the bottom zone allows to control whether values non-explicitly defined in the variable's domain of definition will be accepted and computed on-the-fly or not.

The second page's U.I. can be summarized by the following diagram:

Frequency (MHz)	Voltage (V)	Power consumption (W)
100	1.0	0.10
200	1.5	0.15
300	2.0	0.20

Variables Values



Since the values comprised in a LUT are organized into a **tree structure**, we decided to use a relatively little-known, but clear, and efficient technique of representation of tree structures, in order to let the user browse and edit the LUTs datas.

This technique is known as the **Miller Columns**. These columns have – from the ergonomic point of view – two decisive advantages:

- they allow multiple levels of the hierarchy to be open at once, and
- they provide a clear visual representation of the currently browsed location (“path”) into the tree structure.

Even if they are not used as widely as one could expect into mainstream software, we can however note that the Miller Columns appeared early in development environments (especially the Smalltalk browser); their first use into a general-purpose software was in the file viewer included in the infamous and very influential NeXTstep operating system. Today, such a presentation is still available in the Finder of MacOS X (NeXTstep's heir and “son-in-law”), as well as other Apple software (like iTunes for example)...

The Miller Columns thus occupy the totality of the LUT editor's second page. There, the LUTs values can be browsed and directly edited in-place. Further manipulation of the data (adding, deleting) is provided by the popup menus attached to each of these columns.

We will describe the third page's U.I. in a future version of this document, once the conception and development of that editor page has reached a sufficient maturity level.

## ***5.2. Quantity kind and unit creation***

The purpose of the QUDV DSL is to define units and quantities. For example, it is used to define a 'length' quantity kind whose associated unit is 'metre'. The diagram in figure 10 hereafter shows how the SI seven base units defined as a QUDV model. A more complex instance of a QUDV meta-model can be found in deliverable D4.1 appendice. The quantities and units so defined are to be used in mathematical laws and architecture models definition.

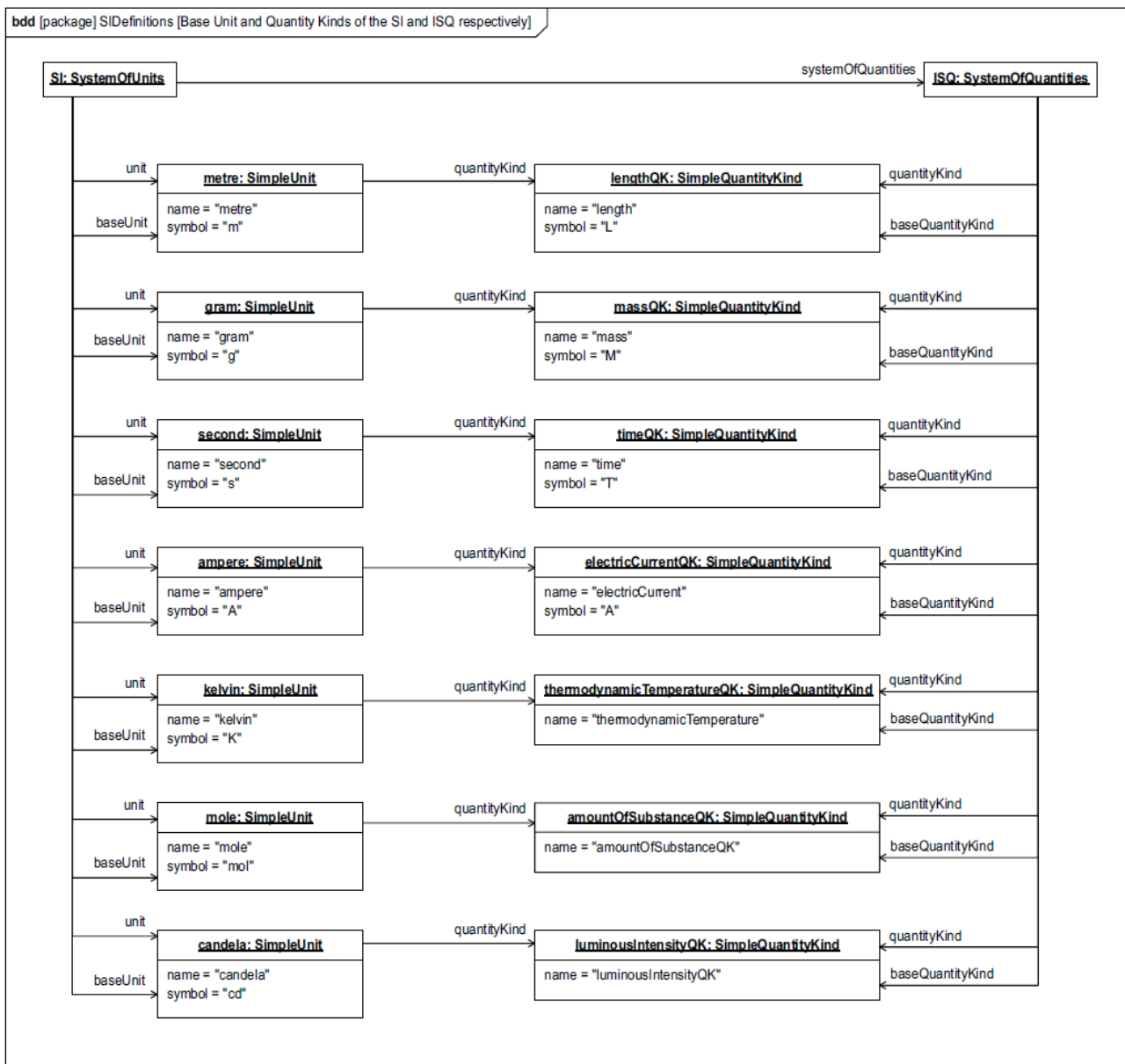
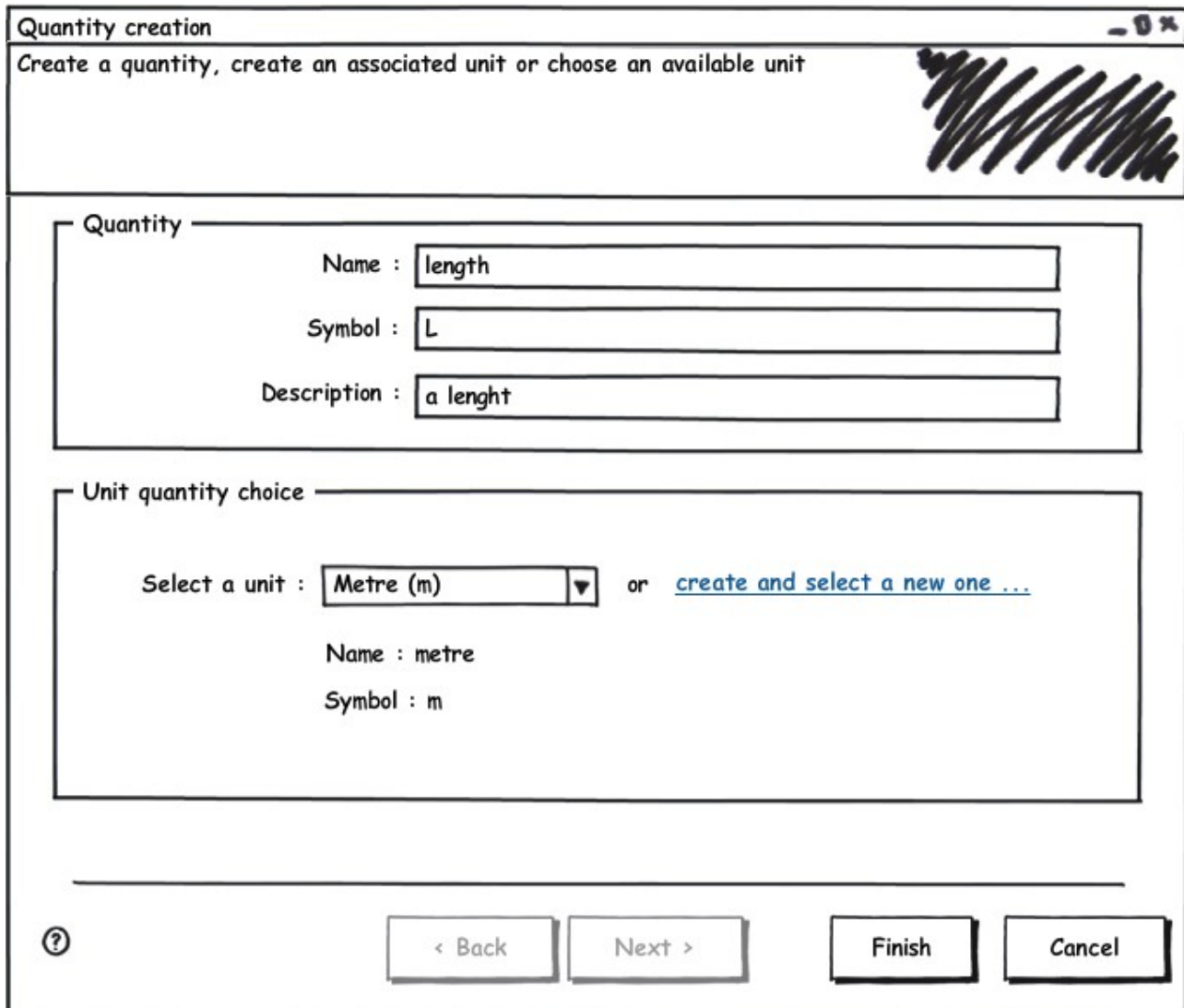


Figure 10: Base unit and QuantityKinds of the SI

The pictures shown in the next figures are good examples of what kind of representation we want to display to the user, instead of bothering him with a cluttered diagram representation of model objects – in the latter case, users would have the unpleasant obligation to create unclear diagrams made of linked boxes. Still, we will have to build a rather more complex UI to take into account multiples- and fractions-related prefixes (kilo, milli, etc.), as well as derived units as defined in deliverable D4.1. However, using tree editors to define such quantities and units would be far more complex and mind-twisting, and as such would result in UI highly likely to be rejected by the users. That's why, we chose to use the common UI elements described in the earlier chapters of the present document to create and edit QUDV-based models.



The mockup shows a window titled "Quantity creation" with a subtitle "Create a quantity, create an associated unit or choose an available unit". The window is divided into two main sections. The first section, "Quantity", contains three input fields: "Name" with the value "length", "Symbol" with the value "L", and "Description" with the value "a lenght". The second section, "Unit quantity choice", contains a dropdown menu labeled "Select a unit" with "Metre (m)" selected, followed by the text "or [create and select a new one ...](#)". Below this, there are two more input fields: "Name" with the value "metre" and "Symbol" with the value "m". At the bottom of the window, there is a help icon (a question mark in a circle) and four buttons: "< Back", "Next >", "Finish", and "Cancel".

Figure 11: Mockup of quantity kind creation user interface.

In the mockup shown in figure 11 above, the user will have to give some details about the quantity kind he wants to create. Then he will be able to choose whether to base his/her new quantity kind on a predefined unit, or else to create a new specific unit. In the latter case, the unit creation wizard will be automatically started for him (her).

In the unit creation wizard (see Figure 12 below), the user can choose between creating a base unit – note, however, that all of the seven SI base units will be already predefined and delivered along the OPSWP – or creating a derived (i.e. compound) unit. In the next step of this wizard (see figure 13 below), the user will define the general information about the unit and above all define the unit-compounding formula that will actually define the new unit. Once this mathematical expression is given, the unit creation process is complete.

Of course, it will be possible to directly create units – there is no need to create a related quantity before creating an unit.

Unit creation

Kind of unit  
Choose the kind of unit you want to create

What kind of unit do you want to create ?

base unit  
 derived unit composed by units

Available units :

Name	Symb	Quantity	Expression (other)	Expression (SI base)
metre	m	length	N.D.(base unit)	N.D.(base unit)
gram	g	mass	N.D.(base unit)	N.D.(base unit)
second	s	time	N.D.(base unit)	N.D.(base unit)
amper	A	electric current	N.D.(base unit)	N.D.(base unit)
kelvin	K	thermodynamic	N.D.(base unit)	N.D.(base unit)

Figure 12: Choice of kind of unit in the unit creation wizard.

Unit creation

Fill unit information and expression

Unit creation

Name :

Symbol :

Description :

Expression

Watt =

Select Units

Name	Symbol	exponent
metre	m	2
kilogramme	kg	2
second	s	
ampere	A	
kelvin	K	
mole	mol	
candela	cd	

display all types of units

Figure 13: Unit information and derived unit expression in the unit creation wizard.

### 5.2.1. Ergonomic evaluation

To perform the ergonomic evaluation of our UIs, we build use case scenarios. The purpose of these scenarios is to give a concrete task to a test user, then see which difficulties (s)he encounters when realizing that task with our mockups.

In the case of QUDV UI, the task was: create a quantity 'Power' ('P') based on the unit 'Watt' (W) whose composition formula is:  $\text{kg} \cdot \text{m}^2 \cdot \text{s}^{-3}$ .

We made this test with three different users, which is few but enough to detect heavy ergonomic problems.

During these tests, we noticed some difficulties which impelled us to modify mockup at several places: we moved the creation of a base unit in a preference page, thus the first page of the unit creation wizard became obsolete and was deleted; in the second page of the same wizard, we deleted the fraction bar and allowed the user to enter all terms of expression in a single step.

After these modifications, the UI for the unit creation wizard is now as shown on figure 14 hereafter:

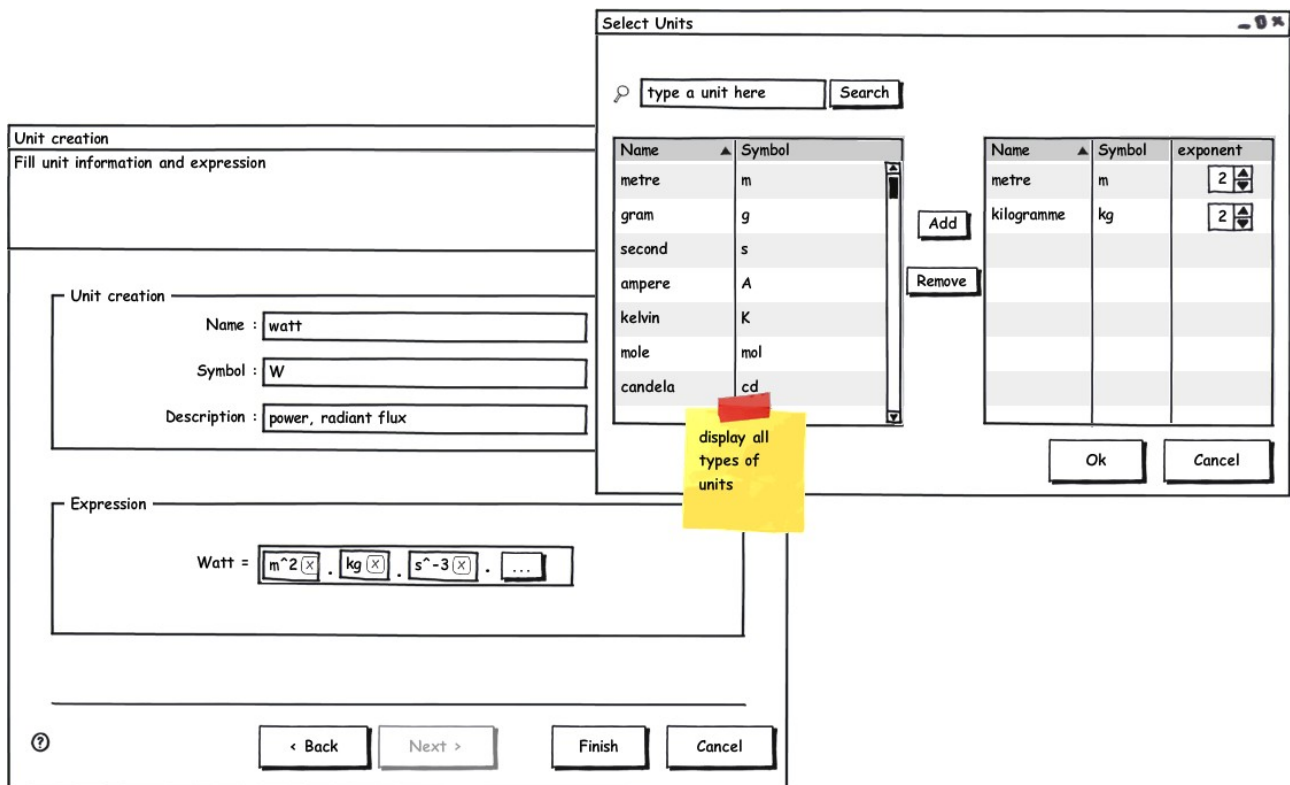


Figure 14: Derived unit information and expression in the unit creation wizard.

The quantity creation wizard stays unmodified from what is shown in figure 12.

### 5.3. Math law creation

Our models use mathematical laws to estimate their final values, e.g.: power consumption or activation time. Deliverable D4.1 details the different kinds of laws that can be used.

$$T_a = \left\{ \begin{array}{l} 31.42 * S_p^{0.1348}, F_p = 100.0 \wedge T_p = UDP \wedge S_p < 1500 \\ 0.6791 * S_p^{0.8705}, F_p = 100.0 \wedge T_p = UDP \wedge S_p \geq 1500 \\ 54.909 * S_p^{0.3189}, F_p = 100.0 \wedge T_p = TCP \wedge S_p < 1500 \\ \dots \end{array} \right.$$

Figure 15: The first 3 equations and their domain of validity for the XUPV2P MAC Ethernet controller time model.

To store it, we use the MathML language, which is a W3C norm. In MathML, a formula can be written by two ways: content formulation or representation formulation. The difference between these two formulations is illustrated by the table hereafter:

Content formulation	Representation formulation
One divided by two	$\frac{1}{2}$ or $1 \div 2$ or $\frac{1}{2}$
Two times a	$2.a$ or $2*a$ or $2 \times a$

As one can see, several representations are possible for a single content. That's why the MathML DSL is divided in two parts. For Open-PEOPLE, we naturally prefer the content representation since it is invariable for every single mathematical content – contrary to representation formulation. On the MathML website can be found a vast list of MathML building software; unfortunately, most of them use the representation formulation, since these tools are editors that don't actually need to use and compute the expressions they manipulate. That's why we didn't choose to use one of them.

On the other hand, we will try to “take a leaf out” of the GUI of these softwares to create our own MathML editor.

There are two ways to build mathematical user interfaces:

- one can use a pseudo programming language to describe the formula (see for example figure 16 – the OpenOffice/LibreOffice equation editor), or
- use a graphical “box pattern” system (e.g.: the Formulator MathML editor UI, shown in figure 17 below).

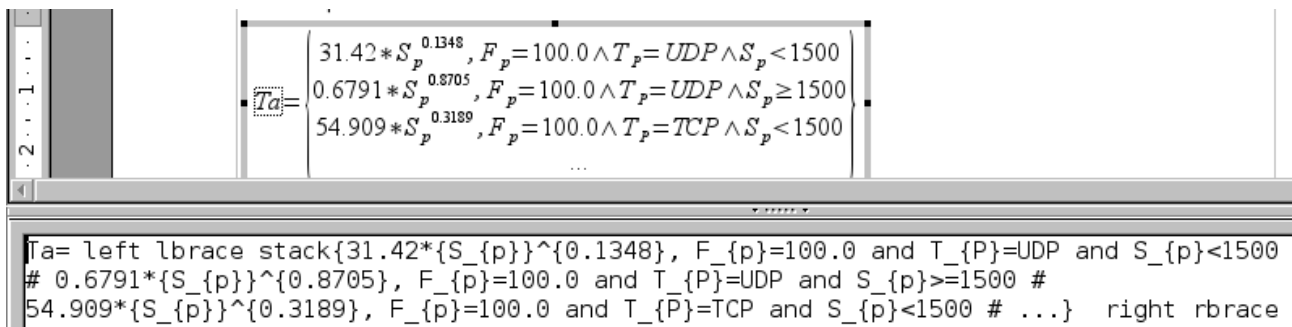


Figure 16: Software with pseudo programming language (Open Office).

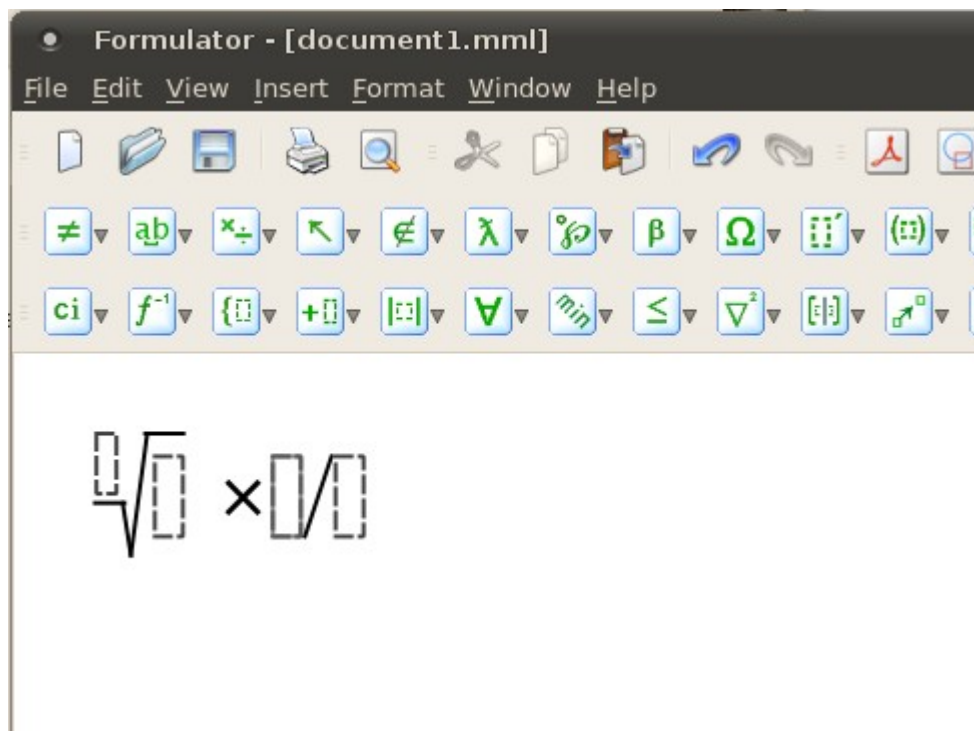


Figure 17: Software with box patterns edition (Formulator).

The pseudo programming language of Figure 16 mainly uses keyboard entry to define a mathematical formula, whereas “box patterns” UIs equally uses both mouse and keyboard. So, in the first case, it is possible to write a math formula with the sole keyboard – which is better for long formulas – whereas “box patterns” UIs force the user to constantly switch between the keyboard and mouse. Research literature on UIs confirm that direct manipulation with such “mixed” procedures need to be extra precise, and are finally less productive than keyboard-only edition. Moreover, the keyboard edition is more adapted for repetitive entries that “expert” users are likely to use.

That's why we choose for the OPSWP to implement a pseudo-programming editor like the one shown in Figure 16 for mathematical objects, as it is clearly the best option in our opinion.



## 5.4. PCMD creation

The purpose of this task is the creation of an energy consumption model composed by several mathematical laws. Use cases and activity diagrams describe this task more precisely (see deliverable D2.1). From these data, several static and dynamic mockups were realized and several ergonomics interviews allowed to validate the mockups described hereafter.

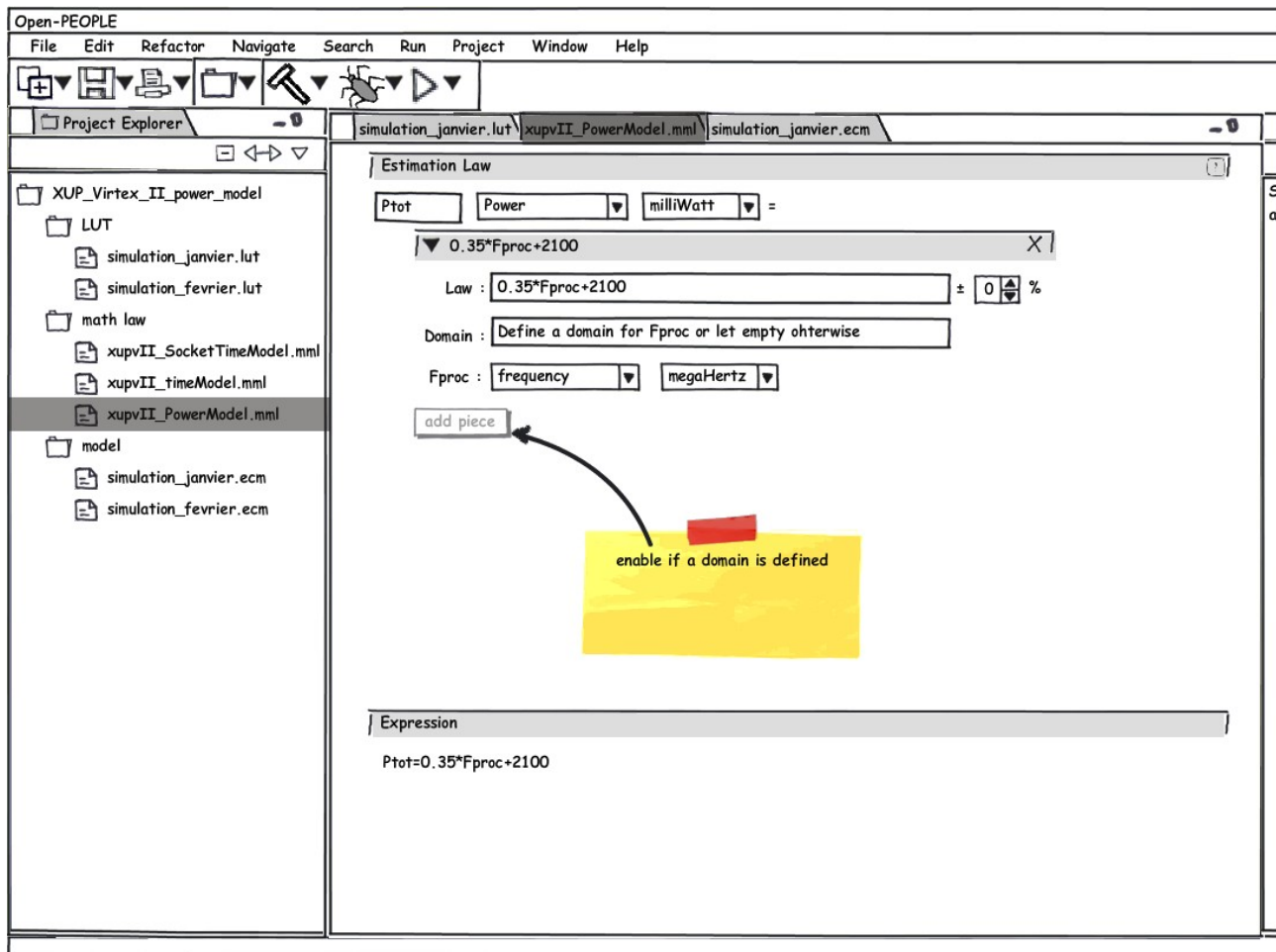


Figure 18: Mockup of the estimation law editor.

The mockup shown in Figure 18 above is divided into two parts.

- On the left, we can see the project explorer view, showing a newly created Open-PEOPLE project. An Open-PEOPLE project is divided in three sub folders: **LUTs** (Lookup Tables); **math law** containing all the available (entered) mathematical laws; and **models** containing the energy consumption models that regroup an arbitrary set of mathematical laws and LUTs selected from the two preceding subfolders, as shown in Figure 19.
- On the right stands a mathematical (estimation or composition) law editor, containing among others an area displaying the complete, well-formed current math. expression.



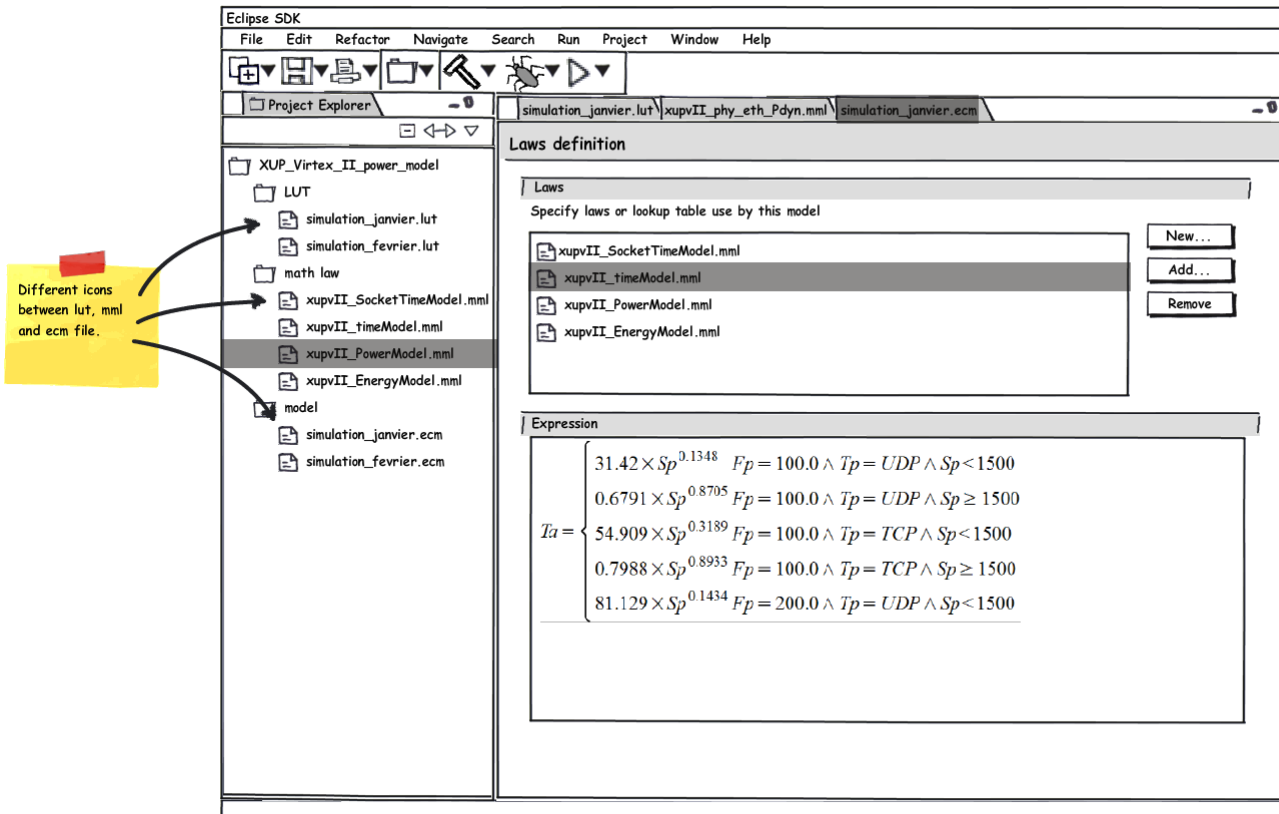


Figure 19: Mockup of the energy consumption model editor

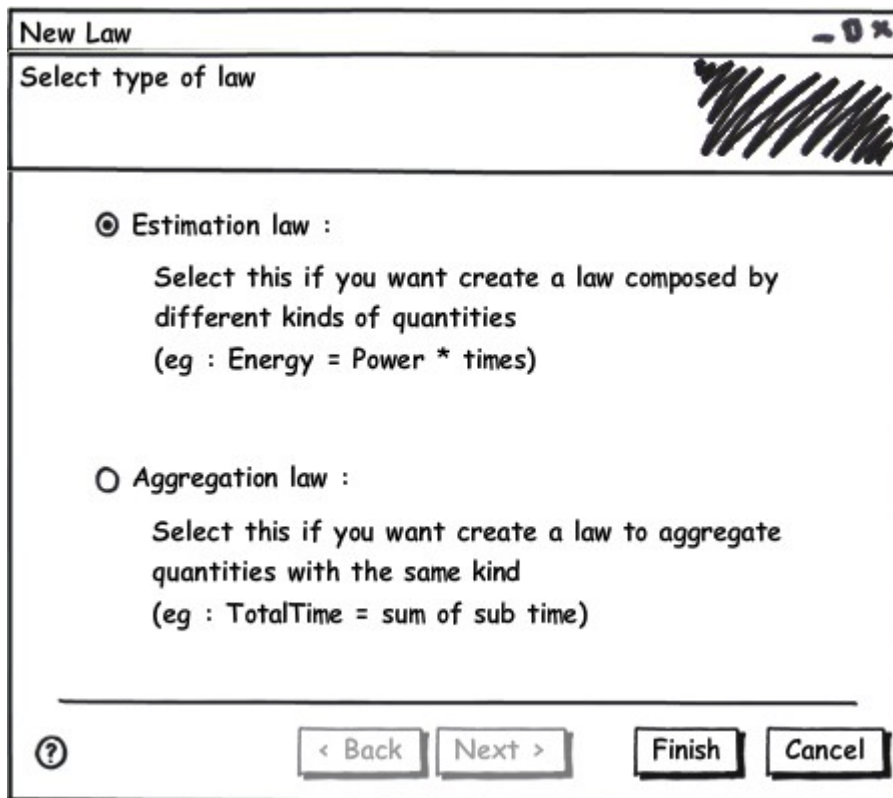
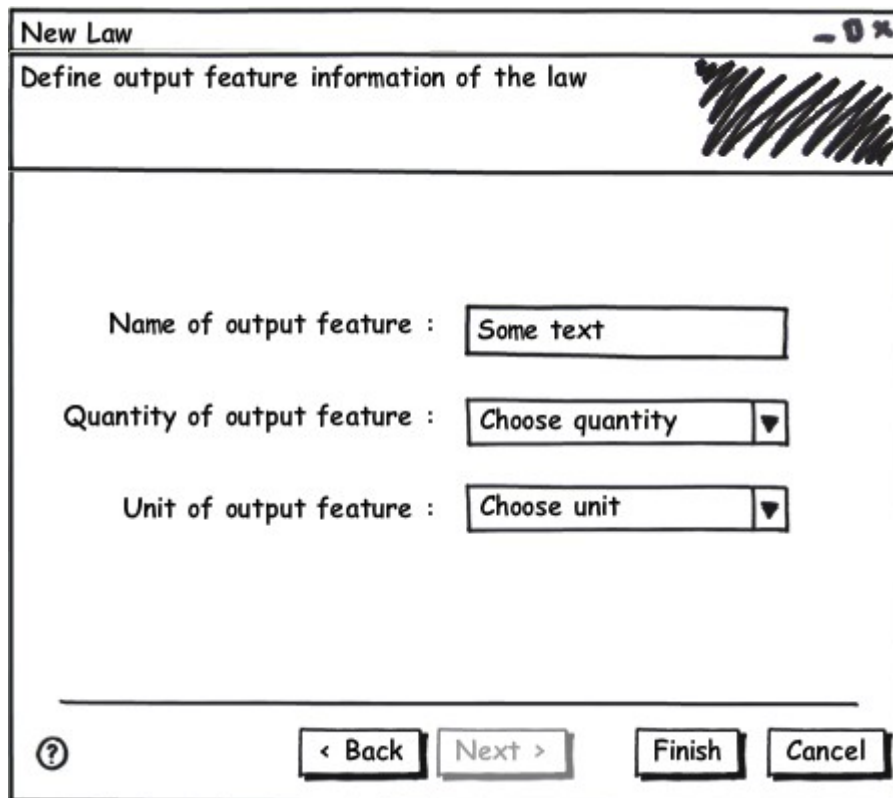



Figure 20: Mockup of the "New law" wizard's first page.



New Law - 0 x

Define output feature information of the law 

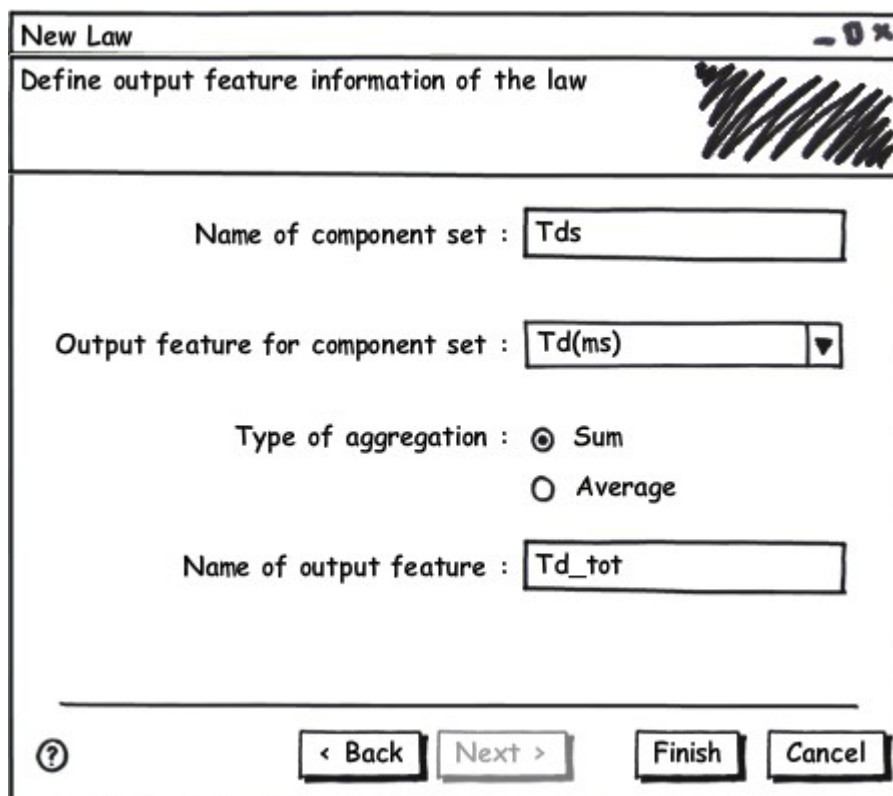
Name of output feature :

Quantity of output feature :  ▼


Unit of output feature :  ▼

---

Figure 21: Mockup of the “New law” wizard's second page, when creating an estimation law.



New Law - 0 x

Define output feature information of the law 

Name of component set :

Output feature for component set :  ▼

Type of aggregation :  Sum  
 Average

Name of output feature :

---

Figure 22: Mockup of the “New law” wizard's second page, when creating an aggregation law.

The mathematical law creation wizard can be invoked through the project view's popup menu. The first page of this wizard (shown in Figure 20) allows the user to choose between an estimation law and an composition law. As the next and last step, the wizard second page asks the user to provide the general characteristics of the new law; this second page differs according to the nature of this new law. Figure 21 shows the page displayed for an estimation law, Figure 22 for an aggregation law.

Figure 18 – previously seen – shows the editor dedicated to estimation laws.

Figure 23 displays the (somewhat simpler) aggregation law editor.

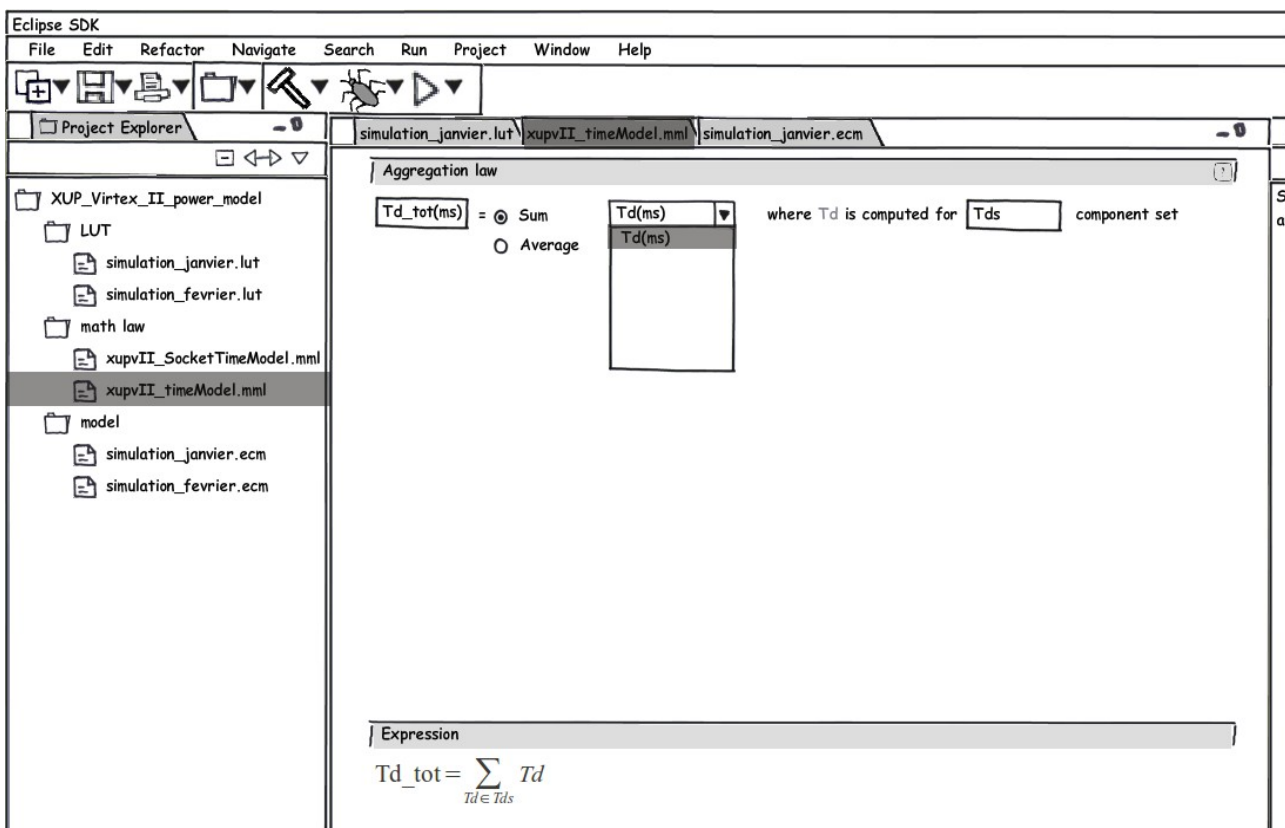


Figure 23: Mockup of the composition law editor.

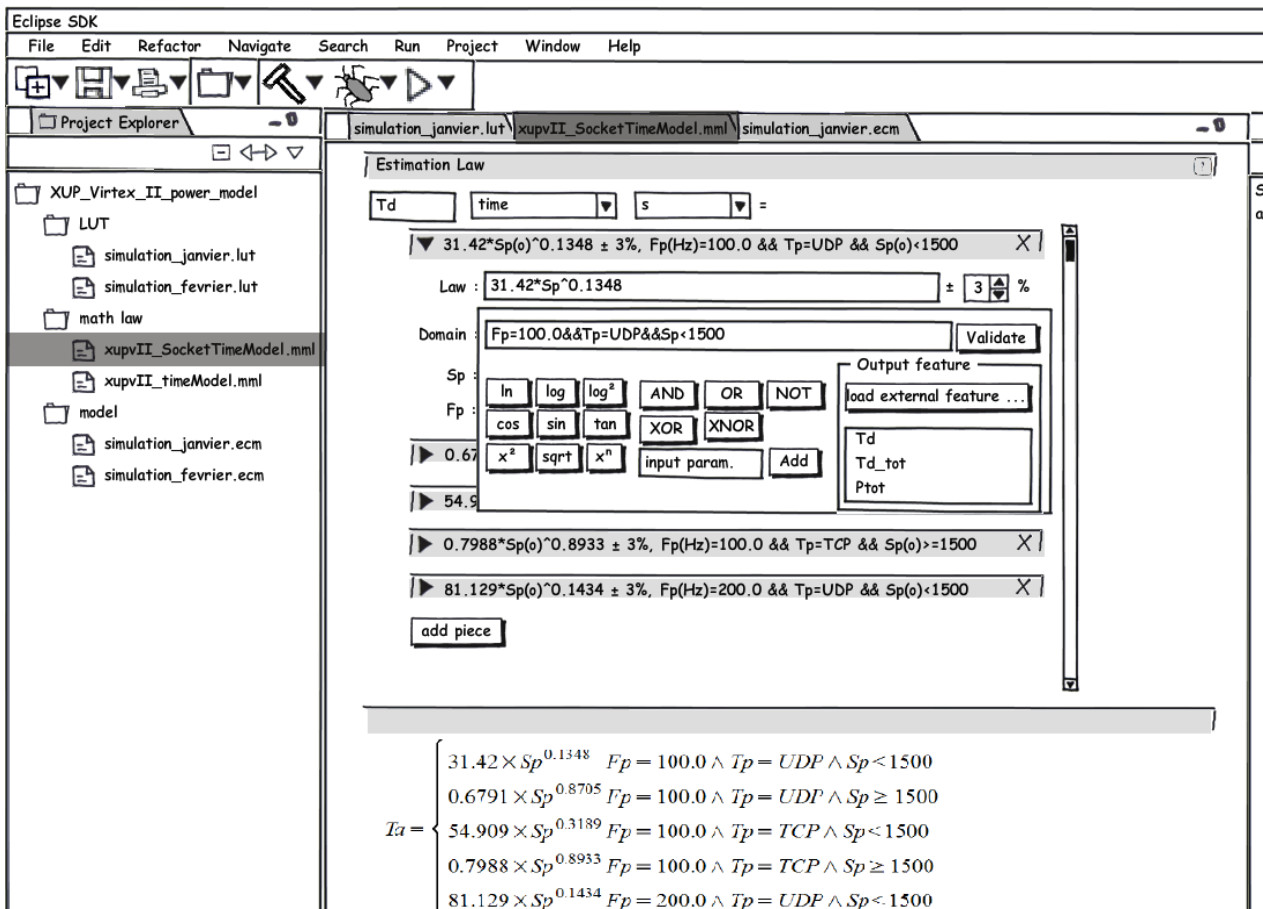


Figure 24: Mockup of math field popup.

Finally, Figure 24 displays how the specialized mathematical fields in our UI offer a popup panel that includes several buttons, thus allowing easy addition of operators (e.g.: exponent or square root) to the currently edited mathematical expression.

## 5.5. PCAO: weaving model

The weaving model's purpose is to link each hardware or software component of an architecture with the appropriate energy consumption model. As explained in deliverable D4.1, two tools already exist to create a weaving model : *Atlas Model Weaver* and *Epsilon*. Both tools provide a three-part editor (see figures 25 and 26 hereafter) to create weaving models.

Both of these tools have a UI centered on weaving models: the related business task “choose a model for each component” is only implied.

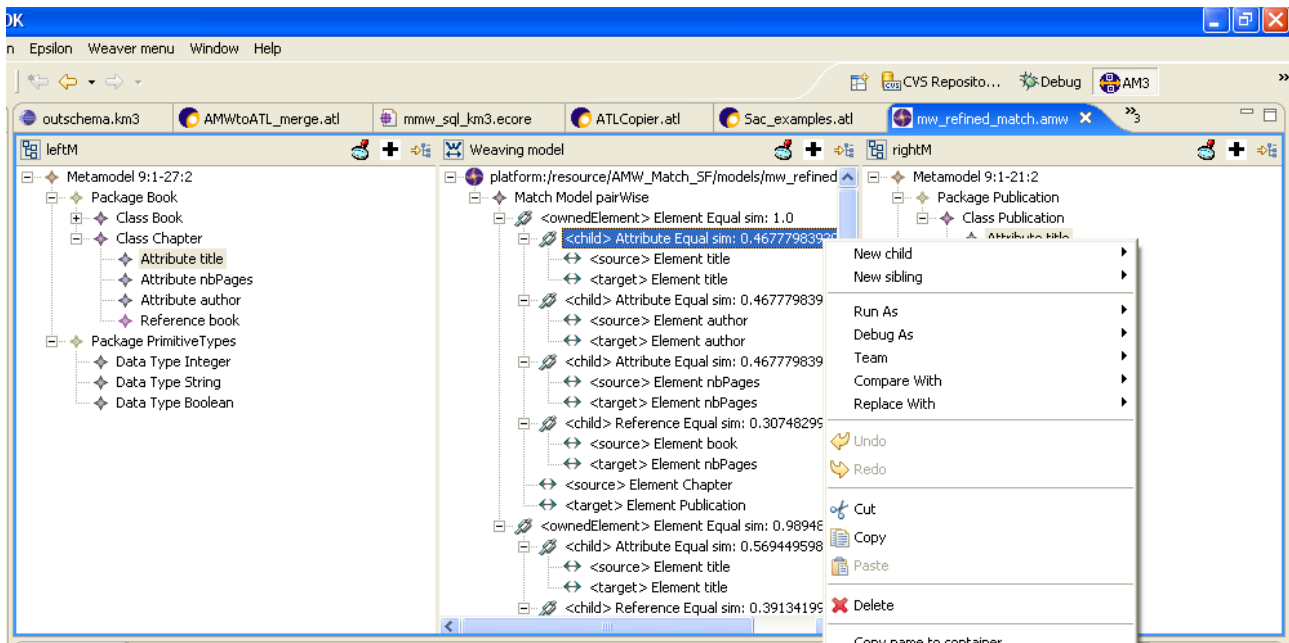


Figure 25: The three parts editor of AWM.

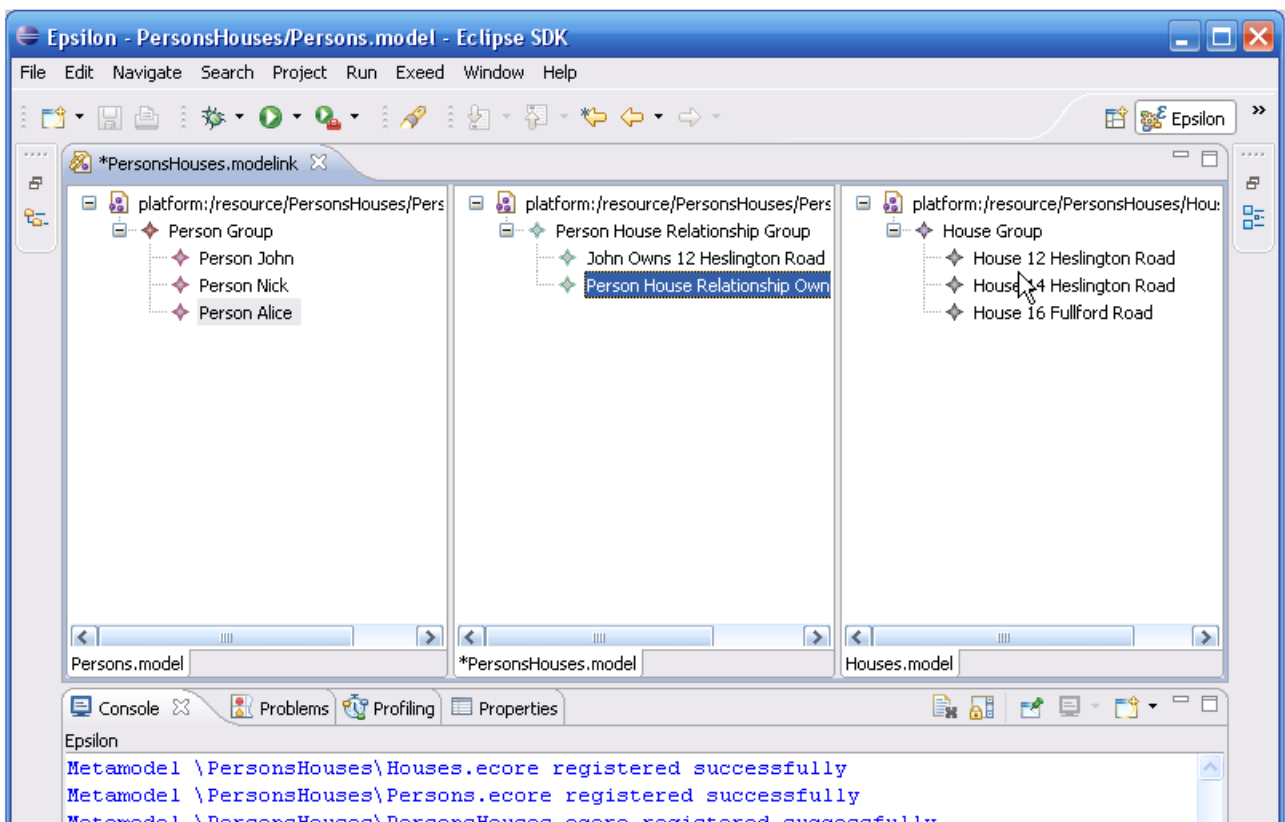


Figure 26: The three parts editor of Epsilon.

Choosing this kind of UI would be incoherent with the previously shown UI: the user would be disturbed by the different, lower-level paradigm. That's why we concluded that it was much preferable to develop an higher-level UI, centered on the “choose a model for each component” business task: such an UI will show more coherence with the rest of our software platform, and shall prove to be more user-friendly.

## Special acknowledgment

We thank following persons who was interviewed to realize this document : Dominique Blouin, Mickael Lanoe, Eric Senn, Damien Bodenes, Cecile Belleudy, Christian Samoyeau and Victor Tissier.