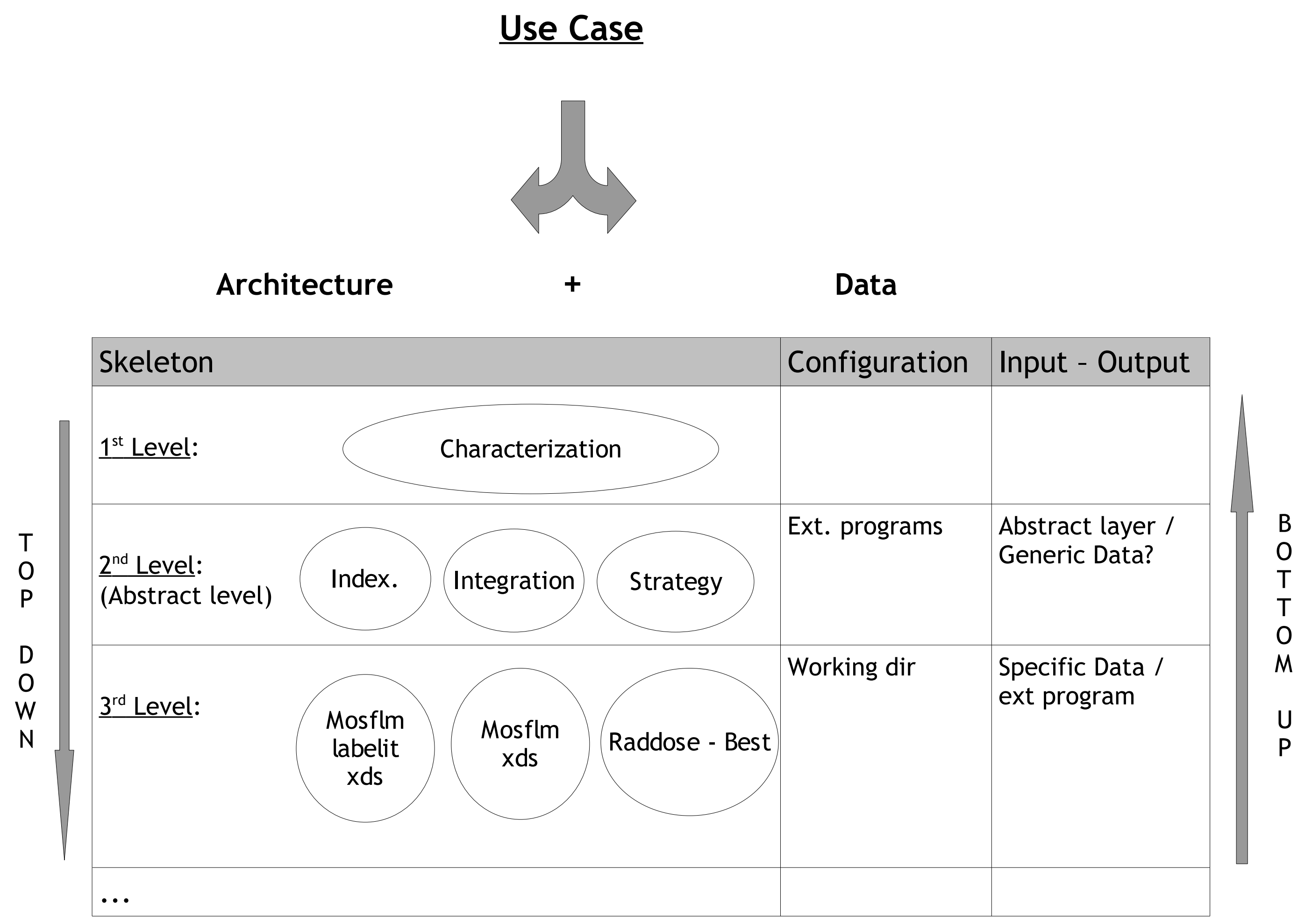
Prototype Implementation Technical Approach

1/ From the Use Cases to the prototype implementation...

The objective of this document is to address how, from a given Use Case, to reach a working application. The general idea is to adopt both top-down and bottom-up strategies depending on what has to be defined.
In a top-down approach, an overview of the system is first formulated, first-levels sub-systems are specified but not detailed. Each subsystem is then refined in greater details, until the entire specification is reduced to base elements. This approach is typically applicable to analyze the main use cases in order to bring an application skeleton to light (see “EDNA skeleton” section for more details)
Once a certain level reached, a bottom-up analysis can then be carried out. This approach allows the base elements to be specified in great details (including configuration parameters and Data Model). This bottom-up approach is essential to guarantee the modularity of the system (see “Checking plugins” section) and should also guarantee an homogeneous integration of a base element in the whole application

The figure below illustrates the concepts mentioned above:



2/ EDNA Skeleton

2.1/ Objectives

The objective of the skeleton is to launch sequentially the indexing, the integration and the strategy steps within an application described in Figure1. The term skeleton is used to mention that the steps will either be empty or will launch the appropriate executable but no scientific results will be provided, that is to say that no Data Model will be needed in this context.

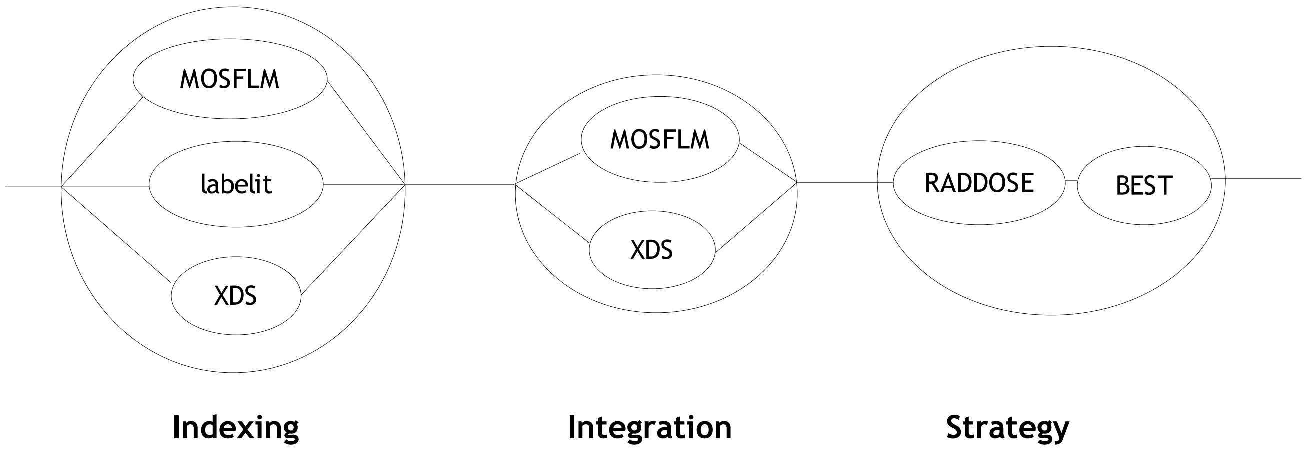


Figure 1: Indexing/Integration/Strategy steps sequence.

The application can be configured to execute one or several parallel external programs for a particular step (i.e: the indexing step could either be performed using MOSFLM or labelit or XDS or using the 3 programs in parallel) (see section 2.2.1 for more technical details).

2.2/ Technical approach

2.2.1/ Configuration

2.2.1.1. List of parameters

The objectives of the configuration are:

- to define the list of plugins which are necessary to perform a particular step (i.e indexing: MOSFLM or labelit or XDS, or the 3)
- to define the technical parameters for a particular plugin to work properly (i.e the working directory)

- to overwrite default hard-coded parameter values/behaviours (MOSFLM as a default external program to be launched could be replaced by the plugin defined in the configuration file, etc...)

2.2.1.2. Format

2 proposals are currently being studied for the configuration file: an ini-like and an xml format.

- ini-like configuration file:

```
[indexing]
	software = mosflm, labelit, xds
	var2 = value2
[integration]
	software = mosflm
	var1 = value1
```

<u>Advantages / drawbacks</u>:

The organization is not really object-oriented-like. A special parser is needed
Easy to read and to configure manually

- xml configuration file:

The xml is a more appropriate format to organize the configuration on different levels of depth. several approaches are proposed here:

<u>1st approach</u>: pragmatical

The first approach proposes a pragmatical solution to configure the skeleton described in section 2.1. as a sequence of steps that encapsulates plugins

```
<step name = “indexing”>
	<plugin name = “mosflm” />
	<plugin name = “labelit” />
	<plugin name = “xds”>
		<workdir=“/path/to/workdir” />
	</plugin>
</step>
<step name = “integration”>
	...
</step>
```

<u>2nd approach</u>: “all is plugin”

The 2nd approach is more generic. The principle is “All is plugin”. A plugin configuration contains the configuration of its childs. It follows closely the way the application is structured (one level of depth per level of plugins). Even if a plugin has no direct configuration (no param element), it can have an indirect configuration due to its childs configuration. This approach proposes that if a plugin is present in the configuration file, it will be executed by its parent.

```
<plugin>
        <name> Indexing </name>
        <param>
                <name> time </name>
                <value> 17:05 </value>
        </param>
        <plugin>
                <name> labelit </name>
                <param>
                        <name> x </name>
                        <value> y </value>
                </param>
                <param>
                        <name> w </name>
                        <value> z </value>
                </param>
                <plugin>
                </plugin>
        </plugin>
</plugin>
```

3rd approach: “all is plugin” / enabling plugins parameters

The information is given in the param element (direct configuration) : a plugin is enabled if the ext program param is “on”. We can note that even if a plugin is not activated, its related configuration is nevertheless present in the file (see labelit or xds in this example)

```
<plugin>
        <name> indexing </name>
        <param>
                <name> indexing_mosflm </name>
                <value> on </value>
        </param>
        <param>
                <name> indexing_labelit </name>
                <value> off </value>
        </param>
        <param>
                <name> indexing_xds </name>
                <value> off </value>
        </param>
        <plugin>
```

```
                <name> mosflm </name>
                ...
        </plugin>
        <plugin>
                <name> labelit </name>
                ...
        </plugin>
        <plugin>
                <name> xds </name>
                ...
        </plugin>
</plugin>
```

4th approach: “all is plugin” / enabling plugins parameters (other way)

Enabling a specific plugin from a plugin parent is managed within the option element. The plugin to be enabled is part of the parent option):

```
<plugin>
        <name> indexing </name>
        <optionList>
                <optionItem>
                        <name> indexing_mosflm </name>
                        <enabled> true </enabled>
                        <plugin>
                                <name> mosflm </name>
                                <paramList>
                                        <param>
                                                <name> workDir </name>
                                                <value> /path/to/workDir </value>
                                        </param>
                                </paramList>
                        </plugin>
                </optionItem>
                <optionItem>
                        <name> indexing_labelit </name>
                        <enabled> false </enabled>
                        <plugin>
                                <name> labelit </name>
                                ...
                        </plugin>
                </optionItem>
                <optionItem>
                        <name> indexing_xds </name>
                        <enabled> false </enabled>
                        <plugin>
                                <name> xds </name>
                                ...
                        </plugin>
                </optionItem>
        </optionList>
        <paramList>
              <param>
                    <name> my_param </name>
                    <value> my_value </value>
              </param>
        </paramList>
  </plugin>
```

The main differences are:

- Format
- Difference in determining whether a plugin has to be executed or not (present in the file, via param element or option element).
- Verbosity (problem?).
- ...

2.2.1.3. Auto-configuration

Possibility to generate a configuration file if it is missing. The generated configuration will be the default application configuration (mosflm should be the default external program to be executed for the indexing and the integration steps). Depending on the configuration file format, the auto-generation will be different (i.e: in the 2nd approach, the configuration generation will be recursive (a plugin parent will have to request the configuration of its direct child if any, etc...).

2.2.1.4. When will the system read the configuration file:

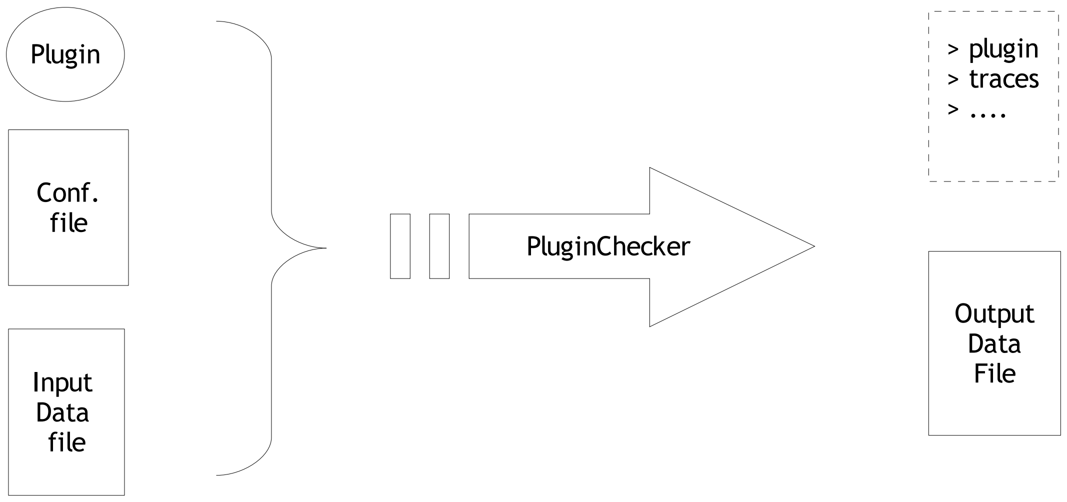
Once, at the initialization step of the application

2.2.1.5. Which configuration verifications should be done when loading the configuration file?

- Check that all the plugins are available
- Check that all the available plugins can access their related 3rd parties.
- The configuration parameters are valid
- ...

3/ Checking Plugins

Checking plugins independently is a crucial point to guarantee the modularity of the system. The aim is to verify that a particular plugin works properly in several given environments with the expected results (even the error cases). For that, it is proposed to implement a tool (i.e “ PluginChecker”) that will take as input parameters the plugin name, its configuration file (to simulate the environment) and an input xml file that will store the scientific input data needed to execute the plugin. The result will be in one hand the execution traces of the plugin and in the other hand, the scientific output data stored in a xml file. Executing the plugin in several configuration environments with several input data (even bad cases) will be a crucial point to take into account.



One way to extend the PluginChecker to a Benchmark would be to compare the obtained output data file to an expected one.

4/ Open questions

Regarding Use Case Implementation:

- Data Model: it is proposed to check if the experimental Data Model applied to the strategy during the spike can suit to the indexing and the integration.
- Data Modeling Tools: Enterprise Architect / Umbrello
- Persistence (should not be included in the prototype)

Regarding AALib:

- How to propagate Data between plugins ?
- Shall the prototype work in jython ?
- How to make the prototype becoming a server ?
- GRID