Advanced Software Engineering Project

ACADEMIC YEAR 2017/2018

EVA BAFARO 893961
SUSANNA POZZOLI 897788

Abstract

Using Eclipse Modeling, we defined the abstract syntax of our modeling language, then we designed a concrete graphical syntax and created a modelling environment with Eclipse Sirius.

Then, we defined two sample models.

Finally, we created two Model-to-Text (M2T) transformations with Eclipse Acceleo. The first one generates HTML reports and the second one generates R scripts.

Contents

User Guide

Import the following projects:

- `it.polimi.ase.project`
- `it.polimi.ase.project.design`
- `it.polimi.ase.project.edit`
- `it.polimi.ase.project.editor`
- `it.polimi.ase.project.pipeline2html`
- `it.polimi.ase.project.pipeline2html.ui`
- `it.polimi.ase.project.pipeline2r`
- `it.polimi.ase.project.pipeline2r.ui`
- `it.polimi.ase.project.validation`

Right-click on the `it.polimi.ase.project.editor` project and select the menu **Run As > Eclipse Application**.

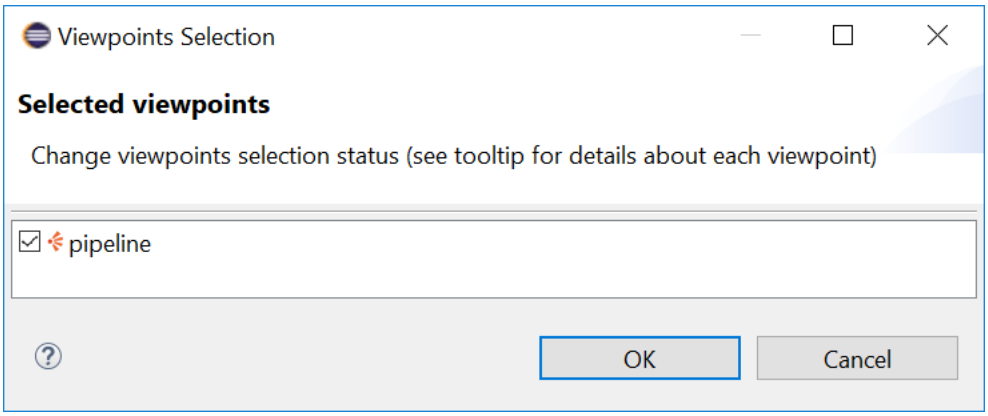
In the new Eclipse runtime, import the following projects:

- `it.polimi.ase.project.samples.a`
- `it.polimi.ase.project.samples.b`

In order to create a new Modeling project, use the menu item **File > New > Modeling Project**.

First, right-click the Modeling project in the *Model Explorer* view and select **New > Pipeline Model**. Choose the name you want for your model, then click **Next >**. Select *Pipeline* as the model object to create, then click **Finish**.

To create a new Pipeline diagram, right-click in the *Model Explorer* view on the Modeling project and select the menu **Viewpoints Selection**. You should see the viewpoint *pipeline*. You must activate this viewpoint to be able to create the representation which is defined by this viewpoint.

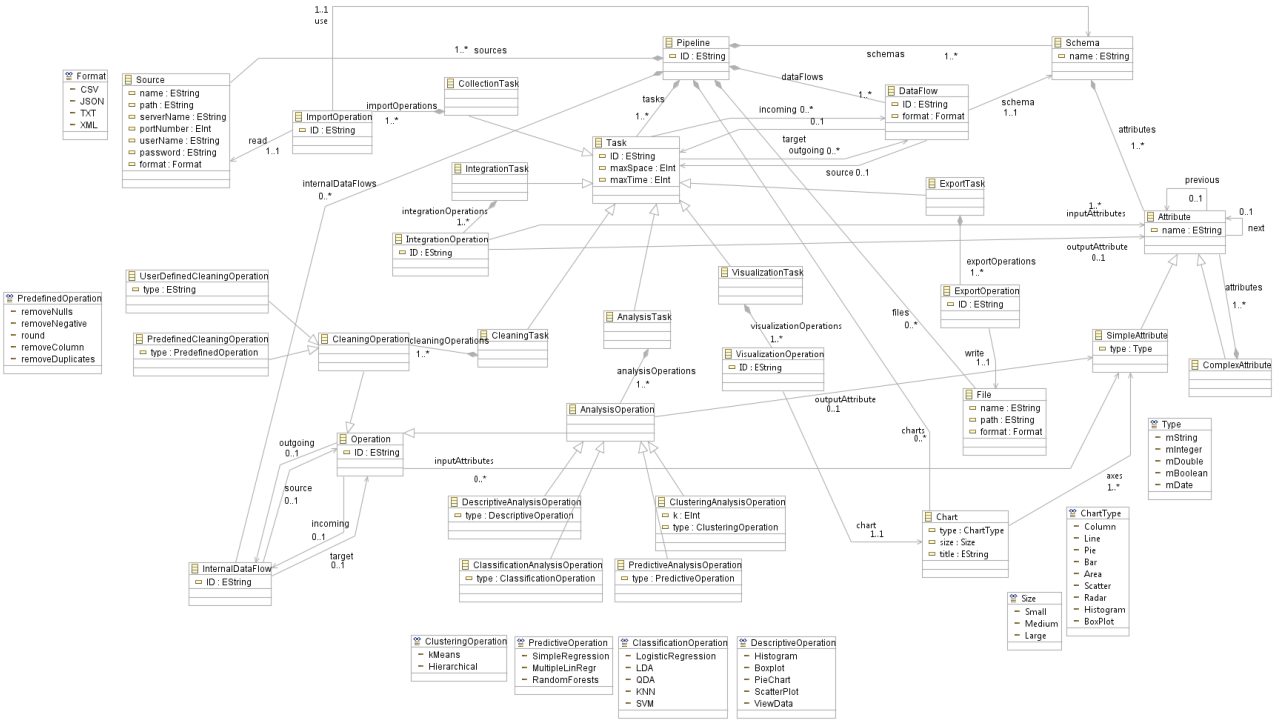


Then right-click on the Pipeline model and select the menu **New Representation > new Pipeline diagram**.

You can validate the Pipeline diagram by selecting the **Diagram > Validate** menu.

Abstract Syntax

The metamodel is defined by a *.ecore* file and some constraints (validation rules).



Ecore

The container of our model is the *EClass* **Pipeline**, which is an aggregation of the following *EClasses*:

- **Task**: the abstract class representing the various tasks, it has a name as identifier, it specializes into the concrete EClasses **CollectionTask**, **IntegrationTask**, **CleaningTask**, **AnalysisTask**, **VisualizationTask** and **ExportTask**;
- **DataFlow**: this class represents the flow of data through the various tasks;
- **Source**: this class represents the sources of the data, and is binary associated with the CollectionTask;
- **File**: this class represents the generated output, and is binary associated with the ExportTask;
- **InternalDataFlow**: it's used to link operations inside the various tasks, to define their order.

Data Flows, Schemas and Attributes

Data flows represent the flow of data between tasks. Each data flow has a format (*CSV*, *JSON*, *TXT* or *XML*), and is composed by a **Schema**. The latter is composed by **Attributes**, which can be **SimpleAttributes** or **ComplexAttributes**. The latter are composed by attributes (and it must be composed at least by two attributes, otherwise a SimpleAttribute should be used instead), while the first just have a *name* and a *type* (*mString*, *mInteger*, *mDouble*, *mBoolean* or *mDate*). Within the same schema (or the same complex attribute) attributes can't have the same name, not even if they are of different type. Attributes are ordered via the *next/previous* association.

Tasks and Operations

Each task is composed by its operations. For almost each operation we defined an *EEnumeration* to define its type.

- **CollectionTask**: for each **Source**, there's an **ImportOperation** linked to it and to its schema. The task has many outgoing data flows as many sources.
    - Sources have the attributes: *name*, *path*, *format*, and to grant remote sources *serveName*, *serverPort*, *userName* and *password*. If the source is local, only the first three parameters are set.
- **IntegrationTask**: if there are many sources, for each couple an **IntegrationOperation** is needed to join data in a unique data flow (if the source is only one this task is not needed). Every operation needs at least 2 attributes (one for each source), and can produce an output attribute.
- **CleaningTask**: it's composed by the abstract class **CleaningOperation**, which specializes in **UserDefinedCleaningOperation** and **PredefinedCleaningOperation**. Both have an ID and are associated with one or more attributes, and the latter has the attribute *type* which is an enumeration: *removeNulls*, *removeNegative*, *round*, *removeColumn* and *removeDuplicates*. This task isn't mandatory, but in the case, there must be at least one cleaning operation.
- **AnalysisTask**: it's composed by the abstract class **AnalysisOperation**, which has an ID, one or more input attributes and possibly an output attribute. There must be at least one analysis in the pipeline. We have four types of analysis:
    - **DescriptiveAnalysisOperation**, which requires input attributes but don't generate any output, they show the data as a table or in some type of plots;
    - **ClassificationAnalysisOperation**, just *KNN* generate an output attributes, the others (*LogisticRegression*, *LDA*, *QDA* and *SVM*) don't because they build a model;
    - **ClusteringAnalysisOperation**, these operations require an integer K (the number of clusters) besides the input attributes; only *kmeans* generates an attribute, *hierarchical clustering* doesn't;
    - **PredictiveAnalysisOperation**, they don't generate any attribute as they build models, they are *Linear regression*, *Multiple linear regression* and *Random forests*; for the regression we have the condition that the input attribute must be a number (integer or float).
- **VisualizationTask**: it's composed by the **VisualizationOperation**s, which have an ID and are associated with a **Chart** each. Charts are then associated with attributes which will be their axis. This task is not mandatory.
- **ExportTask**: for each generated **File**, there's an **ExportOperation** linked to it.
    - Each File has a *name*, *path* and *format* (we suppose to export files only locally).

For the Cleaning and Analysis task, and order between operation is required, so we have the *internal data flow*.

Tasks and Data Flows

Task is linked to DataFlow by two binary associations, so from the point of view of:

- task, we can get the *incoming* and *outgoing* data flow;
- data flow, we can get the *source* and *target* task. Operations are linked by the InternalDataFlow as tasks are linked by the DataFlow.

Epsilon Validation Language

To complete our model, we made some other assumptions, which are represented by constraints we added to the model.

The tasks are unique and executed in the order we defined them:

Collection > Integration > Cleaning > Analysis > Visualization > Export

This assumption is expressed by these type of constraints:

```
context Pipeline {

    constraint uniqueTasks {
        check: self.tasks -> select(t|t.isTypeOf(CollectionTask)) -> size()
<= 1 and self.tasks -> select(t|t.isTypeOf(IntegrationTask)) -> size() <= 1
and self.tasks -> select(t|t.isTypeOf(CleaningTask)) -> size() <= 1 and
self.tasks -> select(t|t.isTypeOf(AnalysisTask)) -> size() <= 1 and
self.tasks -> select(t|t.isTypeOf(VisualizationTask)) -> size() <= 1 and
self.tasks -> select(t|t.isTypeOf(ExportTask)) -> size() <= 1
        message: 'There can be at most 1 task per type'
    }

}
context CollectionTask {

    constraint initialTask {
        check: self.incoming -> size() = 0
        message: "Collection task can't have an incoming data flow"
    }

    constraint nextTypeCollection {
        check: (self.outgoing -> size() == 1) and (self.outgoing.target ->
select(t | t.isTypeOf(CollectionTask) or t.isTypeOf(VisualizationTask) or
t.isTypeOf(ExportTask)) -> size() = 0) or (self.outgoing -> size() > 1)
        message: "Collection task must be linked to integration, cleaning or
analysis task"
    }

}
```

Furthermore, not all of them must be present, the mandatory ones are Collection, Analysis and Export: if the source is unique, there is no need for Integration; the data can be already cleaned (no need for Cleaning); a user may not want to see the results of analysis (no need for Visualization).

All tasks except Collection and Export, must have a unique outgoing data flow, and all tasks except Integration and Collection must have a unique incoming data flow. From Collection to Integration task there can be multiple data flows, precisely as many as the sources are.

```
context CollectionTask {

    constraint manyDataFlowsFromCollectionAsManyImports {
        check: self.importOperations -> size() = (self.outgoing -> size())
        message:  "In collection task there must be as many outgoing data
flows as many input sources"
    }

    constraint allOutgoingDataFlowsTargetingSameIntegrationTask {
```

```
        check: (self.outgoing -> size() > 1) implies (self.outgoing.target -
> forAll(t | t.isTypeOf(IntegrationTask)))
        message: "If there are many outgoing data flows from the collection
task, all of them must be linked to the same integration task"
    }

}
```

Also, the internal data flow between operations must be unique (and obviously a cleaning operation can be linked only to another cleaning operation, and an analysis operation can be linked only to another analysis operation, so the internal data flow can't exit the task).

```
context Pipeline {

    constraint dataFlowBetweenCleaningOperation {
        check: self.tasks -> collect(t : CleaningTask | t.cleaningOperations
-> size()) -> sum() <= (self.internalDataFlows -> select(d |
d.source.isKindOf(CleaningOperation) and
d.target.isKindOf(CleaningOperation)) -> size() + 1) or collect(t :
CleaningTask | t.cleaningOperations -> size()) -> sum() == 0
        message: "Missing one or more data flows between cleaning
operations"
    }

}
```

Within Cleaning and Analysis Task, operation must be all executed once (they all have one incoming and one outgoing internal data flow, except for the first and the last, and there must not be cycles). This is expressed by the cardinality of the relation between operations and internal data flow, and the following constraints:

```
context CleaningTask {

    constraint uniqueInternalDataFlowOut {
        check: self.CleaningOperations -> forAll(o | o.outgoing -> size() <=
1)
        message: "Cleaning operations can have at maximum one outgoing
internal data flow"
    }

    constraint uniqueInternalDataFlowIn {
        check: self.cleaningOperations -> forAll(o | o.incoming -> size() <=
1)
        message: "Cleaning operations can have at maximum one incoming
internal data flow"
    }

    constraint initialCleaningOperation {
        check: self.cleaningOperations -> select(op | op.incoming = null) ->
size() = 1
        message: "There can be just one initial cleaning operation. Some
internal data flows are wrong"
    }

    constraint finalCleaningOperation {
        check: self.cleaningOperations -> select(op | op.outgoing = null) ->
size() = 1
        message: "There can be just one final cleaning operation. Some
internal data flows are wrong"
```

```
    }

}
```

Source can be imported only once, as files can be exported only once (and obviously each import[export] operation is associated with a unique source[file]).

```
context Pipeline {

    constraint sourceImportedOnce {
        check: self.sources -> forAll(s | (self.tasks -> select(t |
t.isTypeOf(CollectionTask)).importOperations -> forAll(i | i.read = s)) ->
size() = 1)
        message: "Imports must be linked to different input sources"
    }

}
```

Each schema must have at least one attribute, and all attributes within the same schema must have different names.

```
context Schema {

    constraint SchemaHasAttributes {
        check: self.attributes -> size() > 0
        message: "Each schema must have at least one attribute"
    }

    constraint uniqueNameAttribute {
        check: self.attributes -> size() > 0 and self.attributes -> forAll
(a1 | self.attributes -> forAll (a2 | a1 <> a2 implies a1.name <> a2.name))
        message: "There can't be more attributes with the same name in the
same schema"
    }

}
```

As we already said, if there are many sources the Integration Task is needed, and the number of *integrationOperation* is the number of sources − 1 (they are merged one pair at a time). Obviously, the attributes on which we join must be of the same type.

```
context Pipeline {

    constraint ifManySourcesIntegration {
        check: self.sources -> size() > 1 implies self.tasks -> select(t |
t.isTypeOf(IntegrationTask)) -> size() = 1
        message: "If there are many input sources there must be an
integration task"
    }

    constraint numberOfIntegrationOperation {
        check: self.sources -> size() > 1 implies self.sources -> size() =
self.tasks -> select(t | t.isTypeOf(IntegrationTask)).integrationOperations
-> first() -> size() + 1
        message: "The number of integration operation must be the number of
sources - 1"
    }

}
```

Since the Analysis Task can produce other attributes, the incoming and outgoing data flows must have compatible schemas, which means that all the incoming schema attributes must be present in the outgoing schema attributes, and the output schema must include all the attributes generated by the analysis operations:

```
context AnalysisTask {

    constraint outputSchemaIsCompatibleWithInputSchema {
        check: self.incoming.schema.attributes -> first() -> forAll(attr1 |
self.outgoing.schema.attributes -> first() -> exists(attr2 | attr1.name =
attr2.name and attr1.type = attr2.type))
        message: "The outgoing data flow schema must contain all the
attributes of the incoming data flow schema"
    }

    constraint outgoingDataFlowHasRightSchema {
        check: self.analysisOperations.outputAttribute -> select(attr | attr
<> null) -> size() > 0 implies self.analysisOperations.outputAttribute ->
select(attr | attr <> null) -> forAll(attr1 |
self.outgoing.schema.attributes -> first() -> exists(attr2 | attr1.name =
attr2.name and attr1.type = attr2.type))
        message: "The outgoing data flow schema must contain all the
generated output attributes"
    }

}
```

For clarity and implementation reasons, analysis and cleaning operations must all have different IDs, and there is a series of constraints regarding the specific analysis operations (number and type of input and output attributes), which we have mentioned introducing them.

Concrete Syntax

Graphical Concrete Syntax

Eclipse Sirius

We created a modeling workbench with Eclipse Sirius. This diagram editor allows users to visualize and edit a pipeline with its elements and their relationships.

A *Viewpoint Specification Project* contains the definition of our modeling workbench. The Viewpoint Specification Project creation wizard creates a new project containing a *.odesign* file. This file describes the modeling workbench that we created. It will be interpreted by the Sirius runtime. In this file the wizard has created a first viewpoint we renamed to `pipeline`. This viewpoint provides a diagram that the user will be able to instantiate. We configure this diagram to graphically represent instances of *Pipelines*.

A *Diagram* shows *Nodes*, *Containers*, *Element Base Edges* and *Relation Based Edges* which are elements of the model. To define the way a diagram element is graphically represented on the diagram, it must declare a *Style*. The *Style* defines the graphical attributes of the *Diagram Element* (e.g. its color).

Sources, *Charts* and *Files* are displayed as nodes.

A *Container* is a kind of diagram element that can contain other diagram elements. On the *Pipeline diagram*, we used seven containers to represent:

- Schema
- CollectionTask
- IntegrationTask
- CleaningTask
- AnalysisTask
- VisualizationTask
- ExportTask

Now, to display children inside these containers, we added *Sub Nodes* and *Bordered Nodes*. Here, we have chosen a workspace image to represent an Operation.

The following aggregations and associations of the Domain Model are represented by *Relation Based Edges*:

- attributes
- axes
- chart
- inputAttribute
- inputAttributes
- next
- outputAttribute
- outputAttributes
- read

- use
- write

To display instances of *DataFlow* and *InternalDataFlow*, we created two *Element Based Edges*.

Palette

We completed this designer with a palette containing tools to allow users to create new model elements. We added five *Sections* named *Tools*, *Data Flows*, *Schema*, *Tasks* and *Operations* to the *Layer*. The palette is composed of tools which will allow the user to create new objects.

To create new instances of Domain Classes, we added a *Node Creation* element to the *Section Tools*. To define which kind of nodes will be created by the tool, the *Node Creation* element must be associated to an existing node (e.g. *SourceNode*). When the user will use the tool, actions will be executed. We had to define them. The first step consisted in adding a *Change Contex* element to define the context of the actions. The *Change Contex* element contains an expression which allows Sirius to find the model element which will be the context. Then, we added a *Create Instance* which will create a new instance of *Source*. We specified the type of the new instance and how it will be attached to the context. We copied, pasted and adapted this tool to provide a *Node Creation* tool for all the elements.

An *Edge Creation* tool allows the user to create relationships directly from the diagram, by using the palette. We used this tool to allow the user to set the relations of an element. We created an *Edge Creation* tool to set the successor of an *Attribute*. Then we defined the operations that will be performed by this tool each time the user will click on it. Under the *Begin* object, we created a *Change Context*. We set its *Browse Expression* to `var:source` in order to define the execution context of the next operations. Under the *Change Context* we created a *Set*. It will set the *next* of the first Attribute clicked (source) to the second Attribute clicked (target):

- **Feature Name:** `next`
- **Value Expression:** `var:target`

For the element based case, it is slightly different as we have to create the element and store it in the model and then we need to set the references between this element and its "source" and "target". We proceeded the same way to create an *Edge Creation* tool for all the relationships.

A *Reconnect Edge* tool allows the user to change the end of a relationship by moving it directly from the diagram. We created a *Reconnect Edge* tool to change the *next* of an Attribute. We associated the *nextEdge* to this reconnect tool. Then we created a *Change Context* and set its expression to `var:element` (the attribute that will change its successor). Finally, we created a *Set* to assign the new selected attribute (`var:target`) as *next* of this attribute. We copied, pasted and updated this tool to create a *Reconnect Edge* tool for all the relationships.

A *Delete Element* tool specifies which actions have to be performed when the user hits the *delete* key on a diagram element. It is necessary on elements which deletion can't be interpreted by Sirius (for example *edges*). We created a *Delete Element* tool to specify what to do when the user deletes a succession relation. We associated the *nextEdge* to this Delete tool. So, we created a *Change Context* and set its expression to `var:element` (the attribute which is the predecessor). Then we created an *Unset* on the feature *next* to remove this relation. We proceeded the same way to create a *Delete Element* tool for all the relationships.

Samples

Sample A

The first example is a simple in case in which we have a unique source, representing some social media data, on which we want to perform some statistical analyzes.

The source *posts.csv* is a simple *csv* file, with the following columns:

- *date*: date on which the data were collected;
- *source*: the social media which data are referred to (Facebook, Instagram, Twitter, ...);
- *numbPosts*: the number of posts/photos posted that day;
- *numbUsers*: the number of users that posted something that day;
- *mostActiveHour*: the hour in which the number of users was maximum for that day;
- *holiday*: if that day was a holiday or a day off.

Data are collected in the first task (*Collection task*) through an *import operation*, which uses the *schema* described above. The *import operation* is linked to its *source* by a dotted arrow, and to its *schema* by a dashed arrow.

A *data flow* is connected to its *schema* by a dotted line.

Then some *cleaning operation* in the *Cleaning task* are done:

- a *user defined* operation, in which data are ordered by date;
- a *predefined* operation, precisely *removeColumn*, removing the *source* column;
- another *user defined* operation, in which data are aggregated by date (number of posts and number of users are added, and as mostActiveHour we will consider the maximum).

The order is defined by the *internal data flow*. *Cleaning operation* that requires input attributes are linked to them by dashed arrows. (Since *user defined* operation are defined by a simple short description by the user, in the implementation of the pipeline the user will have to complete these operations.)

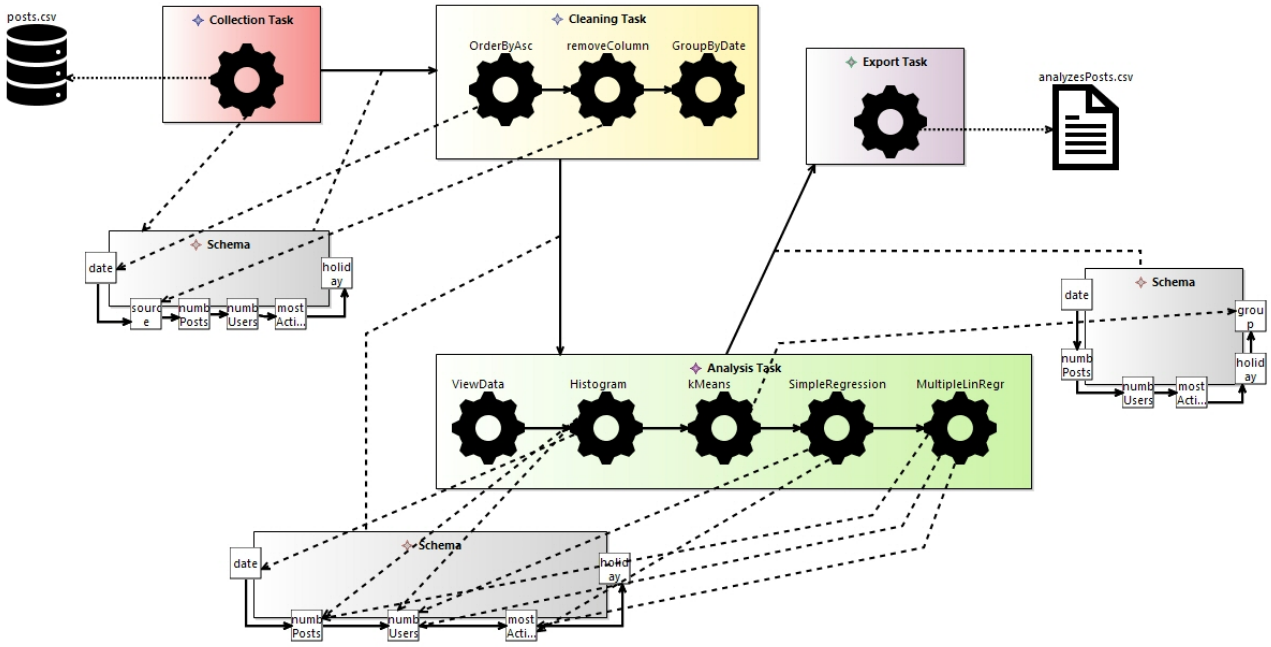
The *data flow* from *Cleaning task* to *Analysis Task* has a new *schema*, which is like the initial one, except for the *source* attribute which was removed. Now analysis operations are performed (again the order is defined by the *internal data flow*):

- *ViewData*: the user will simply view a table representing all data (no input attributes are required);
- *Histogram*: by specifying the input attributes, an histogram representing there attributes will be built;
- *kMeans*: observation will be grouped into k (in this case we specify k = 5) clusters, and a new attribute *group* is added to the schema;
- *LinearRegression*: a simple linear model predicting the *mostActiveHour* on *numbUsers*;
- *MultipleLinearRegression*: a linear model predicting the *mostActiveHour* on *numbUsers* and *numbPosts*. Again, *analysis operations* are linked to their input/output attributes (if present) by dashed arrows.

The final *data flow*, from *Analysis Task* to *Export task*, has again another schema, which is the previous one plus the attribute *group*.

In the final task (*Export task*), just one file is exported through an *export operation*, and the file *analyzesPosts.csv* is generated. The *export operation* is linked to its *file* by a dotted line.

Here is a representation of the model:



Sample B

Now a little more complex example. We have some sensors that measure air quality: one for temperature and humidity, one for pressure and another for pollution. Each sensor writes data in a different file, the first and the latter in *csv* format, the other in *txt*.

The *schemas* are:

- temperature and humidity:
  - *timestamp*: an integer that represents date and time in the *epoch* format;
  - *temperature*: a double value that represents the temperature;
  - *humidity*: an integer that represents the percentage of humidity;
- pressure:
  - every row will be a string in the format: *"timestamp_value"*;
- pollution:
  - *timestamp*;
  - *pm1*: integer representing the pm1 particles value;
  - *pm25*: integer representing the pm2.5 particles value;
  - *pm10*: integer representing the pm10 particles value.

Since we have three sources, we have an *Integration task*, and we have three data flows between *Cleaning* and *Integration* tasks.

In the *Integration Task*, we have 2 *integration operations*, which join couple by couple the sources in a unique one.

(Since one source is a txt format, in the implementation part the user will have to complete the join part, after splitting the file in the right columns.)

After the integration, the schema has the following attributes: *timestamp*, *temperature*, *humidity*, *pressure*, *pm1*, *pm25* and *pm10*.

In the *Cleaning task* we have the following *cleaning operations*:

- a *user defined* operation will format *timestamp* into *date* and *hour*;
- *removeColumn pm1*;
- *removeNulls* on the whole table.

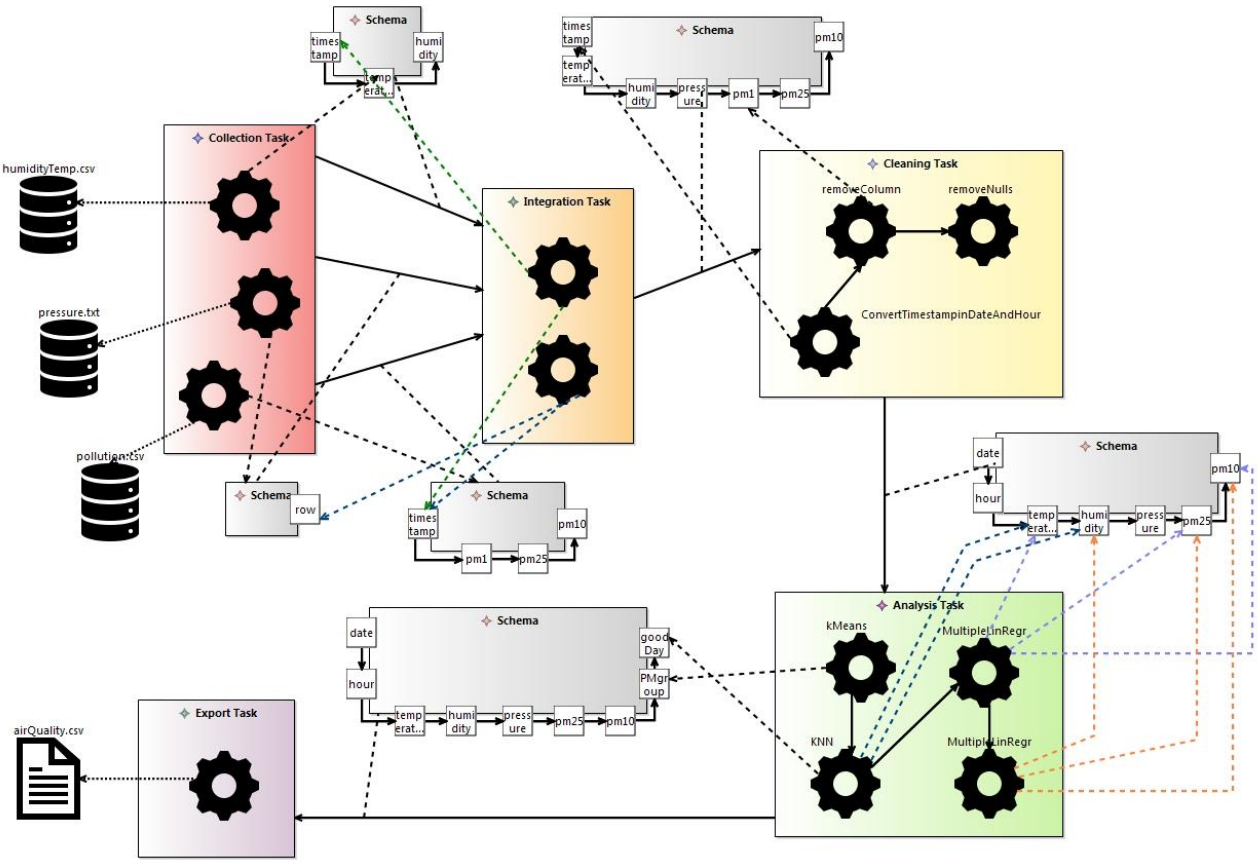
Since one column was splitted in two other columns, and another was removed, the outgoing data flow from *cleaning task* has a new *schema*.

Then we do the following *analysis operations* in the *Analysis Task*:

- *kMeans* with k = 4;
- *KNN* (K Nearest Neighbour, the default k is 3, in the implementation part it can be changed) to classify data according to *temperature* and *humidity*;
- *MultipleLinearRegression* to estimate *temperature* from *pm* values;
- *MultipleLinearRegression* to estimate *humidity* from *pm* values.

Finally, in the *Export task* we export the final schema in a new *csv* file, named *airQuality.csv*.

Here is the diagram (arrows and links have different colors for clarity):



Model-to-Text Transformations

HTML Report

it.polimi.ase.project.pipeline2html

reportHtmlFile.mtl

The `generateHtml()` template uses the "file block" to generate *.html* files. Bootstrap requires the use of the HTML5 doctype. To ensure proper rendering and touch zooming for all devices, we added the responsive viewport metatag to our `<head>`.

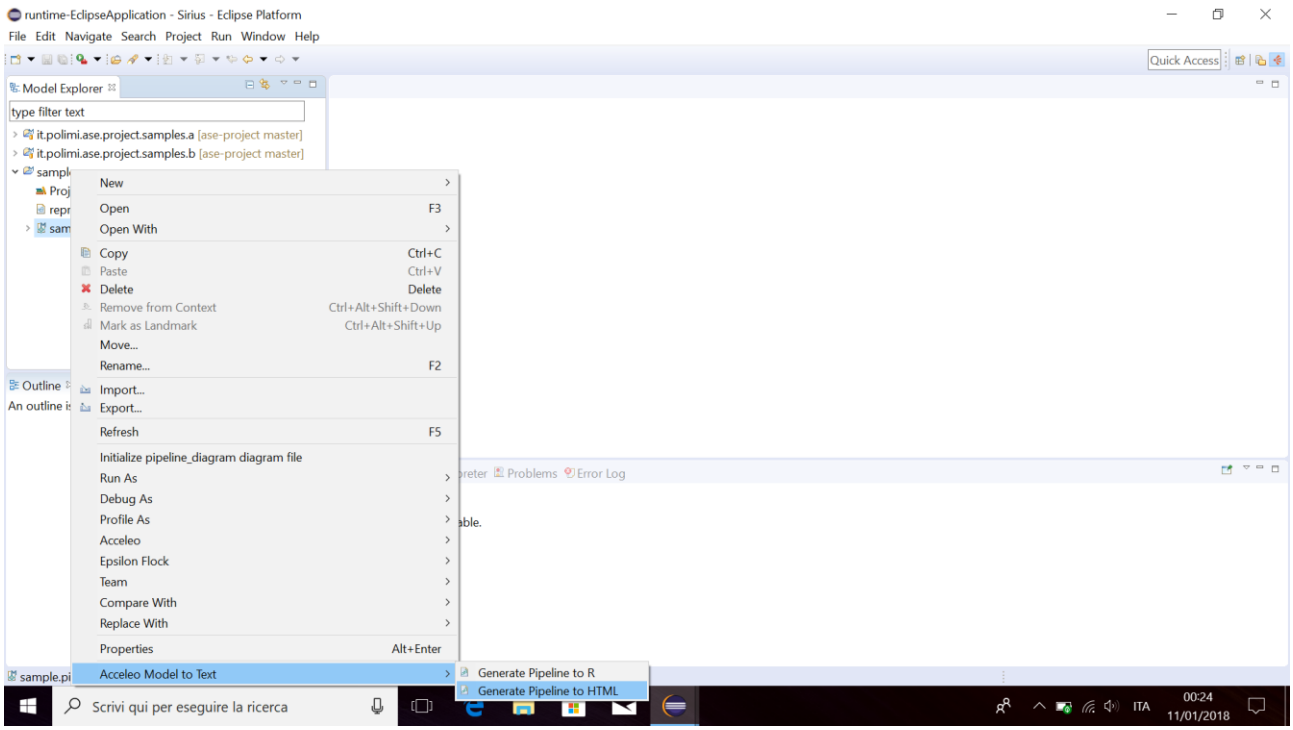
We defined three divisions in the HTML document. The first `<div>` contains information about the pipeline. The second `<div>` tag is used to group block-elements containing information on the tasks inside the pipeline. For each task, the user is provided with information on it, including incoming data flow(s) or source(s), operations and outgoing data flow(s) or file(s). The third `<div>` tag is used to group block-elements containing information on the operations inside the tasks of the pipeline. For each operation, the user is provided with information on it, including incoming internal data flow(s) or source(s), input attribute(s), output attribute(s) and outgoing internal data flow(s) or file(s).

mainModule.mtl

We created a "main" module which role is to delegate to all of the modules that will create files. This module is placed alone in its own *it.polimi.ase.project.pipeline2html.main* package.

it.polimi.ase.project.pipeline2html.ui

Now that our generation modules are ready, we want to have some wizards to launch the generation from within Eclipse. The **New Acceleo UI project** wizard will create a new Eclipse project which will allow the end user to launch the generation with a right-click action on any appropriate model. The wizard created a new plugin with all the necessary code to display a new action for the selected model file that will generate *.html* files in the specified folder. The result of this plugin is a **Generate Pipeline to HTML** action on the *.pipeline* files.



R Script

it.polimi.ase.project.pipeline2r

scriptRFile.mtl

This module implements a *partial code generation* (so the user will have to complete the script in some points identified by a *TODO*), and will generate a *.r* file.

The script, generated through the *file* tag, imports first all the needed libraries. Then all sources are imported, saved in variables named as the file (removing any space). According to the format:

- *csv*: the file is simply imported as a data frame;
- *txt*: the file is imported and should automatically be read as a data frame, but if the file is badly formatted, this won't happen, so the user will have to check if the file was correctly imported, otherwise he will have to fix it (e.g., if every row is read as a unique column he will have to split it in the various columns); a *TODO* will mark this situation;
- *xml*: the file is imported as a xml structured variable, but the program can handle just tables and data frames, so only the following structure of xml file can be read (and converted to data frames, this also because the purpose of the program is to handle data, which can be fairly expressed in this way in an *xml* file):

```
<rootNode>
    <listNode>
        <attribute1>…</attribute1>
        <attribute2>…</attribute2>
        …
        <attributeN>…</attributeN>
    </listNode>
    …
    <listNode>
```

```
        …
    </listNode>
</rootNode>
```

- *json*: the behavior is similar to the *csv* files, the *json* file is imported and the user will have to complete the conversion to a data frame, by pointing the correct path to the elements (note that in the end the structure must be an array of objects structured in the same way).

If the sources are local, they are simply imported, if the *path* is not specified, they are assumed to be in a folder named *sources* at the same level of the *.r* file.

If the sources are remote: if they need credentials, the user will have to manually download them (a *TODO* will highlight link and credentials), if not they are automatically downloaded and imported by the script.

If there are many sources they are all merged in a unique data frame, named sourceDF, obtained by joining in cascade all the data frames, then if the user specified a different output attribute for the integration operation, the corresponding column will be renamed.

Example of joining 3 data frames:

```
merge(merge(df1, df2, by.x = 'c1', by.y = 'c2'), df3, by.x = 'c1', by.y = 'c3')
```

Then we have the cleaning operations: the *predefined* ones are already defined and ready to use, but the *user defined cleaning operations* must be written by the user (these are marked by *TODO*).

In the SampleA, we created a source file (with random values). For the cleaning task we defined two user-defined operations, and we manually added the corresponding code. All sources and output files are included in the *it.polimi.ase.project.samples.a/src-gen* folder.

Also, the analysis operations are already defined, just for the *kMeans* (*classification operation*), the user can change the *k* parameter (default set to 3).

For the visualization task, all plots are rendered through the *ggplot2* library. Note that it's very important to link *Charts* to the attributes (axes) in the right order (first x axis, then y axis, finally eventual grouping attribute), otherwise the plots will be rendered in the wrong way, and the user will have to manually change the order and/or the parameters of the various functions. Attributes are copied in a new data frame called data2plot, so the user can see which attributes (and with which names) he can use, without modifying the primary data frame sourceDF.

Finally, files are exported, for *txt* and *csv* format the data frame is just saved in the corresponding format, but for:

- *json*: the data frame must be converted to a *json* structure, precisely it will be an array of objects, with as attributes the columns of the data frame, then the result will be saved in a *json* file;
- *csv*: the data frame must be converted to the tree structure, precisely a root element is created, then for each row a child is added to the root, and for each column another child is added to the root child; the result is then saved in a *xml* file.

For some operations (like regression), there are no output attributes generated, but we wanted to save the result anyway, so the *summary* of the operation is saved in a *.txt* file in the *output* folder. Also plots are saved in a *.pdf* file in the *output* folder.

mainModule.mtl

We created a "main" module which role is to delegate to all of the modules that will create files. This module is placed alone in its own *it.polimi.ase.project.pipeline2r.main* package.

it.polimi.ase.project.pipeline2r.ui

Now that our generation modules are ready, we want to have some wizards to launch the generation from within Eclipse. The **New Acceleo UI project** wizard will create a new Eclipse project which will allow the end user to launch the generation with a right-click action on any appropriate model. The wizard created a new plugin with all the necessary code to display a new action for the selected model file that will generate *.r* files in the specified folder. The result of this plugin is a **Generate Pipeline to R** action on the *.pipeline* files.