Rapport de projet : Ingénierie dirigée par les modèles

BARILLY Victor &
BEUHORRY-SASSUS Nicolai &
DAVESNE Adrien &
JOMA'A Ainji &
RAGOT Cyrian

Décembre 2024

Contents

Introduction

L'informatique moderne repose sur la capacité à traiter des données et à effectuer des calculs, comme le fait Excel, largement utilisé depuis les années 80. Cependant, ces outils peuvent être complexes à manipuler et limitent l'automatisation des tâches. Le but de notre projet est donc d'offrir une solution pour permettre aux experts non informaticiens de créer des outils adaptés à leurs besoins.

Le projet se concentre sur deux types d'utilisateurs avec des approches adaptées à chacun d'entre eux. Le premier, l'utilisateur principal (end user), utilise la suite d'outils pour définir des schémas de données et des calculs automatisés servant à générer des outils spécifiques. Le second, l'utilisateur final (end user target), exploite ces outils générés pour importer, visualiser et exporter des données en accord avec les schémas définis. Nous détaillerons les fonctionnalités mises en place pour chacun : les outils de conception pour le premier et les outils d'exploitation pratiques pour le second.

Fonctionnalités mises en place dans notre projet :

Voici un résumé de ce que nous avons développé dans notre projet. Chacun de ces points sera expliqué en détails dans la suite du rapport.

1. Nous avons défini des méta-modèles de tables et d'algorithmes.
2. Une interface Sirius a été développée pour permettre au "End-user" de créer des modèles dans une interface ergonomique.
3. Des contraintes OCL ont été intégrées pour permettre aux utilisateurs de valider la définition de leurs modèles.
4. Nous avons créé une librairie de manipulation des tableaux que nous avons utilisée nous-mêmes pour développer notre projet et que nous avons mise à disposition du "End-user"
5. Pour donner vie à tout cela, nous avons utilisé l'intercession et l'introspection de Java pour pouvoir lire les fonctions fournies dans le script de l'algorithme et les appliquer sur des colonnes issues des modèles de tables.
6. Ensuite, nous avons défini un méta-modèle de mini-langage avec son Sirius et ses contraintes OCL permettant d'écrire un code dans un langage simple sous forme de blocs et de valider ce code.
7. Nous avons commencé une transformation de ce modèle en un code Java, mais nous n'avons pas réussi à mener cette partie à terme.
8. Enfin, nous avons créé une interface Swing pour le "end user target" qui lui permet d'afficher des fichiers CSV, d'effectuer des calculs sur un modèle de table et un algorithme définis et finalement d'afficher le résultat obtenus par les calculs.

1 Schéma de table :

1.1 Méta-Modèle :

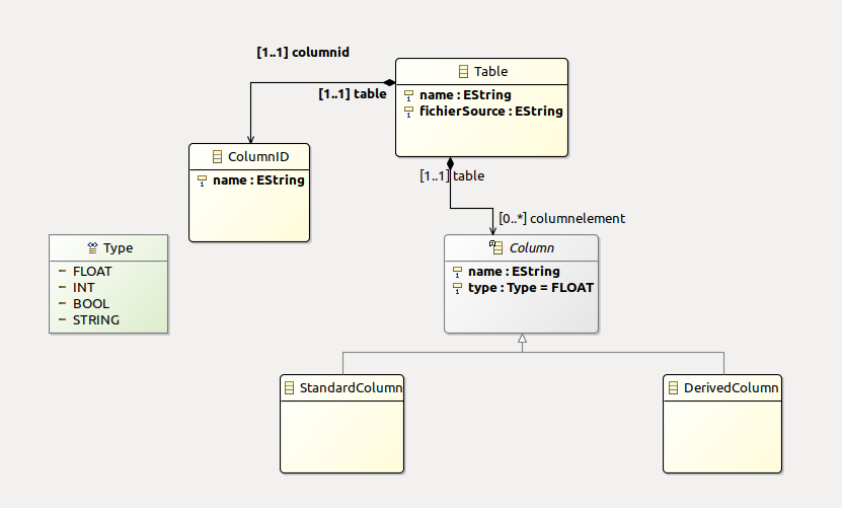

Figure 1: Méta-Modèle du schéma de table

On peut voir dans la figure 1, le méta-modèle du schéma de table défini dans le fichier "Table.ecore"

1. La Classe `Table`

- Représente le schéma de table qui va être manipulé par le "End User".
- L'attribut `fichierSource` : Indique le fichier CSV contenant les données associées à cette table.

2. La Classe `ColumnID`

- Représente les identifiants de ligne, cette colonne est rajoutée aux colonnes déjà existentes, il s'agit d'une colonne où les lignes sont numérotées de 1 à nbLignes.
- Elle est liée à une unique `Table` via la relation `[1..1] table`. En effet, chaque column ID a une unique tabe et chaque table a une unique column ID.

3. La Classe `Column`

- Représente une colonne dans le schéma de table.
- `type` : est le type réel des éléments de la colonne, avec une valeur par défaut `FLOAT`. Ce type peut être changé en n'importe quel élément de l'énumeration "Type".

4. Spécialisation de `Column`

- Deux sous-classes héritent de `Column` :
  - `StandardColumn` : Représente une colonne standard, c'est à dire une colonne dont les éléments sont donnés en entrée dans le fichier CSV.
  - `DerivedColumn` : Représente une colonne dérivée, calculée à partir d'autres colonnes.

5. Classe `Type`

Définit les types de données possibles pour une colonne.

1.2 Contraintes Ocl-Validation :

Nous avons défini les contraintes Ocl suivantes qu'une `Table` doit respecter :

1. Le nom d'une colonne doit être unique.
2. Le nom d'une colonne doit commencer par le nom de la table à laquelle elle appartient. Le nom d'une colonne doit donc être de la forme "nomTable.nomColonne".
3. Le fichier source de la table doit être indiqué.
4. Le nom de chaque élément doit respecter les conventions Java.

1.3 Interface graphique - Sirius

Afin de répondre à l'exigence globale **F1** du Cahier Des Charges Fonctionnel (CDCF), nous avons implanté une interface graphique ergonomique à l'aide de Sirius. Cette interface permet également au "end user" de composer des modèles de schéma de table à l'aide de la palette à outil.

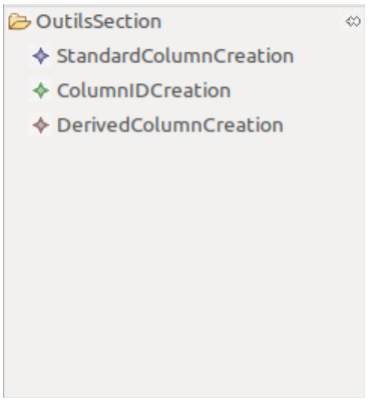

Figure 2: Palette à outil de l'interface Sirius de *Schema de table*

Comme on peut l'observer sur la Figure 2, la palette possède trois composants :

- *StandardColumnCreation* : permet de créer une nouvelle colonne "standard" en cliquant sur la table correspondante.
- *ColumnIDCreation* : permet de créer une nouvelle colonne "identifiant" en cliquant sur la table correspondante. Cette caractéristique est un choix de conception permettant d'assurer l'existence d'une colonne avec des clés primaires.
- *DerivedColumnCreation* : permet de créer une nouvelle colonne "dérivée" en cliquant sur la table correspondante.

Pour chaque élément du modèle de *Table* créé, ses attributs sont directement modifiables à l'aide de la vue propriété déjà intégrée dans Eclipse (Figure 3).

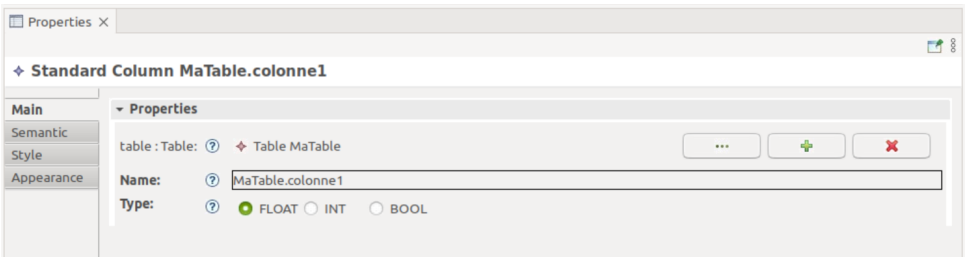

Figure 3: Apercu de la section "Propriété" pour l'élément *MaTable.colonne1* (Figure 5)

Ainsi en se basant sur un exemple d'ensemble de données fourni par le "end user target" (Figure 4), voici le schéma de table correspondant que pourrait poser le "end user" (Figure 5).

```
1 MaTable.colonne1,MaTable.colonne2
2 3.14,2.71
3 1.62,0.58
```

Figure 4: Exemple de fichier (au format *.csv*) fourni par le "end user target"

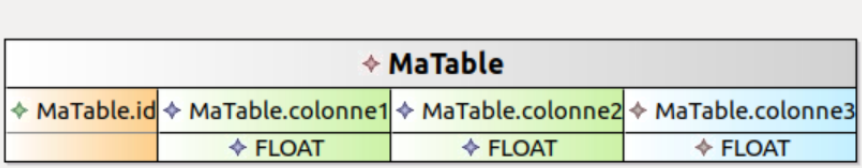

Figure 5: Schéma de table posé par le "end user" basé sur l'exemple de la Figure 4

De ce fait on comprend assez facilement que ce schéma de table est composé de quatre colonnes :

- Une colonne identifiant qui doit être obligatoirement renseignée comme indiqué dans le métamodèle de schéma de table.
- Deux colonnes "standard[s]" qui correspondent aux colonnes déjà présentes dans la ressource liée au schéma de table.
- Une colonne "dérivée" qui correspond à la colonne dans laquelle le résultat du traitement sera stocké.

1.4 Librairie de manipulation de modèles de table :

Vous trouverez ci-dessous l'interface commentée de la librairie "TableCode.java" qui permet de manipuler les schémas de table. Celle-ci comporte :

- **Un Constructeur:** Le constructeur `TableCode` permet de construire un tableau en définissant les colonnes comme elles sont données dans le modèle .xmi et d'attribuer à chaque colonne les valeurs qui lui correspondent dans le fichier CSV.
- **Des Attributs:** Chaque attribut stocke une partie des informations sur les colonnes (entêtes, données ou chemins des fichiers).
- **Méthodes principales:** Ces méthodes gèrent l'affichage, la récupération, l'ajout ou l'export des colonnes. La méthode getColonne permet par exemple de récupérer des colonnes à partir de leurs noms et la méthode setColonne d'attribuer des valeurs aux colonnes dérivées.
- **Méthodes utilitaires:** Ce sont des méthodes dont on a eu besoin pour implémenter les méthodes principales.

```
1 public class TableCode {
2
3     // Attributs prives de la classe
4
5     // Chemin du fichier XMI
```

```
6       private String cheminFichierXmi;
7       // Chemin du fichier CSV
8       private String cheminFichierCsv;
9       // Entetes de toutes les colonnes
10      private Map<String, Integer> entetes;
11      // Entetes des colonnes standards
12      private Map<String, Integer> entetesStandard;
13      // Entetes des colonnes derivees
14      private Map<String, Integer> entetesDerivees;
15      // Le tableau genere a partir des entetes et des donnes du fichier CSV
16      private Map<String, List<String>> Tableau;
17      // Une copie du tableau qui sera utile pour l'affichage
18      private Map<Integer, List<String>> TableauAffichage;
19
20      // Constructeur principal pour initialiser un objet TableCode
21      public TableCode(String cheminFichierXmi);
22
23
24      // Getters pour les attributs principaux
25
26      public String getCheminFichierCsv();
27      public String getCheminFichierXmi();
28      public Map<String, Integer> getEntetesStandard();
29      public Map<String, List<String>> getTableau();
30      public Map<Integer, List<String>> getTableauAffichage();
31      public Map<String, Integer> getEntetes();
32      public Map<String, Integer> getEntetesDerivees();
33
34
35      // Methodes principales de la classe
36
37      // Affiche les donnees de la table en console
38      public void afficherTableau();
39      // Modifie les donnees associees a une colonne derivee
40      public <T> void setColonneDerivee(String nom, List<T> list);
41      // Recupere la colonne "nom" de type "type"
42      public <T> List<T> getColonne(String nom, Class<T> type);
43      // Exporte la table en fichier CSV
44      public void exportTableau();
45      // Affiche une colonne specifique de nom "nom"
46      public void afficherColonne(String nom);
47
48
49
50      // Methodes utilitaires statiques
51      // Cree une liste d'entiers de p a n
52      public static List<Integer> createList(int p, int n);
53       // Affiche les entetes standards et derivees
54      public void printEntetes();
55      // Trouve une cle par sa valeur
56      public static String findKeysByValue(Map<String, Integer> map, Integer
            valeur);
57      // Charge un modele XMI et retourne les entetes
58      public static Map<String, Integer> chargerModeleXMI(String cheminFichierXMI
            );
59  }
```

Listing 1: Interface de la classe TableCode

A noter que pour pouvoir lire et manipuler un fichier CSV nous avons également crée une classe "CSV-Parser" que vous pouvez consulter dans le fichier "CSVParser.java" qui permet de faire différentes

opérations sur un fichier CSV dont :

1. `parseCSV(String cheminFichier)` : Lit un fichier CSV et retourne une liste de listes représentant les colonnes. Chaque colonne est stockée comme une liste de chaînes.
2. `openFile(String path)`: Vérifie l'existence d'un fichier donné par son chemin et affiche un message correspondant.
3. `compterLignes(String cheminFichier)` : Compte et retourne le nombre de lignes dans un fichier.
4. `ecrireCSV(Map< Integer, List<String> > map, Map<String, Integer> entetes, String cheminFichier)`
   - **Description** : Écrit un fichier CSV en utilisant une (`map`) pour les colonnes et une autre (`map`) pour les en-têtes.
   - **Entrée** :
     - `map` : Les données à écrire (clé = numéro de colonne, valeur = liste de valeurs pour cette colonne).
     - `entetes` : Une (`map`) associant le nom des colonnes à leurs indices.
     - `cheminFichier` : Le chemin où enregistrer le fichier CSV.
   - **En Sortie** : Le fichier est écrit sur le disque.

2 Algorithme :

2.1 Méta-Modèle :

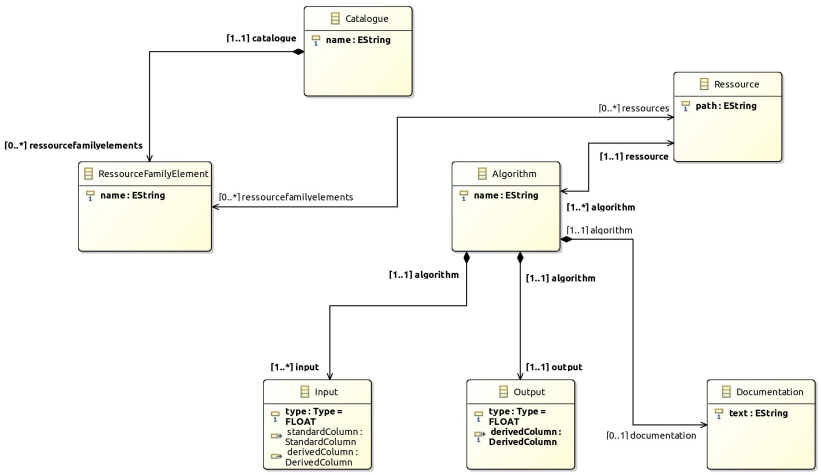

Figure 6: Méta-Modèle d'Algorithm

On peut voir dans la figure 6, le méta-modèle d'Algorithm défini dans le fichier "Algorithm.ecore"

1. La classe `Algorithm`

- Représente l'algorithme, qui va être utilisé par le "End User".
- Lié à `Input` via la relation `[1..*] input`. En effet, un `Algorithm` peut avoir une ou plusieurs colonnes en entrée.
- Lié à `Output` via la relation `[1..1] output`. En effet, un `Algorithm` a forcément une unique colonne en sortie.

- Lié à `Documentation` via la relation `[0..1] documentation`. En effet, on peut choisir d'affecter ou non une documentation à un `Algorithm`.
- Lié à `Ressource` via la relation `[1..1] ressource`. En effet, un algorithme est obligatoirement lié à une unique ressource, c'est une implémentation de la ressource.

2. La classe `Input`

- Représente les colonnes données en entrée d'un `Algorithm`
- Liée à un unique `Algorithm` via la relation `[1..1] algorithm`. En effet, nous avons choisi, pour des raisons de simplicité, qu'une colonne ne pouvait être utilisée que pour un `Algorithm`.
- `type : Type = FLOAT` : Le type déclaré des éléments de la colonne, avec une valeur par défaut `FLOAT`. Ce type peut être changé en n'importe quel élément de l'énumeration "Type".
- Une colonne en entrée d'un `Algorithm` peut être une `StandardColumn` ou une `DerivedColumn`.

3. La classe `Output`

- Représente la colonne en sortie d'un `Algorithm`
- Liée à un unique `Algorithm` via la relation `[1..1] algorithm`. En effet, une colonne est calculée à partir d'un unique `Algorithm`.
- `type : Type = FLOAT` : Le type déclaré des éléments de la colonne, avec une valeur par défaut `FLOAT`. Ce type peut être changé en n'importe quel élément de l'énumeration "Type".
- Une colonne en sortie d'un `Algorithm` ne peut être qu'une `DerivedColumn`.

4. La classe `Documentation`

- Liée à un unique `Algorithm` via la relation `[1..1] algorithm`. En effet, une documentation doit être affectée à un unique `Algorithm`.

5. La classe `Ressource`

- Représente le code associé à un `Algorithm`.
- Liée à `Algorithm` via la relation `[1..*] algorithm`. En effet, une ressource est une implémentation d'un algorithme. On peut donc en faire plusieurs.
- Liée à `Algorithm` via la relation `[0..*] ressourcefamilyelements`. En effet, une ressource peut appartenir à une ou plusieurs familles de ressource pour pouvoir être regroupée avec d'autres ressources du même type.
- `path : EString` : Le chemin d'accès de la ressource.

6. La classe `RessourceFamilyElement`

- Représente une famille de `Ressource`, comme les librairies Math ou NumPy en Python
- Liée à `Ressource` via la relation `[0..*] ressources`. En effet, elle permet de regrouper des ressources ensemble.
- Liée à `Catalogue` via la relation `[1..1] catalogue`. En effet, une famille de ressource doit appartenir à un unique catalogue.

7. La classe `Catalogue`

- Représente un ensemble de familles de `Ressource`
- Lié à `RessourceFamilyElement` via la relation `[0..*] ressourcefamilyelements`. En effet, un catalogue se compose de plusieurs familles de ressources, ce qui permet de facilement les partager.

2.2 Contraintes Ocl-Validation :

Nous avons défini les contraintes Ocl suivantes qu'un `Algorithm` doit respecter :

1. Le type déclaré d'une colonne doit être le même que le type réel de la colonne.
2. Une colonne en Input doit être exclusivement une StandardColumn ou une DerivedColumn.
3. Il faut indiquer le chemin d'accès d'une ressource.
4. Les noms de chaque élément doivent respecter les conventions Java.

2.3 Interface graphique - Sirius

Afin de répondre à l'exigence globale **F2** ainsi que l'exigence **F1.3** du CDCF, nous avons de nouveau implanté une interface graphique ergonomique à l'aide de l'outil Sirius. En suivant l'exemple introduit dans la partie concernant Sirius pour Schéma de Table (Figure 4 et Figure 5), nous pouvons imaginer que le "end user" veut definir un algorithme afin de lier la ressource utilisée au modèle de schéma de table posé précédemment (Figure 5). La ressource liée à l'algorithme posé est écrite en Java et contient le code suivant :

```java
public class ressourceAlgoImplTest1 {

    public static Double monTraitement(Double a, Double b) {
        return a+b;
    }
}

```

Figure 7: Contenu de la ressource lié à au modèle *Algorithme* posé par le "end user"

Ainsi, le traitement effectué sur le tableau et dont le résultat est stocké dans la colonne *MaTable.colonne3* (Figure 5) correspond à la somme des deux colonnes *MaTable.colonne1* et *MaTable.colonne2* (Figure 5).

Voici ci-dessous (Figure 8) un apercu de l'interface graphique dans le cadre de l'exemple de notre fil conducteur :

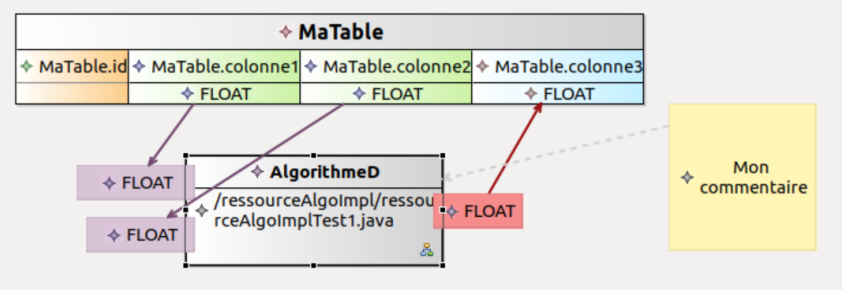

Figure 8: Interface graphique d'un modèle d'algorithme posé par le "end user"

Afin de fournir un environnement graphique aussi clair et ergonomique que possible, la representation graphique de *Schema de table* a été incluse dans celle de *Algorithme*. De ce fait des flèches, respectivement violettes et rouges pour les ports d'entrée et le port de sortie, permettent de représenter graphiquement les colonnes liées à chacun des ports.

De plus, une palette à outil est mise à disposition du "end user" afin que ce dernier puisse directement créer un modele d'algorithme à partir de l'environnement graphique.

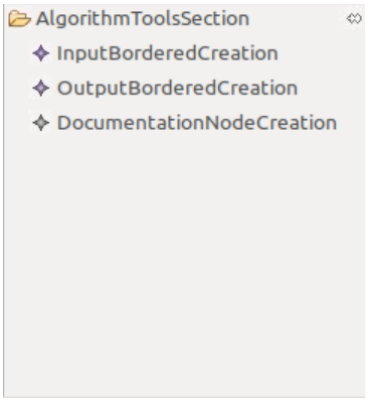

Figure 9: Palette de l'interface Sirius d'*Algorithme*

Comme on peut l'observer Figure 9, la palette possède également trois composants :

- *InputBorderedCreation* : permet de créer un nouveau port "Input" en cliquant sur l'algorithme correspondant.
- *OutputBorderedCreation* : permet de créer un nouveau port "Output" en cliquant sur l'algorithme correspondant.
- *DocumentationNodeCreation* : permet de créer une nouvelle "Note" representée par un carré jaune sur le modèle graphique afin de commenter des modèles d'*Algorithme*.

De même que pour les schémas de table pour chaque élément du modèle d'*Algorithme* créé, ses attributs sont directement modifiables à l'aide de la vue propriété déjà intégrée dans Eclipse (Figure 10).

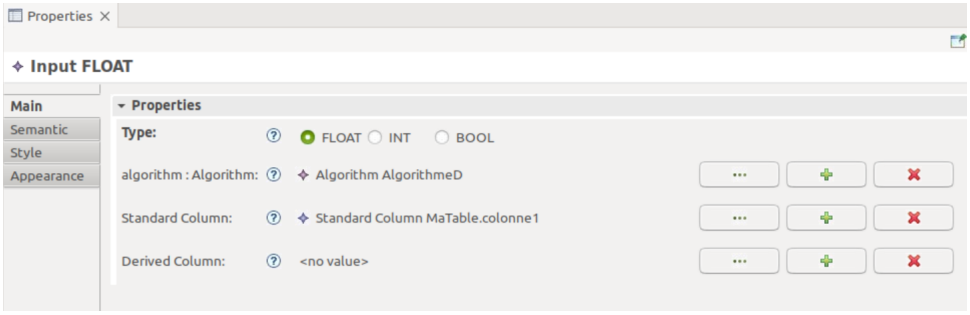

Figure 10: Apercu de la section "Propriété" pour un des ports "Input" (Figure 8)

N.B. : L'exigence **F2.6** n'a pas été pas été satisfaite malgré la réalisation partielle de son implantation, comme l'atteste le métamodèle de Algorithme.

2.4 Librairie de Manipulation de modèles d'*Algorithme* - Intercession et Introspection

Afin de lier concrètement la ressource d'Algorithme au modèle de schéma de table, c'est à dire d'exécuter le code contenu dans le fichier ressource pour que ce dernier effectue le bon traitement dans la colonne "dérivée" correspondante, il faut implanter un algorithme d'introspection et d'intercession.

Dans un premier temps, le choix a été fait de réaliser ce code en java car on a supposé en premier lieu que tous les fichiers correspondants aux ressources des modèle d'Algorithme seraient écrits en java. L'unique classe contenant l'implantation de cet algorithme (*DynamicAlgorithmExecutor*) est contenue dans le package *projetIDM.algorithm.executor*. Le code contenu dans *DynamicAlgorithmExecutor* s'appuie sur la librairie *projetIDM.table.accesseur* dont la classe principale est *TableCode.java*.

Afin de simplifier l'implantation de la partie introspection de l'algorithme, il a été decidé de considérer que la structure du code java des fichiers ressources des modèles Algorithme serait la suivante (Figure 11):

```
1 public class MaRessource {
2
3     public static MonType monTraitement(MonType param1, ..., MonType param2) {
4         // traitement
5         return ...;
6     }
7
8 }
```

Figure 11: Format imposé des ressources des modèles d'*Algorithme*

Ainsi, le fonctionnement de la classe *DynamicAlgorithmExecutor* est résumé par le pseudo-raffinage suivant :

- **R0 :** Appliquer le "traitement" de la ressource du modèle d'*Algorithme* à la colonne lié à l'"output" du même modèle.
- **R1 :** Comment **R0** ?
  Charger le modèle d'*Algorithme*

  Charger les modèles de *Table* correspondants respectivement aux colonnes contenues dans les "Input[s]" du modèle d'*Algorithme*

  Compiler et "charger" la ressource (afin de pouvoir faire de l'introspection sur cette dernière)

  "Récupérer" la première méthode *public* et *static* de la ressource (cette méthode est censée être unique si le format imposé est respecté (Figure 11))

  Lier les colonnes en "Input" avec les arguments de la méthode récupérée

  Appliquer le méthode et lier le resultat à la colonne en "Output"

3 Mini langage :

3.1 Méta-Modèle :

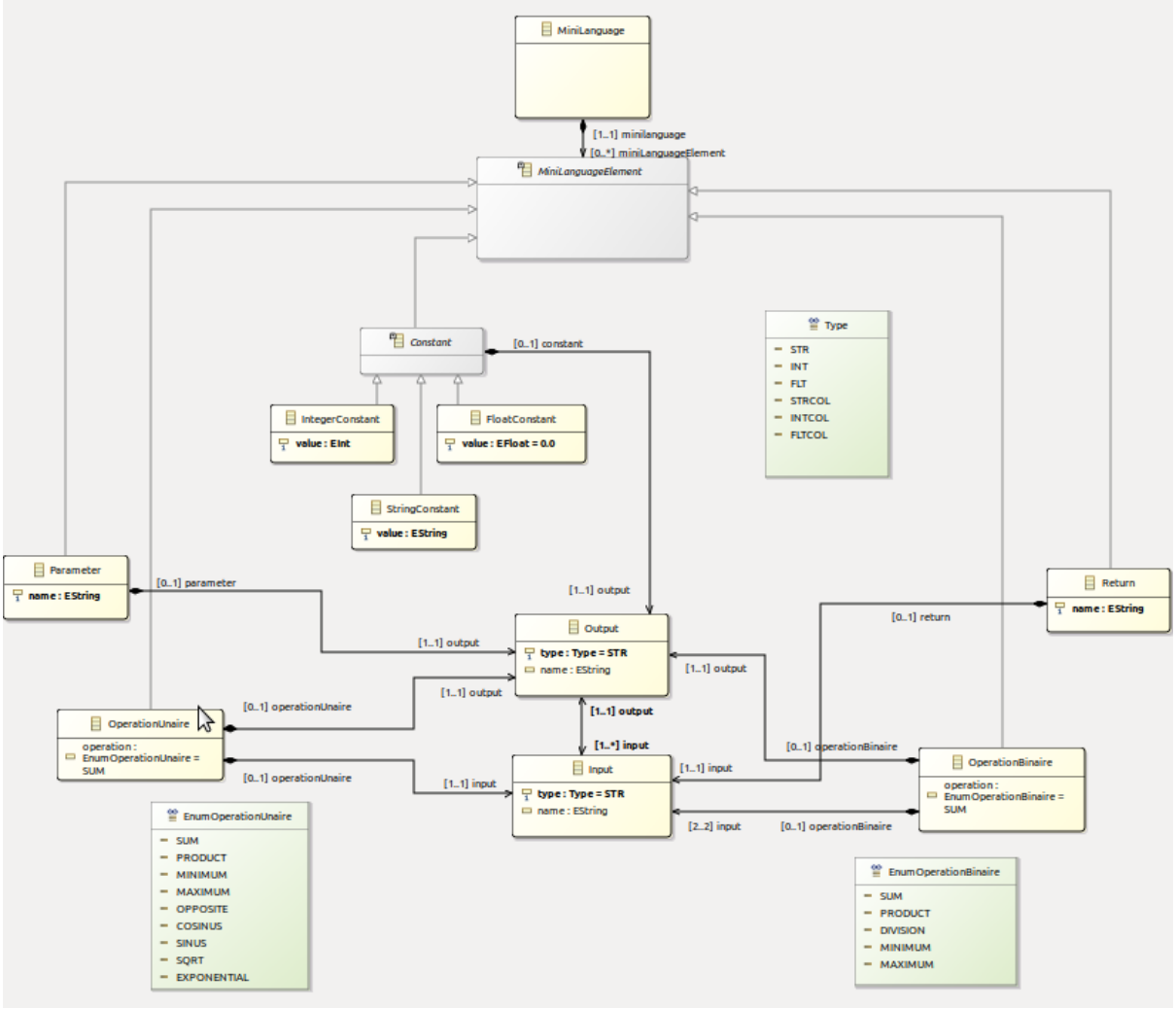

Figure 12: Méta-Modèle du mini langage

L'objectif est d'introduire un mini-langage permettant à l'utilisateur d'effectuer un calcul en utilisant un algorithme dans un autre langage ou en créant un script en utilisant ce mini-langage. Nous avons donc défini le métamodèle ci-dessus en se basant sur le principe qu'un mini-langage est composé d'éléments *miniLanguageElement* pouvant être une constante, une opération ou autre.
Chaque élément est associé à une ou plusieurs entrées (input) et/ou une sortie (output) afin de les relier entre eux dans le but d'obtenir un schéma tel que celui ci-dessous:

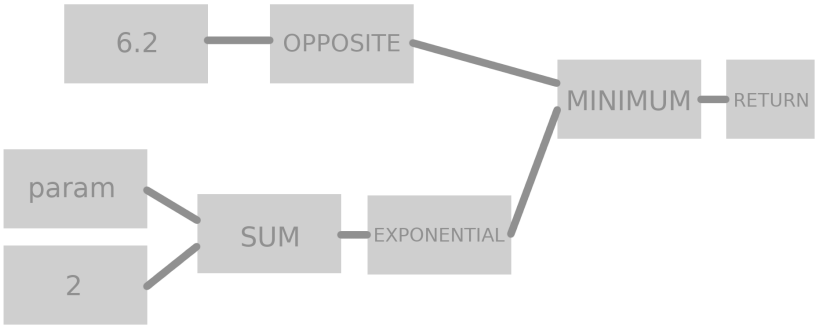

Figure 13: Exemple d'utilisation du mini langage

3.2 Contraintes Ocl-Validation :

A l'aide de *MiniLanguage/validation*, nous pouvons valider un mini-langage en vérifiant si ce dernier se plie à des contraintes en plus de son respect du méta-modèle MiniLanguage. Ce dernier vérifie entre autres:

1. Le fait qu'un identifiant soit bien formé.
2. Le fait qu'un mini-langage contienne exactement une sortie (Return).
3. Le fait qu'une opération unaire (resp. binaire) contienne 1 (resp. 2) entrée(s) compatibles entre elles et compatibles au type de sortie.
4. Le fait qu'un paramètre ou retour respecte les conventions Java.
5. Le fait qu'un paramètre mentionné existe.

3.3 Interface graphique - Sirius :

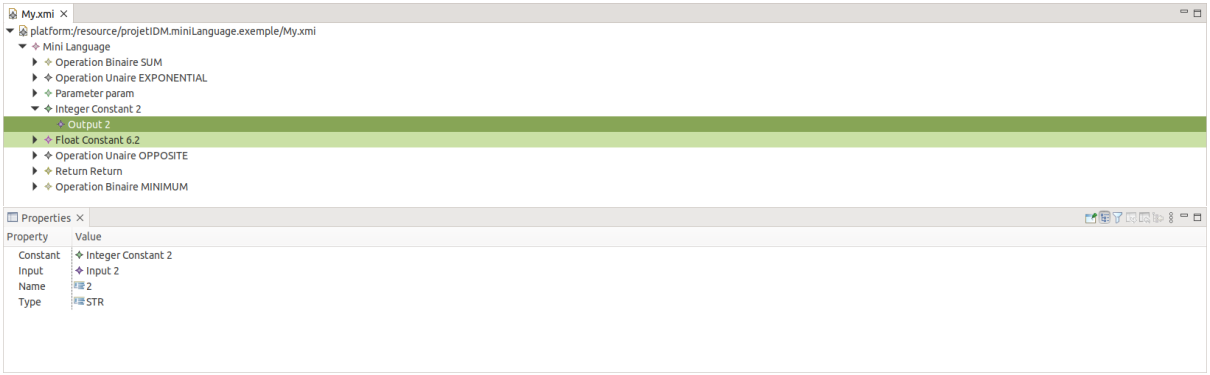

Figure 14: Représentation de l'exemple ci-dessus (My.xmi) dans le Sirius

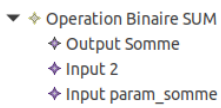

Figure 15: Dépliage de l'opération SUM dans l'exemple My.xmi

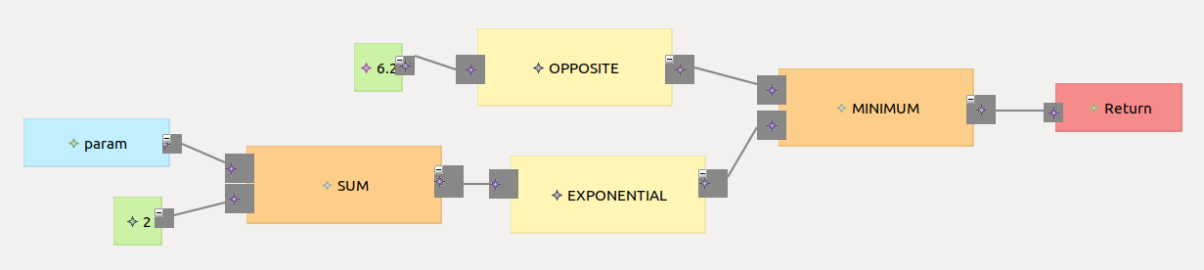

Figure 16: Rendu de l'exemple dans l'implémentation dans Sirius

3.4 Transformation du mini-langage en un code java :

Pour traduire le "code" du MiniLanguage en un script exécutable en java, on a voulu décrire un méta-modèle Xtext qui correspondrait au code java exécutable pour des calculs sur des tables. Ce langage est appelé JavaScriptForTable et fonctionne par-dessus java. Ainsi, on peut faire une transformation de notre modèle MiniLanguage vers un modèle JavaScriptForTable.

Ce langage inclut une seule classe, et une seule méthode qui prend en paramètre 0 ou plusieurs paramètres qui sont des ArrayList en java, et représentent des colonnes au vu des tables. Dans cette méthode on écrit alors toutes les opérations une à une. Chaque opération possible dans le MiniLanguage est donc écrite dans ce Xtext avec des manipulations java sur les ArrayList. Il y a beaucoup d'opérations différentes car on a considéré des types String, Float, Int et aussi des calculs avec des nombres ou des listes. On enregistre chaque résultat d'une opération dans une variable.

Il faut alors, lors de la transformation, réussir à effectuer les opérations dans le bon ordre pour que chaque opération utilise bien un résultat déjà calculé précédemment. Puis la méthode renvoie le résultat final correspondant au résultat prévu par le MiniLanguage.

4 Utilisation de l'outil généré par le "End-User target" :

Le "End-User target" peut utiliser l'outil généré par le "End-User" via une interface Swing comme celle de la Figure 17.

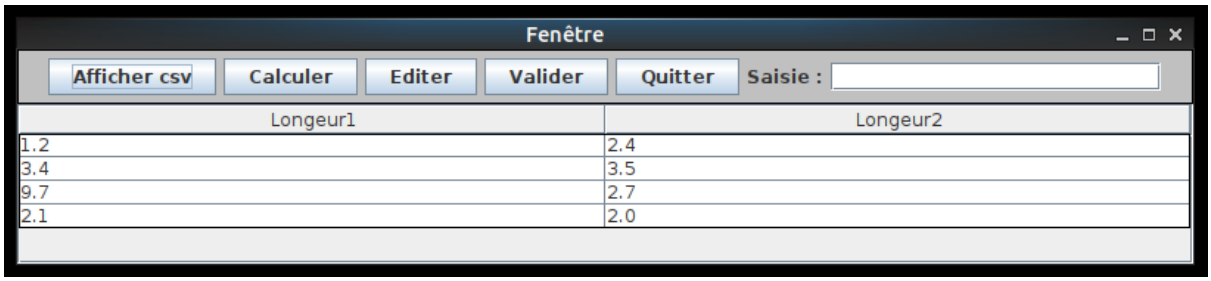

Figure 17: Interface Swing

Le "End-User target" dispose de 5 boutons sur cette interface :

- `Afficher csv` : Permet de charger un csv et de l'afficher sous forme de tableau
- `Calculer` : Permet d'effectuer l'algorithme donné sur les colonnes des tables données
- `Editer` : Permet d'enregistrer l'algorithme et les tables sur lesquelles l'effectuer après avoir été saisis dans la barre `Saisie`
- `Valider` : Permet de vérifier une contrainte sur une ou plusieurs colonnes (cette fonctionnalité n'a pas été implémentée)

- `Quitter` : Permet de quitter l'application

Conclusion

Ce projet illustre le potentiel de l'ingénierie dirigée par les modèles pour simplifier la conception et l'exploitation d'outils adaptés aux besoins des experts non informaticiens. En mettant à disposition des méta-modèles, des interfaces ergonomiques, et des librairies de manipulation de données, nous avons démontré comment rendre accessibles des processus complexes tels que la gestion de schémas de tables et l'exécution d'algorithmes sur ces données.

Malgré quelques limites, les fonctionnalités développées assurent une bonne continuité entre les étapes de conception, de validation et d'exploitation. Ce travail constitue une base solide pour de futures améliorations, comme l'enrichissement des interfaces, l'ajout d'avantages de langages de programmation ou encore l'extension des capacités d'automatisation.