Environnement de Calcul Domaine-Spécifique

Groupe L34-2

MANGENOT Guilhem
SCHEYDER Claire
ROURE Mathéo
SCHLÖGEL Benjamin

Département Sciences du Numérique - Deuxième année
2023-2024

Table des matières

Table des figures

1 Introduction

L'objectif de ce projet est de créer un ensemble d'outils permettant aux utilisateurs (***end user***) de créer des schémas de table et d'automatiser les calculs qui leur sont associés. Pour cela, nous avons utilisé la plateforme Eclipse et les technologies EMF afin de permettre à l'utilisateur de créer des modèles de schéma de table définissant les différentes colonnes, la façon dont elles sont obtenues et les contraintes qui s'y appliquent. Enfin, ces outils sont destinés à être envoyés à des utilisateurs finaux (***end user target***), qui pourront importer leurs données dans les schémas de table à l'aide d'une interface spécifique.

2 Description détaillée du rendu

Workspace eclipse-workspace-idm

Projet SchemaTable
— `schemaTable.ecore` : métamodèle de SchemaTable
— `schemaTable.ocl` : contraintes OCL de SchemaTable
— `Recettes.xmi`, `Dépenses.xmi`, `Détail.xmi` et `Resume.xmi` : schémas de table du scénario d'utilisation

Projet Algo
— `algo.ecore` : métamodèle d'Algo
— `algo.ocl` : contraintes OCL d'Algo
— `Algo_MeilleurTrimestre.xmi, Algo_Positif.xmi, Algo_RapportRecettesSurDepenses.xmi, Algo_SommeTernaire_m.xmi, Algo_SoustractionBinaire.xmi` : algorithmes du scénario d'utilisation
— `meilleurTrimestre.py, positif.py, sommeTernaire.m` : scripts extérieurs du scénario d'utilisation

Projet Expression
— `expression.ecore` : métamodèle d'Expression
— `expression.ocl` : contraintes OCL d'Expression
— `RapportRecettesSurDepenses.xmi, SommeTernaire.xmi, SoustractionBinaire.xmi` : expressions du scénario d'utilisation
— `TestTout.xmi` : expression pour tester la transformation Acceleo

Projets SchemaTable2Python et STWithMatlab2Python
— `ToPython.mtl` : code Acceleo de la transformation SchemaTable vers Python
— `Config.py` : fichier Python à compléter avec le path de l'end user target vers le workspace
— `Interface_Utilisateur.py` : fichier Python à lancer pour obtenir la vue utilisateur
— `Recettes.py, Depenses.py, Detail.py, Resume.py` : fichiers exemples générés par notre code Acceleo

Dossier CSV_Exemples
— `Recettes.csv, depenses.csv, detail.csv, resume.csv` : CSV du scénario d'utilisation

Workspace runtime-EclipseApplication

Projets *.design
— `*.odesign` : description des interfaces Sirius

Projets *.samples
— `Dependencies` : modèles du scénario d'utilisation et leurs représentations

3 Scénario d'utilisation

3.1 Scénario

Nous avons deux tables au format CSV présentant respectivement les recettes et les dépenses d'une entreprise sur chaque trimestre, pour chaque année. On souhaite dans un premier temps effectuer une série de calculs sur ces données, puis dans un second temps récupérer les résultats de certains de ces calculs pour pouvoir les présenter.

On définit d'abord les schémas de table `recettes` et `depenses` collant au format des données à importer.

Pour effectuer les calculs, on commence par définir un schéma de table `détail` important les colonnes des tables `recettes` et `dépenses` et définissant les colonnes dérivées suivantes :
— `Bilan T1`, `Bilan T2` et `Bilan T3` : la soustraction des dépenses aux recettes pour chaque trimestre. Calculé par une expression du langage dédié.
— `Bilan` : la somme des trois colonnes précédentes. Calculé par un script MatLab.
— `Meilleur trimestre` : le trimestre au plus haut bilan. Calculé par un script Python.
— `Rapport recettes/dépenses` : rapport des recettes globables par les dépenses globales, en pourcentage. Calculé par une expression du langage dédié.

On pose des contraintes de positivité sur plusieurs de ces colonnes.

On définit ensuite la table `resumé` important les colonnes `Bilan`, `Meilleur trimestre` et `Rapport recettes/dépenses` de la table `détail`.

Une interface graphique permet au client de charger les tables CSV conformes aux schémas définis, puis de vérifier les données, de calculer les colonnes dérivées, et de visualiser les tables obtenues.

3.2 Implantation

La création et l'édition de tables et d'expressions du langage dédié se fait par une syntaxe graphique concrète avec Sirius. Les deux sont ensuite converties en scripts **Python** à l'aide d'une transformation Acceleo.

Une colonne dérivée est spécifiée par un algorithme, aussi défini avec Sirius, à partir d'une ressource (script **Python** ou **MatLab**) et de colonnes en entrée. Les contraintes sont également spécifiées par des algorithmes.

L'utilisateur peut vérifier la validité de ces modèles par une évaluation de contraintes OCL.

4 Les métamodèles

Notre projet s'articule autour de trois méta-modèles qui définissent les outils nécessaires au fonctionnement du programme.

4.1 Métamodèle SchemaTable

Le schéma de table est le méta-modèle central, qui regroupe tous les outils du projet :

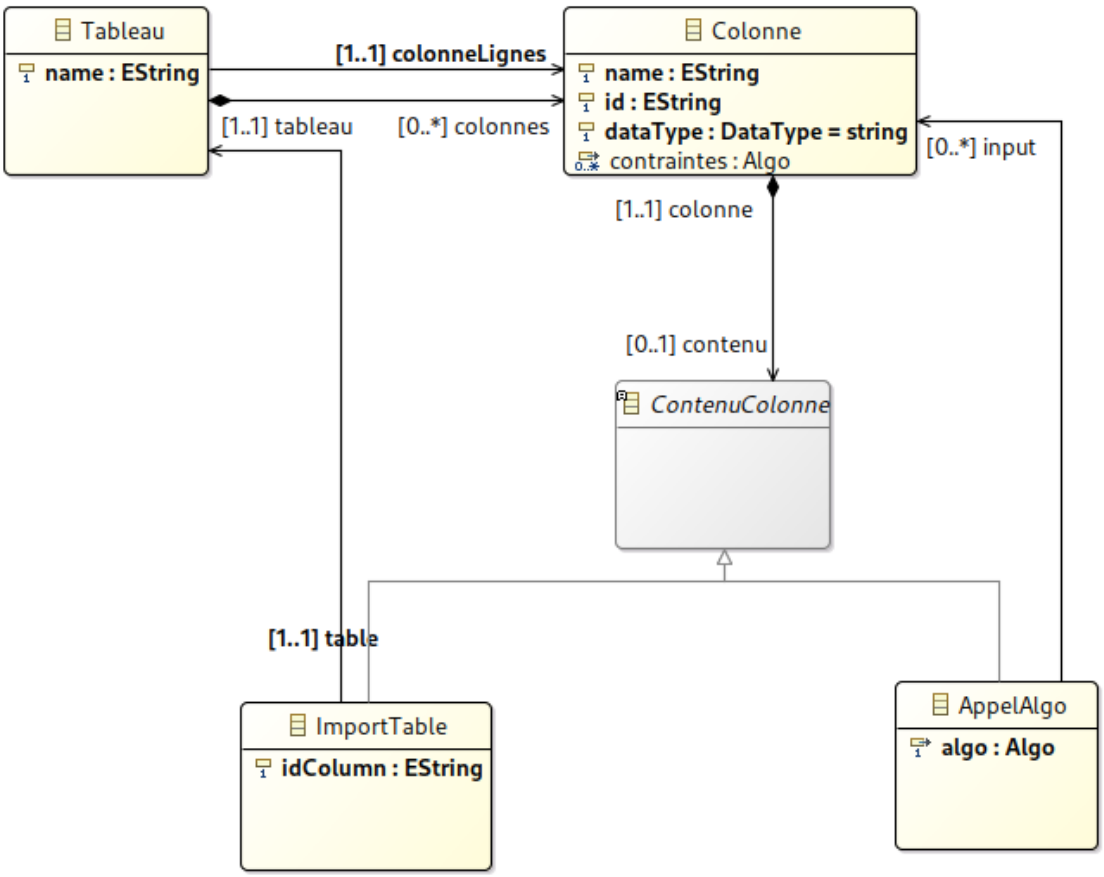

FIGURE 1 – Diagramme de classe du métamodèle SchemaTable

Un tableau possède un nom, une colonne qui servira d'identifiant pour les lignes ("colonneLignes") et un nombre indéfini d'autres colonnes.

Une colonne a un nom, un identifiant, des contraintes et un type ("DataType") qui peut être un string, un int, un bool ou un float.

Une colonne peut également posséder un ContenuColonne, qui indique si la colonne est importée d'une autre table (type ImportTable) ou si elle est le résultat d'un Algorithme (type AppelAlgo).
ImportTable référence la table de laquelle la colonne est importée et l'identifiant de cette colonne.
AppelAlgo référence un algorithme (cf métamodèle Algo) et les colonnes en entrée de l'algorithme.

Avoir un DataType pour une colonne n'est pas absolument nécessaire pour la génération de la librairie actuelle, mais d'une part, cela permet de vérifier une certaine cohérence dans les entrées et les sorties d'un algorithme grâce aux contraintes OCL et d'autre part, cela aurait été nécessaire si

nous voulions générer une librairie dans un langage où l'on doit spécifier le type des objets.

Pour les contraintes, nous avons choisi de les représenter également à l'aide d'un algorithme car cela permet de répondre parfaitement à nos besoins. Actuellement, une contrainte n'est définie que sur une colonne, mais cette implantation comporte aussi l'avantage de pouvoir évoluer facilement vers des contraintes sur plusieurs colonnes, en mettant plusieurs arguments dans la fonction de l'algorithme et en effectuant les modifications nécessaires dans la génération de la librairie.

4.2 Métamodèle Algo

Les algorithmes représentent la partie calculatoire :

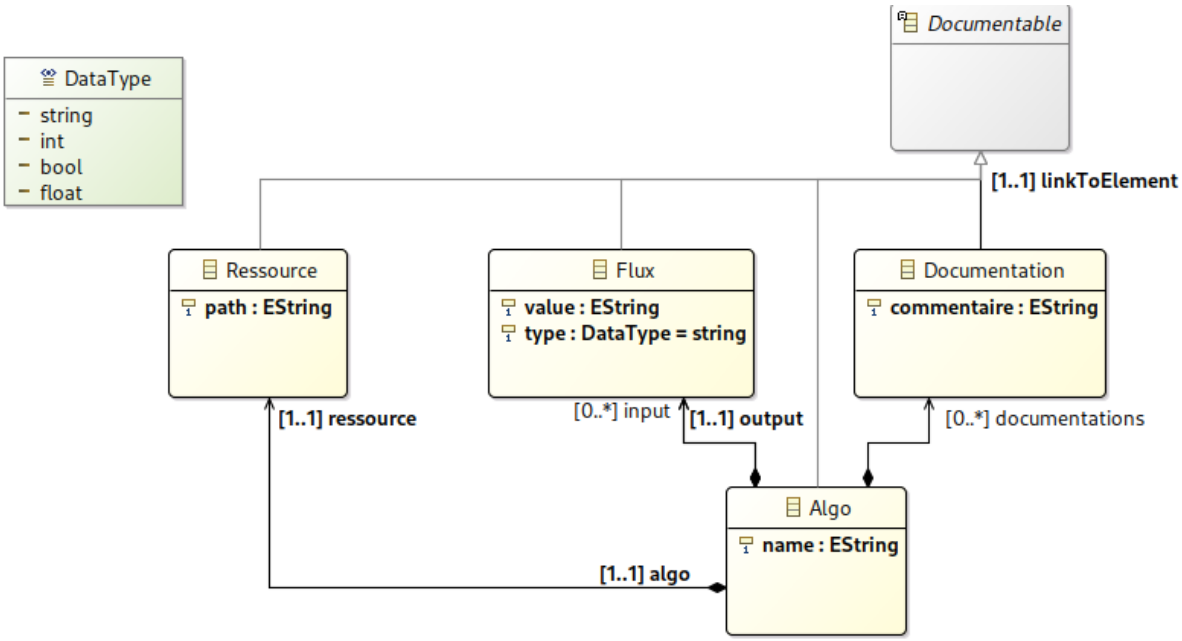

FIGURE 2 – Diagramme de classe du métamodèle Algo

oups j'arrive Un algorithme a un nom, une ressource définie par un path, qui peut-être un fichier **Python** ou **Matlab**. Si l'utilisateur souhaite écrire un algorithme à l'aide du langage dédié, il doit auparavant effectuer la transformation Acceleo Expression2Python (section 7.1) puis donner le path du fichier python généré.

Dans un fichier Python, l'utilisateur peut choisir de mettre autant de fonctions qu'il le souhaite mais la fonction utilisée pour le calcul final doit avoir le nom "calcul" et s'il s'agit d'une contrainte à vérifier sur la colonne, elle doit avoir le nom "check". Dans un fichier Matlab, l'utilisateur doit avoir une unique fonction et ce sera celle-ci qui sera évaluée.

Un algo possède aussi des entrées et une sortie, qui sont représentées par des flux, qui sont à nouveau d'un certain type DataType et qui ont une certaine valeur. Tous ces éléments sont documentables **(F2.4)**.

4.3 Métamodèle Expression

Les expressions sont les algorithmes directement implémentés dans le projet :

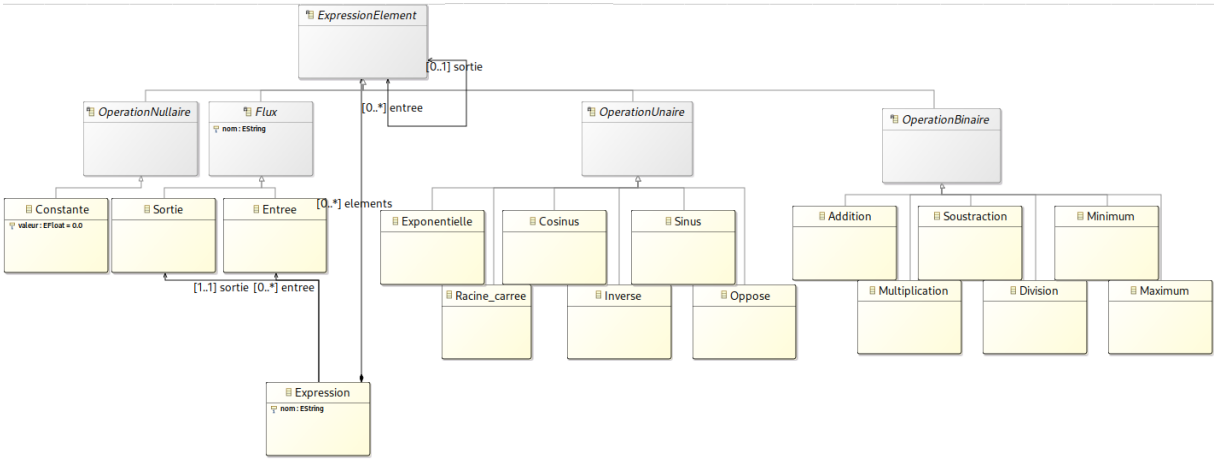

FIGURE 3 – Diagramme de classe du métamodèle Expression

Une expression possède un nom, des entrées et une sortie. Une expression peut représenter plusieurs objets, cela correspond soit à une opération, (OperationNullaire, OperationUnaire, ...), soit à un flux de données, c'est-à-dire les entrées de l'expression (les valeurs des colonnes en entrée) ou les sorties (les valeurs de la colonne en sortie). Les flux possèdent des noms étant donné que l'on ne connait pas leurs valeurs, c'est nos "variables" des modèles de *expression* (très utile pour la transformation Acceleo 7.1).

Notons juste que nous avons décidé d'utiliser des classes abstraites pour *OperationUnaire* et *OperationBinaire*, que *Addition*, *Soustraction*, etc. implémentent étant donné qu'elles possèdent la même forme (respectivement un argument et deux arguments). *OperationNullaire* n'a que *Constante* qui possède un attribut **valeur** *Float*, mais pour un souci de logique et cohérence, nous avons quand même créé la classe abstraite. Nous aurions par exemple pu lui rajouter des constantes fixes comme *Pi*.

5 Contraintes sur les modèles

5.1 SchemaTable

Mis à part les contraintes sur les noms conformes, nous assurons ces différents éléments à l'aide de contraintes OCL :

- Un algorithme définissant une contrainte doit avoir une entrée unique (qui correspond à un élément de la colonne sur laquelle la contrainte est définie)
- Un algorithme définissant une contrainte doit renvoyer un booléen
- Un algorithme définissant une contrainte doit avoir une entrée du même type que la colonne en entrée
- Le type des colonnes doivent être les mêmes que les entrées de l'algo dans AppelAlgo
- Tous les types d'entrées d'AppelAlgo doivent être identiques (ainsi que les types des colonnes dans input)
- Le type de sortie de l'algorithme doit être identique au type de la colonne sur laquelle l'algorithme est assigné

— Lors de l'importation d'une autre table, il existe bien une colonne de l'ID défini dans la table définie

Certaines de ces contraintes ne sont pas optimales, par exemple la cinquième de la liste ci-dessus restreint l'ingénieur à mettre des colonnes toutes de mêmes types dans les inputs d'AppelAlgo, alors qu'il est possible qu'un algorithme ait des entrées différentes.

5.2 Algo

Mis à part les contraintes sur les noms conformes, nous assurons ces différents éléments à l'aide de contraintes OCL :

— S'il y a une documentation, le commentaire ne doit pas être vide
— Le chemin d'accès à la ressource doit être conforme, entre autres, il doit finir par **.py** ou **.m**.

5.3 Expression

Mis à part les contraintes sur les noms conformes, nous assurons ces différents éléments à l'aide de contraintes OCL :

— Le nombre d'entrées est correcte (0 pour OperationNullaire, 1 pour OperationUnaire, 2 pour OperationBinaire)
— Une sortie ne peut pas avoir de sortie
— Une entrée ne peut pas avoir d'entrée
— Une sortie ne peut pas avoir plusieurs entrées

Pour ces contraintes, nous aurions pu les modéliser directement dans le métamodèle en donnant une multiplicité précise aux *entree* et *sortie*, mais il est pratique les garder attributs de *ExpressionElement*, car nous conservons le *EOpposite* lors des connexions avec les autres *ExpressionElement*.

6 Interfaces de spécification de modèles

Pour spécifier des modèles conformes à chacun nos méta-modèles de façon plus ergonomique et adaptée qu'avec l'éditeur arborescent, nous utilisons des syntaxes graphiques concrètes avec Sirius.

6.1 Expressions

Les expressions sont visualisées sous la forme d'un diagramme (figure 4). Les ExpressionElement sont représentés par des nœuds rectangulaires reliés entre eux par des flèches modélisant leurs relations entrée/sortie.

Pour faciliter la composition d'expressions, nous avons créé une palette d'outils (figure 5) :

— Connecter : permet de définir une relation entrée/sortie entre deux ExpressionElements, représentée par une flèche étiquetée de la position de l'élément dans la liste des entrées du suivant. Cette étiquette n'est pas un attribut, elle est obtenue avec la fonction indexOf.
— Créer entrée : définit une entrée de l'expression, représentée par un nœud étiqueté du nom de l'entrée et précédé de sa position dans les entrées de l'expression et d'une flèche.
— Créer sortie : définit une sortie de l'expression, représentée par un nœud étiqueté du nom de la sortie suivi d'une flèche.
— Sélectionner sortie : pendant l'édition, l'utilisateur peut se retrouver avec plusieurs nœuds de sortie. Cet outil permet de sélectionner celle qui est effectivement la sortie de l'expression.
— Créer constante : définit une constante représentée par un nœud étiqueté de sa valeur.
— Créer <opération> : définit une opération, représentée par un nœud étiqueté de son nom.

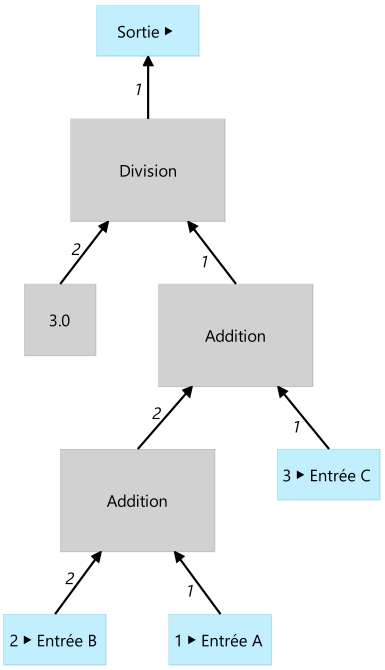

FIGURE 4 – Exemple d'Expression spécifiée avec Sirius

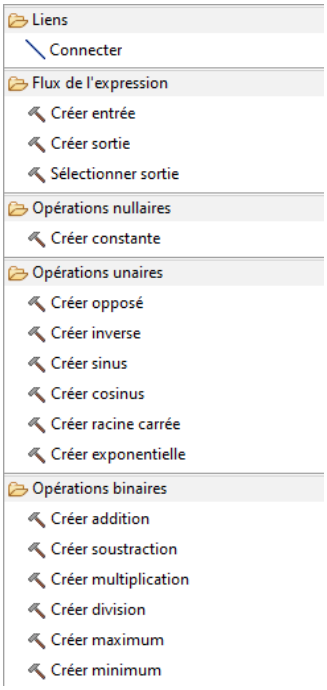

FIGURE 5 – Palette d'outils pour la spécification d'Expressions

6.2 Algorithmes

Les algorithmes sont, eux aussi, visualisés par un diagramme, et sont édités par une palette d'outils :

- Créer ressource : définit une ressource, représentée par un nœud étiqueté de son chemin.
- Créer entrée / sortie : définit une entrée/sortie de façon semblable aux Expressions. L'étiquette du nœud fait également apparaître le type du flux.
- Spécifier un <type> : change le type d'un flux en cliquant dessus.
- Créer / Lier une doc : calque optionnel permettant d'écrire un commentaire et de le lier à un élément de l'algorithme.

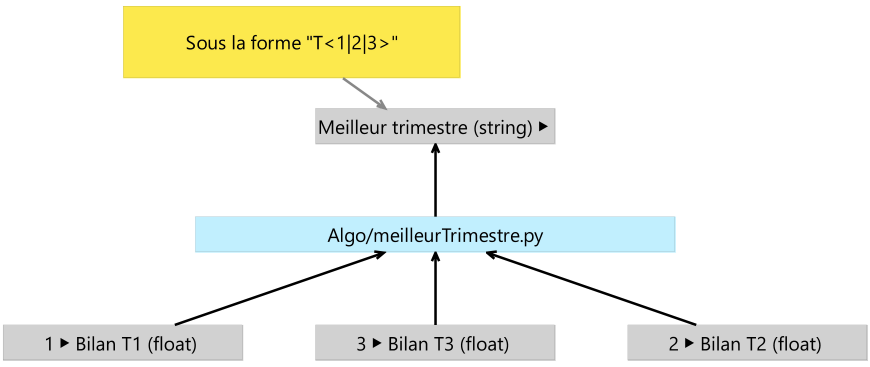

FIGURE 6 – Algorithme `Meilleur trimestre` spécifié avec Sirius

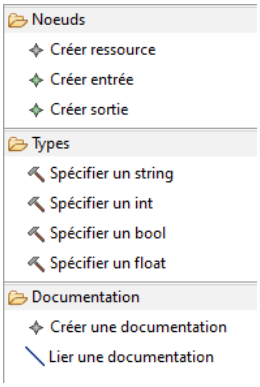

FIGURE 7 – Palette d'outils pour la spécification d'Algos

6.3 Schémas de table

Les schémas de table sont représentés par une table avec en lignes, les colonnes du schéma et leur contenu ; et en colonnes leurs attributs.

La première ligne de la table est réservée au nommage du tableau lui-même. Ensuite, chaque ligne représente une colonne du tableau. La colonne des identifiants de ligne est bleue et son étiquette à la mention "[i]". Le contenu d'une colonne (ContenuColonne) : un appel à une autre table (ImportTable) ou une dérivation (AppelAlgo) ; est représenté par une sous-ligne repliable.

La saisie du texte se fait directement depuis la table, la sélection de l'énumération "type" se fait par un menu déroulant, et la sélection des modèles Algo se fait facilement par un pop-up.

Pour composer la table, nous avons ajouté des outils contextuels accessibles depuis un clic droit sur une ligne :

— Nouvelle colonne : créé une nouvelle colonne, donc une nouvelle ligne dans la table.
— Appeler un algorithme / Importer depuis une autre table : ajoute une sous-ligne "Import" ou "Algorithme". Accessible seulement depuis les lignes "Colonne".
— Désigner comme colonne 'identifiant' : désigne la colonne comme colonne des identifiants de la table. Accessible seulement depuis les lignes "Colonne".

	Nom	Identifiant	Type	Contrainte(s)	Algorithme: Algorithme	Algorithme: Entrée(s)	Import: Table	Import: Identifiant
Tableau	detail							
Colonne 1 [i]	Annee	detail.annee	int	[]				
Colonne 2	Recettes T1	detail.recettes-t1	float	[Positif]				
Import							Tableau recettes	recettes.t1
Colonne 3	Recettes T2	detail.recettes-t2	float	[Positif]				
Import							Tableau recettes	recettes.t2
Colonne 4	Recettes T3	detail.recettes-t3	float	[Positif]				
Import							Tableau recettes	recettes.t3
Colonne 5	Depenses T1	detail.depenses-t1	float	[Positif]				
Import							Tableau depenses	depenses.t1
Colonne 6	Depenses T2	detail.depenses-t2	float	[Positif]				
Import							Tableau depenses	depenses.t2
Colonne 7	Depenses T3	detail.depenses-t3	float	[Positif]				
Import							Tableau depenses	depenses.t3
Colonne 8	Bilan T1	detail.bilan-t1	float	[]				
Algorithme					SoustractionBinaire	[Recettes T1, Depenses T1]		
Colonne 9	Bilan T2	detail.bilan-t2	float	[]				
Algorithme					SoustractionBinaire	[Recettes T2, Depenses T2]		
Colonne 10	Bilan T3	detail.bilan-t3	float	[]				
Algorithme					SoustractionBinaire	[Recettes T3, Depenses T3]		
Colonne 11	Bilan	detail.bilan	float	[]				
Algorithme					SommeTernaire_m	[Bilan T1, Bilan T2, Bilan T3]		
Colonne 12	Meilleur trimestre	detail.meilleur	string	[]				
Algorithme					MeilleurTrimestre	[Bilan T1, Bilan T2, Bilan T3]		
Colonne 13	Rapport recettes/de	detail.rapport	float	[Positif]				
Algorithme					RapportRecettesSurDepenses	[Recettes T1, Recettes T2, Recette		

FIGURE 8 – Schéma de table `détail` spécifié avec Sirius

7 Transformations Acceleo

Pour faire les calculs sur les CSV entrés par ***l'end-user target*** (utilisateur final), nous avons opté pour le langage de calcul *Python*. Ainsi, le noyau de notre projet s'exprime dans ce même langage et nous force donc à faire des transformations **Acceleo** faisant le lien entre nos *métamodèles/modèles* et l'aspect concret et pratique du calcul.

7.1 Projet Expression2Python

De cette transformation découlent les fonctionnalités F1.3, F1.5, F3.4, F3.5 et F3.6

En effet, l'objectif est de pouvoir utiliser en **Python** les calculs sur les colonnes que ***l'end user*** (utilisateur) a définies grâce à son interface dédiée dans *Sirius*. Comme vu précédemment, *expression* contient des fonctions simples de **Python** comme : *sinus, cosinus, addition,etc*.

La seule difficulté que nous avons rencontrée dans cette implémentation, est que l'on ne peut pas faire un parcours des objets (assez classique en Acceleo) et écrire le calcul en brut. En effet, on peut imaginer que l'ordre de calcul soit autre que l'ordre dans lequel l'utilisateur a défini les objets. Ainsi, il nous semblait complexe d'associer correctement les sorties aux entrées appropriées pour chaque fonction en Python.

La solution que nous avons trouvée est de faire une fonction récursive qui part de l'unique sortie du programme et *"parse"* au fur et à mesure de manière descendante les entrées des calculs. Les flux en entrée, autrement dit les valeurs des colonnes sur lesquelles est appelé le script, sont remplacés par leurs noms qui apparaissent également dans la signature de la fonction. Ce sont les variables d'entrée de notre fonction.

Nous avons créé un modèle permettant de tester cette transformation en utilisant absolument toutes les fonctions possibles : *TestTout.xmi*.

```
    return cosinus(expo(racine(maximum(soustraction(minimum(sinus(multiplication(inverse(oppose(division(addition(A,B),2.0))),4.0)),
10.9),19.0),0.0))))
```

FIGURE 9 – Résultat de *TestTout.xmi*

Toutes les fonctions disponibles sont codées en brut dans le **Python** généré. Une amélioration possible de cette transformation serait d'implémenter juste les fonctions utilisées dans le modèle. Ici le nombre de fonctions disponibles est faible, mais on peut imaginer en rajouter pour une future mise à jour. Il y en aurait beaucoup, et cette fonctionnalité augmenterait grandement la lisibilité du code.

7.2 Génération d'une librairie Python

Couvre les fonctionnalités contenues dans F4

Une fois que les .xmi sont créés et vérifiés grâce aux contraintes OCL, on peut générer une librairie en Python à l'aide d'Acceleo. Si l'utilisateur n'utilise que des scripts Python, il est plus intéressant pour lui de lancer le projet Acceleo `SchemaTable2Python`, s'il utilise au moins un script Matlab, il doit utiliser `STWithMatlab2Python`. Dans les 2 cas, il doit lancer le fichier `ToPython.mtl` et renseigner le bon modèle dans la configuration.

Ainsi, l'utilisateur obtiendra un fichier Python par schema de table. Dans ces fichiers, il y aura des opérations pour :

- Charger un csv dans le schéma de Table (fonction `load`)
- Vérifier que la colonneLigne a des identifiants différents à chaque ligne et que les contraintes sur les colonnes sont vérifiées (fonction `checkAll`)
- Importer les colonnes des autres tables et calculer les colonnes qui sont issues d'un algorithme (fonction `calcAll`)
- Exporter la table en csv (fonction `export`)

Pour la construction de cette librairie, nous avons utilisé la programmation objet et nous avons été confrontés à quelques difficultés. Dans un premier temps, lors de l'import d'une table, il est nécessaire de pouvoir retrouver l'instance de la classe de la bonne table. Pour cela, nous avons utilisé le patron de conception Singleton, qui permet d'avoir une seule instance par classe et de pouvoir récupérer l'instance à partir du nom de la classe. Il est également nécessaire que cette classe ait été instanciée avant, mais cela est géré au niveau de l'interface utilisateur.

Une autre difficulté réside dans la fonction "CalcAll" car il est nécessaire que la fonction fasse les calculs dans le bon ordre. Pour cela, nous nous assurons d'importer dans un premier temps toutes les colonnes nécessaires, puis nous calculons les colonnes dans l'ordre d'apparitions. Il est donc nécessaire que l'end user place dans le bon ordre les colonnes si une colonne est calculée à partir d'une autre colonne qui a été calculée. Sinon l'end user target devra cliquer deux fois sur CalcAll. Nous aurions aussi pu utiliser un attribut entier sur les colonnes de type "AppelAlgo", qui définit l'ordre d'exécution des algorithmes.

Nous avons aussi rencontré des difficultés concernant le chemin vers une ressource. En effet, nous avions 2 possibilités : mettre le chemin absolu ou le chemin relatif. L'inconvénient de mettre le chemin relatif est que les fichiers Python générés doivent se situer au même endroit que le modèle algo (ou que l'ingénieur écrive le chemin par rapport à l'emplacement des fichiers Python générés, plutôt que par rapport à l'emplacement du modèle). L'inconvénient du chemin absolu est que l'end-user et l'end-user target doivent avoir le même chemin absolu vers la ressource, ce qui ne nous semblait pas adapté à la problématique, car ces deux utilisateurs doivent pouvoir travailler sur des machines différentes. Nous avons donc fait un compromis : l'end-user doit écrire le chemin absolu de la ressource en prenant comme racine le workspace, l'end-user target inscrit le chemin vers ce workspace dans le fichier Config.py, et nous pouvons ensuite concaténer les deux chemins.

8 Interface Utilisateur

L'interface utilisateur se présente sous forme d'un script python nommé *Interface_Utilisateur.py* à compiler. L'objectif ici est de récupérer les fonctions des classes résultantes des transformations réalisées par l'ingénieur (avec *SchemaTable2Python* ou *STWithMatlab2Python*) et de les rendre disponible à l'utilisateur pour qu'il puisse les appliquer sur ses propres tableaux **.csv**. Nous avons développé une interface en python, car nos transformations sont faites en python. Au départ, nous voulions coder l'interface en html, mais nous nous sommes confrontés à l'utilisation de fonctions python depuis html. Nous avions pensé à utiliser Flask, mais nous avons décidé de concevoir une interface python, car il est plus facile à implémenter une telle interface.

8.1 Côté End-User

Du côté de l'ingénieur, il est nécessaire d'apporter quelques modifications pour que le script fonctionne correctement.

```
9  ################################## Modifications End-user ##################################
10
11 from depenses import depenses
12 from recettes import recettes
13 from detail import detail
14 from resume import resume
15
16 liste_instances = [depenses(), recettes(), detail(), resume()] #liste des instances à créer
17
18 ############################################################################################
```

FIGURE 10 – Modifications à apporter dans le script python

Sur la figure précédente, on peut voir un ensemble d'import et une liste contenant des objets. L'ingénieur doit :

1. écrire à la main dans le script les imports à effectuer par rapport aux transformations de SchemaTable qu'il a précédemment effectué
2. écrire dans la *liste_instances* les différents objets à créer dans l'ordre des dépendances des tables : *depenses()* et *recettes()* qui ne dépendent de rien, puis *detail()* qui dépend de *depenses()* et *recettes()*, etc...
3. placer ce script au même endroit que les scripts résultants des transformations qu'il a créées

Une fois cela fait, l'interface utilisateur est prête à l'utilisation.

8.2 Côté Utilisateur

Du côté utilisateur, il faut dans un premier qu'il se munisse de ses fichiers **.csv** de la bonne forme (imposée par l'ingénieur) puis il doit exécuter le script *Interface_Utilisateur.py*. Nous avons imaginé une interface user-friendly, avec des boutons et une zone de dialogue pour guider l'utilisateur lors des calculs qu'il souhaite réaliser. Lors de l'exécution du script, une fenêtre s'ouvre :

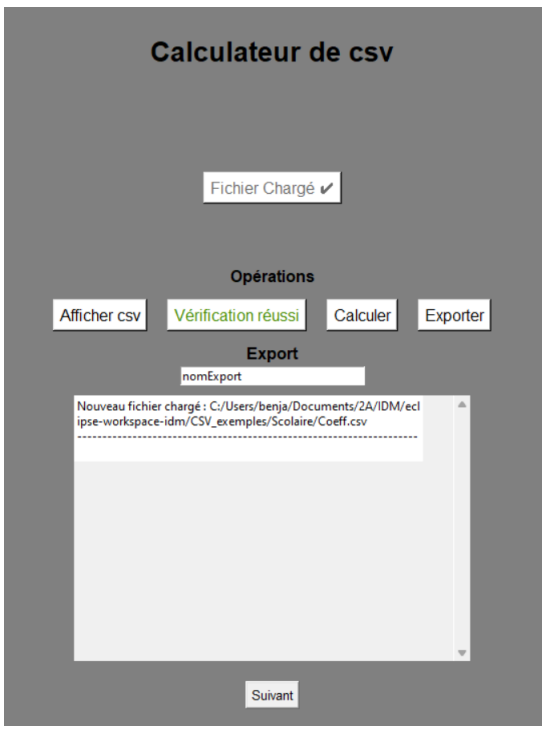

FIGURE 11 – Exemple d'affichage de l'interface Utilisateur

Chaque fenêtre sert à travailler uniquement sur un fichier **.csv** à la fois, pour charger un fichier suivant, il faut cliquer sur le bouton *Suivant* et une nouvelle fenêtre apparaît. L'utilisateur doit charger ses fichiers dans l'ordre des dépendances des tables, le même ordre que dans la liste *liste_instances*. À chaque fois qu'il charge un fichier **.csv**, il peut :

— Charger un fichier **.csv**
— Afficher le fichier
— Vérifier les éventuelles contraintes
— Faire les calculs nécessaires
— Exporter le fichier **.csv**

Pour passer au chargement du prochain **.csv**, il clique sur *Suivant* et une nouvelle interface apparaît. Pour l'export d'un fichier, une *Entry* est disponible pour donner un nom au fichier **.csv** à exporter, puis l'utilisateur clique sur *Exporter* pour choisir un répertoire où le fichier sera enregistré. Une boite de dialogue permet l'affichage de certaines informations lorsque l'utilisateur utilise l'interface.

Pour le scénario fourni, l'exécution est la suivante :

1. Exécuter le script python
2. Charger le fichier **.csv** correspondant aux **dépenses**, puis cliquer sur Suivant
3. Charger le fichier **.csv** correspondant aux **recettes**, puis cliquer sur Suivant
4. Charger le fichier **.csv** correspondant aux **détails**, vérifier le fichier, calculer le fichier puis cliquer sur Suivant
5. Charger le fichier **.csv** correspondant aux **résumé**, vérifier le fichier, calculer le fichier puis l'exporter en lui donnant un nom au préalable dans la zone d'entrée
6. Cliquer sur Quitter

Nous pouvons noter que l'utilisateur peut visualiser le **.csv** couramment importé, le vérifier, le calculer et l'exporter à n'importe quelle étape de l'exécution. Si aucune vérification ne doit être faite sur un tableau, alors la vérification fonctionne, de même pour les calculs.

9 Résumé de la couverture des fonctionnalités

Nous estimons avoir mis au point les fonctionnalités listées dans le tableau suivant :

Fonctionnalité	Complètement implémentée	Partiellement implémentée	Non implémentée
F1	✓		
F1.1	✓		
F1.2	✓		
F1.3	✓		
F1.4	✓		
F1.5	✓		
F2		✓	
F2.1	✓		
F2.2	✓		
F2.3			✓
F2.4	✓		
F2.5	✓		
F2.6		✓	
F3	✓		
F3.1	✓		
F3.2	✓		
F3.3	✓		
F3.4	✓		
F3.5	✓		
F3.6	✓		
F4	✓		
F4.1	✓		
F4.2	✓		
F4.3	✓		
F4.4	✓		
F4.5	✓		
F4.6	✓		
F5		✓	
F6	✓		

TABLE 1 – État d'implémentation des fonctionnalités

Dans ce projet, nous n'avons pas pu mettre au point la fonctionnalité suivante :
— F2.3 : L'utilisateur ne peut pas fixer une entrée du programme.
Les fonctionnalités partiellement implémentées sont les suivantes :
— F5 : Comme expliqué dans la section *Interface Utilisateur*, l'ingénieur est contraint d'écrire à la main quelques lignes de code pour les imports et les instances à créer. Donc cette étape n'est pas complétement automatisée.
— F2.6 : Les algorithmes sont clairement identifiés et partageables, mais ne sont pas actuellement regroupés dans des catalogues. Il est néanmoins possible de le faire.

10 Pistes d'amélioration

— Créer un type de ContenuColonne qui indique si une colonne doit être remplie à la main pour signaler une erreur si l'utilisateur ne remplit pas la colonne requise
— Permettre l'édition de données par l'end-user target via l'interface utilisateur
— Pour l'interface utilisateur et pour le problème des lignes que l'ingénieur doit écrire manuellement pour le bon fonctionnement de l'interface, nous avons pensé à créer une transformation acceleo pour automatiser les imports et la liste d'instance à créer
— Donner la possibilité à l'end-user de modifier l'ordre des colonnes de son schéma de table via l'interface Sirius
— Permettre au langage dédié de manipuler des objets autre que des valeurs numériques
— Permettre à l'end-user d'importer des algorithmes dans un *SchemaTable* avec des entrées de types différents (impossible dû aux contraintes ocl)

11 Conclusion

N'étant visiblement pas des utilisateurs aguerris d'éclipse, une des grandes difficultés de ce projet a été les erreurs récurrentes et problèmes hors de notre compréhension qui ont été, pour la plupart, résolus de manière autonome par éclipse lui-même.

Nous avons également eu des difficultés concernant la compréhension du sujet, notamment en ce qui concerne les schémas de table. Par exemple, nous n'avions pas tout de suite compris qu'ils ne devaient pas contenir de données. Cela a été éclairci grâce aux séances de projet.

Malgré ces difficultés, nous avons réussi à rendre un projet qui nous semble complet par rapport à notre interprétation des fonctionnalités demandées. Pour conclure sur notre ressenti, nous trouvons que c'est une application fonctionnelle et utile. Nous avons aimé travailler dessus et sommes satisfaits du travail final rendu.