Rapport Projet IDM

Jérémy Angulo - Loïc Blanc - Lucas Bolbènes - Ethan Boswell - Margaux Garric

2SN L1 - L2

Département Sciences du Numérique - SIL
2023-2024

1. Table des matières

1. Table des figures

2. Introduction

Ce rapport présente notre projet d'IDM, axé sur le développement d'une interface graphique Java et l'intégration d'algorithmes variés pour le traitement et l'analyse de données. L'objectif principal de ce projet est de concevoir un système flexible et performant, capable de s'adapter à différents besoins d'analyse de données, tout en offrant une expérience utilisateur intuitive et efficace. Pour une meilleure compréhension du rapport, nous allons définir quelques termes clés utilisés tout au long de ce document.

2.1 Définitions

- **Algorithme** : Un algorithme dans le contexte de notre projet définit les entrées et les sorties ainsi qu'un chemin vers une ressource. Cette ressource spécifie la fonction utilisée par l'algorithme pour traiter les données.
- **Schéma de table** : Un schéma de table est une structure organisée comprenant plusieurs tables. Chaque table est constituée de plusieurs colonnes, qui définissent les différents types de données que nous pouvons manipuler.
- **Librairie** : Une librairie, dans notre cadre, est un ensemble d'algorithmes regroupés. Chaque librairie sert à traiter un type spécifique de données ou à effectuer un ensemble particulier de tâches d'analyse.

Dans ce rapport, nous discuterons des différentes composantes de notre projet, y compris les défis rencontrés et les pistes d'améliorations envisagées pour optimiser davantage notre système.

3. Création d'un schéma de table

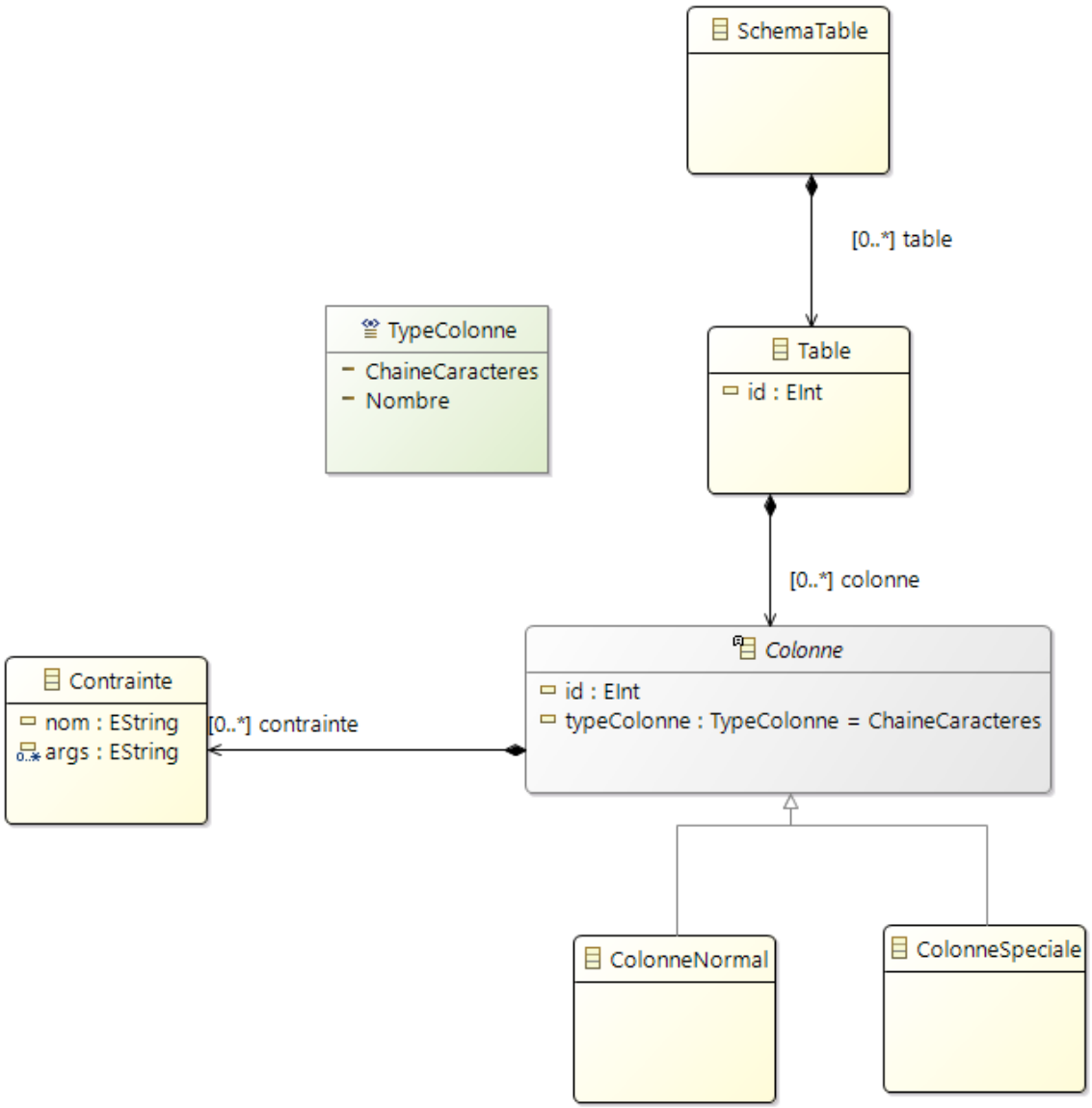

FIGURE 1 – Diagramme du métamodèle schema de table

Nous avons commencé par modéliser un schéma de table. Un schéma de table est composé de différentes tables identifiées par un identifiant. Cette table est elle-même composée de différentes colonnes. Il y a deux types de colonnes : les colonnes normales et les colonnes spéciales. Les colonnes spéciales représentent la colonne comportant l'en-tête de chaque ligne. Une colonne a plusieurs attributs : son identifiant, une référence vers un objet Contrainte et le type de colonne. Nous avons pris la décision de considérer que tous les identifiants seraient des entiers. L'attribut type de colonne a pour but de vérifier l'unicité des types rentrés en donnée de la colonne. En effet, les algorithmes s'appliquent sur des colonnes entières et il est donc nécessaire que la colonne ait le même type de donnée.

Chaque instance de Colonne peut posséder plusieurs contraintes optionnelles, définies par un nom et une liste d'arguments. Lorsqu'un fichier CSV est importé dans notre application, ces attributs sont parsés et la méthode correspondante, est exécutée pour vérifier une contrainte quelconque. Nous vérifions au même moment le bon typage des colonnes et leur bonne taille.

Ce metamodèle est la version finale, mais au début, nous avions proposé d'ajouter un objet cellule à ce métamodèle. Cependant, nous avons abandonné l'idée, car nous nous sommes rendu compte que l'utilisateur n°1 qui créait le schéma de table ne gérait pas de données.

Nous avons considéré que l'utilisateur 1 était en capacité d'utiliser les outils Eclipse et pouvait donc créer le schéma de table directement à partir Sample Reflective Ecore Model Editor. Cela permet de générer de façon arborescente les modèles xmi.

4. Créations d'un outil graphique pour pouvoir créer une ressource

Pour représenter graphiquement les ressources, nous avons décidé de créer un système de création par bloc sur le même principe qu'à la figure 2 du sujet.

4.1 Réalisation du métamodèle représentant les ressources graphiques

Nous avions décidé de créer un métamodèle ressourceGraphique qui permettra de modéliser les ressourcesGraphiques.

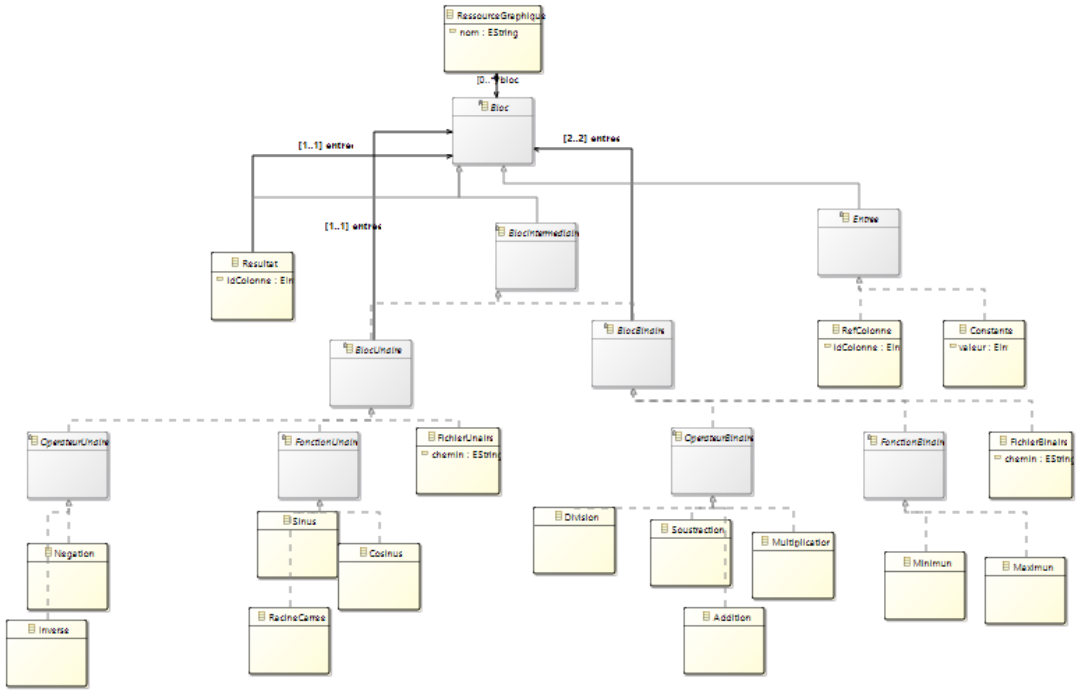

FIGURE 2 – Diagramme du métamodèle RessourceGraphique

Une ressource graphique est composée de blocs. La difficulté était de pouvoir gérer les différents nombres d'entrées et de sorties. Nous avons différencié 3 cas :

- les "Entrées " de la ressource qui ont 1 sortie, mais pas d'entrée dans le bloc
- les "Blocs intermédiaires" qui ont un nombre d'entrées et de sorties
- les "Résultats " qui ont une entrée, mais pas de sortie.

Nous avons donc créé 3 classes héritant de la classe Bloc.

La classe Entrée se redivise elle-même en deux classes : "RefColonne " qui est une référence vers une colonne et "Constante" qui permet d'intégrer les valeurs constantes dans l'algorithme.

Pour les blocs intermédiaires, nous avons décidé de représenter seulement les fonctions prenant 1 ou 2 entrées et nous avons divisé la classe en deux : BlocBinaire et BlocUnaire. Cette décision a été prise, car nous avons jugé rares les fonctions qui avaient 3 entrées et ne pouvaient pas utiliser de propriété d'associativité pour représenter l'algorithme avec seulement des BlocsBinaires et des BlocUnaires. Il est toutefois possible d'améliorer en créant une autre classe BlocTernaire ou BlocQuaternaire par exemple si cela répond à un besoin.

Nous nous sommes aussi posés la question de limiter le nombre de sorties des blocs intermédiaires à une et d'un point de vue mathématique, réutiliser deux fois le résultat d'un calcul intermédiaire n'est pas un problème et d'un point de vue implémentation, cela facilitait grandement. En effet, si nous avions créé un objet Sortie, nous aurions dû gérer la concordance entre la sortie d'un bloc et l'entrée de l'autre. Par exemple, vérifier que si le bloc A a pour sortie le bloc B, le bloc B a pour sortie le bloc A. Pour cela, il aurait fallu créer des objets lien qui prend en paramètre une entrée et une sortie. Mais tous les types de bloc n'auraient pas eu le même nombre d'attributs "entrée" relié au lien. Par exemple, la cardinalité de l'attribut entrée qui aurait référencé un lien aurait été de 2 pour BlocBinaire, mais de 1 pour BlocUnaire. Il aurait donc fallu faire plein de sous-classes de la classe Lien pour représenter tous les types de Lien entre les différents objets. C'est pourquoi, nous avons décidé de ne gérer cela qu'avec les entrées.

Pour revenir à la description du modèle, le bloc Unaire (respectivement Binaire) est divisé en OperateurUnaire (resp OperateurBinaire), FonctionUnaire (resp fonctionBinaire) et FichierUnaire(resp FichierBinaire) Nous avions implémenté un certain nombre de blocs opérateur et fonction, mais il est possible d'en rajouter dans une application plus complète en rajoutant de nouvelle classe héritant d'opérateur/fonction unaire/binaire. Les blocs FichierUnaire et FichierBinaire permettent de rajouter dans la ressource un script venant de l'extérieur. Pour cela, il faut réaliser un script en python suivant le même principe que les blocs de base codés en Python (voir sections script python). Ce script devra être mis dans le dossier fonctions du fr.l127.fonctions. Lors de l'utilisation du bloc fichierUnaire/Binaire, il faut mettre en chemin le nom de la fonction python à appeler.

4.2 Réalisation de l'outil avec Sirius

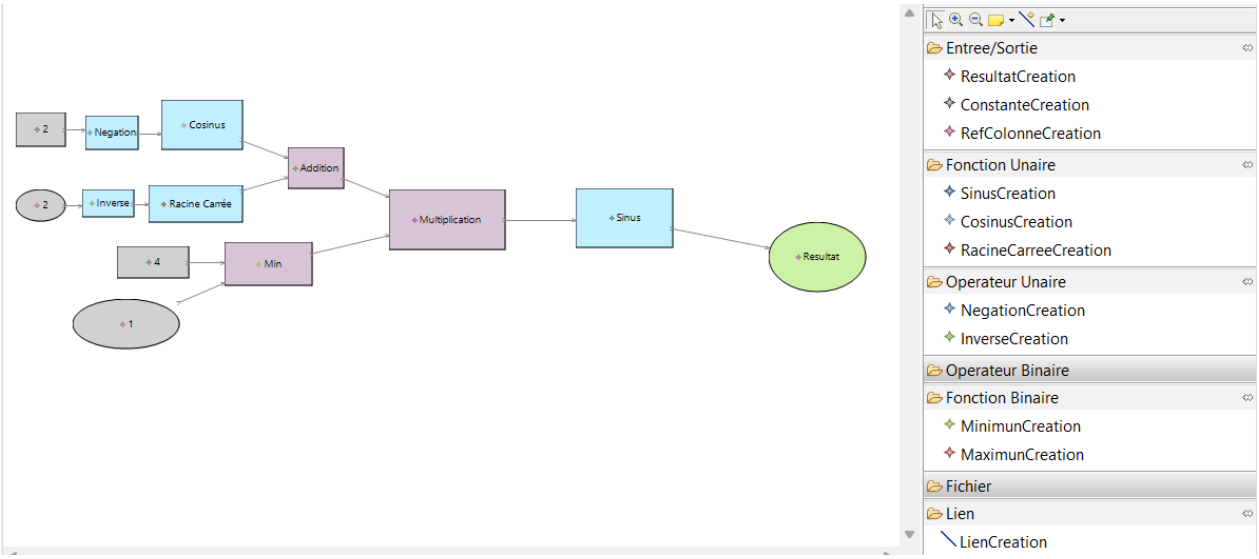

FIGURE 3 – Outil Sirius pour représenter les ressources graphiques

Pour représenter, l'application, nous avions eu décidé d'utiliser Sirius. Dans le fichier ressourceGraphique.odesign, nous avons déterminé comment était représenté les différents objets du modèle RessourceGraphique.

Pour un aspect pratique, nous avons mis en place un code couleur/forme selon les objets concernés

- Bloc Binaire : rectangle violet clair
- Bloc Unaire : rectangle bleu clair
- Resultat : elipse vert
- Entree : gris clair et ellipse pour une référence à une colonne et un rectangle pour une colonne.

Nous avons également développé les outils nécessaires pour concevoir graphiquement les différents objets, facilitant ainsi la création des modèles. Ces outils ont été organisés par catégories, permettant à l'utilisateur de se repérer. Ces catégories sont :

- Entrée/Sortie
- FonctionUnaire
- FonctionBinaire
- OperateurBinaire
- OperateurUnaire

- Fichier
- Lien

Nous avons séparé les opérateurs/fonctions unaire et binaire pour clarifier si le bloc avait besoin d'un ou deux entrées. À part pour les blocs fichiers, mais pour ces blocs, la fonction vient de l'utilisateur donc il doit savoir s'il faut un ou deux entrées.

Pour que l'utilisateur puisse utiliser l'outil graphique, il faut dans l'Eclipse de déploiement, choisir comme viewPoint le viewPoint qui a été créé par le odesign nommé "ressourceGraphique" et ajouter dans le projet un modèle vide ou à compléter au format .ressourceGraphique.

Lors de la création d'une ressource graphique, il est important de le faire valider à la fin. En effet, certaines choses sont possible avec l'outil, mais bloqué par le métamodèle. Par exemple, sur le graphique, il est possible de relier un bloc binaire à un nombre quelconque de bloc et donc d'avoir un nombre quelconque d'entrée. Ceci est bloqué par la validation, car il attend 2 blocs. Pour créer un Lien, il faut d'abord cliquer sur le bloc de destination du lien puis après le bloc source du lien.

Un Algorithme est constitué d'une ressource ainsi que d'entrées et de sorties. Comme une ressource graphique n'est qu'un attribut de Algorithme, les ressources générées graphiquement peuvent être exploitées par plusieurs algorithmes. Ainsi, une ressource graphique peut correspondre à plusieurs entrées/sorties différentes. Les identifiants de colonnes spécifiés dans l'objet RefColonnes ne doivent donc pas correspondre à l'identifiant d'une colonne du modèle, mais l'ordre dans lequel la colonne sera appelée lors de l'utilisation de la ressource. Par conséquent, une ressource utilisant 2 colonnes devrait seulement avoir comme id de colonnes 0 et 1. Exemple : on veut utiliser un même algorithme sur les colonnes 3/4 et 3/10. Si les identifiants spécifiés devaient correspondre à une colonne du modèle, on devrait créer deux algorithmes identiques ayant pour entrées 3/4 et 3/10. Dans notre cas, il suffit de créer un seul algorithme avec comme entrée 0 et 1. Quand l'utilisateur spécifie les entrées, l'algorithme associe automatiquement 3/4 ou 3/10 avec les entrées 0 et 1 de l'algorithme. Nous avons décidé qu'un algorithme pouvait avoir plusieurs sorties. En pratique, cela veut dire qu'il faut retourner non pas une colonne, mais les identifiants de colonnes à sortir. Nous avons donc procédé de manière similaire aux entrées, en imposant un ordre dans les sorties pour savoir exactement quelle sortie correspondait à quelle colonne.

4.3 Utilisation des ressources graphiques dans l'application

Nous avons décidé de gérer les algorithmes avec Python. Le modèle représentant la ressource graphique créée avec Sirius doit donc être convertie en Python. Nous avons utilisé pour cela une transformation Accéléo. Pour illustrer cette transformation, nous partirons d'un exemple de ressource graphique.

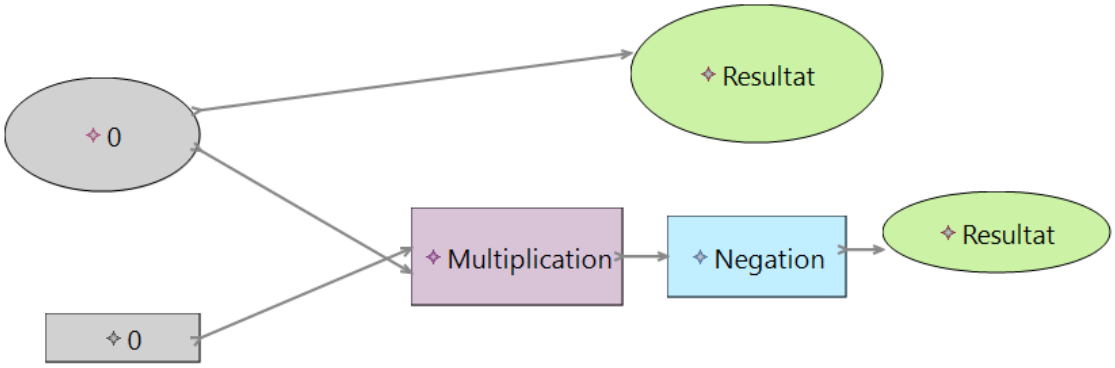

FIGURE 4 – Exemple de Ressource Graphique créé avec Sirius

Pour cet exemple, voici le résultat souhaité avec la transformation :

Listing 1 – Code souhaité d'après la ressource Graphique

```
from fonctions import *

def test2(*colonnes):
        return [colonnes[0], opposite(multiply(colonnes[0], cons(0)))]
```

Dans le code généré par la transformation, le "from fonctions import *" permet de récupérer le package python contenant toutes les fonctions des blocs élémentaires.

Le premier problème consiste à pouvoir appeler les fonctions de manière imbriquée, en considérant que les seules liaisons connues sont les entrées des blocs. La solution adoptée implique de partir du bloc résultat et de remonter progressivement pour déterminer les liens entre les blocs, permettant ainsi l'appel des fonctions appropriées.

À cette fin, nous avons développé des templates de manière récursive. Pour chaque bloc, la template génère le nom de la fonction, puis entre parenthèses, elle appelle la même template sur les entrées du bloc.

Une autre difficulté rencontrée a été la diversité des classes des blocs, avec certains attributs absents dans certaines situations. Pour surmonter cette complexité, nous avons conçu la template appelé de façon récursive avec comme paramètre un objet de type Bloc rendant ainsi la méthode utilisable pour tous les types d'entrées. Dans cette template, le type réel de l'objet est déterminé, et une autre template personnalisée est invoquée en fonction de ce type réel des objets.

Lorsque l'on écrit en Accéléo, les indentations et les espaces écrits sont répercutés sur le fichier généré. Or, Python a besoin d'une certaine indentation précise. Il a donc fallu mettre sur la même ligne de nombreuses lignes de codes pour respecter la syntaxe Python. Cela rend le code de la transformation difficilement lisible.

4.4 Résumé du processus de création de la ressource graphique

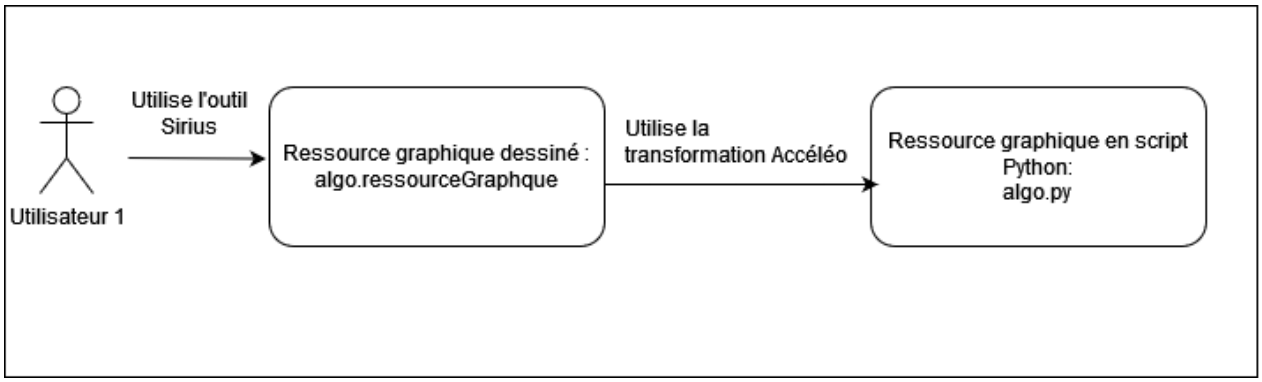

FIGURE 5 – Processus de la création de la ressource graphique

Le processus de création de la ressource graphique a donc deux étapes : la création graphique avec Sirius puis la transformation avec Accéléo en code Python. Cependant, nous avons rencontré des problèmes liés à la gestion entre l'Eclipse de développement et l'Eclipse de déploiement. Le projet Accéléo est en effet dans l'Eclipse de développement et le projet Sirius se trouve dans l'Eclipse de déploiement. Les fichiers .ressourceGraphique ne sont détectables comme respectant un metamodèle que dans l'Eclipse de déploiement et donc ne peuvent pas être détectés par la transformation Accéléo qui est dans l'Eclipse de développement. Pour cela, nous avons essayé plusieurs solutions :

- Générer avec Sirius un fichier Xmi puis le déplacer dans l'éclipse de développement, mais la transformation Accéléo ne donnait rien.
- Générer un Plug In Eclipse pour pouvoir réaliser la transformation depuis l'éclipse de déploiement sans succès non plus.

Il est donc actuellement possible de tester les deux étapes de la chaîne en créant le même algorithme que celui créé graphiquement à la main avec Sample Reflective Ecore Model Editor. Il aurait évidemment trouvé une solution pour que la chaîne puisse fonctionner dans une version finale.

5. Définition des algorithmes par l'utilisateur 1

5.1 Réalisation du métamodèle représentant le catalogue d'algorithme

Nous avions d'abord déterminé la modélisation de l'algorithme :

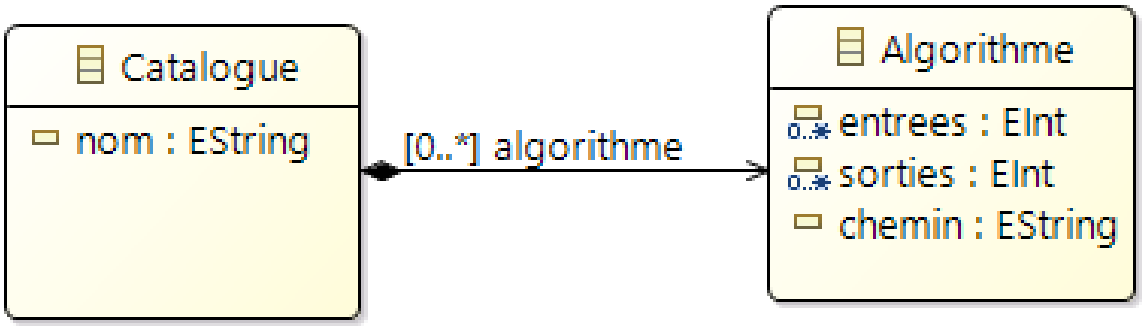

FIGURE 6 – Diagramme du métamodèle catalogue d'algorithme

Ce metamodèle réunissait les algorithmes composés de leur attribut entrees, sorties et chemin sous un objet catalogue défini par son nom.

5.2 Création de l'outil pour définir les algorithmes

Pour permettre à l'utilisateur de définir ses algorithmes, nous avons décidé de réaliser un outil textuel. Pour cela, nous avons utilisé Xtext et défini une syntaxe pour représenter les éléments du métamodèle Catalogue d'algorithme. Cette syntaxe a pour but de faciliter la définition des algorithmes par l'utilisateur. Elle doit permettre à l'utilisateur de renseigner les informations nécessaires à la création d'algorithme qui sont les entrées, les sorties et le chemin vers la ressource. Les fichiers respectant cette syntaxes sont des fichiers d'extension .la pour librairie algorithme.

Listing 2 – Syntaxe Xtext représentant les catalogues d'algorithme

```
Librairie:
'librairies' name=ID '{'
elements+=Algorithme*
'}';

Algorithme :
'algo' name=ID '{'
'entrees' '(' (entrees+=Entree)+ ')'
'sorties' '(' (sorties+=Sortie)+ ')'
```

```
'chemin' chemin=STRING
'}';

Entree :
idColonne = INT;
Sortie :
idColonne = INT;
```

Listing 3 – Exemple de fichier écrit par l'utilisateur pour definir les algorithmes

```
librairies listeAlgo {
algo test{
entrees(1 0)
sorties (2)
chemin "test.py"
}
}
```

Pour améliorer l'expérience utilisateur, nous avons introduit un nouvel attribut, "nom", dans notre syntaxe Xtext. Par ailleurs, nous avons aussi créé deux nouveaux objets, "Entree" et "Sortie", en plus de ceux présents dans le métamodèle d'origine. À posteriori, nous reconnaissons que ces ajouts ont été effectués principalement parce que nous avions l'habitude d'inclure des objets derrière l'opérateur "+=" sans apporter une contribution significative par rapport à l'utilisation d'un simple entier (INT). Cela ne changeait rien pour l'expérience utilisateur donc nous avons décidé de laisser comme cela.

Le principe général de Xtext est de réaliser une transformation Text-To-Model. Cette grammaire a donc permis de générer un nouveau metamodèle nommé "LA.ecore". Voici le métamodèle :

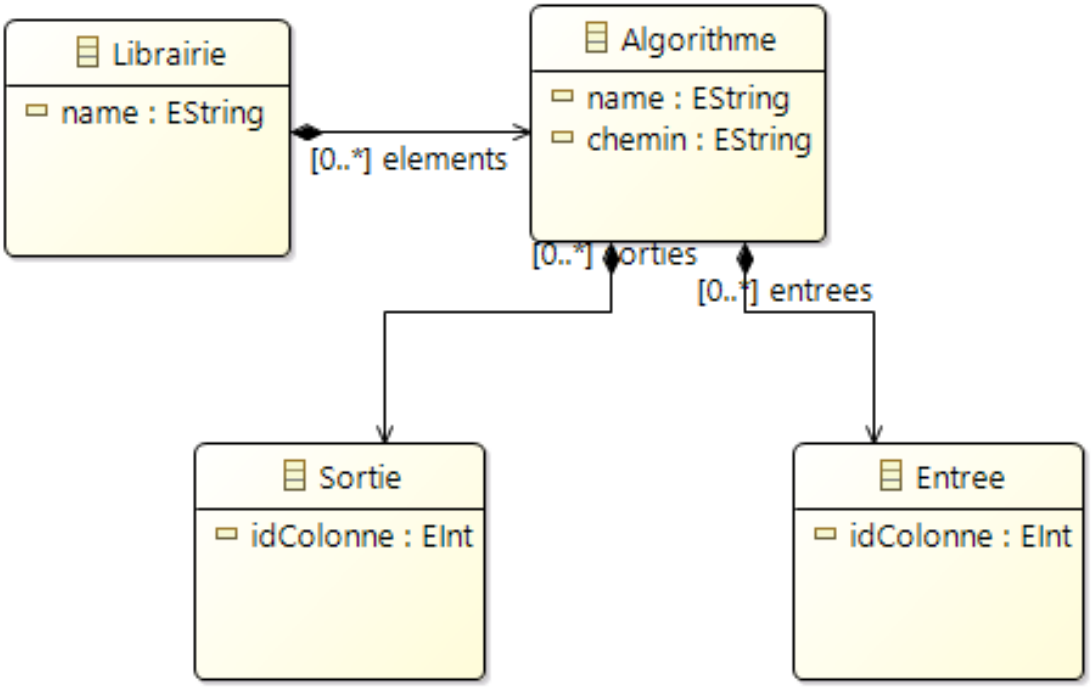

FIGURE 7 – Diagramme du métamodèle librairie d'algorithme généré par la syntaxe Xtext

Le métamodèle correspond au métamodèle "CatalogueAlgorithme", en prenant en compte les modifications mentionnées précédemment, telles que l'ajout des classes "Entree" et "Sortie" ainsi que de l'attribut "name". Pour assurer la cohérence entre le modèle généré par Xtext (LA.ecore) et le modèle de base (CatalogueAlgorithme.ecore), nous avons recouru à une transformation ATL.

Nous avons rencontré des difficultés similaires à celle rencontrée avec les modèles dans Sirius. Le fichier avec l'extension ".la" est uniquement utilisable dans l'Eclipse de déploiement, tandis que la transformation ATL ne fonctionne que dans l'Eclipse de développement. Néanmoins, nous avons constaté que la transformation s'opère correctement à partir d'un fichier .xmi créé dans l'Eclipse de developpement.

Finalement, en raison de contraintes expliquées plus en détail dans la section sur la Transformation en application Java, nous avons pris la décision de supprimer le modèle "CatalogueAlgorithme" et la transformation ATL associée. Nous avons choisi de conserver uniquement le métamodèle généré par Xtext. Bien que le modèle ne soit plus transformé en "CatalogueAlgorithme", une transformation sera quand meme réalisé. Elle sera exécutée à partir d'un fichier .xmi créé dans l'Eclipse de développement, résolvant ainsi les problèmes d'incompatibilité entre les deux environnements Eclipse.

5.3 Choix des ressources utilisables pour un algorithme

Pour un algorithme, un utilisateur a trois possibilités comme ressource :

- dessiner la ressource grâce à l'outil Sirius
- utiliser un fichier extérieur
- utiliser un fichier Python fourni

Nous avions initialement envisagé de permettre à l'utilisateur d'utiliser un fichier Python en spécifiant uniquement son chemin, offrant ainsi la flexibilité de le placer n'importe où sur la machine. C'est pourquoi nous avons choisi le terme "chemin" comme attribut dans l'objet Algorithme. Toutefois, dans la pratique, l'utilisateur doit finalement déplacer le fichier dans le dossier "script" et ajouter l'importation du fichier Python dans le fichier "___init___.py" du même dossier, en utilisant la commande suivante : from .nomDuFichier import nomDeLaFonction.

5.4 Scripts Python Fournis

Afin de simplifier et faciliter le travail de l'utilisateur nous avons mis à disposition des fichiers python. Ces fichiers contiennent une liste non exhaustive de fonctions qui implémentent des opérations mathématiques classiques comme l'addition ou la soustraction et qui pourront être utiles à l'utilisateur. En effet il n'aura alors pas besoin de les écrire et pourra les utiliser directement. En particulier la fonction "cons" permettra de gérer les constantes en les mettant dans un liste avec un seul élément. Ensuite les fonctions comme "add" sont conçues pour s'adapter à deux entrées de la forme une colonne (sous forme de liste) et une constante (sous forme de liste avec un seul élément). Toutefois par rapport au codage de la fonction, si on prend comme entrée une constante et une colonne, la première entrée doit être une colonne.

6. Réalisation de l'application de l'utilisateur 2

6.1 Préparation des modèles à la transformation en code Java

Transformer un modèle en code Java correspond à une transformation Model-To-Text. Nous avons donc décidé de réaliser cette transformation avec Accéléo. Cependant, nous avions deux métamodèles d'entrée (schemaTable et CatalogueAlgorithme). Nous avons essayé de trouver une solution pour réaliser une transformation depuis deux métamodèles mais nous n'avons pas trouvé comment la lancer. Nous avons donc décidé de mixer ces deux métamodèles en un metamodèle SchemaTableAlgo avec ATL. Le processus de création de ce modèle aurait alors été :

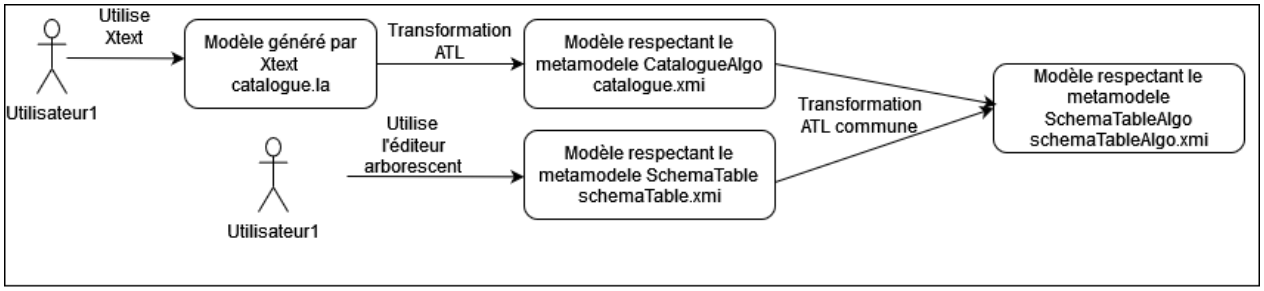

FIGURE 8 – Processus originel de création du modèle schemaTableAlgo

Cependant, l'intérêt de créer le modèle correspondant au métamodèle CatalogueAlgo pour le retransformer par Atl directement après et que ce soit sa seule utilisation n'était pas très utile. Nous avons donc pris la décision de supprimer cette étape et donc ce métamodèle et de transformer directement depuis le modèle généré par Xtext :

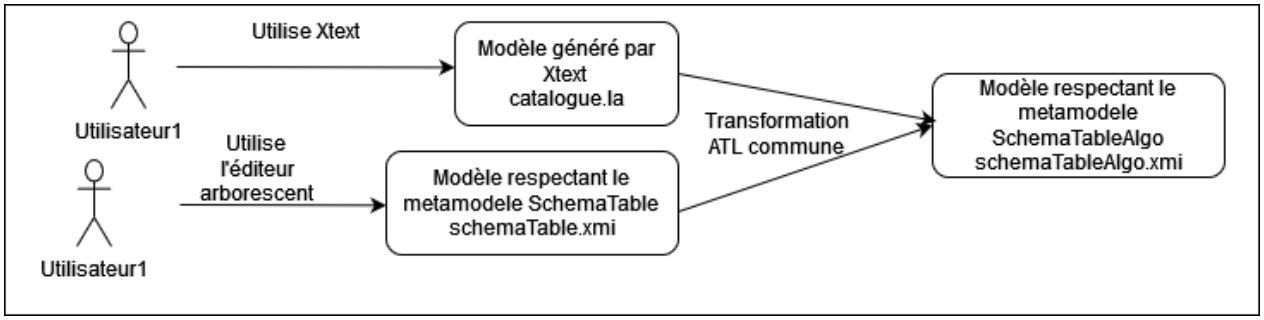

FIGURE 9 – Processus de création du modèle schemaTableAlgo

La transformation ATL, consiste principalement à fusionner deux modèles. En effet, à l'exception de la réunion des objets "Entree", "Sortie", et "Algorithme" du métamodèle Xtext en un objet "Algorithme", ainsi

que la transformation de "schemaTable" en "schemaTableAlgo" (qui était essentiellement le même objet avec un attribut supplémentaire pour lier les catalogues d'algorithme), la plupart des objets sont restés inchangés.

Voici le métamodèle de schemaTableAlgo :

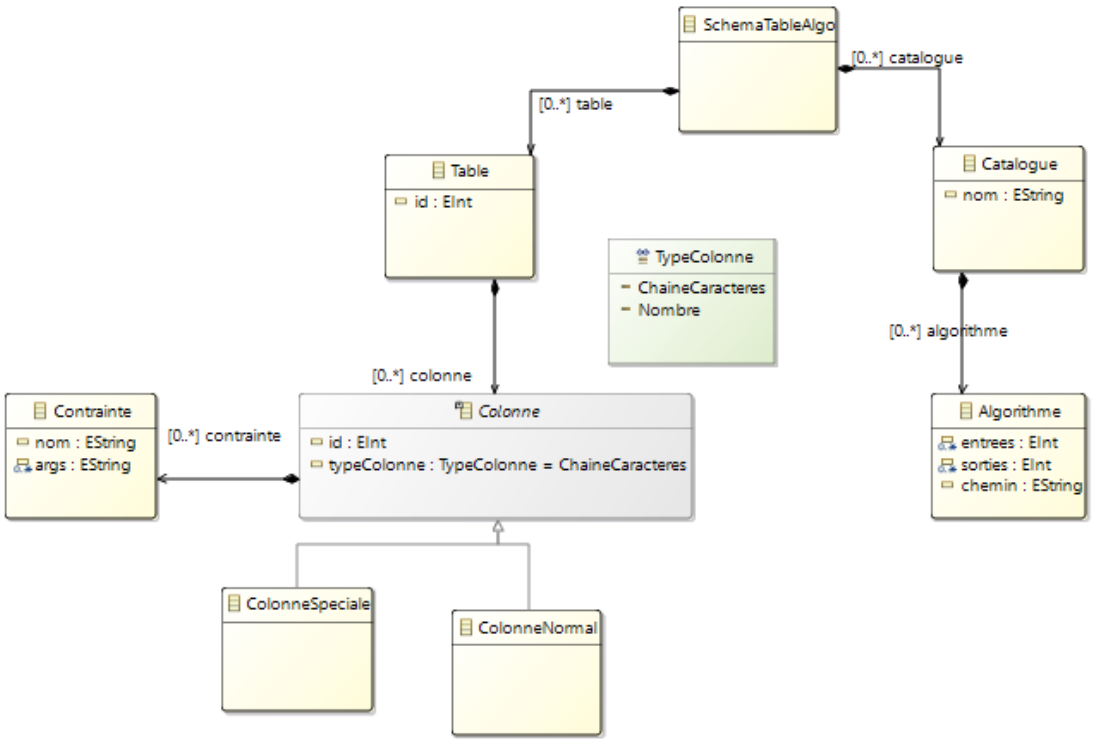

FIGURE 10 – Diagramme du métamodèle schema de table et algo

Comme mentionné précédemment, ce métamodèle résulte de la combinaison des deux métamodèles, avec les modifications évoquées lors de la description de la transformation ATL.

6.2 Transformation du modèle en code Java

6.2.1 Modèle de l'application Java produite

À ce stade, l'utilisateur n°1 dispose d'outils pour générer un modèle (.xmi), respectant le métamodèle schemaTableAlgo, représentant ses décisions concernant l'application prochainement utilisée par l'utilisateur n°2. Cependant, il ne peut pas encore en tirer l'application finale. C'est ici qu'Acceleo entre en jeu. L'objectif est de produire une application Java. Pour ce faire, nous avons créé le projet : fr.l127.app qui contient les fichiers sources de notre application. Il inclut différentes classes disposées dans le projet de manière fixée (Column, Algorithm, Constraint ...) qui vont permettre de représenter les données induites par le modèle .xmi en Java ainsi qu'une classe générée dynamiquement : Model.java. Pour générer ce fichier, nous utilisons Acceleo, un projet est associé à cette transformation, il s'agit du projet fr.l127.schemaTableToJavaApp. Le fichier au cœur de cette transformation est donc schemaTableToJavaApp.mtl qui parcourt les éléments du modèle .xmi et selon les éléments, instancie les classes fixées précédemment citées, et saisie les données. (Par exemple, on instancie Algorithm et on lui ajoute les colonnes d'entrées / sorties avec les méthodes addInput(int id) / addOutput(int id)).

Remarque : Nous aurions pu insérer toutes les classes fixées directement dans le Acceleo à la suite du code généré. Mais pour des questions de compréhension et de lisibilité, nous avons décidé de placer ces classes

6.2.2 Gestion des contraintes

La gestion des contraintes est une particularité à décrire. C'est le dernier élément paramétrable par l'utilisateur n°1. En effet, pour créer ses propres contraintes, ou consulter celles prédéfinies, l'utilisateur n°1 doit modifier ou lire le fichier ConstraintsDefinition.java du projet fr.l127.app évoqué précédemment. Par exemple la contrainte suivante est définie.

```
public static boolean beginBy(String element, char begin) {
    return element.charAt(0) == begin;
}
```

Ici la contrainte retourne vrai si l'élément commence par un char choisi. Une contrainte définie dans ConstraintsDefinition se traduit en une méthode statique retournant un booléen. Le premier paramètre doit être soit un String, soit un int. Ensuite, on peut mettre autant d'argument de type natif qu'on veut pour appliquer le traitement. Pour rappel, la classe Contrainte définie dans le métamodèle schemaTable dispose des attributs nom (String) et args (ensemble de String) représentant le nom de la contrainte, donc de la méthode associée dans ConstraintsDefintition, et l'ensemble des arguments mis sous forme de chaine de caractère. On peut saisir "0", qui sera converti en 0 si la signature de la méthode associée l'exige. Une contrainte est associée à une colonne, et sera vérifiée lorsque c'est nécessaire, suite à un import csv par exemple. Cela est géré dans la partie suivante.

6.2.3 Développement de l'Interface Graphique

L'interface utilisateur graphique, développée en Java Swing, sert de pont entre l'utilisateur final et la transformation Accéléo, générant des instances de classes (`Algorithm`, `Table`, etc), réunies en une seule classe `SchemaTable`. Cette interface permet une interaction intuitive avec les données sous forme tabulaire, similaire à un tableur Excel.

6.2.4 Fonctionnalités Principales

- **Visualisation et Manipulation des Données :** Les utilisateurs peuvent visualiser les données dans un format de tableau et effectuer des opérations telles que l'ajout, la modification, et la suppression de données dans les cellules. La gestion des lignes et des colonnes offre une flexibilité dans l'organisation des données.
- **Calcul automatique des algorithmes :** L'application exécute les algorithmes Python transmis dès que possible, de façon automatique, en manipulant ou calculant des données à partir des entrées des colonnes de la table. Les résultats sont dynamiquement affichés dans la table.
- **Importation et Exportation en CSV :** Des fonctionnalités d'importation et d'exportation au format CSV sont implémentées pour faciliter l'échange de données. Les utilisateurs peuvent charger et exporter des données depuis et vers des fichiers CSV.

6.2.5 Caractéristiques Supplémentaires

- **Gestion Avancée des Colonnes :** Des fonctionnalités ont été ajoutées pour identifier et traiter différemment les colonnes utilisées comme sorties d'algorithmes. Ces colonnes sont visuellement distinctes et non modifiables.
- **Interaction Contextuelle :** Un menu contextuel a été intégré, permettant aux utilisateurs de supprimer des lignes sélectionnées de manière intuitive via un clic droit.
- **Synchronisation avec les Objets Sous-Jacents :** Toutes les modifications dans l'interface graphique sont synchronisées en temps réel avec les instances des classes générées, assurant une cohérence entre l'interface utilisateur et la structure des données.

6.2.6 Conclusion

Cette interface graphique joue un rôle crucial en rendant les concepts de modélisation complexes accessibles à l'utilisateur final. Elle offre une expérience utilisateur intuitive tout en préservant la flexibilité du modèle sous-jacent.

7. Requêtes OCL

Pour éviter une mauvaise utilisation du métamodèle ressource graphique il a fallu mettre en place des contraintes. Nous les avons alors implémentées en OCL. Tout d'abord un objet résultat ne doit pas pouvoir prendre un objet résultat en entrée, en effet le résultat est censé être l'arrivée du modèle. De la même façon pour vérifier la bonne construction, une entrée d'un bloc intermédiaire, dans notre cas un bloc unaire ou binaire, ne peut pas être la sortie du même bloc. Il faut éviter cette boucle.

8. Améliorations envisagées

8.1 Amélioration de la robustesse des Algorithmes

Actuellement, la configuration des algorithmes autorise l'utilisation d'identifiants de colonnes en tant qu'entrées, même s'ils ne correspondent à aucune colonne existante. C'est lors de l'execution du Java que le roblème va etre detecté. Le but serait de le bloquer dès la création du modèle.

En outre, il est possible de spécifier des identifiants de colonnes en tant que paramètres d'un algorithme, même s'ils ne sont pas du même type que ceux requis par l'algorithme en question. Ces situations peuvent entraîner des blocages actuels dans la définition et l'exécution des algorithmes.

Un risque potentiel réside dans la possibilité de définir de manière récursive les colonnes et les algorithmes. Actuellement, il est possible de définir une colonne en utilisant une autre colonne qui, à son tour, peut être définie en remontant jusqu'à la colonne d'origine. Pour illustrer ce phénomène, voici un schéma d'exemple :

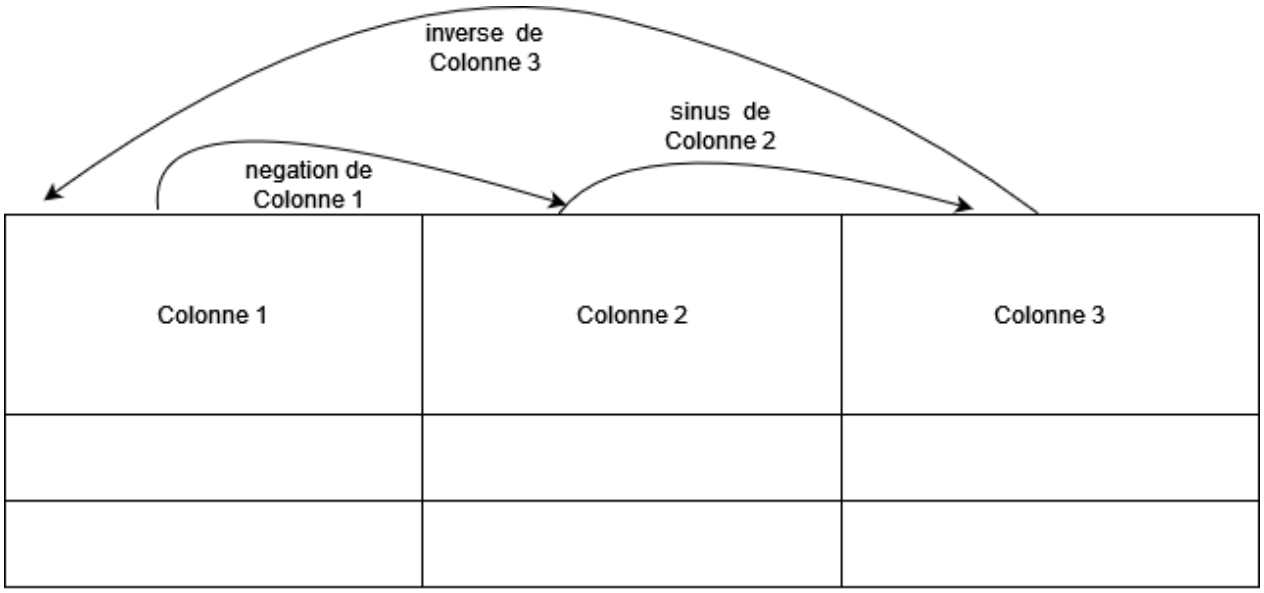

FIGURE 11 – Schéma de colonnes qui sont définis de manière récursive

8.2 Permettre l'utilisation de ressource d'autres langage

Nous avons envisagé la possibilité de permettre aux algorithmes d'utiliser des ressources externes dans des langages autres que Python. Cela aurait nécessité la mise en œuvre d'une traduction entre différents langages, impliquant une transformation Texte-à-Texte. Pour réaliser cette conversion, il aurait fallu la diviser en deux étapes distinctes : une conversion Texte-à-EMF et Modèle-à-Texte, utilisant respectivement des outils comme Xtext et Acceleo. Cette approche aurait nécessité la création d'un nouveau modèle représentant chaque langage supplémentaire que nous aurions voulu implémenter. Finalement, nous avons décidé de ne pas poursuivre, estimant que cela n'était pas le plus pertinent sur le plan pédagogique. En outre, effectuer

une conversion vers Python s'est avéré assez désagréable

8.3 Meilleur traitement des données entrées par l'utilisateur

Actuellement, si une donnée ne respecte pas les contraintes, toutes les données en dessous sont ignorées jusqu'à ce que l'erreur soit corrigée. Pour améliorer cela, nous proposons l'approche suivante :

1. Traiter les données par segments de colonnes.
2. À la rencontre d'une cellule invalide, ignorer cette cellule.
3. Continuer l'exécution des algorithmes sur les lignes suivantes.

Cette méthode permet de réduire le nombre d'appels Python, optimisant ainsi la performance globale du système.

8.4 Prise en charge des types Float dans les méta-modèles

Une autre amélioration envisageable concerne l'extension des types de données pris en charge dans nos méta-modèles. Jusqu'à présent, nous avons uniquement considéré les types `String` et `Integer`. Cependant, il serait bénéfique d'inclure également le type `Float`.

Raisons pour l'ajout de Floats L'ajout des `Floats` se justifie par plusieurs raisons :

- Plusieurs algorithmes, notamment ceux impliquant des fonctions trigonométriques comme le sinus et le cosinus, renvoient des résultats en `Float`.
- Notre programme Java, les contraintes définies, et les scripts Python associés sont déjà capables de gérer les `Floats`.

Bénéfices attendus L'intégration des `Floats` dans nos méta-modèles permettra une plus grande flexibilité et précision dans le traitement des données, et élargira la gamme des algorithmes applicables à notre système.

9. Bilan des Fichiers Rendus

Projets	**Fichiers**	**Fonctionnalité**
fr.l127.schemaTable	schemaTable.ecore, schemaTableDemo.xmi, constraints.ocl	Modélisation d'un schéma de table
fr.l127.schemaTableAlgo	schemaTableAlgo.ecore, schemaTableAlgoDemo.xmi	Modélisation mixant les métamodèles LA et schemaTable
fr.l127.ressourceGraphique	ressourceGraphique.ecore, test.xmi , test2.xmi constraints.ocl	Modélisation des ressources graphiques
fr.l127.rg2Python	ToPython.mtl	Transformation d'un modèle suivant ressourceGraphique en code Python
fr.l127.librairieAlgorithme	LA.xtext, LA.ecore, Librairie.xmi	Permettre à l'utilisateur 1 de définir textuellement les algorithmes utilisés
fr.l127.la_schemaTable2 schemaTableAlgo	la_schemaTable2 schemaTableAlgo.atl	Transformation des modèles suivant les métamodèles LA et schemaTable en un modèle suivant un métamodèle schemaTableAlgo
fr.l127.ressourceGraphique .app	script python dans fonction et algo gen , code de l'application Java dans le dossier src ,	Création de l'application Java utilisé par l'utilisateur 2
fr.l127.librairies.examples	exemple_algo.l a	Ecriture des algorithmes utilisés avec la syntaxe textuelle
fr.l127.ressourceGraphique .design	ressourceGraphique.odesign	Création de l'outil de réalisation de ressource graphique
fr.l127.ressourceGraphique .samples	ressourceGraphique .ressourcegraphique	Création d'une ressource graphique avec l'outil de création

Nous avons défini par convention que les projets commenceront par fr.l127 car nous sommes le groupe 7 du TD L1-2. Les modèles et métamodèles se trouvent dans le dossier model des projets concernés. Les fichiers OCL se trouvent dans le dossier ocl des projets concernés. Les trois derniers projets sont des projets de l'Eclipse de deploiement donc ils se trouvent dans le dossier runtime-EclipseApplication.

Les fichiers modèles ne trouvaient pas leur metamodèle. Nous avons donc configuré le lien en dur entre le modèle et le métamodèle ce qui oblige à ne pas les déplacer.

10. Conclusion

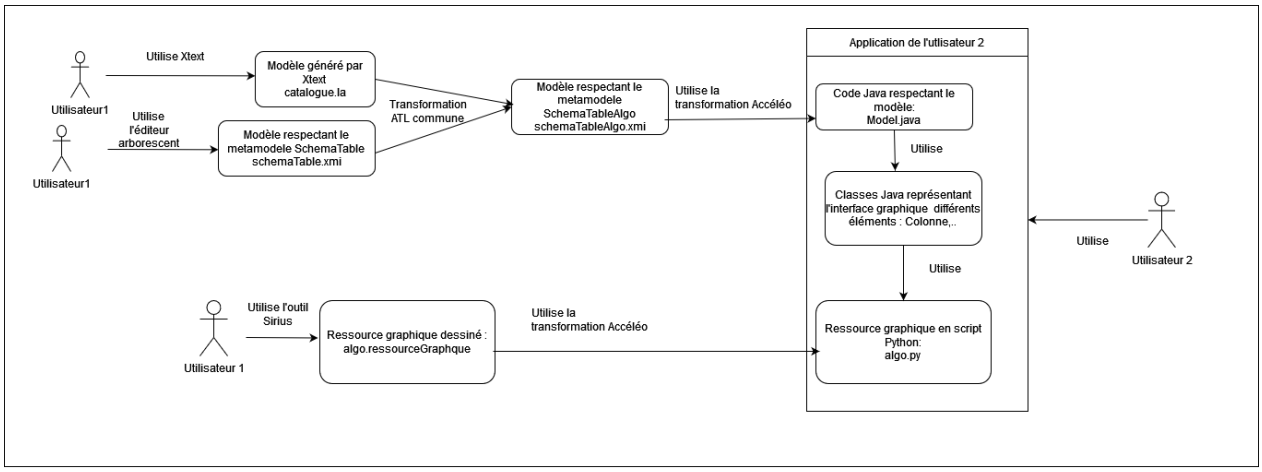

FIGURE 12 – Schéma bilan générale de l'utilisation de l'application

En conclusion, ce projet d'ingénierie dirigée par les modèles a non seulement abouti à la création d'une interface graphique Java performante et intuitive, mais a également mis en lumière l'importance de la génération de modèles pour assurer le bon fonctionnement de cette interface. L'utilisation d'outils tels qu'Accéléo, Sirius et ATL a été cruciale pour modéliser efficacement les données et les processus, et pour intégrer harmonieusement ces modèles avec l'interface utilisateur.

Contributions des outils de modélisation Le recours à des outils de modélisation comme Acceleo, Sirius et ATL a permis :

- Une génération de modèles précise et flexible, adaptée aux besoins spécifiques du projet.
- Une intégration efficace des modèles générés avec l'interface Java, garantissant une interaction utilisateur fluide.
- Une amélioration de la maintenabilité et de l'évolutivité de l'application grâce à une architecture bien structurée.

Perspectives futures Bien que le projet ait atteint de nombreux objectifs, des opportunités d'amélioration et d'extension subsistent. Les travaux futurs pourraient explorer des améliorations dans la génération de modèles, ainsi que l'intégration de nouvelles fonctionnalités dans l'interface Java.

Ce projet a non seulement atteint ses objectifs initiaux, mais a également posé les bases pour des développements futurs.