AFKER Samy, AZDAD Bilal,

HUANG Julien, JANSOU Vincent,

POIRIER Romain

Groupe M2 : Rapport du Projet d'Ingénierie Dirigée par les Modèles

Département Sciences du Numérique - Deuxième année
2023-2024

Table des matières

Table des figures

1 Introduction

Ce projet vise à créer une suite d'outils pour aider les utilisateurs à gérer et effectuer des calculs automatiques sur des données. Les fonctionnalités attendues sont structurées autour de la création, la sauvegarde et la consultation de schémas de table, d'algorithmes, et de scripts de calculs. Ce projet est fortement autonome, c'est à nous de développer des solutions aux fonctionnalités présentées.

Nous avons passé beaucoup de temps à essayer de comprendre les technologies nécessaires et à mettre en place les parties du projet, comme l'utilisation des fonctionnalités nécéssitant Acceleo, Xtext et Sirius. Malheureusement, nous n'avons pas réussi à terminer toutes les fonctionnalités prévues, comme la création des outils spécifiques aux schémas de données et la mise en place complète des fonctionnalités d'importation, visualisation et exportation des données.

Ce rapport explique nos difficultés, ce que nous avons appris et ce que nous pourrions faire à l'avenir dans le cadre de l'ingénierie dirigée par les modèles. Même si nous n'avons pas réussir à tout terminer, nous avons acquis une bonne compréhension des technologies et des processus impliqués dans ce projet. Nous espérons que ce rapport montrera ce que nous avons appris et les défis que nous avons rencontrés en essayant d'utiliser ces technologies pour créer ces outils logiciels.

2 Fonctionnalités sur les schémas de table

Dans cette partie, nous présentons le méta-modèle principal de l'outil d'environnement de calcul qui sert à construire un ou plusieurs schémas de table que nous avons nommé `(eCDS.ecore)` pour Environnement de Calcul Domaine-Spécifique.

2.1 Le méta-modèle des tableaux

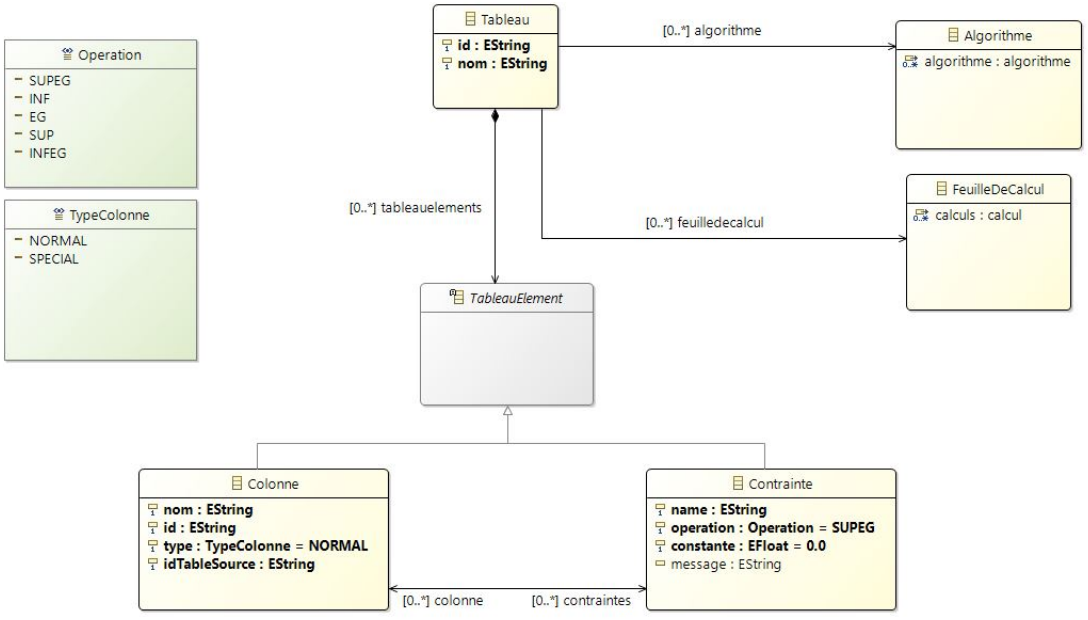

Figure 1 – Le Méta-modèle Tableau

2.2 Explication des choix de conception

— *Tableau* : C'est la classe racine, on pourra l'identifier avec son attribut id (qui est unique) ou avec son nom. Les deux attributs sont des chaines de caractères.

— *TableauElement* : Un tableau sera composé de tableauElement qui sera soit une contrainte ou une colonne.

— *Contrainte* : définit par son nom, une ou plusieurs colonnes, une opération, une constante et un message d'erreur.

— *Colonne* : definit par son nom, un identifiant unique, le type de colonne (normal par défaut), et l'identifiant tableau dont est issu la colonne (on peut déclarer qu'une colonne provient d'une autre table en indiquant le schéma de la table étrangère et la colonne à extraire de cette table).

— Pour un tableau donné on va lui associer un ou plusieurs Algorithmes, et FeuilleDeCalcul qui vont être spécifier dans les parties 3 et 4.

— On a préféré utiliser des enumérations pour pouvoir avoir la possibilité d'ajouter de nouvelles opération ou type de colonne plus facilement. Cela permet de rendre le modèle plus extensible.

Les defauts de cette conception :

— On ne peut pas definir de contraintes entre deux colonnes. Seulement entre une colonne et un entier. Pour le faire, nous aurions du ajouté des attributs opérandes gauche et droite pour pouvoir avoir deux colonnes entre un opérateur.

— On aura besoin de définir des contraintes OCL en plus. Notamment sur le nombre de colonne spécial (une et une seule)

— Pour les contraintes, on ne peut définir qu'une contrainte à la fois. On ne peut pas par exemple borner les valeurs d'une colonne entre deux entiers en créant une seule contrainte, on doit en créer une pour la borne supérieure et une pour la borne inférieure.

Nous avons créer un modèle (`Tableau.xmi`) conforme à un exemple d'utilisation de l'outil, mais à cause des références croisés, nous n'avons pas pu compléter les attributs algorithme et calculs lors de la création du modèle ce qui nous a empécher notamment de tester la transformation Acceleo sur un modèle correct.

2.3 Les contraintes OCL

Nous avons implémenter des contraintes OCL (`eCDS.ocl`) pour vérifier les contraintes classiques prédéfinis au sein de l'application dont :

— Contrainte sur les noms : on ne peut pas définir deux colonnes avec le même nom, deux contraintes avec le même nom, la syntaxe du nom doit correspondre à un formalisme standard.

2.4 Sirius

Pour des contraintes de temps, nous n'avons fait que le début de l'implémentation d'une syntaxe graphique pour les tableaux. Nous avons implenter les différents éléments pour la création des colonnes et contraintes mais ils ne s'affichaient pas sur la vue graphique. (`eCDS.design & eCDS.samples`). Tout de même, nous avons réflêchit à ce qu'on devrait être capables de faire à l'aide de Sirius. La figure 2 ci-dessous résume ce à quoi devrait ressembler notre modèle crée avec Sirius.

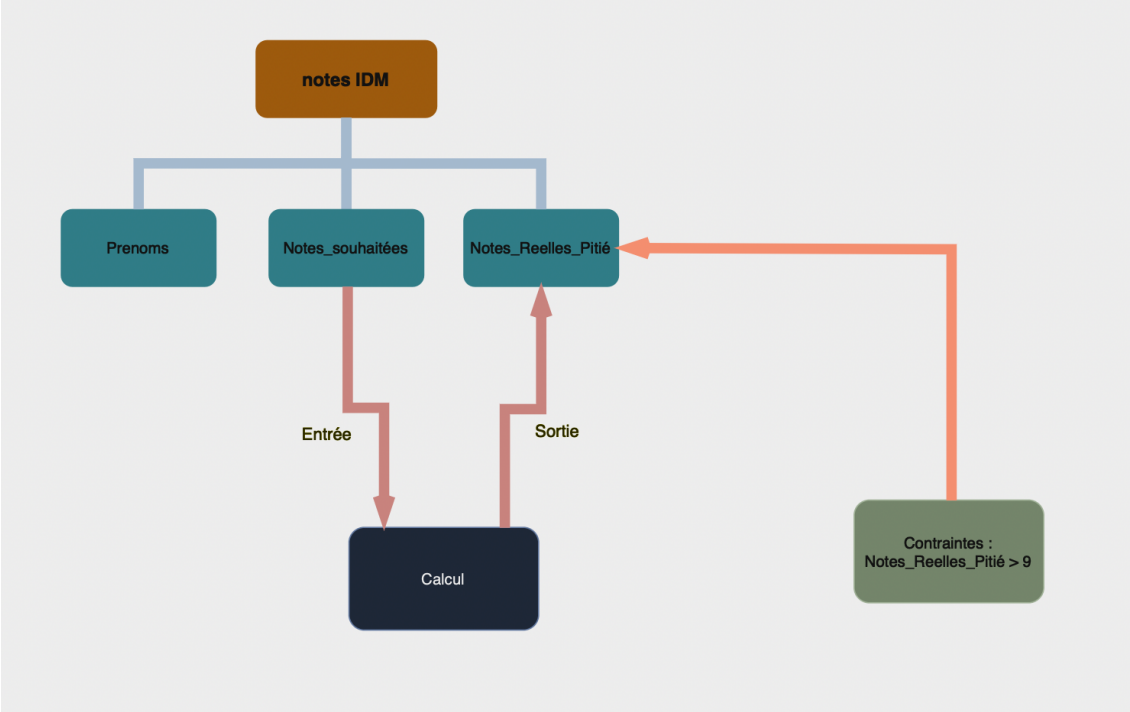

FIGURE 2 – Aperçu de la syntaxe graphique pour la création de tableaux

2.5 Xtext

Nous avons également construit une syntaxte textuelle concrête des modèles de tableaux avec Xtext `(eCDS.xtext)` pour permettre à l'utilisateur de spécifier son modèle de schéma de table. Pour définir la feuille de calcul éventuelle associé au schéma de table et les algorithmes que l'utilisateur peut utiliser nous inclut leur définition dans le fichier XText pour faciliter la représentation structurelle de l'environnement.

3 Fonctionnalités sur les algorithmes

Le but de cette partie est de permettre à l'utilisateur d'utiliser des algorithmes pour les appliquer sur ses données dans un langage de programmation qu'il aura choisit. Les algorithmes qu'il peut utiliser sont internes ou externes.

Pour les algorithmes internes, nous avons choisis de créer un méta-modèle calcul (calcul.ecore) décrit dans la section F3. Nous voulions donc transformer les modèles générés conformes au méta-modèle calcul en algorithme (`calcul.py` par exemple) pour ensuite pouvoir les appliquer sur nos données. Pour les algorithmes externes nous avons écrit le script d'un algorithme pour donner un exemple `(script/python/calcul_exemple.py)` avec la documentation associée à cet algorithme `(script/python/calcul_exemple.txt)`. Dans cet exemple, nous avons choisit d'additionner deux colonnes.

3.1 Le méta-modèle de algorithme

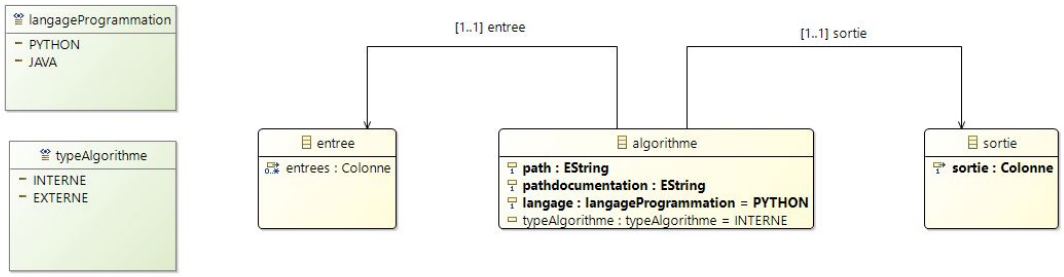

FIGURE 3 – Le Méta-modèle des algorithmes

3.2 Explication des choix de conception

Pour implémenter l'utilisation des algorithmes, nous avons conçu un autre méta-modèle (`algorithme.ecore`). Il est composé de :

— *algorithme* : C'est la classe mère, qui contient le chemin vers le fichier où est définit l'algorithme si l'utilisateur souhaite ajouté un algorithme externe à l'outil. On a également le chemin vers la documentation associée à l'aogithme. Nous avons deux attribut langage et typeAlgorithme qui permettent respectivement de connaitre le langage de programmation dans lequel l'algorithme est définit et si c'est un algorithme interne (prédéfinit dans l'outil) ou externe (importé par l'utilisateur).

Enfin, nous avons deux référence, une entree et une sortie pour l'algorithme.

— *entree* : C'est la classe qui définit l'entrée de l'agorithme c'est à dire les colonnes sur lesquelles on va l'appliquer. Il peut il y en avoir plusieurs d'où le choix des bornes de l'attribut.

— *sortie* : C'est la classe qui définit la sortie de l'agorithme, l'attribut sortie est donc la colonne résultante de l'application de l'algorithme sur les entrées.

Comme on peut le voir dans les classes entree et sortie, le méta-modèle algorithme utilise une référence croisé associé au méta-modèle des schémas de table pour pouvoir avoir accès aux colonnes en entrée et générer une colonne en sortie.

Nous avons écrit un modèle (algorithme.xmi) conforme au méta-modèle des algorithme mais dans ce modèle nous n'avons pas pu spécifié les entrées et les sorties à cause de la référence croisées.

Pour les algorithmes, nous n'avons pas implémenter de Sirius puisque les modèles conformes à ce méta-modèle sont relativement simple à créer sans Sirius. De même pour le Xtext des algorithmes.

3.3 Les contraintes OCL

Les contraintes OCL que nous avons implentés vérifient que les différents attributs de la classe algorithme (les entrées et sorties, les chemins etc) sont bien non vide.

4 Fonctionnalités sur les scripts de calculs

Le but de cette partie est de pouvoir réaliser des calculs dans un langage propre dédié à la plateforme. Le but est de pouvoir, par exemple, créer des calculs de ce type :

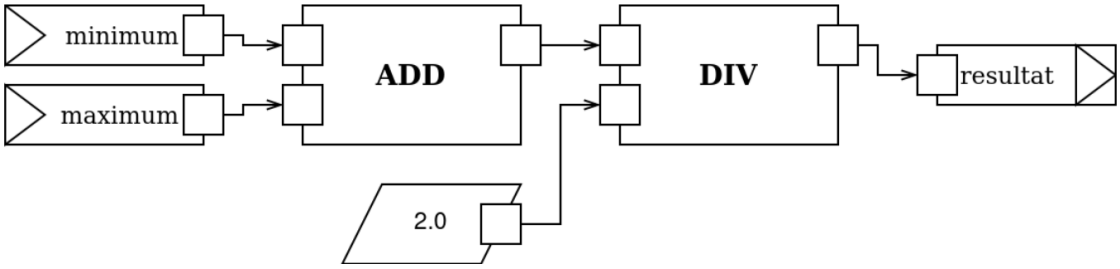

FIGURE 4 – Exemple de calcul de moyenne (sujet du projet)

4.1 Le méta-modèle calcul

Pour cela, nous avons créé un métamodèle appelé `(calcul.ecore)` qui décrit cette fonctionnalité. Il est présenté ci-dessous :

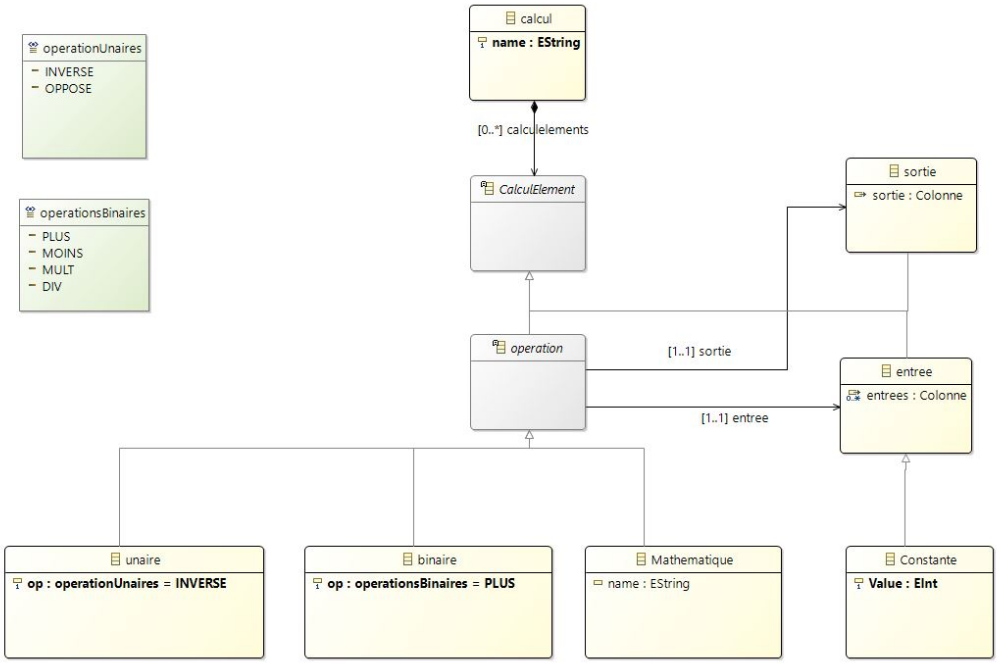

FIGURE 5 – Le Méta-modèle Calcul

Ce métamodèle est composé de plusieurs éléments :

— `calcul` : La classe principale de notre script de calcul caractérisée par un nom (moyenne pour l'exemple précédent).

— `CalculElement` : Tous les autres éléments du méta-modèle sont considérés comme étant des `CalculElement`.

— `operation` : Représente l'opération que l'on veut utiliser. Cette dernière peut être une opération binaire, une opération unaire ou une opération mathématique (sin, cos, exp etc.).

— `entree` : Représente les colonnes en entrée pour un calcul.

— `Constante` : Il s'agit d'un type particulier d'entrée qui est caractérisé par une valeur `Value`.

— `sortie` : Représente la colonne de sortie du calcul. Nous avons considéré qu'un calcul ne peut produire qu'une seule colonne en sortie.

— `operationsBinaires & operationsUnaires` : Nous avons quelques opérations binaires et unaires basiques qui sont déjà disponibles dans ces énumération du méta-modèle.

Les defauts de cette conception :

— De même que pour les contraintes dans le méta-modèle des tableaux, nous aurions pu faciliter la création des opération en ajoutant des attributs opérandes gauche et droite pour pouvoir avoir deux colonnes entre un opérateur.

— Pour la classe Constante, l'héritage de l'attribut entrees doit être d'arité 0 ce qui n'est pas vraiment un comportement naturel quand on souhaite créer un calcul avec une colonne et une constante au lieu de deux colonnes.

4.2 Les contraintes OCL

Afin de garantir une cohérence de notre méta-modèle, nous avons définit des contraintes OCL `(calcul.ocl)` sur ses éléments :

— Le nom d'un modèle calcul doivent respecter l'expression régulière suivante : `[A-Za-z_][A-Za-z0-9_]*`

— Une entrée doit être unique

— Nous avons fait le choix d'avoir uniquemnt une seule sortie.

4.3 Sirius

Nous avions commencer l'implémentation de la syntaxe graphique `(calcul.design & calcul.samples)` pour que l'utilisateur puisse créer avec des blocs un calcul comme les exemples des figures 6 et 7 ci-dessous. Cependant, comme pour le Sirius du schéma de table, nous avons rencontrer des problèmes dans l'affichage des éléments sur la vue graphique. Tout de même, nous avons réflêchit à ce qu'on devrait être capables de faire à l'aide de Sirius.

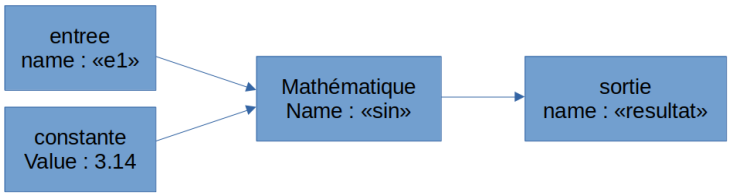

FIGURE 6 – Exemple d'utilisation de Sirius

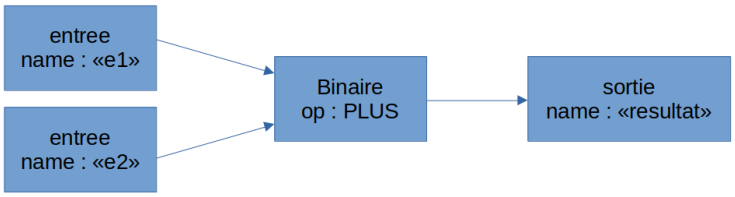

FIGURE 7 – Exemple d'utilisation de Sirius

4.4 Xtext

De même que pour les tableaux, nous avons définit la syntaxte textuelle concrête des modèles de calcul avec Xtext `(calcul.xtext)` pour faciliter la construction de ces modèles à l'utilisateur. Nous avons réussi a bien définir les classes du méta-modèle. Pour les références au méta-modèle des tableaux, nous avons choisit de le représenter en indiquant l'identifiant unique des colonnes.

5 Génération de la librairie et traitement de données

Cette partie explique comment nous souhaitions traiter la génération de la librairie. La librairie serait dans un langage appartenant à langageProgrammation (en l'occurence dans notre exemple Python).

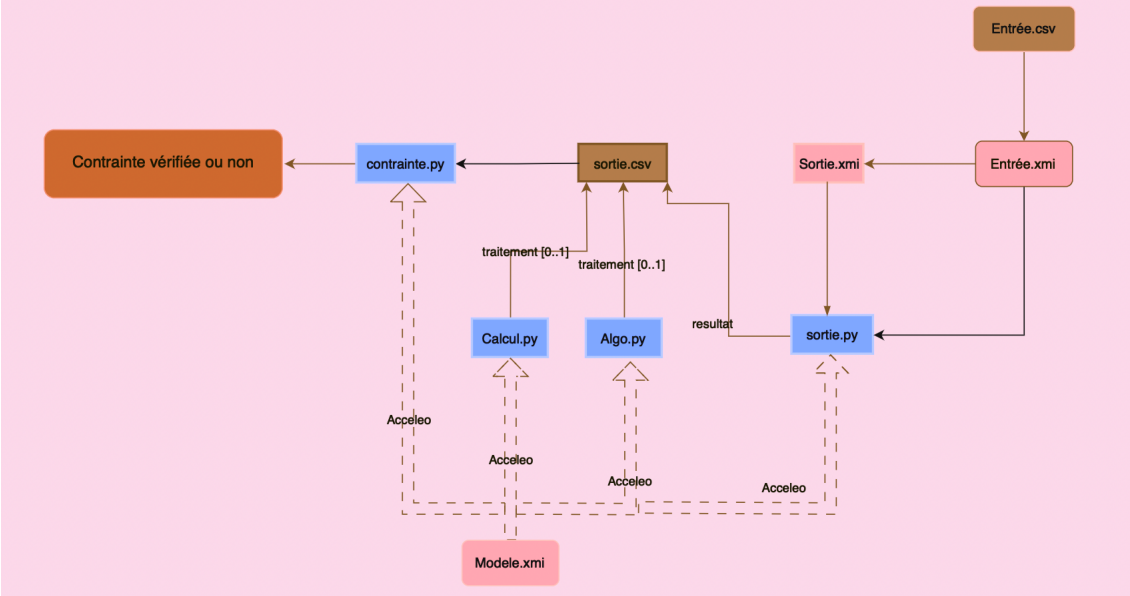

FIGURE 8 – Processus de traitement de données

Voici une explication du processus de traitement de données représenté par la figure 8 :

- Dans un premier temps on aura un fichier en entrée (**Donnee.csv** qui va contienir toutes les données initiales de l'utilisateur)
- Créer **donnee.xmi**, ca sera le modèle squelette qui est calqué sur les données de départ.
- Créer **sortie.xmi** : ca sera le squelette du modèle transformé, on ajoute des colonnes issues d'un algorithme (F2) ou d'un calcul (F3). (Crée à l'aide du méta-modèle, ou sirius).
- **modele.xmi** contient toutes les informations du schéma de table (**sortie.xmi, calcul.xmi, algo.xmi**).
- Chaque flèche acceleo correspond à une transformation : dans un language (ici python), qui aura comme entrée un fichier .xmi et comme sortie un fichier python.
- **Calcul.py** et **Algo.py** permettent d'effectuer les calculs et de modifier le fichier **Sortie.csv** (généré a l'aide de calcul.xmi et algo.xmi et en prenant comme entrée Donnee.csv).
- Contrainte.py permet de vérifier les contraintes que le developpeur a fixé sur sortie.csv.

5.1 Exemple d'utilisation des outils

Nous proposons un exemple d'un déroulement de A-Z d'un traitement nominal que l'utilisateur pourrait effectuer.

Voici un exemple de données en entrée importées par l'utilisateur : **Entrée.csv :**

donnee

Prenom	Notes_Souhaitees
Julien	20
Bilal	20
Vincent	20
Samy	20
Romain	20

FIGURE 9 – Donnee de départ : Entree.csv

Nous partons donc d'un modèle calqué sur ces données en entrée :

```
Model {
   Tab Entree {
      ColonneS Prenom;
      colonne Notes_Souhaitees;
   }

   Tab Sortie {
      ColonneS Prenom;
      colonne Notes_Reelles_Pitie;

      contrainte C1 ("contrainte non vérifiée", warning) :
Sortie.Notes_Reelles_Pitie > 9;
   }

   Calcul {
      calcul SqueletteSortie {
         Entree { Entree.Notes_Souhaitees }
         Sortie { Sortie.Notes_Reelles_Pitie }
         Operation { DIV }
         Constante { 2 }
         Sortie.Notes_Reelles_Pitie = Entree.Notes_Souhaitees / 2
      }
   }
}
```

FIGURE 10 – Exemple de modèle conforme aux données en entrée

Ce modèle est issue de la syntaxe xTest que l'on aurait eu, si notre métamodèle était compris par les outils de EMF. Pour pallier ce problème nous aurions pu opter pour un unique méta-modèle qui regroupait tout les méta-modèles mais il était déconseillé de le faire dans le sujet du projet, même si d'autres groupes ont connu le même problème et ont décidé de tout regroupper dans un unique métamodèle. On pourrait créer ce modèle via Sirius comme le montre la vue graphique de la figure 2.

Une fois ce modèle mis en place, grâce à des transformations Acceleo nous pouvons générer des scripts python qui vont permettre d'appliquer les traitements suivants (les propositions de scripts acceleo sont en annexe du rapport) :

Sortie.csv :

donnee

Prenom	Notes_Reelles_Pitie
Julien	10
Bilal	10
Vincent	10
Samy	10
Romain	10

FIGURE 11 – Sortie suite à l'application des scripts Acceleo

Processus de Traitement des Données

Explication :
Le processus de traitement des données se déroule comme suit :

- **Données en entrée :** Nous commençons avec des données au format **csv**.
- **Traitement :**
  - Application d'un calcul sur une colonne spécifique.
  - Stockage du résultat du calcul dans une nouvelle colonne.
  - Ces opérations sont réalisées grâce au script Python **Calcul.py** (Voir Annexe).
- **Extraction et Sortie :**
  - Extraction des colonnes nécessaires à l'aide du script **Sortie.py** (Voir Annexe).
  - Le fichier **csv** résultant constitue notre sortie.
- **Vérification de la contrainte :**
  - Contrôle de la condition que chaque note est supérieure ou égale à 9.
  - Cette vérification est effectuée par le script **Contrainte.py** (Voir Annexe).

5.2 Transformations Accéléo

Nous avions commencer à réaliser les transformations Accéléo décrites sur la figure 8.
Dans les fichiers `(eCDS.acceleo & calcul.acceleo)`. Cependant pour plusieurs attributs nous n'avons pas réussi à compléter entièrement le code python qui doit résulter de ces transformations.

6 Visualisation des données

Nous n'avons pas eu le temps d'implémenter cette fonctionnalité qui consistait à représenter graphiquement les données générées. Cependant, nous avions réfléchit à comment illustrer nos données lors du processus de génération de ces données.

Nous souhaitions réaliser une transformation avec Acceleo pour transformer le modèle en sortie de notre outil (`sortie.xmi`) en créant du code HTML, pour pouvoir représenter les colonnes (avec la balise *table* notamment).

7 Conclusion

Améliorations possibles

Avec un regard critique sur notre travail, comme on a seulement réfléchi sur l'implémentation, et commencé a mettre en oeuvre les outils (Acceleo, Sirius...) sans pour autant avoir une version complète et aboutie, plusieurs pistes d'amélioration s'offrent à nous :

- ***Implémentation concrète des outils*** : L'une des principales pistes d'amélioration consiste à effectuer l'implémentation complète des outils. En dépit de nos réflexions sur la théorie, l'absence de cette implémentation représente une lacune significative dans notre projet. L'implémentation de ces outils permettrait non seulement de compléter le projet de manière plus concrète, mais également d'offrir à l'utilisateur une utilisation plus intuitive et visuelle.
- ***Validation des failles potentielles dans le métamodèle*** : Nous reconnaissons que notre métamodèle pourrait présenter des failles. Nous avons changé plusieurs fois de métamodèle ce qui a donc à plusieurs reprises changer notre approche vis à vis du sujet. De plus, nous sommes conscient que les méta-modèles rendus peuvent contenir des erreurs non identifiées puisque nous avons peu réaliser la conception concrête des outils. Une démarche d'amélioration cruciale serait de procéder à une implémentation concrète afin de mettre en lumière ces éventuelles failles. Cela nécessiterait des tests approfondis sur plusieurs exemples et des analyses des cas d'utilisation des modèles dans des scénarios réels. Comme on a pu l'apprendre durant notre mini-projet, lors de l'implémentation de la syntaxe graphique, on avait modifié à de nombreuse reprise notre méta-modèle de départ.
- ***Organisation du temps et du groupe*** : En dehors de l'aspect programmation, on a vraiment mal géré notre temps. La manière dont nous avons organisé notre temps et notre équipe a posé des problèmes importants dans notre projet. On a manqué d'une bonne planification au départ et la répartition des tâches n'a pas été idéale. Nous nous sommes réparties les taches en fonction des fonctionnalités ($F1..F6$) à traiter dans le sujet mais finalement nous nous sommes rendu compte qu'il fallait partir sur des fondations solides puisques nos solutions pour résoudre ces fonctionnalités ne pouvaient pas être mise en commun selon l'approche que nous avions employé. Le fait que ce projet soit extrêment libre de les choix d'implantation nous a posé beaucoup de problèmes puisque nous n'étions pas guidés. Le fait de ne pas évaluer régulièrement nos progrès ont entraîné des retards. Si on avait mieux coordonné nos efforts, fait des bilans réguliers, et été plus flexibles face aux imprévus, ça aurait pu aider. Nous retenons que pour les futurs projets, il est essentiel d'avoir une planification plus sérieuse, une meilleure communication, des évaluations fréquentes afin de mieu gérer le temps et les personnes dans l'équipe.

8 Annexe

8.1 Proposition de scripts Acceleo

Dans cette partie nous fournissons des scripts Acceleo (non compilés) qui permettraient d'obtenir les scripts python nécessaires pour les traitements des tableaux

8.1.1 Calcul : Acceleo + python

Acceleo Calcul :

```
[module generate('http://www.example.org/MyModel')/]
[template public generatePythonScript(model : Model)]
[file ('Calcul.py', false, 'UTF-8')]
"""
import pandas as pd
# Chemin vers le fichier CSV
file_path = '[model.tabs.get(0).sourceFilePath/]'
try:

    # Lecture du fichier CSV
    df = pd.read_csv(file_path)
    # Applique le calcul spécifié dans le modèle pour modifier la colonne

    df['[calculElement.sortie.name/]'] =
df['[calculElement.entree.name/]'] [calculElement.operation.symbol/]
[calculElement.constante.value/]

    # Écriture des modifications dans le fichier CSV existant
    df.to_csv(file_path, index=False)

    message = "Le fichier CSV a été modifié avec succès :
{}".format(file_path)

except Exception as e:

    message = "Une erreur est survenue lors de la modification du
fichier CSV : {}".format(e)

print(message)
"""
[/file]
[/template]
```

FIGURE 12 – Script permettant d'obtenir un script Calcul.py

Calcul.py

```python
import pandas as pd

# Chemin vers le fichier CSV
file_path = 'chemin/vers/Entree.csv'

try:
    # Lecture du fichier CSV
    df = pd.read_csv(file_path)

    # Applique le calcul spécifié dans le modèle pour modifier la colonne
    df['Notes_Reelles_Pitie'] = df['Notes_Souhaitees'] / 2

    # Écriture des modifications dans le fichier CSV existant
    df.to_csv(file_path, index=False)

    message = "Le fichier CSV a été modifié avec succès : {}".format(file_path)
except Exception as e:
    message = "Une erreur est survenue lors de la modification du fichier CSV : {}".format(e)

print(message)
```

FIGURE 13 – Script python qui permet de faire un calcul définit dans le modèle

8.1.2 Contrainte : Acceleo + python

Acceleo Contrainte :

```
[module generate('http://www.example.org/MyModel)/]

[template public generatePythonScript(model : Model)]
[file ('Contrainte.py', false, 'UTF-8')]
"""
import pandas as pd

# Chemin vers le fichier CSV
file_path = '/Users/bilalazdad/Downloads/Sortie.csv'

try:
    # Lecture du fichier CSV
    df = pd.read_csv(file_path)

    [for (tab : Tab | model.tabs)]
        [for (c : Contrainte | tab.contraintes)]
            # Nom de la colonne à vérifier
            column_name = '[c.column.name/]'

            # Vérification de la contrainte sur la colonne spécifiée
            if (df[column_name] [c.operator.symbol/] [c.threshold/]).all():
                message = f"Toutes les valeurs dans la colonne
'{column_name}' sont conformes à la contrainte [c.name/]."
            else:
                message = f"Certaines valeurs dans la colonne
'{column_name}' ne respectent pas la contrainte [c.name/]."
            print(message)
        [/for]
    [/for]

except Exception as e:
    message = f"Une erreur est survenue lors de la lecture du fichier
CSV : {e}"

print(message)
"""
[/file]
[/template]
```

FIGURE 14 – Script permettant d'obtenir un script Contrainte.py

Contrainte.py

```python
import pandas as pd

# Chemin vers le fichier CSV
file_path = '/Users/bilalazdad/Downloads/Sortie.csv'

try:
    # Lecture du fichier CSV
    df = pd.read_csv(file_path)

    # Nom de la colonne à vérifier
    column_name = 'Notes_Reelles_Pitie'

    # Vérification de la positivité des valeurs dans la colonne spécifiée
    if (df[column_name] >= 9).all():
        message = f"Toutes les valeurs dans la colonne '{column_name}' valides."
    else:
        message = f"Certaines valeurs dans la colonne '{column_name}' ne sont pas valides."

except Exception as e:
    message = f"Une erreur est survenue lors de la lecture du fichier CSV : {e}"

print(message)
```

FIGURE 15 – Script python qui permet de vérifier une contrainte sur une colonne

8.1.3 Sortie : Acceleo + python

Acceleo Sortie :

```
[module generate('http://www.example.org/MyModel')/]

[template public generatePythonScript(model : Model)]
[file ('Sortie.py', false, 'UTF-8')]
"""
import pandas as pd

# Chemin vers le fichier CSV source et de sortie
source_file_path = '[model.tabs.get(0).sourceFilePath/]'
output_file_path = '[model.tabs.get(0).outputFilePath/]'

try:
    # Lecture du fichier CSV source
    df = pd.read_csv(source_file_path)

    # Noms des colonnes spécifiées dans le modèle
    nom_colonne = '[model.tabs.get(0).colonnes->select(clc.name =
'Notes_Reelles_Pitie')->first().name/]'
    nom_colonneS = '[model.tabs.get(0).colonnes->select(clc.name =
'Prenom')->first().name/]'

    # Création d'un nouveau DataFrame avec les colonnes spécifiées
    new_df = df[[nom_colonneS, nom_colonne]]

    # Écriture du nouveau DataFrame dans un fichier CSV
    new_df.to_csv(output_file_path, index=False)

    message = "Le fichier CSV a été créé avec succès :
{}".format(output_file_path)
except Exception as e:
    message = "Une erreur est survenue lors de la création du fichier
CSV : {}".format(e)

print(message)
"""
[/file]
[/template]
```

FIGURE 16 – Script permettant d'obtenir un script Sortie.py

sortie.py

```python
import pandas as pd

# Chemin vers le fichier CSV source
source_file_path = '/Users/bilalazdad/Downloads/Entree.csv'
# Chemin vers le fichier CSV de sortie
output_file_path = '/Users/bilalazdad/Downloads/Sortie.csv'

try:
    # Lecture du fichier CSV source
    df = pd.read_csv(source_file_path)

    # Nom des colonnes à copier
    nom_colonne = 'Notes_Reelles_Pitie'
    nom_colonneS = 'Prenom'

    # Création d'un nouveau DataFrame avec les colonnes spécifiées
    new_df = df[[nom_colonneS, nom_colonne]]

    # Écriture du nouveau DataFrame dans un fichier CSV
    new_df.to_csv(output_file_path, index=False)

    message = f"Le fichier CSV a été créé avec succès : {output_file_path}"
except Exception as e:
    message = f"Une erreur est survenue lors de la création du fichier CSV : {e}"

print(message)
```

FIGURE 17 – Script python qui permet de créer la colonne