Rapport - Projet IDM
Un Environnement de Calcul Domaine-Spécifique

Groupe B-04 :
Assala ASSELLALOU
Nour El Houda MHAMEDI
Maxime LAURENT
Youri RENOUD-GRAPPIN
Julien GAUTHIER

Département Sciences du Numérique - HPC et Big Data
2024-2025

Table des matières

Table des figures

1 Introduction

L'ingénierie dirigée par les modèles est une approche méthodologique qui repose sur la création et la manipulation de modèles pour résoudre des problématiques complexes de manière automatisée.

Dans le cadre de ce projet, nous avons appliqué les concepts vus en cours et en TPs pour développer un environnement de calcul domaine-spécifique destiné à la gestion des données financières et logistiques d'une compagnie aérienne.

Notre démarche a été guidée par les spécifications fournies notamment les fonctionnalités F1,F2 et F3 que nous avons traduites en plusieurs méta-modèles interconnectés. Ces méta-modèles servent de base au développement d'outils dédiés dans le but de réaliser les fonctionnalités F4, F5 et F6. Ce rapport présente d'abord la conception des méta-modèles, une étape fondamentale du projet, avant de décrire les outils développés et les résultats obtenus. Enfin, il propose une analyse critique de notre approche et des pistes d'amélioration possibles.

2 Conception des méta-modèles avec Ecore

La conception des metamodele est la première étape de résolution d'un problème d'ingénierie dirigée par les modèles car constitue la base de tout le reste. Tout au long du projet on a fait évalue nos metamodele pour essayer de se rapprocher au mieux des fonctionnalités exiges par le sujet du projet. La version présentée dans ce rapport est celle que nous avons retenue et présentée lors de l'oral.

2.1 Méta-modèle Table

En s'appuyant sur la fonctionnalité **F1** du projet, nous avons conçu le métamodèle *Table* pour représenter les différents aspects du schéma de table.

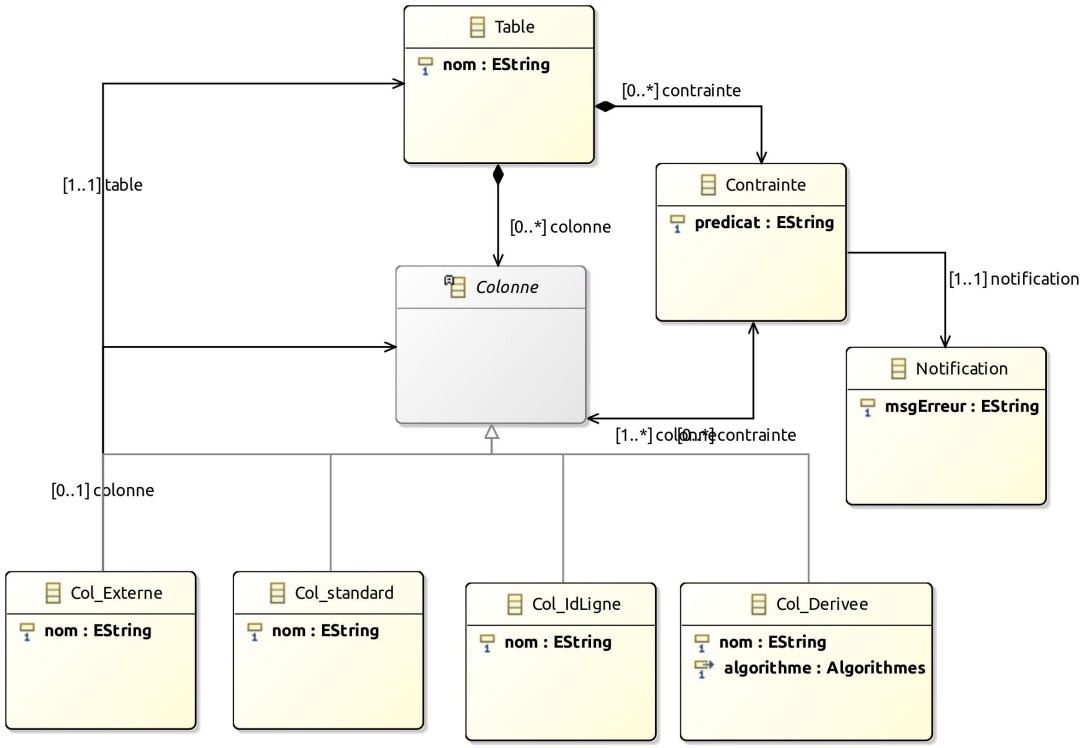

FIGURE 1 – Méta modèle Table.ecore

Table est la classe principale qui représente un schéma de table et se compose des deux classes : **Colonne (F1.1)** qui représente les colonnes de celle ci et de **Contrainte (F1.5)** qui définit les contraintes à appliquer sur les colonnes d'une table.

Colonne est une classe abstraite qui représente une colonne générique d'une table et peut être spécialisée en différentes catégories de colonnes :

- **Col_standard** : une colonne normale dans la table **(F1.1)**
- **Col_idLigne** : une colonne spéciale pour les identifiants des lignes **(F1.2)**
- **Col_Derivee** : une colonne dérive d'autres colonnes en precisant un **algorithme (F1.3)**
- **Col_Externe** : une colonne qui provient d'une autre table **(F1.4)**

Et **Contrainte** est une classe définit les contraintes à appliquer sur les colonnes d'une table. Chaque contrainte est un **prédicat** qui vérifie si les données respectent certaines conditions et la classe Notification est utilisée pour notifier l'utilisateur par un message d'erreur : **msgErreur (F1.5)**.

2.2 Méta-modèle Algorithme

En s'appuyant sur la fonctionnalité **F2** du projet, nous avons conçu le métamodèle *Algorithmes* pour représenter les différents aspects d'un algorithme.

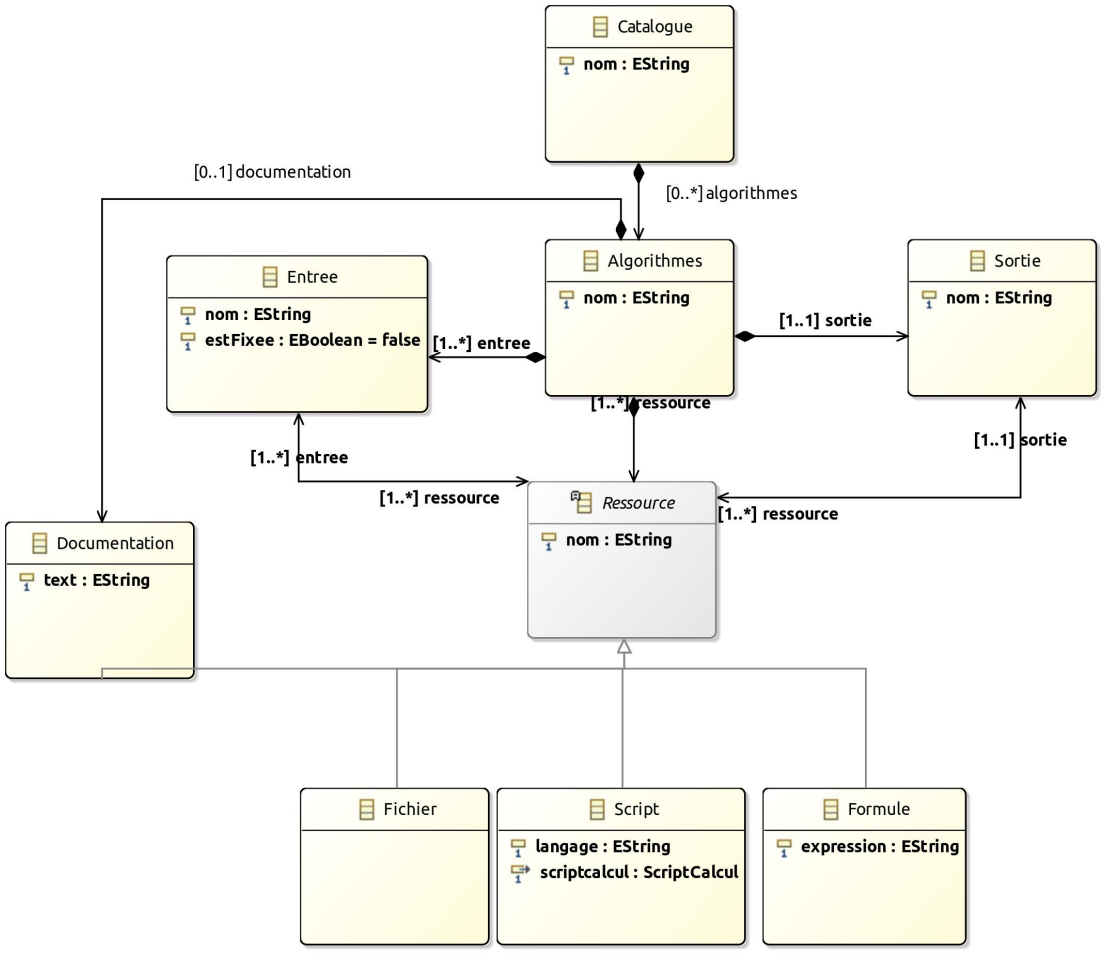

FIGURE 2 – Méta modèle Algorithme.ecore

Catalogue est la classe qui permet de regrouper les algorithmes **(F2.6)**

Algorithmes *(normalement sans s)* est la classe qui définit un algorithme particulier. Un algorithme est composé de plusieurs éléments :

- **Entree** : définit les paramètres d'entrée de l'algorithme et peut être fixée d'où l'attribut **estFixee** de type booléen , ce qui permet de proposer un algorithme avec un paramètre particulier **(F2.1 ; F2.3)**.
- **Sortie** : le résultat produit par l'algorithme, c'est-à-dire la donnée générée à la fin du calcul **(F2.2)**.
- **Ressource** : classe abstraite qui représente la ressource utilisée par l'algorithme pour effectuer le calcul **(F2.5)**. Elle peut être un **Fichier**, un **Script**, ou une **Formule**, et on a ajouté **Documentation**.
- **Documentation** : permet d'ajouter des explications sur l'algorithme ou ses entrées **(F2.4)**.

2.3 Méta-modèle ScriptCalcul

En s'appuyant sur la fonctionnalité **F3** du projet, nous avons conçu le métamodèle *ScriptCalcul* pour représenter les différents aspects d'un script de calcul.

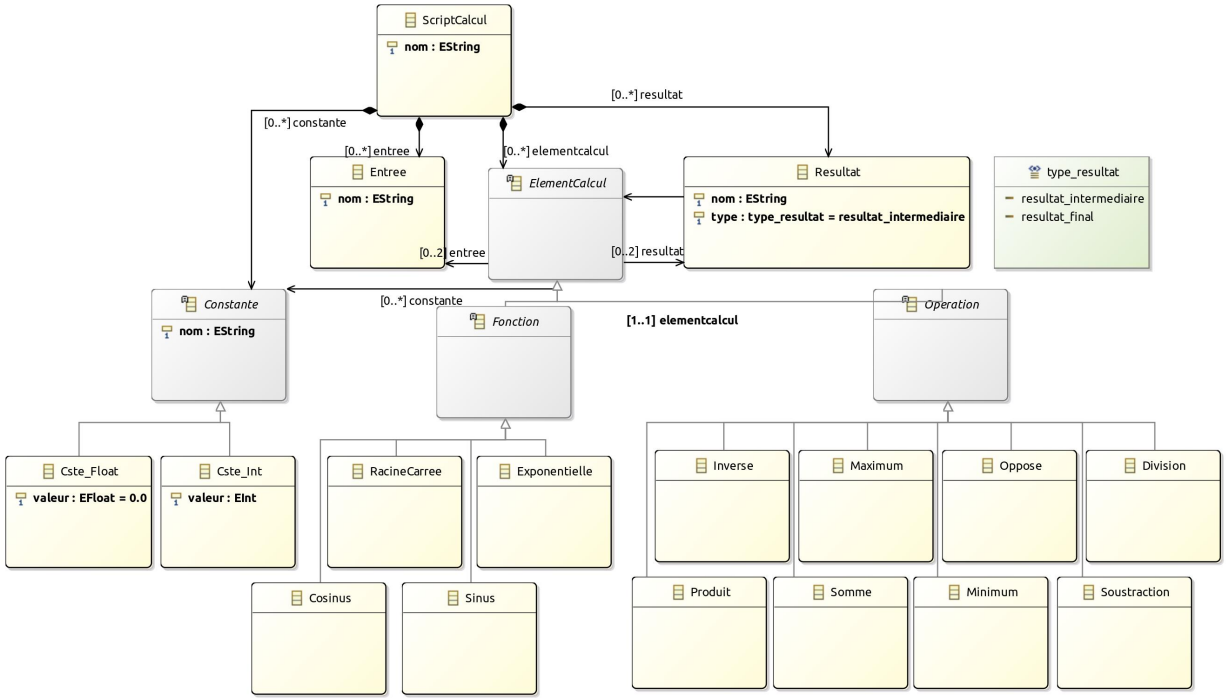

FIGURE 3 – Méta modèle ScriptCalcul.ecore

ScriptCalcul est la classe principale. Un script permet de définir un ensemble d'opérations et de calculs, en spécifiant les entrées, les sorties, les constantes, et les éléments de calcul à utiliser. Il est composé des éléments suivants :

- **Entree** : définit les paramètres d'entrée du script de calcul. Chaque entrée représente un argument nécessaire pour l'exécution des opérations du script **(F3.2)**.
- **Resultat** : représente le résultat produit par le script de calcul, indiquant le type de résultat (final ou intermédiaire) et les éléments impliqués dans le calcul **(F3.5)**.
- **type-resultat** : énumération qui distingue les résultats intermédiaires des résultats finaux du calcul.
- **Element Calcul** : une classe abstraite qui regroupe les éléments de calcul de base du script, comme les opérations et les fonctions : representes respectivement par les classes abstraites **Operation** et **Fonction**.
  - **Operation** : une classe abstraite qui représente les différentes opérations disponibles dans le script, comme les sommes, produits, inverses, opposés, minimum et maximum de valeurs **(F3.4)**.
  - **Fonction** : une classe abstraite qui représente une fonction spécifique, telle que le sinus, le cosinus, la racine carrée, l'exponentielle **(F3.5)**.
- **Constante** : définit les constantes utilisées dans les calculs, avec des sous-classes **Cste_Float** et **Cste_Int** pour les valeurs de type réel ou entier **(F3.6)**.

3 Conception d'outils graphiques avec Sirius

3.1 Objectifs

On propose Sirius comme outil de création graphique des modèles afin de faciliter ce processus pour des utilisateurs techniques. Ainsi, nous avons créé un viewpoint adapté pour chacun des Méta Modèles : Table , Algorithme et ScriptCalcul.

3.2 Point de vue pour Table

3.2.1 Table.odesign

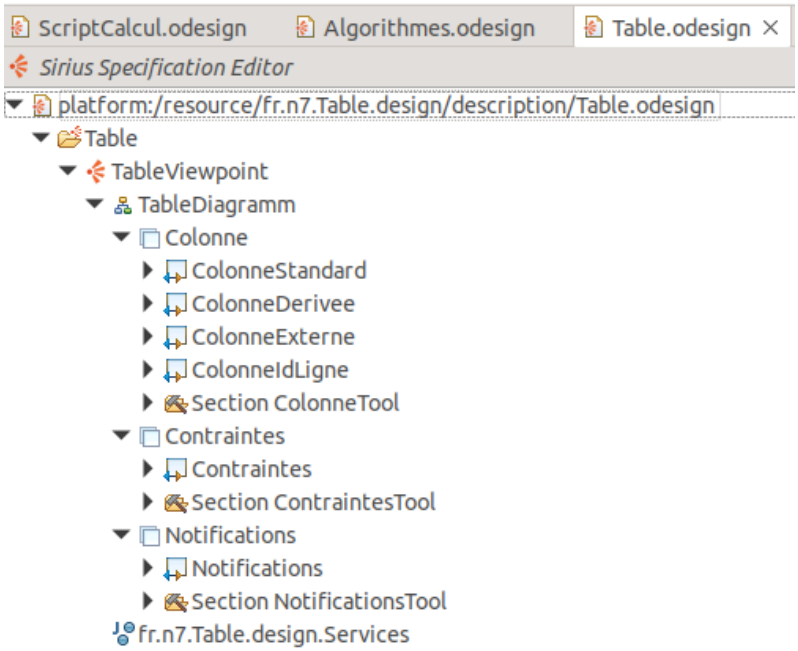

FIGURE 4 – Table.odesign

Pour Table on a créé :

— **Colonne** : un calque pour l'affichage et la création des colonnes de la table.
  — *ColonneStandard* : Node qui représente les colonnes normales dans une table
  — *ColonneDerivee* : Node qui représente les colonnes provenant d'autres colonnes de la même table.
  — *ColonneExterne* : Node qui représente les colonnes provenant d'une autre table.
  — *ColonneIdLigne* : Node qui représente les colonnes spéciales pour les identifiant de lignes dans la table.
— **Contraintes** : un calque pour l'affichage et la création des contraintes sur les colonnes.
— **Notifications** : un calque pour l'affichage et la création des notifications si les contraintes ne sont pas respectées.

3.2.2 Résultats et Validation

Le point de vue de table est valide.

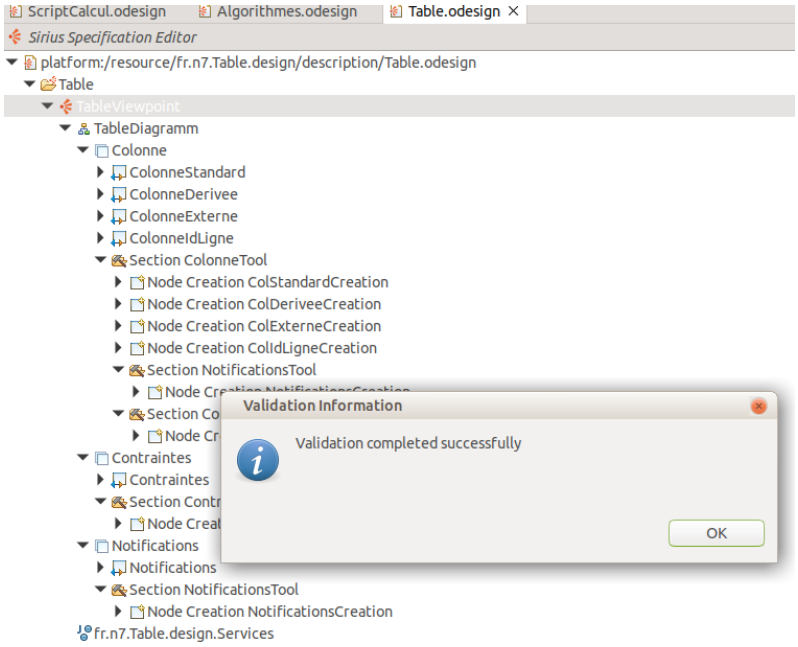

FIGURE 5 – Validation Table.odesign

Les exemples créés s'ouvrent correctement sous une forme graphique, offrant la possibilité de modifier les attributs définis dans le méta-modèle. La création de colonnes est également fonctionnelle.

On peut remplir les champs pour déclarer les provenances des colonnes (notamment pour les colonnes externes) ou encore ajouter des références vers des algorithmes permettant pour calculer les colonnes dérivées, par exemple.

Nous avons ainsi obtenu une modélisation complète de Table et permis une gestion précise des données associées aux colonnes.

3.3 Point de vue pour Algorithme

3.3.1 Algorithme.odesign

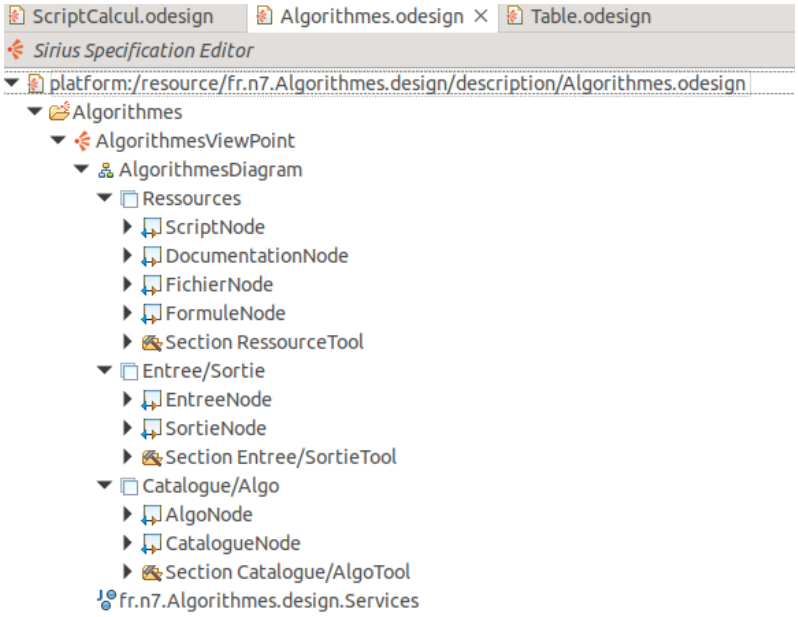

FIGURE 6 – Algorithme.odesign

Pour Algorithme on a créé :

- **Ressources** : un calque pour l'affichage et la création des ressources d'un algorithme.
  - *ScriptNode* : Node qui représente les ressource de type script.
  - *DocumentationNode* : Node qui représente les ressources de type documentation.
  - *FichierNode* : Node qui représente les ressources de type fichier.
  - *FormuleNode* : Node qui représente les ressources de type formule.
- **Entree/sortie** : un calque pour l'affichage et la création des entrées et sorties d'un algorithme.
  - *EntreeNode* : Node qui représente les entrées (arguments) d'un algorithme.
  - *SortieNode* : Node qui représente la sortie (résultat) d'un algorithme.
- **Catalogue/Algo** : un calque pour l'affichage et la création du catalogue qui regroupe les algorithmes et les algorithmes en question.
  - *AlgoNode* : Node qui represente les algorithmes du catalogue.
  - *CatalogueNode* : Node qui represente le catalogue qui englobe les algorithmes.

3.3.2 Résultats et Validation

Le point de vue Algorithme est valide.

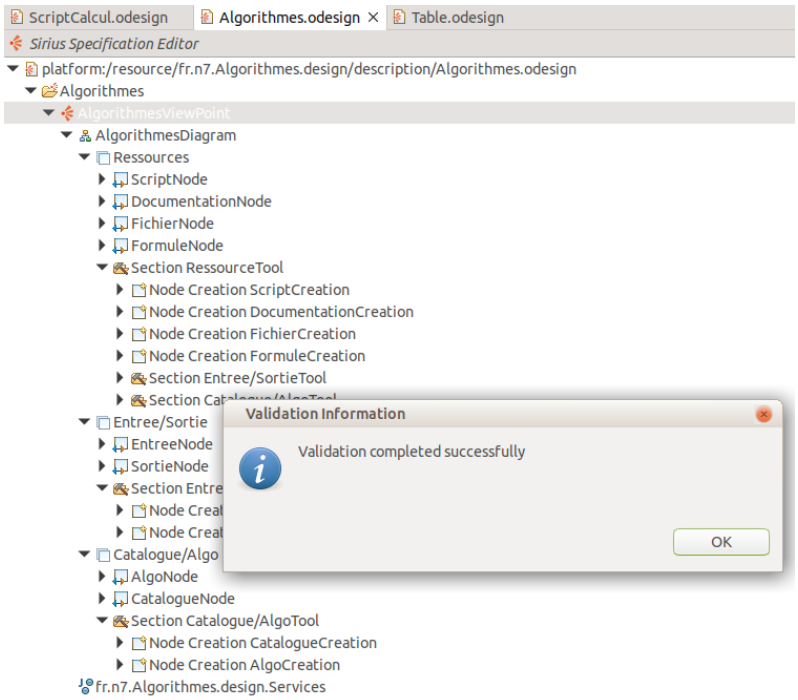

FIGURE 7 – Validation Algorithme.odesign

Les exemples créés s'ouvrent correctement sous une forme graphique, avec la possibilité de modifier les attributs définis dans le méta-modèle. Les algorithmes introduits dans les exemples sont affichés de manière claire, et il est possible de leur associer une description détaillant leur fonctionnement. De plus, des scripts, formules ou fichiers peuvent être ajoutés pour définir précisément le comportement des algorithmes.

Cependant, un problème similaire à celui rencontré pour ScriptCalcul se pose lors de la création des différents éléments. Si un algorithme est déjà déclaré dans l'exemple utilisé, ses entrées et sorties sont définies correctement.

Toutefois, une fois l'interface graphique lancée, il est impossible de définir de nouvelles entrées ou sorties à l'algorithme via cette interface. Cette restriction peut s'avérer problématique, car elle limite l'adaptabilité du modèle lors des modifications ou des ajustements nécessaires une fois l'interface lancée.

3.4 Point de vue pour ScriptCalcul

3.4.1 ScriptCalcul.odesign

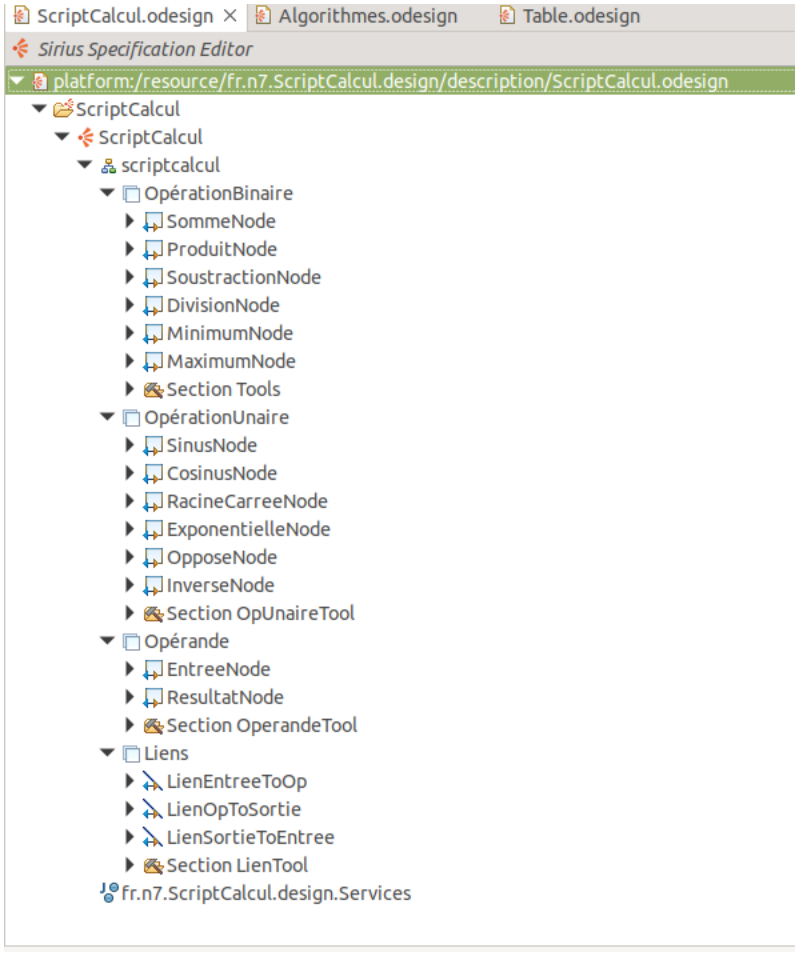

FIGURE 8 – ScriptCalcul.odesign

Pour ScriptCalcul on a créé :

— **OperationBinaire** : un calque pour l'affichage et la création des opérations binaire dans un calcul.
  — *SommeNode* : Noeud qui représente la somme .
  — *ProduitNode* : Noeud qui représente le produit.
  — *SoustractionNode* : Noeud qui représente la soustraction.
  — *DivisionNode* : Noeud qui représente la division.
  — *MinimumNode* : Noeud qui représente le minimum.
  — *MaximumNode* : Noeud qui représente le maximum.
— **OperationUnaire** : un calque pour l'affichage et la création des opérations unaire y compris les fonctions.
  — *SinusNode* : Noeud qui représente la fonction sinus.
  — *CosinusNode* : Noeud qui représente la fonction cosinus.
  — *RacineCarreeNode* : Noeud qui représente la fonction Racine carrée.
  — *ExponnentielleNode* : Noeud qui représente la fonction exponentielle.
  — *OpposeeNode* : Noeud qui représente l'opération unaire "x -¿ -x".
  — *InverseNode* : Noeud qui représente l'opération unaire "x -¿ 1/x".

— **Opérandes** : un calque pour l'affichage et la création du catalogue qui regroupe les algorithmes et les algorithmes en question.
  — *EntreeNode* : Noeud qui représente les paramètres d'une opération/fonction dans un calcul.
  — *ResultatNode* : Noeud qui représente le résultat d'un calcul et peut être soit un résultat intermédiaire ou final.
— **Liens** : un calque pour l'affichage et la création des liens entre les différents éléments d'un script de calcul, on distingue entre :
  — *LienEntreeToOp* : un "Edge" qui représente le lien qui va d'une entrée à une opération.
  — *LienOpToSortie* : un "Edge" qui représente le lien qui va d'une opération à son résultat.
  — *LienSortieToEntree* : un "Edge" qui represente le lien qui va d'une sortie intermédiaire à l'entrée d'une nouvelle opération/fonction.

3.4.2 Résultats et Validation

Les exemples créés s'ouvrent correctement sous une forme graphique, et les attributs définis dans le méta-modèle peuvent être modifiés sans difficulté.
Les différentes opérations déclarées dans les exemples sont correctement affichées avec leurs entrées et sorties correspondantes.
De plus, les informations spécifiques à chaque opération s'affichent comme attendu, offrant une vue détaillée et cohérente de la structure des calculs.

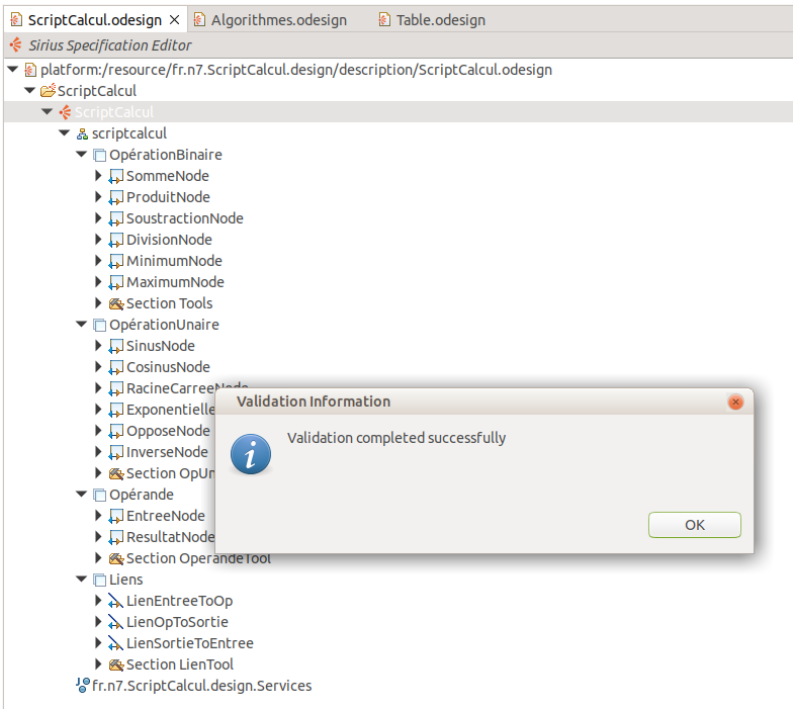

FIGURE 9 – Validation ScriptCalcul.odesign

Cependant, nous avons rencontré quelques difficultés lors de la création des opérations. Une opération peut certes être créée et affichée, mais nous n'avons pas réussi à faire fonctionner correctement les liens entre les différents éléments. Par exemple, lors de la création d'une opération, il n'est pas possible d'ajouter directement une entrée ou une sortie.

En revanche, si l'opération est déjà définie dans l'exemple chargé dans Sirius, les entrées et sorties s'affichent correctement et sont associées aux opérations correspondantes. Les modifications survenant une fois l'interface graphique lancée ne sont ainsi pas toutes fonctionnelles.

4 Génération de code avec Acceleo

Nous exploitons les modèles générés par l'utilisateur pour produire un code Python conforme à ses attentes à l'aide de l'outil **Acceleo**.

4.1 ToPython.mtl

Le fichier *ToPython.mtl* contient les *templates* nécessaires à la transformation des modèles en code Python, permettant de traduire automatiquement la structure et la logique des modèles en un script exécutable.

4.1.1 Structure du fichier

Plusieurs éléments structurent le fichier *.mtl*. Les plus importants sont les suivants :

— **Module :** Le fichier commence par la déclaration du module et des URI des modèles cibles, indiquant les classes et attributs concernés par la transformation.
— **Templates :** Ce sont des unités de transformation associées aux éléments spécifiques du modèle (classe, attribut, etc.). Un template principal contrôle l'exécution globale de la transformation, souvent en fonction du type d'élément du modèle.
— **Expressions et Contrôles :** Les boucles (*[for ...]*) permettent de parcourir les éléments définies dans les modèles. Les conditions (*[if ...]*) permettent de personnaliser le contenu généré en fonction de critères spécifiques liés aux propriétés des éléments du modèle.

4.1.2 Fonctionnalités du script généré

Le script Python généré par le fichier *ToPython.mtl* suit les étapes suivantes :

1. **Importation des données :** Le script commence par importer les bibliothèques nécessaires (*math*, *pandas*), puis lit un fichier CSV portant le même nom que la table spécifiée dans le modèle.
2. **Gestion des colonnes :** Une liste de toutes les colonnes de la table (standards, externes, dérivées, etc.) est générée. Pour chaque colonne dérivée, l'algorithme associé est traduit en une fonction Python, en utilisant les ressources et entrées définies dans le modèle. Les formules et fonctions mathématiques, telles que *Somme*, *Produit*, ou *Sinus*, sont générées à partir des modèles correspondants (*ScriptCalcul*).
3. **Ajout des colonnes dérivées :** Les colonnes dérivées calculées par les fonctions générées sont ajoutées au tableau en tant que nouvelles colonnes.
4. **Exportation des résultats :** Enfin, le tableau mis à jour est sauvegardé dans un fichier *CSV* nommé *result.csv*.

4.2 Description de l'exemple

Pour illustrer le fonctionnement, nous avons conçu un exemple basé sur une table nommée *TableVols.table*, contenant des informations sur les vols.

— **Colonnes standards :** Ces colonnes incluent les informations de base, telles que les horaires et les numéros de vol.
— **Colonnes externes :** Certaines colonnes sont reliées à une autre table appelée *TableComvols.table*.
— **Colonne dérivée : Poids_Somme :** Cette colonne représente la somme des deux colonnes *"Poids des bagages en cabine (kg)"* et *"Poids des bagages en soute (kg)"*. Elle est calculée à l'aide de l'algorithme *Somme_Colonnes.algorithme*, qui s'appuie sur le script calcul *Somme.script*.

4.3 Résultats et Validation

Le fichier Python généré *Exemple_Vols.py* est placé dans le dossier *toPythonFinal*. Cependant, après génération, plusieurs problèmes ont été constatés :

— Seule l'initialisation des colonnes et une fonction vide *funPoids_total* sont présentes dans le fichier généré.

— Les calculs associés à la colonne dérivée *Poids_Somme* ne sont pas implémentés correctement.

4.3.1 Améliorations nécessaires

Pour corriger ces problèmes, les améliorations suivantes ont été mises en place :

1. **Vérification des modèles :** Les entrées, sorties et algorithmes associés aux colonnes dérivées ont été revus pour garantir leur conformité avec les spécifications du modèle.
2. **Correction des templates :** Les templates responsables des calculs (*codeElementCalcul*, *codeOperation*, etc.) ont été également revus pour éviter la génération de fonctions vides.

4.3.2 Résultats attendus après correction

Après application des corrections, le fichier Python généré doit contenir :

— Une fonction *funPoids_Somme* fonctionnelle, prenant en entrée les colonnes nécessaires et calculant correctement la somme des poids.

— L'application de cette fonction sur les données du tableau CSV, avec le résultat ajouté comme une nouvelle colonne.

— Un fichier final *result.csv* contenant toutes les colonnes, y compris la colonne dérivée *Poids_Somme*. Cependant, même après correction le problème persiste.

5 Génération de graphes avec Pandas

5.1 ScriptTracerGraphique.py

Afin de permettre au client de visualiser les données sous forme graphique, nous avons développé un script Python nommé *ScriptTracerGraphique.py*. Ce script permet de modéliser les données sous forme de graphiques et d'histogrammes à partir des informations contenues dans un fichier CSV.

Le script s'appuie sur la bibliothèque **Pandas**, qui permet de convertir les données du fichier CSV en un **DataFrame**. Cette structure de données facilite les opérations sur les lignes et les colonnes, telles que le filtrage, la transformation ou l'agrégation des données. Une fois les données préparées, nous utilisons la bibliothèque **Matplotlib** pour générer des graphiques visuels à partir des données traitées.

5.2 Résultats et Validation

```
pc-maxime@pcmaxime-Latitude-3420:~/projet-IDM/graphiques$ python3 scriptTracerGraphique.py

Entrez le chemin du fichier CSV : Exemple_vols.csv
Fichier chargé avec succès. Colonnes disponibles :
Index(['Code du vol', 'Durée du vol (heures)', 'Nombre de passagers',
       'Poids des bagages en cabine (kg)', 'Poids des bagages en soute (kg)',
       'Revenus totaux (€)', 'Carburant (€)', 'Maintenance (€)',
       'Personnel (€)', 'Autres frais (€)', 'Coût total (€)',
       'Profit brut (€)', 'Rentabilité par vol (%)',
       'Revenu par passager (RPP, €)', 'Coût par passager (CPP, €)',
       'Marge brute (%)', 'Rendement par siège (€)', 'Taux de remplissage (%)',
       'Profit net (€)', 'Coût d'opération par heure (€)'],
      dtype='object')

Colonnes disponibles :
0: Code du vol
1: Durée du vol (heures)
2: Nombre de passagers
3: Poids des bagages en cabine (kg)
4: Poids des bagages en soute (kg)
5: Revenus totaux (€)
6: Carburant (€)
7: Maintenance (€)
8: Personnel (€)
9: Autres frais (€)
10: Coût total (€)
11: Profit brut (€)
12: Rentabilité par vol (%)
13: Revenu par passager (RPP, €)
14: Coût par passager (CPP, €)
15: Marge brute (%)
16: Rendement par siège (€)
17: Taux de remplissage (%)
18: Profit net (€)
19: Coût d'opération par heure (€)
```

FIGURE 10 – Vue 1 sur le terminal

Le script commence par demander à l'utilisateur de spécifier le fichier CSV qu'il souhaite analyser. Ensuite, il affiche le nom de chaque colonne du fichier pour faciliter la sélection des données à visualiser.

```
Entrez l'indice de la colonne pour l'axe X : 2
Entrez l'indice de la colonne pour l'axe Y : 10
Graphique enregistré : plot_Nombre de passagers_vs_Coût total (€).png
```

FIGURE 11 – Vue 2 sur le terminal

À ce stade, le script invite l'utilisateur à choisir les colonnes à utiliser pour les axes des graphiques : une pour l'axe des abscisses (X) et une autre pour l'axe des ordonnées (Y).
Une fois les colonnes sélectionnées, le script génère automatiquement un graphique sous forme de fichier image *.png*.

Par exemple, dans ce rapport, nous avons utilisé le script pour créer un graphique représentant le coût total en euros en fonction du nombre de passagers. Ce graphique permet de visualiser facilement les relations entre ces deux variables.

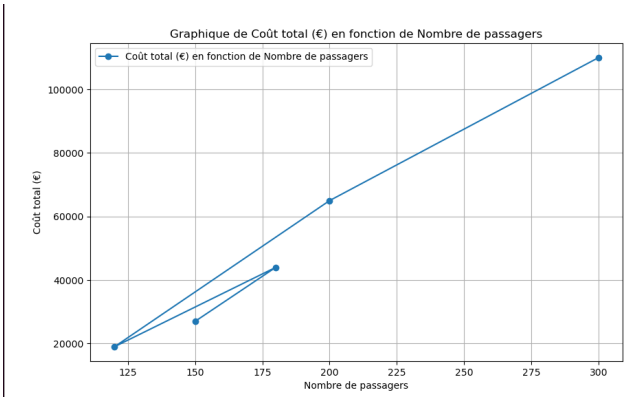

FIGURE 12 – graphique de coût total en fonction du nombre de passagers

En plus des graphiques linéaires, le script permet également de générer des histogrammes. Nous présentons ci-dessous des exemples d'analyse graphique réalisés avec le script, tels que l'évolution du profit brut par code de vol et l'histogramme de la rentabilité par vol, exprimée en pourcentage, en fonction de la rentabilité.

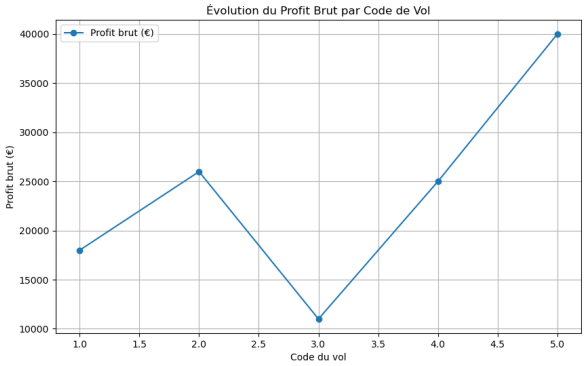

FIGURE 13 – Courbe d'évolution du profit brut par code de vol

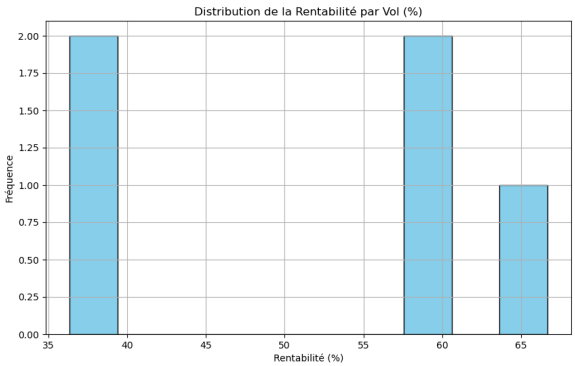

FIGURE 14 – Histogramme de distribution de la rentabilité par vol

5.3 Améliorations possibles

Plusieurs pistes d'amélioration peuvent être envisagées pour enrichir ce script :

— **Ajout de nouveaux types de graphiques** :
Il serait pertinent d'ajouter la possibilité de créer d'autres types de graphiques, tels que des graphique circulaires (*camembert*) ou des boîtes à moustaches (*box plot*), permettant ainsi une meilleure visualisation de la répartition des données.

— **Intégration avec le reste du projet** :
Afin de centraliser et simplifier l'analyse visuelle des données il faut lier ce script à d'autres parties du projet, notamment pour automatiser la génération de graphiques à partir des résultats obtenus par d'autres modules,

6 Exemple de Gestion des données financières et logistiques d'une compagnie aérienne

6.1 Présentation et Motivations

Le suivi financier et logistique des vols d'une compagnie aérienne permet de gérer efficacement les coûts et les revenus associés à chaque vol. En tant qu'élèves ingénieurs en Master 1 HPC et Big Data à l'ENSEEIHT Toulouse, une école bien ancrée dans le domaine aéronautique, nous proposons un environnement de calcul destiné à faciliter et à rendre plus accessible ce suivi.

Cet environnement est basé sur une table principale qui comporte des colonnes essentielles à la gestion logistique : le code du vol, la date du vol, la durée du vol (avec ou sans escale), le type d'aéronef, le nombre de passagers, le poids des bagages, les revenus totaux (billets, suppléments et amendes), ainsi que les différents coûts liés à chaque vol comme le carburant, la maintenance, le personnel, etc. Ces colonnes permettent de déduire les données financières de la compagnie aérienne, telles que le coût de chaque vol, le profit brut et la rentabilité.

6.2 Colonnes de la table principale

Exemples de colonne de notre table :

— **Code du vol** : Un identifiant unique pour chaque vol.
— **Date du vol** : La date à laquelle le vol est effectué.
— **Durée du vol** : Temps estimé de vol, incluant ou non des escales.
— **Nombre de passagers** : Lié aux revenus générés par la vente des billets.
— **Poids des bagages en cabine** : Moins d'impact sur la consommation de carburant, mais lié aux revenus en cas de suppléments.
— **Poids des bagages en soute** : Directement lié au poids de l'avion et donc à la consommation de carburant.
— **Revenus totaux** : Incluent les ventes de billets, les suppléments divers (bagages supplémentaires), les amendes (ex. bagages dépassant les limites de poids) et les ventes à bord (produits hors taxes, repas, boissons ou autres services spécifiques).
— **Coûts spécifiques** : Comprennent le carburant, la maintenance, le personnel, ainsi que d'autres frais associés à chaque vol.

Ces colonnes servent de base à des calculs financiers essentiels pour la compagnie aérienne, tels que :

— **Coût total par vol** : La somme de tous les frais associés.
— **Profit brut** : La différence entre les revenus totaux et les coûts spécifiques.
— **Rentabilité par vol** : Calculée en pourcentage, elle évalue l'efficacité économique du vol.
— **Revenu par passager (RPP)** : $\text{RPP} = \frac{\text{Revenus totaux}}{\text{Nombre de passagers}}$.
— **Coût par passager (CPP)** : $\text{CPP} = \frac{\text{Coût total}}{\text{Nombre de passagers}}$.
— **Taux de remplissage (Load Factor)** : $\text{Load Factor} = \left(\frac{\text{Nombre de passagers}}{\text{Nombre de sièges disponibles}}\right) \times 100$.
— **Marge brute (Gross Margin)** : $\text{Marge brute} = \left(1 - \frac{\text{Coût total}}{\text{Revenus totaux}}\right) \times 100$.
— **Rendement par siège (Yield)** : $\text{Yield} = \frac{\text{Revenus totaux}}{\text{Nombre de sièges disponibles}}$.
— **Profit net** : Profit net = Profit brut – Coûts indirects.
— **Coût d'opération par heure de vol** : Évalue l'efficacité des coûts en fonction de la durée du vol.

6.3 Contraintes sur les données

— Le **code du vol** doit être alphanumérique, unique, non nul et respecter un format valide.
— Le **nombre de passagers** doit être positif et inférieur ou égal au nombre de sièges disponibles.
— Le **poids des bagages** doit respecter les limites autorisées.
— La **durée du vol** doit se situer dans un intervalle logique.
— La **date du vol** ne peut pas être dans le passé par rapport à la date actuelle.
— Les calculs financiers doivent respecter les règles définies pour éviter les incohérences.

6.4 Conclusion globale

Ce projet nous a permis de développer un environnement de calcul basé sur une approche d'ingénierie dirigée par les modèles. À travers les différentes phases, nous avons conçu des méta-modèles, développé des interfaces graphiques adaptées à ceux-ci avec Sirius, et intégré des outils de génération de code et de visualisation des données. Ces contributions étaient essentielles pour répondre aux spécifications précises du projet, notamment pour obtenir une bonne gestion de l'exemple que nous avons proposé, à savoir l'analyse de données financières et logistiques d'une compagnie aérienne.

Les résultats obtenus démontrent la bonne implémentation de nombreux points concernant notamment une bonne représentation graphique ainsi qu'une génération de code satisfaisante. Cependant, nous avons également pu relever certaines limitations, telles que l'impossibilité de définir de nouvelles entrées ou sorties via l'interface graphique. Ces points nécessitent une revue plus approfondie pour identifier l'origine du problème. De plus, un travail reste nécessaire pour terminer de lier les différentes parties de ce projet ensemble notamment concernant la visualisation des données proposée au point 5. .

En conclusion, ce projet nous a permis d'améliorer nos connaissances dans la mise en œuvre pratique des concepts d'ingénierie dirigée par les modèles. Il offre une base solide pour la conception d'applications et ouvre des perspectives d'amélioration, notamment concernant l'implémentation de certaines des fonctionnalités existantes.

7 Ressources

Dans le cadre de ce projet, plusieurs ressources ont été consultées pour enrichir notre compréhension et construire nos modèles. Voici les principales sources d'information utilisées :

- **Cirium** – The world's most trusted source of aviation analytics.
  `https://www.cirium.com/`
- **Case Study : Madrid to Beijing Flight** – Air Transport Management.
  Une étude de cas analysant les paramètres économiques et logistiques d'un vol long-courrier.
  `https://airtransportmanagement.org/content/`
- **Investors — Financial Results & Annual Reports — Airbus**
  `https://www.airbus.com/en/investors/financial-results`
  Rapports financiers et analyses des coûts et revenus des opérations aériennes, publiés par Airbus, une référence clé dans l'industrie aéronautique.