Projet IDM

BONCOUR Maxime
CORSETTI Theo
EL MOSSLIH Amal
ROCHDI Adam

Année: 2024/2025

Sommaire

List of Figures

1 Introduction

Ce projet propose une suite d'outils basée sur la plateforme Eclipse et les technologies EMF, permettant de définir des schémas de données et des calculs automatisés associés. À partir d'un schéma de table défini par l'utilisateur, incluant des colonnes et leurs relations, des outils spécifiques sont générés pour importer, visualiser, exporter et valider des données.

Les calculs, exprimés via des scripts dans des langages comme Python, permettent d'automatiser des opérations sur les données, tandis que des contraintes personnalisées assurent leur qualité. Cette approche flexible génère des outils adaptés aux besoins des utilisateurs finaux, simplifiant la gestion et l'exploitation des données complexes.

Le travail se fera en suivant ces étapes :

- Définition des méta-modèles avec Ecore .
- Spécification des règles de cohérence avec OCL (implantation en Java).
- Manipulation des modèles avec l'infrastructure EMF.
- Conception de syntaxes concrètes graphiques via Sirius.
- Définition des transformations de modèle à texte (M2T) nécessaires avec Acceleo.

2 Définition des métamodèles avec Ecore

2.1 Table

Le métamodèle conçu pour ce projet repose sur une structure hiérarchique et modulaire permettant de représenter les éléments essentiels pour gérer des schémas de tables et des calculs automatisés. Cette section décrit les composantes principales.

2.1.1 Structure générale

La classe principale `Table` est au centre du métamodèle. Elle représente une table de données et inclut les éléments suivants :

- Un **nom** (`nom`, de type `EString`) permettant d'identifier la table.
- Une référence vers les tables sources (`tablesource`) et résultantes (`tableresultante`), définies comme des relations de composition multiples.
- Une association à des scripts (`script`) pour spécifier les transformations ou calculs dérivés.

2.1.2 Colonnes et types

La gestion des colonnes repose sur une hiérarchie de classes :

- La classe abstraite `Colonne` définit les propriétés communes :
  - `nom` : le nom de la colonne.
  - `id` : un identifiant unique pour les références croisées.

  - `valeurs` : une liste de valeurs associées.
  - `contrainte` : une liste de contraintes applicables à la colonne.

- Les classes concrètes spécialisent les colonnes :
  - `ColonneId` : représente une colonne identifiante.
  - `ColonneSaisie` : capture des données saisies.
  - `ColonneDeriv` : définit des colonnes dérivées à l'aide de scripts.

2.1.3 Contraintes

Les contraintes sur les colonnes sont modélisées via la classe abstraite `Contrainte` et ses spécialisations :

- `ContrainteOperation` applique des règles spécifiques basées sur un type d'opération (`OpContrainte`, comme `SUP`, `INF`, `EGAL`) et une valeur de comparaison (`vComparee`).

Ces contraintes garantissent la cohérence des données importées ou calculées.

2.1.4 Scripts et calculs

Les calculs sont modélisés à l'aide de la classe `Script`, qui inclut :

- Des références d'entrée (`entree`) et de sortie (`sortie`) pour lier les colonnes concernées, la sortie d'un script est une colonne dérivée.
- Un **nom** permettant d'identifier le script.

Cette structure facilite l'intégration de scripts dans des langages comme Python ou Matlab.

2.1.5 Tables sources et résultantes

- La classe `TableSource` regroupe les colonnes saisies (`colonnesaisie`) et les colonnes identifiantes (`colonneid`). Elle représente les données brutes importées.
- La classe `TableResultante` inclut les colonnes identifiantes, les colonnes dérivées et les colonnes saisies après transformation pour permettre la réutilisation des tables.

2.1.6 Flexibilité et extensibilité

Le métamodèle est conçu pour être modulaire et réutilisable grâce à :

- L'utilisation de références croisées et de relations de composition.
- Une définition flexible des calculs via des scripts externes et des contraintes personnalisables.

Cette conception permet d'adapter le métamodèle à divers besoins et garantit la qualité des données manipulées.

Le schéma du métamodèle est le suivant:

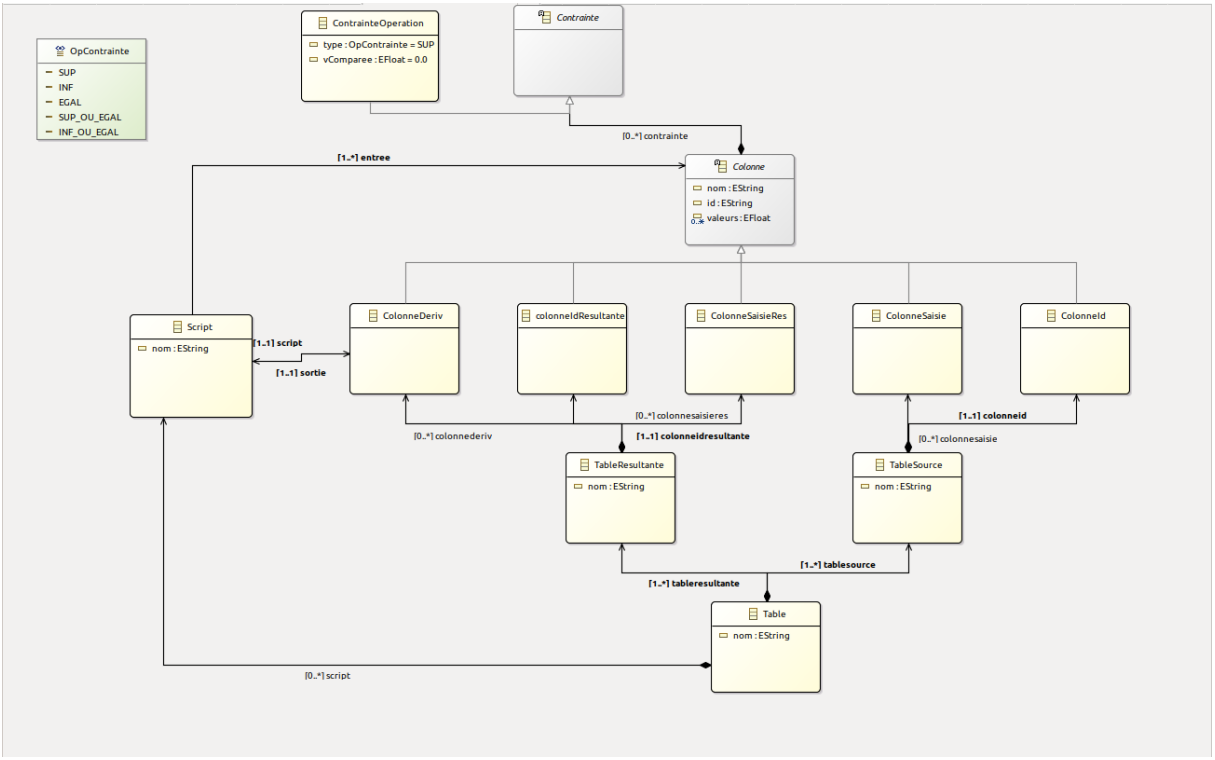

Figure 1: Le métamodèle Table

2.2 Calcul

Ce métamodèle a pour objectif de générer des scripts qui seront utlisés afin de créer des colonnes résultantes pour nos tables. Nous devons donc garder pour objectif la création d'un script Python via Acceleo de notre métamodèle.

2.2.1 Structure générale

La classe principale `Calcul` est ce que nous voulons transformer en scripts. Elle représente un calcul complexe et inclut les éléments suivants :

- Un **nom** (`nom`, de type `EString`) permettant d'identifier le calcul.
- Une ou plusieurs opérations qui représentent les "actions" réalisées par notre calcul (`operateur`).

2.2.2 Opérateur

Les opérateurs reposent sur une classe abstraite, et des classes concrètes qui les spécialisent :

- La classe abstraite `Operateur` définit les propriétés communes :
  - `nom` : le nom de la colonne.
  - `entreePrincipale` : une liste des entrées de notre opérateur réalisant la classe abstraite `Entrée`.

  - `sortie` : la sortie unique de notre opérateur réalisant la classe abstraite `Sortie`. Si cette sortie est unique c'est dans le but de simplifiant la génération de code. Dans le cas où un utilisateur veut obtenir plusieurs sorties, il peut faire plusieurs calculs en utilisant la sortie d'un autre calcul.

- Les classes concrètes spécialisent les opérateurs :
  - `OperateurBinaire` : représente un opérateur à deux entrées.
  - `OperateurUnaire` : représente un opérateur à deux entrées.
  - Ces deux opérateurs possèdent les mêmes attributs, le type de leur opérande (`operandetype`), permettant de réaliser des vérifications de typage à la validation du modèle. Ainsi que l'opération qu'ils représentent (`type`).

2.2.3 Entrée

Les entrées des différents opérateurs pouvant être de nature multiple, nous avons décicidé d'implémenter la classe abstraite `Entree` et ses spécialisations :

- La classe abstraite `Entree` définit les propriétés communes :
  - `ordre` : L'ordre dans l'opération de l'entrée en question. Ceci permet de créer un sens de lecture des opérations.
  - `type` : Le type de la valeur de l'entrée, il permet de faire des validations de typage.
- Les classes concrètes spécialisent les entrées :
  - `PortEntree` : Les entrées globales du calcul, celles qui seront plus tard utilisées pour être les colonnes de notre table.
  - `EntreeOperateur` : Une entrée qui provient de la sortie d'un autre opérateur.
  - `EntreeConstante` : Une classe abstraite dont les entrées de constante hérite.
  - `EntreeConstante<type>` : Contient une `valeur` constante du type indiqué dans le nom de la classe.

2.2.4 Sortie

Les sorties des différents opérateurs pouvant être de nature multiple, nous avons décicidé d'implémenter la classe abstraite `Sortie` et ses spécialisations :

- Elle est héritée par :
  - `PortSortie` : La sortie globale du calcul.
  - `SortieOperateur` : Une sortie qui provient de l'entrée d'un autre opérateur.

2.2.5 Flexibilité et extensibilité

Le métamodèle est conçu pour être modulaire et réutilisable grâce à :

- L'ajout potentiel d'autres opérateurs qui peuvent hériter de la classe `Operateur`. Cependant, si le paradygme de programmation visiteur aurait pu être implémenté, il ne l'a pas était fait, il faudra donc ajouter dans les différents validateurs et objets de traitements les caractéristiques de l'opérateur.
- La possibilité de rajouter d'autres types par la modification de `TypeElement`.

Le schéma du métamodèle est le suivant:

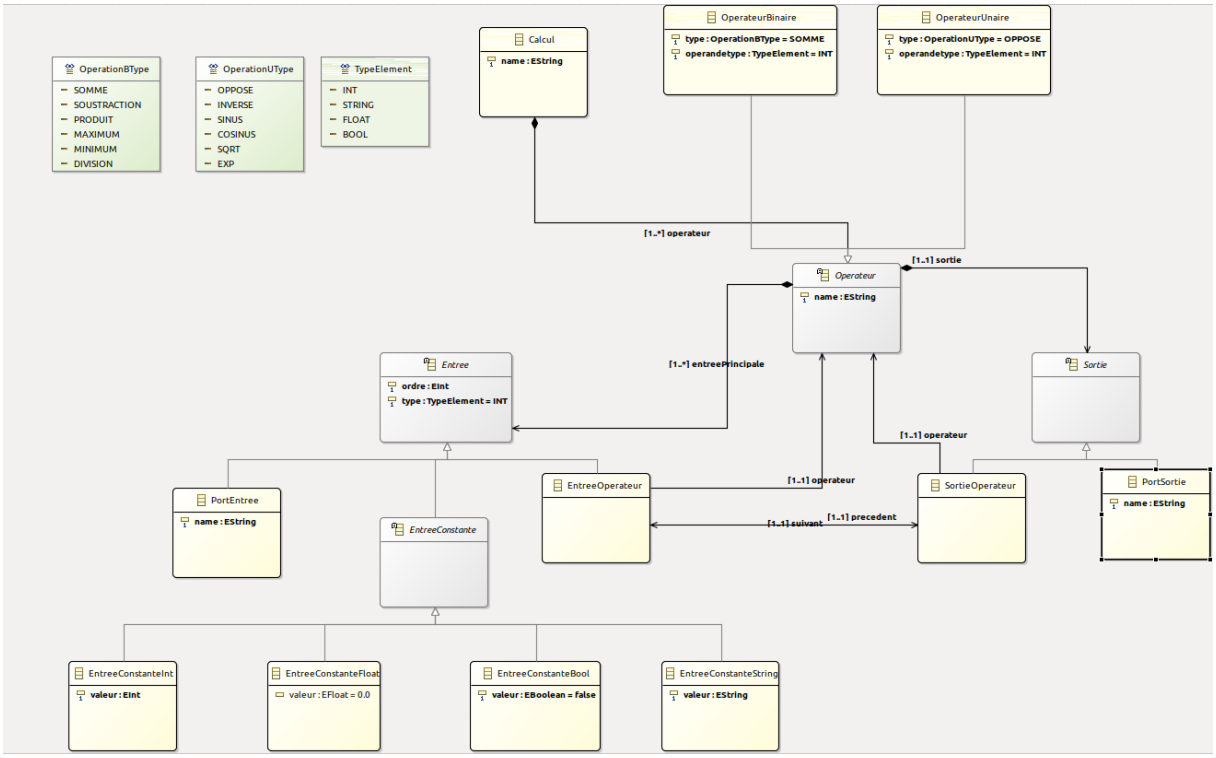

Figure 2: Le métamodèle Calcul

3 Définitions des contraintes OCL (implantées en Java)

3.1 Contraintes pour Table

Les contraintes Java pour Table sont définies comme suit :

- Les noms des tables, `tableSources` ou `tableResultantes` doivent respecter la règle suivante :

$$[A-Za-z][A-Za-z0-9]^*$$

- Les noms des tables principales (`Table`), des tables sources (`TableSource`) et des tables résultantes (`TableResultante`) doivent être uniques dans tout le modèle.

- Les identifiants (`id`) des colonnes doivent être absolument uniques dans tout le modèle. Une duplication d'identifiants, que ce soit dans les tables sources ou résultantes, déclenche une erreur.
- Les scripts (`Script`) ne peuvent pas accepter des colonnes de type `ColonneId` ou `colonneIdResultante` comme entrées. Toute violation de cette règle génère une erreur de validation.
- Tous les éléments d'une même colonne doivent avoir le même type. Si les valeurs d'une colonne ne sont pas homogènes en termes de type, une erreur est enregistrée.

3.2 Contraintes pour Calcul

Les contraintes Java pour`Calcul` sont définies comme suit :

- Les noms des calculs, `PortSortie`, `PortEntree` et `Operateur` doivent respecter la règle suivante :
$$[A-Za-z][A-Za-z0-9]^*$$
- Un `OperateurBinaire` doit posséder deux entrées, un `OperateurUnaire` doit posséder une seule entrée.
- L'ordre des entrées est unique, et maximisé par le nombre d'entrées de l'opérateur. En combinant ces deux contraintes on obtient donc que pour un opérateur binaire, les ordres sont forcéments [1,2].
- Le port de sortie est unique et forcement présent.
- Le typage est valide. Notre typage ici est simplifié, les types de toutes les entrées sont égaux, et sont égaux au type demandé par l'opérateur.

Comme un opérateur ne possède qu'une seule sortie, et qu'un port de sortie est forcément présent, il ne peut pas y avoir de boucles de calcul dans nos opérateurs.

4 Syntaxes concrètes

4.1 Syntaxe concrète graphique de Table avec Sirius

Les syntaxes graphiques sont essentielles pour rendre la visualisation des modèles plus intuitive et agréable. C'est pourquoi nous avons utilisé l'outil Sirius d'Eclipse, basé sur les technologies Eclipse Modeling (EMF), afin de générer un éditeur graphique à partir d'un modèle Ecore. Nous avons réussi à définir une syntaxe graphique pour le métamodèle défini par Ecore, permettant de représenter graphiquement, pour une table source ou une table dérivée, les colonnes ID, les colonnes saisies, les colonnes dérivées, les contraintes et les scirpts.

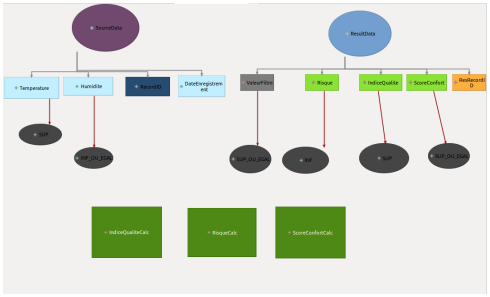

Figure 3: Syntaxe graphique de table

4.2 Syntaxe concrète graphique de Calcul avec Sirius

Pour cette partie, nous avons essayé de se rapprocher au plus au modèle qui est présenté dans le sujet. Pour cela nous avons créé des *"BorderedNodes"* sur les opérateurs unaires et binaires. Pour le choix de l'opérateur nous avons 2 boutons : créer un opérateur binaire et créer un opérateur unaire, cela correspond à notre modèle Ecore, pour le choix précis, il faudra le choisir dans les propriétés ainsi que de lui choisir un nom. Nous pouvons après choisir les entrées : entrée constante en jaune. Pour cela nous avons le choix du type de l'entrée constante directement dans la palette. En cliquant sur le bouton, nous pouvons créer ce *BorderedNode* uniquement du côté gauche (côté des entrées) et cela crée le rectangle avec la valeur, le lien et le borderedNode. Il en est de même pour les portEntrée. Pour les ports sorties, nous pouvons le positionner uniquement à droite. Pour les *BorderedNode* rouge pâle, ce sont les entrées/sorties opérateurs, qui permettent de lier 2 opérateurs entre eux. Pour cela, il y a 2 boutons pour créer les entrées et sorties et le lien se créer automatiquement en renseignant l'opérateur suivant ou précédent dans les properties de l'entrée ou sortie opérateur.

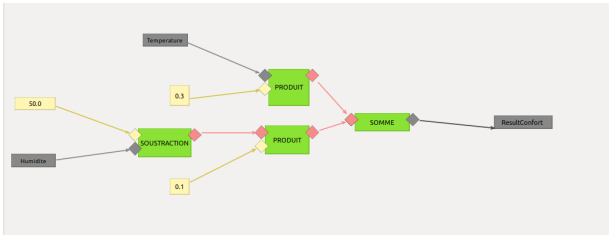

Figure 4: Syntaxe graphique de table

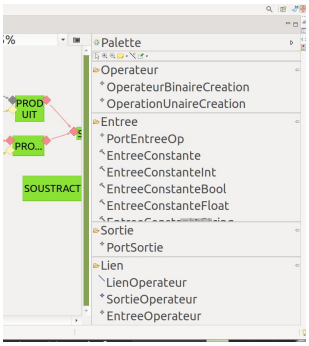

Figure 5: Palette de calcul dans Sirius

5 Génération de code

5.1 Génération des calculs

Nous utilisons l'outil Acceleo afin de générer les scripts de nos calculs en partant du modèle.

5.1.1 calcul.py

Le fichier calcul.py change de nom en fonction du nom de notre calcul. Pour parler de manière générique nous utiliserons ce nom. Ce fichier contient une fonction python qui doit prendre pour entrée les différents port d'entrée nommés correctement et retourner le port de sortie de notre metamodèle `Calcul`. Pour simplifier le contenu de ce fichier, une banque d'opérations détaillée dans la section suivante est importée.

Les simplifications de notre métamodèle, comme la présence d'une unique sortie par opérateur nous permet d'avoir une unicité du chemin de calcul au travers des opérateurs. Ainsi en partant de la sortie, nous pouvons à l'aide d'une query, calculer les entrées de notre opérateur final récursivement, et remonter tous nos opérateurs un à un. Ce choix d'une sortie unique nous permet d'éviter d'avoir à pottentiellement attendre qu'un opérateur ai fini de calcul une sortie, et donc d'avoir à utiliser des paradygmes de programmation concurrente.

On calcul donc les entrées de l'opérateur qui renvoit le port de sortie, et si une des entrées est la sortie d'un autre opérateur, on rapelle notre query sur les entrées de notre nouvel opérateur, et ainsi de suite.

5.1.2 operations.py

Ce programme implémente un module Python nommé `operations.py`, conçu pour effectuer des opérations binaires et unaires tout en respectant les types de données d'entrée (`int`, `float`, `bool`, `str`). Les opérations binaires, telles que la somme, la soustraction, la multiplication et la division, sont adaptées pour conserver ou transformer intelligemment les types. Par exemple, les chaînes de caractères sont concaténées, les booléens sont traités comme des valeurs logiques, et les entiers et flottants conservent leurs propriétés arithmétiques. Les opérations unaires incluent des fonctions mathématiques classiques comme le cosinus, le sinus, l'exponentielle et le logarithme naturel. Pour ces dernières, des mécanismes de contrôle d'erreur garantissent la validité des calculs (par exemple, éviter un logarithme de valeur négative).

Une attention particulière a été portée à la robustesse du module, notamment pour éviter les erreurs courantes comme les divisions par zéro ou les incompatibilités de type. Cette solution modulaire offre une grande flexibilité pour le traitement des données hétérogènes, tout en étant extensible pour des besoins futurs. Elle est particulièrement adaptée à des applications qui nécessitent de manipuler différentes données dans un environnement contrôlé, avec une logique intuitive adaptée aux types d'entrée.

5.2 Génération de l'interface graphique

Cette partie s'appuie sur le développement d'une interface graphique avec Python. Cet outil permet aux utilisateurs de visualiser et de manipuler facilement leurs données grâce à une interface intuitive créée avec Tkinter.

Le point fondamental de cette génération réside dans le traitement des tables dérivées. Le système crée dynamiquement une structure de données (DataFrame) pour chaque table dérivée spécifiée dans le modèle. Cette création s'effectue en respectant la structure de colonnes définie, où les valeurs sont automatiquement calculées selon les expressions établies dans la transformation des modèles calculs en scripts Python.

Afin de répondre à l'objectif principal du projet : Faciliter l'interaction avec l'utilisateur final. Nous avons défini, sous forme de boutons, différentes fonctionnalités :

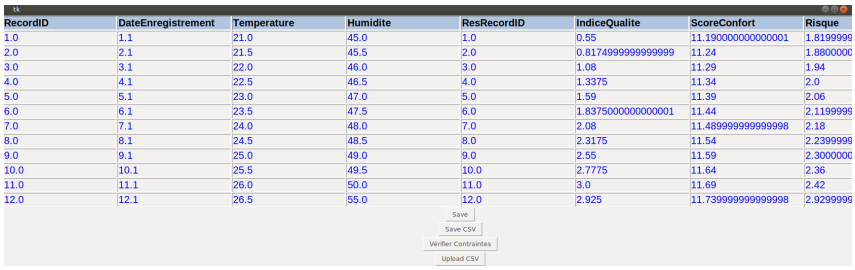

Figure 6: Interface graphique

- Import/Export de données via des fichiers CSV ;
- Sauvegarde des modifications en temps réel ;
- Vérification des contraintes définies dans le modèle.

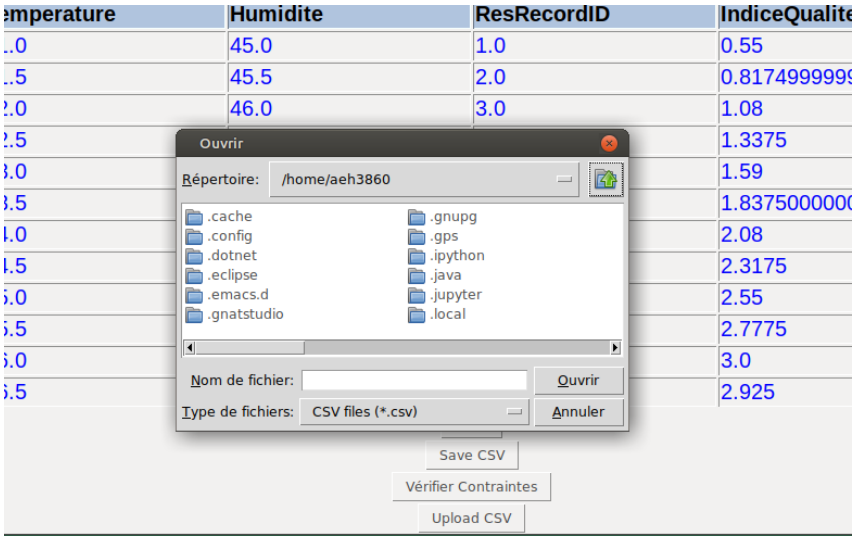

Figure 7: Uploader un nouveau CSV

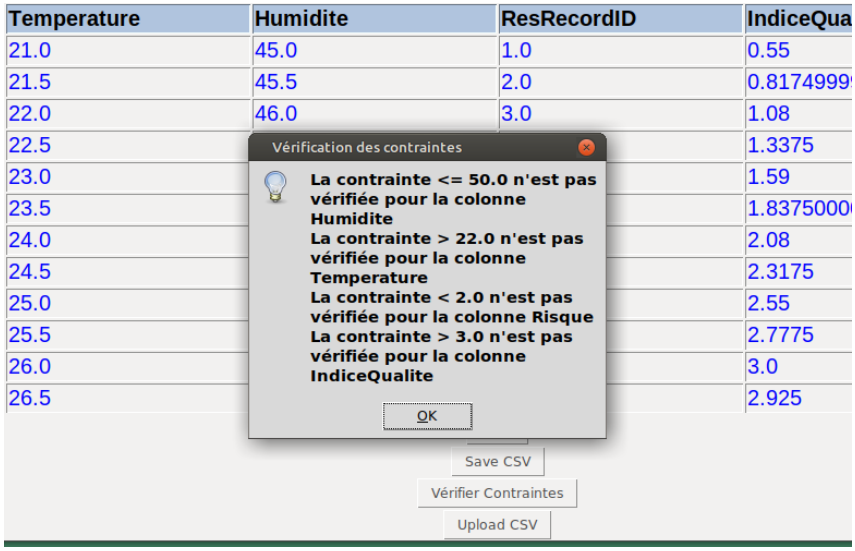

Figure 8: Vérification des contraintes

6 Conclusion

6.1 Difficultés rencontrées

Lors du développement du métamodèle Table, nous avons rencontré une difficulté majeure liée à la gestion des types de données des colonnes. Au début de notre conception, nous souhaitions affecter des types spécifiques à chaque colonne (par exemple, int, float, string) afin de garantir une gestion plus précise des données et d'assurer la cohérence des contraintes associées. Cependant, la mise en place de cette fonctionnalité s'est avérée complexe.

Nous n'avons pas réussi à définir une méthode robuste permettant d'attribuer dynamiquement des types aux colonnes dans notre métamodèle tout en respectant les contraintes de typage. En conséquence, nous avons opté pour une solution simplifiée en définissant toutes les colonnes comme des float. Bien que cette approche ait permis de finaliser le projet dans les délais, elle a limité la précision et la flexibilité du traitement des données, ce qui constitue une piste d'amélioration pour des versions futures.

6.2 Remarques personnelles

Theo Corsetti

Amal El Mosslih

Adam Rochdi

Maxime Boncour

Le sujet de ce projet était intéressant en soi, car il nous a permis d'explorer différentes facettes de la modélisation et de la programmation. Cependant, l'utilisation de l'outil Eclipse s'est révélée particulièrement frustrante. En effet, nous avons souvent été confrontés à la nécessité de réaliser des tâches beaucoup plus avancées que celles décrites dans les tutoriels disponibles sur internet. La documentation de Sirius, en particulier, est quasi inexistante, ce qui complique énormément la prise en main et la réalisation de projets complexes. À notre avis, il serait indispensable de disposer de tutoriels poussés et détaillés pour mieux exploiter le potentiel de cet outil. De plus, Eclipse donne souvent l'impression de fonctionner de manière aléatoire, avec des erreurs qui apparaissent sans explication claire et des comportements inattendus qui ralentissent considérablement le travail.

Malgré ces difficultés, nous avons trouvé très intéressant de pouvoir mettre en œuvre différents langages de programmation au cours de ce projet. Cela a enrichi notre expérience et nous a permis de mieux comprendre les interactions possibles entre les différents outils.

Par ailleurs, nous étions assez satisfaits de notre modèle Ecore. Bien que nous ayons investi beaucoup de temps dans sa conception, ce travail a considérablement facilité les étapes suivantes, notamment l'utilisation de Sirius et des autres outils. Cette préparation rigoureuse a donc été un atout majeur dans la réussite de notre projet.