Ingénierie Dirigée par les Modèles

Projet IDM : chaiseMinute

Groupe L34-2

Élèves :

LEBOBE Timothée

LECUYER Simon

SABLAYROLLES Guillaume

THEVENET Louis

Novembre 2024 - Decembre 2024

Table des matières

1. Introduction 3
2. Méta-Modèles 3
   2.1. `ChaiseMinute` 3
   2.2. `Algorithm` 4
   2.3. `Function` 4
3. Transforamtions Texte à Modèle de `ChaiseMinute` 5
   3.1. Syntaxe Textuelle 5
   3.2. Transformation M2M 5
4. Génération de la librarie et du script de calcul (Transformation M2T) 5
5. Transformations Modèle à Modèle 6
6. Edition graphique 6
   6.1. Limitation 6
7. Exemples 6
   7.1. Equation du second degré 6
8. Conclusion 10
9. Annexes 10

1. Introduction

Ce projet consiste en la réalisation d'un environnement de calcul Domaine-Scientifique nommé : ChaiseMinute.

ChaiseMinute permet à tous ses utilisateurs de développer des `schémas de tables` et des `librairies` pour traiter automatiquement des données respectant ces schémas. Ainsi l'utilisateur pourra produire des outils pour d'autres utilisateurs finaux souhaitant manipuler des données sans avoir à créer leur propres outils.

L'utilisateur de `ChaiseMinute` dipose d'une syntaxe textuelle ainsi qu'un outil graphique permettant de définir ses schémas de tables. Les schémas de tables pourront être transformés dans des langages plus spécifiques au calcul pour pouvoir transformer et vérifier des données.

2. Méta-Modèles

Nous avons décidé de séparer le modèle en trois sous-méta-modèles :

1. ChaiseMinute.Ecore
   - Point d'entrée, il permet de définir les schéma de table
2. Algorithm.Ecore
   - Définir les algorithmes utilisés pour les `ComputedColumns` et les `Constraints`
3. Function.Ecore
   - Pour spécifier les `Functions` utilisées par les `Algorithm`

2.1. `ChaiseMinute`

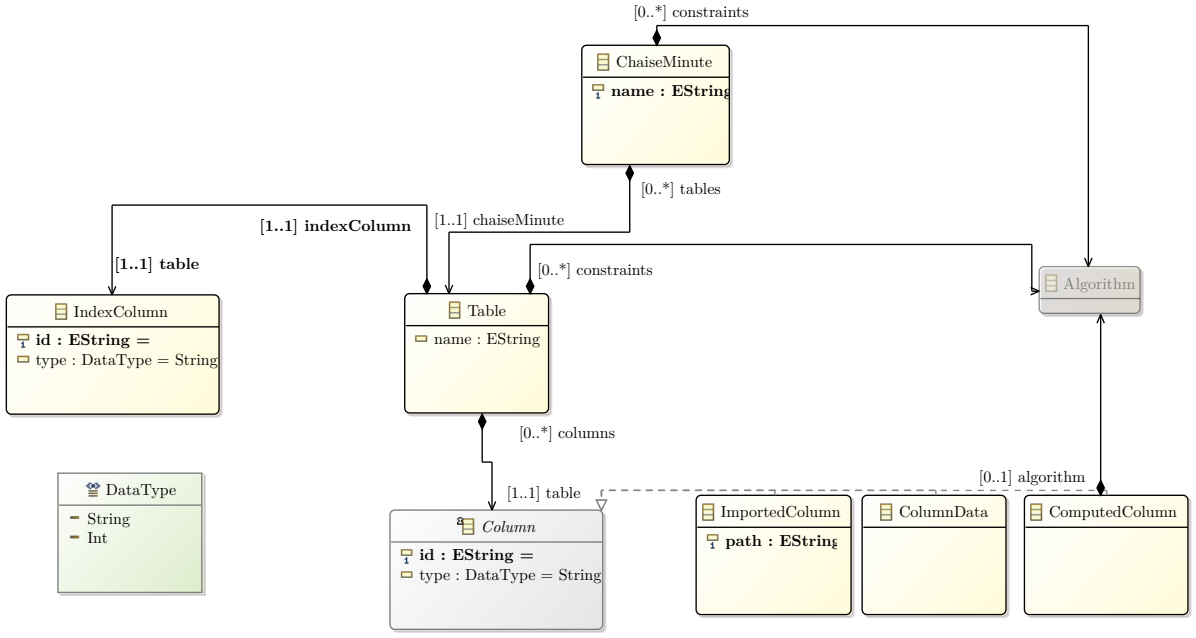


Fig. 1. – ChaiseMinute Ecore Diagram.

- `ChaiseMinute` représente le container permettant de regrouper les schémas de tables entre eux.
- `Table` défini un schéma de table comme un ensemble de `Column` et d'une `IndexColumn`, des contraintes peuvent également être ajoutées pour vérifier les données d'entrée
- `IndexColumn` : est une colonne spéciale pour les index des lignes
- `Column` est une super-classe représentant une colonne, elle contient un type de données et d'éventuelles contraintes sur les données d'entrée ou de sortie
  - `ColumnData` : une colonne simple pour représenter une donnée,

- `ImportedColumn` : une colonne provenant d'une autre `Table`,
- `ComputedColumn` : représentant une colonne calculée avec un `Algorithm`

Toutes les contraintes sont représentées par des algorithmes qui renvoient un booléen.

2.2. `Algorithm`

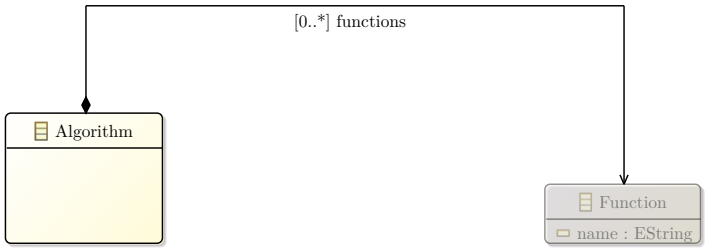


Fig. 2. – Algorithm Diagram.

`Algorithm` représente le point d'entrée d'un algorithme, il contient les fonctions qui seront appliquées.

L'application des fonctions respecte les règles suivantes :
- La première fonction est appliquée à ses arguments déclarés dans le modèle
- Les fonctions suivantes reçoivent en premier argument la sortie de la fonction précédente, puis leur arguments déclarés

Ces règles nous permettent de chaîner les fonctions dans un algorithme.

2.3. `Function`

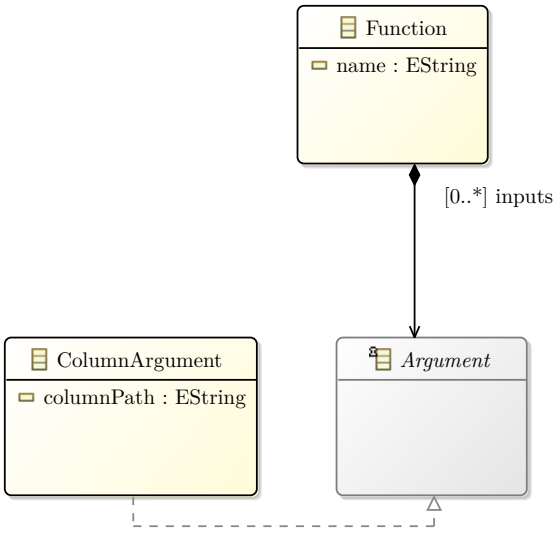


Fig. 3. – Algorith Diagram.

Une `Function` est représentée par un identifiant qui référence un programme Python, elle contient des arguments, qui sont des références vers des colonnes. Les colonnes sont représentées sous la forme `nomTable.nomColonne` dans tout le projet (pour les références croisées de Fig. 1, les arguments de fonctions, etc).

2.3.1. Limitations

On aurait pu faire en sorte que les `Function` soient des arguments, ainsi on aurait pu construire un arbre d'appels et prendre la sortie de plusieurs fonctions à la fois en arguments. On peut quand même obtenir ce résultat avec le système actuel en adaptant les fonctions Python pour qu'elles renvoient plusieurs colonnes, ainsi en adaptant la fonction suivante pour qu'elle récupère ces deux colonnes, on imite le fonctionnement d'un arbre d'appels.

3. Transforamtions Texte à Modèle de `ChaiseMinute`

3.1. Syntaxe Textuelle

En utilisant le langage Xtext, nous avons pu définir une syntaxe textuelle pour les schémas de tables comme suit :

```
nomSchemaTable {
  nomTable1 (
    nomColonne1 of typeColonne1,
    nomColonne2 of typeColonne2,
    nomColonne3 of typeColonne3 computed with [
      nomFonction1("nomTable1.nomColonne1", "nomTable1.nomColonne2") >
      nomFonction2(nomTable1.nomColonne2)
    ]
  )
  constrained by [nomFonction3("nomTable1.nomColonne1", "nomTable1.nomColonne2")]

  nomTable2 (
    nomColonne1 of typeColonne1 imported from nomTable1.nomColonne1
  )
}
constrained by [nomFonction4("nomTable2.nomColonne1" "nomTable1.nomColonne2")]
```

La syntaxe permet de définir des modèles de Table dans un méta-modèles proche de `ChaiseMinute` mais néamoins différent : `TabouretSeconde`. En effet, Xtext ne supportant pas certaines spécificités des ecore, comme les e-references, il est nécéssaire d'effecter une transformation M2M pour obtenir un modèle de `ChaiseMinute` à partir d'un modèle conforme à `TabouretSeconde`.

3.2. Transformation M2M

Pour finaliser la transformation texte à modèle, nous avons écrit une transformation modèle à modèle en Acceleo (la `menuiserie`) qui consiste simplement à rétablir les e-references et avoir un modèle conforme à `ChaiseMinute`.

4. Génération de la librarie et du script de calcul (Transformation M2T)

Nous avons décidé d'utiliser le language Python pour nos algorithmes. Ainsi, pour générer une librarie de calcul à partir d'un schéma de table, il nous suffit de générer un programme Python qui appelle les fonctions référencées par les algorithmes et les contraintes.

Nous générons aussi une fonction `main` dans notre librarie qui prend en entrée des fichier au format CSV et qui en génèrent de nouveaux en appliquant le schéma de table, résultant en un script de transformation automatique des données.

5. Transformations Modèle à Modèle

6. Edition graphique

Nous avons développé un outils graphique permettant de modifier des fichiers .cml (`ChaiseMinute`) pour modifier les différents `schémas de tables` et obtenir une visualisation plus pratique pour l'utilisateur.

Les `Tables` et les `Columns` sont visualisées comme des `containers`, des boîtes, pour montrer l'imbrication des `Columns` dans les `Tables` et la fraternités des `Columns`.

- Les `Tables` sont représentées par des `containers` verts clairs,
- Les `IndexColumn` sont représentées par des `containers` bleus clairs,
- Les `DataColumn` sont représentées par des `containers` rouges clairs,
- Les `ImportedColumn` sont représentées par des `containers` violets clairs,
- Les `ComputedColumn` sont représentées par des `containers` jaunes,
- Les `Algorithm` sont représentés par des `containers` marrons dans les `ComputedColumn`
  - Les informations dans les `containers Algorithm` affichent le nom des `Functions` qu'ils utilisent.

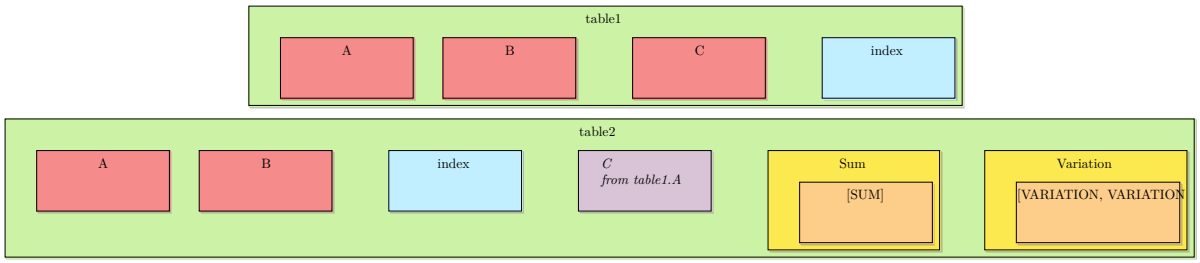


Fig. 4. – Sirius de ExempleComplique.cml.

6.1. Limitation

Il est possible pour l'utilisateur de rajouter des fonctions utiles pour une `ComputedColumn`. Cependant nous avons rencontré des difficultés à choisir des fonctions inutile. En effet, nous ajoutons et enlevons les fonctions en écrivant leur chemin dans une boite de dialogue texte mais pour l'enlever nous n'avons pas réussi à utiliser la valeur renvoyée pour vérifier si elle correspondait à une `Function` présente et donc la supprimer en conséquence.

7. Exemples

7.1. Equation du second degré

7.1.1. Modèle `TabouretSeconde` (syntaxe textuelle)

On utilise un schéma de table qui représente des polynômes du second degré dont on cherche les racines. Les données d'entrée sont un fichier CSV qui contient une colonne par coefficient (a, b, c).

```
exemple_equations {
        equations indexed on Index of Int(
        Solution of String computed with [
          compute_delta("equations.a","equations.b", "equations.c")
          > solve("equations.a","equations.b", "equations.c")
        ],
        delta of Int computed with [
          compute_delta("equations.a","equations.b", "equations.c")
        ],
        a of Int imported from "table.a",
        b of Int imported from "table.b",
        c of Int imported from "table.c"
        )
}
constrained by [ensure_is_not_null("table.a") ]
```

Liste 1. – Description textuelle du schéma de table

Les colonnes a, b et c sont importées de la table d'entrée.

Le schéma de table spécifie deux nouvelles colonnes `Solution` et `delta` qui représentent les racines et les déterminants de chaque polynôme. Elles sont produites à partir des fonctions `compute_delta` et `solve`.

```
import numpy as np

def compute_delta(a,b,c):
    return np.multiply(b,b) - 4 * np.multiply(a, c)
```

```
import numpy as np
import math

def solve(deltas, a,b,c):
    solutions = []
    for i in range(len(a)):

        if deltas[i] > 0:
            sqrt_delta = math.sqrt(deltas[i])
            solutions.append([
              (-b[i] - sqrt_delta)/(2*a[i]),
              (-b[i] + sqrt_delta)/(2*a[i])
            ])
        elif deltas[i] < 0:
            solutions.append(["No solution"])
        else:
            solutions.append([-b[i] / (2*a[i])])
    return solutions
```

Finalement, une contrainte est vérifiée sur la colonne a, dont tous les éléments doivent être non nuls.

```
import numpy as np

def ensure_is_not_null(x):
    return (np.array(x) != 0).all()
```

7.1.2. Modèle `ChaiseMinute`

On transforme ensuite ce modèle intermédiaire en `ChaiseMinute` par notre transformation ATL `cmtToCm`

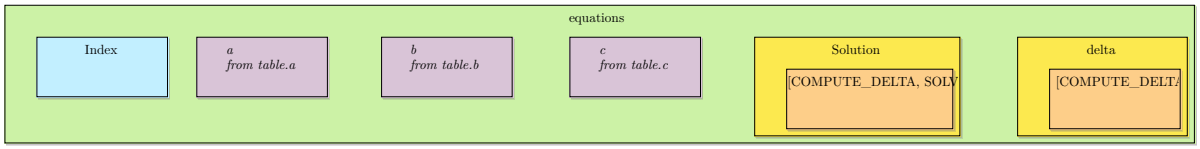


Fig. 5. – Modèle `chaiseMinute` obtenu vu sous Sirius

7.1.3. Génération de la librarie Python

On génère ensuite la librarie via la transformation modèle vers texte, un extrait est disponible ci-dessous.

La fonction `generate_output` applique le schéma de table à partir des données chargées en entrée.

```
def generate_output(input):
  """Returns a Dict<TableName, Dict<ColumnName, Data>> by applying the the model on the input data."""
  tables = {}
  out = {}
  ################################################################################
  # Table: equations
  ################################################################################
  ### Imported column: a from table.a ###
  out["a"]=input["table"]["a"]

  ### Imported column: b from table.b ###
  out["b"]=input["table"]["b"]

  ### Imported column: c from table.c ###
  out["c"]=input["table"]["c"]

  ################################
  ## Computed column: Solution
  ################################
  ### Apply compute_delta ##
  from compute_delta import compute_delta
  out["Solution"] = compute_delta(
    search(input, out, "equations", "a"),
    search(input, out, "equations", "b"),
    search(input, out, "equations", "c"),
  )
  ###############
  ### Apply solve ##
  from solve import solve
  out["Solution"] = solve(
    out["Solution"], # Previous result used in next function
    search(input, out, "equations", "a"),
    search(input, out, "equations", "b"),
    search(input, out, "equations", "c"),
  )
  ###############
  ################################
  ## Computed column: delta
  ################################
  ### Apply compute_delta ##
  from compute_delta import compute_delta
```

```
  out["delta"] = compute_delta(
    search(input, out, "equations", "a"),
    search(input, out, "equations", "b"),
    search(input, out, "equations", "c"),
  )
  ###############
  tables["equations"] = out

  return tables
```

La fonction `check_constraints` applique les contraintes et vérifie la cohérence des types.

```
def check_constraints(input):
  """Ensure all contraints are respected in input and output data"""
  ################################
  # Verifying Input Constraints
  ################################
  ### Apply ensure_is_not_null ##
  from ensure_is_not_null import ensure_is_not_null
  res = ensure_is_not_null(
    input["table"]["a"].to_list(),
  )

  if not res:
    return (False,("ensure_is_not_null constraints failed"))

  ################################
  # Verifying Table Constraints
  ################################
  out = generate_output(input)

  ### Verify data types
  for x in out["equations"]["Solution"]:
    break
  for x in out["equations"]["delta"]:
    if (type(x)!=int and type(x)!=numpy.int64):
      return (False, "Type constraints failed on equations.delta")

  for x in out["equations"]["a"]:
    if (type(x)!=int and type(x)!=numpy.int64):
      return (False, "Type constraints failed on equations.a")

  for x in out["equations"]["b"]:
    if (type(x)!=int and type(x)!=numpy.int64):
      return (False, "Type constraints failed on equations.b")

  for x in out["equations"]["c"]:
    if (type(x)!=int and type(x)!=numpy.int64):
      return (False, "Type constraints failed on equations.c")


  return (True, "Constraints are respected")
```

7.1.4. Application de la librarie

On peut finalement appeler le script automatique (fonction `main` de la librarie), qui à partir des données d'entrée au format CSV :

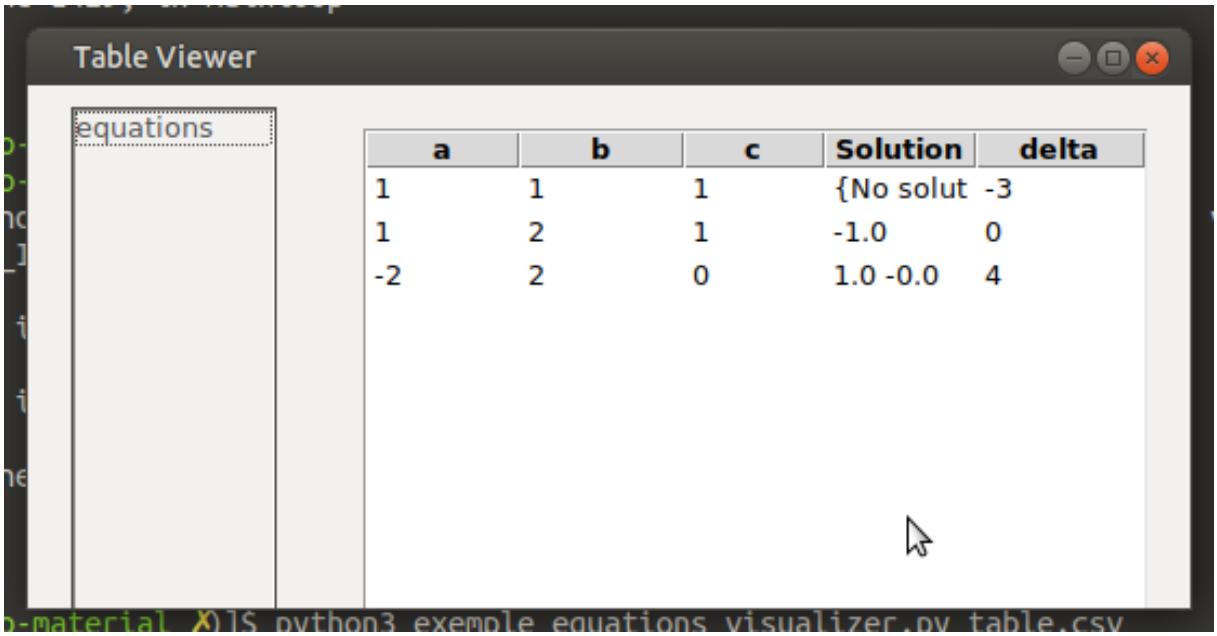
Index	a	b	c
0	1	1	1
1	1	2	1

2	-2	2	0

Produit le fichier :

Index	a	b	c	Solution	del-ta
0	1	1	1	['No solution']	-3
1	1	2	1	[-1.0]	0
2	-2	2	0	[1.0, -0.0]	4

On peut également voir les données dans l'outil de visualisation généré



8. Conclusion

9. Annexes