



Ingénierie Dirigée par les Modèles

Projet IDM : chaiseMinute

Groupe L34-2

Élèves :

LEBOBE Timothé

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	5
2.4. Calculus	6
3. Transforamtions Texte à Modèle de ChaiseMinute	7
3.1. Syntaxe Textuelle	7
4. Transformation Modèle à Modèle	7
5. Transformations Modèle à Texte	7
5.1. Librairie et script de calcul automatique	7
5.2. Outil de visualisation des données	8
6. Edition graphique	8
6.1. ChaiseMinute	8
6.2. Calculus	9
7. Exemples	11
7.1. Equation du second degré	11
8. Conclusion	14
8.1. Guillaume	14
8.2. Timothé	15
8.3. Louis	15
8.4. Simon	15
9. Annexes	15
9.1. Workspaces fr.n7.ChaiseMinute	15
9.2. fr.n7.ChaiseMinute.calculus	16
9.3. Workspaces Sirius	16
9.4. Workspaces Exemples	16
9.5. Figures Content	17

1. Introduction

Ce projet consiste en la réalisation d'un environnement de calcul Domaine-Scientifique nommé **ChaiseMinute** (c'est un tableur).

ChaiseMinute permet à tous ses utilisateurs de développer des schémas de tables et des bibliothèques 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** dispose 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, vérifier et visualiser des données.

2. Méta-Modèles

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

1. ChaiseMinute.Ecore

- Point d'entrée, il définit les schémas de table

2. Algorithm.Ecore

- Définit les algorithmes utilisés pour les colonnes calculées et les contraintes

3. Function.Ecore

- Pour spécifier les fonctions utilisées par les algorithmes

2.1. ChaiseMinute

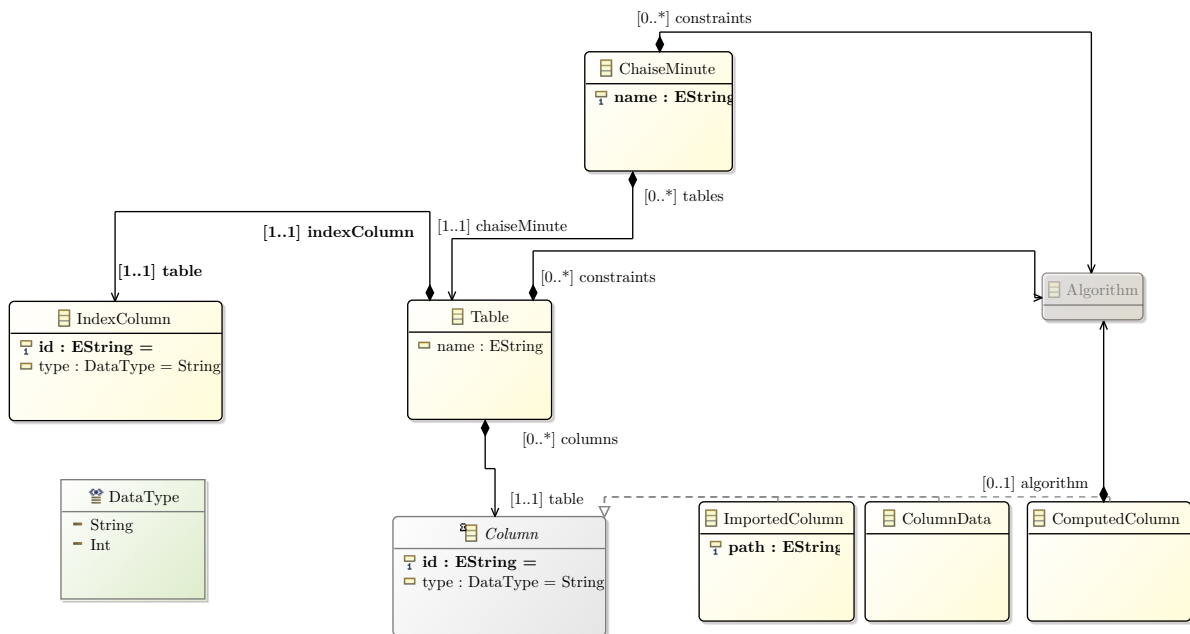


Fig. 1. – ChaiseMinute Ecore Diagram.

- **ChaiseMinute** représente le container permettant de regrouper les schémas de tables entre eux.
- **Table** définit 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.

Le méta-modèle seul ne permet pas d'éviter toute erreur que pourrait faire l'utilisateur lors de la conception d'un modèle. Nous avons donc mis en place un certains nombre de contraintes statiques décrites ci-dessous :

NomValide Le nom d'un **ChaiseMinute** doit être ni vide ni null, et respecter les conventions java

NomCorrect Le nom d'une **Table** doit respecter les mêmes conditions

NomCorrect Le nom d'une **Column** doit respecter les mêmes conditions

Typé Une **Column** doit posséder un type

NomUniqueColumn Une **Column** doit posséder un nom qui l'identifie dans la table

CheminCorrect Une **ImportedColumn** doit posséder un chemin d'accès valide vers la colonne qui est importée. C'est-à-dire que si la colonne est importée depuis une autre **Table** du **ChaiseMinute** le chemin doit être **<NomDeLaTable>.<NomDeLaColonne>**. Sinon, **<NomDeLaColonne>** car la colonne est dans la même table (duplication d'une colonne).

Il n'y a pas de contraintes statiques supplémentaire pour les **ColumnData**, et les **ComputedColumn**.

2.2. Algorithm

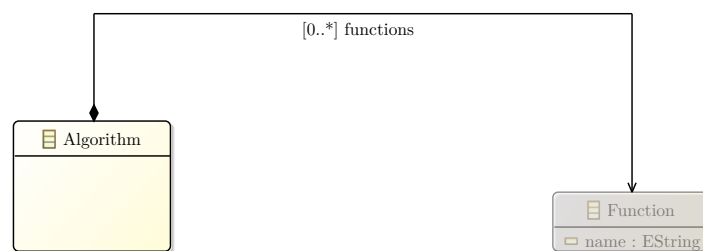


Fig. 2. – Algorithm Diagram.

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

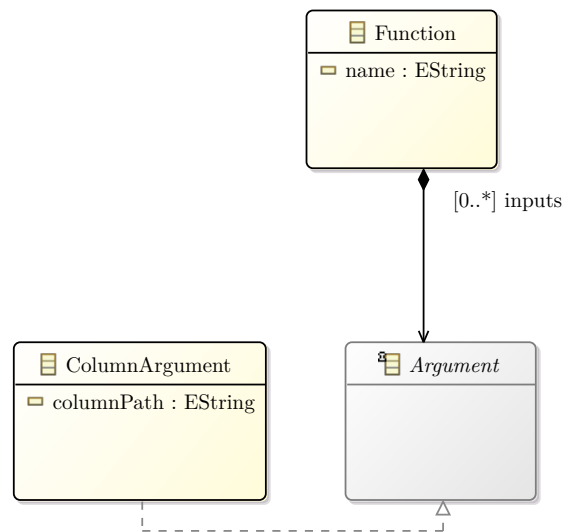


Fig. 3. – Function Diagram.

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, puis en adaptant la fonction suivante pour qu'elle récupère ces deux colonnes, on imite le fonctionnement d'un arbre d'appels.

2.4. Calculus

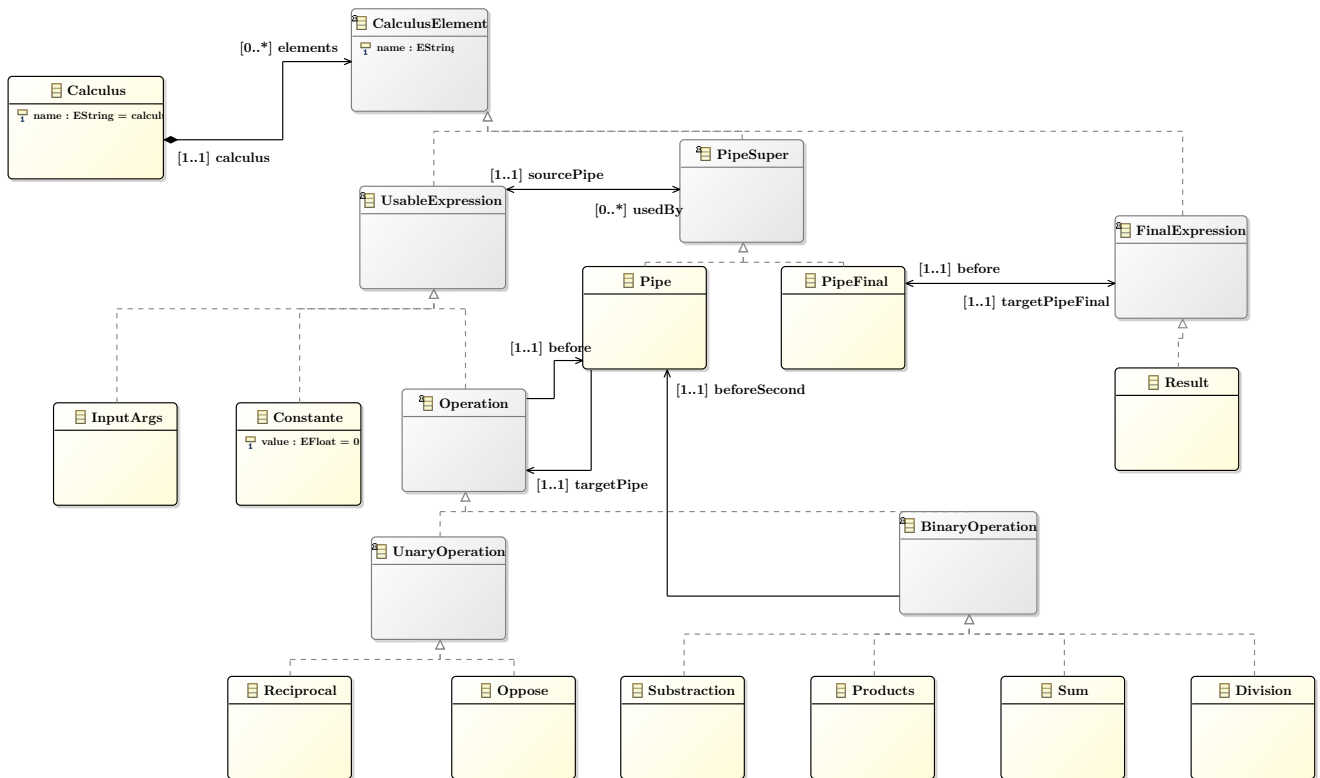


Fig. 4. – Calculus Diagram.

Calculus représente le méta-modèle des fonctions/calculs éditables à l'aide d'une syntaxe graphique par un utilisateur. Un Calculus représente un ensemble de CalculusElement qui peuvent se décomposer en trois grandes parties : UsableExpression, PipeSuper et FinalExpression.

- UsableExpression décrit les expressions pouvant être appelées pour réaliser le calcul :
 - InputArgs pour les arguments en entrée du Calculus,
 - Constante pour décrire les constantes pouvant être écrites en dure dans les suites de calculs. Ces deux premiers ne peuvent que fournir des valeurs à l'intérieur du Calculus et donc n'ont pas d'expression en entrée,
 - Operation décrit une opération possible à l'intérieur du calcul, pouvant être à deux entrées (BinaryExpression) ou à une seule entrée (UnaryExpression)
- PipeSuper représente les liens entre les UsableExpression pour décrire l'ordre des calculs et l'ordre des appels :
 - Pipe pour décrire les liens, les entrées et les sorties, entre les calculs,
 - PipeFinal indique que les calculs redirigent vers la sortie finale, la donnée retournée du Calculus,
- FinalExpression indique la fin du calcul.

2.4.1. Limitations

Nous avons donné des opérations de calcul « classique » utilisable directement par l'utilisateur : Sum, Substraction, Products, Division, Oppose et Reciprocal mais nous aurions pu rajouter des opérations mathématiques plus avancées (sin, cos, exp, modulo, ...) ou juste définir des opérations unaire ou binaire en les décrivant par un argument opération.

Les Pipe sont utiles car il permettent de définir des expressions pendantes non reliées qui sont facilement éditables ensuite mais cela rajoute du poids sur l'architecture du méta-modèles qui nous pourrions corriger dans une version future.

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 :

```
1  nomSchemaTable {
2    nomTable1 (
3      nomColonne1 of typeColonne1,
4      nomColonne2 of typeColonne2,
5      nomColonne3 of typeColonne3 computed with [
6        nomFonction1("nomTable1.nomColonne1", "nomTable1.nomColonne2")
7        > nomFonction2(nomTable1.nomColonne2)
8      ]
9    )
10   constrained by [nomFonction3("nomTable1.nomColonne1", "nomTable1.nomColonne2")]
11
12   nomTable2 (
13     nomColonne1 of typeColonne1 imported from nomTable1.nomColonne1
14   )
15 }
16 constrained by [nomFonction4("nomTable2.nomColonne1" "nomTable1.nomColonne2")]
```

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

4. Transformation Modèle à Modèle

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.

5. Transformations Modèle à Texte

5.1. Librairie et script de calcul automatique

On souhaite générer une librairie de calcul qui permet de charger et transformer des données conformes à un schéma de table.

Nous utilisons le langage Python pour la produire. Ainsi, l'utilisateur peut écrire les différentes fonctions appelées dans son schéma de table dans ce langage, la librairie leur fera appel.

La librairie propose différentes fonctions pour manipuler des données à l'aide du schéma de table.

```

1  def load(files):
2      """Load all tables as CSV files. All tables must have a corresponding CSV file
3      with the same name in working dir."""
4  def check_constraints(input):
5      """Ensure all constraints are respected in input and output data"""
6
7  def generate_output(input):
8      """Returns a Dict<TableName, Dict<ColumnName, Data>> by applying the the model
9      on the input data."""
10
11 def save_to_csv(tables):
12     """Saves each table from he input argument as a CSV file."""
13
14 def main():
15     input = load(sys.argv[1:])
16     out = generate_output(input)
17     res, msg = check_constraints(input)
18     print(msg)
19     save_to_csv(out)
20     print("Exported files")

```

Liste 1. – Signatures des fonctions de la librairie et fonction main

Nous générons aussi une fonction `main` dans notre librairie 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.

Lorsque qu'une colonne est référencée par un import ou en argument de fonction, le programme la cherchera en priorité dans les données de sortie (i.e. au sein de la table courante), puis parmi les données d'entrée.

5.2. Outil de visualisation des données

En s'appuyant sur la même librairie, on crée un outil de visualisation de schéma de table en Python. (Voir Fig. 11)

6. Edition graphique

6.1. ChaiseMinute

Nous avons développé un outil graphique permettant de modifier des fichiers `.cm` (`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.

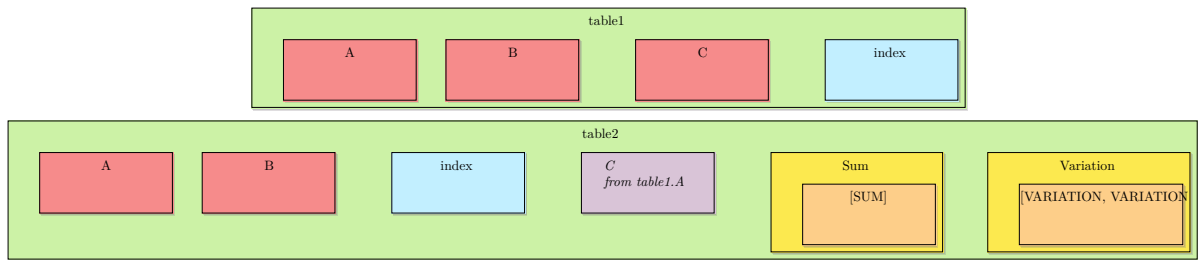


Fig. 5. – Sirius de ExempleComplice.cm.

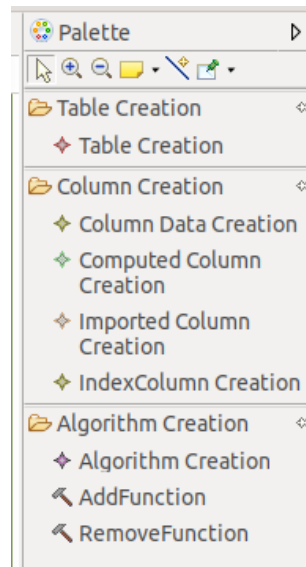


Fig. 6. – Palette de création de ChaiseMinute.

6.1.1. Limitations

Il est possible pour l'utilisateur de rajouter des fonctions utiles pour une `ComputedColumn`. Cependant nous avons rencontré des difficultés à supprimer des fonctions avec l'éditeur graphique. En effet, nous ajoutons et enlevons les fonctions en écrivant leur chemin dans une boîte 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.

6.2. Calculus

Les fichiers `.clc`, pour `Calculus`, sont éditables dans Sirius et permettent une alternative au code Python.

Les `UsableExpression` et `FinalExpression` sont représentés par des `Node` et les `Pipes` comme des `Element Based Edge` :

- Les `InputArgs` en blanc,
- Les `Constante` en orange clair avec leur `value`,
- Les `BinaryExpression` en vert clair,
- Les `UnaryExpression` en bleu clair,
- Les `Pipe` et les `PipeFinal` en gris clair,
- Les `FinalExpression` en bleu clair.

Les noms sur les `Nodes` ou les `Edges` représentent les arguments `name` des `CalculusElement` associé.

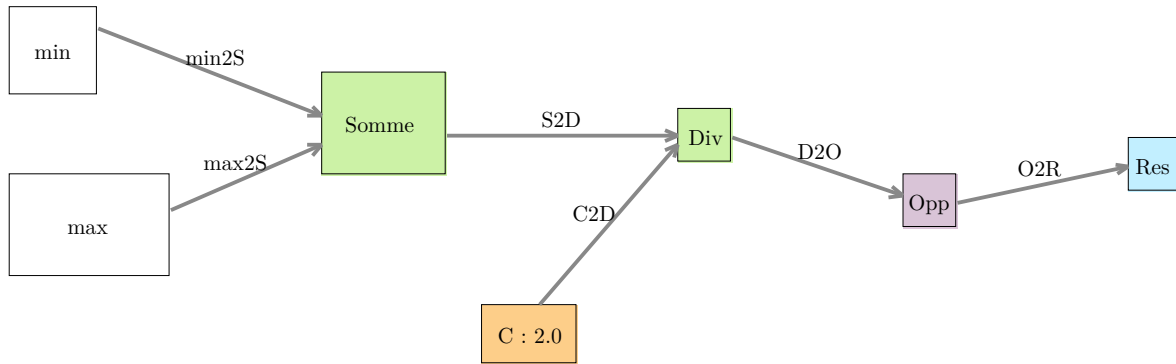


Fig. 7. – Sirius de Mean.clc.

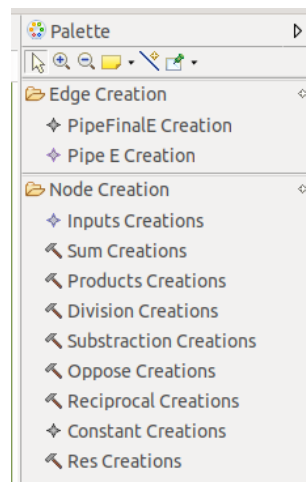


Fig. 8. – Palette de création de Calculus.

6.2.1. Limitations et améliorations

Dans l'état actuel de notre représentation graphique, nous pouvons définir et créer des `CalculusElement` avec la palette de création. Cependant, nous n'arrivons pas pour les `BinaryExpression` à définir uniquement la valeur du lien `beforeSecond` lorsque le lien `before` existe déjà. En effet, quand nous relions ce nouveau lien, les valeurs de `before` et `beforeSecond` sont définies avec cette nouvelle valeur. Nous avons isolé la partie et compris d'où venait le problème et n'arrivons pas à implémenter une solution convenable.

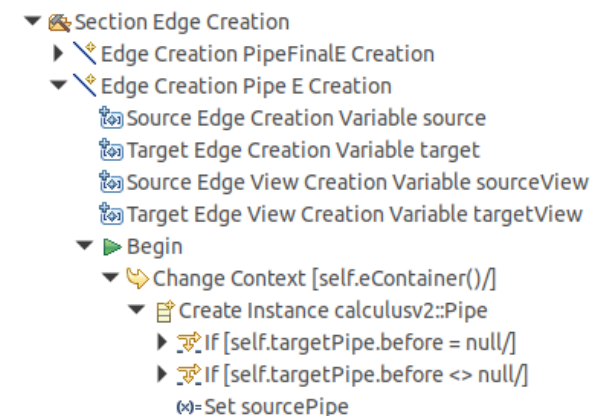


Fig. 9. – Erreur conception Pipe Edge.

Une piste d'amélioration de ce problème serait de créer des *Bordered Nodes* indiquant le **before** et **beforeSecond** pour les `BinaryExpression` pour isoler le `Pipe` à modifier et ainsi résoudre notre problème. Dans la même idée, rajouter des *Bordered Nodes* pour chaque entrée (**before/****beforeSecond**) et sortie (**targetPipe**) pour visualiser correctement le nombre d'E/S nécessaire par expression.

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).

```

1  exemple_equations {
2      equations indexed on Index of Int(
3          Solution of String computed with [
4              compute_delta("equations.a","equations.b", "equations.c")
5              > solve("equations.a","equations.b", "equations.c")
6          ],
7      delta of Int computed with [
8          compute_delta("equations.a","equations.b", "equations.c")
9      ],
10     a of Int imported from "table.a",
11     b of Int imported from "table.b",
12     c of Int imported from "table.c"
13 )
14 }
15 constrained by [ensure_is_not_null("table.a") ]

```

Liste 2. – 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`.

```

1  import numpy as np
2
3  def compute_delta(a,b,c):
4      return np.multiply(b,b) - 4 * np.multiply(a, c)

```

```

1  import numpy as np
2  import math
3
4  def solve(deltas, a,b,c):
5      solutions = []
6      for i in range(len(a)):
7
8          if deltas[i] > 0:
9              sqrt_delta = math.sqrt(deltas[i])
10             solutions.append([
11                 (-b[i] - sqrt_delta)/(2*a[i]),
12                 (-b[i] + sqrt_delta)/(2*a[i])

```

```

13         ])
14         elif deltas[i] < 0:
15             solutions.append(["No solution"])
16         else:
17             solutions.append([-b[i] / (2*a[i])])
18     return solutions

```

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

```

1 import numpy as np
2
3 def ensure_is_not_null(x):
4     return (np.array(x) != 0).all()

```

7.1.2. Modèle ChaiseMinute

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

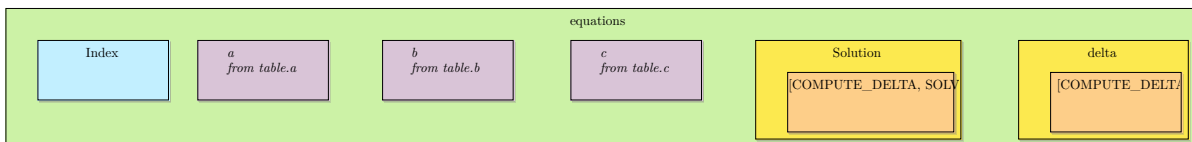


Fig. 10. – Modèle chaiseMinute obtenu vu sous Sirius.

7.1.3. Génération de la librairie Python

On génère ensuite la librairie de calcul 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.

```

1 def generate_output(input):
2     """Returns a Dict<TableName, Dict<ColumnName, Data>> by applying the the model
3     on the input data."""
4     tables = {}
5     out = {}
6     #####
7     # Table: equations
8     #####
9     ### Imported column: a from table.a ###
10    out["a"] = input["table"]["a"]
11
12    ### Imported column: b from table.b ###
13    out["b"] = input["table"]["b"]
14
15    ### Imported column: c from table.c ###
16    out["c"] = input["table"]["c"]
17
18    #####
19    ## Computed column: Solution
20    #####
21    ### Apply compute_delta ##
22    from compute_delta import compute_delta
23    out["Solution"] = compute_delta(
24        search(input, out, "equations", "a"),

```

```

24     search(input, out, "equations", "b"),
25     search(input, out, "equations", "c"),
26 )
27 #####
28 ### Apply solve ##
29 from solve import solve
30 out["Solution"] = solve(
31     out["Solution"], # Previous result used in next function
32     search(input, out, "equations", "a"),
33     search(input, out, "equations", "b"),
34     search(input, out, "equations", "c"),
35 )
36 #####
37 #####
38 ## Computed column: delta
39 #####
40 ### Apply compute_delta ##
41 from compute_delta import compute_delta
42 out["delta"] = compute_delta(
43     search(input, out, "equations", "a"),
44     search(input, out, "equations", "b"),
45     search(input, out, "equations", "c"),
46 )
47 #####
48 tables["equations"] = out
49
50 return tables

```

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

```

1 def check_constraints(input):
2     """Ensure all constraints are respected in input and output data"""
3     #####
4     # Verifying Input Constraints
5     #####
6     ### Apply ensure_is_not_null ##
7     from ensure_is_not_null import ensure_is_not_null
8     res = ensure_is_not_null(
9         input["table"]["a"].to_list(),
10    )
11
12    if not res:
13        return (False, ("ensure_is_not_null constraints failed"))
14
15    #####
16    # Verifying Table Constraints
17    #####
18    out = generate_output(input)
19
20    ### Verify data types
21    for x in out["equations"]["Solution"]:
22        break
23
24    for x in out["equations"]["delta"]:
25        if (type(x) != int and type(x) != numpy.int64):
26            return (False, "Type constraints failed on equations.delta")
27
28    for x in out["equations"]["a"]:
29        if (type(x) != int and type(x) != numpy.int64):
30            return (False, "Type constraints failed on equations.a")
31
32    for x in out["equations"]["b"]:

```

```

33     if (type(x)!=int and type(x)!=numpy.int64):
34         return (False, "Type constraints failed on equations.b")
35
36     for x in out["equations"]["c"]:
37         if (type(x)!=int and type(x)!=numpy.int64):
38             return (False, "Type constraints failed on equations.c")
39
40
41     return (True, "Constraints are respected")

```

7.1.4. Application de la librairie

On peut finalement appeler le script automatique (fonction `main` de la librairie).

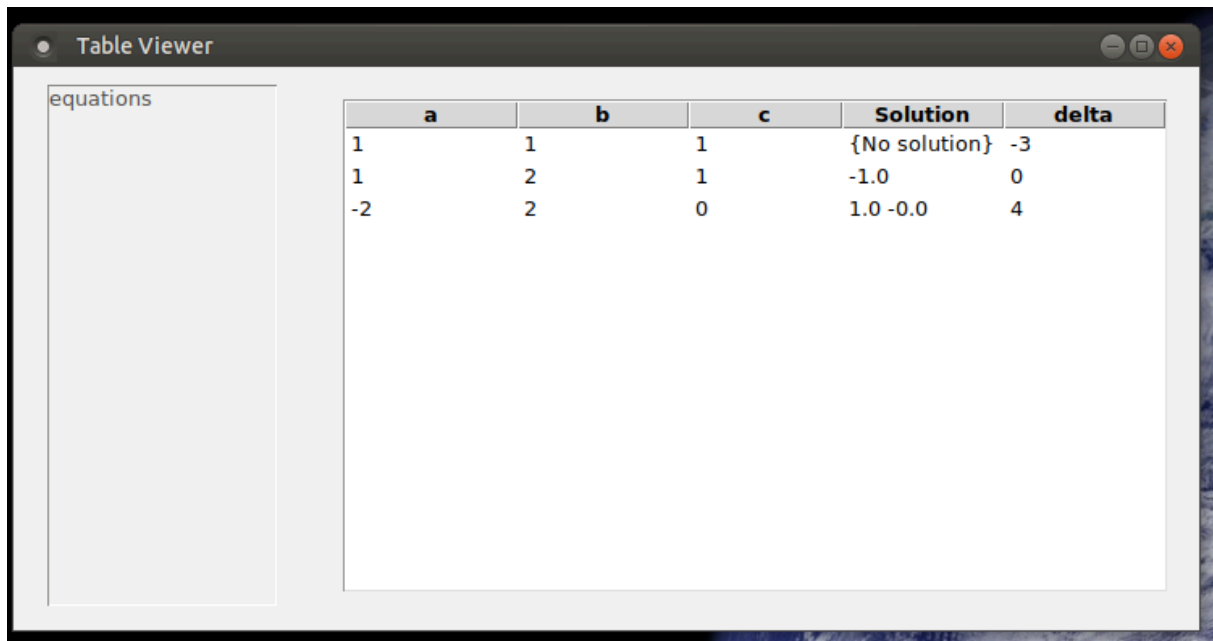
Index	a	b	c
0	1	1	1
1	1	2	1
2	-2	2	0

Tableau 1. – Données d'entrée (CSV)

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

Tableau 2. – Données transformées (CSV)

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



	a	b	c	Solution	delta
1	1	1	1	{No solution}	-3
1	2	1		-1.0	0
-2	2	0		1.0 -0.0	4

Fig. 11. – Visualisation des données générées.

8. Conclusion

8.1. Guillaume

J'ai trouvé le sujet un peu compliqué au début à devoir s'adapter à des contraintes d'utilisations et imaginer une solution sans avoir des contraintes techniques. Mais j'ai apprécié travailler en équipe pour trouver ces solutions et aboutir à un résultat exploitables et je pense y avoir gagné en management d'équipe et communication. Du côté technique j'ai pu travailler sur presque tous les types de transformations sauf texte à modèle ce qui m'a permis de mieux concevoir le travail d'un ingénieur logiciel qui doit s'adapter au problème qu'il rencontre.

8.2. Timothé

C'était un travail difficile pour moi car j'ai eu du mal avec la liberté d'interprétation. Dans ce projet, j'ai pu travailler sur la première version de la transformation de texte à modèle, la syntaxe concrète graphique et les contraintes statiques. Ce qui m'a permis de m'améliorer dans ces domaines.

8.3. Louis

J'ai également eu du mal à interpréter le sujet qui est très libre, mais je pense que cette difficulté fait partie de l'exercice.

J'ai apprécié le fait que le projet nous force à nous coordonner pour délivrer les différentes parties et éviter les conflits. Nous avons, par exemple, eu un problème de communication quant à la manière d'utiliser notre schéma de table en sortie (Entre le programme Python et les contraintes statiques).

8.4. Simon

J'ai, tout comme mes camarades trouvé le sujet assez vague, ne proposant pas réellement de scénario ce qui a notamment causé des dissonances au sein de l'équipe sur ce qui devait être produit. Avec la complexité (parlons plutôt d'impossibilité) de modifier les métas-modèles au cours du projet, cela empêche de commencer réellement différents aspects du projet sereinement. De plus l'outil Eclipse semble nécessiter plus d'expérience pour être utilisé que ce que nous avons, notamment en ce qui concerne la collaboration via git (et certains mystères qui resteront irrésolus et disparus après quelques redémarrage). Tout cela nous a fait « perdre » beaucoup de temps, ce qui reste très frustrant car nous avons l'impression de passer plus de temps à se battre contre l'outil plutôt que de travailler avec. Cependant je tiens aussi à souligner que les concepts appris dans la matière sont très intéressants, et que mes camarades sont restés particulièrement sérieux dans l'exercice de collaboration.

9. Annexes

9.1. Workspaces **fr.n7.ChaiseMinute**

9.1.1. **fr.n7.ChaiseMinute**

Workspace contenant le méta-modèle `chaiseMinute.ecore` décrivant les schémas de tables. Ce projet contient aussi le code pour la validation des modèles. Pour valider un modèle vis-à-vis des contraintes statiques, il faut lancer la classe `ValidateChaiseMinute.java` suivi du chemin d'accès vers un modèle.

9.1.2. **fr.n7.ChaiseMinute.function**

Worksapce contenant le méta-modèle `function.ecore` décrivant les fonctions utilisées dans les `Algorithm` d'un modèle.

9.1.3. **fr.n7.ChaiseMinute.script**

Workspace contenant le méta-modèle `algorithm.ecore` décrivant les algorithmes (utilisations des fonctions) dans le méta-modèle `ChaiseMinute` pour pouvoir appliquer des opérations pour les `ComputedColumns`.

9.1.4. fr.n7.ChaiseMinute.library

Workspace Acceleo permettant la transformation du modèle instancié de `ChaiseMinute` vers un fichier Python en sortie pour vérifier les contraintes, appliquer le schéma et afficher les tables obtenues.

9.1.5. fr.n7.ChaiseMinute.tabouretSeconde

Workspace comportant un fichier `TabouretSeconde.xtext` décrivant un langage textuel concret propre à `ChaiseMinute` pour décrire des schémas de tables (d'extension *cmt*). Cette transformation ne supportant pas les `eOpposite`, nous avons dû créer une transformation Modèle à Modèle `menuiserie.atl` pour obtenir des vrais schémas de tables `ChaiseMinute`.

9.1.6. fr.n7.ChaiseMinute.chaiseMinute.cmtToCm

Workspace comportant la transformation Modèle à Modèle `menuiserie.atl` pour engendrer des `ChaiseMinute` (d'extension *xmi* ou *cm*) à partir de la transformation `xText`.

9.2. fr.n7.ChaiseMinute.calculus

Workspace contenant le méta-modèle `calculusv2.ecore` décrivant les fonctions/scripts que l'utilisateur pourra définir par un éditeur graphique Sirius. Une fois l'édition faite, nous voulions pouvoir transformer ces scripts en fichiers Python pour pouvoir être appelé dans les `Algorithm` au travers des `paths`, mais ayant eu des problèmes avec l'édition graphique, expliquées plus haut (Limitations et améliorations), nous n'avons pu aboutir à une transformation utilisable.

9.3. Workspaces Sirius

9.3.1. fr.n7.ChaiseMinute.design

Workspace contenant le fichier `chaiseMinute.odesign` représentant le `ViewPoint` sous forme de containers d'un `ChaiseMinute` pour éditer des schémas de tables de façon graphique.

9.3.2. fr.n7.ChaiseMinute.samples

Workspace contenant des fichiers *cm* de `ChaiseMinute` éditables par les utilisateurs.

9.3.3. fr.n7.ChaiseMinute.calculus.v2design

Workspace contenant le fichier `v2design.odesign` représentant le `ViewPoint` sous forme d'arbre d'un `Calculus` pour être modifié par les utilisateurs ensuite.

9.3.4. fr.n7.ChaiseMinute.calculus.v2samples

Workspace contenant des fichiers *clc* de `Calculus` éditables par les utilisateurs.

9.4. Workspaces Exemples

9.4.1. fr.n7.ChaiseMinute.exemples

Workspace contenant différents exemples d'utilisation de schémas de tables avec les résultats de leur transformation vers le modèle et les générations python.

- Dans ce projet, nous avons fournis dans `exemples_contraintes_statiques` deux exemples différents de modèle qui passent l'entièreté des contraintes statiques (nom commence par « ok »). Ainsi que 9 autres exemples qui ne passent pas les tests le nom du fichier précise l'erreur qui doit être levé par le validator.

9.5. Figures Content

Figures

Fig. 1: ChaiseMinute Ecore Diagram.	3
Fig. 2: Algorithm Diagram.	4
Fig. 3: Function Diagram.	5
Fig. 4: Calculus Diagram.	6
Liste 1: Signatures des fonctions de la librairie et fonction <code>main</code>	8
Fig. 5: Sirius de ExempleComplice.cm.	9
Fig. 6: Palette de création de ChaiseMinute.	9
Fig. 7: Sirius de Mean.clc.	10
Fig. 8: Palette de création de Calculus.	10
Fig. 9: Erreur conception Pipe Edge.	10
Liste 2: Description textuelle du schéma de table	11
Fig. 10: Modèle <code>chaiseMinute</code> obtenu vu sous Sirius.	12
Tableau 1: Données d'entrée (CSV)	14
Tableau 2: Données transformées (CSV)	14
Fig. 11: Visualisation des données générées.	14