Rapport PIDR

Programmes polygraphiques

Année Universitaire 2008 - 2009

Encadrant universitaire : **GUIRAUD Yves (LORIA – Paréo)**

Membres du groupe : **MARTINELLE Sébastien (2A – TRS)**
SERRA David (2A – SIE)

Enoncé : réaliser une interface graphique pour la construction d'objets polygraphiques en Java

Sommaire

1. Introduction 4
2. Traitement du sujet de recherche 5
2.1. Analyse de l'existant 5
2.1.1. Polygraphes 5
2.1.2. Programmes polygraphiques 6
2.2. Langages et outils utilisés 7
2.2.1. GOM 7
2.2.2. TOM 7
2.2.3. Compilateur 8
2.3. Objectifs 9
2.4. Organisation 10
2.4.1. Méthode et répartition du travail 10
2.4.2. Matériel utilisé 10
2.4.3. Répartition des tâches 10
2.5. Résolution 10
2.5.1. Première approche 11
2.5.2. Passage entre Java et TOM 12
2.5.3. Deuxième approche 13
2.5.4. Notre solution 14
3. Conclusion 19
Annexe 21

Remerciements

Nous tenons à remercier Monsieur Yves Guiraud, notre encadrant universitaire, pour son accueil et sa disponibilité durant toute la durée du projet ainsi que pour la confiance qu'il nous a accordée.

Nous tenons également à remercier Monsieur Pierre-Etienne Moreau pour l'aide qu'il nous a apportée sur l'apprentissage de Tom.

Enfin nous remercions aussi le LORIA pour nous avoir permis de découvrir le monde de la recherche.

1. Introduction

Ce projet de PIDR (Projet interdisciplinaire ou de découverte de la recherche) s'inscrit dans le cadre du tronc commun de deuxième année à l'ESIAL. Il a pour but, comme son nom l'indique, de nous faire découvrir le monde de la recherche.

Notre sujet consiste en la réalisation d'une interface graphique de visualisation et de construction pour les programmes polygraphiques : des programmes décrits par des règles de transformation de circuits. Il fait suite au projet d'un élève ingénieur qui a réalisé un compilateur pour ces programmes polygraphiques.

Notre projet de PIDR s'est déroulé au LORIA (Laboratoire Lorrain de Recherche en Informatique et ses Applications) au sein de l'équipe PAREO. Cette équipe a été créée le 1er janvier 2008 ; elle est dirigée par M. Pierre Etienne Moreau et M. Yves Guiraud y occupe la place de chercheur.

Cette équipe s'est fondée sur une réflexion commune portant sur le fait que les langages de programmation basés sur des règles peuvent améliorer la qualité, la sûreté et la sécurité des systèmes informatiques. C'est pourquoi ils proposent des extensions de langages pour les programmeurs en leur permettant d'écrire une partie de leur code dans un style basé sur des règles (types abstraits, filtrage, règles et stratégies). Grâce à ces extensions, des outils d'analyse pourront être développés et le programmeur pourra les utiliser pour produire des certificats qui garantissent les propriétés de son code. De plus, cette équipe va élaborer des concepts et des outils pour rendre possible l'intégration des nouvelles constructions sans modifier le code existant. Le but de cette démarche est d'inciter les programmeurs à créer des programmes fondés sur des règles et certifiés, ce qui augmentera la confiance des utilisateurs dans leur code, permettra d'améliorer leur productivité, et ainsi d'accroître la qualité du code. Ce qui, à son tour, devrait convaincre les autres programmeurs d'utiliser ces techniques…

L'activité de l'équipe est matérialisée par le langage de développement Tom, initié à l'intérieur du projet Protheo. Son but est d'être une mise en œuvre concrète de la programmation fondée sur des règles.

Le domaine des programmes polygraphiques s'inscrit parfaitement dans cette philosophie de programmation basée sur des règles. Cependant il n'existe pas d'application informatique permettant d'afficher ou de construire des programmes de ce type, du moins pas facilement. Notre projet représente donc une étape fondamentale dans l'étude et la manipulation des programmes polygraphiques. Elle permettra une utilisation plus simple au quotidien de ces programmes et rendra possible la simulation d'objets complexes, fastidieuse à faire manuellement.

2. Traitement du sujet de recherche

2.1.Analyse de l'existant

2.1.1. Polygraphes

Une façon d'aborder les polygraphes consiste à les considérer comme des systèmes de réécriture sur des circuits algébriques. Les polygraphes sont des objets mathématiques composés de cellules de différentes dimensions.

2.1.1.1. Types

De dimension 1, ces cellules sont représentées par des fils et appelés « 1-chemin ». Chaque fil contient une information de type élémentaire : nat, list et bool par exemple un 1-chemin peut être composé de plusieurs fils dont les types peuvent être différents.

Figure 1 : Exemple de 1-chemin

2.1.1.2. Opérations

Les opérations sont représentées par des circuits appelés « 2-chemins ». Par convention, ils sont orientés du haut vers le bas. Comme les 2-chemins peuvent communiquer avec d'autres, ils possèdent des sources et des buts. Les sources correspondent aux paramètres d'entrée et les buts permettent d'obtenir les résultats. Les sources et les buts sont en réalité des 1-chemins et donc possèdent un type. Les opérateurs sont représentés par des composants appelés « 2-cellules ».Les 2-cellules sont de deux types : les constructeurs, permettant de décrire les données, et les symboles de fonction. On compose les 2-chemins de deux façons : soit en les montant en parallèle (juxtaposition notée *0) soit en les montant en série (connexion notée *1).

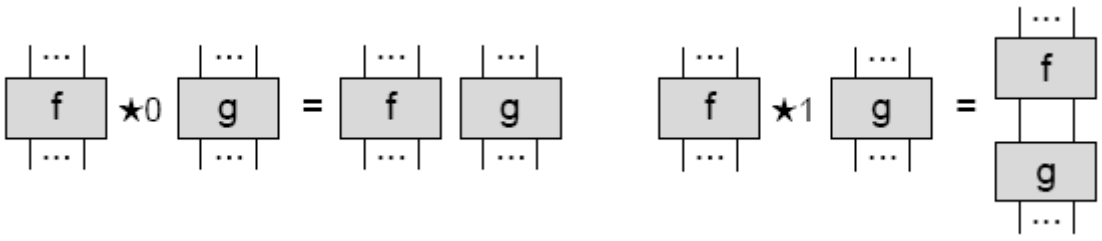

Figure 2 : Exemples de composition

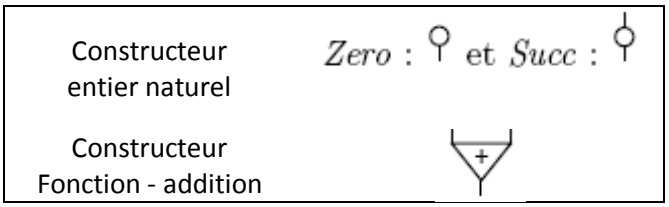

Figure 3 : Exemple de constructeurs

2.1.1.3. Calculs

La structure des 2-chemins et 1-chemins permet uniquement de définir le type des fonctions du programme, mais ne suffisent pas à effectuer un processus de calcul. En effet, il ne suffit pas de définir une addition comme étant une 2-cellule composée de deux entrées et d'une sortie pour traduire que la sortie sera l'addition des deux entrées. C'est le rôle des 3-cellules.

En effet, les 3-chemins sont des règles, qui permettent de décrire les calculs effectués par le programme, en fonction de cas élémentaires.

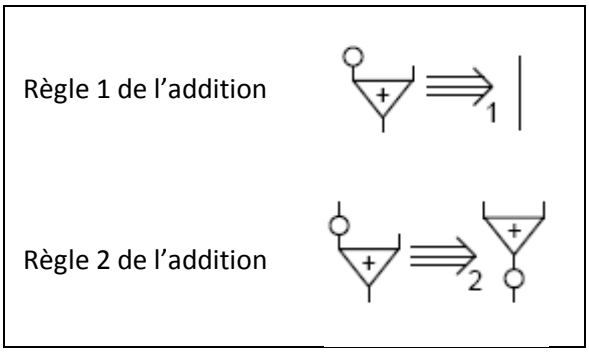

Figure 4 : Exemple de règles de calcul

Dans le cas ci-dessus, l'addition est définie via deux règles simples qui permettent de traiter par la suite tous les cas possibles d'une addition. Ces deux cas correspondent aux relations algébriques suivantes qui caractérisent l'addition d'entiers naturels :

- $0 + x = x$
- $(x + 1) + y = (x + y) + 1$

2.1.2. Programmes polygraphiques

Les programmes polygraphiques sont des cas particuliers des polygraphes de dimension 3. En effet, un programme polygraphique est la donnée de 1-cellules, 2-cellules et 3-cellules satisfaisant certaines propriétés calculatoires usuelles.

2.2.Langages et outils utilisés

Maintenant que nous connaissons la structure algébrique des programmes polygraphiques, nous pouvons passer à la partie technique. Afin de pouvoir faire fonctionner les programmes polygraphiques nous utilisons un langage de programmation intermédiaire nommé TOM. Ce langage de programmation consiste à faire le lien entre des langages de programmation fonctionnelle et la programmation par objet (comme Java ou C++). Dans le cadre des programmes polygraphiques, TOM est donc un langage particulièrement adapté puisqu'il permet d'établir des règles de calcul très simplement.

2.2.1. GOM

GOM permet de générer les structures de base sous forme d'arbres directement en Java. C'est donc ici que l'on spécifie ce que sont les 1/2/3-chemins. Ce générateur est cependant régi par des règles strictes. Ces règles impliquent que la génération doit être compatible avec TOM et sans effets de bord, c'est-à-dire sans pouvoir modifier les attributs d'un objet.

GOM fournit ainsi la structure de base via les termes sur lesquels seront implantés les traitements. Pour les programmes polygraphiques, voici la description des structures de base telles que nous les utilisons pour l'interface graphique :

ProgramPolygraphicGui.gom

```
OnePath = Id()
    | OneCell (Name : String, x : int, y : int, hauteur : int, largeur : int)
    | OneC0 (OnePath*)
TwoPath = TwoId (onePath : OnePath)
    | TwoCell (Name : String, Source : OnePath, Target : OnePath, Type : CellType, x : int,
y : int, largeur : int, hauteur : int)
    | TwoC0 (TwoPath*)
    | TwoC1 (TwoPath*)
ThreePath = ThreeCell (Name : String, Source : TwoPath, Target : TwoPath)
CellType = Constructor()
    | Function()
```

Note : * représente une liste de 0 à n éléments

Figure 5 : Description de la structure des programmes polygraphiques en TOM

2.2.2. TOM

La programmation fonctionnelle est une technique de programmation dans laquelle les programmes source représentent une fonction des données initiales. Les langages destinés à la programmation fonctionnelle se distinguent en général des langages traditionnels par l'absence de structure itérative (boucle) comme les classiques « `for` » ou « `while` » et par le fait que les variables ne varient plus après leur affectation initiale. La récursivité remplace les boucles et le fait que l'itération disparaisse supprime la majorité des cas où la modification de valeur de variables est nécessaire. Il existe deux grandes classes de langages fonctionnels :

- Les tableurs : une feuille de calcul implémente un programme fonctionnel ; chaque cellule s'exprime en fonction de valeurs initialisées une seule fois (valeurs brutes) ou calculées en fonction d'autres valeurs (formules), sans itération ou modification après l'affectation initiale.
- Les langages spécialisés tels que CAML ou Haskell.

TOM est une extension de langage destinée à manipuler des structures d'arbre comme, par exemple, des fichiers XML. Il inclut des notions de filtrage permettant d'inspecter et de récupérer des valeurs. Quand TOM est utilisé dans un environnement Java, des fonctionnalités supplémentaires sont disponibles :

- Un langage de règles puissant qui peut être utilisé pour contrôler les transformations.
- Un générateur d'objets orientés sur des structures d'arbre (GOM).

TOM permet de produire du code Java à partir du code TOM en utilisant les structures de base produites par GOM.

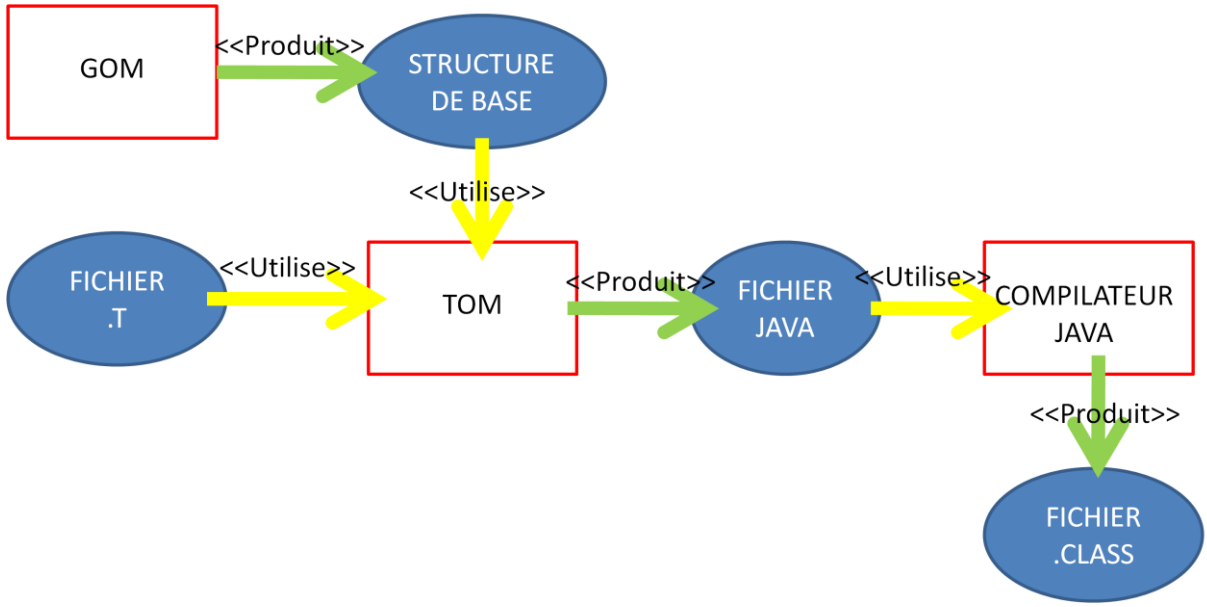

Figure 6 : Représentation graphique du cheminement d'informations par TOM

2.2.3. Compilateur

Un précédent stagiaire est déjà intervenu sur le projet afin de permettre la résolution des programmes polygraphiques en TOM. Pour pouvoir faire cela, il a utilisé GOM afin de créer les structures de base des programmes polygraphiques ; puis, grâce à TOM, il a créé un compilateur qui produit les règles de calcul à partir d'un fichier XML d'entrée.

Le fichier XML contenant les règles de calcul est d'abord traité par le compilateur, qui va produire une suite de règles de calcul en TOM. Ces règles de calcul seront ensuite transformées par le compilateur TOM afin d'obtenir un fichier exécutable. Ce dernier pourra alors récupérer un programme polygraphique à résoudre (sous forme d'un fichier XML). Ce fichier appliquera alors les règles de calcul et retournera alors un autre fichier XML de sortie avec le résultat du programme.

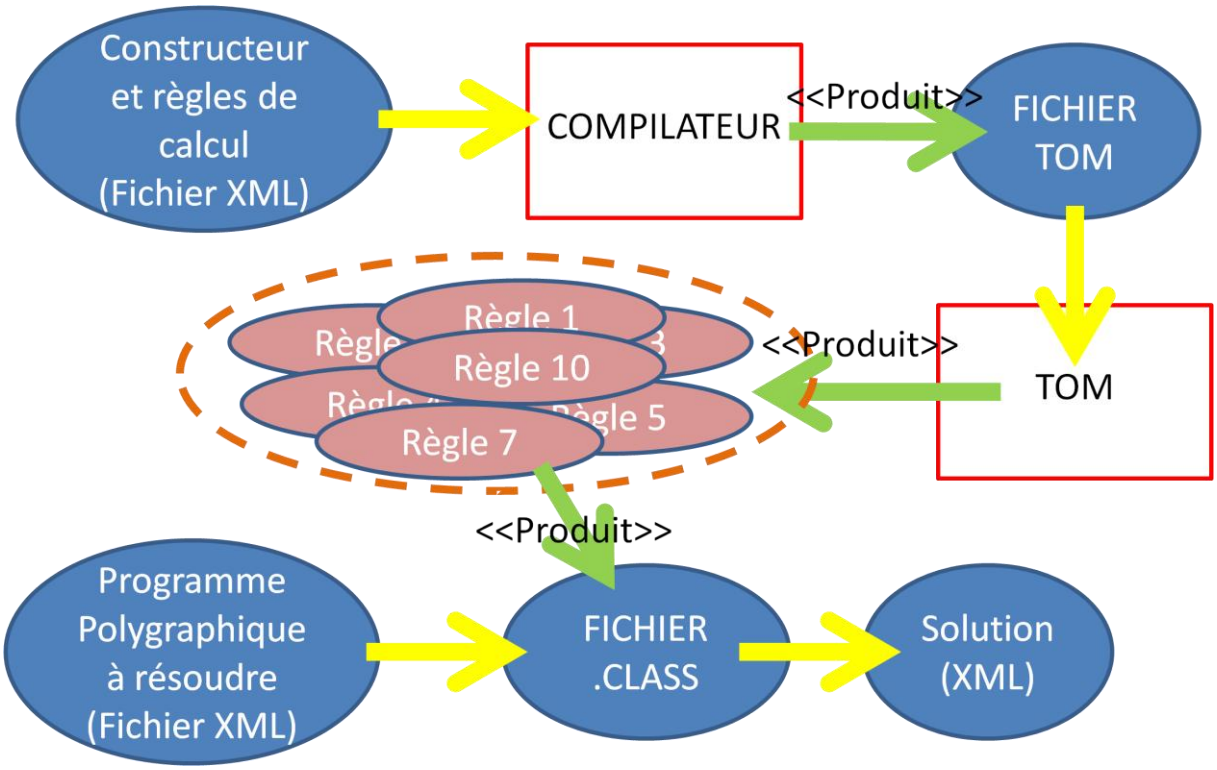

Figure 7 : Représentation graphique du fonctionnement du compilateur

2.3.Objectifs

Le compilateur déjà existant permet de calculer et de résoudre les programmes polygraphiques à l'aide de TOM et GOM. Malheureusement, le compilateur n'offre des résultats affichables qu'en version texte, ce qui rend leur lecture difficile pour un utilisateur dès que le programme polygraphique est important.

Le but à atteindre est de conserver le système actuel de calcul qui se base sur la lecture de document XML et qui est traduit en termes GOM. Ces termes sont utilisés en TOM pour obtenir un affichage graphique dynamique, rapide et clair.

Pour pouvoir obtenir cet affichage, il a donc fallu retravailler les éléments existants afin de leur permettre de prendre en compte de nouveaux paramètres, tels les positions des éléments dans le graphique. La définition des méthodes d'affichage ainsi que l'aspect et le rendu des résultats ont dû être définis de différentes manières afin de pouvoir tester les meilleures dispositions.

2.4.Organisation

2.4.1. Méthode et répartition du travail

Notre projet étant un sujet de recherche, cela a reflété une manière de travailler différente des projets que nous avons réalisés à l'ESIAL. La principale différence a été l'exploration de plusieurs pistes de recherche permettant de savoir si telle ou telle méthode était viable ou non.

Nous avons eu des réunions hebdomadaires avec notre encadrant, ce qui a eu le mérite d'un suivi régulier de notre travail. La séance se déroulait avec la présentation du travail accompli suivie d'une discussion sur les problèmes soulevés ou plus simplement sur les informations nouvelles obtenues. Enfin, au vu de l'avancement réalisé, nous fixions les prochaines étapes de travail.

2.4.2. Matériel utilisé

Le matériel utilisé a été le langage de programmation JAVA. L'environnement de travail est Windows même si des essais ont été effectués sous Linux pour s'assurer de l'interopérabilité entre plateformes. Le développement s'est fait grâce à l'IDE (environnement de développement intégré, en français) Eclipse. Le langage TOM également utilisé correspondait à la version en cours de développement au sein de l'équipe Paréo. Le grand avantage était de profiter des nouveautés du langage et de nous assurer de mises à jour réactives en cas de problèmes. Pour cela nous avons été intégrés à l'équipe de développement afin de profiter du serveur de développement par le biais de Subversion (SVN). SVN est un système de gestion de versions, ce qui permet de garder une trace du travail effectué sur le serveur. Il offre également la possibilité de récupérer une version antérieure. Un autre avantage de ce système est la possibilité de travailler à plusieurs sur le même programme tout en gardant les modifications de chacun.

2.4.3. Répartition des tâches

Il n'y a pas eu de répartition stricte des tâches puisque celle-ci a évolué en fonction du travail effectué et des difficultés rencontrées. En effet, quand cela était possible nous avons séparé le travail en deux afin de pouvoir explorer plusieurs pistes de recherche. Par exemple, pour la construction des fils droits ou courbes, chaque solution a été réalisée par un membre du binôme et nous avons ensuite comparé les résultats. Utiliser cette séparation des tâches a été utile pour étudier plus de possibilités. Cependant certaines étapes plus critiques ont été réalisées conjointement par les deux membres du groupe.

2.5.Résolution

Le sujet étant assez libre, il a fallu tout d'abord définir à quels objectifs précis devrait correspondre le programme de visualisation. C'est à cette époque et notamment pour des raisons techniques, que nous avons décidé de faire une séparation stricte entre l'interface de visualisation et l'interface de construction. Nous avons donc commencé par l'interface de visualisation qui est une étape

primordiale. Nous avons tout d'abord étudié les possibilités qui s'offraient à nous pour la réalisation de schémas graphiques avec le langage JAVA.

La possibilité de changer de langage de programmation a été étudiée, mais à cause de la possible utilisation conjointe avec TOM, nous sommes restés sur le langage JAVA. De plus, comme nous l'avons vu précédemment utiliser Java et TOM permet d'obtenir des fonctionnalités supplémentaires que nous avons effectivement utilisées par la suite.

Il nous a fallu réfléchir sur les critères qui permettent de diriger notre travail. Plusieurs critères ont ainsi permis de déterminer la meilleure solution :

- Difficultés de programmation et faisabilité en JAVA
- Compatibilité avec le compilateur et avec TOM
- Gestion des éléments
- Temps de traitement
- Possibilité d'exporter les informations ayant pour intérêt les graphismes dans différents langages
- Eviter les erreurs d'interprétation de lecture

2.5.1. Première approche

2.5.1.1. Disposition spatiale

En JAVA, la disposition des objets graphiques sur une fenêtre se fait à partir de conteneurs appelés « layout ». Aucun layout ne correspondait à une représentation dynamique d'objets nous permettant de les placer en coordonnées absolues. En effet, les layouts en Java s'occupent de placer les éléments les uns par rapport aux autres. Malgré de nombreux essais, aucun n'a permis d'obtenir la flexibilité que nous désirions. C'est pourquoi nous avons décidé pour la partie visualisation de n'utiliser aucun layout sur la fenêtre graphique. De ce fait, nous avons perdu la flexibilité mais nous pouvons choisir les coordonnées des objets à placer.

2.5.1.2. Gestion des fils

La méthodologie pour le placement des objets étant établie, nous nous sommes intéressés à la question de la représentation des programmes et plus précisément de l'enchaînement des différentes 2-cellules. La problématique méritait une réflexion puisque le but de ce projet est de permettre un affichage qui puisse être rapidement lisible, cela impliquant que la lecture du programme soit claire. Dans les programmes polygraphiques et en raison de la structure de base des programmes, deux éléments peuvent être modifiés : les 2-cellules ou les fils.

Deux solutions ont été étudiées :

- Fils courbes : dans ce cadre, les 2-cellules sont positionnées avec une taille par défaut, et leur positionnement est relatif à leur construction (i.e. : décaler vers le bas dans le cadre d'une construction en série (*1) ou vers la gauche dans le cadre d'une construction en parallèle (*0)) mais les liaisons entre les constructions sont faites par des fils courbes. La

gestion graphique doit donc être capable de déterminer le point de départ ainsi que le point d'arrivée, et d'établir un fil qui peut se courber.

- Fils droits : le concept est différent puisqu'ici les fils doivent être maintenus droits. Il faut donc que les cellules s'adaptent par un agrandissement vertical ou horizontal afin de pouvoir établir une liaison droite.

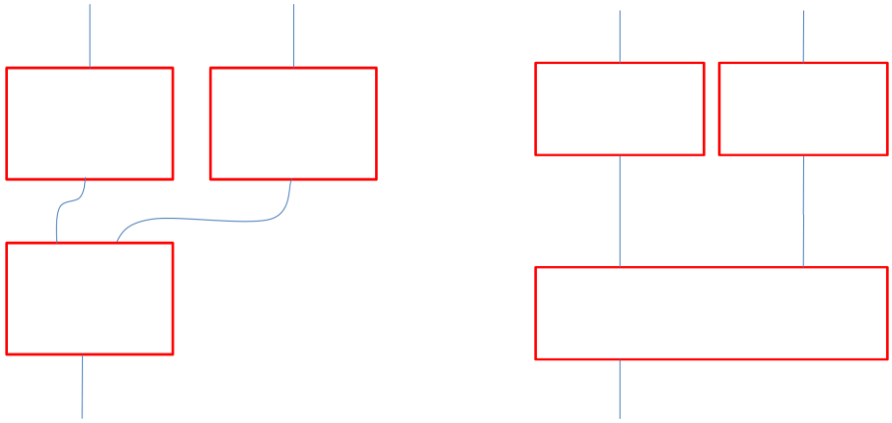

Figure 8 : Représentation graphique des deux solutions étudiées

Ces deux solutions sont potentiellement intéressantes et pourtant nous nous sommes tournés vers le concept de fils droit. Il y a plusieurs raisons à cela. Les fils courbes nécessitent la représentation en mémoire d'un rectangle dans lequel on doit mémoriser les extrémités supérieures, inférieures et le point d'inflexion. De plus, les courbures d'apparence différentes en fonction de la position des éléments rendent la visibilité de l'ensemble difficile. Enfin, il est tout à fait possible que des fils se croisent en raison des angles de courbure différents.

Nous avons donc abandonné cette voie pour nous concentrer plus particulièrement sur la partie des fils droits. Cette méthode a pour avantage de rendre impossible une erreur d'interprétation lors de la lecture car les cellules sont organisées pour suivre les fils.

2.5.1.3. Premier résultat

Une fois ce choix effectué, nous nous sommes donc concentrés sur les calculs de positionnement des objets les uns par rapport aux autres, qui se sont avérés plus compliqués qu'au premier abord, notamment avec les compositions *0 et *1. L'étape suivante a donc été l'implantation de ces calculs dans un programme JAVA pur pour démontrer la faisabilité et la fiabilité de ces calculs.

2.5.2. Passage entre Java et TOM

Ensuite, nous avons repris la base des calculs pour l'intégrer dans un programme similaire mais cette fois en utilisant TOM et ses améliorations spécifiques à JAVA. Cependant cette phase s'est révélée beaucoup plus compliquée à mettre à en œuvre et nous a pris beaucoup plus de temps que prévu, c'est pourquoi la partie construction n'a finalement pas été réalisée. Cependant tout le cœur du

programme a été programmé dans une optique de réutilisation pour des améliorations futures telles que l'interface graphique de construction.

Des recherches ont également été effectuées sur le fonctionnement de TOM car malgré une documentation existante, l'installation sous Windows s'est révélée chaotique. Nous avons transmis cette expérience à l'équipe afin d'améliorer ce point pour des versions ultérieures de TOM.

Dans cette optique, nous avons également repris le compilateur réalisé par l'élève ingénieur pour le rendre compatible avec Windows, ce qui n'était pas le cas auparavant. Cela nous a permis de mieux comprendre le fonctionnement du compilateur et la manière d'utiliser TOM avec les programmes polygraphiques.

2.5.3. Deuxième approche

Cette deuxième approche concerne le programme utilisant TOM.

2.5.3.1. Règles de composition

Plusieurs difficultés sont pourtant apparues. Nos premiers tests étant réalisés avec Java, fortement adapté aux affichages graphiques, la transition avec la programmation TOM nous a forcés à repenser une partie des calculs mis en place. En effet, lorsque nous avions fini la programmation dans le cadre de notre première approche, chaque élément était sous forme d'objet. Cependant pour formuler les compositions *0 et *1, nous avons utilisé des objets auxquels nous avons appliqué la théorie des programmes polygraphiques. En effet, nous nous sommes rendus compte, lors du passage en TOM, que nous avions arbitrairement forcé une composition comme étant composée obligatoirement de deux éléments simples alors que le compilateur utilise des règles différentes puisque les compositions sont des listes pouvant contenir de 0 à n éléments.

2.5.3.2. Effets de bord

Un autre point de difficulté concerne cette fois-ci le langage TOM/GOM. En effet, GOM interdit les effets de bord et, par conséquent, il est impossible de modifier un objet. Or, pour pouvoir déplacer un objet, il faut pouvoir en changer les coordonnées. Il a donc fallu travailler sur cet aspect afin de créer des méthodes qui permettent de créer un nouvel élément capable de remplacer celui que nous souhaitons modifier. Ce problème a pris d'autant plus longtemps à être réglé que TOM ne permet pas de remplacer facilement des informations dans une liste (d'une composition).

2.5.3.3. Sens de la construction

Enfin un dernier problème rencontré est dû au fait qu'un programme polygraphique peut se lire de différentes manières. Dans ce contexte, le compilateur utilisait un parseur de lecture pour lire les fichiers XML d'entrée. Le parseur construit par portions différentes et dans le désordre le programme polygraphique (afin d'éviter tout problème d'interprétation) et non dans un sens de lecture précis. Ceci nous a obligés à établir des méthodes pouvant pallier à tous les cas de figure et à tous les sens

de lecture. Ceci a retenu notre attention un certain temps car il est difficile d'établir une solution qui soit valable pour tous les cas possibles.

2.5.4. Notre solution

2.5.4.1. Introduction

Progressivement nous avons obtenu une interface graphique, qui nous a permis de représenter les différents programmes polygraphiques. Nous avons porté une attention toute particulière à la conservation de la structure du compilateur original, c'est pourquoi nous avons apporté des modifications mineures sur les termes GOM, notamment avec les attributs de position (x, y, largeur, hauteur). En effet, chaque modification dans les structures de base à un impact direct sur le compilateur, qu'il faut alors modifier pour prendre en compte les nouveaux paramètres. Nous avons donc décidé de modifier notre approche. En effet, dans notre approche en Java pur, chaque élément était un objet pour lequel nous avions donné des coordonnées. Notamment, une composition avait les coordonnées de ses différents sous-éléments, ce qui lui permettait de connaître la place que prenait l'ensemble de ses sous-éléments. Nous avons aussi travaillé de concert avec les méthodes de création et de génération des structures de base du compilateur afin de produire, en même temps que la création des objets du compilateur, les objets graphiques dans une optique de gain de temps de traitement.

2.5.4.2. Placement des 2-cellules

Pour pouvoir placer nos 2-cellules, nous avons aussi utilisé la puissance du langage TOM puisque nos méthodes utilisent le filtrage. En effet, TOM permet, en fonction de ce que l'on cherche à filtrer, d'effectuer une action plutôt qu'une autre. En Java, il s'agirait alors d'une sorte de « switch-case » permettant d'aiguiller un objet.

Nous avons donc fortement utilisé les effets de filtrage afin de gérer l'ensemble des cas de génération. Cependant, en raison de la manière dont est produit le code, il nous est impossible d'attacher les méthodes de placement aux termes GOM (et donc aux structures de base), ce qui a conduit à la création de multiples méthodes ayant un rôle très proche. Ainsi, il existe en moyenne trois variantes d'une même méthode. Par exemple, nous avons une méthode : récuperationFils() qui permet de récupérer les fils d'un objet 2-cellules. Mais en réalité, il existe les méthodes récuperationFilsHaut(), récuperationFils(), récuperationFilsBas(). Ce système, bien que lourd, permet en fait de gérer tous les cas possibles de construction. Ces méthodes sont appelées par les méthodes de construction des compositions qui, à l'aide des filtrages, permettent de conduire un objet vers ses propres méthodes.

Voici un exemple de construction :

Soit TwoC1(TwoC0(C1,C2),C3) une composition verticale d'une cellule C3, avec un élément qui est une composition horizontale de C1 et C2.

Dans ce cas, la construction s'opère suivant les étapes suivantes :

- Génération et initialisation de C1 (position de départ x=0 et y=0)

- Génération et initialisation de C2 (position de départ x=0 et y=0)
- Construction de la composition horizontale *0 entre C1 et C2
    - décalage C2 par rapport à C1 vers la gauche
- Génération et initialisation de C3 (position de départ x=0 et y=0)
- Construction de la composition verticale *1 entre l'élément précédent et C3
    - Décalage de C3 par rapport à (C1 *0 C2) vers le bas

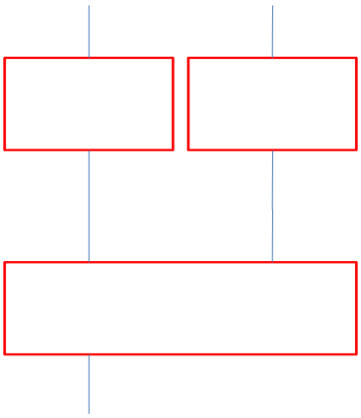

Figure 9 : Résultat de cette construction

Cependant, si ceci nous a permis de gérer le positionnement des éléments, il faut savoir que cette construction n'est pas forcement dans cet ordre (ici lecture de haut en bas et de gauche à droite). C'est donc toute l'utilité des méthodes de filtrage qui nous permettent, notamment dans les cas de constructions complexes et partielles, de pouvoir récupérer des constructions simples (la partie basse d'une composition *1 par exemple) afin de construire de manière simple les compositions.

Cependant, si le placement d'éléments est effectué, il nous faut aussi prendre en compte le fait que nous souhaitons avoir des fils droits.

2.5.4.3. Gestion des fils

Pour conserver et gérer les fils, plusieurs conditions doivent être assurées. D'une part, les cellules doivent être suffisamment larges pour recevoir les fils venant d'autres cellules et ce notamment afin de préserver les fils droits. D'autre part, les 2-cellules doivent récupérer des fils provenant de la même distance du niveau supérieur. La gestion des fils droits s'opère à l'aide de deux stratégies qui correspondent aux deux compositions.

2.5.4.3.1. Stratégie horizontale

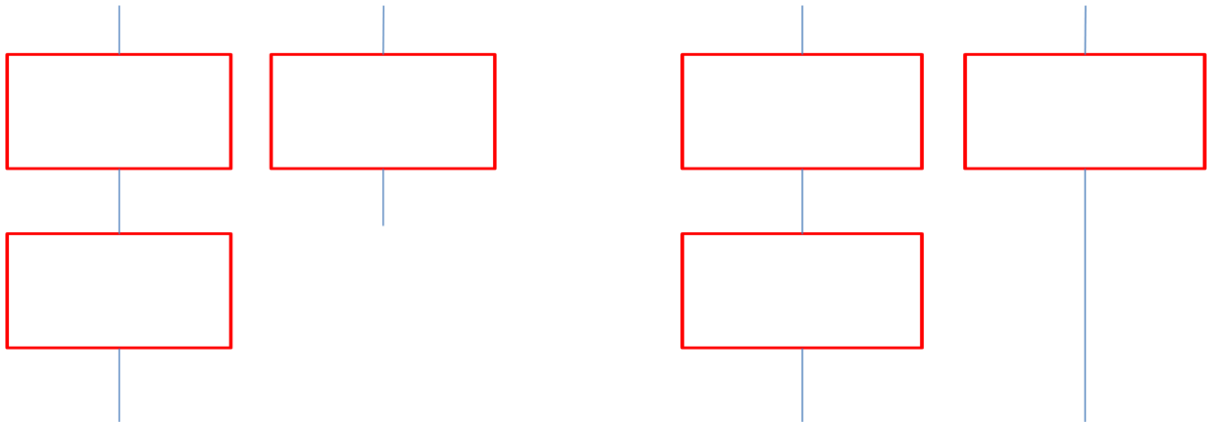

Figure 10 : Exemple de la construction graphique*0

Dans les compositions *0, on cherche à conserver une taille en hauteur équivalente entre les éléments. En effet lors de la construction des compositions *0, les différents éléments peuvent être de hauteurs différentes. Or la composition *1, pour des raisons pratiques, nécessite que les cellules hautes soient toutes à la même hauteur afin de décaler efficacement et de manière standardisée les cellules basses. Pour cela, on cherche dans un premier temps à déterminer dans la liste des éléments quel est l'élément le plus grand (en hauteur). Nous effectuons alors un allongement des fils buts (les fils du bas) des autres cellules, ceci dans le but d'allonger les éléments et de leur donner la même taille.

2.5.4.3.2. Stratégie verticale

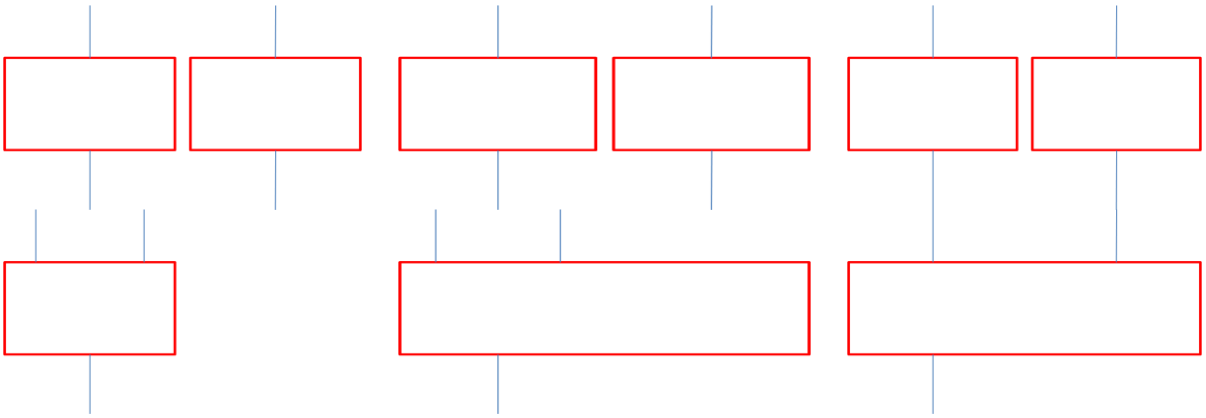

Figure 11 : Exemple de la construction graphique*1

La stratégie verticale impose de nombreuses règles, notamment pour obtenir des fils droits. Il y a pour cela deux étapes qui travaillent ensemble. Dans un premier temps, on cherche à agrandir les cellules afin de leur donner une taille horizontale qui correspond à la position du dernier fils qu'une cellule du bas doit recevoir d'une cellule de la partie haute. Dans un deuxième temps, les fils de la

cellule du bas voient leur abscisse modifiée afin de correspondre également aux fils bas des cellules du haut.

L'étape de construction *1 étant particulièrement critique en raison des contraintes (fils droits, gestion des différents cas), la gestion et la construction de cette étape passent par une série de filtrages avant d'aborder la construction de celui-ci. Cela permet de déterminer quelles sont les entités qui entrent en jeu lors de la construction, quelles doivent être les entités qui vont être modifiées et quelles sont celles qui, au contraire, sont à conserver.

L'une des principales difficultés de cette étape est de modifier et de transmettre les changements de taille de la cellule à tout l'ensemble du programme déjà construit. C'est pourquoi les méthodes qui gèrent les compositions *1 sont récursivement appelées afin de modifier les éléments de niveaux inférieurs. Il existe donc un coût important de traitement en raison de ces multiples appels.

De plus, chaque modification sur les cellules nous oblige à recréer un nouvel objet (puisque TOM ne tolère aucun effet de bord). Afin de limiter ce coût, nous avons travaillé à limiter le nombre de créations à son strict minimum et à ne modifier le programme polygraphique qu'une seule fois par appel. Ceci s'est traduit par d'importantes duplications d'objets afin de nous permettre de gagner du temps de traitement au détriment de la mémoire.

2.5.4.4. Résultat final

Voici un exemple de représentation graphique que nous obtenons avec la règle $(x+1)^3 \rightarrow x^3+3x^2+3x+1$ permettant de calculer la fonction « cube » sur des entiers naturels :

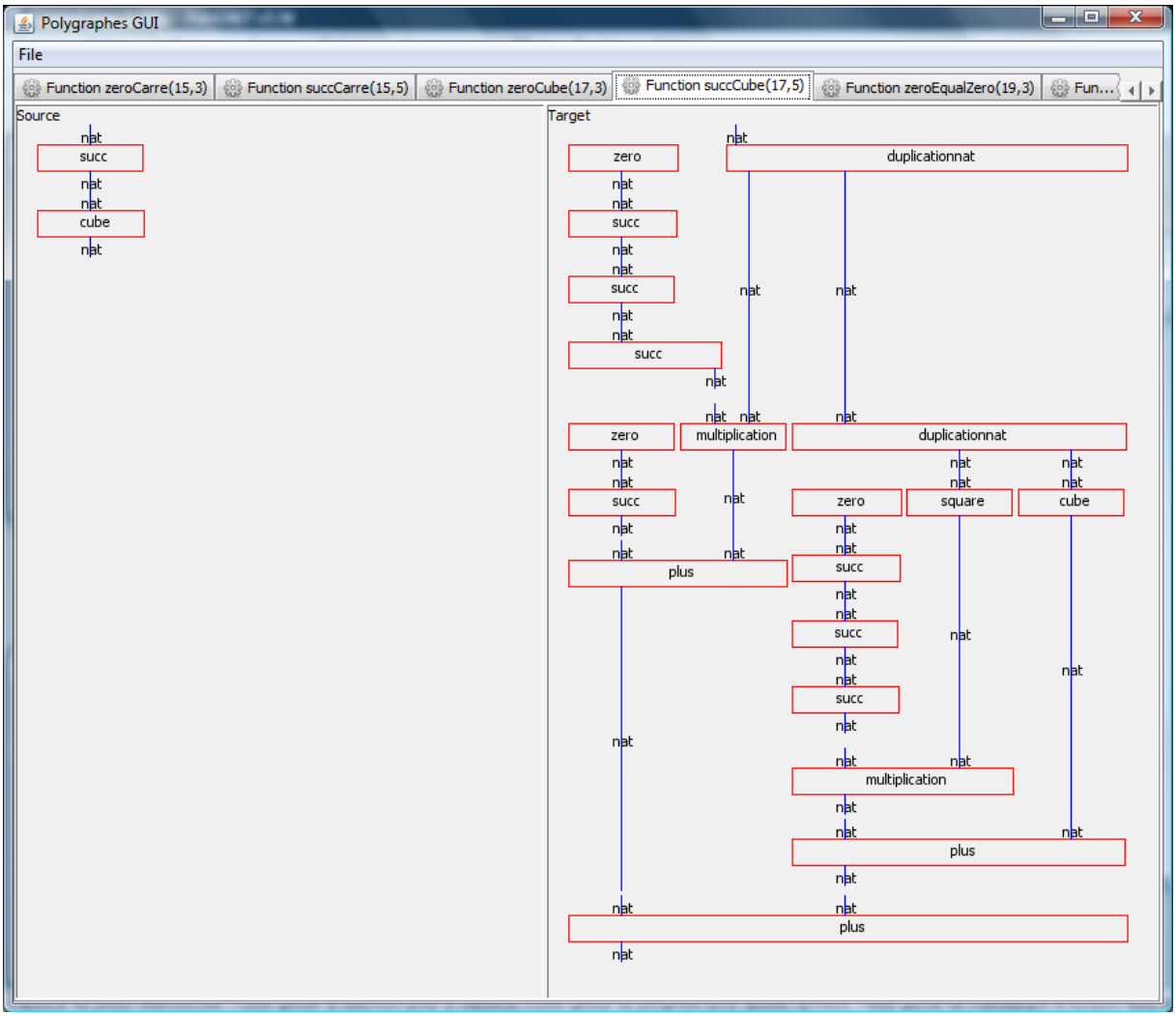

Figure 12 : Résultat obtenu

3. Conclusion

Nous avons découvert, durant ce projet interdisciplinaire ou de découverte de la recherche, le monde de la recherche et son fonctionnement. Nous avons, dans un premier temps, déterminé différentes approches et de nombreuses questions ont dû être résolues notamment par l'expérimentation. Ce projet a été l'occasion pour nous de confronter notre vision pro-développeur à une vision plus théorique et plus abstraite du problème afin d'en retenir les points les plus conflictuels.

Malgré cela, nous avons pris un temps important de notre projet à tester différentes approches. En effet, le fait d'avoir d'abord débuté par une approche purement Java, avant de passer sur les langages GOM et TOM, nous a permis d'appréhender certains problèmes, notamment sur la gestion graphique, mais nous a éloigné de la problématique de placement des composants.

De plus, le passage en GOM/TOM nous a particulièrement surpris en raison des difficultés dues aux restrictions sur les effets de bord imposées par ce langage. C'est pourtant grâce à cette restriction que nous avons entrevu une approche totalement différente du problème. Nous avons aussi travaillé de manière à conserver une forte compatibilité avec l'existant tout en améliorant celui-ci. L'aspect du temps de traitement a été pris en compte afin de conserver des performances satisfaisantes. Elles se traduisent notamment par la capacité du logiciel à pouvoir exploiter les résultats partiels sans avoir à attendre la fin de la lecture de l'ensemble du programme polygraphique.

Malgré nos efforts pour réaliser dans les meilleures conditions le projet, nous n'avons pas pu réaliser la deuxième partie, qui consistait à créer un éditeur capable de produire un programme polygraphique. En effet, les multiples interprétations qu'offre le parseur XML de TOM nous a demandé plus de temps que prévu pour réaliser une solution capable de prédire la position des éléments quelque soit la façon dont est lu le document et qui soit capable de transmettre les modifications aux seuls éléments touchés par ces modifications.

Nous estimons que nous avons déjà réalisé une base solide pour la représentation graphique des polygraphes mais que ce travail peut être amélioré notamment avec l'intégration d'une application de construction de programmes polygraphiques.

Bibliographie

- Guillaume Bonfante et Yves Guiraud, *Polygraphic programs and polynomial-time functions*, Logical Methods in Computer Science (à paraître), 2007.
- Aurélien Monot, *Compilateur pour programmes polygraphiques en TOM*, 2008.
- Tom, *tom.loria.fr .*

Annexe

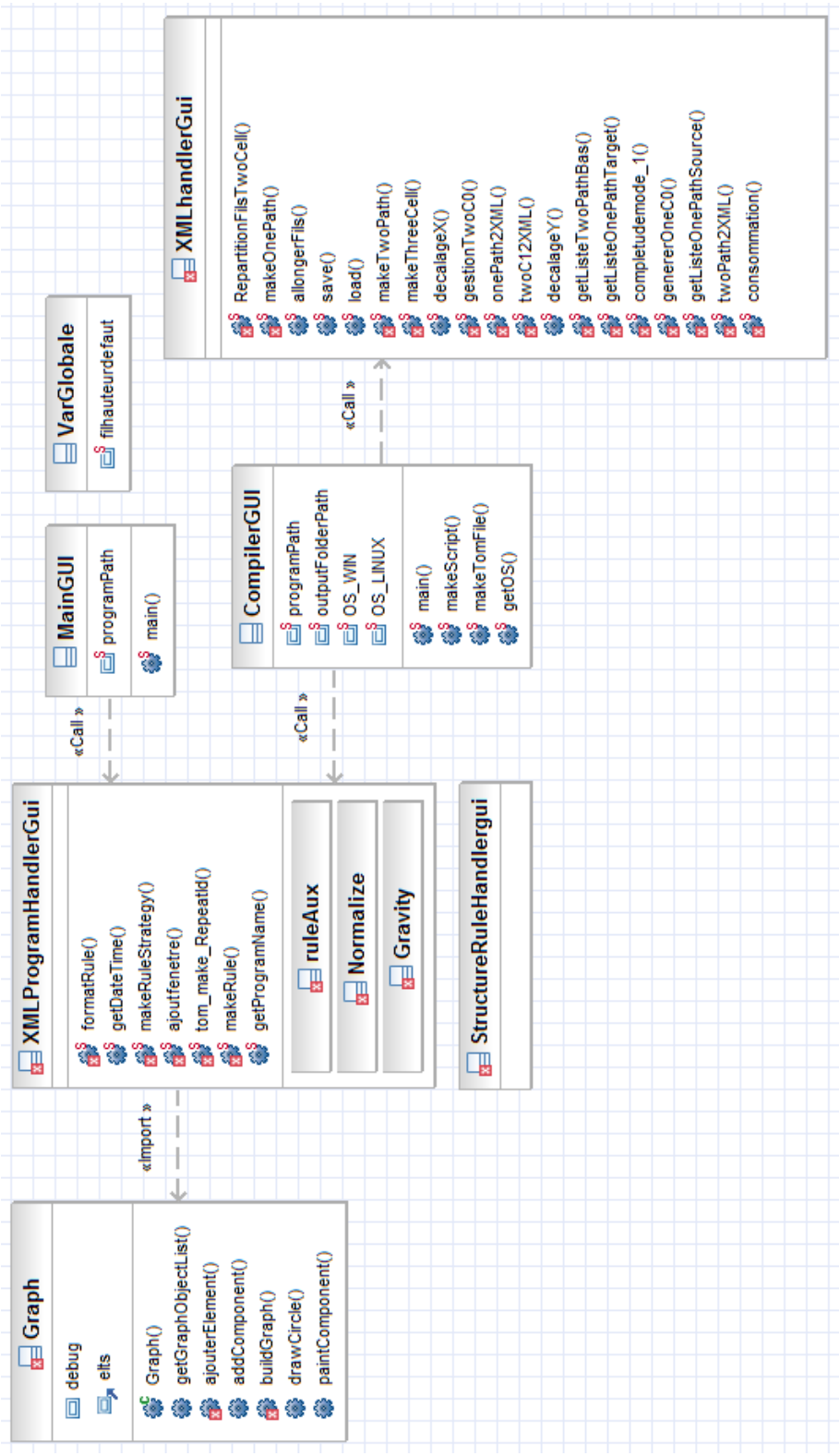

Figure 13 : Diagramme UML du compilateur et du système graphique