# Une approche catégorique unifiée pour la récriture de graphes attribués

Maxime Rebout


## ▶ To cite this version:

Maxime Rebout. Une approche catégorique unifiée pour la récriture de graphes attribués. Informatique [cs]. Université Paul Sabatier - Toulouse III, 2008. Français. NNT : . tel-00365230

## HAL Id: tel-00365230
## https://theses.hal.science/tel-00365230

Submitted on 2 Mar 2009

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# THESE

En vue de l'obtention du

## DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

**Délivré par** *l'Université Toulouse III - Paul Sabatier*
**Discipline ou spécialité :** *Informatique*

---

**Présentée et soutenue par** *Maxime REBOUT*
**Le** *16 juillet 2008*

**Titre :** *Une approche catégorique unifiée pour la récriture de graphes attribués*

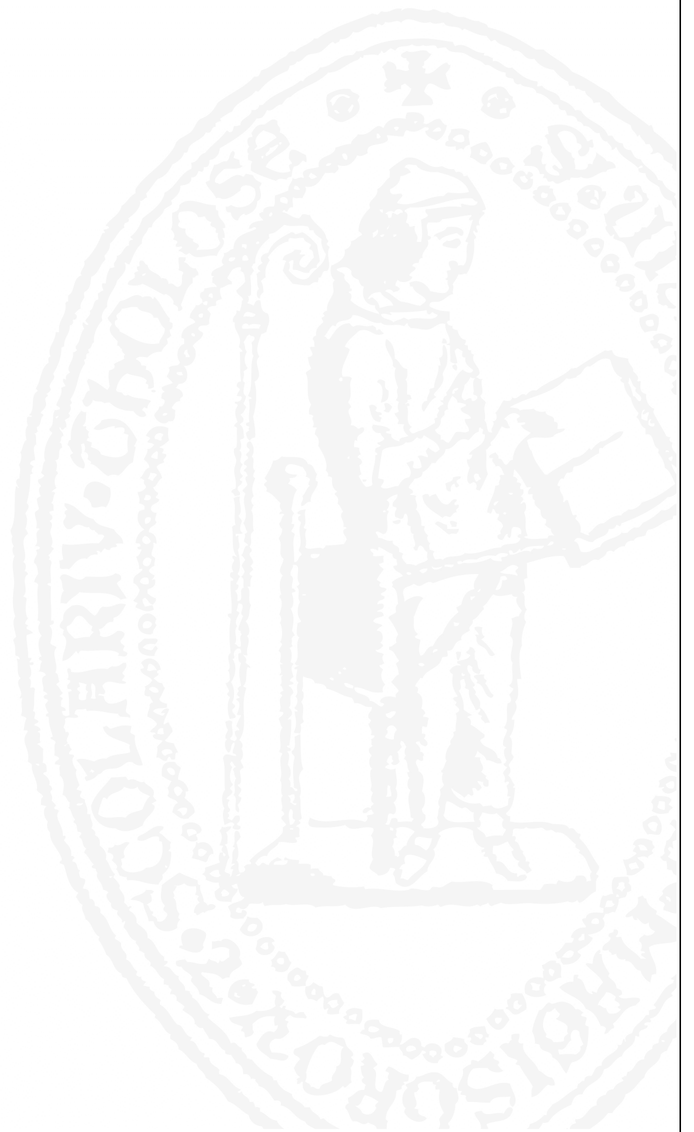
---

**JURY**

*Jean-Paul Bodeveix, Professeur, Université Toulouse III, Président*
*David Chemouil, Ingénieur, CNES*
*Louis Féraud, Professeur, Université Toulouse III*
*Reiko Heckel, Professeur, University of Leicester*
*Sergei Soloviev, Professeur, Université Toulouse III*
*Konstantin Verchinine, Professeur, Université Paris 12*

---

**Ecole doctorale :** *MITT*
**Unité de recherche :** *IRIT*
**Directeur(s) de Thèse :** *Sergei Soloviev et Louis Féraud*
**Rapporteurs :** *Reiko Heckel et Konstantin Verchinine*


SCOLARIV•THOLOSE

# Une approche catégorique unifiée pour la récriture de graphes attribués

**Maxime Rebout**

# Remerciements

Je tiens à remercier Louis FÉRAUD et Sergei SOLOVIEV, mes directeurs de thèse, sans qui cette thèse n'aurait pas existé. Ils ont su me proposer un sujet passionnant dans le domaine de l'ingénierie des systèmes, sans pour autant s'éloigner de l'algèbre qui forme la base de ma culture scientifique. Je leur en suis très reconnaissant.

Au delà de la science, je les remercie aussi pour tout ce qui fait que les années de doctorat se passent bien : leur bureau toujours ouvert pour répondre à une question, les discussions informelles autour d'un thé...

Cette thèse ne serait pas arrivé à son terme sans les rapporteurs - Reiko HECKEL et Konstantin VERCHININE - et les membres du jury - Jean-Paul BODEVEIX et David CHEMOUIL. Je les remercie tous très vivement.

Enfin, pour terminer les remerciements scientifiques, je tiens à exprimer ma gratitude envers tous les chercheurs que j'ai cotoyé au cours de ces années et dont les remarques m'ont aidé à avancer. Je pense en particulier aux équipes de recherche ACADIE et MACAO, mais aussi encore une fois à Reiko HECKEL avec qui j'ai eu de très fructueuses discussions lors de ses venues à Toulouse.

Par ailleurs, je tiens bien entendu à remercier ma famille qui m'a toujours soutenu tout au long de mes études ; en particulier mes parents, ma sœur pour ses traductions allemandes et mon frère comme fournisseur officiel de bonne musique.

J'ai aussi une pensée pour tous mes amis (toulousains, vosgiens ou d'ailleurs ; mathématiciens, informaticiens ou autres) qui m'ont toujours aidé au cours de ces années.

Enfin, je remercie Soazig pour tout ce qu'elle m'apporte.

# Table des matières


**1 Contexte : Les transformations de graphes 7**

**I Les systèmes inspirés par les grammaires textuelles 9**
1 L'approche «Hyperedge Replacement» . . . 9
1.1 Un exemple simple pour comprendre cette approche . . . 9
1.2 Hypergraphe . . . 9
1.3 Remplacement d'hyperarc . . . 10
1.4 Propriétés . . . 10
1.5 Productions et grammaires . . . 11
2 L'approche «Node Replacement» . . . 11
2.1 Substitution d'un nœud . . . 11
2.2 Les grammaires *edNCE* . . . 15
2.3 Les propriétés «context-free» . . . 15
3 Discussion . . . 16

**II Les approches catégoriques 19**
1 L'approche par «pullback» . . . 19
1.1 Graphe, produit et pullback . . . 19
1.2 Récriture de nœud . . . 21
2 L'approche par «pushout» . . . 23
2.1 Graphes, morphismes et catégorie . . . 24
2.2 Productions et grammaires de graphes . . . 26
2.3 Les propriétés de cette approche . . . 27
2.4 Évolutions du système . . . 27
3 Discussion . . . 27

**III Ajouter de l'information sur les graphes : les systèmes adhésifs HLR 29**
1 Les catégories adhésives . . . 29
1.1 Carré de Van Kampen . . . 29
1.2 Catégorie et système adhésifs HLR . . . 30
1.3 Quelques propriétés des systèmes adhésifs HLR . . . 31
2 Application aux graphes attribués typés . . . 33
2.1 Signatures et algèbres . . . 33
2.2 Graphes attribués sur les sommets et sur les arcs. Typage . . . 34
2.3 Transformations des graphes typés attribués . . . 35
3 Discussion . . . 36

**2 L'approche «Double Pushout Pullback» 39**

**IV Les catégories AttGraph et $\mathbf{AttGraph_{TG}}$ 41**
1 Définition des graphes . . . 41
1.1 La structure des graphes . . . 41
1.2 Les attributs . . . 42
1.3 Relation d'équivalence . . . 43

1.4 Exemple . . . . . 44
2 Définition des morphismes de graphes attribués . . . . . 45
2.1 Sur la structure . . . . . 45
2.2 Pour les attributs . . . . . 45
2.3 Relation d'équivalence . . . . . 48
2.4 Exemple . . . . . 48
2.5 Identification des arbres . . . . . 48
3 La catégorie **AttGraph** . . . . . 48
3.1 Composition de morphismes . . . . . 50
3.2 Morphismes identités . . . . . 51
3.3 Isomorphismes . . . . . 51
3.4 Catégorie . . . . . 51
4 Typage des graphes . . . . . 51
4.1 Morphisme de typage . . . . . 51
4.2 Exemple . . . . . 52
4.3 Catégories de graphes attribués typés . . . . . 52
4.4 Typages successifs . . . . . 53
5 Discussion . . . . . 53

**V Les constructions de base** **55**
1 Le pushout dans la catégorie **AttGraph** . . . . . 55
1.1 Existence du pushout . . . . . 55
1.2 La construction explicite du pushout . . . . . 56
1.3 Stabilité des classes $\mathcal{M}'$ et $\mathcal{N}$ . . . . . 60
1.4 La preuve de la propriété universelle du pushout dans la catégorie **AttGraph** 60
1.5 Exemple . . . . . 63
2 Le pushout-complement dans la catégorie **AttGraph** . . . . . 63
2.1 «Gluing condition» et existence du pushout-complement . . . . . 63
2.2 Construction explicite . . . . . 65
2.3 Stabilité des classes $\mathcal{M}''$ et $\mathcal{N}$ . . . . . 66
2.4 Exemple . . . . . 66
3 Compatibilité des constuctions catégoricielles avec le typage . . . . . 66
3.1 Avec le pushout . . . . . 66
3.2 Avec le pushout-complement . . . . . 68

**VI Système de transformations de graphes dans l'approche DPoPb** **69**
1 Règles de transformations . . . . . 69
2 Règles composées . . . . . 69
3 Comportement vis-à-vis du typage . . . . . 71
4 Conditions d'application . . . . . 72
4.1 Type I . . . . . 72
4.2 Type II . . . . . 72
5 Exemples . . . . . 73
5.1 Des arbres syntaxiques aux graphes directs acycliques . . . . . 73
5.2 Des diagrammes de classes UML aux bases de données relationnelles . . . . 74
5.3 Des diagrammes d'activité UML vers les réseaux de Petri . . . . . 76
5.4 Optimisation . . . . . 76

## 3 Outils et propriétés supplémentaires 85

**VII Les outils** **87**
1 Pullback . . . . . 87
2 Pullback faible . . . . . 88
3 Carrés de Van Kampen . . . . . 88
4 Factorisation des paires . . . . . 92

**VIII Extensions et restrictions de transformations** **95**
1 Extension de transformations . . . . . 95
1.1 Pushout initial et condition de collage . . . . . 95
1.2 Empan et extension . . . . . 98
2 Restriction de transformations . . . . . 100

**IX La confluence** **103**
1 Indépendance parallèle et indépendance séquentielle . . . . . 103
2 Paires critiques . . . . . 106
3 Confluence locale . . . . . 108

**X La terminaison** **111**
1 Différents types de strates de règles . . . . . 111
1.1 Terminaison d'une couche de destruction . . . . . 112
1.2 Terminaison d'une couche de calcul . . . . . 112
1.3 Terminaison d'une couche de création . . . . . 112

**XI Vers l'implantation** **115**
1 Quelques logiciels actuels de transformations de graphes . . . . . 115
1.1 PROGRES . . . . . 115
1.2 VIATRA2 . . . . . 115
1.3 AGG . . . . . 115
2 Quels choix pour l'implantation du système DPoPb ? . . . . . 116
2.1 Coq . . . . . 116
2.2 Ocaml . . . . . 117
2.3 Haskell . . . . . 117

**A Théorie des categories** **123**
1 Catégories . . . . . 123
2 Propriétés des flèches . . . . . 123
3 Diagrammes commutatifs . . . . . 124
4 Propriétés universelles . . . . . 124
4.1 Objets initiaux et terminaux . . . . . 124
4.2 Pushout . . . . . 124
4.3 Pullback . . . . . 126
5 Foncteurs . . . . . 127

# Introduction

*De l'importance des transformations de modèles en génie logiciel*

Influencé par la complexité croissante de logiciels dans l'informatique moderne, les chercheurs et les développeurs ont imaginé des nouvelles approches pour faciliter la conception de ces logiciels. Cette recherche vise plusieurs buts ; les nouvelles techniques doivent permettre de

- spécifier de manière simple et très précise le cahier des charges du logiciel ;
- s'adapter aux nouveaux concepts introduits par les langages de programmation modernes (en particulier les langages orientés objets ou composants) afin de les utiliser de manière efficace ;
- réutiliser des parties plus ou moins importantes du programme, le logiciel doit donc être découpé en module de manière simple et logique ;
- maintenir facilement le code source du logiciel, c'est-à-dire corriger un erreur ou bien rajouter une fonctionnalité *a posteriori*. Il doit donc être possible de modifier localement le code source sans introduire des incohérences avec le reste du programme.

On est alors arrivé au concept de modèles : la modélisation d'un système est un processus d'abstraction permettant de retenir uniquement les éléments nécessaires à la représentation du domaine étudié. Elle consiste à représenter le système sous forme de modèles en utilisant un certain nombre des concepts prédéfinis dans un langage de modélisation. Un modèle est, par définition, une entité manipulable avec des propriétés précises. Les premières descriptions de langages de modélisation sont apparues dans les années 70 avec, par exemple, un modèle relationnel proposé par Codd [14] et le modèle CODASYL utilisé pour spécifier le langage COBOL [74], mais c'est dans les années 90 que ces méthodes ont véritablement pris leur essor avec Booch, Rumbaugh et Jacobson qui ont chacun proposé leur vision du problème [9, 62, 44] avant de réunir leurs idées pour définir l'UML (*Unified Modeling Language*) [61, 36], qui est aujourd'hui normalisé par l'OMG (*Object Management Group*) et qui est le langage de modélisation le plus répandu.

L'UML est un langage de modélisation graphique qui permet grâce à treize types de diagrammes de décrire tous les aspects et les caractéristiques d'un logiciel. Certains diagrammes se focalisent plus sur la structure du programme, d'autres sur l'interaction du logiciel avec l'extérieur ou encore sur la chronologie interne lors de l'execution. Par exemple :

- les diagrammes de classes qui donnent une représentation de la structure du domaine étudié en terme de classes et d'associations. Une classe décrit les responsabilités, le comportement et le type d'un ensemble d'objets. Elle est décrite par un ensemble de fonctions et de données (attributs) ;
- les diagrammes de cas d'utilisation qui sont uilisés pour donner une vision globale du comportement fonctionnel d'un système logiciel. Un cas d'utilisation représente une unité discrète d'interaction entre un utilisateur (humain ou machine) et un système. Il est une unité significative de travail. Dans un diagramme de cas d'utilisation, les utilisateurs sont appelés acteurs (actors), ils interagissent avec les cas d'utilisation (use cases) ;
- les diagrammes d'activité qui permettent de modéliser un processus interactif, global ou partiel pour un système donné. Ils présentent une vision macroscopique et temporelle du système modélisé ;

D'un point de vue théorique, une des forces de l'UML est que la sémantique du langage se décrit par l'UML lui-même. Par contre, ce langage ne propose aucune méthode particulière pour d'une part construire la modélisation du logiciel et d'autre part utiliser la modélisation pour obtenir le code source (ou tout du moins un squelette du code source) du programme.

Dans cette optique, l'OMG a défini le standard du *Model Driven Architecture* (abrégé en MDA) [7, 55] qui offre une approche complète du développpement du logiciel. La première étape de ce développement est entièrement conceptuelle et ne s'inquiète pas des problèmes d'implantation : on décrit en UML le cahier des charges de l'application par des sous-modèles indépendants des technologies d'implantation - on nomme ces modèles des PIM (*Plateform Independant Model*). À la fin de cette étape, l'interaction du logiciel avec l'extérieur et son fonctionnement interne global doivent être entièrement décrits. Par la suite, on raffine ces modèles petit à petit pour prendre en compte les problèmes liés à la plateforme d'implantation. On construit ainsi des PSM (*Plateform Specific Model*). Après ces transformations, on doit obtenir un ou plusieurs modèles assez précis pour pouvoir générer automatiquement le squelette du code source du logiciel. Par ailleurs, le MDA introduit la notion de méta-modèle permettant de vérifier à chaque étape du raffinage la conformité du modèle obtenu par rapport à un méta-modèle connu (par exemple, celui correspondant à la plateforme d'implantation).

Ces raffinements se font donc grâce à des transformations de modèles. Dans sa version originale, l'approche MDA ne proposait aucun mécanisme pour ces transformations. Pour autant, pour obtenir un logiciel cohérent, il est nécessaire de s'assurer du bon déroulement de ces transformations et donc d'avoir un outil sûr pour executer ces constructions. Plusieurs approches ont été proposées pour cela :

- QVT (pour *Query/View/Transformation*) [38] est aujourd'hui le standard défini par l'OMG. C'est une spécifiaction de langage de transformations de modèles qui permet d'adapter un modèle à de nouvelles contraintes et de transformer n'importe quel langage dédié vers un autre ;
- ATL (pour *Atlas Transformation Language*) [45] est un langage de transformation de modèles développé à l'INRIA de Nantes par l'équipe de Bézivin et disponible sous forme d'un plugin Eclipse ;
- les système de récriture de graphes basés sur des grammaires de graphes. Plusieurs méthodes ont été développées pour faire de manière efficace des transformations de modèles à l'aide des grammaires de graphes ; on peut par exemple citer l'approche TGG (pour *triple graph grammars*) introduite par Schürr [67]). Différents logiciels implantent ces méthodes : les plus connus sont PROGRES, ATOM$^3$, VIATRA2 ou bien encore AGG.

Notre travail s'inscrit dans la recherche d'une base théorique solide pour les systèmes de transformations de graphes portant des attributs afin de construire une approche efficace et sûre pour les transformations de modèles. Il faut par ailleurs souligner que le champ d'application de la récriture de graphes n'est pas restreint au génie logiciel : on peut par exemple citer le domaine de l'optimisation [3] ou de l'imagerie [2, 49] et pour trouver des applications en dehors de l'informatique, l'étude de la structure des molécules d'ARN [70] en biologie peut aussi s'appuyer sur la récriture de graphes.

La suite de cette thèse est organisée comme suit :

- la première partie est consacrée à l'étude des théories connues de récritures de graphes. Plus précisement, les deux premiers chapitres se focalisent sur les différentes approches pour transformer un graphe simple (sans attributs) et le troisième chapitre complète l'étude avec l'introduction des attributs sur les graphes ;
- la seconde partie introduit notre contribution avec respectivement la définition de la catégorie des graphes attribués que l'on considerera, les constructions associés et enfin les systèmes de transformation dans notre système ;
- enfin, dans la troisième et dernière partie, on s'intéresse plus précisement aux propriétés plus fines de notre système.

# Première partie

# Contexte : Les transformations de graphes

On présente ici quelques approches parmi les plus répandues pour faire de la transformation de graphes. Dans un premier temps, on s'attardera sur des généralisations aux graphes des grammaires textuelles avant de s'intéresser aux approches dites algébriques, c'est-à-dire basées sur la théorie des catégories.

# Chapitre I

# Les systèmes inspirés par les grammaires textuelles

Les récritures sont très répandues sur les chaînes de caractères et s'expriment par le concept de grammaires. Ce concept a été repris et adapté pour récrire les graphes. Dans ce chapitre, nous présentons quelques formalisme de cette approche.

Les approches «Hyperedge Replacement» et «Node Replacement» sont des généralisations naturelles des grammaires textuelles : on substitue à un élément atomique du graphe portant une étiquette spécifiée une nouvelle structure. Bien entendu, la nature complexe des graphes exige plus d'informations que dans le cas textuel pour réaliser cette substitution ; en particulier, il faut indiquer de quelle manière la nouvelle structure va se rattacher à l'ancienne.

Pour l'approche «Hyperedge Replacement», dont on peut trouver une introduction dans [19], les éléments atomiques considérés sont les arcs ou les hyperarcs.

Pour l'approche «Node Replacement», on travaillera avec les nœuds du graphes. Plusieurs constructions ont été développées pour contrôler l'étape de recollement de la nouvelle sous-structure à l'ancienne structure. Nous présentons ici l'approche «NLC» (pour «Node Label Controlled») et plus particulièrement la variante «edNCE» (pour «edge labelled and directed Neighboorhood Controlled Embedding») qui traite le cas des graphes orientés avec des étiquettes sur les arcs. On peut trouver une présentation informelle de cette construction dans [34] et une description plus poussée dans [35].

## 1 L'approche «Hyperedge Replacement»

### 1.1 Un exemple simple pour comprendre cette approche

On considère ici les graphes orientés munis d'un nœud «début» et d'un nœud «fin». On peut remplacer un arc $e$ d'un graphe $G$ par un autre graphe $G'$ : on commence par effacer l'arc $e$ puis on fusionne le nœud «début» du graphe $G'$ avec la source de l'arc $e$ et le nœud «fin» du graphe $G'$ avec la cible de l'arc $e$. Le résultat est noté $G[e/G']$. Un exemple est dessiné figure I.1.

### 1.2 Hypergraphe

*Notation.* Soit $A$ un ensemble quelconque. L'ensemble de toutes les chaînes sur $A$ est noté $A^*$. Pour un élément $\omega \in A^*$, on notera $|\omega|$ la longueur de $\omega$.

Dans la suite, $C$ désignera un alphabet et $type : C \to \mathbb{N}$ une fonction de typage sur $C$.

**Définition 1.** *Un hypergraphe $H$ au-dessus de $C$ est donné par le quintuplet $(V, E, att, lab, ext)$ où :*

- *$V$ est un ensemble fini de nœuds ;*
- *$E$ est un ensemble fini d'hyperarcs ;*
- *$att : E \to V^*$ est une fonction qui associe à chaque hyperarc une suite de nœuds d'attachement distincts deux à deux ;*

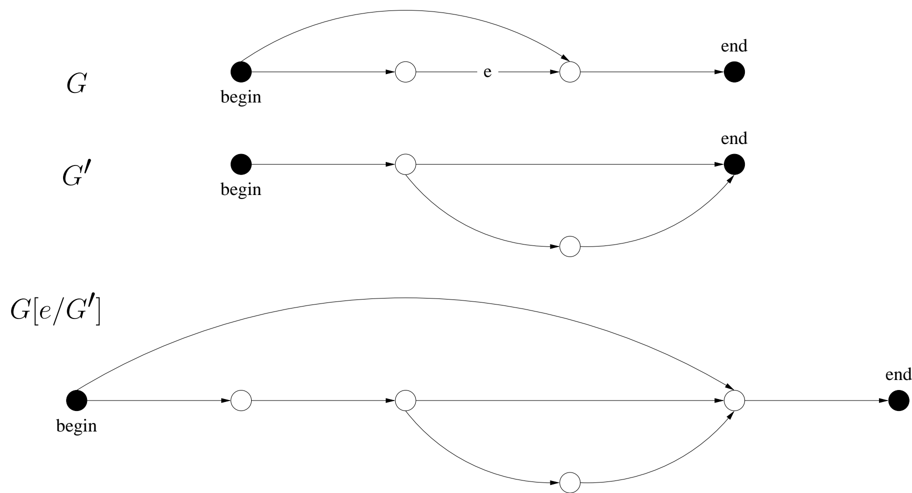



FIG. I.1 – Un exemple simple de remplacement d'arc

– *$lab : E \to C$ est une fonction d'étiquetage vérifiant $type(lab(e)) = |att(e)|$ ;*
– *$ext \in V^*$ est une séquence représentant les nœuds extérieurs de l'hypergraphe.*

*On note $\mathcal{H}_C$ l'ensemble des hypergraphes au-dessus de $C$.*

Dans les représentations graphiques, les nœuds extérieurs de l'hypergraphe seront représentés par des points noirs contrairement aux nœuds intérieurs qui seront blancs.

**Remarque 1.** Dans l'exemple introductif, les nœuds «début» et «fin» représentaient les nœuds extérieurs.

*Notation.* Pour un hypergraphe $H$, on note $type(H)$ le nombre de ses nœuds extérieurs (*i.e.* $|ext_H|$).

## 1.3 Remplacement d'hyperarc

Soit $H \in \mathcal{H}_C$ un hypergraphe et $e$ un hyperarc appartenant à $H$. Soit $R$ un second hypergraphe au-dessus de $C$. Si $type(R) = |att(e)|$ (on dit alors que $R$ est compatible avec l'hyperarc $e$), alors l'hyperarc $e$ va pouvoir être remplacé par l'hypergraphe $R$. Pour cela, on procède en trois étapes :
– on commence par effacer l'arc $e$ de l'hypergraphe $H$ ;
– on ajoute de manière disjointe le graphe $R$ au résultat de l'étape précédente ;
– on fusionne ensuite les nœuds extérieurs de $R$ avec les nœuds d'attachement de l'hyperarc $e$ (en respectant l'ordre).

Le résultat est noté $H[e/R]$.

On peut aussi faire plusieurs remplacements à la fois en se donnant un sous-ensemble $B = \{e_1, \ldots, e_n\}$ inclus dans $V_H$ et une fonction $repl : B \to \mathcal{H}_C$ vérifiant, pour tout $e \in B$, $type(repl(e)) = |att(e)|$. Le résultat sera alors noté $H[e_1/R_1, \ldots, e_n/R_n]$ où $R_i = repl(e_i)$.

## 1.4 Propriétés

On donne ici quelques propriétés importantes de l'approche «Hyperedge Remplacement».

#### *1.4.1 Séquençage et parallélisation*

Soit $H$ un hypergraphe et $e_1, \ldots, e_n$ des hyperarcs de $H$ distincts deux à deux. Pour chacun de ces arcs, on se donne un hypergraphe $H_i$ tel que $type(H_i) = |att(e_i)|$, alors on a :

$$H[e_1/H_1, \ldots, e_n/H_n] = ((H[e_1/H_1]) \ldots)[e_n/H_n].$$

#### 1.4.2 *Confluence*

Soit $H \in \mathcal{H}_C$ et $e_1$ et $e_2$ deux hyperarcs distincts de $H$. Soit $H_1$ et $H_2$ deux hypergraphes compatibles avec $e_1$ et $e_2$. On a alors

$$H[e_1/H_1][e_2/H_2] = H[e_2/H_2][e_1/H_1].$$

#### 1.4.3 *Associativité*

Soit $H$, $H_1$ et $H_2$ trois hypergraphes. On se donne un premier hyperarc $e_1$ appartenant à $E_H$ et un second $e_2$ dans $E_{H_1}$ tels que $e_1$ soit compatible avec $H_1$ et $e_2$ avec $H_2$. L'équation suivante est alors vérifiée :

$$(H[e_1/H_1])[e_2/H_2] = H[e_1/(H_1[e_2/H_2])].$$

### 1.5 Productions et grammaires

Dans la suite, l'ensemble $N$ designera le sous-ensemble de $C$ représentant les symboles non-terminaux.

**Définition 2.** *Une production sur $N$ est une paire ordonnée $p = (A, R)$ avec $A \in N$ et $R \in \mathcal{H}_C$ vérifiant $type(A) = type(R)$. Le symbole $A$ est appelé la partie gauche de la règle et $R$ la partie droite.*

Soit $H$ un hypergraphe et $P$ un ensemble de productions au-dessus de $N$. Soit $e \in E_H$ tel que la paire $(lab(e), R)$ appartienne à $P$. Alors $H$ se dérive directement en $H' = H[e/R]$. On écrit alors $H \implies_P H'$ ou bien encore $H \implies H'$ si il n'y a pas de confusion. On appelle cette étape une dérivation directe.

Une suite de dérivations directes $H_0 \implies H_1 \implies \ldots \implies H_k$ est appelé une dérivation de longueur $k$ et est notée $H_0 \implies^k H_k$ ou $H_0 \implies^* H_k$.

**Définition 3.** *Une grammaire est donnée par un quadruplet $HRG = (N, T, P, S)$ où $N$ et $T$ doivent former une partition de l'ensemble $C$ et représente respectivement l'ensemble des symboles non-terminaux et l'ensemble des symboles terminaux, $P$ est un ensemble fini de productions au-dessus de $N$ et enfin $S \in N$ est le symbole de départ.*

*Exemple.* On considère ici la grammaire $(\{A, S\}, \{a, b, c\}, P, S)$ où $P$ contient les quatre productions décrites dans la figure I.2. Pour représenter celles-ci de manière plus lisible, on utilise ici la notation de Backus-Naur : la production $(A, R)$ sera donc notée $A ::= R$ et les barres verticales apparaissent ici pour traduire une alternative. Cette grammaire permet de produire toutes les chaînes de la forme $a^n b^n c^n$ (la figure I.3 montre la dérivation donnant $a^3 b^3 c^3$).

## 2 L'approche «Node Replacement»

### 2.1 Substitution d'un nœud

**Définition 4.** *On se donne deux alphabets : $\Sigma$ pour les étiquettes des sommets et $\Gamma$ pour les étiquettes des arcs.*

*Un graphe $H$ au-dessus de $\Sigma$ et $\Gamma$ est donné par un triplet $(V, E, \lambda)$ où $V$ est un ensemble fini de sommets, $E$ est un sous-ensemble fini de $\{(v, \gamma, w) | v, w \in V, v \neq w, \gamma \in \Gamma\}$ et $\lambda : V \to \Sigma$ est une fonction d'étiquetage des nœuds.*

**Remarque 2.** La définition donnée ici interdit donc les boucles mais autorise le fait d'avoir plusieurs arcs entre deux sommets à condition qu'ils aient des étiquettes différentes.

Un graphe sera non orienté si pour chaque arête $(v, \gamma, w) \in E$, alors $(w, \gamma, v)$ appartient aussi à $E$.

**Remarque 3.** Pour modéliser des graphes qui possèdent des sommets ou des arcs sans étiquette, on rajoute aux alphabets $\Sigma$ et $\Gamma$ un élément particulier qui ne sera pas indiqué sur les représentations graphiques (*cf.* l'exemple de la section 2.2).

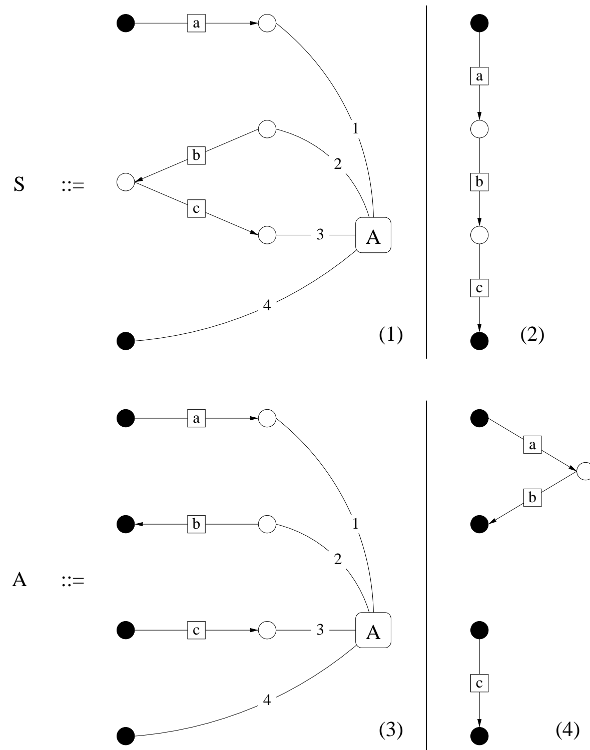



FIG. I.2 – Les productions générant les chaînes $a^n b^n c^n$

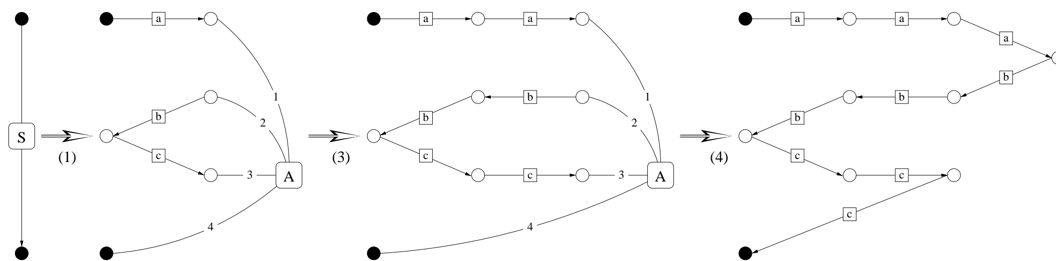



FIG. I.3 – Dérivation pour $a^3b^3c^3$ (il faut suivre l'hypergraphe dans le sens des flèches pour reconstruire cette chaîne)

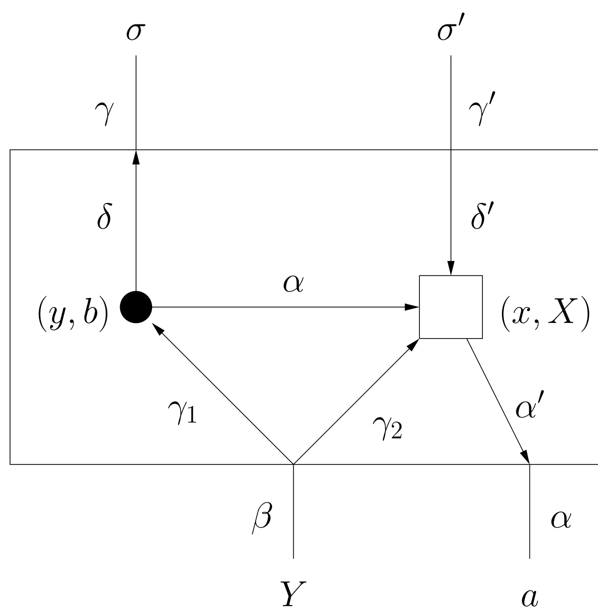



FIG. I.4 – Le graphe avec plongement $(D, C_D)$

**Définition 5.** *Une production d'une grammaire edNCE est de la forme $X \to (D, C)$ où $X$ est une étiquette de nœud non terminale, $D$ est un graphe et $C$ est un ensemble d'instructions de connexion ; c'est-à-dire un sous-ensemble de $\Sigma \times \Gamma \times \Gamma \times V \times \{in, out\}$.*

**Remarque 4.** Si $(\alpha, \beta, \gamma, x, d)$ est un élément de $C$, on préférera écrire $(\alpha, \beta/\gamma, x, d)$ pour bien indiquer que l'étiquette $\gamma$ va remplacer l'étiquette $\beta$.

Intuitivement, l'application d'une production $X \to (D, C)$ à un graphe $H$ se déroule en plusieurs étapes :

- on commence par retirer un nœud portant l'étiquette $X$ (le nœud principal) du graphe de départ $H$ ainsi que tous les arcs adjacents à ce sommet ;
- on ajoute alors le graphe $D$ ;
- on connecte enfin les nœuds du graphe $D$ à ceux de $H$ en suivant les instructions de connexion.

Une instruction de connexion telle que $(\alpha, \beta/\gamma, x, out)$ doit se comprendre comme suit : s'il existe dans le graphe $H$ un arc portant l'étiquette $\beta$ et partant du nœud principal pour arriver à un sommet $w$ portant l'étiquette $\sigma$, alors on va créer un arc du sommet $x \in D$ vers $w$ portant l'étiquette $\gamma$. Si à la place du *out* se trouve le symbole *in*, alors on doit considérer les arcs arrivant sur le nœud principal.

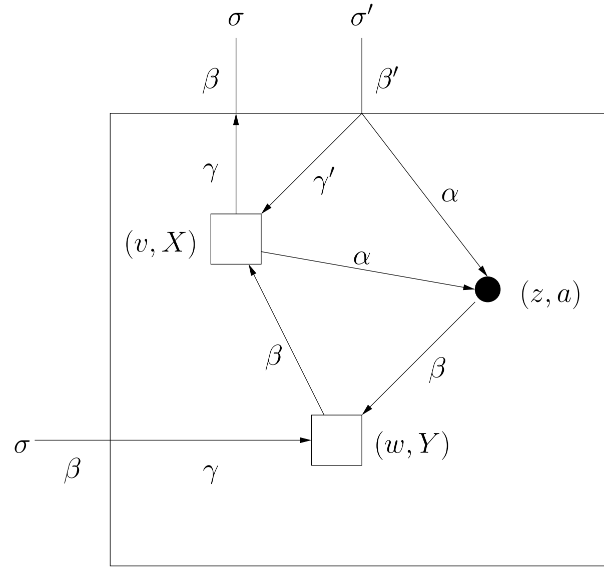
**Définition 6.** *Un graphe avec plongement est la donnée d'un couple $(H, C_H)$ où $H$ est un graphe et $C$ est un ensemble d'instructions de connexion.*

On peut représenter graphiquement un graphe avec plongement de manière simple (voir l'exemple suivant) : on rajoute au graphe des arcs qui n'ont qu'une seule attache pour coder les instructions de connexion. Par exemple, l'instruction $(\alpha, \beta/\gamma, x, out)$ va être dessinée comme un arc sortant du sommet $x$. On indiquera au bout de cette flèche l'attribut $\alpha$ et sur la flèche, on ajoutera les étiquettes $\beta$ et $\gamma$ (pour plus de lisibilité, on peut coupler les représentations de ces instructions si ces dernières se rapportent au même arc dans le graphe initial).

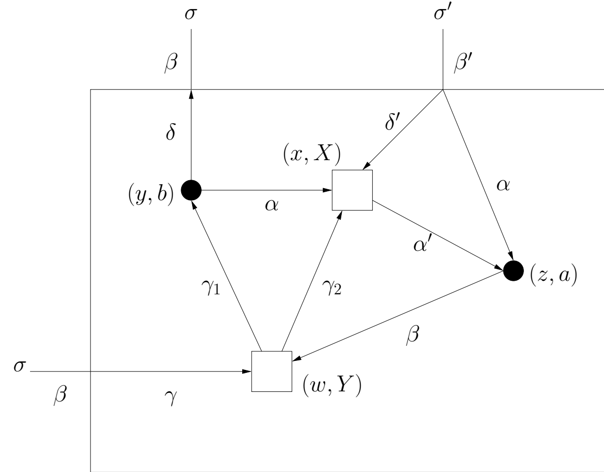
*Exemple.* On définit le graphe $D$ avec $V_D = \{x, y\}$, $\lambda_D(x) = X$, $\lambda_D(y) = b$ et $E_D = \{(y, \alpha, x)\}$. Les instructions de connexion sont les suivantes : $(\sigma, \gamma/\delta, y, out)$, $(\sigma', \gamma'/\delta', x, in)$, $(Y, \beta/\gamma_1, y, in)$, $(Y, \beta/\gamma_2, x, in)$ et $(a, \alpha/\alpha', x, out)$. Ce graphe avec plongement est représenté figure I.4.

On peut maintenant définir la substitution avec plus de précisions :

**Définition 7.** *Soient $(H, C_H)$ et $(D, C_D)$ deux graphes avec plongement tels que $V_H$ et $V_D$ soient disjoints et soit $v$ un sommet de $H$. La substitution de $v$ par $(D, C_D)$ dans $(H, C_H)$ (notée $(H, C_H)[v/(D, C_D)]$)*




(a) Le graphe avec plongement $(H, C_H)$




(b) Le résultat de la substitution

FIG. I.5 – La substitution $(H, C_H)[v/(D, C_D)]$

*est un nouveau graphe avec plongement défini par :*

$$
\begin{aligned}
V &= (V_H \backslash \{v\}) \cup V_D \\
E &= \big\{(x, \gamma, y) \in E_H \mid x \neq v, y \neq v\big\} \cup E_D \\
&\quad \cup \big\{(w, \gamma, x) \mid \exists \beta \in \Gamma \textit{ tel que } (w, \beta, v) \in E_H \textit{ et } (\lambda_H(w), \beta/\gamma, x, in) \in C_D\big\} \\
&\quad \cup \big\{(x, \gamma, w) \mid \exists \beta \in \Gamma \textit{ tel que } (v, \beta, w) \in E_H \textit{ et } (\lambda_H(w), \beta/\gamma, x, out) \in C_D\big\} \\
\lambda(x) &= \begin{cases} \lambda_H(x) & si\ x \in V_H \backslash \{v\} \\ \lambda_D(x) & si\ x \in V_D \end{cases} \\
C &= \big\{(\alpha, \beta/\gamma, x, d) \in C_H \mid x \neq v\big\} \\
&\quad \cup \big\{(\alpha, \beta/\delta, x, d) \mid \exists \gamma \in \Gamma \textit{ tel que } (\alpha, \beta/\gamma, v, d) \in C_H \textit{ et } (\alpha, \gamma/\delta, x, d) \in C_D\big\}.
\end{aligned}
$$

*Exemple.* On montre ici un exemple de substitution. Pour le graphe $(D, C_D)$, on reprend celui de la figure I.4. Le graphe avec plongement $(H, C_H)$ est dessiné figure I.5(a). Le résultat est représenté figure I.5(b).

**Propriété.** (Associativité)

Soit $K$, $H$ et $D$ trois graphes avec plongement mutuellement disjoints. Soit $w$ un sommet du graphe $K$ et $v$ un sommet du graphe $H$. Alors

$$\big(K[w/H]\big)[v/D] = K\big[w/(H[v/D])\big].$$

## 2.2 Les grammaires *edNCE*

**Définition 8.** *Une grammaire edNCE est donnée par un n-uplet $G = (\Sigma, \Delta, \Gamma, P, S)$ où $\Sigma$ est l'alphabet d'étiquettes des nœuds, $\Delta$ est un sous-ensemble de $\Sigma$ représentant les éléments terminaux de ce dernier, $\Gamma$ est l'alphabet d'étiquettes des arcs, $P$ est un ensemble fini de productions (de la forme $X \to (D, C)$ où $X \in \Sigma \backslash \Delta$) et $S \in \Sigma \backslash \Delta$ est le symbole initial.*

Soit $G = (\Sigma, \Delta, \Gamma, P, S)$ une grammaire *edNCE* et $p : X \to (D, C)$ une production de $G$ Soient $H$ et $H'$ deux graphes avec plongement et $v$ un sommet de $H$. On écrit $H \Longrightarrow_{v,p} H'$ si $\lambda_H(v) = X$ et $H' = H[v/(D, C)]$. Comme auparavant, si une suite de telles applications permet de passer du graphe $H_0$ au graphe $H_n$, on écrit $H_0 \Longrightarrow^* H_n$.

Soit $S$ une étiquette de nœud non terminale. On note $sn(S, z)$ le graphe avec plongement possédant un seul sommet $z$ étiqueté par $S$ et un ensemble d'instructions de connexion vide. On appelle le langage généré par la grammaire $G$ l'ensemble suivant :

$$L(G) = \big\{H \text{ tel que } sn(S, z) \Longrightarrow^* H\big\}.$$

*Exemple.* On montre ici une grammaire *edNCE* $G_{arbre}$ qui permet de générer tous les arbres binaires avec un arc supplémentaire pour chaque feuille de l'arbre qui remonte vers la racine. On pose ici :

$$\Sigma = \{S, X, \#\},$$

$$\Delta = \{\#\},$$

$$\Gamma = \{r, *\}.$$

Les étiquettes # et $*$ sont pour les nœuds et les arcs non étiquettés.

Il y a ici trois productions représentées figures I.6(a), I.6(b) et I.6(c). Dans la figure I.7, on donne un exemple de dérivation associée à cette grammaire.

## 2.3 Les propriétés «context-free»

**Définition 9.** *Une grammaire edNCE $G = (\Sigma, \Delta, \Gamma, P, S)$ est dite confluente si pour toute paire de productions $X_1 \to (D_1, C_1)$ et $X_2 \to (D_2, C_2)$, pour tout nœud $x_1 \in V_{D_1}$ et tout nœud $x_2 \in V_{D_2}$ et pour toute paire d'étiquettes d'arc $\alpha$ et $\delta$, la propriété suivante est vérifiée :*

$$\exists \beta \in \Gamma \text{ tel que } (X_1, \alpha/\beta, x_1, out) \in C_1 \text{ et } (\lambda_{D_1}(x_1), \beta/\delta, x_2, in) \in C_2$$

$$\Longleftrightarrow$$

$$\exists \gamma \in \Gamma \text{ tel que } (X_1, \alpha/\gamma, x_2, in) \in C_2 \text{ et } (\lambda_{D_2}(x_2), \gamma/\delta, x_1, out) \in C_1.$$

**Proposition 1.** *Une grammaire edNCE $G = (\Sigma, \Delta, \Gamma, P, S)$ est confluente si et seulement si la propriété suivante est vraie pour tout graphe $H \in L(G)$ : si on se donne deux dérivations (de la grammaire $G$)*

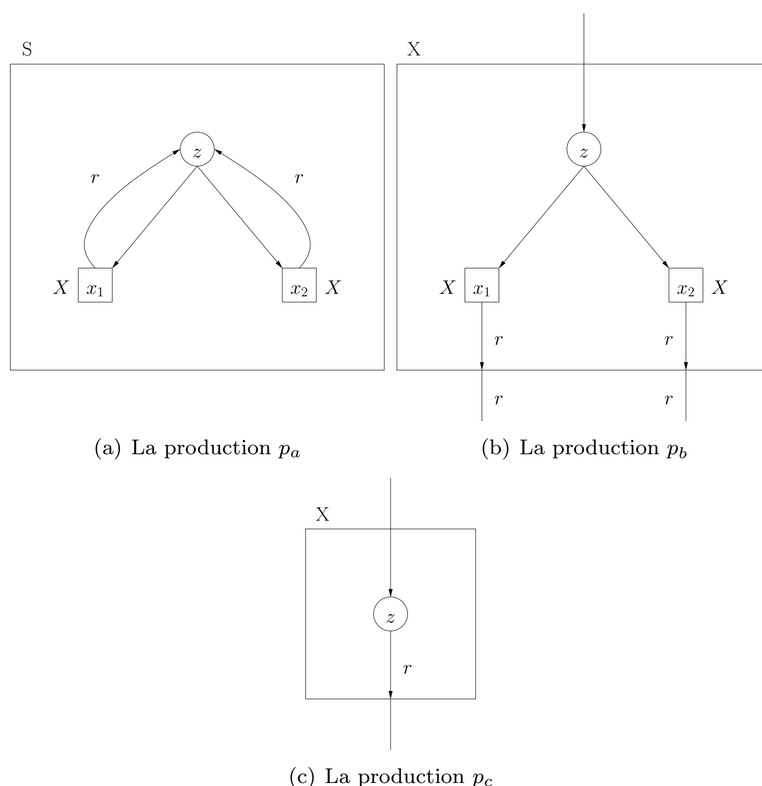
$$H \Longrightarrow_{u_1,p_1} H_1 \Longrightarrow_{u_2,p_2} H_{12}$$

*et*

$$H \Longrightarrow_{u_2,p_2} H_2 \Longrightarrow_{u_1,p_1} H_{21}$$

*avec $u_1$ et $u_2$ des sommets distincts de $H$, alors*

$$H_{12} = H_{21}.$$




(a) La production $p_a$

(b) La production $p_b$

(c) La production $p_c$

FIG. I.6 – Les productions de la grammaire $G_{arbre}$

# 3 Discussion

Les deux approches présentées ici offrent chacune des possibilités très intéressantes pour l'étude de grammaires de graphes grâce à des propriétés très fortes telles que la confluence. Pour cela, elles se basent sur des généralisations des grammaires textuelles : les transformations sont basées sur le remplacement d'un élément atomique du graphe (sommet ou arc). Il est alors naturel de trouver dans ces approches des caractéristiques d'indépendance vis-à-vis du contexte car on ne s'intéresse pas à ce qui entoure l'élément atomique avant d'appliquer la règle.

Si l'on est intéressé par la manipulation des modèles, ces approches permettent certains traitements intéressants comme par exemple la génération de tous les diagrammes d'activité UML possibles à l'aide de règles comme celle présentée dans la figure I.8 (*cf.* [41]).

Pour autant, les opérations effectuées lors de transformations de modèles nécessitent très souvent des informations liées au contexte afin de pouvoir être appliquées. On peut par exemple citer le cas du *refactoring* (*cf.* [54, 71]) pour lequel il est indispensable de connaître le contexte pour empêcher certaines règles de s'appliquer au mauvais moment.

De plus, toujours dans l'exemple du *refactoring*, il est nécessaire de pouvoir travailler avec les attributs lors des transformations (par exemple, ajouter un prefixe à une chaîne de caractères quelconque lors de l'application de la règle) ; or ces deux approches ne permettent pas de faire des calculs avec les attibuts. Enfin, une caractéristique recherchée pour les transformations de modèles est la réversibilité des modifications apportées à un graphe et les deux approches détaillées dans cette section ne le permettent pas.

Dans le cadre des transformations de modèles, l'approche «Hyperedge Replacement» et l'approche «Node Replacement» ne vont donc pas véritablement convenir.

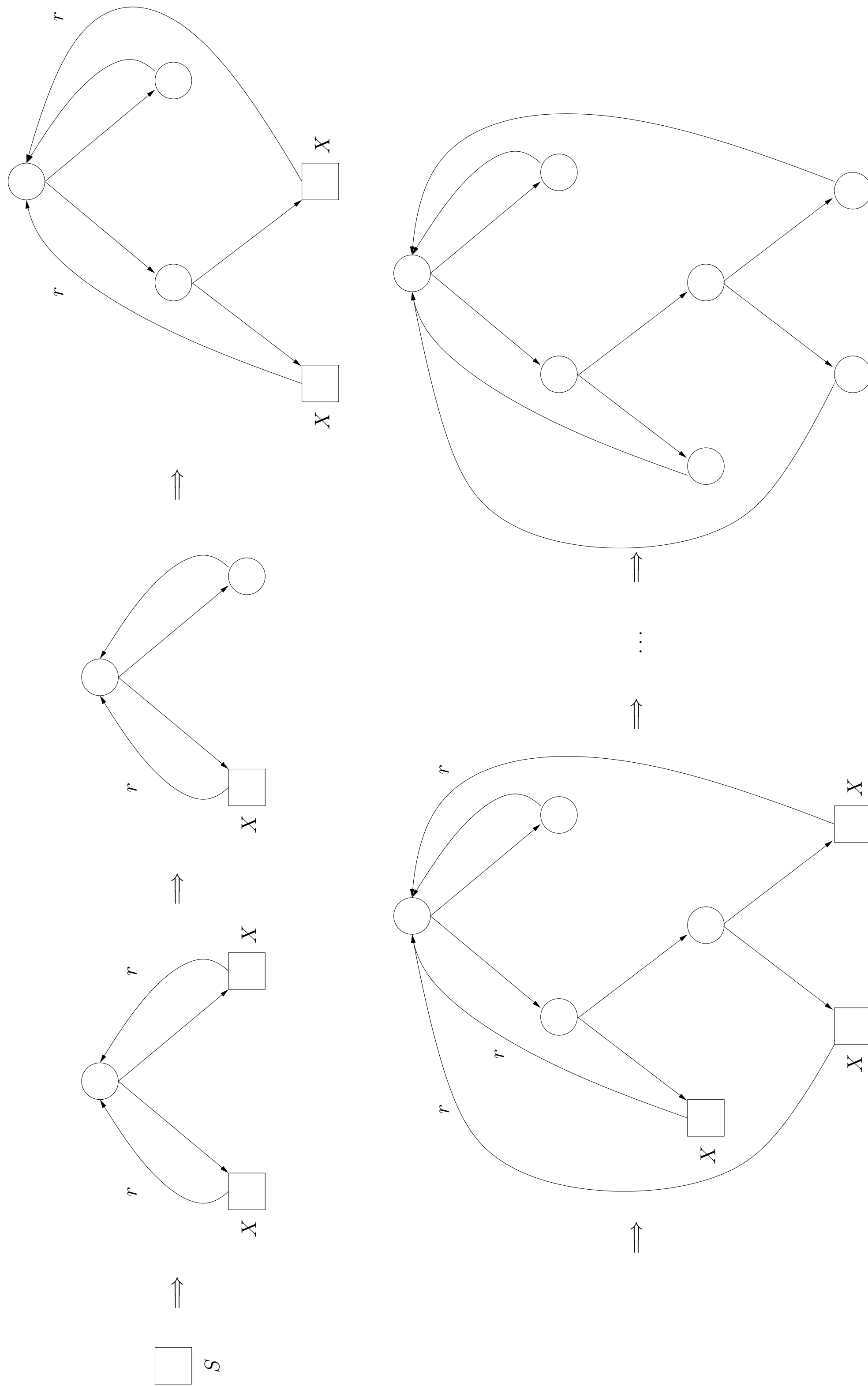
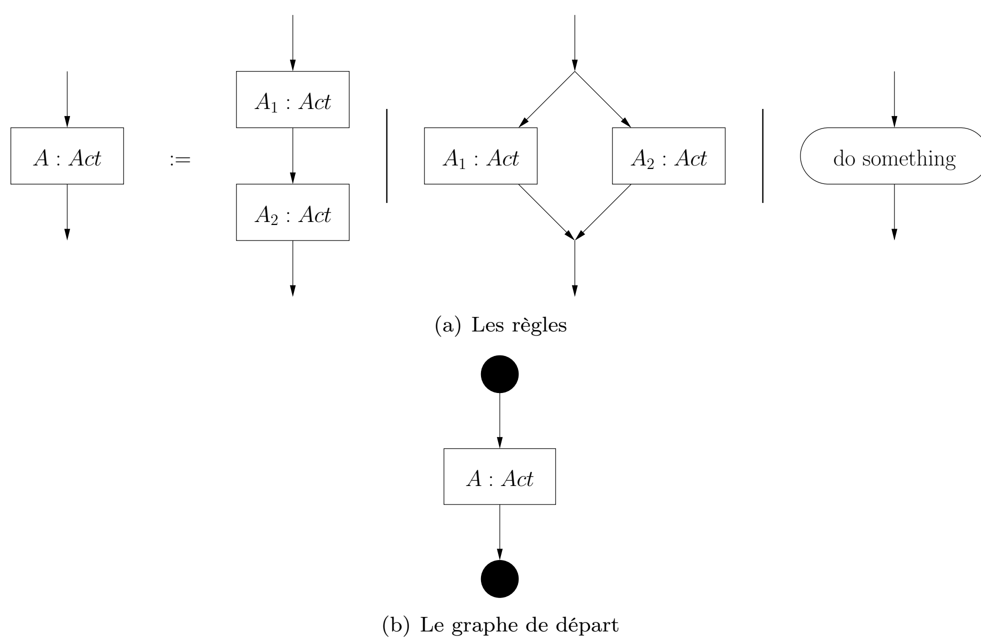



FIG. I.7 – Une dérivation dans la grammaire $G_{arbre}$




(a) Les règles

(b) Le graphe de départ

FIG. I.8 – La grammaire pour générer les diagrammes d'activité UML

# Chapitre II

# Les approches catégoriques

Initialement, les approches catégoriques de la récriture de graphe ont été introduites au cours des années 70 par EHRIG, PFENDER et SCHNEIDER (*cf.* [31, 21]) afin de généraliser aux graphes les grammaires de CHOMSKY définies sur les chaînes de caractères. Ces approches ont connu un grand succès car le cadre théorique offrent un excellent outil de travail dont la puissance permet de démontrer un grand nombre de propriétés. Beaucoup de variantes ont été proposées au cours des décennies qui ont suivies. On présente ici deux méthodes reposant chacune d'elles sur les deux principales opérations catégoriques employées dans la récriture de graphes. La première s'appuie sur le pullback et la seconde, qui est en réalité l'approche originelle, sur le pushout.

Une breffe introduction à la théorie des catégories avec les définitions utiles à la compréhension de ce chapitre se trouve en annexe A.

## 1 L'approche par «pullback»

Cette théorie a été introduite par M. BAUDERON en 1995 dans l'article [4]. Cette approche utilise principalement le pullback, outil issu de la théorie des catégories (*cf.* annexe A), pour construire les transformations de graphes. Cette construction permet de réinterpréter certaines théories à l'aide des catégories : on propose de montrer ici comment catégoriser la variante *NLC* (pour Node Labelled Controlled) de l'approche *Node Replacement*.

### 1.1 Graphe, produit et pullback

**Définition 10.** *Un graphe $G$ est donné par une paire $(V, E)$ où $V$ est un ensemble de sommets et $E$ est un sous-ensemble de $V \times V$ représentant les arêtes.*

Un sommet $v$ est dit réflexif si le couple $(v, v)$ appartient à $E$. Pour un ensemble $S$, le graphe complet $K_S$ au-dessus de $S$ est défini par la paire $(S, S \times S)$.

On prend ici la définition naturelle pour les morphismes de graphes (*cf.* [37]) : c'est-à-dire que le morphisme va être composé de deux fonctions, une pour les sommets et une pour les arêtes vérifiant une condition de compatibilité : l'image de la source d'une arête (*i.e.* le premier élément du couple) doit être la source de l'image de l'arête ; de même pour la cible de chacun des arcs.

Dans la suite, pour plus de clarté, on se restreint aux graphes non orientés. Cela se traduit par

$$(u, v) \in E \iff (v, u) \in E.$$

**Remarque 5.** Les notions de produit et de pullback développées ci-dessous sont des notions catégoriques et sont donc définies de la manière la plus générale possible dans les catégories à l'aide de propriétés universelles (*cf.* annexe 4). Par exemple, le pullback de deux flèches $b' : C \to D$ et $c' : B \to D$, défini par un objet $A$ et et de deux morphismes $b : A \to B$ et $c : A \to C$, est caractérisé par la propriété universelle suivante : pour tout objet $E$ appartenant à la catégorie $\mathbf{C}$, pour toutes flèches $f : E \to B$ et $g : E \to C$ tels que $c' \circ f = b' \circ g$, alors il existe une flèche $a : E \to A$ qui

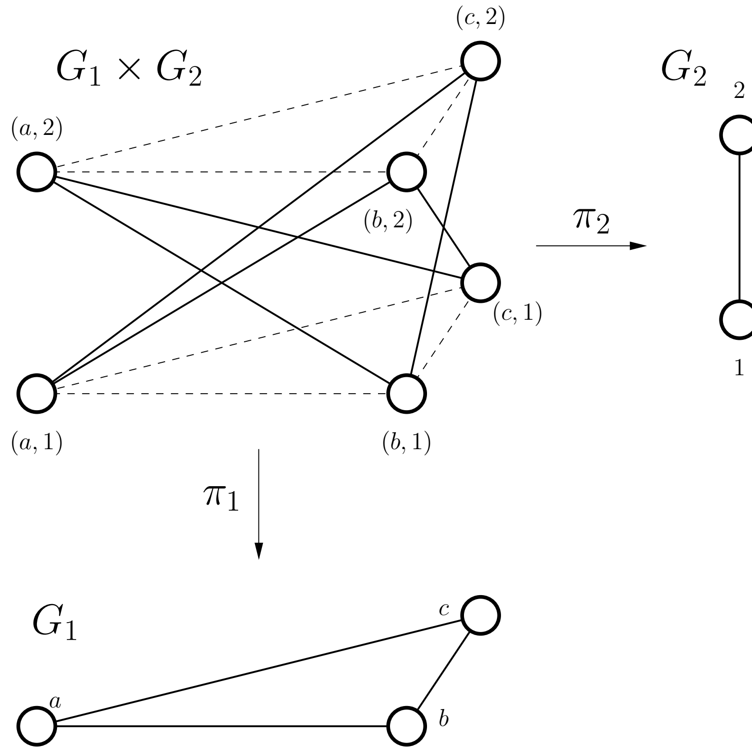
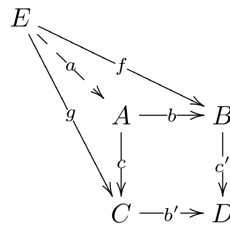



FIG. II.1 – Un exemple de produit

factorise $f$ et $g$, c'est-à-dire $f = b \circ a$ et $g = c \circ a$.




Pour autant, dans le cadre de travail présent, on peut se contenter de travailler de manière ensembliste, ce qui permet d'avoir une meilleure vision des constructions.

**Proposition 2.** *Soient $G_1$ et $G_2$ deux graphes. Le produit $G = G_1 \times G_2$ est donné par :*
- $V = V_1 \times V_2$;
- $E = \{((u_1, u_2), (v_1, v_2))$ *tel que* $(u_1, v_1) \in E_1 \wedge (u_2, v_2) \in E_2\}$.

On associe à ce produit deux projections $\pi_i : G \to G_i$ dont la définition est évidente.

*Exemple.* La figure II.1 montre un exemple simple de produit de graphes.

**Proposition 3.** *Soient $G_1$, $G_2$ et $F$ trois graphes et $f_1 : G_1 \to F$ et $f_2 : G_2 \to F$ deux morphismes. Le pullback de $f_1$ et $f_2$ est donné par le plus grand sous-graphe $H$ de $G_1 \times G_2$ sur lequel les morphismes $f_1 \circ \pi_1$ et $f_2 \circ \pi_2$ coïncident. Les morphismes associés sont alors les restrictions des projections $\pi_1$ et $\pi_2$.*

*Exemple.* Dans le figure II.2, on donne un exemple de pullback. Pour calculer simplement ce pullback, on commence par chercher quels sommets vérifient les bonnes conditions : en regardant les morphismes $f_1$ et $f_2$, on s'aperçoit que le sommet $a$ et les sommets 1 et 3 ont la même image, de même que les sommets $b$ et $c$ et le sommet 2. On en déduit donc que le pullback va contenir 4 sommets : $(a, 1)$, $(a, 3)$, $(b, 2)$ et $(c, 2)$. Pour les arcs, on regarde si les différentes arêtes possibles

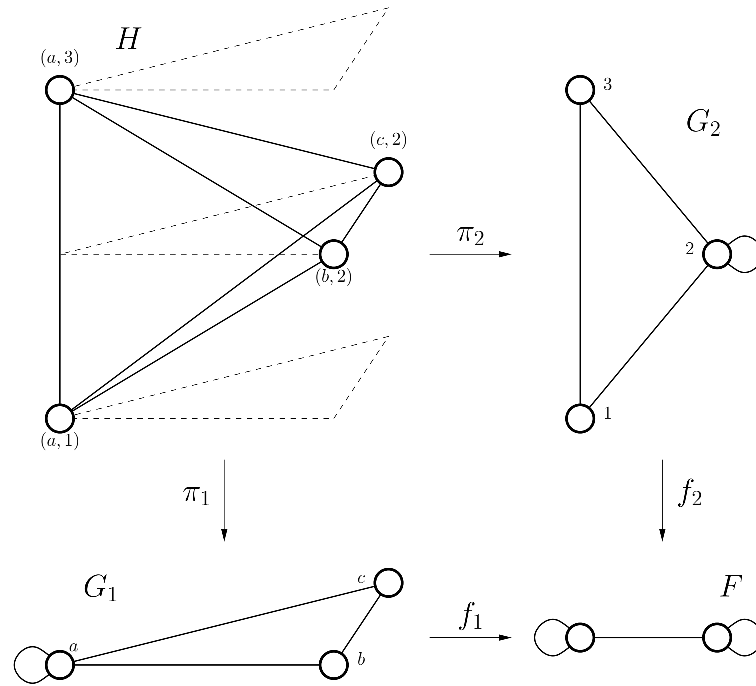



FIG. II.2 – Un exemple de pullback

sont compatibles avec la condition de commutativité. On s'aperçoit ici que les six arcs possibles entre deux sommets différents doivent être présents dans le pullback, mais aucune boucle n'apparait dans le résultat.

Cet exemple peut s'interpréter comme une récriture de graphes avec l'approche *Node Replacement*. Pour cela, on interprète le morphisme $f_2$ comme étant une règle de récriture qui prend le sommet de gauche du graphe $F$ et va le remplacer par deux nouveaux sommets (les sommets 1 et 3). Ces sommets vont de plus être reliés à tous les autres nœuds du graphes car dans le graphe $G_2$ les sommets 1 et 3 sont reliés au sommet 2.

## 1.2 Récriture de nœud

Afin de formaliser l'idée précédente, on a besoin d'un graphe particulier qui va nous permettre de différencier le nœud à récrire de ses voisins d'une part et du reste du graphe d'autre part. C'est le rôle du graphe alphabet.

#### *1.2.4 Le graphe alphabet*

Dans la suite de ce chapitre, le symbole $\mathbb{N}^+$ représente l'ensemble des entiers naturels strictement positifs.

**Définition 11.** *Le graphe alphabet $\mathcal{A}$ est défini comme le graphe complet $K_{\mathbb{N}^+}$ auquel on rajoute deux sommets réflexifs $0$ et $-1$ que l'on va relier à tous les autres sommets. En pratique, on pourra restreindre ce graphe alphabet en remplaçant l'ensemble $\mathbb{N}^+$ par un intervalle du type $[\![1, k]\!]$ ; on notera alors ce nouveau graphe $\mathcal{A}_k$.*

*Exemple.* La figure II.3 montre le graphe alphabet $\mathcal{A}_3$.

L'idée derrière ce graphe est la suivante : le nœud $-1$ va représenter le nœud à remplacer, les sommets avec des étiquettes strictement positives vont s'identifier aux voisins immédiats de ce premier nœud portant toutes les étiquettes possibles de l'alphabet de départ (on va ainsi pouvoir coder les instructions de connexion dans le règle de récriture) et enfin le nœud 0 va représenter le reste du graphe qui n'intervient pas dans la transformation. Le graphe $\mathcal{A}$ (ou bien l'une de ses

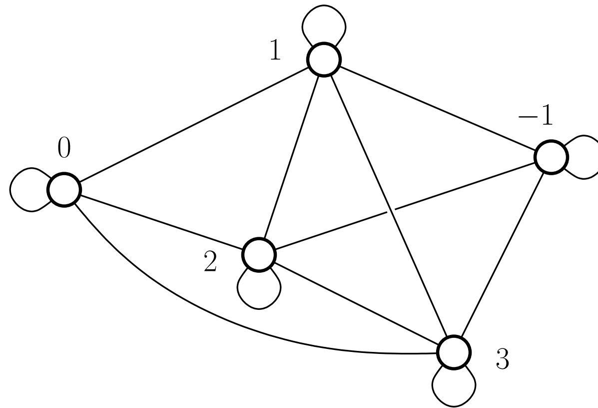



FIG. II.3 – Le graphe alphabet $\mathcal{A}_3$

restrictions) vont donc avoir un rôle fondamental dans l'approche de transformations par pullback car il va intervenir dans chacune des règles de récriture.

#### 1.2.5 Récriture

**Définition 12.** *Une règle de récriture est un morphisme $r : R \to \mathcal{A}$, où $R$ est un graphe, vérifiant les condition suivantes :*
- *l'ensemble $r^{-1}(0)$ possède exactement un élément ;*
- *les ensembles $r^{-1}(i)$ pour $i \in \mathbb{N}^+$ possèdent au plus un élément.*

Intuitivement, la préimage par $r$ du sommet $-1$ et sa boucle représente la nouvelle structure à substituer à un nœud.

**Définition 13.** *Soit $G$ un graphe. Une variable est un morphisme $a : G \to \mathcal{A}$ tel que l'image réciproque de $-1$ possède exactement un élément et que ce sommet soit réflexif. De plus, les antécédents des sommets strictement positifs doivent être des voisins immédiats de l'antécédent de $-1$.*

Intuitivement, le sommet qui est envoyé sur $-1$ est le sommet que l'on va remplacer par un nouveau graphe.

**Définition 14.** *Soit $G$ un graphe, $a$ une variable sur $G$ et $r : R \to \mathcal{A}$ une règle. Le résultat de l'application de la règle $r$ à $G$ est donné par le pullback des morphismes $a$ et $r$.*

#### 1.2.6 Traduction d'une règle Node Replacement dans ce formalisme

**Proposition 4.** *Toute règle de récriture de type NLC (cf. [35] pour une description de l'approche NLC) est exprimable dans ce formalisme.*

Soit $\rho = (a, X, C)$ une règle *NLC* défini sur un alphabet dénombrable $A$ (les élément de $A$ sont énumérés sous la forme : $a_1, a_2, \ldots$) où $a \in A$ est l'étiquette de nœud où $\rho$ peut être appliquée, $X$ est la partie droite de la règle et $C$ est l'ensemble des instructions de connexion. On commence par définir le graphe $R$ par
- l'ensemble des sommets de $R$ est composé des sommets du graphe $X$, d'un sommet $a_x$ pour chaque élément $x$ de $A$ et d'un sommet supplémentaire $u$ ;
- l'ensemble des arêtes de $R$ est composé des arêtes du graphe $X$, d'un arc entre chaque sommet $a_x$ et $u$ et si la relation $(x, b)$ appartient à $C$, alors on rajoute un arc entre $a_x$ et chaque sommet étiqueté par $b$ dans le graphe $X$.

Le morphisme $r : R \to \mathcal{A}$ est quant à lui défini par
- $r(u) = 0$ ;
- $r(i) = i$ pour tout $i \in \mathbb{N}^+$ ;
- $r(v) = -1$.

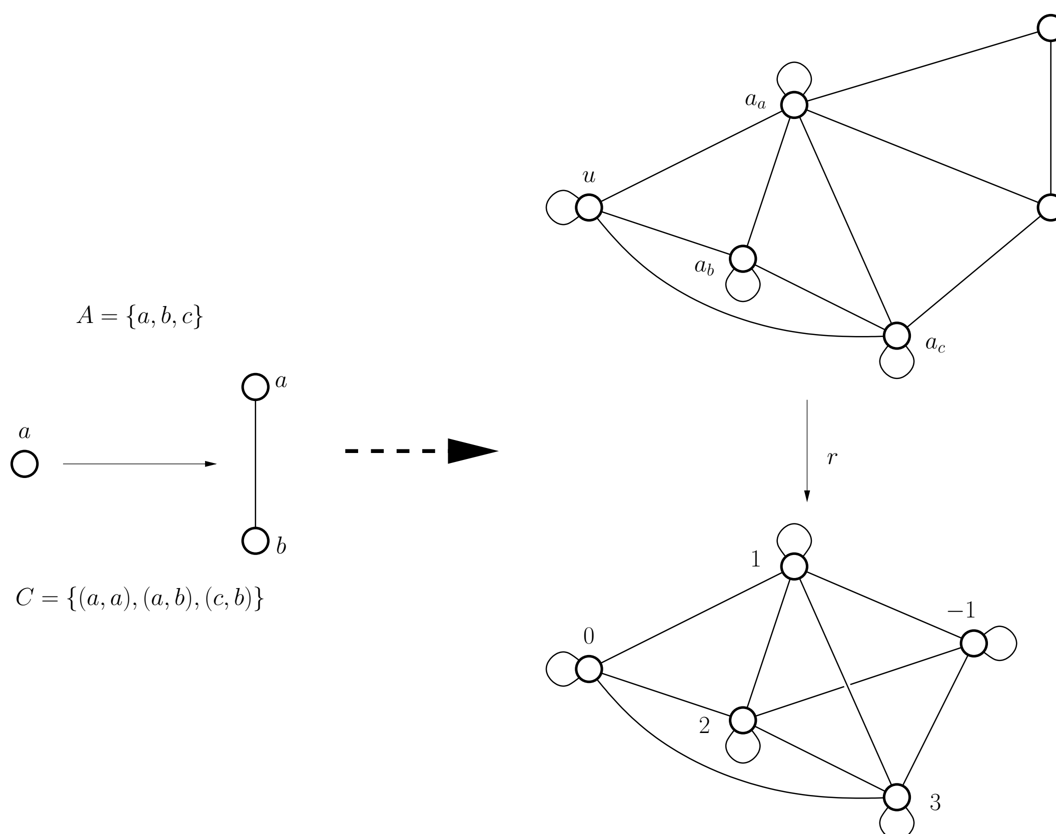



FIG. II.4 – Une règle *NLC* et son équivalent dans l'approche pullback

*1.2.7 Exemple*

La figure II.4 montre la traduction d'une règle *NLC* dans l'approche pullback. L'alphabet possède ici trois éléments, c'est pourquoi on restreint le graphe $\mathcal{A}$ au graphe $\mathcal{A}_3$.

Dans la figure II.5, on montre l'application de la règle précédente à un graphe simple (en bas à gauche dans la figure) ; le sommet récrit étant le sommet le plus à droite du graphe. Le résultat est le graphe en haut à gauche.

## 2 L'approche par «pushout»

L'idée d'utiliser les catégories pour enrichir les grammaires de graphes a été introduite au début des années 70 par H. Ehrig, M. Pfender et H.J. Schneider [31, 21]. Le but était alors de généraliser les grammaires de CHOMSKY connues sur les chaînes de caractères aux graphes et donc de pouvoir décrire de manière algébrique des règles de transformation complexes ainsi que le processus de transformation. Ce dernier repose sur une construction catégorique appelé le pushout. Pour transformer un graphe, deux pushouts vont être calculés : le premier pour enlever les éléments du graphe qui n'appartenant plus au graphe cible et le second pour ajouter les nouveaux éléments (et les relier au reste du graphe). L'interêt de cette approche et de ses dérivés réside dans la possibilité grâce à la solide base théorique de démontrer des propriétés puissantes sur les systèmes de récriture, comme par exemple, le paralellisme [25], la confluence ou bien encore la terminaison [23]. . .De plus, la souplesse d'utilisation des graphes permet d'adapter cette approche à beaucoup de problèmes dans des domaines variés. Un outil implémentant cette théorie a d'ailleurs été développé à cet effet ; il s'agit de l'environnement AGG [72] qui offre un moyen visuel très simple de définir les règles de transformation avant de les appliquer à un graphe [22]. Enfin, un des avantages de l'approche originelle est la propriété de reversibilité des règles qui s'inscrit dans le cadre du *reverse engineering*. On peut d'ailleurs signaler qu'un des dérivés les plus répandus de ce cadre

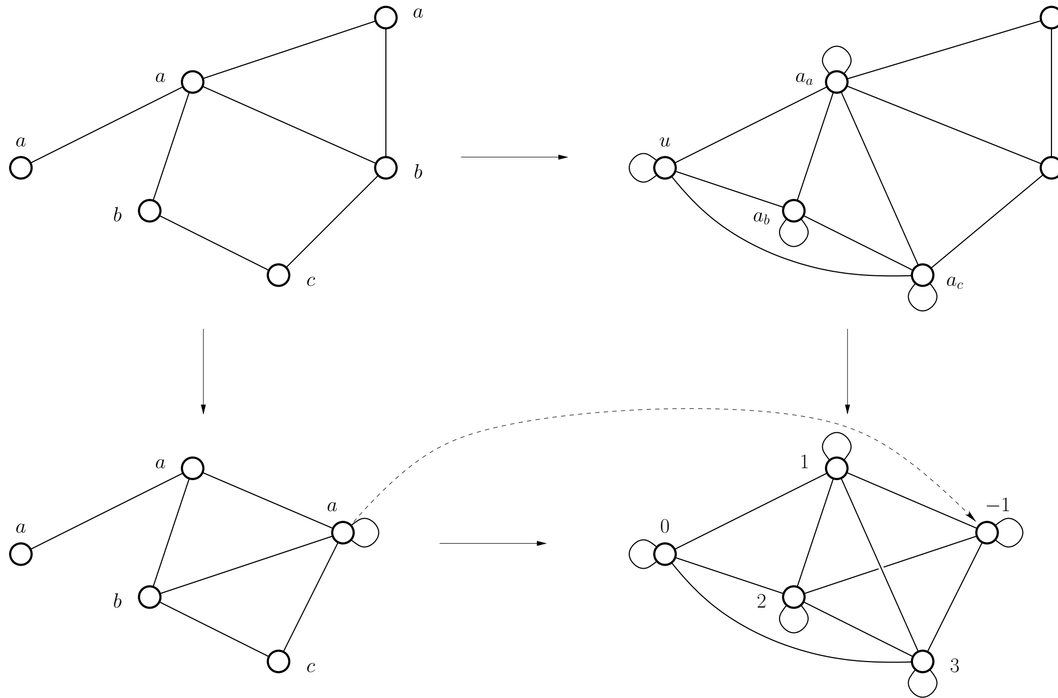



FIG. II.5 – L'application de la règle de la figure II.4 à un graphe simple

théorique, l'approche « *Single Pushout* » [50, 28, 27], perd cette propriété de reversibilité dans le but d'autoriser plus facilement l'exécution d'une règle.

## 2.1 Graphes, morphismes et catégorie

On donne ici les définitions formelles de l'approche «double pushout» (en abrégé, DPO), historiquement introduite dans [31] et [21]. On pourra lire une introduction plus complète dans [32].

**Définition 15.** *On se donne deux alphabets $\Omega_S$ et $\Omega_A$ afin d'étiqueter respectivement les sommets et les arêtes des graphes.*

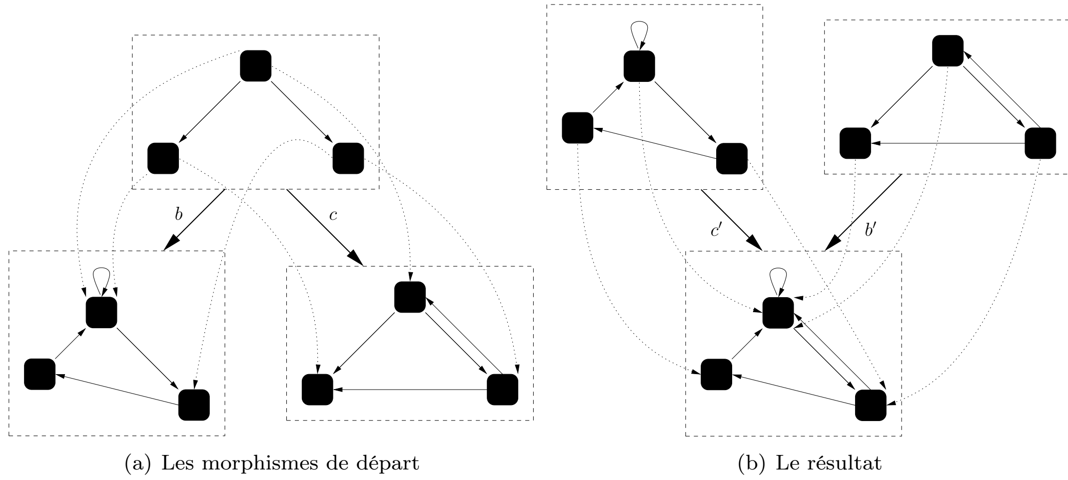
*Un graphe $G$ est défini par :*
- *un ensemble $G_S$ représentant l'ensemble des sommets ;*
- *un ensemble $G_A$ pour les arêtes ;*
- *deux applications $s : G_A \rightarrow G_S$ et $t : G_A \rightarrow G_S$ qui associent respectivement à chaque arc sa source et sa cible.*
- *deux applications $l_S : G_S \rightarrow \Omega_S$ et $l_A : G_A \rightarrow \Omega_A$ qui associent à chaque élément du graphe son étiquette.*

*Un morphisme $f$ entre deux graphes $G$ et $G'$ est défini par deux applications $f_S : G_S \rightarrow G'_S$ et $f_A : G_A \rightarrow G'_A$ préservant les sources et les cibles des arêtes ainsi que les étiquettes, c'est-à-dire :*
- *pour les sources : $f_S \circ s^G = s^{G'} \circ f_A$ ;*
- *pour les cibles : $f_S \circ t^G = t^{G'} \circ f_A$ ;*
- *pour les étiquettes : $l_S^G = l_S^{G'} \circ f_S$ et $l_A^G = l_A^{G'} \circ f_A$ ;*

Les graphes ainsi que les morphismes de graphes forment une catégorie, noté **Graph**.

Avant de donner les propriétés relatives au pushout dans **Graph**, on commence par rappeler les définitions de ce dernier et du pushout-complement car dans le reste de cette thèse, nous ferons constamment appel à ces deux constructions :

**Définition 16.** *Soient $A$, $B$ et $C$ trois objets de la catégorie* **C** *et $b : A \rightarrow B$ et $c : A \rightarrow C$ deux flèches de cette catégorie. Le pushout de $b$ et $c$ sera donné par*
- *un objet $D$ ;*

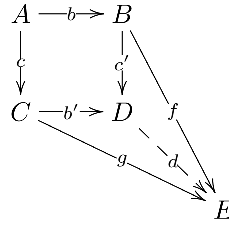



(a) Les morphismes de départ (b) Le résultat

FIG. II.6 – Construction d'un pushout

– *deux flèches $b' : C \to D$ et $c' : B \to D$*

*vérifiant la propriété universelle suivante : pour tout objet $E$ appartenant à la catégorie* **C***, pour toutes flèches $f : B \to E$ et $g : C \to E$ tels que $f \circ b = g \circ c$, alors il existe une flèche $d : D \to E$ qui factorise $f$ et $g$, c'est-à-dire $f = d \circ c'$ et $g = d \circ b'$.*




Intuitivement, le pushout est une construction consistant à coller deux objets de la catégorie le long d'un sous-graphe défini par un troisième objet.

**Définition 17.** *Soit $b : A \to B$ et $c' : B \to D$ deux flèches dans la catégorie* **C***. On dira que le pushout-complement de $b$ et $c'$ existe si on peut trouver un objet $C$ et deux flèches $c : A \to C$ et $b' : C \to D$ tels que le carré $ABCD$ soit un pushout.*

**Proposition 5.** *Dans la catégorie* **Graph***, le pushout de deux morphismes $b : A \to B$ et $c : A \to C$ existe toujours.*

Comme les graphes que l'on considère ici sont décrits de manière ensembliste, on peut donner une construction ensembliste du pushout. Pour cela, on introduit une relation $\mathcal{R}$ sur le graphe $A$ défini par la condition suivante : deux éléments (sommets ou arcs) $x$ et $y$ du graphe $A$ seront en relation si $b(x) = b(y)$ ou $c(x) = c(y)$. Grâce à cette relation, on peut facilement décrire le pushout $D$ de $b$ et $c$ par :

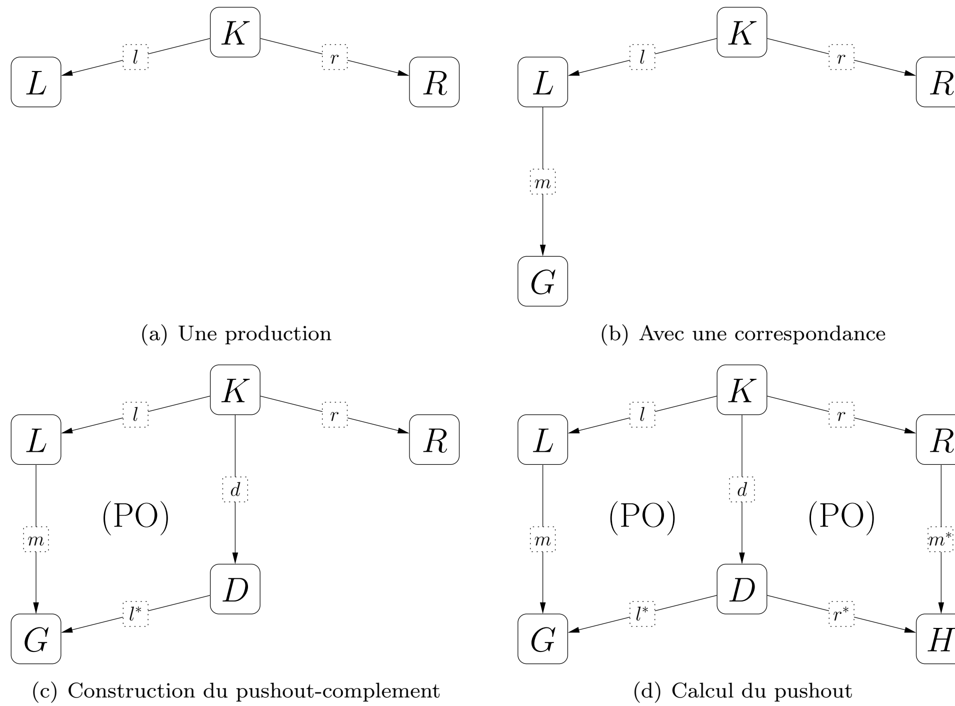
$$D = A/\mathcal{R} + B \backslash b(A) + C \backslash c(A).$$

Pour construire ce pushout, on décompose les graphes et les morphismes en deux parties pour se placer dans la catégories des ensembles : on considère tout d'abord les sommets et les applications entre les sommets et ensuite les arcs et les applications associées.

*Exemple.* La figure II.6 montre un exemple de pushout.

**Proposition 6.** *Soit $b : A \to B$ et $c' : B \to D$ deux morphismes de graphes. Le pushout-complement de $b$ et $c'$ existe si et seulement si les conditions suivantes sont remplies :*

– *il n'existe pas d'arête $e \in D_A \backslash c'_A(B_A)$ tel que $s^D(e)$ ou $t^D(e)$ appartienne à $c'_S(B_S \backslash b_S(A_S))$ ;*
– *il n'existe pas de couple $x, y$ appartenant à $B_S \cup B_A$ tel que $x \neq y$, $c'(x) = c'(y)$ et $y \notin b(A_S \cup A_A)$.*




(a) Une production (b) Avec une correspondance

(c) Construction du pushout-complement (d) Calcul du pushout

FIG. II.7 – Une transformation de graphe pas à pas

*Ces deux conditions sont regroupées sous le nom de condition de collage ou «gluing condition». Si de plus le morphisme $b$ est injectif, alors le pushout-complement est unique à isomorphisme près.*

Cette condition de collage est utile car elle assure *a priori* qu'on ne va pas effacer au cours de la construction un sommet qui fera perdre la cohérence du graphe : en effet, si l'on n'y prend pas garde, certaines arêtes peuvent ne plus avoir de source ou de cible à la fin de la construction, violant ainsi la définition d'un graphe.

*Exemple.* On reprend ici les morphismes $b$ et $c'$ de la figure II.6. Comme le morphisme $b$ n'est pas injectif, le résultat n'est pas unique : en effet, on va soit pouvoir retrouver le graphe $C$, soit trouver le même graphe où les deux sommets inférieurs auront fusionnés.

## 2.2 Productions et grammaires de graphes

**Définition 18.** *Une production (ou règle de transformation) $p$ est composée de trois graphes $K$, $L$ et $R$, appelés respectivement l'interface, la partie gauche et la partie droite de $p$, ainsi que deux morphismes $l : K \to L$ et $r : K \to R$.*

*Une grammaire de graphes $\mathcal{G}$ est la donnée d'un ensemble de productions et d'un graphe de départ $G_0$.*

Intuitivement, la partie gauche d'une production va représenter le sous graphe d'un graphe $G$ que l'on va modifier. Les éléments qui appartiennent à $L \backslash l(K)$ vont être effacés au cours de la transformation et ceux du graphe $K$ vont représenter les sommets et les arêtes conservés et enfin les éléments appartenant à $R \backslash r(K)$ vont être ajoutés pour obtenir le graphe cible de la transformation.

En détail, la transformation du graphe $G$ suivant la règle $p : L \leftarrow K \to R$ va donc se dérouler comme suit. On commence par chercher un morphisme $m$ entre $L$ et $G$. Ce morphisme, nommé correspondance, sert donc à repérer un sous graphe de $G$ correspondant à $L$. À partir des morphismes $l$ et $m$, s'ils vérifient la condition de collage, on va pouvoir construire le pushout-complement. On obtient alors un graphe $D$, appelé le graphe de contexte, accompagné de deux morphisme $d : K \to D$ et $l^* : D \to G$. Il reste alors à calculer le pushout des morphismes $d$ et $r$ pour obtenir le transformé $H$ de $G$ suivant la production $p$.

Par ailleurs, il est possible d'obtenir un contrôle plus fin sur l'exécution des transformations à l'aide des conditions d'application (*cf.* [39]). Ces conditions d'application permettent de définir des contextes autour des sous-graphes à remplacer permettant ainsi, par exemple, d'interdire l'application d'une règle si un certain sommet est presént. Une exemple classique est la grammaire de graphe permettant de calculer la fermeture transitive d'un graphe : elle possède une règle qui va chercher trois sommets tels que le premier est relié au second et le second au troisième et qui va alors ajouter une arête entre le premier et le troisième. Bien évidemment, si un tel arc existe déjà, on ne va pas vouloir appliquer la règle et une condition d'application va alors être introduite pour cela. On reviendra sur les conditions d'application et leur définition précise dans la section 4.

## 2.3 Les propriétés de cette approche

Le base théorique pour cette approche permet de démontrer de manière simple des propriétés très importantes pour l'étude des grammaires de graphes. On peut citer par exemple l'étude de l'indépendance parallèle et séquentielle ou bien l'analyse par paires critiques, on reviendra plus en détails sur ces propriétés dans les chapitres III et 3.

## 2.4 Évolutions du système

En modifiant ou enrichissant la catégorie **Graph**, il est possible d'obtenir des variations de l'approche DPO. Une des plus répandues est l'approche «simple pushout». Pour construire ce cadre théorique, on autorise dans la catégorie de départ les morphismes partiels $f : G \rightarrow H$, c'est-à-dire définis seulement sur un sous-graphe du graphe de départ $G$. Cette nouvelle définition des morphismes permet alors d'effacer et d'ajouter des éléments à un graphe en ne calculant qu'un seul pushout : en effet, si on définit une production par un morphisme partiel $l$ entre deux graphes $L$ et $R$, les éléments qui n'ont pas d'image par $l$ seront effacés lors du calcul du pushout et ceux qui n'ont pas de préimage par $l$ seront créés.

La principale différence avec l'approche «double pushout» se trouve dans la gestion des cas problématiques. Plus précisément, on a vu que l'application d'une production $p$ à un graphe $G$ dans l'approche DPO passe par la vérification de la condition de collage avant de calculer un pushout-complement afin d'éviter de trouver des arcs attachés à un seul ou aucun sommet dans le résultat final. Dans l'approche SPO, on calcule seulement un pushout, il n'y a donc aucune condition préalable à remplir avant de commencer la transformation. Comment sont alors traités ses arêtes problématiques ? Contrairement à l'approche DPO qui est conservatrice et interdit l'application de la règle en présence d'un tel arc, l'approche SPO est non-conservatrice et la construction explicite du pushout montre que les arêtes problématiques vont être naturellement effacées lors du calcul. Bien entendu, dans ce cas, on perd la caractéristique de réversibilité des transformations de graphes.

Un autre système dérivé de l'approche DPO consiste à enrichir la catégorie afin de pouvoir ajouter des attributs sur les graphes. Cet enrichissement sera détaillé dans le chapitre III.

# 3 Discussion

Ces systèmes catégoriques offrent donc une base théorique très solide pour transformer des graphes. Comme nous l'avons vu, l'approche par pullback permet de traduire le formalisme *NLC* dans un cadre catégorique. On peut trouver dans la littérature d'autres approches qui sont interprétables avec le pullback : dans [4] il est expliqué comment appliquer des règles de transformations *NCE* grâce à un double pullback et on peut lire dans [5] comment traduire l'approche *Hyperedge Replacement* à l'aide de graphes bipartites. Une autre possibilité offerte par le pullback est la copie simple de graphe (*cf.* [46]). En effet, grâce au fait que la catégorie **Graph** possède un élément final (un seul sommet avec une arête formant une boucle sur ce sommet), il suffit d'une seule règle pour pouvoir copier n'importe quel graphe, alors qu'avec le pushout, il est nécessaire de définir une règle pour chacun des graphe que l'on veut dupliquer.

L'approche par pushout nous donne donc un moyen très intuitif de coder les transformations de graphes. De plus, le manière de définir une production dans cette approche permet de définir un contexte pour appliquer la règle et il est par ailleurs possible d'ajouter à une règle des conditions

d'application afin d'empêcher l'application de cette production en présence d'un certain sommet ou d'une certaine arête (cette idée sera développée dans le paragraphe 4 du chapitre VI). Cette caractéristique, en plus des propriétés évoquées dans la section 2.3, en fait un candidat naturel pour représenter les transformations de modèle dans l'approche MDA. Pour cela, il faut enrichir la définition des graphes afin de prendre en compte les attributs portés par les modèles avec lesquels on travaille en génie logiciel. Cette possibilité va être développé dans le chapitre suivant.

Pour conclure cette discussion, on peut ajouter quelques remarques sur la complexité d'une récriture de graphes dans l'approche DPO. L'étape la plus gourmande en calcul et en temps de cette construction est la reconnaissance du sous-graphe isormorphe à la partie gauche $L$ de la règle dans le graphe source $G$. On sait que cette recherche est en réalité un problème NP-complet ; plus précisément, chercher un sous-graphe constitué de $l$ éléments dans un graphe composé de $n$ éléments possède une complexité en $O(n^l)$. Pour autant, cette compléxité exponentielle n'est pas rédhibitoire dans l'optique de la transformation de modèles à l'aide de graphes : dans [68], l'auteur montre que sous certaines conditions, le cout des calculs reste tout à fait acceptable : en particulier, il faut limiter la taille des parties gauches des règles et ne chercher à appliquer à un moment donné qu'un nombre restreint de règles. Dans la plupart des exemples de transformations de modèles, ces condtions sont remplies.

# Chapitre III

# Ajouter de l'information sur les graphes : les systèmes adhésifs HLR

En UML, et plus généralement dans le cadre de travail défini par le MDA, les graphes représentant les modèles comportent presque tout le temps des attributs (par exemple pour préciser le nom des classes ainsi que les méthodes associées). Les transformations de modèles doivent prendre en compte ces attributs afin, d'une part, de ne pas perdre d'information lors de l'application de la transformation, mais aussi d'autre part, de pouvoir faire des calculs plus ou moins complexes avec ces attributs. Le problème est donc maintenant de trouver un formalisme adapté pour ajouter cette information dans la catégorie des graphes qui permettent lors de la construction du double pushout ce travail avec les attributs.

Dans ce but, les systèmes adhésifs HLR ont été introduits par Hartmut Ehrig et ses collègues [32] dans l'optique de fusionner l'approche catégorique par double pushout pour les transformations de graphes et le calcul sur les attributs. D'un point de vue théorique, cette approche s'appuie sur une généralisation [26] des catégories adhésives introduites par Lack et Sobocinski en 2004 [48]. Le but de ces catégories est de définir le cadre de travail minimum pour obtenir les bonnes propriétés pour un système de transformations basé sur le pushout. Plusieurs raffinements de cette approche existe, on pourra par exemple citer [47] dans lequel on ne considère plus que les opérations unaires dans les signatures algébriques.

**Remarque 6.** Dans la suite de la thèse, le concept de typage va apparaître avec deux sens différents : le premier est la traduction sur les graphes de la relation entre un modèle et son méta-modèle ou entre un méta-modèle et son méta-méta-modèle comme définie dans le cadre du MDA ; le seconde utilisation du mot typage sera faite dans le cadre de la théorie des types (et plus particulièrement des types inductifs) pour décrire les attributs : par exemple, si un attribut est défini comme un entier naturel, comme un caractère alphanumérique, *etc.* Il n'y a *a priori* aucun risque de confusion entre ces deux concepts et on utilisera ainsi le terme type ou un de ses dérivés sans plus de précision.

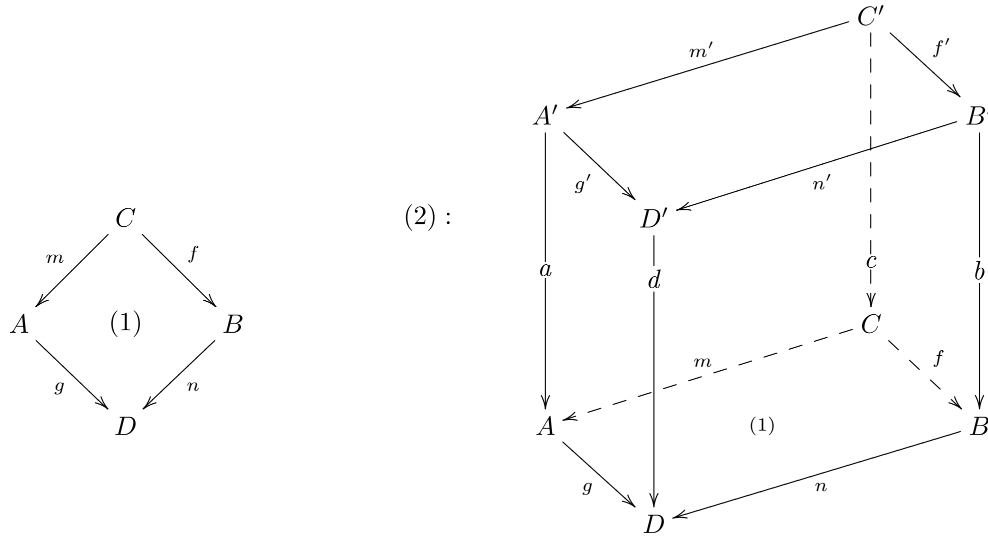
On commence ici par rappeler les définitions et quelques propriétés d'une catégorie adhésive. La suite du chapitre est consacré à la construction d'une catégorie de graphes attribués et typés satisfaisant à ces conditions.

## 1 Les catégories adhésives

### 1.1 Carré de Van Kampen

**Définition 19.** *Un pushout* (1) *est un carré de* Van Kampen *si pour chaque cube commutatif* (2)*, construit avec le pushout* (1) *pour face inférieure et des pullbacks pour les faces arrière, la propriété suivante est vérifiée :*

*le haut du cube est un pushout* $\iff$ *les faces avant sont des pullbacks.*




Si le morphisme $m$ est un monomorphisme (voir annexe 2), ce carré sera appelé pushout le long d'un monomorphisme. Dans ce cas, $n$ aussi sera un monomorphisme et le carré sera de plus un pullback.

**Définition 20.** *Une catégorie* $\mathbf{C}$ *sera dite adhésive si :*

1. $\mathbf{C}$ *a des pushouts le long de tous les monomorphismes ;*
2. $\mathbf{C}$ *a des pullbacks ;*
3. *les pushouts le long des monomorphismes sont des carrés VK.*

## 1.2 Catégorie et système adhésifs HLR

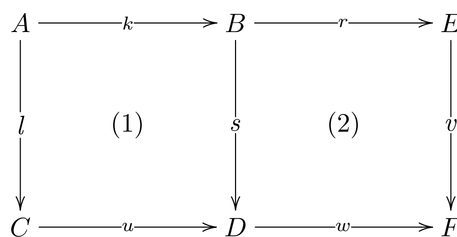
La principale différence entre les catégories adhésives et les catégories adhésives "high level replacement" (HLR) vient du fait qu'au lieu de considérer les pushouts le long de monomorphismes, on va étudier ces pushouts le long d'une classe particulière de monomorphismes.

**Définition 21.** *Une catégorie* $\mathbf{C}$ *avec une classe* $\mathcal{M}$ *de morphismes sera dite adhésive HLR si*

1. $\mathcal{M}$ *est une sous-classe de monomorphismes fermée pour les isomorphismes, fermée par composition (i.e. si* $f \in \mathcal{M}$ *et* $g \in \mathcal{M}$ *alors* $g \circ f \in \mathcal{M}$*) et par décomposition (i.e. si* $g \circ f \in \mathcal{M}$ *et* $g \in \mathcal{M}$ *alors* $f \in \mathcal{M}$*) ;*
2. $\mathbf{C}$ *a des pushouts le long de tous les* $\mathcal{M}$*-morphismes et* $\mathcal{M}$ *est stable par pullbacks et pushouts (c'est-à-dire que si un des deux morphismes de départ est dans la classe* $\mathcal{M}$*, alors le morphisme opposé dans le carré du pushout sera lui aussi dans la classe* $\mathcal{M}$*) ;*
3. *les pushouts dans* $\mathbf{C}$ *le long de* $\mathcal{M}$*-morphismes sont des carrés VK.*

**Théorème 1.** *Soit* $(\mathbf{C}, \mathcal{M})$ *une catégorie adhésive HLR. Alors,*

1. *les pushouts le long des* $\mathcal{M}$*-morphismes sont des pullbacks ;*
2. *Si on se donne le diagramme suivant avec* $l$*,* $w$ *appartenant à* $\mathcal{M}$ *ainsi que* $(1) + (2)$ *étant un pushout et* $(2)$ *un pullback.*




*Alors* $(1)$ *et* $(2)$ *sont des pushouts.*

3. *On se donne un cube commutatif où tous les morphismes de la face supérieure et de la face inférieure sont dans* $\mathcal{M}$*, où la face supérieure est un pullback et les faces de devant sont des pushouts. On a alors : le bas est un pullback* $\iff$ *les faces arrière sont des pushouts ;*
4. *si on se donne* $k : A \to B$ *et* $s : B \to D$*, alors il existe au plus un objet (à isomorphisme près)* $C$ *avec* $l : A \to C$ *et* $u : C \to D$ *tel que le diagramme* (1) *soit un pushout.*

**Définition 22.** *Un système adhésif HLR* $(\mathbf{C}, \mathcal{M}, S, P)$ *est constitué d'une catégorie adhésive HLR* $(\mathbf{C}, \mathcal{M})$*, un objet de départ* $S$ *ainsi qu'un ensemble de productions* $P$*, où une production* $p = L \xleftarrow{l} K \xrightarrow{r} R$ *est donnée par trois objets* $L$*,* $K$ *et* $R$ *et des morphismes* $l$ *et* $r$ *appartenant à* $\mathcal{M}$*.*

*Une transformation directe* $G \overset{p,m}{\Longrightarrow} H$ *via une production* $p$ *et un morphisme* $m : L \to G$*, appelé correspondance, est donné par le DPO-diagramme suivant, où* (1) *et* (2) *sont des pushouts :*

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ \downarrow m & (1) & \downarrow d & (2) & \downarrow m^* \\ G & \xleftarrow{l^*} & D & \xrightarrow{r^*} & H \end{array}$$

**Remarque 7.** Pour construire le transformé d'un graphe $G$ suivant une règle $p$ dans le formalisme adhésif HLR, la première étape est donc comme auparavant le calcul du pushout-complement. Et comme auparavant, une condition de collage (ou « *gluing condition* ») apparaît ici afin de ne pas avoir d'incohérence dans le graphe final (voir paragraphe 2.1 du chapitre II).

## 1.3 Quelques propriétés des systèmes adhésifs HLR

Les système adhésifs HLR vérifient les propriétés suivantes :

#### *1.3.8 Indépendance et* CHURCH-ROSSER *local*

**Remarque 8.** Dans la littérature sur la récriture de graphes [25, 20], le terme « CHURCH-ROSSER local » est employé pour signifier la possiblité de permuter l'application de deux règles, alors que, plus généralement, dans la théorie de la récriture, ce même terme recouvre une propriété plus faible qui affirme que si on applique deux règles différentes au même objet, alors il va exister deux dérivations (de longueur *a priori* inconnue) qui vous nous ramener au même objet.

**Définition 23.** *Soit deux productions* $p_1 = L_1 \leftarrow K_1 \to R_1$ *et* $p_2 = L_2 \leftarrow K_2 \to R_2$*. Deux dérivations directes* $G \overset{p,m}{\Longrightarrow} H$ *et* $G \overset{p',m'}{\Longrightarrow} H'$ *seront dites parallèlement indépendantes s'il existe des morphismes* $k_1 : L_2 \to D_1$ *et* $k_2 : L_1 \to D_2$ *tels que* $l_2^* \circ k_2 = m_1$ *et* $l_1^* \circ k_1 = m_2$ *(cf. figure III.1).*

**Définition 24.** *Soit deux productions* $p_1 = L_1 \leftarrow K_1 \to R_1$ *et* $p_2 = L_2 \leftarrow K_2 \to R_2$*. Une dérivation séquentielle* $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2}{\Longrightarrow} H_2$ *sera dite séquentiellement indépendante s'il existe des morphismes* $k_2 : R_1 \to D_2$ *et* $k_1 : L_2 \to D_1$ *tels que* $l_2^* \circ k_2 = m_1^*$ *et* $r_1^* \circ k_1 = m_2$ *(cf. figure III.2).*

Intuitivement, les conditions imposées dans ces deux définitions de l'indépendance s'assurent que aucune des deux productions considérées ne va consommer un élément essentiel pour l'application de la seconde production. Le théorème suivant nous affirme qu'en réalité, ces deux définitions sont équivalentes.

**Théorème 2.** *Soit* $G \overset{p_1,m_1}{\Longrightarrow} H_1$ *et* $G \overset{p_2,m_2}{\Longrightarrow} H_2$ *deux dérivations directes parallèlement indépendantes. On pose* $m_2' = r_1^* \circ k_1$*. Alors les morphismes* $l_2$ *et* $m_2'$ *satisfont la condition de collage. On peut donc appliquer la production* $p_2$ *avec la correspondance* $m_2'$*. De plus, la dérivation* $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2'}{\Longrightarrow} H_2$ *est séquentiellement indépendante.*

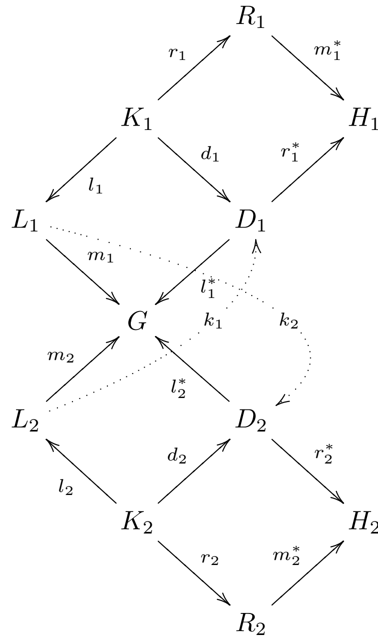



FIG. III.1 – Indépendance parallèle

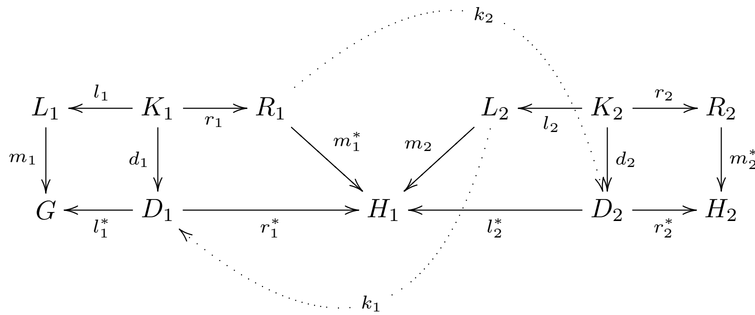



FIG. III.2 – Indépendance séquentielle

*Soit* $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2}{\Longrightarrow} H_2$ *une dérivation séquentiellement indépendante. On pose* $m_2' = l_1^* \circ k_1$. *Alors les morphismes* $l_2$ *et* $m_2'$ *satisfont la condition. On peut donc appliquer la production* $p_2$ *avec la correspondance* $m_2'$. *De plus, les dérivations directes* $G \overset{p_1,m_1}{\Longrightarrow} H_1$ *et* $G \overset{p_2,m_2'}{\Longrightarrow} H_2$ *sont parallèlement indépendantes.*

*1.3.9 Parallélisme*

Pour formuler le parallélisme, on doit supposer que la catégorie $(\mathbf{C}, \mathcal{M})$ possède un coproduit binaire (que l'on notera $+$) qui soit compatible avec $\mathcal{M}$, *i.e.* $m_1$ et $m_2$ appartenant à $\mathcal{M}$, on a $m_1 + m_2$ qui fait aussi partie de la classe de monomorphismes $\mathcal{M}$. Dans la catégorie des ensembles, cela revient à prendre l'union disjointe.

**Théorème 3.** *On se donne deux productions* $p_1 : (L_1 \leftarrow K_1 \rightarrow R_1)$ *et* $p_2 : (L_2 \leftarrow K_2 \rightarrow R_2)$. *Alors les deux assertions suivantes sont équivalentes :*

- *il existe une dérivation parallèle* $G \overset{p_1+p_2,m}{\Longrightarrow} X$ *;*
- *il existe une dérivation séquentiellement indépendante* $G \overset{p_1,m}{\Longrightarrow} H_1 \overset{p_2,m'}{\Longrightarrow} X$ *.*

En conclusion, on s'aperçoit donc ici de la richesse théorique de ce cadre de travail car les principales propriétés pour l'étude d'une grammaire de graphes sont ici vérifiées *a priori*. La suite de cette approche est donc de trouver une catégorie adéquate pour les graphes attribués vérifiant les conditions d'une catégorie adhésive HLR. Bien entendu, les transformations dans cette catégorie doivent permettre de véritablement travailler avec les attributs (les copier, faire des calculs avec...).

# 2 Application aux graphes attribués typés

L'idée directrice emprunté par Ehrig et ses collègues pour la récriture de graphes attribués est d'ajouter à une structure de graphes classique une signature algébrique pour traîter les attributs et les calculs avec ceux-ci. Cela conduit donc à une solution hybride utilisant deux formalismes algébriques différents pour construire le système de récriture.

## 2.1 Signatures et algèbres

On rappelle ici quelques définitions se rapportant aux signatures algébriques et aux $\Sigma$-algrèbres (*cf.* [20, 30]).

**Définition 25.** *Une signature* $SIG = (S, OP)$ *est la donnée*

- *d'un ensemble* $S$*, l'ensemble des sortes ;*
- *d'un ensemble* $OP$*, l'ensemble des symboles de constante et des symboles d'opération.*

*L'ensemble* $OP$ *est l'union disjointe des ensembles suivants :*

- $(K_s)_{s \in S}$*, l'ensemble des symboles de constantes de sorte* $s \in S$ *;*
- $(OP_{w,s})_{w \in S^+, s \in S}$*, l'ensemble des symboles d'opération de sorte d'argument* $w \in S^+$ *et de sorte d'arrivée* $s \in S$.

**Définition 26.** *Une algèbre* $A = (S_A, OP_A)$ *de signature* $SIG = (S, OP)$*, appelé* $SIG$*-algèbre, est donnée par deux familles* $S_A = (A_s)_{s \in S}$ *et* $OP_A = (N_A)_{N \in OP}$ *telles que :*

- $A_s$ *sont des ensembles pour tout* $s \in S$*, appelés ensembles de base ou domaines de* $A$ *;*
- *Si* $N$ *est un symbole de constante appartenant à* $K_s$*, alors* $N_A$ *est un élément de* $A_s$*, appelé constante de* $A$ *;*
- *Si* $N$ *est un symbole d'opération appartenant à* $OP_{s_1 \ldots s_n, s}$*, alors* $N_A$ *est un fonction de* $A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$*, appelé opération de* $A$.

**Définition 27.** *Un morphisme de* $SIG$*-algèbres* $f : A \rightarrow A'$ *est la donnée, pour tout* $s \in S$*, de fonctions* $f_s : A_s \rightarrow A'_s$ *qui commutent avec les opérations.*

**Définition 28.** *Soit* $SIG = (S, OP)$ *une signature. On définit l'algèbre finale* $Z$ *de signature* $SIG$ *par*

$$Z_s = s, \text{ pour tout } s \in S.$$

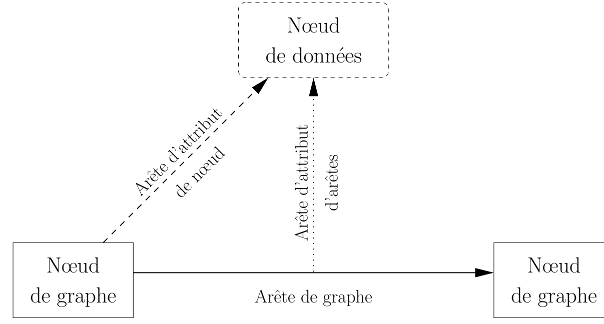



FIG. III.3 – Représentation schématique d'un $E$-graphe

**Remarque 9.** Si l'on se place dans la catégorie des $SIG$-algèbres, l'algèbre finale de signature $SIG$ peut être vu comme l'objet terminal de la catégorie.

**Définition 29.** *Soit $SIG = (S, OP)$ une signature. On se donne un ensemble $X_s$ pour chaque $s \in S$, cet ensemble est appelé l'ensemble des variables de sorte $s$. On suppose que les $X_s$ sont deux à deux disjoints, ainsi qu'ils sont disjoints avec $OP$. L'union $X = \bigcup_{s \in S} x_s$ est appelé l'ensemble des variables par rapport à $SIG$.*

*L'ensemble des termes de sortes $s$, noté $T_{OP,s}$ est inductivement défini par :*

1. *termes basiques : $X_s \cup K_s \subset T_{OP,s}$ ;*
2. *termes composés : $N(t_1, \ldots, t_n) \in T_{OP,s}$ pour tout symbole d'opération $N : s_1 \ldots s_n \to s$ et tout terme $t_1 \in T_{OP,s_1}$, ..., $t_n \in T_{OP,s_n}$ ;*
3. *il n'y a pas d'autre terme dans $T_{OP,s}$.*

*On pose enfin $T_{OP} = \bigcup_{s \in S} T_{OP,s}$.*

## 2.2 Graphes attribués sur les sommets et sur les arcs. Typage

**Définition 30.** *Un E-graphe $G = (V_1, V_2, E_1, E_2, E_3, (source_i, target_i)_{i \in [\![1,3]\!]})$ est constitué*
- *d'ensembles*
  - *$V_1$ et $V_2$ appelés respectivement nœuds de graphe et nœuds de données,*
  - *$E_1$, $E_2$ et $E_3$ appelés respectivement arêtes de graphe, d'attributs de nœuds et d'attributs d'arcs ;*
- *des fonctions source et cible*
  - *$source_1 : E_1 \to V_1$, $source_2 : E_2 \to V_1$ et $source_3 : E_3 \to E_1$,*
  - *$target_1 : E_1 \to V_1$, $target_2 : E_2 \to V_2$ et $target_1 : E_3 \to V_2$.*

*Ces différents éléments sont schématisés dans la figure III.3. Un morphisme de E-graphes $f : G' \to G'$ est la donnée de 5 fonctions $(f_{V_1}, f_{V_2}, f_{E_1}, f_{E_2}, f_{E_3})$ avec*
- *$f_{V_i} : V_i \to V_i'$ pour $i \in [\![1, 2]\!]$;*
- *$f_{E_j} : E_j \to E_j'$ pour $j \in [\![1, 3]\!]$,*

*telle que $f$ commute avec toutes les fonctions source et cible.*

*L'ensemble des E-graphes muni des morphismes de E-graphes forme la catégorie **EGraph**.*

Ces $E$-graphes vont servir de support pour les graphes attribués que l'on cherche à définir. Il reste donc maintenant à ajouter les signatures algébriques pour les attributs :

**Définition 31.** *On considère une signature de données $DSIG = (S_D, OP_D)$ avec des sortes de valeurs d'attributs $S_D' \subset S_D$. Un graphe attribué $(G, D)$ est la donnée d'un E-graphe $G$ ainsi que d'une DSIG-algèbre $D$ telle que $\bigcup_{s \in S_D'} D_s = V_2$. Un morphisme de graphes attribués est une paire $f = (f_G, f_D)$ constituée d'un morphisme de E-graphes $f_G$ et d'un homomorphisme d'algèbre $f_A$*

*telle que le diagramme suivant soit un pullback pour tout s appartenant à* $S'_D$.

$$\begin{array}{ccc} D_s & \xrightarrow{f_{D,s}} & D'_s \\ \downarrow & & \downarrow \\ V_2 & \xrightarrow{f_{G,V_2}} & V'_2 \end{array}$$

*Les graphes attribués munis de ces morphismes forment la catégorie* ***AGraphs****.*

**Remarque 10.** La possibilité de choisir $S'_D$ strictement plus petit que $S_D$ permet de limiter les sortes utiles pour le graphe au minimum et ainsi de laisser de côté les sortes qui ont servi à en construire de plus compliqués mais qui n'ont plus d'utilité dans le graphe [43].

Par ailleurs, malgré cette possibilité, il faut bien remarquer que cette définition entraine que pour un graphe attribué $G$, l'ensemble $V_2$ contient un élément pour chaque valeur possible de la sorte. Ainsi, théoriquement, $V_2$ est un ensemble inifini dès que l'on travaille avec une sorte infinie.

**Définition 32.** *Un graphe-type attribué est un graphe attribué* $(G_0, Z)$ *où* $Z$ *est la* $DSIG$*-algèbre finale.*

*Un graphe typé attribué* $((G, D), t)$ *au-dessus de* $G_0$ *est la donnée d'un graphe attribué* $(G, D)$ *ainsi que d'un morphisme de graphes attribués* $t : G \to G_0$.

*Un morphisme de graphes typés attribués* $f : G \to G'$*, où* $G$ *et* $G'$ *sont deux graphes typés par le même graphe-type* $(G_0, Z)$ *est un morphisme de graphes attribués vérifiant la condition supplémentaire* $t' \circ f = t$.

*Les graphes typés attribués au-dessus de* $ATG = (G_0, Z)$ *munis des morphismes décrits ci-dessus forment la catégorie* $\textbf{\textit{AGraphs}}_{ATG}$.

*Exemple.* On considère les sortess classiques `Caract`, `Chaine`, `Nat`. On definit la signature de données $DSIG$ par

$$DSIG = \texttt{Caract} + \texttt{Chaine} + \texttt{Nat} + \quad \begin{array}{l} \textit{sortes} : \texttt{ParametreDirection} \\ \textit{opns} : \textit{in}, \textit{out}, \textit{inout}, \textit{return} :\to \texttt{ParametreDirection}. \end{array}$$

L'ensemble des sortes de données utilisés est $S'_D = \texttt{Chaine} + \texttt{Nat} + \texttt{ParametreDirection}$. La figure III.4 montre un graphe-type attribué $(G_0, Z)$, où $Z$ est l'algèbre finale associée à la signature $DSIG$. Dans la figure III.5, on donne un exemple de graphe typé attribué $G$ au-dessus de $G_0$. Il est important de remarquer que tous les attributs ne sont représentés pas ici : on ne dessine que ceux qui sont "rattachés" au graphe c'est-à-dire ceux qui nous apportent une information. Par exemple, on devrait voir apparaître ici une case *inout* ; elle n'est pas repésentée car *inout* n'est l'attribut d'aucun élémént du graphe. Le graphe $G$ est typé au-dessus de $G_0$ par le morphisme $t$ défini par :

- $t(m) = Methode$
- $t(par_1) = t(par_2) = t(par_3) = Parametre$
- $t(c) = Class$
- $t(1) = t(2) = t(3) = \texttt{Nat}$
- $t(return) = t(in) = \texttt{ParametreDirection}$
- $t(p_1) = t(p_2) = t(add) = t(Nat) = \texttt{Chaine}$

## 2.3 Transformations des graphes typés attribués

**Définition 33.** *La classe de monomorphismes distingués* $\mathcal{M}$ *pour la catégorie* $\textbf{\textit{AGraph}}_{ATG}$ *est definie par* $f = (f_G, f_A) \in \mathcal{M}$ *si* $f_G$ *est injective et* $f_D$ *est un isomorphisme.*

**Théorème 4.** *La catégorie* $(\mathbf{AGraph_{ATG}}, \mathcal{M})$ *des graphes typés attribués au-dessus de* $ATG$ *est une catégorie adhésive HLR.*

**Définition 34.** *On définit une production* $p = L \xleftarrow{l} K \xrightarrow{r} R$ *en se donnant :*

- *trois graphes typés attribués* $L$*,* $K$ *et* $R$ *attribués au-dessus de l'algèbre de termes* $T_{DSIG}$*. Ces trois graphes sont respectivement appelés la partie gauche, l'interface et la partie droite de la production ;*

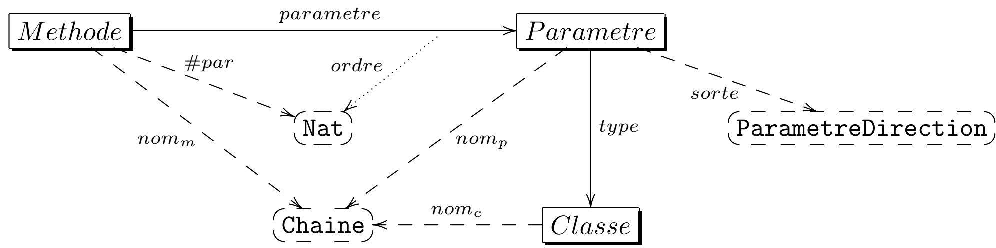



Fig. III.4 – Graphe-Type attribué

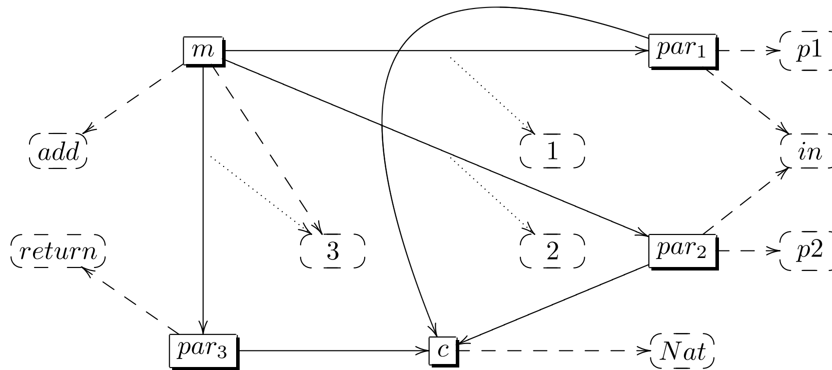



Fig. III.5 – Graphe typé attribué

– *deux morphismes l et r appartenant à la classe* $\mathcal{M}$.

Pour obtenir des transformations de graphes typés attribués, on applique maintenant la construction habituelle avec le double pushout : si on se donne une production $p = L \xleftarrow{l} K \xrightarrow{r} R$ et une correspondance $m : L \to G$, on commence donc par calculer, s'il existe, le pushout-complement $D$ et on construit ensuite le pushout $H$. La figure III.6 donne un exemple simple d'une telle transformation appliqué à un compteur.

Le fait que l'on trouve une structure de catégorie adhésive HLR permet donc d'appliquer tous les résultats propres à ces constructions algébriques. En particulier, pour les transformations de graphes typés attribués, on retrouve bien les propriétés de Church-Rosser local et de parallélisme.

# 3 Discussion

L'approche présentée ici permet donc d'ajouter des attributs sur les graphes et de faire des calculs avec ces attributs. Pour autant, sur certains aspects, cette théorie est perfectible. En effet, la manière de construire les graphes attribués engendre des définitions hétérogènes mélangeant les signatures algébriques et les ensembles. Ce cadre théorique engendre alors des graphes théoriquement infinis dès que l'on travaille avec une $\Sigma$-algèbre infinie, c'est-à-dire où une sorte a une infinité d'habitants. C'est un inconvénient majeur en vue de l'implémentation sur machine du système : la gestion d'ensembles infinis sur machine n'est pas quelque chose de facile ; il faut user de techniques telles que l'évaluation paresseuse par exemple. À ce propos, dans le logiciel AGG [72], qui est l'implémentation de ce système, le traitement des attributs ne suit pas exactement l'approche théorique et est délégué à un langage extérieur (Java en l'occurrence). Par ailleurs, en utilisant les $\Sigma$-algèbres finales pour définir le graphe-type d'un graphe, on limite la possibilité de typage à un «étage» et il est alors impossible de traduire dans ce formalisme la notion de méta-méta-modèle introduite dans le MDA.

Le but de notre contribution sera donc de proposer un nouveau système de transformation de graphes attribués et typés permettant de dépasser les problèmes soulevés par les systèmes adhésifs

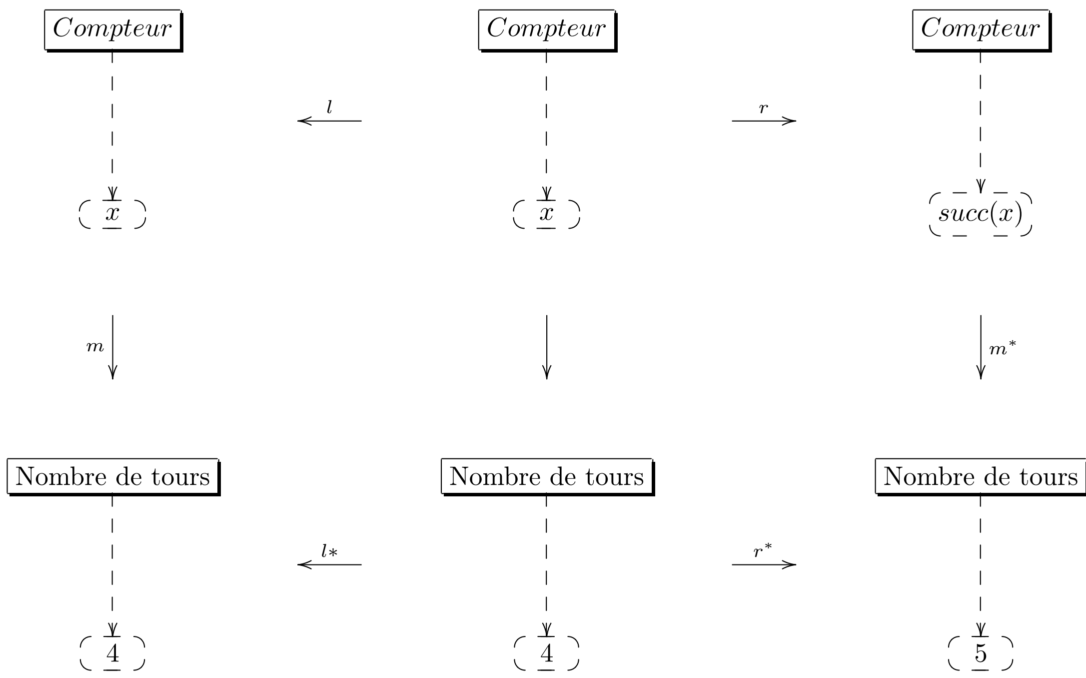
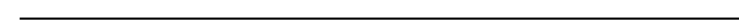



Fig. III.6 – Transformation de graphes typés attribués

HLR.

## Deuxième partie

# L'approche «Double Pushout Pullback»



On présente maintenant notre contribution pour la transformtion des graphes attribués et « typés » s'appuyant principalement sur le cadre théorique introduit par l'approche « Double Pushout ».

# Chapitre IV

# Les catégories AttGraph et $\mathbf{AttGraph}_{\mathbf{TG}}$

## 1 Définition des graphes

Dans l'approche DPoPB, une des idées directrices des recherches était de trouver un formalisme unique pour décrire la structure des graphes ainsi que les attributs qui les accompagnent. Ce formalisme doit de plus permettre, lors des transformations de graphes, d'effectuer des calculs avec les attributs sans pour autant quitter le cadre théorique des catégories (et les pushouts).

La solution trouvée utilise les types inductifs (*cf.* [51, 11]), à la fois pour décrire la structure des graphes et pour définir les attributs attachés à cette structure. Ce choix a été fait pour plusieurs raisons. La première est de tirer partie de la puissance de la théorie des types dans le but de prouver des propositions formelles sur les transformations de graphes. La seconde raison est que les types inductifs permettent de construire des «espaces d'attributs» plus généraux que ce qu'il est possible de faire avec par exemple les Σ-algèbres car les constructeurs des types inductifs peuvent recevoir des arguments fonctionnels (par exemple, utiliser les arbres ou les ordinaux pour construire de nouveaux types) ; cette caractéristique nous permet d'obtenir une plus grande expressivité pour les calculs que l'on pourra appliquer sur les attributs. Par ailleurs, ce choix permet de résoudre le problème du calcul des attributs sans faire apparaître de structure infinie. Enfin, la dernière raison concerne la mise en œuvre du système : certains assistants à la preuve tels que Coq sont très efficace pour manipuler les types inductifs ; de plus, il existe des bibliothèques pour travailler avec les catégories (*cf.* [18, 69]).

### 1.1 La structure des graphes

On introduit le type spécial *TypeFini* qui représente les types inductifs finis et qui est défini par :

$$\mathtt{Ind}()(\mathtt{F_n} : \mathtt{Set} := \mathtt{c_1} : \mathtt{F_n}, ..., \mathtt{c_n} : \mathtt{F_n}).$$

On utilise ici les notations du logiciel Coq [18] pour décrire les types inductifs.

Dans le cadre théorique des types inductifs avec la notion de types dépendants, un graphe (simple) peut être défini par la fonction suivante :

$$A^G : S^G \to S^G \to TypeFini$$

où $S^G$ : *TypeFini* représente l'ensemble des nœuds du graphe $G$. Le nombre de constructeurs dans $S$ donne le nombre de sommets présents dans le graphe $G$ et pour chaque couple $(s_1, s_2)$ de sommets, le type inductif $G(s_1, s_2)$ représente de la même manière des arcs orientés entre $s_1$ et $s_2$. Une telle représentation permet donc d'avoir plusieurs arcs avec la même orientation entre deux sommets.

**Remarque 11.** On peut définir les fonctions *src* et *tgt* qui associent respectivement à chaque arc du graphe son origine et sa cible.

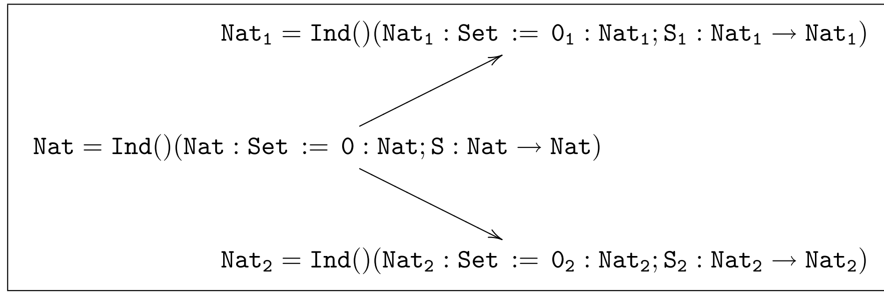



Fig. IV.1 – Copies du type inductif `Nat`

## 1.2 Les attributs

Afin de ne pas se perdre dans des détails techniques, on ne considère dans ce chapitre que les attributs attachés aux nœuds. L'extension aux attributs attachés aux arcs se fait sans aucune difficulté supplémentaire.

D'un point de vue conceptuel, pour attacher de l'information aux sommets, on procède en deux étapes : lors d'une première phase, on indique quels sont les types des attributs (nommé «espaces d'attributs») que l'on veut lier à un nœud. Ensuite, on donne, pour chacun de ces types, la valeur précise de l'attribut ou un joker pour garder la place libre sans donner de valeur.

Durant la première étape, on veut pouvoir placer sur un même nœud plusieurs espaces d'attributs (ce nombre pouvant être nul). Dans [40], une fonction partielle entre les sommets du graphe et les $\Sigma$-algèbres est introduite. Celle-ci permet en effet de choisir si un attribut est accroché ou non à un sommet, mais on ne peut avoir plusieurs attributs par sommet. Afin de généraliser cette approche, on remplace alors la fonction partielle par une relation entre les sommets et les espaces d'attributs. Un autre impératif du système est de pouvoir placer plusieurs attributs du même type sur un même sommet et bien entendu de pouvoir distinguer entre eux ces attributs. À cette fin, on utilise la notion de copie de type inductif (*cf.* [12]).

**Définition 35.** *Soit $T$ et $T'$ deux types inductifs. $T'$ sera dit copie conforme de $T$ si $T'$ diffère de $T$ seulement par les noms de ces constructeurs.*

Si un nœud possède deux entiers naturels comme attributs, il sera alors en relation avec deux copies de :

$$\mathtt{Nat} = \mathtt{Ind}()(\mathtt{Nat} : \mathtt{Set} := \mathtt{0} : \mathtt{Nat}; \mathtt{S} : \mathtt{Nat} \rightarrow \mathtt{Nat}),$$

le type inductif représentant $\mathbb{N}$ : on utilisera alors par exemple les noms $\mathtt{Nat}_1$ et $\mathtt{Nat}_2$ (voir figure IV.1). Les deux attributs seront alors bien distincts.

On arrive donc à la définition suivante :

**Définition 36.** *Une relation d'étiquetage est une relation entre les sommets $S^G$ du graphe $G$ et les copies des espaces d'attributs. Cette relation est notée $\mathcal{R}^G$ et pour $s \in S^G$, l'ensemble $\mathcal{R}^G_s$ est l'ensemble des espaces d'attributs en relation avec le sommet $s$.*

On se donne maintenant un graphe $G$ ainsi qu'une relation d'étiquetage $\mathcal{R}^G$ associée. On veut alors définir les valeurs des attributs. Il va donc falloir choisir pour chaque nœud et pour chaque espace d'attributs en relation avec ce nœud un élément de l'espace d'attributs.

Par ailleurs, on veut offrir une alternative à cette sélection : afin d'avoir un élément qui puisse nous servir de variable ou d'inconnue dans le but d'introduire des calculs entre les attributs dans les transformations de graphes attribués, on autorise la possibilité que l'attribut ne soit pas explicitement défini (on ne connaît alors que son type). Cet attribut, nommé joker, sera alors noté ❀. On arrive alors à la définition suivante :

**Définition 37.** *Une fonction d'attribution est la donnée d'une fonction de type*

$$Attrib^G : \left(\Pi\, s : S^G . \left(\Pi\, \mathtt{EA} : \mathcal{R}^G_s . (\mathtt{EA} \cup ❀)\right)\right)$$

*où la notation $\Pi$ désigne le produit dépendant (cf. [51, 6]).*

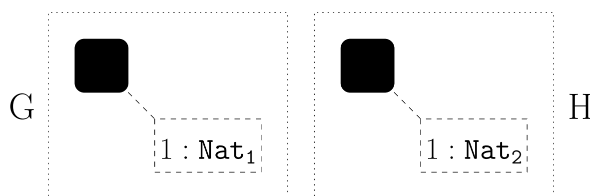



FIG. IV.2 – Deux graphes de même nature

Pour résumer :

**Définition 38.** *Un graphe attribué $G$ sera donc donné par un type fini $S^G$ et une fonction $A^G$ représentant à eux deux la structure du graphe, ainsi qu'une relation d'étiquetage $\mathcal{R}^G$ et une fonction d'attribution $Attrib^G$ pour les attributs.*

On peut comprendre ici une des différences majeures entre cette approche pour définir les graphes attribués et l'approche développée dans [32]. Dans l'approche HLR pour les graphes attribués, en utilisant les $\Sigma$-algèbres, les attributs sont introduits en rajoutant des sommets spéciaux aux graphes. Pour attacher un attribut à un nœud, un arc est alors créé entre les deux sommets. En suivant cette approche, il est nécessaire de créer dans le graphe un nœud pour chaque valeur possible de l'attribut, c'est-à-dire qu'il va y avoir autant de sommets spéciaux qu'il y a d'élément dans la $\Sigma$-algèbre associée aux graphes. Dès que l'on travaille avec une sorte infinie, le graphe sort lui aussi du domaine fini. Si cette solution est théoriquement acceptable, elle reste très difficile à mettre en œuvre. Pour cette raison, le logiciel AGG [72] sort du cadre théorique et utilise Java pour implémenter l'attribution des graphes ainsi que le calcul sur les attributs. Avec les définitions introduites ici, même si le type inductif utilisé pour attribuer un sommet est infini, le graphe reste fini.

## 1.3 Relation d'équivalence

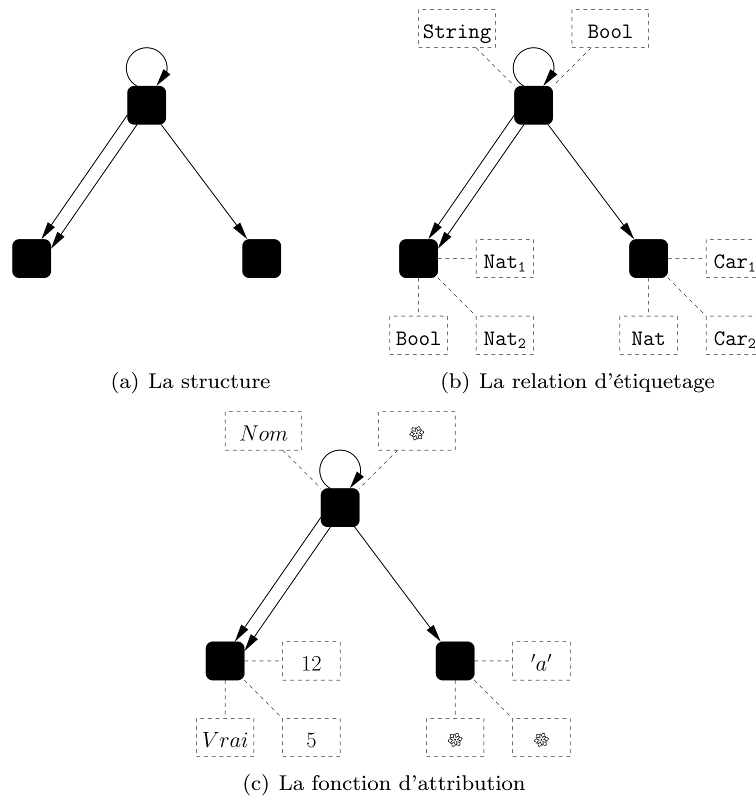
La définition que l'on vient de présenter pour les graphes ne permet pas d'identifier des graphes qui semblent *a priori* identiques. En effet, si on regarde les deux graphes dessinés figure IV.2, il est impossible de les identifier car ce n'est pas le même espace d'attributs attaché à chacun des sommets. Pour autant, intuitivement, ces deux graphes sont identiques. Afin de pouvoir dire que ces deux graphes représentent bien la même structure, on introduit une relation d'équivalence. Cette relation sera utile dans la construction du pushout pour le placement des espaces d'attributs (voir chapitre V).

**Définition 39.** *Deux graphes attribués $G$ et $H$ seront dits équivalents si :*
- *ils possèdent la même structure, i.e. $S^G = S^H$ et $A^G = A^H$ ;*
- *pour tout sommet $s \in S^G$, il existe une bijection entre $\mathcal{R}_s^G$ et $\mathcal{R}_s^H$ telle que*
  - *un espace d'attributs et son image par cette bijection soient des copies conformes l'un de l'autre,*
  - *l'attribut associé à un espace d'attribut de $\mathcal{R}_s^G$ soit égal à l'attribut associé à l'image de cet espace par la bijection (même dans le cas de jokers ❀).*

Avec cette définition, les graphes $G$ et $H$ de la figure IV.2 sont équivalents. Par la suite, on confondra un graphe attribué et sa classe d'équivalence pour cette relation.

**Remarque 12.** On peut maintenant toujours trouver un représentant de la classe d'équivalence d'un graphe où tous les espaces d'attributs en relation avec un nœud du graphe possèdent un nom différent.

**Remarque 13.** On peut se rapporter au travail de CHEMOUIL et SOLOVIEV [13] et de MARIE-MAGDELEINE et SOLOVIEV [53] sur l'introduction de règles de réductions afin de traiter ce type d'équivalence.




(a) La structure (b) La relation d'étiquetage

(c) La fonction d'attribution

Fig. IV.3 – Un graphe attribué

Du point de vue de la complexité, vérifier que deux graphes $G$ et $H$ sont équivalents n'est pas très gourmand. En effet, pour cela, on commence par vérifier que $S^G = S^H$ et $A^G = A^H$ (pour cette seconde éqalité, il est nécessaire de vérifier $n^2$ où $n$ est le nombre de sommets). Après cette étape, il reste à vérifier les correspondances entre les espaces d'attributs et les attributs des deux graphes. Une manière simple de faire cela est de construire pour chaque sommet deux tableaux à double entrée (un correspondant au graphe $G$, l'autre au graphe $H$) ; la première entrée étant les valeurs des attributs présents sur le nœud et la seconde les espaces d'attributs présents. Une croix est alors placée à chaque intersection pour laquelle l'attribut correspond à l'espace d'attributs :

| | `Nat` | $\texttt{Nat}_1$ | $\texttt{Nat}_2$ | `String` | `String'` |
|---|---|---|---|---|---|
| 3 | X | | X | | |
| 5 | | X | | | |
| Hello | | | | X | X |

## 1.4 Exemple

La figure IV.3 montre un exemple simple de construction d'un graphe attribué $G$. Sur le schéma IV.3(a), la première étape de la construction est finie : on a la structure du graphe. Ici, le type fini $S^G$ possède donc trois éléments et les types $A^G(s_1, s_2)$ sont non vides dans seulement trois cas. La figure IV.3(b) montre le résultat de l'application de la relation d'étiquetage : à chacun des sommets sont alors attachés plusieurs espaces d'attributs. Ici, `String` est le type inductif représentant les chaînes de caractères, `Car` et `Bool` sont respectivement les types finis des caractères et des valeurs booléennes. On peut voir que la notion de copie de types inductifs a été utilisé pour plusieurs nœuds. Enfin, sur la figure IV.3(c), les attributs apparaissent véritablement, avec l'utilisation de jokers ❀ dans 3 cas.

## 2 Définition des morphismes de graphes attribués

Soit $G$ et $H$ deux graphes attribués. On définit alors en deux temps la notion de morphisme entre $G$ et $H$. En suivant le schéma de la définition des graphes attribués, on regarde dans un premier temps ce qu'il se passe sur la structure, puis on s'occupe des attributs.

De plus, pour bien décomposer la construction, on ne considère pas au départ la relation d'équivalence introduite dans le paragraphe 1.3. Le paragraphe 2.3 expliquera comment prendre en compte cette identification des graphes.

### 2.1 Sur la structure

Sur la structure, on retrouve les définitions classiques pour un morphisme de graphes. C'est-à-dire une fonction qui va envoyer les sommets du graphe $G$ vers ceux du graphe $H$ et une seconde fonction qui va s'occuper des arêtes avec la condition de compatibilité qui affirme que la source (respectivement la cible) de l'image d'une arête doit valoir l'image de la source (respectivement de la cible) de cette arête. Dans notre approche, pour ne pas perdre d'information durant l'application du morphisme, on impose une condition supplémentaire : on veut que l'image d'un sommet possède au moins les mêmes espaces d'attributs que le sommet de départ. Cela se traduit par la définition suivante :

**Définition 40.** *Un morphisme structurel $f$ entre $G$ et $H$ est donné par un couple $(f_S, f_A)$ avec*
- *$f_S$ est une fonction de $S^G$ vers $S^H$ vérifiant la condition qui affirme que si un espace d'attributs* `EA` *est en relation avec le sommet $s \in S^G$, alors* `EA` *est aussi en relation avec $f_S(s)$ ;*
- *$f_A$ est une fonction du type :*

$$\Big(\Pi\, s_1 : S^G \,.\, \Big(\Pi\, s_2 : S^G \,.\, \big(A^G(s_1, s_2) \rightarrow A^H(f_S(s_1), f_S(s_2))\big)\Big)\Big)$$

**Remarque 14.** Quand il n'y aura pas d'ambiguïté, on abrégera la notation $f_A(s_1, s_2)(e)$ en $f_A(e)$. Le type de la fonction $f_A$ assure automatiquement que la condition imposée entre les arcs et les sommets est vérifiée, c'est-à-dire que pour toute arête $e$ appartenant au graphe $G$, on a bien

$$f_S(src(e)) = src(f_A(e)) \qquad \text{et} \qquad f_S(tgt(e)) = tgt(f_A(e)).$$

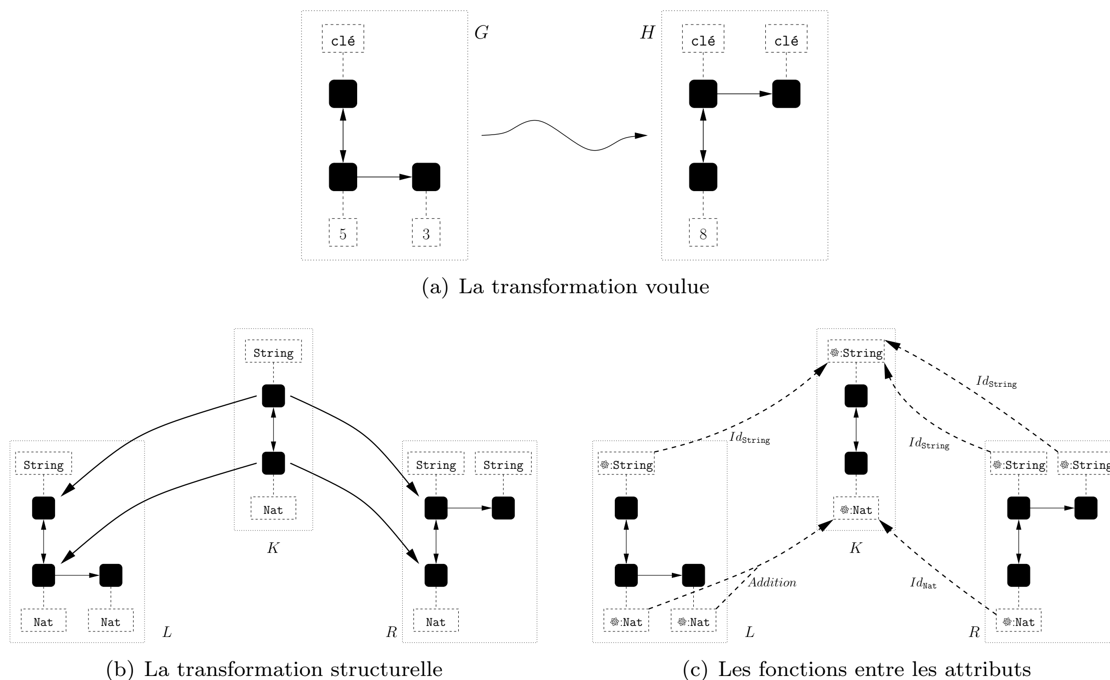
### 2.2 Pour les attributs

C'est dans la manière de traiter les morphismes entre les attributs que réside la principale originalité de notre approche. En effet, durant les transformations de graphes, on veut pouvoir copier les attributs et effectuer avec ceux-ci des calculs plus ou moins complexes (en particulier, la définition de fonctions prenant plusieurs attributs dans le graphe source comme arguments pour obtenir un attribut dans le graphe cible doit être possible). Pour répondre à cette problématique, la solution mise en œuvre dans le morphisme$f = (f_S, f_A)$ est de renverser les flèches entre les attributs par rapport à un morphisme classique. Intuitivement, ce ne sont plus les attributs de $G$ qui vont influencer ceux de $H$, mais les attributs de $H$ qui vont influencer ceux de $G$.

Pour comprendre les raisons qui nous ont amenés à changer cette orientation pour les flèches entre les attributs, étudions un exemple simple. La figure IV.4(a) montre la transformation qui doit être appliquée à un graphe source $G$. Les deux entiers naturels présents sur ce graphe doivent être additionnés et le nœud supportant le nombre 3 supprimé. En parallèle, un nouveau sommet est créé pour accueillir une copie de la chaîne de caractères `clé`.

La partie structurelle de la transformation (*cf.* figure IV.4(b)) est facile à construire : l'interface $K$ de la règle va être constituée des deux sommets et de l'arc bidirectionnel qui sont conservés pendant la transformation. Pour les espaces d'attributs attachés à ces sommets, les possibilités sont là aussi très restreintes : étant donné que les espaces d'attributs présents dans le graphe $K$ doivent se retrouver à l'identique dans les graphes $L$ et $R$ et qu'une information qui ne se trouve pas dans ce graphe d'interface est forcément perdue, il va y avoir deux espaces d'attributs dans le graphe $K$ : `Nat` et `String`.

À partir de cette figure, on peut comprendre pourquoi il est logique d'inverser les flèches pour travailler sur les attributs (voir figure IV.4(c)). Entre le graphe $L$ et le graphe $K$, on perd un




(a) La transformation voulue

(b) La transformation structurelle

(c) Les fonctions entre les attributs

FIG. IV.4 – Le renversement des flèches

attribut portant comme information un nombre entier. L'attribut restant dans le graphe $K$ doit donc déjà porter le résultat de l'addition que l'on souhaite effectuer. Par conséquent, cette addition doit être portée par le morphisme $l$ de la règle de transformation. Il est alors naturel de faire partir cette fonction (à 2 arguments) du graphe $L$ pour se diriger vers le graphe $K$ ; il serait de plus impossible de trouver une fonction qui en partant d'un nombre entier en donne une décomposition en deux entiers, le résultat ne serait alors pas unique.

Le graphe interface $K$ permet donc de stocker temporairement les résultats des calculs lors de la transformation.

Pour les mêmes raisons, les copies d'attributs ont forcément lieu durant l'execution du morphisme $r$ et imposent là aussi de renverser l'orientation des flèches : une flèche dans le sens classique ne permettrait d'envoyer l'attribut qu'à un seul endroit du graphe $R$. Retourner les flèches permet à n'importe quel attribut de $R$ d'aller «chercher» l'information qui l'intéresse où il le souhaite dans le graphe $K$ et ainsi de partager cette information.

Ces considérations qui nous ont conduits à inverser le sens des flèches pour travailler sur les attributs rappelle dans une certaine mesure l'approche pullback pour la transformation de graphes (voir par exemple l'introduction de [46]). Pour cette raison, nous avons baptisé l'approche de transformation présentée dans cette thèse «Double Pushout Pullback» (abrégé en DPoPb).

**Remarque 15.** On peut signaler qu'une idée similaire mélangeant les concepts du pushout et du pullback a déjà été introduite dans le domaine des spécifications algébriques par Pavlovic et Smith [58].

Cette originalité va nécessiter de redéfinir une nouvelle catégorie de graphes attribués **AttGraph** et toutes les opérations associées. On donne maintenant les définitions précises pour la construction de la partie des morphismes traitant des attributs. On suppose maintenant qu'un morphisme structurel $f = (f_S, f_A)$ est donné entre deux graphes attribués $G$ et $H$.

### *2.2.10 Sur les espaces d'attributs*

La première étape pour formaliser ces morphismes est de donner une relation entre les espaces d'attributs du graphe $H$ et ceux du graphe $G$ pour indiquer quels attributs de $H$ vont «influencer» ceux de $G$. Cette relation, pour être la plus générale possible, doit permettre que deux espaces

d'attributs soient plusieurs fois en relation. Pour cela, on donne la définition d'une relation multivaluée.

**Définition 41.** *Soit $A$ et $B$ deux ensembles. Une relation multivaluée entre $A$ et $B$ est une relation qui permet à un élément $a \in A$ et un élément $b \in B$ d'être plusieurs fois en relation. Formellement, une relation multivaluée entre $A$ et $B$ peut être vu comme un multi-ensemble au dessus du produit cartésien $A \times B$.*

Encore une fois, afin d'être sûr de ne pas perdre de l'information durant l'application du morphisme, on rajoute une condition sur la relation multivaluée que l'on construit. Comme on sait qu'on retrouve tous les espaces d'attributs du sommet $s$ attachés à son image $f_S(s)$, on demande alors que la relation relie les espaces d'attributs identiques entre un sommet et son image. De cette manière, on est sûr que tous les espaces d'attributs du graphe $G$ ont au moins un lien dans le graphe $H$.

**Définition 42.** *Pour chaque morphisme $f : G \to H$, on définit une relation multivaluée entre les types dépendants $\left(\Pi\, s : S^H . \mathcal{R}_s^H\right)$ et $\left(\Pi\, s : S^G . \mathcal{R}_s^G\right)$. Cette relation doit vérifier la condition suivante :*

*pour tout sommet $s \in S^G$ et pour tout espace d'attributs $\mathtt{EA} \in \mathcal{R}_s^G$, alors $(f_s(s), \mathtt{EA})\mathcal{R}^f(s, \mathtt{EA})$.*

*Cette condition définit ce que l'on nomme les « éléments nécessaires » à la relation $\mathcal{R}^f$.*

*Notation.* Pour un nœud $s \in S^G$ et pour un espace d'attributs $\mathtt{EA}$ en relation avec $s$, on note $\mathcal{R}^f_{(s,\mathtt{EA})}$ le (multi-)ensemble des espaces d'attributs en relation avec $(s, \mathtt{EA})$.

On sait maintenant quels attributs du graphe $H$ vont influencer un attribut quelconque du graphe $G$. Si un espace d'attributs $\mathtt{EA}$ de $G$ est en relation avec plusieurs espaces d'attributs de $H$, on peut affiner cette connaissance : en effet, on va pouvoir voir apparaître dans le morphisme soit des fonctions de plusieurs variables pointant vers $\mathtt{EA}$ (par exemple, l'addition décrite figure IV.4(c)) soit plusieurs fonctions d'une variable (utile pour faire des copies d'un attribut). Pour décrire ce mécanisme, on introduit pour chaque espace d'attributs de $G$ une partition de l'ensemble des espaces d'attributs qui sont en relation avec lui.

**Définition 43.** *Un partitionnement associé à la relation $\mathcal{R}^f$ est la donnée pour chaque élément $s$ de $S^G$ et pour chaque espace d'attributs $\mathtt{EA}$ en relation avec $s$ d'une partition de l'ensemble $\mathcal{R}^f_{(s,\mathtt{EA})}$.*

*Un élément de ces partitions va alors être appelé un arbre et la totalité des arbres sera naturellement nommée la forêt associée au morphisme $f$.*

### 2.2.11 Sur les attributs

Il reste maintenant à définir comment se passe l'interaction entre les attributs. Pour cela, on définit pour chaque arbre une fonction entre les espaces d'attributs qui se trouvent aux «feuilles» de l'arbre et l'espace d'attributs à la «racine» de l'arbre.

**Définition 44.** *Pour chaque arbre de la forêt associé au morphisme $f$, on définit une fonction, nommée fonction de calcul, dont le domaine est le produit cartésien des espaces d'attributs du graphe $H$ associés à cet arbre et dont le codomaine est l'espace d'attributs de $G$ associé à cet arbre.*

*Une telle fonction de calcul devra vérifier la condition suivante : l'image des valeurs explicites des attributs du graphe $H$ par cette fonction de calcul devra coïncider avec l'attribut à la racine de l'arbre dans $G$. Dans le cas où des jokers sont utilisés, la condition est affaiblie : une fonction de calcul peut envoyer n'importe quelle valeur sur un joker, mais un joker est nécessairement envoyé sur un joker.*

**Remarque 16.** Quelles fonctions peuvent être acceptables comme fonction de calcul ? Un des intérêts de l'utilisation de la théorie des types inductifs se trouve ici dans la large palette des fonctions utilisables pour définir les morphismes de graphes attribués. En effet, on peut se servir de n'imprte quelle fonction calculable dans le formalisme des types inductifs ; ce qui inclut par exemple les fonctions récursives. Par rapport à l'approche HLR, on augmente ainsi très largement l'expressivité du système car dans l'approche allemande, les calculs sur les attributs sont limités pour une règle de transformation aux termes exprimables dans la $\Sigma$-algèbre (les fonctions récursives sont donc exclues).

Pour résumer :

**Définition 45.** *Un morphisme $f$ entre deux graphes attribués va être donné par sa partie structurelle, une forêt et pour chaque arbre de cette forêt, une fonction de calcul compatible avec les valeurs des attributs.*

### 2.3 Relation d'équivalence

On peut maintenant se demander comment se comporte cette notion de morphisme vis-à-vis de la relation d'équivalence entre les graphes attribués introduite paragraphe 1.3. On va pouvoir «prolonger» la relation d'équivalence aux morphismes.

**Définition 46.** *Soient $g : G \to H$ et $\tilde{g} : \tilde{G} \to \tilde{H}$ deux morphismes de graphes attribués. On dira que $g$ est équivalent à $\tilde{g}$ si*
- *les graphes $G$ et $\tilde{G}$ d'une part et les graphes $H$ et $\tilde{H}$ sont équivalents ;*
- *les parties structurelles de $g$ et $\tilde{g}$ sont identiques ;*
- *pour tout sommet $s_H$ du graphe $H$, on a*

$$(s_H, \mathtt{EA}_H)\mathcal{R}^g(s_G, \mathtt{EA}_G) \iff (s_H, renomme_H(\mathtt{EA}_H))\mathcal{R}^{\tilde{g}}(s_G, renomme_G(\mathtt{EA}_G))$$

  *où $renomme_H$ et $renomme_G$ sont les bijections entre les espaces d'attributs induites par la relation d'équivalence entre respectivement les graphes $H$ et $\tilde{H}$ et les graphes $G$ et $\tilde{G}$ ;*
- *le partitionnement et les fonctions de calculs sont identiques entre les morphismes $g$ et $\tilde{g}$.*

Cette définition traduit l'idée intuitive que si un morphisme est défini entre deux graphes attribués $G$ et $H$, alors un morphisme identique va exister entre des graphes équivalents à $G$ et $H$. Par la suite, on confondra un morphisme et sa classe d'équivalence.

### 2.4 Exemple

Sur l'exemple représenté figure IV.5, on peut voir comment cette construction reflète l'idée intuitive d'un morphisme entre deux graphes attribués. La figure IV.5(a) montre les graphes $G$ et $H$ entre lesquels on veut définir un morphisme. Après avoir donné la partie structurelle du morphisme (figure IV.5(b), il faut alors indiquer quels attributs de $H$ vont influencer ceux de $G$ : sur la figure IV.5(c), les éléments nécessaires de la relation sont indiqués par les traits en gros pointillés alors que les trois en petits pointillés signifie une relation optionnelle. La dernière étape de la construction est d'ajouter les fonctions qui représentent cette influence (figure IV.5(d)).

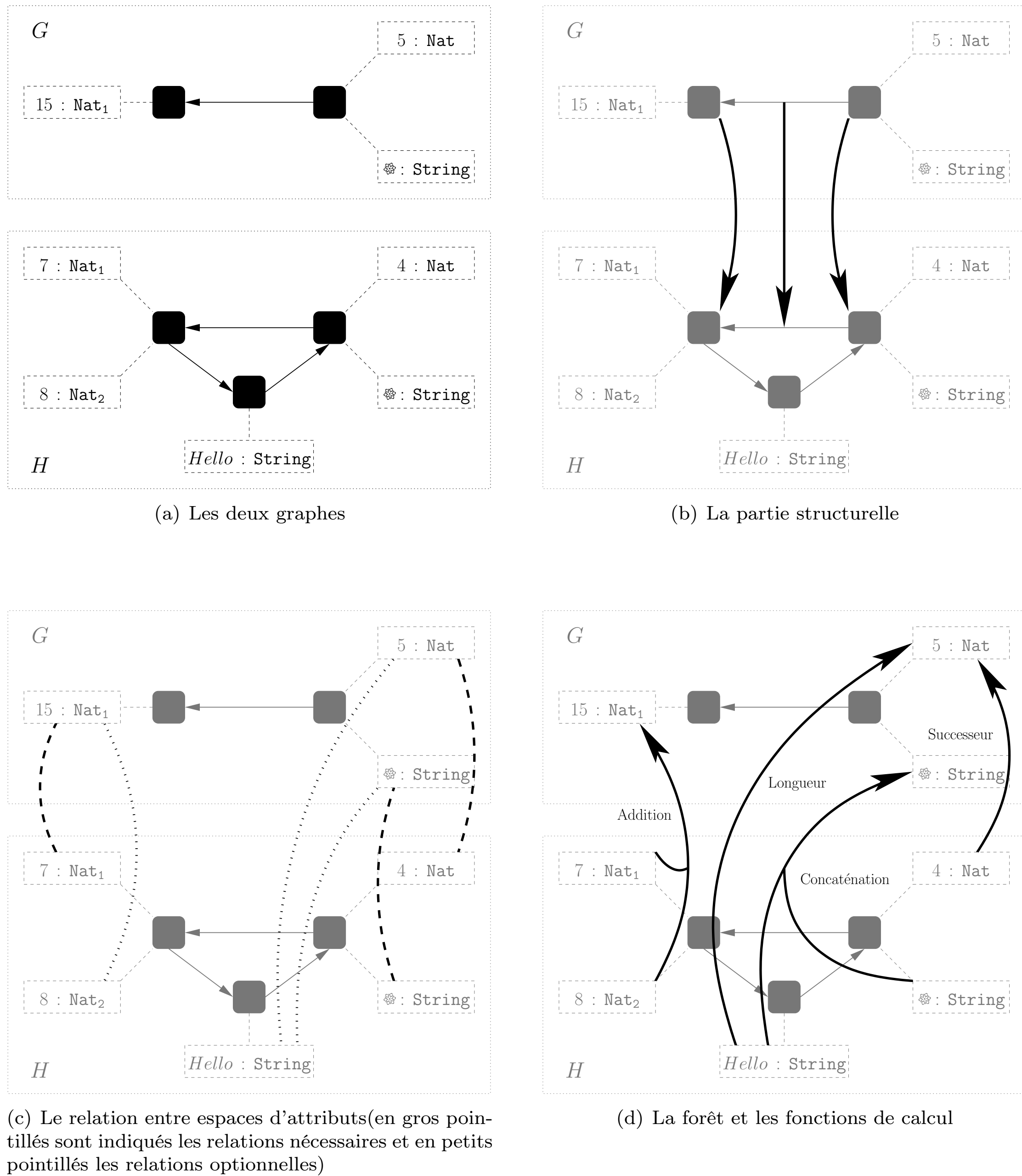
### 2.5 Identification des arbres

Pour terminer la construction des morphismes, il faut indiquer qu'il reste avec cette définition un point peu satisfaisant. En effet, si un arbre possède plusieurs feuilles qui partent du même espace d'attributs appartenant au graphe $H$, alors la fonction de calcul associée prendra plusieurs fois l'attribut associé comme argument. Le morphisme peut alors être simplifié en identifiant les feuilles de chaque arbre qui sont attachées au même espace d'attributs du graphe $H$.

Par la suite, ces différents arbres seront toujours identifiés. De plus, on verra que lors de compositions de morphismes, ce type d'arbre avec des feuilles portant la même information peut naturellement apparaître et il est important de faire la simplification afin de pouvoir démontrer les propriétés universelles sur le pushout (*cf.* section 1.2.17).

## 3 La catégorie AttGraph

Les graphes attribués, tels qu'on vient de les définir, accompagnés des morphismes de graphes attribués vont former une catégorie. Afin de le prouver, on définit maintenant la composition de deux morphismes de graphes, ainsi que les morphismes identités.



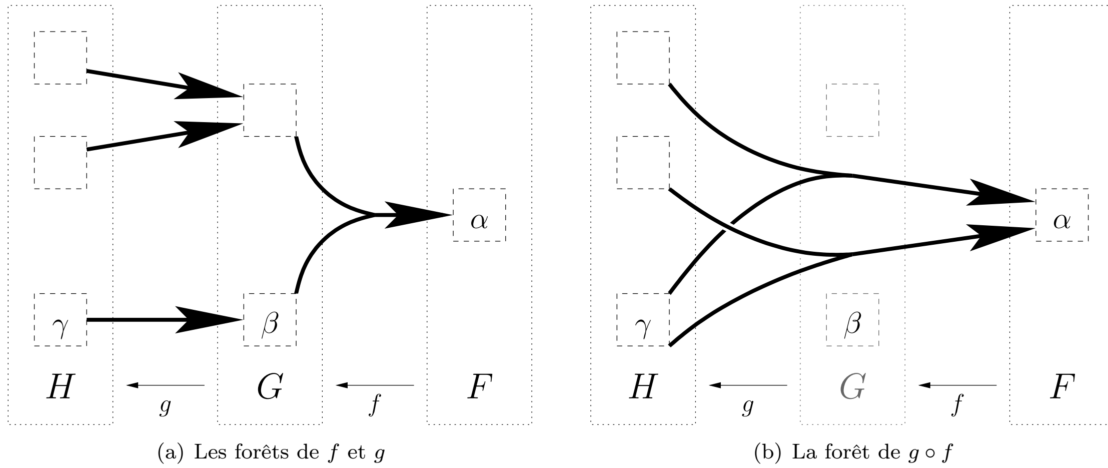

(a) Les deux graphes

(b) La partie structurelle

(c) Le relation entre espaces d'attributs(en gros pointillés sont indiqués les relations nécessaires et en petits pointillés les relations optionnelles)

(d) La forêt et les fonctions de calcul

FIG. IV.5 – Construction d'un morphisme




(a) Les forêts de $f$ et $g$

(b) La forêt de $g \circ f$

FIG. IV.6 – Composition des forêts

## 3.1 Composition de morphismes

Soient $F$, $G$ et $H$ trois graphes attribués et $f : F \to G$ et $g : G \to H$ deux morphismes des graphes attribués.

*3.1.12 Pour la structure.*

La composition des parties structurelles des morphismes ne pose aucun problème : c'est une composition classique de fonctions entre ensembles.

*3.1.13 Pour la forêt.*

La partie délicate de la composition de deux morphismes se trouve dans la construction de la nouvelle relation $\mathcal{R}^{g \circ f}$ et de la forêt associée au morphisme $g \circ f$. En effet, il n'est pas suffisant de dire que si $(s', \mathtt{EA}')\mathcal{R}^f(s, \mathtt{EA})$ et $(s'', \mathtt{EA}'')\mathcal{R}^g(s', \mathtt{EA}')$ alors $(s'', \mathtt{EA}'')\mathcal{R}^{g \circ f}(s, \mathtt{EA})$. Des multiplicités peuvent apparaître pendant la composition. Pour comprendre ce phénomène, on peut regarder le diagramme IV.6. Sur le schéma IV.6(a), on a représenté quelques uns des espaces d'attributs appartenant aux graphes $F$, $G$ et $H$ ainsi que les arbres correspondants. On peut voir que 2 arbres différents du morphisme $g$ arrivent sur la même feuille de l'arbre du morphisme $f$ qui possède deux sources. Pour ne pas perdre d'information durant la composition de ces deux morphismes, il est logique d'introduire dans la forêt du morphisme $g \circ f$ deux arbres avec chacun deux feuilles (figure IV.6(b)). Par conséquent, on avait au départ $\beta \mathcal{R}^f \alpha$ et $\gamma \mathcal{R}^g \beta$ ; et à l'arrivée, on se trouve avec $\gamma$ en relation deux fois avec $\alpha$. Si jamais on n'ajoutait pas cette multiplicité, on ne pourrait construire qu'un seul arbre dans la forêt de $g \circ f$ ; une des deux fonctions de calcul du morphisme $g$ serait alors perdue.

**Proposition 7.** *On suppose $(s', \mathtt{EA}')\mathcal{R}^f(s, \mathtt{EA})$ et $(s'', \mathtt{EA}'')\mathcal{R}^g(s', \mathtt{EA}')$. Si l'arbre $\mathfrak{T}$ qui relie $\mathtt{EA}'$ et $\mathtt{EA}$ possède plus d'une feuille, alors la multiplicité de la relation $(s'', \mathtt{EA}'')\mathcal{R}^{g \circ f}(s, \mathtt{EA})$ sera égale au produit du nombre d'arbres de la forêt du morphisme $g$ arrivant sur chacune des feuilles de $\mathfrak{T}$, excepté la feuille correspondante à l'espace d'attributs $\mathtt{EA}'$.*

Le partionnement associé au morphisme $g \circ f$ se fait alors de manière naturelle.

*3.1.14 Pour les fonctions de calcul.*

Pour les fonctions de calcul, la composition se fait comme une classique composition de fonctions. La condition de compatibilité entre les valeurs des attributs et les fonctions de calcul est automatiquement vérifiée.

**Propriété.** Grâce à la proposition 7, la composition des morphismes est associative.

### 3.2 Morphismes identités

**Définition 47.** *Soit $G$ un graphe attribué. Le morphisme identité de $G$, noté $Id_G$, est donné par :*
- *la partie structurelle est l'identité ;*
- *la relation $\mathcal{R}^{Id_G}$ est réduite aux relations nécessaires (il n'y a alors qu'un seul choix possible pour le partionnement) ;*
- *chaque fonction de calcul vaut l'identité du type inductif associé (la condition de compatibilité est alors automatiquement vérifiée).*

**Proposition 8.** *Les morphismes identités sont des éléments neutres pour la composition des morphismes à gauche et à droite.*

### 3.3 Isomorphismes

**Définition 48.** *Soient $G$ et $H$ deux graphes attribués. Un morphisme $f$ de $G$ vers $H$ sera qualifié d'isomorphisme si :*
- *la partie structurelle est un isomorphisme sur les ensembles correspondants ;*
- *la relation $\mathcal{R}^f$ est réduite aux relations nécessaires ;*
- *chaque fonction de calcul vaut l'identité du type inductif associé (la condition de compatibilité doit être vérifiée).*

### 3.4 Catégorie

Les propriétés sur la composition des morphismes et les morphismes identités permettent maintenant d'affirmer que les (classes d'équivalence des) graphes attribués ainsi que les (classes d'équivalence des) morphismes de graphes attribués forment un catégorie, nommé **AttGraph**.

**Proposition 9.** *Il existe un foncteur d'oubli $\mathcal{F}$ entre la catégorie* **AttGraph** *et celle des graphes simples* **Graph**. *Ce foncteur ne garde que les sommets et les arêtes pour les objets et ne conserve que la partie structurelle des morphismes.*

## 4 Typage des graphes

L'UML et l'approche dirigée par les modèle (MDA) introduit la notion de métamodèle et de métamétamodèle pour décrire la conformité d'un modèle par rapport à une idée plus générale. Par exemple, on sait de décrire de manière simple et générale à quoi doit ressembler un diagramme d'activité. La définition des graphes et des morphismes introduite dans cette thèse permet d'introduire la notion de typage pour les graphes afin de refléter l'idée introduite par les métamodèles et les métamétamodèles dans le cadre du MDA : un graphe spécial, appelé graphe-type, va représenter tout une famille de graphes (appelés les instances) en imposant certaines contraintes sur la structure (par exemple, un nœud en relation avec l'espace d'attributs `Nat` ne pourra pas être relié à un nœud en relation avec l'espace d'attributs `String`), sur les espaces d'attributs ou bien encore sur la valeur des attributs eux-mêmes (par exemple, imposer une valeur à un attribut ou au contraire le laisser complètement libre). Ce concept est inspiré par l'approche développée par Ehrig *et al.* dans [32].

Dans notre approche, le graphe-type va être un simple graphe attribué et c'est une classe spéciale de morphisme qui va coder cette notion de typage.

### 4.1 Morphisme de typage

Ces morphismes spéciaux, dirigés de l'instance vers le graphe-type, doivent permettre de reconnaître tous les espaces d'attributs en relation avec chacun des sommets d'une instance du graphe-type et vérifier si la présence de ces espaces est tolérée par le graphe-type. Cela se fait naturellement grâce à la condition imposée sur les morphismes dans la définition de la section 2.1. Par ailleurs, voulant être en mesure de donner des conditions sur les valeurs des attributs, on ne veut pas que ce morphisme de typage «modifie» les attributs. Par conséquent, la relation associée à

ce morphisme doit être restreinte, ainsi que les fonctions de calcul possibles : ces dernières doivent forcément être égales à l'identité.

**Définition 49.** *La classe $\mathcal{T}$ des morphismes de typage est définie par la propriété suivante : un graphe $t$ sera dans la classe $\mathcal{T}$ si*

- *la relation entre les espaces d'attributs est limitée aux éléments nécessaires ;*
- *le partionnement ne crée que des arbres avec une seule feuille ;*
- *toutes les fonctions de calculs sont égales à l'identité.*

*Pour cette classe de morphismes, la condition imposée sur les jokers est inversée : un joker peut être envoyé sur n'importe quelle valeur du type inductif correspondant par un morphisme de typage ; par contre, une valeur précisément définie ne peut être envoyée sur un joker.*

**Définition 50.** *Soit $G$ et $TG$ deux graphes attribués. On dira que le graphe $G$ est typé par $TG$, ou encore que $G$ est une instance de $TG$, s'il existe un morphisme $t$ entre un représentant de la classe de $G$ et $TG$ appartenant à la classe $\mathcal{T}$. Le morphisme $t$ est alors appelé le morphisme de typage de $G$.*

**Remarque 17.** Par rapport à la notion de métamodèles introduite dans le MDA, notre approche ne permet pas d'avoir des conditions de cardinalité sur les éléments des graphes-types. En effet, si un graphe-type autorise un arc entre un sommet $A$ et un sommet $B$, sur les instances de ce graphe-type, on pourra trouver zéro, une, deux ou encore plus d'arêtes entre les nœuds $A$ et $B$.

## 4.2 Exemple

Le célèbre jeu de pacman offre un exemple simple pour illustrer le typage des graphes (*cf.* [41]). Le graphe-type représenté figure IV.7(a) montre la définition générique d'une carte de jeu de pacman. Les cases reliées entre elles schématisent l'aire de jeu ; les nœuds avec des fantômes, des perles ou bien pacman permettent de repérer sur la carte la position des différents éléments du jeu. Les chaînes de caractères comme attributs permettent de bien differencier les différents types de sommets. De plus, un nombre entier pouvant prendre n'importe quelle valeur permet de compter les vies du héros. La figure IV.7(b) montre une instance de ce graphe-type.

## 4.3 Catégories de graphes attribués typés

Afin de définir la notion de morphismes entre deux graphes attribués et typés par le même graphe-type, on rajoute une condition sur les morphismes : il est en effet logique lorsqu'on travaille en référence à un métamodèle qu'un élément du modèle et son image aient le même représentant dans ce métamodèle. Cela se traduit par le fait qu'un sommet et son image par le morphisme doivent être typés par le même sommet du graphe-type. De même, les espaces d'attributs semblables entre un sommet et son image (c'est-à-dire ceux liés par une relation nécessaire par le morphisme) doivent être représentés par le même espace d'attributs dans le graphe-type. On arrive donc à la définition suivante :

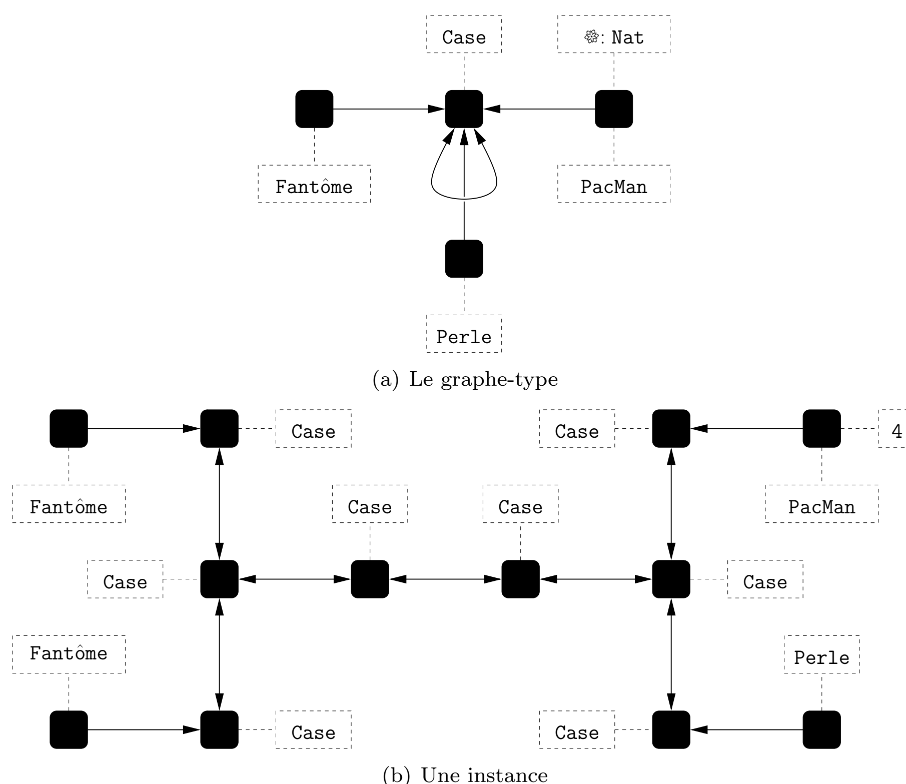
**Définition 51.** *Soient $G$ et $G'$ deux graphes typés par le même graphe-type $TG$ et dont les morphismes de typage sont respectivement $t$ et $t'$. Un morphisme de graphes attribués et typés $f$ sera donné par un morphisme de graphes attribués vérifiant la condition suivante :*

- *les parties structurelles commutent, i.e.*

$$\mathcal{F}(t') \circ \mathcal{F}(f) = \mathcal{F}(t);$$

- *les relations nécessaires commutent.*

Les graphes attribués typés par le graphe-type $TG$ et les morphismes de graphes attribués typés forment une sous-catégorie de la catégorie **AttGraph**, notée **AttGraph**$_{TG}$.




(a) Le graphe-type

(b) Une instance

Fig. IV.7 – Un exemple de typage

### 4.4 Typages successifs

La classe $\mathcal{T}$ est close par composition. Par conséquent, on peut introduire la notion de typage successifs afin de traduire l'idée de métamétamodèles. En effet, si le graphe $TG$ est lui-même typé par un graphe $TTG$ avec le morphisme de typage $t_{TG}$, alors tous les objets de la catégorie $\mathbf{AttGraph}_{TG}$ seront eux aussi typés par $TTG$ et leur morphisme de typage correspondant sera la composition de leur morphisme de typage par rapport à $TG$ et de $t_{TG}$. Par conséquent, si le graphe $TG$ est typé par le graphe $TG'$, la catégorie $\mathbf{AttGraph}_{TG}$ est une sous-catégorie de $\mathbf{AttGraph}_{TTG}$.

Cette notion de typage successif est propre à notre approche DPoPb. En effet, dans l'approche reposant sur les catégories adhésives HLR, le typage d'un graphe s'appuie sur la notion d'algèbre finale. En faisant ainsi, on restreint à un seul étage la possibilité de typer un graphe et il n'est alors plus possible de représenter formellement la notion de métamétamodèle.

## 5 Discussion

Après avoir précisément défini les graphes et les morphismes, on souhaite maintenant pouvoir construire des transformations en suivant les idées introduites dans l'approche « double pushout », en particulier la structure générale des règles. Le chapitre suivant sera donc consacré à la construction du pushout et du pushout-complement dans la catégorie **AttGraph**. Avant cela, on va décrire de manière intuitive l'aspect de nos règles de transformations afin de comprendre les restrictions que l'on va imposer pour construire le pushout.

Comme dans le cas classique, une règle va donc être décrite par trois graphes $K$, $L$ et $R$ et deux morphismes $l : K \to L$ et $r : K \to R$. Les fonctions de calcul utilisées afin de définir les morphismes vont nous autoriser une grande expressivité car on peut se servir de n'importe quelle fonction exprimable dans le cadre de la théorie des types inductifs. Pour appliquer une telle règle à un graphe $G$, il faut trouver un morphisme $m$ entre la partie gauche $L$ de la règle et le graphe source $G$. Quelles propriétés doit vérifier ce morphisme $m$ ?

Le but de ce morphisme étant de repérer un sous-graphe dans le graphe $G$ semblable au graphe $L$ ; il est donc logique d'imposer l'injectivité sur la partie structurelle du graphe. Pour les attributs, afin de respecter les valeurs, on demande à ce que les fonctions de calcul soient toutes égales à l'identité.

À propos des attributs, il faut remarquer l'utilisation presque généralisée des jokers pour les attributs des nœuds des trois graphes de la règles. Les jokers vont ainsi pouvoir jouer le rôle de variables lors des transformations et les fonctions de calcul (ici, des fonctions identités) vont alors faire le lien entre les variables et les valeurs précises du graphe source. De plus, ce lien bien défini permet d'éviter de donner un nom aux variables. Par ailleurs, la possibilité de laisser une valeur particulière pour un ou plusieurs attributs dans le graphe $L$ permet de restreindre le champ d'application de la règle car il faudra avoir la même valeur dans le graphe $G$ afin de contruire le morphisme $m$.

Que dire sur les morphismes $l$ et $r$ ? Dans la descrition des morphismes pour les attributs (*cf.* paragraphe 2.2), on a vu que la première construction de la transformation (le pushout-complement) va être utile pour faire les calculs complexes sur les attributs : pour cela, il va être nécessaire d'autoriser n'importe quelles fonctions de calcul attachées à des arbres pouvant avoir plusieurs feuilles. Il ne va donc pas y avoir de restriction sur la partie concernant les attributs dans le morphisme $l$. On peut d'ailleurs remarquer que l'utilisation des types inductifs permet d'avoir une très grande puissance de calcul disponible pour ces fonctions de calcul (en particulier, les fonctions récursives sont acceptables dans la définition. Pour la seconde partie de la transformation, on a vu que la construction du pushout va, sur le attributs, principalement permettre de recopier les valeurs des attributs. Cette copie se fait grâce à des fonctions de calcul égales à l'identité. Dans la construction générale du pushout, on va donc se restreindre à des fonctions de calcul bijectives, cela sera suffisant pour construire les transformations de graphes.

# Chapitre V

# Les constructions de base

## 1 Le pushout dans la catégorie AttGraph

Dans ce chapitre, on utilise les notations introduites dans l'annexe 4 pour le pushout. Ainsi, si on construit le pushout de deux morphismes $b$ et $c$, les morphismes opposés dans le carré obtenu sont appelés repsectivement $b'$ et $c'$.

### 1.1 Existence du pushout

Dans la catégorie des graphes simples, les objets sont construits exclusivement à base d'ensembles. Or, dans la catégorie des ensembles, si on se donne 3 ensembles $A$, $B$ et $C$ ainsi que deux applications $b : A \to B$ et $c : A \to C$ alors le pushout de $b$ et $c$ existe toujours. Par conséquent, le pushout de deux morphismes de graphes simples est lui aussi toujours constructible.

En revanche, le fait de rajouter des attributs à la structure des graphes complique les choses ; l'existence du pushout de deux morphismes n'est pas assuré et il faut alors rajouter des conditions à ces morphismes pour être sûr d'arriver à un résultat. Dans l'approche $HLR$, une classe $\mathcal{M}$ de morphismes est définie dans ce but ; Dans notre cadre de travail, les forêts et les fonctions de calcul compliquent encore un peu plus les constructions. On peut facilement décrire un exemple problématique. En effet, des contradictions peuvent apparaître lorsque l'on cherche à calculer les fonctions de calcul des morphismes $b'$ et $c'$ : sur le même arbre, on peut avoir à placer deux fonctions différentes. Pour éviter ces problèmes, on va ici définir deux classes de morphismes de graphes attribués. Les restrictions que nous allons apporter à ces classes ne vont pas être contraignantes car elles vont principalement suivre le cahier des charges présenté à la fin du chapitre précédent (*cf.* section IV.5).

La première classe définie ici est utile pour repérer exactement un sous-graphe dans un graphe plus grand. Ce type de morphismes est utile pour appliquer une règle de transformation à un graphe $G$ ; pour cela, il va falloir chercher un sous-graphe de $G$ correspondant exactement à la partie gauche de la règle. Dans ce but, il est alors logique d'imposer l'injectivité sur la partie structurelle des morphismes. Comme dans le cas des morphismes de typage (voir section 4.1), la partie structurelle va aussi nous assurer de la présence des bons espaces d'attributs. Il reste alors à vérifier que les attributs associés ont la bonne valeur : pour cela, on limite la relation du morphisme aux éléments nécessaires (on a alors qu'un seul choix pour le partitionnement) et les fonctions de calcul aux fonctions identités.

**Définition 52.** *La classe $\mathcal{N}$ de morphismes de graphes attribués est composée par les morphismes vérifiant les propriétés suivantes :*
- *la partie structurelle est injective ;*
- *la relation entre espaces d'attributs est limitée aux éléments nécessaires ;*
- *les fonctions de calcul sont les fonctions identités des espaces d'attributs correspondants.*

La seconde classe de morphismes que l'on va définir nous servira à décrire les parties droites des règles de transformation de graphes. En suivant la discussion du chapitre précédent, on définit ainsi la classe $\mathcal{M}'$ :

**Définition 53.** *Un morphisme $f$ de graphes attribués sera dans la classe $\mathcal{M}'$ si chaque fonction de calcul de $f$ est une bijection.*

**Théorème 5.** *Si le morphisme $b : A \to B$ appartient à la classe $\mathcal{M}'$ et le morphisme $c : A \to C$ appartient à la classe $\mathcal{N}$, alors le pushout de $b$ et $c$ existe à la vérification de la compatibilité des attributs près.*

Dans ce théorème, la construction du pushout peut être bloquée par des incohérences entre les valeurs concrètes des attributs. Ce problème n'est pas un problème théorique, mais il reflète des situations réelles : par exemple, si on essaye de fusionner deux attributs qui ont au départ des valeurs différentes, il est normal de tomber sur un écueil : il n'y a pas de valeur naturelle pour le nouvel attribut.

Il y a alors plusieurs manières de traiter les cas problématiques. La solution conservatrice est d'affirmer que la construction est impossible et ainsi de ne renvoyer aucun résultat. Ainsi, on est sûr de ne pas produire des résultats incohérents. La seconde solution consiste à «factoriser» les espaces d'attributs correspondants aux attributs problématiques afin d'identifier les valeurs incompatibles (la résolution de contraintes induites par cette factorisation peut alors conduire dans certains cas à des espaces d'attributs avec seulement un élément). Avec cette approche, même si de l'information est perdue pendant la factorisation des espaces d'attributs, la construction peut être finie. Dans certaines situations, il peut s'avérer utile d'obtenir toujours un résultat, même s'il est partiel. Dans la suite, on privilégiera la solution conservatrice.

La section suivante est consacrée à la construction explicite du pushout afin de fournir une preuve au théorème 5.

## 1.2 La construction explicite du pushout

On suppose ici que le morphisme $b$ appartient à la classe $\mathcal{M}'$ et que le morphisme $c$ appartient à la classe $\mathcal{N}$.

Comme pour la définition des graphes et des morphismes, on décompose la construction du pushout en plusieurs étapes. La première pour la structure et les suivantes pour tout ce qui concerne les espaces d'attributs et les attributs eux-mêmes.

#### *1.2.15 Pour la structure*

Pour construire la structure du graphe que l'on cherche, on peut utiliser le foncteur d'oubli $\mathcal{F}$ sur $A$, $B$ et $C$ ainsi que sur les morphismes $b$ et $c$ pour se ramener dans la catégorie des graphes simples. On peut alors calculer le pushout dans cette catégorie (voir section 2). Les résultats trouvés sont intéressants pour notre construction : la pushout dans la catégorie des graphes simples sera alors la partie structurelle de notre graphe attribué et les morphismes $b'$ et $c'$ vont aussi servir de base à l'écriture des morphismes souhaités. On rappelle que comme $b$ et $c$ sont des morphismes structurellement injectifs, les morphismes $b'$ et $c'$ vont partager cette caractéristique.

#### *1.2.16 Pour les espaces d'attributs*

Soit $s_D$ un sommet du graphe $D$. Pour savoir quels sont les espaces d'attributs en relation avec $s_D$, on distingue trois cas :

- le sommet $s_D$ possède une préimage seulement par le morphisme $c'$ ;
- le sommet $s_D$ possède une préimage seulement par le morphisme $b'$ ;
- le sommet $s_D$ possède une préimage par les deux morphismes $b'$ et $c'$ ; et donc le carré est complété par un unique nœud $s_A$ du graphe $A$.

Comme on ne considère ici que des morphismes injectifs, aucun autre cas n'est possible. Dans le premier cas, le sommet $s_D$ sera mis en relation avec les mêmes espaces d'attributs que sa préimage $s_B$ appartenant au graphe $B$. Le second cas se traite de la même manière et $s_D$ sera relié aux mêmes espaces d'attributs que sa préimage $s_C$.

Enfin, pour le troisième cas, suivant l'idée intuitive que le pushout permet de faire un collage des graphes $B$ et $C$ suivant le graphe $A$, on va retrouver comme espaces d'attributs en relation avec $s_D$ les mêmes espaces d'attributs qui sont en relation avec $s_A$, plus ceux qui sont en relation avec

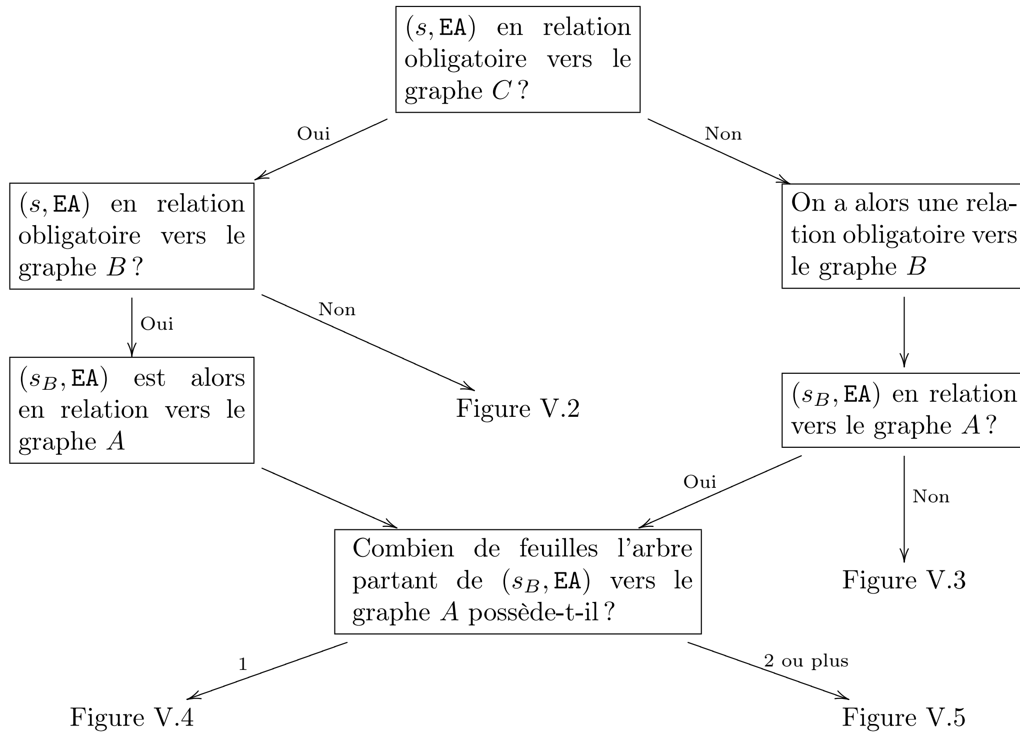



FIG. V.1 – Les différents cas pour l'étude des nouvelles forêts

$s_B$ sans être en relation avec $s_A$, c'est-à-dire $\mathcal{R}^B_{s_B} \backslash \mathcal{R}^A_{s_A}$ et enfin $\mathcal{R}^C_{s_C} \backslash \mathcal{R}^A_{s_A}$. Il faut faire attention ici car un renommage des espaces d'attributs peut être nécessaire avant de calculer $\mathcal{R}^D_{s_D}$. Prenons un exemple simple pour illustrer cette difficulté, si jamais $s_A$ n'est en relation avec aucun espace d'attributs et si $s_B$ et $s_C$ sont chacun en relation avec `Nat`, le sommet $s_D$ devra être en relation deux fois avec l'espace `Nat`. Ce qui est impossible d'après la définition d'un graphe. Il faudra donc prendre un graphe $C'$ équivalent à $C$ et remplacer le nom `Nat` par `Nat'`. Ainsi, le nœud du graphe $D$ sera en relation avec deux copies du même espace d'attributs et la définition sera respectée.

Placer ainsi les espaces d'attributs rend licite la définition des morphismes structurels que l'on a donnés avant : en effet, un sommet du graphe $D$ possède au moins les même espaces d'attributs que ses préimages, les relations obligatoires peuvent donc exister ici.

*1.2.17 Pour les forêts*

La description des forêts et des fonctions de calcul des morphismes $b'$ et $c'$ est le point délicat de la construction du pushout. Il faut étudier localement, pour chaque espace d'attributs `EA` en relation avec un sommet $s$ du graphe $D$, les relations des morphismes $b$ et $c$ associées à `EA` en relation avec les préimages de $s$ dans les graphes $B$ et $C$. Il faut pour cela distinguer plusieurs cas. Pour autant, cette étude est facilitée par les conditions imposées sur la classe $\mathcal{N}$. Le schéma dessiné figure V.1 permet de différencier ces différents cas.

Les deux premiers cas sont simples (figures V.2 et V.3) : un attribut qui est seulement dans le graphe $B$ ou bien seulement dans le graphe $C$ se retrouvera à l'identique dans le graphe $D$. Par conséquent, il est logique de créer un arbre simple entre les deux espace d'attributs correspondants et de lui attribuer la fonction de calcul $Id_{\mathtt{EA}}$.

Les deux derniers cas (figures V.4 et V.5) sont les plus intéressants. Il faut tout d'abord signaler qu'entre les espaces d'attributs du graphe $D$ et ceux du graphe $C$, il peut, ou non, avoir des relations obligatoires. De plus, dans un souci de clarté, sur la figure V.5, on a représenté un arbre avec seulement deux feuilles, mais on peut facilement généraliser à des arbres avec un plus grand nombre de feuilles. Enfin, toujours à propos de cet arbre, les espaces d'attributs attachés aux feuilles

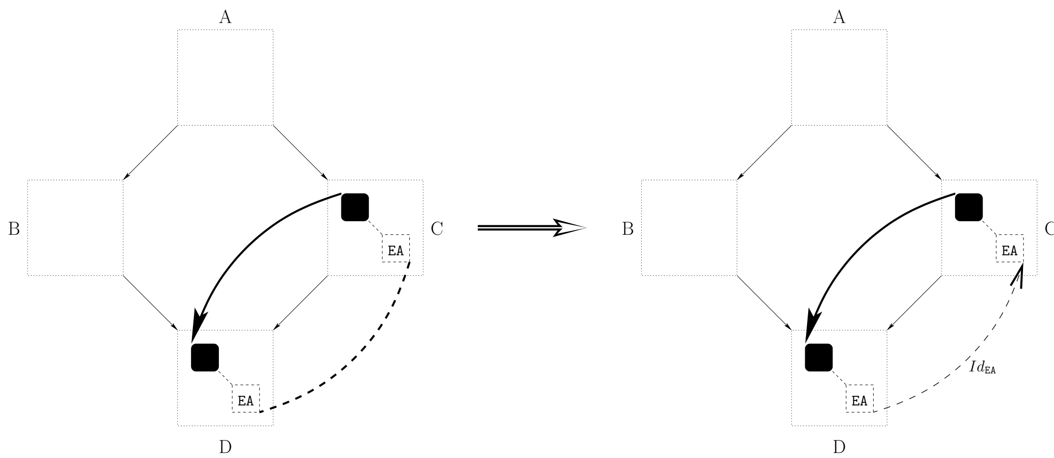



FIG. V.2 – Construction de forêt : Cas 1

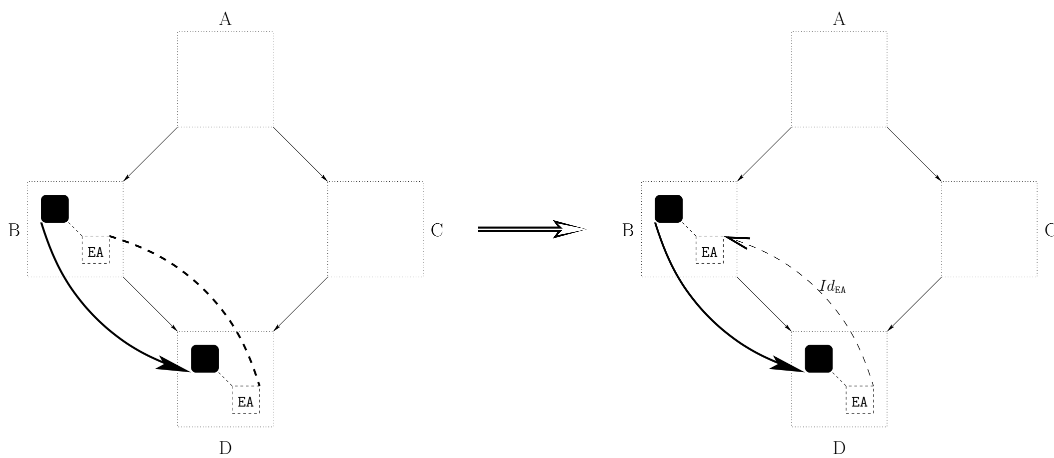



FIG. V.3 – Construction de forêt : Cas 2

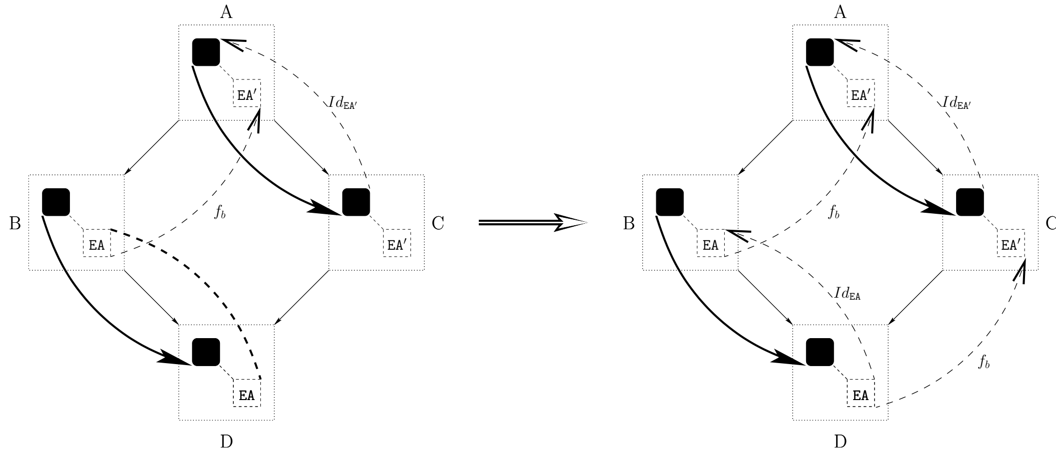



FIG. V.4 – Construction de forêt : Cas 3

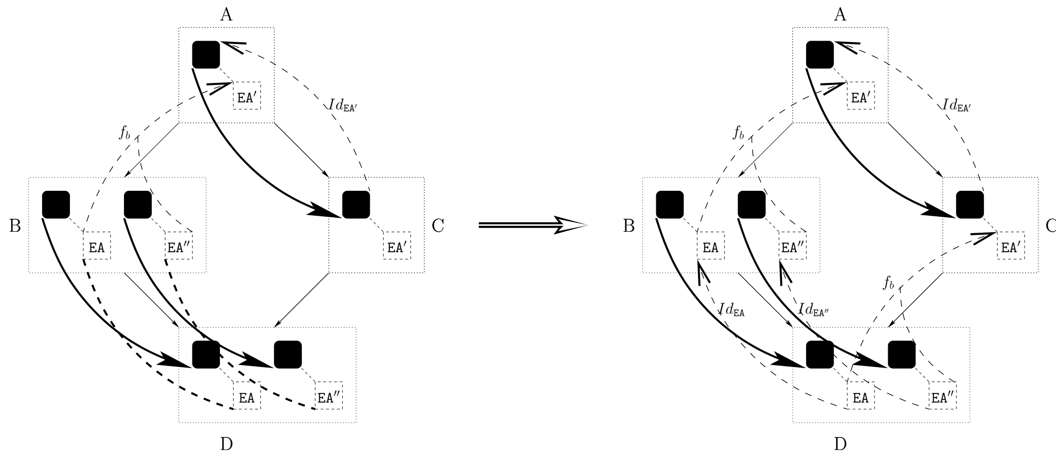
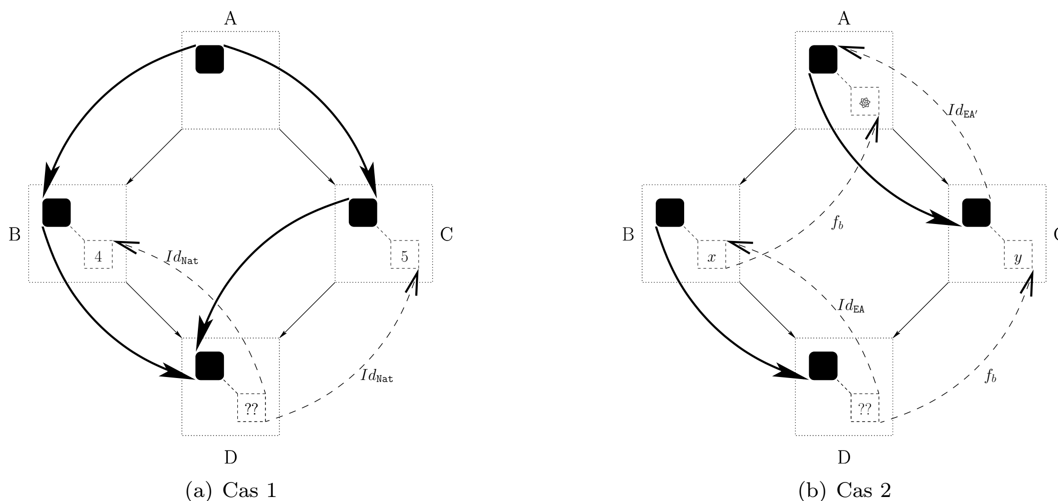



FIG. V.5 – Construction de forêt : Cas 4

pourraient être en relation avec le même sommet du graphe $B$ sans changer le résultat. Ces deux cas sont les seuls où l'on voit apparaître des éléments non nécessaires dans les relations associée aux morphismes et des fonctions de calcul différentes de l'identité. On s'aperçoit ici qu'une telle fonction de calcul définie dans le morphisme $b$ va se retrouver à l'identique dans le morphisme $b'$. Cette propriété nous sera très utile dans l'optique des transformations de graphes : en effet, si on définit une règle avec une fonction particulière, l'application de cette règle à un graphe $G$ (c'est-à-dire le calcul de pushouts) permettra d'appliquer la même fonction à $G$. Enfin, le figure V.5 permet aussi de comprendre pourquoi les fonctions de calcul du morphisme $c$ doivent valoir l'identité : si cette fonction ne vaut plus $Id_{\mathtt{EA}'}$, il faudra aussi changer les valeurs des fonctions de calcul du morphisme $c'$ afin de rendre le carré commutatif et il n'y a *a priori* aucun moyen de savoir s'il existe des fonctions pour remplacer $Id_{\mathtt{EA}}$ et $Id_{\mathtt{EA}''}$ et de les construire.

### *1.2.18 Pour les attributs*

Afin de terminer la construction du pushout, il reste à compléter la définition du graphe $D$ en plaçant les valeurs explicites des attributs. Celles-ci doivent satisfaire la condition imposée sur les fonctions de calcul donnée dans le définition 44. Dans la plupart des cas, ce calcul est évident. Pour autant, il existe des cas problématiques. Deux de ces cas sont représentés figure V.6. Le premier pose clairement problème : il est impossible de trouver un entier qui soit à la fois égal à 4 et à 5. Pour le deuxième, on va être bloqué si $f_b^{-1}(y) \neq x$. Dans les deux configurations, la construction du pushout est alors impossible.




(a) Cas 1 (b) Cas 2

FIG. V.6 – Deux cas problématiques pour le placement des attributs

Par ailleurs, la figure V.6(b) permet de comprendre l'importance d'avoir des fonctions de calcul bijectives dans la définition de la classe $\mathcal{M}'$. En effet, si l'attribut $x$ est remplacé par un joker ❀, le seul moyen de calculer la valeur de l'attribut présent dans le graphe $D$ est de calculer une préimage de $y$ par $f_b$. Si jamais cette fonction n'est pas bijective, ni l'existence ni l'unicité de l'attribut ne sont assurées.

## 1.3 Stabilité des classes $\mathcal{M}'$ et $\mathcal{N}$

La construction explicite du pushout de deux morphismes de graphes attribués, l'un appartenant à la classe $\mathcal{M}'$ et l'autre à la classe $\mathcal{N}$, permet de remarquer que, dans le carré commutatif considéré, les deux morphismes sur des côtés opposés du carré appartiennent à la même classe.

**Proposition 10.** *Les classes $\mathcal{M}'$ et $\mathcal{N}$ sont conjointement stables par passage au pushout.*

## 1.4 La preuve de la propriété universelle du pushout dans la catégorie AttGraph

Afin de prouver que la construction précédente est valide, il faut vérifier que la propriété universelle du pushout est satisfaite. Pour cela, on se donne trois graphes attribués $A$, $B$ et $C$ ainsi que deux morphismes $b : A \to B$ appartenant à la classe $\mathcal{M}'$ et $c : A \to C$ appartenant à la classe $\mathcal{N}$. On suppose que le pushout $D$ de $b$ et $c$ existe. On se donne un graphe supplémentaire $E$ avec deux morphismes $f : B \to E$ et $g : C \to E$ tels que $f \circ b = g \circ c$. Le but est donc de factoriser conjointement et de manière unique les morphisme $f$ et $g$ par le graphe $D$. C'est-à-dire trouver un morphisme $d : D \to E$ unique tel que $f = d \circ c'$ et $g = d \circ b'$.

La construction de $d$ va elle aussi être décomposée suivant les même étapes : on commence par chercher la partie structurelle de $d$, puis on construit la forêt accompagnée des fonctions de calcul.

Pour la partie structurelle, on utilise encore une fois le foncteur d'oubli $\mathcal{F}$. Le diagramme est alors envoyé dans la catégorie des graphes simples et la propriété universelle du pushout dans cette catégorie nous permet d'obtenir la partie structurelle du morphisme recherché entre $D$ et $E$. Afin de vérifier que cette partie structurelle est valide, il faut s'assurer que la condition 40 sur les espaces d'attributs est bien satisfaite. Cette condition est facilement vérifiable : en effet, si on considère un sommet $s_D$ du graphe $D$, deux cas se présentent : le nœud $s_D$ possède une préimage $s_B$ seulement par le morphisme $c'$ (respectivement une préimage $s_C$ seulement par le morphisme $b'$), on a alors $\mathcal{R}^D_{s_D} = \mathcal{R}^B_{s_B}$ (respectivement $\mathcal{R}^D_{s_D} = \mathcal{R}^C_{s_c}$) et la condition est alors trivialement vérifiée. Dans le second cas, le sommet $s_D$ va posséder une préimage par chacun des morphisme $c'$ et $b'$. On sait alors que $s_B$ et $s_C$ vont partager la même préimage $s_A$ dans le graphe $A$. La construction du

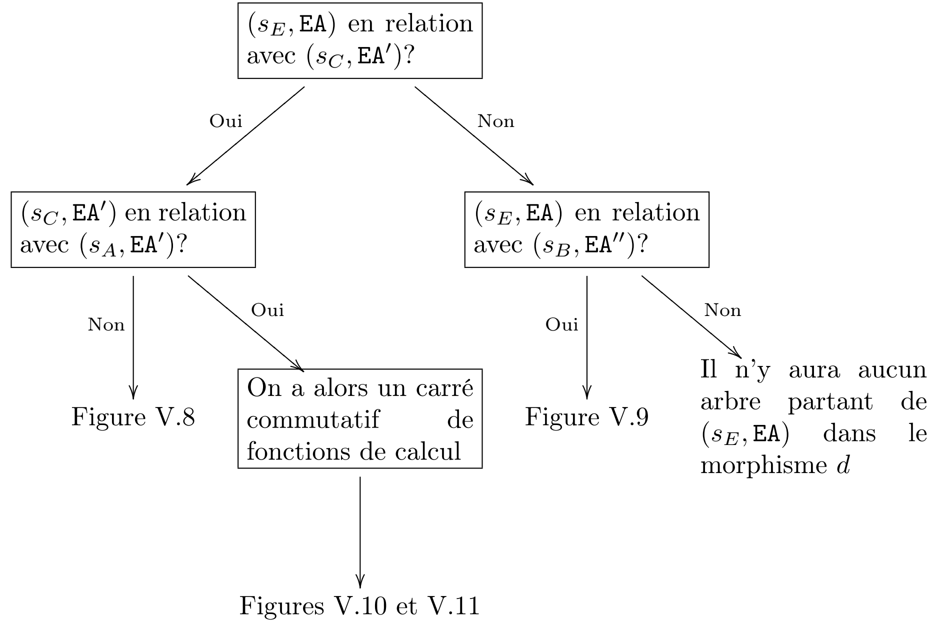



FIG. V.7 – Les différents cas pour l'étude de la forêt du morphisme de la propriété universelle

pushout nous affirme alors que $\mathcal{R}^D_{s_D} = \mathcal{R}^C_{s_C} \cup \mathcal{R}^B_{s_B}$. Enfin, la commutativité du diagramme $ABCE$ nous assure alors que $\mathcal{R}^D_{s_D} \subset \mathcal{R}^E_{d_S(s_D)}$.

Pour le traitement des arbres et des fonctions de calcul, on va de nouveau regarder localement pour chaque espace d'attributs en relation avec un sommet du graphe $E$ les différents arbres dans les morphismes le concernant. Plusieurs cas peuvent se produire et le schéma présenté figure V.7 permet alors pour un espace d'attributs en relation avec un sommet du graphe $E$, noté $(s_E, \mathtt{EA})$, de différencier ces cas.

Les deux premiers cas (figures V.8 et V.9) sont très simples. Ils correspondent aux cas où un espace d'attributs est rajouté par le graphe $B$ ou le graphe $C$ lors du pushout. Le traitement de telles configurations se fait juste en reprenant l'arbre et la fonction de calcul du morphisme $f$ ou $g$ pour les replacer dans le morphisme $d$. Sur ces schémas sont représentés des arbres avec plusieurs feuilles, mais la démarche est la même avec des arbres simples.

Le dernier cas (figures V.10 et V.11) est plus intéressant car on voit apparaître des fonctions de calcul différentes de l'identité dans le pushout. Ici, la commutativité du diagramme est indispensable pour conclure. En effet, pour le cas décrit figure V.10, la condition imposée aux morphismes $f$ et $g$ permet d'affirmer que $f_b \circ f_f = f_g \circ Id_{\mathtt{EA}''} = f_g$. Par conséquent, comme la fonction de calcul $f_b$ se retrouve aussi dans le morphisme $b'$ entre $C$ et $D$, on va pouvoir placer $f_f$ dans le morphisme entre $D$ et $E$. De cette manière, on va bien réussir à factoriser de manière conjointe les fonctions de calcul des morphismes $f$ et $g$. L'intérêt du le second cas représenté réside dans le fait qu'il apparaît naturellement des arbres à identifier (*cf.* section 2.5) dans la composition des morphismes $b'$ et $d$. En effet, dans le morphisme $g$, un seul arbre part du l'attribut $\mathtt{EA}'''$. Or dans le morphisme $d$, on a deux arbres qui partent de ce même espace d'attributs. Ces deux arbres sont réunis grâce à un arbre à deux feuilles dans le morphisme $b'$. Mais pour pouvoir conclure que la composition des arbres équivaut bien à l'arbre du morphisme $g$, il est nécessaire d'identifier les deux feuilles.

Enfin, il faut signaler qu'il existe d'autres configurations pour la construction de la forêt du morphisme $d$ qui viennent compléter celles présentées figures V.10 et V.11, mais ces cas supplémentaires ne présentent pas d'intérêt particuler et le processus pour la construction des arbres est le même et c'est toujours les fonctions de calcul du morphisme $f$ que l'on va retrouver dans le morphisme $d$.

Il reste à vérifier que le morphisme $d$ ainsi construit est bien unique. Pour la partie structurelle, la propriété universelle du pushout dans la catégorie des graphes simples nous assure de son unicité.

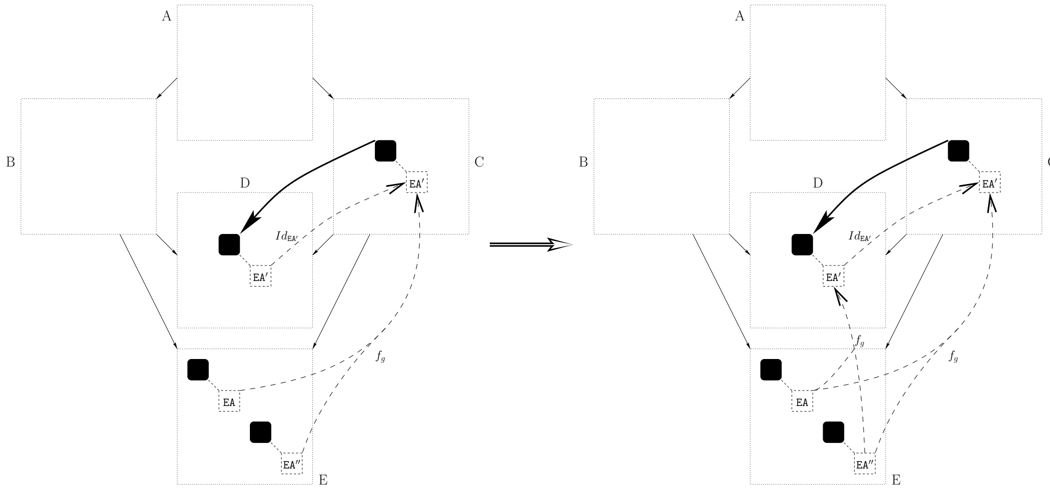



Fig. V.8 – Construction de forêt pour la propriété universelle : cas 1

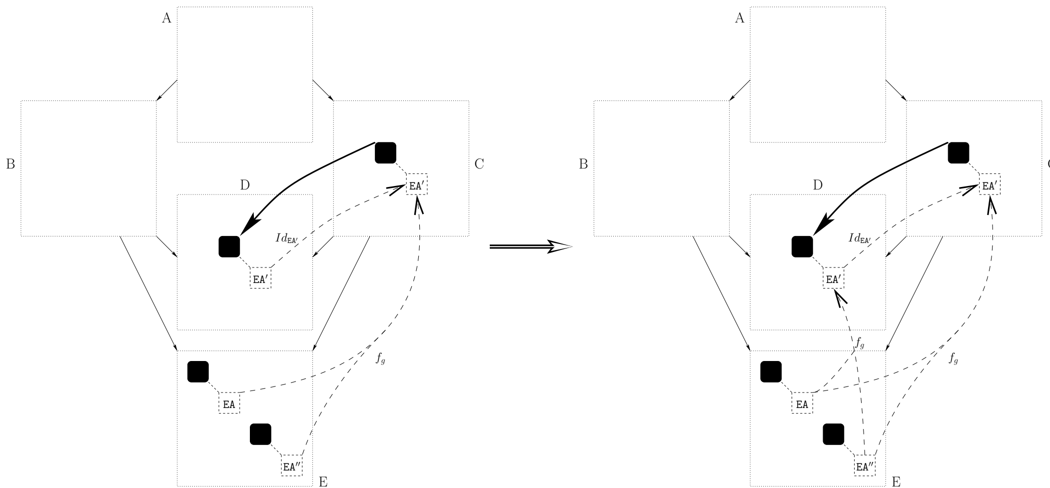



Fig. V.9 – Construction de forêt pour la propriété universelle : cas 2

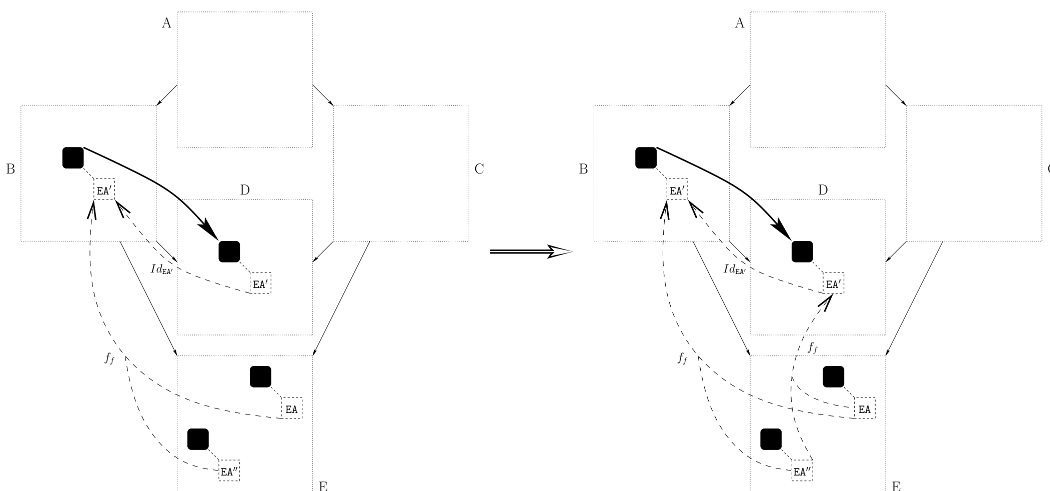



Fig. V.10 – Construction de forêt pour la propriété universelle : cas 3

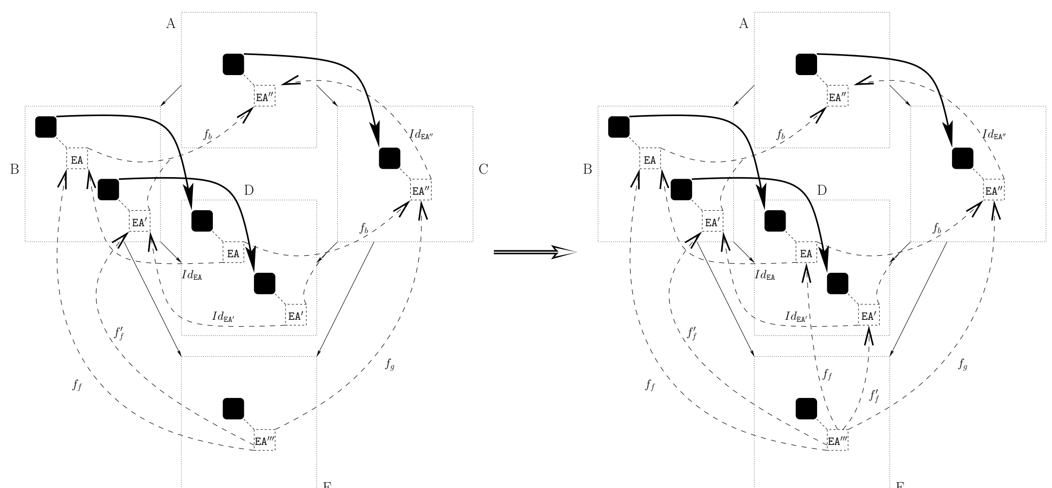


FIG. V.11 – Construction de forêt pour la propriété universelle : cas 4

Pour la relation associée au morphisme $d$, il faut remarquer que chaque espace d'attributs du graphe $D$ est en relation avec au moins un espace d'attributs du graphe $B$ ou du graphe $C$. Par conséquent, le fait de rajouter une relation dans le morphisme $d$ se répercutera forcément dans la composition $d \circ c'$ ou la composition $d \circ b'$. Ces deux morphismes ne pourront alors plus être respectivement égaux à $f$ et $g$. La relation associée au morphisme $d$ est donc unique. La même remarque peut être faite pour les forêts. Enfin, la manière de placer les fonctions de calcul dans la construction du pushout nous assure ici qu'il n'y a pas d'autre choix pour les fonctions de calcul portées par les arbres de $d$.

Cette construction de la pripriété universelle du pushout montre donc que la construction explicite du pushout proposé dans le paragraphe 1.2 est la bonne.

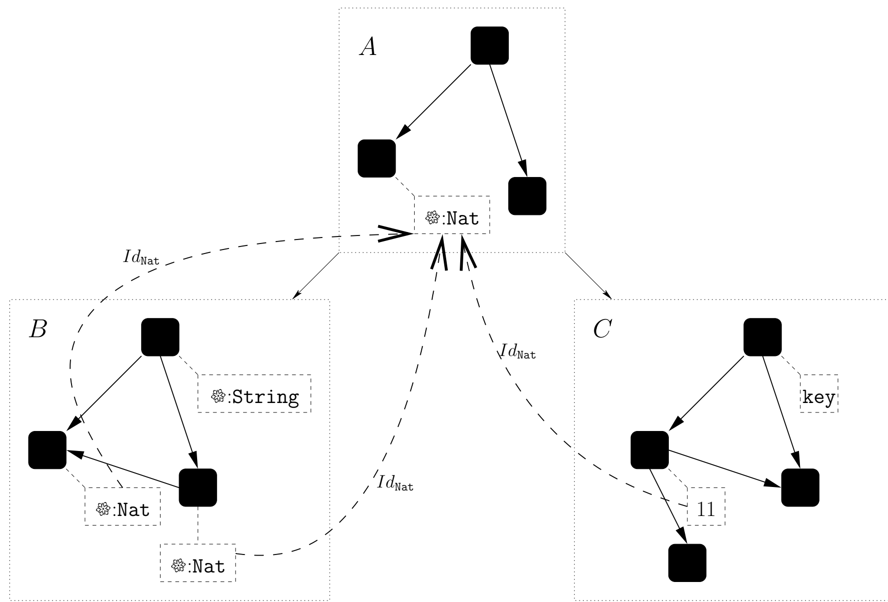
### 1.5 Exemple

On termine la partie sur le pushout en donnant un exemple de construction de pushout. La figure V.12(a) représente les données de départ. Sur ce schéma, afin qu'il reste lisible, les parties injectives des morphismes ne sont pas dessinées. La figure V.12(b) montre la partie structurelle du résultat. Sur la figure V.12(c), on a rajouté les espaces d'attributs en relation avec les sommets : il faut en particulier remarquer que les espaces d'attributs `String` ont été renommés afin de pouvoir accrocher deux chaînes de caractères sur le même sommet. Enfin, le dernier schéma montre le placement explicite des attributs ainsi que les forêts associées aux morphismes nouvellement construits. Chacun de ces arbres porte ici la fonction identité correspondante à l'espace d'attributs du domaine (ou du codomaine) de la fonction.

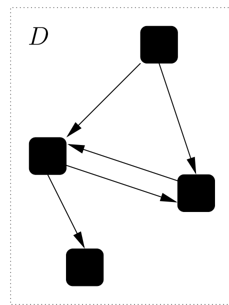
## 2 Le pushout-complement dans la catégorie AttGraph

### 2.1 «Gluing condition» et existence du pushout-complement

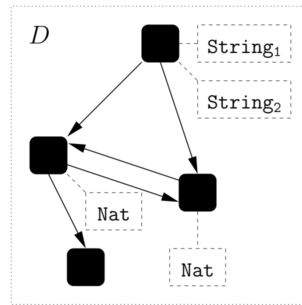
Dans le cadre le l'approche «double pushout» des transformations de graphes, l'execution de la première étape suppose que la contruction du pushout-complement de deux morphismes est possible. le pushout-complement joue alors le rôle de gomme et permet d'enlever des éléments du graphe. Il faut toutefois faire attention lorsque des éléments sont ainsi retirés du graphe car le résultat peut alors ne plus être cohérent vis-à-vis de la définition même d'un graphe : on peut par exemple se retrouver avec un arc qui n'est plus attaché à aucun sommet. Il faut donc s'assurer *a priori* que le résultat sera cohérent. On introduit pour cela une condition, appelé «gluing condition» ou condition de collage, que doivent vérifier les morphismes $b$ et $c'$ avant de commencer le calcul du pushout-complement. Cette condition très classique dans la théorie algébrique de transformation



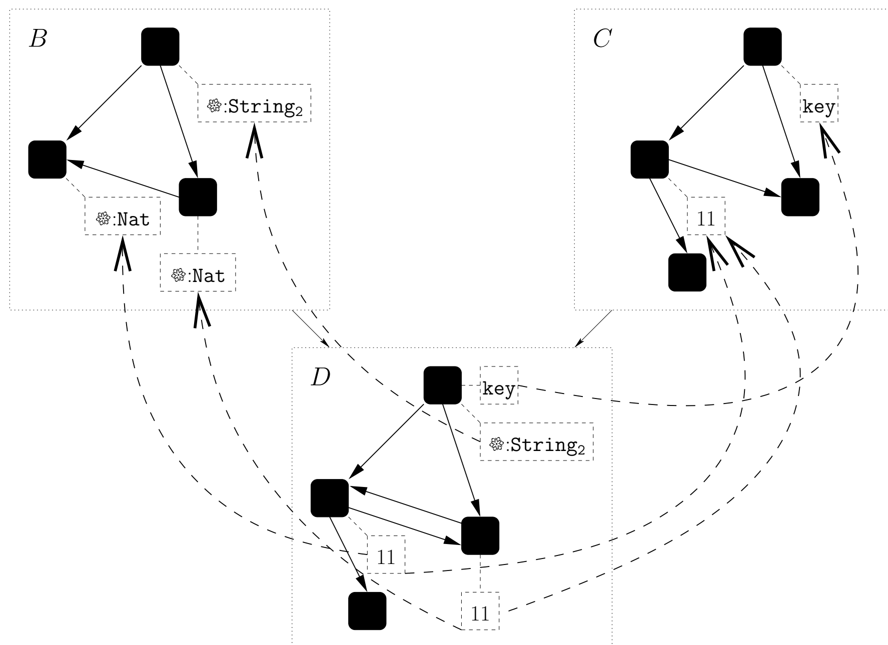

(a) Les données




(b) La structure




(c) Les espaces d'attributs




(d) Les forêts et le placement des attributs

FIG. V.12 – Les différentes étapes de la construction

de graphes (*cf.* [21]) est ici légèrement renforcée afin de prendre en compte les espaces d'attributs attachés aux sommets.

On définit pour cela trois ensembles :

$$Gluing = \{s_B \in S^B \mid \exists s_A \in S^A \text{ tel que } b(s_A) = s_B\},$$

$$\begin{aligned} Dangling \quad = \quad & \Big\{ s_b \in S^B \mid \exists s_D \in S^D,\ \exists a \in A^D(c'(s),t) \cup A^D(t,c'(s)) \\ & \text{tel que } a \text{ n'ai pas de préimage par } c' \Big\}, \end{aligned}$$

$$Floating = \Big\{ s_B \in S^B \mid \exists \mathtt{EA} \in \mathcal{R}^D_{c'(s_B)} \text{ tel que } \mathtt{EA} \notin \mathcal{R}^B_{s_B} \Big\}.$$

L'ensemble *Dangling* représente l'ensemble des nœuds du graphe $B$ auxquels un arc (sortant ou rentrant) est rajouté au cours du morphisme $c'$. Si jamais un de ces nœuds est effacé au cours du pushout-complement, un arc pendouillant apparaîtra alors dans le graphe $C$. L'ensemble *Floating* représente pour sa part l'ensemble des nœuds du graphe $B$ auxquels un espace d'attributs est rajouté au cours du morphisme $c'$. Effacer un de ses sommets implique de retrouver un espace d'attributs dans le graphe $C$ en relation avec aucun sommet. Il faut donc s'assurer qu'aucun des sommets des ensembles *Dangling* et *Floating* ne soit effacé. Cela se fait grâce au troisième ensemble. L'ensemble *Gluing* représente en effet les sommets qui sont conservés lors de la construction du pushout-complement.

**Définition 54.** *On dit que les morphismes $b$ et $c'$ vérifient la «gluing condition» si on a :*

$$Dangling \cup Floating \subset Gluing.$$

Afin de suivre les conclusions de la discussion du paragraphe IV.5, on introduit une troisième classe de morphismes qui nous sera utile pour construire les parties gauches des règles de transformation :

**Définition 55.** *Un morphisme $f$ de graphes attribués sera dans la classe $\mathcal{M}''$ si la partie structurelle de $f$ est injective.*

**Théorème 6.** *Si le morphisme $b : A \to B$ appartient à la classe $\mathcal{M}''$ et le morphisme $c' : B \to D$ appartient à la classe $\mathcal{N}$, si de plus les morphismes $b$ et $c'$ vérifient la «gluing condition», alors le pushout de $b$ et $c'$ existe à la vérification de la compatibilité des attributs près.*

Comme pour le pushout, la construction du pushout-complement peut être bloquée à la dernière étape si pour un même espace d'attributs plusieurs valeurs différentes sont calculées lors de la construction.

## 2.2 Construction explicite

La construction explicite du pushout-complement suit exactement le même schéma que la construction du pushout. On commence par se ramener dans la catégorie des graphes simples à l'aide du foncteur d'oubli $\mathcal{F}$. Il faut remarquer que la condition de collage imposée ici (*cf.* définition 54) est plus contraignante que la «gluing condition» définie dans l'approche classique «double pushout» [17]. Ceci nous assure que les images par $\mathcal{F}$ des graphes et des morphismes considérés vont satisfaire la condition de collage dans la catégorie des graphes simples et donc que la partie structurelle du pushout-complement va pouvoir être construite.

Soit $s_c$ un sommet du graphe $C$ construit. Définir l'ensemble $\mathcal{R}^C_{s_C}$ va se faire de deux manières différentes suivant que $s_C$ possède une préimage $s_A$ dans le graphe $A$ par $c$ ou non. Si $s_C$ ne possède pas de préimage, alors $\mathcal{R}^C_{s_C} = \mathcal{R}^D_{b'(s_C)}$. Sinon, l'ensemble $\mathcal{R}^C_{s_C}$ contiendra tout d'abord $\mathcal{R}^A_{s_A}$, plus les espaces d'attributs en relation avec $b'(s_C)$ qui ne proviennent pas du graphe $B$ ; c'est-à-dire :

$$\mathcal{R}^C_{s_C} = \mathcal{R}^A_{s_A} \cup \left( \mathcal{R}^D_{b'(s_C)} \backslash \mathcal{R}^B_{b(s_A)} \right).$$

La vérification que les parties structurelles des morphismes satisfassent bien la condition de la définition 40 se fait de la même manière qu'expliquée dans la construction du pushout.

**Remarque 18.** Contrairement à ce qui se passe dans la construction du pushout, il n'y a ici jamais besoin de faire de renommage pour construire $\mathcal{R}^C_{s_C}$ car on ne fait que «retirer» des éléments à $\mathcal{R}^D_{b'(s_C)}$. Il ne peut donc pas apparaître des espaces d'attributs avec le même nom.

La construction des forêts et des fonctions de calcul se fait aussi par l'étude locale des espaces d'attributs en relation avec les sommets du graphe $C$. On retrouve alors des schémas similaires à ceux présentés dans le chapitre précédent pour la construction du pushout. Enfin, le placement des attributs se fait lui aussi de la même manière.

### 2.3 Stabilité des classes $\mathcal{M}''$ et $\mathcal{N}$

On retrouve ici la même propriété que pour la construction du pushout.

**Proposition 11.** *Les classes $\mathcal{M}''$ et $\mathcal{N}$ sont stables par passage au pushout-complement.*

### 2.4 Exemple

Un exemple de construction d'un pushout-complement est montré dans la figure V.13. Le schéma V.13(a) représente les données du problème. Si on calcule les ensembles associés à la condition de collage, on trouve :

$$Gluing = \{s_1, s_2\},$$

$$Dangling = \{s_1\},$$

$$Floating = \{s_2\}.$$

La «gluing condition» est donc bien vérifiée sur cet exemple. Le résultat est dessiné sur la figure V.13(b).

## 3 Compatibilité des constuctions catégoricielles avec le typage

### 3.1 Avec le pushout

Soit $TG$ un graphe attribué. Soient $A$, $B$ et $C$ trois graphes attribués et typés par $TG$. On se donne par ailleurs deux morphismes de graphes attribués typés $b : A \to B$ appartenant à la classe $\mathcal{M}'$ et $c : A \to C$ appartenant à la classe $\mathcal{N}$. La question que l'on se pose maintenant est de savoir si le pushout de $b$ et $c$, s'il existe, est aussi typé par le graphe $TG$. En d'autres mots, savoir si on peut construire le pushout dans la catégorie $\mathbf{AttGraph}_{TG}$.

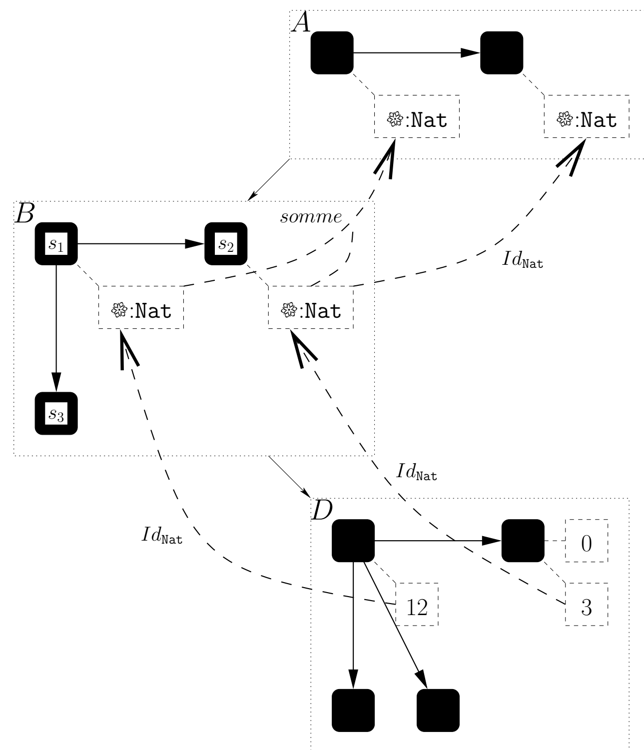
La réponse est négative dans le cas général. En effet, si lors de la construction du pushout a lieu un renommage des espaces d'attributs du graphe $B$ ou du graphe $C$, alors des problèmes peuvent apparaître. La figure V.14 montre un exemple problématique. L'unique sommet du graphe $D$, pushout des morphismes $b$ et $c$, va être en relation avec deux copies du type `Nat` ; ce qui n'est pas prévu par le graphe-type $TG$. Pour éviter cet écueil, il faut donc que le graphe-type autorise dès le départ la possibilité d'avoir deux copies de `Nat` sur le même sommet.

Pour s'assurer *a priori* que ce problème ne va pas apparaître, il faut vérifier la condition suivante sur les morphismes de typage :

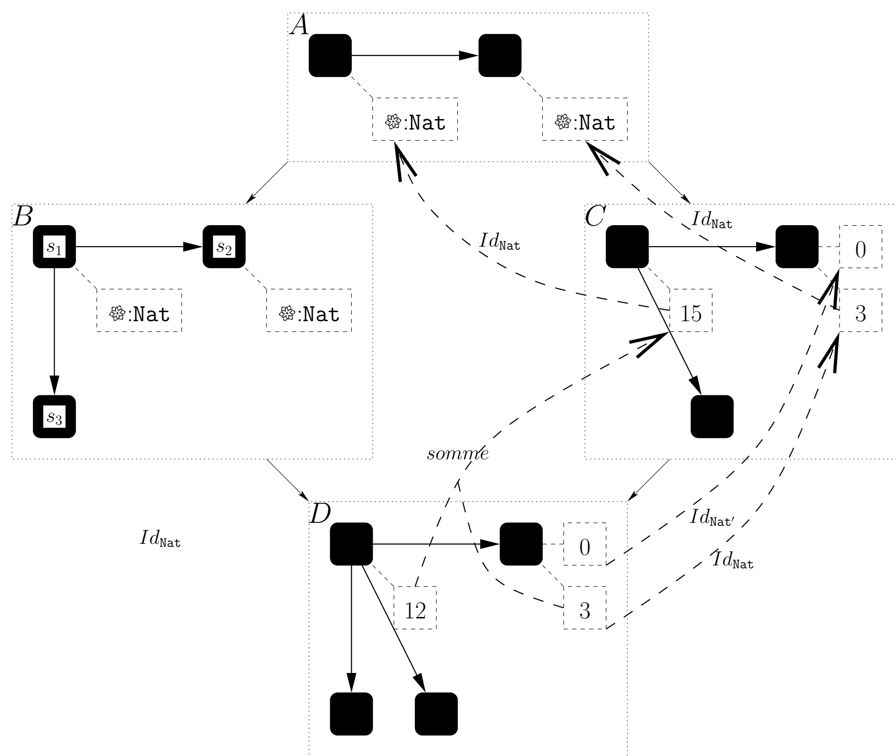
**Théorème 7.** *S'il existe un sommet $s_A$ appartenant au graphe $A$ et un espace d'attributs* `EA` *tels que $(b(s_A), \texttt{EA})\mathcal{R}^{t_B}(t_A(s_A), \texttt{EA})$ et $(c(s_A), \texttt{EA})\mathcal{R}^{t_C}(t_A(s_A), \texttt{EA})$, alors l'espace d'attributs* `EA` *est aussi en relation avec le sommet $s_A$ et par conséquent, on a $(s_A, \texttt{EA})\mathcal{R}^{t_A}(t_A(s_A), \texttt{EA})$.*

*Dans la catégorie $\mathbf{AttGraph}_{TG}$, si cette condition est vérifiée, le pushout existe avec les mêmes restrictions que dans la catégorie* **AttGraph**.

Pour prouver ce théorème, on commence par se placer dans la catégorie **AttGraph** et on calcule le pushout de $b$ et $c$. Le but est donc maintenant de trouver le morphisme de typage entre $D$ et $TG$. Pour cela, comme d'habitude, on commence par se ramener dans la catégorie des graphes simples grâce au foncteur d'oubli $\mathcal{F}$. D'après la construction du pushout (*cf.* section 1.2), on sait




(a) Les données




(b) Le résultat

FIG. V.13 – Exemple de construction de pushout-complement

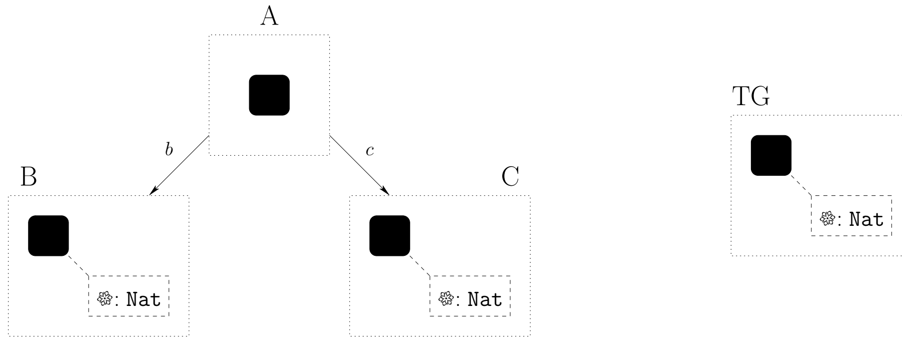



FIG. V.14 – Le pushout ne respecte pas le typage

que $\mathcal{F}(D)$ est le pushout de $\mathcal{F}(b)$ et $\mathcal{F}(c)$. De plus, comme $b$ et $c$ sont des morphismes de graphes attribués et typés par $TG$, on sait que $\mathcal{F}(t_B) \circ \mathcal{F}(b) = \mathcal{F}(t_A) = \mathcal{F}(t_C) \circ \mathcal{F}(c)$. Cette équation nous permet donc d'invoquer la propriété universelle du pushout dans la catégorie des graphes simples. Par conséquent, il est existe un morphisme $\tilde{t}_D$ des graphes entre $\mathcal{F}(D)$ et $\mathcal{F}(TG)$ qui vérifie

$$\mathcal{F}(t_B) = \tilde{t}_D \circ \mathcal{F}(c') \qquad \text{et} \qquad \mathcal{F}(t_C) = \tilde{t}_D \circ \mathcal{F}(b'). \tag{V.1}$$

Ce morphisme $\tilde{t}_D$ va nous servir de base pour construire le morphisme de typage de $D$. En effet, les équations précédentes et la condition imposée dans le théorème sur les morphismes de typage nous assurent que la condition de la définition 40 est bien satisfaite par le morphisme structurelle $\tilde{t}_D$ car comme les espaces d'attributs en relation avec un sommet $s_D$ du graphe $D$ ne peuvent provenir que des espaces d'attributs en relation avec les préimages de $s_D$ par $b'$ et $c'$, on est sur que le nœud $\tilde{t}_D(s_D)$ est relié aux mêmes espaces d'attributs que $s_D$.

Pour la forêt et les fonctions de calcul du morphisme de typage $t_D$ du graphe $D$, la définition de la classe $\mathcal{T}$ ne laisse qu'un choix possible : des arbres simples limités aux relations nécessaires et portant la fonction identité. Il reste donc à s'assurer la compatibilité des attributs afin de vérifier que la définition de $t_D$ est licite. Cette vérification est simple à prouver. En effet, en étudiant les schémas de construction du pushout (voir figure V.2 à V.5), on s'aperçoit qu'un attribut du graphe $D$ partage forcément la même valeur que son attribut correspondant dans le graphe $B$ ou $C$. Par conséquent, si la valeur d'un attribut est imposée par le graphe-type $TG$, comme $B$ et $C$ sont typés par $TG$, alors l'attribut correspondant dans le graphe $D$ aura la bonne valeur.

On a donc bien construit un morphisme de typage pour le graphe $D$ par rapport au graphe $TG$. Enfin, les égalités (V.1) nous assurent que les morphismes $b'$ et $c'$ sont bien des morphismes de graphes attribués typés. Le pushout est donc constructible dans la catégorie $\mathbf{AttGraph}_{TG}$.

## 3.2 Avec le pushout-complement

Le cas du pushout-complement est plus simple que le cas du pushout car comme nous l'avons vu dans la section 2.2, aucun renommage n'a lieu durant la construction du pushout-complement. Le problème soulevé dans la section précédente ne peut donc pas apparaître ici.

**Théorème 8.** *Dans la catégorie* $\mathbf{AttGraph}_{TG}$*, le pushout-complement existe avec les mêmes restrictions que dans la catégorie* **AttGraph***.*

Pour prouver ce théorème, on suit la même démarche que pour la preuve précédente. La principale différence vient de la construction du morphisme structurel entre $C$ et $TG$. On n'utilise plus ici la propriété universelle de pushout dans la catégorie des graphes simples, mais on calcule seulement la composition $\mathcal{F}(t_D) \circ \mathcal{F}(b')$. Enfin, pour vérifier que les attributs ont les bonnes valeurs, il suffit de remarquer qu'un attribut du graphe $C$ possède la même valeur que l'attribut correspondant dans le graphe $D$ ou dans le graphe $A$.

# Chapitre VI

# Système de transformations de graphes dans l'approche DPoPb

Après avoir décrit la catégorie **AttGraph** ainsi que les opérations catégoricielles qui lui sont associées, nous définissons maintenant précisément les règles de transformation de graphes et le déroulement de leur application pour construire des systèmes de transformations de graphes.

## 1 Règles de transformations

**Définition 56.** *Une règle de transformation $p$ (ou production) dans l'approche* DPoPB *est donnée par 3 graphes attribués $K$, $L$ et $R$ ainsi que deux morphismes $l : K \to L$ appartenant à la classe $\mathcal{M}''$ et $r : K \to R$ appartenant à la classe $\mathcal{M}'$. La graphe $L$ (respectivement $R$) s'appelle la partie gauche (respectivement droite) de la règle et $K$ est nommé le graphe de collage.*

**Définition 57.** *Soient $p = (L \leftarrow K \rightarrow R)$ une production et $G$ un graphe attribué. Une correspondance associée à $p$ et $G$ est un morphisme $m$ appartenant à la classe $\mathcal{N}$.*

L'application d'une règle $p$ à un graphe $G$ suivant une correspondance $m$ se fait alors en suivant exactement le même schéma que dans l'approche «double pushout» (*cf.* section 2) : si les morphismes $l$ et $m$ satisfont la «gluing condition», alors on peut calculer le pushout pour construire le graphe de contexte $D$ ainsi qu'un morphisme $m^*$ entre $K$ et $D$ appartenant à la classe $\mathcal{N}$. On peut alors terminer le calcul en construisant le pushout $H$ de $r$ et $m^*$. Ce dernier graphe obtenu est alors le transformé direct de $G$ suivant la règle $p$, on note alors $G \Longrightarrow_p H$. Une suite de transformations directes $G_0 \Longrightarrow G_1 \Longrightarrow \ldots \Longrightarrow G_n$ sera parfois notée $G_0 \Longrightarrow^* G_n$.

**Définition 58.** *Un système de transformations de graphes attribués est donné par un ensemble de règles de transformation. Une grammaire de graphes attribués est définie par un système de transformations et un graphe de départ.*

**Remarque 19.** Si l'on souhaite travailler avec des règles réversibles, il faut bien entendu imposer quelques restrictions supplémentaires afin de rendre la définition des règles de transformation symétrique. Plus précisément, il faut supprimer les parties qui peuvent ne pas être réversibles : les calculs complexes et la fusion d'éléments du graphes. Pour cela, on demande à ce que les morphismes $l$ et $r$ appartiennent tous deux à l'intersection des classes $\mathcal{M}'$ et $\mathcal{M}''$.

## 2 Règles composées

Pour décrire certaines règles, l'application d'une seule règle suivant la définition ci-dessus ne suffit pas ; étudions par exemple la transformation suivante : à partir d'un sommet auquel sont attachés deux entiers naturels, on veut créer un second sommet relié au premier avec comme attributs la somme des deux entiers de départ (*cf.* figure VI.1). Une telle transformation ne peut pas s'effectuer en une seule étape. En effet, avant de calculer effectivement la somme $a + b$, il faut

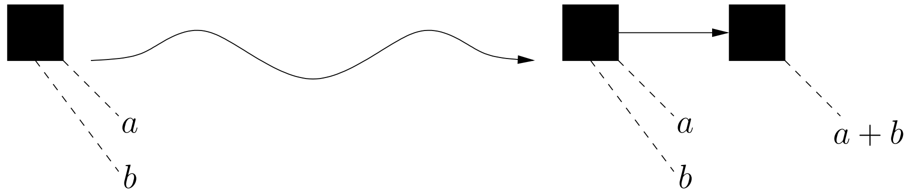



Fig. VI.1 – Une transformation simple

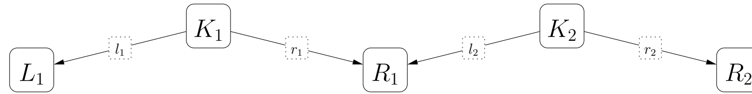



Fig. VI.2 – Règle composée

commencer par créer un emplacement pour recevoir ce résultat. Cette création ne peut se faire que lors du second pushout d'une règle. Par contre, comme les fonctions de calcul de la partie droite d'une règle sont forcément bijectives, on ne peut pas faire l'addition en même temps. Il faut donc définir une seconde règle pour l'addition (qui aura lieu durant le calcul du pushout-complement) et imposer que les deux règles s'appliquent toujours de manière successive. On arrive alors à la définition de règle composée.

**Définition 59.** *Soient* $p_1 = (L_1 \leftarrow K_1 \rightarrow R_1)$ *et* $p = (L_2 \leftarrow K_2 \rightarrow R_2)$ *deux règles telles que* $R_1 = L_2$*. Il est alors possible de «concaténer» les deux règles* $p_1$ *et* $p_2$ *pour obtenir une règle composée (voir figure VI.2). L'application d'une telle règle se fait de manière naturelle avec dans l'ordre un calcul de pushout-complement, un calcul de pushout, encore un pushout-complement puis un dernier pushout.*

Typiquement, ce genre de règles nous servira lors de l'ajout d'un attribut qui n'est pas la simple copie d'un autre attribut (voir l'exemple presenté dans la section 5.2). La première partie permettra alors de créer l'espace d'attributs qui nous intéresse (lors du pushout) et lors de la seconde partie de l'application, le calcul de la valeur pourra être effectué (lors du pushout-complement).

**Remarque 20.** Dans l'approche HLR (*cf.* [32]), on ne recontre pas de règles composées car dès que l'on a deux « Double Pushout », on peut calculer le pullback des morphismes $r_1$ et $l_2$ pour finalement n'obtenir qu'une seule règle simple (*cf.* figure VI.3). Pour autant, cette possibilité de grouper les règles ne permet pas d'augmenter l'expressivité du système HLR : les fonctions complexes ne sont toujours pas exprimable malgré cette construction. Par exemple, il est impossible de construire une règle calculant la factorielle d'un nombre entier : la factorielle, qui est définie récursivement, n'est pas une fonction exprimable dans le formalisme des Σ-algèbres ; il est donc nécessaire de définir un ensemble[1] de règles pour arriver au résultat. Comme le nombre d'applications de ces règles va varier suivant l'argument de départ, on ne peut pas utiliser la construction du pullback pour simplifier dans le cas général le calcul de la factorielle.

[1] *cf.* `http://tfs.cs.tu-berlin.de/agg/examples_V122/Actor/`

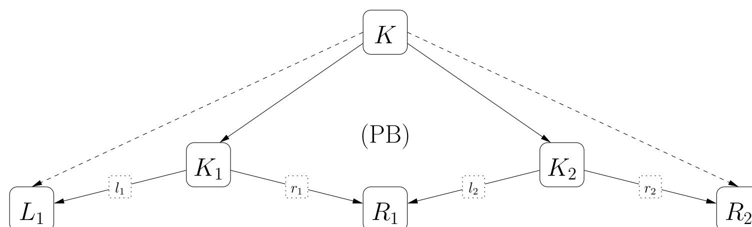



Fig. VI.3 – Compiler deux règles en une seule dans l'approche HLR

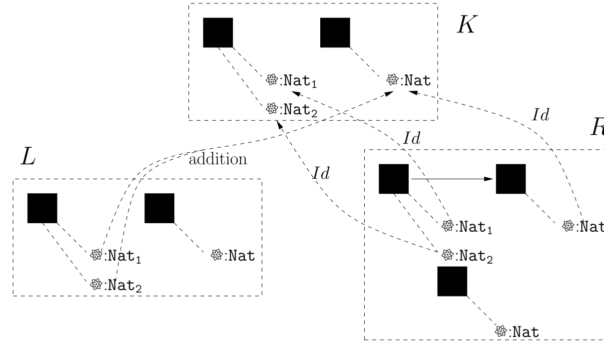



FIG. VI.4 – Additionner deux attributs en une seule règle avec un sommet supplémentaire

Techniquement, la construction de la figure VI.3 n'est pas possible dans l'approche DPoPB car, d'une part, le pullback (*cf.* chapitre 3.1) n'est pas toujours constructible et d'autre part, même s'il est constructible, il ne comportera pas forcément assez d'information pour coder la règle : en reprenant l'exemple introductif, le pullback ne comporte qu'un sommet avec comme attributs les deux entiers de départ et l'on perd donc le calcul de l'addition. Cette complication est le prix à payer pour la plus grande expressivité de notre approche (avec les types inductifs, les fonctions complexes telles que la factorielle sont exprimables et on peut donc l'utiliser comme fonction de calcul dans nos morphismes) et l'économie de sommets faites dans nos graphes.

En effet, ce problème n'apparait pas dans l'approche HLR car tous les attributs possibles sont présents dans tous les graphes (*cf.* chapitre III). Ainsi, aucun attribut n'est supprimé lors du calcul du pushout. Pour traduire les règles HLR dans notre approche, on peut imaginer rajouter aux graphes considérés un sommet particulier avec comme attributs un joker du bon type ; ainsi, on aura toujours à disposition un attribut prêt à recevoir le résultat d'un calcul lors de l'application de la partie gauche de la règle et la partie droite de la règle va permettre de déplacer cet attribut à l'endroit voulu dans le graphe et de recréer un nouveau joker. La règle codant l'addition avec cette astuce est représenté figure VI.4. On peut ainsi exprimer n'importe quelle règle de l'approche HLR dans notre formalisme.

## 3 Comportement vis-à-vis du typage

*Transformation en conformité vis-à-vis d'un méta-modèle*

Les résultats obtenus dans la partie 3 nous affirment que les transformations de graphes attribués conservent bien le typage des graphes. Plus précisément, soit $p = (L \leftarrow K \rightarrow R)$ une règle de transformation où les trois graphes sont typés par un graphe $TG$ et les morphismes $l$ et $r$ sont des morphismes de graphes attribués typés, soit $G$ un graphe typé par $TG$ et soit $m$ une correspondance entre $G$ et $p$ telle que $m$ soit un morphisme de graphes attribués typés. Si maintenant les morphismes de graphes attribués et typés satisfont la condition déjà évoquée dans la section 3.1, alors on est sûr que la transformation de $G$ suivant la règle $p$ sera elle aussi typée par le graphe $TG$.

*Transformation exogène*

La question qui apparaît alors est de savoir s'il est possible de transformer un graphe $G$ typé par un graphe $TG$ en un graphe $H$ typé par un autre graphe $TG'$. Dans l'optique de transformation de modèles décrite par le MDA, cela correspond à transformer un modèle $M_1$ conforme à un métamodèle $MM_1$ vers un modèle $M_2$ conforme à un métamodèle $MM_2$. Afin de réaliser une telle transformation, l'idée appliqué est de trouver une troisième graphe-type $TTG$ pour typer simultanément les graphes $TG$ et $TG'$. Généralement, on prendra pour $TTG$ l'union disjointe de $TG$ et $TG'$ avec des arêtes et des sommets additionnels représentant les correspondances entre des concepts similaires des deux méta-modèles (*cf.* [67]). Par transitivité, le graphe $G$ va, lui aussi, être un instance de $TTG$, ainsi que toutes les règles de transformation. On peut donc appliquer la

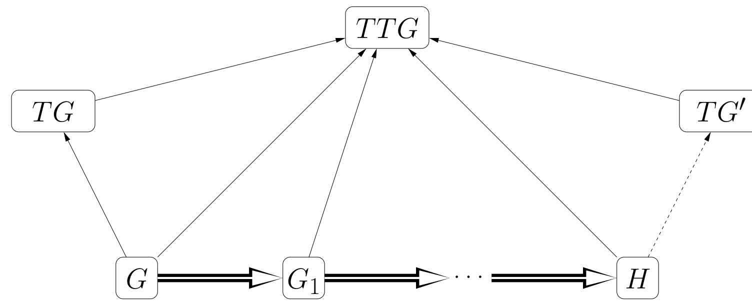



FIG. VI.5 – D'un graphe-type à un autre

construction précédente (voir figure VI.5). Enfin, si le morphisme de typage $t'$ entre $TG'$ et $TTG$ est une injection (sur la structure) et si l'image du morphisme de typage de $H$ par $TTG$ est inclus dans l'image de $t'$ (on ne regarde ici que la structure), alors on peut facilement construire un typage de $H$ par $TG'$.

# 4 Conditions d'application

Il est souvent utile de pouvoir contrôler plus finement l'application d'une règle de transformation. L'idée est de pouvoir utiliser le contexte autour du sous-graphe à récrire pour autoriser ou non l'execution d'une règle : par exemple, on peut vouloir interdire cette application si un certain sommet est présent dans le graphe source $G$ ou encore calculer la transformation si et seulement si deux attributs du graphe $G$ sont égaux. Pour cela, on utilise les conditions d'application (*cf.* [39]).

Pour une règle $p = (L \leftarrow K \rightarrow R)$, une condition d'application est donnée par un graphe supplémentaire $N$ ainsi qu'un morphisme $n$ entre les graphes $L$ et $N$. Une règle peut être accompagnée de plusieurs conditions d'application et l'application de cette règle à un graphe $G$ ne sera possible que si $G$ satisfait chacune de ces conditions. Plus précisément, il existe deux types de conditions d'application que l'on définit maintenant.

## 4.1 Type I

Ce premier type de condition d'application est utile pour vérifier la présence ou non d'un élément (nœud, arête ou encore espace d'attributs) dans le graphe source $G$. Il peut aussi servir pour interdire l'application d'une règle si un attribut est égal à une valeur donnée.

**Définition 60.** *Si $N$ est une condition d'application positive de type I, alors il doit exister un morphisme $n' : N \rightarrow G$ tel que $n' \circ n = m$ pour appliquer la production $p$ au graphe $G$.*

*Si $N$ est une condition d'application négative de type I, un tel morphisme ne doit pas exister afin de pouvoir appliquer la production $p$ au graphe $G$.*

## 4.2 Type II

Le second type de condition d'application est utile pour vérifier des conditions entre les valeurs des attributs. Ce type de condition est nécessaire car, comme les attributs ne possèdent pas de nom dans notre approche, c'est le seul moyen de comparer les valeurs des attributs. Il ne peut pas être utilisé pour des conditions sur la structure.

**Définition 61.** *Si $N$ est une condition d'application positive de type II, alors le pushout de $m$ et $n$ doit être constructible pour appliquer la production $p$ au graphe $G$.*

*Si $N$ est une condition d'application négative de type II, alors ce pushout ne doit pas exister afin de pouvoir appliquer la production $p$ au graphe $G$.*

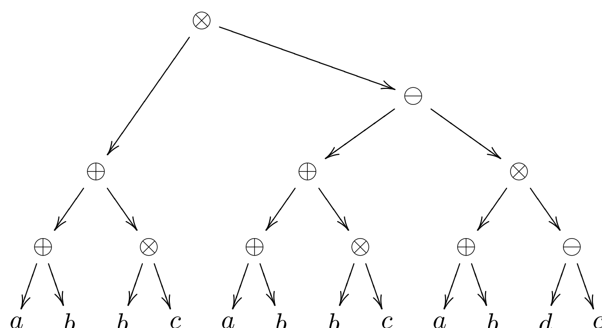



FIG. VI.6 – L'arbre syntaxique représentant l'expression $(a+b+b*c)*(((a+b+b*c))-((a+b)*(d-c)))$

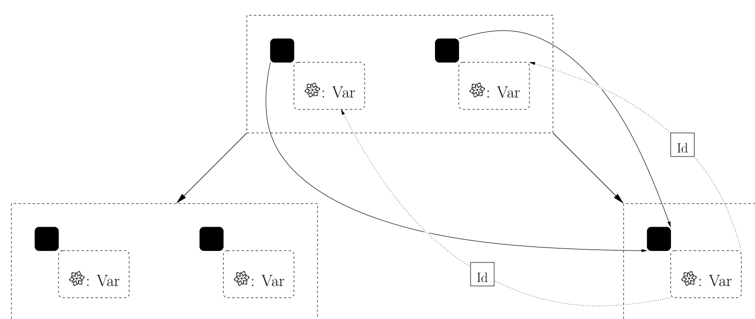



FIG. VI.7 – Règle 1 : fusion des variables

# 5 Exemples

## 5.1 Des arbres syntaxiques aux graphes directs acycliques

Le but de cet exemple est de montrer un système permettant de passer d'un arbre représentant une expression algébrique à un graphe direct acyclique (abrégé en DAG pour «Direct Acyclic Graph») qui simplifie la représentation de cette expression. Dans la pratique, une telle transformation est utile pour limiter la mémoire qu'occupe une expression et aussi permettre de gagner du temps lors de son évaluation. Les compilateurs mettent fréquemment en place une telle simplification afin d'optimiser simplement le code [1]. Pour arriver à ce but, le principe de la transformation consiste à rassembler les sous-expressions de l'expression de départ qui sont identiques.

Les arbres syntaxiques sont représentés par des graphes : sur chacune des feuilles de l'arbre possède comme attributs une variable ou une constante et à chaque nœud de l'arbre, on va trouver une opération. Pour être complet, il faut rajouter la possibilité de numéroter les arcs sortant d'un nœud portant une opération afin de différencier les attributs dans le cas où l'opération n'est pas commutative. Pour une opération commutative, on peut choisir la convention de numéroter tous les arcs avec des zéros. Un exemple d'un tel arbre syntaxique est donné figure VI.6 ; pour alléger le schéma, les zéros ne sont pas représentés. Le système de transformation va contenir deux règles. La première va s'occuper de fusionner les feuilles de l'arbre lorsque les variables ou les constantes sont identiques et la seconde va rassembler deux opérations identiques dès qu'elles partagent les mêmes arguments. Ces deux règles sont respectivement représentées figures VI.7 et VI.8 (sur ce deuxième schéma, les fonctions de calcul, qui sont toutes égales à l'identité, ne sont pas représentées afin de ne pas surcharger le graphe).

Dans ces deux règles, on peut remarquer que les parties structurelles des morphismes définissant les parties droites des productions sont non injectives. Cela permet de fusionner les sommets lors de l'application de la transformation sans se préoccuper des flèches incidentes à ces sommets et qui ne sont pas représentées dans la production. Ces flèches seront alors automatiquement incidentes

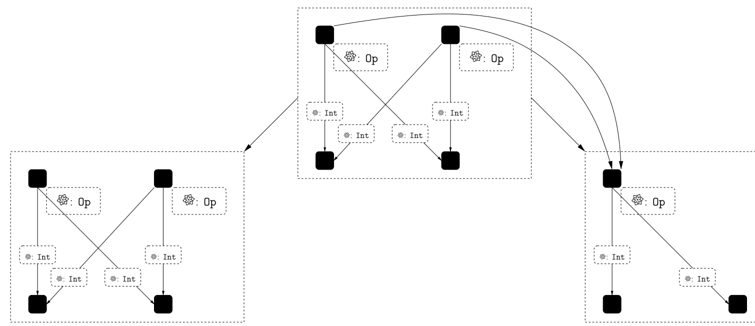


Fig. VI.8 – Règle 2 : fusion des opérations

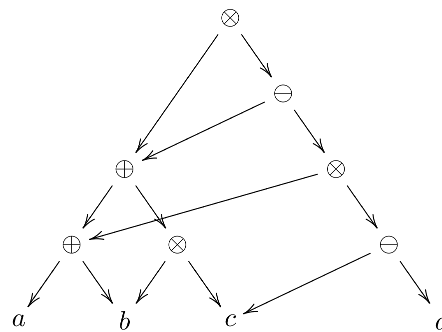



Fig. VI.9 – Le DAG correspondant à l'arbre de la figure

au sommet fusionné et ce, quelque soit leur nombre. Par ailleurs, afin de contrôler l'égalité des attributs avant l'application d'une règle, on utilise ici des conditions d'application négatives de type II : ces NAC sont dans l'exemple présenté ici égales aux parties droites des règles. Le résultat de la transformation est représenté figure VI.9.

## 5.2 Des diagrammes de classes UML aux bases de données relationnelles

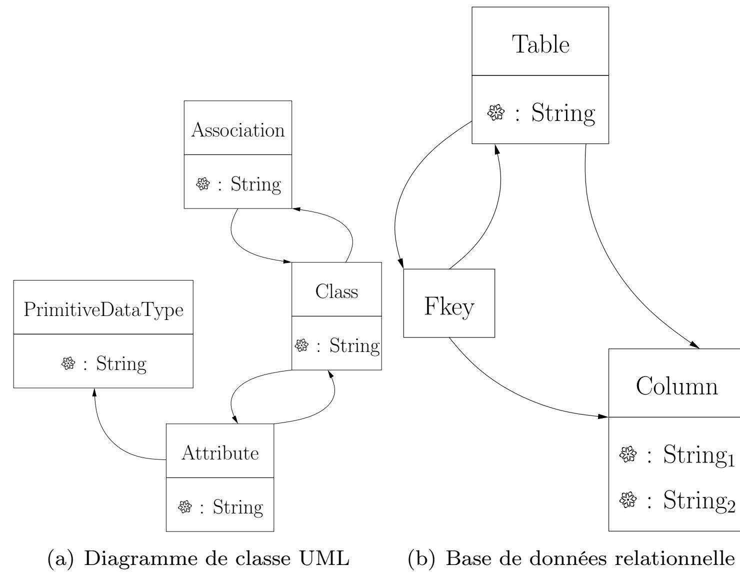
On s'intéresse maintenant à un exemple classique de transformations de modèles : comment transformer un diagramme de classes UML en une base de données relationnelle. Le cahier des charges est présenté dans [8]. On ne présente ici qu'une approche simplifiée du problème où les clefs primaires, l'héritage et les classes non persistantes ne sont pas pris en compte. La solution proposée ici reprend les idées décrites dans la section 3 de [33].

Les diagrammes de classes et les bases de données relationnelles sont décrites par des méta-modèles : les graphes-types correspondants sont représentés figure VI.10. Pour réaliser cette transformation exogène, on va utiliser un graphe-type plus général constitué de l'union disjointe des deux graphes-types et de quelques éléments additionnels pour faire le lien entre les deux modèles ; comme indiqué dans la section 3. Ce graphe-type général est dessiné figure VI.11.

Dans cette grammaire de graphes, quatre règles différentes sont présentes :

– pour chaque classe, une table est créée ;
– pour chaque attribut primaire, une colonne est ajoutée ;
– pour chaque attribut avec une classe comme type d'attribut, une colonne et une clef sont créées ;
– pour chaque association, une colonne et une clef sont créées.

Ces quatre règles sont représentées figure VI.12 à VI.15. Pour chacune d'entre elles, il faut signaler qu'une condition d'application de type I égale à la partie droite de la règle est présente afin d'empêcher l'exécution d'une même règle au même endroit (ce qui conduirait, par exemple, à créer plusieurs tables pour une même classe). Enfin, on peut remarquer que les règles 3 et 4 sont des




(a) Diagramme de classe UML (b) Base de données relationnelle

FIG. VI.10 – Graphes-types pour les deux méta-modèles

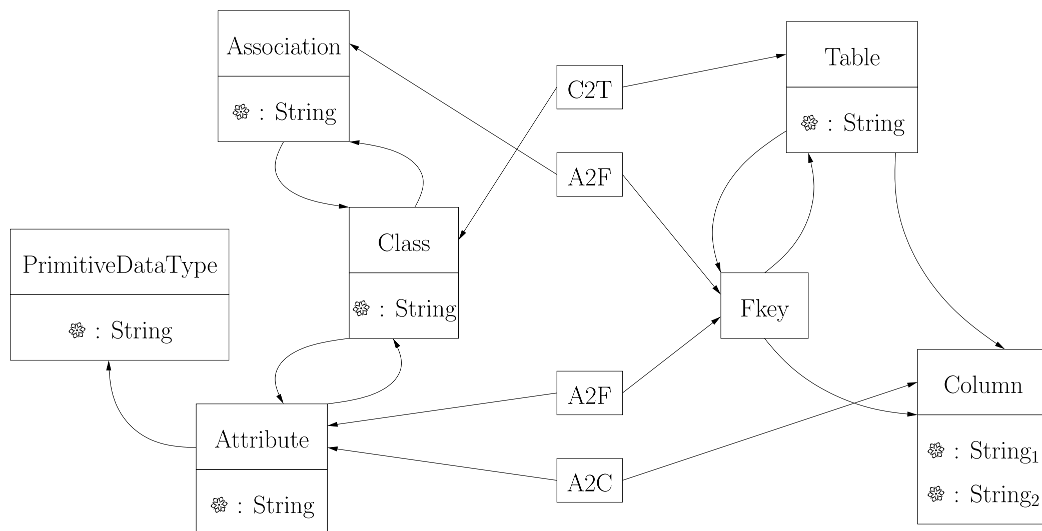



FIG. VI.11 – Type-graph for the transformation

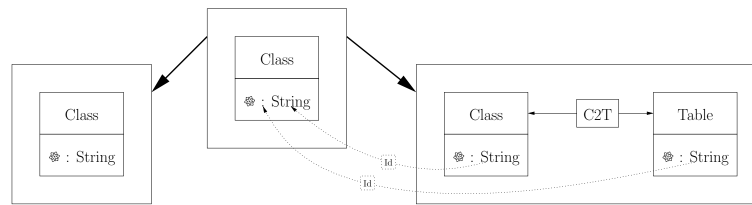



Fig. VI.12 – Règle 1

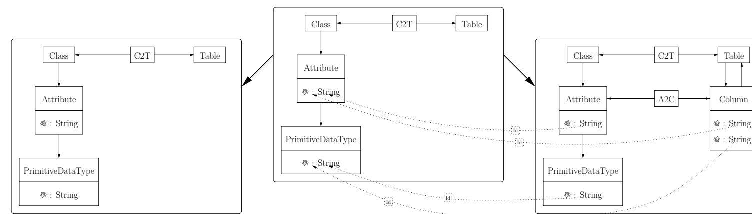



Fig. VI.13 – Règle 2

règles composées : la première partie sert à ajouter les éléments structurels et les espaces d'attributs et au cours de la seconde phase, la valeur d'un de ces espaces d'attributs est calculée : on utilise pour cela une fonction de calcul non triviale afin de concaténer deux chaînes de caractères.

## 5.3 Des diagrammes d'activité UML vers les réseaux de Petri

On présente ici un second exemple se rapportant à l'approche MDA ; il s'agit du problème de traduction des diagrammes d'activité UML vers les réseaux de Petri (on ne considère ici les réseaux avec au plus un jeton par place). Cet exemple est inspiré par les articles [23, 22].

Dans cet exemple, on utilise encore la possiblité de typer les graphes en traduisant les méta-modèles des diagrammes d'activité et des réseaux de Petri. Ces deux graphes sont représentés figure VI.16. Durant la transformation, on utilise le graphe de la figure VI.17 comme graphe de typage : celui-ci est constitué des deux graphes-types auxquels on rajoute quelques éléments pour relier les concepts similaires ; par exemple, une transition dans un reseau de Petri et un pas dans le diagramme d'activité. Quelques règles composant le système de transformation sont représentés figure

## 5.4 Optimisation

On cherche à optimiser un programme avec une simple optimisation (*cf.* [1]) : remplacer une multiplication dans une boucle dépendant de la variable de contrôle de la boucle par une addition (voir figure VI.19). Une addition étant plus rapide à calculer qu'une multiplication pour le processeur, l'execution du programme est accelérée.

**Remarque 21.** Quand on sort de la boucle, selon que l'on soit dans le cas optimisé ou non, la variable $r$ n'a pas forcément la même valeur. Ce n'est pas un vrai problème car les compilateurs savent traiter cette différence.

Pour simplifier l'exemple, on suppose ici qu'il n'y a pas de dépendance entre les variables. En réalité, cette question n'est pas triviale et doit être traitée auparavant en analysant le graphe de flot de contrôle du programme ([1]).

On représente le code source du programme par un graphe de contrôle (voir figure VI.20). Chaque sommet représente une étape caractérisée par un quadruplet. Afin de pouvoir récupérer de l'information à partir de chaque élément du quadruplet, on préfère séparer chaque élément et ainsi placer quatre attributs sur chaque sommet.

Par ailleurs, afin de juste montrer les règles de transformation intéressantes, on suppose qu'on sait reconnaître les multiplications qu'il va être utile de modifier. Afin de traîter ce problème, on

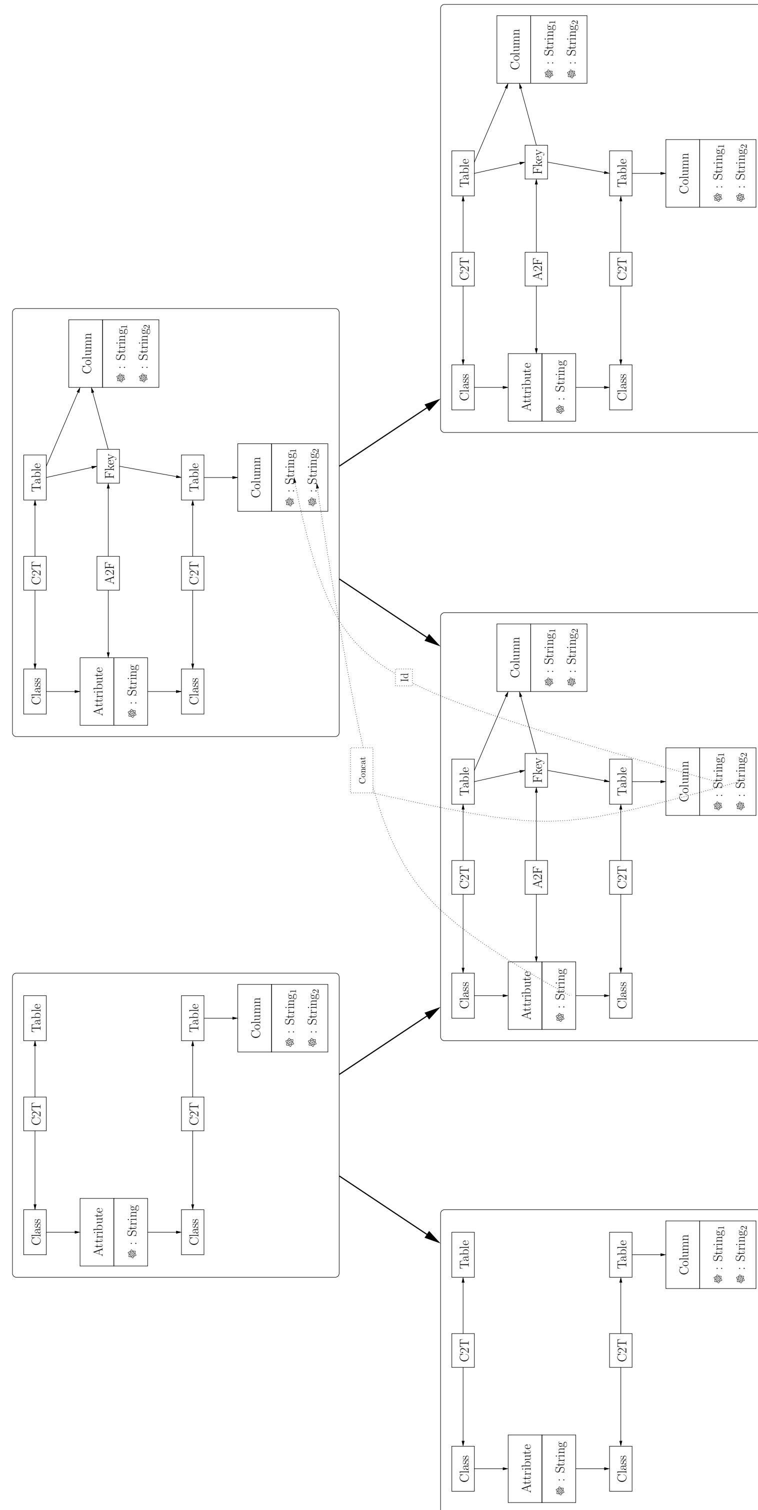



Fig. VI.14 – Règle 3

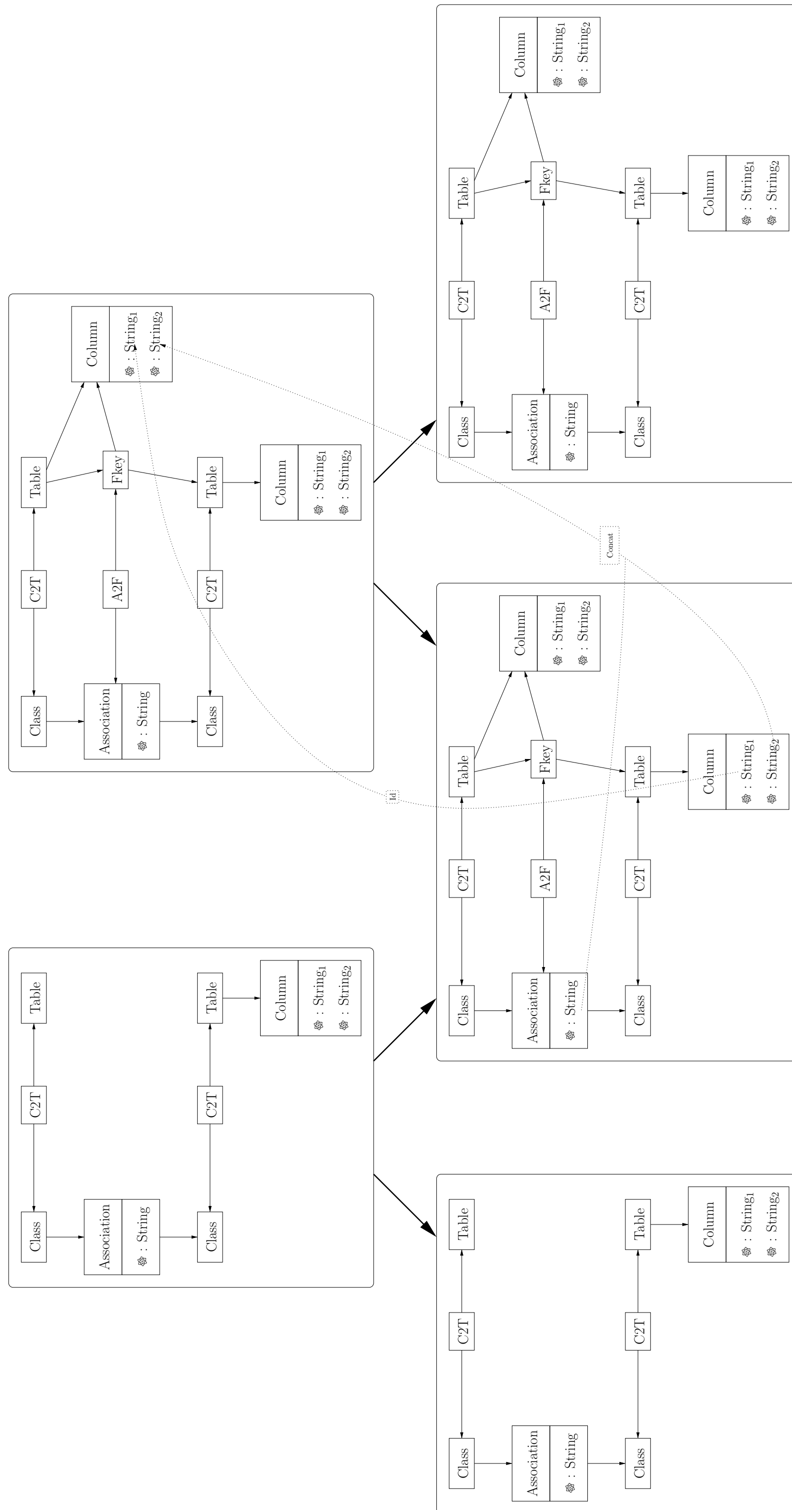
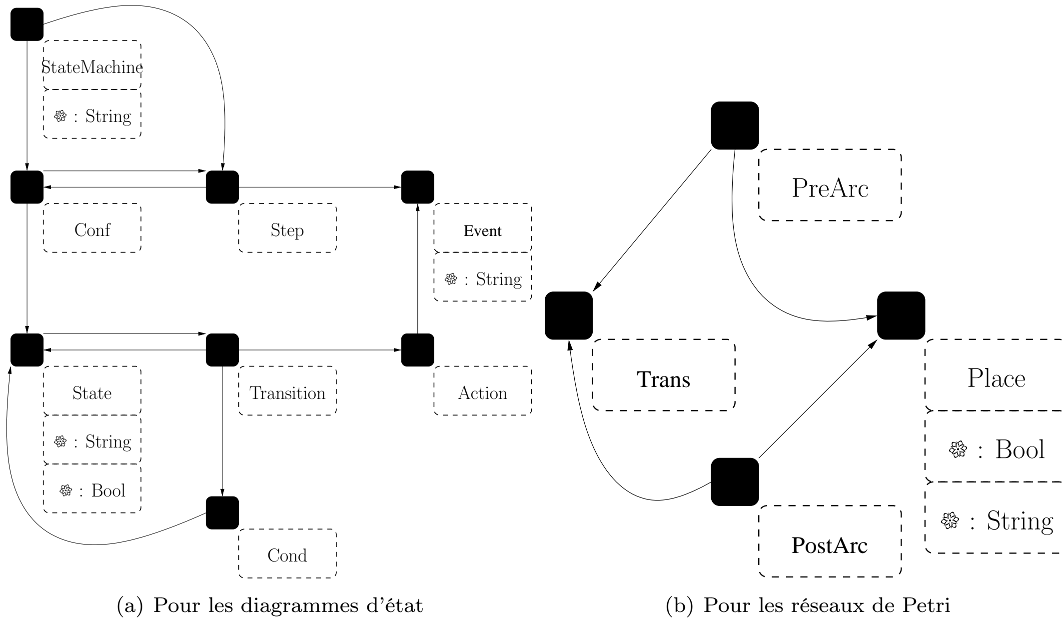



Fig. VI.15 – Règle 4




(a) Pour les diagrammes d'état (b) Pour les réseaux de Petri

Fig. VI.16 – Graphes-types

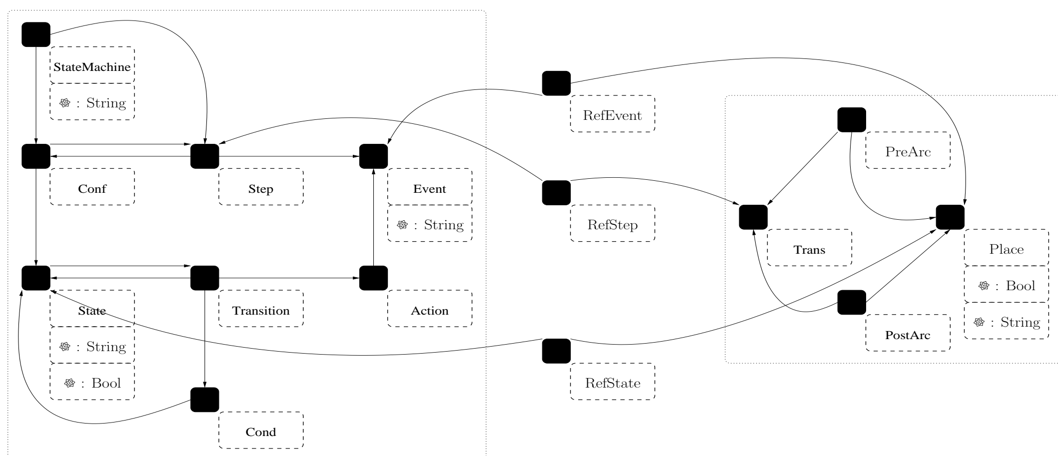
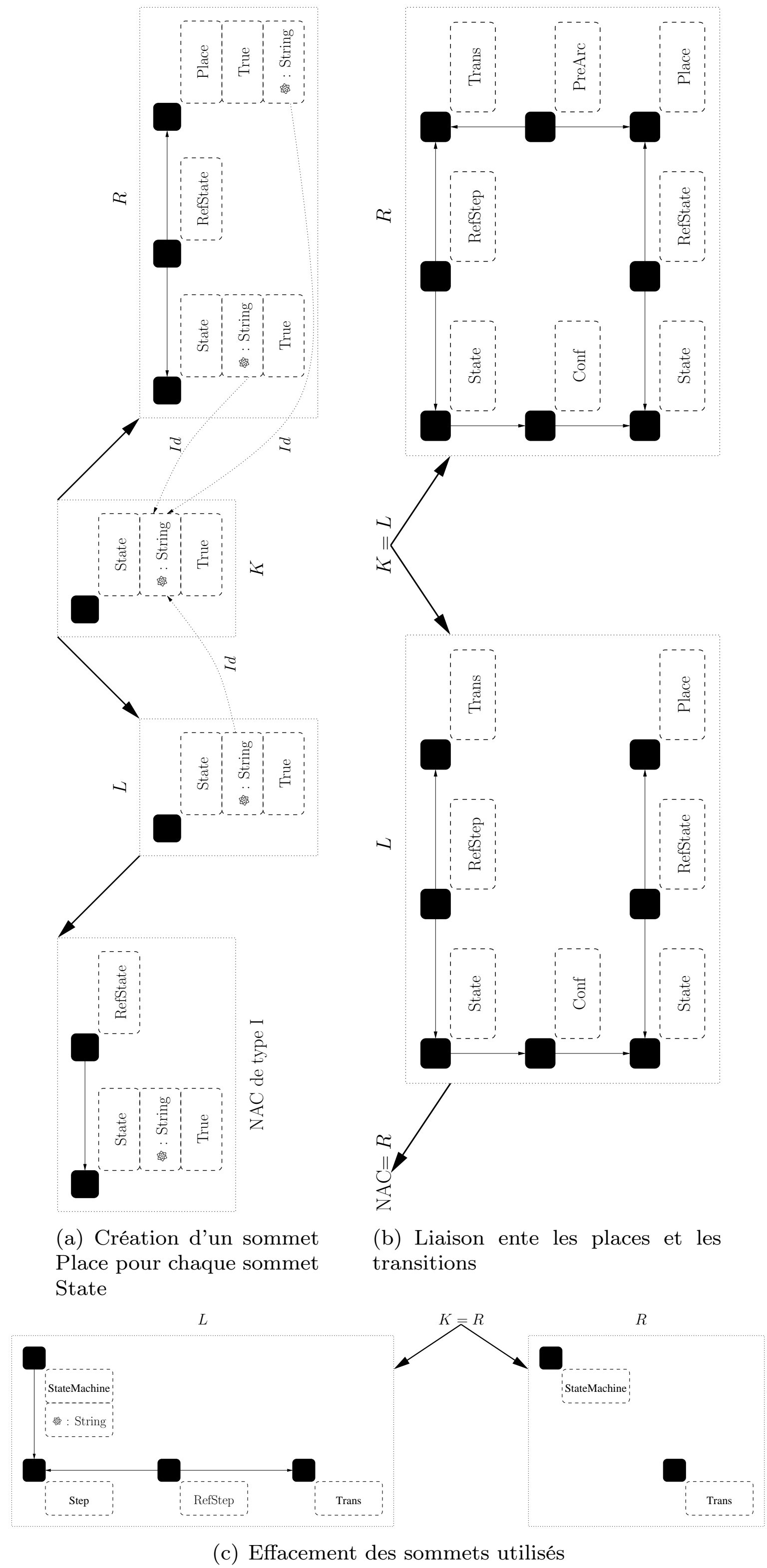



Fig. VI.17 – Graphe-type pour la transformation




(a) Création d'un sommet Place pour chaque sommet State

(b) Liaison ente les places et les transitions



(c) Effacement des sommets utilisés

FIG. VI.18 – Trois règles du système de récriture

```
init : i ← valeur initiale
test : if i >max goto over
...
r ←k*i
...
incr : i ← i+pas
goto test
over ...
```

devient

```
init : i ← valeur initiale
r ←k*i
test : if i >max goto over
...
...
incr : i ← i+pas, r ← r+k*pas
goto test
over ...
```

FIG. VI.19 – L'optimisation d'une boucle

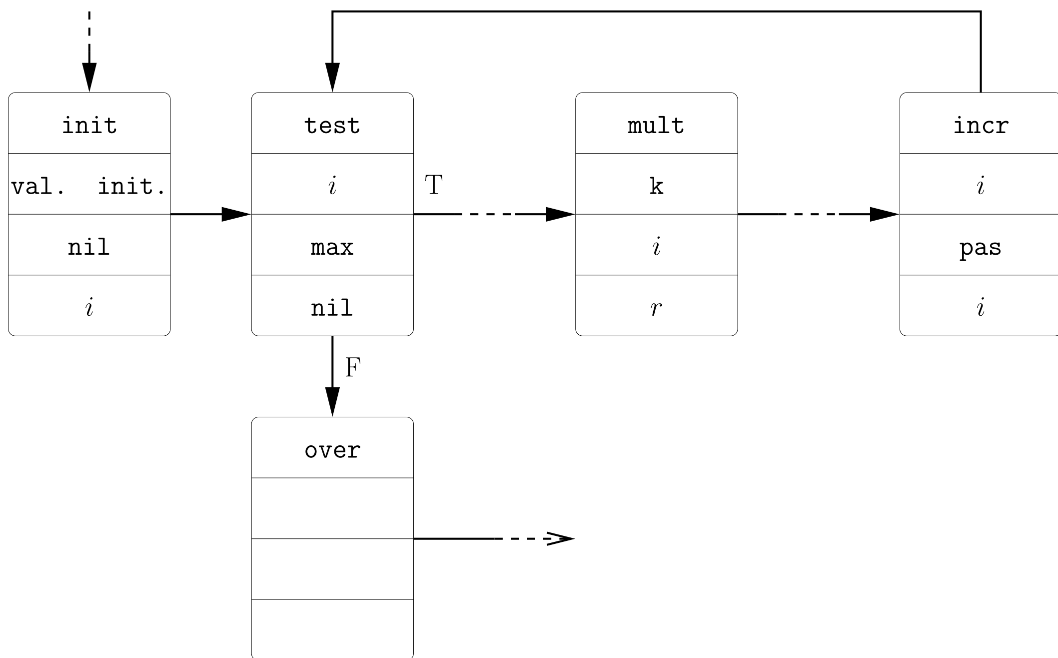



FIG. VI.20 – Graphe de contrôle

peut imaginer faire descendre dans les boucles les noms des variables pour repérer quelle multiplication satisfait à nos critères. Ainsi, les sommets correspondants pourront porter un attribut supplémentaire repérable par les règles de transformation.

On propose une solution avec deux règles de transformation pour ce problème :
- la première règle va permettre de remonter la multiplication dans la boucle en elle-même jusqu'à arriver sur le test de boucle ;
- la seconde va transformer la multiplication en une addition et faire sortir une phase d'initialisation de la boucle.

#### *5.4.19 Règle n° 1*

Le principe de cette règle est d'intervertir deux sommets. En pratique, on modifie les flèches touchant ces sommets (voir figure VI.21). Comme ces deux sommets font partie d'une "chaîne" dans le graphe, les 3 graphes de la règle comporte 4 sommets afin de ne pas voir apparaître d'arc pendouillant pendant le processus de transformation.

Une condition d'application négative de type 2 se charge enfin d'empêcher d'appliquer cette règle si le sommet supérieur est le test de la boucle qui correspond à la multiplication.

#### *5.4.20 Règle n° 2*

Dans cette seconde règle qui modifie les attributs du sommet de multiplication et rajoute un sommet avant le test de la boucle (voir figure VI.22), on ajoute une condition d'application positive chargée de vérifier que la variable de contrôle de la boucle correspond bien à une variable de la multiplication (non représentée ici).

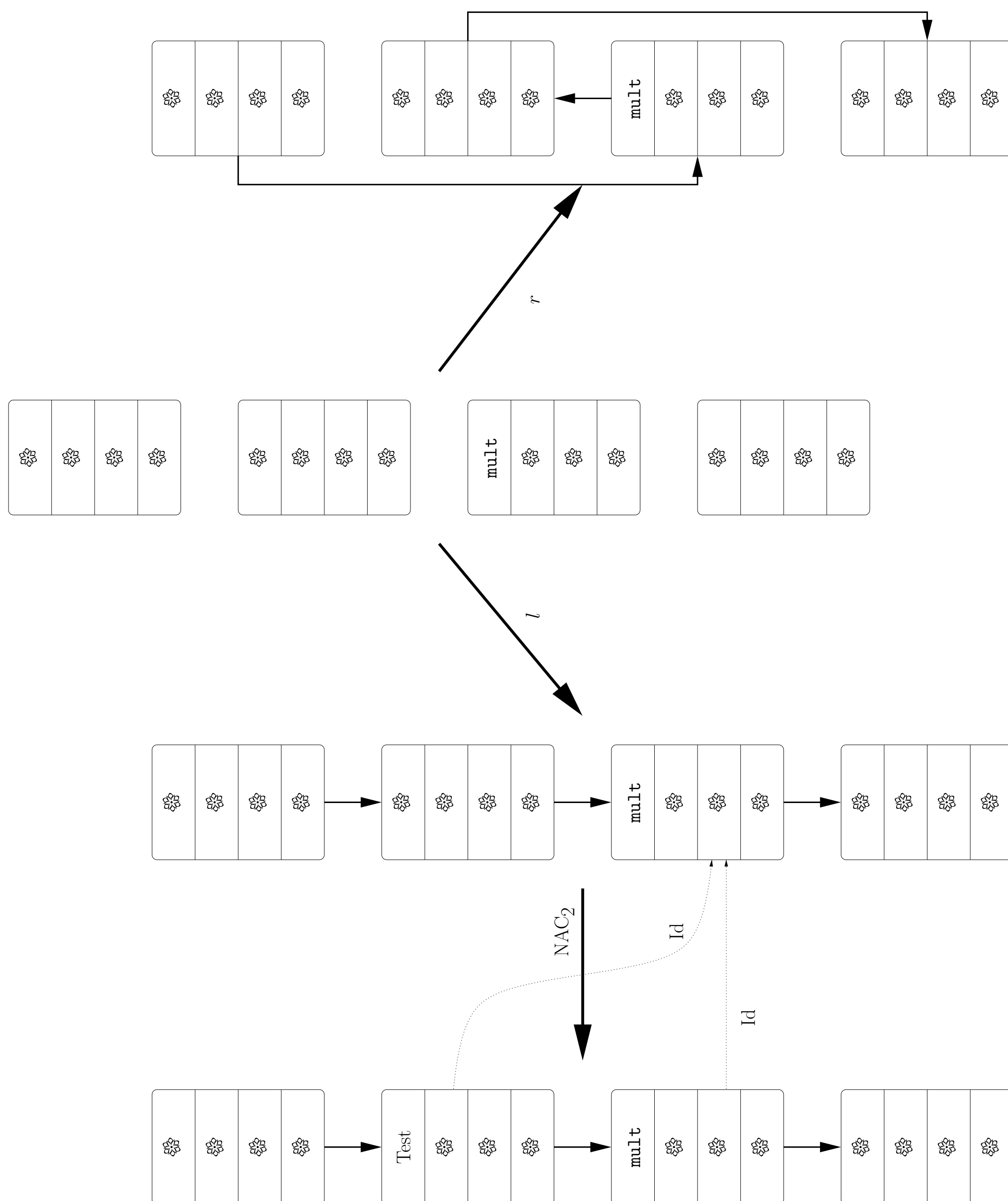



FIG. VI.21 – Première règle

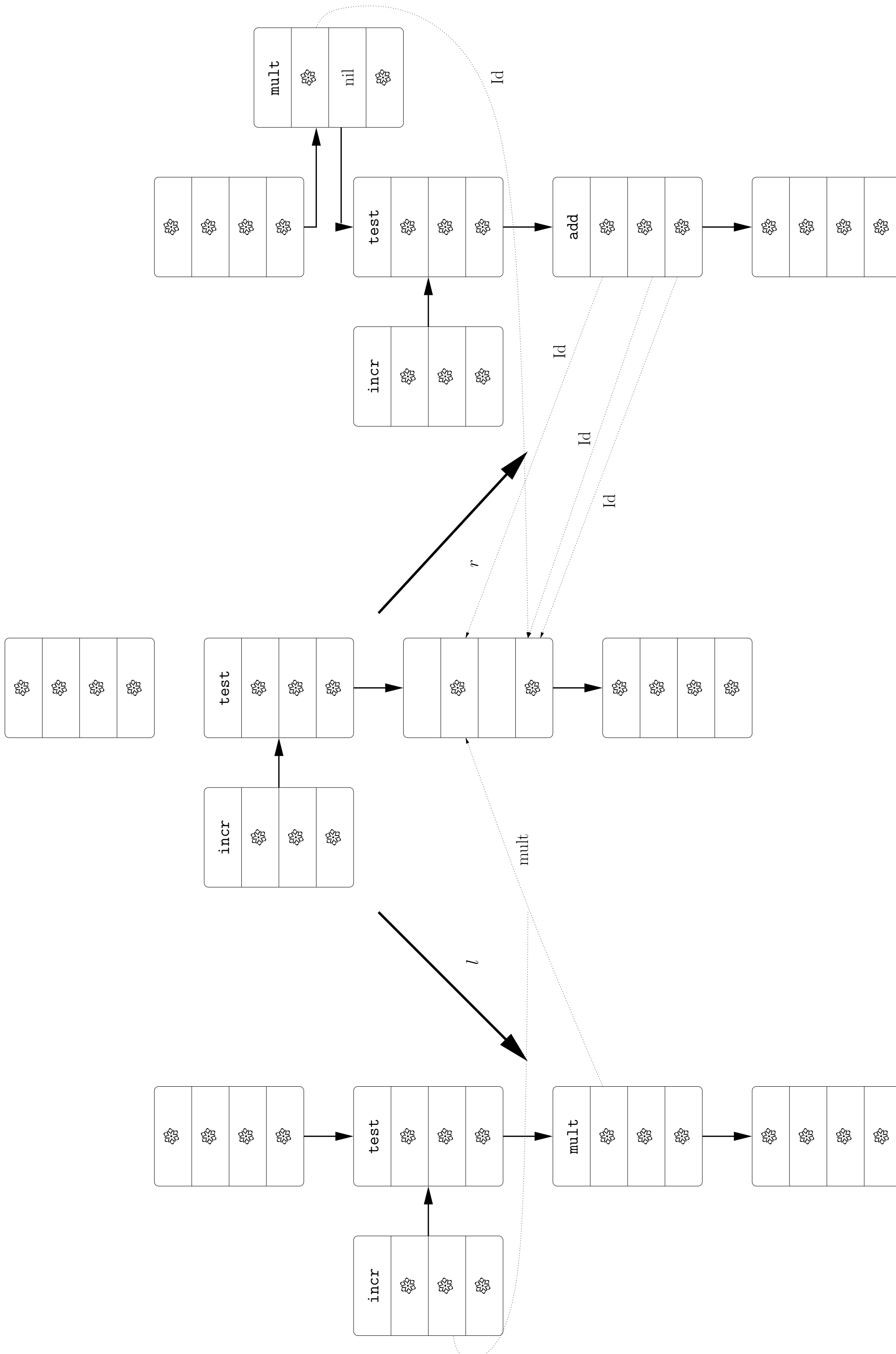



Fig. VI.22 – Seconde règle

# Troisième partie

# Outils et propriétés supplémentaires

---

Dans cette partie, on observe le comportement de notre système de transformation de graphes vis-à-vis des propriétés classiques. Comme on l'a vu, notre système ne rentre pas dans le cadre des catégories adhésives HLR. Pour autant, les différences ne sont pas énormes entre les deux approches et il peut être intéressant de voir si les bonnes propriétés des systèmes adhésifs HLR (confluence, paires critiques, parallélisme...) sont transposables dans notre système.

Pour cela, on introduit quelques outils supplémentaires comme le pullback ou l'équivalent des carrés de Van Kampen afin de pouvoir transposer dans notre formalisme les démonstrations de ces propriétés.

# Chapitre VII

# Les outils

## 1 Pullback

Comme pour le pushout, l'existence du pullback n'est pas automatique dans la catégorie **AttGraph**. Au cours de la construction, les mêmes problèmes sur les fonctions de calcul peuvent apparaître. On va donc dans un premier temps considérer les mêmes hypothèses pour les morphismes que celles utilisées dans le cadre de la construction du pushout. On travaille donc avec les morphismes $c' : B \to D$ et $b' : C \to D$ appartenant respectivement aux classes $\mathcal{N}$ et $\mathcal{M}'$ ou $\mathcal{M}''$.

**Proposition 12.** *Si $c' : B \to D$ appartient à la classe $\mathcal{N}$ et $b' : C \to D$ appartient à la classe $\mathcal{M}'$ ou $\mathcal{M}''$, alors le pullback de $c'$ et $b'$ existe.*

Pour construire ce pullback, on suit le même schéma que pour le pushout. On commence par se ramener dans la catégorie des graphes simples à l'aide du foncteur d'oubli $\mathcal{F}$ afin de calculer la structure du pullback $A$. Il y a par contre une petite subtilité pour définir les espaces d'attributs en relation avec les sommets du graphe $A$. En effet, si on regarde les morphismes définis figure VII.1, on a envie de placer sur l'unique sommet du pullback l'espace d'attributs `EA` car celui-ci se retrouve dans les trois graphes $B$, $C$ et $D$. Ce choix conduirait donc à définir des fonctions de calcul entre les graphes $B$ et $A$ d'une part et $C$ et $A$ d'autre part. Mais alors, dans la composée des morphismes $b$ et $c'$, on trouverait un arbre avec une seule feuille et dans la composée de $c$ et $b'$, un arbre avec deux feuilles apparaîtrait ; ce qui contredit la commutativité du carré dans la construction du pullback. Dans ce cas, le pullback ne va donc pas posséder l'espace d'attributs `EA`.

Il y a donc ici une condition supplémentaire à vérifier avant de placer un espace d'attributs sur la structure du pullback. Pour exprimer cette condition, on étudie les espaces d'attributs de $D$ un par un : soit $\texttt{EA}_D$ un espace d'attributs du graphe $D$, celui-ci peut se retrouver dans le pullback s'il existe dans les morphismes $b'$ et $c'$ une relation nécessaire le concernant. On regarde ensuite les relations non nécessaire l'impliquant dans le morphisme $b'$. S'il n'en existe aucune, alors

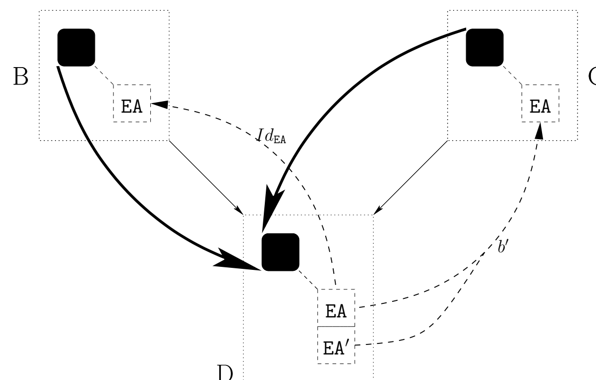



Fig. VII.1 – Quel espace d'attributs pour le pullback ?

l'espace d'attributs $\mathtt{EA}_D$ sera en relation avec le sommet correspondant dans le pullback. Dans le cas contraire, s'il existe une relation non nécessaire entre $\mathtt{EA}_D$ et $\mathtt{EA}_C$, on observe le comportement de l'espace $\mathtt{EA}_C$ dans le graphe $D$ (après avoir suivi le relation nécessaire associée à $\mathtt{EA}_C$). Si cet espace d'attributs possède une relation nécessaire dans le morphisme $c'$, alors l'espace d'attributs $\mathtt{EA}_D$ sera bien en relation avec le sommet correspondant du pullback.

Pour la construction des arbres, la définition des fonctions de calcul et le calcul explicite des attributs, on procède alors exactement de la même manière que dans le construction du pushout.

**Remarque 22.** On peut dans certains cas affaiblir les conditions sur les morphismes $b'$ et $c'$ tout en conservant l'existence du pullback. Un exemple est donné dans la démonstration du théorème 14.

# 2 Pullback faible

Dans le cas général, même si le pullback n'est pas constructible, une construction affaiblie est possible. Il s'agit alors de ne considérer que la partie structurelle des graphes et des morphismes. Comme il n'y a alors aucun espace d'attributs attaché au graphe $A$, aucune condition n'est imposée sur les morphismes $b$ et $c$. La construction est alors valide. Dans la suite, on appellera cette construction le pullback faible de $b'$ et $c'$. Bien entendu, la propriété universelle du pullback n'est pas respectée dans la catégorie **AttGraph**. Pour autant, cet outil est tout de même utile pour suivre les transformations structurelles qui se produisent au fur et à mesure que l'on applique des productions à un graphe.

# 3 Carrés de Van Kampen

Comme on l'a vu dans la section 1 du chapitre III, les bonnes propriétés présentées par les systèmes adhésifs HLR sont assurées par l'existence des carrés de VAN KAMPEN le long de monomorphismes (ou bien de morphismes dans une classe particulière de monomorphismes) dans la catégorie concernée.

Une question naturelle est alors de savoir si cette notion de carré de VAN KAMPEN est transposable dans notre approche afin de traduire sans trop d'effort les bonnes propriétés déjà évoquées.

**Définition 62.** *On se donne un pushout* (1) *(cf. figure VII.2). On dira que ce pushout est un carré de* VAN KAMPEN *si pour tout cube commutatif* (2) *dont les morphismes verticaux sont dans la classe $\mathcal{N}$ et construit avec le pushout* (1) *comme face inférieure et des pullbacks comme faces arrière, on a la propriété suivante :*

*la face supérieure est un pushout si et seulement si les faces avant sont des pullbacks.*

L'image d'un carré de VAN KAMPEN dans la catégorie **AttGraph** par le foncteur d'oubli $\mathcal{F}$ est encore un carré de VAN KAMPEN dans la catégorie des graphes simples.

**Théorème 9.** *Si dans le pushout* (1) *de la figure VII.2, le morphisme $m$ est dans la classe $\mathcal{M}'$ ou $\mathcal{M}''$ et le morphisme $f$ dans la classe $\mathcal{N}$ alors ce pushout est un carré de* VAN KAMPEN.

*Démonstration.* On utilise pour cette preuve le foncteur d'oubli qui va de notre catégorie **AttGraph** vers la catégorie des graphes simples **Graph**. Dans cette dernière, on sait que tout pushout le long d'un monomorphisme est un carré de VAN KAMPEN [48].

On se donne un cube vérifiant les propriétés de construction de la définition.

On commence par supposer que les deux faces avant sont des pullbacks. On veut montrer que la face supérieure est un pushout. Pour cela, on prouve que la propriété universelle est vérifiée. On se donne donc un graphe attribué $E'$ ainsi que deux morphismes $b' : B' \to E'$ et $c' : C' \to E'$ tels que $b' \circ m' = c' \circ f'$. En passant dans la catégorie **Graph**, comme on sait que la face supérieure est un pushout, on en déduit qu'il existe un morphisme $d' : D' \to E'$ qui factorise $b'$ et $c'$. Ce morphisme va nous servir de base pour construire celui qui nous intéresse. La première chose à vérifier est que ce morphisme structurel est compatible avec les espaces d'attributs des graphes $D'$ et $E'$. Soit

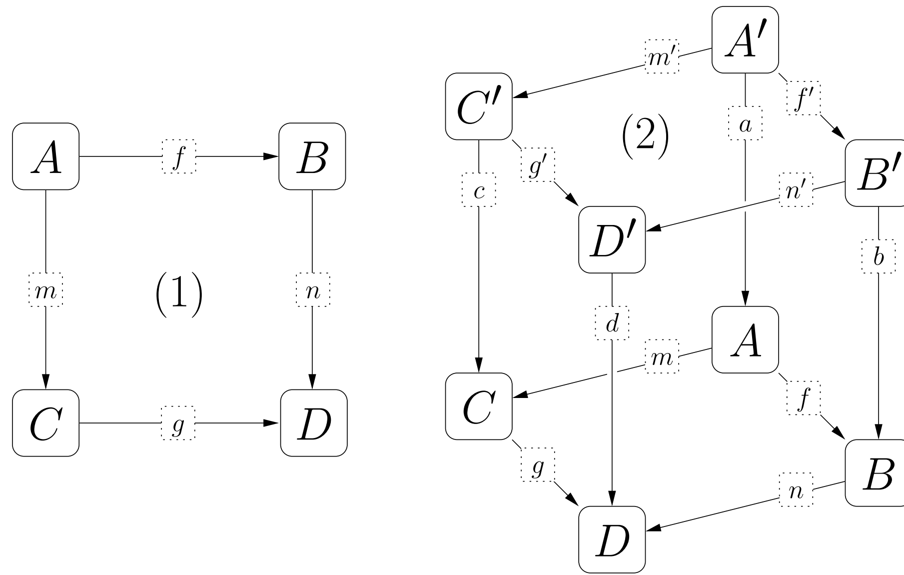



Fig. VII.2 – Carré de Van Kampen

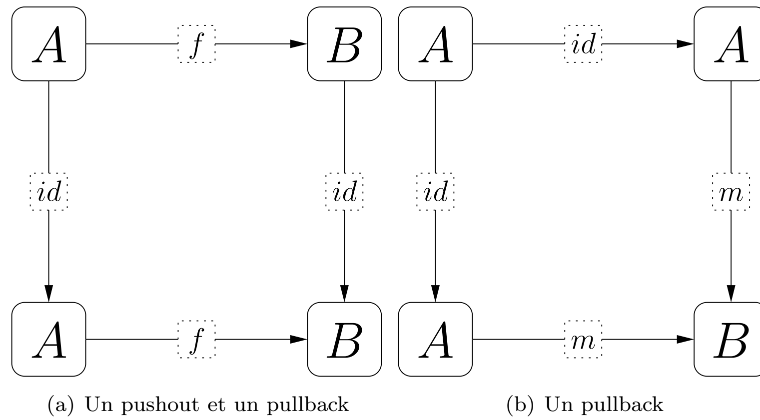
donc $s$ un sommet de $D'$ et EA un espace d'attributs en relation avec le sommet $s$. Comme on a un morphisme de $D'$ vers $D$, on sait que EA se retrouve aussi en relation avec l'image de $s$ dans $D$. Or $D$ est le pushout de la face inférieur du cube ; donc cet espace d'attributs provient du graphe $B$ ou du graphe $C$. Or les faces avant sont des pullbacks donc EA va se retrouver avec la préimage du sommet $s$ dans le graphe $B'$ ou dans le graphe $C'$ et donc par conséquent, l'espace d'attributs qui nous intéresse va aussi se retrouver en relation avec l'image de $s$ par $d'$ dans le graphe $E'$. La condition de compatibilité est donc assurée. Maintenant, pour les fonctions de calcul, on applique exactement la même construction que dans la preuve pour la propriété universelle. La face supérieure est donc bien un pushout.

Pour la réciproque, on suppose maintenant que la face supérieure est un pushout. On montre que la face avant droite est un pullback (c'est la même chose pour la face avant gauche). On procède de manière analogue : on cherche à prouver que la propriété universelle du pullback est vraie. On se donne donc un graphe $E'$ avec deux morphismes vers $D'$ et $B$ tels que le nouveau carré commute. Encore une fois, on passe dans la catégorie **Graph** pour trouver un morphisme structurel entre $E$ et $B'$. Soit $s$ un sommet de $E'$ et EA un espace d'attributs en relation avec $s$. Par conséquent, EA se retrouve dans les graphes $B$, $D'$ et dans $D$. Par hypothèse, $D'$ est le pushout de la face supérieure. Donc un espace d'attributs de ce graphe provient soit du graphe $B'$, soit du graphe $C'$. s'il vient de $B'$, c'est gagné. Sinon, on le retrouve alors dans le graphe $C$ et comme la face inférieure est un pushout, on retrouve encore EA attaché à un sommet du graphe $A$. On utilise alors le fait que la face arrière gauche est un pullback pour affirmer que la graphe $A'$ contient aussi l'espace d'attributs EA et par conséquent $B'$ aussi. Encore une fois, la condition de compatibilité est vérifiée. Il n'y a pas de problème pour définir les fonctions de calculs. Les faces antérieures sont donc bien des pullbacks. □

**Remarque 23.** Les conditions imposées aux morphismes du cube dans le théorème servent lors de l'étape de construction des fonctions de calcul pour assurer que celles-ci existent bien. En particulier, si on ne demande plus que les morphismes verticaux soient dans la classe $\mathcal{N}$, on ne pourra plus construire les morphismes de la propriété universelle des pullbacks, par contre, on pourra toujours déduire de la présence des pullbacks l'existence du pushout sur la face supérieure.

**Lemme 1.** Soient $A$ et $B$ deux graphes attribués et $f : A \to B$ un morphisme appartenant à la classe $\mathcal{M}'$ ou $\mathcal{M}''$. Alors le carré de la figure VII.3(a) est un pushout et un pullback.

**Lemme 2.** Soient $A$ et $B$ deux graphes attribués et $m : A \to B$ un morphisme appartenant à la classe $\mathcal{N}$. Alors le carré de la figure VII.3(b) est un pullback.




(a) Un pushout et un pullback (b) Un pullback

FIG. VII.3 – Construction de pullbacks et de pushouts

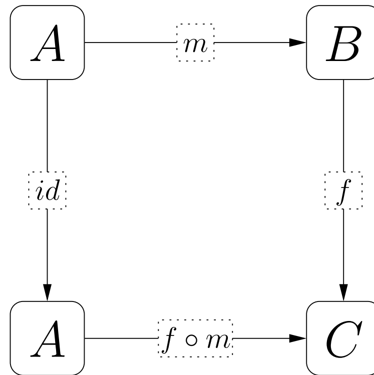



FIG. VII.4 – Un pullback spécial

**Remarque 24.** Pour le second lemme, il suffit que le morphisme $m$ soit un monomorphisme.

*Démonstration.* Pour ces deux lemmes, il est très facile de vérifier les propriétés universelles recherchées. □

**Lemme 3.** Soient $m : A \to B$ un morphisme appartenant à $\mathcal{N}$ et $f : B \to C$ un morphisme de $\mathcal{M}'$ ou $\mathcal{M}''$ tels que la composition $f \circ m$ soit dans la classe $\mathcal{N}$. alors le carré de la figure VII.4 est un pullback.

*Démonstration.* Comme $f \circ m$ appartient à la classe $\mathcal{N}$, si un sommet de $B$ est dans l'image de $m$, alors pour tous les espaces d'attributs du sommet, en analysant le morphisme $f$, on voit qu'il n'y a que la relation obligatoire de présente et la fonction de calcul associée est forcément l'identité. Donc, $f_{|Im(m)}$ est un monomorphisme. Par conséquent, si maintenant, on se donne un graphe $D$ avec deux morphismes $a : D \to B$ et $b : D \to B$ tels que $f \circ m \circ a = f \circ b$, alors, d'un point de vue structurelle, l'image de $b$ est inclus dans l'image de $m$ (dans le cas contraire, comme $f$ est structurellement injectif, on arrive à une contradiction). D'où l'égalité $m \circ a = b$. Le morphisme $a$ peut donc servir pour vérifier la propriété universelle du pullback. □

**Proposition 13.** *Soit $f : A \to B$ un morphisme de la classe $\mathcal{M}'$ ou $\mathcal{M}''$ et $m : A \to C$ un morphisme de la classe $\mathcal{N}$. alors le pushout de $f$ et de $m$ est aussi un pullback.*

*Démonstration.* On note $D$ le quatrième coin du pushout de $f$ et $m$. On construit le cube de la figure VII.5 au-dessus de ce pushout. D'après les lemmes précédents, les deux faces arrière sont des pullbacks alors que la face supérieure est un pushout. De plus, on sait que le pushout inférieur est un carré de VAN KAMPEN. Par conséquent, les deux faces avant sont des pullbacks. En particulier, celle de gauche nous donne le résultat voulu. □

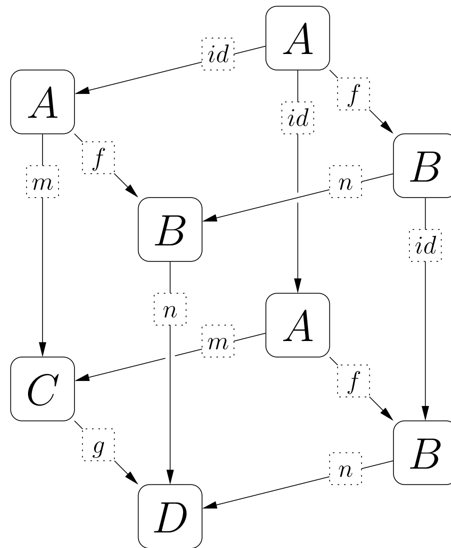



Fig. VII.5 – Le cube construit pour montrer que pushout implique pullback

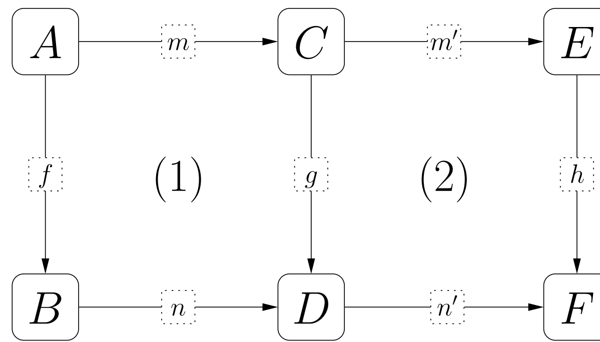



Fig. VII.6 – Décomposition pushout-pullback

**Proposition 14.** *On se donne le diagramme de la figure VII.6 avec $f$, $g$ et $h$ dans $\mathcal{N}$ et $m$, $m'$, $n$ et $n'$ dans $\mathcal{N}$. On suppose que* (1) + (2) *est un pushout et que* (2) *est un pullback. Alors* (1) *et* (2) *sont tout deux des pushouts.*

*Démonstration.* On construit les cubes de la figure VII.7 au-dessus des carrés (1) et (2). Par hypothèse, le fond du cube est un pushout, et la face antérieure droite est un pullback. De plus, d'après les lemmes précédents et les propriétés de composition des pullbacks, on sait que les faces postérieures, ainsi que la face avant droite sont des pullbacks. Par conséquent, comme on est parti d'un carré de Van Kampen, on en déduit que le haut (et donc (1)) est un pushout. Pour terminer, on utilise la décomposition du pushout pour montrer que (2) est aussi un pushout. □

**Proposition 15.** *On reprend le même diagramme que dans la proposition précédente, mais on suppose cette fois que $m'$ et $n'$ sont dans la classe $\mathcal{M}'$ ou $\mathcal{M}''$ et que $m' \circ m$ et $n' \circ n$ sont dans la classe $\mathcal{N}$. Les autres hypothèses restent inchangées. Alors le résultat tient toujours.*

*Démonstration.* On reprend exactement le même schéma de preuve. Il faut ici appliquer le troisième lemme pour trouver que la face antérieure droite et la face postérieure gauche sont des pullbacks. Ensuite, comme les morphismes verticaux ne sont pas dans $\mathcal{N}$, on n'a pas un vrai cube de Van Kampen, mais, d'après la remarque suivant le théorème, on peut tout de même en déduire que la face supérieure est un pushout. □

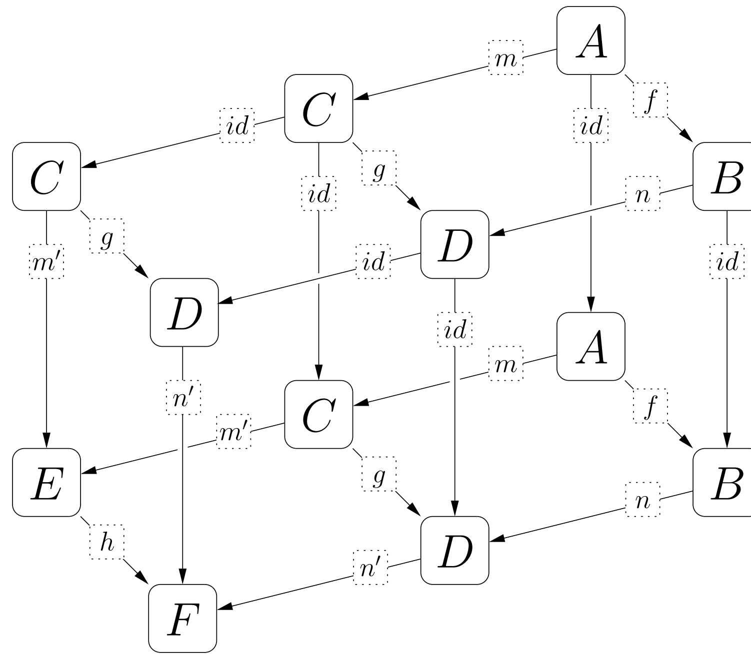



FIG. VII.7 – Le cube construit pour la preuve de la décomposition pushout-pullback

# 4 Factorisation des paires

**Définition 63.** *Soient $e_1 : E_1 \to G$ et $e_2 : E_2 \to G$ deux morphismes avec le même codomaine $G$. On dit que $e_1$ et $e_2$ sont conjointement épimorphiques si pour tout couple de fonctions $f$ et $g$ de $G$ vers $H$, on a :*

$$\forall i \in [\![1;2]\!], f \circ e_i = g \circ e_i \implies f = g.$$

**Remarque 25.** Pour la catégorie des ensembles ou celle des graphes simples, "conjointement épimorphiques" signifie "conjointement surjectifs". Dans notre catégorie **AttGraph**, à cause des fonctions de calcul, les choses sont un peu plus compliquées, et il ne suffit plus de regarder la surjectivité de la partie structurelle des morphismes. Ce n'est plus qu'une condition nécessaire.

Mais si $e_1$ et $e_2$ sont dans la classe $\mathcal{N}$, comme toutes les fonctions de calcul sont l'identité, cela redevient une condition suffisante.

**Remarque 26.** Si on se donne deux morphismes $b : A \to B$ et $c : A \to C$, alors les deux morphismes $b' : C \to D$ et $c' : B \to D$ construits lors du pushout sont conjointement épimorphiques.

**Définition 64.** *Soient $e_1$ et $e_2$ deux morphismes avec le même codomaine. On dira que la paire $(e_1, e_2)$ est dans la classe $\mathcal{E}$ de paires de morphismes si :*
- *ces deux morphismes sont dans la classe $\mathcal{N}$ ;*
- *ils sont conjointement épimorphiques.*

**Proposition 16.** *La catégorie* **AttGraph** *possède la propriété de $\mathcal{E} - \mathcal{N}$-factorisation des paires. C'est-à-dire que si l'on se donne une paire de morphismes de la classe $\mathcal{N}$ avec le même codomaine $f_1 : G_1 \to H$ et $f_2 : G_2 \to H$, alors il existe un objet $K$ dans* **AttGraph***, une paire de morphismes $(e_1 : G_1 \to K, e_2 : G_2 \to K)$ appartenant à la classe $\mathcal{E}$ et un morphisme $m \in \mathcal{N}$ de $K$ vers $H$ tels que le diagramme de la figure VII.8 commute.*

*Démonstration.* Comme les morphismes $f_i$ sont dans la classe $\mathcal{N}$ et qu'ils ont le même codomaine, il est possible de calculer leur pullback. Cela nous donne la figure VII.9(a).

Cette construction a donc permis d'extraire la partie commune des morphismes $f_1$ et $f_2$. De plus, d'après la construction générale du pullback, on sait que les morphismes $a_1$ et $a_2$ ainsi construit sont aussi dans la classe $\mathcal{N}$. On peut donc construire le pushout $K$ basé sur $a_1 : A \to G_1$ et $a_2 : A \to G_2$ (*cf.* figure VII.9(b)).

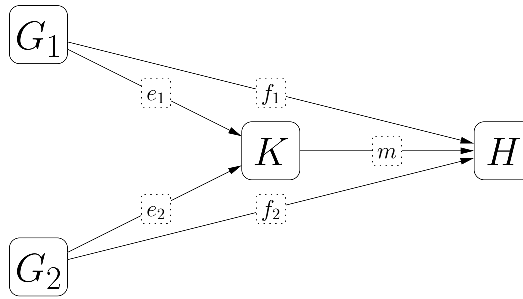
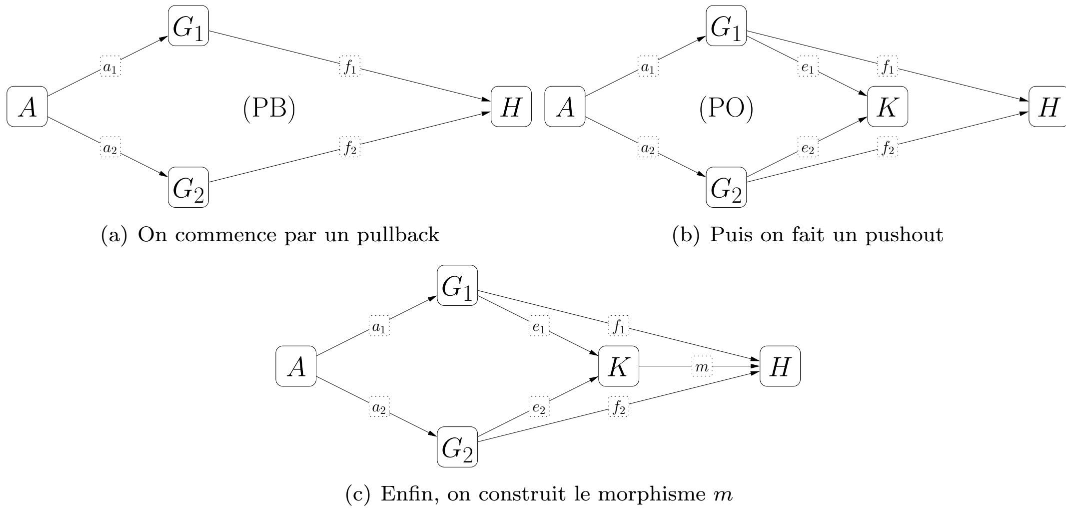



FIG. VII.8 – Factorisation des paires




(a) On commence par un pullback

(b) Puis on fait un pushout

(c) Enfin, on construit le morphisme $m$

FIG. VII.9 – Les trois étapes de la démonstration de la proposition 16

Là encore, la construction générale nous permet d'affirmer que les morphismes $e_1$ et $e_2$ sont dans la classe $\mathcal{N}$ et qu'ils sont conjointement épimorphiques. Enfin, la propriété universelle du pushout, nous permet de conclure pour l'existence du morphisme $m$ appartenant à la classe $\mathcal{N}$ (*cf.* figure VII.9(c)).

□

# Chapitre VIII

# Extensions et restrictions de transformations

## 1 Extension de transformations

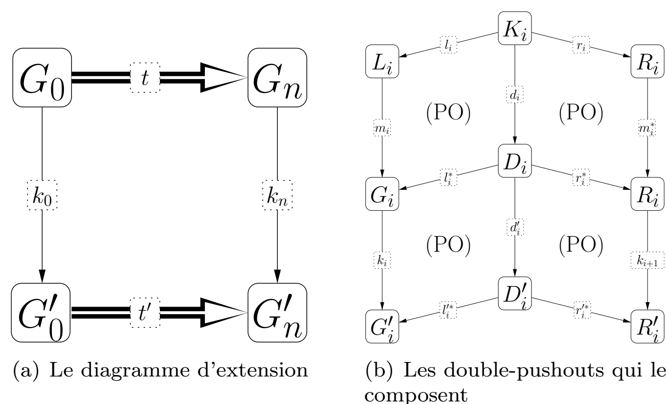
**Définition 65.** *La figure VIII.1(a) présente la forme générale d'un diagramme d'extension. Dans ce schéma, $k_0 : G_0 \to G'_0$ est un morphisme appartenant à la classe $\mathcal{N}$ et $t : G_0 \stackrel{*}{\Longrightarrow} G_n$ et $t' : G'_0 \stackrel{*}{\Longrightarrow} G'_n$ sont des transformations suivant les même productions $(p_0, \ldots, p_n)$ et ayant respectivement pour correspondance $(m_0, \ldots, m_{n-1})$ et $(k_0 \circ m_0, \ldots, k_{n-1} \circ m_{n-1})$ définies par les double-pushouts de la figure VIII.1(b).*

**Remarque 27.** Le diagramme d'extension, s'il existe, est uniquement déterminé (à isomorphisme près) par $t : G_0 \stackrel{*}{\Longrightarrow} G_n$ et $k_0 : G_0 \to G'_0$.

La question est alors de savoir sous quelles conditions, si on se donne une transformation $t : G_0 \stackrel{*}{\Longrightarrow} G_n$ et un morphisme d'extension $k_0 : G_0 \to G'_0$, on arrive à construire un diagramme d'extension. Le problème à résoudre est de savoir si au départ et après chaque étape, la "gluing condition" est vérifiée pour pouvoir appliquer la transformation directe suivante. Pour cela, on va introduire plusieurs notions : les pushout initiaux et les empans dérivés.

### 1.1 Pushout initial et condition de collage

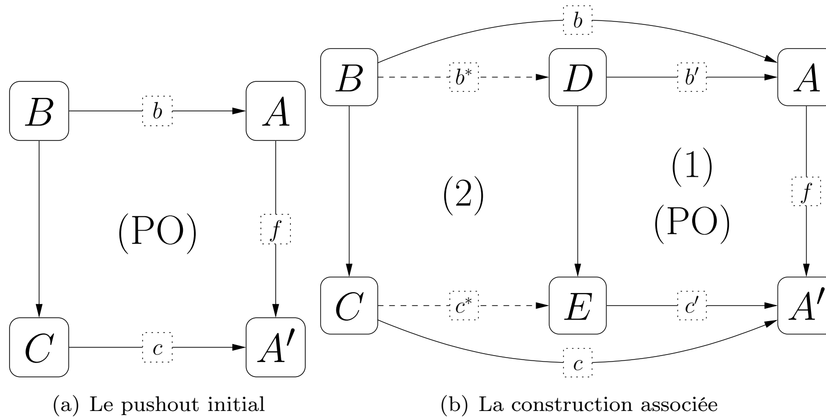
**Définition 66.** *On se donne un pushout (cf. figure VIII.2(a)). On dira que ce pushout est initial au-dessus de $f$ si pour tout pushout* (1) *(cf. figure VIII.2(b)) avec $b' \in \mathcal{M}$, il existe $b^* : B \to D$ et*




(a) Le diagramme d'extension

(b) Les double-pushouts qui le composent

Fig. VIII.1 – Extension de transformation




(a) Le pushout initial (b) La construction associée

Fig. VIII.2 – Le pushout initial

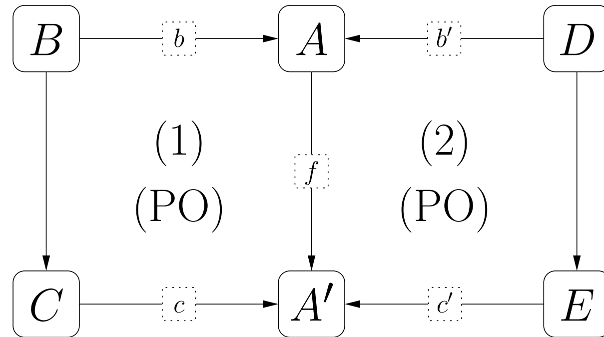



Fig. VIII.3 – Un second pushout basé sur le morphisme $f$

$c^* : C \to E$ *appartenant à* $\mathcal{M}'$ *ou* $\mathcal{M}''$ *tels que* $b' \circ b^* = b$, $c' \circ c^* = c$ *et tels que le carré* (2) *soit un pushout.*

*Le morphisme* $b$ *est alors appelé la frontière de* $f$ *et* $B$ *l'objet frontière.*

**Proposition 17.** *Si le morphisme* $f$ *appartient à la classe* $\mathcal{N}$*, alors la frontière au-dessus de* $f$ *existe toujours.*

*Démonstration.* On prend pour le graphe $B$ l'ensemble des noeuds du graphe $A$ sur lesquels le morphisme $f$ vient "rajouter" une arrête ou un espace d'attributs. On n'ajoute aucun arc ou aucun espace d'attributs à ces sommets et le morphisme qui relie le graphe $B$ à $F$ est le morphisme trivial qui envoie chaque sommet sur celui qui lui correspond. Ce graphe $B$ correspond en réalité exactement à la réunion des ensembles *Dangling* et *Floating* considérés pour la vérification de la "gluing condition" (*cf.* section V.2). On peut remarquer que ce graphe peut être vu comme un graphe simple (il ne possède aucun attribut).

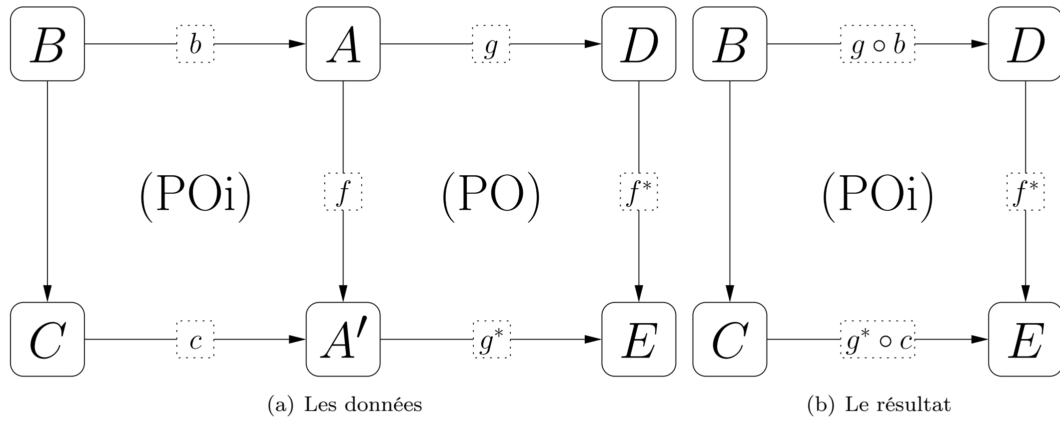
La condition de collage étant validée par construction, on peut donc construire le pushout-complement. Il reste à prouver le caractère initial de ce pushout. Pour cela, on se donne un second pushout basé sur le morphisme $f$ (*cf.* figure VIII.3).

Comme le carré (2) est un pushout, la "gluing condition" est vérifiée pour les morphismes $b'$ et $f$. Par conséquent, l'image du graphe $B$ par $b$ est inclus dans l'image de la structure du graphe $D$ dans $A$ et comme le morphisme $b'$ est structurellement injectif, on peut prendre l'image réciproque de chaque sommet qui nous intéresse pour construire le morphisme $b^*$ entre $B$ et $D$. Il est clair que $b = b' \circ b^*$.

Enfin, pour trouver le morphisme $c^*$ entre $C$ et $E$, on fait une construction similaire à celle effectuée ci-dessus pour trouver $b^*$.

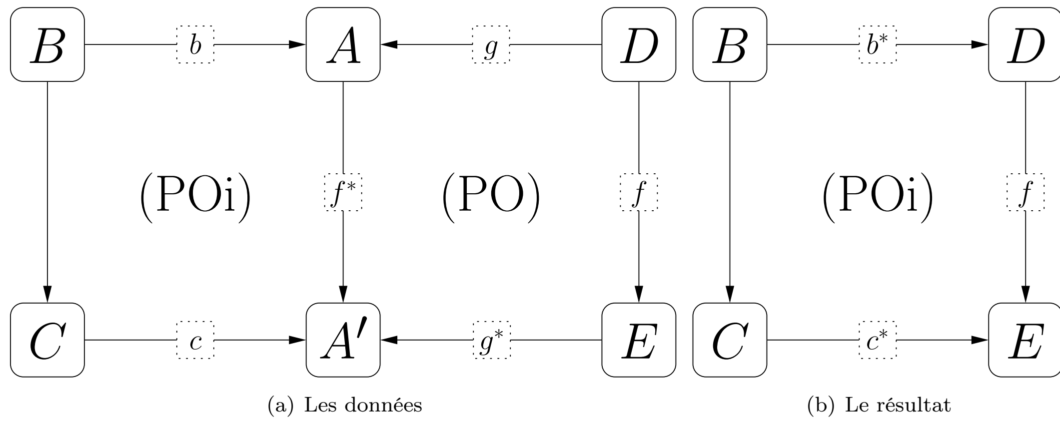
On a donc bien la propriété voulue. □

On va maintenant énoncer et démontrer un lemme qui nous servira dans le démonstration du




(a) Les données (b) Le résultat

Fig. VIII.4 – Propogation du pushout initial (direction semblable)




(a) Les données (b) Le résultat

Fig. VIII.5 – Propogation du pushout initial (direction opposée)

théorème de confluence locale.

**Lemme 4.** Les pushouts initiaux au dessus de morphisme de la classe $\mathcal{N}$ se propagent le long des pushouts de direction semblable ou opposé. Plus précisément, si on se donne un pushout et un pushout initial comme sur la figure VIII.4(a), alors on trouve un pushout initial en composant les carrés (*cf.* figure VIII.4(b)). D'autre part, si les données initiales correspondent à la figure VIII.5(a), alors on trouve encore un pushout initial (*cf.* figure VIII.5(b)) avec les morphismes $b^*$ et $c$ provenant de la définition du pushout initial.

*Démonstration.* On se donne un pushout initial basé sur le morphisme $f : A \to A'$ et un second pushout où le retrouve le morphisme $f$ (*cf.* figure VIII.4(a)). On veut montrer que la composition (figure VIII.4(b)) est un pushout initial (on sait déjà que c'est un pushout).

Pour cela, on écrit le pushout initial au dessus de $f'$, on note $b'$ sa frontière et $B'$ l'objet frontière. Son caractère initial nous donne les morphismes $b'^* : B' \to B$ et $c'^* : C' \to C$ de telle sorte que le carré de la figure soit un pushout. En composant ce pushout avec le pushout initial au dessus de $f$, on trouve un nouveau pushout basé sur $f$ et donc des morphismes $b^* : B \to B'$ et $c^* : C \to C'$. Comme $b^*$ et $b'^*$ peuvent être vus comme des morphismes injectifs de graphes simples, on est déduit que $B = B'$. De même, $C = C'$. CQFD. Pour la seconde partie du lemme, on se donne un pushout basé sur $f^*$, mais de direction opposé (*cf.* figure VIII.5(a)). On veut montrer que le carré de la figure VIII.5(b) est initial. Pour cela, on se donne un pushout supplémentaire basé sur $f$. Par composition, on trouve un pushout basé sur $f$ et l'hypothèse de départ nous permet de déduire l'existence de morphismes $b'^* : B \to F$ et $c'^* : C \to G$. Ces morphismes sont exactement ceux que l'on cherche. □

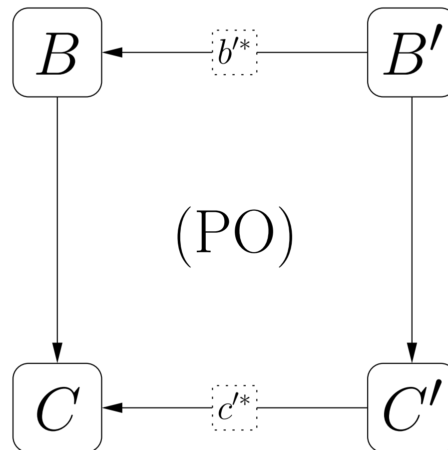



Fig. VIII.6 – Le pushout entre les objets frontières

**Remarque 28.** Les deux points de ce lemme permettent d'affirmer que les pushouts initiaux se propagent le long des transformations.

**Théorème 10.** *Soit $p : L \leftarrow K \rightarrow R$ une production, $G$ un graphe attribué et $m$ une correspondance entre $L$ et $G$. Alors la "gluing condition" pour le morphisme $m$ est équivalente à l'existence d'un morphisme $b^*$ entre l'objet frontière de $m$ et le graphe $K$ tel que $l \circ b^* = b$ où $b$ est la frontière de $m$.*

*Démonstration.* Évidente d'après la construction des pushouts initiaux. □

## 1.2 Empan et extension

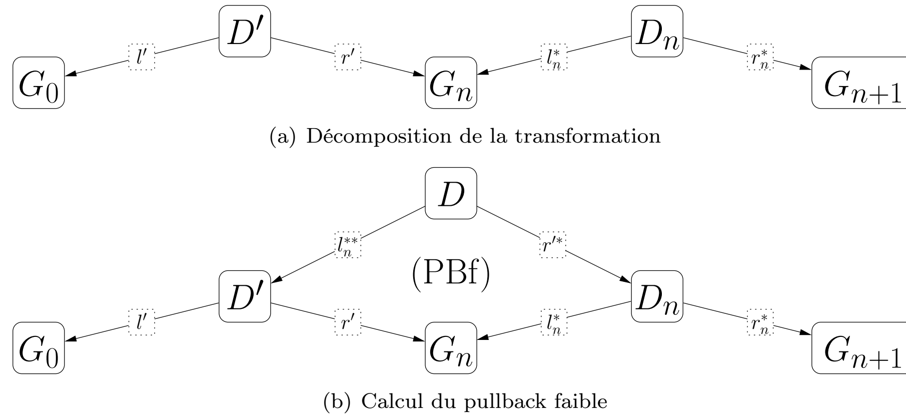
Le pushout initial et un morphisme de $B$ vers $K$ permettent donc de vérifier que rien ne disparaît au cours de la première étape d'une transformation directe qui serait nécessaire pour construire le graphe $D$. Maintenant, on voudrait appliquer cet outil aux transformations quelconques. Il faut donc trouver un moyen pour connaître la partie du graphe $G_0$ qui reste inchangée tout au long de la transformation. L'empan dérivé, que l'on va définir maintenant, est la solution.

**Définition 67.** *Soit $t : G \stackrel{*}{\Longrightarrow} H$ une transformation. On définit l'empan dérivé de $t$, noté $der(t)$, par récurrence suivant sur sa longueur :*
- *si $t$ est de longueur 1, alors $der(t) = (G \leftarrow D \rightarrow H)$ ;*
- *si la transformation $t$ est de longueur $n+1$, on la décompose en une transformation $t'$ de longueur $n$ et d'empan dérivé $(G_0 \leftarrow D' \rightarrow G_n)$ et une transformation directe de $g_n$ vers $G_{n+1}$. On a alors le diagramme de la figure VIII.7(a). On calcule alors le pullback faible de $r'$ et de $l_n^*$ (cf. figure VIII.7(b)). L'empan dérivé de $t$ est alors $der(t) = (\ G_0 \xleftarrow{l' \circ l_n^{**}} D \xrightarrow{r_n^* \circ r'^*} G_{n+1}\ )$.*

**Remarque 29.** Le calcul de l'empan dérivé ne dépend pas de l'ordre dans lequel on calcule les pullbacks faibles.

**Remarque 30.** Il y a ici une différence importante avec l'approche classique : l'empan dérivé ne permet pas dans notre cas de définir une transformation directe entre $G_0$ et $G_n$, à l'inverse du cas classique. Ceci vient du fait que l'empan dérivé ne prend absolument pas en compte l'aspect attribué de nos graphes.

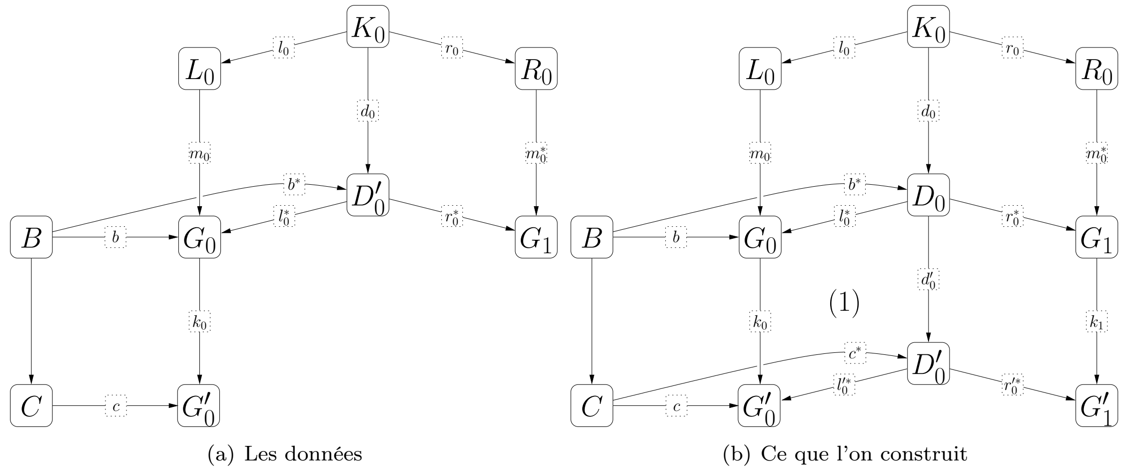
**Définition 68.** *On se donne une transformation $t : G_0 \stackrel{*}{\Longrightarrow} G_n$ dont l'empan dérivé est $der(t) = (\ G_0 \xleftarrow{l'} D \xrightarrow{r'} G_n\ )$. Un morphisme $k_0 : G_0 \rightarrow G_n$ appartenant à la classe $\mathcal{N}$ sera dit consistent par rapport à la transformation $t$ s'il existe un morphisme entre $B$ l'objet frontière de $k_0$ et le graphe $D$ tel que $l' \circ b = b_0$, où $b_0$ est la frontière de $k_0$.*




(a) Décomposition de la transformation

(b) Calcul du pullback faible

FIG. VIII.7 – L'empan dérivé




(a) Les données

(b) Ce que l'on construit

FIG. VIII.8 – La démonstration au rang 1

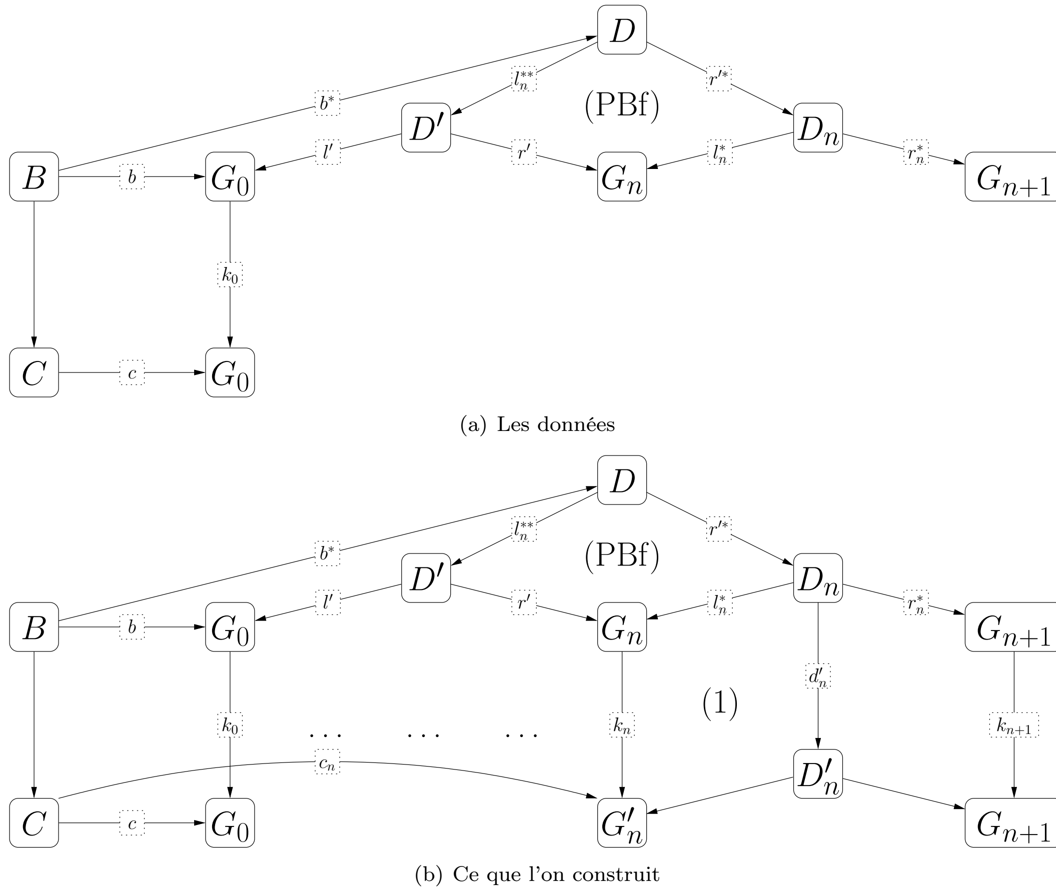
**Théorème 11.** *Soient* $t\,:\ G_0 \overset{*}{\Longrightarrow} G_n$ *une transformation et* $k_0 : G_0 \rightarrow G_n$ *un morphisme consistent par apport à* $t$*. Alors, il existe un diagramme d'extension basé sur* $t$ *et* $k_0$*.*

*Démonstration.* On raisonne par récurrence sur la longueur de la transformation $t$. Si $t$ est de longueur 1 (la figure VIII.8(a) représente les données de départ), on effectue ces différentes étapes :

– on calcule le pushout $D_0'$ de $B \rightarrow C$ et $b^*$ ;
– par propriété universelle, on construit le morphisme $l_0'^* : D_0' \rightarrow G_0'$ ;
– par décomposition du pushout, on sait que le carré (1) est un pushout ;
– on construit ensuite le graphe $G_1$, pushout de $d_0'$ et $r_1^*$ ;
– enfin, les propriété de composition de pushout permettent de voir que ce diagramme correspond à ce que l'on cherche (*cf.* figure VIII.8(b)).

On suppose maintenant que la propriété est vérifiée jusqu'au rang $n$. Vérifions la pour le rang $n+1$. On se donne donc une transformation $t$ de longueur $n+1$. On peut décomposer celle-ci en une transformation de longueur $n$ suivi d'une transformation directe $t:\ G_0 \overset{n}{\Longrightarrow} G_n \overset{p_n,m_n}{\Longrightarrow} G_{n+1}$ . Les données sont regroupées dans la figure VIII.9(a). Les étapes de la construction sont les suivantes :

– on applique l'hypothèse de récurrence pour construire $G_n'$, ainsi que les morphismes $k_n$ et $c_n$. Il est possible d'utiliser la récurrence ici car le morphisme $l_n^{**} \circ b^*$ nous donne la consistance de $k_0$ par rapport à la transformation de longueur $n$ ;
– on calcule le pushout $D_n'$ de $B \rightarrow C$ et $r'^* \circ b^*$ ;
– la propriété universelle de ce pushout nous permet d'obtenir un morphisme entre $D_n'$ et $G_n'$ ;
– en décomposant les pushouts, on sait que le carré (1) est un pushout ;
– enfin, on construit le pushout $G_{n+1}'$ à partir de $d'^*$ et de $r_n^*$ ;




(a) Les données

(b) Ce que l'on construit

FIG. VIII.9 – La démonstration au rang $n+1$

– les compositions de pushouts nous donne exactement ce que l'on cherche (*cf.* figure VIII.9(b). □

**Théorème 12.** *La réciproque du théorème 11 est juste : si le diagramme d'extension existe, alors le morphisme $k_0$ est consistent par rapport à $t$.*

*Démonstration.* On raisonne encore par récurrence sur la longueur de la transformation $t$. Si $t$ est une transformation directe, alors, d'après le théorème 10, on a le résultat voulu.

Si $t$ est de longueur $n+1$, on la décompose en une transformation de longueur $n$ et une transformation directe, comme dans la figure VIII.10. Par hypothèse de récurrence, on sait qu'il existe un morphisme $b_0$ entre $B$ et $D'$ tel que $l' \circ b_0 = b$. De plus, par propagation du pushout initial, on sait que $r' \circ b_0$ est la frontière de $k_n$, donc il existe un morphisme $b_1$ entre $B$ et $D_n$ tel que $l_n^* \circ b_1 = r' \circ b_0$. Cette dernière égalité nous permet d'appliquer la propriété universelle du pushout faible et ainsi de construire un morphisme structurelle entre $B$ et $D_{n+1}$. Or comme ces deux graphes ne sont pas attribués, ce morphisme structurelle est un morphisme dans notre catégorie. □

## 2 Restriction de transformations

**Théorème 13.** *Si on se donne une transformation directe $G' \overset{p,m'}{\Longrightarrow} H'$, un morphisme $s : G \to G'$ de classe $\mathcal{N}$ et une correspondance $m : L \to G$ tels que $m' = s \circ m$, alors on a une transformation directe de $G$ par $p$ qui nous donne le diagramme d'extension de la figure VIII.11.*

*Démonstration.* Les étapes de cette démonstration sont résumées dans la figure VIII.12.

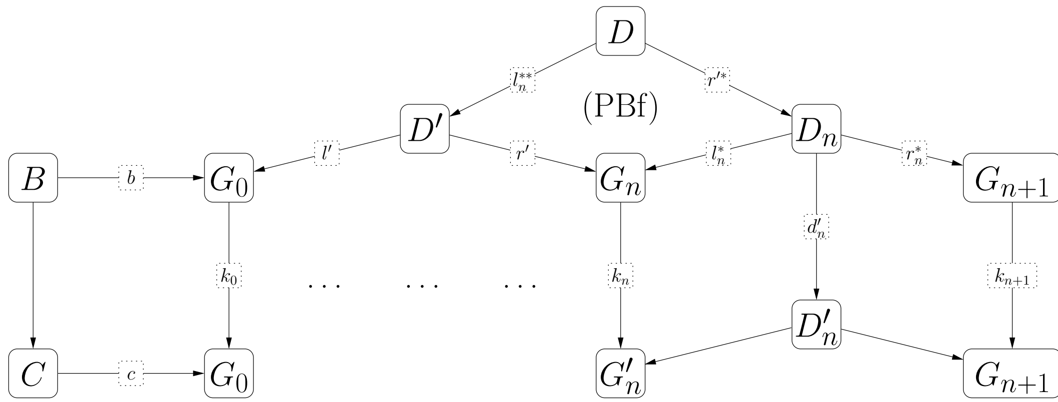



Fig. VIII.10 – Les données pour la démonstration du théorème 12

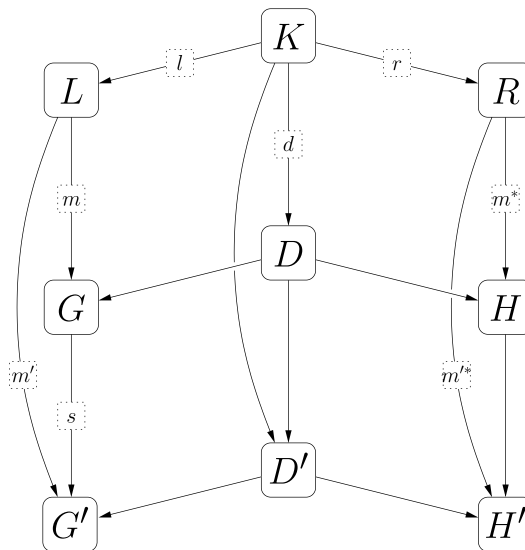
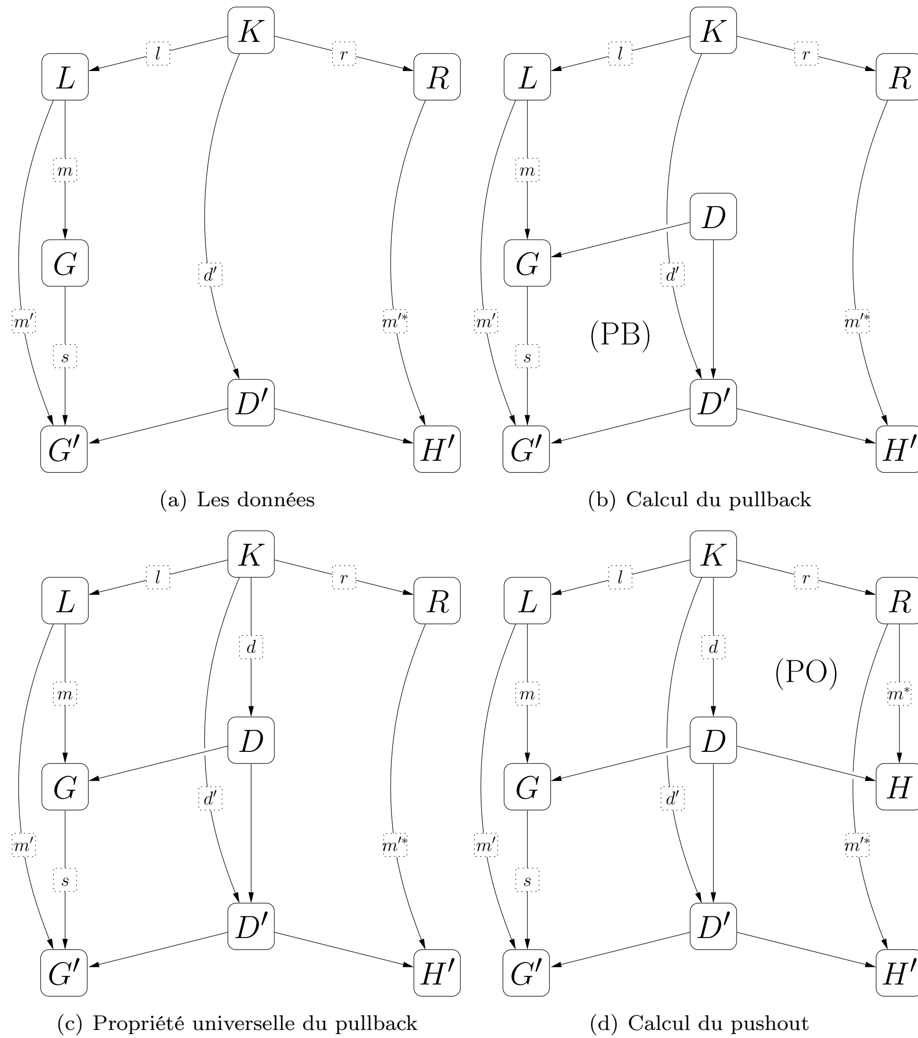



Fig. VIII.11 – Restriction de transformation




(a) Les données

(b) Calcul du pullback

(c) Propriété universelle du pullback

(d) Calcul du pushout

FIG. VIII.12 – La démonstation du théorème 13

On commence par calculer le pullback $D$ de $s : G \rightarrow G'$ et de $l'' : D' \rightarrow G'$ (figure VIII.12(b)). Comme le carré $KLD'G'$ est commutatif, on est sûr que les espaces d'attributs du graphe $G'$ impliqués dans le morphisme $l''$ apparaissent aussi dans le morphisme $s$ (dans une relation nécessaire car $s$ appartient à la classe $N$) ; le pullback de $s$ et $l''$ existe donc bien. La propriété universelle du pullback nous permet de construire un morphisme $k$ entre $K$ et $D$ qui factorise $k'$ et $m \circ l$ (figure VIII.12(c)). Grâce à la décomposition pushout-pullback, on sait que le carré supérieur est un pushout.

Pour terminer, on calcule le graphe $H$ par pushout (figure VIII.12(d)). La propriété universelle du pushout permet de construire un morphisme entre $H$ et $H'$ en composant les pushouts, on arrive à la conclusion voulue.

□

# Chapitre IX

# La confluence

Le but de ce chapitre est d'étudier la confluence d'un système de récriture de graphes [42]. On commence par rechercher les conditions nécessaires à la permutation de l'application de deux règles, avant de passer à l'étude des paires critiques du système.

## 1 Indépendance parallèle et indépendance séquentielle

**Définition 69.** *Soit deux productions $p_1 = L_1 \leftarrow K_1 \rightarrow R_1$ et $p_2 = L_2 \leftarrow K_2 \rightarrow R_2$. Deux dérivations directes $G \overset{p,m}{\Longrightarrow} H$ et $G \overset{p',m'}{\Longrightarrow} H'$ seront dites parallèlement indépendantes s'il existe des morphismes $k_1 : L_2 \rightarrow D_1$ et $k_2 : L_1 \rightarrow D_2$ appartenant à la classe $\mathcal{N}$ tels que (cf. figure IX.1) :*

- *$l_2^* \circ k_2 = m_1$ et $l_1^* \circ k_1 = m_2$ ;*
- *les morphismes $r_2^* \circ k_2$ et $r_1^* \circ k_1$ appartiennent à la classe $\mathcal{N}$.*

**Définition 70.** *Soit deux productions $p_1 = L_1 \leftarrow K_1 \rightarrow R_1$ et $p_2 = L_2 \leftarrow K_2 \rightarrow R_2$. Une dérivation séquentielle $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2}{\Longrightarrow} G'$ sera dite séquentiellement indépendante s'il existe des morphismes $k_2 : R_1 \rightarrow D_2$ et $k_1 : L_2 \rightarrow D_1$ appartenant à la classe $\mathcal{N}$ tels que (cf. figure IX.2) :*

- *$l_2^* \circ k_2 = m_1^*$ et $r_1^* \circ k_1 = m_2$ ;*
- *les morphismes $l_1^* \circ k_1$ et $r_2^* \circ k_2$ sont dans la classe $\mathcal{N}$.*

**Remarque 31.** Dans le cas de transformations réversibles, alors les transformations $G \overset{p_1}{\Longrightarrow} H_1 \overset{p_2}{\Longrightarrow} G'$ sont séquentiellement indépendantes si et seulement si $G \overset{p_1^{-1}}{\Longleftarrow} H_1 \overset{p_2}{\Longrightarrow} G'$ . Mais on a vu que si les fonctions de calcul du morphisme $l_1$ ne sont pas des isomorphismes alors la production $p_1$ n'est pas réversible. Cette équivalence n'est donc pas toujours vérifiée.

**Théorème 14.** *Soient $G \overset{p_1,m_1}{\Longrightarrow} H_1$ et $G \overset{p_2,m_2}{\Longrightarrow} H_2$ deux dérivations directes parallèlement indépendantes. Alors on va pouvoir appliquer la transformation $p_1$ au graphe $H_2$ et la transformation $p_2$ au graphe $H_1$ pour trouver le même graphe $G'$ à l'arrivée. De plus, les dérivations $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2'}{\Longrightarrow} G'$ et $G \overset{p_2,m_2}{\Longrightarrow} H_2 \overset{p_1,m_1'}{\Longrightarrow} G'$ seront séquentiellement indépendantes.*

*Soit $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2}{\Longrightarrow} G'$ une dérivation séquentiellement indépendante. Alors, on va pouvoir inverser l'ordre des deux transformations pour trouver le même résultat $G'$ : $G \overset{p_2,m_2'}{\Longrightarrow} H_2 \overset{p_1,m_1'}{\Longrightarrow} G'$ . De plus, les transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ et $G \overset{p_2,m_2'}{\Longrightarrow} H_2$ seront parallèlement indépendantes.*

*Démonstration.* Les conditions imposées sur $k_1$ et $k_2$ vont permettre dans un premier lieu de construire le pullback de $l_1^*$ et de $l_2^*$. En effet, si d'un espace d'attributs d'un sommet de $G$ démarre un arbre du morphisme $l_1^*$ portant une fonction de calcul $b$ différente de l'identité, alors on sait par construction que cette fonction provient du morphisme $l_1$ et donc par conséquent le sommet de $G$ considéré appartient à l'image de $l_1$ (on note $s$ sa préimage), et donc aussi à celle de $l_2^* \circ k_2$.

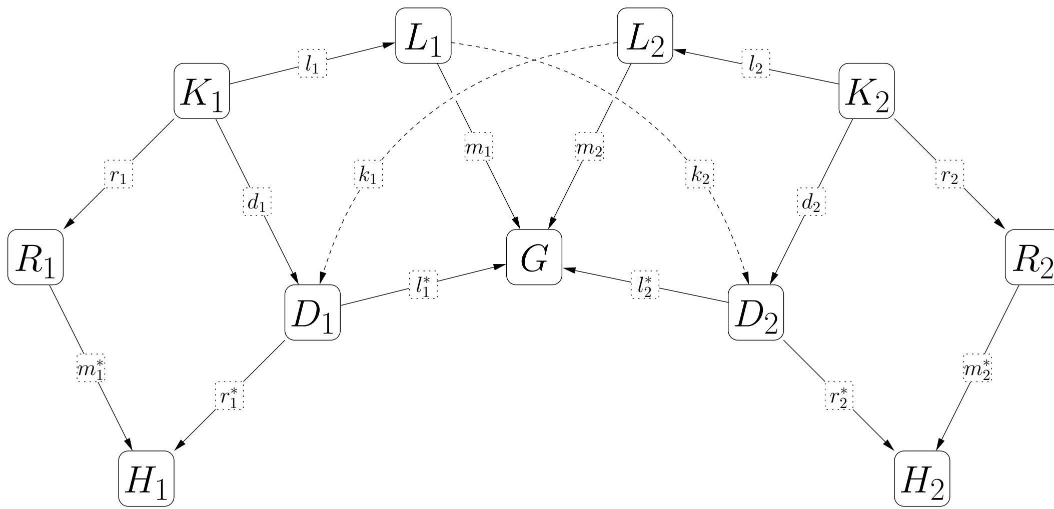



Fig. IX.1 – Indépendance parallèle

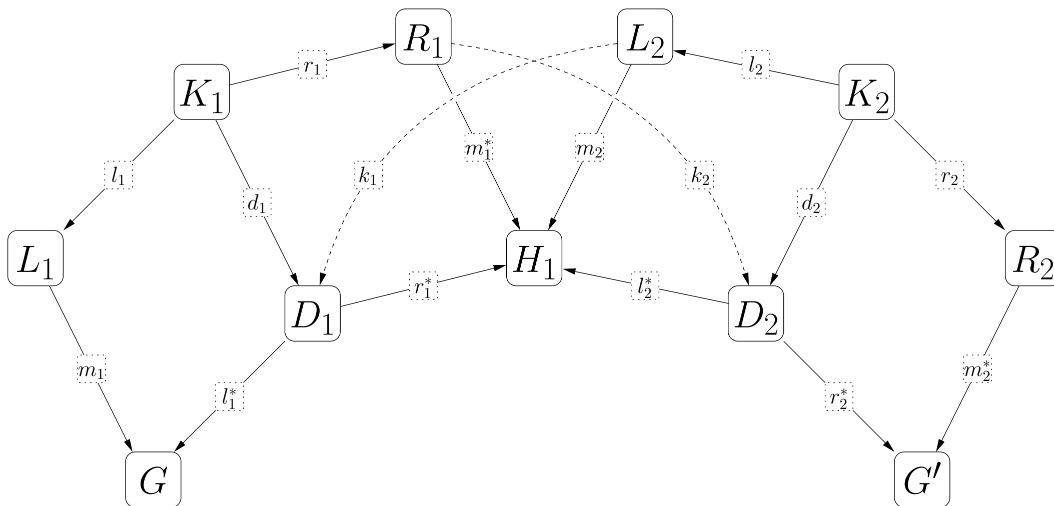



Fig. IX.2 – Indépendance séquentielle

Or ce dernier morphisme est dans la classe $\mathcal{N}$, cela implique donc que sur les espaces d'attributs de $k_2(s)$, il n'y a que les relations obligatoires portant l'identité. Symétriquement, on a la même chose si on part d'une fonction de calcul du morphisme $l_2^*$. C'est exactement la condition dont on a besoin pour pouvoir construire le pullback.

On utilise ensuite la propriété universelle du pullback pour construire des morphismes $k_1^* : K_2 \to D$ et $k_2^* : K_1 \to D$. Ces morphismes appartiennent à la classe $\mathcal{N}$. On le montre pour $k_1^*$. On a $d_1 \circ k_2^* = k_1 \in \mathcal{N}$ ; donc si il y a une relation non obligatoire dans $k_2^*$, alors elle se retrouvera dans la composition, ce qui est impossible. De plus, il est facile de voir que les fonctions de calcul sont alors forcément égales à l'identité.

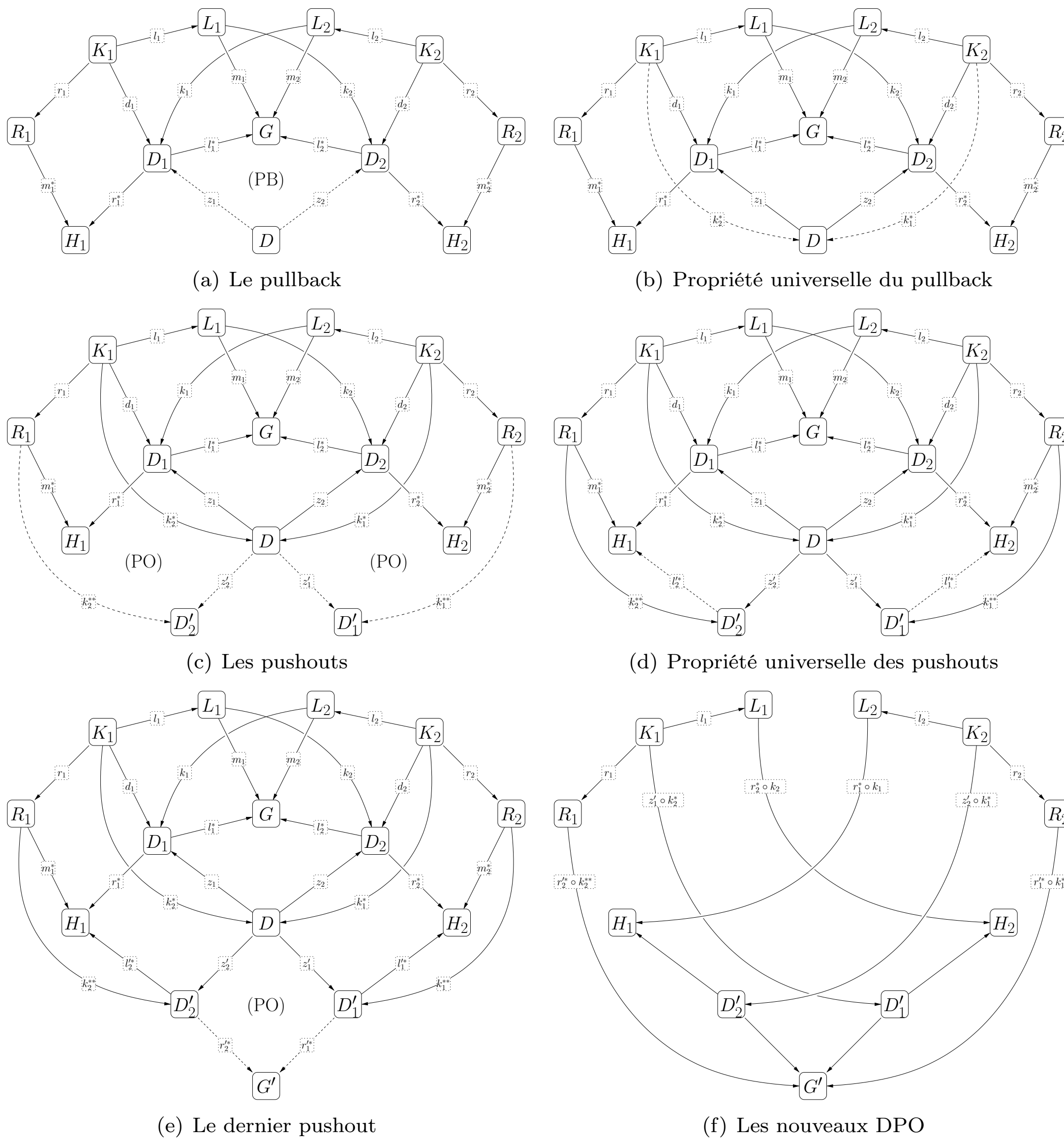
On peut alors maintenant construire le pushout $D_2'$ basé sur $k_1^*$ et sur $r_1$ et le pushout $D_1^*$ basé sur $k_2^*$ et $r_2$. Ces pushouts sont parfaitement constructibles car les fonctions de calculs de $r_1$ et $r_2$ sont inversibles. x La propriété universelle du pushout nous permet ensuite de connaître les morphismes $l_2^{*'} : D_2' \to H_1$ et $l_1^{*'} : D_1' \to H_2$.

Enfin, on va construire le pushout de $d_1'$ et $d_2'$. *A priori*, ces deux morphismes sont dans $\mathcal{M}$, il faut donc utiliser la généralisation de la construction du pushout pour obtenir le résultat voulu. On commence donc par montrer que $d_1' \circ k_2^*$ et $d_2' \circ k_1^*$ sont dans la classe $\mathcal{N}$. On montre la propriété voulue pour la composition $d_1' \circ k_2^*$. Comme $k_2^*$ est dans $\mathcal{N}$, il suffit de montrer que sur les espaces d'attributs des sommets appartenant à l'image de ce morphisme, il n'arrive (pour le morphisme $d_1'$) que les relations obligatoires et qu'elles portent l'identité comme fonction de calcul. On raisonne pour cela par l'absurde : si on une fonction de calcul $b$ différente de l'identité dans le morphisme $d_1'$, alors par construction du pushout, elle provient de $r_2$ et on la retrouve aussi dans $r_2^*$. Or, par hypothèse, la composition $r_2^* \circ k_2$ est dans la classe $\mathcal{N}$, donc le sommet sur lequel arrive cette fonction $b$ pour le morphisme $r_2^*$ n'appartient pas à l'image de $k_2$, mais il a forcement une préimage dans $K_2$ et par commutativité du diagramme une préimage dans $D$. On sait par ailleurs que $k_2 \circ l_1 = d_2 \circ k_2^*$, par conséquent, le sommet considéré dans $D$ n'a pas de préimage par $k_2^*$. Il faut maintenant montrer que l'on peut découper le graphe $D$ en deux sous-graphes de telle sorte que les morphismes $d_1'$ et $d_2'$ se décomposent correctement dessus. On prend pour $D'$ et $D''$ la même structure que $D$, maintenant si un espace d'attributs d'un sommet de $D$ était déjà en relation avec la préimage du sommet (si cette préimage existe) dans $K_1$, on place cet espace d'attributs sur le graphe $D'$, sinon il va sur le graphe $D''$. Les restrictions des morphismes $d_1'$ et $d_2'$ aux graphes $D'$ et $D''$ vérifient bien les conditions voulues. En effet, $d'_{1|D'}$ appartient bien à $\mathcal{N}$ d'après la discussion précédente. Pour $d'_{2|D''}$, il faut en plus remarquer que si un espace d'attributs n'a aucun “antécédent” dans les graphes $K_1$ ou $K_2$ alors, vu que les morphismes $d_1'$ et $d_2'$ sont construits par pushout, cela implique que le seule arbre qui arrive sur cet espace d'attributs est la relation obligatoire et qu'il porte l'identité. On peut donc bien calculer le pushout de $d_1'$ et $d_2'$.

On termine la preuve grâce aux propriétés de composition et de décomposition des pushouts et des pullbacks pour faire apparaître nos doubles pushouts. On applique donc la règle $p_2$ au graphe $H_1$ grâce à la correspondance $r_1^* \circ k_1$. Un pushout-complement nous donne le graphe $D_2'$ et un pushout nous permet de tomber sur le graphe $G'$. De même, en appliquant la règle $p1$ à $H_2$, on passe par le graphe $D_1'$ avant de retomber sur $G'$.

On a donc prouvé le premier point du théorème. Pour le second, on va faire exactement la même chose avec ici, le graphe $H_1$ qui va tenir le rôle du graphe $G$ auparavant. Il y a cependant deux endroit où il faut faire un peu plus attention : comme on n'a pas supposé les productions $p_1$ et $p_2$ réversibles, les fonctions de calcul du morphisme $l_1$ ne sont pas forcement inversibles et par conséquent, comme on l'a déjà remarqué, le calcul des attributs du graphe $D_2'$ peut bloquer. De même ensuite, pour le calcul du pushout $G'$.

Pour la première difficulté, on s'en sort en réutilisant les attributs du graphe $G$. On montre dans la figure IX.4 les données du problème. Tout ce qu'il reste à faire c'est placer les attributs sur le graphe $D_2'$. On sépare alors les cas : si un attribut provient du graphe $D$ (mais pas de $K_1$), la fonction de calcul entre les deux est l'identité et on peut donc placer le même dans $D_2'$. Sinon il provient soit de $L_1$, soit de $K_1$ ; dans les deux cas, il existe un attributs correspondant dans le graphe $G$. C'est celui-là qu'on prend pour remplir la case qui nous intéresse. Il est logique de procéder ainsi car le graphe le graphe $D_2'$ représente le graphe $G$ après le premier pushout de la transformation directe $p_2$ (en particulier, les attributs que modifie $p_1$ doivent encore être




(a) Le pullback
(b) Propriété universelle du pullback
(c) Les pushouts
(d) Propriété universelle des pushouts
(e) Le dernier pushout
(f) Les nouveaux DPO

FIG. IX.3 – Les différentes étapes de la preuve

inchangés). On vérifie ensuite facilement que $D_2'$ ainsi construit vérifie bien la propriété universelle du pushout.

Pour construire $G'$, c'est encore plus simple. En effet, le cas de figure précédent pose des problèmes lorsque dans la règle de transformation les attributs sont des jokers ❀ alors que dans les graphes que l'on transforme on a des vrais attributs. Ce cas de figure ne peut plus arriver dans le calcul du dernier pushout : si on a un joker sur le graphe $D$, alors soit il se retrouve tel que sur le graphe $D_2'$, soit il est aussi présent sur le graphe $K_1$. Les règles de construction des transformations permettent d'affirmer que ce joker se retrouve aussi sur le graphe $L_1$ et donc, par pushout,sur le $D_2'$. Idem pour $D_1$. On peut donc construire facilement les attributs du graphe $H_2$.

□

## 2 Paires critiques

Après avoir étudié l'indépendance de deux productions et donc la confluence locale, on s'intéresse maintenant à la confluence globale d'un système de transformation ; On introduit pour cela

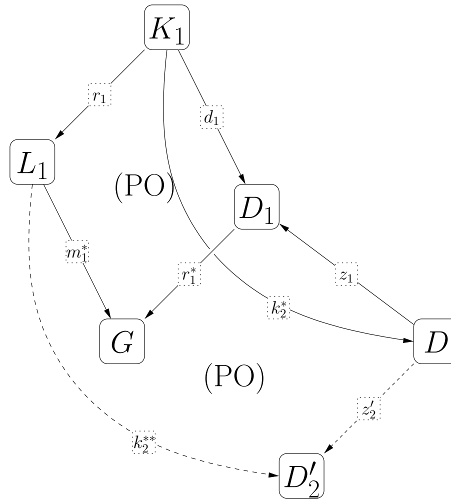



FIG. IX.4 – Les données (en trait plein) et le pushout que l'on souhaite construire (en pointillés)

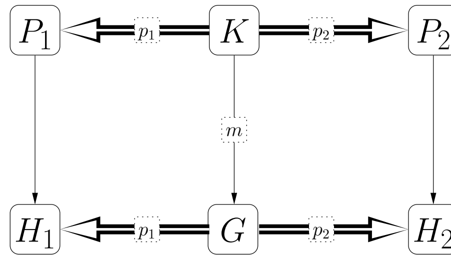



FIG. IX.5 – Complétude des paires critiques

la notion de paire critique.

**Définition 71.** *Une paire critique est une paire de transformations parallèlement dépendantes* $P_1 \overset{p_1,e_1}{\Longleftarrow} K \overset{p_2,e_2}{\Longrightarrow} P_2$ *telle que le couple* $(e_1, e_2)$ *de correspondances appartienne à la classe* $\mathcal{E}$ *(cf. définition 63).*

**Proposition 18.** *Les paires critiques sont complètes. Cela veut dire que si l'on se donne une paires de transformations parallèlement dépendante* $H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$ *, alors il existe une paire critique* $P_1 \overset{p_1,e_1}{\Longleftarrow} K \overset{p_2,e_2}{\Longrightarrow} P_2$ *et un morphisme* $m : K \to G$ *dans la classe* $\mathcal{N}$ *qui permettent de construire les diagrammes d'extension de la figure IX.5.*

*Démonstration.* On commence par factoriser les morphismes $m_1 : L_1 \to G$ et $m_2 : L_2 \to G$ grâce à la propriété de factorisation en $\mathcal{E} - \mathcal{N}$-paires (*cf.* proposition 16). On obtient donc un graphe $K$ et les morphismes $e_1 : L_1 \to K$, $e_2 : L_2 \to K$ et $m : K \to G$ appartenant à la classe $\mathcal{N}$ tels que $m_1 = m \circ e_1$ et $m_2 = m \circ e_2$. On peut alors appliquer le principe de restriction de transformations (*cf.* théorème 11) pour arriver aux diagrammes d'extensions voulus. De plus, en raisonnant par l'absurde, il est simple de prouver que la paire de transformations $P_1 \overset{p_1,o_1}{\Longleftarrow} K \overset{p_2,o_2}{\Longrightarrow} P_2$ est parallèlement dépendante. □

**Définition 72.** *Une paire critique* $P_1 \overset{p_1,o_1}{\Longleftarrow} K \overset{p_2,o_2}{\Longrightarrow} P_2$ *est dite strictement confluente, si l'on a :*

1. *la paire critique est confluente, i.e. il existe des transformations* $P_1 \overset{*}{\Longrightarrow} K'$ *et* $P_2 \overset{*}{\Longrightarrow} K'$ *de empans dérivés respectivement der(* $P_1 \overset{*}{\Longrightarrow} K'$ *)* $=$ $P_1 \overset{v_3}{\longleftarrow} N_3 \overset{w_3}{\longrightarrow} K'$ *et der(* $P_2 \overset{*}{\Longrightarrow} K'$ *)* $=$ $P_2 \overset{v_4}{\longleftarrow} N_4 \overset{w_4}{\longrightarrow} K'$ *;*

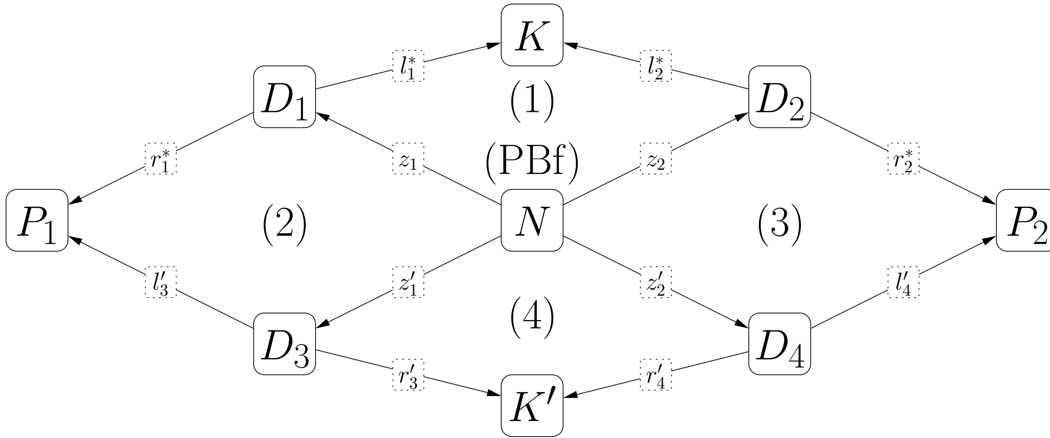



Fig. IX.6 – Le caractère strict

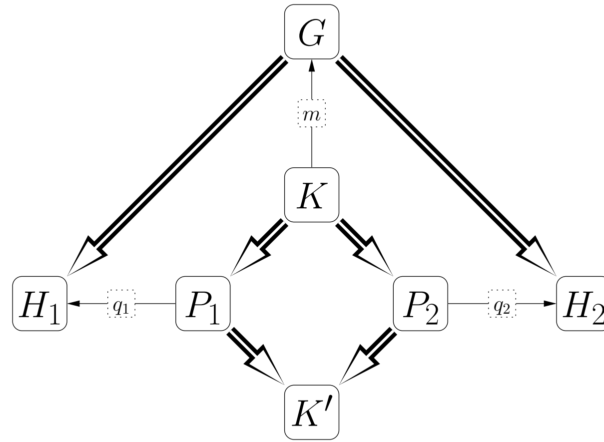



Fig. IX.7 – Les diagrammes d'extension pour la confluence

2. *si, dans le diagramme de la figure IX.6, on note $N$ le pullback faible du carré* (1)*, alors il doit exister les morphismes $z_3$ et $z_4$ tels que les carrés* (2)*,* (3) *et* (4) *commutent.*

**Remarque 32.** L'idée intuitive de cette dernière condition est que l'objet $N$ qui est le plus grand sous-graphe préservé par chaque transformation de la paire critique doit aussi être préservé par les transformations $P_1 \overset{*}{\Longrightarrow} K'$ et $P_2 \overset{*}{\Longrightarrow} K'$ .

# 3 Confluence locale

**Théorème 15.** *Un système de transformation sera localement confluent si toutes ses paires critiques sont strictement confluentes.*

*Démonstration.* On se donne une paire de transformations directes $H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$ .

Si ces deux transformations sont parallèlement indépendantes, alors on sait qu'elles sont localement confluentes (*cf.* théorème 14). Si elles sont dépendantes, la proposition précédente nous assure de l'existence d'une paire critique $P_1 \overset{p_1,o_1}{\Longleftarrow} K \overset{p_2,o_2}{\Longrightarrow} P_2$ et d'un morphisme $m : K \to G$ qui nous donne les diagrammes d'extension de la figure IX.7 (l'hypothèse de confluence locale pour les paires critiques nous permet de déduire de rajouter sur le schéma le graphe $K'$). De plus, la confluence stricte de la paire critique nous permet de développer le diagramme pour obtenir la figure IX.8. On écrit alors maintenant le pushout initial associé au morphisme $m : K \to G$ : on a

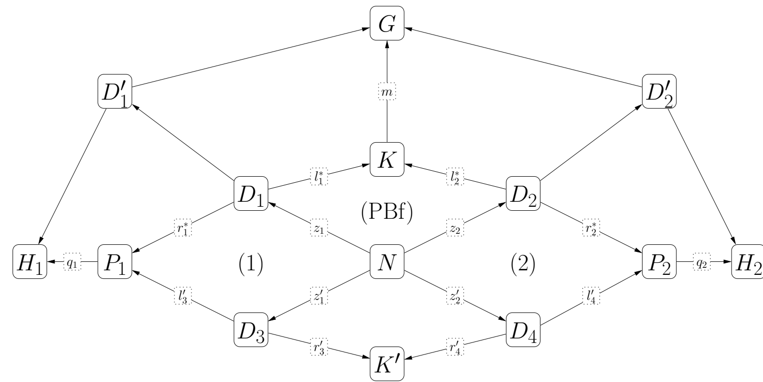



FIG. IX.8 – Le diagramme développé pour la confluence

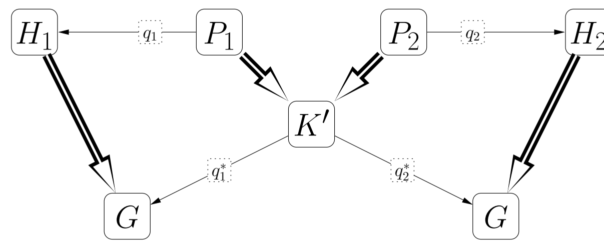



FIG. IX.9 – Construction des extensions

donc deux graphe $B$ et $C$ avec un morphisme $b$ entre $B$ et $K$. Le théorème 13 nous donne alors l'existence de morphismes $b_1 : B \to N_1$ et $b_2 : B \to N_2$ tels que $v_1 \circ b_1 = b$ et $v_2 \circ b_2 = b$.

On aimerait maintenant pouvoir factoriser les morphismes $b_1$ et $b_2$ par le graphe $N$. Mais ce dernier n'est pas un vrai pullback dans notre catégorie, mais seulement un pullback faible. Cela va tout de même suffire car le graphe $B$, comme le graphe $N$, sont des graphes simples (*i.e.* sans attributs) ; donc en utilisant le foncteur d'oubli pour revenir dans la catégorie des graphes simples, le graphe $N$ va alors devenir un véritable pushout pour les morphismes $\mathcal{F}(v_1)$ et $\mathcal{F}(v_2)$. On peut alors construire un morphisme entre les graphes $\mathcal{F}(B)$ et $\mathcal{F}(N)$ et ainsi, en revenant dans notre catégorie de graphes attribués, on obtient le morphisme $b_3 : B \to N$.

Par ailleurs, on sait d'après le lemme 4 sur la propagation des pushouts initiaux que $B$ est l'objet frontière de $P_1$ et de $P_2$ ; la frontière étant respectivement $w_1 \circ b_1$ et $w_2 \circ b_2$. De plus, le morphisme $b_3$ permet de construire un morphisme entre $B$ et $N_3$ qui, grâce à la commutativité du carré (2), rend consistent le morphisme $q_1$ avec la transformation $t_1$. On a le même résultat pour le morphisme $q_2$ vis-à-vis de la transformation $t_2$. En construisant les extensions correspondantes, on obtient donc la figure IX.9.

Il reste donc à montrer que $G'_1 = G'_2$ et que les morphismes qui le relient à K' sont eux aussi égaux. Pour cela, on se sert encore des propriétés du pushout initial. Toujours d'après le lemme 4, les deux carrés de la figure IX.10 sont des pushouts initiaux. Or, par hypothèse, on a $w_3 \circ z_3 = w_4 \circ z_4$.

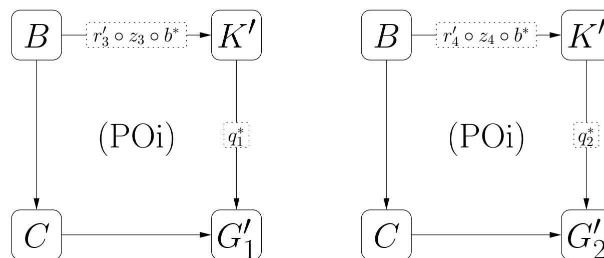



FIG. IX.10 – Les pushouts initiaux nouvellement construits

En conséquence, grâce à l'unicité du pushout, on obtient exactement ce que l'on veut. □

# Chapitre X

# La terminaison

On présente ici un jeu de critères pour déterminer si un système de transformation de graphes typés attribués est terminant ou non.

Comme ce problème est indécidable en général, on le restreint ici à une classe très particulière de système de transformation. En particulier, on ne traitera ici que des systèmes typés par un graphe attribué $TG$.

L'idée directrice pour ce critère est reprise de [23] : on trie toutes les règles de transformation du système en couche selon leur ordre d'application : on appliquera les règles de la couche 1 autant que possible puis, on passe à la seconde couche sans possibilité de revenir à la précédente et ainsi de suite... On obtient ainsi $k_0$ couches de transformations.

De plus, à chaque élément $x$ du graphe type (un arc, un sommet ou un type d'attributs) vont être associé une couche de création $cl(x)$ et une couche de destruction $dl(x)$ vérifiant

$$0 \leq cl(x) < dl(x) \leq k_0 + 1.$$

Cela indique qu'un élément de type $x$ ne pourra être créé qu'entre les couches 1 et $cl(x)$ (si $cl(x) = 0$, alors aucun élément de ce type ne pourra être créé dans ce système de transformation) et qu'il ne pourra être effacé qu'entre les couches $dl(x)$ et $k_0$. De plus, si l'élément est un attribut, sa valeur ne pourra être modifiée qu'entre les couches $cl(x)$ et $dl(x)$ (exclus).

Le système de transformation termine si et seulement si chaque strate (entre 1 et $k_0$) termine.

## 1 Différents types de strates de règles

Les couches de règles de transformations sont alors classées dans trois catégories suivant leur propriétés :

- couche de destruction ;
- couche de création ;
- couche de calcul.

**Définition 73.** *On dira que la couche $k$ est une couche de destruction si chaque règle $p$ de la couche efface au moins un élément (un sommet, un arc ou bien un attribut).*

**Définition 74.** *On dira que la couche $k$ est une couche de création si chaque règle $p$ de la couche rajoute un ou plusieurs éléments au graphe. Plusieurs conditions sont de plus imposées aux règles d'une telle couche :*

- *on demande à ce que la partie gauche de la règle soit sans effet sur la structure du graphe (i.e. aucun effacement), cela implique que $\mathcal{F}(l_p) = Id$ ;*
- *si un élément de type $x$ se trouve dans la parie gauche d'une règle, alors le nombre $cl(x)$ devra être inférieur au numéro de la couche. Cela implique qu'aucun élément de type $x$ ne pourra être créé au cours de l'exécution des transformations de cette couche ;*
- *enfin, la règle $p$ doit posséder au moins une condition d'application négative pour éviter de créer indéfiniment des élément avec le même règle appliquée au même endroit (on précisera plus tard les propriétés que doivent vérifier ces NAC).*

**Définition 75.** *On dira que la couche $k$ est une couche de calcul si pour chaque règle $p$ de la couche, la structure du graphe est inchangée au cours de l'application de la règle, i.e.* $\mathcal{F}(l_p) = \mathcal{F}(r_p) = Id$.

*De plus, pour travailler plus facilement sur la terminaison, on demande à ce que toute les fonctions de calcul de la partie droite de la règle soient égales à l'identité : on ne peut donc que faire de la copie d'attributs lors du second pushout. Cette condition n'est en réalité pas restrictive car les calculs intéressants (i.e. l'application d'une fonction non bijective) n'est possible que lors du premier pushout.*

Le fait que toutes les règles puissent se répartir d'une telle manière n'est pas automatique. En réalité, les système de transformations qui s'écrivent sous cette forme représente une petite partie des système de transformation. Pour autant, une telle approche, même si elle est restrictive, permet de traiter certains cas intéressants, en particulier la plupart des cas reliés aux transformations de modèles.

## 1.1 Terminaison d'une couche de destruction

L'application d'une couche de destruction termine toujours.

En effet, du fait de la condition $cl(x) < dl(x)$, les éléments effacés par une des règles de la couche ne pourront pas être recréées par d'autres règles. Donc, en comptant le nombre total d'éléments pouvant être effacés par une des règles dans le graphe de départ, on trouve un majorant du nombre de transformations possibles dans cette couche. Et comme on ne travaille qu'avec des graphes finis, ce nombre est lui aussi fini.

## 1.2 Terminaison d'une couche de calcul

Comme la structure du graphe reste inchangée au cours de l'exécution de cette couche, il suffit de travailler sur les attributs. L'idée est alors de trouver une fonction strictement décroissante envoyant ces attributs vers un ordre bien fondée et d'avoir une condition d'application négative correspondante au plus petit élément pour l'ordre.

## 1.3 Terminaison d'une couche de création

Soit $p$ un règle de transformation de graphes attribués typés appartenant à une couche de création. Si cette règle ne crée que des attributs et aucun élément structurel du graphe, alors on sait qu'on ne pourra appliquer qu'un nombre fini de fois cette règle sur un graphe donné car le nombre d'attributs attachés à un sommet (ou à un arc) est limité par le graphe type. Ce type de règle ne pose donc aucun problème pour la terminaison.

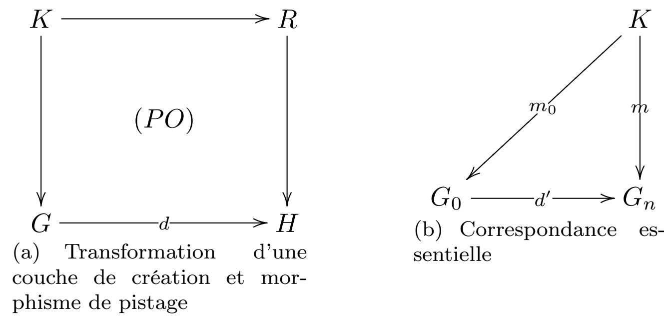
Maintenant si la règle $p$ rajoute des éléments structurels au graphe, l'idée pour ne pas avoir un graphe qui grossit indéfiniment va être de s'assurer que si la règle $p$ est appliquée une première fois avec une correspondance $m$, si on essaye une seconde fois de l'appliquer avec la même correspondance, alors une condition d'application négative empêchera la transformation de se poursuivre. Cette NAC ne s'intéressera d'ailleurs qu'à la structure et pas aux attributs. Il faut pour cela imposer quelques conditions supplémentaires sur ce type de règle pour s'assurer des propriétés voulues :

- la règle $p$ ne doit effacer aucun élément structurel, *i.e.* $\mathcal{F}(K) = \mathcal{F}(L)$ et $\mathcal{F}(l) = Id$ et pour chaque espace d'attributs du graphe $L$, le morphisme $l$ possède une relation nécessaire associé à cet espace ;
- si un élément de type $x$ se trouve dans la parie gauche d'une règle, alors le nombre $cl(x)$ devra être inférieur au numéro de la couche ;
- la règle $p$ possède une condition d'application négative $n : L \to N$ telle qu'il existe un morphisme structurellement injectif $n' : N \to R$ vérifiant $n' \circ n = r$.

Avec ces restrictions, une couche de création termine toujours. Pour le montrer, on introduit la notion de correspondance essentielle.

### *1.3.21 Correspondance essentielle*

On se place ici dans une couche de création. On suppose de plus que pour les règles de création, le morphisme structurel de la partie droite est injective. Cela interdit alors d'avoir des règles qui



(a) Transformation d'une couche de création et morphisme de pistage

(b) Correspondance essentielle

FIG. X.1 – Morphisme de pistage

fusionnent des éléments (*cf.* exemple section 5.1 pour voir de telles règles).

**Définition 76.** *Soit $p$ un règle de transformation d'une telle couche. Structurellement, cette règle est entièrement caractérisée par sa partie droite : $r : K \to R$. Soit $G$ un graphe attribué et $m$ une correspondance entre la règle $p$ et le graphe $G$.*

*Comme on n'efface aucun élément du graphe, la condition de collage est automatiquement vérifié et, sous réserve de compatibilité des attributs, cette transformation est automatiquement constructible ; on obtient donc une transformation directe $G \Longrightarrow H$ via le couple $(p, m)$ donnée par le pushout de la figure X.1(a). En se limitant à la structure, on trouve donc un morphisme structurel entre le graphe-source et le graphe modifié.*

*Le morphisme structurel $d$ sera appelé le morphisme de pistage de la transformation directe $(p, m)$.*

**Définition 77.** *Soit $G_0 \Longrightarrow G_n$ une transformation dans la couche de création. Cette transformation possède donc un morphisme de pistage $d' : G_0 \to G_n$. On dira qu'une correspondance $m$ entre un graphe $K$ et le graphe $G_n$ possède une correspondance essentielle s'il existe un morphisme $m_0 : K \to G_0$ appartenant à la classe $\mathcal{N}$ et vérifiant $\mathcal{F}(d') \circ \mathcal{F}(m_0) = \mathcal{F}(m)$ (voir figure X.1(b)).*

**Remarque 33.** On remarque que si $m$ possède une correspondance essentielle alors elle est unique du fait de l'injectivité structurelle de $d'$.

**Proposition 19.** *Dans une couche de création, toutes les correspondances possèdent une correspondance essentielle.*

*Démonstration.* Dans une couche de création, on sait que les règles n'effacent ni ne fusionnent aucun élément. De plus, on sait que si un élément se trouve dans la partie gauche d'une des règles de la strate, alors aucune des règles de la couche ne pourra créer un élément de ce type (voir la définition 74).

À partir de ces données, si on a une transformation $G_0 \Longrightarrow G_n$ dont on connaît le morphisme de pistage $d$ et une correspondance $m : K \to G_n$, on est alors sûr que l'on peut trouver un antécédent unique par $d$ pour tous les éléments structuraux de l'image de $K$ par $m$. On peut donc facilement définir un morphisme structurel entre $K$ et $G_0$. □

**Proposition 20.** *On ne peut appliquer qu'une seule fois une règle avec la même correspondance essentielle.*

*Démonstration.* On veut appliquer la règle $p$ au graphe $G_n$ avec la correspondance $m'$ et la correspondance essentielle $m_0$. On suppose que cette même règle ait déjà été appliquée une fois au cours de la transformation avec une correspondance $m$ entre $K$ et $G_l$ et avec la même correspondance essentielle.

On peut alors montrer que nécessairement la correspondance $m'$ ne vérifie pas la condition d'application négative attaché à la règle $p$. On sait que pour cette NAC, il existe un morphisme $n'$ entre le graphe $N$ la représentant et la partie droite $R$ de la règle tel que $n' \circ n = r$. Ceci nous permet de construire un morphisme entre $N$ et $G_{l+1}$, puis en utilisant le morphisme de pistage entre $G_{l+1}$ et $G_n$, on trouve un morphisme entre $N$ et $G_n$ qui vient bloquer l'application de la règle $p$ avec la correspondance $m'$. □

**Proposition 21.** *Une couche de création termine toujours.*

*Démonstration.* Pour un graphe $G$, on définit le nombre $c(G)$ de correspondances possibles pour toutes les règles de la couche pour lesquelles les NAC sont vérifiées. On montre très facilement à l'aide des deux propositions précédentes que ce nombre décroît strictement à chaque application d'une des règles de la transformation :

$$c(G_0) > c(G_1) > \cdots > c(G_n).$$

On en conclut donc qu'une couche de création termine toujours. □

*Exemple.* Le terminaison de l'exemple présenté dans le chapitre VI de traductions des diagrammes d'activité UML vers les réseaux de Petri peut être démontré grâce à cette approche [22]. Les deux premières règles vont se trouver dans deux couches de création différentes et elles sont bien accompagnées de conditions d'application négatives comme demandé dans les critères de terminaison. Enfin, la dernière règle sera placée dans une couche de destruction.

# Chapitre XI

# Vers l'implantation

Dans ce chapitre, nous présentons quelques systèmes implantant des récritures de graphes. Nous discutons ensuite des choix techniques pour l'implantation informatique de l'approche DPoPb.

## 1 Quelques logiciels actuels de transformations de graphes

### 1.1 PROGRES

PROGRES [66, 59], apparu en 1986, est sans doute l'un des premiers systèmes permettant de faire des transformations de graphes. Il a été développé en Allemagne à l'université d'Aachen.

Le logiciel repose sur l'approche orientée logique des grammaires de graphes : une règle est décrite de manière visuelle par une partie gauche et une partie droite. Par aileurs, il est possible de définir des règles plus complexes à l'aide de structures de contrôle textuelles et d'un langage de type OCL [77] pour formuler des contraintes sur les chemins dans les graphes. Le couplage avec la théorie n'est ici pas évident ; il est alors difficle d'enchaîner les concepts théoriques et la programmation.

### 1.2 VIATRA2

VIATRA2 [75] est un plugin Eclipse développé depuis 2005 à l'université de Budapest. Le logiciel utilise un langage de modélisation particulier pour répresenter les modèles appelé VPM [76] qui va englober les modèles UML afin d'avoir un langage plus général. Pour définir les règles de transformations, l'utilisateur peut utiliser des motifs récursifs et des motifs de négations (l'équivalent des conditions d'application négatives) avec une profondeur quelconque de négation. Un des avantages du système est la possiblité d'ordonnancement dans l'application des règles à l'aide d'*abstract state machine*. Ce dernier point permet de donner une réponse pour l'implantation aux questions posées dans le chapitre X.

### 1.3 AGG

AGG [72, 73] est le logiciel développé par l'équipe berlinoise à l'origine de l'approche adhésive HLR. La philosophie de ce logiciel est donc de suivre ce système dont on a largement discuté dans cette thèse. Les détails de l'implantation du noyau peuvent se trouver dans [78].

La définition des règles de transformation dans le logiciel se fait de manière graphique : on dessine les parties gauches et droites des règles de transformation en précisant à l'aide de numéros sur chacun des éléments des graphes l'action des morphismes. Il est par ailleurs possible d'ajouter autant de conditions d'application voulues pour chacune des règles et de classer les règles par couche afin de définir la séquence d'application des règles (en suivant les idées de la preuve de terminaison des systèmes de récriture). Par ailleurs, une des propriétés intéressantes du logiciel est sa capacité à analyser les paires critiques d'un système. Des exemples de systèmes de récriture se trouvent sur la page internet du logiciel : `http://tfs.cs.tu-berlin.de/agg/examples_V122/index.html`.

Par contre, l'implantation s'éloigne de la théorie sur plusieurs points. Premièrement, comme on l'a vu dans le chapitre III, les graphes manipulés sont théoriquement infinis dès que l'on travaille avec une $\Sigma$-algèbre infinie, ce qui est le cas avec les entiers naturels par exemple. Comme il n'est pas facile de travailler avec des objets infinis avec un ordinateur, dans le logiciel AGG, ces graphes sont simplifiés et on ne considère que les sommets avec les attributs réellement utilisés pour définir le graphe ; on se ramène ainsi à des graphes finis. Deuxièmement, l'implantation du calcul des attributs s'éloigne de la théorie en autorisant n'importe quelle fonction acceptable en Java. Ce choix permet donc d'augmenter l'expressivité du système car ainsi, on n'est plus limité aux termes de la $\Sigma$-algèbre pour travailler avec les variables, mais on sort largement du cadre théorique, la sémantique de Java n'étant pas accessible[1].

# 2 Quels choix pour l'implantation du système DPoPb ?

Deux caractériques importantes doivent être présentes dans le langage choisi pour implanter le système DPoPb. Notre approche se basant sur la théorie des types inductifs, il sera intéressant de travailler avec un langage, d'une part, fortement typé pour être certain de toujours avoir des résultats cohérents et d'autre part, permettant de définir des structures récursives afin de traduire facilement les constructions employées pour la définition des graphes et des morphismes. Par ailleurs, les transformations reposant sur des constructions catégoriques, il est nécessaire d'utiliser un langage possédant des bibliothèques ou des facilités pour travailler avec les catégories. Il doit ainsi être possible de définir des morphismes, de vérifier quelques propriétés dessus (monomorphisme, isomorphisme...) et d'appliquer les constructions catégoriques qui nous intéressent (principalement le pushout). De cette manière, les transformations de graphes pourront véritablement suivre la théorie et seront donc vérifiées *a priori*.

Les langages fonctionnels ou les systèmes basés sur ces derniers semblent alors de bons condidats pour l'implantation.

## 2.1 Coq

Coq [18, 6] est un outil d'aide à la preuve. Il existe de nombreux logiciels de ce type, certains automatiques et d'autres interactifs : on peut par exemple citer Isabelle [56], PVS [57] ou encore SAD [52]. Notre choix s'est porté sur le logiciel Coq car il est basé sur un formalisme nommé le « Calcul des Constructions Inductives » qui est à la fois une logique et un langage de programmation purement fonctionnelle. En conséquence, la manipulation des types inductifs est un des points forts de Coq : on peut facilement définir une grande variété de types inductifs et construire des fonctions récursives au-dessus de ces types pour ensuite vérifier des propriétés sur ces fonctions.

```
Inductive piece : Set := Pile : piece | Face : piece.
Inductive nat : Set := 0 : nat | S : nat -> nat
```

On remarque avec la syntaxe des types dans Coq qu'un type fini est bien un cas particulier d'un type inductif. Pour les fonctions récursives, la syntaxe rappelle la construction par point fixe d'une telle fonction :

```
Fixpoint factorielle (n:nat) : nat :=
  match n with
      0 => 1
   |  S p => (S p)*(factorielle p)
  end.
```

Par ailleurs, Coq intègre aussi un langage de programmation fonctionnelle qui, grâce à un mécanisme d'extraction, permet d'engendrer automatiquement des programmes certifiés en Objective Caml, Haskell ou Scheme à partir de preuves de leurs spécifications.

[1] Le problème de la sureté de Java a beaucoup été discuté dans la littérature. Les problèmes de consistance du langage posent alors des questions sur l'utilisation sans vérification interne comme module externe de Java dans un système comme AGG

De plus, deux contributions faites par les utilisateurs écrites respectivement par Saïbi [64] et Simpson [69] sont disponibles pour manipuler les concepts catégoriques avec le logiciel. On peut aussi citer le travail récent de Spiwack [15] qui s'intéresse à la formalisation de la théorie des catégories dans ce système logique.

Même si Coq n'est pas un logiciel parfaitement adapté pour faire des calculs, il est très puissant pour la vérification. On peut donc utiliser ce système pour démontrer certaines propriétés sur les règles de transformation ou bien pour tester si une transformation est correcte ou non : par exemple, savoir si le résultat d'une transformation est bien conforme à la sémantique du système.

## 2.2 Ocaml

Le langage Ocaml (`http://caml.inria.fr/`) est un langage de programation fonctionnelle fortement typé. Dans ce langage [10], il est facile de définir des nouveaux types de données, qu'ils soient finis :

```
# type piece = Pile | Face;;
```

ou bien inductifs :

```
# type int_or_char_list =
       Nil
    |  Int_cons of int * int_or_char_list
    |  Char_cons of char * int_or_char_list ;;
```

Bien entendu, à partir de ces types récursifs, il est possible de définir et manipuler des fonctions récursives :

```
# let rec factorielle = function
       0 -> 1
    |  n -> n * (factorielle (n-1)) ;;
```

De plus, il existe des développements pour manipuler les concepts catégoriques conçus pour les langages de la famille ML (dont Ocaml fait partie). On pourra voir à ce sujet le livre de Rydeheard et Burstall [63] qui offre un panorama assez complet du travail avec les catégories pour avoir plus de détails.

## 2.3 Haskell

Haskell (`http://www.haskell.org/`) est une des voies qui nous semble la plus prometteuse. Là aussi, on a affaire à un langage fortement typé. Il est là aussi possible de définir tous les sortes de types qui nous intéressent et les fonctions récursives qui les accompagnent :

```
data Piece = Pile | Face
data Arbre a = Feuille a | Branche (Arbre a) (Arbre a)

factorielle 0 = 1
factorielle n = n * factorielle (n-1)
```

De plus, ce langage de programmation repose sur le concept de monades. Ces objets assurent au langage d'être fonctionnel pur (c'est-à-dire que n'importe quelle opération peut être codée de manière fonctionnelle, même celles qui habituellement nécessitent la séquentialité comme par exemple les entrées-sorties). Ces monades semblent bien adaptées pour manipuler les catégories. Schneider [65] a proposé quelques pistes pour l'implantation avec ce langage d'un système de transformations de graphes simples. Les idées développées dans cet article peuvent servir de point de départ pour notre implantation. Pour cela, il commence par définir de manière générale une catégorie avec ses briques de bases. En voici les lignes directrices : d'abord les objets et les morphismes, puis le domaine, le codomaine d'une morphisme, l'identité de chaque objet et la composition des morphismes :

```
module Category where
  class (Eq o) => Obj o
  class (Eq m) => Mor m where

  class Category c where
    dom :: (Obj o,Mor m) => c o m -> m -> o
    codom :: (Obj o,Mor m) => c o m -> m -> o
    ident :: (Obj o,Mor m) => c o m -> o -> m
    compose :: (Obj o,Mor m) => c o m -> m -> m -> m
    composable :: (Obj o,Mor m) => c o m -> m -> m -> Bool
```

Par la suite, il met en place les opérations catégoriques de base : coproduit et coéqualiseur et enfin pushout dans un second module `Colimits`.

```
data (Obj o, Mor m) =>
     Coproduct o m = Coproduct {cpObj1::  o,
                                cpObj2:: o,
                                cpObj::  o,
                                cpMor1:: m,
                                cpMor2:: m,
                                cpUniv:: m -> m -> m
                              }
data (Obj o, Mor m) =>
     Coequalizer o m = Coequalizer {ceqObj1:: o,
                                    ceqObj2:: o,
                                    ceqObj::  o,
                                    ceqMor1:: m,
                                    ceqMor2:: m,
                                    ceqMor::  m,
                                    ceqUniv:: m -> m
                                   }

data (Obj o, Mor m) =>
     Pushout o m = Pushout {poInt::   o,
                            poObj1::  o,
                            poObj2::  o,
                            poObj::   o,
                            poMor1::  m,
                            poMor2::  m,
                            poMor1b:: m,
                            poMor2b:: m,
                            poUniv::  m -> m -> m
                           }

pushout c m1 m2
   = if (dom c m1) /= (dom c m2) then
        error "Pushout: Domains not identical"
     else Pushout {poInt = dom c m1,
                   poObj1 = obj1,
                   poObj2 = obj2,
                   poObj = obj,
                   poMor1 = m1,
                   poMor2 = m2,
                   poMor1b = compose c r g',
                   poMor2b = compose c r f',
                   poUniv = u
                  }
            where obj1 = codom c m1
                  obj2 = codom c m2
                  cp = coproduct c obj1 obj2
                  f' = cpMor1 cp
```

```
g' = cpMor2 cp
f'' = compose c f' m1
g'' = compose c g' m2
ceq = coequalizer c f'' g''
obj = ceqObj ceq
r = ceqMor ceq
```

La suite de son travail consiste à particulariser (grâce à la relation d'héritage très dévelopée en Haskell) ces concepts généraux pour définir la catégorie des ensembles et enfin la catégorie des graphes simples. Il peut alors implanter la transformation de graphes en suivant l'approche « Double Pushout ».

En partant de ces idées, plusieurs problèmes techniques vont encore se poser pour adapter notre système. Par exemple : comment vont être codées les fonctions de calcul qui vont dans l'autre sens par rapport aux morphismes ? Comment vérifier qu'un morphisme appartient ou non à une des trois classes $\mathcal{N}$, $\mathcal{M}'$ ou $\mathcal{M}''$ que l'on a défini ? Est ce possible d'utiliser la même décomposition coproduit-coéqualiseur pour calculer le pushout dans notre catégorie ou bien faut il construire directement le résultat voulu ?

# Conclusion et perspectives

Au cours de thèse, nous avons discuté d'une approche de transformation de graphes attribués fondée sur la théorie des types et sur la théorie des catégories dans l'optique d'une application aux transformations de modèles. Le travail exposé dans ce mémoire a été réalisé suivant une démarche en plusieurs temps :

- Le premier temps a consisté en une définition du contexte de l'étude qui concerne les différents cadres théoriques de récritures de graphes ; en s'attardant plus particulièrement sur les approches catégoriques par pushout et leurs dérivées permettant d'ajouter de l'information sur les graphes qui nous ont servi de base pour la suite.
- Dans un deuxième temps, nous avons proposé les définitions et les constructions précises de notre système de récriture de graphes attribués : après avoir décrit les graphes et les morphismes entre ces derniers, le pushout et le pushout-complement ont été introduit de manière constructive. Cela a permis de donner la définition d'un système de récriture dans l'approche DPoPb avec des exemples classiques du MDA.
- Enfin, le troisième et dernier temps a permis de montrer des propriétés intéressantes de l'approche DPoPb telles que des considérations sur la confluence des systèmes ou bien encore la terminaison.

L'étude des systèmes adhésifs HLR a mis en lumière plusieurs défauts dont le principal est le mélange de deux cadres théoriques différents, les ensembles et les $\Sigma$-algèbres, pour traîter d'une part la structure des graphes et d'autre part les attributs. Comme nous l'avons vu, ce choix entraîne des problèmes à la fois théoriques avec l'expressivité limitée des termes des $\Sigma$-algèbres pour définir les calculs sur les attributs lors des transformations, mais aussi pratiques car les graphes théoriquement infinis sont simplifiés dans le logiciel AGG, sortant ainsi du cadre formel.

La principale contribution de cette thèse consiste alors en la proposition d'une approche unifiée pour la catégorie de graphes attribués s'appuyant sur les types inductifs pour définir à la fois la structure des graphes manipulés grâce à des types énumérés, mais aussi les attributs qui viennent se rajouter sur cette structure. Le premier avantage de cette idée réside alors dans la grande liberté offerte pour le choix des attributs portés par les éléments des graphes : les types inductifs, acceptant les arguments fonctionnels dans leur définition, génèrent des espaces plus généraux que les $\Sigma$-algèbres (par exemple les ordinaux). De plus, cette idée nous a permis d'introduire dans les morphismes de graphes attribués des fonctions récursives entre les attributs des deux graphes ouvrant la voie à des calculs complexes lors des transformations de graphes : en effet, les applications définies par récursion mettent à disposition un très grand pouvoir expressif. Contrairement à l'approche adhésive HLR où les calculs sur les attributs étaient codés dans les attributs eux-mêmes, nous avons donc choisi de faire porter par les morphismes l'information concernant les calculs ; ce qui nous a permis de nous passer de nom pour les variables.

Après avoir montré comment dans cette catégorie se construisent les opérations catégoriques de base, le pushout et le pushout-complement, on a ainsi pu définir simplement les règles de transformations et les systèmes de récritures de graphes attribués en reprenant les idées classiques de l'approche algébrique. En suivant alors le même chemin que les approches classiques et l'approche adhésive HLR, on a ainsi réussi à prouver toutes les propriétés essentielles requises pour un système de récriture de graphes attribués : en particulier, l'analyse par paires critiques et la confluence locale. Par ailleurs, on a aussi développé des critères pour conclure à la terminaison d'un système de récriture.

La suite de ce travail se situe naturellement dans le développement d'implantations. Dans ce but, nous avons exploré quelques pistes qu'il faut maintenant approfondir afin d'obtenir un logiciel fonctionnel.

Du côté de la théorie, une question intéressante à appronfir serait d'essayer de généraliser la définition des correspondances afin de pouvoir appliquer une même règle de transformation à plusieurs endroit du graphe source en même temps. Ainsi, lors d'une transformation de modèles, on pourra repérer dans le graphe source tous les sommets représentant le même concept et appliquer une seule fois la règle correspondante. Pour autant, cette généralisation n'est pas immédiate car plusieurs problèmes peuvent survenir, par exemple :

- Que se passe-t-il si deux sous-graphes du graphe $G$ à récrire sont d'intersection non vide ?
- Comment gérer les différentes copies d'espaces d'attributs sur un même sommet ?

Toutes ces questions constituent une suite naturelle à notre étude.

# Annexe A

# Théorie des categories

Nous présentons ici les bases de la théorie des catégories qui nous serviront par la suite pour définir des transformations de graphes. En particulier, on développe la notion de pushout qui est centrale dans notre travail.

Introduction historique des catégories.

## 1 Catégories

**Définition 78.** *Une catégorie* **C** *est la donnée des éléments suivants :*
- *d'une collection, dont les éléments sont appelés objets ;*
- *pour chaque paire d'objets $A$ et $B$, d'une collection $Hom_{\mathbf{C}}(A, B)$, dont les éléments $f$ sont appelés morphismes (ou flèches) entre $A$ et $B$, et sont parfois notés $f : A \to B$.*

*Les collections de morphismes doivent en plus satisfaire aux conditions suivantes :*
- *pour chaque objet $A$, il existe un morphisme $\mathrm{id}_A : A \to A$, appelé identité sur $A$ ;*
- *pour toute paire de morphismes $f : A \to B$ et $g : B \to C$, il existe un morphisme $g \circ f : A \to C$, appelé composé de $f$ et $g$ vérifiant que la composition est associative et que les morphismes identités sont neutres pour la composition.*

**Remarque 34.** La notion de collection est ici très vague. Elle indique que l'on peut considérer des objets plus grand que les ensembles (par exemple, la classe de tous les ensembles).

*Exemple.* On trouve des exemples de catégories partout en mathématiques ;

| Catégorie | Objets | Flèches |
|---|---|---|
| **Ens** | les ensembles | les applications |
| **Top** | les espaces topologiques | les applications continues |
| **Grp** | les groupes | les homomorphismes |
| $\mathbf{Vect}_{\mathbf{K}}$ | les $K$-espaces vectoriels | les applications linéaires |

## 2 Propriétés des flèches

**Définition 79.** *Une flèche $f : A \to B$ sera qualifiée de monomorphisme si elle vérifie la propriété suivante :*

*pour tout objet $A'$ et toute paire de morphismes $g$ et $h$ de $A'$ vers $A$,*
*si $f \circ g = f \circ h$ alors $g = h$.*

*Une flèche $f : A \to B$ sera qualifiée d'épimorphisme si elle vérifie la propriété suivante :*

*pour tout objet $B'$ et toute paire de morphismes $g$ et $h$ de $B$ vers $B'$,*
*si $g \circ f = h \circ f$ alors $g = h$.*

**Définition 80.** *Une flèche $f : A \to B$ sera qualifiée d'isomorphisme si elle vérifie la propriété suivante :*

*il existe une flèche* $g : B \to A$ *telle que* $g \circ f = \mathrm{Id}_A$ *et* $f \circ g = \mathrm{Id}_B$.

*Exemple.* Dans la catégorie **Ens**, les monomorphismes sont les applications injectives, les épimorphismes sont les applications surjectives et les isomorphismes sont les bijections.

**Remarque 35.** En général, un morphisme qui est à la fois un monomorphisme et un épimorphisme n'est pas nécessairement un isomorphisme.

# 3 Diagrammes commutatifs

**Définition 81.** *Un diagramme dans une catégorie* **C** *est la donnée d'un graphe orienté où les sommets représentent des objets de la catégorie et les arcs des morphismes.*

**Définition 82.** *Dans une catégorie* **C**, *un diagramme est dit commutatif si pour tout couple d'objets* $A$ *et* $B$, *tous les chemins de* $A$ *vers* $B$ *dans le diagramme déterminent une même et unique flèche.*

*Exemple.* Dire que le diagramme ci-dessous est commutatif revient à dire que $g \circ f = t \circ s$.

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle s}\downarrow & & \downarrow{\scriptstyle g} \\ C & \xrightarrow[t]{} & D \end{array}$$

# 4 Propriétés universelles

Dans la théorie des catégories, beaucoup de constructions se font à l'aide des propriétés universelles. Celles ci assurent l'existence et l'unicité de la construction. Elles sont souvent formulés sous la forme «quel que soit… il existe un unique… tel que…». Voici quelques exemples de telles constructions qui nous sont utiles.

## 4.1 Objets initiaux et terminaux

**Définition 83.** *Un objet* 0 *d'une catégorie* **C** *est dit initial si pour tout objet* $A$, *il existe une unique flèche de* 0 *vers* $A$.

**Définition 84.** *Un objet* 1 *d'une catégorie* **C** *est dit terminal si pour tout objet* $A$, *il existe une unique flèche de* $A$ *vers* 1.

**Proposition 22.** *S'ils existent, les objets initiaux et terminaux sont uniques à isomorphisme près.*

*Exemple.* Dans la catégorie **Ens**, l'ensemble vide est le seul objet initial et les singletons sont tous des objets terminaux (isomorphes entre eux).

## 4.2 Pushout

Intuitivement, le pushout est une construction consistant à coller deux objets de la catégorie le long d'un modèle (un troisième objet).

**Définition 85.** *Soient* $A$, $B$ *et* $C$ *trois objets de la catégorie* **C** *et* $b : A \to B$ *et* $c : A \to C$ *deux flèches de cette catégorie. Le pushout de* $b$ *et* $c$ *sera donné par*

– *un objet* $D$ ;
– *deux flèches* $b' : C \to D$ *et* $c' : B \to D$

*vérifiant la propriété universelle suivante : pour tout objet* $E$ *appartenant à la catégorie* **C**, *pour toutes flèches* $f : B \to E$ *et* $g : C \to E$ *tels que* $f \circ b = g \circ c$, *alors il existe une flèche* $d : D \to E$

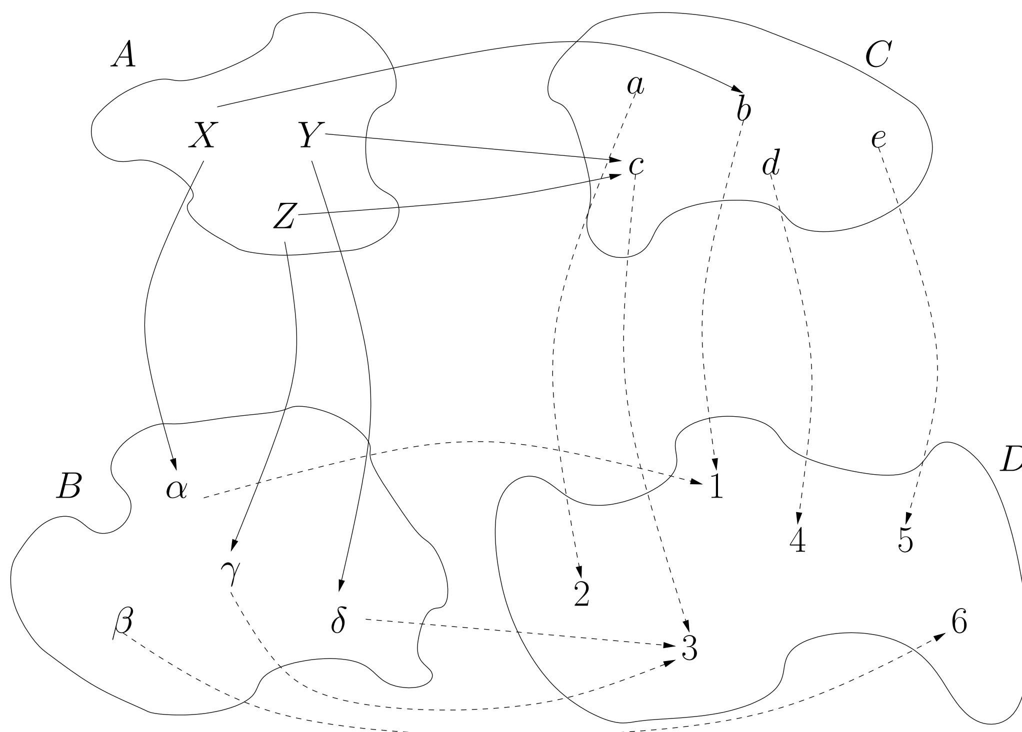
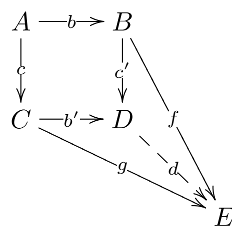



FIG. A.1 – Une construction concrète de pushout dans la catégorie **Ens**

*qui factorise $f$ et $g$, c'est-à-dire $f = d \circ c'$ et $g = d \circ b'$.*




**Remarque 36.** Le pushout, s'il existe, est unique à isomorphisme près.

*Exemple.* Dans la catégorie **Ens**, le pushout de deux applications existe toujours. On donne maintenant sa construction explicite. On commence par définir une relation d'équivalence $\tilde{\mathcal{R}}$ sur l'ensemble $A$ en prenant le fermeture transitive de la relation $\mathcal{R}$ suivante

$$x\mathcal{R}y \text{ si } b(x) = b(y) \text{ ou } c(x) = c(y).$$

L'ensemble $D$ est alors défini par

$$D = A/_{\tilde{\mathcal{R}}} \cup B \backslash b(A) \cup C \backslash c(A)$$

et les morphismes $b'$ et $c'$ par

$$b' : \begin{array}{ccc} C & \longrightarrow & D \\ x & \longmapsto & \begin{cases} [a] \text{ si } \exists a \in A | c(a) = x \\ x \text{ sinon} \end{cases} \end{array} \quad \text{et} \quad c' : \begin{array}{ccc} B & \longrightarrow & D \\ x & \longmapsto & \begin{cases} [a] \text{ si } \exists a \in A | b(a) = x \\ x \text{ sinon.} \end{cases} \end{array}$$

**Lemme 5.** On se donne le diagramme commutatif suivant dans la catégorie $\mathbf{C}$ :

$$\begin{array}{ccccc} A & \longrightarrow & B & \longrightarrow & C \\ \downarrow & (1) & \downarrow & (2) & \downarrow \\ D & \longrightarrow & E & \longrightarrow & F \end{array}$$

Alors

- si (1) et (2) sont des pushouts, alors la composition (1) + (2) est aussi un pushout ;
- si (1) et (1) + (2) sont des pushouts, alors le carré (2) est aussi un pushout.

*Démonstration.* On suppose que les carrés (1) et (2) sont des pushouts. Pour montrer que (1) + (2) est aussi un pushout, on cherche à démontrer que la propriété universelle du pushout est vraie pour le diagramme $ACDF$. Pour cela, on se donne donc un objet $G$ appartenant à la catégorie $\mathbf{C}$ ainsi que deux flèches de $D$ vers $G$ et de $C$ vers $G$ tels que le diagramme $ACDG$ soit commutatif. La composition des flèches nous permet d'obtenir une flèche entre $B$ et $G$ tel que le carré $ABDG$ commute. On peut alors appliquer la propriété universelle du pushout (1) pour obtenir une unique flèche entre $E$ et $G$. On sait alors que le carré $BCEG$ est commutatif et la propriété universelle du pushout (2) nous permet d'obtenir là flèche recherchée entre $F$ et $G$.

Pour le deuxième point, on veut prouver la propriété universelle des pushouts pour le diagramme $BCEF$. On se donne donc un objet $G$ et deux flèches de $C$ vers $G$ et de $E$ vers $G$. Par composition des flèches, on obtient une nouvelle flèche de $D$ vers $G$ telle que le carré $ACDG$ soit commutatif. Après application de la propriété universelle du pushout (1) + (2), on obtient l'unique flèche entre $F$ et $G$ factorisant les flèches entre $C$ et $G$ et entre $D$ et $G$. Pour conclure, il reste à montrer que cette flèche factorise bien aussi la flèche entre $E$ et $G$. Pour cela, en composant les flèches $E \to F$ et $F \to G$, on obtient une seconde flèche entre $E$ et $G$. On peut alors conclure en invoquant la propriété universelle du pushout (1). □

Une question naturelle qui intervient lorsque l'on travaille avec les pushouts est d'essayer de renverser la construction :

**Définition 86.** *Soit $b : A \to B$ et $c' : B \to D$ deux flèches dans la catégorie $\mathbf{C}$. On dira que le pushout-complement de $b$ et $c'$ existe si on peut trouver un objet $C$ et deux flèches $c : A \to C$ et $b' : C \to D$ tels que le carré $ABCD$ soit un pushout.*

## 4.3 Pullback

Le pullback est la construction duale du pushout.

**Définition 87.** *Soient $B$, $C$ et $D$ trois objets de la catégorie $\mathbf{C}$ et $b' : C \to D$ et $c' : B \to D$ deux flèches de cette catégorie. Le pullback de $b'$ et $c'$ sera donné par :*

- *un objet $A$ ;*
- *deux flèches $b : A \to B$ et $c : A \to C$*

*vérifiant la propriété universelle suivante : pour tout objet $E$ appartenant à la catégorie $\mathbf{C}$, pour toutes flèches $f : E \to B$ et $g : E \to C$ tels que $c' \circ f = b' \circ g$, alors il existe une flèche $a : E \to A$ qui factorise $f$ et $g$, c'est-à-dire $f = b \circ a$ et $g = c \circ a$.*

$$\begin{array}{ccccc} E & & & & \\ & \overset{a}{\searrow} & & \overset{f}{\searrow} & \\ & g & A & \xrightarrow{b} & B \\ & & \downarrow c & & \downarrow c' \\ & & C & \xrightarrow{b'} & D \end{array}$$

**Remarque 37.** Le pullback, s'il existe, est unique à isomorphisme près.

*Exemple.* Dans la catégorie **Set** des ensembles, le pullback de deux applications est toujours constructible.

# 5 Foncteurs

Intuitivement, un foncteur peut se voir comme une application entre catégories.

**Définition 88.** *Soient* **C** *et* **D** *deux catégories. Un* ***foncteur (covariant)*** $\mathcal{F}$ *de* **C** *vers* **D** *est donné par :*

- *une fonction qui à tout objet* $A$ *de* **C** *associe un objet* $\mathcal{F}(A)$ *de* $B$ *;*
- *pour tout couple d'objets* $A$ *et* $B$ *de la catégorie* **C***, une fonction qui à toute flèche* $f \in \mathrm{Hom}_{\mathbf{C}}(A, B)$ *associe une flèche* $\mathcal{F}(f)$ *appartenant à* $\mathrm{Hom}_{\mathbf{D}}(\mathcal{F}(A), \mathcal{F}(B))$*.*

*Ces fonctions doivent de plus respecter l'identité et la composition.*

**Proposition 23.** *Les foncteurs préservent les isomorphismes.*

*Exemple.* On parle de foncteur d'oubli d'oubli $\mathcal{F} : \mathbf{C} \to \mathbf{D}$ si pour chaque objet et chaque lèche de la catégorie **C**, le foncteur $\mathcal{F}$ «oublie» quelques propriétés de l'élément considéré pour calculer l'image. Par exemple, un foncteur qui est défini sur la catégorie des anneaux **Ring** vers la catégorie de groupes **Grp** sera qualifié de foncteur d'oubli si son rôle consiste à oublier toute la structure multiplicative des objets manipulés.

# Bibliographie


[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Kunio Aizawa and Akira Nakamura. Path-controlled graph grammars for multi-resolution image processing and analysis. In Hans Jürgen Schneider and Hartmut Ehrig, editors, *Dagstuhl Seminar on Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1993.

[3] U. Assmann. How to uniformly specify program analysis and transformation with graph rewrite systems. *Lecture Notes in Computer Science*, 1060, 1996.

[4] Michel Bauderon. A uniform approach to graph rewriting : The pullback approach. In Manfred Nagl, editor, *WG*, volume 1017 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 1995.

[5] Michel Bauderon and Hélène Jacquet. Pullback as a generic graph rewriting mechanism. *Applied Categorical Structures*, 9(1), 2001.

[6] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development.* SpringerVerlag, 2004.

[7] Jean Bézivin. From object composition to model transformation with the mda. In *TOOLS (39)*, pages 350–354. IEEE Computer Society, 2001.

[8] Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt. Model transformations in practice workshop. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 120–127. Springer, 2005.

[9] Grady Booch. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[10] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective CAML*. O'Reilly, 2000.

[11] David Chemouil. *Types inductifs, isomorphismes et récriture extensionnelle.* Thèse de doctorat, Université Paul Sabatier, Toulouse, France, septembre 2004.

[12] David Chemouil. Isomorphisms of simple inductive types through extensional rewriting. *Mathematical. Structures in Comp. Sci.*, 15(5) :875–915, 2005.

[13] David Chemouil and Sergei Soloviev. Remarks on isomorphisms of simple inductive types . In *Mathematics, Logic and Computation , Eindhoven, 04/07/03-05/07/03*, pages 1–19, Electronic Notes in Theoretical Computer Science 85, 7, juillet 2003. Elsevier.

[14] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6) :377–387, 1970.

[15] Thierry Coquand and Arnaud Spiwack. Towards constructive homological algebra in type theory. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *Calculemus/MKM*, volume 4573 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2007.

[16] Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, volume 2505 of *Lecture Notes in Computer Science*. Springer, 2002.

[17] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation - part i : Basic concepts and double pushout approach. In Rozenberg [60], pages 163–246.

[18] The Coq development team. The Coq proof assistant reference manual : Version 8.1. Technical report, LogiCal Project, 2006.

[19] Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. Hyperedge replacement, graph grammars. In Rozenberg [60], pages 95–162.

[20] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[21] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer, 1978.

[22] Hartmut Ehrig and Karsten Ehrig. Overview of formal concepts for model transformations based on typed attributed graph transformation. *Electr. Notes Theor. Comput. Sci.*, 152 :3–22, 2006.

[23] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In Maura Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2005.

[24] Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors. *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science*. Springer, 2004.

[25] Hartmut Ehrig, Annegret Habel, Hans-Jörg Kreowski, and Francesco Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Mathematical Structures in Computer Science*, 1(3) :361–404, 1991.

[26] Hartmut Ehrig, Annegret Habel, Julia Padberg, and Ulrike Prange. Adhesive high-level replacement categories and systems. In Ehrig et al. [24], pages 144–160.

[27] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation - part ii : Single pushout approach and comparison with double pushout approach. In Rozenberg [60], pages 247–312.

[28] Hartmut Ehrig, Martin Korff, and Michael Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In Ehrig et al. [29], pages 24–37.

[29] Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Graph-Grammars and Their Application to Computer Science, 4th International Workshop, Bremen, Germany, March 5-9, 1990, Proceedings*, volume 532 of *Lecture Notes in Computer Science*. Springer, 1991.

[30] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1 : Equations und Initial Semantics*, volume 6 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1985.

[31] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars : An algebraic approach. In *FOCS*, pages 167–180. IEEE, 1973.

[32] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In Ehrig et al. [24], pages 161–177.

[33] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation : A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 2005.

[34] Joost Engelfriet and Grzegorz Rozenberg. Graph grammars based on node rewriting : An introduction to nlc graph grammars. In Ehrig et al. [29], pages 12–23.

[35] Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. In Rozenberg [60], pages 1–94.

[36] Martin Fowler and Kendall Scott. *UML distilled : applying the standard object modeling language*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1997.

[37] Claude Godbillon. *Élements de topologie algébrique*. Hermann, 1997.

[38] Object Management Group. Meta object facility (mof) 2.0 query/view/transformation specification. Technical report, OMG, 2008.

[39] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundam. Inform.*, 26(3/4) :287–313, 1996.

[40] Annegret Habel and Detlef Plump. Relabelling in graph transformation. In Corradini et al. [16], pages 135–147.

[41] Reiko Heckel. Introductory tutorial : Foundations and application of graph transformation. Talk, ICGT'06 in Natal, Brazil, 2006.

[42] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In Corradini et al. [16], pages 161–176.

[43] Frank Hermann. A typed attributed graph grammar for syntax-directed editing of uml sequence diagrams. Master Thesis, 2005.

[44] Ivar Jacobson. Object-oriented software engineering. In Boris Magnusson, Bertrand Meyer, Jean-Marc Nerson, and Jean-François Perrot, editors, *TOOLS (13)*, page 539. Prentice Hall, 1994.

[45] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl : a qvt-like transformation language. In Peri L. Tarr and William R. Cook, editors, *OOPSLA Companion*, pages 719–720. ACM, 2006.

[46] Wolfram Kahl. *A relational-algebraic approach to graph structure transformation*. PhD thesis, Universität der Bundeswehr München, 2001.

[47] Harmen Kastenberg. Towards attributed graphs in groove : Work in progress. *Electr. Notes Theor. Comput. Sci.*, 154(2) :47–54, 2006.

[48] Stephen Lack and Pawel Sobocinski. Adhesive categories. In Igor Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2004.

[49] Josep Lladós and Gemma Sánchez. Symbol recognition using graphs. In *ICIP (2)*, pages 49–52, 2003.

[50] Michael Löwe and Hartmut Ehrig. Algebraic approach to graph transformation based on single pushout derivations. In Rolf H. Möhring, editor, *WG*, volume 484 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 1990.

[51] Zhaohui Luo. *Computation and reasoning : a type theory for computer science*. Oxford University Press, Inc., New York, NY, USA, 1994.

[52] Alexander V. Lyaletski, Andrey Paskevich, and Konstantin Verchinine. Sad as a mathematical assistant - how should we go from here to there ? *J. Applied Logic*, 4(4) :560–591, 2006.

[53] Lionel Marie-Magdeleine and Sergei Soloviev. Non-standard reductions in simply-typed, higher order and dependently-typed systems. HOR workshop, affiliated with TLCA'07 conference, Paris, 2007.

[54] Tom Mens. On the use of graph transformations for model refactoring. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *GTTSE*, volume 4143 of *Lecture Notes in Computer Science*, pages 219–257. Springer, 2006.

[55] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[56] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[57] S. Owre, J. M. Rushby, , and N. Shankar. PVS : A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[58] D. Pavlovic and R. Smith. Composition and refinement of behavioral specifications, 2001.

[59] Ulrike Ranger and Erhard Weinell. The graph rewriting language and environment PROGRES. *Applications of Graph Transformations with Industrial Relevance, LNCS*, 5088, 2008.

[60] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1 : Foundations.* World Scientific, 1997.

[61] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual.* Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[62] James E. Rumbaugh, Michael R. Blaha, William J. Premerlani, Frederick Eddy, and William E. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[63] David E. Rydeheard and Rod M. Burstall. *Computational category theory.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988.

[64] Amokrane Saïbi. La théorie des catégories dans coq v6.2. *Coq contribution*, 1998.

[65] Hans Jürgen Schneider. Implementing the Categorical Approach to Graph Transformations With Haskell *in* An Introduction to the Categorical Approach, Draft March 7, 2007.

[66] A. Schürr, A. J. Winter, and A. Zündorf. *The PROGRES approach : language and environment*, pages 487–550. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[67] Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *WG*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1994.

[68] Andy Schürr, Andreas J. Winter, and Albert Zündorf. Graph grammar engineering with progres. In Wilhelm Schäfer and Pere Botella, editors, *ESEC*, volume 989 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 1995.

[69] Carlos Simpson. Formalized proof, computation, and the construction problem in algebraic geometry. *Coq contribution*, 2004.

[70] Karine St-Onge, Philippe Thibault, and Sylvie Hamel Francois Major. Modeling rna tertiary structure motifs by graph-grammars. *Nucleic Acids Research*, 35 :1726–1736, 2007.

[71] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring uml models. In Martin Gogolla and Cris Kobryn, editors, *UML*, volume 2185 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 2001.

[72] Gabriele Taentzer. AGG : A tool environment for algebraic graph transformation. In Manfred Nagl, Andy Schürr, and Manfred Münch, editors, *AGTIVE*, volume 1779 of *Lecture Notes in Computer Science*, pages 481–488. Springer, 1999.

[73] Gabriele Taentzer. Agg : A graph transformation environment for modeling and validation of software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003.

[74] Robert W. Taylor and Randall L. Frank. Codasyl data-base management systems. *ACM Comput. Surv.*, 8(1) :67–103, 1976.

[75] Daniel Varró, András Balogh, and András Pataricza. The viatra2 transformation framework - model transformation by graph transformation. *Eclipse Modeling Symposium*, 2006.

[76] Dániel Varró and András Pataricza. Vpm : A visual, precise and multilevel metamodeling framework for describing mathematical domains and uml (the mathematics of metamodeling is metamodeling mathematics). *Software and System Modeling*, 2(3) :187–210, 2003.

[77] Mandana Vaziri and Daniel Jackson. Some shortcomings of ocl, the object constraint language of uml. In Qizoyan Li, Donald Firesmith, Richard Riehle, and Bertrand Meyer, editors, *TOOLS (34)*, pages 555–562. IEEE Computer Society, 2000.

[78] D. Wolz. Colimit library for graph transformations and algebraic development techniques, 1998.

**TITLE:** A unified categorical approach for attributed graph rewriting

---

**RÉSUMÉ:**

Due to the new requirements of modern software, researchers in software engineering have created more efficient development methods based on the concept of modeling (for example, the MDA) to control every stage of development.
From a theoretical point of view, these methods are based on graphs and graph transformations. The theoretical difficulty lies in adding on these graphs data on which it must be possible to do computations.
Our work has focused on developing a mathematical framework to implement these changes. The theories of categories (through the double pushout) and inductive types (very expressive computation functions) allowed us to provide a unified solution to this problem in which a single operation can transform the structure and compute with the attributes. In addition, the usual properties of rewriting systems are checked.

---

**KEYWORDS:**

Model Driven Architecture, attributed graphs, graph rewriting, categories, type theory, double pushout, pullback.

**AUTEUR :** Maxime REBOUT

**TITRE :** Une approche catégorique unifiée pour la récriture de graphes attribués

**DIRECTEUR DE THESE :** Sergei SOLOVIEV et Louis FÉRAUD

**DATE DE SOUTENANCE :** le 16 juillet 2008
**LIEU DE SOUTENANCE :** IRIT

---

**RÉSUMÉ :**

En génie logiciel, les méthodes modernes de développement (*ex.* le MDA) s'appuient de manière cruciale sur les notions de modélisation et de transformation.

Ces méthodes peuvent s'interpréter à l'aide de la théorie des graphes. La difficulté théorique réside aujourd'hui dans l'ajout sur ces graphes de données supplémentaires sur lesquelles il est nécessaire de pouvoir effectuer des calculs.

Notre travail s'est focalisé sur le développement d'un cadre mathématique sûr afin d'appliquer ces transformations. Les théories des catégories (à travers le double pushout) et des types inductifs (fonctions de calcul très expressives) nous ont permis de donner une solution unifiée à ce problème dans laquelle une seule opération permet de travailler sur la structure et de calculer avec les attributs en définissant des fonctions entre graphes possédant une partie contravariante pour le travail sur les attributs. De plus, les propriétés usuelles des systèmes de récriture sont vérifiées.

---

**MOTS-CLÉS :**

Architecture dirigée par les modèles, graphes attribués, récriture de graphes, catégories, théorie des types, double pushout, pullback.

---

**DISCIPLINE :** Informatique

---

**LABORATOIRE D'ACCEUIL:**

Institut de Recherche en Informatique de Toulouse (IRIT)
Université Paul Sabatier,
118 Route de Narbonne,
F-31062 TOULOUSE CEDEX 9