INSTITUT SUPERIEUR INFORMATIQUE ISI المعهد العالي للإعلامية
UNIVERSITÉ TUNIS ELMANAR
INSTITUT SUPÉRIEUR D'INFORMATIQUE
UNITÉ DE RECHERCHE
MOSIC

Mémoire de Mastère

Présenté en vue de l'obtention
du Diplôme de Mastère en Génie Logiciel

Par
MOHAMED GARA

Exécutabilité de Modèles de Styles Architecturaux

Soutenu le 12 Octobre 2011 devant le jury composé de :

Mme NARJES ROBBANA - Présidente
M. NAOUFEL KRAIEM - Membre
M. ADEL KHALFALLAH - Directeur du Mémoire

Année universitaire : 2011/2012

Remerciements

En préambule à ce mémoire, je souhaite adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et qui ont contribué à l'élaboration de ce travail.

J'exprime ma gratitude à monsieur Adel Khalfallah*, mon directeur de mémoire, pour m'avoir soutenu, conseillé et encadré avec compétence et patience tout le long de ce travail.*

Résumé

Les styles architecturaux et l'ingénierie des modèles sont deux approches qui permettent de maîtriser la complexité d'un système logiciel via la notion d'abstraction. Dans ce mémoire, nous combinons les deux en proposant un langage de modélisation exécutable dédié au style architectural Pipes et Filtres dénommé xPFL (eXecutable Pipes and Filters Language). Nous définissons formellement tous les aspects de ce langage (la syntaxe abstraite, la syntaxe concrète et la sémantique). Pour les connecteurs du style architectural, nous adoptons une approche formelle pour définir leur sémantique en utilisant comme domaine sémantique la transformation de graphe et en utilisant la transformation de modèle pour définir le mapping sémantique. Nous offrons également un outil de support pour ce langage sous forme de plugins Eclipse conforme à la définition formelle de la syntaxe et de la sémantique. Cet outil est principalement composé d'un éditeur graphique, d'un interpréteur et d'un générateur de code final en Java.

Mots Clés : style architectural, pipes et filtres, ingénierie des modèles, méta-modélisation, modèle exécutable, sémantique opérationnelle, transformation de graphe.

Abstract

Architectural styles and Model Driven Engineering are two approaches that control the complexity of a software system with the notion of abstraction. In this paper, we combine the two by providing an executable modeling language dedicated to the Pipes and Filters architectural style called xPFL (eXecutable Pipes and Filters Language). We define formally all aspects of the language (abstract syntax, concrete syntax and semantics). For connectors of the architectural style, we adopt a formal approach to define their semantics using graph transformation as semantic domain and using model transformation to define the semantic mapping. We also offer a support tool for this language as Eclipse plugins according to the formal definition of syntax and semantics. This tool is mainly composed of a graphical editor, an interpreter and a final code generator in Java.

Keywords : architectural style, pipes and filters, model driven engineering, meta-modeling, executable model, operational semantic, graph transformation.

Table des matières

Table des figures

Liste des symboles

ADL Architecture Description Language

BNF Bachus-Naur Form

DMM Dynamic Meta Modeling

DSL Domain Specific Language

EMF Eclipse Modeling Framework

GMF Graphical Modeling Framework

IDM Ingénierie Dirigée par les Modèles

JDT Java Developpement Tools

MDE Model Driven Engineering

MOF Meta Object Facility

QVTo Operational QVT

xPFL eXecutable Pipes and Filters Language

Chapitre 1

Introduction et Problématique

Introduction

Les systèmes informatiques se caractérisent par une évolution croissante de leur complexité et de leur caractère critique. Pour maîtriser cette complexité croissante et améliorer la productivité, les principes d'abstraction, de modularité, de réutilisation et d'automatisation devraient être fortement intégrés dans le processus et les techniques de développement logiciel.

Pour adopter ces principes, une description des systèmes logiciels à un haut niveau d'abstraction a été introduite : c'est l'architecture logicielle. L'architecture décompose le système en termes de modules définissant des sous-systèmes plus simples, appelés composants, interconnectés avec des connecteurs. Dans ce niveau intermédiaire entre la spécification et l'application, a émergé la notion de styles architecturaux. Un style architectural dénote une représentation abstraite regroupant des caractéristiques et des décisions de conception communes pour des instances d'architectures similaires. L'utilisation de ces styles architecturaux favorise la réutilisation aussi bien au niveau des modèles qu'au niveau du code.

Pour représenter l'architecture plusieurs approches ont été proposées. La première approche est la description informelle avec des graphiques. Une deuxième solution semi-formelle consiste à utiliser UML pour décrire et documenter l'architecture logicielle. Par ailleurs, il est possible d'utiliser des formalismes dédiés à la description de l'architecture appelés Langage de Description d'Architecture (en anglais Architecture Description Language ADL). Certains ADL sont semi-formels (basés sur des notations graphiques nor-

malisées) alors que d'autres sont strictement formels (disposant de règles syntaxiques et sémantiques ne souffrant d'aucune ambiguïté) [Aza07]. Ces ADL peuvent être accompagnés par des environnements permettant la vérification et la validation des propriétés des architectures et éventuellement la simulation et la génération de l'application finale.

Dans un autre axe de l'ingénierie des logiciels, à savoir la notion de modèle, plusieurs techniques de l'ingénierie des modèles ont vu le jour, dont principalement la métamodélisation et la transformation de modèle. Cette approche, l'ingénierie des modèles, préconise de passer d'une vision plutôt contemplative à une vision réellement productive des modèles. Ceci a donné naissance au concept de modèle exécutable. Un modèle exécutable est un modèle dont tous les aspects sont complètement et précisément définis. Un modèle est créé en utilisant un langage de modélisation. Pour avoir des modèles exécutables tous les éléments du langage de modélisation (syntaxe et sémantique) devraient donc être formellement définis.

Les instances des styles architecturaux constituent des modèles abstraits du code final des applications. L'objectif de ce travail est de rendre exécutables les instances de modèles du style architectural Pipes et Filtres, en proposant un langage de modélisation exécutable dont la sémantique est formellement définie.

1.1 Problématique

L'ingénierie dirigée par les modèles propose d'utiliser différents langages spécialisés à chaque domaine rencontré dans le processus de développement logiciel. Ce domaine peut être relatif au métier auquel s'adresse la solution logicielle ou bien aux techniques de conception de cette solution. Avec cette tendance, il devient impératif de simplifier la tâche de définition des langages dédiés aux domaines et des outils qui les accompagnent. Nous aurons donc besoin d'avoir recours aux approches génératives dans le développement des éditeurs, des interpréteurs, des compilateurs, des debuggers et tout autre outil concernant les langages de modélisation.

Pour automatiser la création de ces différents outils, nous nous retrouvons devant la nécessité de définir formellement les langages de modélisation. La définition formelle d'un langage revient à définir l'ensemble de ses constituants. Un langage est défini avec sa syntaxe abstraite, sa syntaxe concrète et sa sémantique. Tous ces éléments devraient donc être formellement définis pour avoir une définition précise et complète du langage.

L'ingénierie des modèles propose la technique de méta-modélisation comme méthode standard pour définir formellement la syntaxe abstraite d'un langage graphique. La définition des deux types de syntaxe est, à nos jours, maîtrisée et outillée. Pour la définition de la sémantique des langages de modélisation, il n'existe pas encore d'approche standard et mature. La sémantique est exprimée avec un mapping du domaine syntaxique vers le domaine sémantique. Il est ainsi exigé de proposer une approche bien définie pour préciser le mapping et le domaine sémantique.

La définition formelle et complète d'un langage de modélisation permet de développer des outils compatibles et d'éviter les interprétations subjectives de chaque constructeur d'outil.

Par définition, un style architectural regroupe des concepts communs à un ensemble d'architectures. La sémantique de ces concepts est généralement définie à un haut niveau d'abstraction. Ainsi, La sémantique ne peut pas être précisément définie dans la phase de méta-modélisation. Nous avons donc besoin d'un mécanisme pour le raffinement de la sémantique du modèle lors de son instanciation. Par ailleurs, les styles architecturaux présentent plusieurs variantes. Il est nécessaire de les prendre toutes en considération lors de la définition de la sémantique d'un langage dédié à un style architectural.

1.2 Contribution

Notre principale contribution consiste en la définition formalisée de la sémantique d'un langage de modélisation pour le style architectural « Pipes et Filtres ». La définition de la sémantique revient à définir un mapping sémantique à partir du domaine syntaxique vers un domaine sémantique formellement défini.

L'approche utilisée définit le mapping sémantique avec une transformation de modèle en QVTo. L'application de cette transformation sur le méta-modèle syntaxique renvoie le méta-modèle du domaine sémantique. Le méta-modèle sémantique obtenu contient tous les concepts nécessaires à la définition de la sémantique dynamique : cette sémantique dynamique est exprimée en utilisant le formalisme Transformation de Graphe [EH00]. Avec ce formalisme la sémantique dynamique est exprimée avec un ensemble de règles de transformation qui sont appliquées consécutivement sur un graphe de départ représentant l'état initial. Le résultat est un système d'état/transition dans lequel les états sont des graphes et les transitions sont des règles de transformation définissant le comportement

dynamique.

Nous avons appliqué cette approche sur différentes variantes du style architectural « Pipes et Filtres », dont la syntaxe est définie avec la technique de méta-modélisation. Le résultat est un langage exécutable, dénommé xPFL (eXecutable Pipes and Filters Language), dédié à ce style architectural. Les modèles exprimés dans ce langage sont des modèles exécutables : leurs aspects statique et dynamique sont complètement spécifiés et leur sémantique est formellement exprimée.

De plus, un ensemble d'outils a été proposé pour rendre notre langage opérationnel conformément à sa définition formelle. Cet ensemble est formé de trois constituants principaux : un éditeur graphique, un interpréteur et un générateur du code final en Java.

Dans notre implémentation de ces outils, le comportement des opérations associées à chaque concept défini par le méta-modèle est exprimé en utilisant le langage Java. Le style architectural représente des caractéristiques et des comportements communs à plusieurs instances d'architecture. Ainsi, dans la définition de la sémantique comportementale au niveau du méta-modèle, nous ne pouvons pas introduire les comportements spécifiques à chaque instance. Il existe donc des points de variation sémantique au niveau du méta-modèle qui devraient être précisés au niveau du modèle instance. Dans notre environnement, nous avons résolu ce problème par les choix architecturaux de notre solution.

1.3 Organisation du Mémoire

Le reste du document est organisé comme suit. Le deuxième chapitre introduit la notion d'architecture logicielle et des styles architecturaux, plus particulièrement le style architectural Pipes et Filtres. Le troisième chapitre est une description des éléments définissant un langage de modélisation en se focalisant sur la définition de la sémantique. Le quatrième chapitre est consacré à la description de la définition formelle d'un langage de modélisation exécutable pour le style architectural choisi. Avant de conclure, l'avant dernier chapitre est consacré à la présentation de l'outil développé conformément à la définition formelle du quatrième chapitre.

Chapitre 2

Styles Architecturaux

Nous introduisons dans ce chapitre le domaine pour lequel nous définissons un langage de modélisation spécialisé. Notre contexte général est l'architecture logicielle et plus précisément les styles architecturaux. Nous commençons par introduire ces notions et nous terminons par la présentation du style architectural qui représente le domaine du langage de modélisation à définir.

2.1 Architecture Logicielle

Les solutions logicielles sont devenues de plus en plus complexes en termes d'éléments constituants et en termes de leur organisation et structure. Une description à un niveau architectural a été introduite pour aider à comprendre ces systèmes complexes, simplifier leur maintenance et promouvoir la réutilisation. Ce niveau architectural des systèmes logiciels fournit une description à un haut niveau d'abstraction : seuls les aspects de haut niveau sont identifiés.

L'architecture logicielle est généralement définie comme étant la description de la structure et l'organisation globales du système en termes de composants, du rôle de chaque composant et de l'interaction entre les différents composants. L'architecture logicielle peut être considérée comme un niveau intermédiaire entre les spécifications et l'implémentation. La description de l'architecture satisfait les besoins et sert de base dans la conception. Les

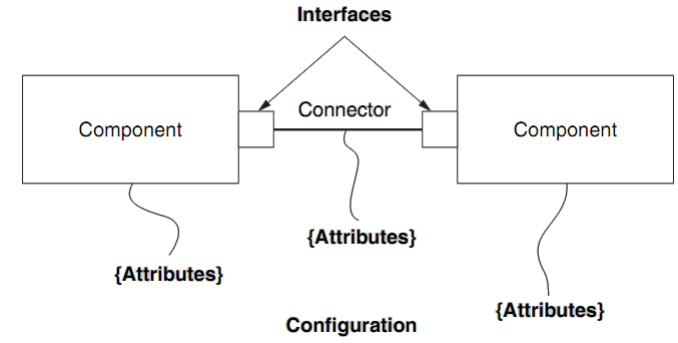


FIGURE 2.1 – Concepts de l'architecture logicielle [Kai05]

concepts utilisés afin de décrire l'architecture sont les suivants :

Composants : Ce sont les briques de base et les éléments actifs dans la description. Ils encapsulent les traitements et les données. Ils accomplissent leurs rôles avec des traitements internes et des communications avec les autres composants. Ces communications sont effectuées en utilisant les ports dédiés à interagir avec l'environnement.

Ports/Interfaces : Ce sont des points d'interactions utilisés par les composants pour communiquer avec le monde extérieur. Un port peut être implémenté avec une variable partagée, une procédure, un évènement ou tout autre mécanisme.

Connecteurs : Ils définissent les interactions entre les composants. Ils connectent les composants entre eux via leurs ports. Le concept Connecteur représente les interactions comme une entité de première classe dans l'architecture.

Attributs/Propriétés : Ils associent aux composants, connecteurs et ports différentes informations utilisées pour analyser, paramétrer ou contraindre la configuration.

Configuration/Système : C'est un ensemble de composants et de connecteurs reliés pour décrire une structure architectural particulière. Elle contient des instances concrètes des différents concepts précédents.

Pour représenter ces différents concepts il est possible d'utiliser des diagrammes et des descriptions informels. Pour avoir une description plus formelle, les Architectural Description Languages (ADL) ont été introduits. Un ADL fournit une notation permettant de composer une configuration en instanciant différents composants et connecteurs.

Dans ce mémoire, nous utilisons la technique de méta-modélisation pour proposer un

DSL[1] pour le style architectural Pipes et Filtres. Nous définissons ainsi un langage graphique permettant de créer des configurations conformes à ce style architectural.

2.2 Style Architectural : Pipes et Filtres

[BMR+96] définit les styles architecturaux comme étant un schéma fondamental de l'organisation structurelle d'un système logiciel. Un style architectural est constitué d'un ensemble prédéfini de sous-systèmes (composants, ports et connecteurs) et de leurs responsabilités. Il est aussi constitué d'un ensemble de règles et de directives pour organiser les relations entre les différents sous-systèmes.

Les styles architecturaux définissent des familles d'architecture qui utilisent les mêmes types de composants, d'interactions, de contraintes structurelles et d'analyses [Kiw04]. Ce sont des templates pour les architectures concrètes.

Dans les outils d'édition, les types de composants, de ports et de connecteurs sont présents dans les palettes. L'utilisateur peut utiliser uniquement les types définis dans la palette pour créer des éléments dans le diagramme. Lors de la connexion des éléments, seules les combinaisons possibles sont tolérées. Ainsi, la palette impose d'utiliser les types définis par le style architectural et le contrôle de la création des connexions impose de satisfaire les contraintes structurelles.

Il existe plusieurs styles architecturaux tels que : l'architecture Client/Serveur, l'architecture en couche, l'architecture Microkernel, l'architecture Blackboard, l'architecture MVC, l'architecture PAC, etc... Le style architectural que nous avons adopté est le style : Pipes et Filtres. C'est l'un des styles de la catégorie flux de données.

Le style Pipes et Filtres décrit une structure utile pour les systèmes qui traitent un flux de données. L'architecture est composée de plusieurs étapes de traitement successives. Chaque étape reçoit des données en entrée et génère des données en sortie. Les étapes de traitement sont encapsulées dans des composants appelés Filtres. Ces derniers consomment et délivrent les données d'une manière incrémentale. Les connecteurs véhiculant les données entre les filtres interconnectés sont appelés Pipes.

Cette architecture offre un haut degré de flexibilité. Elle procure aussi la possibilité de réutilisation des filtres définis. En contrepartie, il y aura une surcharge due au transfert

1. Domain Specific Language (Langage Dédié au Domaine)

des données entre les threads ou les processus et ainsi une perte de performance.

Il existe plusieurs variantes de ce style architectural. Dans notre méta-modèle du style Pipes et Filtres, nous essayerons de représenter le maximum de ces différentes variantes. Les deux invariants de ce style sont les suivants :

- Les filtres ne partagent aucune donnée commune et sont indépendants.
- Les filtres ne connaissent pas l'identité des filtres connectés à l'entrée et les filtres connectés à la sortie.

La structure du style « Pipes et Filtres » définit donc un seul type de composant (Filtre), deux types de ports (Input et Output) et un seul type de connecteur (Pipe).

Filtre :

Les filtres encapsulent les traitements à effectuer sur les flux de données. Une multitude de traitements sont possibles : un filtre peut enrichir, raffiner, filtrer ou transformer les données reçues en entrée. Il peut aussi multiplexer ou démultiplexer les données reçues. En bref, le comportement des filtres concrets est très variable.

Le style architectural ne précise pas le comportement du composant filtre. Ce composant est défini à un niveau d'abstraction assez élevé. Chaque instance concrète du composant filtre aura un fonctionnement interne spécifique.

Deux types de filtre sont définis par ce style. Le premier est un filtre actif. Le filtre actif représente généralement un thread ou un processus. La dynamique des filtres actifs est généralement incluse dans une boucle récupérant les données à partir des entrées des filtres pour en générer des données sur leurs sorties.

Le deuxième type de filtre sont les filtres passifs. L'exécution des traitements réalisés par des filtres passifs dépend des filtres actifs. Seuls les filtres actifs sont auto-déclenchés. Un filtre passif est activé par une demande de récupération d'une donnée déclenchée par un filtre voisin (appel Pull) ou par un envoi de donnée réalisé par un filtre voisin (appel Push).

Un cas particulier des filtres sont les sources de données. Ce type de filtre présente uniquement des sorties de données. Ce type ne reçoit aucune entrée. Un autre type particulier est représenté par les filtres ne possédant aucune sortie. Ils forment la fin de la chaine de traitement du flux de données.

Pipe :

La communication entre les différents filtres est réalisée par le biais des pipes. La communication avec un filtre actif se fait d'une manière asynchrone : la pipe synchronise l'interaction. La communication avec un filtre passif est une communication synchrone. Nous avons donc deux types de pipes définies par ce style architectural : pipe synchrone et pipe asynchrone. Les pipes synchrones sont aussi divisées en deux types : des pipes Push et des pipes Pull.

Les pipes synchrones sont des appels de procédures implémentant le principe de délégation. Le comportement de la pipe asynchrone est défini avec le pattern Pipe, qui est un cas particulier du pattern Producteur-Consommateur [Gra02], incluant uniquement un seul Producteur et un seul Consommateur.

Les pipes peuvent être implémentées avec plusieurs mécanismes : variables partagées, fichiers, messages ou appels de procédure à distance. Les pipes asynchrones dotées de buffer de synchronisation peuvent avoir des tailles de buffers limitées. Les pipes asynchrones peuvent aussi avoir deux comportements différents. Ils peuvent avoir un comportement bloquant ou un comportement non bloquant. Dans le cas d'un comportement bloquant, les threads accédants à la pipe seront bloqués jusqu'à la récupération de la donnée requise (pattern Guarded Suspension [Gra02]). Pour le comportement non bloquant, les threads ne seront pas bloqués (pattern Balking [Gra02]). Un type de données peut être associé aux pipes : les deux bouts de la pipe doivent avoir donc le même type.

Pour le style architectural Pipes et Filtres, les pipes présentent des comportements dynamiques plus concrets que les comportements des filtres. La sémantique dynamique des pipes peut être décrite avec précision pour les différentes variantes de ce connecteur.

Port Input/Output :

Chaque filtre peut avoir un ou plusieurs ports de communication avec le monde externe (les autres filtres de la configuration). Deux types de port sont définis : les ports d'entrée de données (les inputs) et les ports de sortie des données (les outputs). Chaque port constitue un point de communication avec un autre filtre voisin connecté.

Chaque port peut avoir un type. Les pipes connectent les points de sortie de données (outputs) aux points d'entrée de données (inputs). Les pipes ne doivent donc connecter que les ports de même type.

Le comportement des ports est une simple délégation : ils appellent le service corres-

pondant au niveau de la pipe.

Configuration :

Les contraintes structurelles reliées à la configuration, peuvent limiter les connexions possibles entre les différents composants en termes de type et de nombre. Plusieurs contraintes sont possibles. L'utilisation de la méta-modélisation avec les contraintes OCL permet de préciser les différentes contraintes structurelles éventuellement imposées par le style architectural.

L'exécution d'une instance du style architectural Pipes et Filtres commence par lancer tous les filtres actifs qui la composent. Chaque filtre s'occupe de traiter les données qu'il reçoit en entrée et génère un flux de sortie vers les autres filtres auxquels il est connecté.

Pour définir formellement la sémantique des différents éléments définis par ce style architectural, nous serons confrontés au problème de la variation de la sémantique. Certaines de ces variations sémantiques peuvent être traitées en définissant plusieurs variantes du composant ou du connecteur correspondant. Il est aussi envisageable d'offrir la possibilité de paramétrer la définition formelle pour qu'elle représente plusieurs variantes.

En revanche, d'autres types de variation de la sémantique ne peuvent pas être traités avec ces deux solutions. En effet, certains composants sont définis à un haut niveau d'abstraction et les variantes seront infinies. La notion d'exécution et d'abstraction sont dans ce cas en conflit. C'est le cas des différents composants filtres définis par le style architectural. La sémantique comportementale de ces composants ne peut être spécifiée qu'au moment de la définition des instances concrètes de ces derniers.

Dans le reste de ce document, nous définissons formellement la sémantique des éléments structurels définis au niveau d'un méta-modèle correspondant au style architectural Pipes et Filtres. Ce méta-modèle devra refléter les contraintes structurelles imposées par ce style et inclure les différentes variantes des composants et des connecteurs de ce style. Par la suite, nous présentons un ensemble d'outils développés conformément à cette définition formelle.

Chapitre 3

Langage de Modélisation et Sémantique Formelle

Dans ce chapitre, nous commençons par présenter les fondements de l'ingénierie dirigée par les modèles. Par la suite, nous abordons le problème de la définition des langages dédiés aux domaines en mettant l'accent sur le problème de la définition de leur sémantique.

3.1 Fondements de l'IDM

L'Ingénierie Dirigée par les Modèles (IDM) est la discipline qui place les modèles au centre des processus de l'ingénierie logicielle [DLC10]. Selon cette approche, les modèles sont des entités de première classe. Le code final est obtenu suite à une série de transformation de modèle.

En conséquence de cette nouvelle position, les caractères d'informalité et d'imprécision des modèles ne sont plus tolérés. Ils doivent plutôt être suffisamment précis et riches afin de pouvoir être interprétés ou transformés par les différents outils [DLC10].

Dans cette section, nous introduisons les notions fondamentales de l'approche MDE.

3.1.1 Modèle

Un modèle est une description d'un système (ou d'une partie d'un système) à un niveau d'abstraction permettant de masquer les détails inutiles. Dans notre contexte, l'ingénierie des logiciels, le système peut être soit le métier soit le système logiciel.

3.1.2 Langage de Modélisation

Un modèle est exprimé dans un langage. Le type du langage diffère d'un langage à un autre : langage textuel, langage graphique, langage général, langage spécifique etc... Dans l'approche MDE, nous avons besoin de langages bien définis en matière de forme ainsi que de signification pour être automatiquement interprétés. L'ingénierie dirigée par les modèles propose de définir un langage spécialisé pour chaque domaine (technique ou métier).

3.1.3 Méta-modélisation

[Com08] définit la méta-modélisation comme étant l'activité consistant à définir le méta-modèle d'un langage de modélisation. Le rôle de ce méta-modèle est de définir la structure des modèles exprimés dans le langage.

Or, ce méta-modèle est lui-même un modèle. Il devrait donc être défini en utilisant un langage bien défini nommé méta-langage. Ce méta-langage possède une syntaxe abstraite définie avec un méta-métamodèle. Pour éliminer le chainage infini de cette pile, on convient que le méta-métamodèle est auto-défini.

Les différents niveaux de modèles et leurs relations sont les suivants :

Modèle : Il peut être défini comme une abstraction d'un système réel ou logiciel qu'il représente.

Méta-modèle : C'est un modèle de modèles. C'est aussi un langage utilisé pour décrire des modèles, dans la mesure où il englobe un ensemble de concepts nécessaires à la description d'une famille de modèles donnée [DLC10].

Méta-métamodèle : Il permet de décrire le modèle du méta-modèle, autrement dit c'est un langage de description de méta-modèle [DLC10].

Relation Conforme : Elle indique que le modèle instance du niveau supérieur est conforme à la structure décrite par son méta-modèle.

Donc, pour définir la syntaxe abstraite d'un langage, nous devrons créer un méta-modèle en utilisant un méta-langage. Nous disposons, à ce jour pour définir cette syntaxe, de nombreux environnement et langages de méta-modélisation : Eclipse-EMF/Ecore, GME/-MetaGME , AMMA/KM3 ou XMF-Mosaic/Xcore [CRC+06]. L'OMG nous propose aussi son Meta Object Facility (MOF) pour créer des méta-modèles.

Un méta-modèle est, généralement, défini par un langage visuel de méta-modélisation. Bien que l'aspect graphique de ce langage le rende plus intuitif que les langages textuels, il présente un inconvénient majeur. Cette représentation graphique ne permet pas de spécifier (convenablement) certaines propriétés ou contraintes structurelles sur le méta-modèle. Nous avons besoin donc en plus du langage de méta-modélisation d'un langage pour exprimer ces contraintes.

L'OMG propose le langage textuel OCL largement utilisé pour définir les règles de bonne formation des modèles qui ne peuvent pas être exprimées avec le langage de méta-modélisation. Ces règles sont ajoutées au niveau du méta-modèle lui-même pour lui imposer des contraintes additionnelles. Chacune de ces contraintes est rattachée à un contexte bien précis dans le méta-modèle.

Ainsi, la définition des éléments syntaxes des modèles avec la technique de méta-modélisation consiste à définir un méta-modèle enrichis avec des contraintes supplémentaires sous forme d'expressions OCL.

3.1.4 Transformation de Modèle

Une transformation de modèle est une génération automatique d'un modèle cible à partir d'un modèle source. La transformation est définie avec un ensemble de règles qui, réunies ensembles, décrivent comment une instance de modèle du langage source sera transformée en une instance de modèle du langage cible. Si le langage source et cible sont identiques, la transformation est dite endogène sinon elle est exogène. La transformation définie est appliqué par un outil de transformation.

Nous pouvons considérer que les langages de modélisation sont le fondement de l'MDE. Un modèle est défini par le bais d'un langage. Une transformation de modèle reçoit en entrée un modèle et en génère un autre. Ainsi, cette transformation dépend fortement du

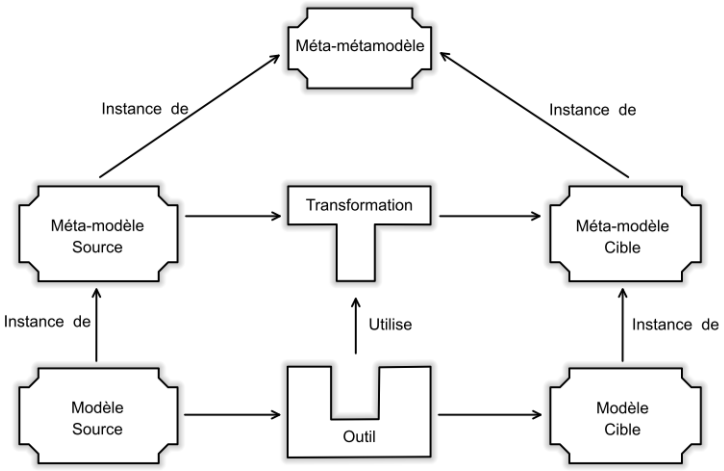


FIGURE 3.1 – Transformation de modèle [KWB03]

langage du modèle source et du modèle généré.

Dans cette section, nous avons introduit les principaux concepts de l'ingénierie des modèles. Dans la section suivante, nous présentons en détail les langages de modélisation spécialisés aux domaines et leurs constituants.

3.2 Langage Spécialisé au Domaine (DSL)

L'approche MDE encourage la définition de plusieurs langages dédiés aux différents domaines afin de créer les différents modèles le long du processus du développement logiciel. Pour donner à ces modèles la dimension opérationnelle voulue et automatiser les transformations de modèles, il est essentiel de décrire de manière précise les langages utilisés pour les représenter [DLC10]. Afin de répondre à ces exigences, la description d'un langage de modélisation bien défini devrait contenir une définition précise des éléments suivants (figure 3.2) :

- une syntaxe abstraite,
- une ou plusieurs syntaxes concrètes,
- et une description de la sémantique.

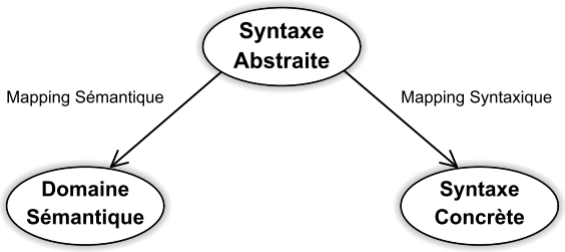


FIGURE 3.2 – Composantes d'un Langage

La spécification de la syntaxe abstraite ne suffit pas à elle seule pour définir un langage utilisable dans le contexte MDE. Bien que non suffisante, la syntaxe abstraite reste l'élément central dans un langage de modélisation bien défini. C'est un élément pivot entre la syntaxe concrète et la sémantique. Les relations entre la syntaxe abstraite et les deux autres sont établies avec deux types de mapping : un mapping syntaxique et un mapping sémantique.

La syntaxe du langage est définie avec les trois constituants : syntaxe abstraite, syntaxe concrète et mapping syntaxique. La sémantique du langage, quant à elle, elle est définie avec le domaine sémantique et le mapping sémantique. La syntaxe d'un DSL fournit les constructions de modélisation qui forment une interface au domaine sémantique. La sémantique d'un DSL donne la signification derrière chaque modèle de domaine bien formé composé par les constructions de modélisation syntaxiques du langage.

3.2.1 Syntaxe Abstraite

La syntaxe abstraite d'un langage de modélisation exprime, de manière structurelle, l'ensemble de ses concepts et leurs relations [CRC+06].

La syntaxe des langages textuels est généralement définie en utilisant une grammaire BNF (Bachus-Naur Form). Dans une approche basée sur les modèles, la syntaxe abstraite d'un langage est décrite avec un méta-modèle. Le méta-modèle pour les DSML est analogue à la grammaire pour les langages textuels. Ce méta-modèle capture les concepts du domaine utilisés pour exprimer les instances de modèle. La technique utilisée pour créer ce méta-modèle est la méta-modélisation.

3.2.2 Syntaxe Concrète

La syntaxe abstraite décrit les concepts du langage mais elle ne décrit pas comment ces derniers sont présentés à l'utilisateur. La syntaxe concrète précise cette représentation. La syntaxe concrète n'est donc qu'une forme équivalente à la syntaxe abstraite plus adaptée à l'exploitation par l'utilisateur [CRC+06]. Cette présentation peut avoir trois types : textuelle, graphique ou les deux conjointement. C'est une interface à laquelle l'utilisateur du langage fait recours pour manipuler les concepts définis par la syntaxe abstraite et ainsi créer des instances de modèle conforme au méta-modèle. A une seule syntaxe abstraite, nous pouvons associer plusieurs syntaxes concrètes.

La définition de la syntaxe concrète est à ce jour bien maîtrisée et outillée [CRC+06]. Il existe en effet de nombreux projets qui s'y consacrent, principalement basés sur EMF (Eclipse Modeling Framework) : GMF (Generic Modeling Framework), TOPCASED, Merlin Generator, GEMS, TIGER, etc [Com08]... Des générateurs d'éditeurs graphique existe, qui, à partir de la syntaxe abstraite et concrète et le mapping syntaxique génèrent un outil d'édition de modèle.

La définition de la syntaxe concrète peut influencer le méta-modèle en lui imposant des contraintes facilitant la génération automatique de l'éditeur graphique correspondant ou en lui ajoutant des éléments additionnels.

Jusqu'ici, nous avons dit que les deux syntaxes sont fortement couplées. Le mapping syntaxique défini les correspondances entre les éléments graphiques et les concepts abstraits. Ce mapping peut être défini formellement ou informellement. La définition formelle est nécessaire pour la génération automatique des éditeurs graphiques.

3.3 Sémantique

Après avoir défini la syntaxe d'un langage il est nécessaire de lui associer une définition de la sémantique. La définition de cette sémantique est cruciale pour que les instances de cette syntaxe soient bien comprises. Sinon il pourrait y avoir des suppositions et des interprétations subjectives qui mènent à une mauvaise utilisation et à un problème de communication pour les utilisateurs et un problème d'interopérabilité pour les outils. Sans une sémantique précise l'utilité d'un langage est fortement réduite.

Définir la sémantique d'un langage revient à définir le domaine sémantique et le map-

ping entre la syntaxe abstraite et le domaine sémantique [Com08]. La sémantique est définie à partir de la syntaxe abstraite. Le domaine sémantique doit capturer les concepts exprimés par le langage. La signification des éléments syntaxiques est exprimée en les reliant aux éléments du domaine sémantique. Dans notre contexte, contrairement aux syntaxes qui sont formalisées et outillées, la sémantique des langages de modélisation est, à ce jour, rarement définie et fait actuellement l'objet d'intenses travaux de recherche [Com08]. La sémantique est un sujet ouvert : il n'y a pas encore une approche standardisée ou sur laquelle il y a un consensus. Par exemple UML se concentre sur la définition de la syntaxe abstraite et concrète et sa sémantique reste définie informellement en utilisant le langage naturel. Des travaux existent afin de définir sa sémantique formellement, mais il reste encore un langage semi-formel.

3.3.1 Sémantique Statique et Dynamique

[HR04] attire notre attention sur la confusion courante entre la sémantique et le comportement. Il déclare que la structure et le comportement sont deux vues importantes dans un système de modélisation. Les deux sont représentés par les concepts syntaxiques et les deux ont besoin d'une sémantique. Nous pouvons, ainsi distinguer entre deux types de sémantique. La sémantique statique exprime la sémantique structurelle d'un terme du langage : la définition de cette sémantique répond à la question « Qu'est-ce que c'est ? ». La sémantique dynamique est concernée par le comportement exprimé par un terme du langage. Elle répond à la question « Qu'est-ce qu'il fait ?) ». Ces deux types de sémantique sont définis avec un mapping du domaine syntaxique vers le domaine sémantique. La définition de la deuxième est généralement la plus difficile [Hau05].

Un autre courant utilise le terme sémantique statique pour désigner les contraintes de bonne formation vérifiables au moment de la compilation et le terme sémantique dynamique pour désigner le comportement dynamique lors de l'exécution. [Lan09, Chapitre 2] considère l'utilisation du terme sémantique statique pour nommer les contraintes syntaxiques supplémentaires définies en OCL ou autre comme une mauvaise utilisation.

3.3.2 Sémantique Formelle

La description du domaine sémantique peut avoir plusieurs degrés de formalité, partant du langage naturel jusqu'à une définition mathématique rigoureuse [HR04]. La méthode informelle utilisant le langage naturel et les exemples pour décrire la sémantique d'un langage, bien qu'elle soit plus facile, peut s'avérer imprécise et ambigue. Dans ce cas, la sémantique est une interprétation subjective de la description informelle, ce qui peut mener à des problèmes de communication : un seul modèle est interprété de différentes manières selon la compréhension de chaque acteur. Un autre problème peut émerger. Les outils basés sur la sémantique d'un modèle tels que les générateurs de code aurons un problème d'interopérabilité : chacun présente une sémantique relative à l'interprétation de son constructeur. Idéalement la sémantique doit être formellement définie pour éliminer les problèmes engendrés par les descriptions informelles. Une sémantique est dite formelle quand elle est précise et non ambigue. Elle est définie dans un formalisme mathématique.

Dans l'approche MDE, la possibilité de transformation, d'analyse et d'exécution des modèles sont exigés. Pour atteindre cet objectif, il est impératif d'avoir une sémantique précise et non ambigue de ces derniers. La définition formelle de cette sémantique est fondamentale pour que les modèles constituent un moyen non ambigu de communication et une base précise pour l'automatisation des tâches dans le développement logiciel [Hau05].

Nous présentons dans la suite l'état de l'art de la définition de la sémantique des méta-modèles. Mais avant de passer à ce point, nous décrivons les différents types de sémantique.

3.3.3 Types de Sémantique

Les types de sémantique déjà exploités dans le domaine des langages de programmation, sont réutilisés et adaptés aux langages de modélisation. Une sémantique comportementale peut être axiomatique, dénotationnelle ou opérationnelle. Dans [Com08], ces trois types de sémantiques sont illustrés à travers des expérimentations pour définir la sémantique d'exécution d'un langage simple de modélisation de processus et la sémantique d'une fonction décrémenter dans un modèle. Nous décrivons ces trois approches en partant de la plus abstraite à la plus concrète :

3.3.3.1 Sémantique Axiomatique

Une sémantique axiomatique propose une vision déclarative en décrivant l'évolution des caractéristiques d'un élément lors du déroulement d'un programme [Com08]. Avec cette approche, la sémantique d'un langage est définie avec un ensemble d'axiomes (appelés encore affirmations) sur les concepts structurels du domaine syntaxique du langage concerné. Ces axiomes sont dénommés triplet de Hoare dont la forme est la suivante [Wol09] :

$$\{P\}S\{Q\}$$

Chaque axiome est formé de trois éléments. P est une précondition, S est une structure syntaxique du langage et Q est une postcondition. L'ensemble de ces triplés définit le comportement des concepts structurels du langage.

Un exemple pratique de ce type de sémantique est l'utilisation d'OCL pour définir la sémantique comportementale des opérations d'un modèle. P sera donc la précondition, Q la postcondition et S l'opération. La définition de la précondition et de la postcondition pour une opération d'un modèle constitue une spécification axiomatique de la sémantique de cette opération.

3.3.3.2 Sémantique Dénotationnelle

La sémantique dénotationnelle s'appuie sur un formalisme rigoureusement (mathématiquement) défini pour exprimer la sémantique d'un langage donné. Une traduction est réalisée des concepts du langage d'origine vers ce formalisme. C'est cette traduction qui donne la sémantique du langage d'origine [CRC+06]. Ainsi, les outils matures de simulation, de vérification et d'exécution du domaine cible peuvent être exploités.

Dans sa forme mathématique pure, l'approche dénotationnelle reste très théorique est non compris par la majorité. Dans la cadre de l'approche MDE, nous pouvons assister à un défaut de rigueur mathématique dans la définition de ce type de sémantique (au niveau du mapping sémantique), elle est ainsi dénommée sémantique par traduction (ou par compilation) [Hau05]. La sémantique des expressions du domaine syntaxique du langage est exprimée par une traduction de celles-ci vers des expressions du domaine sémantique formellement défini. Ce type de sémantique décrit l'exécutabilité d'un modèle dans un autre formalisme (compilation). Le domaine syntaxique est donc différent du domaine sémantique.

3.3.3.3 Sémantique Opérationnelle

Une sémantique opérationnelle donne une vision impérative en décrivant un programme par un ensemble de transitions (ou transformations) entre les états du contexte d'exécution [Com08]. La sémantique opérationnelle permet de décrire le comportement dynamique des constructions d'un langage. Dans le cadre de l'MDE, elle vise à exprimer la sémantique comportementale d'un méta-modèle afin de permettre l'exécution des modèles qui lui sont conformes [CRC+06].

Contrairement à l'approche par traduction qui décrit les éléments du domaine syntaxique à travers les éléments du domaine sémantique, l'approche opérationnelle décrit l'effet engendré par les éléments du domaine syntaxique sur l'état actuel. Elle est définie en termes de règles de transformation. Chaque règle est formée par une précondition à satisfaire pour être appliquée et l'effet qui transforme l'état courant. L'approche opérationnelle est donc plus intuitive que l'approche dénotationnelle dans l'expression de l'aspect comportemental [Hau05].

Ce type de sémantique décrit l'exécutabilité d'un modèle dans le même formalisme (interprétation). Le domaine syntaxique est donc identique au domaine sémantique.

3.4 Définition de la Sémantique des DSL

Dans l'approche MDE, la structure syntaxique des modèles est définie avec la technique de méta-modélisation. En revanche, la définition de la sémantique d'un modèle reste un sujet crucial et ouvert pour cette approche [GRS09a]. Plusieurs travaux ont été proposés dans ce sens, intégrant la technique de méta-modélisation avec différents techniques formelles afin d'associer aux modèles une sémantique précise et rigoureuse.

[Hau05] présente une taxonomie des techniques de définition de la sémantique des modèles. Il divise ces travaux en deux catégories. Les techniques du premier groupe visent à atteindre des objectifs particuliers (vérification, analyse ou génération de code). Le deuxième groupe contient les approches générales de définition de la sémantique. Pour le premier groupe, le but spécifique guide le processus de formalisation et le domaine sémantique est une technique formelle outillée. Cette catégorie se concentre sur son objectif spécifique et ne fournit pas de solution universelle pour la définition de la sémantique. [Hau05] présente une comparaison de ces approches. Nous ne nous intéressons qu'aux approches univer-

selles de description de la sémantique opérationnelle d'un langage défini avec la technique de méta-modélisation.

En utilisant la méta-modélisation, la syntaxe abstraite est définie avec un méta-modèle. Ce méta-modèle est défini avec les concepts objets (Classe, Attribut, Association, Opération) imposés par le méta-métamodèle. Le méta-métamodèle associe à chaque élément défini avec le concept classe un ensemble d'opérations encapsulant le comportement dynamique relatif à cet élément. Le méta-métamodèle fournit uniquement les concepts nécessaires à la définition de la structure du méta-modèle : aucun aspect comportemental ne peut être associé aux opérations.

La définition de la sémantique d'exécution de l'ensemble des opérations associées aux instances de classe permet de définir la sémantique d'exécution des modèles. Le corps des opérations manipule le graphe d'objets (création d'objets, destruction d'objets, test de conditions, changement de valeurs, etc ...).

[TC08] introduit cinq niveau pour évaluer la qualité et la complétude de la méta-modélisation.

Niveau 1 : Dans ce premier niveau, une simple syntaxe abstraite est définie. La sémantique du langage est informelle et incomplète.

Niveau 2 : Dans ce niveau, la syntaxe abstraite est relativement complète. Un ensemble de règles de bonne formation est défini. La sémantique reste informellement définie.

Niveau 3 : Dans ce niveau, la syntaxe abstraite est complètement définie. Une syntaxe concrète est associée au langage, mais elle est partiellement formalisée.

Niveau 4 : Dans ce niveau, la syntaxe concrète est formalisée. L'utilisateur peut créer des modèles visuels ou textuels et vérifier leur conformité à la syntaxe abstraite.

Niveau 5 : Dans ce dernier niveau, tous les aspects du langage sont formellement définis, y compris la sémantique comportementale. Les instances de modèle dans ce niveau sont des modèles exécutables. Dans un modèle exécutable les aspects statique et dynamique sont complètement spécifiés et la sémantique est formelle exprimée, permettant, ainsi, aux utilisateurs du langage de faire des opérations sémantiquement riches sur les modèles comme la simulation, l'évaluation et l'exécution. Le langage ne dépend d'aucune technologie extérieure : il est indépendant de toute plateforme et peut être utilisé pour générer et développer différents outils de support.

Dans ce mémoire, nous proposons un langage de modélisation du cinquième niveau dont

tous les aspects sont formellement définis, offrant par conséquent, la possibilité d'exécuter les modèles. Il nous faut donc une approche formelle pour définir la sémantique du langage. Avant de présenter l'approche utilisée pour définir la sémantique opérationnelle de notre langage, nous présentons quelques approches proposées pour formaliser la sémantique des modèles.

3.4.1 Transformation de Graphe

La Transformation de Graphe est une approche théoriquement bien fondée, tout en étant simple et intuitive : étant donné un graphe (une structure de nœuds et de connecteurs), une transformation de graphe produit un autre graphe. La modification effectuée par la transformation est capturée dans une règle de transformation [Hau05]. Un graphe peut avoir un type décrit avec un graphe de type imposant des contraintes structurelles au graphe typé. Avec cette approche, le domaine sémantique sera un système de transition d'états dont les états sont des graphes et les transitions sont des règles de transformation [Wol09].

[Com08] décrit une expérimentation de cette approche pour la définition de la sémantique d'exécution d'un modèle en utilisant un outil formel nommé AGG [AGG] (figure 3.3).

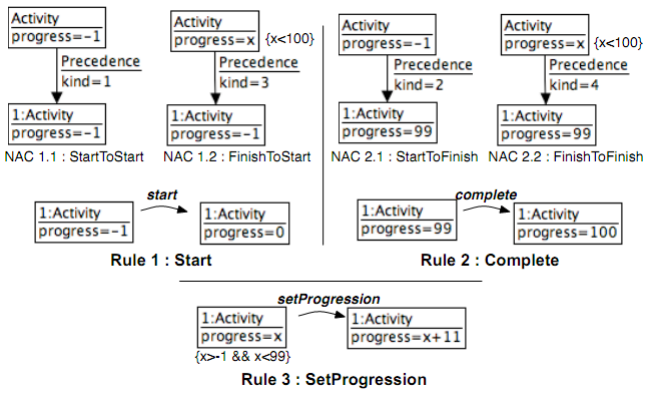


FIGURE 3.3 – Exemple de transformation de graphe [Com08]

La transformation de graphes est un formalisme visuel permettant de manipuler intuitivement des éléments structurés sous forme de graphes. Les caractères visuel, intuitif

et rigueur mathématique de ce domaine sémantique en font un moyen intéressant pour la spécification de la sémantique dynamique d'un modèle.

Les deux piliers de la technique sont les graphes et les règles de transformation. A partir d'un graphe en entrée l'application d'une règle de transformation modifie l'état actuel en produisant un autre graphe en sortie.

3.4.1.1 Graphe

Dans sa version la plus basique, un graphe est formé par un ensemble de nœuds connectés avec des liaisons. Il existe plusieurs variantes structurelles des graphes : par exemple les liaisons peuvent être unidirectionnelles ou bidirectionnelles, deux nœuds peuvent être connectés avec plusieurs liaisons ou une seule liaison est tolérée, etc ...

Il existe aussi d'autres variantes plus évoluées que la version basique :

Graphe étiqueté : Des étiquettes textuelles sont attachées aux nœuds, aux liaisons ou les deux en même temps. Elles sont utilisées pour attribuer des noms aux nœuds et aux liaisons et éventuellement aux bouts des liaisons.

Graphe avec attributs : Les nœuds peuvent avoir des attributs. Ces attributs sont définis avec des noms et auxquels des valeurs sont affectées. Les valeurs possibles sont définies avec le type de l'attribut (entier, réel, chaine de caractères, etc...).

Graphe typé : Des contraintes structurelles supplémentaires sont imposées sur un graphe typé en plus des contraintes générales. Ces contraintes sont exprimées par le biais d'un graphe de type. Par analogie à la approche orientée objet, le graphe de type correspond aux classes et le graphe typé correspond aux objets instances de ces classes.

Graphe avec héritage : La notion d'héritage est une notion principale dans l'approche orientée objet. Dans un graphe avec héritage, cette relation est représentée avec un arc de liaison particulier.

Dans notre contexte, nous avons besoin d'un graphe étiqueté, typé et avec attributs et héritage pour reproduire tous les concepts définis dans un méta-modèle défini avec les concepts objet.

3.4.1.2 Règle de Transformation

Contrairement au graphe, qui représente l'aspect structurel, une règle de transformation représente l'aspect dynamique dans le formalisme. L'application des règles de transformation sur un graphe de départ permettent de le faire évoluer.

Une règle est composée de deux parties. La première partie est un sous-graphe à rechercher dans le graphe de départ, représentant ainsi la partie du graphe à manipuler par la règle. La deuxième partie est un autre graphe qui remplacera le sous graphe retrouvé dans le graphe initial.

Différentes approches sont disponibles pour contrôler l'application des règles de transformation (exemple : affecter des priorités aux règles, utiliser un programme pour contrôler l'application des règles).

La figure 3.4 représente une illustration de l'application d'une règle sur un graphe cible.

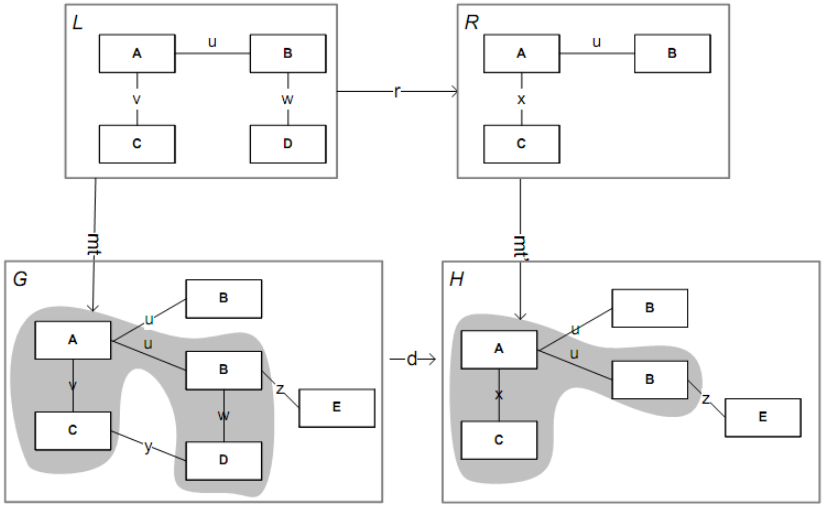


FIGURE 3.4 – Exemple d'application d'une règle sur un graphe [Hau05]

3.4.1.3 Avantages

Les avantages de la technique de transformation de graphe pour la définition formelle de la sémantique dynamique sont [Hau05] :

Compréhensibilité : avec son aspect visuel le formalisme est plus intuitif que d'autres formalismes non visuels et purement théorique. Par ailleurs, nous gardons le

même niveau d'abstraction et les mêmes concepts que le domaine syntaxique.

Adéquation : En plus de permettre d'exprimer certains concepts, le formalisme nous permet d'exprimer convenablement des situations fréquemment rencontrées. En effet, dans notre manipulation des instances de méta-modèles, nous avons un support adéquat pour exprimer et manipuler les concepts orientés objet.

Analysabilité : Nous avons la possibilité d'appliquer certaines analyses à la spécification de la sémantique. Nous pouvons utiliser la technique de model checking et l'exploration des différents états du système et ainsi vérifier certaines propriétés.

Outils : Différents outils sont disponibles pour créer, manipuler et analyser des graphes.

3.4.2 Transformation Endogène de Modèle

Une approche par transformation de modèle endogène permet de définir de manière déclarative le comportement sous la forme d'un système de transition de modèles, en fonction des états possibles du modèle [Com08]. Le domaine sémantique est donc un système de transition d'états. Ses états sont des modèles instances du méta-modèle. Ses transitions sont définies avec un langage de transformation de modèle (ATL, QVT ...).

[Com08] expérimente cette technique pour définir la sémantique opérationnelle d'un méta-modèle en utilisant le langage de transformation de modèle ATL.

L'avantage de l'utilisation de la transformation de graphe par rapport à l'utilisation de la transformation de modèle pour définir la sémantique dynamique est la possibilité d'exploiter des méthodes formelles tel que le model checking [Wol09]. La transformation de graphe l'emporte grâce à son fondement théorique.

3.4.3 Méta-modélisation Dénotationnelle

Dans l'approche de définition de la sémantique par méta-modélisation en plus de la syntaxe abstraite, le domaine sémantique est défini en tant que modèle [Lan09]. Le méta-modèle syntaxique défini l'ensemble des expressions du langage. L'approche propose de formuler le domaine sémantique dans la même structure, en utilisant un méta-modèle sémantique afin de définir l'ensemble des concepts sémantique nécessaire pour exprimer le sens du modèle [Hau05]. Le mapping sémantique est défini au niveau méta-modèle. Ce mapping est projeté au niveau instance.

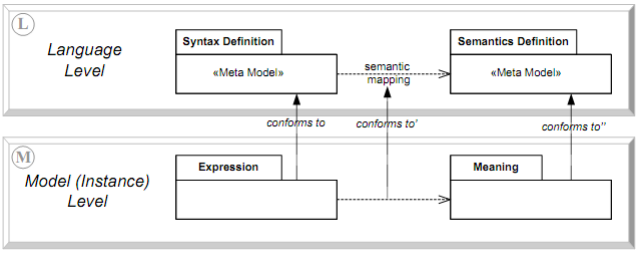


FIGURE 3.5 – Architecture de la méta-modélisation dénotationnelle [Hau05]

Dans cette approche, le domaine sémantique peut être construit librement. Les concepts adéquats à la définition de la sémantique sont définis sans être obligé à utiliser des concepts prédéfinis dans le domaine sémantique. En général, certains concepts du domaine sémantique correspondront à des concepts du domaine syntaxique et d'autres seront ajoutés pour décrire des éléments nécessaires à la description de la sémantique de l'exécution. Cette approche est adéquate à la définition de la sémantique statique.

La méta-modélisation dénotationnelle est utilisée dans la définition de la sémantique d'OCL [Lan09, Chapitre 7] et dans l'approche DMM [Hau05] décrite par la suite. D'après [Hau05], c'est le style de définition de sémantique le plus influant pour UML.

3.4.4 Méta-modélisation Dynamique

[Hau05] déclare que le point faible majeur de l'approche précédente est la difficulté d'exprimer l'aspect comportemental convenablement. Pour résoudre ce problème, il propose une approche de description de la sémantique d'un langage de modélisation visuel, qui combine la méta-modélisation dénotationnelle pour exprimer la sémantique statique avec des règles opérationnelles définissant l'aspect dynamique des éléments (figure 3.6). Pour définir la sémantique d'une opération dans un méta-modèle, il lui associe une règle opérationnelle. Il propose aussi la notion d'invocation de règle utilisé pour le contrôle de l'application des règles opérationnelles.

3.4.5 Approche Proposée

Pour définir formellement la sémantique d'un langage de modélisation graphique, il n'existe pas encore une approche standard. Peu importe l'approche à utiliser pour définir

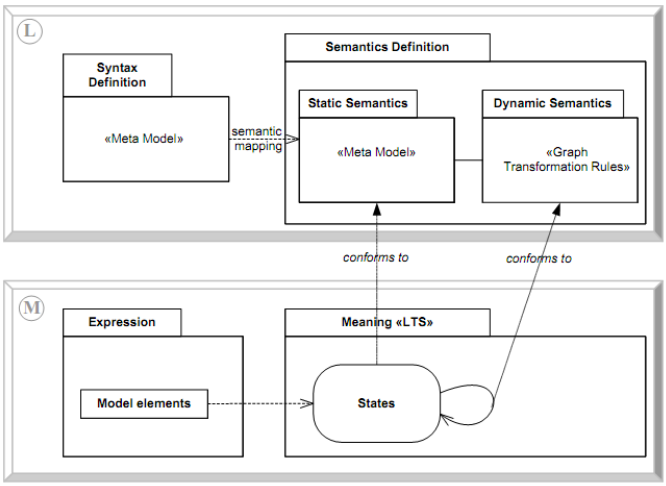


FIGURE 3.6 – Approche de méta-modélisation dynamique [Hau05]

la sémantique générale de notre langage, elle devra satisfaire les critères : formalité et compréhensibilité. Par le critère compréhensibilité, nous éliminons les approches purement mathématiques compréhensibles uniquement par des experts du domaine sémantique. Nous voudrions adopter une approche intuitive.

L'approche opérationnelle permet de spécifier la sémantique comportementale tout en restant dans le même formalisme en décrivant l'évolution du modèle d'un état à un autre. Elle est plus adaptée dans le cas où l'objectif est l'exécution du modèle. Il serait donc plus judicieux d'utiliser l'approche opérationnelle pour définir la sémantique dynamique des modèles exprimés dans notre langage.

Puisque le modèle représente une abstraction du code final, il est fortement apprécié de rester dans le même formalisme et le même niveau d'abstraction. La définition de la sémantique dynamique à ce niveau est plus simple et intuitive. Le modèle est considéré comme étant un ensemble de données sur lesquelles nous appliquons des règles de transformation. Le domaine sémantique est ainsi un système de transition d'états dans lequel les états sont des instances du méta-modèle et les transitions sont les règles de transformation.

Avec cette approche, nous pouvons superviser le comportement au niveau des états intermédiaires à chaque application d'une règle de transformation. L'approche opérationnelle est plus intuitive et plus compréhensible pour exprimer une sémantique dynamique que les approches par traduction vers un autre domaine formel dont les concepts sont totalement différents.

La structure des modèles graphiques est définie formellement en tant que graphe. Donc pour définir nos règles de transformation nous pouvons soit utiliser les langages de transformation de modèle directement soit utiliser les outils formels de transformation de graphe. Nous avons choisi d'utiliser la deuxième technique afin d'exploiter la possibilité de suivre étape par étape l'évolution de l'état courant suite à l'application des règles de transformation offerte par les outils de ce domaine. Ainsi, notre domaine sémantique sera un système état/transition dans lequel les états sont des modèles représentés avec des graphes et les règles de transformation sont définies dans le langage de l'outil utilisé qui sera présenté par la suite.

La technique de transformation de graphe est un domaine formel qui répond à notre besoin : c'est un formalisme visuel qui permet de reprendre le domaine syntaxique tel qu'il est. Par ailleurs, c'est un domaine rigoureusement défini permettant de manipuler intuitivement les structures objets (rappelons que notre domaine syntaxique est une instance d'un méta-modèle défini avec les concepts de l'approche orientée objet). A partir d'un état initial une opération relative à un concept donné du domaine syntaxique peut être décrite par une application d'une série de règles définissant ainsi le comportement désiré.

Jusqu'ici, nous avons détaillé l'approche opérationnelle utilisée pour définir le comportement dynamique. Mais pour définir ce comportement des concepts relatifs à l'exécution sont nécessaires pour détailler la sémantique dynamique. Ces concepts doivent être intégrés avec les concepts du domaine syntaxique. Pour expliquer ce point prenons l'exemple d'un programme Java. Tout programme Java est décrit avec un ensemble de classes. Ces classes sont des instances du domaine syntaxique du langage. Nous ne nous pouvons pas comprendre la sémantique d'exécution de ce programme avec uniquement ses classes. D'autres notions relatives à l'environnement d'exécution (interpréteur) doivent être prises en considération (exemple : le tas, la pile d'exécution, le ramasse miettes ...) pour comprendre précisément le comportement du programme.

Pour ajouter ces concepts, nous utilisons le principe de l'approche de méta-modélisation dénotationnelle. Au méta-modèle syntaxique nous associons un autre méta-modèle dans lequel nous reprenons les concepts définis dans le domaine syntaxique et nous leur ajoutons les concepts désirés. Nous obtenons ainsi un méta-modèle définissant les concepts du domaine sémantique. Ce nouveau méta-modèle est obtenu avec une transformation Modèle vers Modèle définie en QVTo.

Le comportement de chaque opération dans le méta-modèle est défini par un ensemble

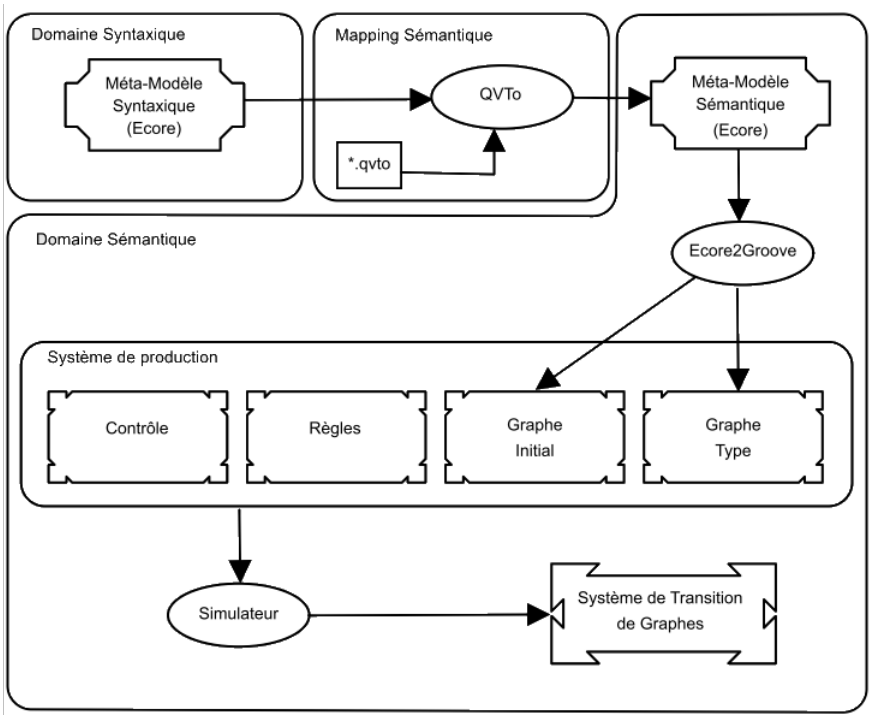


FIGURE 3.7 – Approche proposée

de règles de transformation. A chaque méthode nous associons au moins deux règles de transformation : une pour l'appel de la méthode et une autre pour le retour de la méthode. Il est possible de lui associer d'autres règles correspondant au corps de la méthode.

L'élément de base dans la description de l'exécution est l'appel de méthode. Pour représenter la notion d'appel de méthode dans le domaine sémantique, nous ajoutons le concept « MethodFrame » possédant le nom de la méthode, le pointeur « this » vers l'objet sur lequel la méthode est invoquée et l'ensemble des paramètres de la méthode. Au besoin, nous ajoutons d'autres concepts pour décrire le comportement de ces méthodes.

La figure 3.7 résume cette approche. Sur le méta-modèle définissant la syntaxe abstraite, nous appliquons une transformation de modèle définie en QVTo pour générer le méta-modèle sémantique contenant des concepts supplémentaires nécessaires à la définition de la sémantique dynamique. Nous avons défini deux transformations. Une première transformation à appliquer pour les exécutions mono-thread et une deuxième transformation à appliquer pour les exécutions multi-threads.

Nous avons choisi d'utiliser l'outil formel de transformation de graphe Groove. Avec l'utilitaire Ecore2Groove, nous transformons automatiquement le méta-modèle sémantique

généré en un graphe de type et un graphe initial inclus dans une grammaire de transformation de graphe. Pour compléter le système de transformation de graphe, nous devons créer les règles de transformation associées aux opérations du méta-modèle et éventuellement un script contrôlant l'exécution de ces règles.

Après avoir défini le système de production dans l'outil Groove utilisé, nous utilisons le simulateur de cet outil pour générer automatiquement ou manuellement le système de transition d'états.

Dans le chapitre suivant, nous utilisons cette approche pour définir la sémantique opérationnelle des modèles.

Chapitre 4

Définition Formelle du Langage xPFL

Dans ce chapitre, nous proposons un langage de modélisation exécutable et orienté modèle pour style architectural Pipes et Filtres dénommé : xPFL (eXecutable Pipes and Filters Language). Dans notre définition de ce langage, nous utilisons la technique de méta-modélisation. Notre définition sera conforme au cinquième niveau de méta-modélisation détaillé dans le chapitre précédent. Etant donné que nous avons défini tous les aspects du langage, les modèles instances du méta-modèle de ce langage seront donc des modèles exécutables. Nous présentons dans ce chapitre une description formelle du langage xPFL.

Les différents éléments utilisés pour définir le langage de modélisation sont schématisés sur la figure 4.1. Dans notre définition de la syntaxe abstraite, de la syntaxe concrète et de la sémantique du langage, nous avons utilisé les outils et les frameworks des projets suivants :

- EMF (Eclipse Modeling Framework [DS08])
- GMF (Graphical Modeling Framework [Gro09])
- QVTo (Operational QVT [Gro09])

Tous les frameworks énumérés sont englobés au sein du projet « Eclipse Modeling Project » [Gro09]. Ce projet se concentre sur l'évolution et la promotion des technologies de développement orientées modèle dans la plateforme Eclipse en fournissant un ensemble

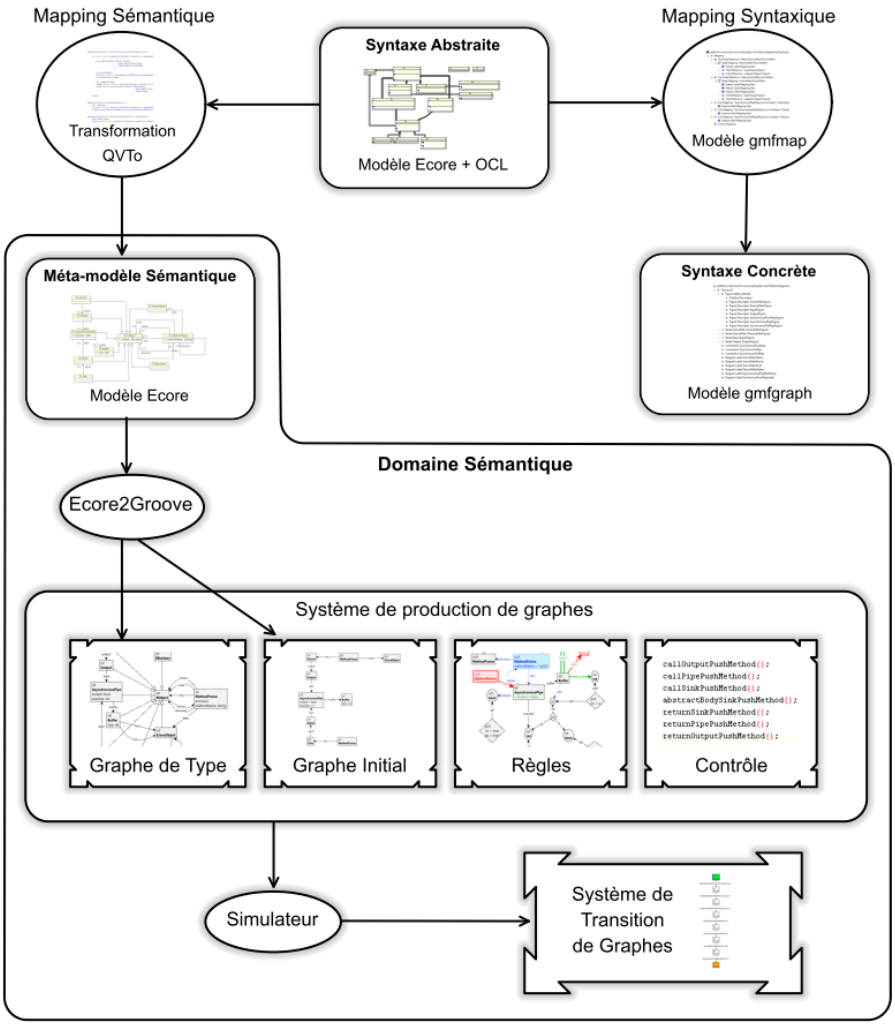


FIGURE 4.1 – Définition de xPFL

unifié de frameworks, d'outils et d'implémentation de standards.

4.1 Syntaxe Abstraite

Dans la définition d'un langage avec la technique de méta-modélisation, la syntaxe abstraite, définie avec un méta-modèle, occupe la place centrale dans sa description et sert de base dans la définition des autres éléments. Nous commençons par définir cette syntaxe avec un modèle Ecore représenté dans la figure 4.2.

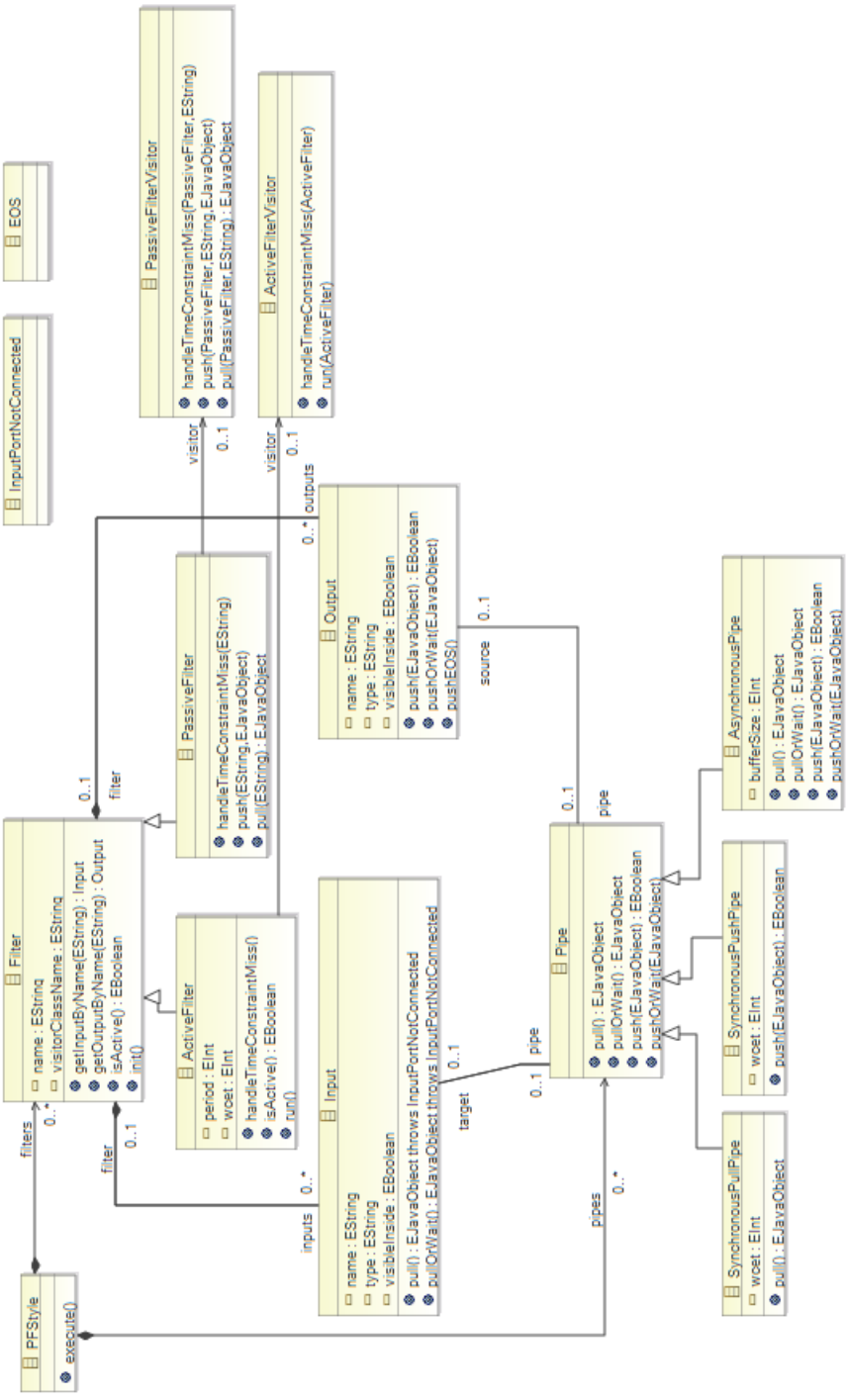


FIGURE 4.2 – Méta-modèle

4.1.1 Méta-modèle

Le cœur du méta-modèle est composé par les concepts suivants :

- Les composants (filtres) : Les filtres encapsulent les traitements à réaliser sur les données reçues en entrée pour générer le flux de sortie. Nous avons deux types de filtres :
  - Filtre Actif : Le traitement réalisé par les filtres actifs est lancé automatiquement au début de l'exécution. La méthode « run » d'un filtre actif définie son comportement, qui consiste à lire les données entrantes et les traiter pour enfin générer les données de sortie. La tâche du filtre peut être exécutée d'une manière périodique en affectant une valeur non nulle à l'attribut « period ».
  - Filtre Passif : Les filtres passifs contiennent des traitements qui sont invoqués par d'autres filtres. Ils peuvent contenir une méthode « push » (entrée de données) ou/et une méthode « pull » (sortie de données). L'exécution de ces méthodes dépend donc des filtres reliés.
- Les ports de communication : chaque port possède obligatoirement un nom avec lequel il est repéré au sein de son filtre parent. Le type du port définit le type des données véhiculées. Les ports sont de deux types :
  - Input : c'est un point d'entrée des données au filtre parent à partir d'un autre filtre source.
  - Output : c'est un point de sortie des données du filtre parent vers un autre filtre cible.
- Les connecteurs (pipes) : au lieu de séparer le comportement bloquant et non bloquant de la communication via la pipe dans deux types de pipes : pipe bloquant et pipe non bloquant, nous avons défini dans l'interface « Pipe » ces deux possibilités conjointement. Dans ce cas, la classe héritante définira le comportement désiré ou probablement les deux en même temps.

  Nous avons deux catégories de pipes :
  - Synchrone : Cette catégorie contient deux types de pipes : « SynchronousPullPipe » et « SynchronousPushPipe ». Chacune d'entre elles permet d'invoquer sa méthode respective.
  - Asynchrone : Cette catégorie contient un seul type de pipe qui est « AsynchronousPipe ». Avec ce type de pipe les deux filtres connectés communiquent via un tampon dont la taille est limitée par la valeur de l'attribut « bufferSize ».

Nous ne nous attardons pas à décrire les connecteurs dans ce paragraphe, puisque la sémantique de ces derniers sera définie formellement par la suite.

4.1.2 Contraintes Structurelles

Les ports supportent les deux types de communications : synchrone et asynchrone. Ainsi, selon le type du connecteur, d'autres contraintes syntaxiques sur les connexions possibles entre les ports devraient être déterminées. Des règles de bonne formation devraient donc être ajoutées au niveau du méta-modèle.

- Si un des deux ports connectés possède un type, le type du port de l'autre bout doit être identique au premier.

```
context Pipe
        inv invarant_CheckType :
                source.type = target.type
```

- Toutes les combinaisons possibles des connexions sont autorisées sauf les deux cas suivants :
  - Une pipe « SynchronousPullPipe » avec un filtre source actif.

```
context SynchronousPullPipe
        inv invariant_CheckSourceActiveFilter:
                source.oclIsTypeOf(ActiveFilter)=false
```

  - Une pipe « SynchronousPushPipe » avec un filtre cible actif.

```
context SynchronousPushPipe
        inv invariant_CheckTargetActiveFilter:
                target.oclIsTypeOf(ActiveFilter)=false
```

Le point commun de ces deux cas est la présence d'une invocation directe d'une méthode d'un filtre actif. Or, un filtre actif ne contient ni de méthode « push » ni de méthode « pull ».

Dans le cas de l'utilisation d'une pipe synchrone, un seul filtre des deux situés aux bouts de la connexion peut accéder à l'opération effectuée par la pipe. Pour éviter les problèmes et simplifier la tâche de l'utilisateur, il est nécessaire de modifier la visibilité

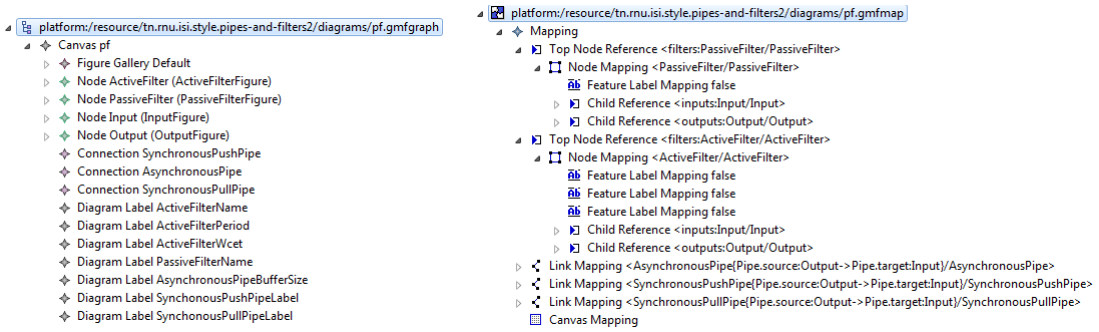
du port concerné. Il ne sera plus accessible à l'intérieur de son filtre parent. L'attribut « visible » des ports Input et Output offre la possibilité de les masqués.

4.2 Syntaxe Concrète

La syntaxe abstraite est le cœur du langage, à partir duquel la syntaxe concrète est définie. Une syntaxe abstraite peut avoir plusieurs syntaxes concrètes graphiques et/ou textuelles. Après avoir défini la syntaxe abstraite du langage, nous introduisons dans ce paragraphe une syntaxe concrète graphique qui sera utilisée par l'utilisateur pour manipuler les concepts de la syntaxe abstraite et en créer ainsi des instances conformes au méta-modèle.

La définition d'une notation graphique pour un langage passe par deux étapes. La première consiste à définir la syntaxe concrète en précisant les symboles graphiques à utiliser. La deuxième consiste à créer le mapping syntaxique entre les deux syntaxes. Pour concrétiser cet objectif, nous avons utilisé le framework GMF, un framework du projet EMP qui permet de construire une syntaxe concrète graphique pour des méta-modèles EMF.

Dans GMF, la syntaxe concrète et le mapping syntaxique sont définis sous forme de modèle. Le modèle instance de « gmfgraph » (figure 4.3a) définit les différents éléments graphiques (les nœuds, les liaisons et leurs détails de représentation graphique). Le modèle instance de « gmfmap » (figure 4.3b) fait la correspondance entre les éléments du formalisme graphique et les concepts du méta-modèle.



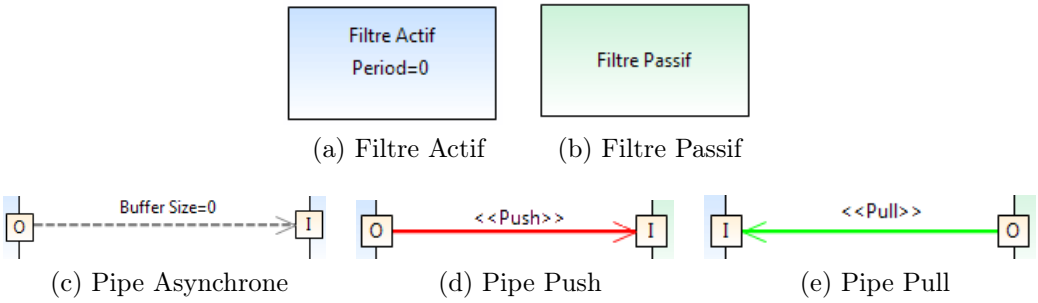
(a) Modèle de la syntaxe concrète (b) Modèle du mapping syntaxique

FIGURE 4.3 – Modèles de la syntaxe concrète

Pour bénéficier de GMF, le modèle de la syntaxe abstraite doit respecter certaines contraintes imposées par le framework sur la structure. En respectant ces conditions, nous pouvons générer automatiquement un éditeur graphique pour les instances de modèle.

Les concepts définis par la syntaxe abstraite et leurs correspondants dans la syntaxe concrète sont associés comme suit (figure 4.4) :

- Les filtres actifs sont représentés par des rectangles bleus.
- Les filtres passifs sont représentés par des rectangles verts.
- Les ports de chaque filtre sont représentés par des petits carrés accrochés sur son bord.
- Les ports d'entrée se distinguent par la lettre I et les ports de sortie de distinguent par la lettre O.
- Les pipes synchrones sont représentées par des flèches continues.
- Les pipes asynchrones sont représentées par des flèches discontinues.
- Le sens de la flèche des différentes pipes indique le sens du flux des données véhiculées.



(a) Filtre Actif (b) Filtre Passif

(c) Pipe Asynchrone (d) Pipe Push (e) Pipe Pull

FIGURE 4.4 – Représentation graphique

4.3 Sémantique Formelle

Jusqu'ici, nous avons défini la syntaxe abstraite et la syntaxe concrète. A cette étape, il est possible de construire des éditeurs graphiques pour le langage. Ces éditeurs seront capables de vérifier la conformité des instances de modèle créées à la structure imposée par le méta-modèle et les contraintes structurelles précédemment définies. L'étape suivante consiste à définir la sémantique opérationnelle en appliquant l'approche proposée dans le chapitre précédent. La définition de la sémantique dynamique des éléments (composants

et connecteurs) définis par le style architectural Pipes et Filtres revient à définir la sémantique dynamique des connecteurs. Le style n'associe pas à ses composants (filtres) un comportement dynamique précis. Ce comportement dépend de l'instance concrète du composant.

Nous avons appliqué l'approche sur trois connecteurs avec des variations selon deux critères : copie des données véhiculées et blocage pour les connecteurs multi-threads. Tous nos connecteurs sont de type flux de données. Les différents connecteurs et leurs variantes sont classés comme suit :

- Connecteurs Synchrones :
  - Push Pipe :
    - sans copie
    - avec copie
  - Pull Pipe :
    - sans copie
    - avec copie
- Connecteurs Asynchrones
  - Pipe :
    - bloquant avec copie
    - bloquant sans copie
    - non bloquant avec copie
    - non bloquant sans copie

Nous avons donc en totalité deux classes de connecteurs (synchrone et asynchrone), trois types de connecteurs (Push Pipe, Pull Pipe et Pipe) et huit variantes.

Chaque connecteur présente une interface composée d'une ou plusieurs opérations possibles invocables à partir d'un port de l'un ou de l'autre des deux bouts de la connexion. Le comportement de chaque opération associée au connecteur est défini par un ensemble de règles de transformation.

Nous avons choisi de présenter deux connecteurs : le premier est un connecteur synchrone, Push Pipe sans copie, et le deuxième est un connecteur asynchrone, Pipe avec accès bloquant sans copie. Chacun d'entre eux représente une classe particulière (synchrone et asynchrone) et représente un type différent (mono-thread et multi-threads).

Pour chaque connecteur nous devons définir manuellement son méta-modèle du domaine syntaxique et un ensemble de règles définissant le comportement dynamique des

opérations du connecteur. Le reste est généré automatiquement avec les différents outils utilisés (l'interpréteur QVTo, Ecore2Groove et le simulateur Groove).

4.3.1 Push Pipe : connecteur synchrone sans copie

Dans le méta-modèle du connecteur, nous représentons uniquement les éléments et les concepts nécessaires à détailler le fonctionnement interne du connecteur. Toutes les données ajoutées aux éléments du connecteur et utilisables par des éléments externes non utilisés pour décrire sa sémantique dynamique sont supprimées. Dans ce type de pipe, nous n'avons besoin d'aucun attribut au niveau des concepts définis. Seules les méthodes à invoquer lors de la définition de la dynamique du connecteur sont représentées.

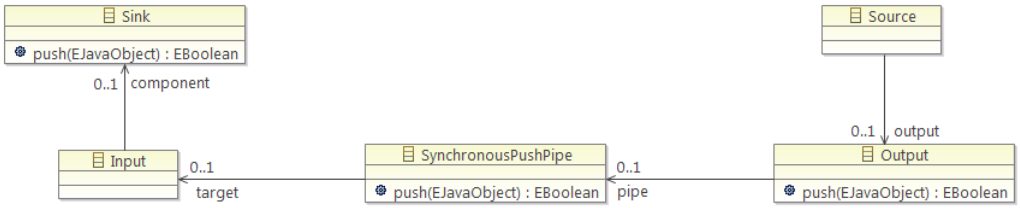


FIGURE 4.5 – Méta-modèle syntaxique

Nous avons maintenant un méta-modèle (figure 4.5) auquel nous voudrions définir la sémantique dynamique. L'étape suivante dans notre approche consiste à définir le mapping vers le domaine sémantique. Le domaine sémantique sera lui aussi représenté par un méta-modèle. Ce méta-modèle sémantique est obtenu en appliquant une transformation de modèle définie en QVTo. Cette transformation représente formellement le mapping entre le domaine syntaxique et le domaine sémantique.

Notre connecteur est un connecteur synchrone dans lequel un seul thread ou processus est en jeu. Nous avons défini deux transformations pour le mapping sémantique : une transformation vers un espace mono-thread et une autre vers un espace multi-threads. Dans ce premier cas, nous utilisons la transformation mono-thread et dans le connecteur suivant nous utiliserons la deuxième transformation multi-threads.

La transformation se résume en deux règles globales. La première règle est la transformation des différents concepts du méta-modèle source vers le méta-modèle cible, à l'exception des opérations. Ainsi, les classes, les attributs et les relations sont réutilisés intégralement. La deuxième règle est l'ajout des différents concepts nécessaires à modé-

liser la dynamique des opérations. Cette deuxième règle se divise en deux parties. La première partie consiste à créer une relation d'héritage entre toutes les classes obtenues suite à l'application de la première règle et une classe parente « RObject » (Root Object). La deuxième partie consiste à ajouter le concept « MethodFrame » qui représente une méthode invoquée et l'ensemble de ces « MethodFrame » représente la pile d'appels des méthodes. L'appel d'une méthode commence par la création de l'instance de « MethodFrame » et se termine par sa destruction. Une « MethodFrame » possède un seul attribut « methodName », définissant le nom de la méthode invoquée (défini dans le méta-modèle de la syntaxe abstraite). Différentes relations sont y reliées. La relation « this » définie l'objet sur lequel la méthode est appelée. La relation « previous » définie la « MethodFrame » précédente (la méthode qui a appelé la méthode courante). Les autres relations définissent les paramètres des méthodes et la valeur de retour.

L'application de cette transformation avec l'interpréteur QVTo nous donne le méta-modèle suivant :

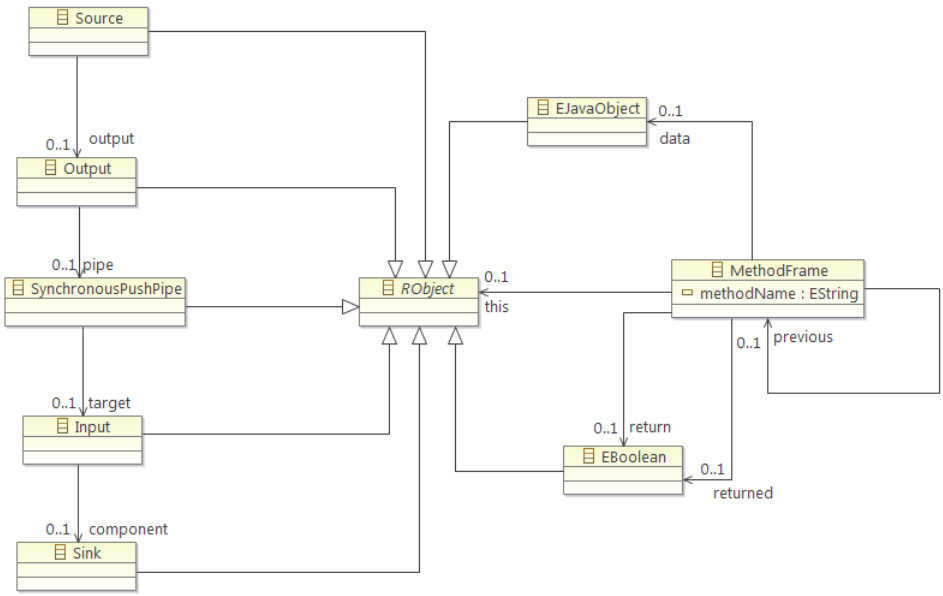


FIGURE 4.6 – Méta-modèle sémantique

Après la génération de ce méta-modèle, nous nous retrouvons à l'intérieur du domaine sémantique. L'étape suivante est exercée avec l'outil Ecore2Groove que nous avons adapté à notre besoin. En fait, l'outil original, dans sa génération, prend soin de reproduire tous les concepts définis au niveau du méta-modèle Ecore. Nous avons éliminé toute la partie

inutile dans notre contexte, afin de simplifier l'affichage des graphes de définition de type.

En fournissant en entrée de l'outil Ecore2Groove, le méta-modèle précédent, nous obtenons une nouvelle grammaire Groove, contenant un graphe de type (figure 4.7) et un graphe initial (figure 4.8) générés automatiquement. Il nous reste l'ensemble des règles de transformation de graphe.

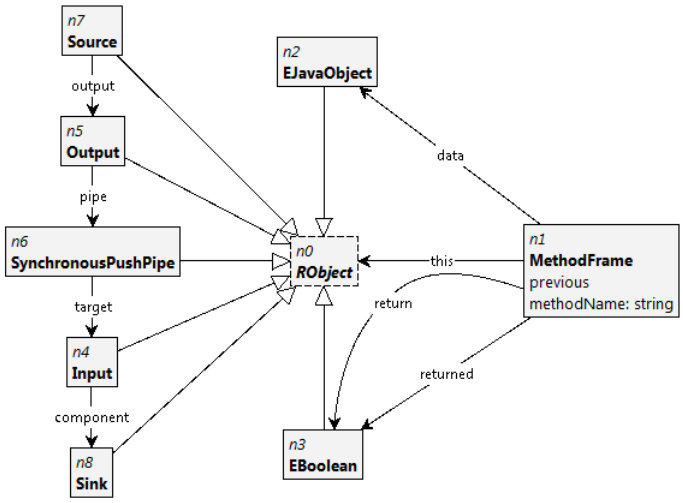


FIGURE 4.7 – Graphe de type

Avant de définir les règles, il faudrait comprendre le principe de ce connecteur. Le composant source de données invoque la méthode « push » de notre connecteur en lui passant comme paramètre les données à transférer. L'interface du connecteur contient une seule opération : la méthode « push » accessible à partir du port de type Output. Le connecteur « Push Pipe » ne fait aucun traitement sur les données reçues, son rôle se limite au transfert des données. La méthode « push » en question fait appel à la méthode « push » de « SynchronousPushPipe ». Cette dernière invoque elle-même la méthode « push » du composant relié au port de type Input. Nous avons donc au niveau de la pipe deux délégations consécutives.

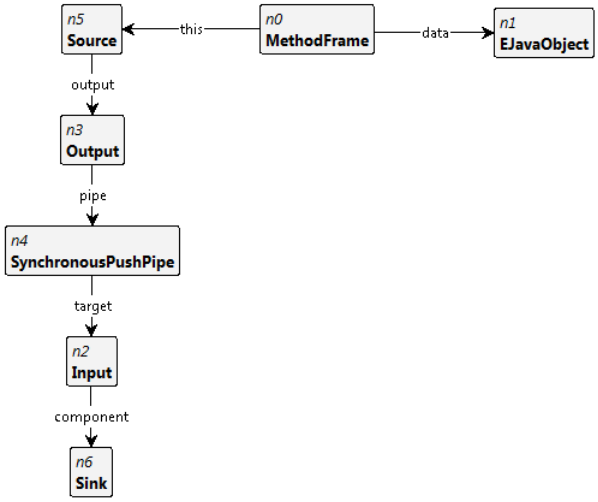
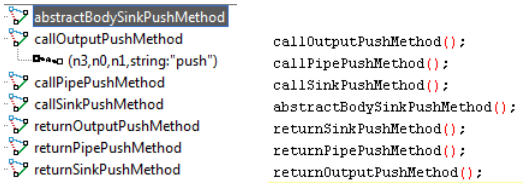


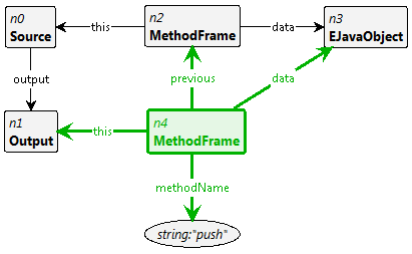
FIGURE 4.8 – Graphe initial

Nous partons d'un graphe de départ (figure 4.8) dans lequel une méthode du composant source est invoquée. C'est elle qui fait appel au service de la pipe. Suite à l'accès à la pipe nous aurions deux délégations. Les méthodes délégantes ne comprennent pas de traitement. Pour chacune nous définissons deux règles : une qui lance la « MethodFrame » et une qui la détruit et renvoie une valeur de retour. La seule méthode qui contient un traitement est la méthode « push » du « Sink ». Nous exprimons le comportement de cette méthode d'une manière abstraite, en créant l'objet à retourner. Les règles ne contiennent pas de transformations complexes. Ces dernières se limitent à créer les « MethodFrame » et les détruire. L'application des règles est contrôlée par un programme de contrôle défini avec le langage de l'outil. Ce programme simplifie la définition des règles.

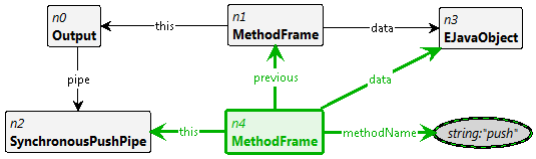


(a) Règles (b) Contôle

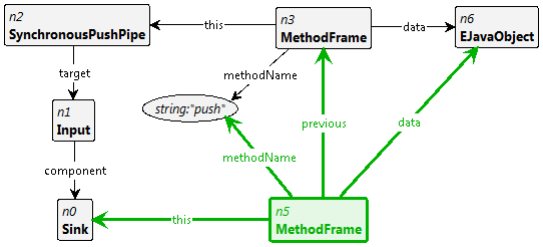
FIGURE 4.9 – Règles et script de contrôle



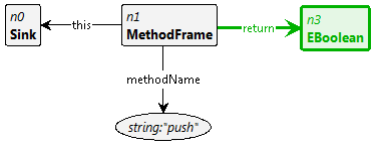
(a) callOutputPushMethod



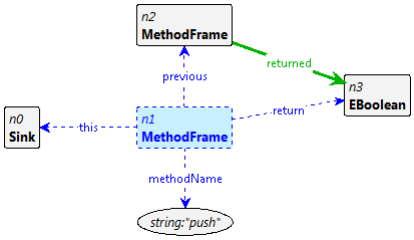
(b) callPipePushMethod



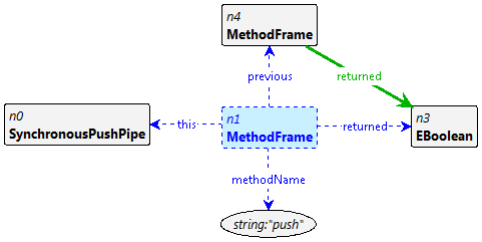
(c) callSinkPushMethod



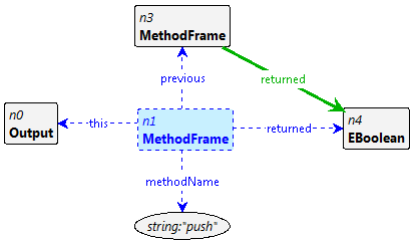
(d) abstractBodySinkPushMethod



(e) returnSinkPushMethod



(f) returnPipePushMethod



(g) returnOutputPushMethod

FIGURE 4.10 – Règles de transformation

Nous avons en place tous les éléments nécessaires pour produire le système de transition définissant la dynamique du connecteur. Les états de ce système sont des graphes. Après l'utilisation de l'explorateur automatique des états atteignables par notre système de production, nous obtenons le système état/transition de la figure 4.11. L'état en vert est

le graphe de départ. L'état en orangé est le graphe final. Sur chaque transition, représentée avec une flèche, il est indiqué le nom de la règle appliquée sur le graphe source pour avoir le graphe cible.

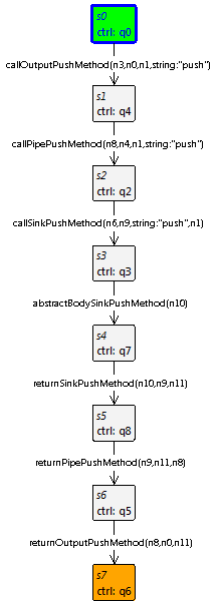


FIGURE 4.11 – Système de transition de graphes

4.3.2 Pipe : connecteur asynchrone avec accès bloquant et sans copie

Identiquement au connecteur précédent, le méta-modèle syntaxique ne contient que les concepts nécessaires à l'expression de la sémantique dynamique interne. Dans le méta-modèle de la pipe asynchrone, nous avons besoin d'un attribut exprimant le nombre maximal d'éléments à retenir dans le tampon. Nous avons aussi besoin d'un buffer dans lequel nous stockons les données échangées. Ce connecteur asynchrone contient deux méthodes. Une méthode « push », qui ajoute des données au buffer du connecteur, invocable à partir du port Output. Une méthode « pull », qui retire des données à partir du buffer, invocable à partir du port Input.

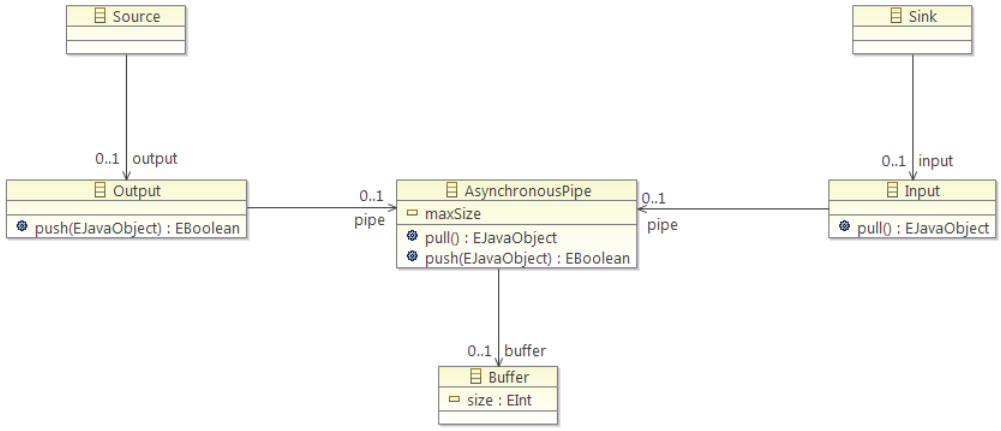


FIGURE 4.12 – Méta-modèle syntaxique

Ce type de connecteur forme un pont de communication entre deux threads. Les deux méthodes du connecteur sont chacune invoquée à partir d'un thread différent. Dans la deuxième étape de l'application de notre approche de définition de la sémantique sur le connecteur Pipe, nous utilisons donc la deuxième transformation de modèle (transformation vers un environnement multi-threads). En appliquant cette transformation, définie en QVTo, sur le méta-modèle de la syntaxe abstraite du connecteur (figure 4.12), nous obtenons le méta-modèle du domaine sémantique (figure 4.13).

La transformation multi-threads applique les mêmes règles de transformation que la première transformation précédemment définie, tout en ajoutant au modèle quelques concepts et relations nécessaires à une exécution multi-threads. La classe « RObject » contient un attribut boolean reflétant si l'accès à l'objet est bloqué ou non. La relation « locker » détermine l'instance de méthode qui a réservé l'objet bloqué. Une méthode en exécution peut être suspendue et reste en attente jusqu'elle reçoive l'autorisation à accéder à un objet particulier. La relation « wait » pointe vers cet objet.

Tous ces éléments du méta-modèle permettent de décrire la sémantique dynamique dans un environnement multi-threads. Chaque sous graphe formé par un ensemble connexe de « MethodFrame » interconnectées représente un thread à part.

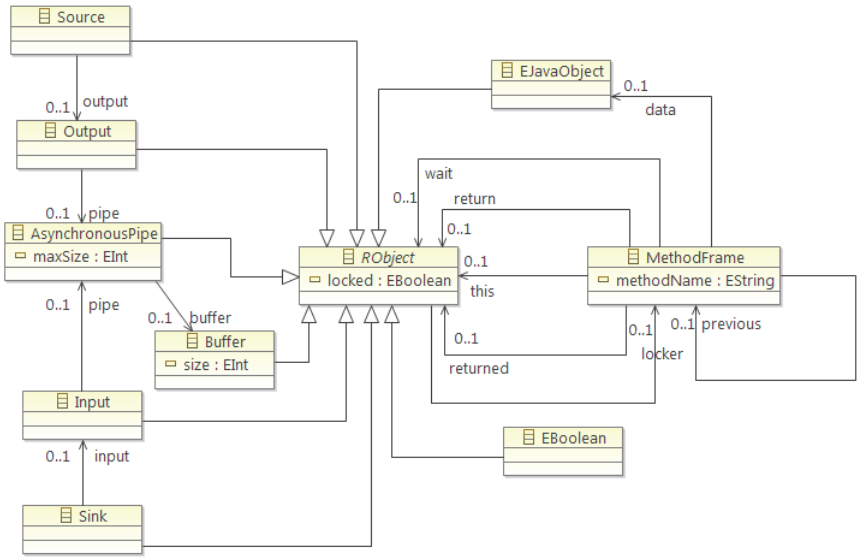


FIGURE 4.13 – Méta-modèle sémantique

En utilisant l'outil Ecore2Groove, nous générons automatiquement une nouvelle grammaire dans l'outil Groove contenant un graphe de type (figure 4.14) et un graphe initial (figure 4.15) dont la structure est conforme à la structure imposée par le graphe type. Nous avons besoin d'ajouter quelques relations pour pouvoir gérer le contenu du buffer. Ces relations sont « first », « last » et « next ».

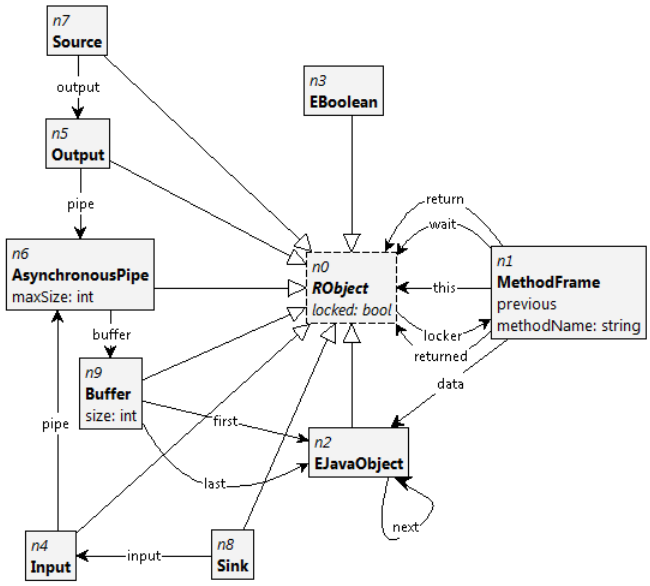


FIGURE 4.14 – Graphe de type

La pipe relie deux threads concurrents. Le premier thread, source de données, utilise la fonction « push » pour envoyer les données au deuxième thread via la pipe. Le composant source de donnée fait appel à la méthode « push » du port de type Output. Le port délègue la tâche à la méthode « push » de la pipe. Le deuxième thread, destination des données, utilise la fonction « pull » pour récupérer les données à partir du deuxième thread via la pipe. Le composant récepteur de données fait appel à la méthode « pull » du port de type Input. Le port à son tour délègue la tâche à la méthode « pull » de la pipe. Pour la méthode « push » si le buffer de la pipe est plein le thread passe à l'état de suspension jusqu'à la disposition d'un emplacement vide. A ce moment le thread peut reprendre sa tâche. Pour la méthode « pull », si le buffer est vide le thread passe à l'état de suspension jusqu'à la présence des données. La politique de gestion du contenu du buffer est FIFO (First In First Out).

Dans le graphe de départ, nous avons deux méthodes invoquées en parallèle. Chacune des « MethodFrame » appartient à un thread à part entière.

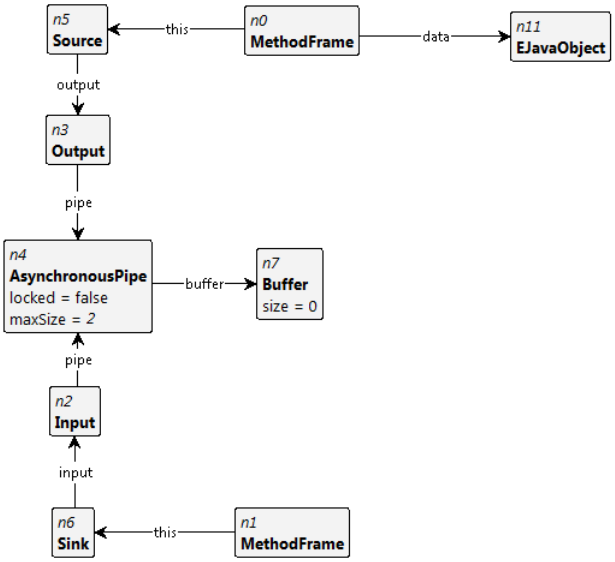


FIGURE 4.15 – Graphe initial

Le rôle du connecteur précèdent était une simple délégation. A chacune des méthodes, nous associons une règle pour le lancement d'une « MethodFrame » et une deuxième règle pour terminer l'appel de la méthode et renvoyer la valeur de retour. Le connecteur courant contient des traitements et détient des données stockées. Les règles correspondantes aux

méthodes « push » et pull sont plus nombreuses et plus complexe (figure 4.16). Dans ces règles, nous effectuons des modifications et des tests sur les valeurs des attributs.

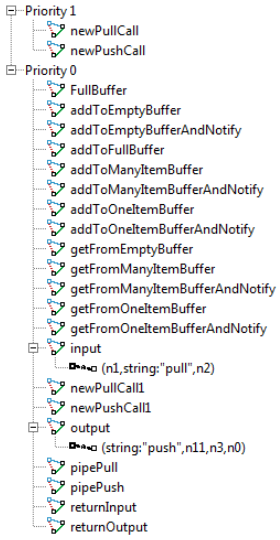


FIGURE 4.16 – Règles de transformation

Les règles sont nombreuses. Dans la figure suivante, nous présentons un seul exemple de règle contenant des tests logiques et des modifications des valeurs de certains attributs.

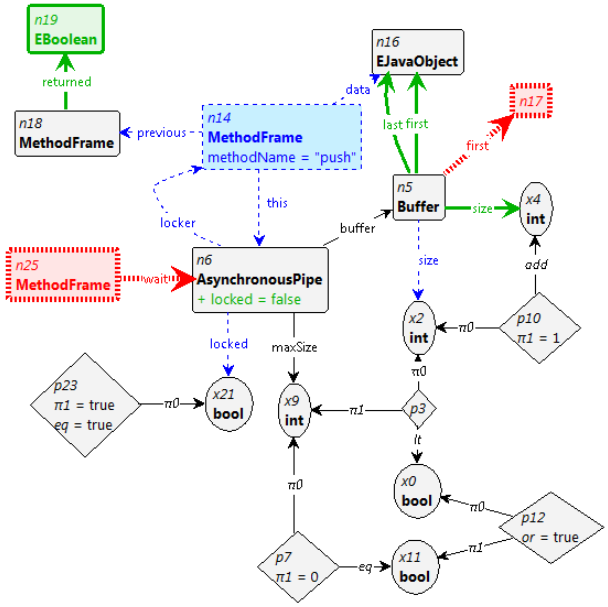


FIGURE 4.17 – Règle addToEmptyBuffer

Cette règle décrit le comportement de la méthode « push » de la pipe asynchrone dans le cas où le buffer est vide et il n'existe pas de méthode qui attend la suppression du blocage. Son rôle est d'ajouter le paramètre de la méthode au buffer, de terminer l'exécution de la méthode push et de renvoyer à la méthode appelante une valeur de retour de type booléen.

Tous les éléments du système de production de graphes sont définis. A l'aide du simulateur de l'outil Groove nous explorons automatiquement tous les états du système de transition simulant la dynamique du connecteur. Les états de ce système sont des graphes. L'état en vert est le graphe de départ. A la différence du connecteur précèdent, le système de transitions de ce connecteur ne présente aucun état final. Sur chaque transition, il est indiqué le nom de la règle appliquée. Le connecteur asynchrone présente une dynamique plus complexe que le premier connecteur synchrone. Pour un buffer de taille maximale 2, le nombre total des états est 94 et le nombre total des transitions est 182.

Chapitre 5

Outil

Après une présentation formelle du langage xPFL dans le chapitre précédent, nous présentons dans ce dernier chapitre les outils développés conformément à cette définition formelle du langage. Ces outils sont développés sous forme de plugins pour la plateforme Eclipse. Pour réaliser ces plugins, nous avons utilisé les frameworks EMF, GMF et Xpand du projet « Eclipse Modeling Project ». Sur la figure 5.1 ces frameworks sont représentés avec des rectangles arrondis.

L'outil est composé principalement de trois éléments qui sont :

– un éditeur graphique,
– un interpréteur,
– et un générateur de code en Java.

La réalisation de tous ces éléments dépend du framework EMF qui permet de définir la syntaxe abstraite du langage et fournit des interfaces taillées sur mesure pour manipuler les modèles instance du méta-modèle. Dans l'implémentation concrète de ces interfaces, plus précisément dans le corps des opérations définies au niveau du méta-modèle, nous avons ajouté le comportement dynamique en utilisant le langage Java.

Pour générer ces interfaces relatives au méta-modèle, il faudrait tout d'abord créer un méta-modèle sous forme d'un modèle Ecore. L'étape suivante consiste à générer, à partir de ce modèle, un deuxième modèle « genmodel », utilisé par le framework dans la génération du code. A partir de ce modèle, éventuellement personnalisé, nous générons les interfaces spécifiques au méta-modèle et un plugin nommé « edit ».

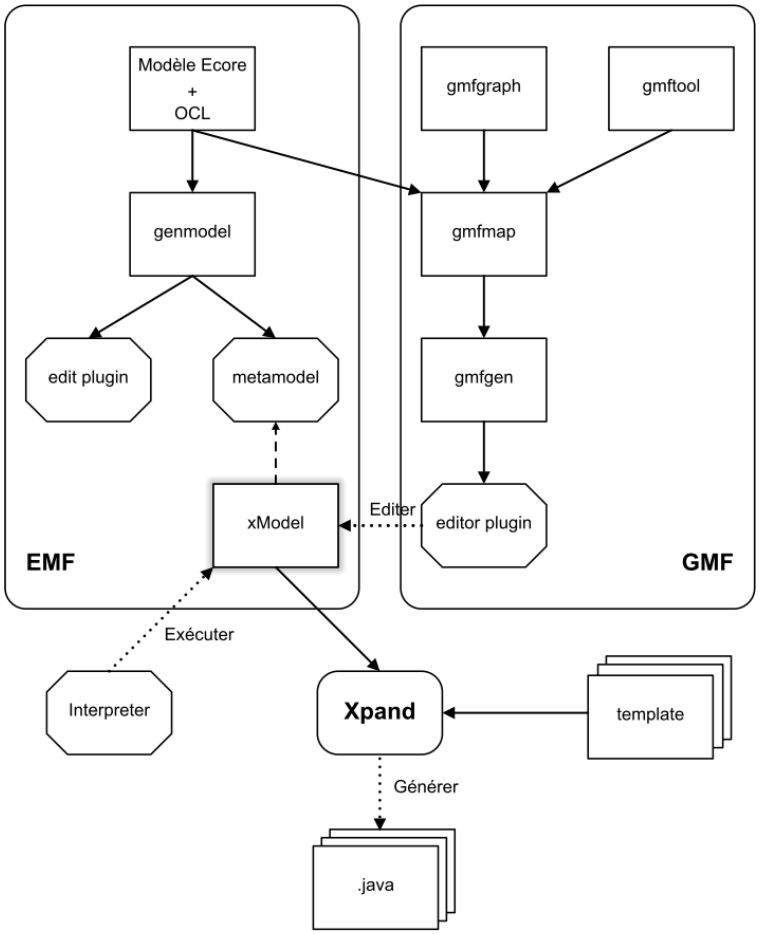


FIGURE 5.1 – Plugins développés

Nous utilisons EMF pour générer un plugin contenant les interfaces dédiées et leurs implémentations et un plugin d'édition utilisable par d'autres plugins. Le premier plugin sert à créer des instances et de les manipuler dynamiquement. Le deuxième est utilisé par le plugin de l'éditeur graphique. A cette étape, nous avons la possibilité de créer des instances du méta-modèle sous forme d'objets Java. L'étape suivante consiste à créer un éditeur graphique pour créer ces instances. Pour le faire nous utilisons le framework GMF.

5.1 Editeur Graphique

Comme nous l'avons déjà mentionné, en plus de la définition formelle de la syntaxe concrète d'un méta-modèle défini avec Ecore, GMF offre la possibilité de générer automa-

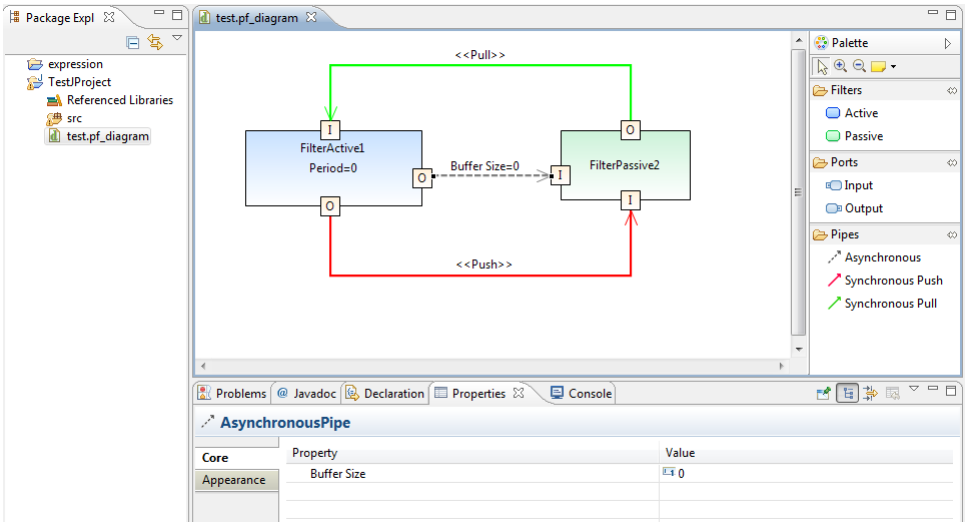


FIGURE 5.2 – Editeur

tiquement un éditeur graphique pour ce dernier. Pour le faire, il faudrait créer un ensemble formé de quatre modèles :

- un modèle Ecore définissant la syntaxe abstraite,
- un modèle « gmfgraph » définissant la syntaxique concrète avec un ensemble d'éléments graphiques (Nœuds et Connexions) et leur représentation visuelle,
- un modèle « gmftool » définissant la structure de la palette en termes de groupes et de leurs éléments sous-jacents,
- et un modèle de mapping « gmfmap » qui combine tous les trois modèles précédents en définissant des associations entre les concepts de chacun d'entre eux.

Avant de pouvoir générer l'éditeur graphique, il est nécessaire de générer un modèle « gmfgen ». Ce modèle sert à générer le code final de l'éditeur. Pour paramétrer la génération et la personnaliser, il faudrait en premier lieu modifier ce modèle avant de penser à modifier le code généré. A partir du modèle « gmfgen » et grâce à l'outil GMF Tools, nous obtenons le plugin de l'éditeur. Ce plugin utilise le plugin « edit » déjà généré.

L'éditeur est composé de trois parties :

- une palette contenant les concepts définis au niveau du méta-modèle,
- une zone de dessin contenant une représentation graphique de l'instance selon la syntaxe concrète définie,
- et une vue « Propriétés » offrant la possibilité de consulter et de modifier les attributs de l'élément sélectionné dans le modèle.

Avec cet éditeur, nous avons la possibilité de créer une instance conforme au méta-modèle avec les notations définies par la syntaxe concrète. Par ailleurs, nous avons besoin d'exécuter cette instance.

5.2 Interpréteur

Pour exécuter les instances de modèle créées, nous avons besoin d'un interpréteur. Une partie de cet interpréteur existe déjà dans le corps des opérations relatives à chaque concept du méta-modèle. Derrière la représentation visuelle de l'instance du modèle, l'éditeur détient une autre représentation de cette même instance, sous forme d'objets Java, instances des classes définies dans le plugin « metamodel » généré avec EMF. Nous avons juste besoin d'un élément dans l'environnement qui exécute la fonction « execute » de « PFStyle » qui se préoccupe de lancer les threads nécessaires. Cette action est réalisée dans l'éditeur à travers le menu contextuel de l'environnement.

Le comportement de chaque concept est décrit par un ensemble d'opérations. L'enchaînement d'exécution de ces différentes opérations à partir de l'opération principale (l'opération « execute » de « PFStyle ») représente le comportement dynamique du modèle. Pour notre langage, seule la sémantique des connecteurs est définie avec précision. La sémantique comportementale des composants reste abstraite. En effet, la sémantique de l'opération « run » des filtres actifs ne peut pas être définie au niveau du méta-modèle puisque elle dépend de l'instance du filtre (de même pour les opérations des tâches des filtres passifs). Chaque filtre possède sa propre sémantique pour la méthode « run ». Par conséquent, dans notre définition formelle de la sémantique du langage, nous nous sommes contentés de définir la sémantique des connecteurs. L'utilisateur a donc besoin de spécifier pour chaque instance concrète du composant filtre un fonctionnement interne spécifique permettant de transformer les données reçues en entrée en des données à envoyer en sortie.

Les méthodes « run » constituent ainsi des points de variation sémantique dans notre méta-modèle. Dans notre implémentation, nous devons donc proposer une solution à l'utilisateur pour lui permettre de modifier le comportement de chacune de ces méthodes au niveau du modèle créé.

Dans la littérature, la définition du langage intitulé : UML4SPM (Langage De Modélisation De Procédés De Développement Logiciel Exécutable Et Orienté Modèle) a confrontée le besoin d'ajout d'un comportement spécifique (non existant au niveau du méta-modèle)

au niveau du modèle. UML4SPM adopte l'approche suivante. Le code java est contenu dans une variable de type String. Au moment de l'exécution ce code est récupéré à partir de cette variable. Par la suite, une classe Java est créée avec une méthode bien précise (nommée « executeBody ») renfermant le code Java extrait précédemment. Ensuite, la classe générée est compilée. Enfin, la classe est chargée et instanciée en utilisant l'API Java Reflect, et la méthode « executeBody » est invoquée.

Notre solution adopte une approche différente de celle-ci. Au lieu de définir le code Java dans une propriété au niveau du modèle, nous utilisons les services offerts par le JDT (Eclipse Java Developpement Tools) pour créer une classe Java contenant le code désiré. Pour un filtre actif, cette classe devra implémenter l'interface « ActiveFilterVisitor » définie au niveau du méta-modèle :

```
interface ActiveFilterVisitor {
        void run(ActiveFilter filter);
}
```

Dans la variable, nous nous contentons d'affecter le nom complet de la classe créée et compilée avant l'exécution.

Au niveau de la méthode « run » du méta-modèle, nous utilisons un ClassLoader pour charger dynamiquement la classe Java dont le nom est spécifié avec l'attribut « visitorClassName » de la classe « Filter ». En faisant recours à l'API Java Reflect, nous instancions la classe chargée et nous exécutons sa méthode « run ». Nous pouvons dire que le corps de la méthode « run » du méta-modèle ira chercher le corps de sa méthode.

Avec cette approche, pour résoudre le problème rencontré, nous utilisons une version modifiée du pattern GOF : Visiteur. Cette solution architecturale pour le problème de points de variation sémantique ressemble au pattern Liaison Dynamique (Dynamic Linkage) décrit dans [Gra02].

Le pattern Visiteur classique est représenté par le code source Java :

```
interface ActiveFilterVisitor {
        void run(ActiveFilter filter);
}

class ActiveFilter {
        void run(ActiveFilterVisitor visitor){
                visitor.run(this);
        }
}
```

L'adaptation du pattern Visiteur est représentée par le code source Java :

```
interface ActiveFilterVisitor {
        void run(ActiveFilter filter);
}

class ActiveFilter {
        // Valeur affectée via la vue propriétés avec l'assistant
        private String visitorClassName;

        void run(ActiveFilterVisitor visitor){
                ...
                // Charger la classe
                ClassLoader cl = new URLClassLoader(...);
                Class aClass = cl.loadClass(visitorClassName);
                // Créer une instance
                ActiveFilterVisitor visitor =
                        (ActiveFilterVisitor) aClass.newInstance();
                // Exécuter la méthode
                visitor.run(this);
        }
}
```

Les deux classes « ActiveFilterVisitor » et « PassiveFilterVisitor » définies dans le méta-modèle, n'ont pas de correspondants dans la syntaxe concrète. Elles ne sont pas directement utilisables dans les modèles instances.

Dans la spécification du comportement particulier à chaque filtre, nous utilisons les assistants du JDT. A partir du filtre et avec l'assistant de création de classe Java, nous pouvons lancer la création d'une classe implémentant l'interface visiteur correspondante au type du filtre. Nous offrons aussi la possibilité de choisir une classe déjà existante au sein du projet avec l'assistant de sélection d'une ressource du projet.

5.3 Générateur de Code

Avec l'éditeur nous éditons un modèle et avec l'interpréteur nous l'exécutons. Nous possédons jusqu'ici des instances de modèle exécutables et réellement productives. Sauf que ces modèles instances, représentées avec des objets Java, contiennent des données et des traitements supplémentaires non utilisés dans l'application à déployer. Ces données et traitements inutiles dans l'environnement de déploiement sont hérités du framework EMF. Nous avons donc besoin de générer à partir du modèle un code final à installer.

Nous utilisons le framework Xpand pour générer le code final à partir du modèle. Nous définissons des templates pour une génération de code qui garde le pattern visiteur et des templates qui supprime le pattern visiteur puisqu'il n'est plus nécessaire dans le code final. La génération est lancée à partir d'un assistant avec lequel nous la paramétrons.

5.4 Exemple

Dans cette section, nous utilisons l'ensemble des outils avec un exemple de démonstration. Notre exemple comprend quatre filtres. Le premier filtre est un filtre actif. Son rôle est de lire le contenu d'un fichier et le segmenter. Chaque segment est envoyé au filtre suivant. Le filtre suivant est un filtre actif qui permet de compter le nombre de mots qui ont véhiculés à travers de lui. Si le segment reçu n'est pas un mot il le renvoie au filtre suivant. Le dernier filtre dans la chaine de traitement est un filtre actif qui permet de compter le nombre total de nombres reçus.

Les deux filtres de comptage utilisent les deux un filtre passif pour afficher le résultat final de leur traitements. Le filtre passif « Afficheur » affiche les résultats sur la console.

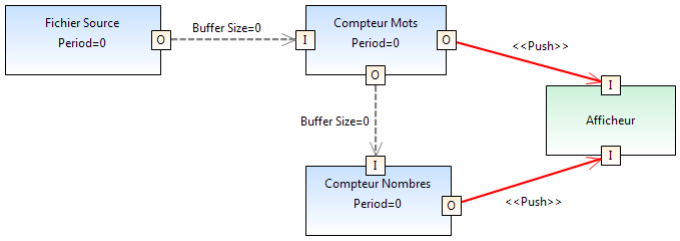


FIGURE 5.3 – Modèle de l'exemple

Après la définition des classes des traitements correspondants à chaque filtre. Nous avons exécuté le modèle sur un fichier de code source Java. Nous avons obtenu sur la console l'affichage suivant :

```
Nombre de mots : 106
Nombre de nombres : 4
```

FIGURE 5.4 – Résultat sur la console

Nous avons enfin utilisé le générateur de code pour générer le code final en Java.

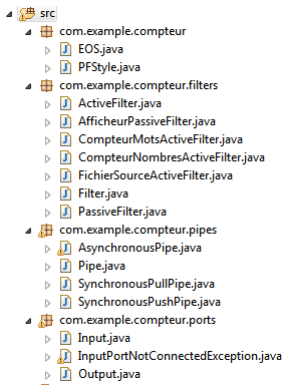


FIGURE 5.5 – Résultat de la génération

Le code généré est composé de quatre paquetages Java. Le premier paquetage contient la classe principale. Les trois autres paquetages contiennent chacun les classes relatives aux filtres, ports et pipes.

Chapitre 6

Conclusion et Perspectives

Conclusion

Les styles architecturaux permettent de décrire l'architecture d'un système logiciel avec un modèle dont les entités sont définies à un haut niveau d'abstraction. Ces entités sont principalement les composants et les connecteurs. L'ingénierie des modèles, de son côté, vise à utiliser les modèles, dans le processus de développement logiciel, non seulement pour des fins de description et de documentation mais pour qu'ils soient aussi des artéfacts productifs au sein du processus. De ce fait, la notion de modèles exécutables a fait son apparition. L'approche MDE propose aussi de définir un langage dédié au domaine des styles architecturaux.

Dans le cadre de ce mémoire, nous avons appliqué les principes de l'ingénierie des modèles pour pouvoir exécuter les instances du style architectural Pipes et Filtres. Pour le faire, nous avons défini un langage de modélisation exécutable dédié à ce style architectural. Un langage est complètement défini en précisant sa syntaxe abstraite, sa syntaxe concrète et sa sémantique. Il existe plusieurs possibilités formelles pour la définition des deux types de syntaxe. Nous avons utilisé le framework EMF dans la définition de la syntaxe abstraite et le framework GMF dans la définition de la syntaxe concrète. Contrairement à la syntaxe, la définition de la sémantique formelle des modèles ne possède pas encore d'approches matures et standardisées. Le coeur de notre travail était donc de définir la sémantique formellement.

L'approche proposée consiste à transformer le méta-modèle syntaxique vers un méta-

modèle sémantique. Cette transformation, qui est définie en QVTo, ajoute aux concepts syntaxiques des concepts nécessaires à la description du comportement dynamique. Le méta-modèle sémantique définit tous les concepts à utiliser dans le domaine sémantique. Le domaine sémantique que nous avons choisi est la transformation de graphe. En utilisant un outil de transformation de graphe, nous définissons un système de production de graphe constitué d'un graphe de départ, d'un graphe de type et d'un ensemble de règles de transformation correspondantes aux méthodes définies dans le méta-modèle syntaxique. Avec ce système, nous générons un système état/transition détaillant la sémantique comportementale.

En plus de la définition formelle du langage xPFL, nous avons aussi développé un outil sous forme de plugins pour la plateforme Eclipse. L'outil est composé de trois éléments : un éditeur graphique des instances de modèle, un interpréteur de ces instances et un générateur de code final en Java.

Pour les connecteurs de notre style architectural, nous avons pu définir leur sémantique au niveau du méta-modèle. Par contre pour les composants (les filtres), nous n'avons pas pu le faire au niveau du méta-modèle, puisque ces derniers présentent des points de variation sémantique. Grâce à des choix architecturaux, nous avons donc proposé un mécanisme de raffinement de la sémantique de certaines méthodes lors de la création des modèles. Cette possibilité de préciser le comportement d'une méthode au niveau de l'instance du modèle résout le problème de conflit entre l'abstraction et l'exécutabilité.

Perspectives

Les perspectives identifiées sont les suivantes :

- Il est judicieux d'ajouter à la simulation des modèles instance du méta-modèle du style Pipes et Filtres la possibilité d'animer l'exécution avec une représentation graphique des flux de données.

- Il est aussi intéressant d'offrir la possibilité d'analyse et de vérification des instances d'architecture créées. Suite à l'analyse, des suggestions devraient, éventuellement, être proposées.

Bibliographie

[AC07] F. T. S. G. A. Cuccuru, C. Mraidha. "*Métamodèles et Points de Variation Sémantique.*" In "Atelier sur la Sémantique des Modèles (SéMo'07)," 2007.

[AGG] "AGG : The Attributed Graph Grammar System." http ://user.cs.tu-berlin.de/~gragra/agg/index.html.

[Are03] M. Arends. "*A Simulation of the Java Virtual Machine using Graph Grammars.*" Master's thesis, University of Twente - Department of Computer Science, November 2003.

[Aza07] S. Azaiez. "*Approche Dirigée par les Modèles pour le Développement de Systèmes Multi-agents.*" Ph.D. thesis, Université de Savoie, 2007.

[BD07] J.-P. Babau and J. Deantoni. "*Architectures Logicielles pour Les systèmes Embarqués Temps Réel.*" In "La 5ème Ecole d'été Temps Réel (ETR'07)," 2007.

[Ben07] R. Bendraou. "*UML4SPM : Un Langage De Modélisation De Procédés De Développement Logiciel Exécutable Et Orienté Modèle.*" Ph.D. thesis, Université Pierre et Marie Curie, 2007.

[BGBJ08] R. Bendraou, et al. "*Vers l'Exécutabilité des Modèles de Procédés Logiciels.*" In "Langage Modèles et Objets LMO'08," 2008.

[BMR+96] F. Buschmann, et al. *"Pattern-Oriented Software Architecture, Volume 1 : A System of Patterns".* Wiley, 1996. ISBN 978-0-471-95869-7.

[Bus98] F. Buschmann. "*Real-time Constraints as Strategies.*" In "Proc. EUROPLOP 98, Third European Conf. on Pattern Languages of Programming and Computing," pages 1–1. 1998.

[Béz05] J. Bézivin. "*On the Unification Power of Models.*" Software and Systems Modeling (SoSym), 4 :171–188, 2005.

[Com08] B. Combemale. "*Approche de Métamodélisation pour la Simulation et la Vérification de Modèle - Application à l'Ingénierie des Procédés.*" Ph.D. thesis, Institut National Polytechnique, Université de Toulouse, 2008.

[CRC+06] B. Combemale, et al. "*Expériences pour Décrire la Sémantique en Ingénierie des Modèles.*" In H. Sciences/Lavoisier, editor, "2ième journées sur l'Ingénierie Dirigée par les Modèles (IDM, in french)," pages 17–34. 2006.

[CSAJ05] K. Chen, et al. "*Semantic Anchoring with Model Transformations.*" In A. Hartman and D. Kreische, editors, "Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings," volume 3748 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2005. ISBN 3-540-30026-0.

[DLC10] S. Diaw, R. Lbath, and B. Coulette. "*Etat de l'Art sur le Développement Logiciel basé sur les Transformations de Modèles.*" Technique et Science Informatiques, Ingénierie Dirigée par les Modèles, 29(4-5/2010) :505–536, 2010.

[DS08] M. P. E. M. Dave Steinberg, Frank Budinsky. *"EMF : Eclipse Modeling Framework, 2nd Edition"*. Addison-Wesley Professional, 2008. ISBN 0-321-33188-5.

[EH00] G. Engels and R. Heckel. "*Graph Transformation and Visual Modeling Techniques.*" Bulletin of the EATCS, (71) :186–202, 2000.

[FMRS07] C. Fuss, et al. "*A Comparison of AGG, Fujaba, and PROGRES.*" ECEASST, pages –1–1, 2007.

[Gra02] M. Grand. *"Patterns in Java, Volume 1- A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition"*. Wiley, 2002. ISBN 0471227293.

[Gro09] R. C. Gronback. *"Eclipse Modeling Project : A Domain-Specific Language (DSL) Toolkit"*. Addison-Wesley Professional, 2009. ISBN 0-321-53407-7.

[GRS09a] A. Gargantini, E. Riccobene, and P. Scandurra. "*A Semantic Framework for Metamodel-Based Languages.*" Automated Software Engg., 16 :415–454, 2009.

[GRS09b] A. Gargantini, E. Riccobene, and P. Scandurra. "*Integrating Formal Methods with Model-Driven Engineering.*" In "Proceedings of the 2009 Fourth International Conference on Software Engineering Advances," ICSEA '09, pages 86–92. IEEE Computer Society, 2009. ISBN 978-0-7695-3777-1.

[GTVG05] E. G. J. d. L. L. L. T. L. U. P. D. V. G. Taentzer, K. Ehrig and S. Varro-Gyapay. "*Model Transformation by Graph Transformation : A Comparative Study.*" In "MTiP '05 : Proceedings of the International Workshop on Model Transformations in Practice," 2005.

[Hau05] J. H. Hausmann. "*Dynamic Meta Modeling : A Semantics Description Technique for Visual Modeling Languages.*" Ph.D. thesis, University of Paderborn, 2005.

[HKM06] F. Hermann, H. Kastenberg, and T. Modica. "*Towards Translating Graph Transformation Approaches by Model Transformations.*" In G. Karsai and G. Taentzer, editors, "Proceedings of the Second International Workshop on Graph and Model Transformation," volume 4 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2006.

[HR04] D. Harel and B. Rumpe. "*Meaningful Modeling : What's the Semantics of Semantics ?*" Computer, 27 :64–72, 2004.

[Kai05] S. H. Kaisler. *"Software Paradigms"*. Wiley, March 2005. ISBN 0471483478.

[Kiw04] A. W. Kiwelekar. "*Architectural Connectors.*" Ph.D. thesis, Department of Computer Science and Engineering - Indian Institute of Technology, Bombay Mumbai, December 2004.

[Kle07] A. Kleppe. "*A Language Description is More than a Metamodel.*" In "Fourth International Workshop on Software Language Engineering," megaplanet.org, 2007.

[KM08] P. Kelsen and Q. Ma. "*A Lightweight Approach for Defining the Formal Semantics of a Modeling Language.*" In "Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems," MoDELS '08, pages 690–704. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-87874-2.

[KWB03] A. G. Kleppe, J. Warmer, and W. Bast. *"MDA Explained : The Model Driven Architecture : Practice and Promise"*. Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 032119442X.

[Lan09] K. Lano. *"UML 2 Semantics and Applications"*. John Wiley & Sons, Inc., 2009. ISBN 0470409088.

[MdM09] M. Z. Maarten de Mol. "*A GROOVE Solution for the GraBaTs 09 BPMN to BPEL Model Case Study.*" In "5th International Workshop on Graph-Based Tools," July 2009.

[MG04] T. Mens and P. V. Gorp. "*A Taxonomy of Model Transformation and its Application to Graph Transformation.*" Online Available : http ://www.win.ua.ac.be/ lore/refactoringProject/publications/-Mens2004MtransTaxoGT.pdf, 2004.

[PM05] J. J. P Muller, F Fleurey. "*Weaving Executability into Object-Oriented Meta-Languages.*" In "Proceedings of MODELS/UML'2005," pages 264–278. S. Kent. Springer, 2005.

[Ren03] A. Rensink. "*The GROOVE Simulator : A Tool for State Space Generation.*" In "Applications of Graph Transformations with Industrial Relevance (AGTIVE), volume 3062 of Lecture Notes in Computer Science," pages 479–485. Springer, 2003.

[SDL+08] Y. Sun, et al. "*Model Transformations Require Formal Semantics.*" In "Workshop on Domain-Specific Program Development (DSPD)," October 2008.

[TC04] P. S. J. W. Tony Clark, Andy Evans. "*An eXecutable Metamodelling Facility for Domain Specific Language Design.*" In "The 4th OOPSLA Workshop on Domain-Specific Modeling," 2004.

[TC08] J. W. Tony Clark, Paul Sammut. *"Applied Metamodeling : A Foundation for Language Driven Developement"*. Ceteva, 2nd edition, 2008.

[Wol09] T. Wolterink. "*Operational Semantics Applied to Model Driven Engineering.*" Master's thesis, Faculty of Electrical Engineering, Mathematics and Computer Science, 2009.

[XB05] O. S. Xavier Blanc. *"MDA en Action"*. Eyrolles, 2005. ISBN 2212115393.

Annexe A

Transformations QVTo

A.1 Transformation Mono-thread

Le code QVTo suivant définit la transformation vers un espace mono-thread :

```
modeltype ECORE uses ecore('http://www.eclipse.org/emf/2002/Ecore
    ');

transformation SemanticMappingTransformation(in syntaxicModel:
    ECORE, out semanticModel:ECORE);

main() {
  syntaxicModel.rootObjects()[EPackage]->map ePackage2EPackage();
  syntaxicModel.objects()[EClass]->map eClass2EReference();
}

mapping EPackage::ePackage2EPackage(): EPackage{
  name:=self.name+'Exe';

  var rObject:EClass:= object EClass{
                    name:="RObject";
```

```
                _abstract:=true;
              };
  eClassifiers+=rObject;

  var methodFrame:EClass:= object EClass{
            name:="MethodFrame";

            eStructuralFeatures+=object EAttribute{
                        name:="methodName";
                        eType := ECORE::EString.oclAsType(
                            EClassifier);
                      };

            eStructuralFeatures+=object EReference{
                        name:="this";
                        eType:=rObject;
                      };

          };

  methodFrame.eStructuralFeatures+=object EReference{
                        name:="previous";
                        eType:=methodFrame;
                      };
  eClassifiers+=methodFrame;

  eClassifiers+=self.eClassifiers[EClass]->map eClass2EClass(
     rObject);
}

mapping EClass::eClass2EClass(superType:EClass): EClass{
  name:=self.name;
  eSuperTypes+=superType;
```

```
  eStructuralFeatures+=self.eStructuralFeatures[EAttribute]->map
      eAttribut2EAttribut();
  self.eOperations.eParameters->map eParameter2EClass(superType);
  self.eOperations.eType->map eType2EClass(superType);
}

mapping EAttribute::eAttribut2EAttribut(): EAttribute{
  name:=self.name;
  eType:=self.eType;
}

mapping EClassifier::eType2EClass(superType:EClass){

  if semanticModel.objects()[EClass]->select(c|c.name=self.name)
     ->isEmpty() then{

    var type:EClass:= object EClass{
              name:=self.name;
              eSuperTypes+=superType;
            };

    var p:EPackage;
    p:=semanticModel.rootObjects()[EPackage]->asOrderedSet()->
       first();
    p.eClassifiers+=type;

    var frame:EClass;
    frame:=semanticModel.objects()[EClass]->select(o|o.name='
       MethodFrame')->asOrderedSet()->first();
    frame.eStructuralFeatures+=object EReference{
                      name:='return';
                      eType:=type;
                    };
```

```
    frame.eStructuralFeatures+=object EReference{
                        name:='returned';
                        eType:=type;
                       };
  }endif
}

mapping EParameter::eParameter2EClass(superType:EClass){

  if semanticModel.objects()[EClass]->select(c|c.name=self.eType.
     name)->isEmpty() then{

    var type:EClass:= object EClass{
                name:=self.eType.name;
                eSuperTypes+=superType;
               };

    var p:EPackage;
    p:=semanticModel.rootObjects()[EPackage]->asOrderedSet()->
       first();
    p.eClassifiers+=type;

    var frame:EClass;
    frame:=semanticModel.objects()[EClass]->select(o|o.name='
       MethodFrame')->asOrderedSet()->first();
    frame.eStructuralFeatures+=object EReference{
                        name:=self.name;
                        eType:=type;
                       };
  }endif
}

mapping EClass::eClass2EReference(){
```

```
  var o:EClass;
  o:=semanticModel.objects()[EClass]->select(c|c.name=self.name)
    ->asOrderedSet()->first();
  o.eStructuralFeatures+=self.eStructuralFeatures[EReference]->
    map eReference2EReference();
}

mapping EReference::eReference2EReference(): EReference{
  name:=self.name;
  eType:=semanticModel.objects()[EClass]->select(c|c.name=self.
    eType.name)->asOrderedSet()->first();
}
```

A.2 Transformation Multi-threads

Le code QVTo suivant définit la transformation vers un espace multi-threads :

```
modeltype ECORE uses ecore('http://www.eclipse.org/emf/2002/Ecore
  ');

transformation SemanticMappingTransformation(in syntaxicModel:
  ECORE, out semanticModel:ECORE);

main() {
  syntaxicModel.rootObjects()[EPackage]->map ePackage2EPackage();
  syntaxicModel.objects()[EClass]->map eClass2EReference();
}

mapping EPackage::ePackage2EPackage(): EPackage{
  name:=self.name+'Exe';

  var rObject:EClass:= object EClass{
                  name:="RObject";
```

```
                _abstract:=true;

                eStructuralFeatures+=object EAttribute{
                        name:="locked";
                        eType := ECORE::EBoolean.oclAsType(
                            EClassifier);
                        };
            };
eClassifiers+=rObject;

var methodFrame:EClass:= object EClass{
          name:="MethodFrame";

          eStructuralFeatures+=object EAttribute{
                        name:="methodName";
                        eType := ECORE::EString.oclAsType(
                            EClassifier);
                        };

          eStructuralFeatures+=object EReference{
                        name:="this";
                        eType:=rObject;
                        };

          eStructuralFeatures+=object EReference{
                        name:="return";
                        eType:=rObject;
                        };

          eStructuralFeatures+=object EReference{
                        name:="returned";
                        eType:=rObject;
                        };
```

```
            eStructuralFeatures+=object EReference{
                        name:="wait";
                        eType:=rObject;
                      };
          };

  methodFrame.eStructuralFeatures+=object EReference{
                        name:="previous";
                        eType:=methodFrame;
                      };

  rObject.eStructuralFeatures+=object EReference{
                        name:="locker";
                        eType:=methodFrame;
                      };
  eClassifiers+=methodFrame;

  eClassifiers+=self.eClassifiers[EClass]->map eClass2EClass(
      rObject);
}

mapping EClass::eClass2EClass(superType:EClass): EClass{
  name:=self.name;
  eSuperTypes+=superType;
  eStructuralFeatures+=self.eStructuralFeatures[EAttribute]->map
      eAttribut2EAttribut();
  self.eOperations.eParameters->map eParameter2EClass(superType);
  self.eOperations.eType->map eType2EClass(superType);
}

mapping EAttribute::eAttribut2EAttribut(): EAttribute{
  name:=self.name;
```

```
  eType:=self.eType;
}

mapping EClassifier::eType2EClass(superType:EClass){

  if semanticModel.objects()[EClass]->select(c|c.name=self.name)
    ->isEmpty() then{

    var type:EClass:= object EClass{
              name:=self.name;
              eSuperTypes+=superType;
            };

    var p:EPackage;
    p:=semanticModel.rootObjects()[EPackage]->asOrderedSet()->
       first();
    p.eClassifiers+=type;

  }endif
}

mapping EParameter::eParameter2EClass(superType:EClass){

  if semanticModel.objects()[EClass]->select(c|c.name=self.eType.
    name)->isEmpty() then{

    var type:EClass:= object EClass{
              name:=self.eType.name;
              eSuperTypes+=superType;
            };

    var p:EPackage;
    p:=semanticModel.rootObjects()[EPackage]->asOrderedSet()->
```

```
        first();
    p.eClassifiers+=type;

    var frame:EClass;
    frame:=semanticModel.objects()[EClass]->select(o|o.name='
        MethodFrame')->asOrderedSet()->first();
    frame.eStructuralFeatures+=object EReference{
                        name:=self.name;
                        eType:=type;
                       };
  }else{

    var cl:EClass;
    cl:=semanticModel.objects()[EClass]->select(c|c.name=self.
        eType.name)->asOrderedSet()->first();

    var frame:EClass;
      frame:=semanticModel.objects()[EClass]->select(o|o.name='
          MethodFrame')->asOrderedSet()->first();

    if frame.eStructuralFeatures[EReference]->select(ref|(ref.
        name=self.name))->isEmpty() then{

      frame.eStructuralFeatures+=object EReference{
                          name:=self.name;
                          eType:=cl;
                         };
    }endif

  }endif
}

mapping EClass::eClass2EReference(){
```

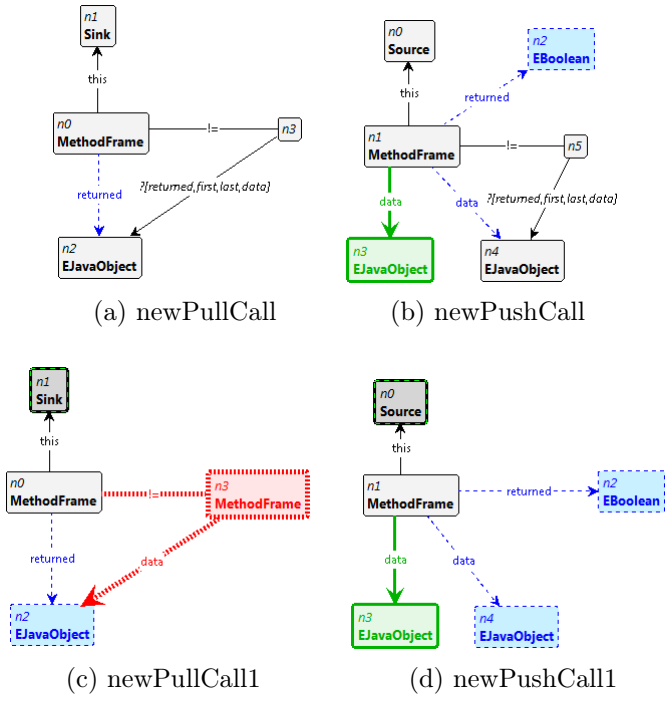
```
  var o:EClass;
  o:=semanticModel.objects()[EClass]->select(c|c.name=self.name)
    ->asOrderedSet()->first();
  o.eStructuralFeatures+=self.eStructuralFeatures[EReference]->
    map eReference2EReference();
}

mapping EReference::eReference2EReference(): EReference{
  name:=self.name;
  eType:=semanticModel.objects()[EClass]->select(c|c.name=self.
    eType.name)->asOrderedSet()->first();
}
```

Annexe B

Règles de transformation

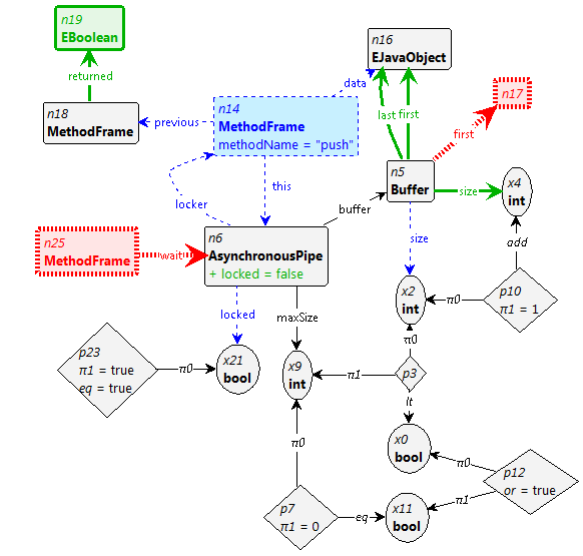
Les figures suivantes présentent les différentes règles de transformation utilisées pour définir la sémantique dynamique du deuxième connecteur : pipe avec accès bloquant et sans copie.



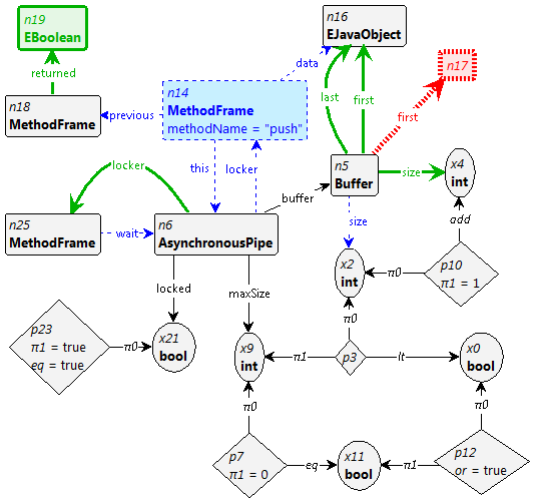
(a) newPullCall (b) newPushCall

(c) newPullCall1 (d) newPushCall1

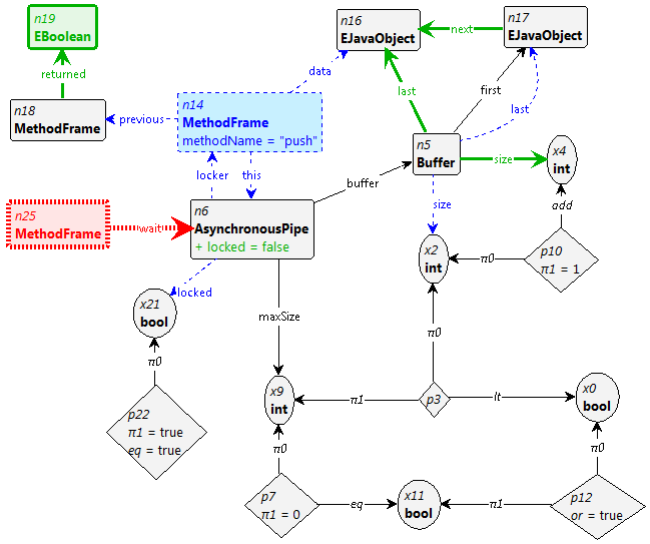
Figure B.1 – Règles d'initialisation



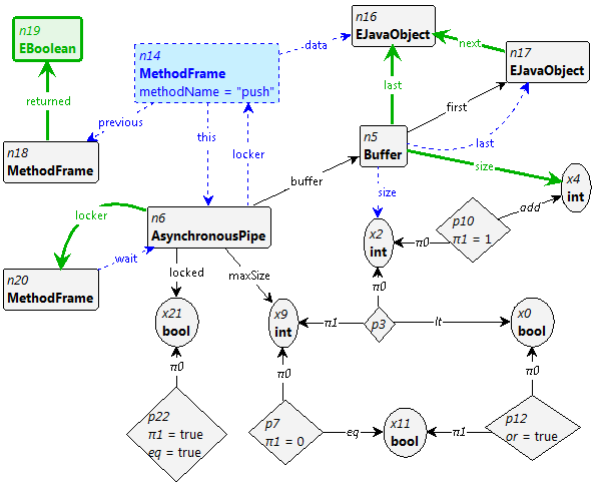
(a) addToEmptyBuffer



(b) addToEmptyBufferAndNotify

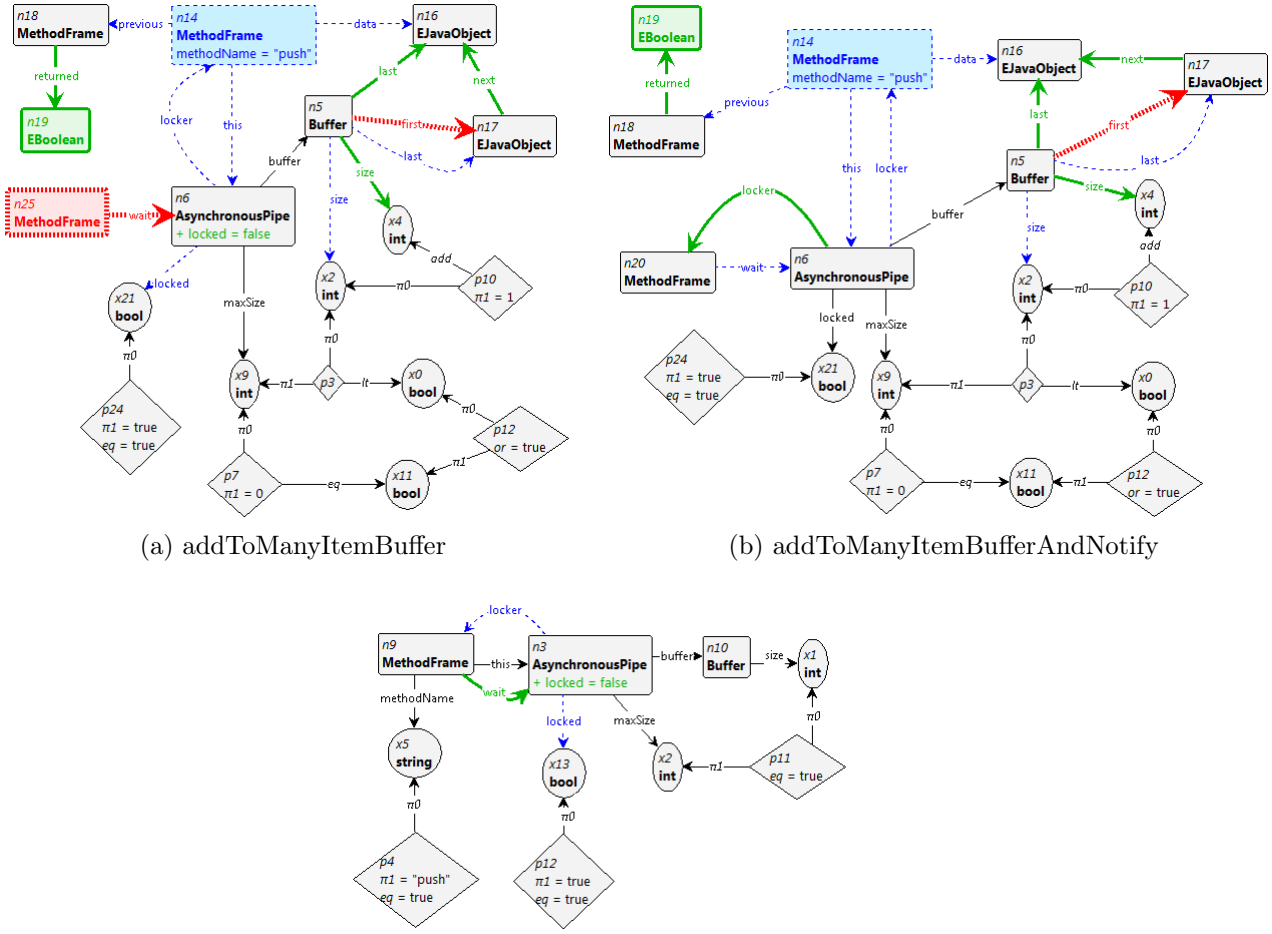


(c) addToOneItemBuffer



(d) addToOneItemBufferAndNotify

FIGURE B.2 – Règles addToBuffer (1)

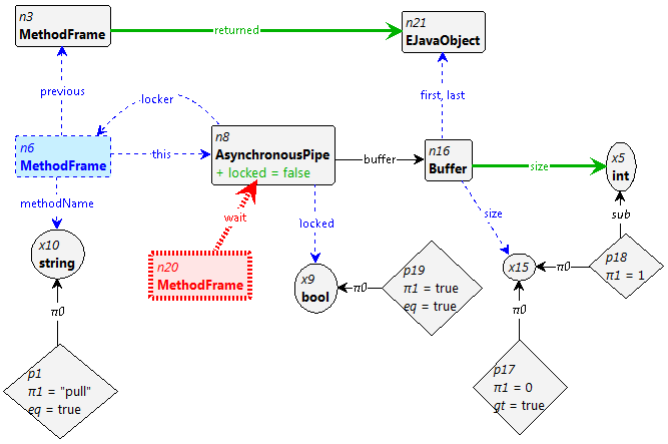


(a) addToManyItemBuffer

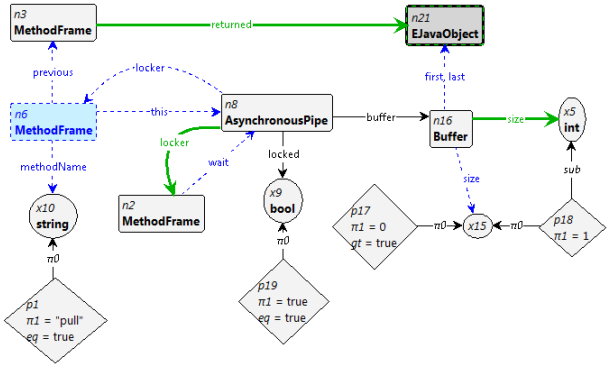
(b) addToManyItemBufferAndNotify

(c) addToFullBuffer

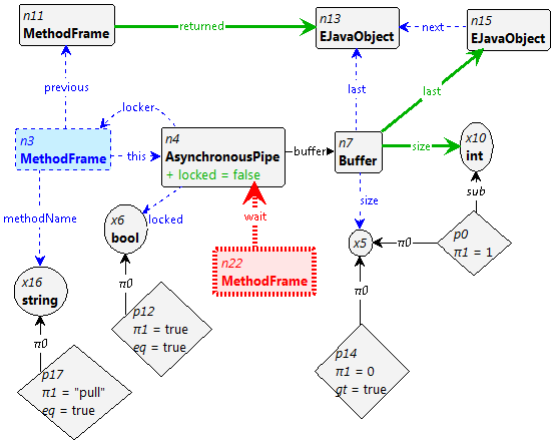
FIGURE B.3 – Règles addToBuffer (2)



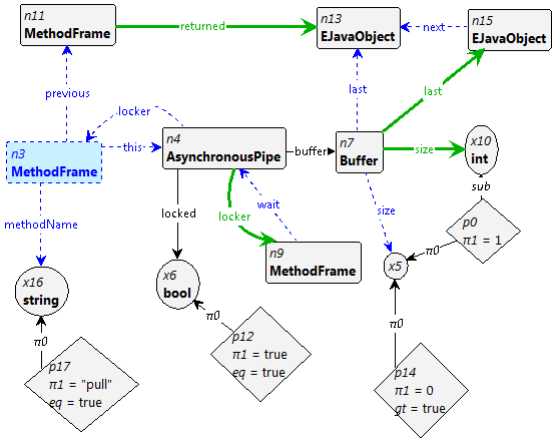
(a) getFromOneItemBuffer



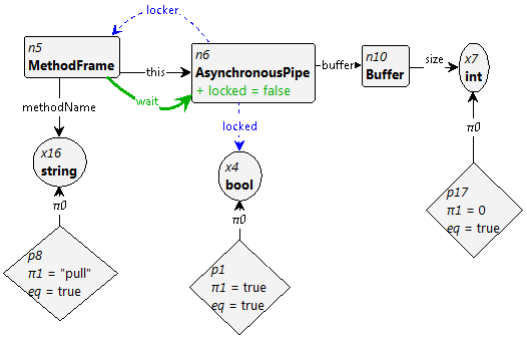
(b) getFromOneItemBufferAndNotify



(c) getFromManyItemBuffer

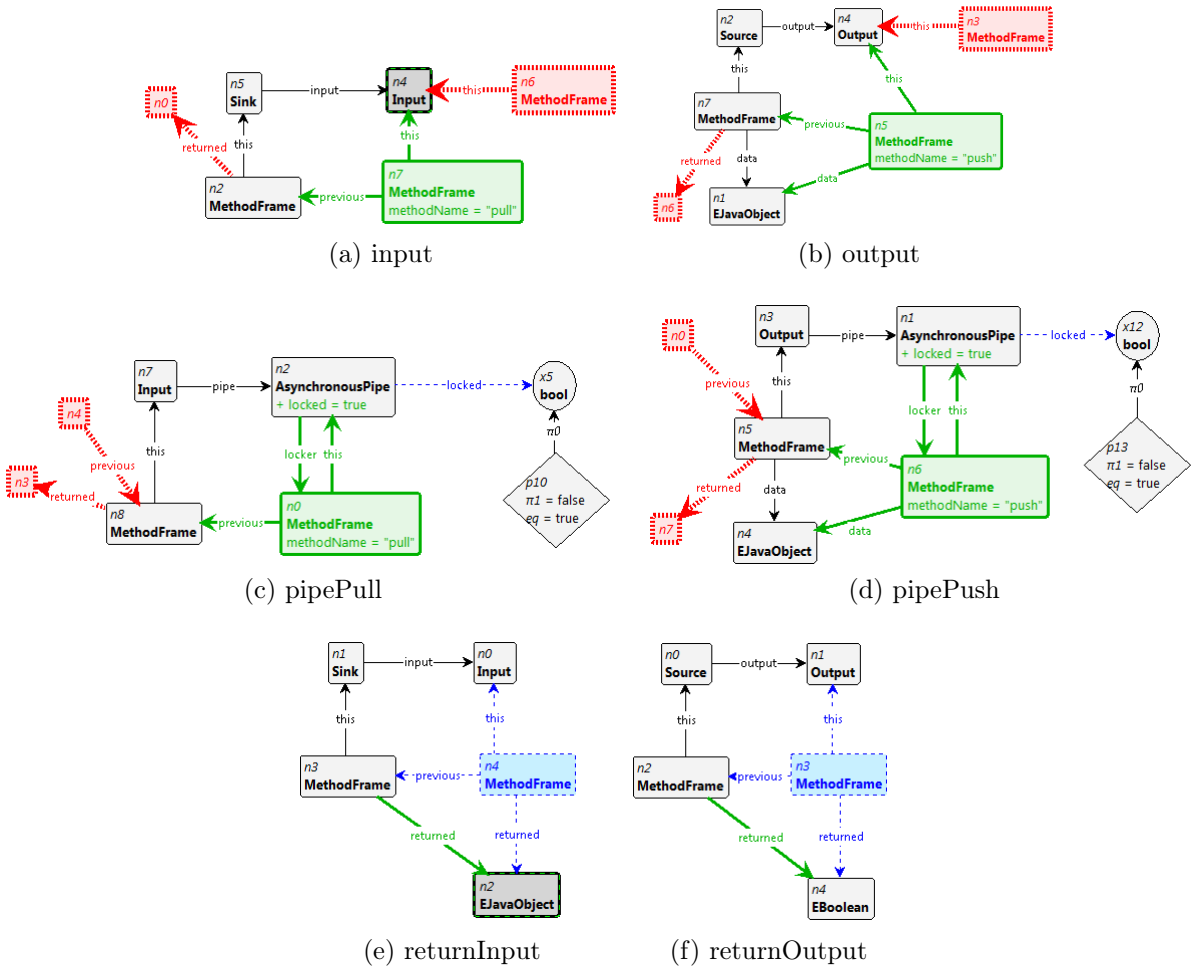


(d) getFromManyItemBufferAndNotify



(e) getFromEmptyBuffer

FIGURE B.4 – Règles getFromBuffer



(a) input (b) output

(c) pipePull (d) pipePush

(e) returnInput (f) returnOutput

FIGURE B.5 – Règles input/output