Spécification d'un Métamodèle pour l'adaptation des outils UML

Amine El Kouhen

Université Lille1
Sciences et Technologies

THÈSE DE DOCTORAT

pour obtenir

le grade de DOCTEUR DE L'UNIVERSITÉ DE LILLE 1
Mention INFORMATIQUE

Présentée par

Amine EL KOUHEN

Équipe d'accueil : DaRT du LIFL et LISE du CEA/LIST
École Doctorale : Sciences Pour l'Ingénieur (SPI) - ED 72
Composante universitaire : IEEA

Titre de la thése :

Spécification d'un Métamodèle pour l'adaptation des outils UML

Soutenue le jeudi 19 décembre 2013 devant la commission d'examen.
Composition du jury :

Président		
Pierre-Alain	MULLER	Professeur à l'Université de Haute-Alsace
Rapporteurs		
Jean-Michel	BRUEL	Professeur à l'Université de Toulouse
Gaëlle	CALVARY	Professeur à l'ENSIMAG, INP Grenoble
Examinateurs		
Pierre	BOULET	Professeur à l'Université de Lille 1 (Directeur de thèse)
Cédric	DUMOULIN	Maître de Conférences à l'Université de Lille 1 (Encadrant)
Sébastien	GÉRARD	Directeur du laboratoire (LISE) du CEA/LIST (Co-encadrant)

Spécification d'un Métamodèle pour l'adaptation des outils UML

Résumé : Dans cette thèse, nous présentons une approche basée sur les modèles pour la conception des éditeurs graphiques de diagrammes. Ceci permet la spécification rapide de ces éditeurs à un niveau d'abstraction élevé, afin de modéliser et de réutiliser, les langages visuels qui y sont manipulés.
Dans un premier temps, nous voulons pouvoir spécifier, à l'aide de modèles, des éditeurs de diagrammes avec une fidélité et une expressivité graphique exemplaire sans besoin d'intervenir manuellement dans cette tâche. En effet, de nombreux environnements de modélisation ont encore besoin d'une quantité considérable de programmation manuelle pour construire leurs éditeurs graphiques de diagrammes. Cette faiblesse devient problématique pour les développeurs, qui doivent fournir une quantité importante de temps et d'autres ressources pour terminer des tâches considérées comme secondaires par rapport à l'objectif principal : le développement et l'intégration du langage de modélisation et la génération de code.
Le second axe est consacré au besoin de réutiliser, d'étendre et de spécialiser cette spécification (sous forme de modèles) dans d'autres contextes utilisations. En effet, la conception de l'outil UML Papyrus fait apparaître un besoin important en termes de réutilisabilité dans la définition des diagrammes. La redondance au niveau de la spécification (actuellement sous forme de modèle), pose une problématique d'incohérence lors de l'évolution ou la modification de cette spécification.
L'objectif est de pouvoir décrire une bibliothèque de syntaxe concrète indépendante de la sémantique, de réutiliser si nécessaire cette description dans différents diagrammes et de pouvoir factoriser les points communs de la syntaxe concrète dans des définitions communes et les variantes dans des définitions spécifiques basées, elles aussi, sur les définitions communes. Ceci nous permettra de définir chaque diagramme indépendamment de l'autre avec la possibilité de réutiliser et de redéfinir les éléments communs.
L'ensemble de ces travaux a été implémenté dans un Framework appelé *MID* (Metamodels for user Interfaces and Diagrams), qui nous a permis d'effectuer une validation sur le langage UML et d'autres langages visuels.

Mots clés : Outils MetaCASE, Métamodélisation basée sur les composants, Réutilisation, Langages visuels, Ingénierie Dirigée par les Modèles (IDM), Syntaxe concrète, Éditeurs graphiques

Specifying a Metamodel for UML tools adaptation

Abstract : Model-Driven Engineering (MDE) encourages the use of graphical modeling tools, which facilitate the development process from modeling to coding. Such tools can be designed using the MDE approach into meta-modeling environments called *metaCASE tools*.
It turned out that current metaCASE tools still require, in most cases, manual programming to build full tool support for the modeling language.
First of all, we want to specify, using models, diagrams editors with a high graphical expressiveness without any need for manual intervention to perform this task.
The second axis is dedicated to this specification reuse in other contexts of use. The design of Papyrus revealed an important need in terms of reusability of diagrams specification. The redundancy of diagrams editors specification raises the problem of inconsistency during the evolution or the update of this specification.
The objective is to describe a library of graphical concrete syntax, reuse this description in different diagrams and create derivative elements (graphical variations) based on the common definitions.
In this context, we propose *MID*, a set of meta-models supporting the easy specification of modeling editors by means of reusable components.

Keywords : MetaCASE tools, Component-based metamodeling, Reusability, Visual languages, Model-Driven Engineering (MDE), Concrete syntax, Graphical editors.

Remerciements

La réalisation d'une thèse est certes un long travail scientifique mais aussi une aventure humaine sans laquelle nos recherches auraient peu de sens. Je vous prie de m'excuser pour les oublis éventuels.

A l'issue de ces trois agréables années au sein du Laboratoire d'Informatique Fondamentale de Lille, j'adresse mes remerciements particuliers à toute l'équipe DaRT pour la qualité d'accueil qui m'a été dispensé tout au long de ma thèse, je tiens aussi à leur exprimer ma sincère reconnaissance pour m'avoir procuré l'environnement propice pour le bon déroulement de mes travaux.

En premier lieu, je tiens à remercier mon directeur de thèse, monsieur Pierre Boulet, pour la confiance qu'il m'a accordée en acceptant d'encadrer ce travail doctoral, pour ses multiples conseils et pour toutes les heures qu'il m'a consacré à diriger cette recherche. J'ai été extrêmement sensible à ses qualités humaines d'écoute et de compréhension tout au long de ma thèse.

Ma profonde gratitude s'adresse à Cédric Dumoulin pour m'avoir encadré, conseillé et soutenu tout au long de cette thèse et pour m'avoir guidé et aidé lors de l'écriture de ce manuscrit.

Merci également à Sébastien Gérard pour m'avoir permis d'intégrer le projet Papyrus et de participer à son développement. Par l'occasion, je tiens à remercier le Commissariat à l'Énergie Atomique pour avoir financé ma thèse.

Je remercie également Jean-Michel Bruel et Gaëlle Calvary pour avoir accepté de rapporter cette thèse, ainsi que pour leurs remarques, questions et recommandations.

Je tiens aussi à remercier Pierre-Alain Muller pour avoir accepté d'examiner cette thèse et de présider son jury.

Je souhaite dédicacer cette thèse à mes parents, mes frères Achraf et Tarik, mon épouse Karima et notre futur bébé que nous attendons avec impatience.

Je tiens à remercier mes amis pour leur soutien au cours de ces trois années de thèse, spécialement Christophe CALVES et Majdi EL HAJJI.

Une partie du matériel présenté dans cette thèse a également été publiée dans les articles suivants :

CONFÉRENCES INTERNATIONALES À COMITÉ DE LECTURE ET ACTES [2]

2013 A Component-Based Approach for Specifying Reusable Visual Languages (Amine El Kouhen, Sébastien Gerard, Cedric Dumoulin, Pierre Boulet), *In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL-HCC'13)*, IEEE, 2013.

2013 A Component-Based Approach for Specifying DSML's Concrete Syntax (Amine El Kouhen, Cedric Dumoulin, Sébastien Gerard, Pierre Boulet), *In Proceedings of the Second Workshop on Graphical Modeling Language Development (GMLD) @ EC-MFA'13*, ACM, 2013.

CONFÉRENCES NATIONALES À COMITÉ DE LECTURE ET ACTES [1]

2013 Specifez vos éditeurs de diagrammes à l'aide de composants réutilisables (Amine El Kouhen, Cedric Dumoulin, Sébastien Gerard, Pierre Boulet), *In Proceedings of 2ème Conférence en Ingénierie Logiciele (CIEL'13)*, 2013.

AUTRES PUBLICATIONS [1]

2012 Evaluation of Modeling Tools Adaptation (Amine El-kouhen, Cédric Dumoulin, Sebastien Gérard, Pierre Boulet), *Technical report, HAL CNRS*, 2012. (http ://hal.archives-ouvertes.fr/hal-00706701)

Table des matières

CHAPITRE 1

Introduction générale

Le monde dans lequel nous vivons, ne cesse d'évoluer et de faire évoluer les données que nous utilisons. Cette évolution influence bien évidemment, les systèmes qui traitent ces données et la manière dont ces systèmes sont conçus. La complexité de ces systèmes a ouvert la voie à ce qu'on appelle l'ingénierie logicielle.

Après l'approche objet des années 80 et un bref passage de l'approche aspect, l'ingénierie logicielle s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles (IDM), au sein de laquelle un système est vu non pas comme une suite de lignes de code mais comme un ensemble de modèles plus abstraits et décrivant chacun une vue (c'est-à-dire une préoccupation) particulière sur le système. L'enjeu de l'IDM est de capitaliser les savoir-faire de conception et de validation d'un système en capturant et en réutilisant les connexions entre ces points de vue, que ce soit de manière horizontale (tissage de liens entre modèles) ou verticale (transformation de modèles). Il s'agit principalement d'augmenter le niveau d'abstraction des développements (au sens plus large que du simple codage) en permettant aux développeurs de se concentrer sur leurs préoccupations à l'aide des langages spécifiques à leurs domaines. Ainsi, les modèles sont devenus des outils puissants pour exprimer la structure, le comportement et d'autres propriétés dans tous les domaines de l'ingénierie et plus précisément dans l'ingénierie logicielle.

L'évolution du paradigme oblige ainsi l'évolution des outils et des langages qui le supportent. On voit alors l'émergence de UML (Unified Modeling Language) [OMG 2011c], qui s'est imposé comme le langage de modélisation le plus utilisé dans l'industrie et dans le milieu universitaire.

Alors que les modèles sont de plus en plus répandus, la définition d'un langage de modélisation et la manipulation explicite de ses modèles sont étroitement liés à certains outils d'ingénierie logiciel assisté par ordinateur ou en abréviation des outils CASE (Computer-Aided Software Engineering). Ces outils ont connu un grand succès dans le génie logiciel, ils manipulent des modèles et proposent le passage au code à partir d'un modèle et inversement (rétro-ingénierie). Cependant, de nombreux environnements de modélisation ont encore besoin d'une quantité considérable de programmation manuelle pour construire les éditeurs graphiques de diagrammes. Les approches utilisées actuellement ne permettent de spécifier des éditeurs de diagrammes que superficiellement, l'ergonomie des éditeurs générés n'est donc pas au niveau des attentes. Un effort de programmation supplémentaire est nécessaire pour implémenter des formes complexes, gérer les différentes interactions, éditer les labels, fournir les vues de propriété, etc. Ceci devient un problème pour les développeurs, qui doivent fournir une quantité importante de temps et d'autres ressources pour terminer des tâches considérées comme

secondaires par rapport à leur objectif principal : le développement et l'intégration du langage de modélisation et sa génération de code.

En plus de l'intervention programmatique pour les spécifier, ce genre d'outils a fait apparaitre plusieurs lacunes [El-kouhen 2011, Amyot 2006, Mohagheghi 2010], principalement en termes de leur expressivité graphique et de la faiblesse de réutilisation de leur spécification. La conception d'un outil tel que Papyrus par exemple, fait apparaitre un besoin important en termes de réutilisation de la définition des diagrammes. Les differents diagrammes comportent plusieurs éléments communs, ces éléments sont spécifiés actuellement par redondance (c'est-à-dire par copie dans tous les diagrammes). Cette redondance au niveau de la spécification, pose une problématique d'incohérence lors de l'évolution ou la modification de cette spécification.

L'objectif est de pouvoir décrire une bibliothèque de syntaxe concrète graphique indépendante de la sémantique, de réutiliser si nécessaire cette description dans différents diagrammes et de pouvoir factoriser les points communs de la syntaxe concrète dans des définitions communes et les variantes dans des définitions spécifiques basées sur les définitions communes. Ceci nous permettra de définir chaque diagramme indépendamment de l'autre avec la possibilité de réutiliser et de redéfinir les éléments en commun.

Dans cette thèse nous voulons proposer un cadre permettant de définir, à l'aide de modèles, la syntaxe concrète graphique d'un langage de modélisation, c'est-à-dire les diagrammes avec toutes leurs préoccupations (les formes, les styles et la composition des éléments de diagramme) et les éditeurs sous-jacent à ces langages (outillage, menus, interactions sur les diagrammes, etc.) et cela en appliquant une démarche d'ingénierie dirigée par les modèles.

La démarche que nous avons suivie tout au long de cette thèse est empirique, elle est basée sur le recensement des besoins constatés lors du développement de l'outil Papyrus et sur l'évaluation de l'existant dans le domaine de la spécification de la syntaxe concrète graphique. Notre contribution essaie de porter des réponses à ces besoins et limitations.

Cette thèse s'inscrit dans une prise en compte plus forte de l'ingénierie des langages visuels dans le contexte de l'IDM. Elle se place dans un champ de recherche encore peu exploré, mais dont l'intérêt ne cesse de grandir. Elle s'inscrit ainsi, dans le cadre du projet Papyrus qui est actuellement l'un des projets majeurs du Laboratoire d'Ingénierie dirigée par les modèles pour les Systèmes Embarqués (LISE) du Commissariat à l'Energie Atomique CEA/LIST [CEA 2013]. Elle s'appuie pour cela sur l'expérience acquise par les différents membres de ce laboratoire sur l'IDM, la définition des langages de modélisation et le développement des outils de support de cette activité. Cette thèse a été dirigée conjointement entre le laboratoire LISE du CEA/LIST et le Laboratoire d'Informatique Fondamentale de Lille (LIFL) qui est aussi un des contributeurs de l'outil Papyrus.

Dans un premier temps, nous présentons dans le chapitre 2 les problématiques de cette thèse et les besoins constatés en termes d'expressivité graphique, de réutilisation de la spécification et d'interactions.

Dans la première partie de cette thèse, nous nous intéressons aux concepts de base de l'IDM et plus précisément à l'étude de l'existant sur la définition de la syntaxe concrète graphique et sa réutilisation.

Dans les chapitres 3 et 4, nous discutons respectivement les fondements de l'IDM qui

représentent le contexte général de notre thèse et les notions clefs des langages visuels qui entrent dans le processus de définition de la syntaxe concrète graphique dans un langage de modélisation. Par la suite, au chapitre 5, nous présentons et comparons les différentes approches et outils existants pour définir une syntaxe concrète graphique et nous concluons par une évaluation motivant le besoin d'une nouvelle proposition pour définir et réutiliser cette syntaxe.

La seconde partie de cette thèse présente notre proposition utilisée ensuite dans le reste de la thèse. Au chapitre 6, nous proposons l'architecture, un aperçu général de la proposition ainsi qu'un début de réponse pour les questionnements soulevés dans le chapitre 2. Le chapitre 7 détaille et explique notre solution et répond à nos problématiques.

La troisième et dernière partie de cette thèse présente la validation de notre proposition. Dans le chapitre 8 nous validons notre approche en spécifiant tous les diagrammes UML et en proposant un prototype de transformation qui permet de générer des éditeurs opérationnels pour ces diagrammes. Le chapitre 9 présente les applications de notre proposition au-delà de la spécification du langage UML. En spécifiant d'autres langages visuels avec une autre technologie d'implémentation, nous avons validé le critère de l'expressivité graphique et l'indépendance à la technologie cible. Le chapitre 10 récapitule la validation en évaluant notre solution selon les objectifs soulevés dans le chapitre 2.

Enfin, nous concluons en faisant un bilan des contributions apportées dans cette thèse, en soulignant leurs faiblesses et en ouvrant les perspectives d'utilisation de notre solution.

CHAPITRE 2

Problématiques

Sommaire

Papyrus est un outil d'édition graphique pour UML développé principalement par le CEA LIST et le LIFL, les deux laboratoires dans lesquels cette thèse a été réalisée. Papyrus est composé de plusieurs éditeurs (actuellement une quinzaine), principalement graphiques, complétés par d'autres types d'éditeurs tels que les éditeurs textuels ou arborés. L'ensemble de ces éditeurs permettent la visualisation simultanée de plusieurs diagrammes où chacun pouvant exprimer un point de vue différent.

Tout comme Papyrus, de nombreux environnements de modélisation ont encore besoin d'une quantité considérable de programmation manuelle pour construire les éditeurs graphiques de diagrammes. Ceci devient un problème pour les développeurs, qui doivent fournir une quantité importante de temps et d'autres ressources pour terminer des tâches considérées comme secondaires par rapport à l'objectif principal : le développement et l'intégration du langage de modélisation et sa génération de code.

Notre problématique de recherche est constituée de deux axes (besoins) principaux. Le premier est de pouvoir spécifier, à l'aide de modèles, des éditeurs de diagrammes avec une fidélité et une expressivité graphique exemplaire et le deuxième besoin est de pouvoir réutiliser, étendre et spécialiser cette spécification (sous forme de modèles) dans d'autres utilisations.

Dans ce chapitre, nous présentons sous forme de questions tous les besoins en termes de ces axes et nous exposons par la suite les questions subsidiaires pertinentes à l'intérieur de chaque axe principal, mais avant cela, nous présentons l'outil de modélisation Papyrus et le contexte de nos travaux de recherche. Par la suite, nous expliquons la notion d'éditeur de diagramme et sa composition.

2.1 Contexte de la thèse

Dans le cadre de ses initiatives pour le MDE (Model-Driven Engineering), l'Object Management Group (OMG) [OMG 1989] - un consortium international qui représente de nombreuses institutions académiques et industrielles - a fourni une série complète de recommandations techniques normalisées à l'appui du développement basé sur les modèles. Parmi ces recommandations nous trouvons la propagation de la méta-modélisation, les transformations de modèles, les langages spécifiques (DSML) ou même les langages d'usage général (General-purpose modeling language). Un élément clé de cette dernière catégorie est UML (Unified Modeling Language), qui s'est imposé comme le langage de modélisation le plus utilisé dans l'industrie et le milieu universitaire [Gérard 2011]. Un certain nombre d'outils de soutien UML sont disponibles à partir de diverses sources. Toutefois, ceux-ci sont généralement des solutions propriétaires restrictives dont les capacités et disponibilités sur le marché sont contrôlées par leurs éditeurs / propriétaires. Cela peut représenter un réel blocage pour les utilisateurs industriels, qui exigent des outils très spécifiques ainsi qu'un support à long terme qui, du point de vue du fournisseur, se prolonge souvent au-delà du point de viabilité commerciale. En conséquence, certains industriels sont à la recherche de solutions open source pour leurs outils UML. La même situation est constatée au niveau de la recherche académique connu par la forte utilisation des outils open-source flexibles pour la mise en place de nouvelles idées et prototypes. Par conséquent, la plateforme Eclipse avec ses outils de développement de model (Projet MDT), est l'environnement de choix pour développer les outils de modélisation open-source.

Pour répondre à ce besoin, un nouvel éditeur graphique, nommé Papyrus, a été proposé par le CEA LIST, le LIFL et Atos Origin et accepté par le projet Eclipse MDT en août 2008. Cet outil d'édition graphique pour UML est basé sur Eclipse et utilise le Framework de modélisation graphique d'Eclipse (GMF).

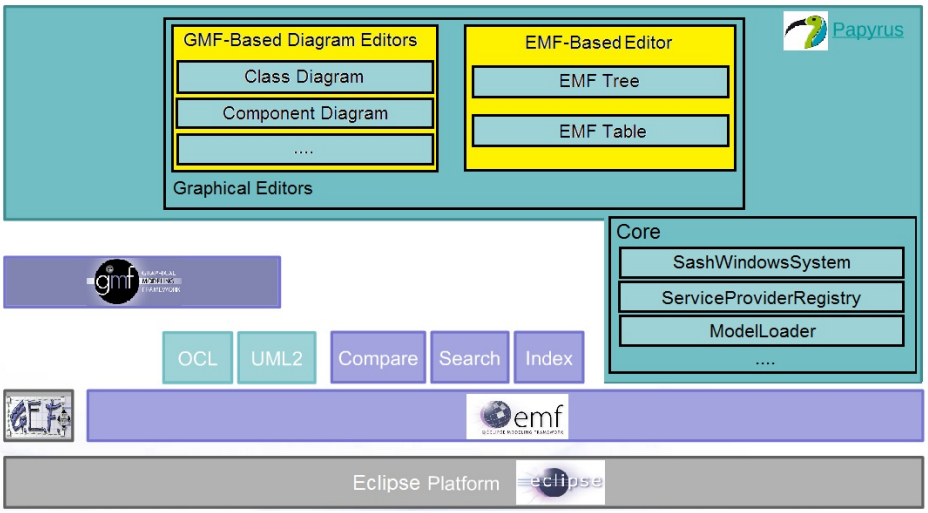


FIGURE 2.1 – Architecture de Papyrus

Papyrus est composé de plusieurs éditeurs (actuellement une quinzaine), principalement graphiques, complétés par d'autres types d'éditeurs tels que les éditeurs textuels ou arborés. L'ensemble de ces éditeurs permettent la visualisation simultanée de plusieurs diagrammes d'un modèle UML donné, chaque diagramme pouvant exprimer un point de vue différent. La modification d'un élément dans l'un des diagrammes se reflète immédiatement dans les autres diagrammes visualisant celui-ci. Papyrus est intégré dans Eclipse comme un éditeur unique lié à un modèle UML. Il offre une vue principale, montrant des diagrammes de modèle et des vues supplémentaires, y compris un mode plan, une vue de propriétés et un explorateur de modèle. Les diagrammes multiples sont gérés par Papyrus plutôt que par Eclipse. Il est hautement personnalisable et permet l'ajout de nouveaux types de diagrammes développés par des technologies compatibles avec Eclipse (GEF, GMF, EMF trees, tables, etc.). La figure suivante illustre l'architecture générale de Papyrus.

Parmi les autres caractéristiques intéressantes pour la modélisation des systèmes en utilisant Papyrus on trouve la création de profils UML. Le profil est créé en sélectionnant les stéréotypes et les métaclasse qui seront étendus par ces stéréotypes. Il est par la suite possible de créer les diagrammes de ces profils avec la réutilisation de la syntaxe concrète d'UML ou en utilisant des images représentant les métaclasses. Papyrus supporte aussi le langage OCL pour définir des contraintes, le même que celui utilisé pour valider le schéma conceptuel généré.

Tout comme Papyrus, de nombreux environnements de modélisation ont encore besoin d'une quantité considérable de programmation manuelle pour construire les éditeurs graphiques de diagrammes. Ceci devient un problème pour les développeurs, qui doivent fournir une quantité importante de temps et d'autres ressources pour terminer des tâches considérées comme secondaires par rapport à l'objectif principal : le développement et l'intégration du langage de modélisation et sa génération de code.

2.2 Spécifier les éditeurs de diagrammes

Nous voulons proposer un cadre permettant de spécifier, à l'aide de modèles, des éditeurs graphiques de diagrammes. Afin de pouvoir modéliser un éditeur de diagrammes, il faut comprendre de quoi est-il composé et quelles sont les problématiques qui se posent lors de la description d'un tel éditeur.

Suite à notre expérience dans le développement de l'outil Papyrus, nous pouvons déduire qu'un éditeur de diagramme est une interface qui permet d'intercepter et de gérer les interactions des utilisateurs pour manipuler un langage visuel (appelé communément "*Diagramme*"). Cette interface (exemple d'éditeur dans la figure 2.2) fournie l'outillage nécessaire pour accomplir cette tâche (palette de création, vue de propriétés) ainsi que des mécanismes intrinsèques comme l'interception des interactions et l'association des éléments graphiques du langage visuel aux éléments d'une syntaxe abstraite (un métamodèle de domaine) qui permettent respectivement de gérer les actions effectuées par les utilisateurs et de donner un sens aux différents éléments graphiques. Dans les sous-sections qui suivent, nous allons parcourir les problématiques soulevées dans

chacune des parties d'un éditeur de diagramme.

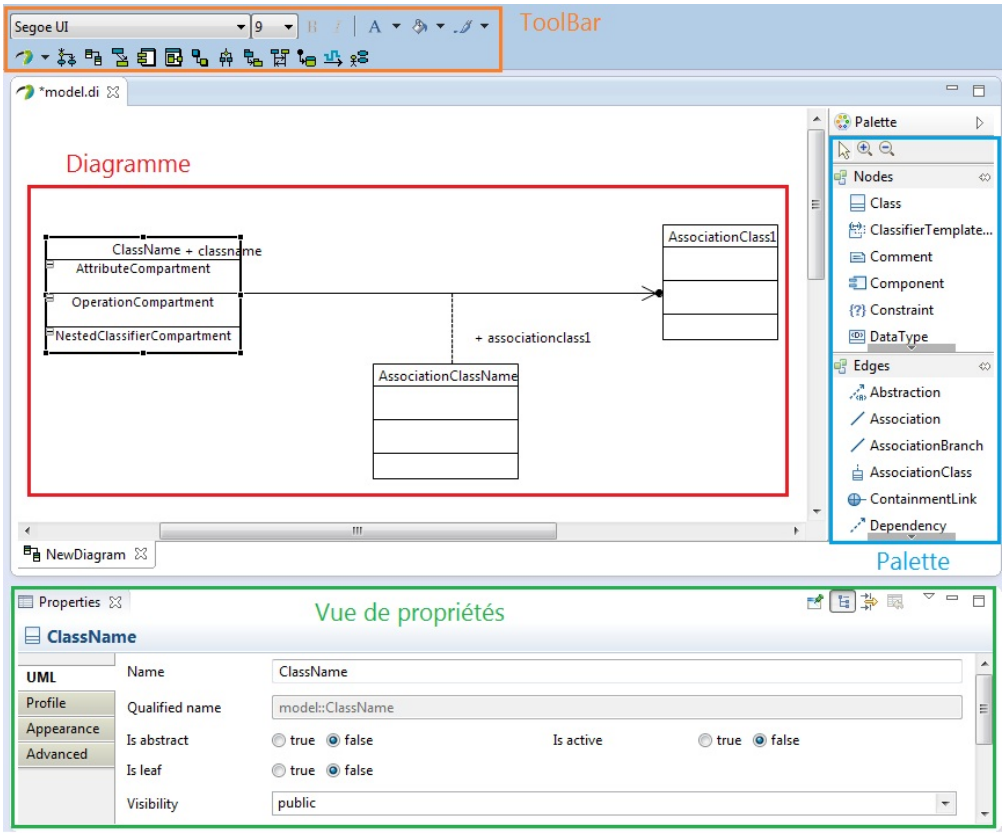


FIGURE 2.2 – Composition d'un éditeur de diagramme

2.2.1 Décrire un diagramme

Décrire un éditeur de diagramme, c'est aussi décrire les éléments graphiques qui y sont manipulés. Proposer un cadre permettant de définir des éditeurs de diagrammes, nécessite une forte connaissance de la nature des langages visuels, leurs compositions, leurs caractéristiques ainsi que les besoins réels des utilisateurs de ces langages. Pour décrire un langage visuel, nous avons été confrontés à deux problématiques majeures. La première est l'expressivité graphique de notre proposition, c'est-à-dire la possibilité de proposer une description plus exhaustive et complète pour définir un langage visuel quelle que soit sa nature. La deuxième problématique concerne la variabilité visuelle des éléments graphiques et comment on peut définir et gérer cette diversité.

2.2.1.1 Expressivité graphique

Dans le cadre de développement des diagrammes UML sous Papyrus, nous avons soulevé des besoins en termes d'expressivité graphique c'est-à-dire la possibilité de décrire aisément des formes graphiques complexes et expressives. Quelques approches utilisées

actuellement, permettent de spécifier des diagrammes à partir de modèles. Cependant, l'ergonomie des éditeurs générés n'est pas au niveau des attentes. Les diagrammes sont pour la plupart faiblement expressives et ne permettent pas de transcrire avec fidélité le langage visuel spécifié. Un effort de programmation supplémentaire est donc nécessaire pour implémenter des formes complexes, éditer les labels, etc. Les questions de bases qui se posent sont :

Question 1. Qu'est ce qu'un diagramme et de quoi est-il composé ?
Question 2. Quelles sont les catégories de diagrammes qui existent ?
Question 3. Comment décrire les diagrammes de type entité-relation ?
Question 4. Comment décrire d'autres catégories de diagramme, en général plus complexes graphiquement ?

Les deux premières questions nous interpellent, car en connaissant la composition d'un diagramme et sa nature, il nous serait possible de décrire et de généraliser les éléments qui font partie de sa description. De plus, ces deux questions exigent de notre proposition une expressivité graphique exemplaire, c'est-à-dire que notre proposition doit pouvoir décrire les langages visuels de type entités-relations comme les autres types de langage, souvent plus complexes graphiquement, d'une manière exhaustive.

La troisième question se concentre sur la nature des diagrammes de type entité-relation (ER) qui sont très répondus dans le domaine de modélisation. Comme leur nom l'indique, ces langages visuels sont constitués principalement de *graphes* composés d'entités (sommets ou noeuds) et de relations (arêtes ou edges) pour les liens entre ces entités. Cette question nous amène à d'autres problèmes subsidiaires, que nous essayons aussi de répondre. Ces problèmes concernent les différents types de liens qui existent, qui peuvent être purement graphiques c'est-à-dire qu'ils ne représentent pas d'éléments dans le modèle de domaine (par exemple, les liens de commentaire dans les diagrammes UML) ou peuvent représenter des éléments dans la syntaxe abstraite. Dans les deux cas, plusieurs besoins se montrent, notamment pour les liens n-aires (plus que deux extrémités pour un lien), les liens qui représentent une métaclasse dans la syntaxe abstraite (classes d'association), les liens qui représentent des références unitaires (unidirectionnelles) ou binaires (dans les deux sens) entre deux métaclasses, les liens qui contiennent des entités (par exemple, les bus), etc.

Les entités ou les noeuds dans les langages ER, soulèvent aussi des questions sur la manière de les décrire ainsi que les attributs graphiques qui leur sont associés (la forme, la couleur, la taille, etc.). La relation entre ces éléments graphiques et les éléments de domaine (syntaxe abstraite) qu'ils représentent sera discutée dans la partie 2.2.3.

La quatrième question s'intéresse à la spécification d'autres catégories de diagrammes, généralement plus complexes graphiquement que les diagrammes de type ER. Ces langages visuels, comportent les caractéristiques de plusieurs catégories de langages visuels et sont donc plus riches en termes de contenu graphique. Le Langage UML fait partie de cette catégorie de langage, puisqu'il propose de construire des diagrammes de type entité-relation, en plus de la possibilité de faire de la composition (relation composant-composé) et d'autres relations spatiales comme l'adjacence (dans le cas du diagramme

de séquence). Une réponse plus détaillée de cette problématique, nous permettra de répondre à nos besoins en termes de spécification des diagrammes UML dans l'outil Papyrus (décrire une classe, décrire un élément contenant un autre, décrire une zone de texte, décrire un élément glissant dans la bordure d'un autre élément, définir des diagrammes complexes comme le diagramme de séquence, etc.).

Par ailleurs, notre solution doit être exhaustive et ne pas se limiter à décrire les diagrammes UML. Elle doit pouvoir définir d'autres éléments graphiques, par exemples, les liens complexes (sous forme de bus), les formes imbriquées/composés, les schémas électriques complexes ou la possibilité de définir avec précision les points d'attaches des liens sur les noeuds, etc.

2.2.1.2 Variabilité visuelle

Un autre problème constaté lors du développement de l'outil Papyrus, concerne la variation de la représentation graphique de quelques éléments UML. Par exemple une classe dans le diagramme de classe n'a pas la même représentation graphique que celle dans le diagramme de structure composite. Cette situation est constatée aussi pour l'interface dans le diagramme de composant et dans les autres diagrammes ou bien l'élément Acteur dans le diagramme de cas d'utilisation et les autres diagrammes. Ces éléments ont la même sémantique dans le modèle UML, parfois ils ont la même structure, mais ils sont représentés différemment.

Question 5. Comment décrire la variabilité visuelle d'un élément de diagramme ?

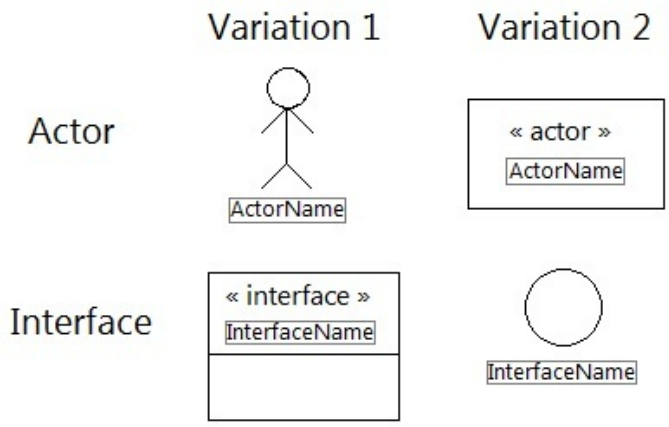


FIGURE 2.3 – Variation des notations visuelles

Par exemple, la figure 2.3 montre quelques variations graphiques dans UML. Dans le diagramme de cas d'utilisation un *acteur* peut être représenté par un homme de bâtons (stick man) avec une étiquette de nom (label) ou bien par un rectangle stéréotypé avec l'étiquette de nom. Ces deux éléments graphiques ont la même signification (sémantique), mais deux représentations différentes. L'autre exemple est celui de l'élé-

ment *Interface* qui est représenté en tant que rectangle (semblable à la classe) dans le diagramme de classes et en tant que cercle (lollipop) dans le diagramme de composant.

Une autre forme de variabilité visuelle que nous devons prendre en compte est celle des attributs graphiques (la forme, la couleur, etc.) d'un élément de diagramme, qui changent selon les valeurs du modèle de domaine. Selon la valeur d'un attribut du modèle de domaine, nous pouvons imaginer que les éléments graphiques changent d'aspect ou de taille, etc. Par exemple, La police du nom d'une classe qui s'affiche en italique si la classe est abstraite.

2.2.2 Décrire les interactions

Un éditeur permet a l'utilisateur d'interagir avec le diagramme en lui offrant les outils nécessaires à la manipulation des concepts du diagramme. Nous nous concentreront ainsi sur les interactions des utilisateurs qui influencent l'aspect graphique du diagramme ainsi que le modèle de domaine associé. La question qui se pose donc est :

Question 6. Quelles sont les *interactions* gérées par un éditeur et comment les définit-on ?

Nous avons analysé les éditeurs graphiques de Papyrus pour en extraire les composants de base et les interactions prises en charge par cet outil. Ces interactions, qui sont souvent associées à des éléments graphiques, nécessitent un grand effort d'analyse pour les recenser et les généraliser dans une description. Nous procédons d'une manière incrémentale dans ce recensement, en analysant les besoins dans l'outil Papyrus. Nous essayons aussi de définir qu'est ce qu'une interaction et quelles sont ses différentes catégories.

2.2.3 Décrire les liens entre le domaine et la syntaxe concrète graphique

Dans notre contexte, un langage de modélisation graphique est défini avec une association de la syntaxe abstraite (métamodèle de domaine) vers sa (ses) représentation(s) concrète(s) graphique(s). Pour spécifier des langages de modélisation, tels que UML, nous avons ainsi besoin de définir les liens entre ces deux syntaxes (abstraite et concrète). Plusieurs questions secondaires se posent :

Question 7. Comment fait-on le *lien* entre les éléments graphiques d'un diagramme et la syntaxe abstraite du domaine ?
Question 8. Le lien entre un élément graphique et un élément du domaine est-il souvent de cardinalité 1 :1 ?
Question 9. Quelles sont les autres possibilités d'association entre les éléments graphiques d'un diagramme et les éléments du domaine ?

En analysant les liens existants entre les éléments graphiques et les concepts du langage UML qui y sont associés (éléments du domaine), nous avons été confrontés à des besoins en termes d'association au-delà de simples liens unitaires (1 :1). Ces besoins nous motivent à construire notre proposition d'une manière à permettre la création

d'associations sémantiques plus complexes (associations multiples, associations calculées, etc.). Par exemple, pour l'élément *InstanceSpecification* de UML, nous avons besoin du nom de l'instance ainsi que le nom de sa classe. Dans ce cas la valeur est calculée à partir de deux attributs de domaine et donc c'est une association de cardinalité (1 :N). Un autre besoin que nous avons relevé précédemment est celui des relations (liens) qui peuvent représenter un concept (une métaclasse) ou une propriété d'une métaclasse. Ces éléments nous poussent à proposer une solution qui permet de généraliser cette problématique et de fournir un mécanisme plus complet pour l'association de l'aspect graphique aux éléments du domaine.

2.2.4 Décrire l'outillage de l'éditeur

En plus d'un diagramme et ses interactions, un éditeur de diagrammes définit aussi les outils offerts aux utilisateurs pour accomplir leurs tâches de modélisation. L'outil le plus important est l'outil de création (palette), qui permet d'instancier les éléments de diagrammes, qu'ils soient représentés par des noeuds ou des connections. La vue de propriétés (PropertyView) est aussi parmi les outils disponibles dans un éditeur graphique. Elle permet de visualiser et/ou éditer les propriétés d'un élément graphique sélectionné. D'autres éléments peuvent être proposés dans un éditeur de diagramme comme les barres d'outils ou des vues spécialisées, etc. Les questions qui se posent sont :

Question 10. Comment décrit-on la palette de création dans un éditeur de diagramme ? Pourait-on la déduire à partir des éléments de diagrammes ?
Question 11. Comment peut-on éditer les valeurs du modèle de domaine à partir de l'éditeur graphique de diagrammes ?

2.3 Réutiliser, étendre et spécialiser les éléments d'un diagramme

En spécifiant les diagrammes de papyrus avec les outils existants (GEF, GMF, etc), nous avons constaté d'autres problèmes à différents niveaux. Le premier problème commun constaté est celui des éléments redondants dans tous les diagrammes UML, comme les éléments de commentaires (comment) ou les contraintes (constraints) qui sont présents dans tous les diagrammes Papyrus.

Cette redondance au niveau de la spécification (actuellement sous forme de modèle), pose une problématique d'incohérence lors de l'évolution ou la modification de cette spécification. Par exemple, pour modifier un paramètre visuel dans l'élément « commentaire », qui est présent dans tous les diagrammes, il faut effectuer les modifications manuellement sur cet élément dans chacun des diagrammes. Ce qui représente une charge de travail considérable et sujette à des erreurs de manipulation. La redondance au niveau de la spécification engendre une génération redondante du même élément dans chaque diagramme. La question qui se pose donc est :

Question 12. Comment *Réutiliser* un élément graphique dans plusieurs diagrammes ?

Le deuxième problème commun concerne quelques diagrammes spécifiques comme le diagramme de package qui est composé d'un sous ensemble d'éléments du diagramme de classes (package, import, merge, etc.). Un autre exemple est celui des langages qui réutilisent les diagrammes d'autres langages (par exemple, UML et SysML figure 2.4) ou bien les langages qui sont très similaires graphiquement (par exemple, le diagramme de Workflow de BPMN et le diagramme d'activité d'UML). Ce point nous a interpellé par la question suivante :

Question 13. Comment réutiliser, étendre et spécialiser tout ou une partie d'un diagramme ?

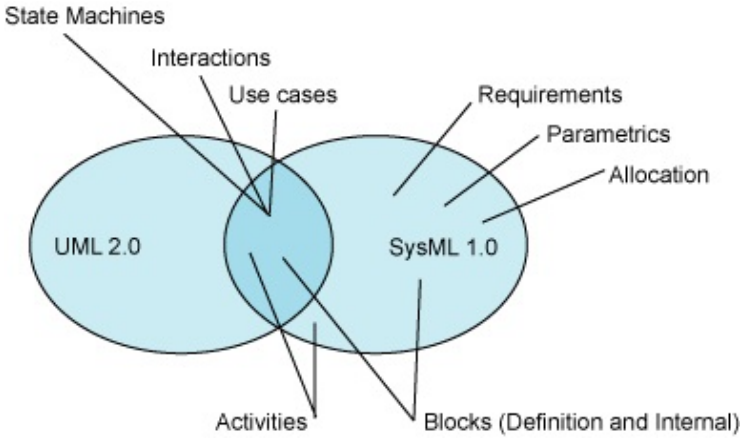


FIGURE 2.4 – Réutilisation des diagrammes UML 2.0 dans SysML 1.0

La conception de Papyrus fait donc apparaitre un besoin important en termes de réutilisabilité dans la définition des diagrammes. L'objectif est de pouvoir décrire une bibliothèque de syntaxe concrète indépendante de la sémantique, de réutiliser si nécessaire cette description dans différents diagrammes et de pouvoir factoriser les points communs de la syntaxe concrète dans des définitions communes et les variantes dans des définitions spécifiques basées elles aussi sur les définitions communes. Ceci nous permettra de définir chaque diagramme indépendamment de l'autre avec la possibilité de réutiliser et redéfinir les éléments en commun.

Première partie

IDM et Langages visuels : État de l'Art

CHAPITRE 3

Ingénierie Dirigée par Modèles

Sommaire

Après l'approche objet des années 80, l'ingénierie logicielle s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles (IDM). Selon [Bézivin 2004, Bézivin 2005], cette approche peut être considérée à la fois comme continuité et point de rupture avec les précédents travaux. En continuité, car les technologies objet ont été la base de l'évolution vers les modèles. Le point de rupture de l'IDM par rapport aux précédents travaux était de classifier les objets en fonction de leurs différentes origines (objet fonctionnel, technologique, etc.). L'IDM vise donc, d'une manière plus poussée que les patterns ou les aspects, à fournir un cadre qui permet d'exprimer séparément chacune des préoccupations des utilisateurs, des concepteurs, des architectes, etc. Le principe de séparation des préoccupations, est considéré comme le point de rupture par rapport aux travaux précédents. L'IDM offre un cadre méthodologique et technologique permettant d'unifier et de favoriser en un processus homogène l'étude des différents aspects du système. L'IDM est une forme d'ingénierie générative qui aboutit au code du système décrit à haut niveau.

L'enjeu de l'IDM est de capitaliser les savoir-faire de conception et de validation d'un système en capturant et en réutilisant les connexions entre ces points de vue, que ce soit de manière horizontale, par tissage de liens entre modèles, ou verticale, par transformation de modèles. Visant à automatiser une partie du processus de développement, l'IDM requiert un effort d'abstraction plus important de la part des développeurs. En contrepartie, l'IDM promet de conserver le savoir-faire de conception proche des centres de décision grâce aux économies d'échelle dues à l'automatisation [Jezequel 2012].

Dans ce chapitre, nous allons présenter les concepts généraux de l'IDM, la différence entre l'IDM et la MDA, nous présentons les langages dédiés de modélisation et leur composition et nous présentons un bref aperçu des transformations de modèles.

3.1 Concepts généraux de l'IDM

L'ingénierie dirigée par les modèles (IDM) a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique. Il s'agit d'une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles.

Ponctuées d'acronymes (MDA, PIM, MOF, CIM, SPEM, XMI, UML, DSM) mais aussi de standards aussi variés que complexes à maîtriser, le ticket d'entrée dans la jungle de l'IDM est très élevé [Favre 2006]. On constate souvent une ambiguïté dans la relation entre ces termes et le monde de l'IDM. Pour faire simple, l'IDM peut être vu comme une famille d'approches qui se développent à la fois dans les laboratoires de recherche et chez les industriels impliqués dans les grands projets de développement logiciel [Estublier 2006]. Cette approche vise non seulement à favoriser un *génie* logiciel plus proche des métiers en autorisant une appréhension des applications selon différents points de vues (modèles) exprimés séparément, mais elle intègre également comme étape fondamentale la composition et la mise en cohérence de ces perspectives. De plus, elle se veut productive en automatisant la prise en charge des outils relatifs à la validation des modèles, les transformations et les générations de code. Malgré ses balbutiements initiaux, l'IDM cible une production logicielle bien fondée [Estublier 2006].

La modélisation est vue comme un moyen d'exprimer une solution à un haut niveau d'abstraction. En ingénierie logicielle on souhaite en général décomposer un système complexe en autant de modèles que nécessaire pour aborder efficacement toutes les préoccupations pertinentes [Baniassad 2004]. L'expression explicite de solution en gérant les préoccupations, permet d'établir des compromis adéquats le plus tôt possible dans le cycle de vie du logiciel et de gérer efficacement les variantes du système. De nos jours, il est de plus en plus rare de construire des systèmes informatiques, qui ne soient pas déclinés en de multiples variantes. En effet, il faut avouer/reconnaitre que l'IDM est une meilleure solution pour gérer cette variabilité connue sous le nom de ligne de produit. Il existe dans l'industrie d'importants savoir-faire, souvent capturés sous la forme de patrons de conception. Prendre en compte plusieurs aspects à la fois s'avère un peu plus complexe, mais les ingénieurs y arrivent la plupart du temps. Le vrai défi est plutôt lié, dans un contexte d'agilité des développements, aux besoins fréquents de s'adapter au changement des exigences. Ce changement se traduit par un nouveau choix de variante par aspect du système et nécessite d'être recomposé pour obtenir de nouvelles déclinaisons / configurations du système, c'est-à-dire un nouveau produit dans la ligne de produits. Bien évidemment, cette nouvelle déclinaison du système doit être obtenue rapidement, de manière fiable et bon marché [Combemale 2008]. L'IDM ne propose pas de résoudre directement ce problème, mais simplement de mécaniser et de reproduire le processus que les concepteurs expérimentés suivent à la main [Ho 2002]. La solution est donc de rejouer la plus grosse partie de ce processus de conception (en apportant quelques petites modifications), lorsqu'une nouvelle variante a besoin d'être dérivée dans une ligne de produits [Lahire 2007].

L'IDM est basée sur un jeu de concept et de relations. Le concept de base de l'IDM est

la notion de modèle, pour laquelle il existe plusieurs définitions. Selon [Jezequel 2012], un **modèle** est un ensemble de faits caractérisant un aspect d'un système dans un objectif donné. Un modèle représente donc un système selon un certain point de vue, à un niveau d'abstraction facilitant la compréhension et la validation de cet aspect particulier du système. Un **modèle** doit par définition être une abstraction pertinente du système qu'il modélise pour un point de vue particulier. Il doit être suffisant pour répondre aux questions sous-jacentes à ce point de vue particulier, en lieu et place du système qu'il représente et exactement de la même façon que le système aurait répondu lui-même. Ce principe assure que le modèle peut se substituer au système pour permettre d'analyser de manière plus abstraite certaines de ces propriétés [Minsky 1968, Favre 2006]. La notion de modèle dans l'IDM fait explicitement référence à la définition des langages utilisés pour les construire. La définition d'un langage de modélisation prend la forme d'un modèle, appelé métamodèle.

Un **métamodèle** est un modèle qui définit le langage d'expression d'un modèle [OMG 2006b], c'est-à-dire le langage de modélisation. Il permet de capitaliser un domaine de connaissance. Il devient naturellement le coeur d'un outillage visant à systématiser certaines étapes de développement. De nombreux métamodèles ont émergés afin d'apporter chacun leurs spécificités dans un domaine particulier. Devant le danger de voir émerger indépendamment et de manière incompatible cette grande variété de métamodèles, il y avait un besoin urgent de donner un cadre général pour leur description. La réponse logique fut donc d'offrir un langage de définition de métamodèle qui prit lui-même la forme d'un modèle : ce fut le métamétamodèle. En tant que modèle, le métamétamodèle doit être défini à partir d'un langage de modélisation. Pour limiter le nombre de niveau d'abstraction, le métamétamodèle doit alors avoir la propriété de métacircularité, c'est-à-dire la capacité de se décrire lui-même.

Un **métamétamodèle** est un modèle qui décrit un langage de métamodélisation, c'est-à-dire les éléments de modélisation nécessaires à la définition des langages de modélisation. Il a de plus la capacité de se décrire lui-même.

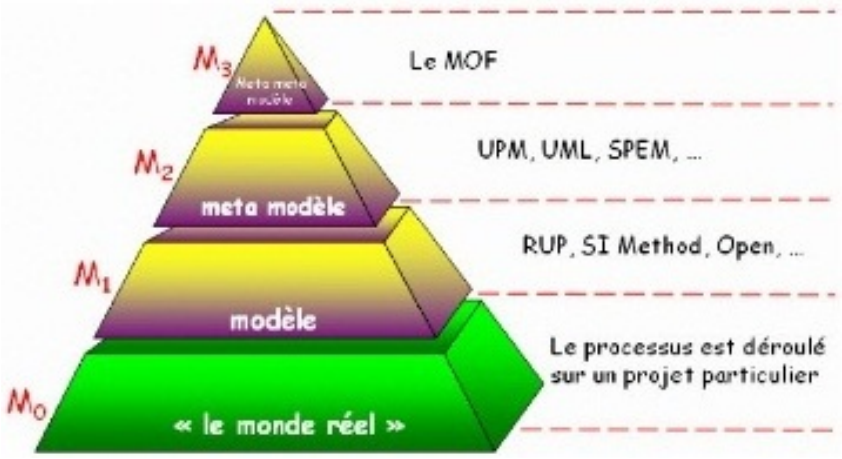


FIGURE 3.1 – Niveaux d'abstraction de l'IDM

L'IDM se compose donc de trois niveaux d'abstraction, à savoir :

1. Niveau M1 ou couche modèle où on peut définir des modèles en utilisant les langages de modélisation ;
2. Niveau M2 ou couche métamodèle où les métamodèles sont définis en utilisant les langages de métamodélisation (ex, MOF) ;
3. Niveau M3 ou couche métamétamodèle où seul les langages de métamodélisation sont auto- définis/décris.

L'OMG a défini le MDA (Model Driven Architecture) en 2000 [Soley 2000] pour promulguer de bonnes pratiques de modélisation et exploiter pleinement les avantages des modèles. Le MDA inclut pour cela la définition de plusieurs standards, notamment UML, MOF et XMI. Le principe clé du MDA consiste à s'appuyer sur le standard UML pour décrire séparément des modèles de différentes préoccupations (variabilité horizontale) et de différentes phases du cycle de développement d'une application (variabilité verticale). Le MDA préconise l'élaboration de modèles :

- D'exigences (CIM - Computation Independent Model) dans lequel on doit représenter l'application dans son environnement afin de définir quels sont les services offerts par l'application et quelles sont les autres entités avec lesquelles elle interagit ;
- D'analyse et de conception (PIM - Plateform Independent Model) ;
- De Code (PSM - Platform Specific Model).

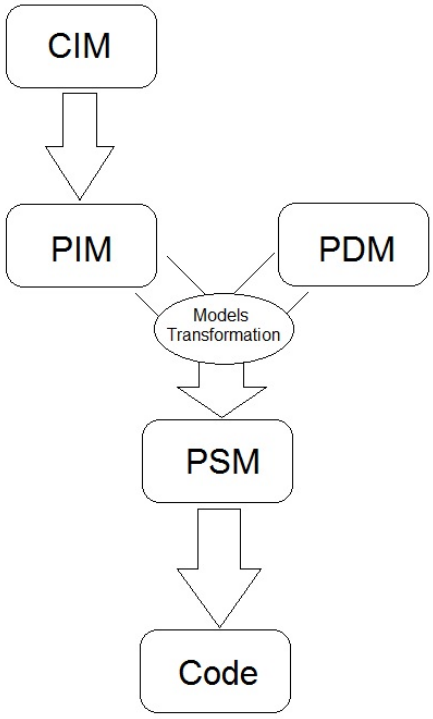


FIGURE 3.2 – Processus MDA en "Y"

L'objectif du MDA est l'élaboration de modèle pérenne (PIM) indépendants des détails techniques (Plateforme d'exécution, préférences d'utilisation...), afin de permettre la génération automatique de la totalité des modèles de code PSM et d'obtenir un gain significatif de productivité. Le passage de PIM à PSM fait intervenir des mécanismes de composition et transformation de modèle avec un modèle de description de plateforme (PDM - Platform Description Model). Cette démarche s'organise selon un cycle de développement en « Y » propre au développement dirigé par les modèles.

C'est sur ces principes de base que s'appuie l'OMG pour définir l'ensemble de ses standards, en particulier UML (Unified Modeling Language) dont le succès industriel est largement reconnu.

3.2 Langages dédiés de modélisation

Selon Wikipedia [Wikipedia 2013h], un langage de modélisation est un langage artificiel qui peut être utilisé pour exprimer de l'information ou de la connaissance dans une structure qui est définie par un ensemble cohérent de règles (Métamodèle). Un langage de modélisation peut être graphique ou textuel. Les langages de modélisation graphiques utilisent les diagrammes pour représenter les concepts. Les langages de modélisation textuels utilisent typiquement des mots-clés spécifiques pour rendre les expressions interprétables. Un grand nombre de langages de modélisation apparaissent dans la littérature. Ils sont classifiés selon différents critères : exécutables, non-exécutables, généralistes ou spécifiques, etc. La classification qui nous intéresse dans notre étude est celle du champ d'application (Scope) du langage de modélisation : c'est-à-dire les langages généralistes (GPML) et les langages spécifiques à un domaine particulier (DSML).

3.2.1 GPML (General-Purpose Modeling Language)

C'est une catégorie de langages de modélisation dont le champ d'application est générique. Ce type de langages est utilisé dans divers domaines pour représenter les différentes facettes d'un objet ou d'un système. Il existe une multitude de langages qui sont classifiés comme GPML. Par exemple (non-exhaustif), on trouve le langage EXPRESS [ISO 2004] qui est un standard ISO 10303-11 pour spécifier formellement les données. Ce langage permet donc de définir une représentation non ambiguë des données, interprétable par un système informatique ce qui permet de créer directement et automatiquement un grand nombre d'éléments à partir d'un modèle Express. C'est un langage de modélisation ayant une approche objet comme UML. Contrairement à UML, son but est seulement de spécifier une base de données et non de modéliser un système. Un modèle EXPRESS peut être écrit sous forme graphique ou sous forme textuelle [Wikipedia 2013e]. D'autres méthodes sont étiquetées comme langages GPML, on peut citer comme exemple, la méthode Merise (France), SSADM (UK), Information Engineering (US) ou MACAO qui est une méthode complète du génie logiciel qui s'appuie sur la notation et les modèles UML. Elle se base sur UP (Unified Process) et se place ainsi au même niveau que le RUP (Rational Unified Process) proposé par IBM [Combemale 2008].

Un élément clé de cette catégorie de langage est UML (Unified Modeling Language), qui s'est imposé comme le langage de modélisation de référence, le plus utilisé dans l'industrie et le milieu universitaire.

Divers langages de modélisation existent dans la littérature. Actuellement, avec les facilités proposées par les outils de métamodélisation, il est aisé de définir et construire son propre métamodèle et par conséquent, son propre langage de métamodélisation. Cependant, la confusion des langages et des formalismes, même au niveau d'un même domaine, peut rendre difficile la compréhension entre les différents acteurs du développement. Dans un souci d'uniformisation des langages de modélisation, UML a été proposé comme standard par l'OMG.

L'Unified Modeling Langauge (UML) est un langage de modélisation graphique standardisé par l'Object Management Group (OMG). Il est défini par un métamodèle auquel lui est associé une représentation graphique. UML combine les techniques de modélisation de données (diagrammes entité-relation), modélisation de processus métier (Workflows), modélisation objet et la modélisation par composant. Il peut être utilisé dans tous les processus du cycle de vie du développement logiciel et à travers différentes technologies en implémentation [Mishra 1997]. Historiquement, UML avait synthétisé les notations de la méthode Booch, la technique de modélisation Objet (OMT) et le langage OOSE (Object-Oriented Software Engineering) en les fusionnant en un seul langage de modélisation.

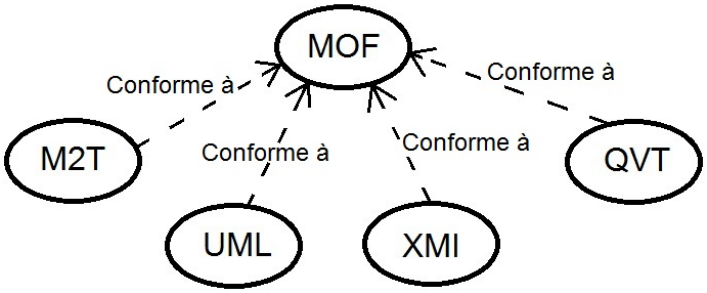


FIGURE 3.3 – Conformité des Standards OMG à MOF

Basé sur UML, l'OMG a conçu une architecture de métamodélisation qui s'appelle Meta-Object Facility (MOF) [OMG 2006a]. Ce langage de méta-métamodélisation (c'est-à-dire qu'il permet de créer de définir des langages/métamodèles) est utilisé actuellement comme standard proéminent pour la définition des langages de modélisation que ce soit d'usage générique (GPML) ou spécifique (DSML). Il permet par sa compatibilité avec différents autres standards (dont il est l'élément de base) de bénéficier du format de persistance et d'échange de modèles (MOF-XMI), de la possibilité de transformation de modèles vers modèles (MOF-QVT), des transformations de modèles vers texte (MOF-M2T) et même d'intégrer les langages de contrainte comme OCL.

Comme tout GPML, UML permet une modélisation uniforme quel que soit le système modélisé. Cependant, sa sémantique trop faible est un véritable inconvénient. Ainsi, la notion de profil lui a été ajoutée pour enrichir la sémantique des concepts de base proposés par UML et les étendre par l'intermédiaire de stéréotypes. En utilisant les

stéréotypes, il est possible d'ajouter des attributs et relations aux éléments qu'il raffine grâce à l'utilisation de références et de tagged values. Le package de Profil d'UML contient des mécanismes qui permettent d'étendre les métaclasses du métamodèle existant pour les adapter aux différents usages. Ceci inclut la capacité à adapter le métamodèle UML aux différentes plates-formes (J2EE ou .Net) ou domaines (temps réel, processus métier, etc.) [OMG 2011c].

Le mécanisme de profil a été défini spécifiquement pour fournir un mécanisme léger d'extension pour le standard UML. En UML, les stéréotypes sont des méta-classes spécifiques, les tagged values sont des méta-attributs standards et les profils sont des Packages spécifiques.

Pour récapituler, un profil UML permet de :

- Ajouter des concepts (Terminologies) relatifs à une plateforme ou un domaine particulier ;
- Fournir / Changer la représentation de ces concepts, avec les images associées aux stéréotypes, pour utiliser la représentation « traditionnelle » spécifique du domaine ;
- Fixer les points de variation sémantique : faire un choix parmi les possibilités sémantiques laissées ouvertes par UML. Par exemple : les signaux (*signal*) peuvent être en broadcased ou mulitcasted, les State-Machines ont une queue FIFO ou LIFO, etc. ;
- Ajouter des contraintes sur les associations entre ces concepts, c'est-à-dire restreindre les associations existantes au niveau des méta-classes UML, limiter le nombre d'attribut du concept à N, forcer des règles logiques d'associations, etc. ;
- Ajouter des contraintes sur l'utilisation ou non de certains concepts selon le contexte ;
- Ajouter de l'information qui peut être utilisée lors de la transformation entre un modèle vers un autre ou vers du code.

3.2.2 DSML (Domain-Specific Modeling Language)

Notez bien qu'il faut bien séparer clairement les approches IDM du formalisme UML. En effet, la portée de l'IDM est plus large que celle d'UML. UML est un standard obtenu par consensus *a maxima*, dont on doit réduire ou étendre la portée à l'aide de mécanismes comme les profils. Ces mécanismes n'ont pas toujours la précision souhaitable et mènent parfois à des contorsions dangereuses pour rester dans l'espace UML. Au contraire, l'IDM favorise la définition des langages de modélisation dédiés à un domaine particulier (DSML) offrant ainsi aux utilisateurs, des concepts propres à leurs métiers et dont ils ont la maitrise. Ces langages sont généralement de petites tailles et doivent être facilement manipulables, transformables, etc. Parmi les avantages d'utiliser les DSML, est de bénéficier des langages dédiés permettant d'exprimer des solutions avec le niveau d'abstraction adéquat au domaine traité, faciliter la documentation du code, améliorer la productivité, la qualité, la fiabilité, la maintenabilité et les possibilités de réutilisation, permettre la validation au niveau domaine ainsi que de fournir, au besoin, une ou plusieurs syntaxes concrètes (présentations graphiques ou textuelles) associées à

ce langage. Un langage que ce soit en informatique ou en linguistique est caractérisé par une syntaxe et une sémantique. La syntaxe décrit les différentes constructions du langage et les règles d'agencement de ces constructions. La sémantique désigne un sens à chacun des constructions des langages. En linguistique et en domaine de langages visuels, on distingue généralement la syntaxe concrète, manipulée par l'utilisateur du langage elle-même composée d'un vocabulaire (les blocs élémentaires de construction) et d'une grammaire (les règles de compositions et d'agencement des constructions), de la syntaxe abstraite qui est la représentation interne manipulée par l'ordinateur [Aho 1986]. Par ailleurs, on distingue également la sémantique du langage qui représente le sens des éléments du langage. Il est donné en associant (mapping / binding) les constructions de la syntaxe concrète avec le sens auquel elles correspondent dans le domaine sémantique. Dans le contexte de l'IDM, la syntaxe abstraite est placée au coeur de la description du langage de modélisation. Elle est donc généralement décrite en premier et sert de base pour définir la syntaxe concrète. La définition de la syntaxe concrète consiste à définir des représentations textuelles ou graphiques et à définir un lien (M_{ac}) entre les constructions de la syntaxe abstraite et les représentations de la syntaxe concrète. On peut même envisager de définir plusieurs syntaxes concrètes pour une même syntaxe abstraite (M^*_{ac}). La sémantique est exprimée à partir des constructions de la syntaxe abstraite par un lien vers un domaine sémantique (M_{as}). Plusieurs mappings (M^*_{as}) peuvent être définis et donnent lieu à différentes sémantiques, pour des objectifs différents, à des niveaux d'abstraction différents. Un langage de modélisation (Lm) est donc défini selon le tuple AS, CS^*, M^*_{ac}, SD^*, M^*_{as} où AS est la syntaxe abstraite, CS^* est la (les) syntaxe(s) concrète(s), M^*_{ac} est le mapping de syntaxe abstraite vers sa (ses) représentation(s) concrète(s), SD^* est le (les) domaine(s) sémantique(s) et M^*_{as} est l'ensemble des mappings de la syntaxe abstraite vers le(s) domaine(s) sémantique(s) [Jezequel 2012].

3.3 Processus de métamodélisation en pratique

La métamodélisation est l'activité consistant à définir le métamodèle d'un langage de modélisation. Elle vise donc à modéliser un langage, qui joue alors le rôle de système à modéliser.

3.3.1 Sémantique

La sémantique d'un langage est la signification des constructions de ce langage. Elle permet ainsi de donner un sens précis aux modèles construit à partir à celle-ci. On dit qu'une sémantique est formelle si elle est exprimée dans un formalisme mathématique et permet de vérifier la cohérence et la complétude de cette définition. Définir une sémantique d'un langage de modélisation consiste à choisir/définir le domaine sémantique SD et à définir le mapping M_{as} entre ce dernier et la syntaxe abstraite ($AS \longleftrightarrow SD$). Dans le contexte de l'IDM, la définition du domaine sémantique et du mapping prend la forme de modèle [Hausmann 2005, Harel 2000, Harel 2004].

– Sémantique statique : on distingue la sémantique statique qui correspond à des propriétés indépendantes de l'exécution ou valable pour toutes les exécutions,

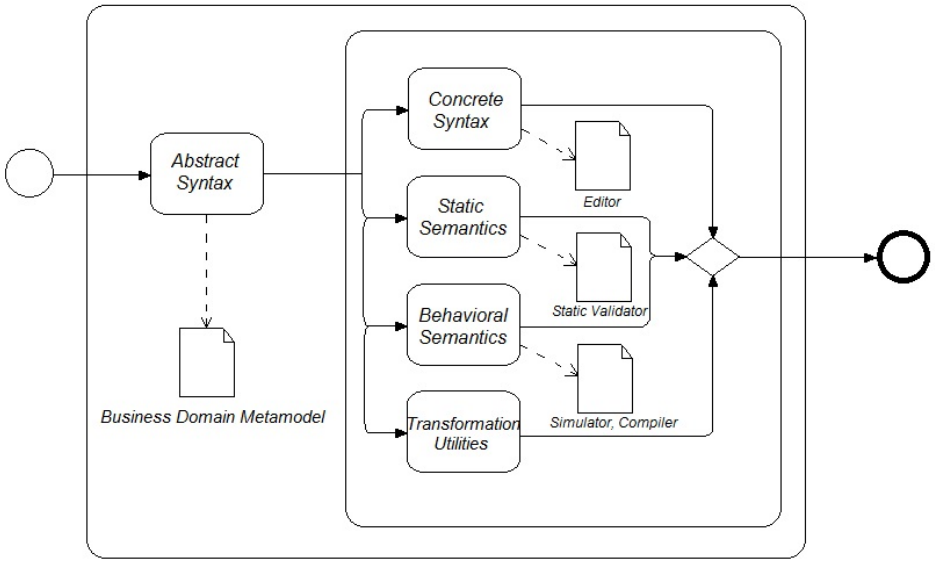


FIGURE 3.4 – Processus de métamodélisation

de la sémantique dynamique (ou comportementale) qui permet de décrire le comportement des programmes à l'exécution. La sémantique statique est vérifiée généralement statiquement lors de la compilation des programmes et exprime des règles qui ne peuvent pas être exprimés par la syntaxe, comme le typage, etc.). Cette sémantique peut être exprimée sur la syntaxe abstraite, soit à l'aide du langage de métamodélisation lui-même soit en la complétant de contraintes exprimée à l'aide d'un langage de contraintes (OCL par exemple) ;

– Sémantique dynamique : elle peut être décrite de différentes techniques allant du plus concrète au plus abstraite, elle peut être opérationnelle, dénotationnelle ou axiomatique [Winskel 1993]. Elle permet de donner un caractère opératoire à un DSML, on dit alors que le métamodèle est exécutable et permet de faire évoluer les modèles qui lui sont conformes au cours d'une exécution conformément à la sémantique du langage.

    1. Sémantique axiomatique.

       La sémantique axiomatique encore appelée logique de Hoare [Hoare 1983] donne une vision déclarative en décrivant l'évolution des caractéristiques d'un élément lors du déroulement d'un programme. Elle peut être exprimées sur la syntaxe abstraite, sois à l'aide du langage de modélisation lui-même soit en la complétant de contraintes exprimées à l'aide d'un langage comme OCL.

    2. Sémantique opérationnelle.

       Une sémantique opérationnelle propose une vision impérative en décrivant un programme par un ensemble de transformations entre les états du contexte d'exécution. Elle s'exprime sur une représentation abstraite similaire à celle de la syntaxe abstraite du langage considéré, à laquelle sont rajoutées

les informations que l'on souhaite capturer sur l'exécution, c'est-à-dire les informations permettant de distinguer les différents états du modèle au cours de l'exécution, en fonction du niveau d'abstraction choisi. Elle considère donc comme domaine sémantique une extension du domaine syntaxique. Cela permet d'exprimer la sémantique comportementale directement sur les concepts de base du langage dédié au domaine, instanciés à partir des concepts de métamodélisation. Dans le contexte des langages naturels, une sémantique opérationnelle revient à manipuler une phrase mentalement, dans la même langue que celle utilisée pour l'écrire initialement [Jezequel 2012].

Dans la pratique, différentes solutions ont été explorées pour implémenter une sémantique opérationnelle directement dans le métamodèle. Une première solution est l'utilisation d'un langage de métamodélisation exécutable comme Kermeta, xOCL ou même java avec l'Api d'EMF. La seconde solution est d'exprimer une transformation endogène (voir section 3.3.4) sur la syntaxe abstraite à l'aide d'un langage de transformation déclaratif (QVT par exemple).

3. Sémantique dénotationnelle.

   Une sémantique dénotationnelle décrit, sous forme de fonctions, l'effet d'un programme et non la manière dont celui-ci est exécuté. Elle s'appuie sur une représentation abstraite différente de celle définie dans la syntaxe abstraite du langage considéré, elle traduit chaque état du modèle dans un autre espace technique, s'appuyant ainsi sur la sémantique de celui-ci. Pratiquement, définir une sémantique dénotationnelle consiste à définir la traduction de chaque construction du langage de modélisation en construction(s) du langage de l'espace technique cible choisi. Dans le contexte des langages naturels, cela revient à interpréter une phrase dans une langue différente que celle dans laquelle elle a été initialement écrite. Ce type de sémantique, correspond dans l'IDM à une sémantique par traduction (translational semantics) qui définit un langage par sa transformation vers un autre formellement défini [Clark 2004]. D'autres nommages décrivant la sémantique des langages ont été proposés. Clarck et al. [Clark 2004, Clark 2008] proposent la notion de sémantique par extension, consistant à étendre les concepts et la sémantique d'un langage existant. La sémantique est néanmoins donnée de manière opérationnelle ou par traduction (dénotationnelle) [Jezequel 2012].

3.3.2 Syntaxe abstraite

La syntaxe abstraite (AS) d'un langage de modélisation exprime, de manière structurelle, l'ensemble de ses concepts et leurs relations. Les langages de métamodélisation tels que MOF, offrent les constructions élémentaires qui permettent de décrire une telle syntaxe abstraite à travers un modèle appelé métamodèle. Il existe à ce jour de nombreux environnements et langages de métamodélisation : Eclipse EMF/Ecore [Budinsky 2003], GME/MetaGME [Ledeczi 2001b], XMF-Mosaic/Xcore [Clark 2004], MetaEdit+/GOPPR

[Kelly 1996] ou Kermeta [Muller 2005]. Tous ces langages s'inspirent de l'approche orientée objet, les langages de métamodélisation objet offrent le concept de classe pour définir les concepts du langage. Une classe est composée de propriétés (attributs ou références) qui la caractérisent. Une propriété est appelée référence lorsqu'elle est typée par une autre classe et et attribut lorsqu'elle typée par un type de donnée (booléen, chaîne de caractère, entier, etc.). Par convention, la présentation graphique d'un métamodèle utilise le diagramme de classe d'UML.

Un métamodèle définit un ensemble de modèles valides. Cependant, certaines contraintes ne peuvent pas directement être exprimées en utilisant un langage de métamodélisation non-exécutable tel que MOF. Il y a pas moyen par exemple, d'exprimer que les noms des attributs doivent être différents les uns des autres lorsqu'on déclare les propriétés d'une classe. Ces contraintes font partie de ce qu'on appelle la sémantique statique du langage et sont majoritairement exprimés à l'aide d'un langage de contraintes comme OCL. OCL (Object Constraint Language) [OMG 2010] est défini par l'OMG pour exprimer des contraintes sous la forme d'expressions de premier ordre sur des métamodèles objet décrit à l'aide de langage non-exécutable (comme MOF). Il permet ainsi d'exprimer la sémantique statique du domaine que l'on souhaite capturer dans la syntaxe du DSML. Une expression OCL est décrite par les éléments suivants :

1. Un contexte qui définit la situation dans laquelle la contrainte doit être valide. Si le contexte est une métaclasse du DSML, la contrainte sera appliquée à l'ensemble des instances de cette métaclasse. Si le contexte est une propriété d'une métaclasse, la contrainte sera évaluée sur toutes les instances de la métaclasse portant cet attribut, mais ne s'appliquera qu'à cette propriété ;
2. Un contrat qui définit l'expression qui devra être respectée. Plusieurs types de contrat existent : invariant (inv), pré-condition (pre), post-condition (post), dérivation d'attribut (derive), initialisation d'attribut (init) et résultat d'opération (body). Le type le plus fréquent et le plus utile dans le cadre de métamodélisation est l'invariant. Ce type de contrat exploite une expression booléenne pour exprimer les règles de bonne formation de nos modèles (utilisées pour construire des vérificateurs de modèles ou model checkers).

   Les pré et post conditions sont aussi des expressions booléennes qui doivent s'appliquer respectivement avant et après l'exécution d'une opération pour vérifier cette dernière. La dérivation d'attributs permet de définir un attribut calculé à partir d'autres informations dans le modèle (par exemple, l'attribution du nom par défaut d'un élément du modèle). L'initialisation d'attribut est une expression définissant la valeur initiale d'un attribut. Le résultat d'opération permet de spécifier le retour d'une opération.

Pour persister physiquement les modèles dans le système de fichiers et pouvoir les échanger, il est nécessaire de disposer d'un format de fichier compris par tous les outils qui manipulent ces modèles. Pour cela l'OMG a proposé le standard XMI (XML Metadata Interchange) [OMG 2013]. XMI indique comment les modèles MOF peuvent être sérialisés en format XML. Le but de ce standard est de permettre à des outils de modélisation

ou outils CASE (Computer-Aided Software Engineering) d'explorer et d'échanger les définitions des structures de données, leurs propriétés, les relations qui les unissant, etc.

Pour définir la syntaxe abstraite d'un langage de modélisation dans un langage comme Ecore, on spécifie les éléments du langage à modéliser (les concepts) sous forme d'arbre (Arbre syntaxique) en spécifiant leurs attributs dans la vue de propriétés de l'éditeur arboré.

Le principe global à appliquer est de définir des associations de type composition entre les principaux types d'éléments d'un modèle afin de former un arbre. On commence donc par créer ce qui joue le rôle du sommet de l'arbre, puis à partir de là, on crée les éléments suivants. On retrouve ce principe quand on crée un modèle Ecore : l'élément principal créé est un package, dans lequel on peut créer des classes puis les attributs de ces classes ainsi que les associations entre elles. Il s'agit donc de définir dans les métamodèles Ecore un élément particulier qui servira de racine. Cet élément racine, qui sera créé en premier dans le modèle, devra permettre de créer les autres éléments. De manière concrète, l'éditeur permettra de créer des types d'éléments qui sont en relation directe avec l'élément racine via une association de type composition (de même, à partir d'un élément créé, on pourra créer d'autres éléments qu'il contiendra s'il y a une relation de composition, comme pour un modèle Ecore quand on rajoute des méthodes ou des attributs à une classe) [Cariou 2011].

Il est aussi possible d'afficher et d'éditer le modèle via une syntaxe graphique (à la UML). Pour cela, Ecore propose un diagramme pour spécifier les métamodèles. On peut alors modifier le modèle selon l'une et l'autre des vues (arbre ou graphe), les modifications sont prises en compte dans l'autre vue. Notez qu'il est aussi possible d'obtenir un métamodèle au format Ecore à partir de sources externes. Par exemple, on pourra convertir un schéma XML (.xsd), ou un profil UML conçu par un modeleur comme Papyrus ou Rational Rose pour définir les concepts du métamodèle et d'importer ce fichier dans Ecore pour générer l'arbre syntaxique. Ainsi les instances (modèles) conforme à ce métamodèle, sont aussi créées dans des éditeurs arborés ou, si une syntaxe concrète (graphique ou textuelle) a été définie, seront manipuler sur les éditeurs de celle-ci. Ce format Ecore, joue un rôle clé dans le développement de DSML, car c'est à partir des informations issues des métamodèles écrits en Ecore que seront construits les outils qui manipulent les modèles. Les Frameworks tels que EMF, exploitent des métamodèles Ecore pour charger et sauver les modèles au format XMI [Jezequel 2012].

D'autres méthodes pour définir la syntaxe abstraite utilisent une description textuelle, c'est le cas de Kermeta. Ce langage peut être vu comme étant composé d'un méta-méta-modèle (une extension d'EMOF) auquel est associé un langage permettant de définir des actions sur les métamodèles. Ces actions peuvent servir par exemple à vérifier la cohérence d'un modèle (avec des validations de contraintes OCL) ou bien encore à réaliser des transformations de modèles [Cariou 2011]. Il est possible de passer d'une représentation textuelle à une autre (représentation UML ou arborée), elles sont toutes équivalentes. Il est également possible de transformer un modèle Ecore en un modèle Kermeta et inversement. Parmi les fonctionnalités de Kermeta, les opérateurs *require* et *aspect* qui permettent de mettre en relation les préoccupations de conception de DSML (Syntaxe abstraite, Syntaxe concrète et Sémantiques) en complétant un

métamodèle existant avec de nouveaux éléments. L'opérateur *aspect* permet de rouvrir une classe créée précédemment et de lui ajouter de nouvelles informations telles que de nouvelles propriétés ou opérations, etc. L'opérateur *require* est un opérateur de composition statique qui permet d'assembler différentes unités (c'est-à-dire fichiers) et de les composer automatiquement dans un métamodèle unique. Le modèle composé subit une vérification de type pour garantir la bonne intégration de toutes ces unités. Dans le cas des langages de métamodélisation exécutable tel que Kermeta par exemple, les contraintes OCL peuvent être tissée directement dans le métamodèle. Ce langage permet aussi de choisir quand les contraintes doivent être vérifiées.

3.3.3 Syntaxe concrète

Les syntaxes concrètes (CS) d'un langage fournissent à l'utilisateur un ou plusieurs formalismes pour manipuler les concepts de la syntaxe abstraite et ainsi créer des instances du métamodèle, c'est-à-dire des modèles. Les modèles ainsi obtenus seront conforme à la structure définie par la syntaxe abstraite. La définition d'une syntaxe concrète consiste à définir un des mapping M^{*}_{ac} ($M_{ac} : AS \longleftrightarrow CS$) et permet ainsi de *décorer* ou d'*annoter* chaque construction ou un ensemble de constructions du langage définie(s) dans la syntaxe abstraite par une ou plusieures décorations de la syntaxe concrète qui peuvent être manipulées par l'utilisateur du langage.

La syntaxe concrète des langages de modélisation est l'objet de notre thème de recherche. Nous détaillons cet aspect dans le chapitre 4.

3.3.4 Transformation de modèles

Tout processus de développement logiciel englobe un certain nombre d'activités (comme l'expression des besoins, l'analyse, la conception, l'implantation ou encore la validation) qui chacune produit un ou plusieurs artefacts tels que la documentation, des diagrammes, des codes sources, des fichiers de configuration, des fiches de tests, des rapports de qualification, etc. Ces artefacts donnent de multiples points de vue sur le logiciel en cours de développement. Cependant, le développement logiciel est souvent confronté à assurer une cohérence entre ces vues, ou au minimum une traçabilité entre les éléments des différents artefacts. L'IDM offre un cadre méthodologique et technologique unificateur grâce à l'utilisation intensive des modèles et des transformations entre les modèles. Les transformations permettent de choisir l'espace technique et le formalisme le plus adapté à chaque activité [Favre 2006].

Le principe même de l'IDM est de capitaliser le savoir-faire au niveau des modèles et non plus au niveau du code source. On passe des approches centrées code vers les approches centrées modèles tout en ayant comme objectif de générer/synthétiser une application sous la forme d'un code, binaire ou autre, afin de pouvoir l'exécuter ou l'interpréter sur une machine (des techniques de synthèse d'applications logicielles à partir de modèles : l'interprétation de modèles, la compilation de modèles et la génération de code [Favre 2006]. Au niveau pratique, on peut trouver deux catégories de transformation : les transformations de modèles à modèles (M2M) et les transformations de modèles à texte

(M2T). Les transformations de modèles à modèles permettent de produire et de modifier automatiquement des modèles. Cette manipulation automatique est essentielle pour plusieurs raisons :

1. Les modèles peuvent être relativement complexes, impliquant une complexité équivalente pour les manipuler manuellement ;
2. Les modèles à traiter peuvent être nombreux et un même traitement peut devoir être appliqué plusieurs fois pour chacun d'eux ;
3. Le gain de temps obtenu grâce au travail à un niveau d'abstraction plus élevé est perdu si le passage vers différents niveaux n'est pas automatisé.

De manière générale, une transformation est définie à partir de ses métamodèles source et destination. Elle établit un ensemble de relations exprimant les actions à effectuer sur les modèles d'entrée pour produire les modèles de sortie correspondants. Pour écrire une transformation, différentes approches peuvent être employées. Ils sont classifiés dans [Czarnecki 2003].

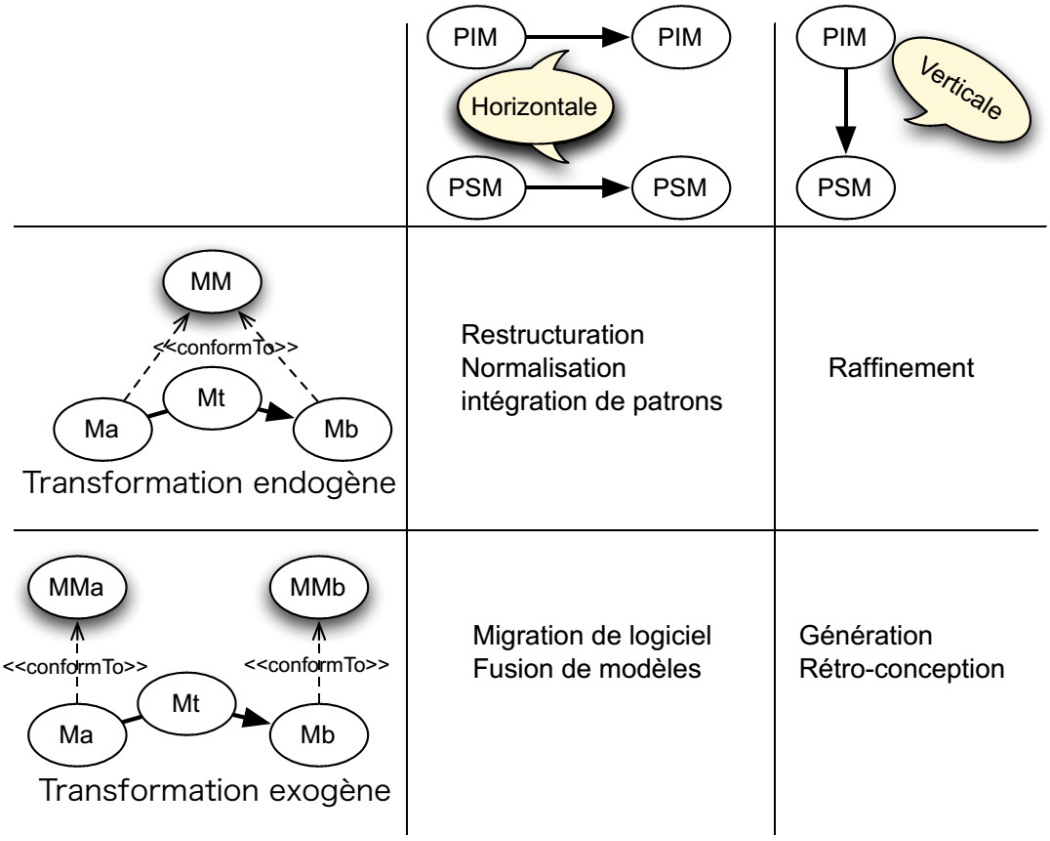


FIGURE 3.5 – Type de Transformations [Combemale 2008]

Le processus général d'une transformation consiste à transformer un modèle M1 en un modèle M2 qui sont respectivement conformes à leurs métamodèles MM1 et MM2.

Si les métamodèles MM1 et MM2 sont identiques, on parle alors d'une transformation endogène ou sur place (in place). Si les métamodèles MM1 et MM2 sont différents, la transformation est dite exogène (in vers out). Ce dernier type de transformation est le plus fréquemment utilisé dans l'architecture MDA proposée par l'OMG. Les transformations de modèles sont utilisées pour changer de niveau d'abstraction et produire un modèle dédié à une plate-forme (figure 3.5) [Gerber 2002a]. D'autres classifications des différents types de transformation peuvent être faites selon le changement de niveau d'abstraction. Lorsqu'on produit un modèle dans le même espace d'abstraction que celui du modele source, cette transformation est dite horizontale. Lorsqu'on produit un modèle dans un espace d'abstraction différent que celui du modèle source (on introduit un changement de niveau abstraction, cette transformation est nommée verticale [Mens 2006].

Les travaux réalisés dans le domaine de la transformation de modèle peuvent être chronologiquement classés selon plusieurs générations en fonction de la structure de données utilisée pour représenter le modèle [Bézivin 2003].

- Première génération : transformation des structures d'enregistrement. Dans ce cas un script spécifie comment un flux d'entrée (input flow) est réécrit/structuré en un flux de sortie (output flow). Par exemple, les scripts système : shell, batch ou Perl. Malgré la difficulté de maintenance et de lisibilité de ces systèmes, ils nécessitent une analyse grammaticale du texte d'entrée et une adaptation du texte de sortie [Gerber 2002b, Bézivin 2003] ;
- Deuxième génération : transformation d'arbres. Ces transformations consistent à parcourir un arbre d'entrée (généralement au format XML) et de générer des fragments de l'arbre de sortie. La sortie peut être d'un format balisé (XML, HTML, SVG), comme elle peut être du simple texte. Ces méthodes se basent sur les technologies XSLT ou XQuery ;
- Troisième génération : transformation de graphes. Avec ces méthodes, un modèle d'entrée est considéré comme un graphe orienté étiqueté. Cette approche considère l'opération de transformation comme un autre modèle conforme à son propre métamodèle, lui-même défini à l'aide d'un langage de métamodélisation (ex. MOF). Ceci est illustré dans la figure 3.6.

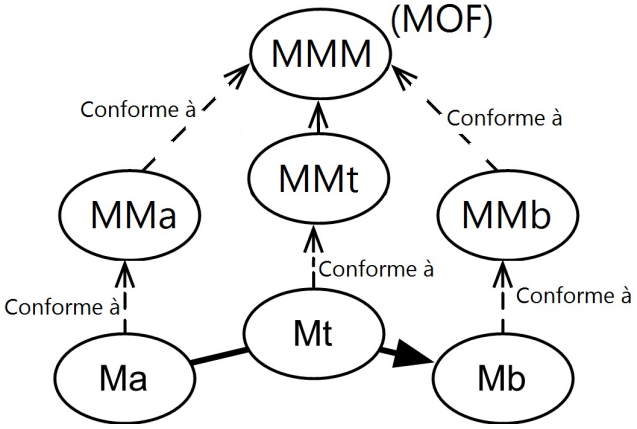


FIGURE 3.6 – Transformation de modèles

Cette dernière génération a donné lieu à d'importants travaux de recherche. Celles-ci peuvent être caractérisées à partir de critères comme le paradigme pour la définition des transformations, les scénarios de transformation, le nombre de modèles sources et cibles, la traçabilité, le langage de navigation utilisé, l'organisation et l'ordonnancement des règles, etc. [Jouault 2006].

Plusieurs langages de transformation sont disponibles à ce jour pour décrire des transformations de troisème génération. On peut citer les langages généralistes qui s'appuient directement sur la représentation abstraite du modèle comme l'API EMF qui permet de manipuler un modèle sous forme de graphe tout en l'associant à du code java. Dans ce cas c'est au programmeur de faire la recherche d'information dans le modèle, d'expliciter les règles de transformation et de gérer les éléments cibles construits, etc. D'autres langages reposent sur la définition d'un métamodèle dédié à la transformation de modèle et des outils permettant d'exécuter les modèles de transformation. Nous citons par exemple ATL, qui est un langage hybride (déclaratif et impératif) qui permet de définir des transformations de modèles à modèles (Modules) et des transformations de modèles à texte (Query). Cependant, un standard est proposé par l'OMG pour normaliser l'implémentation des différents langages dédiés à la transformation de modèles. Il s'appelle QVT (Query/View/Transformation) [OMG 2011b] et il repose sur l'utilisation du langage OCL pour naviguer à travers les modèles. Afin d'écrire les transformations, le standard QVT définit trois sous-langages : *Relation*, *Operational Mapping* et *Core*. Chacun de ces trois sous-langages repose sur un paradigme de transformation de langage différent à savoir déclaratif, impératif et hybride [Kurtev 2008]. Les langages *Relations* et *Core* sont tous deux déclaratifs, mais placés à différents niveaux d'abstraction. L'un des buts de Core est de fournir une base pour la spécification de la sémantique de *Relations*. La sémantique de *Relations* est donnée comme une transformation de *Relations* vers *Core*. Il est parfois difficile de définir une solution complètement déclarative à un problème de transformation donné. Pour adresser cette question, QVT propose deux mécanismes pour étendre *Relations* et *Core* : un troisième langage appelé *Operational Mapping* et un mécanisme d'invocation de fonctionnalités de transformation implémentées dans un langage arbitraire (boîte noire ou black box). *Operational Mappings* étend *Relations* avec des constructions impératives et des constructions OCL avec effets de bord.

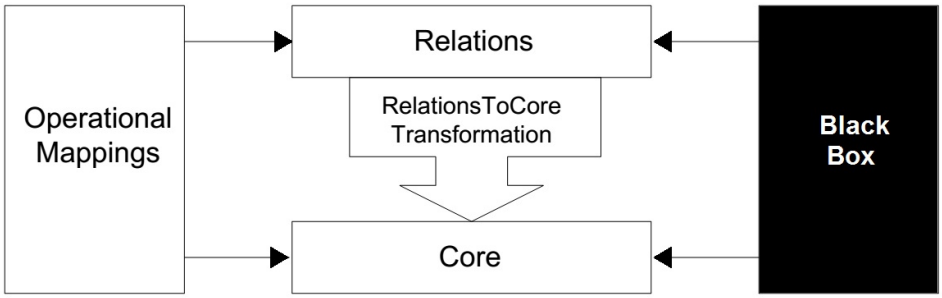


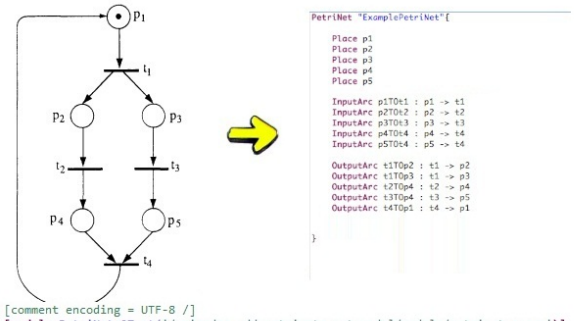
FIGURE 3.7 – Architecture du Standard QVT

Le standard QVT propose une définition syntaxique et sémantique des trois sous-

langages, mais ne donne aucune direction concernant l'implémentation. Ainsi plusieurs implémentations existent actuellement, comme QVTo proposé par Borland et Smart-QVT proposé par France Telecom. Ces deux implémentations concernent le sous-langage *Operational Mapping*.

Les transformations de modèles à modèles ne sont pas les seuls types de transformations que l'on peut retrouver en IDM. Les transformations de modèles vers texte (M2T) prennent des modèles en entrée et produisent du texte qui peut être, par exemple, du code source ou encore de la documentation.

Le processus général d'une transformation de modèles vers texte consiste à générer un morceau de code pour un objet ou un ensemble d'élément des modèles sources. Afin de réaliser cette tâche on utilise souvent des template [Czarnecki 2003]. Un template consiste en un block de texte contenant des méta-codes permettant d'effectuer diverses tâches sur les modèles source et ce itérativement. Finalement, la structure du template est généralement assez proche du code généré. Les templates peuvent être composés pour répondre aux besoins complexes de transformation. Les grandes transformations peuvent être structurées en modules comportant des parties publiques et privées. Les templates ont une nature WYSIWYG, le texte spécifié est disposé exactement de la façon dont il devrait être dans la sortie. Dans la plupart des cas, ce type de spécification de template est intuitif.




```
PetriNet "ExamplePetriNet"{

    Place p1
    Place p2
    Place p3
    Place p4
    Place p5

    InputArc p1TOt1 : p1 -> t1
    InputArc p2TOt2 : p2 -> t2
    InputArc p3TOt3 : p3 -> t3
    InputArc p4TOt4 : p4 -> t4
    InputArc p5TOt4 : p5 -> t4

    OutputArc t1TOp2 : t1 -> p2
    OutputArc t1TOp3 : t1 -> p3
    OutputArc t2TOp4 : t2 -> p4
    OutputArc t3TOp4 : t3 -> p5
    OutputArc t4TOp1 : t4 -> p1

}
```

```
[comment encoding = UTF-8 /]
[module PetriNets2Text('/uniandes.mdd.petrinets.metamodel/models/petrinet.ecore')]

[template public generateElement(aPetriNet : PetriNet)]
[comment @main/]
[file (aPetriNet.name + '.pnl2', false, 'UTF-8')]
PetriNet "[aPetriNet.name/]"{

    [for (element : Element | aPetriNet.elements)]
    [if (element.oclIsTypeOf(Place))]
    Place [element.oclAsType(Place).name/]
    [/if]
    [/for]

    [for (element : Element | aPetriNet.elements)]
    [if (element.oclIsTypeOf(InputArc))]
    InputArc [element.oclAsType(InputArc).name/] : [element.oclAsType(InputArc).from.name/] -> [element.oclAsType(InputArc).to.name/]
    [/if]
    [/for]

    [for (element : Element | aPetriNet.elements)]
    [if (element.oclIsTypeOf(OutputArc))]
    OutputArc [element.oclAsType(OutputArc).name/] : [element.oclAsType(OutputArc).from.name/] -> [element.oclAsType(OutputArc).to.name/]
    [/if]
    [/for]
}
[/file]
[/template]
```

FIGURE 3.8 – Transformation d'un modèle (réseau de petri) vers du texte avec une template Acceleo

Plusieurs langages de transformation de modèles vers texte sont disponibles à ce

jour pour décrire les règles de génération du code source, documentation, etc. On peut citer le projet M2T d'Eclipse qui chapeaute plusieurs sous-projets comme JET (un dérivé des JSP - JavaServer Pages) proposé par IBM et utilisé dans EMF pour la génération de code. Le projet XPand proposé par Itemis comme langage de template, il supporte le polymorphisme de template et la programmation par aspect (il est possible d'intercepter le comportement des templates liées) et permet via le langage d'expression Xtend, d'appeler des bouts de code java dans les templates.

Dans l'optique de compléter QVT, un standard est proposé par l'OMG pour normaliser l'implémentation des différents langages dédiés à la transformation de modèles vers texte. Ce standard s'appelle MOF-M2T (Model to Text). Ainsi, plusieurs implémentations existent actuellement pour ce standard, comme Acceleo proposé par Obeo et MOFScript qui est un moteur de génération développé en réponse à la RFP de l'OMG.

3.4 Synthèse et discussion

Nous avons introduit dans ce chapitre les principes généraux de l'IDM, c'est à-dire la métamodélisation d'une part et la transformation de modèle d'autre part. L'IDM favorise la définition de langages de modélisation dédiés à un domaine particulier (DSML) offrant ainsi aux utilisateurs des concepts propres à leurs métiers et dont ils ont la maitrise. Ces langages sont généralement de petites tailles et doivent être facilement manipulables, transformables, etc. Parmi les avantages d'utiliser les DSML est de bénéficier des langages dédiés permettent d'exprimer des solutions avec le niveau d'abstraction adéquat au domaine traité, faciliter la documentation du code, améliorer la productivité, la qualité, la fiabilité, la maintenabilité et les possibilités de réutilisation, permettre la validation au niveau domaine ainsi que de fournir, au besoin, une ou plusieurs syntaxes concrètes (présentations textuelles ou graphiques) associées à ce langage. Comme tout autre langage, les DSML sont caractérisés par une syntaxe et une sémantique. La syntaxe décrit les différentes constructions du langage et les règles d'agencement de ces constructions alors que la sémantique désigne un sens à chacune des constructions de ce langage. Il existe deux types de syntaxe : la syntaxe concrète, manipulée par l'utilisateur du langage elle-même composée d'un vocabulaire (les blocs élémentaires de construction) et d'une grammaire (les règles de compositions et d'agencement des constructions) et la syntaxe abstraite qui est la représentation interne manipulée par l'ordinateur. Par ailleurs, nous n'avons pas abordé dans ce chapitre les aspects liés à la syntaxe concrète graphique, puisqu'ils font l'objet du chapitre 4.

Les techniques de transformation de modèle, clé du succès de l'IDM afin de pouvoir rendre opérationnels les modèles, sont issues des principes qui sont bien antérieurs à l'IDM. Les travaux récents de normalisation (QVT de l'OMG par exemple [OMG 2011b]) et d'implantation d'outils ont permis de faire évoluer ces techniques.

CHAPITRE 4

Langages visuels

Sommaire

Les syntaxes concrètes (CS) d'un langage fournissent à l'utilisateur un ou plusieurs formalismes pour manipuler les concepts de la syntaxe abstraite et ainsi créer des instances du métamodèle, c'est-à-dire des modèles. Les modèles ainsi obtenus seront conforme à la structure définie par la syntaxe abstraite. La définition d'une syntaxe concrète consiste à définir un des mapping M^*_{ac} ($M_{ac} : AS \longleftrightarrow CS$), et permet ainsi de *décorer* ou d'*annoter* chaque construction ou un ensemble de constructions du langage, définies dans la syntaxe abstraite par une ou plusieurs décorations de la syntaxe concrète et pouvant être manipulées par les utilisateurs du langage.

En fonction de ses besoins, un utilisateur peut choisir de représenter son modèle de différentes façons. On pourra utiliser des syntaxes concrètes textuelles pour manipuler le modèle sous la forme d'un fichier texte. Il est également possible de décrire des syntaxes concrètes graphiques pour représenter le modèle sous forme d'arbres, de tableaux, de formulaire ou plus classiquement sous forme de diagramme qui est la forme pour laquelle on s'est intéressé durant nos travaux de recherche.

Notez qu'il est important de distinguer un modèle d'un diagramme. Un diagramme est une vue graphique sur un modèle [Jezequel 2012]. Pour cela, il peut ne pas présenter la totalité du modèle, mais présente une partie des éléments du modèle d'une manière lisible. Plusieurs diagrammes peuvent donc être utilisés pour présenter le même modèle.

Dans ce chapitre, nous présentons les concepts généraux des langages visuels, avec lesquels on construit la syntaxe concrète graphique (sous forme de diagrammes), leur classification ainsi que les éléments de bases qui les composent. Ainsi, nous répondons à la première et deuxième questions de la problématique.

4.1 Nature des langages visuels

Les représentations visuelles sont parmi les plus anciennes formes de représentations de connaissance et précèdent l'apparition de l'écriture d'environ 25.000 ans [Tufte 1983]. Les Philosophes de la science avaient étudié la linguistique et les représentations mathématiques dans l'ordre de comprendre la science, mais ce n'est que très récemment que les investigations ont commencé sur la manière avec laquelle les représentations

visuelles contribuent à la compréhension. Les représentations visuelles jouent un rôle significatif dans le raisonnement scientifique [Perini 2005].

FIGURE 4.1 – Les notations visuelles existent depuis la pré-histoire

Nous présentons dans cette section, les bases de cette forme de représentation dans l'optique d'en extraire les concepts qui décrivent un diagramme. Pour cela, il sera nécessaire de définir la notion de *diagramme*, sa nature, ses différents types et les points essentiels pour atteindre une efficacité cognitive d'un diagramme.

Dans la littérature, plusieurs définitions existent pour le concept de diagramme, la définition la plus connue est celle de [Kosslyn 1980, Larkin 1987, Tversky 1997] : *Les diagrammes sont des moyens efficaces pour présenter la pensée humaine et la résolution des problèmes. Les diagrammes sont des représentations graphiques d'informations, symbolique et orienté-humain ; ils sont créés par des humains pour des humains.* Ils ont une petite (voire nulle) valeur dans la communication avec les machines dont la capacité de traitement visuelle est au mieux primitive [Moody 2009b].

Les composants élémentaires d'une représentation visuelle (*diagramme*) sont appelées les notations visuelles (ou aussi langage visuel, notation diagrammatique, notation graphique). Comme tous les langages, les notations ou langages visuels sont constituées d'un ensemble de symboles graphiques (**le vocabulaire visuel**), d'un ensemble de règles de composition structurant la notation visuelle (**la grammaire visuelle**) et d'une définition du sens de chaque symbole (**la sémantique**). Le vocabulaire et la grammaire visuelle forment tous les deux ce que l'on appelle la **syntaxe concrète graphique** ou visuelle.

Les symboles graphiques sont utilisés pour représenter visuellement des constructions sémantiques, définies typiquement par un métamodèle [Amyot 2006]. Les sens des symboles graphiques sont définis par leurs associations avec les concepts sémantiques qu'ils représentent. Dans les notations visuelles, une expression valide (Celle qui respecte un vocabulaire et une grammaire) s'appelle **phrase visuelle** ou **diagramme**. Les diagrammes sont composés d'instances de symboles (tokens), arrangés et structurés selon les règles de la grammaire visuelle [Moody 2009a]. Une telle distinction entre le contenu (sémantique) et la forme (syntaxe : vocabulaire et grammaire) nous a permis de séparer les différentes préoccupations de notre proposition. Ces définitions sont illustrées dans la figure 4.2.

Les notations visuelles sont uniquement des représentations orientées-humains :

leur seul objectif est de faciliter la communication humaine et la résolution de problèmes [Harel 1988]. Pour être efficaces dans ces tâches, elles ont besoin d'être optimisées pour un traitement efficace par le cerveau humain.

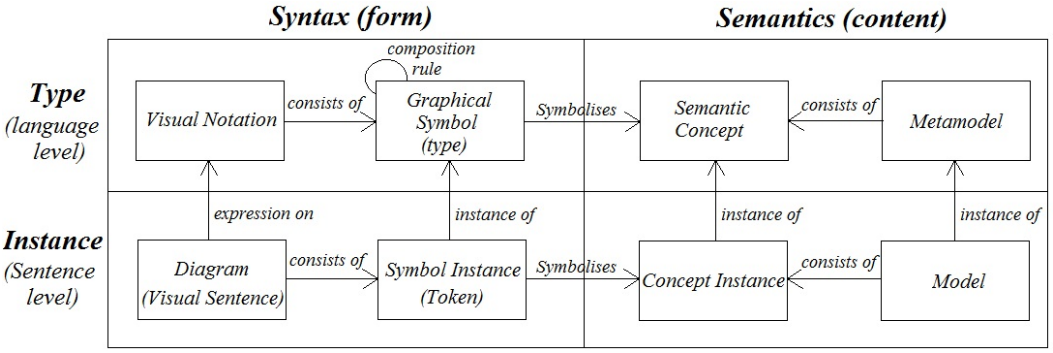


FIGURE 4.2 – La Nature d'une Notation Visuelle

Un diagramme est une phrase dans un langage visuel [Mackinlay 1986]. Sachant que l'objectif premier de n'importe quel langage est de communiquer, un *bon* diagramme est celui qui communique efficacement. L'efficacité de communication (efficacité cognitive) est mesurée par la rapidité, la facilité et la précision avec lesquelles le contenu de l'information peut être compris.

Le travail fondamental dans le domaine de la communication graphique est celui de Jacques Bertin dans la *Sémiologie des graphes* [Bertin 1983]. Bertin identifie huit variables visuelles élémentaires qui sont utilisées pour l'encodage de l'information (figure 4.3). Celles-ci sont classées en variables planaires (les deux dimensions spatiales) et les variables rétiniennes (les propriétés de l'image).

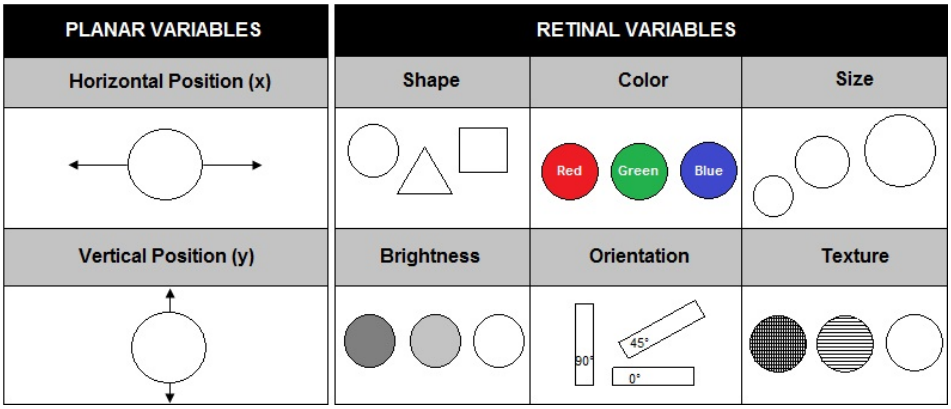


FIGURE 4.3 – Les variables visuelles

L'ensemble des variables visuelles définissent un *vocabulaire* pour la communication graphique : un ensemble de bloc de constructions atomiques qui peuvent être utilisées pour construire n'importe quelle représentation graphique. Les différentes variables

visuelles sont appropriées pour encoder les différents types d'informations. Le choix des variables visuelles a un impact majeur sur l'efficacité cognitive, car il affecte à la fois la vitesse et la précision de l'interprétation [Cleveland 1984, Lohse 1993, Winn 1991].

4.2 Classification des langages visuels

Récemment, des approches basées sur la classification des langages visuels ont vu le jour afin de guider dans le choix des approches adaptées à la réalisation d'outils interactifs, d'établir les visualisations les plus viables pour les domaines complexes, d'évaluer les compromis entre la complexité algorithmique et la puissance expressive lors de l'adoption d'un outil formel pour la définition langage visuel [Bottoni 2004]. Cette classification a été réalisée comme suit.

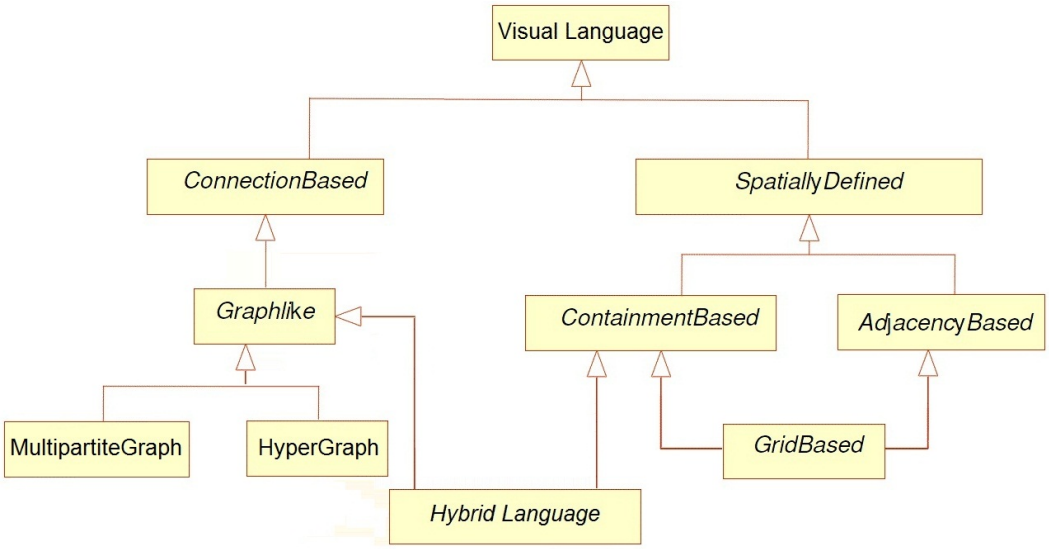


FIGURE 4.4 – Classification des langages visuels

La figure 4.4 montre les relations entre les différentes familles de langages visuels étudiés en [Bottoni 2004]. Cette classification nous a permis de distinguer les différents types de diagramme utilisés dans des domaines variés comme la géométrie, l'économétrie, le génie logiciel, analyse des systèmes, etc.

Les langages basés sur les connections (Connection-based languages) sont parmi les langages visuels les plus utilisés. La représentation explicite des connexions exprime rapidement l'existence de relations sémantiques entre les éléments. Les connections ont deux caractéristiques : la direction et le poids qui peut signifier la distance, le cout ou simplement la multiplicité. Les graphes sont considérés comme des langages basés sur les connections. Cette catégorie est utilisée dans divers domaines, comme les mathématiques, les cartes géographiques (figure 4.5) et même dans la structuration des big data (fichiers balisés) et algorithmes de fouille de données (DFS, Dijkstra, etc.).

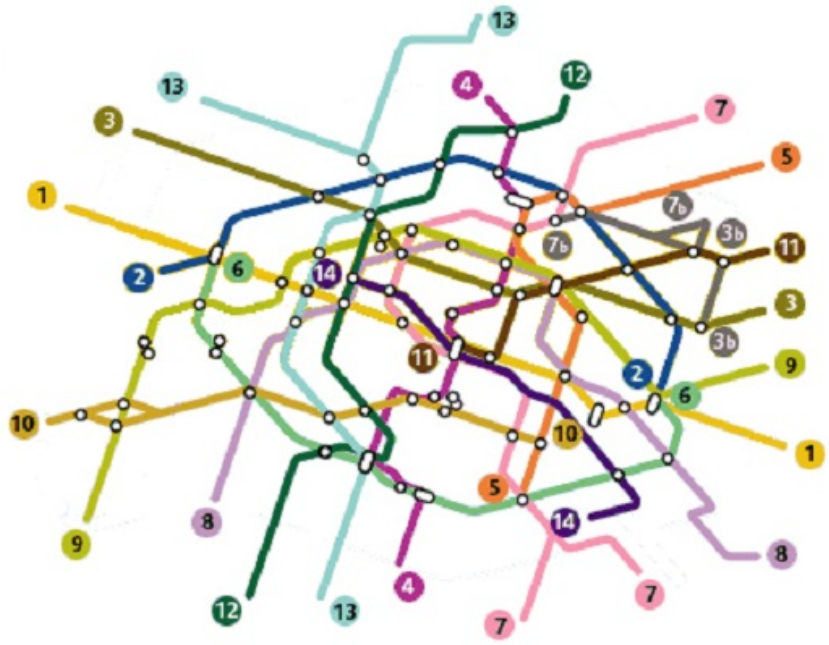


FIGURE 4.5 – Un graphe représentant le réseau de métro de Paris

Un graphe G se compose du tuple (V, E) et défini par l'ensemble d'éléments V appelés sommets (Vertices en anglais), et par l'ensemble d'éléments E appelés arêtes (Edges en anglais). Une arête e de l'ensemble E est définie par une paire non ordonnées de sommets appelés les extrémités de e. Si l'arête e relie les sommets a et b, on dira que ces sommets sont adjacents ou incidents avec e ou bien que l'arête e est incidente avec les sommets a et b. On appelle ordre d'un graphe le nombre de sommets v de ce graphe. Voici quelques terminologies :

Si on peut dessiner un graphe G dans le plan sans qu'aucune arête n'en coupe une autre, on dit que G est planaire.

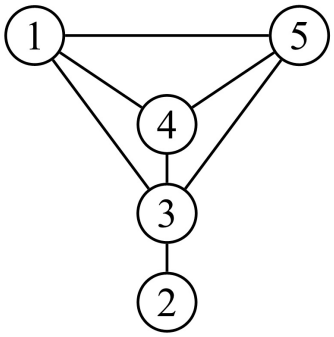


FIGURE 4.6 – Graphe planaire

Un graphe est simple si au plus une arête relie deux sommets et s'il n'y a pas de boucle sur un sommet. On peut imaginer des graphes avec une arête qui relie un sommet à

lui-même (une boucle), ou plusieurs arêtes reliant les deux mêmes sommets. On appelle ces graphes des multigraphes.

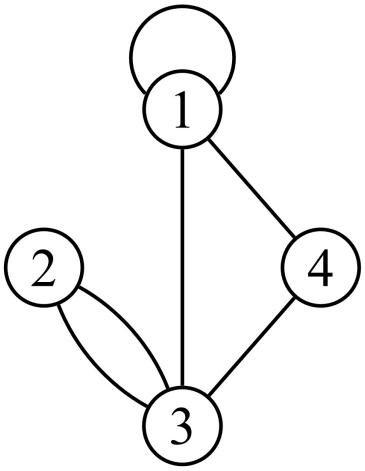


FIGURE 4.7 – Multi-graphe

Un graphe est connexe s'il est possible, à partir de n'importe quel sommet, de rejoindre tous les autres en suivant les arêtes. Un graphe est complet si chaque sommet du graphe est relié directement à tous les autres sommets.

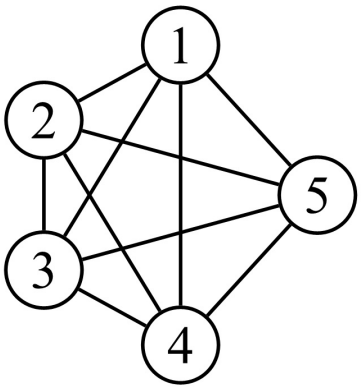


FIGURE 4.8 – Graphe complet

Un graphe est biparti si ses sommets peuvent être divisés en deux ensembles X et Y, de sorte que toutes les arêtes du graphe relient un sommet dans X à un sommet dans Y ou vice versa.

Un hypergraphe H = (V, E) est la donnée de deux ensembles V et E. L'ensemble V, comme dans le cas d'un graphe, est l'ensemble des sommets de H. Par contre, E est appelé hyper-arête, il ne relie plus un ou deux sommets, mais un nombre quelconque de sommets (compris entre un et le nombre de sommets de l'hypergraphe).

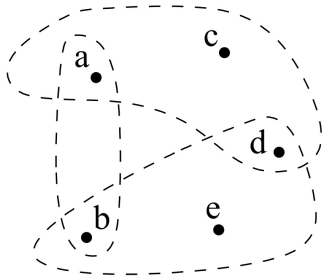


FIGURE 4.9 – Hypergraphe

Les langages basés sur la composition (containment-based languages) incluent les diagrammes de types composé-composant, comme le digramme Euler-Venn [Wiki 2013] et une grande partie des diagrammes UML. Ils sont généralement basés sur le patron *Composite*, dans lequel un élément peut contenir plusieurs éléments, y compris les composés à leur tour.

Plusieurs types de langages visuels n'ont pas besoin d'éléments spéciaux pour établir l'existence d'une relation. Un exemple typique est les langages basés sur l'adjacence (adjacency-based) dans lesquels les éléments sont composés dans *un puzzle*. Leur contiguïté définit la possibilité de transition d'une cellule à une autre. D'autre part, les éléments peuvent être positionnés dans des positions relatives différentes. La disposition d'éléments peut également être considérée comme signifiante. En règle générale, la définition de la disposition (Layout) d'une interface est basée sur cet ordre de considération. Par exemple, nous sommes habitués à interpréter un label à gauche d'un champ de texte (textbox) comme une indication de la signification de ce champ de texte. Les langages de représentation de jeu sont des cas réels de cette problématique, ils considèrent souvent les compositions, chevauchements, ou disjonctions des éléments graphiques comme des relations significatives [Bottoni 2004].

Les langages visuels qui définissent des grilles, tableaux ou matrices, combinent généralement l'adjacence et la composition dans la spécification des cellules pour indiquer respectivement les possibilités de communication et la présence de l'information dans la bonne position.

Les langages hybrides sont basés sur l'existence simultanée de plusieurs relations spatiales et règles de composition. Dans le génie logiciel, la plupart des langages visuels sont hybrides, si on considère par exemple l'utilisation des labels (attachés ou contenus dans les éléments graphiques) comme un moyen additionnel de fournir des informations sur ces éléments. On peut penser que pour avoir un lien de composition entre éléments graphiques, il faut que le composant soit englobé par le composé. Or, dans les langages hybrides il est commun qu'une connexion entre deux éléments graphiques, soit considérée comme une composition. Un des exemples qui existent est le langage UML dont les connections *Association* peuvent être utilisées pour indiquer le lien entre les éléments et la relation de composition entre eux.

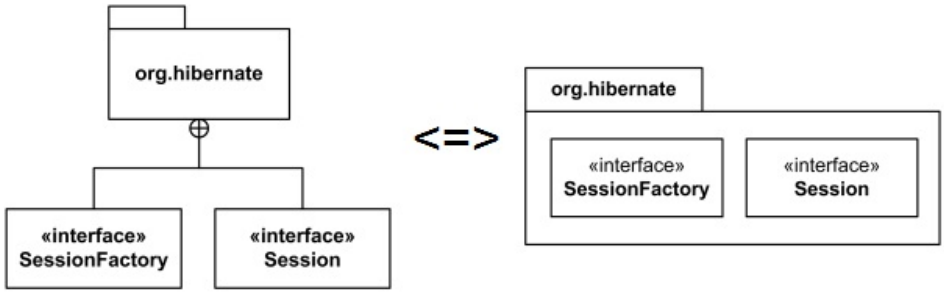


FIGURE 4.10 – Différentes représentations de la relation de composition

4.3 Synthèse et discussion

Les syntaxes concrètes (CS) d'un langage fournissent à l'utilisateur un ou plusieurs formalismes pour manipuler les concepts de la syntaxe abstraite et ainsi créer des instances du métamodèle, c'est-à-dire des modèles. Nous avons présenté dans ce chapitre, les concepts de base de cette syntaxe ainsi qu'une classification des différents langages visuels qui existent. Dans le chapitre suivant, nous présentons les principaux outils qui permettent de spécifier les éditeurs de la syntaxe concrète graphique d'un langage ainsi que d'une évaluation de ces outils selon les critères et les besoins qui nous intéressent dans notre thème de recherche.

CHAPITRE 5

Approches de spécification des éditeurs de diagrammes

Sommaire

Le présent chapitre donne un bref aperçu des observations basées sur notre expérience avec les outils de spécification des éditeurs graphiques de diagrammes. Nous tenons à souligner que tous les outils utilisés dans cette évaluation sont utilisés sur une grande échelle et ont une notoriété dans la communauté des langages visuels et du génie logiciel. Chacun de ces outils fournit des concepts et des méthodologies innovantes dans ce domaine. Le processus de la définition et de l'outillage d'une syntaxe concrète est assez couteux du fait qu'il nécessite de se familiariser avec chacune des techniques dédiées et connaitre les Frameworks spécifiques pour leur implémentation (Par exemple, Antlr pour parser du texte, le patron visiteur pour traduire le modèle vers du texte et le MVC pour synchroniser le modèle avec la représentation, les technologies tels SWT et GEF pour définir l'interface utilisateur (UI), etc.). Les spécialistes du domaine se sont penchés sur la question et ont construit des langages et outils dédiés pour cette partie de métamodélisation. La définition d'une syntaxe concrète et plus précisément la syntaxe concrète graphique est à ce jour bien outillée.

Le processus général pour la génération automatique d'un éditeur de syntaxe concrète est assez générique. L'éditeur sera construit à partir des constructions du domaine spécifique (le métamodèle représentant la syntaxe abstraite), d'une description de la syntaxe concrète (modèle ou code) décrivant les représentations textuelles ou graphiques et d'une description des liens entre les deux syntaxes précédentes. Certains outils proposent de générer un éditeur à partir du modèle du domaine. Cette pratique permet d'avoir un point de départ pour ensuite adapter l'éditeur en retouchant les différents modèles. Le plus simple des éditeurs de syntaxe concrète graphique est l'éditeur arborescent [Jezequel 2012]. Il est fourni par défaut avec des technologies telles que EMF de Eclipse. S'appuyant sur la structure de contenance des éléments du métamodèle pour créer un arbre, il peut être complètement dérivé du métamodèle. Cet éditeur sert généralement de base pour construire une vue outline (model explorer) pour d'autres éditeurs plus sophistiqués (textuelles ou diagrammatiques).

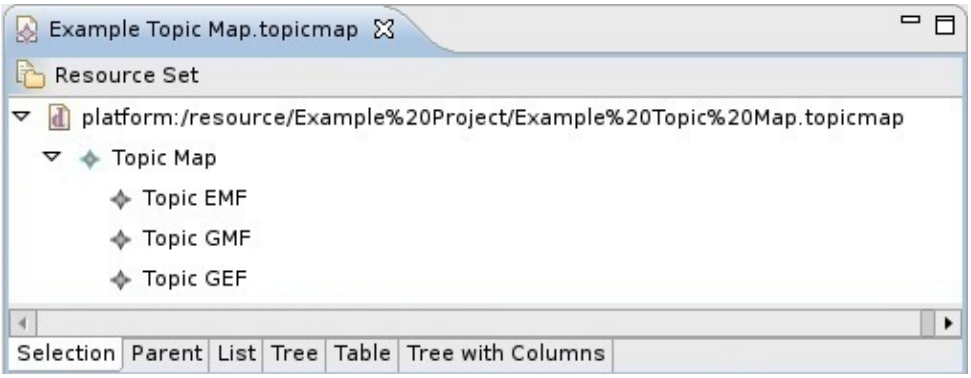


FIGURE 5.1 – Éditeur arborescent généré par EMF

Certains langages de modélisation préfère des éditeurs textuels pour manipuler leurs modèles à cause de l'expressivité de ceux-ci (figure 5.2). Grace à la définition d'un modèle de correspondance entre la syntaxe concrète textuelle et les concepts des métamodèles, des outils tels EMFText et Xtext permettent de générer automatiquement le code des différents éléments d'un éditeur textuel. Ces outils permettent d'obtenir des parsers qui permettent de traduire le texte en format XMI. De plus, ils permettent de bien représenter les éléments des concepts avec une meilleure mise en valeur (highlight) souvent en les représentant en gras ou en couleur, cette fonctionnalité souvent appelé pretty-printer permet de traduire les données du fichier XMI vers du texte bien structuré.

Tout comme les éditeurs arborescents, une bonne pratique consiste à générer une préversion utilisant une syntaxe générique comme HUTN (Human Usable Textual Notation) de l'OMG [OMG 2004], puis de la faire évoluer vers la syntaxe envisagée.

La représentation diagrammatique (éditeurs de diagrammes) quant à elle, est souvent celle qui est préférée des utilisateurs. Elle permet une bonne réflexion en phase amont du processus de développement. Toutefois, les compétences requises pour créer ce genre d'éditeurs freinent souvent le développement.

Il existe en effet de nombreux projets qui s'y consacrent et qui suivent différentes façons pour spécifier les éditeurs de diagrammes. Nous les avons classifiés selon différentes

FIGURE 5.2 – Éditeur textuel du langage Ecore

catégories d'outils :

1. Spécification par code. Comme c'est le cas pour GEF et ses dérivés (draw2d touch, GEF3D), UMLet [Auer 2003] ou Graphiti[SAP 2011]. Cette catégorie est la plus utilisée, elle permet de construire des éditeurs graphiques à partir d'une description programmatique (code) ;
2. Spécification basée sur des langages de méta-description. Cette catégorie d'outils utilise des méta-langages de description pour spécifier les diagrammes, c'est le cas de MetaEdit+, qui utilise le langage de métamodélisation GOPPRR [Kelly 1996] et GME [Ledeczi 2001a], basé sur une méta-description (Meta-GME) pour définir les syntaxes abstraites et concrètes d'un domaine spécifique ;
3. Spécification basée sur la grammaire des graphes. Ce genre d'outil est basé sur la définition de la grammaire de graphe et permet de spécifier des éditeurs de diagrammes à partir de celle-ci (par exemple Visual Diagen [Minas 1995], $AToM^3$ [Lara 2002], VLDesk [Costagliola 2002], etc.) ;
4. D'autres outils permettent la spécification de dessinateurs de diagrammes/charts, ils sont exclusivement graphiques c'est-à-dire qu'ils ne se préoccupent que de l'aspect graphique sans associer les éléments du diagramme à une syntaxe abstraite (par exemple, Microsoft Visio ou Dia).

Il existe cependant, d'autres méthodes et outils, qui ont récemment adopté des approches dirigées par les modèles. On distingue deux grandes catégories dans cette méthode de spécification :

- Outils basés sur les profils UML : Certains outils de modélisation UML permettent de spécifier des éditeurs de diagramme pour les profils UML comme Papyrus [Gérard 2011], IBM RSA[IBM 2011] et MagicDraw[Silingas 2008]. Ils permettent ainsi de créer des représentations visuelles pour les méta-classes des profiles (mécanisme des stéréotypes UML) en réutilisant la syntaxe concrète d'UML ;
- Outils basés sur un langage spécifique de modélisation (DSML). Les langages spécifiques de modélisation (DSML) sont plus spécifiques aux exigences du domaine, ils offrent aux utilisateurs des concepts propres à leur métier et leur savoir-faire. Spray [Itemis 2012], GMF [Eclipse 2006b], TopCased et Obeo Designer [Juliot 2010] sont des exemples de ce type de spécification. Ils permettent de décrire (via des modèles) la syntaxe concrète et de l'associer à un métamodèle spécifique à un

domaine. Ce genre de technologie, notamment GMF, est celle utilisée dans papyrus pour plusieurs raisons (facilité de gérer un modèle par rapport au code, technologie open-source, facilité d'intégration avec l'écosystème Eclipse, etc.). Cependant, ces outils nécessitent encore des interventions programmatiques pour l'adaptation. De nombreuses autres limitations sont détectées, principalement en termes de faiblesses de réutilisation et de rigidité des outils [El-kouhen 2011, Amyot 2006].

Nous avons évalué un échantillon représentatif d'outils dans chaque catégorie mentionnée ci-dessus avec les besoins que nous avons soulevés dans notre problématique. Nous avons choisi dans cette étude, les outils qui offrent les caractéristiques d'un outil Meta-Case (c'est-à-dire qui permettent de spécifier et de générer des outils de modélisation). Nous avons choisi le Framework GEF et Graphiti pour représenter la catégorie de spécification par code, MetaEdit+ et GME pour la catégorie de spécification par langages de méta-description, $ATOM^3$ pour la catégorie de spécification par grammaire des graphes et DIA pour représenter les dessinateurs graphiques. Pour les outils basées sur une approche IDM, nous avons choisi IBM RSA pour représenter les outils supportant le mécanisme d'extension UML (profile) utilisant les stéréotypes et finalement nous présentons un large éventail d'outils basés sur des DSML comme GMF, Obeo Designer (Eclipse Sirius) et le projet Spray.

5.1 Spécification par code

Ce type de spécification est le plus classique et historiquement le plus commun dans le développement des outils de modélisation. Des bibliothèques java comme le Framework d'édition graphique (GEF), Draw2D Touch et GEF 3D sont la base de plusieurs éditeurs de diagrammes comme TopCased, IBM RSA, MagicDraw et d'autres. Toutes ces bibliothèques et outils permettent la spécification des éditeurs de diagramme en décrivant plusieurs concepts comme la notion de noeuds, liens, palette et d'autres paramètres comme la couleur, la taille, les bordures, etc. Ces éléments communs dans toutes les technologies de spécifications de diagrammes sont les concepts de bases de notre proposition. Hormis la difficulté de l'implémentation des diagrammes complexes, le temps pour étudier/apprendre ces bibliothèques, le risque d'erreur et d'incohérence dans le codage et la difficulté de maintenance et d'évolution des éditeurs générés, les outils cités dans cette catégorie bénéficient des bonnes pratiques en patrons de conception : architecture MVC, la possibilité d'extension, etc.

5.1.1 GEF

Le Framework d'édition graphique (GEF) fourni un cadre technologique pour créer des applications riches de type éditeurs graphique sous l'environnement Eclipse. Il est composé de trois composants :

1. Draw2d : c'est une boite à outils de mise en page et de rendu graphique en java. Elle se base sur SWT pour les figures, les écouteurs (listener) d'événements et la mise en page dans ce qu'on appelle un canvas. Les figures peuvent être composées

par une relation Parent/fils (composé/composant). Chaque figure a une bordure rectangulaire, sur laquelle, les éléments fils sont dessinés. Un gestionnaire de disposition (Layout Manager) peut être utilisé pour placer les éléments fils selon leurs indexes et/ou contraintes [GEF 2013]. Les figures, les gestionnaires de disposition et les bordures peuvent être combinés et imbriqués pour créer des figures plus complexes et répondre ainsi à tous les besoins. Choisir la bonne combinaison de figures et de Layouts pour créer l'effet désiré peut s'avérer une tâche délicate et fastidieuse [Lee 2003].

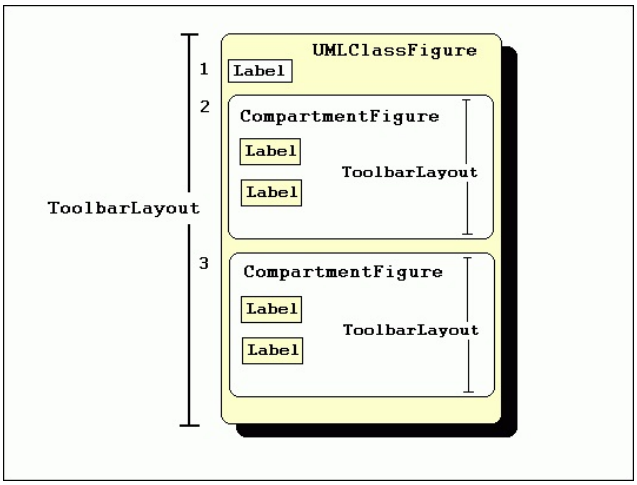


FIGURE 5.3 – Structure d'une Classe UML

Par exemple pour concevoir une classe UML avec cette bibliothèque, on doit décomposer la figure en trois parties. Dans cet exemple la composition s'appellera *UMLClassFigure*. Le premier composant est une figure de Label (ceci va afficher le nom de la classe). Les deux autres composants sont des conteneurs pour les attributs et les méthodes de la classe. Dans cet exemple, la figure créée pour cet effet s'appellera *CompartementFigure*. A la fois la figure de la classe et du compartiment, vont utiliser un ToolbarLayout pour placer leurs composants. Le schéma conceptuel de la structure de la classe est illustré dans la figure 5.3.

Le code ci-dessous permet le rendu désiré. La classe *CompartmentFigure* est utilisée pour contenir les méthodes et les attributs de la classe.

```
public class CompartmentFigure extends Figure {
  public CompartmentFigure() {
    ToolbarLayout layout = new ToolbarLayout();
    layout.setMinorAlignment(ToolbarLayout.ALIGN_TOPLEFT);
    layout.setStretchMinorAxis(false);
    layout.setSpacing(2);
    setLayoutManager(layout);
    setBorder(new CompartmentFigureBorder());
  }
}
```

La classe *UMLClassFigure* est assez similaire à la classe *CompartmentFigure*. Elle contient trois composants : deux CompartmentFigures pour les attributs et méthodes et une étiquette (label) pour afficher le nom de la classe. Elle utilise également un ToolbarLayout orienté verticalement pour placer ses enfants. La classe *UMLClassFigure* utilise une ligne de bordure d'épaisseur égale à un pour tracer un cadre autour de ses bords. Le code de la classe *UMLClassFigure* est comme suite :

```
public class UMLClassFigure extends Figure {

  public static Color classColor = new Color(null
     ,255,255,206);
  private CompartmentFigure attributeFigure = new
     CompartmentFigure();
  private CompartmentFigure methodFigure = new
     CompartmentFigure();

  public UMLClassFigure(Label name) {
          ToolbarLayout layout = new ToolbarLayout();
    setLayoutManager(layout);
    setBorder(new LineBorder(ColorConstants.black,1));
    setBackgroundColor(classColor);
    setOpaque(true);
    add(name);
    add(attributeFigure);
    add(methodFigure);
     }
     public CompartmentFigure getAttributesCompartment() {
    return attributeFigure;
  }
     public CompartmentFigure getMethodsCompartment() {
    return methodFigure;
  }
}
```

Draw2d offre aussi la possibilité de créer des connexions ou liens entre deux figures. Pour cela, il est nécessaire d'établir deux extrémités de cette connexion. Ces deux extrémités sont nommées les ancres (anchors) de source et de destination.

2. GEF-MVC : ce composant permet la séparation des préoccupations selon l'architecture MVC (Modèle-Vues-Contrôleurs). Les contrôleurs mappent la vue et le modèle (figure 5.4). Chaque contrôleur, ou EditPart comme on les appelle dans GEF, est chargé de lier le modèle à sa représentation (sous forme d'arbre en utilisant SWT trees ou sous forme de figure en utilisant Draw2d) et d'apporter ainsi les modifications au modèle ;

   L'EditPart observe également le modèle et met à jour la vue pour refléter les changements des états dans le modèle. Ce composant offre aussi la possibilité d'interaction entre les différentes représentations (outline, properties view, etc.). Les EditParts sont les objets avec lesquels l'utilisateur interagit. Ce composant gère donc les interactions de type outils (Tool-Based Interactions) comme la Création, le dépla-

cement, etc. ou les interactions basées sur les actions (Action-Driven Interactions) comme l'alignement, la suppression, etc. [Hudson 2003].

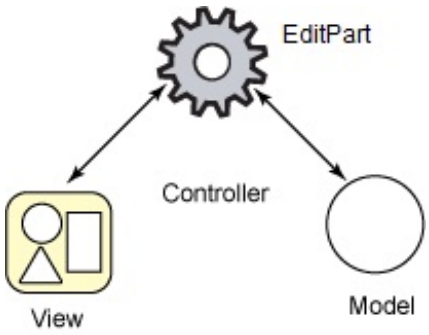


FIGURE 5.4 – Architecture MVC de GEF

3. Zest : c'est une boite à outils de visualisation de graphe sous le Framework GEF. Basée sur Draw2d, elle fournie une implémentation de vues sur la plateforme Eclipse. Développé par l'université de Victoria et le centre d'études avancées d'IBM en 2005, elle rejoint le projet GEF en 2007.

La version GEF4 propose une interface déclarative qui prend en charge la description textuelle en pseudo-code et permet de l'interpréter et d'afficher le rendu dans une autre fenêtre et vice versa.

Plusieurs variantes de GEF sont disponibles à ce jour pour décrire des diagrammes dans des pages web. On peut citer la bibliothèque Draw2d Touch écrite en JavaScript sous licence GPL2 qui utilise des fonctionnalités intégrées du navigateur pour fournir des dessins interactifs et diagrammes sur les pages HTML5. De cette façon, il est possible de manipuler des diagrammes dans des navigateurs web, quel que soit la plateforme technologique (tablette, smartphone, etc.). Il s'agit d'une réécriture complète de l'ancienne bibliothèque Draw2D, en utilisant RaphaelJS [Raphael 2013] et jQuery [Wikipedia 2013g] comme base. Il s'agit d'une solution basée sur du SVG (Scalable Vectorial Graphics) pur.

Ci-dessous un exemple du script java et le rendu dans un navigateur web décrivant un diagramme avec deux éléments et un lien entre eux.

```
var canvas = new draw2d.Canvas("gfx_holder");

// create and add two Node which contains Ports
var start = new draw2d.shape.node.Start();
var end   = new draw2d.shape.node.End();
canvas.addFigure( start, 50,450);
canvas.addFigure( end, 230,150);

// Create a Connection and connect he Start and End node
var c = new LabelConnection();
c.setSource(start.getOutputPort(0));
c.setTarget(end.getInputPort(0));
canvas.addFigure(c);
```

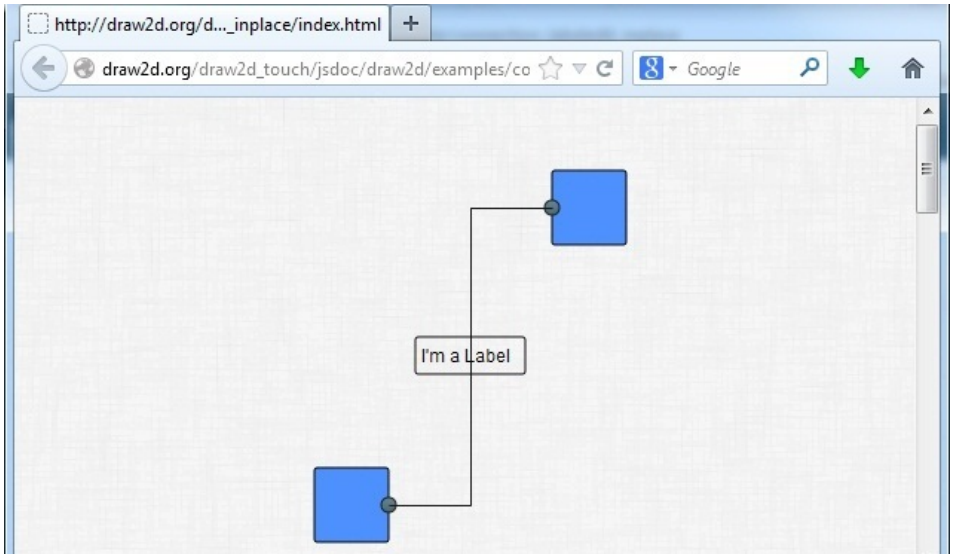


FIGURE 5.5 – Éditeur de diagrammes construit par Draw2d touch

Le Framework GEF3D est une autre extension de GEF qui a fait l'objet d'une thèse en 2008 [Bauer 2008] et qui a pour but de concevoir des diagrammes avec un rendu en trois dimensions. Avec GEF3D on peut créer des diagrammes 3D, des diagrammes 2D ou une combinaison entre les deux.

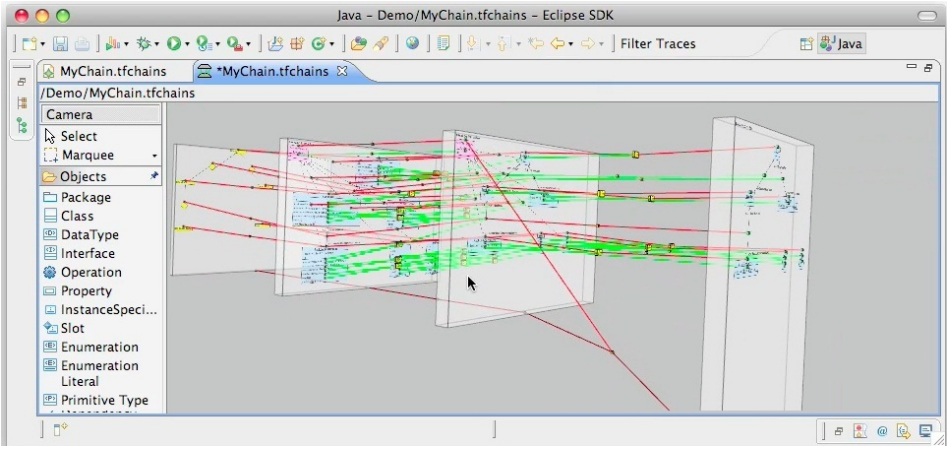


FIGURE 5.6 – Éditeur de diagrammes 3D affichant des connexions inter-modèles

Les éditeurs de diagrammes 2D spécifié avec GEF restent compatibles et facilement intégrables dans GEF3D. Ceci est réalisé en projetant la sortie des éditeurs 2D sur des plans 3D et en comblant l'écart entre le contenu 2D et 3D : les objets 3D peuvent accéder au contenu 2D et vice versa. En outre, les éditeurs multi-diagrammes peuvent être créés en combinant plusieurs éditeurs simples. Par exemple plusieurs diagrammes 2D peuvent être affichés et modifiés simultanément sur différents plans et des éléments 3D peut être utilisée pour afficher les connexions inter-modèles (figure 5.6)[Eclipse 2008].

5.1.2 Graphiti

Un autre projet Eclipse utilise une description par code, c'est le cas du Projet Graphiti (Graphical Tooling Infrastructure) [SAP 2011]. Ce projet considéré comme évolution de GMF/GEF a été créé par SAP en 2009 et donné à la communauté Eclipse en Octobre 2010. Il consiste à décrire les diagrammes d'une manière déclarative (des annotations) pour qu'un moteur d'interprétation et de rendu, puissent afficher le diagramme à la volée. Cela permet l'affichage d'un diagramme au sein d'autres environnements qu'Eclipse (par exemple, l'utilisation du flash dans un navigateur) simplement en fournissant un nouveau moteur de rendu pour l'environnement qui peut interpréter le code déclaratif Graphiti. (Actuellement seul un moteur de rendu pour Eclipse GEF existe).

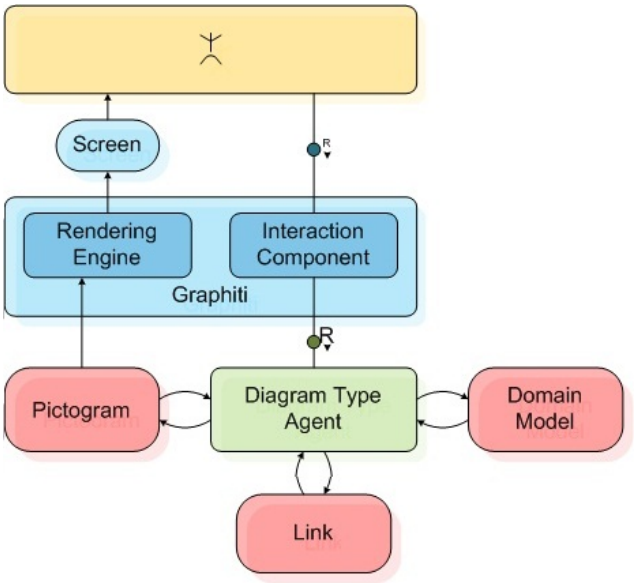


FIGURE 5.7 – Architecture du Framwork Graphiti - Copyright 2011 SAP

Graphiti est strictement autonome et indépendant des technologies de rendu. L'utilisateur du Framework spécifie ses diagrammes en code java pur, il n'a besoin de connaitre le Framework EMF pour créer un éditeur. Aucune connaissance de Draw2D, GEF ou autres Frameworks de rendu n'est nécessaire. Toutes les interactions avec un éditeur Graphiti sont interceptées par le composant d'interaction (Interaction Component), ce composant traite ces requêtes et les achemine à l'agent des types du diagramme (Diagram Type Agent). Ce dernier qui joue le rôle du contrôleur dans l'architecture MVC de Graphiti, permet de créer/modifier les données dans modèle de domaine (Model), créer la visualisation graphique de l'élément (View) et de créer le lien entre les pictogrammes et les objets du modèle de domaine.

Toutes les choses complexes dans GEF comme les EditPolicy, requêtes et commandes sont considérés dans Graphiti comme des Features. Toute chose est Feature dans la spécification de diagramme dans ce Framework. Les features sont manipulées (ajoutés, supprimés et mis à jour) selon la stratégie appropriée (c'est-à-dire lorsque la synchro-

nisation entre le domaine et le pictogramme doit être faite) par l'agent des types du diagramme suite à la requête interceptée par le composant d'interaction. Toutes les features doivent être implémentées par le développeur de l'éditeur. Les éditeurs Graphiti ne sont donc que les interpréteurs des features et types de diagramme déclarées.

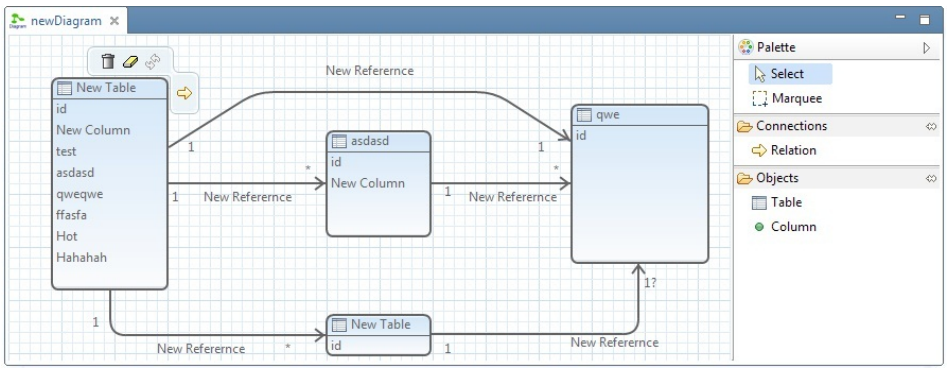


FIGURE 5.8 – Éditeur de diagrammes construit avec Graphiti

Hormis les avantages de créer des features comme la réutilisation, le coût de développement d'un éditeur sous Graphiti est assez grand. Pour chaque concept du métamodèle (syntaxe abstraite), le développeur doit implémenter les features et leurs méthodes (Add, Create, Update), les interactions (remove, delete, move, layout, etc.), l'enregistrement de ces features dans le FeatureProvider et la configuration de la palette dans la classe ToolBehaviourProvider. Ceci est assez fastidieux dans le cas où le métamodèle à manipuler est d'une taille considérable.

5.2 Spécification par langages de méta-description

5.2.1 Generic Modeling Environment (GME)

GME (Generic Modeling Environment) est un outil metaCase configurable développé à l'université de Vanderbilt en C++. Il propose une boite à outil pour la création d'environnements de DSML. La configuration est effectuée par la spécification de ce qu'on appelle « un paradigme de modélisation » qui représente le langage spécifique au domaine. Ceci est fait en utilisant le langage de description MetaGME (GMeta). Hormis la syntaxe abstraite du domaine, un paradigme de modélisation comporte aussi les informations de représentation (syntaxe concrète) liées au domaine [Ledeczi 2001a].

Un projet contient un ensemble de dossiers (Folders). Les dossiers sont des conteneurs de modèles. Les Modèles, les Atoms, les Références, les connexions et les ensembles (Sets) sont tous des objets de première-classe ou FCO (First-Class Object). Les Atoms sont des objets élémentaires (ne peuvent contenir de Part). Chaque type d'Atom possède un ensemble d'attributs et est associé à une icône. Les valeurs des attributs peuvent être changées par l'utilisateur.

Les Modèles sont les composants de base de ce Framework. Ils peuvent contenir des Parts et une structure interne. Un Part est un conteneur de modèle qui a toujours un Rôle.

Le paradigme de modélisation détermine quel type de Part est autorisé dans un modèle sur un Rôle. La relation Containment crée une décomposition hiérarchique des modèles. Si un modèle contient un autre modèle du même type, la profondeur de la hiérarchie est théoriquement infinie. Chaque objet doit avoir au plus, un parent et celui-ci doit être un modèle. Il existe au moins un modèle qui n'a pas de parent. Il s'appelle « modèle de racine ». Les Aspects fournissent avant tout un contrôle de visibilité. Chaque Modèle a un ensemble prédéfini d'aspects. Chaque Part a un ensemble d'aspects primaires qui peuvent être crées ou supprimés.

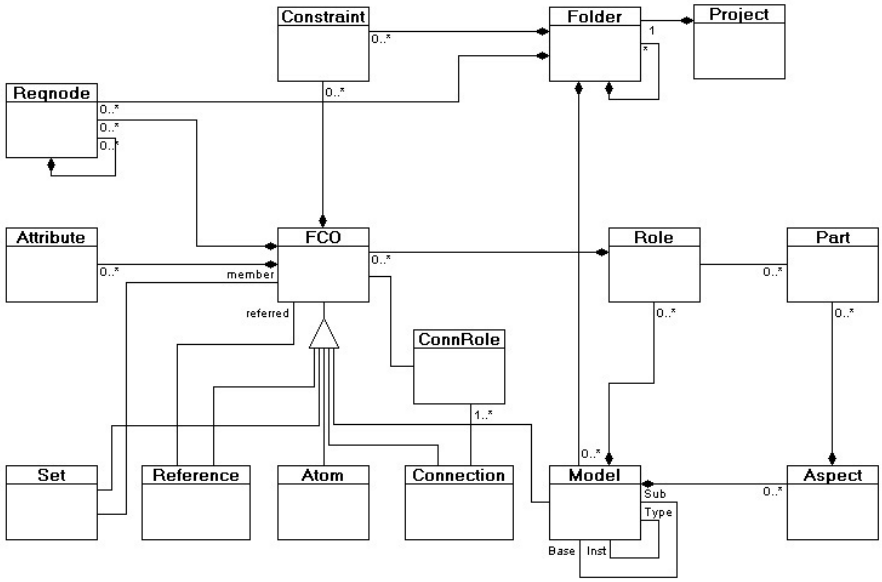


FIGURE 5.9 – Concepts du langage MetaGME (GMeta)

Pour définir une relation entre deux objets dans GME, on utilise le concept Connection. Une connexion possède des attributs et peut être dirigé ou non. Pour définir une relation entre deux objets, ils doivent être dans la même hiérarchie et être visibles dans le même Aspect. Les connexions peuvent en plus être limitées par des contraintes explicites en spécifiant leurs multiplicités. Une connexion ne peut exprimer que la relation entre deux éléments contenus dans le même modèle. Notez bien que le modèle de racine, ne peut participer à aucune connexion. Les connexions et les références sont des relations binaires. Les ensembles (Sets) peuvent être utilisés pour spécifier une relation entre un groupe d'objets. La seule restriction est que tous les membres de l'ensemble doivent avoir le même conteneur (Parent) et doivent être visible dans le même Aspect. Les dossiers (Folders), les FCO (Modeles, Atoms, Sets, Références, Connections), les Rôles, les Contraintes et les Aspects sont les concepts principaux utilisés pour définir un paradigme de modélisation (figure 5.9). En d'autres termes, le langage de modélisation est construit avec des instances de ces concepts. Le choix des ces concepts génériques est certainement, la décision de conception la plus critique. Les modèles de GME sont similaires aux classes en java. Quand un modèle particulier est créé dans GME, il devient un type (Classe). Il

peut être étendu et instancié autant de fois que l'utilisateur en a besoin [Ledeczi 2001a].

Ce Framework supporte la réutilisation et la maintenance de modèles, puisque chaque changement dans un type, se propage automatiquement dans sa descendance hiérarchique. Il est aussi possible de créer des librairies de modèles type qui peuvent être utilisés dans diverses applications pour un domaine donné.

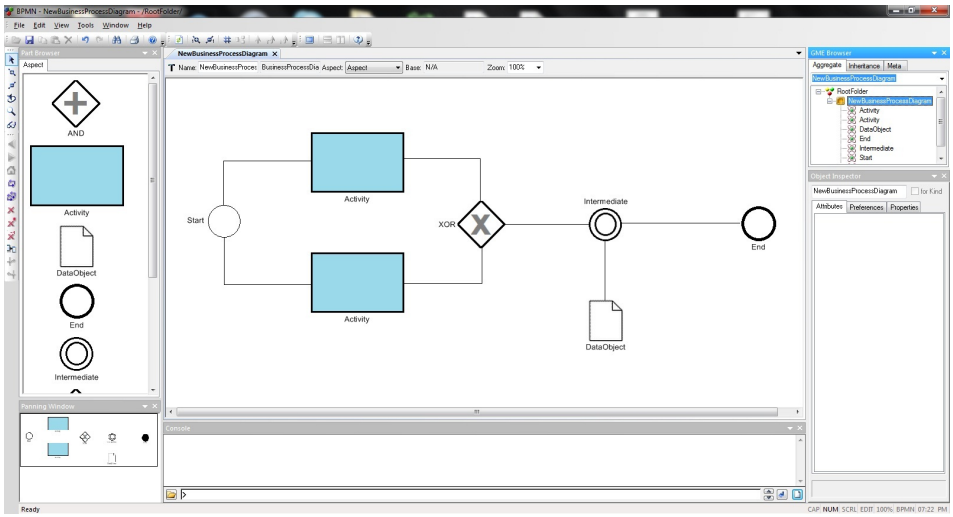


FIGURE 5.10 – Éditeur graphique construit dans GME

Une fois le paradigme créé, on peut associer des décorateurs (représentations graphiques) aux concepts du domaine et les enregistrer dans GME en tant qu'éditeur tel que montré dans la figure 5.10. L'outil propose plusieurs fonctionnalités : le chargement et la sauvegarde sous format XMI, undo/redo, le drag and drop, la validation du modèle à travers les multiplicités et contraintes OCL, l'impression, les vues d'ensemble (overviews), les vues de propriétés, etc. Nous pouvons également utiliser le mécanisme d'aspects pour créer des perspectives (points de vue) qui limitent la vision des concepts du métamodèle. Le projet est très bien documenté [ISIS 2011]. Cependant, nous avons constaté qu'il est difficile de créer des styles personnalisés pour les liens. Pour les éléments de diagrammes, nous ne pouvons pas créer des formes complexes en dehors des représentations en image bitmap.

GME a une architecture modulaire et extensible qui utilise les interfaces COM pour l'intégration [Wikipedia 2013a]. Des composants externes peuvent être intégrés à condition qu'ils soient écrits dans un langage qui prend en charge COM (C++, Visual Basic, C#, Python, etc.). GME dispose d'un gestionnaire de contraintes qui applique toutes les contraintes de domaine dès les premières étapes de la construction du modèle. GME propose aussi la composition du métamodèle pour la réutilisation et la combinaison de langages de modélisation existants et les concepts des nouveaux langages. Il prend en charge les bibliothèques de modèles pour la réutilisation au niveau du modèle via l'héritage. Il est aussi possible de créer des transformations de modèles dans cet environnement, mais au prix d'une lourde programmation en C++. La syntaxe concrète du langage est personnalisable grâce aux décorateurs associés aux concepts [Davis 2003, Ledeczi 2001a].

Cependant, nous avons constaté d'importantes limitations dans cet outil au niveau de la spécification de cette syntaxe : Le mélange de la définition de la syntaxe abstraite et la syntaxe concrète, la difficulté de réutiliser la syntaxe concrète seule (c'est-à-dire réutiliser les représentations graphiques sans réutiliser les concepts de la syntaxe abstraite associés) et l'impossibilité de créer des diagrammes complexes puisque l'outil ne supporte que des représentations sous forme d'images.

Un autre inconvénient dans ce type d'environnement, est le temps et l'effort fournis pour l'étude et la compréhension du langage Méta-GME.

5.2.2 MetaEdit+

MetaEdit+ est un environnement Meta-CASE développé à l'université de Jyväskylä en Finlande et fait partie du projet MetaPHOR [MetaPHOR 1991] qui a pour but la construction et l'utilisation des solutions Domain-Specific Modeling (DSM). MetaEdit+ fourni un ensemble de fonctionnalité pour créer des éditeurs graphiques pour les langages spécifiques et les intégrer avec d'autres outils via son open API.

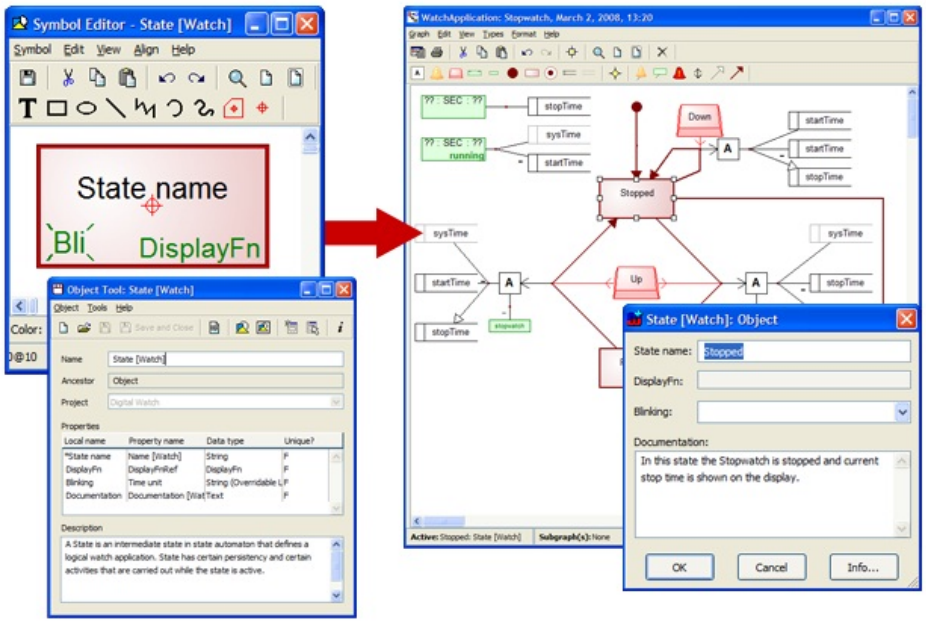


FIGURE 5.11 – Environnement MetaEdit+ - Copyright 2011 MetaCase

Tout comme GME, MetaEdit+ est basé sur un langage de métamodélisation propriétaire. Ce langage s'appelle GOPPRR [Kelly 1996]. GOPPRR est l'acronyme formé par les types de base du langage qui sont les Graphes, les Objets, les Ports, les Propriétés, les Relations, et les Rôles. Le Graphe est la structure de haut niveau du métamodèle. Elle définie un langage ou diagramme. La sémantique réelle du Graphe est définie comme la liaison d'Objets, les Relations, les Rôles et les Ports dans le Graphe. Les Propriétés caractérisent les attributs qui peuvent être attachées à chacun de ces autres types [Pohjonen 2005]. Un Graphe (équivalent au concept de « Model » en GME) désigne un concept global qui

contient un ensemble d'objets et leurs relations, avec des rôles spécifiques. Par exemple, un Graphe serait équivalent à l'élément package dans un digramme de Classe UML. Pratiquement, le concept de Graphe est fondamentalement un concept de décomposition généralisé : il peut être inclus dans un Graphe parent, attaché à un Objet ou un rôle, etc. [Kelly 1996]

Le processus de construction des outils de modélisation en MetaEdit+ est assez simple. Nous commençons par concevoir le langage de modélisation avec sa syntaxe concrète dans un dessinateur WYSIWYG (MetaEdit+ Workbench) de SVG semblable à Microsoft Paint et par la suite nous produisant l'éditeur manipulant ce langage dans MetaEdit+ Modeler (figure 5.11).

MetaEdit+ Workbench est un outil pour la conception de langages de modélisation : les concepts, les règles, les notations et les générateurs graphiques. La définition du langage est stocké comme un métamodèle dans le référentiel de MetaEdit+. MetaEdit+ Modeler suit la définition du langage de modélisation de données défini précédemment dans MetaEdit+ Workbench par extraction à partir du référentiel et propose automatiquement, toutes les fonctions des outils de modélisation : les éditeurs de diagramme, les vues de propriétés, les générateurs de code, une fonctionnalité pour le travail collaboratif (multiutilisateur), etc.

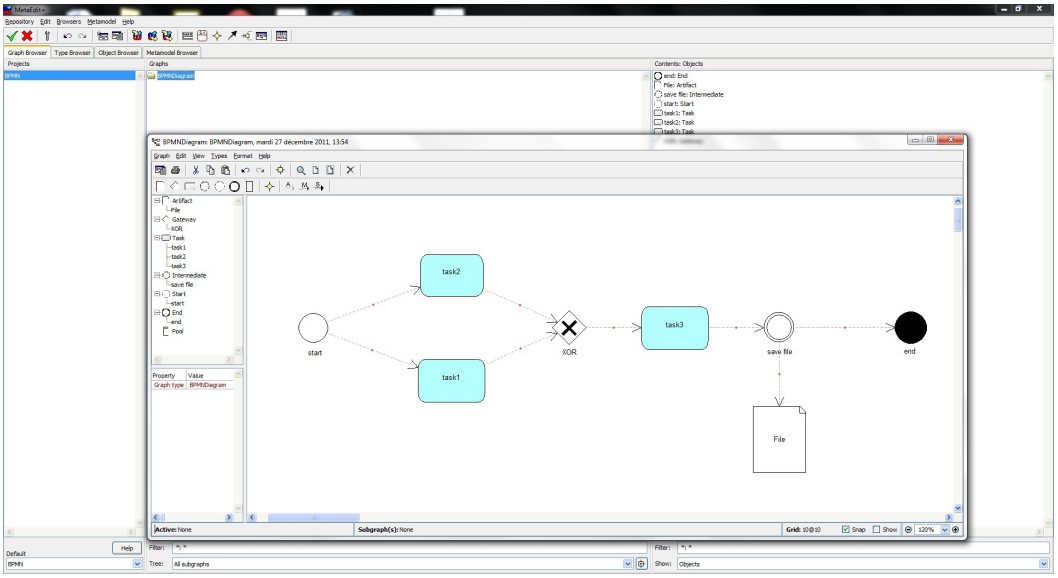


FIGURE 5.12 – Éditeur BPMN produit par MetaEdit+

Une fois le projet créé et enregistré comme un métamodèle dans le référentiel MetaEdit+ il peut être utilisé comme un éditeur, comme le montre la figure 5.12. L'outil offre de nombreuses fonctionnalités telles que le chargement/sauvegarde des diagrammes en format XML personnalisé, undo / redo, l'impression, l'export de diagrammes en bitmap, GIF, PNG, l'import/export de représentations graphiques de/vers SVG, zoom, une vue de propriétés, des explorateurs de modèles, un outil de définition de contrainte et

règles pour le modèle. Il offre également le choix de passer d'une représentation à une autre (diagramme, tableau/matrice) sans rechargement ou régénération. Nous avons été étonnés de la rapidité et la convivialité offerte par cet outil, notamment l'éditeur de syntaxe concrète WYSIWYG. Cependant, nous avons trouvé une limitation dans les éditeurs produits comme la difficulté de déplacer les étiquettes (labels) ou de renommer des éléments graphiques sans utiliser le menu contextuel de la propriété, etc. mais cela est dû à la nature des représentations graphiques qui sont en SVG (Scalable Vectorial Graphics).

Tout comme GME, le langage de métamodélisation GOPPRR a pour but de supporter la réutilisation et la maintenance des modèles : tout changement dans le Graphe conceptuel (modèle sémantique) est propagé dans les différents Graphes représentationnels. Cependant, malgré tout l'intérêt de cet outil, nous avons constaté des limitations au niveau de la réutilisation de la syntaxe concrète. Ceci est dû au mélange de la syntaxe concrète et du modèle sémantique (tous les deux sont spécifiés dans le même référentiel) ce qui affaiblit les possibilités de réutilisations du langage visuel sur d'autres applications. Nous avons aussi constaté, que les éditeurs de digrammes générés sont intégrés dans le même environnement de développement avec la même visualisation (disposition des vues et fenêtres). Ceci montre une forte dépendance à la technologie utilisée et à l'environnement de développement (le langage GOPPRR, le référentiel, etc.). Malgré la facilité de définition de la syntaxe concrète avec l'éditeur WYSIWYG fourni par MetaEdit+, on ne peut pas spécifier des diagrammes complexes (diagramme de séquence UML par exemple). Cela peut être expliqué par l'utilisation des images SVG dans la définition de cette syntaxe.

Une des limitation qui peut être problématique pour les utilisateurs industriels, est le contrôle effectué par les éditeurs logiciels sur ce genre d'outil. La même situation est rencontrée dans la recherche, qui dépend typiquement des outils open source. Les outils de méta-description sont assez contraignants et inflexibles pour permettre une implémentation optimale des nouvelles idées et prototypes.

5.3 Approches basées sur la grammaire des graphes

Les approches basées sur une grammaire de graphe pour spécifier les langages visuels sont proches des approches des grammaires textuelles. Les grammaires textuelles peuvent être formellement spécifiées en utilisant des formalismes très connues comme BNF (Backus-Naur Form) ou plus communément EBNF (Extended Backus-Naur Form) dans la mesure où la notation EBNF est un standard ISO [ISO/IEC 1996].

La spécification des langages visuels basée sur une grammaire visuelle, suit la même philosophie dans la mesure où ils permettent la spécification de langages visuels grâce à des règles de grammaire définissant des phrases visuelles valides. Une grammaire de graphe est composée de règles comportant un coté gauche (left-hand side ou LHS) et un coté droit (right-hand side ou RHS). La signification des règles est que le noeud de gauche peut être remplacé par le graphe de droite.

Par exemple, la figure 5.13 est une machine à état avec des rectangles arrondis repré-

sentants des états et les flèches pour représenter les transitions.

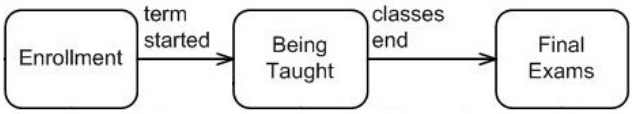


FIGURE 5.13 – Exemple d'une machine à états

Ce petit langage visuel est défini dans la grammaire de graphe dans la figure 5.14.

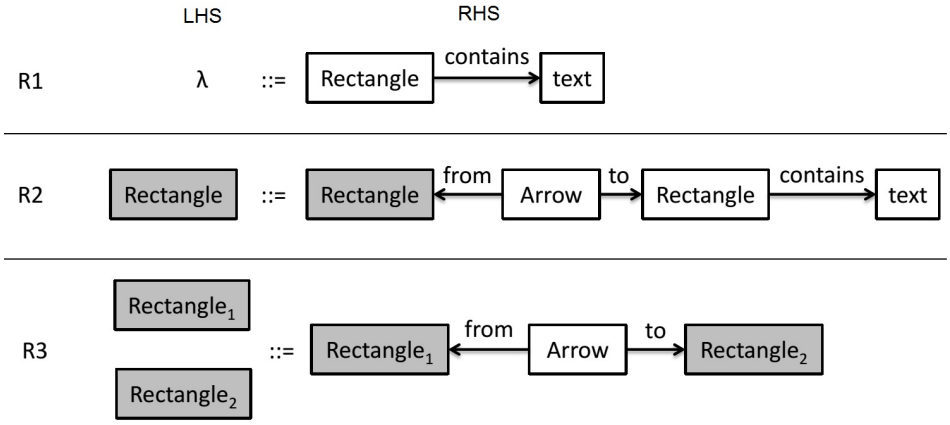


FIGURE 5.14 – Régles LHS et RHS de l'exemple 5.13

Dans cette grammaire, R1 est l'axiome (la règle de haut niveau) qui spécifie comment ajouter un rectangle. La règle R2 est utilisée pour ajouter un nouveau rectangle connecté par une flèche à un rectangle existant. Finalement, la règle R3, est utilisée pour connecter avec une nouvelle flèche, deux rectangles existants.
Les approches les plus marquantes dans la grammaire de graphes selon [Wouters 2013] sont :

- Les grammaires de graphe par couches introduites dans [Rekers 1997], qui divisent la définition d'un langage visuel en *couche physique* pour les attributs graphiques (qui définissent l'apparence visuelle du symbole, par exemple : la taille, la forme, la couleur, etc.) des symboles visuels et en *graphe de relation spatiale* pour les attributs syntaxiques (qui définissent les relations du symbole visuel avec les autres dans le but de déterminer la validité de la phrase visuelle) des symboles visuels ;
- Les grammaires de graphe réservées introduites dans [Zhang 2001] pour améliorer les grammaires de graphe par couches, en simplifiant l'écriture des règles et en supprimant les ambiguités ;
- Les grammaires de graphe par couches contextuelles introduites dans [Bottoni 2000] pour améliorer les grammaires de graphe par couches, en permettant l'expression des conditions positives et négatives dans l'application des règles ;
- Les grammaires de position introduites dans [Costagliola 1997] ;
- Les grammaires de position étendues introduites dans [Costagliola 2004].

Parmi les outils permettant de spécifier les langages visuels selon une grammaire de position étendue, l'outil Visual Diagen. Visual DiaGen est une extension de DiaGen qui permettait déjà de spécifier et de générer des éditeurs graphiques. Cependant, La spécification dans DiaGen était textuelle et moins conviviale.

Pour spécifier un langage visuel dans Visual DiaGen, on dispose d'un éditeur pour définir les règles LHS et RHS (figure 5.15). Dès que l'ensemble des règles sont spécifiées et validés, on peut associer les éléments visuels à une sémantique (dans ce cas des objets java), l'outil peut générer par la suite l'éditeur graphique du langage.

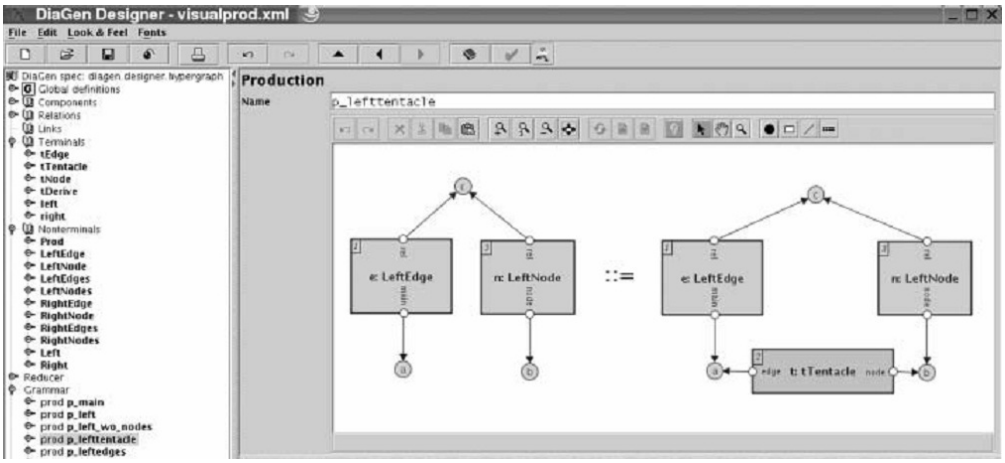


FIGURE 5.15 – Définition de la grammaire de graphe avec Visual DiaGen

Ils existent d'autres outils basés sur la grammaire de graphe. On peut citer $AToM^3$ qui est un outil Meta-CASE écrit en Python en cours de développement au Laboratoire de modélisation, simulation et conception (MSDL) dans la School of Computer Science de l'Université McGill. La définition de la syntaxe abstraite d'un langage dans $AToM^3$ est effectuée avec le formalisme Entité-Relation (ER) étendu avec des contraintes en OCL. Ce formalisme est appelé CD_ClassDiagramsV3. Les modèles sont représentés en interne avec des graphes de syntaxe abstraite. Comme conséquence, la manipulation de ces modèles doit être aussi exprimée sous forme de modèles de grammaires de graphe. Tout comme DiaGen, cet outil utilise des grammaires de graphes pour exprimer la manipulation du modèle. Tel que présenté précédemment, elles sont composées de règles pour lesquelles chacune lie un graphe du côté gauche (LHS) avec un graphe de côté droit (RHS). Une grammaire graphique est appliqué à un graphe d'entrée (appelé aussi graphe hôte ou d'accueil) afin d'effectuer une transformation. Lorsqu'une correspondance est trouvée entre la LHS d'une règle et une partie du graphe d'accueil, ce sous-graphe est remplacé par la partie droite (RHS) de la règle. Dans cet outil, les règles sont classées en fonction de la priorité affectée par l'utilisateur.

Nous avons constaté lors de la manipulation de cette catégorie d'outil, que l'effort fourni pour spécifier un langage visuel est très considérable. Ces outils exigent une forte connaissance des grammaires de langages en général et des grammaires visuelles plus précisément. Un autre point à souligner est celui de la difficulté à réutiliser un langage

visuel déjà spécifié, puisque les langages sont associés dès la création à une sémantique.

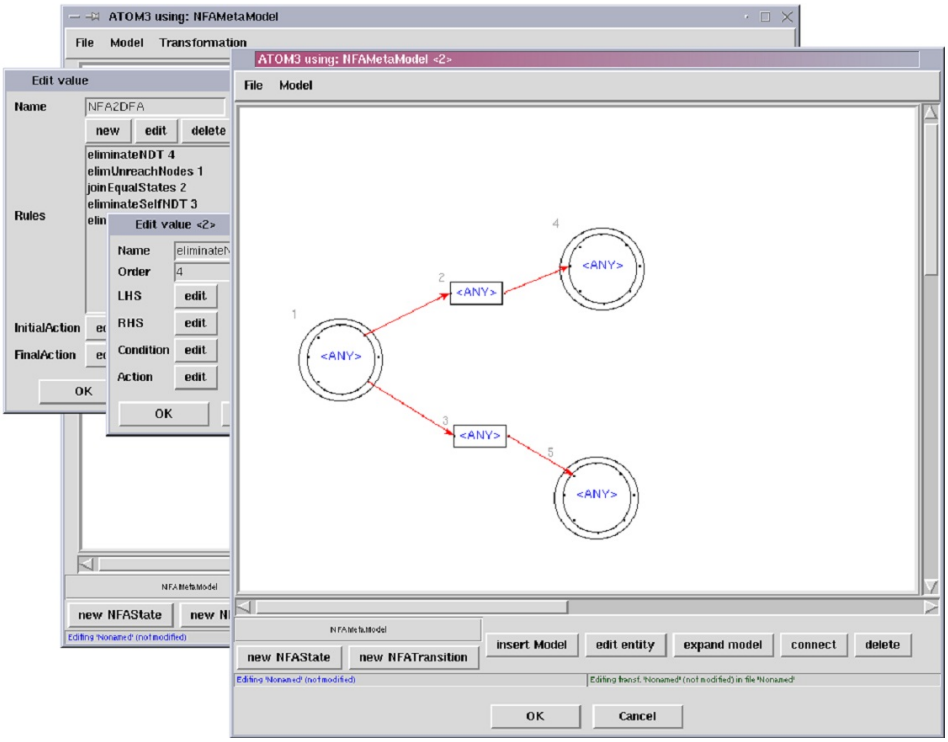


FIGURE 5.16 – Définition de la grammaire de graphe avec AToM3

5.4 Dessinateurs de diagrammes

Ces outils permettent la spécification des diagrammes sans les associer à une syntaxe abstraite. Parmi ceux-ci, on trouve Microsoft Visio et l'outil Dia.

Dia est un logiciel gratuit et open source généraliste, qui peut être utilisées pour dessiner de nombreuses sortes de diagrammes différents. Il utilise des objets spéciaux pour aider à dessiner des modèles entité-relation (ex. DDL SQL), les diagrammes UML (Unified Modeling Language), des organigrammes, diagrammes de réseau et des circuits électriques simples. Il est également possible d'ajouter de nouveaux diagrammes et figures en écrivant de simples fichiers XML, en utilisant SVG (Scalable Vector Graphics).

Scalable Vector Graphics (SVG) est un modèle XML standardisé par le consortium W3C pour décrire des objets graphiques vectoriels. Les coordonnées, dimensions et structures des objets vectoriels sont indiquées sous forme numérique dans ce document XML. Un système spécifique de style (similaire à CSS) permet d'indiquer les couleurs et les polices de caractères à utiliser. Ce format gère quelques formes géométriques de base (rectangles, ellipses, etc.), mais aussi des liens qui utilisent les courbes de Bézier et permettent ainsi d'obtenir presque n'importe quelle forme [Wikipedia 2013b]. Ce standard a été le précurseur d'autres standards comme Diagram Interchange (DI) et dernièrement Diagram Definition (DD) [OMG 2012].

Pour définir un éditeur spécifique dans DIA, l'utilisateur décrit le Sheet (diagramme) en définissant les éléments qui vont éventuellement composer ce diagramme. Par exemple, pour définir un diagramme d'automates dans Dia, le code à intégrer est :

```
<?xml version="1.0" encoding="UTF-8"?>
<sheet xmlns="http://www.lysator.liu.se/~alla/dia/dia-sheet-ns
    ">
<name>Automata</name>
<description>Symbols for designing Automata</description>
<contents>
  <object name="Start State"><description>Start State</
     description></object>
  <object name="Intermediate State"><description>Intermediate
     State</description></object>
  <object name="Final State"><description>Final State</
     description></object>
</contents>
</sheet>
```

Pour les objets du diagramme, il faut définir un fichier Shape pour chaque élément graphique. Dans l'exemple ci-dessus, trois éléments sont nécessaires : Start State, Intermediate State et Final State. Ci-dessous la description de l'élément *Intermediate State* :

```
<?xml version="1.0" encoding="UTF-8"?>
<shape xmlns="http://www.daa.com.au/~james/dia-shape-ns"
    xmlns:svg="http://www.w3.org/2000/svg">
  <name>Intermediate State</name>
  <icon>istate.png</icon>
  <connections>
    <point x="2.5" y="5"/>
    <point x="2.5" y="0"/>
    <point x="0" y="2.5"/>
    <point x="5" y="2.5"/>
    <point x="2.5" y="2.5" main="yes"/>
    <point x="0.732" y="0.732"/>
    <point x="0.732" y="4.268"/>
    <point x="4.268" y="0.732"/>
    <point x="4.268" y="4.268"/>
  </connections>
  <textbox x1="0.3" y1="0" x2="4.7" y2="5"/>
  <aspectratio type="fixed"/>
  <svg:svg width="5.0" height="5.0">
    <svg:circle style="fill:default" cx="2.5" cy="2.5" r="2.5"
       />
  </svg:svg>
</shape>
```

La figure 5.17 montre l'éditeur produit à partir de ces fichiers de configuration. L'outil propose plusieurs fonctionnalités comme undo/redo, le chargement/la sauvegarde de diagrammes sous format XML personnalisé, l'impression, l'export de diagrammes vers de nombreux formats (EPS, SVG, WMF, PNG, etc.). Basé sur les technologies XML, Dia

permet de définir les fonctions de génération de code, par l'écriture de simples fichiers de transformation (XSLT), pour chaque langage cible.

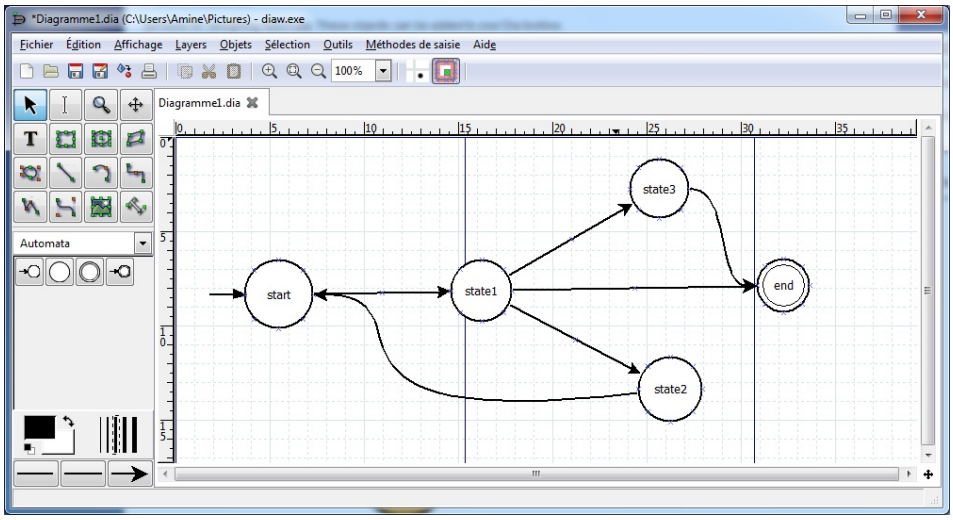


FIGURE 5.17 – Éditeur d'automates avec DIA

5.5 Spécification dirigée par modèles

La première catégorie de spécification des langages visuels basée sur l'IDM est celle des outils qui manipulent des DSML dédiés à la définition des diagrammes. Il existe plusieurs outils dans la littérature et dans l'industrie. Nous avons choisi un éventail représentatif des outils qui ont un fort impacte sur ce domaine, à savoir, les DSL graphiques comme le Framework Eclipse GMF, Obeo Designer ou bien des DSL textuels comme le projet Spray.

5.5.1 Eclipse GMF

L'environnement Eclipse est une plateforme extensible open source basée sur java. Elle offre plusieurs services pour la création des éditeurs graphiques et textuels. Pour construire des éditeurs graphiques sous Eclipse, trois plugins sont nécessaires. Le Framework de modélisation graphique (GMF), le Framework EMF (Eclipse Modeling Framework) [Budinsky 2003] et le Framework d'édition graphique (GEF) [GEF 2013]. Dans le contexte de GMF, EMF est utilisé pour définir le métamodèle ou la syntaxe abstraite du langage (exprimé en Ecore) et pour générer le code de création, d'édition et d'accès aux modèles. GEF est utilisé comme cible technologique dans la génération des modèles GMF, pour implémenter et éditer la syntaxe concrète du langage. Tout comme GEF, le projet GMF est composé de trois sous-projets [Eclipse 2006b] :

1. GMF Runtime : Le projet GMF Runtime est l'environnement d'exécution de la spécification GMF, il offre de nombreuses fonctionnalités :

- Un ensemble de composants réutilisables comme l'impression, l'export de diagramme en image, les menus et Toolbars et plus encore.
- Une infrastructure de commandes qui relie les différentes interactions entre EMF et GEF.
- L'ouverture et l'extensibilité des éditeurs générés.

2. GMF Tooling : ce projet fourni selon une approche IDM, un ensemble de modèles décrivant les éditeurs de diagrammes. En définissant l'outillage, les modèles graphiques et les modèles de mapping avec le domaine (c'est-à-dire le métamodèle du langage spécifique), ce projet permet de générer des éditeurs graphiques opérationnels basés sur GMF Runtime.
3. GMF Notation : ce projet propose un métamodèle notationnel standard. Ce métamodèle sert à stocker les informations des diagrammes séparément de ceux des modèles. Il est basé sur les spécifications du Diagram Interchange [OMG 2006b].

L'OMG a proposé le standard DI (Diagram Interchange) [OMG 2006b] Pour échanger des diagrammes. Ce format a un métamodèle particulier qui capture les concepts de positionnement, de taille et de couleur pour les éléments graphiques des diagrammes. Conçu initialement pour les diagrammes UML, il est devenu applicable/compatible à d'autres types de diagrammes. Malheureusement, si cela arrive souvent d'échanger des modèles via XMI, très peu d'outils mettent en oeuvre correctement la partie DI et perdent souvent la présentation des modèles (c'est-à-dire les diagrammes), quand ils sont transférés d'un outil à un autre [Jezequel 2012].

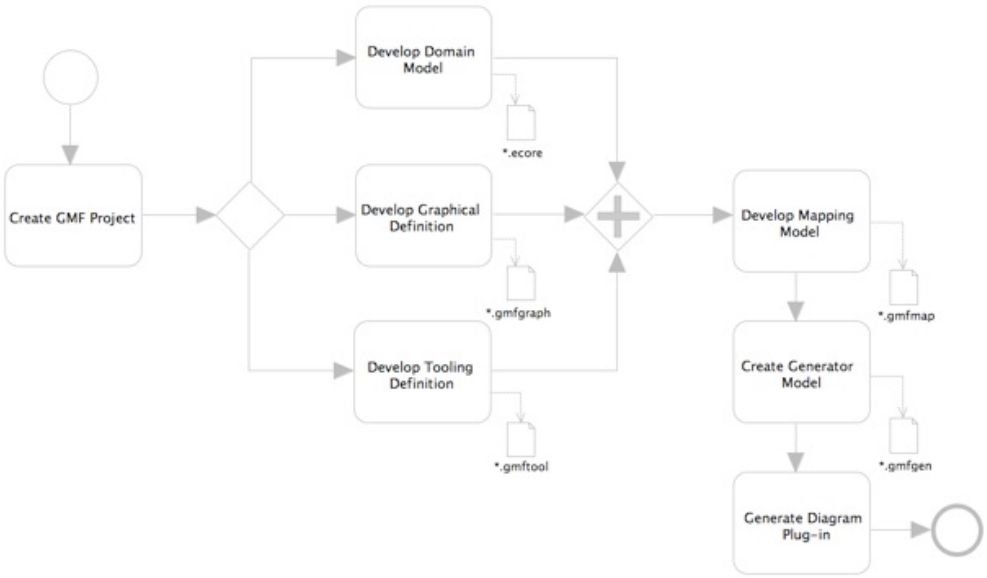


FIGURE 5.18 – Processus de construction des éditeurs graphiques avec GMF

La création d'un éditeur graphique avec GMF se fait donc avec les outils de GMF Tooling dans l'environnement d'exécution GMF Runtime et les diagrammes réalisés avec cet éditeur seront persistés selon la norme d'échange de diagramme défini par GMF

Notation.

Basé sur le paradigme de la séparation des préoccupations, GMF est composé de quatre modèles principaux (figure 5.18) pour la génération des éditeurs graphiques : le modèle de domaine (métamodèle réalisé en Ecore), le modèle de définition graphique (inspirés par les éléments de GEF), le modèle d'outillage qui définie les outils qui seront utilisés dans l'éditeur généré (palette, menus contextuels, etc.) et le modèle de mapping qui définie les relations des éléments graphiques avec les éléments du métamodèle.

La plate-forme Eclipse, avec GMF, offre plusieurs services : le chargement/sauvegarde (en XMI), le zoom, les palettes, les aperçus (outline, bird-eye view), l'exportation des diagrammes en images, les points d'extension pour permettre à d'autres applications d'accéder aux modèles créés, etc. Cependant, un effort de programmation supplémentaire est nécessaire pour implémenter des formes complexes, des undo/redo multiple, l'édition des labels, les vues de propriété, etc. Un effort considérable est nécessaire pour apprendre comment faire un éditeur avec GMF. Toutefois, La documentation (y compris les tutoriels et les livres) et des forums de discussion sont disponibles [Eclipse 2006a]. Malgré la séparation des préoccupations en plusieurs modèles, la partie graphique mélange encore le vocabulaire visuel (les éléments permettant l'affichage graphique comme la forme, la couleur, etc.) et la grammaire visuelle (les règles de compositions des éléments graphiques). Notre expérience dans le développement de Papyrus nous a montré qu'il est impossible de créer des diagrammes complexes en passant par le processus habituel. A cause de diverses limitations constatées dans l'ensemble des modèles GMF comme la difficulté d'introduire des interactions et d'autres problèmes d'ergonomie, nous avons intervenu directement sur le dernier modèle (modèle de génération gmfgen) en changeant la transformation vers le code pour définir les diagrammes Papyrus.

Une fois l'éditeur graphique est en place, ajouter de nouvelles fonctionnalités devient obsolète. Ainsi, adapter l'éditeur aux changements dans le métamodèle est assez simple. Si de nouveaux attributs, classes ou associations sont ajoutées au métamodèle, l'éditeur peut toujours ouvrir les fichiers créés avec la version précédente. Cependant, supprimer ou renommer les nouvelles classes ou attributs peut conduire à des problèmes d'incompatibilité (backward incompatibility).

5.5.2 Obeo Designer/ Eclipse Sirius

Obeo designer (récemment appelé Eclipse Sirius) est un outil adaptatif qui propose un environnement configurable pour définir des représentations graphiques de la syntaxe concrète. Il est basé sur les technologies de l'environnement Eclipse comme EMF, GEF et GMF qui offre les éléments de base pour construire un éditeur graphique [Juliot 2010]. Obeo Designer (version 5.0) permet de créer des espaces de travail de modélisation graphique qui prennent en charge la spécification des langages spécifiques, les notations et les cibles technologiques. Il fournit un outillage pour définir des représentations graphiques tels que les diagrammes, les tableaux ou les arbres avec des interactions riche avec l'utilisateur tout en cachant la complexité. Dans Obeo Designer, un éditeur graphique est décrit en trois étapes principales :

1. La définition de la syntaxe abstraite du langage : consiste à définir les concepts

du domaine, les relations et les propriétés par la création d'un métamodèle Ecore. Obeo Designer offre un éditeur graphique avancé pour ce but.

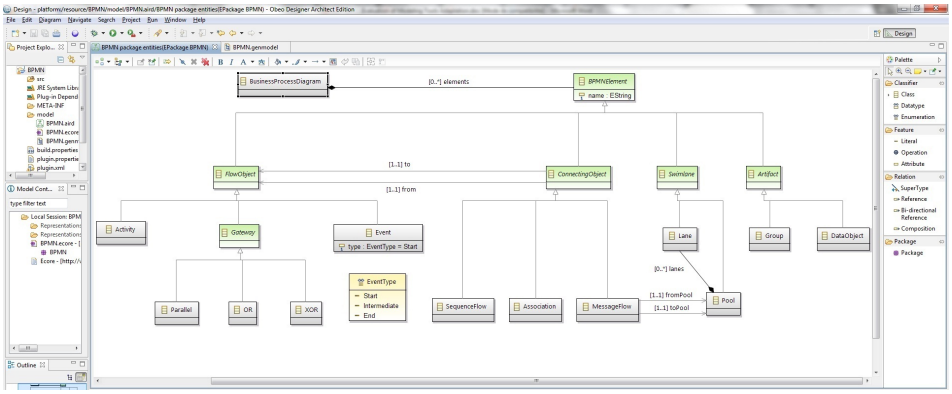


FIGURE 5.19 – Éditeur Ecore sur Obeo Designer

2. La description de l'éditeur graphique : le but de cette description est de définir les points de vue, les représentations visuelles et les éléments graphiques pour chacune de ces représentations.

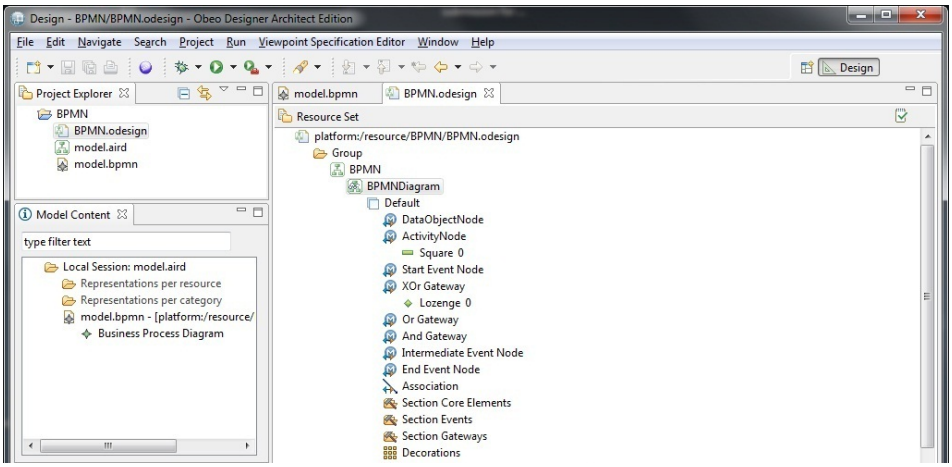


FIGURE 5.20 – Description de la syntaxe concrète dans Obeo Designer

3. La description des représentations graphiques : C'est la dernière étape du processus de construction de l'éditeur. Obeo Designer offre un éditeur arborescent pour décrire la syntaxe concrète (figure 5.20). Dans cet éditeur, nous pouvons créer de nombreux diagrammes, arbres ou tables pour le même DSL. La configuration commence par la définition des éléments graphiques, avec des styles personnalisés (formes, couleurs, etc.) et le mapping avec la syntaxe abstraite.

La notion de points de vue est une abstraction qui fournie une spécification d'un système en se limitant à un ensemble particulier de problèmes. Elle a été introduite par la spécification IEEE 1471 [Wikipedia 2013f]. Dans Obeo designer, les points de vue est

utilisé pour fournir aux utilisateurs un ensemble de représentations visuelles concentré sur une préoccupation particulière. Un point de vue offre un ensemble de représentations. Ce concept définit une projection utilisée pour afficher ou modifier un ensemble de concepts sémantiques. Dans Obeo Designer, une représentation peut être présentée sous forme de diagramme, de tableau ou d'arbre. Le même concept est utilisé dans le RSA, connu sous le nom "Layers" et dans GME sous le nom de «Aspect».

Dans cette étape, on peut créer des validateurs spécifiques, des comparateurs de modèles, des outils comme la palette et les filtres de couche (layer) pour sélectionner les éléments à afficher. On peut aussi créer des générateurs de code en utilisant Acceleo [Obeo 2010]. Les éditeurs graphiques générés par Obeo Designer proposent plusieurs fonctionnalités comme le Undo/Redo, les vues outline, export de diagrammes sous forme d'images, le réarrangement de liens, l'affichage/dissimulation des éléments de diagramme, l'export des modèles en XMI, l'Auto-complétion des noms de classe, la coloration syntaxique, la validation de diagramme, interrogation de modèles/ diagrammes (Query), l'impression, le drag and drop, etc. Obeo Designer offre une bonne interactivité lors de l'élaboration du modèle de domaine et de la description de l'éditeur. En outre, le processus de construction de l'éditeur est progressif. Suite aux modifications, aucune génération du code n'est requise. Le moteur de la transformation, interprète les modifications et les applique à la volée (At Runtime) dans l'éditeur produit.

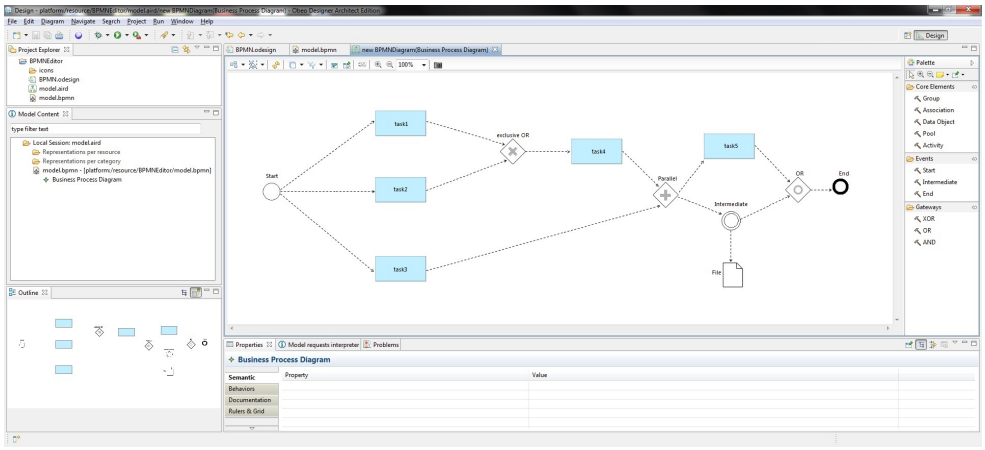


FIGURE 5.21 – Éditeur produit par Obeo Designer

Comme tous les outils basés sur la plateforme Eclipse, Obeo Designer bénéficie de toutes les possibilités d'extension et d'intégration offerte par Eclipse. Cependant, il y'a des améliorations à apporter au niveau du concepteur de l'éditeur. Le concepteur de l'éditeur intègre difficilement des expressions OCL dans la syntaxe concrète graphique. La spécification de la syntaxe concrète est liée étroitement à la syntaxe abstraite, ce qui affaiblit les possibilités de réutilisation des langages visuels.

Le concepteur de l'éditeur ne permet pas d'utiliser la gamme complète des variables visuelles [Bertin 1983]. En effet, il n'est pas possible de créer des formes graphiques complexes comme des lifelines, des compartiments, des polygones complexes, des formes 3D, etc. Les formes de base fournies sont limitées à six formes principales (le carré, le

losange, l'ellipse, le triangle, le point et l'anneau) en plus des représentations iconiques (images). En Mars 2013, Obeo Designer a été proposé conjointement par Obeo et Thales comme projet Eclipse sous le nom de Sirius.

5.5.3 Spray

Le projet Spray est un projet développé et proposé par itemis en 2011, comme solution basée sur les modèles pour générer du code Graphiti. Il se compose de quelques DSL textuels, un générateur et un environnement d'exécution. Pour le générateur, il utilise Xtend qui est un langage de transformations modèles vers texte. L'environnement d'exécution n'est que le Framework Graphiti avec quelques extensions supplémentaires.

Spray [Itemis 2012] propose trois différents DSL textuels : Spray Core, Spray Shape et Spray Style (figure 5.22). Le DSL central est le langage de base de Spray. Pour les éditeurs graphiques simples, ce DSL seul est suffisant. Spray Shape permet de construire des formes complexes à partir de figures primitifs comme les rectangles et les ellipses et de configurer leur placement et leur composition. Les styles des formes, comme la couleur et la police de caractère, peuvent être définies dans le même modèle. Mais afin de réutiliser et de centraliser la définition des styles, le DSL Spray Style permet de définir des classes de style similaires aux CSS pour HTML.

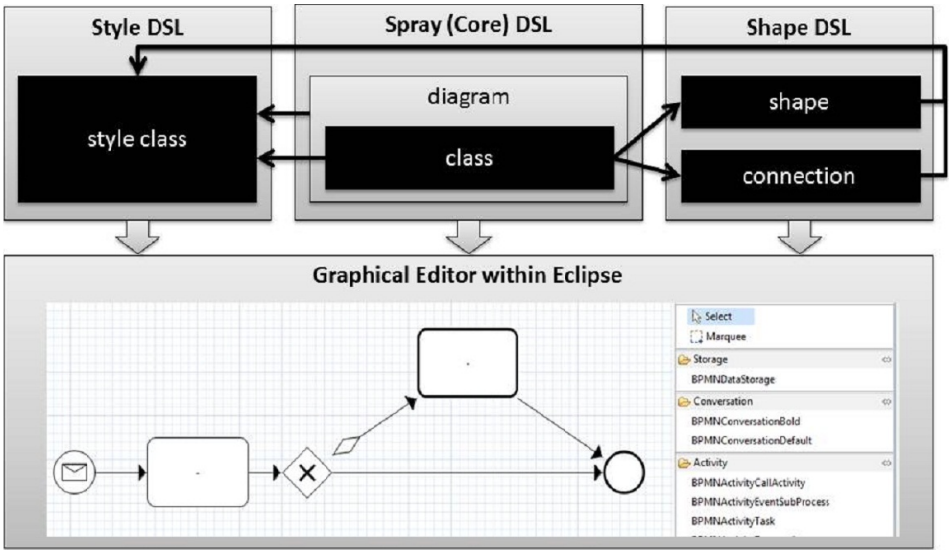


FIGURE 5.22 – Architecture de Spray [Fabio 2013]

```
style BlackAndWhiteStyle {
    description = "White background and black foreground."
    transparency = 0.95
    background-color = white
    line-color = black
    line-style = solid
    line-width = 1
    font-color = black
```

```
    font-name = "Tahoma"
}
```

Le DSL principal permet la création de formes simples et aussi les figures les plus complexes. Il est semblable à SVG dans une certaine mesure. Spray Shape permet de combiner des formes primitives comme des rectangles, ellipses, polygones, polylignes et des lignes pour avoir des formes complexes. Toutes les formes à l'exception des lignes, polylignes et textes peuvent être imbriquées arbitrairement.

Le DSL de style permet de définir un ensemble d'attributs de représentation pour les formes. Si aucun style n'est référencé, un style par défaut est utilisé. Un style peut hériter d'un autre et changer un ou plusieurs attributs. Pour chaque niveau d'imbrication, le style hérite du niveau immédiatement supérieur. Dans l'exemple ci-dessous le style BlackAndYellowStyle hérite de BlackAndWhiteStyle et seul l'attribut background-color est modifié (figure 5.23).

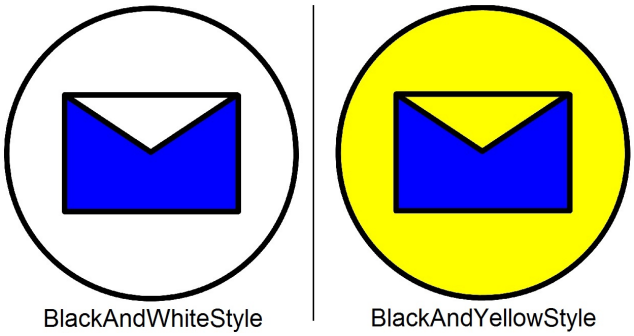


FIGURE 5.23 – Héritage en Spray

```
style BlackAndYellowStyle extends BlackAndWhiteStyle {
  description = "Yellow background and black foreground."
  background-color = yellow
}
```

Une définition Spray core est devisée en quartes parties : les figures, l'outillage, les comportements et les règles (contraintes). Cependant, la définition du Core, mélange l'aspect graphique avec la sémantique. Les éléments de diagrammes (noeud ou lien) sont spécifiques aux concepts du métamodèle. Ce qui affaiblit la réutilisation de la même définition graphique pour d'autres concepts.

```
node <Name> for <EClass> {
    figure { }
    tooling { }
    behavior { }
}
```

Ce projet montre un intérêt remarquable au niveau de la réutilisation et la définition de diagrammes complexes, même s'il est toujours en cours de développement (instable). Il manque aussi d'un concept important qui sont les noeuds de bordure qui permettent

de spécifier des éléments graphiques attachées (par exemple les ports). D'autre projet utilisent des DSL textuels. On peut citer Poseidon DSL [Gentleware 2013] qui a globalement les mêmes caractéristiques que Spray à l'exception de la réutilisation par héritage.

Quelques outils UML propose de définir des éditeurs graphiques pour des langages spécifiques définis en tant que profil UML (mécanisme d'extension par stéréotypes). Nous avons choisi IBM RSA comme représentant de cette catégorie à cause de l'ampleur qu'il représente dans l'industrie.

5.5.4 IBM Rational Software Architect (RSA)

Rational Software Architect de IBM (version 8.0) est un environnement de développement logiciel conforme à la spécification d'UML 2.0, construit sur la plate-forme Eclipse [IBM 2011]. RSA fournit le mécanisme d'extension UML avec les stéréotypes pour définir des profils et permet de générer des éditeurs pour ces profils. Le mécanisme des profils d'UML fait la force du RSA : basé sur UML, elle bénéficie de sa généricité, sa réputation et sa syntaxe concrète. Dans le même temps, il fait aussi sa faiblesse : UML contient beaucoup de concepts pas toujours adaptées aux besoins particuliers des langages spécifiques aux domaines. La création de profil en RSA est assez simple. L'utilisateur a besoin de créer un projet de profil UML (directement pris en charge par l'assistant de création de nouveaux projets Eclipse), sélectionner les méta-classes à stéréotyper, (facultativement) spécifier les icônes et images et créer une version pour ce profil. L'environnement de l'outil offre de nombreuses fonctionnalités, comprenant le chargement/sauvegarde, undo / redo, les filtres (layer), le drag and drop, la validation, l'impression, le zoom, la vue de propriétés, etc. La documentation est très bonne et abondante dans le web. Cependant, la convivialité et l'ergonomie des éditeurs générés en RSA sont plutôt faibles.

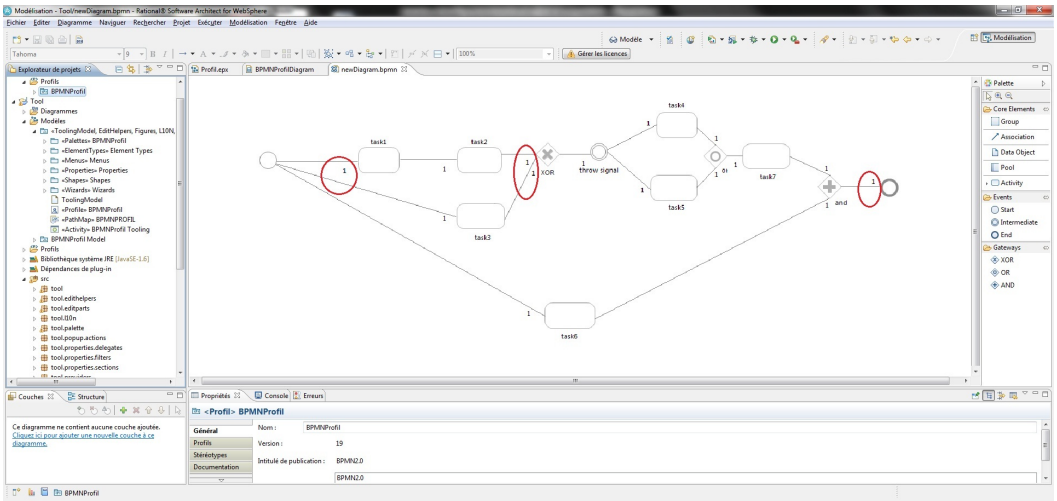


FIGURE 5.24 – Éditeur BPMN dans IBM RSA

RSA se réfère également à un éditeur de diagramme, appelé modèle d'outillage, pour créer des entrées personnalisées de palettes, éléments de menu, les assistants de création

(wizards) et des vues de propriétés. Cependant, RSA ne supporte pas les restrictions personnalisées sur les styles de liens ou sur la personnalisation de la présentation graphique du domaine. Par exemple, dans la figure 5.24 les liens générés sont fidèles à la notation UML et affichent les multiplicités du diagramme de classe dans l'éditeur généré de BPMN. Les possibilités de personnalisation graphique se résument à :

- Par défaut, la réutilisation de la syntaxe concrète d'UML pour les concepts du domaine spécifique (chaque concept est représenté par la notation visuelle de sa méta-classe associée).
- L'utilisation d'une représentation iconique des stéréotypes (l'image associée à l'élément «stéréotype» d'UML).
- Sinon, si on choisie la génération des formes personnalisées, un rectangle est affiché pour les noeuds et une simple ligne pour les liens.

Toutefois, si l'utilisateur a besoin de formes plus complexes, l'outil propose seulement de modifier le code généré. Ceci nécessite une solide connaissance du Framework GMF, GEF, EMF et le développement Eclipse.

5.6 Synthèse et discussion

Pour effectuer une évaluation des différents outils de génération des éditeurs graphiques de diagramme, nous avons identifié les critères qui nous intéressent et qui seront la base d'une telle évaluation.

5.6.1 Critères d'évaluation

Deux approches d'évaluation sont proposés par P. Mohagheghi and Ø. Haugen [Mohagheghi 2010] dans ce domaine. L'étude qualitative et L'approche quantitative.

L'approche qualitative consiste aux cas d'étude, l'analyse des langages et outils de syntaxe concrète par des experts pour diverses caractéristiques et d'analyser les retours d'expédience vis-à-vis les utilisateurs.

Pour l'évaluation quantitative, ils ont identifié plusieurs métriques (effort, compréhension, la convivialité, la réutilisation, etc.) Dans notre étude, nous avons mis un accent particulier sur les critères d'évaluation les plus pertinentes dans le développement des outils de modélisation, à savoir l'expressivité et la complétude graphiques, la réutilisation, la maintenabilité, l'évolution, la facilité d'implémentation, etc.

Différents acteurs interviennent dans le développement des éditeurs graphiques. Les critères d'évaluations des éditeurs graphiques différents d'un acteur à un autre. Ces acteurs sont recensés par (Mohagheghi and Haugen, 2005), avec des exemples de critères qui intéressent chacun d'eux.

Dans [Sprinkle 2010], les différentes parties prenantes du développement des outils de modélisation et les critères qui les intéressent sont :

- Les développeurs d'outils. Les critères pertinents pour ces acteurs sont ceux identifiés dans [Howatt 2001]. Ces critères permettent d'évaluer dans quelle mesure un langage supporte la programmation d'applications spécifiques (éditeurs, simula-

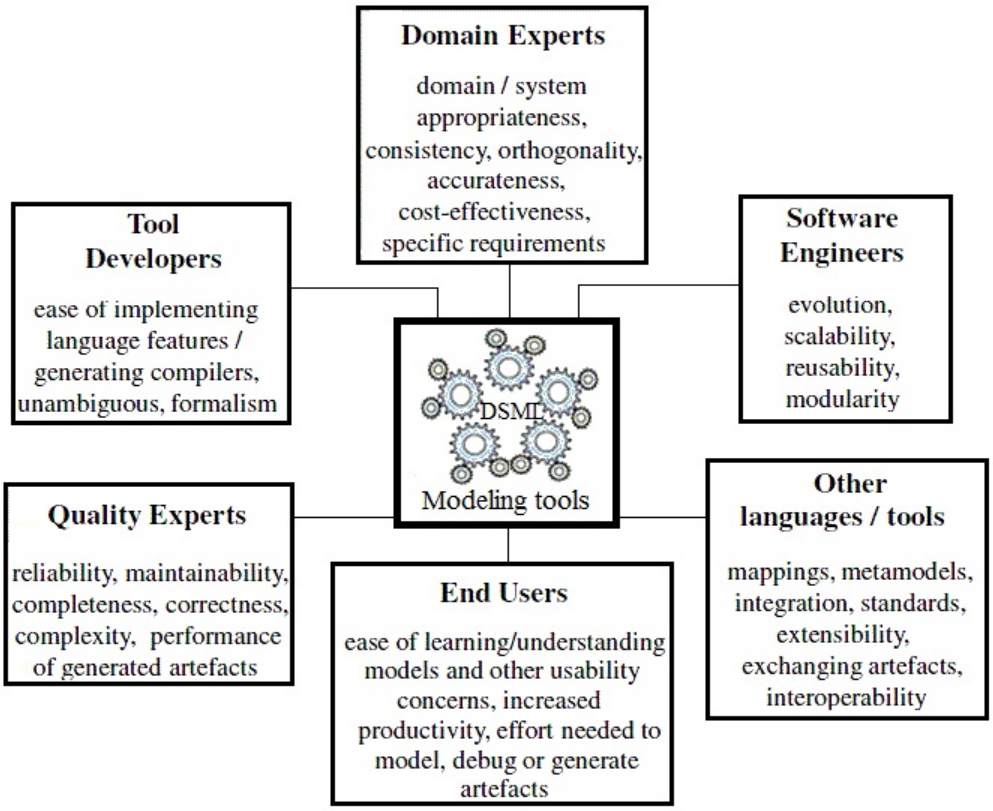


FIGURE 5.25 – Critères d'évaluation selon les acteurs intervenant dans les outils DSML

teurs, compilateur, etc.), ils évaluent aussi la façon avec laquelle ces langages sont définis (formels ou pas) ;

- Les experts domaine. L'évaluation doit donc se concentrer sur les objectifs de la solution ;
- Les ingénieurs logiciels. Ils sont intéressés par les caractéristiques qui conduisent à développer de « bons » logiciels. Des exemples de leurs préoccupations sont la réutilisation des modèles et l'évolutivité de la solution. L'application de certaines pratiques du génie logiciel permet aussi d'améliorer la qualité des modèles et les artefacts générés ;
- Les experts de qualité. Ils sont intéressés par la qualité des modèles ou des artefacts générés à partir de modèles ; Ceux-ci peuvent avoir des exigences concernant l'exhaustivité et la performance du code généré, la facilité de compréhension et la maintenabilité des modèles et artefacts générés ;
- Les critères vis-à-vis des outils couvrent les exigences d'interopérabilité avec d'autres outils ou langages, les mappings entre eux, la construction d'extensions et la conformité aux standards ;
- Les utilisateurs finaux : sont ceux qui utilisent les solutions de modélisation. La convivialité et la facilité de prise en main des outils sont les critères les plus pertinents pour ces acteurs. Le lien entre les développeurs d'outils et les utilisateurs

finaux s'illustre dans la production des supports par les développeurs tels que des bibliothèques, des débogueurs, des interfaces utilisateur intuitives (UI), une documentation abondante et claire, etc.

5.6.2 Résultats d'évaluation

Cette section présente un résumé de tous les critères d'évaluation utilisés pour évaluer chaque outil/approche de modélisation.

5.6.2.1 L'expressivité et la complétude graphique

Selon [Moody 2009a], l'expressivité graphique est définie par le degré de fidélité d'expression d'un système par une notation visuelle. Ceci peut être vérifié si toutes les variables visuelles sont utilisées dans la construction de cette notation. La complétude graphique est définie par la capacité d'utiliser pleinement la variable de forme pour la construction de notations visuelles complexes : composites, imbriqués, 2D/3D, etc. La plupart des outils évalués dans [El-kouhen 2011, Amyot 2006, Mohagheghi 2010] sont confrontés à de sérieuses limitations en termes d'expressivité graphique. Pour les outils basés sur des profils UML, la principale forme d'expression est l'utilisation de la syntaxe concrète d'UML. Chaque concept spécifique est représenté par la notation visuelle de sa méta-classe associée et si les utilisateurs ont besoin d'une représentation plus spécifique, il n'existe qu'une seule autre possibilité qui est l'utilisation de la représentation iconique des stéréotypes (les images associées aux stéréotypes). Les outils de dessin tel Microsoft Visio ou Dia permettent de définir des diagrammes en écrivant des fichiers SVG (Scalable Vector Graphics). A l'exception des outils basés sur un DSML, les possibilités graphiques des autres outils, sont limitées à la représentation en image des concepts. Les outils basés sur une grammaire de graphes fournissent la séparation entre les variables visuelles (vocabulaire visuel) et leur composition (grammaire visuelle). Cependant, pour définir un langage visuel l'utilisateur doit fournir un effort en amont pour analyser et recenser toutes les règles (RHS et LHS). Les outils bases sur des DSML, proposent généralement une description permettant une meilleure expressivité graphique. Ceci est du par le fait que les concepts de ces DSML décrivent les unités atomiques de construction des notations visuelles. Ils peuvent définir différentes formes de représentations visuelles (figures 2D composés) ou images SVG.

5.6.2.2 Séparation des préoccupations

Le principe de la séparation des préoccupations peut être considéré comme l'un des principes clés de l'ingénierie logicielle. Ce principe stipule qu'un problème donné implique différents types de préoccupations qui doivent être identifiés et séparés pour faire face à la complexité et à la réalisation des facteurs de qualité d'ingénierie nécessaires, telles que la robustesse, l'adaptabilité, la maintenabilité et la réutilisabilité [Tekinerdogan 2000b]. Une préoccupation est une abstraction canonique d'une solution pertinente pour un problème donné [Tekinerdogan 2000a]. Notez que cette définition implique que les préoccupations ne sont pas absolues, mais relative par rapport au

problème considéré. Ce qui peut être une préoccupation pour un problème ne peut l'être pour un autre. Les préoccupations sont définies par un consensus d'experts dans le domaine correspondant. La plupart des méthodes de spécification mentionnées ci-dessus mélangent les préoccupations. La forme la plus commune de ce mélange est celle de la forme et du contenu (représentations visuelles et sémantiques). Par exemple, dans le cas de spécification d'un diagramme à l'aide de GME ou MetaEdit+, ces outils permettent de créer des concepts spécifiques et leurs représentations associées dans le même référentiel. Ceci introduit un couplage plus fort entre les aspects sémantiques et graphiques. Le même problème est observé avec Obeo Designer, qui permet d'associer des concepts graphiques aux éléments du méta-modèle du domaine, dans le même modèle. Une autre forme de mélange de préoccupations se situe entre le vocabulaire visuel et la définition de la grammaire. En effet, la plupart des outils qui offrent la séparation de la partie graphique et de la sémantique, comme GMF, Obeo Designer ou Spray ne proposent pas la séparation des préoccupations au niveau de la syntaxe graphique : entre le vocabulaire (formes, couleurs, styles, etc.) et la grammaire (structure et composition des représentations).

5.6.2.3 Réutilisation de la syntaxe concrète graphique

La séparation des préoccupations est le paradigme le plus utilisé afin d'augmenter la réutilisation des éditeurs graphiques, il est utilisé dans des outils comme GMF et TopCased. Désormais, il existe d'autres formes de réutilisation. Plusieurs outils tels que MetaEdit+ et GME ont introduit des langages de description afin de réutiliser les spécifications de ces langages propriétaires. Le but de ces langages est de réutiliser le graphe conceptuel (modèle sémantique) avec ses différents graphes de représentation (modèle de représentation graphique). Cependant, hormis les contraintes liées à la nature de la licence, l'utilisation de ces langages de méta-description affaiblit la portabilité de la spécification. Le mélange de la syntaxe abstraite et concrète limite les possibilités de réutilisation du langage visuel. Nous avons relevé un autre problème dans GMF ou Obeo Designer : le peu de possibilités (voire l'absence) de réutilisation des éléments de spécification, ce qui se traduit généralement par une redondance des spécifications, au niveau modèle et des classes au niveau du code généré. Nous avons relevé d'autres formes de réutilisation dans les outils basés sur les profils UML. Grace à ce mécanisme, des outils tels que IBM RSA ou Magic Draw permettent de réutiliser les concepts UML et leurs syntaxe concrète en étendant leurs méta-classes correspondantes : chaque concept est représenté par la notation visuelle de sa méta-associé. Cependant, l'utilisation de ces outils consiste à utiliser UML pour définir le concept du domaine, qui peut parfois être en contradiction avec la vision de l'IDM : Ces mécanismes n'ont pas toujours la précision souhaitable et mènent parfois à des contorsions dangereuses pour rester dans l'espace UML [Jezequel 2012]. Des outils comme Spray, proposent une réutilisation par héritage. Chose qui permet de reprendre des définitions graphiques existantes et modifié un ou plusieurs paramètres.

Comparaison pratique des outils						
	Temps*	Documentation	Format d'artéfact	Plateforme	$Prix$**	Intégration
GEF	15-30 jours	+++	Code java	Linux, Win	Gratuit	Eclipse
Graphiti	10-20 jours	+	Code java	Linux, Win	Gratuit	Eclipse
GME	5-7 jours	++	Binaire	Win	Gratuit	Standalone
MetaEdit+	1-3 jours	+++	Référenciel	Linux, Win, Mac	Gratuit/150-9500 €	Standalone
$AToM^3$	5-7 jours	++	Non	Linux, Win	Gratuit	Standalone
Dia	1-3 jours	+++	XML	Linux, Win	Gratuit	Standalone
Ms Visio	1-3 jours	+++	XML	Win	400 €	Ms Office
GMF	7-15 jours	+	XMI	Linux, Win	Gratuit	Eclipse
Obeo	7-15 jours	++	XMI	Linux, Win, Mac	Gratuit/ 800-3000 €	Standalone
Spray	1-3 jours	+++	Texte	Linux, Win, Mac	Gratuit	Eclipse
IBM RSA	7-15 jours	++	XMI	Linux, Win, Mac	Gratuit/900-4500 $	Standalone

* Le temps inclut le temps de la documentation et de la prise en main de l'outil. ** MetaEdit+, Obeo Designer et IBM RSA sont gratuits pour une période de 30/45 jours. Pour MetaEdit+, une licence commence de 150 euros jusqu'à 9500 euros pour la version complète. Pour Obeo Designer, une licence individuelle commence à environ 800 euros. Environ 3000 euros pour une licence d'entreprise. Pour IBM RSA, un large choix d'options allant de 900 dollars à 4500 dollars pour une licence complète de l'outil.

Comparaison technique des outils					
	Syntaxe Abstraite (AS)	Syntaxe Concrète (CS)	Lien $AS \longleftrightarrow CS$	Réutilisation[3]	Expressivité Graphique
GEF	EMF/Ecore	Java	Fort	Non	+++(Code)
Graphiti	EMF/Ecore	Java	Fort	Héritage	++++ (Code)
GME	MetaGME (GMeta)	MetaGME (GMeta)	Fort	Non	+ (Images)
MetaEdit+	GOPPRR	WYSIWYG editor	Fort	Non	+++ (SVG)
AToM3	ClassDiagramsV3	Vectorial tool	Fort	Non	++ (SVG)
Dia	Non[1]	SVG	$Aucun$[1]	Non	++ (SVG)
Ms Visio	Non[1]	SVG	$Aucun$[1]	Non	++ (SVG)
GMF	EMF/Ecore	DSML	Faible	SOC[2]	+++
Obeo	EMF/Ecore	DSML	Fort	Non	++
Spray	EMF/Ecore	DSL Textuel	Fort	Héritage	++++
IBM RSA	Profils UML	Image Stéréotype	Fort	CS d'UML	+ (CS UML/Images)

[1] Outils sans Syntaxe abstraite. [2] SOC = Separation Of Concerns (Séparation des préoccupations). [3] Réutilisation de la syntaxe concrète graphique.

Deuxième partie

Contributions pour la spécification des éditeurs de diagrammes

CHAPITRE 6

Vue d'ensemble de la proposition

Sommaire

La spécification des outils de modélisation consiste à définir les langages visuels de modélisation et leurs interactions dans un éditeur graphique à partir de constructions atomiques qui forment la syntaxe concrète. Comme nous l'avons vu dans le chapitre 4, la définition des différentes préoccupations de la spécification des outils de modélisation est une tâche très proche de celle de la modélisation elle-même, en considérant le langage visuel ainsi que son éditeur comme le système à modéliser.

Dans ce chapitre, nous allons donner des éléments de réponses à quelques questions soulevées dans le chapitre 2, en particulier aux questions 3, 4 et 5. Des réponses plus complètes pour l'ensemble de ces questions sont détaillées dans le chapitre 7.

La section 6.1 discute le paradigme utilisé par notre solution pour spécifier des outils de modélisation, tout en expliquant à quel niveau d'abstraction se situe notre proposition. Cette proposition est aussitôt comparée avec le standard Diagram Definition [OMG 2012]. La section 6.2 présente l'architecture générale de notre proposition ainsi que notre approche basée sur les composants qui nous permet de répondre aux problématiques de réutilisation.

Le but de notre travail consiste à concevoir des syntaxes concrètes graphiques et les éditeurs qui permettent : 1) l'édition de cette syntaxe concrète. 2) de faire le lien entre cette syntaxe concrète et la syntaxe abstraite. 3) de réutiliser tout ou partie des modèles de spécification existants. Pour cela, nous proposons d'utiliser une approche dirigée par les modèles, pour assurer l'indépendance à la technologie, faciliter la maintenance de la spécification et permettre une bonne pérennité de celle-ci.

Nous avons aussi utilisé les concepts de l'ingénierie des langages visuels et plus précisément les éléments des langages hybrides comme UML et BPMN dans la description des éditeurs de diagramme.

6.1 Modéliser les langages visuels et leurs éditeurs

Ce n'est que très récemment que les langages visuels commencent à être décrits par des modèles. SVG était parmi les contributions pionnières dans ce domaine. Son rôle principal était de décrire des images vectorielles à partir d'un modèle XML (schéma xsd). Ce langage a inspiré par la suite d'autres contributions comme Diagram Definition qui est devenu le standard pour la définition des langages visuels à partir de modèle. Dans la section **??** nous présentons le standard Diagram Definition et les points de convergence et de divergence entre ce langage et notre proposition. Par la suite nous proposons dans la section 6.1.1 une échelle d'abstraction pour situer notre solution par rapport au niveau d'abstraction dans la définition des langages visuels ainsi que par rapport aux autres contributions dans ce domaine.

Nous utilisons dans notre proposition, une approche dirigée par les modèles pour décrire les langages visuels hybrides ainsi que leurs éditeurs. Pour définir les concepts de base de ces langages, nous avons effectué des études théoriques (chapitre 4) sur la nature de ces langages et des études empiriques [El-kouhen 2011] sur les diagrammes les plus représentatifs et les plus utilisés dans le génie logiciel (par exemple UML, BPMN). Le résultat de ces études est un ensemble de métamodèles et d'outillages compatibles avec les formalismes existants comme le standard OMG Diagram Definition (DD)[OMG 2012].

Le but initial du standard Diagram Definition (DD) est de fournir une base pour la modélisation et l'échange des notations graphiques, et plus spécifiquement les notations graphiques basé sur la notion de graphe (formé de sommets/noeuds et d'arêtes/liens) que l'on trouve dans la syntaxe concrète des langages tels que UML, SysML et BPMN, où les notations sont liées à des syntaxes abstraites définis avec MOF [OMG 2006a]. L'architecture de (DD) est focalisée sur l'échange plus que la spécification de diagramme. Elle distingue deux informations graphiques : la première information concerne le package Graphics qui contient les informations graphiques non-échangeables comme les formes, styles de ligne et les couleurs. Le deuxième type d'informations, concerne les informations graphiques échangeables comme la position et la taille. Ainsi, L'architecture de (DD) comprend deux métamodèles pour permettre la spécification de ces deux types d'information graphique, Diagram interchange (DI) pour les informations échangeables et Diagram Graphics (DG) pour les autres informations graphiques non-échangeables.

En projection sur les notions présentées dans le chapitre 4, les métamodèles (DG) et (DI) de (DD) mélangent les deux parties de la syntaxe concrète graphique (voir figure 4.2), c'est-à-dire la grammaire et le vocabulaire visuels. Diagram Graphics (DG) comporte les éléments de la syntaxe (grammaire et vocabulaire) non-échangeables alors que Diagram Interchange (DI) se focalise sur quelques éléments échangeables du vocabulaire comme la taille et la position. La conception de Diagram Graphics (DG) a été largement inspiré par la spécification du (SVG) pour facilité le passage de (DG) vers les standards graphiques existants dans l'industrie [OMG 2012].

6.1.1 Niveaux d'abstraction

Jusqu'à présent, nous avons considéré le langage visuel et son éditeur comme le système à étudier. Comme nous avons choisi une approche dirigées par les modèles pour spécifier les diagrammes, nous allons nous positionner par rapport au niveau d'abstraction des modèles. Il existe une échelle d'abstraction qui se compose de quatre niveaux [osmoz 2013] : niveau conceptuel, niveau organisationnel, niveau logique et niveau physique (figure 6.1).

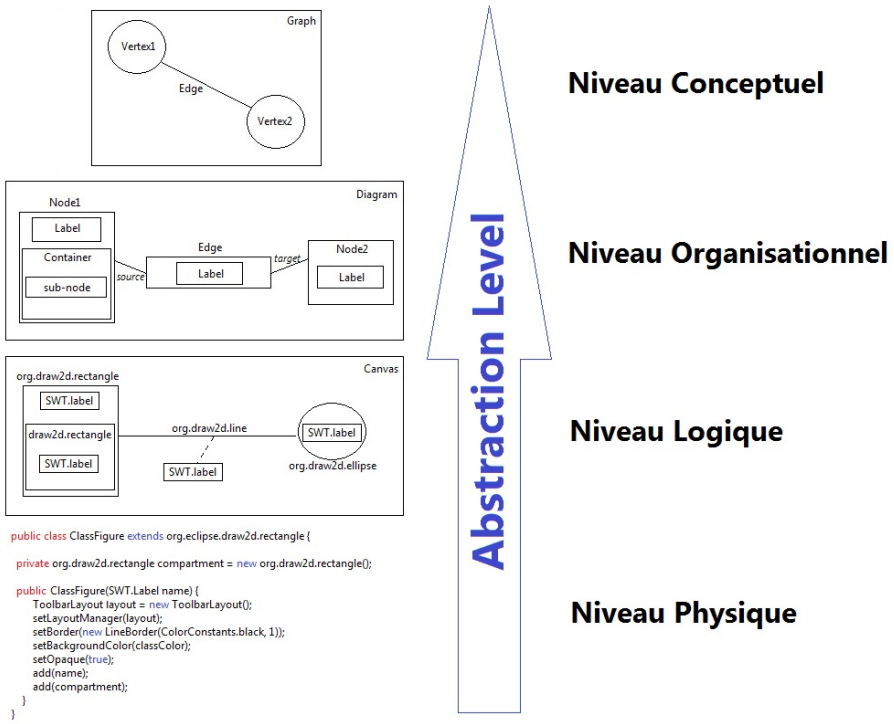


FIGURE 6.1 – Niveaux d'abstraction

1. Le niveau conceptuel (L3). C'est le plus haut niveau d'abstraction. À ce niveau, on considère les éléments du diagramme en tant qu'objets, d'un point de vue extérieur, en faisant totalement abstraction de leur intérieur. Dans ce cas le diagramme est vu comme un graphe (ensemble de sommets et d'arêtes rien de plus). Cette description ne permet pas d'atteindre notre objectif de spécifier des diagrammes hybrides de type UML [Bottoni 2004].
2. Le niveau organisationnel (L2). À ce niveau, on considère les éléments du langage visuel en tant que systèmes, d'un point de vue interne, composés d'autres éléments plus simples. Malgré que notre proposition peut se situer dans le niveau conceptuel de cette échelle (peut décrire un diagramme avec le minimum d'informations), elle ne peut décrire un langage visuel complexe tel que UML qu'à ce niveau. A ce niveau d'abstraction, nous pouvons décrire les éléments du diagramme d'un point de vue structurel et à un niveau de granularité représentatif à la notation des

langages hybrides de type UML.

3. Le niveau logique (L1). À ce niveau, on considère les éléments du langage visuel en tant qu'informations spécifiques à l'implémentation. Ici, on voit l'introduction des détails techniques spécifiques à la plateforme physique cible. C'est l'image du système tel qu'il sera implémenté.
4. Le niveau physique (L0). C'est la réalisation du système étudié, il prend enfin une forme physique (code). À ce niveau on prend en compte toutes les contraintes et limites du monde réel.

Cette échelle peut être mise en relation avec la classification des approches de spécification des langages visuels (chapitre 5). Nous proposons ainsi de les classifier par rapport au niveau d'abstraction.

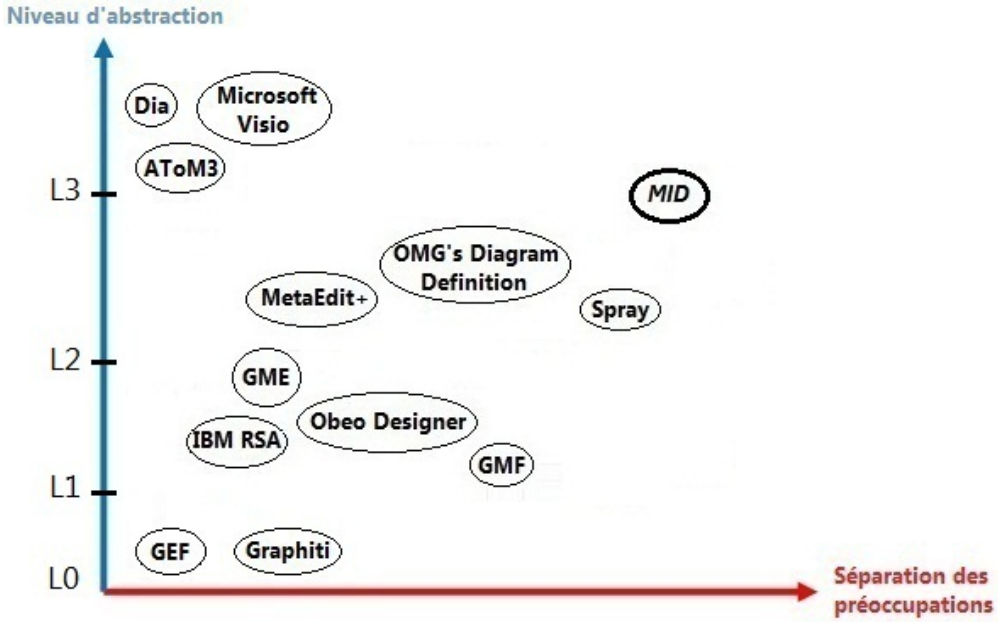


FIGURE 6.2 – Classification des différentes approches selon le niveau d'abstraction

Dans la figure 6.2, nous classifions les différentes approches de spécification de la syntaxe concrète graphique selon le niveau d'abstraction de la spécification et la séparation des préoccupations. Au plus haut niveau d'abstraction, c'est-à-dire au niveau conceptuel, nous trouvons les outils de dessin de diagramme ainsi que les approches basées sur la grammaire des graphes. Ceci peut être expliqué par la manière dont ces outils définissent les langages visuels, qui pour la plupart, utilisent les notions de graphes et d'éléments de graphes, manquant ainsi de possibilité de définir des langages autres que les langages de type entités-relations (ER). Les dessinateurs de diagramme ainsi que les outils à base de grammaire de graphes manquent aussi d'aération des métamodèles, ce qui représente une faiblesse en termes de séparation des préoccupations, puisque la plupart de ces outils ne supportent pas le mapping avec une syntaxe abstraite (source de données). Dans le deuxième niveau d'abstraction (niveau organisationnel), nous trouvons un large

éventail d'outils (Spray, MetaEdit+, Diagram Definition, etc) utilisés pour la spécification des langages visuels. Ces outils basés pour la plupart sur des modèles, offrent une indépendance aux technologies d'implémentation ainsi qu'une forte séparation des préoccupations qui se traduisent par une forte réutilisabilité de la spécification. Notre proposition se situe dans la limite entre ce niveau et le niveau supérieur. Le troisième niveau d'abstraction est celui des outils qui sont dépendant de la technologie d'implémentation cible. On peut citer par exemple GMF (GMF tooling) qui introduit dans ces modèles des détails techniques de Draw2D ou IBM RSA qui est fortement lié à UML. Le dernier niveau d'abstraction n'est autre que le niveau physique (code). Dans cette catégorie nous trouvons des Frameworks comme GMF Runtime, GEF et Graphiti.

À l'exception des dessinateurs de diagramme ainsi que les outils à base de grammaire de graphes, nous avons constaté que la remontée d'abstraction est favorisée par la séparation des préoccupations. Le passage d'un niveau d'abstraction à un autre inférieur augmente la fusion des préoccupations et l'ajout des détails techniques et technologiques dans la spécification.

6.2 Réutiliser les langages visuels

Pour permettre une bonne réutilisabilité, nous proposons une approche basée sur les composants. Cette approche a pour but de bénéficier des avantages de l'encapsulation (facilité de maintenance et de composition) et les avantages de l'interfaçage (mécanisme de nommage des interfaces). De plus, notre approche permet de réutiliser par héritage : un composant peut hériter d'un autre composant et il peut aussi redéfinir (spécialiser) certaines de ses caractéristiques (style, structure, interaction). La spécialisation d'une caractéristique d'un composant se fait par la réutilisation de l'interface correspondante à cette caractéristique en respectant le nommage de celle-ci (voir section 7.3).

Le premier avantage de cette proposition est son indépendance de la syntaxe abstraite, ce qui permet d'assembler les éléments graphiques (parfois les mêmes) différemment d'un diagramme à un autre. Un autre avantage qui se montre pertinent est celui de la facilité de maintenance de cette approche de spécification. En évitant la redondance et les doublons au niveau de la spécification des éditeurs de diagramme (sous forme de modèles), nous pouvons améliorer la maintenabilité de cette spécification. Ceci est possible grâce à un mécanisme qui nous permet de répercuter toute modification d'une définition sur tous les endroits utilisant cette définition. L'approche composant se montre prometteuse dans ce cadre. En définissant la composition de la structure d'un élément graphique et l'encapsulation de ses interfaces, nous facilitons la maintenance et la modification des spécifications.

Nous introduisons dans la section 6.2.1 l'approche basée sur les composants, ses avantages ainsi que les travaux connexes dans ce domaine.

6.2.1 Approche basée sur les composants

Ces dernières années, le développement de logiciels à base de composants s'est imposé comme une technologie de réutilisation très prometteuse [Gössler 2005]. Le

développement de logiciels à base de composants est basé sur le concept de composition ou de l'assemblage des systèmes à partir d'éléments indépendants et connectables (pluggables). L'ingénierie à base de composants est fondée sur un paradigme qui est commun à toutes les disciplines de l'ingénierie : les systèmes complexes peuvent être obtenus par assemblage des composants (blocs de construction) [Gössler 2005].

Selon [Szyperski 2002], un composant est une unité de composition indépendante avec des interfaces typées (points de couplage avec les éléments externes aux composants) et des dépendances de contexte explicites (les relations entre les composants et leurs contextes d'exécution). La composition est utilisée pour construire des composants complexes à partir de composants simples.

L'ingénierie à base de composants met l'accent sur la séparation des préoccupations à l'égard des diverses fonctionnalités disponibles dans un système logiciel. Il s'agit d'une approche basée sur la réutilisation et la composition d'éléments indépendants et faiblement couplés dans un système.

Les composants peuvent être décrits à l'aide d'une ou plusieurs approches conventionnelles. Dans notre contexte, ils pourraient être décrits comme métaclasses d'un métamodèle qui décrivent des caractéristiques comme les interfaces, l'encapsulation et l'héritage. Ces caractéristiques et les concepts orientés-objet sont compatibles (orthogonaux), nous verrons dans le chapitre 7 que cela permet de les concevoir plus facilement. Ainsi, une approche dirigée par les modèles pour décrire les composants d'un éditeur de diagrammes, permet d'exprimer les concepts de base d'un langage visuel et ses interactions dans une syntaxe abstraite d'une manière formelle [Hnetynka 2008].

Pour permettre un développement rapide et efficace d'une application par composition de composants, la proposition doit fournir une infrastructure de développement (outillage) qui permet la composition de ces composants et un référentiel pour stocker les composants existants (à la fois les spécifiques et les génériques). Ces infrastructures peuvent être fournies par une approche basée sur les modèles. La composition sera donc effectuée à l'aide des techniques de composition de modèles [Hnetynka 2008].

Nous utilisons quatre opérateurs de composition pour composer et réutiliser, dans un modèle, les composants d'un diagrammes. Nous donnerons ci-dessous un aperçu des différents opérateurs de composition [Pedro 2009] :

- Confinement : cet opérateur permet de créer des relations d'imbrication (Containment) pour fournir des constructions hiérarchiques. La hiérarchie est définie par une relation d'agrégation d'un composant vers un ou plusieurs composants ;
- Association : cet opérateur est très similaire à l'opérateur de confinement en ce qui concerne la composition du modèle. Cependant, au lieu de créer une agrégation, cet opérateur crée une référence entre un composant et un autre ;
- Héritage : cet opérateur permet de composer des modèles avec un mécanisme similaire à l'héritage d'UML. Il précise qu'un composant est spécialisé par un autre. Cette spécialisation permet, par exemple dans la syntaxe graphique, de substituer un symbole graphique par un autre symbole graphique surchargé (redéfini) ;
- Fusion paramétrée : cet opérateur est utilisé comme "point de jonction" pour composer deux espaces d'abstraction. Ceci est souvent effectué par des tissages de liens entre modèles (transformations horizontales).

D'autres travaux sont basés sur les composants pour spécifier des langages visuels [Costagliola 2002]. Ces travaux qui utilisent la notion de composant comme bloc de construction des langages visuels, utilisent une grammaire de position étendues (voir grammaire des graphe - section 5.3) pour definir des bibliothèques de composants réutilisables. L'outil proposé s'appelle VLDesK et offre un éditeur graphique WYSIWYG pour spécifier les notations visuelles et un analyseur syntaxique du langage dans lequel les composants sont implémentés.

Cependant, cette approche souffre de plusieures limitations. Elle utilise une description textuelle pour définir ces composants, ce qui engendre un coût supplémentaire pour apprendre ce langage et écrire l'implémentation de chaque composant sans parler des risques d'erreur et d'incohérence et la difficulté de maintenance de tels composants. En outre, le développeur a besoin de compétences spécifiques pour l'utilisation de l'outil VLDesK, comme l'expertise dans le formalisme étendue de grammaire de position [Costagliola 2004] et YACC [Johnson 1970].

6.3 Architecture générale de notre proposition

La conception globale de notre solution suit une architecture Modèle / Vue / Contrôleurs, qui sépare le rendu visuel des modèles sous-jacents (modèles de domaine représentés par ce rendu), et fournie un moyen pour garder une cohérence entre les rendus visuelles et les modèles spécifiques au domaine.

Pour une meilleure réutilisabilité, nous proposons aussi une séparation claire des préoccupations. Cette séparation a été inspirée d'un ensemble de travaux dans le domaine de l'ingénierie des langages visuels et les interfaces Homme-Machine (IHM) [Moody 2009a, Bottoni 2004, Bertin 1983, Costagliola 2004] et de notre retour d'experience dans le developpement de Papyrus [Gérard 2011].

Nous avons séparé nos métamodèle en trois grandes parties distinctes. La première partie est celle de la syntaxe concrète du langage de modélisation. Cette partie se divise elle aussi en deux parties : une grammaire visuelle et un vocabulaire visuel. La grammaire visuelle sert à décrire la structure d'une notation visuelle via des régles de composition. Les unités élémentaires de construction et de composition d'une notation visuelle, sont inspirés des concepts de graphes (sommets et arêtes) et de notre expérience dans le développement et la définition des diagrammes. Le vocabulaire visuel est constitué des variables graphiques dédiées à la présentation des notations visuelles (Par exemple, la forme, la couleur, les dimensions, etc.). La grammaire visuelle ne comporte donc aucune information sur la manière dont les notations visuelles vont étre présentées. Le vocabulaire visuel ne spécifie que les informations pertinantes pour le rendu des notations visuelles. La définition du vocabulaire visuel est inspiré des variables visuelles de Bertin [Bertin 1983], des éléments du métamodèle Diagram Graphics (DG) [OMG 2012] et de quelques concepts du Framework Spray [Itemis 2012].

La deuxième partie de notre architecture est celle de l'interface (UI) de l'éditeur de diagramme et les interactions offertes à ses utilisateurs. Cette partie sert à décrire, pour chaque élément du langage visuel, les évènements et interactions pris en charge par

l'éditeur. Les interactions ont été inspirées de GEF [GEF 2013], de GMF [Eclipse 2006b] et de e4 [Eclipse 2012].

La troisième partie concerne l'association de la syntaxe concrète (partie Vue) à la syntaxe abstraite du DSML (la source des données). On appelle ce mécanisme le *DataBinding* ou *DomainBinding*. Cette partie est le résultat d'un grand travail de consolidation des retours d'expériences dans Papyrus. Du côté implémentation, cette partie s'inspire du patron de conception Observateur/Observable pour synchroniser les données entre les différentes parties de l'application.

La figure 6.3 montre le lien entre les différents artefacts concernés. Tout d'abord, nous séparons le contenu du domaine (**la syntaxe abstraite** à droite de la figure) de la forme (**la syntaxe concrète** à gauche de la figure) d'un diagramme à un niveau d'abstraction élevé. La syntaxe abstraite est hors du cadre de notre travail, elle est largement traitée dans des outils et des technologies comme EMF / Ecore [Budinsky 2003]. La forme est séparée en deux parties : le vocabulaire visuel (différentes variables de forme, couleur, taille, etc.) et la grammaire visuelle qui décrit les règles de composition du vocabulaire visuel. Le lien entre la syntaxe concrète et la syntaxe abstraite est également précisé dans un modèle de **Binding**. Ainsi, notre proposition est faite de plusieurs méta-modèles, chacun étant utilisé pour décrire une préoccupation : un métamodèle de grammaire visuelle, un métamodèle de style, un métamodèle d'interactions (ne figure pas dans le schéma) et un métamodèle de binding avec le domaine. L'ensemble de ces méta-modèles forme ***MID*** pour **M**éta-modèles pour les **I**nterfaces utilisateur et les **D**iagrammes.

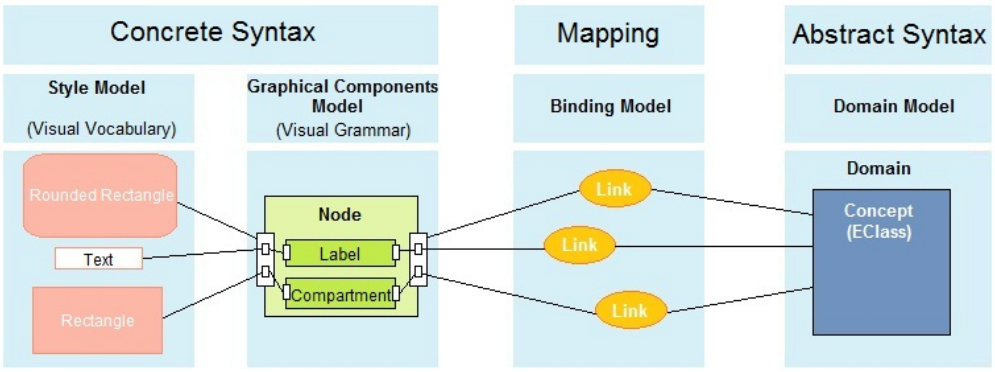


FIGURE 6.3 – MID : les différents artéfacts concernés

La séparation des préoccupations, n'est que le premier élément de réponse pour la problématique de réutilisation. Nous proposons aussi une approche qui permet de réutiliser la description graphique avec un mécanisme très proche de l'héritage dans l'orienté-objet. Un élément de diagramme (appelé composant de diagramme dans toute la thèse), peut hériter d'un autre. Il peut aussi redéfinir la composition, le style et le comportement des composants hérités. Cette fonctionnalité permet la réutilisation des composants ainsi que la création d'autres composants dérivés (spécialisés).

En proposant la séparation des préoccupations avec un mécanisme d'héritage graphique, nous avons trouvé une solution efficace pour la variabilité visuelle (question 5 de la problématique). Il est possible de réutiliser un élément graphique dans deux

diagrammes différents avec deux représentations différentes, simplement en séparant sa composition (grammaire visuelle) de son affichage (vocabulaire visuel) et en utilisant l'héritage et la surcharge pour redéfinir les variations graphiques.

6.4 Synthèse et discussion

MID prend en charge une description des interfaces graphiques pour les éditeurs graphiques de modélisation pour les langages visuels tels que les diagrammes UML, flowchart, petri nets, BPMN, etc. L'objectif de la description est de structurer le langage visuel et les comportements qui y sont associés (dans l'éditeur) tels qu'ils sont perçus par l'utilisateur final.

De point de vue fonctionnelle, **MID** est un ensemble de métamodèles et d'outillages qui permettent de répondre aux aspects suivants :

1. Définir la syntaxe concrète graphique (diagrammes) pour les langages de modélisation (DSML ou autres) ;
2. Effectuer le mapping entre les éléments de la syntaxe concrète et les éléments de la syntaxe abstraite (le métamodèle du domaine) ;
3. Définir l'éditeur qui manipule cette syntaxe concrète graphique en spécifiant les interactions et les outillages nécessaires pour sa manipulation ;
4. Réutiliser les définitions graphiques déjà existantes via :
   - La Séparation des préoccupations (grammaire visuelle, vocabulaire visuel et interactions) pour augmenter les possibilités de réutilisation de chaque partie distinctement des autres ;
   - L'utilisation de l'architecture MVC pour une meilleure cohérence entre les préoccupations ;
   - L'utilisation d'une approche basée sur les composants pour bénéficier des techniques de composition et de la facilité de maintenance de la spécification offertes par cette approche ;
   - Un mécanisme qui permet l'héritage des composants graphiques pour récupérer les descriptions existantes et pour créer d'autres descriptions dérivés de celles-ci.

D'un point de vue global de l'architecture Modèle-Vue-Contrôleur (MVC), MID spécifie la partie vue (View) et les parties de contrôle de l'application et décrit comment la vue dépend du modèle. En particulier :

- En ce qui concerne la vue, MID décrit la composition et la présentation des éléments graphiques des langages visuels qu'il expose à l'utilisateur dans des éditeurs graphiques ;
- En ce qui concerne la partie contrôleur, MID permet au concepteur de spécifier les effets des interactions sur les éditeurs de diagramme en définissant les événements pertinents que le contrôleur doit prendre en charge ;
- En ce qui concerne la partie modèle, MID permet la spécification des références à des objets de données qui incorporent l'état du modèle manipulé et qui sont

exposés dans l'éditeur de diagramme, ainsi que des références à des actions qui sont déclenchées par l'interaction de l'utilisateur ;
- MID décrit la composition des notations visuelles, en fonction de quelques unités de constructions élémentaires et indépendantes, qui peuvent être affichées simultanément et peuvent être imbriqués hiérarchiquement ;
- MID décrit le contenu de la vue, en termes à la fois, de représentation des notations visuelles (attributions des formes et d'autres variables au composant de diagramme) et de synchronisation avec les éléments de modèle (lecture/écriture) ;
- MID permet la définition de l'éditeur graphique manipulant les diagrammes et permettant l'interaction de l'utilisateur avec le langage édité.

CHAPITRE 7

Proposition

Sommaire

Dans l'objectif de spécifier un outil de modélisation graphique, il est nécessaire d'expliciter les concepts de base qui en font partie. Comme nous l'avons montré dans le chapitre 6, nous utilisons une approche dirigée par les modèles pour décrire les concepts élémentaires des langages visuels et leurs éditeurs graphiques dans un ensemble de métamodèles, décrivant chacun une préoccupation de notre système à modéliser.

En plus des informations structurelles du langage visuel, il est donc nécessaire de compléter ces métamodèles par des informations graphiques et des propriétés comportementales des éditeurs générés. Par ailleurs, nous spécifions aussi les liens entre ces informations graphiques/comportementales et la syntaxe abstraite du domaine métier dans un autre métamodèle.

Nous avons séparé nos métamodèle en trois grandes parties distinctes. La première partie est celle de la syntaxe concrète du langage de modélisation. Cette partie se divise elle aussi en deux parties : Une grammaire visuelle et un vocabulaire visuel. La grammaire visuelle sert à décrire la structure d'une notation visuelle via des règles de composition. Les unités élémentaires de construction et de composition d'une notation visuelle, sont inspirés des concepts de la grammaire des graphes et de notre expérience dans le développement et la définition des diagrammes. Le vocabulaire visuel est constitué des variables graphiques dédiées à la présentation des notations visuelles (Par exemple, la forme, la couleur, les dimensions, etc.). La grammaire visuelle ne comporte donc aucune information sur la manière dont les notations visuelles vont être présentées. Le vocabulaire visuel ne spécifie que les informations pertinentes pour le rendu des notations visuelles.

La deuxième partie de notre architecture est celle de l'interface (UI) de l'éditeur de diagramme et les interactions offertes à ses utilisateurs. Cette partie sert à décrire, pour chaque élément du langage visuel, les évènements et interactions pris en charge par l'éditeur.

La troisième partie concerne l'association de la syntaxe concrète à la syntaxe abstraite du DSML (la source des données). On appelle ce mécanisme le *DataBinding* ou *DomainBinding*. Cette partie est le résultat d'un grand travail de consolidation des retours d'expériences dans Papyrus. Du côté implémentation, cette partie s'inspire du patron de conception Observateur/Observable pour synchroniser les données entre les différentes parties de l'application.

Tout au long de ce chapitre nous décrivons notre proposition selon ses différentes préoccupations et nous nous arrêtons sur les solutions que nous avons choisies pour chaque problématique soulevée dans le chapitre 2.

7.1 Métamodèles pour les Interfaces utilisateur et les Diagrammes (MID)

MID est un ensemble de métamodèles qui permettent la conception des langages visuels et les éditeurs graphiques qui les manipulent à un haut niveau d'abstraction. Basés sur des concepts comme les composants, les interfaces et les concepts de l'ingénierie des langages visuels, ces métamodèles couvrent les différentes préoccupations de la syntaxe concrète graphique : la partie du vocabulaire visuel (variables visuelles), la partie structurelle pour définir les règles de composition graphique (grammaire visuelle), la partie comportementale pour la gestion des interactions et l'outillage des éditeurs graphiques sous-jacents à ces langages visuels et la partie de liaison (binding) entre la syntaxe concrète et la syntaxe abstraite.

La partie graphique de notre proposition est composée de deux artefacts : le métamodèle structurel des représentations graphiques (grammaire visuelle), qui représente les règles de composition des symboles visuels et le métamodèle de style (vocabulaire visuel) qui définit les informations élémentaires pour représenter ces symboles (variables visuelles).

La partie comportementale de notre proposition est aussi constituée de deux métamodèles : le premier décrivant les interactions des éléments de diagrammes dans l'éditeur graphique et le deuxième est celui de l'outillage de l'éditeur à savoir la palette, les menus, vue de propriétés, etc.

La dernière partie de notre proposition est celle du mapping entre la syntaxe concrète graphique (les éléments de diagrammes) et la syntaxe abstraite (métamodèle du domaine).

7.1.1 Métamodèle des composants

Le métamodèle des composants (figure 7.1) est le coeur (core) de notre ensemble de métamodèles. Il sert principalement à définir la notion de composant qui est utilisée

dans les autres métamodèles. Il existe trois types d'interface qu'un composant peut avoir : interfaces de domaine, interfaces de style et les interfaces des événements. Les liens appelés *ComponentBinding* servent à faire le lien entre les interfaces. Une interface est utilisée comme point de liaison entre un (sous-)composant et la préoccupation correspondante (la sémantique, les événements, les styles, etc.) avec un couplage faible. Ainsi, elle aide à améliorer la maintenabilité des composants en externalisant et regroupant leurs descriptions dans un endroit unique. La description des liens entre les interfaces et leurs préoccupations correspondantes sont décrites respectivement dans les figures 7.5, 7.13 et 7.21.

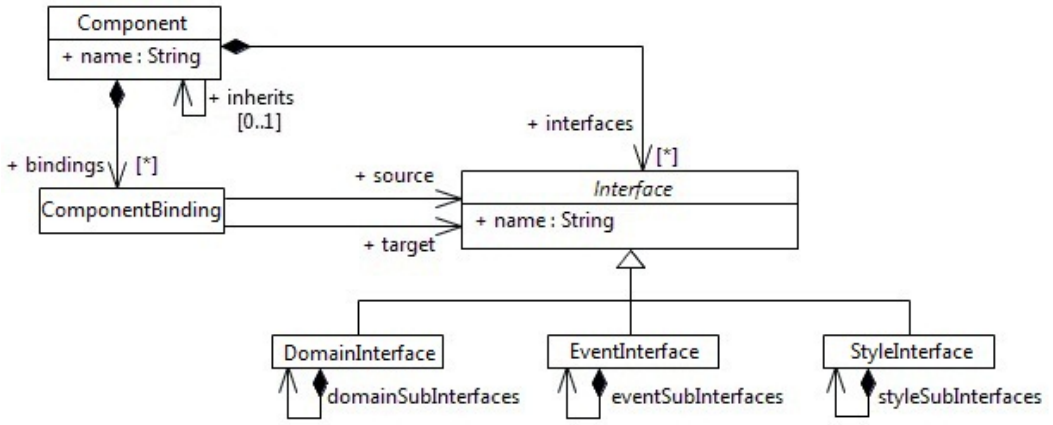


FIGURE 7.1 – Métamodèle des Composants

Les composants ont la capacité d'hériter d'autres composants. Ils peuvent ainsi redéfinir la composition, le style et le comportement des composants hérités. Cette fonctionnalité permet la réutilisation des composants ainsi que la création d'autres composants dérivés. La description complète de cette fonctionnalité est présentée dans la section 7.3 de ce chapitre.

Notre objectif est de spécifier et de composer des diagrammes et leurs éditeurs avec une approche basée sur les caractéristiques des composants. Nous suivons une approche à base de modèles pour la modélisation des composants du diagramme pour résoudre deux problèmes : l'hétérogénéité des composants et leurs techniques de composition.

7.1.2 Primitives communes

Tout au long de notre proposition, et pour définir des langages visuels, nous avons besoin de définir des attributs visuels comme la taille, la couleur, la position, etc. Ces attributs sont utilisés dans plusieurs parties de nos métamodèles, c'est la raison pour laquelle, nous les regroupons dans un package commun. La figure 7.2 présente ces principales primitives communes à l'ensemble des métamodèles de notre proposition.

Le type *Point* est utilisé pour spécifier une coordonnée x,y à partir de l'origine du repère. Les coordonnées de l'origine du repère sont le point (0,0), elles augmentent vers

la droite pour les abscisses (axe des x) et vers le bas pour les ordonnées (axe des y). *Offset* est une donnée qui définie le décalage en x et y (appelés respectivement *xoffset* et *yoffset*) par rapport aux dimensions de la figure hôte.

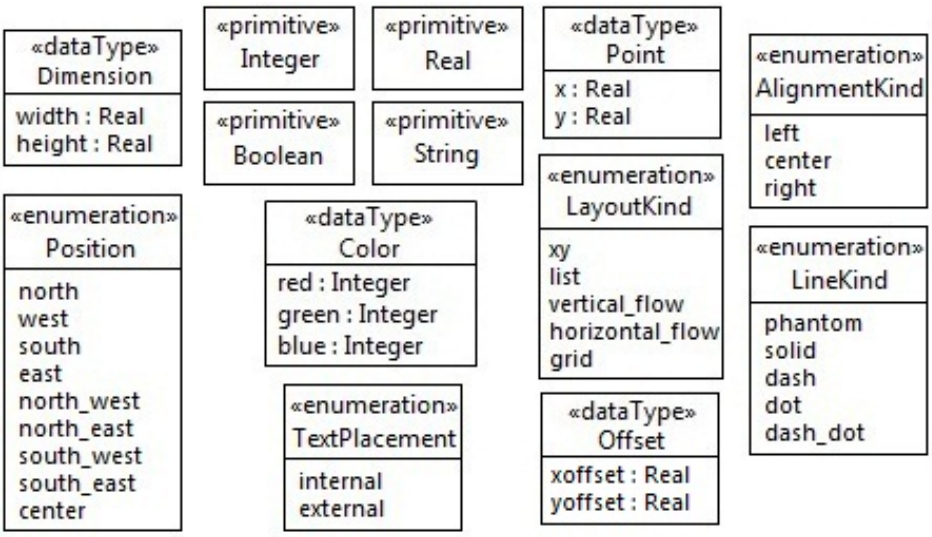


FIGURE 7.2 – Eléments communs : Type de données et primitives

La donnée *Color* est utilisée comme attribut représentant la couleur. Cette valeur entière (de 0 à 255) est calculée en utilisant le format RGB (Red :rouge, Green :vert, Blue :bleu). Par exemple, le jaune est calculé comme (red=255, green=255, blue=0).

AlignmentKind énumère les possibilités d'alignement d'un layout ou d'un texte. *LayoutKind* énumère les possibilités de layout offertes par le métamodèle. Le layout XY est une zone libre de dessin, utilisé souvent dans les diagrammes et les packages. Le layout List est utilisé pour un affichage en liste textuelle de données (par exemple, pour afficher les attributs dans une classe UML). Les layouts *vertical_flow* et *horizontal_flow* servent respectivement à structurer les éléments verticalement et horizontalement dans la figure hôte (Par exemple, les éléments d'une classe UML sont structuré avec un layout *vertical_flow* dans l'ordre : Label, Compartiment des attributs, Compartiment des opérations, etc).

TextPlacement est une donnée qui spécifie le placement d'un texte par rapport à la figure hôte (à l'extérieur de la figure ou à l'intérieur). *LineKind* spécifie la texture d'une ligne ou d'une bordure : phantom pour une ligne invisible, solid pour une ligne simple, dash pour une ligne en traits, dot pour une ligne en pointillés et *dash_dot* pour une ligne en traits et pointillés.

7.1.3 Grammaire visuelle : Régles de composition

La grammaire visuelle permet de décrire la structure des éléments de diagrammes indépendamment du rendu de ceux-ci (figures et formes). Cette description est hiérarchique : un élément racine peut contenir d'autres éléments.

Le métamodèle dans la figure 7.3 représente les concepts principaux de la grammaire visuelle d'un diagramme. Il permet de concevoir tous les langages hybrides [Bottoni 2004], que nous utilisons dans le domaine du génie logiciel comme UML, SysML, BPMN, les graphes d'état, etc. Ce métamodèle comporte les concepts pour décrire des langages de type entité-relation, en plus d'autres concepts visuels comme l'adjacence et le confinement. Cette partie répond aux deux premières questions du chapitre 2.

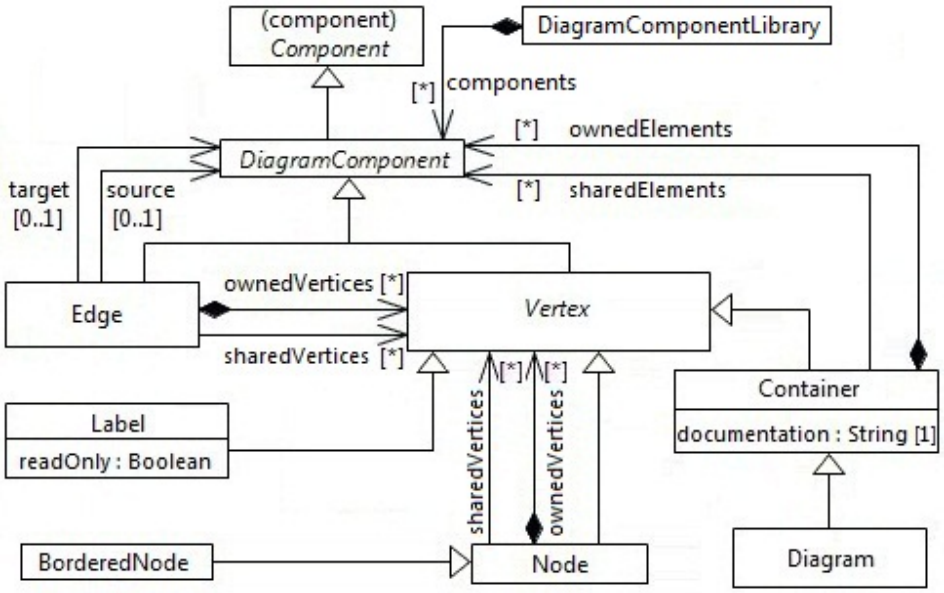


FIGURE 7.3 – Eléments du diagramme

Le composant de diagramme (*DiagramComponent*) est le super type abstrait de tous les éléments graphiques, y compris les diagrammes eux-mêmes. Il hérite directement du concept *Component* vu précédemment. Un composant de diagramme peut être utilisé seulement pour son propre rendu (c'est-à-dire purement notationnel) ou plus communément utilisé comme une représentation d'un élément de la syntaxe abstraite du domaine (par exemple, un modèle UML). Dans ce dernier cas, les composants de diagramme représentent les éléments du modèle et définissent les propriétés de la grammaire visuelle pour ces éléments (par exemple, la composition, les éléments contenus autorisés, etc.).

Comme leur nom l'indique, la spécification des langages Entité-Relation est basée sur la description des entités avec leur composition et les relations entre ces entités. Par analogie aux graphes, ces langages sont en quelques sortes, des graphes composés de sommets ou noeud (entités) et d'arêtes (relations).

Nous proposons ainsi, deux types principaux de composants de diagramme : les sommets (*Vertex*) pour représenter les éléments complexes des diagrammes (les entités) et les arêtes ou connections (*Edge*) pour représenter les liens entre les éléments complexes.

Le concept de *Vertex* est la formalisation d'une figure [Fondement 2008]. Il est le super type abstrait des noeuds (section 4.6) et peut être représenté de différentes manières. Il peut être un noeud principal (noeud supérieur) qui représente l'entité graphique prin-

cipale dans un diagramme, un sous-noeud (noeud appartenant/partagé à/par d'autres noeuds représenté respectivement par *containedVertices* et *sharedVertices* dans la figure 7.3) ou un noeud de bordures (*BorderedNode* : noeud qui peut être apposé sur d'autres noeuds). Une étiquette (*Label*) est un noeud qui permet de spécifier des éléments textuels de noeuds et de leur associer des accesseurs (getters et setters). Cela permet de synchroniser le modèle de données avec la valeur du texte représenté. Un Label peut être en lecture seul (Par exemple, les labels qui affichent les stéréotypes UML).

Les compartiments ou Conteneurs (*Container*) sont des noeuds spécifiques qui servent à structurer, contenir et/ou référencer (*sharedElements*) d'autres éléments. Un diagramme est lui-même un conteneur de composants (*DiagramComponent*). Un diagramme peut faire référence à un élément de modèle à partir d'une syntaxe abstraite, dans ce cas le diagramme entier est considéré comme une représentation de l'élément (par exemple, un diagramme d'activité est une représentation d'une activité UML).

Une arête (*Edge*) est une connection entre les éléments du diagramme. La relation entre l'élément source (*source*) et celui de destination (*target*) peut être spécifiée sémantiquement (dans le métamodèle de domaine). Par exemple, la relation de généralisation d'UML est définie dans le métamodele du langage. Elle indique l'élément source (Generic) et l'élément destination (Specific).

Un lien pourrait être purement graphique, c'est-à-dire qu'il ne fait pas de référence à un élément du modèle (Par exemple, le lien de fixation d'un Commentaire dans UML). Par ailleurs, un lien pourrait être plus complexe qu'une simple ligne (par exemple, la figure 7.4 montre un bus de données qui lie deux circuits électroniques).

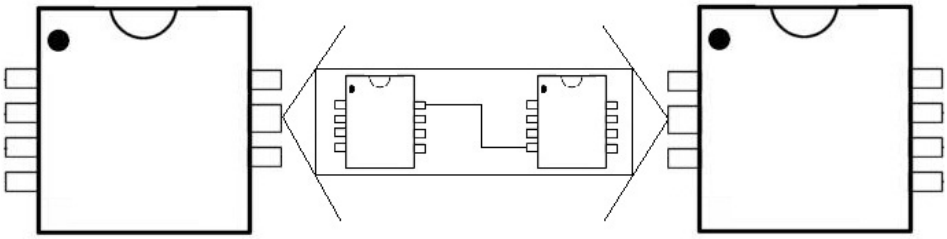


FIGURE 7.4 – Lien composite entre deux éléments graphiques

Les composants de diagramme sont associés, via leurs interfaces de style, à un vocabulaire visuel dont nous décrivons le métamodèle dans la section suivante.

7.1.4 Vocabulaire visuel : Variables visuelles

Comme on l'a mentionné dans le chapitre 6, nous avons séparé la description des langages visuels en deux parties : Une grammaire pour décrire la composition des éléments graphiques et le vocabulaire visuel qui permet de décrire les symboles graphiques des éléments de diagramme. Cette description est composée de différentes primitives graphiques (variables visuelles de Bertin [Bertin 1983] représentant les unités atomiques de construction des notations visuelles), nous les regroupons toutes dans le concept de *Style* (figure 7.5), qui contient des propriétés de mise en forme (exemple, la forme, la cou-

leur, la police de caractères, etc.) et qui affecte l'apparence des éléments du diagramme, y compris les diagrammes eux-mêmes.

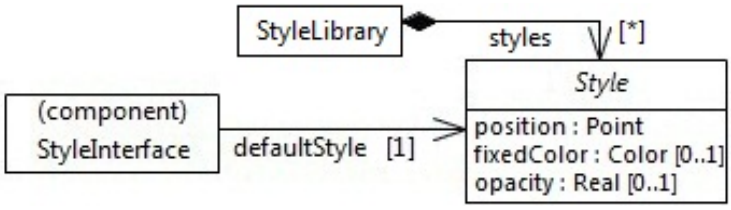


FIGURE 7.5 – Le concept de Style

Tous les composants de diagramme sont associés aux vocabulaires visuels via leurs interfaces de style. Ainsi un style pourrait être partagé par plusieurs éléments du diagramme. Comme toutes autres caractéristiques des éléments graphiques, cette relation peut être réutilisée et surchargée par l'intermédiaire du mécanisme proposé d'héritage (section 7.3).

7.1.4.1 Style

Le style représente la représentation visuelle d'un élément du diagramme. Comme nous l'avons mentionné dans le chapitre 6, nous avons séparé la composition graphique (grammaire visuelle) des éléments du diagramme de leurs représentations (vocabulaire visuel). Une des raisons de cette séparation est la problématique de la variabilité visuelle des éléments du diagramme (question 5 du chapitre 2). Ainsi, on peut attribuer différentes représentations pour le même élément graphique (la même composition avec différentes représentations).

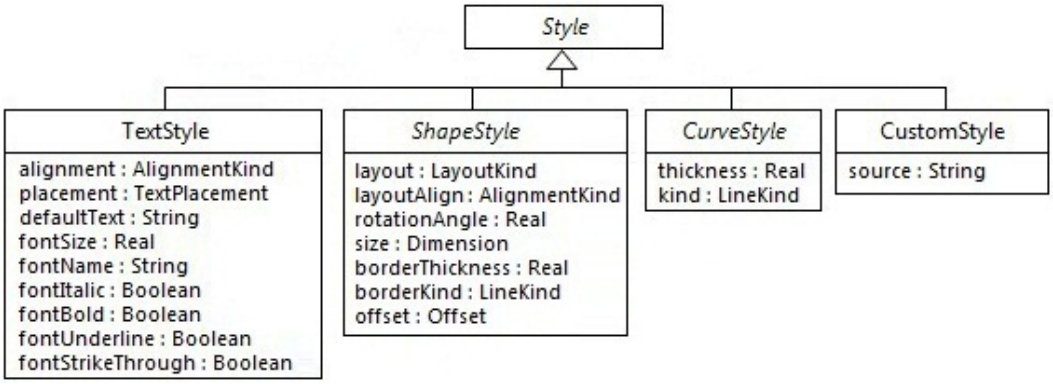


FIGURE 7.6 – Les Styles de base

Même s'ils sont séparés, la représentation d'un élément graphique est définie selon la nature de sa composition (les différents types de la grammaire). Une grammaire est composée d'entités (noeuds), d'élément textuels (labels) et de liens. Le style se décompose ainsi en quatre grandes catégories (figure 7.6) : style de figure (*ShapeStyle*) qui représente

l'unité d'information graphique utilisée pour construire la forme de la notation visuelle. C'est la super-classe abstraite de toutes les formes de noeuds. Le style de texte (*TextStyle*) est une représentation graphique qui restitue une séquence de caractères au sein d'une boite (étiquette). Les styles de texte précisent l'alignement des données affichées, leurs positions et des informations sur la taille, la couleur et la police de caractères utilisée. Le style de courbe (*CurveStyle*) représente la définition graphique d'une connection (un lien sous forme de courbe). C'est la super-classe abstraite de tous les styles de connection. Il est aussi possible de créer des styles personnalisés avec (*CustomStyle*) avec une implémentation dans le code.

7.1.4.2 Style de figure

Les styles de figure (*ShapeStyle*) représentent les formes (des formes 2D, 3D et images) (figure 7.7). Ils sont caractérisés par l'attribut de structure (*Layout*), qui représente les différentes règles d'arrangement dans la figure, la taille (une dimension), la rotation de la figure en degrés (*rotationAngle*) et les propriétés de la bordure (épaisseur et type de la bordure). Il est aussi possible de créer des styles avec des images PNG, JPEG ou SVG (avec l'URL du fichier défini en paramètre).

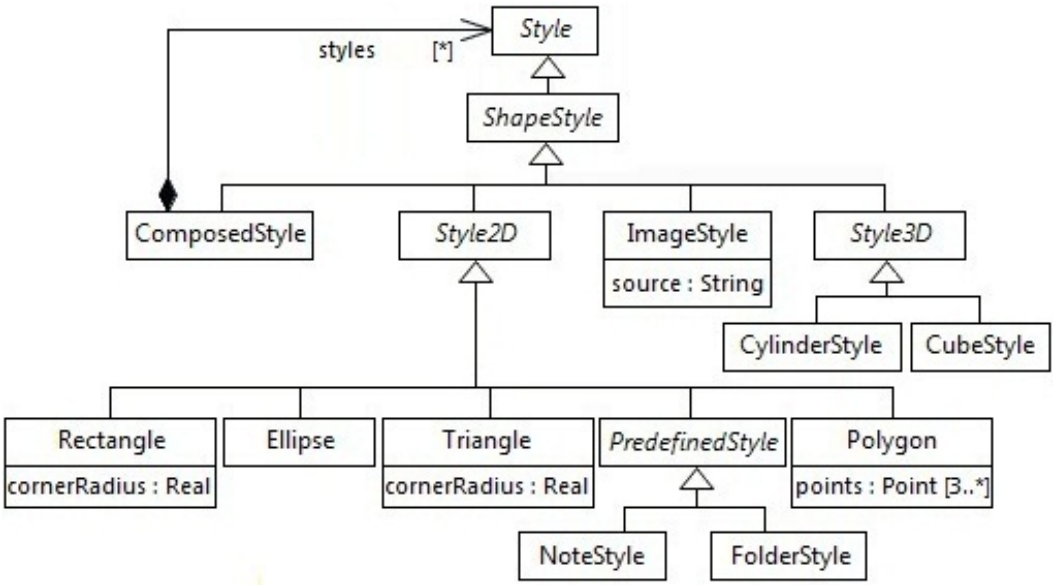


FIGURE 7.7 – Les Styles de forme

Les Styles en 3D (*Style3D*) représentent des formes avec des rendus en trois dimensions (par exemple, le cube, le cylindre, etc). Les Styles en 2D (*Style2D*) proposent une dizaine de formes de base comme *Rectangle* qui permet de définir une forme rectangulaire, *Ellipse* pour les formes elliptiques, *Triangle* pour les formes triangulaires et quelques formes prédéfinies (PredefinedStyle) comme les *Notes* (par exemple, les commentaires dans UML) et les *Folders* (similaire aux formes de package UML). Les formes comme *Rec-*

tangle et *Triangle* peuvent avoir des angles arrondis en définissant l'attribut *cornerRadius* (un cornerRadius égale à 0 définit une forme avec des angles droits).

PolygonStyle permet à l'utilisateur de définir ses propres formes. Dans ce concept, l'utilisateur définit une forme constituée d'une séquence de segments de ligne droite définis par une liste d'au moins trois points. Le résultat est une forme fermée. Le dernier segment est automatiquement défini entre le dernier point et le premier.

Il est aussi possible de créer des formes imbriquées en utilisant le concept (*ComposedStyle*). Ce concept permet aux utilisateurs de combiner plusieurs formes dans un seul afin d'obtenir des formes plus complexes et plus adaptées à leurs besoins.

7.1.4.3 Style de courbe

Pour représenter des liens sous forme linéaire, nous avons proposé le concept de *Courbe* qui généralise les représentations de lignes (une ligne étant une courbe particulière). On peut imaginer aussi des liens sous forme de courbes de bézier ou d'arcs. Un style de courbe (*CurveStyle*) définie une forme personnalisée d'un lien sous forme de courbe (figure 7.8). Une courbe peut être un arc (*ArcStyle*), une courbe de bézier (*BezierCurveStyle*) ou une simple ligne (*LineStyle*).

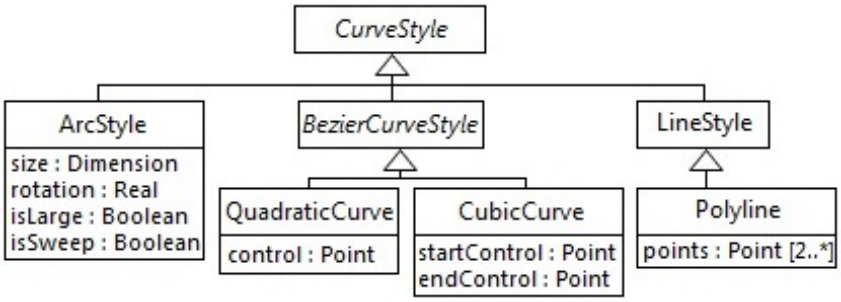


FIGURE 7.8 – Les Styles de courbes

Un arc est une courbe définie par un segment d'un ellipse. Il est spécifié par la taille et la rotation en degrés de l'ellipse à partir duquel l'arc est créé.

Une courbe de bézier [Gilbert 1998] est une courbe polynomiale proposée pour la première fois en 1962 par Pierre Bézier qui les utilisait pour concevoir des pièces mécaniques à l'aide d'ordinateurs. Il existe deux types de courbes de bézier dans ce métamodèle, la courbe cubique (*CubicCurve*) qui est caractérisée par deux points de controle (un pour la source : *startControl* et l'autre pour la destination : *endControl*) et la courbe quadratique (*QuadraticCurve*) qui utilise un seul point de controle.

Un style de ligne (*LineStyle*) définie l'élément graphique d'un segment de ligne entre une figure source et une figure destination. La ligne est l'une des formes possibles pour un lien ente deux éléments de diagramme. Tout lien peut avoir une décoration (*endDecoration*) de source, de milieu et de destination comme des flèches (constitués de polylines : éléments constitués d'une séquence de segment de ligne définis par deux ou plusieurs points) ou des formes (*ShapeStyle*). Ces décorations peuvent être positionnées

sur un lien avec un attribut angle (figure 7.9). L'attribut *angle* sert à décrire la rotation de la décoration en degrés par rapport au lien hôte.

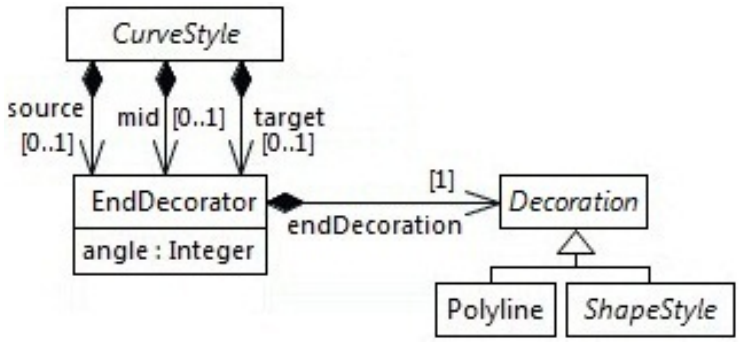


FIGURE 7.9 – Les extremités de liens

7.1.4.4 Points d'attache

Les points d'attache (*Anchorage*) est un autre aspect important dans la conception des figures. A ces points de fixation des liens peuvent être attachés. À l'exception de Spray [Itemis 2012] (à partir duquel nous avons été inspiré pour ce concept), les outils existants ne propose pas de définir ses propres points d'attache, la plupart accroche les liens dans la bordure de l'élément source ou destination.

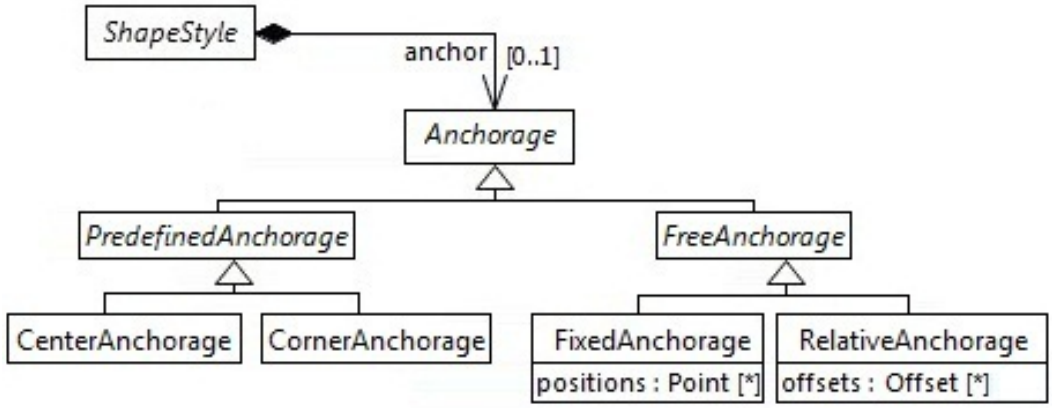


FIGURE 7.10 – Les points d'attache sur les figures

Dans notre métamodèle il y a quatre façons pour la définition des points d'attache. Il existe deux types prédéfinis qui sont appelés *CenterAnchorage* et *CornerAnchorage* (figure 7.10). La définition d'un point d'attache dans le centre (*CenterAnchorage*) fixe l'extrémité du lien au centre de la figure, mais la ligne se termine sur la bordure de cette figure. Le deuxième point d'attache prédéfini est celui du coin qui crée des points de fixation sur tous les coins de la figure, ainsi que sur le centre des quatre côtés de la figure. Un exemple est montré sur la figure 7.11.

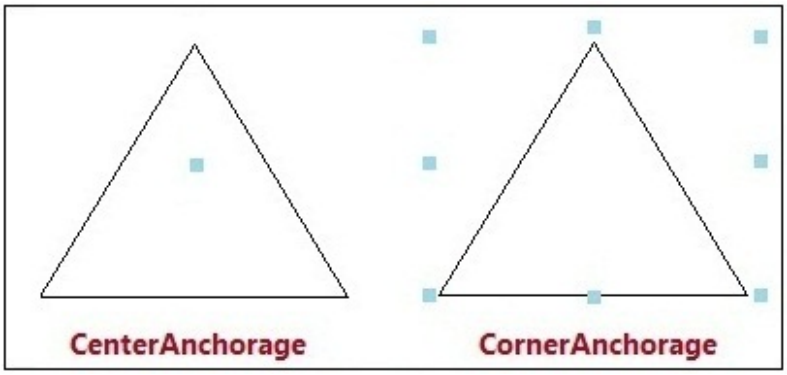


FIGURE 7.11 – Les points d'attache prédéfinis

D'autres points d'attache peuvent être définis sur une figure en utilisant des points spécifiques (*FixedAnchorage*) ou en utilisant la position relative (décalage en X et décalage en Y, appelés *offsets* avec le concept *RelativeAnchorage*. Cette catégorie est utile lors de la définition d'éléments graphiques avec une très grande expressivité graphique c'est-à-dire qu'on exige que le lien ne soit attaché que dans un point précis (par exemple, les schémas électriques - figure 9.7). En utilisant *FixedAnchorage*, les points de fixation sont placés dans la position définie à l'intérieur de la figure dessinée. Les points d'attache relatifs (*RelativeAnchorage*) sont positionnés par rapport à la taille de la forme avec des valeurs comprises entre 0.0 (le côté gauche pour les x et le côté haut pour les y) et 1.0 (le côté droit pour les x et le côté bas pour les y) (figure 7.12).

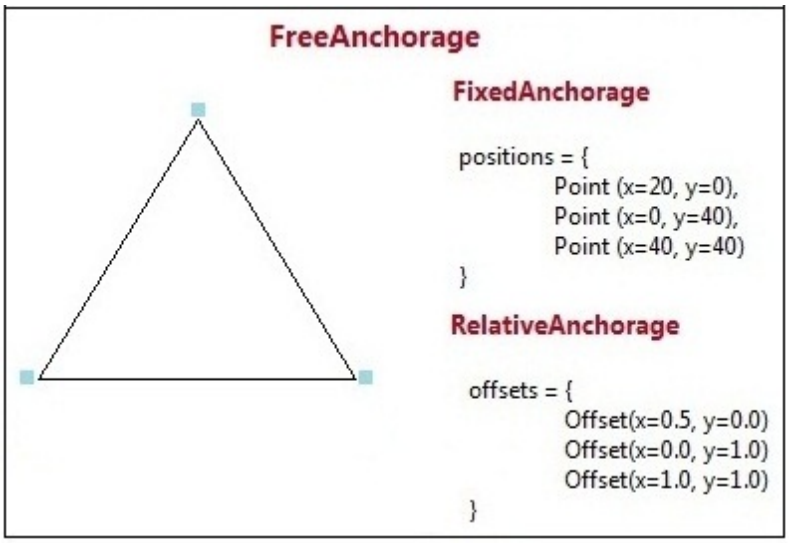


FIGURE 7.12 – Points d'attache relatifs

7.1.5 Comportements et interactions

Une fois le langage visuel défini (avec une grammaire et un vocabulaire visuels), il est possible de l'intégrer dans un éditeur graphique qui va permettre sa manipulation. Un éditeur sert ainsi à interagir avec les utilisateurs du langage en leurs offrant les outils nécessaires à sa manipulation. Les interactions des utilisateurs influencent d'une manière effective sur l'aspect graphique du langage visuel (sa structure et sa présentation) et sur le modèle métier associé (éléments du modèle domaine associés à ce langage).

Ce métamodèle sert à répondre à la question 6 du chapitre 2 pour décrire, pour chaque élément du langage visuel (*DiagramComponent*), les évènements et interactions autorisés (prises en charge) par l'éditeur sur ces éléments graphiques. Les éléments de ce métamodèle sont le résultat d'analyse et de recensement des besoins en termes d'interactions dans Papyrus.

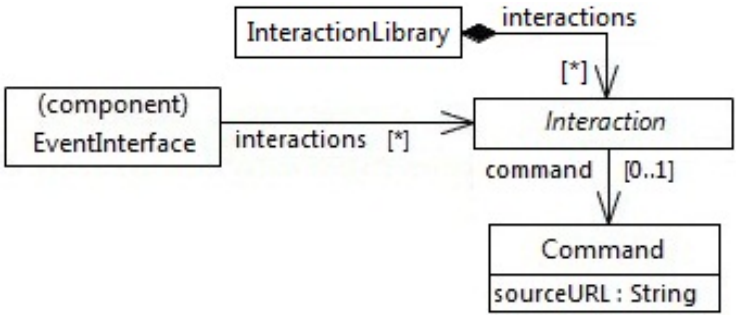


FIGURE 7.13 – Les Interactions

Tous les composants de diagramme sont associés aux interactions via leurs interfaces d'évènement (figure 7.13). Ainsi une Interaction peut être partagée par plusieurs éléments du diagramme. Comme toutes autres caractéristiques des composants, cette relation peut être réutilisée et surchargée par l'intermédiaire du mécanisme proposé d'héritage.

Une interaction est définie comme un changement invoqué par l'utilisateur sur le modèle de domaine. Il existe deux grandes catégories d'interactions dans les éditeurs graphiques [Hudson 2003]. La première, est l'action (*Action*). Les actions apparaissent généralement dans les menus, toolbars, popups et fonctionnent généralement sur la sélection actuelle. (Par exemple, la suppression, l'alignement, show/hide, etc.)

L'autre type d'interaction est graphique. Les interactions graphiques (*GraphicalInteraction*) sont celles effectuées sur un élément de diagramme à l'aide d'une souris (ou un clavier). Ces interactions sont traitées par l'éditeur.

Il est important de souligner que 99% des interactions sont traitées sous forme de commandes qui modifient le modèle (figure 7.13), à l'exception des données non-persistantes ou des données qui ne font pas partie du modèle, telles que l'expansion d'un élément (expand), le niveau de zoom, etc.

7.1.5.1 Interactions graphiques

Les interactions graphiques (*GraphicalInteraction*) définissent les comportements des éléments graphiques suite à une action avec la souris ou avec des touches spéciales du clavier (figure 7.14). Il existe des interactions qui interviennent sur des données non-persistantes. Par exemple, le *Resize* qui permet de définir le mode de redimensionnement de l'élément graphique, le *ExpandCollapse* qui permet d'étendre ou rétrécir une liste graphique, etc. Il est aussi possible de créer ses propres interactions personnalisées (*CustomGraphicalInteraction*) en implémentant du code.

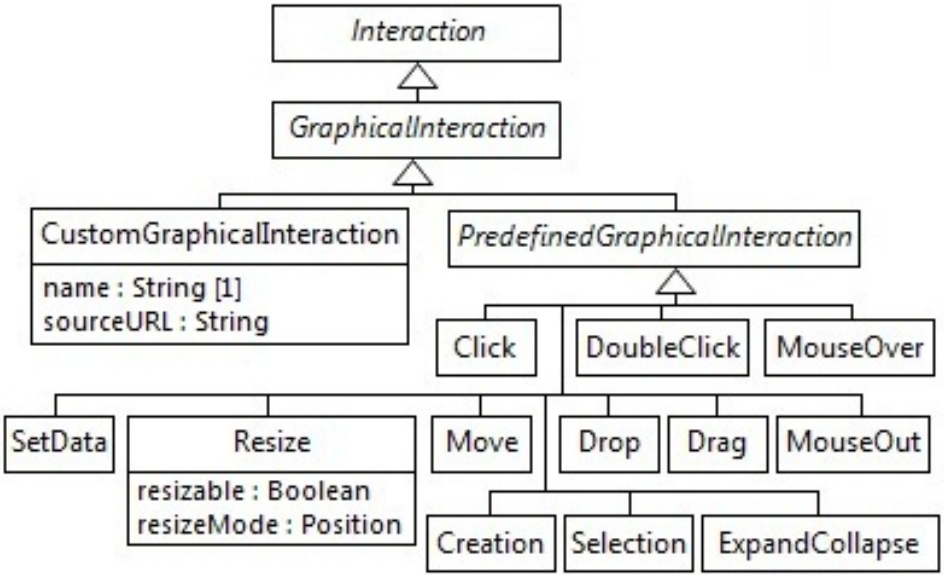


FIGURE 7.14 – Interactions Graphiques

D'autres interactions graphiques nécessitent l'implémentation d'une commande qui effectue des modifications sur l'élément graphique ou le modèle. Par exemple, l'interaction *Selection* permet de définir le comportement lors d'une sélection d'un élément graphique, *SetData* définie les actions à effectuer lors d'une édition d'information dans le diagramme, *Click* et *doubleClick* spécifient les actions à déclencher lors d'un clic simple ou double de souris et le *Drag* qui offre la possibilité de faire glisser un élément graphique dans un autre élément autorisé à l'accueillir. Ce dernier doit déclarer l'interaction *Drop* qui montre que l'élément est susceptible d'accueillir des éléments glissables (Drageable).

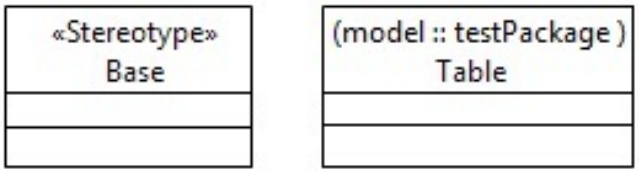


FIGURE 7.15 – Création dynamique de labels dans une Classe UML

Parmi les interactions nécessitant une modification ou une consultation des va-

leurs du modèle, les interactions spécifiques à la grammaire visuelle. Ce cas est souvent rencontré lors de la création dynamique des éléments de diagramme (figure 7.15). Par exemple, lors de l'application d'un métaclasse sur une classe UML, un label est créé dynamiquement comportant le stéréotype de cette métaclasse. Aussi, lors de la création d'une classe UML, il est possible d'afficher ; selon une condition ; un label comportant le *qualifiedName*.

Ceci nous a poussé à proposer le concept *DynamicVertex* qui sert à décrire un élément créé dynamiquement selon une *Condition*. Nous avons aussi ajouté la référence *displayCondition* à l'élément *DiagramComponent* pour décrire les éléments affichés selon des conditions exprimées en *Expression* (figure 7.16). Une expression est une implémentation d'une condition (assertion) ou d'une interrogation (query) définies par un code (body) et un langage (language).

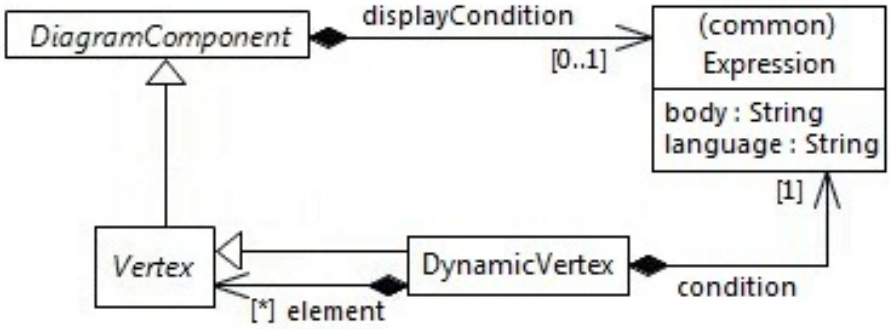


FIGURE 7.16 – La dynamicité dans la grammaire visuelle

Par ailleurs, on peut trouver ce même type d'interaction dans le vocabulaire visuel, dans le cas où la représentation d'un élément dépend d'une valeur dans le modèle (figure 7.17). Par exemple, une association UML est représentée par défaut par une simple ligne. Lorsque la valeur de l'attribut *aggregation* est égale à shared, la représentation de l'association change vers une ligne avec un losange blanc dans l'extrémité de la source. Si la valeur de cet attribut est égale à composite, la représentation sera une ligne avec un losange plein.

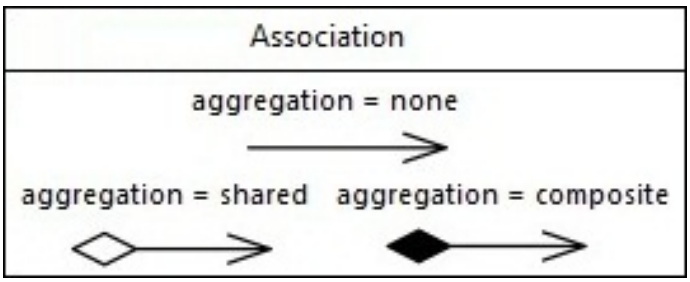


FIGURE 7.17 – Variation de la représentation graphique d'une Association UML

Nous avons ajouté donc ces interactions dans le métamodèle du vocabulaire visuel (figure 7.5). Nous avons ajouté le concept *DynamicStyle* qui sert à décrire une représenta-

tion créée dynamiquement selon une *Condition*. Nous avons conçu ce concept comme un test à choix multiple (Switch/Case) pour définir les différentes possibilités d'affichage d'un élément graphique. Les conditions sont exprimées en *Expression* (figure 7.18).

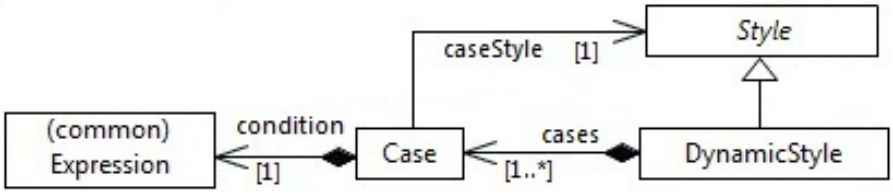


FIGURE 7.18 – La dynamicité dans la vocabulaire visuel

7.1.5.2 Actions

Le métamodèle (figure 7.19) définie les concepts de base des actions et leurs attributs. Les actions (*Action*) apparaissent généralement dans les menus, toolbars (les barres d'outils), popups (menus contextuels) et fonctionnent généralement sur la sélection actuelle. Nous utilisons les actions pour définir les entrées de menus sur un élément de diagramme ou sur le diagramme tout entier. Parmi les exemples d'utilisation de ce concept, les menus contextuels permettant d'effectuer un traitement sur la sélection courante dans un diagramme.

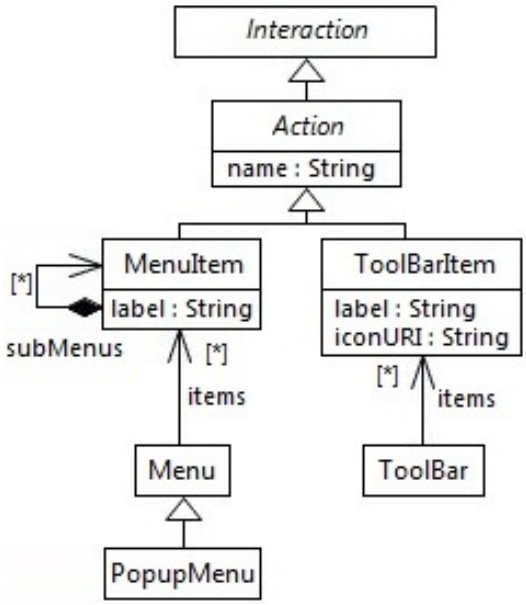


FIGURE 7.19 – Les actions

Le concept d'action est souvent associé à une commande (Command). Une commande est une implémentation qui définie le comportement de l'action (à ne pas

confondre avec les commandes Eclipse [1]).

Un menu (*Menu*) est une liste d'actions ou d'options comportant un comportement et disponible dans un contexte défini [MSDN 2013]. Les menus sont souvent affichés à partir d'une barre de menus, qui se trouve généralement dans la partie supérieure de l'éditeur. En revanche, un menu contextuel (*PopupMenu*) se déroule lorsque l'utilisateur fait un clic-droit sur un élément de diagramme ou une zone de l'éditeur qui prend en charge ce menu contextuel.

Les barres d'outils (*ToolBar*) peuvent être plus efficaces que les barres de menus, car elles offrent un accès direct et immédiat. Elles ont le même rôle que les autres actions, mais elles ont l'avantage d'être facilement reconnaissables et identifiables : les utilisateurs se souviennent des attributs des boutons de la barre d'outils, comme l'emplacement, la forme et la couleur.

7.1.5.3 Palette de création

En réponse à la question 10 de la problématique, ce métamodèle définit quels sont les outils disponibles dans l'éditeur graphique. L'outil le plus important est l'outil de création (*palette*), qui permet d'instancier un élément du métamodèle, qu'il soit représenté par un noeud ou une connection. Les concepts reliés à la création sont représentés dans la figure 7.20.

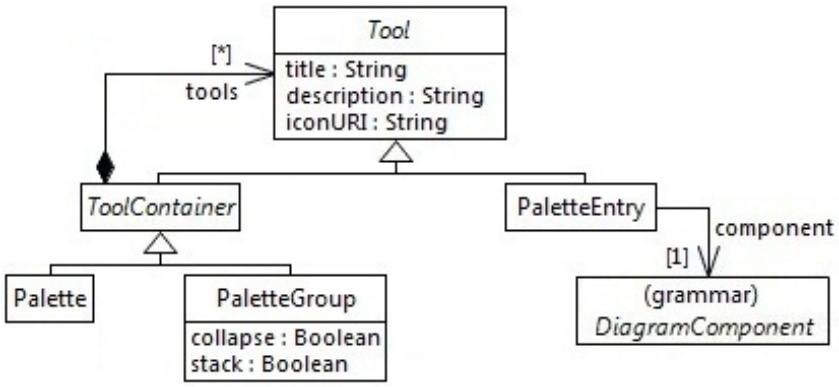


FIGURE 7.20 – Le concept de Palette

La palette d'outils (*Palette*) est composée d'entrées de création (*PaletteEntry*) ou de groupes d'entrées (*PaletteGroup*). Des icônes (*iconURI*) peuvent être définies pour les éléments de la palette. La spécification de la palette est facultative. Dans le cas où elle n'est pas définie explicitement, elle est générée par défaut à partir des éléments spécifiés dans le diagramme qui déclarent l'interaction *Creation*.

1. Une commande dans Eclipse est une est une description déclarative d'un composant et est indépendante des détails d'implémentation.

7.1.6 Lien entre les composants de diagramme et la syntaxe abstraite

Dans notre contexte, un langage de modélisation (Lm) est défini selon le tuple AS, CS^*, M^*_{ac} où AS est la syntaxe abstraite, CS^* est la (les) syntaxe(s) concrète(s) et M^*_{ac} est le mapping de syntaxe abstraite vers sa (ses) représentation(s) concrète(s) [Jezequel 2012].

Le métamodèle d'association ou de binding (figure 7.21) permet d'associer les différents éléments graphiques (noeuds et liens) aux éléments de la syntaxe abstraite du domaine (métamodèle en Ecore).

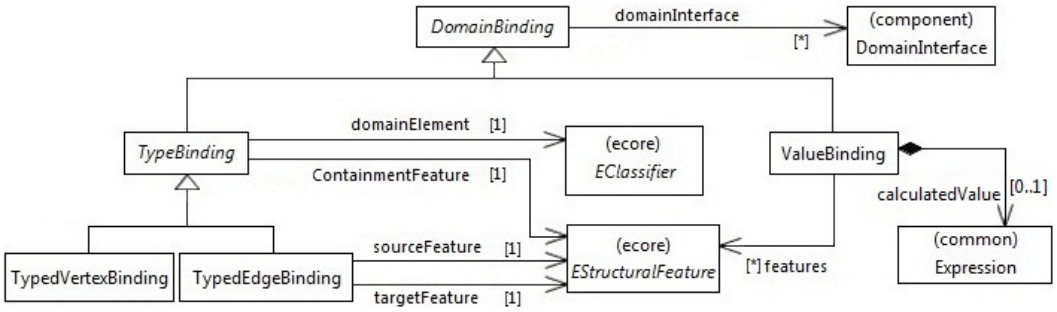


FIGURE 7.21 – Liens entre la syntaxe abstraite et la syntaxe concrète

Le premier besoin était naturellement d'associer *UN* élément de la syntaxe abstraite à *UN* élément graphique de la syntaxe concrète. Ce type d'association est le plus commun et le plus simple dans la partie binding. Pour cela, nous avons proposé le concepts *TypeBinding* fait référence à des concepts (**EClassifier**) qui peuvent représenter un objet de la syntaxe abstraite, un fichier XML, une table dans une base de données, etc. *ValueBinding* associe un élément graphique (label par exemple) à une propriété accessible (**EStructuralFeature**) comme un champ d'une base de données, un attribut d'un objet ou une référence entre deux éléments, etc. Un ValueBinding peut être associé à une expression, qui détermine la valeur calculée spécifique à obtenir de la source de données.

Le concept *TypeBinding* est un concept abstrait pour définir un élément typé. Il peut être *TypedVertexBinding* qui sert à lier un noeud avec un concept (**EClassifer**) et la collection qui le contient (*containmentFeature*) ou un *TypedEdgeBinding* qui permet de lier un lien typé (association sous forme de classe) à son concept correspondant ainsi que la spécification de la source et la destination de ce lien.

Le deuxième besoin en termes d'association avec la syntaxe abstraite est d'associer *UN* élément du domaine à *PLUSIEURS* éléments graphiques (par exemple une lifeline dans le diagramme de séquence). Pour effectuer cette association avec *MID*, plusieurs solutions se proposent : 1) pour les éléments groupés, nous déclarons une interface de domaine unique pour cet ensemble d'éléments. 2) pour les éléments répartis, nous associons l'élément du domaine à toutes les interfaces de domaine de ces éléments.

Le troisième besoin en termes d'association avec la syntaxe abstraite est de pouvoir associer *PLUSIEURS* éléments du domaine à *UN* élément graphique. Dans ce cas on passe par une *ValueBinding* associée à une expression, qui détermine le contenu spécifique à obtenir des différents éléments du domaine. Parmi les exemples d'utilisation

de ce concept, on trouve l'élément *InstanceSpecification* qui comporte dans son label la syntaxe suivante : **<nom de l'instance> : <nom de la classe>**. Ce label sera donc associé à une *ValueBinding* avec une expression qui lie les deux éléments du domaine (nom de l'instance spécification et la classe). Ainsi, nous avons pu répondre aux questions 7, 8 et 9 de la problématique avec ce métamodèle.

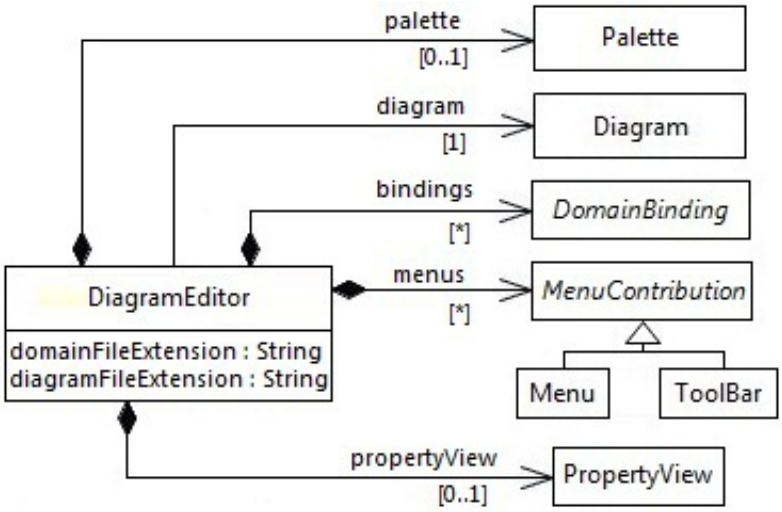


FIGURE 7.22 – Assemblage d'un éditeur de diagramme

Dans la version actuelle, la description des associations avec la syntaxe abstraite sert comme point d'entrée à la description complète de l'éditeur de diagramme (figure 7.22). Ceci est représenté par l'élément *DiagramEditor* qui contient l'ensemble des bindings. Ce concept qui est associé à un diagramme, permet d'ajouter des outillages comme la *Palette*, les menus et une vue de propriétés (*PropertyView*) pour éditer et paramétrer les éléments du diagramme.

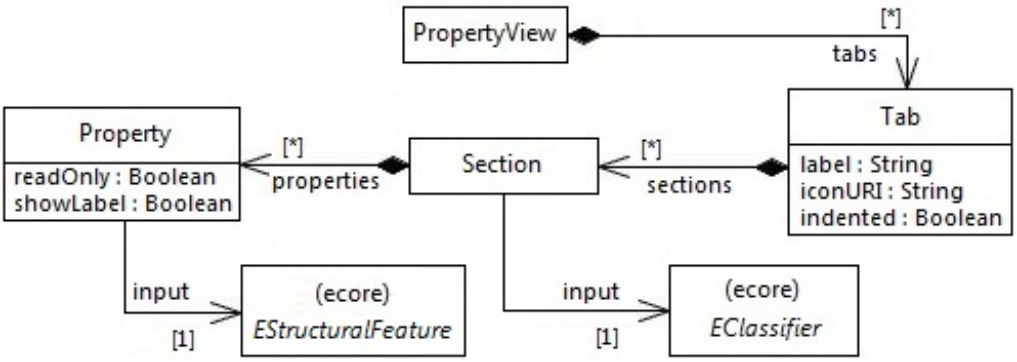


FIGURE 7.23 – Vue de Propriétés

La vue de propriétés (*PropertyView*) permet de visualiser et/ou éditer les propriétés d'un élément sélectionné. La vue de propriétés est associée à un contributeur de propriétés (ici le modèle manipulé par l'éditeur de diagramme) qui contribue à un ou

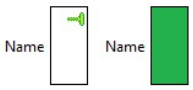
plusieurs onglets (*Tab*) de propriétés. Chaque onglet est rempli avec une ou plusieurs sections (*Section*). Une section est un élément composite contenant un groupe de widgets (SWT, XWT ou autre) qui correspondent à l'ensemble des propriétés (*Property*) d'un concept (**EClassifier**). Une propriété (figure 7.23) est un élément d'un concept (attribut ou référence) associée à un *input* (**EStructuralFeature**) et représentée par le widget correspondant à son type. Par exemple, un attribut en chaine de caractères (String) est représenté par un textbox qui permet d'éditer la valeur de cet attribut.

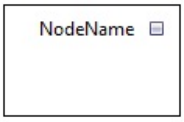
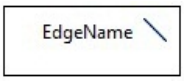
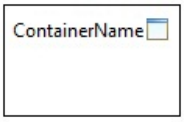
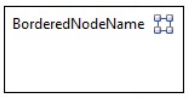
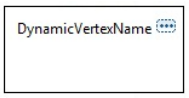
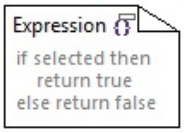
7.2 Syntaxe concrète graphique de MID

Par souci de simplicité, nous proposons un formalisme graphique pour présenter nos concepts. Ce formalisme permet de voir graphiquement la spécification des diagrammes au lieu de la forme textuelle ou arborescente. Ainsi, nous proposons une syntaxe concrète graphique pour nos méta-modèles. Notez que les concepts de nos méta-modèles peuvent également définir ce formalisme (méta-description).

Le tableau 7.1 présente les concepts de base de **MID** et énumère pour chacun sa représentation graphique.

TABLE 7.1: Concepts de MID

Concept	Signification	Notation Visuelle	Exemple
ComponentBinding	Un lien d'association entre interfaces.		
inherits	Héritage graphique des composants de diagramme.		
StyleInterface	Un point de liaison entre la structure d'un élément de diagramme (*DiagramComponent*) et son *style*.		
EventInterface	Un point de liaison entre un élément de diagramme (*DiagramComponent*) et ses interactions.		

Concept	Signification	Notation Visuelle	Exemple
DomaineInterface	Un point de liaison entre un élément de diagramme (*DiagramComponent*) et un élément de la syntaxe abstraite		
Noeud	Le sommet d'un graphe et la formalisation d'une figure.		Une classe UML.
Edge	Une connection entre éléments de diagramme.		Un lien d'héritage UML.
Label	Une étiquette qui permet d'éditer/afficher une valeur textuelle.		Le nom d'une Classe.
Container	Sert à structurer, contenir et/ou référencer d'autres éléments.		Le compartiment d'attributs d'une Classe.
Diagram	Un conteneur d'éléments graphiques (*DiagramComponent*).		Diagrammes UML, Flowchart.
BorderedNode	Un Noeud qui peut être apposé sur la bordure d'autres noeuds.		Le port d'une classe composite.
DynamicVertex	Une séquence dynamique de noeuds.		Les labels de stéréotype et de QualifiedName.
Expression	L'implémentation d'une condition définie par un code .		

La figure 7.24 montre un exemple de spécification d'un composant avec la vue graphique (en haut à droite) et son équivalent en vue arborescente (en haut à gauche). Les

deux vues permettent le résultat dans le bas de la figure.

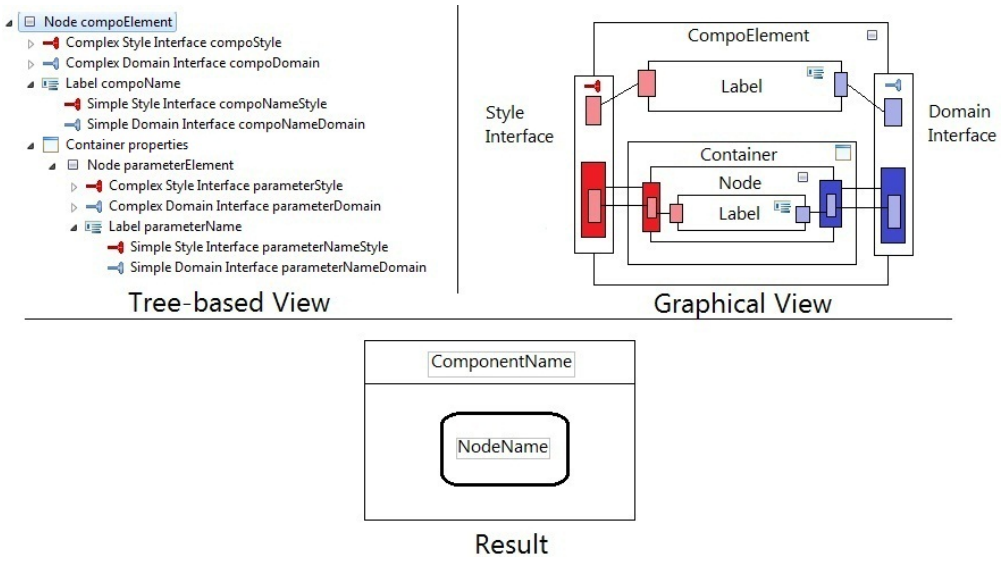


FIGURE 7.24 – MID : formalisme graphique

Dans le reste de cette thèse, nous utilisons ce formalisme graphique dans la définition des langages visuels.

7.3 Mécanisme d'héritage graphique

En réponse à la problématique de la réutilisation (questions 12 et 13 du chapitre 2), nous avons proposé un mécanisme d'héritage graphique permettant de récupérer une partie de la spécification d'un diagramme et de la réutiliser, l'étendre et la spécialiser dans d'autres diagrammes.

L'héritage graphique est un principe inspiré de la programmation orientée objet, permettant de créer une nouvelle notation visuelle (ici représentée par le composant de diagramme ou *DiagramComponent*) à partir d'un autre composant existant. Le nom d'*héritage* (aussi appelé dérivation) provient du fait que la notation dérivée (le composant de diagramme nouvellement créé) contient les caractéristiques/propriétés (contenu, structure, style, comportement, etc.) de la notation dont elle dérive.

L'intérêt majeur de l'héritage est de pouvoir définir de nouveaux éléments dans la notation dérivée, qui viennent s'ajouter à ceux hérités. La possibilité de redéfinir une caractéristique dans des notations héritant d'une notation de base s'appelle la *spécialisation*.

Par ce moyen on crée une hiérarchie de notation de plus en plus spécialisées. Cela a comme avantage majeur de ne pas avoir à repartir de zéro lorsque l'on veut spécialiser un élément graphique existant.

Les composants de diagramme (*DiagramComponent*) ont la capacité d'hériter les uns des autres. Quand un composant B hérite d'un composant A, il récupère toutes ses

propriétés (contenu/structure, style et comportement) (figure 7.25).

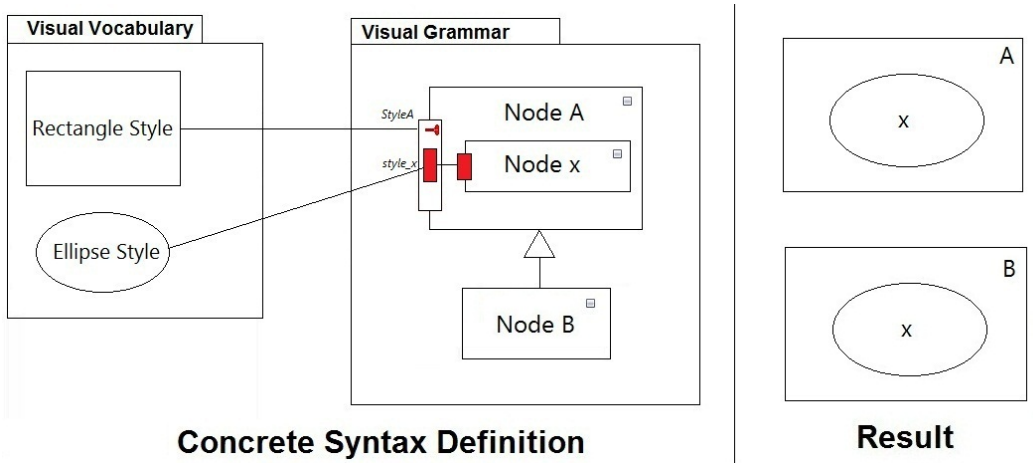


FIGURE 7.25 – MID : Héritage graphique

La spécialisation intervient lorsqu'un composant ou une caractéristique (représentée ici pas une interface) possède le même nom qu'un ou une autre qui existe dans la hiérarchie de l'héritage. Ainsi, si le composant *B* contient un élément *b* avec le même nom qu'un composant *a* dans (*A*), ceci est interprété par une spécialisation (redéfinition) du composant *a*.

Trois possibilités sont donc offertes :

1. Si on a besoin de changer une caractéristique de *a*, on doit redéfinir l'interface correspondante à cette modification dans *b* (par exemple, si on veut changer l'apparence de *a* on doit redéfinir l'interface de style dans le composant *b*).

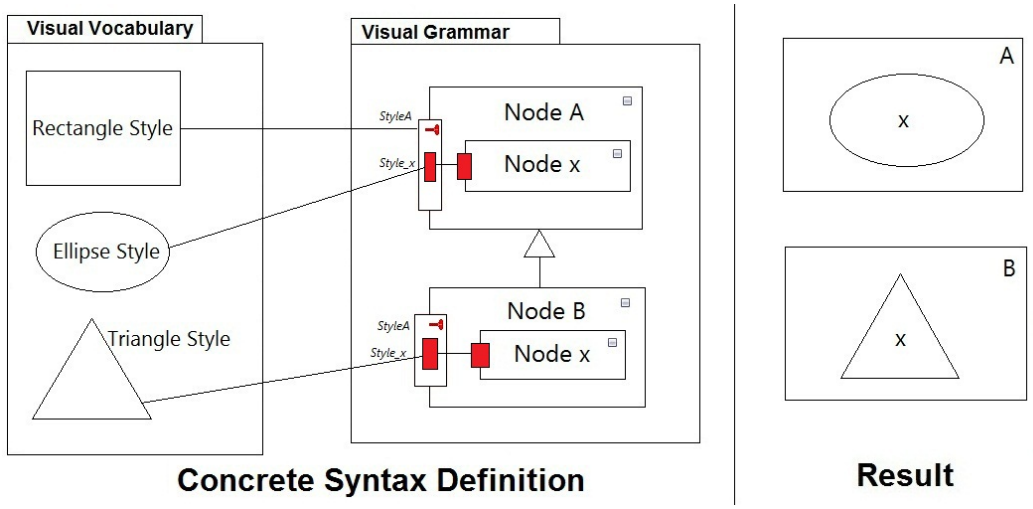


FIGURE 7.26 – Spécialisation d'une caractéristique : redéfinition du style

2. Si on a besoin de supprimer un composant hérité, on doit redéfinir toute la hiérarchie des interfaces héritées sans mettre les interfaces de ce composant dans cette hiérarchie (figure 7.27).

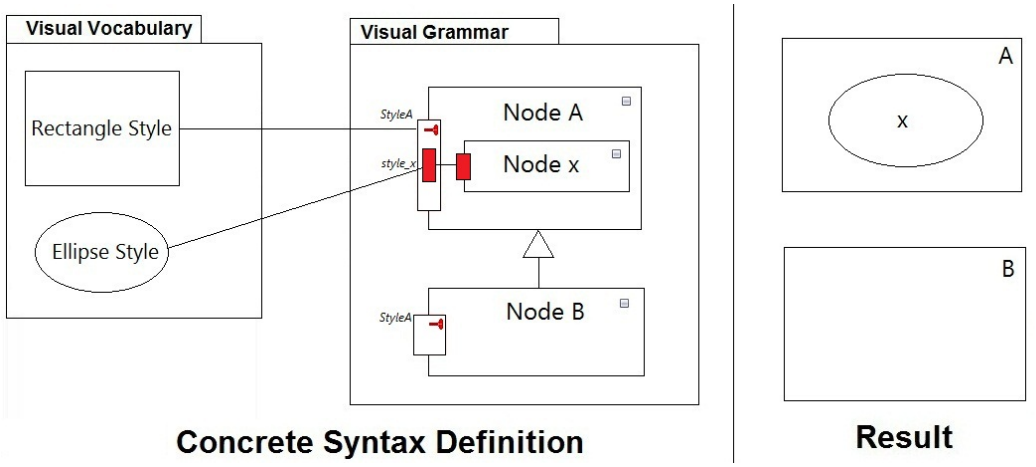


FIGURE 7.27 – Spécialisation de la structure : suppression

3. Dans le cas où on veut ajouter un composant à la structure héritée, on doit le placer avec ses interfaces (caractéristiques) dans le niveau hiérarchique voulu (figure 7.28).

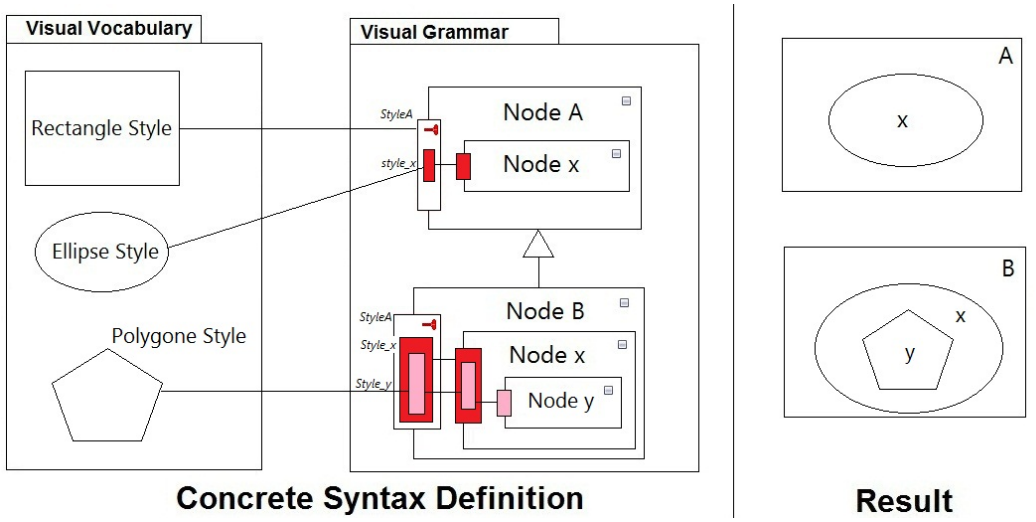


FIGURE 7.28 – Spécialisation de la structure : ajout

Ainsi nous pouvons modifier la structure, le style et le comportement. Cette fonction optimise la réutilisation des composants et permet de créer d'autres composants dérivés. Les règles proposées de l'héritage graphique sont repris dans l'algorithme 1.

Algorithme 1 MID : Mécanisme d'héritage graphique

$\forall A, B \in DiagramComponent, \exists X_A, X_B$, où X_A est l'ensemble de sous-composants de A et X_B l'ensemble de sous-composants de B ;
si B hérite de **A alors**
$B \leftarrow X_A$;
$\forall$ **a**, **b** avec **a** $\in X_A$, **b** $\in X_B$;
si NomDe(**b**) = NomDe(**a**) **alors**
b redéfinie **a** :
– Cas 1 : nous avons besoin de modifier une caractéristique de **a**
Modifier l'interface correspondante à cette caractéristique dans b;
– Cas 2 : nous avons besoin de supprimer le sous-composant hérité **a**
Ne pas spécifier ses interfaces dans la hiérarchie des interfaces héritées ;
– Cas 3 : nous voulons ajouter un nouveau élément à la structure héritée
Ajouter ses interfaces dans la hiérarchie des interfaces héritées ;

7.4 Synthèse et discussion

Dans ce chapitre, nous présentons une approche basée sur les modèles et les composants pour la conception des éditeurs graphiques de diagramme. Ceci permet la spécification rapide des éditeurs graphiques de diagrammes à un niveau d'abstraction élevé, afin de modéliser, de réutiliser, de composer et de générer du code. Dans notre proposition, nous nous concentrons sur le concept de composant pour décrire et ensuite assembler les nouveaux concepts des langages visuels (langages diagrammatiques).

Dans notre approche, nous encourageons l'utilisation de composants réutilisables (avec la composition, l'encapsulation et l'héritage, etc.) et la séparation forte des préoccupations (domaine, élément graphique, les variables visuelles). Cela augmente la réutilisabilité des éditeurs de diagrammes et apporte les avantages du paradigme IDM comme la vérification/validation des modèles ou la possibilité de choisir les technologies "cibles" grâce à des techniques de transformation de modèles. Dans MID, nous résolvons certains problèmes identifiés dans les outils et méthodes existants dans l'industrie comme dans la littérature. Par exemple, la spécification à un haut niveau d'abstraction, sans la nécessité d'une intervention programmatique manuelle, la séparation des préoccupations, l'efficacité graphique et enfin la réutilisation des spécifications, qui faisait partie des problématiques majeures de notre travail de recherche.

Notre approche présente plusieurs avantages : Tout d'abord, grâce à la réutilisation de composants sous forme de modèles : les modèles sont théoriquement plus faciles à comprendre et à manipuler par les utilisateurs finaux, ce qui correspond à un objectif de l'IDM. Ensuite, en gain de productivité : il est possible de constituer des librairies de composants, puis de construire son diagramme par assemblage de ces composants. Les éléments graphiques et les éléments du domaine peuvent évoluer séparément.

Troisième partie

Validation

CHAPITRE 8

Spécification des éditeurs UML

Sommaire

Nous effectuons dans ce chapitre, une première validation de notre proposition selon les axes de problématiques que nous avons soulevé dans le chapitre 2. Le premier critère de validation est donc la capacité de notre proposition de spécifier des éditeurs de diagrammes UML, en mettant l'accent sur l'expressivité graphique et la définition des différentes préoccupations des éditeurs (les interactions, la liaison avec le domaine, etc.). Le deuxième axe de validation est le critère de réutilisation de la syntaxe concrète d'UML.

Pour valider notre proposition, nous avons développé une chaîne de transformations ciblant le Framework GMF et produisant des éditeurs de diagrammes opérationnels. Le choix de GMF n'était pas arbitraire. Nous avons étudié une variété d'outils et de technologies cible et nous avons choisi GMF grâce de son intégration à l'environnement Eclipse, la disponibilité de sa documentation et la facilité de manipulation de ses modèles.

Dans un premier temps nous allons présenter notre chaine de transformation de *MID* vers GMF. Par la suite, nous présentons comment nous avons pu spécifier les éléments courant de la syntaxe concrète d'UML avec MID, ainsi que l'aperçu des éditeurs produits à partir de cette spécification. Par la suite, nous montrons les cas fréquents de réutilisation de la syntaxe concrète d'UML et comment nous les prenons en charge avec notre solution. Ainsi, à la fin de ce chapitre, nous consolidons les résultats de cette partie en montrant le taux de réutilisabilité, offert par *MID* en spécifiant UML.

8.1 Chaîne de transformation *MID* − > GMF

Pour concrétiser notre approche, nous avons développé une chaîne de transformation ciblant le Framework GMF, pour générer le code java opérationnel des éditeurs de diagrammes.

La chaîne de transformation nous permet de passer d'un espace de travail de modélisation à l'autre. Chaque espace de travail est à un niveau d'abstraction et de détail différent de l'autre. Les modèles intermédiaires sont décrits par des méta-modèles et

servent à ajouter des détails techniques et des technologies requises pour la génération de code. Nous avons pris soin de faire apparaître le plus tard possible les détails technologiques propre à la cible visée. Ces détails n'apparaissent que dans les derniers modèles de la chaîne.

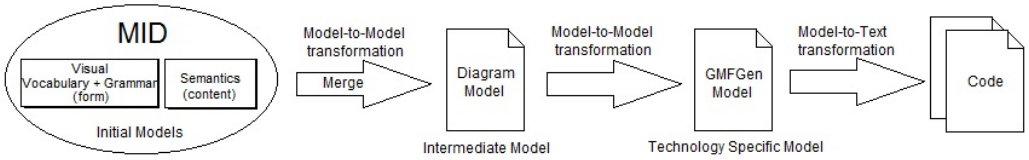


FIGURE 8.1 – Chaîne de transformation *MID* – > GMF

Cette chaîne de transformation nous permet de générer 100% du code opérationnel des éditeurs de diagrammes. Le développeur n'a plus qu'à exécuter l'application. L'éditeur généré permet de manipuler des concepts spécifiques au domaine avec des représentations graphiques spécifiées au niveau du modèle. Actuellement, notre approche permet la réutilisation au niveau de la conception des diagrammes. Cependant, la technologie cible choisie, ici GMF, ne permet pas une réutilisation du code généré. Celui-ci reste dupliqué, mais avec une définition unique au niveau de nos méta-modèles.

8.2 Spécifier les éditeurs UML avec MID

Dans cette partie nous allons spécifier les éléments graphiques les plus courants dans le langage UML. Nous allons commencer par les éléments *classe* et *composant*. Ces deux éléments de formes rectangulaires se définie comme illustré dans la figure 8.2.

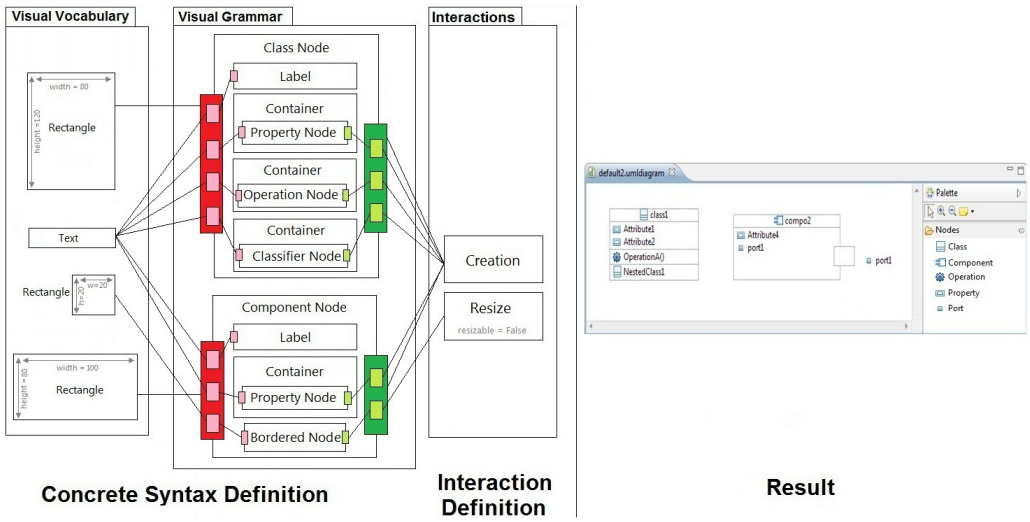


FIGURE 8.2 – Spécification des éléments graphiques et leurs interactions

Une classe décrit un ensemble d'objets qui partagent les mêmes caractéristiques

(attributs, opérations et les classificateurs qui sont imbriqués au sein de la classe), contraintes et sémantique. Elle est constituée graphiquement, d'un noeud représenté par un rectangle. Ce noeud contient un label pour le nom et trois conteneurs qui contiennent respectivement des attributs, des opérations et des classificateurs imbriqués (tous sous forme de labels).

Comme tout autres classificateurs, il se constitue graphiquement d'un noeud rectangulaire avec un label pour le stéréotype *«component»*, un autre label pour le nom du composant et un conteneur de sous-composants. De plus, il contient un noeud de bordure carré pour préciser qu'il accepte les ports. Sa spécification est illustrée dans la figure 8.2.

Cette spécification montre aussi l'association des différents composants aux interactions. L'interaction *Creation* est associée à tous les éléments qu'on désire afficher dans la palette. L'élément graphique *Port* est lié à l'interaction *Resize* avec l'attribut resizable égale à *false* pour spécifier que cet élément n'est pas redimmentionnable.

Par ailleurs, ces composants graphiques sont liés aux éléments de la syntaxe abstraite (métamodèle) de UML, à travers les éléments *Binding*. La figure 8.3 montre le détail de cette spécification.

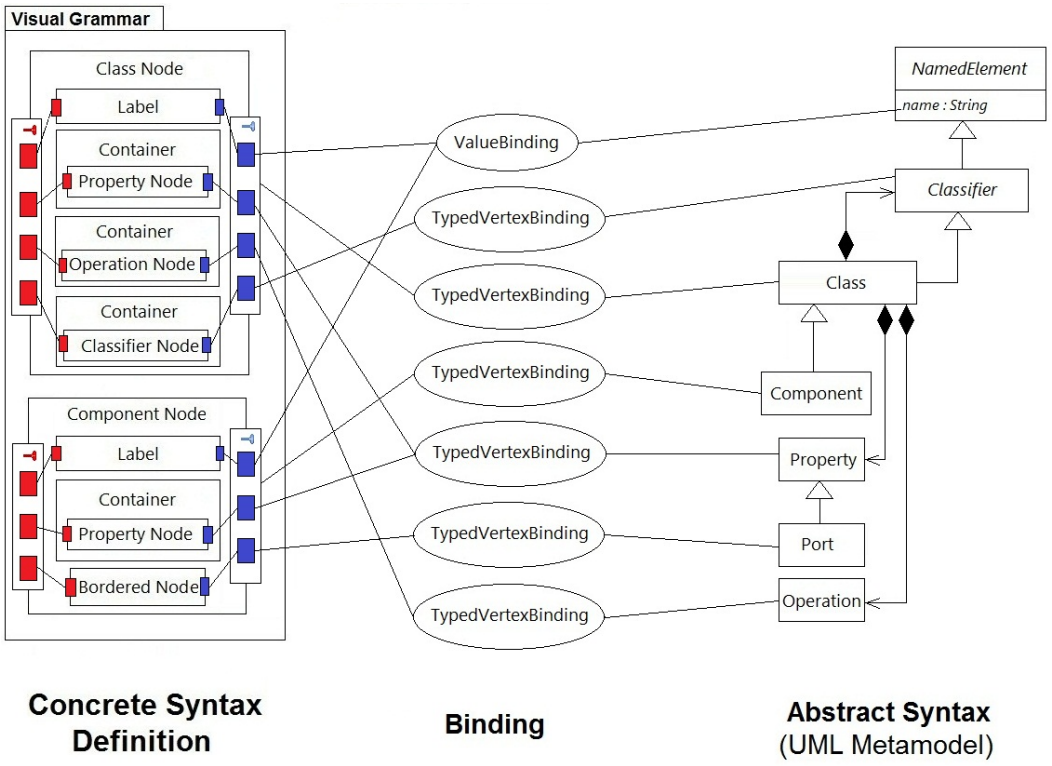


FIGURE 8.3 – Spécification des liens entre les éléments graphiques et ce qu'ils représentent dans le métamodèle UML

Dans la figure 8.4 la spécification d'un Node contenant un artifact (éléments du

diagramme de déploiement). Un node est représenté par un cube contenant un label de nom et un sous-noeud qui représente l'artifact. L'artifact est constitué d'un noeud rectangulaire contenant un label stéréotypé *«artifact»* et un autre label pour le nom.

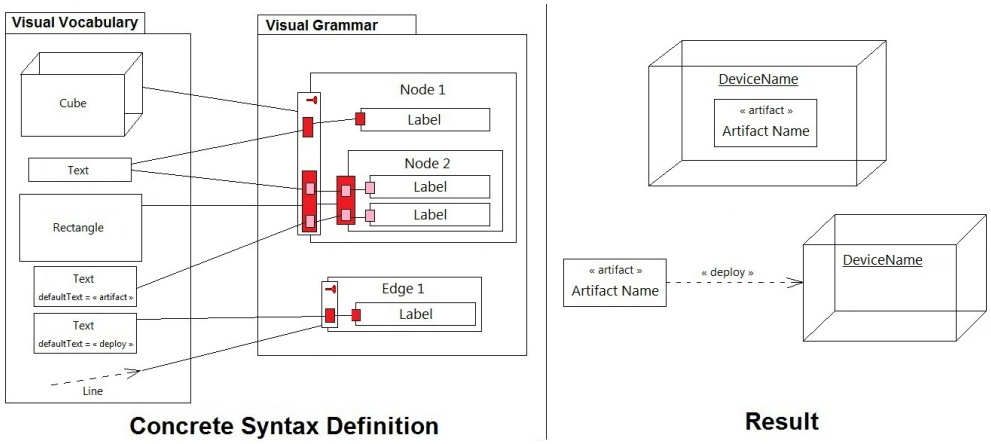


FIGURE 8.4 – Spécification des éléments graphiques du diagramme de déploiement

Le diagramme d'état-transition et le diagramme de cas d'utilisation sont parmi les diagrammes de type entités-relations (ER) qui existent dans UML. On définit les éléments du diagramme d'états-transitions en spécifiant chaque composant graphique tel que montré dans l'exemple illustré dans la figure 8.5. L'éditeur généré à partir de cette spécification est présenté dans la figure 8.6.

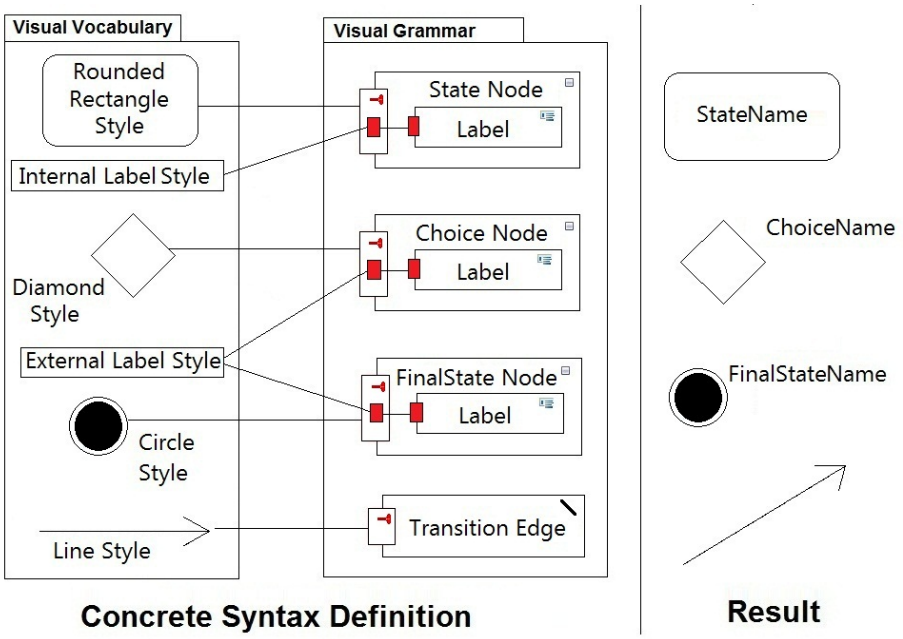


FIGURE 8.5 – Spécification des éléments graphiques du diagramme d'états-transitions

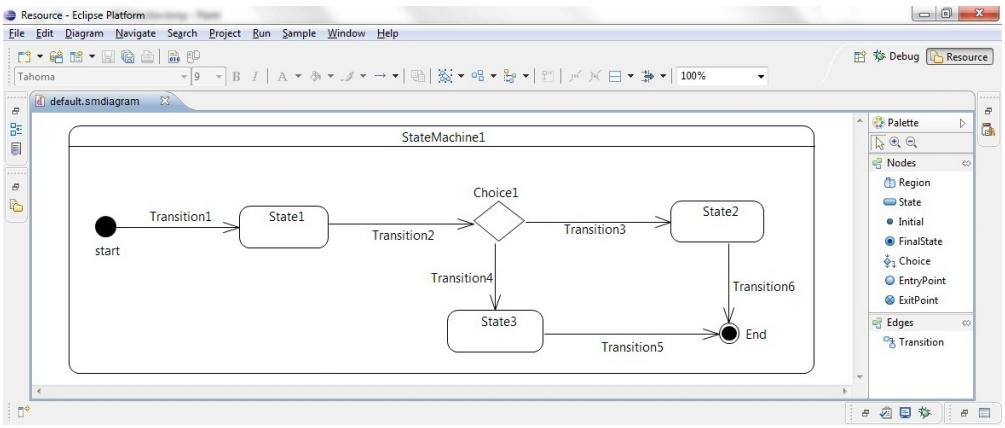


FIGURE 8.6 – Éditeur généré pour l'exemple dans la figure 8.5

La figure 8.7 présente un exemple de spécification des éléments du diagramme du cas d'utilisation. Cet exemple comporte la définition de l'élément *actor* constitué d'un noeud représenté par un polygone sous forme d'un "homme en bâtons" (stick man), un cas d'utilisation construit à partir d'un noeud elliptique avec un label pour le nom du cas.

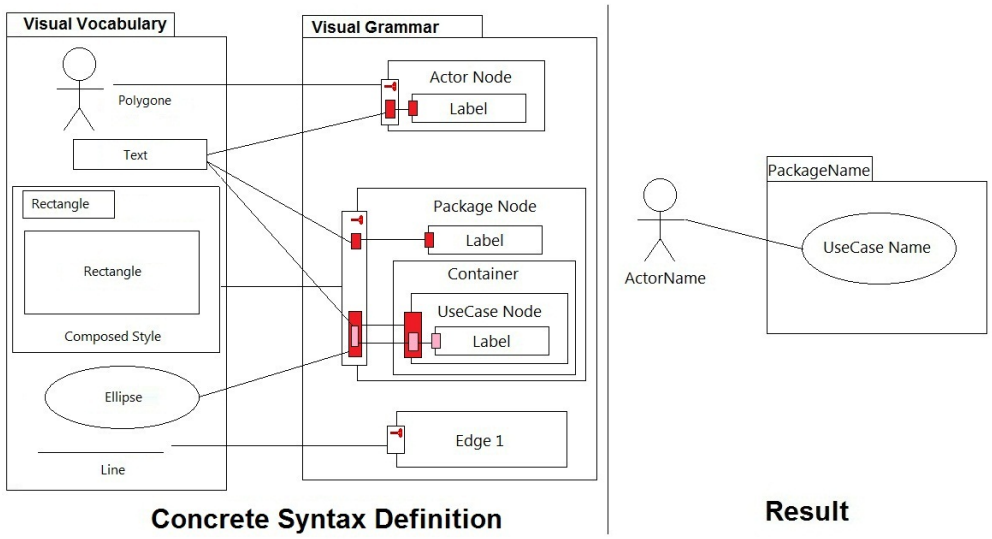


FIGURE 8.7 – Spécification des éléments graphiques du diagramme de cas d'utilisation

La figure 8.8 présente la spécification d'une ligne de temps (Lifeline) du diagramme de séquences. Ce diagramme fait partie des diagrammes hybrides de UML, puisqu'il comporte les caractéristiques de plusieurs catégories de langages visuels et est donc plus riche en termes de contenu graphique. Pour spécifier l'élément *Lifeline*, on définie un noeud représenté par une figure composée d'un rectangle qui représente l'objet et d'une ligne en traits qui représente la ligne de temps. Pour la définition des périodes d'activité de l'objet (*ExecutionSpecifications*), on ajoute un noeud de bordure rectangulaire sur la ligne de temps.

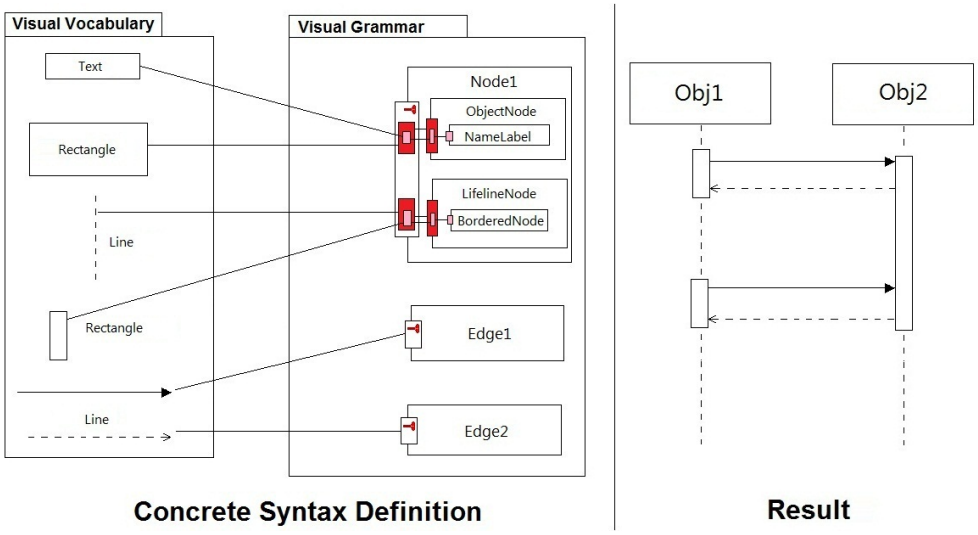


FIGURE 8.8 – Spécification des éléments graphiques du diagramme de séquences

8.3 Réutiliser et étendre les éléments graphiques de UML

Les principaux éléments des diagrammes UML sont les différents classificateurs (*Classifiers*) qui ont généralement la même représentation graphiques à l'exception de quelques variations. Ils sont formés d'un noeud rectangulaire composé d'un label pour le nom, un conteneur d'attributs et un conteneur d'opérations pour la plupart d'entre eux. Les classes et interfaces, contiennent en plus de cela, un troisième conteneur optionnel pour les classificateurs imbriqués (*Nested Classifiers*). Cette spécification est illustrée dans la figure 8.9.

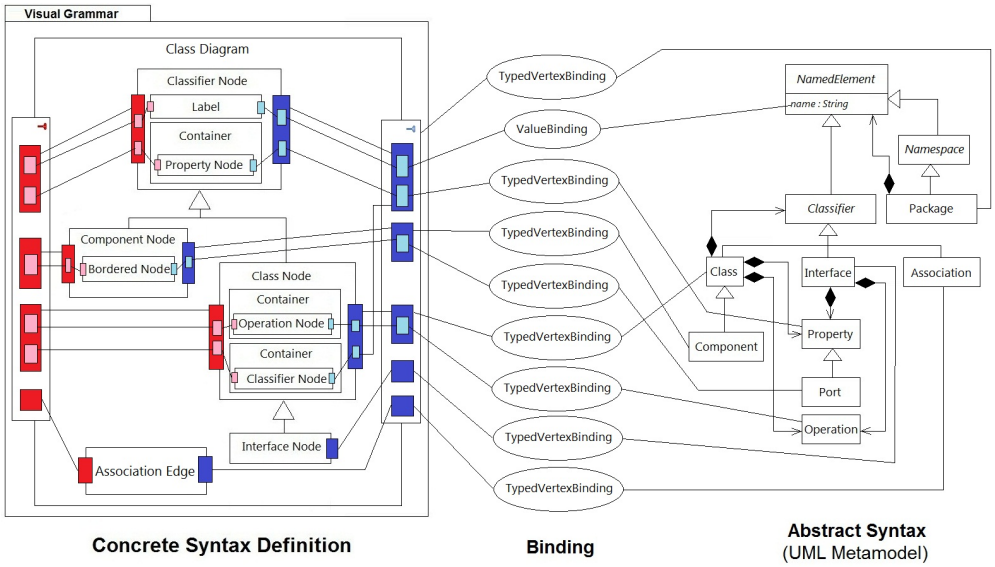


FIGURE 8.9 – Spécification des liens entre les éléments graphiques et leur sémantique

Dans cet exemple, et pour simplifier, l'interface hérite de la notation visuelle de la classe pour montrer la similitude graphique entre les deux concepts. Cette spécification montre aussi l'association des différents composants aux éléments qu'ils représentent dans le métamodèle UML à travers les éléments *Binding*. La figure 8.9 montre le détail de cette spécification.

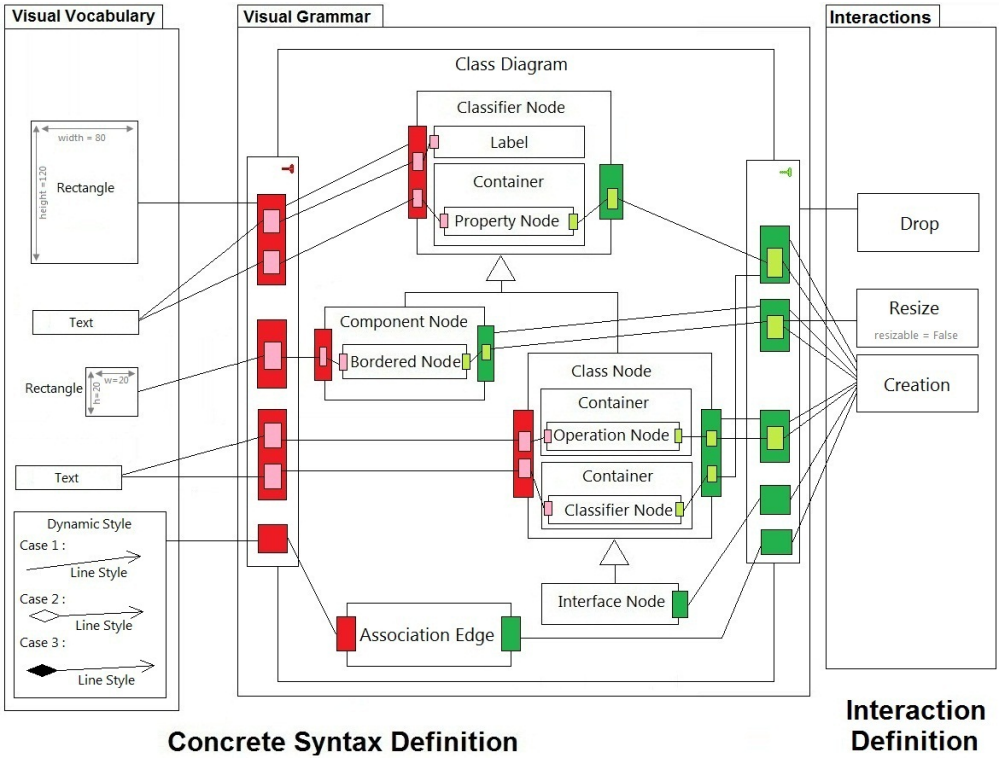


FIGURE 8.10 – Spécification des éléments graphiques et leurs interactions

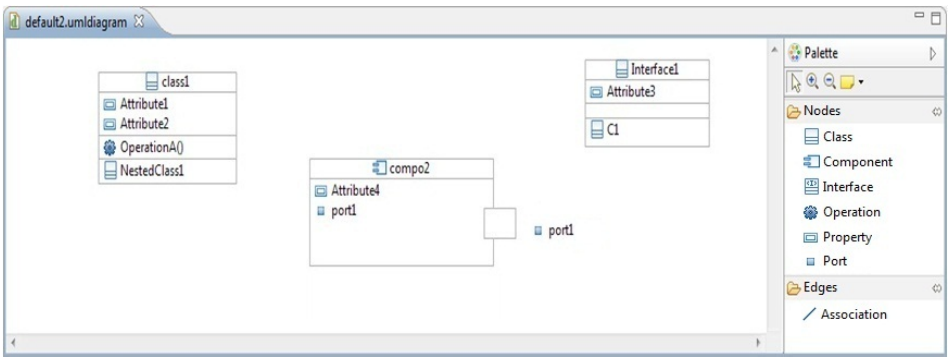


FIGURE 8.11 – Éditeur généré pour l'exemple dans la figure 8.10

D'un autre coté, la spécification figure 8.10 montre l'association des différents composants aux interactions. L'interaction *Creation* est associée à tous les éléments figurant dans la palette. Le digramme quant à lui est associé à l'interaction *Drop* pour définir

sa capacité à accepter les éléments glissables (Draggable). L'élément graphique *Port* est lié à l'interaction *Resize* avec l'attribut resizable égale à *false* pour spécifier que cet élément n'est pas redimmentionnable. La figure 8.11 montre le résultat généré à partir de la spécification dans les figures 8.10 et 8.9.

On peut trouver d'autres similitudes graphiques entre les diagrammes UML. c'est le cas du diagramme d'états-transitions et les éléments du diagramme d'activités. Pour cette raison, nous utilisons le mécanisme d'héritage pour réutiliser la définition de ces éléments (figure 8.12).

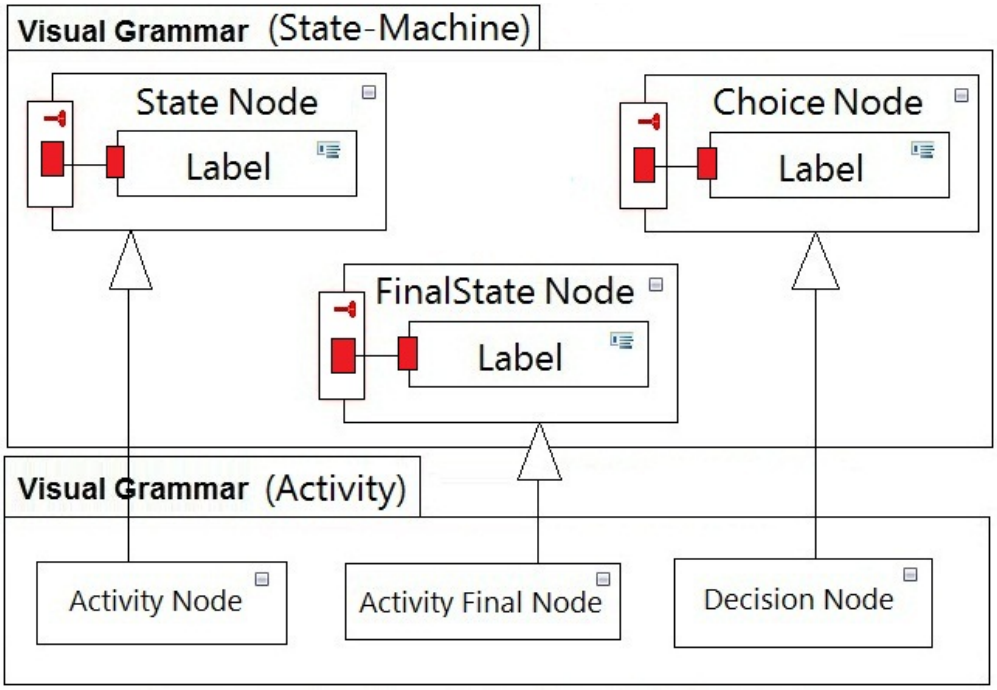


FIGURE 8.12 – Spécification des éléments graphiques du diagramme d'activités

Ainsi, toute redondance graphique dans les diagrammes UML, peut être traitée avec ce mécanisme d'héritage graphique. L'apport en termes de temps de spécification et de maintenance est considérablement réduit en utilisant des éléments graphiques réutilisable. Le taux de réutilisation de la syntaxe concrète graphique d'UML est détaillé dans le tableau 8.1.

8.4 Synthèse et discussion

La conception complète et la génération des éditeurs de diagrammes UML (La spécification complète des diagrammes UML est détaillée dans l'annexe 12) nous a permis d'effectuer une première validation de notre approche. Nous avons complété cette validation par une évaluation selon les critères définis dans le chapitre 5 (section 5.6).

8.4.1 Séparation des Préoccupations

Contrairement aux outils et méthodes existants pour la modélisation et la génération des éditeurs de diagrammes, nous avons séparé initialement les aspects graphiques, comportementaux et sémantiques de l'éditeur, nous avons séparé les deux aspects graphiques, qui sont le vocabulaire visuel (variables visuelles) et la grammaire visuelle (les règles de composition des éléments du diagramme). Par la suite, il était important de créer une autre partie qui effectue l'association entre les différents aspects, notamment entre la syntaxe abstraite et la syntaxe concrète graphique.

La séparation a été également réalisée dans la chaîne de transformation, en introduisant plusieurs modèles de niveau intermédiaire et en retardant l'introduction des détails techniques dans les derniers modèles de la chaîne. Ceci permet une meilleure maintenabilité de la chaîne de transformation lors des évolutions de nos méta-modèles.

Une forte séparation des préoccupations, permet une meilleure réutilisation et maintenance des modèles. Elle diminue les coûts de développement en termes de temps de maintenance en cas de changements dans ces modèles et devrait permettre la conception de nouvelles applications par assemblage de modèles existants.

En proposant la séparation des préoccupations avec un mécanisme d'héritage graphique, nous avons proposé une solution efficace pour la variabilité visuelle (question 5 de la problématique). Il est possible de réutiliser un élément graphique dans deux diagrammes différents avec deux représentations différentes, simplement en séparant sa composition (grammaire visuelle) de son affichage (vocabulaire visuel) et en utilisant l'héritage et la surcharge pour redéfinir les variations graphiques.

8.4.2 Réutilisation

La réutilisation a été depuis le début de nos travaux de recherche, le critère le plus important et le plus recherché. Ce critère nous a motivé à rechercher des méthodes qui permettent une meilleure réutilisation de modèles de spécification. Pour cette raison, nous avons choisi d'introduire une approche de métamodélisation à base de composants pour spécifier les éditeurs graphiques. L'approche basée sur les composants assure une meilleure lisibilité et une meilleure maintenance de modèles. Elle est particulièrement utile pour le travail d'équipe et permet l'industrialisation du développement de ce type d'application. La réutilisation d'un composant permet d'économiser un gain important de productivité, car il permet de réduire le temps de développement, d'autant plus que le composant est réutilisé plus souvent.

Outre la séparation des préoccupations, la réutilisation est effectuée dans notre proposition à travers des concepts inspirés de l'orienté-objet. L'utilisation d'interfaçage (pour le couplage faible entre préoccupations), la composition et l'encapsulation entre les composants et l'utilisation de notre mécanisme d'héritage nous ont permis de composer des éléments graphiques, réutilisables dans plusieurs diagrammes : les composants qui héritent d'autres, réutilisent toutes les propriétés héritées et peuvent redéfinir tout ou une partie de la description.

La spécification de la syntaxe concrète de UML, nous permet d'évaluer le taux de

réutilisation des notations visuelles d'UML avec *MID*. Le tableau 8.1 présente les diagrammes spécifiés avec *MID*, le nombre de notations visuelles dans chaque diagramme et le taux de réutilisation de ces notations.

TABLE 8.1: Taux de réutilisation de la notation UML

Diagramme	Nombre de notations	Nombre de notations réutilisés	Taux de réutilisation
Classes	33	18	54,5%
Composants	15	8	53,3%
Structure Composite	16	14	87,5%
Déploiement	11	8	72,7%
Package	9	9	100%
États-transitions	16	7	43,7%
Activités	16	12	75%
Cas d'utilisation	17	14	82,3%
Séquences	16	5	31,2%
Communication	5	4	80%
Interaction globale	20	20	100%
Taux global de réutilisation			**71 %**

A travers les exemples présentés dans ce chapitre, nous avons validé notre approche en termes de réutilisabilité : Cette approche nous a permis de réutiliser plus de 70 % des composants créés dans l'ensemble de la syntaxe concrète du langage UML, ce qui n'est pas négligeable en termes de coût de développement et de temps. Cette approche nous a permis de définir plus facilement les spécificités des éditeurs avec une approche dirigée par les modèles et sans qu'il soit nécessaire d'intervenir manuellement (programmatiquement) pour effectuer des changements, ce qui augmente le niveau de maintenabilité des éditeurs générées avec notre solution.

Par ailleurs, nous avons évalué le taux de réutilisation des autres outils et nous avons constaté que notre proposition offre un taux de réutilisation largement supérieur à celui des autres outils. Le tableau 8.2 montre les taux de réutilisation des outils existants. Le détail de cette évaluation est dans l'annexe.

TABLE 8.2: Comparatif des taux de réutilisation

Outil	Taux de réutilisation
MetaEdit+	46,9 %
GMF	52,3 %
Obeo Designer	34,8 %

Spary	64 %
MID	**71 %**

D'autres critères comme l'expressivité graphique de notre solution et son indépendance des technologies d'implémentation font l'objet d'une deuxième validation dans le chapitre 9.

CHAPITRE 9

MID au-delà de UML

Sommaire

Dans ce chapitre, nous effectuons une deuxième validation de notre proposition. Cette validation a trois objectifs : le premier est de montrer que notre solution est valable au-delà de la spécification de la syntaxe concrète du langage UML, démontrant ainsi l'ouverture de notre proposition à d'autres domaines d'application. Le deuxième est de montrer notre indépendance à la technologie d'implémentation (technologie cible) et montrer ainsi que notre proposition a un niveau d'abstraction plus élevé que les autres technologies. Le troisième objectif, sert à montrer que les possibilités de spécification offertes par notre proposition, sont largement plus grandes que celles proposées par les autres technologies (GMF, Spray, etc.) puisqu'avec les mêmes concepts au niveau de nos métamodèles, les technologies cibles n'arrivent pas à suivre la description et de la traduire en code opérationnel. Il nous a fallu, pour spécifier les schémas électriques par exemple, de changer la technologie cible (de GMF vers Spray), à cause des limitations rencontrées dans GMF. Le problème est réciproque pour Spray, puisqu'il ne peut pas suivre la description de MID et de générer des éditeurs pour UML (chapitre 8). Ceci est expliqué par le fait que lors de la conception de notre proposition, nous avons été plus généralistes en récupérant les avantages des autres solutions et en essayant de résoudre leurs limitations.

Nous validons aussi le critère de l'expressivité graphique de notre proposition en spécifiant des langages visuels complexes comme des éditeurs de schémas électroniques ou d'autres diagrammes comme celui des workflow de BPMN et en ciblant cette fois, le framwork Spray pour la génération des éditeurs graphiques de ces langages.

Dans la première partie, nous allons présenter notre chaine de transformation de *MID* vers *Spray*. Par la suite, nous spécifions deux langages visuels : dans un premier temps, nous spécifions le diagramme de workflow de BPMN avec MID et nous montrons les modèles Spray générés ainsi que l'éditeur graphique produit à partir de cette spécification. Finalement, nous spécifions un éditeur de schémas électroniques et nous montrons son éditeur généré avec MID.

9.1 Chaîne de transformation *MID* − > Spray

Nous avons développé une chaîne de transformation pour générer le code java pour les éditeurs de diagramme, en ciblant le Framework Spray. A partir d'une description *MID* nous pouvons générer les modèles Spray Core, Shape et Style. Ces modèles permettent de générer 100% du code opérationnel des éditeurs de diagrammes. Le développeur n'a plus qu'à exécuter l'application. L'éditeur généré permet de manipuler des concepts spécifiques au domaine avec des représentations graphiques, spécifiées au niveau du modèle.

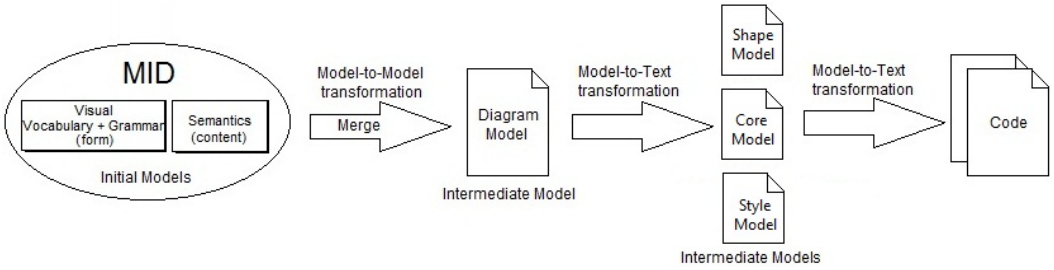


FIGURE 9.1 – Chaîne de transformation *MID* − > Spray

Après spécification des modèles MID, une première transformation permet de faire les associations graphiques et sémantiques nécessaires, applique les règles d'héritage et récupère les éléments réutilisés. Le modèle intermédiaire généré permet de cibler la technologie cible (dans ce cas Spray) et par la suite génère à partir d'une transformation de modèle vers texte les trois modèles spray (core, shape et style). Ces derniers modèles permettent une dernière transformation cette fois vers du code java complètement opérationnel.

9.2 Diagramme BPMN

L'Initiative BPMI (Business Process Management Initiative) a élaboré le standard Business Process Modeling Notation (BPMN) pour fournir une notation qui est facilement compréhensible par tous les analystes d'affaires qui créent les premières ébauches des processus de travail, les développeurs chargés de l'implémentation de la technologie qui va exécuter ces processus et enfin, les gens du métier qui vont gérer et contrôler ces processus [White 2009].

BPMN définit un diagramme de processus métier (BPD ou Business Process Diagram) basé sur un ensemble d'organigrammes adaptés pour créer des modèles graphiques décrivants les opérations des processus métier. Un modèle de processus métier est un réseau d'objets graphiques, qui sont des activités (c'est-à-dire des tâches) et des contrôles de flux qui définissent leur ordre d'exécution.

En termes d'éléments graphiques de la syntaxe concrète, il existe quatre catégories de base qui peuvent varier en termes d'expressivité graphique : les objets de flus (Flow), les

objets de connexion, les swimlanes et les artefacts. Les symboles qui leur correspondent sont illustrés dans la figure 9.2.

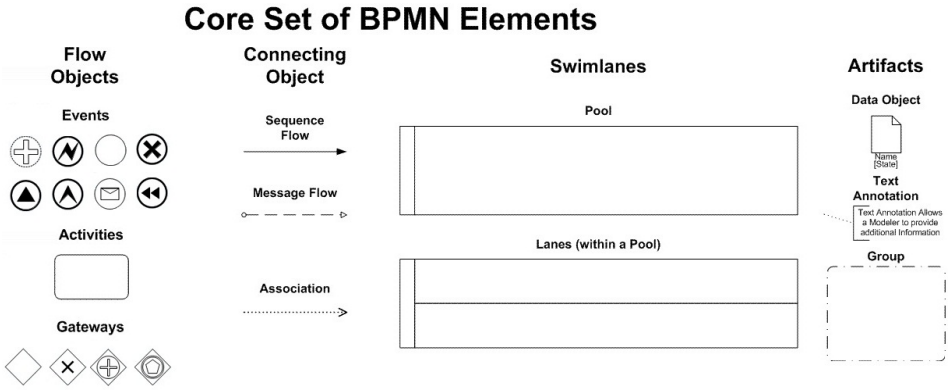


FIGURE 9.2 – Éléments graphiques de BPMN - Copyright 2005 OMG.org

Dans l'objectif de spécifier un éditeur graphique de BPMN avec notre solution, nous avons créé un métamodèle simplifié qui comprend un sous-ensemble représentatif des concepts de ce langage. Ce métamodèle n'est pas destiné à être une représentation exhaustive de BPMN (ce qui est hors de la portée de notre thèse). Une description plus complète de la notation et de la sémantique de BPM peut être trouvée dans [OMG 2011a].

Aussitôt la syntaxe abstraite (métamodèle) de BPMN définie, nous avons spécifié sa syntaxe concrète graphique avec *MID* (figure 9.3). Nous définissons ainsi les éléments du diagramme BPMN en référençant l'élément de domaine avec le composant graphique correspondant. Nous avons trouvé des similitudes graphiques entre les éléments graphiques de BPMN et les éléments du diagramme d'activités spécifié dans le chapitre 8. Pour cette raison, nous utilisons notre mécanisme d'héritage entre ces éléments en surchargeant leurs styles pour les descriptions spécifiques (spécialisation).

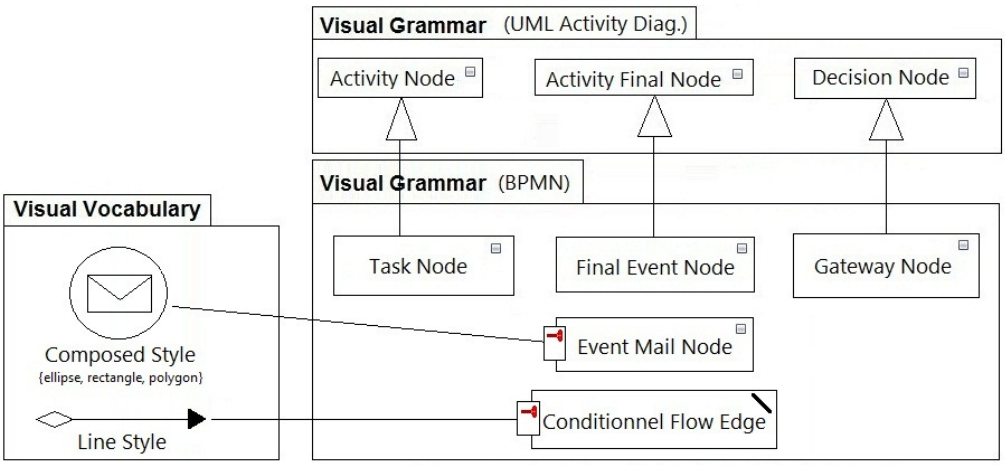


FIGURE 9.3 – Spécification de la syntaxe concrète de BPMN avec *MID*

Le modèle textuel Spray ci-dessous est un aperçu du résultat intermédiaire de la transformation de notre spécification *MID*. Cette description textuelle décrit l'éditeur présenté dans la figure 9.4.

```
//Tasks
class BPMNActivityTask {
    shape BPMN_Activity_Task {
        name into shapeName
    }
    behavior {
        create into modelElements palette "Activity" askFor
           name;
    }
}
shape BPMN_Activity_Task {
    rounded-rectangle {
        size (width=120, height=80)
        curve (width=20, height=20)
        text {
            position (x=5, y=30)
            size (width=110, height=20)
            align (horizontal=center, vertical=middle)
            id = shapeName
        }
    }
}
//Gateways
class BPMNGateway {
    shape BPMN_Gateway_Event behavior {
        create into modelElements palette "Gateway";
    }
}
shape BPMN_Gateway{
    polygon {
        point (x=0, y=30)
        point (x=30, y=0)
        point (x=60, y=30)
        point (x=30, y=60)
    }
}
//Events
class BPMNEventMail {
    shape BPMN_EventMail behavior {
        create into modelElements palette "Event";
    }
}
shape BPMN_EventMail {
    ellipse {
        size (width=50, height=50)
        rectangle {
```

```
            position (x=10, y=15)
            size (width=30, height=20)
            polygon {
                point (x=0, y=0)
                point (x=15, y=10)
                point (x=30, y=0)
            }
        }
    }
}
```

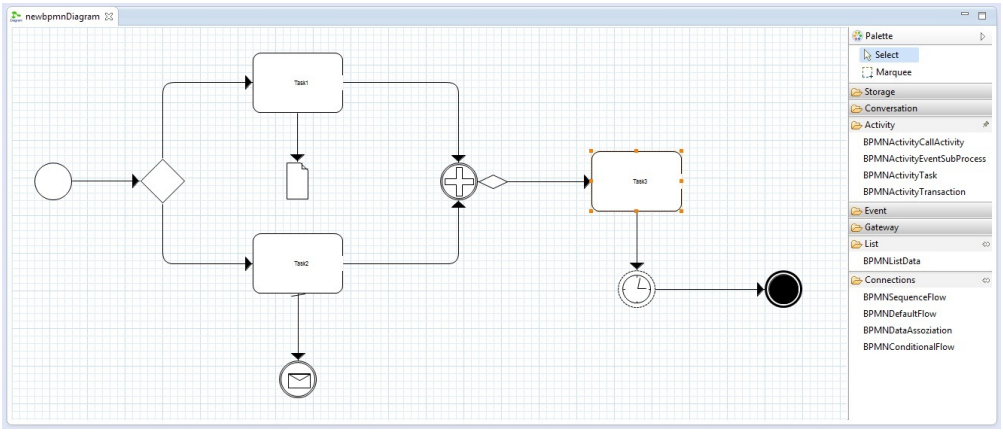


FIGURE 9.4 – Éditeur de BPMN généré avec *MID*

9.3 Schémas électriques

Pour valider le critère de l'expressivité graphique, introduit dans le chapitre 2, nous avons choisi de spécifier des langages complexes hors du périmètre des langages du génie logiciel, qui sont pour la plupart, du genre entités-relations. Ainsi, nous avons choisi de spécifier un éditeur graphique pour les schémas (circuits) électriques.

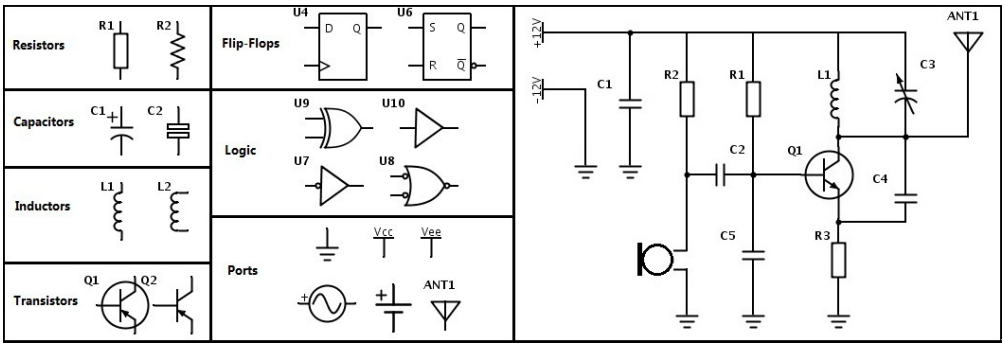


FIGURE 9.5 – Concepts graphiques (à gauche) d'un schéma électrique (à droite)

Un schéma électrique est une représentation graphique d'un circuit électrique, basée sur des conventions visuelles. Il montre les composants du circuit sous forme de symboles normalisés (partie gauche de la figure 9.5), ainsi que l'alimentation et les signaux reliant ces composants [Wikipedia 2013i].

Notre objectif est de spécifier des éditeurs capables de dessiner et de manipuler les concepts graphiques des composants électriques et de construire des schémas tels que montré dans la partie droite de la figure 9.5.

Nous avons pu décrire ce langage visuel avec notre proposition malgré sa complexité graphique. Un aperçu de cette spécification est dans la figure 9.6.

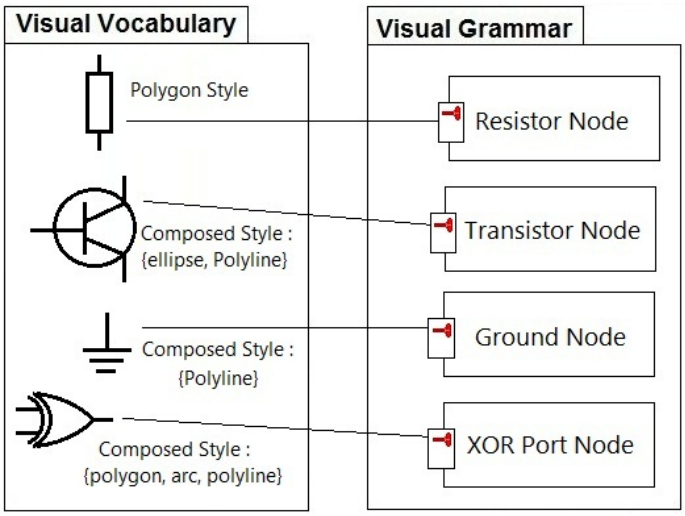


FIGURE 9.6 – Spécification de la notation visuelles des schémas électriques avec *MID*

Le code ci-dessous est le résultat intermédiaire de la transformation de notre spécification *MID* vers Spray. Cette description textuelle décrit l'éditeur de schémas électriques présenté dans la figure 9.7.

```
class Transistor{
  shape TransistorShape
  behavior {
      create into components palette "Components"
  }
}
shape TransistorShape{
  ellipse{
    position(x=4, y=4)
    size(width=20, height=20)
  }
  line{
    point(x=0, y=14)
    point(x=10, y=14)
  }
  line{
```

```
    point(x=10, y=8)
    point(x=10, y=20)
  }
  polyline{
    point(x=18, y=0)
    point(x=18, y=8)
    point(x=10, y=11)
  }
  polyline{
      point(x=18, y=28)
    point(x=18, y=20)
    point(x=10, y=17)
  }
}
class Antenna{
  shape AntennaShape
  behavior {
      create into components palette "Components"
  }
}
shape AntennaShape{
  polygon{
    point(x=0, y=0)
    point(x=10, y=0)
    point(x=5, y=10)
  }
  line{
    point(x=5, y=0)
    point(x=5, y=20)
  }
}
```

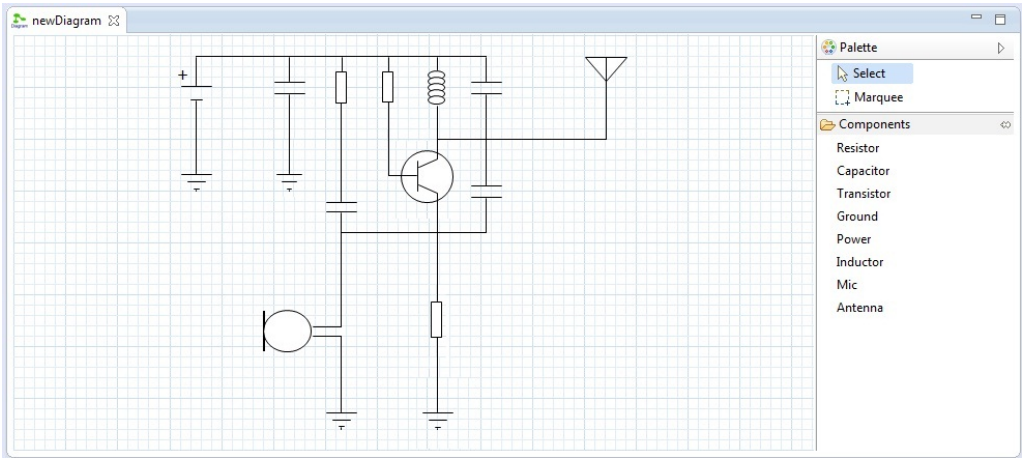


FIGURE 9.7 – Éditeur de schémas électriques généré avec *MID*

9.4 Synthèse et discussion

Ce chapitre nous a permis de valider un critère essentiel dans la spécification des langages visuels. Ce critère est l'expressivité et la complétude graphique. Selon [Moody 2009a], l'expressivité graphique est définie par le degré de fidélité d'expression d'un système par une notation visuelle. Ceci peut être vérifié si toutes les variables visuelles sont utilisées dans la construction de cette notation. La complétude graphique est définie par la capacité d'utiliser pleinement la variable de *forme* dans la construction des notations visuelles complexes : composites, imbriqués, 2D/3D, etc.

Pour prouver que notre proposition permet de spécifier des langages visuels complexes, nous avons choisi de définir un diagramme BPMN et un éditeur pour les schémas électriques. Ce dernier langage est connu par sa complexité et son niveau de détail considérable, chose qui nous a poussé à le choisir comme cas de validation pour vérifier l'expressivité graphique de notre solution.

Ce chapitre nous a permis aussi d'ouvrir les perspectives de notre proposition sur d'autres champs d'application autres que le langage UML ou les langages utilisés dans le génie logiciel (souvent semblables à UML). Par ailleurs, nous avons choisi de changer la cible de notre transformation de GMF vers Spray pour montrer le champs des possibilités offertes par notre solution et son niveau d'indépendance par rapport à la technologie d'implémentation.

CHAPITRE 10

Évaluation

Sommaire

Ce chapitre présente une auto-évaluation de notre solution, selon les objectifs explicitement décrites dans le chapitre 2 et les autres objectifs secondaires implicitement associés aux objectifs principaux. Ainsi, chaque objectif principal est représenté et détaillé dans une section avec les sous-problématiques émanantes de celui-la.

10.1 Spécifier les éditeurs de diagrammes

Un éditeur de diagramme comme son nom l'indique est une interface logicielle qui permet l'édition et la manipulation d'un langage visuel. Construire un éditeur de diagramme consiste à associer le langage visuel à une source de données (dans notre contexte, un métamodèle de domaine) et de spécifier les interactions offertes aux utilisateurs de cet éditeur. Un éditeur sert donc à interagir avec les utilisateurs du langage visuel en leurs offrant les outils nécessaires à sa manipulation. Les interactions des utilisateurs influencent ainsi, d'une manière effective, sur l'aspect graphique du langage visuel (sa structure et sa présentation) et sur le modèle métier associé (éléments du modèle domaine associés à ce langage).

10.1.1 Spécifier les diagrammes de type entités-relations

Pour spécifier des langages visuels de type entités-relations, nous nous sommes appuyé sur les travaux de [Bottoni 2004] qui classifie les langages visuels en différentes categories. Les langages de type entités-relations (relation-based Languages) sont les langages les plus répandus dans le domaine du génie logiciel. Comme leur nom l'indique, la spécification de ce type de langages est basée sur la descritpion des entités (des boites) et les relations entre ces entités (liens entre ces boites). Par analogie aux graphes, ces langages sont en quelques sortes, des graphes composés de noeuds (entités) et d'arêtes (relations). Après la définition de ces deux concepts, il s'est averé que les langages de type

entités-relations nécessitent d'autres types d'élements graphiques, communs à tous les langages, qui servent à enrichir l'expressivité du langage visuel. Parmi les exemples de ces éléments, on trouve le label qui sert de zone d'affichage et d'edition pour les informations textuelles.

À ce niveau, les relations ou comme on les appelés arête (edge), ne sont que des éléments simples qui représentent une connexion entre deux entités. On peut leur associer des entités comme les labels pour ajouter de l'information textuelle à ces liens (par exemple, le nom des associations UML).

La spécification des diagrammes relationnels tel que les diagrammes d'états-transitions et d'activités (Chapitre 8), nous a permis de valider notre proposition pour ce type de langages.

Cependant, la spécification d'autres diagrammes UML, nous a motiver à améliorer notre vision des langages visuels. Même s'il est un langage majoritairement de type entités-relations, le langage UML comporte quelques éléments graphiques complexes, c'est-à-dire, que leur spécification en suit pas le modèle entité-relation comme les autres éléments du langage. Un exemple est le diagramme de séquence qui se compose d'éléments graphiques hybrides, c'est-à-dire qu'ils comportent la notion d'entité-relation en plus d'autres concepts visuels comme l'adjacence et le confinement. Ce qui nous a motivé à faire évoluer notre perception vers une spécification des langages dis hybrides [Bottoni 2004].

10.1.2 Spécifier les langages graphiques hybrides

Pour spécifier des langages hybrides, nous avons procédé d'une manière empirique. Nous avons ainsi spécifié les éléments du langage UML, un par un, pour essayer de proposer le métamodèle le plus complet possible pour ce type de langage. Il s'est avéré donc que les langages hybrides se composent des concepts d'entités-relations en plus d'autres concepts pour le confinement (containment) pour décrire la relation composition/composite et les concepts d'adjacence pour décrire les éléments contigus et attachés à d'autres éléments. Pour cela nous avons ajouté dans notre proposition la notion de *Conteneur* qui servent de groupements d'éléments graphiques regroupé dans une zone unique et la notion de *Bordered Node* qui représente les éléments glissants attachés aux bordures des entités (ils servent pour décrire des notations comme les ports dans une classe). Nous avons pu donc spécifier les autres diagrammes UML et valider ainsi notre proposition par rapport aux diagrammes hybrides. Dans une deuxième validation, nous avons spécifié d'autres langages visuels tels que les schémas électriques (Chapitre 9) et nous avons validé ainsi le niveau d'expressivité graphique (la capacité de description graphique) de notre proposition.

À ce niveau de complexité, nous avons pu être confronté à d'autres problématiques comme les liens composites. Ces liens pourraient être plus complexe qu'une simple ligne (par exemple, la figure 10.1 montre un bus de données, qui lie deux circuits électroniques).

Nous avons ainsi modifié notre proposition de telle sorte que les arêtes peuvent contenir d'autres noeuds et avoir des représentations (Style) plus complexes que des lignes ou des courbes simples. Une arête peut avoir plus d'une représentation grâce au

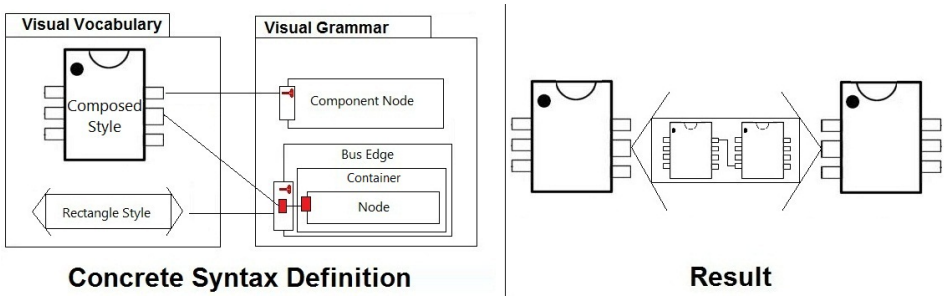


FIGURE 10.1 – Lien sous forme de bus de données entre deux circuits

concept de style dynamique que nous avons proposé (figure 7.18). Ces styles servent généralement pour décrire la variabilité graphique d'un même élément graphique. Ces styles servent aussi à décrire une représentation créée dynamiquement selon une *Condition*. Nous avons conçu ce concept comme un test à choix multiple (Switch/Case) pour définir les différentes possibilités d'affichage d'un élément graphique. Les conditions sont exprimées en *Expression*, qui peuvent évaluer une donnée du domaine et contrôler ainsi la valeur d'un ou plusieurs attributs graphiques (couleur, police, taille, forme, etc.)

Dans le chapitre 9, nous avons validé trois objectifs : le premier est de montrer que notre solution est valable au-delà de la spécification de la syntaxe concrète du langage UML, démontrant ainsi l'ouverture de notre proposition à d'autres domaines d'application. Le deuxième est de montrer notre indépendance à la technologie d'implémentation (technologie cible) et montrer ainsi que notre proposition a un niveau d'abstraction plus élevé que les autres technologies. Le troisième objectif, est de montrer que les possibilités de spécification offertes par notre proposition, sont largement plus grandes que celles proposées par les autres technologies (GMF, Spray, etc.) puisqu'avec les mêmes concepts au niveau de nos métamodèles, les technologies cibles n'arrivent pas à suivre la description et de la traduire en code opérationnel. Il nous a fallu, pour spécifier les schémas électriques par exemple, de changer la technologie cible (de GMF vers Spray), à cause des limitations rencontrées dans GMF. Le problème est réciproque pour Spray, puisqu'il ne peut pas suivre la description de MID et de générer des éditeurs pour UML (chapitre 8). Ceci est expliqué par le fait que lors de la conception de notre proposition, nous avons été plus généralistes en récupérant les avantages des autres solutions et en essayant de résoudre leurs limitations.

10.1.3 Associer les éléments graphiques aux éléments du domaine

Dans notre contexte un langage de modélisation (Lm) est défini selon le tuple AS, CS^*, M^*_{ac} où AS est la syntaxe abstraite, CS^* est la (les) syntaxe(s) concrète(s) et M^*_{ac} est le mapping de syntaxe abstraite vers sa (ses) représentation(s) concrète(s) [Jezequel 2012]. Plusieurs problématiques se sont posées lors de notre spécification de la partie binding de notre proposition. Nous allons les énumérer une par une dans cette section.

Le premier besoin était naturellement d'associer *UN* élément de la syntaxe abstraite à *UN* élément graphique de la syntaxe concrète. Ce type d'association est le plus com-

mun et le plus simple dans la partie binding. Pour cela, nous avons proposé le concepts *TypeBinding* fait référence à des concepts (**EClassifier**) qui peuvent représenter un objet de la syntaxe abstraite, un fichier XML, une table dans une base de données, etc. *ValueBinding* associe un élément graphique (label par exemple) à une propriété accessible (**EStructuralFeature**) comme un champ d'une base de données ou un attribut d'un objet, etc.

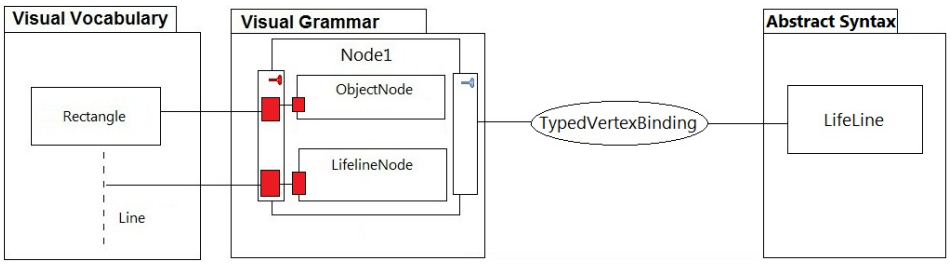


FIGURE 10.2 – Exemple de binding 1-N : Ligne de vie (Lifeline)

Le deuxième besoin en termes d'association avec la syntaxe abstraite est d'associer *UN* élément du domaine à *PLUSIEURS* éléments graphiques (par exemple, une ligne de vie - figure 10.2). Pour effectuer cette association avec *MID*, plusieurs solutions se proposent : 1) pour les éléments groupés, nous déclarons une interface de domaine unique pour cet ensemble d'éléments graphiques. 2) pour les éléments répartis, nous associons l'élément du domaine à toutes les interfaces de domaine de ces éléments.

Le troisième besoin en termes d'association avec la syntaxe abstraite est de pouvoir associer *PLUSIEURS* éléments du domaine à *UN* élément graphique. Dans ce cas on passe par une *ValueBinding* c'est-à-dire un binding associé à une expression, qui détermine le contenu spécifique à obtenir des différents éléments du domaine. Parmi les exemples d'utilisation de ce concept, on trouve l'élément *InstanceSpecification* qui comporte dans son label la syntaxe suivante : "**<nom de l'instance> : <nom de la classe>**". Ce label sera donc associé à une *ValueBinding* avec une expression qui lie les deux éléments du domaine (concaténation du nom de l'instance spécification et le nom de sa classe).

On utilise les expressions aussi au niveau graphique lorsqu'on définie des styles dynamiques, ces expressions peuvent associer des valeurs aux attributs graphiques (par exemple, couleur, taille, forme) selon la (les) valeur(s) d'un (des) élément(s) du domaine. On voit l'utilité de ce concepts dans l'exemple de l'association dans UML qui change de représentation selon la valeur de l'attribut *agregationKind* : Réference, Aggregation et Composition.

Dans notre proposition, les liens d'un diagramme peuvent représenter des références entre les métaclasses du métamodèle (syntaxe abstraite du domaine) ou des classes de liaison (par exemple, les associations UML). Pour cette raison, nous avons proposé le concept *TypedEdgeBinding* qui permet de lier un lien typé (Généralisation UML par exemple) à son concept correspondant ainsi que la spécification de la source et de la destination de ce lien. Dans le cas où le lien représente une référence entre deux métaclasses, on utilise le concept *ValueBinding* pour définir cette relation.

L'autre possibilité offerte par MID, est la définition de liens au niveau graphique. En

absence d'une description de lien au niveau de la syntaxe abstraite, nous pouvons définir la source et la destination d'un lien au niveau de syntaxe concrète lors de la définition de la grammaire visuelle.

10.1.4 Spécifier les interactions

Une interaction est définie comme un changement invoqué par l'utilisateur sur le modèle de domaine. Ce changement peut être de différentes catégories (Section 7.1.5), la plus importante est celle des interactions de l'éditeur ou les interactions graphiques. Parmi ces interactions on trouve l'événement de création (voir *Creation* - section 7.1.5.1) qui définie les éléments à créer dans un diagramme. Ainsi, un élément graphique qui déclare l'événement de création signifie qu'il sera créable à partir de la palette de création de l'éditeur. La palette de création est définie de deux façons : la première est explicite, en définissant chaque entrée de la palette et l'élément graphique qui lui est associé. La deuxième possibilité est la création de la palette automatiquement par déduction, à partir des éléments du diagramme déclarant l'événement *Creation*.

D'autres interactions graphiques concernent les éléments label du diagramme. On peut associer un comportement lorsqu'on édite un label avec l'interaction *SetData*. Ce comportement, peut être un traitement sur le modèle de domaine ou l'affichage d'un éditeur contextuel pour la validation de la valeur du label. Un label est par défaut en lecture/écriture, mais dans des cas précis, on a besoin de label en lecture seule (par exemple, les labels affichant des stéréotypes). Pour cette raison nous avons ajouté à l'élément label un attribut booléen (readOnly) pour définir cette option.

Parmi les interactions d'édition qui manquent dans notre proposition, on trouve la complétion automatique (auto-completion) pour suggérer des valeurs pour les éléments de l'éditeur à partir des données du modèle de domaine et la vérification de la saisie.

10.2 Réutiliser les éléments de diagrammes

La réutilisation a été depuis le début de nos travaux de recherche, le critère le plus important et le plus recherché. Ce critère nous a motivé à rechercher des méthodes qui permettent une meilleure réutilisation de modèles de spécification. Nous avons pu à travers le mécanisme d'héritage de réutiliser des concepts d'un modèle dans un autre. Ceci nous a permis de reprendre tout ou une partie de définition du modèle source et de la récupérer dans le modèle de destination avec la possibilité d'étendre cette définition existante. Cette extension ou spécialisation peut concerner toutes les caractéristiques d'un élément graphique du diagramme, à savoir, sa composition, sa représentation (Style), ses attributs graphiques (Couleur, taille, etc.) et ses interactions.

Nous avons aussi utilisé l'approche basée sur les composants pour assurer une meilleure lisibilité et une meilleure maintenance de modèles. Il est particulièrement utile pour le travail d'équipe et permet l'industrialisation du développement de ce type d'application. La réutilisation d'un composant permet d'économiser un gain important de productivité, car il permet de réduire le temps de développement, d'autant plus que le composant est réutilisé plus souvent.

Contrairement aux outils et méthodes existants pour la modélisation et la génération des éditeurs de diagrammes, nous avons séparé initialement les aspects graphiques, comportementaux et sémantiques de l'éditeur, nous avons séparé les deux aspects graphiques, qui sont le vocabulaire visuel (variables visuelles) et la grammaire visuelle (les règles de composition des éléments du diagramme). Par la suite, il était important de créer une autre partie qui effectue l'association entre les différents aspects, notamment entre la syntaxe abstraite et la syntaxe concrète graphique.

La séparation a été également réalisée dans la chaîne de transformation, en introduisant plusieurs modèles de niveau intermédiaire et en retardant l'introduction des détails techniques dans les derniers modèles de la chaîne. Ceci permet une meilleure maintenabilité de la chaîne de transformation lors des évolutions de nos méta-modèles.

Une forte séparation des préoccupations, permet une meilleure réutilisation et maintenance des modèles. Elle diminue les coûts de développement en termes de temps de maintenance en cas de changements dans ces modèles et devrait permettre la conception de nouvelles applications par assemblage de modèles existants.

En proposant la séparation des préoccupations avec un mécanisme d'héritage graphique, nous avons proposé une solution efficace pour la variabilité visuelle (question 5 de la problématique). Il est possible de réutiliser un élément graphique dans deux diagrammes différents avec deux représentations différentes, simplement en séparant sa composition (grammaire visuelle) de son affichage (vocabulaire visuel) et en utilisant l'héritage et la surcharge pour redéfinir les variations graphiques.

A travers les exemples présentés dans le chapitre 8, nous avons validé notre approche en termes de réutilisabilité : Cette approche nous a permis de réutiliser plus de 70 % des composants créés dans l'ensemble de la syntaxe concrète du langage UML, ce qui n'est pas négligeable en termes de coût et de temps de développement. Cette approche nous a permis de définir plus facilement les spécificités des éditeurs avec une approche dirigée par les modèles et sans qu'il soit nécessaire d'intervenir manuellement (programmatiquement) pour effectuer des changements, ce qui augmente le niveau de maintenabilité des éditeurs générées avec notre solution.

Parmi les choses qui manquent dans notre proposition, la possibilité de compléter un modèle de spécification de diagramme, sans toucher à ce dernier, c'est-à-dire d'ajouter une description graphique à un modèle existant sans modifier la description initiale, ni les artefacts générés à partir de celle-ci. Cette fonctionnalité est faisable au niveau du modèle en définissant des points d'entrées (appelé aussi join point) dans le modèle initial pour ajouter des modèles d'extension éventuels. Cependent, au niveau du code généré, cette fonctionnalité nécessite d'introduire des notions de la programmation orientée-aspect [Kiczales 1997] : ce paradime consiste à ce qu'un programme (Aspect) définisse des points de jonction (join points) où des extensions (greffons ou advices) qui viendront s'exécuter. Un des exemples de ce paradigme est le mécanisme d'extension Eclipse [Eclipse 2007] qui, à partir d'un point d'extension, peut étendre un programme déja existant.

CHAPITRE 11

Conclusion et Perspectives

Dans cette thèse, nous présentons une approche basée sur les modèles et les composants pour la conception des éditeurs graphiques de diagramme. Ceci permet la spécification rapide des éditeurs graphiques de diagrammes à un niveau d'abstraction élevé, afin de modéliser et de réutiliser, les concepts graphiques qui y sont manipulés. Dans notre proposition, nous nous concentrons sur le concept de composant pour décrire et ensuite assembler les nouveaux concepts des langages visuels (langage diagrammatique). Tout d'abord, nous présentons les définitions et les fondements théoriques des représentations visuelles et l'approche basée sur la méta-modélisation des composants tout en positionnant notre travail par rapport aux différents outils et technologies disponibles dans l'industrie et dans la littérature.

Dans notre approche, nous encourageons l'utilisation de composants réutilisables (avec la composition, l'encapsulation et l'héritage, etc.) et la séparation forte des préoccupations (domaine, élément graphique, les variables visuelles). Cela augmente la réutilisabilité des éditeurs de diagrammes et apporte les avantages du paradigme IDM comme la vérification/validation des modèles ou la possibilité de choisir les technologies "cibles" grâce à des techniques de transformation de modèles.

Nous avons donc proposé un cadre permettant de définir, à l'aide de modèles, la syntaxe graphique d'un langage de modélisation, c'est-à-dire les diagrammes avec toutes leurs préoccupations (les formes, les styles et la composition des éléments de diagramme) et les éditeurs sous-jacent à ces langages (outillage, menus, interactions sur les diagrammes, etc.) et cela en appliquant une démarche d'ingénierie dirigée par les modèles. Ce cadre s'appele MID (Metamodels for User interfaces and Diagrams).

MID prend en charge une description des interfaces graphiques pour les éditeurs graphiques de modélisation pour les langages visuels tels que les diagrammes UML, flowchart, petri nets, BPMN, etc. L'objectif de la description est de structurer le langage visuel et les comportements qui y sont associés (dans l'éditeur) tels qu'ils sont perçus par l'utilisateur final.

De point de vue fonctionnelle, **MID** est un ensemble de métamodèles et d'outillages qui permettent de répondre aux aspects suivants :

1. Définir la syntaxe concrète graphique (diagrammes) pour les langages de modélisation (DSML ou autres).
2. Effectuer le mapping entre les éléments de la syntaxe concrète et les éléments de la syntaxe abstraite (le métamodèle du domaine).
3. Définir l'éditeur qui manipule cette syntaxe concrète graphique en spécifiant les interactions et les outillages nécessaires pour sa manipulation.

4. Réutiliser les définitions graphiques déjà existantes via :
   - La Séparation des préoccupations (grammaire visuelle, vocabulaire visuel et interactions) pour augmenter les possibilités de réutilisation de chaque partie distinctement des autres.
   - L'utilisation de l'architecture MVC pour une meilleure cohérence entre les préoccupations.
   - L'utilisation d'une approche basée sur les composants pour bénéficier des techniques de composition et de la facilité de maintenance de la spécification offertes par cette approche.
   - Un mécanisme qui permet l'héritage des composants graphiques pour récupérer les descriptions existantes et pour créer d'autres descriptions dérivés de celles-ci.

D'un point de vue global de l'architecture Modèle-Vue-Contrôleur (MVC), MID spécifie la partie vue (View) et les parties de contrôle de l'application et décrit comment la vue dépend du modèle. En particulier :

- En ce qui concerne la vue, MID décrit la composition et la présentation des éléments graphiques des langages visuels qu'il expose à l'utilisateur dans des éditeurs graphiques.
- En ce qui concerne la partie contrôleur, MID permet au concepteur de spécifier les effets des interactions sur les éditeurs de diagramme en définissant les événements pertinents que le contrôleur doit prendre en charge.
- En ce qui concerne la partie modèle, MID permet la spécification des références à des objets de données qui incorporent l'état du modèle manipulé et qui sont exposés dans l'éditeur de diagramme, ainsi que des références à des actions qui sont déclenchées par l'interaction de l'utilisateur.
- MID décrit la composition des notations visuelles, en fonction de quelques unités de construction élémentaires et indépendantes, qui peuvent être affichées simultanément et peuvent être imbriqués hiérarchiquement.
- MID décrit le contenu de la vue, en termes à la fois, de représentation des notations visuelles (attributions des formes et d'autres variables au composant de diagramme) et de synchronisation avec les éléments de modèle (lecture/écriture).
- MID permet la définition de l'éditeur graphique manipulant les diagrammes et permettant l'interaction de l'utilisateur avec le langage édité.

Dans MID, nous résolvons certains problèmes identifiés dans les outils et méthodes existants dans l'industrie comme dans la littérature. Par exemple, la spécification à un haut niveau d'abstraction, sans la nécessité d'une intervention programmatique manuelle, la séparation des préoccupations, l'efficacité graphique et enfin la réutilisation des éditeurs, qui faisait partie des problématiques majeures de notre travail de recherche.

A travers les exemples présentés dans la validation, nous avons validé notre approche en termes de réutilisabilité : Cette approche nous a permis de réutiliser plus de 70 % des composants créés dans l'ensemble de la syntaxe concrète du langage UML, ce qui n'est pas négligeable en termes de coût de développement et de temps. Cette approche nous a permis de définir plus facilement les spécificités des éditeurs avec une approche dirigée par les modèles et sans qu'il soit nécessaire d'intervenir manuellement (programmati-

quement) pour effectuer des changements, ce qui augmente le niveau de maintenabilité des éditeurs générées avec notre solution.

Nous avons réussi aussi à valider le critère de l'expressivité et la complétude graphique. en choisissant de définir d'autres diagrammes complexes graphiquement que ceux d'UML. Par ailleurs, nous avons choisi deux cibles pour nos transformations pour montrer le niveau d'indépendance de notre solution par rapport à la technologie d'implémentation.

Notre approche présente plusieurs avantages : Tout d'abord, grâce à la réutilisation de composants sous forme de modèles : les modèles sont théoriquement plus faciles à comprendre et à manipuler par les utilisateurs finaux, ce qui correspond à un objectif de l'IDM. Ensuite, en gain de productivité : il est possible de constituer des librairies de composants, puis de construire son diagramme par assemblage de ces composants. Les éléments graphiques et les éléments du domaine peuvent évoluer séparément.

Nous pouvons dire que notre approche ouvre une nouvelle voie qui se montre prometteuse pour une plus large utilisation des outils de modélisation et de génération automatique d'applications. Par rapport aux technologies actuelles de développement, les promesses de cette approche sont importantes, grâce à la possibilité de créer des applications complexes en assemblant de simples fragments de modèles/composants existants et surtout la possibilité pour les non-informaticiens, experts dans leurs domaines d'activité, de créer leurs propres applications à partir d'une description de haut niveau en utilisant un formalisme adapté, facile à comprendre et à manipuler pour eux.

Dans l'état actuel de nos recherches, de nombreuses études sont encore nécessaires pour atteindre une génération complète d'outils tels que Papyrus. Tout d'abord, nous devons introduire progressivement notre proposition dans l'existant de Papyrus pour ne pas repartir à zéro dans la description des éditeurs graphiques de cet outil. Ceci peut être effectué dans un premier temps, en utilisant des styles custom pour lier l'implémentation en code de papyrus à notre solution. Par la suite, nous devons introduire la description des éléments redondants dans Papyrus, tels que les commentaires et les contraintes, qui sont présents dans tous les diagrammes, en suivant la même approche de la réutilisation de composants et d'héritage de MID. Enfin, nous devons alimenter cette description au fur et à mesure pour prendre à moyen terme tous les éléments de Papyrus avec notre proposition. Ainsi, une intégration progressive (douce) doit pouvoir utiliser notre solution dans Papyrus sans représenter une charge ou un cout important de maintenance. Au niveau architectural et toujours dans le but de spécifier Papyrus, notre solution doit prendre en charge certains points : elle doit permettre la description d'éditeur multi-diagrammes avec un modèle unique (la même source de données). Elle doit aussi prendre en charge le système de fichier utilisé dans Papyrus qui est constitué d'un triplet de fichiers : le modèle DI pour échanger les informations des diagrammes, le modèle de notation de GMF et le modèle UML représentant le domaine.

CHAPITRE 12

Annexe

12.1 Diagrammes UML avec MID

Cette section présente un exemple d'utilisation de la spécification *MID* pour définir les diagrammes UML. La sous section 12.1.1 fournie une spécification des diagrammes structurels de UML. La sous-section 12.1.2 présente les diagrammes comportementaux de UML et leur spécification avec *MID*.

Les diagrammes UML sont choisis en raison de leur utilisation répandue et leur familiarité dans le domaine du génie logiciel. Toutefois, afin de contrôler la portée (scope) de ce chapitre, les exemples sont limités à la spécification de la notation UML telle qu'elle est définie dans la spec. OMG de UML [OMG 2011c].

12.1.1 Diagrammes structurels

Cette section définie les concepts structurels, statiques utilisés dans les différents diagrammes structurels, tels que les diagrammes de classes, diagrammes de composants et les diagrammes de déploiement.

Les sous-section ci-dessous décrivent les éléments graphiques qui peuvent être présentés dans chacun des diagrammes de structure. Elles fournissent des informations détaillées sur la sémantique et la notation visuelle pour chaque élément dans ces diagrammes. Elles fournissent également des exemples qui illustrent comment ces éléments graphiques peuvent être spécifiés par *MID*.

12.1.1.1 Diagramme de classes

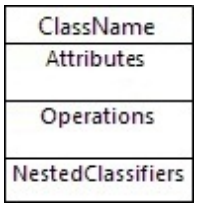
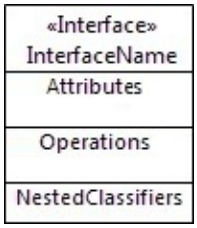
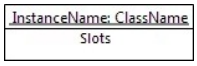
Un objet est une entité identifiable (peut être désignée) du monde réel. Il peut avoir une existence physique (par exemple, une voiture, une personne) ou ne pas en avoir (une structure de donnée). Un ensemble d'objets similaires, c'est-à-dire possédant la même structure et le même comportement et constitués des mêmes attributs et méthodes, forme une *classe* d'objets. La structure et le comportement peuvent alors être définis en commun au niveau de la classe [Debrauwer 2008].

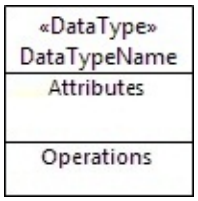
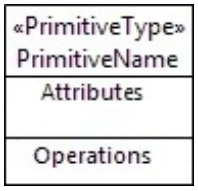
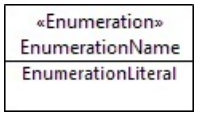
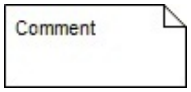
Le diagramme de classes est le diagramme le plus largement répandu dans les spécifications d'UML [Grimaldi 2007]. Il représente la structure statique des classes intervenant dans le système en faisant abstraction des aspects dynamiques et temporels. Le diagramme de classe est une représentation statique des éléments qui composent un système et de leurs relations (relation d'héritage par exemple).

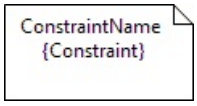
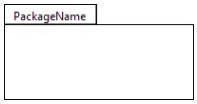
Le tableau 12.1 décrit les éléments graphiques qui peuvent être présentées dans le diagramme de classes et dans d'autres diagrammes de structure (tous les autres dia-

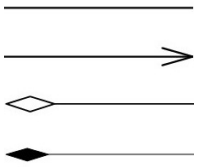
grammes de structure dépendent du diagramme de classes) et fournit des informations sur la sémantique et la notation visuelle pour chaque élément du diagramme.

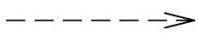
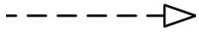
TABLE 12.1: éléments graphiques dans le diagramme de classes

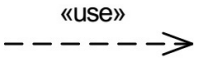
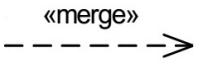
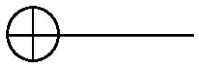
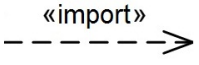
Concept	**Notation**	**Informations**
Class		*Sémantique* : une classe décrit un ensemble d'objets qui partagent les mêmes caractéristiques (attributs, opérations et les classificateurs qui sont imbriqués au sein de la classe), contraintes et sémantique. *Notation* : une classe est constituée d'un noeud représenté par un rectangle. Ce noeud contient un label pour le nom et trois conteneurs qui contiennent respectivement des attributs, des opérations et des classificateurs imbriqués (tous sous forme de labels).
Interface		*Sémantique* : une interface est un classificateur (*Classifier*) qui représente un ensemble de déclarations publiques. Une interface spécifie un contrat, l'instance de l'élément qui réalise l'interface doit remplir ce contrat. *Notation* : une Interface a quasiment la même notation visuelle que la classe à l'exception d'un label qui contient le stéréotype *«interface»*.
Instance Specification		*Sémantique* : une spécification d'instance (*InstanceSpecification*) est une classe concrète qui représente une instance dans un système. *Notation* : une spécification d'instance se constitue graphiquement d'un noeud représenté par un rectangle. Ce noeud contient un label dont la valeur est la concaténation <nom> :<type> suivi d'un conteneur d'attributs.

Concept	Notation	Informations
DataType		*Sémantique* : un type de données (*DataType*) est un type dont les instances sont identifiées uniquement par leur valeur. Un type de données peut contenir des attributs pour soutenir la modélisation des types de données structurées. *Notation* : un type de données est constitué d'un noeud représenté par un rectangle. Ce noeud contient un label avec le stéréotype *«DataType»* suivi d'un autre label pour le nom et deux conteneurs qui contiennent respectivement des attributs et des opérations.
PrimitiveType		*Sémantique* : un type primitif (*PrimitiveType*) définit un type de données prédéfini, sans aucune sous-structure correspondante. Un type de données primitif peut avoir une algèbre et des opérations définies à l'extérieur de UML. *Notation* : un type primitif est constitué d'un noeud représenté par un rectangle. Ce noeud contient un label avec le stéréotype *«PrimitiveType»* suivi d'un autre label pour le nom et deux conteneurs qui contiennent respectivement des attributs et des opérations.
Enumeration		*Sémantique* : une énumération est un type de données dont les valeurs sont énumérées dans le modèle comme des littéraux d'énumération (*EnumerationLiteral*). *Notation* : une énumération est constituée graphiquement d'un noeud représenté par un rectangle. Ce noeud contient un label avec le stéréotype *«Enumeration»* et un autre noeud pour le nom suivi d'un conteneur pour les littéraux d'énumération.
Comment		*Sémantique* : un commentaire est une annotation textuelle qui peut être attachée à un ensemble d'éléments. *Notation* : graphiquement, un commentaire est un noeud avec la forme de *Note* qui contient un label pour le corps.

Concept	Notation	Informations
Constraint		*Sémantique* : une contrainte est une condition ou une restriction exprimée en texte (en langage naturel ou en langage exécutable par une machine) qui peut être attaché à un élément dans le but de lui déclarer une sémantique. *Notation* : graphiquement, Une contrainte est constituée d'un noeud avec la forme de *Note* qui contient un label pour le nom de la contrainte et un autre pour le corps de la contrainte.
Package		*Sémantique* : un paquetage (*Package*) est utilisé pour contenir et grouper des éléments, il représente l'espace de noms (*namespace*) des éléments groupés. *Notation* : un paquetage est constitué d'un noeud qui contient un label pour le nom et un conteneur d'éléments.

Concept	Notation	Informations
Association		*Sémantique* : une association spécifie une relation sémantique qui peut se produire Les associations ont par défaut une navigation bi-directionnelle, c'est-à-dire qu'il est possible de déterminer les liens de l'association depuis une instance de chaque classe d'extrémité. Une association peut représenter une relation de composition entre deux objets. Il existe deux formes de composition, forte ou faible. La composition forte est la relation entre un élément (composite) et les éléments qui en font partie (composant). La suppression de l'objet composé entraîne la suppression de ces composants. La composition faible est appelée agrégation, elle impose beaucoup moins de contraintes aux composants que la composition forte. Dans ce cas, les composants peuvent être partagés par plusieurs composés et destruction du composé ne conduit pas à la destruction des composants. *Notation* : cette relation est construite graphiquement à partir d'une arête représentée par un style dynamique. Ce style comporte trois variations graphiques (voir "case" figure 7.18). Le premier, pour les références avec un lien plain avec un polyline dans l'extrémité de destination pour former la flèche. Le deuxième, pour les agrégations avec un lien plain comportant un polygone sous forme de losange blanc dans l'extrémité de source et un polyline sous la forme d'une flèche dans l'extrémité de destination. La troisième variation pour les compositions, est représentée par un lien plain comportant un polygone sous forme de losange noir dans l'extrémité de source et un polyline sous la forme d'une flèche dans l'extrémité de destination.

Concept	Notation	Informations
Dependency		*Sémantique* : une dépendance est une relation qui signifie qu'un élément nécessite un autre élément pour son implémentation ou sa spécification. Cela signifie que la sémantique complète des éléments dépendants (*client*) nécessite sémantiquement ou/et structurellement sur la définition de l'élément fournisseur (*supplier*). *Notation* : une dépendance est constituée graphiquement d'une arête sous forme de ligne en traits avec une flèche dans l'extrémité de destination.
Generalization		*Sémantique* : une généralisation est une relation taxonomique (du même genre) entre un classificateur plus général et un classificateur plus spécifique. Ainsi les instances d'une classes aussi les instances de sa ou ses superclasses. Par conséquent, elles héritent les attributs et les méthodes définis dans la ou les superclasses. *Notation* : une généralisation est constituée d'une arête sous forme de ligne plaine avec un polygone sous forme de triangle blanc dans l'extrémité de destination.
Realization		*Sémantique* : la réalisation est une relation d'abstraction spécialisée entre deux éléments, l'un représentant une spécification (le fournisseur ou *supplier*) et l'autre représente une implémentation de celui-ci (*client*). La réalisation peut être utilisée pour modéliser les optimisations, les transformations, les templates, la composition des Frameworks, etc. *Notation* : la réalisation est constituée d'une arête sous forme de ligne en traits avec un polygone sous forme de triangle blanc dans l'extrémité de destination.

Concept	Notation	Informations
Usage		*Sémantique* : un usage est une relation dans laquelle un élément nécessite un autre élément (ou un ensemble d'éléments) pour son implémentation ou à son fonctionnement. Dans le métamodèle, un usage est une dépendance dans laquelle le client nécessite la présence du fournisseur (*supplier*). *Notation* : un usage est constitué d'une arête sous forme de ligne en traits avec une flèche dans l'extrémité de destination. Cette arête contient un label stéréotypé par *«use»*.
Package Merge		*Sémantique* : une fusion est une relation orientée entre deux packages qui indique que leurs contenus doivent être combinées. *Notation* : une fusion est constituée d'une arête sous forme de ligne en traits avec une flèche dans l'extrémité de destination. Cette arête contient un label stéréotypé par *«merge»*.
Containment		*Sémantique* : la composition peut être exprimée en imbriquant l'élément composant au sein de l'élément composite ou en utilisant le lien de confinement (*containment*). *Notation* : cette relation est constituée d'une arête sous forme de ligne plaine avec un cercle contenant une croix dans l'extrémité source.
Package Import		*Sémantique* : un import de package est défini comme une relation orientée qui identifie un package dont les membres doivent être importés par un espace de noms (*namespace*). *Notation* : un import de package est constitué d'une arête sous forme de ligne en traits avec une flèche dans l'extrémité de destination. Cette arête contient un label stéréotypé par *«import»*.

Le digramme de paquetage (*Package*) fait aussi partie des diagrammes structuels

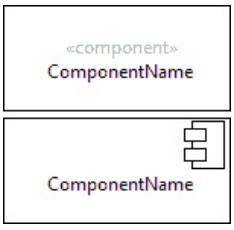
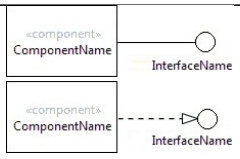
d'UML, il représente les relations existantes entre les paquetages (ou espaces de noms) composant un système. Graphiquement, ce diagramme est composé d'un sous-ensemble du diagramme de classes, il comporte l'élément Package, et les liens import, merge, containment et dependency décrits précédemment.

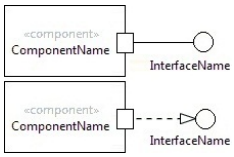
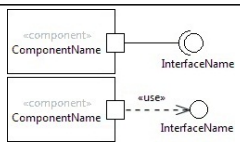
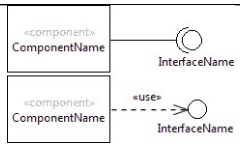
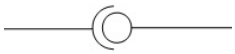
12.1.1.2 Diagramme de composants

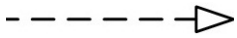
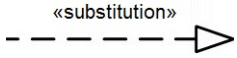
Le diagramme de composants spécifie un ensemble de constructions qui peuvent être utilisées pour définir des systèmes logiciels de taille et de complexité arbitraire. Il décrit l'organisation du système du point de vue des éléments logiciels comme les modules (paquetages, fichiers sources, bibliothèques, exécutables), des données (fichiers, bases de données) ou encore d'éléments de configuration (paramètres, scripts, fichiers de commandes) [Wikipedia 2013c].

Le tableau 12.2 décrit les éléments graphiques qui peuvent être présentées dans le diagramme de composants et dans d'autres diagrammes de structure (par exemple le diagramme de déploiement) et fournit des informations sur la sémantique et la notation visuelle pour chaque élément du diagramme.

TABLE 12.2: éléments du diagramme de composants

Concept	Notation	Informations
Component		*Sémantique* : un composant (*component*) représente une partie modulaire d'un système qui encapsule son contenu et dont la manifestation est remplaçable dans son environnement. Un composant définit son comportement en termes d'interfaces fournies et requises. En tant que tel, un composant sert comme type dont la conformité sémantique est définie par ces interfaces fournies et requises. *Notation* : un composant est constitué graphiquement d'un noeud rectangulaire qui contient un label stéréotypé «*component*», un label pour le nom et un conteneur pour les sous-composants.
Interface implementation		*Sémantique* : un composant fourni une interface.

Concept	Notation	Informations
Un composant avec un port fourni (typé par une interface)		*Sémantique* : un port est une propriété d'un composant qui spécifie un point de liaison (passerelle) entre ce composant et son environnement ou entre le (les) comportements du composant et ses parties internes. Les interfaces associées à un port spécifient la nature des interactions qui peuvent se produire sur un port. Un port peut préciser les services qu'un composant fournie (offre) à son environnement. *Notation* : un port est spécifié graphiquement avec un noeud de bordure attaché au noeud "composant". Ce noeud de bordure est de forme rectangulaire et contient un label externe pour le nom.
Un composant avec un port requis (typé par une interface)		*Sémantique* : un port peut préciser les services qu'un composant attend (exige) de son environnement.
Interface usage		*Sémantique* : un composant exige une interface.
Assembly Connector		*Sémantique* : un connecteur d'assemblage (*assemblyconnector*) est une relation entre un ou plusieurs composants ou ports de composants qui definie qu'une partie (la source du connecteur) fournie un service que l'autre partie utilise. *Notation* : ce connecteur est formé par une arête sous forme d'une ligne plaine avec une decoration de milieu composée d'un cercle et d'un arc qui le contourne.

Concept	Notation	Informations
Component Realization		*Sémantique* : la relation de réalisation de composants (*ComponentRealization*) définie les classificateurs (*Classifier*) qui réalisent le contrat proposé par un composant en fonction de ses interfaces fournies et requises. *Notation* : la réalisation est constituée d'une arête sous forme de ligne en traits avec un polygone sous forme de triangle blanc dans l'extrémité de destination.
Component Substitution		*Sémantique* : un composant peut donc être substitué par un autre, si tout les deux sont conforme à un type. *Notation* : la réalisation est constituée d'une arête sous forme de ligne en traits avec un polygone sous forme de triangle blanc dans l'extrémité de destination. Cette arête contient un label stéréotypé *«substitution»*.

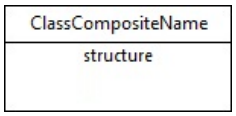
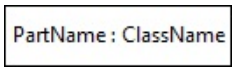
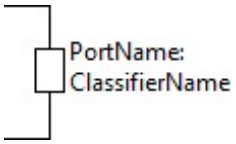
12.1.1.3 Diagramme de structure composite

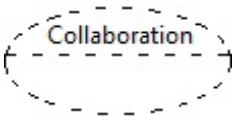
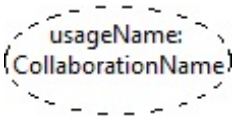
Le diagramme de structure composite ou structure interne prévoit des mécanismes permettant de spécifier des structures d'éléments interconnectés qui sont créés dans une instance d'un classificateur composite. Une structure de ce type représente une décomposition du classificateur désignée comme la "structure interne". Dans le diagramme de structure composite, l'objet composé décrit par un classifcateur tandis que ses composants sont décrits par des parties.

Le tableau 12.2 décrit les éléments graphiques qui peuvent être présentées dans le diagramme de structure composite et fournit des informations sur la sémantique et la notation visuelle pour chaque élément du diagramme.

TABLE 12.3: éléments du diagramme de structure composite

Concept	Notation	Informations

Concept	Notation	Informations
Class		*Sémantique* : étend le concept classe (voir 12.1) en ajoutant la capacité d'avoir une structure interne et des ports. *Notation* : comme les autres classificateurs, une classe composite se constitue graphiquement d'un noeud rectangulaire qui contient un label pour le nom et un conteneur de parts.
Part		*Sémantique* : une propriété (*Part*) représente un ensemble d'instances qui sont détenues par une instance de la classe composite. *Notation* : cet élément graphique est formé d'un noeud rectangulaire contenant un label dont la valeur est la concaténation <nom du Part> :<nom de la classe>
Port		*Sémantique* : un port est une propriété d'une classe composite qui spécifie un point de liaison (passerelle) entre cette classe et sa structure internes. *Notation* : voir diagramme de composants (section 12.1.1.2).

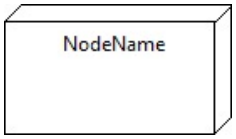
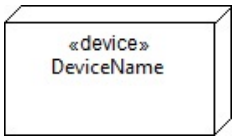
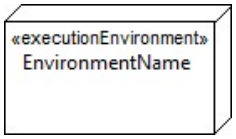
Concept	Notation	Informations
Collaboration		*Sémantique* : une collaboration décrit une structure d'éléments de collaboration (rôles), remplissant chacun une fonction spécialisée et accomlissant collectivement certaines fonctionnalités souhaitées. Son but principal est d'expliquer comment un système fonctionne et par conséquent, il ne comprend généralement que les aspects qui sont jugées pertinentes pour cette explication. *Notation* : une collaboration est constituée graphiquement d'un noeud de forme elliptique et de bordure en traits contenant un label pour le nom de la collaboration et un conteneur de *CollaborationUse*.
Collaboration Use		*Sémantique* : une utilisation de collaboration (*CollaborationUse*) représente une utilisation particulière d'une collaboration pour expliquer les relations entre les propriétés d'un classificateur. *Notation* : elle se contitue d'un noeud elliptique avec une bordure en traits, contenant un label dont la valeur est la concaténation du <nom de l'usage> :<nom de la collaboration>.
Connector		*Sémantique* : indique un lien qui permet la communication entre deux ou plusieurs instances. *Notation* : ce lien est constitué par une arête sous forme de ligne simple.
Role Binding		*Sémantique* : une mapping entre les éléments de la collaboration et les éléments de la classe propriétaire. Cette correspondance indique l'élément de la classe qui joue un/des rôle(s) dans une collaboration. *Notation* : ce lien est constitué par une arête sous forme d'une ligne en traits.

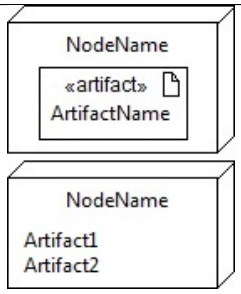
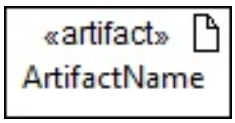
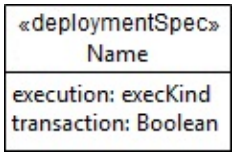
12.1.1.4 Diagramme de déploiement

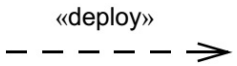
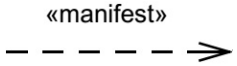
Le diagramme de déploiement spécifie un ensemble de constructions qui peuvent être utilisés pour définir l'architecture (physique) d'exécution des systèmes. Il sert à représenter l'utilisation de l'infrastructure physique par le système et la manière dont les composants du système sont répartis ainsi que les relations entre eux [Wikipedia 2013d].

Le tableau 12.7 décrit les éléments graphiques qui peuvent être présentées dans le diagramme de déploiement et fournit des informations sur la sémantique et la notation visuelle pour chaque élément du diagramme.

TABLE 12.4: éléments du diagramme de déploiement

Concept	Notation	Informations
Node		*Sémantique* : un noeud (*Node*) est une ressource de calcul sur laquelle les artefacts (*Artifact*) peuvent être déployés pour l'exécution. *Notation* : cet élément est constitué d'un noeud graphique sous forme d'un cube et qui contient un label pour le nom.
Device		*Sémantique* : un device est une ressource de calcul physique avec une capacité de traitement sur laquelle les artefacts (*Artifact*) peuvent être déployés pour l'exécution. *Notation* : cet élément est constitué d'un noeud graphique sous forme d'un cube et qui contient un label avec le stéréotype *«device»* et un autre label pour le nom.
Execution Environment		*Sémantique* : c'est un noeud qui offre un environnement d'exécution pour les artifacts qui y sont déployés. *Notation* : la même notation graphique que le device.

Concept	Notation	Information
Node deploying an artifact		Voir Deployment.
Artifact		*Sémantique* : un artefact (*Artifact*) est la spécification d'un élément physique de l'information qui est utilisée ou produite par un processus de développement logiciel. Par exemple les fichiers sources, les scripts, les fichiers executables, etc. *Notation* : cet élément est constitué d'un noeud rectangulaire qui contient un label avec le stéréotype *«artifact»* et un autre label pour le nom de l'artifact.
Deployment Specification		*Sémantique* : une spécification de déploiement (*DeploymentSpecification*) spécifie un ensemble de propriétés qui déterminent les paramètres d'exécution d'un artefact déployé sur un noeud. *Notation* : comme les autres classificateurs, une spécification de déploiement est constituée d'une noeud rectangulaire qui contient un label pour le stéréotype *«deploymentSpec»*, un autre label pour le nom et un conteneur d'attributs.

Concept	Notation	Information
Dependency		*Sémantique* : cette notation est utilisée pour représenter les associations suivantes : (1) la relation entre un artifact et l'élément de modèle qu'il implemente (2) le déploiement d'un artefact sur un noeud (*Node*) par exemple. *Notation* : une dépendance est constituée graphiquement d'une arête sous forme de ligne en traits avec une flèche dans l'extrémité de destination.
Deployment		*Sémantique* : un déploiement est l'attribution d'une instance d'artefact ou d'un artefact à une cible de déploiement. *Notation* : cet élément est constitué graphiquement d'une arête sous forme de ligne en traits avec une flèche dans l'extrémité de destination. Cette arête contient un label stéréotypé «*deploy*».
Manifestation		*Sémantique* : une manifestation physique est le rendu concret d'un ou plusieurs éléments du modèle par un artefact. *Notation* : cet élément est constitué graphiquement d'une arête sous forme de ligne en traits avec une flèche dans l'extrémité de destination. Cette arête contient un label stéréotypé «*manifest*».

12.1.2 Diagrammes comportementaux

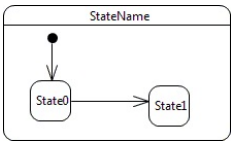
Cette partie spécifie les concepts des comportements dynamiques (par exemple, les activités, les cas d'utilisation, les machines à états) utilisés dans les différents diagrammes comportementaux, tels que les diagrammes d'activité, diagrammes de cas d'utilisation et des diagrammes d'états-transitions.

12.1.2.1 Diagramme d'états-transitions

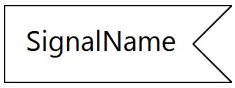
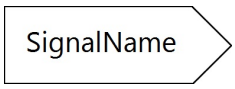
Le diagramme d'état-transitions illustre l'ensemble des états du cycle de vie d'un objet séparés par des transitions. Chaque transition est associée à un événement [Debrauwer 2008]. Il est utilisé en génie logiciel pour représenter des automates déterministes. Il s'inspire principalement du formalisme des statecharts [Harel 1987].

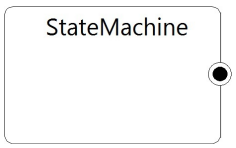
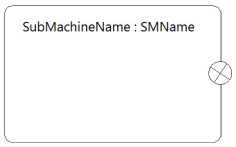
Le tableau 12.5 décrit les éléments graphiques qui peuvent être présentées dans le diagramme d'états-transitions et fournit des informations sur la sémantique et la notation visuelle pour chaque élément du diagramme.

TABLE 12.5: éléments du diagramme d'états-transactions

Concept	Notation	Informations
Choice pseudostate		*Sémantique* : évaluation dynamique des transitions entrantes et déclencheur de transitions sortantes. *Notation* : cet élément est constitué graphiquement d'un noeud représenté par un losange et qui contient un label externe pour le nom.
Composite State		*Sémantique* : un état composite contient une ou deux régions orthogonales. Chaque région contient d'autres états et transitions. *Notation* : cet élément est constitué graphiquement d'un noeud représenté par un rectangle arrondi et qui contient un label pour le nom et un conteneur d'états.
Entry Point		*Sémantique* : le point d'entrée d'une machine à états ou d'un état composite. *Notation* : cet élément est construit graphiquement d'un noeud de forme circulaire avec un label externe pour le nom.
Exit Point		*Sémantique* : le point de sortie d'une machine à états ou d'un état composite. *Notation* : cet élément est construit graphiquement d'un noeud de forme circulaire qui contient une croix au milieu avec un label externe pour le nom.

Concept	Notation	Informations
Final State		*Sémantique* : un état qui signifie que la région qui le contient est terminée. *Notation* : cet élément est construit graphiquement d'un noeud représenté par un deux cercles imbriqués le premier blanc et celui de l'intérieur est noir. Ce noeud contient un label externe pour le nom.
History, Deep pseudostate		*Sémantique* : représente la configuration active la plus récente de l'état composite qui contient directement ce pseudo-état. *Notation* : cet élément est construit graphiquement d'un noeud représenté par un cercle qui contient le mot "H*". Ce noeud contient un label externe pour le nom.
History, Shallow pseudostate		*Sémantique* : représente le sous-état actif le plus récent de l'état composite qui contient ce pseudo-état. *Notation* : cet élément est construit graphiquement d'un noeud représenté par un cercle qui contient la lettre "H". Ce noeud contient un label externe pour le nom.
Initial pseudostate		*Sémantique* : représente l'élément source d'un seul passage d'un état vers un autre. Il ne peut y avoir qu'un seul état initial dans une région. *Notation* : cet élément est construit graphiquement d'un noeud représenté par un cercle noir. Ce noeud contient un label externe pour le nom.

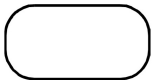
Concept	Notation	Informations
Receive Signal Action		*Sémantique* : il représente un déclencheur de transition à partir d'un signal reçu. *Notation* : cet élément est construit graphiquement d'un noeud représenté par un polygone sous forme rectangulaire à l'exception du côté droit qui est sous forme de flèche entrante. Ce noeud contient un label interne pour le nom.
Send Signal Action		*Sémantique* : il envoie un signal à partir d'une transition reçue. *Notation* : cet élément est construit graphiquement d'un noeud représenté par un polygone sous forme rectangulaire à l'exception du côté droit qui est sous forme de flèche sortante. Ce noeud contient un label interne pour le nom.
Region		*Sémantique* : une région est une partie orthogonale d'un état composite ou d'une machine à états. Elle contient des états et des transitions. *Notation* : cet élément est spécifié à partir d'un conteneur d'élément qui s'imbrique sur un état ou sur d'autres régions.
State		*Sémantique* : une situation implicite d'un élément vérifiant un ensemble de conditions. L'état d'un objet correspond à un moment de son cycle de vie. *Notation* : cet élément est constitué à partir d'un noeud représenté par un rectangle arrondi. Ce noeud contient un label interne pour le nom de l'état.

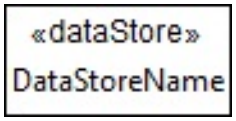
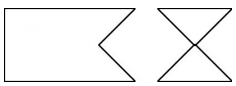
Concept	Notation	Informations
State Machine		*Sémantique* : les machines à états peuvent être utilisées pour exprimer le comportement d'une partie d'un système. *Notation* : même notation que l'état simple, sauf que la machine à état peut contenir d'autres élément (régions, états) et dispose du noeud de bordure "Final State".
Submachine State		*Sémantique* : une sous-machine à états indique l'insertion de la spécification d'une machine à états dans une autre. *Notation* : même notation que la machine à état, à l'exception que cet élément contient un "Exit Point" comme noeud de bordure au lieu de l'élément "Final State".
Terminate		*Sémantique* : ce pseudo-état implique que l'exécution de la machine à états qui le contient est terminée. *Notation* : cet élément est représenté par un noeud de bordure sous forme de croix (polyline) qui s'attache à un état ou une machine à état.
Transition		*Sémantique* : une relation qui représente le passage d'un état à un autre. *Notation* : une transition est constituée graphiquement d'une arête sous forme de ligne plaine avec une flèche dans l'extrémité de destination.

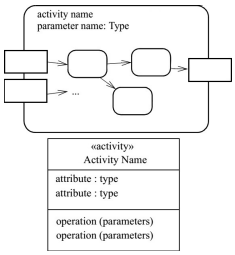
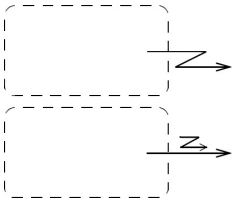
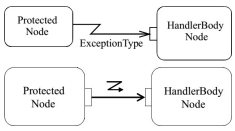
12.1.2.2 Diagramme d'activités

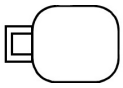
Le diagramme d'activités est basé sur le diagramme d'états-transitions. Il permet de représenter un enchaînement d'événements et/ou activités d'un système ou de ses composants. Ainsi, il peut être utilisé pour décrire un processus métier ou flux de travail (workflow). Ce diagramme offre des alternatives grâce aux conditions de garde. Il peut contenir des enchaînement de type fourche (*fork*) et synchronisation pour gérer des activités parallèles [Debrauwer 2008].

TABLE 12.6: éléments du diagramme d'activités

Concept	Notation	Informations
Action		*Sémantique* : une action est l'unité fondamentale d'une fonctionnalité exécutable. L'exécution d'une action représente une transformation ou traitement dans le système modélisé que ce soit un système informatique ou autre. une action consiste à affecter une valeur à un attribut, créer ou détruire un objet, effectuer une opération, envoyer un signal, etc. *Notation* : cet élément est constitué à partir d'un noeud représenté par un rectangle arrondi. Ce noeud contient un label interne pour le nom de l'action.
Activity Final		*Sémantique* : cet élément arrête toutes les procédures d'exécution de l'activité. *Notation* : cet élément est construit graphiquement d'un noeud représenté par un deux cercles imbriqués le premier blanc et celui de l'intérieur est noir. Ce noeud contient un label externe pour le nom.
Flow Final		*Sémantique* : un élément qui termine un flot (flow). *Notation* : cet élément est construit graphiquement d'un noeud de forme circulaire qui contient une croix au milieu avec un label externe pour le nom.
Initial		*Sémantique* : un élément de commande auquel le flux commence lorsque l'activité est appelée. *Notation* : cet élément est construit graphiquement d'un noeud représenté par un cercle noir. Ce noeud contient un label externe pour le nom.

Concept	Notation	Informations
Decision/Merge		*Sémantique* : un élément de contrôle qui choisit entre les flux sortants. *Notation* : cet élément est constitué graphiquement d'un noeud représenté par un losange et qui contient un label externe pour le nom.
Fork/Join		*Sémantique* : un élément fouche (fork) est un noeud de commande qui divise un flux en plusieurs flux simultanés (concurrents). un élément de jointure (join) est un noeud de commande qui synchronise plusieurs flux. *Notation* : les éléments fork et join sont représenté par des noeuds représentés par un segment de ligne vertical.
DataStore		*Sémantique* : un buffer central pour les informations non-transitoires. *Notation* : cet élément est spécifié graphiquement par un noeud représenté par un rectangle. Ce noeud contient un label stéréotypé «*dataStore*» et un autre label pour le nom de l'élément.
Send Signal Action		*Sémantique* : il envoie un signal à partir d'une transition reçue. *Notation* : cet élément est construit graphiquement d'un noeud représenté par un polygone sous forme rectangulaire à l'exception du côté droit qui est sous forme de flèche sortante. Ce noeud contient un label interne pour le nom.
Accept Event Action		*Sémantique* : il représente un déclencheur de transition à partir d'un signal reçu. *Notation* : cet élément est construit graphiquement d'un noeud représenté par un polygone sous forme rectangulaire à l'exception du côté droit qui est sous forme de flèche entrante.

Concept	Notation	Informations
Activity		*Sémantique* : une activité est une série d'actions. *Notation* : cet élément est constitué à partir d'un noeud représenté par un rectangle arrondi. Ce noeud contient un label interne pour le nom de l'activité. Une autre notation possible pour cet élément. Elle est semblable à celle d'un classificateur, elle se constitue d'un label stéréotypé «*activity*», d'un autre label pour le nom ainsi que de deux conteneurs pour les attributs et les opérations.
Interruptible Activity Region		*Sémantique* : une région d'activité interruptible est un groupe d'activités qui prend en charge la terminaison des jetons circulant dans une activité *Notation* : cet élément est constitué d'un noeud représenté par un rectangle arrondi et pointillé avec une ligne dans la bordure sous forme d'éclair, terminée par une flèche.
Exception Handler		*Sémantique* : un gestionnaire d'exception est un élément qui spécifie un corps à exécuter dans le cas où l'exception spécifiée se produit pendant l'exécution du noeud protégé. *Notation* : Un gestionnaire d'exceptions pour un noeud protégé est représenté par une arête représentée par une ligne sous forme "d'éclair" à partir de la limite du noeud protégé à un noeud de bordure sous forme de petit carré sur la bordure du gestionnaire d'exceptions.

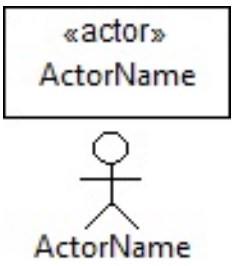
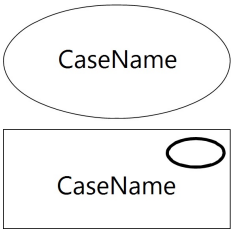
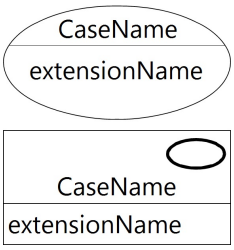
Concept	Notation	Informations
Partition		*Sémantique* : une partition est un groupe d'activités qui permet d'identifier les actions qui ont une caractéristique en commun. *Notation* : cet élément est construit graphiquement à partir d'un noeud de forme rectangulaire qui contient un label pour le nom de la partition et un conteneur d'éléments. La composition de ce noeud, est arrangée horizontalement de gauche à droite (voir layoutkind - figure 7.2).
Parameter Set		*Sémantique* : cet élément fournit des ensembles d'entrées et sorties qu'un comportement peut utiliser. *Notation* : cet élément est constitué graphiquement d'un noeud de bordure rectangulaire. Ce noeud s'attache sur la bordure d'une activité.

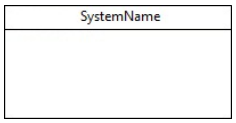
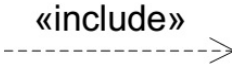
12.1.2.3 Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation décrit sous la forme d'actions et de réactions, le comportement d'un système du point de vue des utilisateurs. Ils définissent les limites du systèmes et ses relations avec son environnement. Ainsi l'ensemble des cas d'utilisation d'un système contient les exigences fonctionnelles attendues ou existantes, les acteurs (utilisateurs du systèmes) ainsi que les relations qui unissent acteur et fonctionnalités. Cet ensemble détermine les frontières du système, à savoir les fonctionnalités remplies par le système et celles qui lui sont externes [Debrauwer 2008].

TABLE 12.7: éléments du diagramme de cas d'utilisation

Concept	Notation	Informations

Concept	Notation	Informations
Actor		*Sémantique* : un acteur spécifie un rôle joué par un utilisateur ou tout autre système qui interagit avec l'objet (*subject*). *Notation* : cet élément est spécifié à partir d'un noeud représenté par un polygone avec la forme d'un bonhomme (stick man) et qui contient un label pour le nom de l'acteur. Une notation alternative est celle d'un noeud rectangulaire qui contient un label stéréotypé *«actor»* et un label pour le nom de l'acteur.
Use Case		*Sémantique* : un cas d'utilisation est la spécification d'un ensemble d'actions effectuées par un système et qui retourne un résultat observable pour un ou plusieurs acteurs ou d'autres intervenants du système. *Notation* : cet élément est construit graphiquement à partir d'un noeud de forme elliptique qui contient un label pour le nom. Ce noeud contient aussi des noeuds de formes textuelles pour représenter les points d'extension.
Extension Point		*Sémantique* : un point d'extension identifie un point dans un cas d'utilisation, dans lequel le comportement de ce cas d'utilisation peut être étendu. *Notation* : une extension est représentée par un noeud de forme textuelle contenu par un cas d'utilisation.

Concept	Notation	Informations
Subject		*Sémantique* : étend le concept classificateur (*classifier*) en ajoutant la capacité de posséder des cas d'utilisation. Le *Subject* représente généralement l'objet auquel les cas d'utilisation appartiennent. *Notation* : cet élément est constitué d'un noeud rectangulaire qui contient un label de nom et un conteneur d'éléments.
Extend		*Sémantique* : une relation entre un cas d'utilisation d'extention et un cas d'utilisation étendu qui précise comment et quand le comportement d'un cas d'utilisation (d'extention) peut être inséré dans le comportement d'un cas d'utilisation (étendu). *Notation* : cet élément est constitué d'une arête sous forme de ligne en traits avec une flèche dans l'extrémité de destination. Cette arête contient un label stéréotypé par *«extend»*.
Include		*Sémantique* : une relation qui définit qu'un cas d'utilisation contient le comportement défini dans un autre cas d'utilisation. *Notation* : cet élément est constitué d'une arête sous forme de ligne en traits avec une flèche dans l'extrémité de destination. Cette arête contient un label stéréotypé par *«include»*.

12.1.3 Diagrammes d'Interaction

Nous avons vu dans la section 12.1.2 que les objets d'un système possèdent leurs propre comportement et interagissent entre eux afin de doter le système de sa dynamique globale. Dans cette section, nous allons présenter les diagrammes d'interaction à savoir le diagramme de séquence et le diagramme de temps. D'autres diagrammes font partie de cette catégorie comme le diagramme de communication ou le diagramme globale d'interaction, ils se composent des mêmes éléments que les diagrammes cités

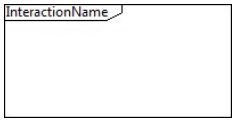
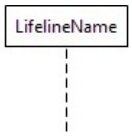
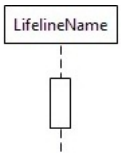
précédemment.

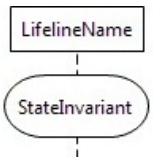
12.1.3.1 Diagramme de séquence

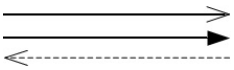
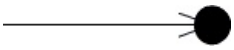
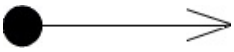
Le diagramme de séquence décrit la dynamique du système étudié. Il décrit les interactions, sous forme de message, entre les acteurs et le système selon un ordre chronologique.

Dans le diagramme de séquences, on représente l'acteur principal à gauche du diagramme, et les acteurs secondaires éventuels à droite du système. Le but étant de décrire comment se déroulent les actions entre les acteurs ou objets.

TABLE 12.8: éléments du diagramme de séquence

Concept	Notation	Informations
Frame		*Sémantique* : une interaction est une unité de comportement qui met l'accent sur l'échange observable d'informations entre les éléments. *Notation* : cet élément est constitué d'un noued rectangulaire avec un label pour le nom du cadre et d'un conteneur d'éléments.
Lifeline		*Sémantique* : une ligne de vie (*lifeline*) représente une unité individuelle d'interaction. *Notation* : l'élément *Lifeline*, est définie par un noeud représenté par une figure composée d'un rectangle qui représente l'objet et d'une ligne en traits qui représente la ligne de temps.
Execution Specification		*Sémantique* : Un périodes d'activité d'un objet (*ExecutionSpecification*) est la spécification de l'exécution d'une unité de comportement ou d'action au sein de la ligne de vie. *Notation* : Pour la définition des périodes d'activité de l'objet (*ExecutionSpecifications*), on ajoute un noeud de bordure rectangulaire sur la ligne de temps.

Concept	Notation	Informations
Combined Fragment		*Sémantique* : Un fragment combiné est une expression d'interaction défini par un opérateur de l'interaction et des opérandes d'interaction correspondantes. *Notation* : cet élément est constitué d'un noued rectangulaire avec un label pour le nom du cadre et d'un conteneur d'éléments.
StateInvariant		*Sémantique* : un StateInvariant est une contrainte d'exécution sur les participants de l'interaction. Il peut être utilisé pour spécifier une variété de différents types de contraintes, telles que les valeurs des attributs ou variables, des états externes ou internes, et ainsi de suite. *Notation* : Pour la définition des *StateInvariant*, on ajoute un noeud de bordure sous forme elliptique sur la ligne de temps.
Destruction Occurrence Sprecification		*Sémantique* : cet élément signifie la distruction de l'objet suite à un message. *Notation* : cet élément est définie par un noeud de bordure sous forme croix (polyline) sur la ligne de temps. .

Concept	Notation	Informations
Message		*Sémantique* : un message définie une communication entre les lignes de vie dans une interaction. *Notation* : cet élément est définie par une arête sous forme de ligne. Un message asynchrone est représenté par une ligne avec une flèche ouverte dans l'extrémité de destination. Un message asynchrone est représenté par une ligne avec un triangle noir dans l'extrémité de destination. Un message de retour est représenté par une ligne en traits. Un message de création d'objets est représenté par une ligne en traits avec une flèche ouverte dans l'extrémité de destination.
Lost Message		*Sémantique* : ce sont des messages avec un expéditeur connu, mais la réception du message n'arrive pas. *Notation* : cet élément est définie par une arête sous forme de ligne avec un cercle noir et une flèche ouverte dans l'extrémité de destination.
Found Message		*Sémantique* : ce sont des messages trouvés par un récepteur, mais l'expéditeur de ces messages est inconnu. *Notation* : cet élément est défini par une arête sous forme de ligne avec un cercle noir dans l'extrémité de source et une flèche ouverte dans l'extrémité de destination.

Concept	Notation	Informations
General Ordering		*Sémantique* : un GeneralOrdering représente une relation entre deux *OccurrenceSpecifications*, pour décrire qu'une occurence qui doit se produire avant l'autre dans une trace valide. *Notation* : cet élément est définie par une arête sous forme de ligne en pointillée avec une flèche au milieu du lien montrant la direction de l'association.

Le diagramme de communication est représentation simplifiée du diagramme de séquence, se concentrant sur les échanges de messages entre les objets. Il est constitué des mêmes éléments que le diagramme de séquence.

Le diagramme global d'interaction permet de décrire les enchaînements possibles entre les scénarios préalablement identifiés sous forme de diagrammes de séquences. Il s'agit d'un diagramme d'activité où chaque activité peut être décrite par un diagramme de séquence.

12.2 Taux de réutilisabilité

La spécification de la syntaxe concrète de UML, nous permet d'évaluer le taux de réutilisation des notations visuelles d'UML avec les outils existants dans l'état de l'art. Les sections suivantes présentent pour chacun des outils, le nombre de notations visuelles pour chaque diagramme UML et le taux de réutilisation de ces notations.

12.2.1 MetaEdit+

TABLE 12.9: Taux de réutilisation de la notation UML avec MetaEdit+

Diagramme	Nombre de notations	Nombre de notations réutilisés	Taux de réutilisation
Classes	33	4	12,12%
Composants	15	4	26,6%
Structure Composite	16	7	43,75%
Déploiement	11	6	45,45%
Package	9	9	100%
États-transitions	16	5	31,25%
Activités	16	6	68,75%

Cas d'utilisation	17	5	29,41%
Séquences	16	3	18,75%
Communication	5	2	40%
Interaction globale	20	20	100%
Taux global de réutilisation			**46,9 %**

12.2.2 GMF

TABLE 12.10: Taux de réutilisation de la notation UML avec GMF

Diagramme	Nombre de notations	Nombre de notations réutilisés	Taux de réutilisation
Classes	33	10	30,3%
Composants	15	6	40%
Structure Composite	16	9	56,25%
Déploiement	11	6	54,54%
Package	9	9	100%
États-transitions	16	4	25%
Activités	16	8	50%
Cas d'utilisation	17	7	41,17%
Séquences	16	3	18,75%
Communication	5	3	60%
Interaction globale	20	20	100%
Taux global de réutilisation			**52,3 %**

12.2.3 Obeo Designer

TABLE 12.11: Taux de réutilisation de la notation UML avec Obeo

Diagramme	Nombre de notations	Nombre de notations réutilisés	Taux de réutilisation
Classes	33	2	6%
Composants	15	4	26,6%
Structure Composite	16	4	25%
Déploiement	11	2	18,18%
Package	9	9	100%
États-transitions	16	2	12,5%
Activités	16	2	12,5%

Cas d'utilisation	17	5	29,41%
Séquences	16	2	12,5%
Communication	5	2	40%
Interaction globale	20	20	100%
Taux global de réutilisation			**34,8 %**

12.2.4 Spray

TABLE 12.12: Taux de réutilisation de la notation UML avec Spary

Diagramme	Nombre de notations	Nombre de notations réutilisés	Taux de réutilisation
Classes	33	18	54,5%
Composants	15	4	26,67%
Structure Composite	16	9	56,25%
Déploiement	11	6	54,54%
Package	9	9	100%
États-transitions	16	7	43,7%
Activités	16	12	75%
Cas d'utilisation	17	14	82,3%
Séquences	16	5	31,2%
Communication	5	4	80%
Interaction globale	20	20	100%
Taux global de réutilisation			**64 %**

Glossaire

Abstraction	L'abstraction est le fait de considérer une qualité ou une caractéristique d'un objet, indépendamment de l'objet lui-même. L'abstraction omet les détails non significatifs pour ne se consacrer qu'à l'essentiel, 18
Artefact	Du latin (*factum*) qui veut dire effet et (*ars, artis*) qui veut dire artificel. Le terme désigne à l'origine un phénomène créé de toutes pièces par les conditions expérimentales. Dans notre contexte, il représente les éléments physiques d'un système (fichier exécutable, script, paquetage de source, etc.), 86
Aspect	Chacune des faces sous lesquelles peut être examinée un système. Selon [Filman 2000], un aspect est la modularisation de préoccupations transverses, 1
Binding	Une liaison d'objets pour les faire communiquer entre eux, 90
Composant de diagramme	Un composant est une unité logicielle offrant des services. Dans notre contexte, il représente un élément de composition réutilisable d'un diagramme, 83
Concept	Le mot Concept vient du latin (*conceptus*), qui signifie contenir entièrement ou former en soi. C'est une représentation générale et abstraite, d'une situation, d'un objet ou d'une idée conçue. Dans notre contexte, un concept est une construction abstraite qui compose un langage et qui est associée à un sens (une sémantique), 93
Couplage	Le couplage est une mesure indiquant le niveau d'interconnexion entre deux ou plusieurs éléments, 83
Diagram Definition (DD)	Un standard proposé par l'OMG pour fournir une base pour la modélisation et l'échange des notations graphiques, 79

Diagramme	Une représentation graphique d'informations. Un diagramme est une construction conforme et correcte dans un langage visuel, 36
DSL	*Domain-Specific Language*, langage spécifique à un domaine. Il peut être textuel (textual DSL) ou graphique (DSML), 62
DSML ou DSM	*Domain-Specific Modeling Language*, langage de modélisation spécifique à un domaine, 6
Editeur graphique (de diagramme)	Une interface qui permet d'intercepter et de gérer les interactions des utilisateurs pour manipuler un diagramme. Elle fournie l'outillage nécessaire pour accomplir cette tâche, 5
Encapsulation	Une action qui consiste à masquer la structure et le comportement internes et propores au fonctionnement du composant de diagramme, 83
Expressivité graphique	la possibilité de décrire et de transcrire avec fidélité les spécificités des langages visuels, 8
GPML	*Generic-Purpose Modeling Language*, langage de modélisation à usage générique (UML par exemple), 21
Grammaire visuelle	Un ensemble de règles de composition structurant une notation visuelle, 36
Granularité	La granularité d'un composant représente sa taille (par rapport aux éléments qui le composent). Un composant de petite taille est dit de granularité fine ou de petit grain. Un composant volumineux est dit de gros grain, 83
Graphe	Structure visuelle d'information définie par un couple (V, E), où V est un ensemble d'objets appelés les sommets ou noeuds du graphe (V pour l'anglais Vertex) et E une relation binaire entre deux sommets. Les éléments de E sont appelés les arêtes ou liens du graphe (E pour l'anglais Edge), 9

Héritage graphique	Le mot *Héritage* (aussi appelé dérivation) provient du fait que l'élément dérivé contient les mêmes caractéristiques/propriétés de l'élément dont il dérive. L'héritage graphique est un principe inspiré de la programmation orientée objet, permettant de créer une nouvelle notation visuelle (représentée par le composant de diagramme) à partir d'une autre composant déja existant, 83
IDM (MDE en anglais)	*Ingénierie Dirigée par le Modèles*, est une forme d'ingénierie logicielle qui offre un cadre méthodologique et technologique permettant d'unifier et de favoriser en un processus homogène l'étude des différents aspects du système. Il s'agit d'une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles, 1
Interaction	Un changement invoqué par un utilisateur dans un support d'interaction (UI), 5
Interface de composant	Dans notre contexte, une interface est utilisée comme point de liaison entre un composant de diagramme (ou un sous-composant) et la préoccupation correspondante à cette interface (domaine, interactions et styles), 83
Langage visuel	Un langage dans lequel l'information est transcrite par assemblage de notations visuelles, 38
langage visuel hybride	Un langage regroupant toutes les caractéristiques des langages visuels existants (les relations spatiales, la composition, l'adjacence, etc.), 38
Lien	Une relation entre deux ou plusieurs noeuds, 9
MDA	*Model-Driven Architecture*, est une proposition de l'OMG dont l'objectif est la conception de systèmes basée seulement sur la modélisation du domaine, indépendamment de la plateforme d'implémentation ou d'exécution, 20
Modèle	Une abstraction pertinente caractérisant un point de vue particulier d'un système, 18
MOF	*Meta-Object Facility*, est un langage de définition de métamodèles, 22

MVC
: *Modèle-Vue-Contrôleur*, est une architecture logiciel destinée à répondre aux besoins des applications interactives en séparant les problématiques (préoccupations) liées aux différents composants au sein de leur architecture respective, 51

Métamodèle
: Un modèle qui définit le langage d'expression d'un modèle. Il permet de capitaliser un domaine de connaissance, 19

Noeud
: Tout point de ramification ou/et de jonction dans un graphe qui admet plus de deux arêtes incidents, 9

Notation visuelle
: Le composant élémentaire d'un langage visuel. Une notation visuelle est pour le langage visuel, ce que le mot représente pour le langage naturel, 36

Objet
: Le mot Objet vient du latin (*objectrum*) qui signifie ce qui est placé devant. Ainsi un objet est tout ce qui se place sous notre regard et qu'on peut clairement dissocier du reste. Dans le monde UML, un objet est une instance d'une classe, 1

OCL
: *Object Constraint Language*, est un langage destiné à exprimer les contraintes au sein d'un langage de modélisation sous forme de conditions logiques, 22

OMG
: *Object Management Group*, est un consortium formé par des industriels et des académiques dans le but de promouvoir les technologies de l'objet, 6

Outils CASE
: *Computer-Aided Software Engineering*, sont des outils d'ingénierie logiciel qui permettent de concevoir un système et de générer ses artefacts, 1

Outils Meta-CASE
: Des outils permettant de construire des outils CASE, 59

Profil UML
: Un mécanisme qui permet d'étendre les métaclasses du métamodèle UML pour les adapter aux différents usages spécifiques, 23

Préoccupation
: Un point d'intérêt sur une caractéristique ou un aspect d'un système, 18

RUP — *Rational Unified Process*, est une version sous base documentaire du processus unifié. Le processus unifié est un processus de conception et d'évolution de logiciels basé sur UML, 21

Spécialisation — La spécialisation est la possibilité de redéfinir une caractéristique héritée, 83

Style — Dans notre contexte, représente l'ensemble des variables visuelles qui permettent la constrcution d'un vocabulaire visuel, 83

SVG — *Scalable Vector Graphics*, est un sous-ensemble de XML standardisé par le consortium W3C pour décrire des objets graphiques vectoriels, 60

Syntaxe abstraite — La syntaxe abstraite d'un langage de modélisation exprime, de manière structurelle, l'ensemble de ses concepts et leurs relations, 26

Syntaxe concrète — Elle fournie à l'utilisateur un ou plusieurs formalismes pour manipuler les concepts de la syntaxe abstraite et ainsi créer des instances du métamodèle, c'est-à-dire des modèles, 29

Syntaxe graphique — Une syntaxe concrète de forme graphique, 46

Syntaxe textuelle — Une syntaxe concrète de forme textuelle, 46

Sémantique — Le mot sémantique est dérivé du grec (*semantikos*)) qui veut dire *signifié*. La sémantique d'un langage est la signification des constructions de ce langage. Dans notre contexte, elle représente les constructions de domaine associés au langage visuel, 24

Sémiologie — Le mot sémiologie est composé de deux mots du grec ancien (*sema*) qui veut dire signe et (*logia*) qui veut dire étude. C'est la science des signes, 37

Transformation de modèles — Un mécanisme permettant de passer d'un espace de modélisation (niveau d'abstraction) à un autre, 30

UML — *Unified Modeling Language*, est un langage visuel destiné à la modélisation de systèmes et de processus, 6

Variable visuelle	L'unité atomique formant un symbole graphique, 36
Vocabulaire visuel	Un ensemble de symboles graphiques représentant une notation visuelle, 36
WYSIWYG	*What You See Is What You Get*, est une interface utilisateur (UI) qui permet de composer visuellement le résultat voulu et dont l'utilisateur voit directement à l'écran à quoi ressemblera le résultat final, 85
XMI	*XML Metadata Interchange*, est un format d'échange basé sur XML pour les modèles exprimés à partir d'un métamodèle MOF, 18

Table des figures

Bibliographie

[Aho 1986] Alfred V. Aho, Ravi Sethi et Jeffrey D. Ullman. Compilers : Princiles, techniques, and tools. Addison-Wesley, 1986. (Cité en page 24.)

[Amyot 2006] Daniel Amyot, Hanna Farah et Jean-François Roy. *Evaluation of Development Tools for Domain-Specific Modeling Languages*. In Reinhard Gotzhein et Rick Reed, editeurs, System Analysis and Modeling : Language Profiles, volume 4320 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin Heidelberg, 2006. (Cité en pages 2, 36, 46 et 72.)

[Auer 2003] M. Auer, T. Tschurtschenthaler et S. Biffl. *A flyweight UML modelling tool for software development in heterogeneous environments*. In Euromicro Conference, 2003. Proceedings. 29th, pages 267 – 272, sept. 2003. (Cité en page 45.)

[Baniassad 2004] Elisa Baniassad et Siobhán Clarke. *Theme : An Approach for Aspect-Oriented Analysis and Design*. In ICSE, pages 158–167, 2004. (Cité en page 18.)

[Bauer 2008] A. Bauer. *Visualisierung von Suchtreffern und Clustern in 3D-Diagrammeditoren*, 2008. (Cité en page 50.)

[Bertin 1983] Jacques Bertin. Semiology of graphics : diagrams, networks, maps. University of Wisconsin Press, Madison, Wisconsin, 1983. (Cité en pages 37, 66, 85 et 94.)

[Bézivin 2003] Jean Bézivin. *La transformation de modèles*. Ecole d'été CEA EDF INRIA, page 35. INRIA-ATLAS & Université de Nantes, 2003. (Cité en page 31.)

[Bézivin 2004] Jean Bézivin. *Sur les principes de base de l'ingénierie des modèles*. L'OBJET, vol. 10, no. 4, pages 145–157, 2004. (Cité en page 17.)

[Bézivin 2005] Jean Bézivin. *On the unification power of models*. Software and System Modeling, vol. 4, no. 2, pages 171–188, 2005. (Cité en page 17.)

[Bottoni 2000] P. Bottoni, G. Taentzer et A. Schurr. *Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation*. In Visual Languages, 2000. Proceedings. 2000 IEEE International Symposium on, pages 59–60, 2000. (Cité en page 58.)

[Bottoni 2004] P. Bottoni et A. Grau. *A Suite of Metamodels as a Basis for a Classification of Visual Languages*. In IEEE Symp. on Visual Languages and Human Centric Computing, pages 83 –90, sept. 2004. (Cité en pages 38, 41, 81, 85, 93, 135 et 136.)

[Budinsky 2003] Franck Budinsky, David Steinberg et Raymond Ellersick. Eclipse modeling framework : A developer's guide. Addison-Wesley Professional, 2003. (Cité en pages 26, 62 et 86.)

[Cariou 2011] Eric Cariou. *Cours en manipulations de modèles*, 2011. `http://web.univ-pau.fr/~ecariou/cours/idm/tp1.html`. (Cité en page 28.)

[CEA 2013] CEA. *Laboratoire d'Intégration de Systèmes et des Technologies (LIST)*, 2013. `http://www-list.cea.fr/fr/conception/`. (Cité en page 2.)

[Clark 2004] Tony Clark, Andy Evans, Paul Sammut et James Willans. *Applied Metamodelling - A foundation for Language Driven Development version 1*, 2004. (Cité en page 26.)

[Clark 2008] Tony Clark, Paul Sammut et James Willans. *Applied Metamodelling - A foundation for Language Driven Development Second Edition*, 2008. (Cité en page 26.)

[Cleveland 1984] William S. Cleveland et Robert McGill. *Graphical Perception : Theory, Experimentation, and Application to the Development of Graphical Methods*. Journal of the American Statistical Association, vol. 79, no. 387, pages 531–554, 1984. (Cité en page 38.)

[Combemale 2008] Benoit Combemale. *Approche de métamodélisation pour la simulation et la vérification de modèle*. PhD thesis, l'Institut National Polytechnique de Toulouse (INPT), 2008. (Cité en pages 18, 21, 30 et 183.)

[Costagliola 1997] Gennaro Costagliola, Andrea De Lucia, Sergio Orefice et Genoveffa Tortor. *A Parsing Methodology for the Implementation of Visual Systems*. IEEE Trans. Softw. Eng., vol. 23, no. 12, pages 777–799, Décembre 1997. (Cité en page 58.)

[Costagliola 2002] G. Costagliola, R. Francese, M. Risi, G. Scanniello et A. De Lucia. *A component-based visual environment development process*. In Proceedings of the 14th international conference on Software engineering and knowledge engineering, SEKE '02, pages 327–334, New York, NY, USA, 2002. ACM. (Cité en pages 45 et 85.)

[Costagliola 2004] Gennaro Costagliola, Vincenzo Deufemia et Giuseppe Polese. *A framework for modeling and implementing visual notations with applications to software engineering*. ACM Trans. Softw. Eng. Methodol., vol. 13, no. 4, pages 431–487, Octobre 2004. (Cité en pages 58 et 85.)

[Czarnecki 2003] Krzysztof Czarnecki et Simon Helsen. *Classification of Model Transformation Approaches*. 2003. (Cité en pages 30 et 33.)

[Davis 2003] James Davis. *GME : the generic modeling environment*. In Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '03, pages 82–83, New York, NY, USA, 2003. ACM. (Cité en page 54.)

[Debrauwer 2008] Laurent Debrauwer et Fien Van Der Heyde. Uml 2 : initiation, exemples et exercices corrigés. Éditions ENI, Nantes, France, 2008. (Cité en pages 145, 160, 163 et 167.)

[Eclipse 2006a] Eclipse. *Eclipse Community Forums : GMF (Graphical Modeling Framework)*, 2006. `http://www.eclipse.org/forums/eclipse.modeling.gmf`. (Cité en page 64.)

[Eclipse 2006b] Eclipse. *Graphical Modeling Project (GMP)*, 2006. `http://www.eclipse.org/modeling/gmp/`. (Cité en pages 45, 62 et 86.)

[Eclipse 2007] Eclipse. *Extension points*, 2007. `http://www.eclipse.org/resources/?category=Extensionpoints`. (Cité en page 140.)

[Eclipse 2008] Eclipse. *GEF3D*, 2008. http://www.eclipse.org/gef3d/. (Cité en page 50.)

[Eclipse 2012] Eclipse. *e4*, 2012. http://www.eclipse.org/e4/resources/e4-whitepaper.php. (Cité en page 86.)

[El-kouhen 2011] Amine El-kouhen, Cédric Dumoulin, Sebastien Gérard et Pierre Boulet. *Evaluation of Modeling Tools Adaptation*. Rapport technique, CNRS, 2011. http://hal.archives-ouvertes.fr/hal-00706701. (Cité en pages 2, 46, 72 et 80.)

[Estublier 2006] Jean-Marie Favre Jacky Estublier et Mireille Blay. L'ingénierie dirigée par les modèles : au-delà du mda. Hermes Science, Lavoisier édition, 2006. (Cité en page 18.)

[Fabio 2013] Filippelli Fabio, Kollosche Steffen, Bauer Michael, Gerhart Markus, Boger Marko, Thoms Karsten et Warmer Jos. *Concepts for the model-driven generation of visual editors in Eclipse*, 2013. http://spray.eclipselabs.org.codespot.com/files/SprayPaper.pdf. (Cité en pages 67 et 184.)

[Favre 2006] J.-M. Favre. Concepts fondamentaux de l'idm. de l'ancienne egypte Ã l'ingénierie des langages. 2èmes Journées sur l'Ingénierie Dirigée par les Modèles (IDM'06). Lille, France, 2006. (Cité en pages 18, 19 et 29.)

[Filman 2000] Robert E. Filman et Daniel P. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*. Rapport technique, 2000. (Cité en page 177.)

[Fondement 2008] Frédéric Fondement. *Documentation succincte sur GMF*. Rapport technique, Université de Haute Alsace, 2008. http://fondement.free.fr/lgl/courses/mde/documentation_succinte_gmf.pdf. (Cité en page 93.)

[GEF 2013] Eclipse Graphical Editing Framework GEF. *draw 2D*, 2013. http://www.eclipse.org/gef/draw2d/. (Cité en pages 47, 62 et 86.)

[Gentleware 2013] Gentleware. *Poseidon for DSLs User Guide*, 2013. http://www.gentleware.com/fileadmin/media/archives/userguides/poseidon-for-dsls/user-guide/Poseidon_for_DSLs_Documentation.html. (Cité en page 69.)

[Gérard 2011] Sébastien Gérard, Cédric Dumoulin, Patrick Tessier et Bran Selic. *Papyrus : A UML2 Tool for Domain-Specific Language Modeling*. In Model-Based Engineering of Embedded Real-Time Systems, Lecture Notes in Computer Science. 2011. (Cité en pages 6, 45 et 85.)

[Gerber 2002a] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel et Andrew Wood. *Transformation : The Missing Link of MDA*. In ICGT, pages 90–105, 2002. (Cité en page 31.)

[Gerber 2002b] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel et Andrew Wood. *Transformation : The Missing Link of MDA*. In ICGT, pages 90–105, 2002. (Cité en page 31.)

[Gilbert 1998] Demengel Gilbert et Pouget Jean-Pierre. Modèles de bézier, des b-splines et des nurbs. Éditions Ellipses, Paris, 1998. (Cité en page 97.)

[Gössler 2005] Gregor Gössler et Joseph Sifakis. *Composition for component-based modeling*. Sci. Comput. Program., vol. 55, no. 1-3, pages 161–183, Mars 2005. (Cité en pages 83 et 84.)

[Grimaldi 2007] Michel Grimaldi. *Diagrammes UML*, 2007. http://grimaldi.univ-tln.fr/LPAI/UML/DiagrammesUML.pdf. (Cité en page 145.)

[Harel 1987] David Harel. *Statecharts : A visual formalism for complex systems*. Sci. Comput. Program., vol. 8, no. 3, pages 231–274, Juin 1987. (Cité en page 160.)

[Harel 1988] David Harel. *On visual formalisms*. Commun. ACM, vol. 31, no. 5, pages 514–530, Mai 1988. (Cité en page 37.)

[Harel 2000] D. Harel et B. Rumpe. *Modeling Languages : Syntax, Semantics and All That Stuff, Part I : The Basic Stuff*. Rapport technique, Jerusalem, Israel, Israel, 2000. (Cité en page 24.)

[Harel 2004] D. Harel et B. Rumpe. *Meaningful modeling : what's the semantics of "semantics" ?* Computer, vol. 37, no. 10, pages 64–72, 2004. (Cité en page 24.)

[Hausmann 2005] Jan Hendrik Hausmann. *Dynamic META modeling : a semantics description technique for visual modeling languages*. PhD thesis, 2005. (Cité en page 24.)

[Hnetynka 2008] Petr Hnetynka et Frantisek Plasil. *The Power of MOF-Based Metamodeling of Components*. In Proceedings of the 2008 Advanced Software Engineering and Its Applications, ASEA '08, pages 67–72, Washington, DC, USA, 2008. IEEE Computer Society. (Cité en page 84.)

[Ho 2002] Wai-Ming Ho, Jean-Marc Jézéquel, François Pennaneac'h et Noël Plouzeau. *A toolkit for weaving aspect oriented UML designs*. In AOSD, pages 99–105, 2002. (Cité en page 18.)

[Hoare 1983] C. A. R. Hoare. *An axiomatic basis for computer programming*. Commun. ACM, vol. 26, no. 1, pages 53–56, Janvier 1983. (Cité en page 25.)

[Howatt 2001] J. Howatt. *A Project-Based Approach to Programming Language Evolution*, 2001. http://academic.luther.edu/~howaja01/v/lang.pdf. (Cité en page 70.)

[Hudson 2003] Randy Hudson. *Standard User Interactions in GEF*, 2003. (Cité en pages 49 et 100.)

[IBM 2011] IBM. *Rational Software Architect (RSA)*, 2011. http://www.ibm.com/developerworks/downloads/r/architect/. (Cité en pages 45 et 69.)

[ISIS 2011] Vanderbilt University ISIS. *The Generic Modeling Environment (GME)*, 2011. http://www.isis.vanderbilt.edu/Projects/gme/. (Cité en page 54.)

[ISO 2004] ISO. *Express ISO 10303-11 :2004*, 2004. Industrial automation systems and integration - Product data representation and exchange. (Cité en page 21.)

[ISO/IEC 1996] ISO/IEC. *Syntactic metalanguage - extended bnf*, 1996. (Cité en page 57.)

[Itemis 2012] Itemis. *A quick way of creating Graphiti*, 2012. http://code.google.com/a/eclipselabs.org/p/spray. (Cité en pages 45, 67, 85 et 98.)

[Jezequel 2012] Jean-Marc Jezequel, Benoit Combemale et Didier Vojtisek. Ingénierie dirigée par les modèles : des concepts à la pratique. Editions ellipses, Paris, 2012. (Cité en pages 17, 19, 24, 26, 28, 35, 44, 63, 73, 105 et 137.)

[Johnson 1970] Stephen C. Johnson. *Yacc : Yet Another Compiler-Compiler*. Rapport technique, AT&T Corporation, 1970. (Cité en page 85.)

[Jouault 2006] Frédéric Jouault. *Contribution à l'étude des langages de transformation de modèles*. PhD thesis, Université de Nantes, 2006. (Cité en page 32.)

[Juliot 2010] E. Juliot et J. Benois. *How to build Eclipse DSM without being an expert developer ?*, 2010. Obeo Designer Whitepaper. (Cité en pages 45 et 64.)

[Kelly 1996] Steven Kelly, Kalle Lyytinen et Matti Rossi. *MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment*. In Panos Constantopoulos, John Mylopoulos et Yannis Vassiliou, editeurs, Advanced Information Systems Engineering, volume 1080 of *Lecture Notes in Computer Science*, pages 1–21. Springer Berlin Heidelberg, 1996. (Cité en pages 27, 45, 55 et 56.)

[Kiczales 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier et John Irwin. *Aspect-oriented programming*. In Mehmet Aksit et Satoshi Matsuoka, editeurs, ECOOP'97 - Object-Oriented Programming, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997. (Cité en page 140.)

[Kosslyn 1980] Stephen M. Kosslyn. Image and mind. Harvard University Press, 1980. (Cité en page 36.)

[Kurtev 2008] Ivan Kurtev. *State of the Art of QVT : A Model Transformation Language Standard*. In Andy Schurr, Manfred Nagl et Albert Zundorf, editeurs, Applications of Graph Transformations with Industrial Relevance, volume 5088 of *Lecture Notes in Computer Science*, pages 377–393. Springer Berlin Heidelberg, 2008. (Cité en page 32.)

[Lahire 2007] Philippe Lahire, Brice Morin, Gilles Vanwormhoudt, Alban Gaignard, Olivier Barais et Jean-Marc Jézéquel. *Introducing Variability into Aspect-Oriented Modeling Approaches*. In MoDELS, pages 498–513, 2007. (Cité en page 18.)

[Lara 2002] Juan de Lara et Hans Vangheluwe. *AToM^3 : A Tool for Multi-formalism and Meta-modelling*. In Ralf-Detlef Kutsche et Herbert Weber, editeurs, Fundamental Approaches to Software Engineering, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2002. (Cité en page 45.)

[Larkin 1987] Jill H. Larkin et Herbert A. Simon. *Why a Diagram is (Sometimes) Worth Ten Thousand Words*. Cognitive Science, vol. 11, no. 1, pages 65 – 100, 1987. (Cité en page 36.)

[Ledeczi 2001a] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle et G. Karsai. *Composing domain-specific design environments*. Computer, vol. 34, no. 11, pages 44 –51, nov 2001. (Cité en pages 45, 52 et 54.)

[Ledeczi 2001b] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle et Peter Volgyesi. *The*

Generic Modeling Environment. In Workshop on Intelligent Signal Processing (WISP), 2001. (Cité en page 26.)

[Lee 2003] Daniel Lee. *Display a UML Diagram using Draw2D*, 2003. http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html. (Cité en page 47.)

[Lohse 1993] Gerald Lee Lohse. *A cognitive model for understanding graphical perception*. Hum.-Comput. Interact., vol. 8, no. 4, pages 353–388, Décembre 1993. (Cité en page 38.)

[Mackinlay 1986] Jock Mackinlay. *Automating the design of graphical presentations of relational information*. ACM Trans. Graph., vol. 5, no. 2, pages 110–141, Avril 1986. (Cité en page 37.)

[Mens 2006] Tom Mens et Pieter Van Gorp. *A Taxonomy of Model Transformation*. Electr. Notes Theor. Comput. Sci., vol. 152, pages 125–142, 2006. (Cité en page 31.)

[MetaPHOR 1991] Jyvaskyla University MetaPHOR. *MetaEdit+*, 1991. http://metaphor.it.jyu.fi/metapubs.html. (Cité en page 55.)

[Minas 1995] M. Minas et G. Viehstaedt. *DiaGen : a generator for diagram editors providing direct manipulation and execution of diagrams*. In Visual Languages, Proceedings., 11th IEEE International Symposium on, pages 203 –210, sep 1995. (Cité en page 45.)

[Minsky 1968] Marvin Minsky. *Matter, mind and models*. Semantic Information Processing, pages 425–432, 1968. (Cité en page 19.)

[Mishra 1997] Satish Mishra. *Visual Modeling and Unified Modeling Language (UML) : Introduction to UML*, 1997. Rational Software Corporation. (Cité en page 22.)

[Mohagheghi 2010] Parastoo Mohagheghi et Øystein Haugen. *Evaluating Domain-Specific Modelling Solutions*. In Juan Trujillo, Gillian Dobbie, Hannu Kangassalo, Sven Hartmann, Markus Kirchberg, Matti Rossi, Iris Reinhartz-Berger, Esteban Zimányi et Flavius Frasincar, editeurs, Advances in Conceptual Modeling - Applications and Challenges, volume 6413 of *Lecture Notes in Computer Science*, pages 212–221. Springer Berlin Heidelberg, 2010. (Cité en pages 2, 70 et 72.)

[Moody 2009a] D. Moody. *The Physics of Notations : Toward a Scientific Basis for Constructing Visual Notations in Software Engineering*. IEEE Transactions on Software Engineering, 2009. (Cité en pages 36, 72, 85 et 134.)

[Moody 2009b] Daniel Moody et Jos Hillegersberg. *Evaluating the Visual Syntax of UML*. In Software Language Engineering, Lecture Notes in Computer Science. 2009. (Cité en page 36.)

[MSDN 2013] MSDN. *Menus*, 2013. http://msdn.microsoft.com/en-us/library/windows/desktop/aa511502.aspx. (Cité en page 104.)

[Muller 2005] Pierre-Alain Muller, Franck Fleurey et Jean-Marc Jézéquel. *Weaving Executability into Object-Oriented Meta-languages*. In MoDELS, pages 264–278, 2005. (Cité en page 27.)

[Obeo 2010] Obeo. *Acceleo*, 2010. http://www.acceleo.org/pages/accueil/fr. (Cité en page 66.)

[OMG 1989] OMG. *Object Management Group*, 1989. http://www.omg.org/. (Cité en page 6.)

[OMG 2004] Object Management Group OMG. *Human-Usable Textual Notation (HUTN) 1.0*, 2004. http://www.omg.org/spec/HUTN/1.0/. (Cité en page 44.)

[OMG 2006a] Object Management Group OMG. *Meta Object Facility (MOF) 2.0*, 2006. http://www.omg.org/spec/MOF/2.0/. (Cité en pages 22 et 80.)

[OMG 2006b] Object Management Group OMG. *UML Diagram Interchange (DI) 1.0*, 2006. http://www.omg.org/spec/UMLDI/. (Cité en pages 19 et 63.)

[OMG 2010] Object Management Group OMG. *Object Constraint Language (OCL) 2.2*, 2010. http://www.omg.org/spec/OCL/. (Cité en page 27.)

[OMG 2011a] OMG. *Business Process Model and Notation (BPMN) 2.0*, 2011. http://www.omg.org/spec/BPMN/2.0/. (Cité en page 129.)

[OMG 2011b] OMG. *Query/View/Transformation 1.1*, 2011. http://www.omg.org/spec/QVT/1.1/. (Cité en pages 32 et 34.)

[OMG 2011c] Object Management Group OMG. *Unified Modeling Language (UML) 2.4.1 Superstructure*, 2011. http://www.omg.org/spec/UML/2.4.1/. (Cité en pages 1, 23 et 145.)

[OMG 2012] OMG. *Diagram Definition (DD) 1.0*, 2012. http://www.omg.org/spec/DD/. (Cité en pages 60, 79, 80 et 85.)

[OMG 2013] Object Management Group OMG. *XML Metadata Interchange (XMI) 2.4.1*, 2013. http://www.omg.org/spec/XMI/2.4.1/. (Cité en page 27.)

[osmoz 2013] osmoz. *Niveau d'abstraction*, 2013. http://osmoz.free.fr/leo/LeGénieObjet/1.theory/5.abstraction/index.html. (Cité en page 81.)

[Pedro 2009] Luis Pedro, Matteo Risoldi, Didier Buchs, Bruno Barroca et Vasco Amaral. *Composing Visual Syntax for Domain Specific Languages*. In Proceedings of the 13th International Conference on Human-Computer Interaction. Part II : Novel Interaction Methods and Techniques, pages 889–898, Berlin, Heidelberg, 2009. Springer-Verlag. (Cité en page 84.)

[Perini 2005] Laura Perini. *Visual Representations and Confirmation*. Philosophy of Science, vol. 72, no. 5, pages 913–926, 2005. (Cité en page 36.)

[Pohjonen 2005] Risto Pohjonen. *Metamodeling Made Easy - MetaEdit+ (Tool Demonstration)*. In Robert Gluck et Michael Lowry, editeurs, Generative Programming and Component Engineering, volume 3676 of *Lecture Notes in Computer Science*, pages 442–446. Springer Berlin Heidelberg, 2005. (Cité en page 55.)

[Raphael 2013] Raphael. *Raphael-JavaScript Library*, 2013. http://raphaeljs.com/. (Cité en page 49.)

[Rekers 1997] J. Rekers et A. Schurr. *Defining and Parsing Visual Languages with Layered Graph Grammars*. JOURNAL OF VISUAL LANGUAGES AND COMPUTING, vol. 8, pages 27–55, 1997. (Cité en page 58.)

[SAP 2011] SAP et Eclipse. *Graphiti*, 2011. http://www.eclipse.org/graphiti/. (Cité en pages 45 et 51.)

[Silingas 2008] Darius Silingas et Ruslanas Vitiutinas. *Towards UML-Intensive Framework for Model-Driven Development*. In Bertrand Meyer, JerzyR. Nawrocki et Bartosz Walter, editeurs, Balancing Agility and Formalism in Software Engineering, volume 5082 of *Lecture Notes in Computer Science*, pages 116–128. Springer Berlin Heidelberg, 2008. (Cité en page 45.)

[Soley 2000] Richard Soley. *Model Driven Architecture (MDA) draft 3.2*, 2000. Object Management Group. (Cité en page 20.)

[Sprinkle 2010] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe et Gabor Karsai. *Metamodelling : state of the art and research challenges*. In Proceedings of the 2007 International Dagstuhl conference on Model-based engineering of embedded real-time systems, MBEERTS'07, pages 57–76, Berlin, Heidelberg, 2010. Springer-Verlag. (Cité en page 70.)

[Szyperski 2002] Clemens Szyperski. Component software : Beyond object-oriented programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd édition, 2002. (Cité en page 84.)

[Tekinerdogan 2000a] B. Tekinerdogan. *Synthesis-Based Software Architecture Design*. PhD thesis, Dept. of Computer Science, University of Twente, the Netherlands, 2000. (Cité en page 72.)

[Tekinerdogan 2000b] Bedir Tekinerdogan et Mehmet Aksit. *Separation and Composition of Concerns through Synthesis-Based Design*. In OOPSLA 2000 Workshop on Advanced Separation of Concerns, 2000. (Cité en page 72.)

[Tufte 1983] Edward R. Tufte. The visual display of quantitative information. Graphics Press, 1983. (Cité en page 35.)

[Tversky 1997] Barbara Tversky. *Cognitive Principles of Graphic Displays*. In AAAI 1997 Fall Symposium on Reasoning with Diagrammatic Representations, November 1997. (Cité en page 36.)

[White 2009] Stephen A. White. *Introduction to BPMN*, 2009. http://www.bpmn.org/Documents/Introduction_to_BPMN.pdf. (Cité en page 128.)

[Wiki 2013] Wiki. *Euler diagram*, 2013. https://en.wikipedia.org/wiki/Euler_diagram/. (Cité en page 41.)

[Wikipedia 2013a] Wikipedia. *Component Object Model*, 2013. http://en.wikipedia.org/wiki/Component_Object_Model. (Cité en page 54.)

[Wikipedia 2013b] Wikipedia. *Courbe de Bézier*, 2013. http://fr.wikipedia.org/wiki/Courbe_de_Bézier. (Cité en page 60.)

[Wikipedia 2013c] Wikipedia. *Diagramme de composants*, 2013. http://fr.wikipedia.org/wiki/Diagramme_de_composants. (Cité en page 152.)

[Wikipedia 2013d] Wikipedia. *Diagramme de déploiement*, 2013. http://fr.wikipedia.org/wiki/Diagramme_de_d{é}ploiement. (Cité en page 157.)

[Wikipedia 2013e] Wikipedia. *Express*, 2013. http://fr.wikipedia.org/wiki/Express/. (Cité en page 21.)

[Wikipedia 2013f] Wikipedia. *IEEE 1471*, 2013. http://en.wikipedia.org/wiki/IEEE_1471. (Cité en page 65.)

[Wikipedia 2013g] Wikipedia. *jQuery*, 2013. http://fr.wikipedia.org/wiki/JQuery. (Cité en page 49.)

[Wikipedia 2013h] Wikipedia. *Langages de Modélisation*, 2013. http://fr.wikipedia.org/wiki/Langage_de_modélisation/. (Cité en page 21.)

[Wikipedia 2013i] Wikipedia. *Schéma Électrique*, 2013. http://fr.wikipedia.org/wiki/Schéma_électrique. (Cité en page 132.)

[Winn 1991] William Winn. *Learning from maps and diagrams*. Educational Psychology Review, vol. 3, pages 211–247, 1991. (Cité en page 38.)

[Winskel 1993] Glynn Winskel. The formal semantics of programming languages - an introduction. Foundation of computing series. MIT Press, 1993. (Cité en page 25.)

[Wouters 2013] Laurent Wouters. *Towards the Notation-Driven Development of DSMLs*. In ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2013. (Cité en page 58.)

[Zhang 2001] Da-Qian Zhang, Kang Zhang et Jiannong Cao. *A Context-sensitive Graph Grammar Formalism for the Specification of Visual Languages*. The Computer Journal, vol. 44, no. 3, pages 186–200, 2001. (Cité en page 58.)