Exploration d'ensembles de modèles

Théo Le Calvar

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ D'ANGERS
COMUE UNIVERSITÉ BRETAGNE LOIRE

École Doctorale N° 601
Mathématique et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Théo LE CALVAR

Exploration d'ensembles de modèles

Thèse présentée et soutenue à ANGERS, le 20 décembre 2019
Unité de recherche : LERIA (EA 2645)

Rapporteurs avant soutenance :

Christophe LECOUTRE, Professeur des Universités, CRIL, Université d'Artois
Jordi CABOT, Professeur, ICREA, Open University of Catalonia

Composition du jury :

Président :	Régine LALEAU, Professeur des Universités, Université Paris-Est Créteil
Rapporteurs :	Christophe LECOUTRE, Professeur des Universités, Université d'Artois
	Jordi CABOT, Professeur des Universités, Open University of Catalonia
Directeur de thèse :	Frédéric SAUBION, Professeur des Universités, Université d'Angers
Co-encadrants de thèse :	Fabien CHHEL, Docteur en Informatique, Enseignant-Chercheur, ESEO
	Frédéric JOUAULT, Docteur en Informatique, Enseignant-Chercheur, ESEO

Version définitive, compilée le 22 janvier 2020

REMERCIEMENTS

Je tiens avant tout à remercier mes encadrants Fabien CHHEL, Frédéric JOUAULT et Frédéric SAUBION qui ont su me guider durant ces trois ans.

Je remercie Jordi CABOT et Christophe LECOUTRE qui m'ont fait l'honneur d'évaluer mes travaux de thèse en acceptant d'être rapporteurs de ce manuscrit. Je remercie également Régine LALEAU, membre du jury, qui a accepté d'évaluer mon travail.

Je remercie mes collègues du LERIA de l'Université d'Angers ainsi que ceux de l'ESEO pour leur accueil et leurs conseils.

Je remercie également Grégoire CHABOT, Florian NERRIÈRE, Marion PIOU et Robin STRAEBLER, étudiants de l'ESEO, pour leur implication dans le développement de plusieurs prototypes de visualiseurs et leurs retours qui nous ont été précieux lors de la conception d'ATL^C.

Enfin, je remercie mes parents, ma famille et mes amis pour leur soutien tout au long de ces trois années.

« *Salut, et encore merci pour le poisson.* »

TABLE DES MATIÈRES

INTRODUCTION

Avec la complexité croissante des systèmes logiciels, l'**I**ngénierie **D**irigée par les **M**odèles (IDM) s'est imposée comme une méthode centrale de la conception. Un modèle est une abstraction pouvant se substituer à l'objet modélisé dans un contexte particulier. L'IDM considère ces modèles comme l'artéfact principal de la conception.

Cet artéfact sert d'abstraction facilitant la conception de systèmes logiciels. Chaque étape de la conception utilise donc un modèle adapté à ses besoins, c'est-à-dire un modèle qui n'expose que les concepts nécessaires à cette étape. Ainsi, le cycle de conception d'un système peut être vu comme une série de raffinements de modèles abstraits en modèles de plus en plus concrets, et ce jusqu'à un modèle pouvant être exécuté sur une cible (le plus souvent du code).

Cependant, l'abondance de modèles adaptés à chaque étape de la conception nécessite des outils permettant de les manipuler. Afin de faciliter au maximum l'automatisation des traitements sur ces modèles ceux-ci sont, le plus souvent, stockés et exprimés dans des formats adaptés aux machines et non à l'être humain. C'est pourquoi il est indispensable de concevoir des outils permettant de manipuler plus aisément ces modèles. Ces outils peuvent utiliser des syntaxes textuelles, pensées pour l'être humain, ou des syntaxes graphiques, également conviviales et ergonomiques. Néanmoins, la conception d'outils permettant de visualiser et manipuler ces modèles très spécifiques est une tâche complexe et coûteuse.

Il existe plusieurs méta-outils simplifiant la conception d'outils de manipulation graphique de modèles. Mais ces méta-outils sont limités et ne permettent que la création de certains types de visualisation : par exemple, la visualisation d'un modèle sous la forme d'un diagramme composé d'éléments reliés par des flèches. L'ajout de nouveaux types de visualisations est complexe et fastidieux. En effet, chaque nouveau type de diagramme requiert le développement d'algorithmes *ad hoc*.

Dans cette thèse, nous proposons une approche déclarative pour définir et manipuler des modèles, basée sur la transformation de modèles et la programmation par contraintes. Avec cette approche, la structure des diagrammes générés est spécifiée par une transformation de modèles. Très schématiquement, les règles définissant le positionnement des éléments graphiques les uns par rapport aux autres sont décrites sous la forme de contraintes. Celles-ci sont utilisées par un ou plusieurs solveurs de contraintes qui placeront ainsi les éléments graphiques.

Cette approche est construite sur la notion d'exploration d'ensembles de modèles que nous définissons à partir de la notion d'espace de modèles. Simplement, un espace de modèles M est un graphe $M = (M_\bullet, M_\Delta)$ contenant un ensemble de modèles $M_\bullet$ (nœuds) ainsi qu'un

ensemble de mises à jour, ou deltas, M_Δ permettant de modifier un modèle de l'espace en un autre modèle de l'espace (arcs). Un ensemble de modèles est un sous-graphe de M dans lequel ne sont conservés que les modèles valides. Dans le cas des diagrammes, l'espace des modèles correspondant est composé de tous les modèles conformes à un métamodèle de figures géométriques. Or, dans cet espace de modèles, de nombreux modèles ne sont pas des diagrammes valides. Pour être valide un diagramme doit respecter un ensemble de règles. Ainsi, l'ensemble des modèles valides peut être représenté par un ensemble de modèles.

Pour fonctionner, cette approche repose sur la notion de variable "pont" (*bridge variable*). Ces variables ponts permettent de garder les éléments graphiques et les variables utilisées dans les contraintes synchronisées. Ainsi, chaque changement sur le diagramme est propagé ; le ou les solveurs peuvent alors recalculer une solution prenant en compte le changement. De cette manière, les positions des éléments du diagramme sont mises à jour.

De plus, nous présentons deux langages permettant l'expression de transformations de modèles contenant des contraintes ainsi qu'un ensemble de cas d'étude illustrant les capacités de ces deux langages. Un premier langage est intégré à Xtend, un langage de programmation généraliste, et un second est basé sur **A**TLAS **T**ransformation **L**anguage (ATL), un langage de transformation dédié.

Plan de la thèse

Ce manuscrit de thèse est composé de sept chapitres.

Chapitre 1 : nous abordons le contexte global dans lequel s'inscrivent ces travaux. Nous développons plus en détail l'IDM ainsi que le problème de visualisation de modèles.

Chapitre 2 : nous étudions plus en détail les fondements de la transformation de modèles et de la programmation par contraintes. Nous présentons aussi plusieurs approches alliant des techniques de résolution de problèmes à l'IDM. Plus précisément, nous abordons des méthodes combinant programmation par contraintes et transformation de modèles.

Chapitre 3 : nous proposons une formalisation du problème d'exploration d'ensemble de modèles. Ainsi qu'un ensemble de techniques permettant de construire et d'explorer ces ensembles de modèles grâce à un solveur de contraintes.

Chapitre 4 : nous proposons une formalisation des opérations actives. Les opérations actives sont un ensemble d'opérations agissant sur des collections et permettant, par composition, la création d'expressions complexes. Ces opérations, après une phase d'initialisation, gardent leurs entrées et sorties synchronisées, c'est-à-dire qu'un changement sur une entrée sera reporté sur la ou les sorties de l'opération active. Les opérations communiquent les mises à jour au travers d'un graphe de propagation. Cependant, certaines topologies de ce graphe entraînent des erreurs lors de la propagation. C'est pourquoi, en nous basant sur cette formalisation des opérations actives et du graphe de propagation, nous développons une méthode permettant de détecter les topologies problématiques. Nous présentons également comment

cette méthode peut être appliquée dans le cas où les opérations actives sont utilisées conjointement avec des solveurs de contraintes.

Chapitre 5 : nous abordons un ensemble de spécifications pour la conception d'un langage permettant la génération et l'exploration d'ensembles de modèles. Nous présentons une structure utilisant la transformation de modèles permettant la génération de modèles cibles auxquels s'ajoutent des contraintes. Ces modèles cibles contraints, couplés à un solveur, forment un ensemble de modèles qu'il est possible d'explorer. De plus, nous présentons le métamodèle de contraintes que nous avons mis en place pour représenter les contraintes ainsi que plusieurs transformations que nous avons définies pour réécrire ces contraintes dans notre métamodèle en contraintes spécifiques à chaque solveur.

Chapitre 6 : nous détaillons deux implémentations de cette spécification. Ces deux langages sont basés sur la spécification présentée dans le chapitre précédent. La première est un **D**omain **S**pecific **L**anguage (DSL) interne à Xtend. La seconde est une extension d'ATL, un langage de transformation dédié.

Chapitre 7 : nous présentons six cas d'étude réalisés durant cette thèse, quatre avec le premier outil et deux avec le second. Ces cas d'étude illustrent les capacités des deux langages ainsi que leurs limitations. De plus, ces cas d'étude se concentrent sur la génération de visualiseurs et d'éditeurs de diagrammes. Cependant, un des cas d'étude, le générateur et visualiseur de planning, utilise les contraintes pour générer un planning valide avant de l'afficher dans un éditeur graphique. Ceci démontre ainsi la possibilité d'utiliser plusieurs solveurs de contraintes.

Enfin, nous concluons par un rappel du problème, des contributions de cette thèse, les perspectives et une liste des travaux liés à cette thèse.

CHAPITRE 1

LA PLACE DU MODÈLE DANS L'INGÉNIERIE DIRIGÉE PAR LES MODÈLES

L'**I**ngénierie **D**irigée par les **M**odèles (IDM) (ou ***M**odel-**D**riven **E**ngineering (MDE)* en anglais) s'est imposée comme une méthode clé du développement logiciel. En effet, la complexité croissante des systèmes nécessite la mise en place de méthodes adaptées. Cette méthode consiste à abstraire les différentes parties d'un système sous la forme de modèles. Dans ce chapitre, sauf mention contraire, le terme modèle fera référence à un modèle de l'IDM. Il est ainsi possible de raisonner en termes de concepts et ainsi considérer différents niveaux d'abstractions adaptés aux différentes étapes de conception. Il peut ainsi exister plusieurs modèles d'un même système, chacun adoptant un niveau d'abstraction différent et adapté aux besoins de la personne utilisant ce modèle. Par exemple, un modèle des interactions d'une application avec l'utilisateur et un modèle des communications entre l'application mobile et un serveur représentent deux abstractions d'un même système et répondent à des besoins différents.

1.1 Ingénierie dirigée par les modèles

L'IDM est apparue comme une extension de l'approche du « tout objet » introduite par la **P**rogrammation **O**rientée **O**bjet (POO) dans les années 1980. La POO prône l'utilisation de l'objet comme abstraction représentant des concepts du monde réel. L'objet sert d'abstraction encapsulant les données et traitements spécifiques à un concept, facilitant ainsi la séparation des concepts. Cependant, cette approche nécessite une étape de modélisation indispensable à la conception d'abstractions pertinentes.

Il faut donc mettre en place des méthodes facilitant la conception et l'échange de ces abstractions, ou modèles. L'IDM propose un ensemble de méthodes permettant la modélisation et la gestion de ces modèles. Actuellement, ces méthodes sont portées par l'**O**bject **M**anagement **G**roup (OMG), consortium développant des standards liés aux méthodes de l'IDM.

1.1.1 Le modèle

Au cœur de ces méthodes, on trouve la notion d'objet. L'objet est une abstraction répondant aux besoins de la personne le manipulant. Celui-ci regroupe un ensemble de faits décrivant l'objet modélisé de façon à ce que celui-ci puisse être utilisé en substitution du concept original. Par exemple, pour une pension pour animaux, un animal peut être modélisé par un ensemble de propriétés telles que son nom, le nom des propriétaires, ainsi que le régime alimentaire. Cependant, ce modèle ne serait pas suffisant pour un vétérinaire, qui aurait (par exemple) besoin d'un historique médical.

Ainsi, un bon modèle doit pouvoir remplacer l'objet modélisé pour une utilisation donnée. Il est difficile de définir précisément un *bon modèle*. En effet, un bon modèle doit pouvoir substituer le concept original et répondre pareillement dans le contexte dans lequel le modèle est utilisé. C'est pourquoi concevoir un modèle simple, mais remplaçant parfaitement le concept modélisé est complexe.

De plus, pour profiter des techniques développées dans l'IDM, les modèles doivent pouvoir être lus par des machines. Pour cela, il est nécessaire que ces modèles soient exprimés dans des langages clairement définis. Ces langages modélisant les syntaxes abstraites d'autres modèles sont appelés *métamodèles*. Un métamodèle est un modèle définissant la structure que pourront prendre les modèles qui lui seront conformes.

Par exemple, pour revenir sur l'exemple vu plus tôt, pour une pension, un chat est un animal possédant un nom, des propriétaires, et suivant un certain régime alimentaire. Par conséquent, cette description de ce qu'est un chat correspond au métamodèle de leur modèle de chat.

De même, pour être utilisé par une machine, ce métamodèle doit être spécifié dans un langage qui pourra être lu par une machine. Ainsi, ce langage qui définit les métamodèles est appelé *métamétamodèle*.

Pour représenter ces modèles et métamodèles, l'OMG normalise un ensemble d'outils. Pour cela, ils proposent un système en quatre couches. La Figure 1.1 détaille ces quatre couches en les illustrant d'exemples.

— Tout en bas, la couche `M0` contient les données ou les concepts que l'on souhaite modéliser. Par exemple, pour le chenil, Félix, un chat.

— La couche `M1` regroupe les modèles, qui sont des représentations des données de la couche `M0`. Par exemple, le modèle du chat, son nom, les informations sur son ou ses propriétaires ainsi que son régime alimentaire.

— La couche `M2` regroupe les métamodèles auxquels les modèles de `M1` se conforment. Par exemple, le modèle de chat est conforme au métamodèle **U**nified **M**odeling **L**anguage (UML) décrivant comment construire des modèles.

— Enfin, les métamodèles de la couche `M2` sont conformes à un métamétamodèle de la couche `M3`. Par exemple, pour UML, celui-ci se conforme au métamétamodèle **M**eta-**O**bject **F**acility (MOF).

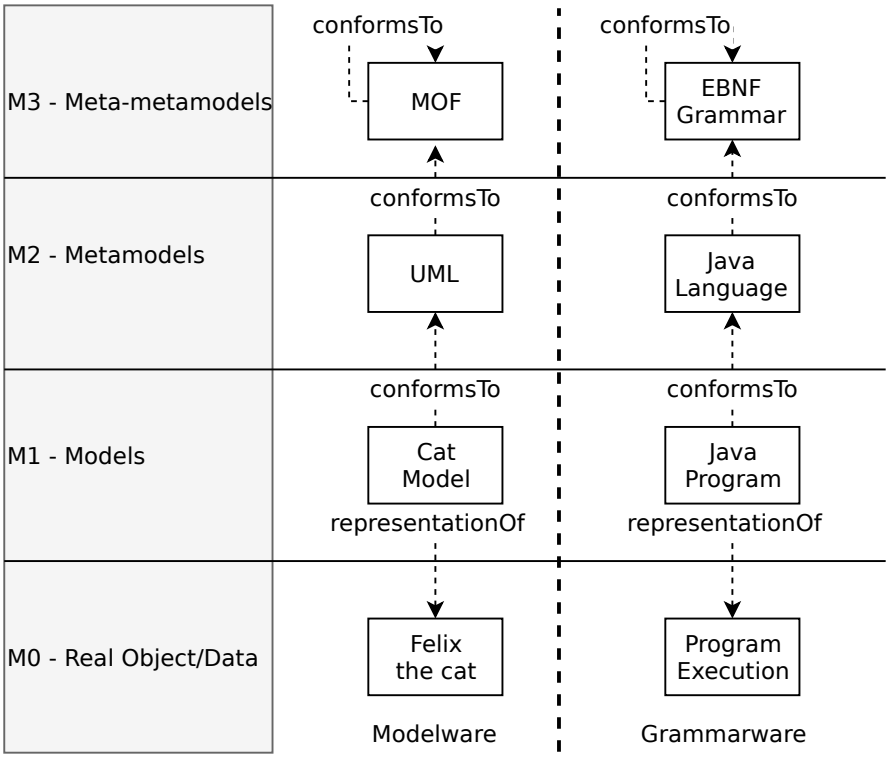


FIGURE 1.1 – Hiérarchie de modélisation à quatre niveaux

Il serait possible d'étendre cette chaîne de conformité indéfiniment, c'est pourquoi les métamétamodèles sont, le plus souvent, conformes à eux-mêmes. C'est à dire qu'il est possible de définir la syntaxe abstraite du métamétamodèle avec ce même métamétamodèle.

De nombreux concepts hors du monde de l'IDM suivent cette structure à quatre couches. Par exemple, un programme informatique est un modèle encodant l'exécution d'un programme. Ce programme est conforme à la syntaxe d'un langage de programmation. La syntaxe du langage de programmation est elle-même exprimée en termes de grammaire se décrivant elle-même. Par exemple, la grammaire **E**xtended **B**ackus-**N**aur **F**orm (EBNF) [ISO96] est une grammaire conforme à elle-même.

La Figure 1.2 montre un second exemple détaillé de cette architecture à quatre couches. Nous avons deux chats, Paco et son fils Jupiter, lesquels sont représentés par deux objets eux-mêmes décrits par un métamodèle de chat conforme au métamétamodèle Ecore. Ecore est une implémentation, portée par la fondation Eclipse, de la spécification MOF. Notre métamodèle de chat décrit un chat comme une entité possédant un nom (`name`), un sexe (`sex`), un âge (`age`) et étant l'enfant d'un autre chat (`childOf`). Cette entité chat, nommée `Cat` dans le diagramme, est plus généralement appelée métaclasse (ou *metaclass* en anglais). Ce métamodèle de chat est utilisé pour modéliser nos deux chats, Paco et son fils Jupiter. Ici, et dans le reste de ce document, on utilisera respectivement les notations du diagramme de classes et du diagramme

d'instances UML pour représenter les métaclasses (niveaux M2) et les éléments de modèles (niveau M1).

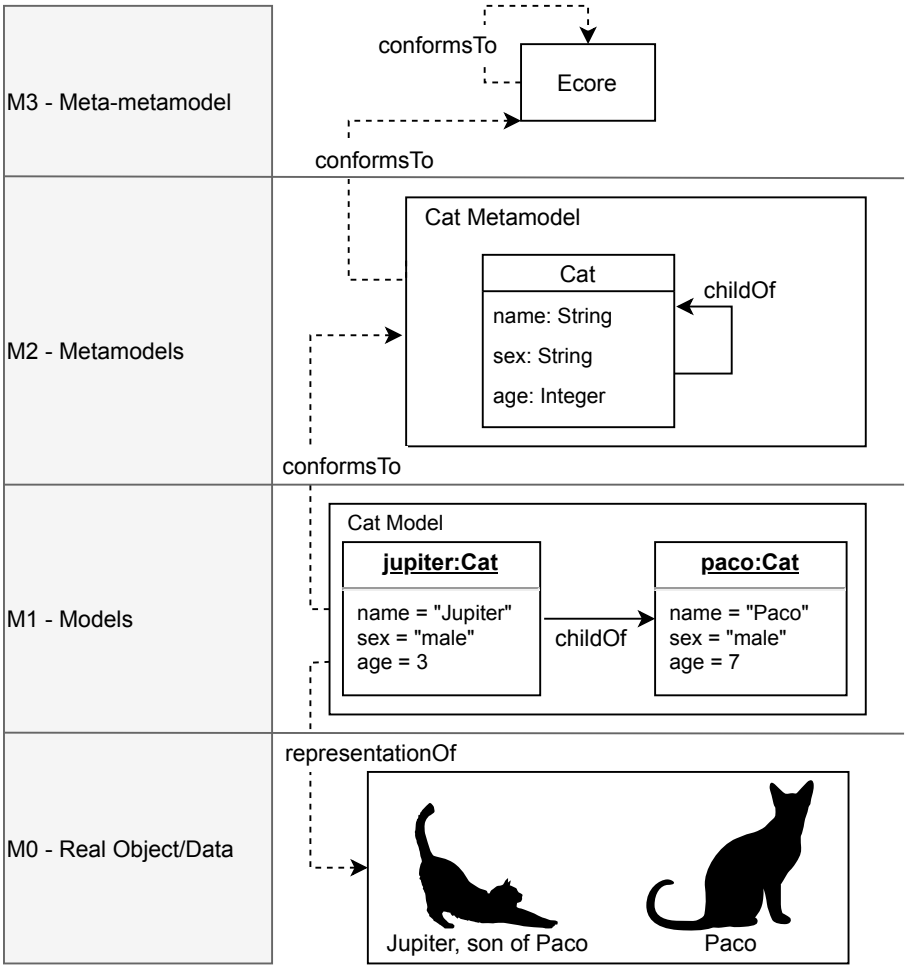


FIGURE 1.2 – Exemple de modélisation avec UML

Cette architecture en quatre couches n'est pas spécifique à l'IDM et peut être adaptée à de nombreux domaines. En fonction du métamétamodèle utilisé, il est possible de décrire de nombreux systèmes. Par exemple, dans la Figure 1.1, nous avons deux exemples de deux domaines différents, le domaine des modèles ou *modelware* et le domaine des grammaires ou *grammarware*. On nomme ces domaines des *espaces techniques* [Béz+05 ; KBA02].

1.1.2 Espaces de modèles

Nous avons vu qu'un métamodèle est comparable à la grammaire d'un langage de programmation dans le sens où celui-ci est utilisé pour décrire un ensemble de modèles corrects. Ainsi, comme il est possible de considérer l'ensemble des mots et des phrases acceptés par un lan-

gage, il est possible de considérer l'ensemble des modèles conformes à un métamodèle. Ces modèles composent une partie de ce que l'on appelle un *espace de modèles* [Dis+14 ; DXC10]. Un espace de modèles est un graphe composé d'un ensemble de modèles (les nœuds) et d'un ensemble de mises à jour ou deltas (les arcs) reliant les modèles. Un delta entre un modèle `A` et un modèle `B` représente les modifications à appliquer au modèle `A` de façon à obtenir le modèle `B`. Ainsi cet espace de modèles permet de représenter l'ensemble des modèles conformes à un métamodèle.

Cependant, comme la syntaxe d'un langage n'est pas suffisante pour construire des programmes corrects, la conformité d'un modèle n'est, souvent, pas suffisante pour qualifier un modèle de valide. C'est la sémantique d'un langage de programmation qui permet de déterminer si un programme syntaxiquement correct est exécutable. De la même façon, il faut un mécanisme pour spécifier quels modèles parmi tous ceux possibles sont valides. L'OMG propose un langage dédié, l'**O**bject **C**onstraint **L**anguage (OCL) [WK98].

Ce langage fonctionnel permet la spécification de contraintes au niveau métamodèle permettant la validation des modèles. Par exemple, pour revenir sur nos chats, il peut être intéressant de contraindre les valeurs possibles pour l'âge. Cela s'exprime en OCL par un invariant, qui est un prédicat devant toujours être vérifié. Par exemple, l'invariant suivant contraint l'âge d'un `Cat` à toujours être positif :

```
context Cat inv : self.age > 0
```

Pareillement, il est possible de définir des contraintes sur des méthodes, lesquelles doivent être vérifiées avant l'application de la méthode (précondition) ou après (post-condition). Par exemple, imaginons que la classe `Cat` ait une méthode `feed(kittenFood : Boolean)` prenant en paramètre une valeur booléenne indiquant si la nourriture donnée est adaptée aux chatons. La contrainte suivante donne une précondition vérifiant qu'un chaton reçoive de la nourriture adaptée :

```
context Cat::feed(kittenFood : boolean) pre : self.age > 1 or kittenFood
```

Si le chat est âgé de moins de 1 an alors celui-ci doit recevoir exclusivement de la nourriture pour chaton.

Le langage OCL s'est imposé comme le standard pour la définition de contraintes sur les modèles dans le contexte de l'IDM. Celui-ci permet de passer outre les limitations d'UML et ainsi pouvoir spécifier quels modèles dans l'espace de modèles sont valides.

1.1.3 Transformation de modèles

Nous avons vu que les métamodèles et modèles sont au cœur de l'approche de l'IDM. Celle-ci promeut l'utilisation de modèles tout au long de la chaîne de développement, passant de modèles abstraits en début de conception à des modèles spécialisés (par exemple, du code). Ainsi, il est possible de schématiser la chaîne de conception sous la forme d'une série de transformations. La Figure 1.3 montre une de ces chaînes.

À partir d'un modèle `Model1` très abstrait est généré un deuxième modèle `Model2`, plus spécialisé. De même, un modèle `Model3` plus spécifique est généré à partir de `Model2`. Enfin, un dernier modèle, `Code`, contenant le code spécifique à une plateforme pour le déploiement de l'application modélisée est généré à partir de `Model3`. Cette série de transformations successives permet de conserver le modèle au cœur de la conception et de ne présenter à chaque étape de conception que les concepts pertinents.

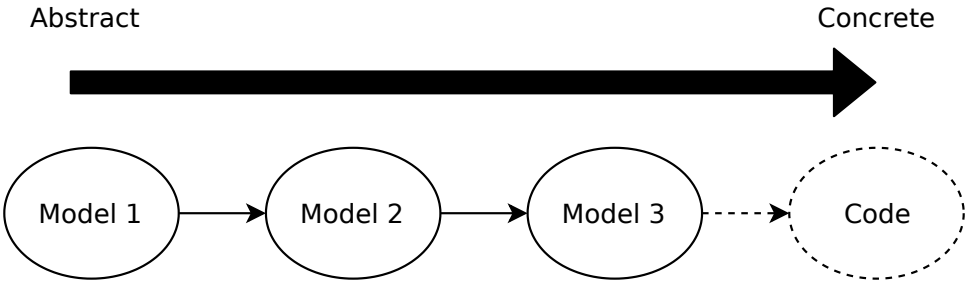


FIGURE 1.3 – Chaîne de transformations

Bien qu'exprimables dans n'importe quel langage, ces transformations utilisent principalement des outils dédiés de transformation de modèles. Ces outils tirent parti de la structure en quatre couches présentée plus tôt. Typiquement, ces outils spécifient les transformations de modèles comme un ensemble de relations, entre un métamodèle source et un métamodèle cible, qui pourront être appliquées à un modèle conforme au métamodèle source pour générer un modèle conforme au métamodèle cible. La Figure 1.4 montre la structure d'une transformation de modèles `Ta2b`, conforme au métamodèle `MMt`, transformant un modèle `Ma`, conforme au métamodèle `MMa`, en un modèle `Mb`, conforme au métamodèle `MMb`.

De cette façon, une spécification de transformation peut être utilisée pour transformer n'importe quel élément conforme au métamodèle source en un élément conforme au métamodèle cible. Il existe de nombreux outils de transformation de modèles adaptés à différentes situations. Par exemple, certains permettent l'application de transformations bidirectionnelles, c'est-à-dire qu'il est possible de générer un modèle cible à partir d'un modèle source et vice-versa. D'autres sont incrémentales, c'est à dire, qu'après le calcul initial du modèle cible, lorsque le modèle source est modifié, le modèle cible est mis à jour avec les changements nécessaires.

Malgré la diversité de ces outils et leurs capacités, le langage OCL s'est imposé comme la brique commune à ces approches. En effet, de nombreux outils utilisent des langages proches d'OCL.

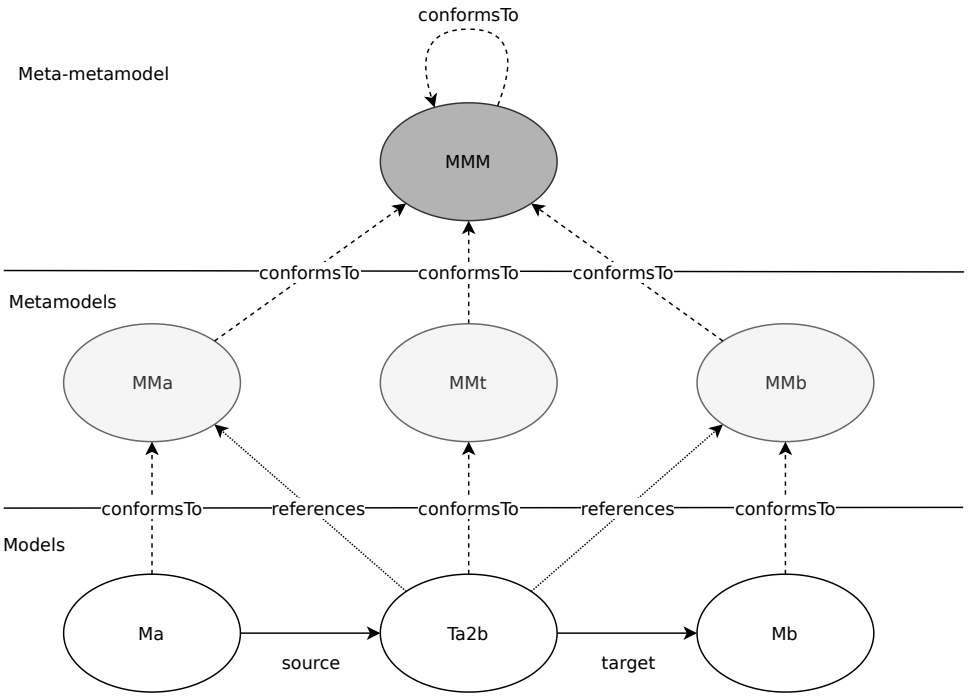


FIGURE 1.4 – Structure d'une transformation de modèles

1.2 Problème de visualisation de modèles

1.2.1 Visualisation de modèles

Durant les étapes de conception, les utilisateurs manipulent un grand nombre de modèles différents représentant les divers aspects d'un système. Les outils permettant de manipuler ces modèles sont donc incontournables dans l'IDM. Ces modèles peuvent être textuels ou graphiques.

Par exemple, la Figure 1.5 montre un diagramme de classes représentant un métamodèle de graphe, le Listing 1 montre le même métamodèle en Xcore (un langage de modélisation d'Eclipse) et le Listing 2 montre le même métamodèle en Ecore (sérialisé en **X**ML **M**etadata **I**nterchange (XMI)).

Ces trois représentations représentent le même métamodèle de graphe composé d'un package `Graph` contenant trois classes `Graph`, `Node` et `Arc`. Un `Graph` est composé de `nodes` et d'`arcs`. Un `Node` possède un `label` et une position (`x` et `y`). En plus de ces attributs, un `Node` possède des références sur les `Arc`s entrants (`incomingArcs`) et sortants (`outgoingArcs`). Enfin, un `Arc` référence deux `Node`s, une `source` et une `target`.

Parmi ces trois représentations, deux sont textuelles (Listings 1 et 2) et une graphique (Figure 1.5). Chacune de ces représentations est adaptée à une situation, par exemple, la représentation Ecore au format XMI est adaptée au traitement par une machine tandis que la

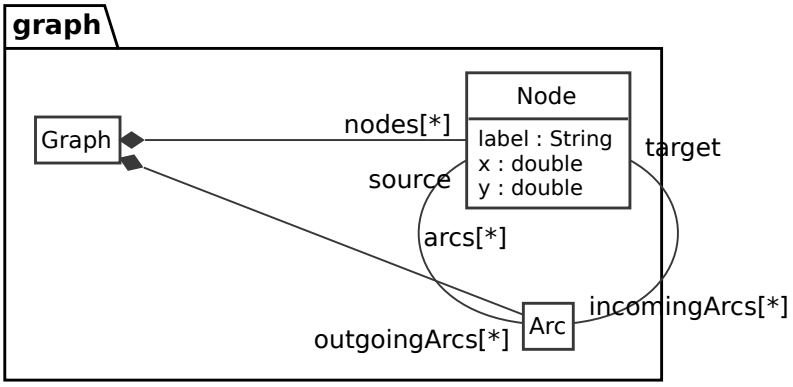


FIGURE 1.5 – Diagramme de classes

représentation sous forme d'un diagramme de classes est adaptée à l'utilisation par un être humain (Figure 1.5).

Avec la généralisation du tout modèle et le développement de métamodèles propres à chaque situation, il est parfois nécessaire de développer des outils permettant de manipuler ces modèles métiers graphiquement.

1.2.2 Méthodes classiques de conception de visualiseurs de modèles

Le développement d'outils permettant de manipuler graphiquement ces modèles métiers est complexe et coûteux. En effet, pour pouvoir manipuler ces modèles graphiquement il faut concevoir une syntaxe visuelle pour chaque métamodèle.

Une telle syntaxe visuelle spécifie la façon dont le modèle est affiché par l'outil et la signification de cet affichage. Par exemple, le diagramme de classes UML possède une syntaxe visuelle précise.

En plus de cette syntaxe visuelle, il faut créer un outil de visualisation gérant un modèle qui sera modifié par l'utilisateur. Il faut donc concevoir un ensemble d'interacteurs que l'utilisateur utilisera pour interagir avec le modèle.

Ceci doit être fait pour chaque métamodèle dont on souhaite pouvoir visualiser et modifier graphiquement les modèles s'y conformant. La construction de ces outils dédiés, souvent spécifiques à une entreprise, représente un coût important.

C'est pourquoi il existe plusieurs outils commerciaux spécialisés dans la création d'outils de modélisation spécifiques. Ces outils permettent la création d'éditeurs graphiques spécialisés pour des modèles métiers spécifiques.

Parmi ces outils, on peut présenter MetaCase [1], et plus spécifiquement MetaEdit+. MetaEdit+ Workbench est un outil propriétaire permettant la spécification de langages visuels pour

1. https://www.metacase.com/

```
1  @Ecore(nsURI="http://trame.eseo.fr/atlc/example/graph")
2  package graph
3
4  class Graph {
5      contains Node[0..*] nodes
6      contains Arc[0..*] arcs
7  }
8
9  class Node {
10     String label
11     double x
12     double y
13     refers Arc[0..*] incomingArcs opposite target
14     refers Arc[0..*] outgoingArcs opposite source
15 }
16
17 class Arc {
18     refers Node source opposite outgoingArcs
19     refers Node target opposite incomingArcs
20 }
```

Listing 1 – Modèle Xcore

des modèles spécifiques. MetaEdit+ Modeler est un outil propriétaire utilisant la spécification créée par MetaEdit+ Workbench pour permettre la visualisation et l'édition de modèles spécifiques.

Toujours dans la conception d'éditeurs graphiques de DSL, on peut aussi présenter Sirius [2]. Sirius est un projet open source basé sur Eclipse, l'**E**clipse **M**odeling **F**ramework (EMF) et le **G**raphical **M**odeling **F**ramework (GMF). Une fois la syntaxe visuelle des modèles définie, l'éditeur correspondant peut être exporté comme plug-in qui pourra facilement être déployé sur d'autres installations Eclipse.

Avec ces outils, l'approche commune consiste à définir une transformation entre les éléments du métamodèle métier et un métamodèle d'éléments graphiques proposé par l'outil. Ensuite, ce métamodèle est utilisé par l'outil pour afficher les éléments correspondants ainsi que pour gérer les interactions avec l'utilisateur. Par exemple, pour générer une syntaxe visuelle basée sur le concept de boîtes et de flèches (basiquement un affichage sous la forme de graphe) le développeur devra spécifier une transformation allant de son métamodèle au métamodèle de l'outil.

La force de cette approche est la facilité de définir des syntaxes visuelles pour des métamodèles spécifiques. Cependant, avec cette approche, il est complexe de développer des syntaxes graphiques non supportées par les outils. De plus, l'ajout de nouvelles familles de syntaxes visuelles par les outils est complexe et coûteux. Par exemple, avec Sirius, l'ajout des syntaxes visuelles basées sur le diagramme de séquence UML s'est fait par le développement de nouveaux *plug-ins* dédiés à ce type de diagrammes. Et il en serait de même pour chaque nouveau type de diagrammes.

2. https://www.eclipse.org/sirius/

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="graf"
    nsURI="http://trame.eseo.fr/atlc/example/graf" nsPrefix="">
    <eClassifiers xsi:type="ecore:EClass" name="Graph">
    <eStructuralFeatures xsi:type="ecore:EReference" name="nodes" eType="#//Node"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="arcs" eType="#//Arc"
        containment="true"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="Node">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="label"
        eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="x"
        eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EDouble"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="y"
        eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EDouble"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="outgoingArcs"
        eType="#//Arc" eOpposite="#//Arc/source"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="incomingArcs"
        eType="#//Arc" eOpposite="#//Arc/target"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="Arc">
    <eStructuralFeatures xsi:type="ecore:EReference" name="source" lowerBound="1"
        eType="#//Node" eOpposite="#//Node/outgoingArcs"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="target" lowerBound="1"
        eType="#//Node" eOpposite="#//Node/incomingArcs"/>
    </eClassifiers>
</ecore:EPackage>
```

Listing 2 – Sérialisation au format XMI d'un modèle Ecore

Ces outils facilitent la conception d'éditeurs graphiques de modèles, cependant, ceux-ci sont limités dans les types de diagrammes qu'ils supportent. L'ajout de nouveaux types de diagrammes est coûteux, car cela nécessite l'ajout ou l'adaptation du métamodèle de diagramme intermédiaire et le développement d'un nouvel algorithme de placement d'éléments et de nouveaux interacteurs.

1.2.3 L'exploration d'ensembles de modèles, une approche déclarative permettant la spécification de diagrammes

Toutes les syntaxes visuelles ne s'alignent pas forcément sur des syntaxes composées de boîtes et de flèches. C'est pourquoi nous souhaitons proposer une approche déclarative générique permettant la spécification de diagrammes.

Les diagrammes que nous considérons sont composés de figures géométriques simples comme des rectangles, des lignes, des cercles et des textes. Par conséquent, un métamodèle représentant ces figures géométriques est suffisant pour représenter les éléments géométriques composant des diagrammes. Ainsi, il est possible de considérer l'espace des modèles engendré par ce métamodèle de figures géométriques. On trouvera, parmi tous les modèles composant cet espace de modèles, l'ensemble des modèles correspondant à des diagrammes

valides. Si un diagramme est défini par un ensemble de règles de construction que celui-ci doit respecter plutôt que par un algorithme *ad hoc*, alors il est possible de réduire l'espace des modèles de façon à ce que celui-ci ne contiennent plus que des diagrammes valides. On nomme ensemble de modèles ce sous-ensemble de l'espace des modèles ne contenant que les modèles valides.

Cette structure n'est pas limitée à la représentation de diagrammes. Celle-ci peut servir à représenter n'importe quel métamodèle auquel on souhaite ajouter des règles conditionnant la validité d'un modèle. Typiquement, ce sont des métamodèles annotés avec des contraintes écrites en OCL.

L'ajout de règles de validité à un modèle n'assure pas que celui-ci soit unique. Lorsqu'il existe plus d'un modèle valide il peut être complexe de sélectionner le plus pertinent pour l'utilisateur. Par exemple, dans le cas du diagramme de classe, certains éléments peuvent être déplacés librement. Il existe donc une infinité de positions pour les éléments graphiques représentant un diagramme de classe. Par conséquent, il est nécessaire de proposer une façon d'explorer ces ensembles de modèles. Ou dans le cas des diagrammes, l'ensemble des diagrammes valides.

C'est pourquoi nous proposons une approche déclarative basée sur la transformation de modèles et la programmation par contraintes, pour permettre la définition de diagrammes. Avec notre approche, nous introduisons un *framework* complet permettant la définition de la syntaxe visuelle par l'utilisation de contraintes. Ces contraintes spécifient les règles de placement des éléments graphiques les uns par rapport aux autres. Ces contraintes sont ensuite traitées par un ou plusieurs solveurs pour calculer la position de ces éléments. La complexité de développement des algorithmes *ad hoc* de placement des éléments graphiques est alors remplacée par un ensemble de contraintes.

Cette approche permet aussi de s'abstraire du métamodèle de diagramme intermédiaire et apporte une flexibilité bien plus importante. En effet, le concepteur de l'éditeur peut concevoir n'importe quel type de diagrammes. Cependant, il est possible de mettre en place des métamodèles intermédiaires permettant de reproduire facilement des diagrammes classiques comme Sirius ou MetaCase.

CHAPITRE 2

RÉSOLUTION DE PROBLÈMES APPLIQUÉE À LA TRANSFORMATION DE MODÈLES

Dans le chapitre précédent, nous avons abordé le contexte général de l'IDM. L'utilisation de modèles spécifiques requiert le plus souvent le développement d'outils permettant de les manipuler. Le développement de ces outils est coûteux et complexe. C'est pourquoi nous proposons une approche basée sur la transformation de modèles et la programmation par contraintes pour spécifier déclarativement des éditeurs de modèles. Cependant, avant de présenter en détail l'approche il nous faut définir les notions de transformation de modèles ainsi que de programmation par contraintes. Dans la section 2.1 nous présenterons donc la transformation de modèles ainsi que plusieurs approches permettant l'exécution de ces transformations. Ensuite, dans la section 2.2 nous présenterons différentes approches de modélisation de problèmes de contraintes ainsi que des méthodes de résolution classiques. Enfin, dans la section 2.3 nous présenterons différentes approches alliant la programmation par contraintes à l'IDM et plus spécifiquement à la transformation de modèles.

2.1 Transformation de modèles

Nous avons vu que dans l'IDM le modèle est considéré comme l'artéfact principal. Celui-ci sert de source pour la génération automatique de tout ou partie du système modélisé. Les techniques de transformation de modèles assurent cette génération.

Classiquement, la transformation suit le schéma montré en Figure 2.1. Une transformation `Ta2b` peut être vue comme un programme générant un modèle `Mb` conforme à un métamodèle `MMb` à partir d'un modèle `Ma` conforme à un métamodèle `MMa`. La transformation `Ta2b` spécifie donc un ensemble de relations entre les métamodèles source(s) (ici `MMa`) et cible(s) (ici `MMb`). Lorsque les métamodèles `MMa` et `MMb` font référence à deux métamodèles différents, on parle de transformation exogène. Par exemple, la transformation d'un fichier XML en fichier JSON. Autrement, lorsque `MMa` et `MMb` font référence au même métamodèle on qualifie les transformations d'endogènes. Par exemple, la transformation d'un modèle UML en un autre modèle UML.

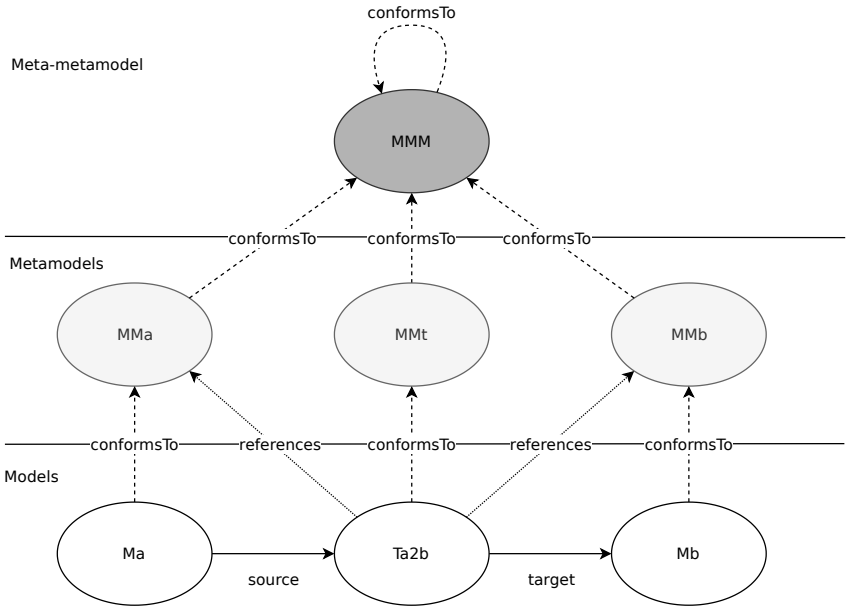


FIGURE 2.1 – Rappel de la structure d'une transformation de modèles

De plus, lors d'une transformation endogène, il se peut que les modèles `Ma` et `Mb` soient identiques. Il n'y a donc pas de création de modèle cible, mais mise à jour du modèle source. Ces transformations sont qualifiées de transformations « en place », « de mise à jour » ou « en mode *refining* » (pour ATL).

Triple Graph Grammars

Il existe de nombreuses techniques de transformation utilisant des approches différentes. On peut citer les approches basées sur la réécriture de graphe, tel que les **T**riple **G**raph **G**rammars (TGGs) [Kön05 ; Sch95].

De nombreux outils sont basés sur cette approche, on peut notamment citer MoTe [1], TGG Interpreter ou eMoflon [2]. Pour plus de détails sur les différents outils de transformation de graphes ainsi que leurs capacités peuvent être trouvés dans [Hil+13 ; Leb+14].

Les modèles étant des graphes typés il est facile d'exprimer des transformations de modèles comme des réécritures de graphes. Trois graphes sont mis en relations, le graphe "source" ou "gauche" (*LG* ou *Left Graph*), le graphe "cible" ou "droit" (*RG* ou *Right Graph*) et le graphe de "correspondance" (*CG* ou *Correspondence Graph*). Les graphes LG et RG font références aux modèles source et cible de la transformation, tandis que le CG correspond à la traçabilité permettant de retracer les correspondances entre les éléments du graphe source et leurs équivalents dans le graphe cible.

1. `http://www.mdelab.de/mote/`
2. `https://emoflon.org/`

Les transformations basées sur les TGGs sont composées d'un ensemble de règles. Ces règles permettent de faire évoluer les trois graphes en assurant qu'ils soient toujours cohérents les uns par rapport aux autres. Ces règles sont composées de deux parties. La première partie, le contexte ou préconditionn, correspond à l'ensemble des éléments qui doivent être présents pour pouvoir appliquer la règle (*application condition* ou AC). Certains moteurs d'exécution proposent, en plus des éléments qui doivent être présents dans le contexte, d'ajouter des éléments qui ne doivent pas être présents (*negative application condition* ou NAC). La seconde partie, ou post-condition est constituée des éléments créés, c'est à dire la partie des graphes qui est générée par l'application de la règle.

Ensuite, cet ensemble de règles de construction permet de générer le graphe de correspondance ainsi que le graphe de droite (respectivement gauche) à partir du graphe de gauche (respectivement droite). L'exécution d'une transformation consiste alors à trouver un ordre d'application des règles tel que l'application de ces règles génère le graphe de droite (respectivement gauche) en partant d'un modèle vide. Ainsi il est possible de construire en même temps les graphes de correspondance et de gauche (respectivement droite).

Viatra

Viatra [VB07] est une autre approche combinant les techniques de réécriture de graphes et de machine à états abstraits (ASM). Les techniques de réécriture de graphes servent notamment à la détection de motifs. Ces motifs (positifs ou négatifs) sont similaires aux contextes présentés dans les TGGs, ce sont des sous-graphes devant être présents (ou absents dans le cas de motifs négatifs). Ces motifs sont utilisés dans les différentes parties des règles Viatra pour extraire les parties des modèles d'entrée ou de sortie impliqués dans les règles.

La définition des règles Viatra se fait en trois parties. La première, la précondition, est constituée d'un ensemble d'appels à des motifs (positifs ou négatifs) devant être trouvés pour pouvoir appliquer la règle. La seconde, la post-condition, est elle aussi constituée de motifs qui devront être présents après l'application de la règle. La partie action contient du code pouvant être exécuté par l'ASM dans le contexte de la règle après que la post-condition ait été vérifiée. Enfin, l'application des différentes règles se fait grâce à du code ASM appelant les différentes règles de transformation.

Viatra utilise un algorithme basé sur l'algorithme Rete [For89]. L'algorithme Rete est un algorithme adapté à la détection de nombreuses occurrences de motifs sur un nombre important d'éléments. Les différents motifs à détecter sont compilés en un réseau qui sera utilisé pour détecter leurs occurrences motifs dans le modèle source.

ATLAS Transformation Language

ATL [Jou+08 ; JK06] est un autre langage de transformation hybride basé sur le concept de règles. Cependant, contrairement aux TGGs et à Viatra, les règles ATL ne se basent pas

sur les motifs de graphes pour l'activation et les effets d'une règle. Les règles ATL déclarent un ensemble d'éléments d'entrée ainsi que des gardes optionnelles sur ces éléments d'entrée. Ce mécanisme permet la sélection fine des éléments d'entrée comme le permettent les AC ou NAC des TGGs et de Viatra. Les éléments de sortie sont déclarés dans une seconde section de la règle. C'est à cet endroit que sont déclarées les relations (ou *bindings*). Ces *bindings* sont fortement inspirés des expressions OCL. Les *bindings* permettent de calculer les valeurs des propriétés du modèle cible à partir des propriétés du modèle source. Sauf cas particulier, le modèle source n'est accessible qu'en lecture seule et le modèle cible n'est accessible qu'en écriture seule.

En ATL il existe deux grands types de règles. Les règles standard sont automatiquement exécutées pour toutes les combinaisons d'entrée présentes dans le modèle source. Les règles paresseuses (*lazy*) ne sont exécutées que sur demande d'une autre règle. Il existe deux types de règles paresseuses, les règles avec cache (*unique*) ou sans. Le cache d'une règle paresseuse permet de retourner le même élément de sortie si la même règle est appelée avec les mêmes éléments d'entrée plusieurs fois.

En plus de ce système de règles déclaratives, ATL permet l'exécution de blocs de code impératif. Ces blocs impératifs facilitent la spécification de transformations difficilement exprimables en ATL déclaratif pur.

L'exécution d'une transformation se fait par l'appel optionnel à une règle spéciale, le point d'entrée (ou *entry point*). L'algorithme cherche ensuite les règles applicables aux éléments d'entrée fournis. Pour cela il compare les éléments d'entrée des règles ainsi que les gardes éventuelles aux éléments du modèle et enregistre les combinaisons valides. Pour chacune de ces combinaisons, l'algorithme crée les liens de traçabilité, servant à associer à chaque élément de sortie le ou les éléments d'entrée correspondants. Le concept de lien de traçabilité est très proche de la notion de graphe de correspondance des TGGs. Ensuite, l'algorithme crée les éléments de sortie correspondant et exécute les *bindings* pour calculer la valeur de leurs propriétés. On notera que l'ordre d'application des règles n'est pas garanti.

Yet Another Model Transformation Language

Toujours basée sur le concept de règles, **Y**et **A**nother **M**odel **T**ransformation **L**anguage (YAMTL) [Bor18] est une approche récente et performante basée sur Xtend. Xtend[3] est un langage de programmation généraliste compilé vers du Java qui permet notamment la redéfinition d'opérateurs que Java ne permet pas. Inspiré des règles ATL, les règles YAMTL sont composées de deux mêmes parties. Une première déclarant une liste de modèles d'entrée ainsi que des filtres facultatifs. Une seconde partie contient les modèles de sortie ainsi que les expressions servant à calculer les valeurs des propriétés des éléments de ces modèles. La principale différence avec ATL est la syntaxe utilisée. YAMTL est limité sur ce point par les

3. `https://www.eclipse.org/xtend/`

contraintes imposées par Xtend. En effet, Xtend offre une certaine liberté grâce à des mécanismes tels que la redéfinition d'opérateurs ou les méthodes d'extensions, mais ne permet pas d'introduire de nouveaux éléments de syntaxe.

L'exécution des règles suit un ordre similaire à celui d'ATL. La transformation est d'abord compilée, puis un algorithme cherche et sauvegarde l'ensemble des règles pouvant être appliquées ainsi que leurs éléments sources. Ensuite, les règles sont exécutées les unes après les autres créant les éléments du modèle de sortie.

Opérations Actives

Les opérations actives [Bea+10] (*Active Operations*) sont un ensemble d'opérations sur les collections permettant la création, par composition, d'expressions incrémentales. Chaque opération (par exemple, `collect`, `select` ou `zip`) est équipée d'algorithmes de propagation assurant la synchronisation entre ses éléments d'entrée et de sortie. Ces algorithmes de propagation calculent les modifications à appliquer à la collection de sortie à partir d'une modification sur la collection d'entrée. Chaque opération écoute sa ou ses sources. Ainsi lorsqu'une opération calcule une mise à jour, elle notifie du changement les opérations l'écoutant. Cette approche permet de limiter les recalculs à ce qui est modifié.

Active **O**peration **F**ramework (AOF) [JB16] est l'implémentation la plus récente des opérations actives. AOF regroupe ces opérations dans une bibliothèque Java. Cependant, contrairement aux approches précédentes, AOF seul ne permet pas l'écriture de règles. AOF ne supporte que l'écriture et le calcul d'expressions incrémentales. Pour palier cette limitation plusieurs langages ont été développés, un langage basé sur Xtend est présenté dans [JB16]. Plus récemment, un nouveau compilateur, ATOL [Le +19c], permet la compilation d'un sous-ensemble d'ATL vers AOF. Par exemple, ATOL ne supporte que les règles *unique lazy* d'ATL, règles devant être appelées explicitement dans les règles de transformation.

Le calcul d'expression incrémentale d'AOF se passe en trois temps :

1. L'enregistrement des opérations : chaque opération s'enregistre auprès de ses sources, s'assurant d'être notifiée lors des mises à jour futures.
2. Le calcul initial : chaque opération calcule les valeurs initiales de sa collection de sortie à partir des valeurs présentes dans sa ou ses collections d'entrée.
3. La synchronisation : lorsqu'un changement est détecté sur une source, les opérations l'écoutant sont notifiées du changement. Elles calculent la ou les modifications à appliquer à leur collection de sortie pour maintenir la cohérence. Ensuite elles notifient les opérations écoutant leur collection de sortie du changement. Ce processus se poursuit jusqu'à ce que toutes les opérations écoutant des valeurs modifiées aient été notifiées et que les nouvelles valeurs aient été calculées.

Il existe trop d'approches différentes pour pouvoir toutes les recenser ici, [Kah+18] propose une classification récente de ces approches.

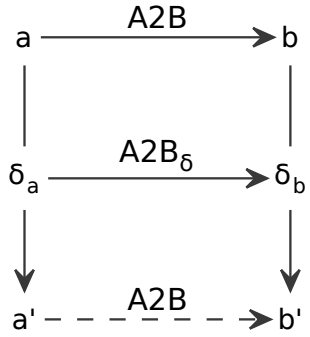


FIGURE 2.2 – Schéma d'une transformation incrémentale

2.1.1 Transformation incrémentale de modèles

Les transformations classiques traitent l'intégralité du modèle source pour créer un modèle cible conforme à la spécification de transformation. Cette opération peut être coûteuse lorsque le modèle source est composé d'un nombre important d'éléments, ou lorsque la transformation requiert des opérations coûteuses en temps de calcul. Ce coût de calcul n'est pas problématique dans le cas de modèles changeant peu. Cependant, si de nombreuses modifications sont appliquées aux éléments du modèle source, alors il peut être intéressant d'utiliser une approche de transformation incrémentale.

L'idée au cœur de la transformation de modèles incrémentale est de n'appliquer la transformation qu'aux éléments sources ayant été modifiés depuis la dernière exécution. Cela réduit ainsi le temps nécessaire à l'application de la transformation au modèle mis à jour. De plus, mettre à jour le modèle cible lorsque des modifications sont détectées sur le modèle source permet de conserver l'identité des éléments du modèle cible.

C'est notamment le cas lorsque le modèle cible est un modèle graphique présenté à l'utilisateur. Dans ce cas, il est important que les modifications apportées au modèle source soient appliquées rapidement au modèle cible. De plus, ici, il est préférable de mettre à jour les éléments visuels présentés à l'utilisateur plutôt que de devoir les remplacer complètement à chaque changement.

La Figure 2.2 reprend l'idée de la transformation incrémentale. En haut à gauche, le modèle source `a` est utilisé par la transformation `A2B` pour générer le modèle `b`. Ensuite, une modification δ_a est appliquée à `a` qui devient `a'`. Une transformation classique repartirait de `a'` pour calculer `b'`. Une transformation incrémentale utilisera δ_a pour calculer δ_b, l'ensemble de modifications à appliquer à `b` pour qu'il devienne `b'`. Cela évite ainsi la transformation `A2B`, en pointillé, généralement plus coûteuse que `A2B`$_\delta$ qui ne transforme que la mise à jour.

Plusieurs approches parmi celles présentées précédemment supportent l'exécution de transformations incrémentales. Par exemple, on peut citer Viatra à partir de sa version 3 [Var+16], des approches basées sur les TGGs [GW06 ; Leb+14], plusieurs extensions d'ATL [JT10 ; Le

+19c ; MTD17], YAMTL [Bor18] ou les opérations actives [Bea+10]. Dans [Kus+13], les auteurs présentent en détail de nombreuses approches de transformations incrémentales.

2.1.2 Transformation de modèles bidirectionnelle

Lors d'une transformation de modèles, le modèle cible est calculé à partir du modèle source. Idéalement un seul modèle est modifié (la source), ainsi recalculer une nouvelle cible (ou calculer un ensemble de modifications à appliquer) est possible à l'aide des approches classiques ou incrémentales présentées précédemment.

Un exemple classique de transformation bidirectionnelle est la génération de code à partir d'un modèle. Il n'est pas rare que le code généré doive être manuellement modifié pour ajouter des fonctionnalités manquantes ou optimiser le code. Cependant, si le modèle ayant servi à générer le code est modifié il est nécessaire de régénérer le code. Une approche classique ne prendrait pas en compte les changements sur le code et les écraserait avec la nouvelle version. Ce comportement n'est pas satisfaisant.

Un autre exemple est le problème de la mise à jour des vues dans une base de données. Une vue dans une base de données est une table virtuelle correspondant au résultat d'une requête. Ces vues sont utiles pour contrôler plus finement l'accès aux données. La plupart des systèmes de gestion de base de données (SGBDs) ne permettent qu'un accès en lecture aux vues. Certains SGBD permettent d'insérer, supprimer ou modifier des enregistrements de la vue, mais dans des contextes très limités.

Généralement, lorsque deux modèles peuvent être modifiés il est intéressant d'être capable de calculer des modifications à appliquer à la source lorsque la cible est modifiée. Dans ces cas, la transformation de modèles peut être vue comme une relation de cohérence entre deux modèles existants. Cette relation assure la cohérence entre les deux modèles et répare les modèles lorsque l'un des deux est modifié. On peut définir cette relation comme deux fonctions de réparation prenant un modèle source $s \in S$ et un modèle cible $c \in C$:

$$\overrightarrow{R} : S \times C \to C$$

$$\overleftarrow{R} : S \times C \to S$$

$\overrightarrow{R}$ correspond à la transformation avant (ou *forward*), et $\overleftarrow{R}$ correspond à la transformation inverse (ou *backward*). Soit $s' \in S$ une mise à jour de s, alors $\overrightarrow{R}$ peut être utilisée pour calculer $c' \in C$, $c' = \overrightarrow{R}(s', c)$. De même, $\overleftarrow{R}$ peut être utilisée pour calculer s' à partir de s et c', $s' = \overleftarrow{R}(s, c')$.

Il est important de s'assurer que $\overleftarrow{R}$ et $\overrightarrow{R}$ soient cohérentes l'une avec l'autre. Une façon de s'assurer que ces deux fonctions soient cohérentes est de les combiner. C'est le cas par exemple de la syntaxe graphique des TGGs, dans ce formalisme, si un motif est détecté sur le LG (respectivement RG) alors les motifs du CG et RG (respectivement LG) seront ajoutés

aux graphes existants. C'est aussi le cas des approches à base de lentilles (*lenses*) [HPW11] telles que [Boh+08]. Cependant, il est parfois plus simple et moins restrictif de définir séparément les deux fonctions $\overleftarrow{R}$ et $\overrightarrow{R}$. [Ste08] et [Hid+15] présentent un apercu des techniques et problématiques de la transformation bidirectionnelle de modèles.

2.2 Programmation par contraintes

La programmation par contraintes est un paradigme général de résolution de problèmes combinatoires (satisfaction de contraintes) qui intègre des langages de modélisation ainsi que des solveurs génériques capable de traiter des modèles et des domaines différents (nombres entiers, nombres réels, booléens). La résolution d'un problème de satisfaction de contraintes consiste à affecter des valeurs à un ensemble de variables de décision tout en respectant un ensemble de contraintes. Ces contraintes sont des relations entre ces variables, exprimées sur des domaines de calcul.

La résolution de contraintes se fonde sur un ensemble de techniques visant à affecter à un ensemble de variables des valeurs le tout en respectant un ensemble de contraintes.

La programmation par contraintes propose une syntaxe ou des langages de modélisation, mais il existe de nombreuses manières d'exprimer ou de formaliser des problèmes de décision ou d'optimisation.

2.2.1 Modélisation des problèmes de contraintes

Un problème de satisfaction de contraintes (en anglais **C**onstraint **S**atisfaction **P**roblem (CSP)) est la plus simple de ces variantes. Celui-ci est composé d'un ensemble de variables auxquelles il faut assigner une valeur de façon à respecter un ensemble de contraintes. Les notations et définitions suivantes sont reprises de [Lec09].

Définition 2.2.1 (Variable)**.** *Une variable est le composant de base des contraintes. Ce sont des objets possédant un nom et prenant une valeur parmi un domaine. Par exemple, une variable x doit prendre une valeur parmi un ensemble de valeurs possibles (son domaine). On note le domaine associé à la variable x $dom(x)$.*

On dit qu'une qu'une valeur a est valide pour une variable x si et seulement si $a \in dom(x)$.

Avant de définir la notion de contrainte, il faut définir la notion de tuple, de produit cartésien et de relation.

Définition 2.2.2 (Tuple)**.** *Un tuple τ est une séquence de valeurs, $v_1, \ldots, v_n$, noté $(v_1, \ldots, v_n)$. On nomme n-uplet un tuple de n valeurs. On note $\tau[i]$ l'élément à l'indice $1 \leq i \leq n$ d'un n-uplet τ.*

Un produit cartésien est composé de l'ensemble des tuples pouvant être construits à partir d'une séquence d'ensembles.

Définition 2.2.3 (Produit cartésien)**.** *Soient $D_1, D_2, \ldots, D_n$ une séquence de n ensembles. Le produit cartésien $D_1 \times D_2 \times \cdots \times D_n$, aussi noté $\prod_{i=1}^{n} D_i$, est l'ensemble $\{(v_1, v_2, \ldots, v_n) | v_1 \in D_1, v_2 \in D_2, \ldots, v_n \in D_N\}$ où chaque élément de l'ensemble est un n-uplet.*

Définition 2.2.4 (Relation)**.** *Soient $D_1, D_2, \ldots, D_n$ une séquence de n ensembles. Une relation R définie sur ces ensembles est un sous-ensemble du produit cartésien de ces ensembles, donc $R \subseteq \prod_{i=1}^{n} D_i$.*

Une contrainte est définie à partir des concepts de variables et de relations.

Définition 2.2.5 (Contrainte)**.** *Une contrainte est une restriction des combinaisons de valeurs que peuvent prendre les variables affectées par la contrainte. Une contrainte c est définie sur un ensemble de variables, constituant la portée de la contrainte (ou* scope*) et noté $scp(c)$. Une contrainte c est définie par une relation, notée $rel(c)$, correspondant aux tuples de valeurs pouvant être prises par les variables dans sa portée. Nous avons donc, $rel(c) \subseteq \prod_{x \in scp(c)} dom(x)$.*

Les contraintes peuvent être définies en extension ou en intension. Une contrainte c est définie en extension si la relation $rel(c)$ est définie explicitement, c'est dire en listant les tuples acceptés par la relation (ou en listant les tuples rejetés).

Une contrainte c est définie en intension si la relation $rel(c)$ est définie par un prédicat $pred : \prod_{x \in scp(c)} dom(x) \rightarrow \{0, 1\}$. Où 1 indique que le tuple est accepté et 0 qu'il est rejeté. Par exemple, la contrainte $x \neq y$ est définie en intension.

À partir des notions de variables et de contraintes, il est possible de construire un CSP.

Définition 2.2.6 (Constraint Satisfaction Problem)**.** *Un CSP P est composé d'un ensemble fini de contraintes agissant sur un ensemble fini de variables. On note $vars(P)$ l'ensemble des variables et $cons(P)$ l'ensemble des contraintes. Nous avons donc $\forall c \in cons(P), scp(c) \subseteq vars(P)$.*

Par exemple, soit un CSP p avec :

— $vars(P) = \{x, y, z\}$, $cons(P) = \{c_1, c_2\}$,

— $c_1 : x < y$,

— $c_2 : x + y > z$,

— $dom(x) = dom(y) = \{0, 1, 2\}$, $dom(z) = \{2, 3\}$.

Définition 2.2.7 (Instanciation)**.** *Une instanciation I d'un ensemble (potentiellement vide) de variables $X = \{x_1, \ldots, x_n\}$ est un ensemble $\{(x_1, a_1), \ldots, (x_n, a_n)\}$ où $a_i \in dom(x_i)$ pour tout $1 \leq i \leq n$. La valeur a_i d'une variable x_i est notée $I[x_i]$. L'ensemble des variables couvertes par une instanciation est noté $vars(I)$.*

Une instanciation I est complète pour un CSP P si et seulement si $vars(P) = vars(I)$. Autrement elle est dite incomplète. On nomme *espace de recherche* de P l'ensemble des instanciations possibles pour P, il est noté $esp(P)$.

Définition 2.2.8 (Satisfaction d'une contrainte)**.** *On dit qu'une contrainte c est satisfaite par une instanciation I si et seulement si l'instanciation couvre le* scope *de la contrainte $scp(c) \subseteq vars(I)$ et que le tuple $(a_1, \ldots, a_n)$ tel que $I[scp(c)] = \{(x_1, a_1), \ldots, (x_n, a_n)\}$ est accepté par c.*

À partir de la notion de contrainte et d'instanciation, il est possible de définir une solution à un CSP.

Définition 2.2.9 (Solution)**.** *Une solution à un CSP P est une instanciation complète sur P satisfaisant toutes les contraintes de P. On note $sol(P)$ l'ensemble des solutions de P.*

Par exemple, il existe trois solutions au CSP p défini plus tôt $sol(p) = \{I_1, I_2, I_3\}$:

— $I_1 = \{(x : 0), (y : 1), (z : 2)\}$,

— $I_2 = \{(x : 0), (y : 1), (z : 3)\}$,

— $I_3 = \{(x : 0), (y : 2), (z : 3)\}$.

Cette modélisation des problèmes est la plus simple, mais elle permet de représenter uniquement des problèmes de satisfaction, en effet, les solutions ne peuvent être que correctes ou non. Dans de nombreux cas, la satisfaction n'est pas suffisante et il est nécessaire de minimiser (ou maximiser) une fonction objectif. On nomme ce type de problème les **C**onstraint **O**ptimization **P**roblem (COP). La fonction objectif est aussi appelée fonction de coût (ou *cost function*). Sauf mention contraire, nous considérerons tous les COP suivants comme des problèmes de minimisation. Tout problème de maximisation peut naturellement être transformé en un problème de minimisation.

Définition 2.2.10 (Constraint Optimization Problem)**.** *Un COP P est un CSP auquel on a ajouté une fonction objectif noté $obj(P)$.*

Définition 2.2.11 (Fonction objectif)**.** *Une fonction de coût d'un COP P est une fonction $f : esp(P) \to \mathbb{R}$ associant un score à chaque instanciation possible de P.*

Par exemple, on étend le CSP p présenté plus tôt en un COP p_o en y ajoutant la fonction d'évaluation f suivante :

$$f(I) = \sum_{x \in vars(I)} I[x]$$

Les solutions à ce COP p_o ne diffèrent pas de celles du CSP p, cependant on leur associe un coût calculé par la fonction d'évaluation, $f(I_1) = 3$, $f(I_2) = 4$ et $f(I_3) = 5$.

Définition 2.2.12 (Solution optimale)**.** *Une solution optimale d'un COP P_o est une solution $s \in sol(P_o)$ telle que $\forall s' \in sol(P_o), f(s) \leq f(s')$ pour un problème de minimisation et $\forall s' \in sol(P_o), f(s) \geq f(s')$ pour un problème de maximisation.*

Dans notre exemple, c'est donc s_1 qui est l'unique solution optimale à ce COP p.

Les contraintes hiérarchiques

Introduites par [MBC93 ; WB93], les contraintes hiérarchiques (en anglais **H**ierarchical **C**onstraint **S**olving **P**roblem (HCSP)) sont une autre variation des COP. Celles-ci reposent sur l'idée de classifier les différentes contraintes d'un problème en fonction de leur importance. À chaque contrainte facultative est associé un poids (ou une force) correspondant à l'importance de la contrainte ; plus le poids est fort plus la contrainte est importante. Les contraintes nécessaires n'ont pas de poids.

Définition 2.2.13 (Hiérarchie de contraintes)**.** *Les poids associés aux contraintes permettent de construire une hiérarchie de contraintes $H = (H_0, H_1, \ldots, H_n)$ avec H_0 l'ensemble des contraintes devant être satisfaites (ou nécessaires), H_1 l'ensemble des contraintes ayant le poids le plus fort et ainsi de suite jusqu'à H_n contenant les contraintes ayant le poids le plus faible. L'ensemble des contraintes contenues dans la hiérarchie H_i, pour $0 \leq i \leq n$, est noté $H[i]$. On note $w_{H_i} \in \mathbb{R}$ le poids associé à chaque niveau H_i de la hiérarchie. Nous avons donc $w_{H_0} > w_{H_1} > \cdots > w_{H_n}$. On note $cons(H_i)$ l'ensemble des contraintes appartenant au niveau H_i avec $0 \leq i \leq n$. On note $hier(P)$ la hiérarchie de contraintes associée au HCSP P.*

Cette hiérarchie contient donc l'ensemble des contraintes d'un problème considéré classées selon leur poids.

Pour pouvoir comparer différentes instanciations d'un HCSP, P_h on introduit une fonction d'erreur e mesurant le degré de validation d'une contrainte $c \in cons(P_h)$ par une instanciation $i \in esp(P_h)$ donnée.

Définition 2.2.14 (Fonction d'erreur)**.** *Soit $e : cons(P_h) \times esp(P_h) \to \mathbb{R}$ une fonction d'erreur mesurant la validité d'une contrainte pour une instanciation donnée. Une erreur $e(c, i) = 0$ signifie que la contrainte est parfaitement vérifiée et $e(c, i) > 0$ indique que la contrainte n'est pas vérifiée. On note $err(P_h)$ la fonction d'erreur associée à P_h.*

Il existe de nombreuses fonctions d'erreurs dépendant principalement du domaine des variables. Par exemple, pour des contraintes sur des entiers, l'erreur associée à la contrainte $x = y$ peut-être $|x - y|$.

Pour pouvoir comparer deux solutions, il est nécessaire de définir deux fonctions d'agrégation. La première, $g : hier(P_h) \times exp(P_h) \to \mathbb{R}$, calculant l'erreur associée à un niveau de la hiérarchie pour une instanciation. Et une seconde, $G : exp(P_h) \to \mathbb{R}$, calculant l'erreur globale associée à une instanciation.

Par exemple il est possible de définir g et G pour un HCSP P_h comme :

$$g(h, i) = \sum_{c \in cons(h)} err(P_h)(c, i)$$

$$G(i) = \sum_{h \in hier(P_h) \setminus hier(P_h)[0]} w_h * g(h, i)$$

Ces fonctions permettent d'agréger les différentes erreurs à chaque niveau en une erreur globale. On note que les contraintes nécessaires du niveau H_0 ne sont pas incluses dans le calcul de l'erreur, car celles-ci doivent forcément être vérifiées. On considère qu'une solution ne validant pas toutes les contraintes nécessaires aura une erreur globale infinie.

Comme pour les COP, on considérera les solutions optimales du problème comme les solutions minimisant l'erreur globale.

Les contraintes pondérées

Similaires aux HCSP dans l'idée, les **W**eighted **C**onstraint **S**atisfaction **P**roblem (WCSP) sont des modèles où un poids est associé à chaque contrainte. Comme pour les HCSP, un score est calculé en calculant la somme des poids des contraintes non satisfaites. Ainsi, on peut voir le WCSP comme une version relâchée du CSP. En effet, il est possible de ne pas satisfaire toutes les contraintes d'un WCSP. C'est donc, comme le HCSP, une modélisation adaptée aux problèmes trop contraints dans lesquels il n'est pas possible de trouver une solution satisfaisant toutes les contraintes.

Les contraintes *molles*

Plus généralement, ces idées de contraintes facultatives peuvent être regroupées dans une famille de formalisations, les contraintes *molles* (*soft constraints* en anglais), aussi appelées **S**oft **C**onstraint **S**atisfaction **P**roblem (SCSP). Ce formalisme, présenté en détail dans [Bis04], permet de modéliser de nombreux problèmes dans lesquels la réponse au problème n'est pas stricte (oui ou non). Ce formalisme permet aussi d'obtenir des solutions imparfaites lorsque le problème est trop contraint et qu'il n'existe pas de solution.

Les contraintes molles regroupent donc de nombreuses autres modélisations. Les HCSP, les CSP flous [Rut94] ou les CSP probabilistes [FL93].

Exemples de domaines des variables

Nous avons vu qu'il existe plusieurs formalismes, chacun adapté à la représentation de problèmes spécifiques. Par exemple, un SCSP ou un HCSP sont particulièrement adaptés aux problèmes trop contraints qu'il faut relâcher (ignorer certaines contraintes).

Parallèlement à ces différents formalismes, il faut aussi prendre en compte les différents domaines des variables. En effet, comme nous l'avons vu, les formalismes présentés plus tôt sont indépendants des domaines utilisés pour les variables.

Le premier grand domaine nous intéressant est le domaine discret, que nous séparons en deux parties.

Le plus simple, le domaine booléen, où l'on ne considère que deux valeurs possibles $\top$ (vrai) et $\bot$ (faux). C'est notamment le domaine utilisé par le **B**oolean **S**atisfiability **P**roblem

(SAT). Ce problème consiste à trouver une affectation des variables booléennes vérifiant une formule booléenne composée. Par exemple, la formule

$$(x \vee \neg y) \wedge (y \vee z)$$

est vérifiée par l'affection suivante $x = \top$, $y = \bot$ et $z = \top$.

Le domaine des entiers (naturels ou relatifs). Ce domaine est utilisé par la majorité des solveurs de contraintes. Il est généralement suffisant pour représenter les problèmes le plus courants. Par exemple, il est possible de modéliser des problèmes tels que le problème des n dames [4] ou le sudoku.

Certains problèmes nécessitent l'utilisation des variables sur $\mathbb{R}$ ou des sous-ensembles de $\mathbb{R}$. Dans ce cas on parle de domaines continus. C'est par exemple le cas lors de l'optimisation d'une fonction mathématique.

Par exemple,

$$\begin{aligned} \underset{x,\,y}{\text{minimize}} \quad & f(x,y) = x^2 - 2x + y^2 + 4y + 5 \\ \text{subject to} \quad & x < 0, \\ & y \geq 0 \end{aligned}$$

possède une unique solution optimale pour $x = 1$ et $y = 0$, $f(1,0) = 4$.

Il existe d'autres domaines, adaptés à des problèmes spécifiques. On peut citer les logiques descriptives spatiales (**R**egion **C**onnection **C**alculus 8 (RCC8)) [RCC92] permettant de raisonner sur des placements de régions dans un espace euclidien. Ou bien l'algèbre des intervalles d'Allen [All90] qui permet de modéliser des intervalles de temps. On qualifie ces domaines de qualitatifs. En plus des méthodes de résolutions présentées précédemment, il est possible de transformer ces problèmes en problèmes SAT.

2.2.2 Méthodes de résolution

Il existe de nombreuses méthodes de résolution pour les CSPs, COPs et HCSPs. Ces méthodes peuvent être classées en deux grandes familles. Les méthodes exactes offrant des garanties de résultats (solution, si elle existe, pour un CSP ou solution optimale, si elle existe, pour un COP ou un HCSP), mais coûteuses en temps, car nécessitant une exploration exhaustive de l'espace des solutions. Les méthodes approchées sacrifient la garantie de résultat au profit du temps d'exécution. Contrairement aux méthodes exactes, les méthodes approchées n'explorent qu'une partie limitée de l'espace de recherche. C'est pourquoi ces méthodes ne donnent généralement pas de garantie sur la qualité des résultats.

4. Ce problème consiste, à l'origine, à placer 8 dames d'un jeu d'échecs sur un échiquier de 8×8 et plus généralement à placer n dames sur un échiquier de $n \times n$.

Propagation de contraintes et backtrack

La méthode du backtack est une méthode de résolution exacte. Celle-ci consiste en l'exploration complète de l'espace de recherche. Pour se faire, l'algorithme du backtrack utilise un arbre de recherche.

Cette structure d'arbre permet de représenter efficacement l'ensemble de l'espace de recherche. La racine de cet arbre correspond à une affectation vide, aucune variable n'est affectée. Les feuilles quant à elles regroupent toutes les affectations complètes possibles. Les nœuds internes représentent des affectations partielles où une partie seulement des variables sont affectées à des valeurs. Les arcs eux encodent l'affectation spécifique d'une variable. Avec cette structure, un parcours de l'arbre de recherche assure une exploration complète de l'espace de recherche.

La méthode naïve du backtrack, détaillée dans l'Algorithme 1, consiste en un parcours en profondeur de cet arbre de recherche jusqu'à la découverte d'une solution valide du CSP (ou parcours complet de l'arbre pour trouver la ou les solutions optimales des COPs et HCSPs). L'algorithme affecte chacune des variables lors des différents points de choix que sont les nœuds intermédiaires. À chaque point de choix, l'algorithme vérifie que l'affectation partielle ne viole aucune contrainte, auquel cas il ne sert plus à rien de compléter l'affectation partielle, car celle-ci ne pourra jamais vérifier toutes les contraintes et il faut donc annuler la dernière affectation et tester une autre valeur possible pour cette variable.

Algorithme 1 Backtrack naïf

```
1 : fonction BACKTRACK(X, D, C, A)
2 :     si VIOLE(A, C) alors
3 :         retourne ∅
4 :     sinon
5 :         si X = ∅ alors
6 :             retourne A
7 :         fin si
8 :         pour v ∈ X faire
9 :             pour x ∈ D_v faire
10 :                A' ← AJOUTEAFFECTATION(A, v, x)
11 :                res ← BACKTACK(X \ {v}, D, C, A')
12 :                si res ≠ ∅ alors
13 :                    retourne res
14 :                fin si
15 :            fin pour
16 :        fin pour
17 :        retourne ∅
18 :    fin si
19 : fin fonction
```

Cette méthode naïve assure qu'une solution sera trouvée si elle existe. Cependant, il n'y

a pas de garantie sur la taille de l'espace de recherche exploré avant de trouver une solution. C'est pourquoi il existe de nombreuses méthodes venant étendre l'algorithme du backtrack afin d'arriver plus vite à une solution. Ces méthodes agissent de plusieurs façons.

La majorité de ces techniques cherchent à minimiser le nombre d'affectations nécessaires avant de détecter que l'affectation partielle n'est pas possible. C'est l'idée du *fail fast*.

Une stratégie couramment utilisée consiste, à chaque sélection de variables (ligne 8), à sélectionner la variable ayant le plus petit domaine. Une autre stratégie complémentaire très utilisée est l'idée de propagation de contraintes.

L'idée est de réduire les domaines des variables au fur et à mesure de la descente dans l'arbre de recherche grâce à l'affection partielle qui est considérée. En effet, il est très probable qu'à chaque point de choix, une partie des domaines des variables non affectées ne satisfasse pas les contraintes et donc ne vaille pas la peine d'être explorée. Pour éviter l'exploration inutile de ces sous-arbres, on utilise les techniques de propagation. Ces techniques visent à retirer des domaines des variables non affectées les valeurs incompatibles avec l'affection partielle considérée. Lors de chaque choix, les conséquences de ce choix sont propagées aux variables non affectées et réduisent ainsi la taille des domaines et donc la taille du sous-arbre à explorer. La plus connue de ces méthodes est l'*Arc Consistency*. Il existe de nombreux dérivés de cette méthode.

En plus de ces méthodes de sélection de variables et de réduction de domaines, on peut aussi citer des méthodes visant à déterminer un ordre d'exploration du domaine de la variable considérée.

Recherches locales

Contrairement à la méthode du backtrack, les méthodes basées sur la recherche locale sont inexactes, ou incomplètes. Celles-ci permettent d'obtenir des résultats plus rapidement, mais le font en n'explorant l'espace de recherche que partiellement. C'est pourquoi ces méthodes n'offrent pas de garantie de résultats. Il est tout à fait possible que la solution à un CSP se trouve dans une partie de l'espace de recherche qui ne sera jamais exploré.

Plutôt que de construire méthodiquement toutes les solutions possibles et les vérifier, les algorithmes de recherches locales construisent une solution initiale puis explorent l'espace de recherche en voyageant de voisin en voisin. Les voisins d'une solution sont les solutions pouvant être atteintes par l'application d'une variation à la solution originale. Il existe de nombreuses méthodes de génération de voisinage dépendant du problème. Pour un CSP on peut considérer le changement d'affection d'une variable comme une variation, ainsi le voisinage d'une solution correspond ici à toutes les solutions n'ayant qu'une variable modifiée. Dans ce voisinage il faut sélectionner un voisin en particulier qui sera la nouvelle solution considérée. Pour cela il existe aussi de nombreuses méthodes telles que le *premier améliorant* qui sélectionnera le premier voisin améliorant. Dans le cas d'un CSP, on considère qu'une solution est améliorante si elle viole moins de contraintes.

Algorithme 2 Recherche Locale

```
1 : fonction LOCALSEARCH(X, D, C, pasMax)
2 :     a ← GÉNÉRATIONALÉATOIRE(X, D)
3 :     pour pas ← 1 à pasMax faire
4 :         si VIOLE(A, C) alors
5 :             a ← CHANGE(a, D, C)
6 :         sinon
7 :             retourne a
8 :         fin si
9 :     fin pour
10 :    retourne "pas de solution"
11 : fin fonction
```

L'Algorithme 2 donne une implémentation naïve d'une recherche locale. Premièrement, une solution aléatoire est construite, d'autres méthodes peuvent être utilisées pour générer de meilleures solutions initiales. Puis, l'algorithme va chercher à l'améliorer jusqu'à ce qu'une solution valide soit trouvée ou qu'un critère d'arrêt soit rencontré. S'il trouve une solution valide alors il la retourne, sinon il indique ne pas avoir trouvé de solutions. Il existe de nombreuses variantes de l'algorithme naïf permettant d'obtenir de meilleurs résultats plus rapidement, on peut par exemple citer la méthode du recuit simulé [KGV83] (*Simulated Annealing*).

Cependant, comme l'espace de recherche n'est pas totalement exploré, il n'est pas possible de conclure qu'un CSP est insatisfiable si aucune solution n'est trouvée. Similairement, pour les COP, la solution retournée n'est pas forcément la solution optimale.

2.3 Méthodes de résolution de problèmes appliquées à l'Ingénierie Dirigée par les Modèles

Les challenges liés à l'IDM sont nombreux et ne se limitent pas à la transformation de modèles. Dans la sous-section 2.3.1 nous aborderons les problèmes de validation de modèles et notamment les approches visant à générer des modèles respectant des conditions de validité (*well formeness*). Puis, dans la sous-section 2.3.2 nous traiterons des méthodes de transformation de modèles utilisant des techniques de résolution de contraintes ou de programmation logique pour calculer la transformation. Enfin, dans la sous-section 2.3.3 nous présenterons quelques applications de méthodes de résolution approchées à l'IDM.

2.3.1 Génération, vérification et réparation de modèles

Les métamodèles spécifient la topologie du graphe des graphes d'objets qui constituent les modèles. Cependant, ces métamodèles seuls ne sont pas toujours suffisants pour totalement

spécifier les modèles corrects.

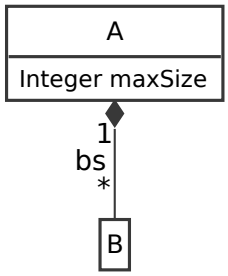


FIGURE 2.3 – Exemple de métamodèle

Prenons l'exemple du métamodèle de la Figure 2.3, celui-ci est composé de deux classes `A` et `B`. La classe `A` possède un attribut `maxSize` et une collection de `B`, `bs`. On voudrait pouvoir préciser que le nombre de `B` référencés dans `bs` ne doit jamais dépasser `maxSize`. Cependant, ça n'est pas directement possible. Pour spécifier cela il faut passer par l'invariant OCL suivant :

```
context A inv : self.maxSize >= self.bs->size()
```

Dans de nombreux cas, il est nécessaire d'exprimer ces règles définissant quels modèles sont valides et quels modèles ne le sont pas. Cependant, lors de la conception du métamodèle il peut être intéressant d'informer l'utilisateur que le métamodèle qu'il est en train de spécifier est incohérent, qu'il n'existe aucun modèle respectant les contraintes. Similairement, il est aussi pertinent de générer des modèles valides afin de servir d'exemples ou de tests pour d'autres outils.

Plusieurs approches ont été proposées pour générer des modèles valides. On peut citer plusieurs approches utilisant directement des solveurs de contraintes [CCR07 ; KJS11 ; SBM09 ; SBP07 ; Soe+10]. D'autres s'inspirent des méthodes utilisées par les solveurs [GBR07 ; HV09 ; HV12 ; SNV18].

Dans leurs papiers [MC16 ; MGC13], les auteurs présentent une approche basée sur la transformation bidirectionnelle et la résolution de contraintes permettant de :

1. générer des modèles conformes à leurs métamodèles et vérifiant des contraintes de validité exprimées en OCL,
2. vérifier la validité d'un modèle par rapport à son métamodèle et ses contraintes OCL,
3. réparer un modèle non conforme à son métamodèle en le modifiant le moins possible.

2.3.2 Spécification de transformation par des contraintes

Les transformations unidirectionnelles permettent de générer un modèle cible à partir d'une source. Cependant, elles ne permettent que rarement de prendre en compte des modifications appliquées à la cible.

Par exemple, considérons une transformation générant du code à partir d'un modèle. Il est courant que le code généré doive être manuellement modifié.

Avec **J**anus **T**ransformation **L**anguage (JTL) [Cic+11 ; CRE06] les auteurs proposent une méthode basée sur l'**A**nswer **S**et **P**rogramming (ASP) [GL88] permettant, à partir d'un modèle cible modifié, de déduire un ensemble de modèles sources compatibles avec la cible modifiée.

Pour se faire, les modèles, leurs métamodèles et la transformation sont encodés en ASP. Puis, à partir de ces informations, la transformation est exécutée, calculant ainsi le modèle cible à partir de la source. La transformation est composée de relations décrivant les liens entre les éléments des métamodèles sources et cibles et de contraintes à respecter. Cette génération se fait en deux temps, premièrement le solveur ASP génère toutes les solutions puis filtre pour ne conserver que les modèles respectant les contraintes. La trace reliant le modèle source au modèle cible est conservée pour permettre la propagation de changements du modèle source au modèle cible ou du modèle cible au modèle source.

Dans [ST09 ; von+13] les auteurs présentent une méthode utilisant un solveur de contraintes pour résoudre des problèmes de refactoring de code. Lors du refactoring de code, de petits changements sur un objet peuvent avoir d'importantes implications ailleurs dans l'application. C'est pourquoi de nombreux outils proposent d'automatiser ces tâches permettant un gain de temps et l'assurance d'un résultat sans erreurs.

Par exemple, dans [ST09] les auteurs modélisent le problème de changement de visibilité de méthodes comme un problème CSP et l'utilisent pour s'assurer qu'un changement de visibilité d'aura pas d'impact sur la sémantique du programme et quels changements faire au besoin. Dans [von+13], la technique est étendue à d'autres types de refactoring tels que le renommage de méthode ou d'attribut ou le déplacement d'attribut.

2.3.3 Search-Based Software Engineering

D'autres approches utilisent aussi des techniques d'optimisation pour résoudre des problèmes liés au génie logiciel. On regroupe ces approches sous le terme générique de **S**earch-**B**ased **S**oftware **E**ngineering (SBSE). Ce terme a été introduit par [HJ01].

L'objectif de ces approches est de résoudre des problèmes de génie logiciel en les modélisant comme des problèmes d'optimisation pouvant être résolus par des méthodes de résolution de problèmes. Ces problèmes d'optimisation sont souvent complexes et les méthodes de recherche exacte, telles que celles présentées dans la section 2.2.2, ne permettent pas d'avoir de résultats dans un temps acceptable. C'est pourquoi on préfère utiliser des métaheuristiques.

Métaheuristiques

Les métaheuristiques sont des algorithmes génériques de résolution approchée de problèmes. Elles ont l'avantage de pouvoir s'adapter à de nombreuses situations tout en offrant

de bonnes performances. On peut regrouper les métaheuristiques en plusieurs catégories en fonction de leurs caractéristiques.

Certaines méthodes, telles que les recherches locales (section 2.2.2) ou les recherches tabou [Glo86], ne considèrent qu'une solution et son voisinage. Ces méthodes voyagent donc de solution en solution jusqu'à satisfaire un critère d'arrêt (généralement un nombre d'itérations).

D'autres considèrent plutôt un ensemble de solutions.

Les méthodes évolutionnaires s'inspirent des mécanismes biologiques de la reproduction. Chaque solution correspond à un individu possédant un ensemble de gènes correspondant aux valeurs affectées aux variables du problème. Les gènes de ces individus sont d'abord brassés comme ils pourraient l'être lors de la reproduction sexuée puis ils peuvent subir des mutations aléatoires. Enfin, les individus les moins bien adaptés (c'est-à-dire ayant les moins bonnes valeurs de la fonction d'objectif) sont éliminés. On recommence ainsi jusqu'à ce qu'une condition d'arrêt soit atteinte. Parmi les algorithmes les plus connus, on peut citer les algorithmes génétiques [Hol92].

Les autres approches cherchent à reproduire les mécanismes d'intelligence collective observable dans la nature (par exemple les déplacements de fourmis). Ces méthodes se basent sur des agents simples, limités à des actions simples, mais capables de communiquer. Les agents n'obéissent pas à une entité centrale, mais respectent tous un ensemble de règles simples. Chaque agent est très simple, mais ensemble des comportements émergeant apparaissent et permettent de résoudre efficacement le problème. Parmi les méthodes les plus connues, on peut citer la méthode des colonies de fourmis [DMC96] ou l'optimisation par essaims particulaires [KE75] (**P**article **S**warm **O**ptimization).

Applications

Les applications de ces méthodes aux problématiques du génie logiciel sont nombreuses, [HMZ12] en liste quelques-unes :

1. Quel est le plus petit ensemble de tests couvrant la totalité du code d'un programme donné ?
2. Quelle est la meilleure façon de structurer un système d'information pour améliorer sa maintenabilité ?
3. Quelles exigences permettent de limiter le coût de développement tout en assurant la satisfaction du client ?
4. Comment répartir efficacement les ressources de ce projet ?
5. Comment réusiner le code source d'un projet pour améliorer la maintenabilité ?

Ces applications couvrent de nombreux domaines du génie logiciel, mais au cœur sont toutes des problèmes d'optimisation. C'est là le cœur du SBSE, la résolution de problèmes d'optimisation liés au génie logiciel. Dans une nouvelle revue plus récente, [BSA17] présente de nombreuses applications et les méthodes utilisées.

Parmi les approches citées, on peut s'attarder sur la génération de transformation de modèles par l'exemple (**M**odel **T**ransformation **B**y **E**xample (MTBE)). Comme le nom l'indique, ces méthodes dérivent les relations nécessaires à la transformation de modèles automatiquement à partir de pairs d'exemple de modèles sources et leurs modèles cibles correspondants. Cela évite l'étape de spécification de la transformation, qui est généralement complexe et nécessite une personne qualifiée. En effet, il est souvent plus simple de fournir un ensemble d'exemples plutôt que de spécifier la transformation. La majorité des approches se concentrent sur la détection de correspondances entre les modèles sources et cibles pour ensuite déduire des règles de transformation.

D'autres s'intéressent plutôt à la génération de modèles pour tester automatiquement les transformations de modèles. Pour automatiser ces tests, les méthodes génèrent des modèles sources. De ces modèles sources sont dérivés des modèles cibles. Ces modèles sont ensuite examinés par un oracle pour déterminer s'ils sont corrects ou non. Les modèles sources ne peuvent pas être générés totalement aléatoirement. En effet, pour assurer une bonne couverture de test il est nécessaire que les modèles :

1. Soient conformes au métamodèle source.
2. Respectent des contraintes de validité comme présentée dans la section 2.3.
3. Couvrent toutes les parties de la transformation.

Le SBSE regroupe de nombreux domaines, on pourra consulter ces deux revues de l'état de l'art [BSA17 ; HMZ12] pour plus de détails.

CHAPITRE 3

CONTRAINTES POUR L'EXPLORATION DE MODÈLES

Ce chapitre est basé sur [Le +19a] et introduit la notion d'ensembles de modèles ainsi que la méthode que nous avons mise au point pour les explorer. Dans la section 3.1, nous définirons les notions d'ensembles de modèles et d'exploration d'ensembles de modèles. Puis, dans la section 3.2, nous présenterons une méthode permettant l'exploration d'ensembles de modèles.

3.1 Ensemble de modèles explorables

Les méthodes de transformation de modèles classiques génèrent un unique modèle à partir d'un modèle source. Cependant, il est parfois préférable de générer non pas un mais plusieurs modèles cibles. Par exemple, dans le cas où la transformation consiste en la génération d'un diagramme. Il existe de nombreux modèles de diagrammes représentant le modèle source. Certains seront plus lisibles que d'autres. Cependant, déterminer programmatiquement quels diagrammes sont préférables est difficilement exprimable dans les langages utilisés pour la transformation.

C'est pourquoi, nous proposons une approche permettant au développeur d'exprimer cette espace de solutions possibles (l'ensemble des diagrammes valides) et de l'explorer (trouver le meilleur diagramme). Nous appelons ensemble de modèles la structure exprimant cet espace de solutions.

3.1.1 Ensemble de modèles

Avant de définir un ensemble de modèles nous rappelons la définition d'un espace de modèles introduite dans [Dis+14 ; DXC10].

Définition 3.1.1. *Un espace de modèles (*model space*) est un graphe orienté $M = (M_\bullet, M_\triangle)$ où $M_\bullet$ est un ensemble de nœuds aussi appelés modèles et où $M_\triangle$ est un ensemble d'arcs appelés deltas ou mise à jour. On note $\delta_s : M_\triangle \to M_\bullet$ et $\delta_t : M_\triangle \to M_\bullet$ deux fonctions prenant un arc $a \in M_\triangle$ et retournant respectivement, le modèle avant l'application du delta (δ_s) et après (δ_t).*

Les modèles d'un espace de modèles sont reliés entre eux par des deltas correspondant aux mises à jour à effectuer pour passer d'un modèle à l'autre.

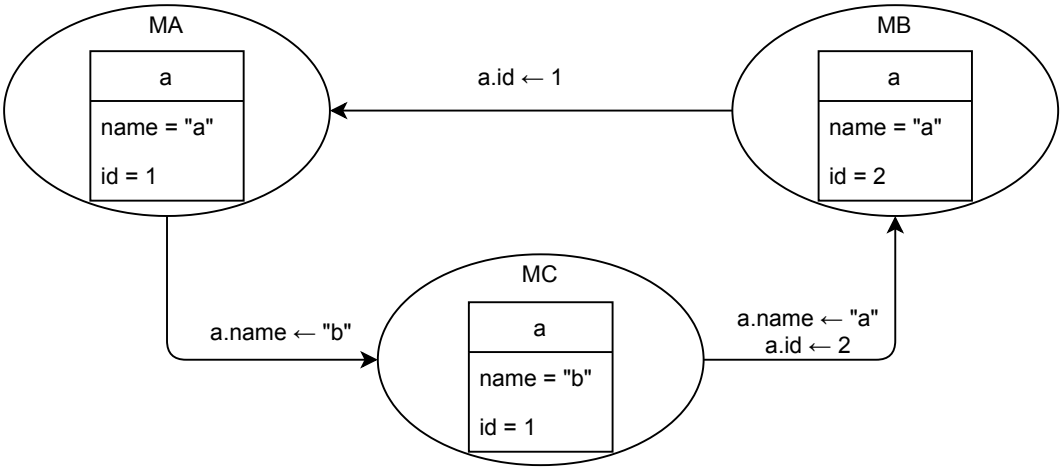


FIGURE 3.1 – Exemple d'espace de modèles

Par exemple, la Figure 3.1 montre un espace de modèles simple. Celui-ci est composé de trois modèles, `MA`, `MB` et `MC`, chacun contenant un objet `a` ayant deux propriétés `name` et `id`. Les deltas sont notés à côté des arcs. On peut voir qu'il est possible de passer de `MA` à `MC` en affectant la valeur `"b"` la propriété `name` de l'objet `a`. Aussi, il est possible de passer de `MC` à `MB` en modifiant les propriétés `name` et `id`. De plus, par transitivité, il est possible de naviguer de `MA` à `MB` en passant par `MC`.

On nomme *exploration d'un espace de modèles* l'application de deltas à des modèles appartenant à $M_\bullet$.

Définition 3.1.2. *L'exploration d'un espace de modèles est définie comme une liste de deltas devant être appliqués les uns après les autres. Soit* $e = \{d_1, d_2, \cdots, d_n\}$ *une exploration de longueur* n*, on nomme modèle départ (ou initial) le modèle* $\delta_s(d_1)$ *et modèle de fin (ou terminal) le modèle* $\delta_t(d_n)$*. Pour être bien construite une exploration doit vérifier :* $\delta_t(d_{n-1}) = \delta_s(d_n)$*. On note* $E(M_\Delta)$ *l'ensemble des explorations bien construites de* M_Δ*.*

Dans notre cas, nous ne considérons qu'un modèle à la fois, on ne montre qu'un seul modèle à l'utilisateur. Celui-ci peut alors le modifier en lui appliquant des modifications correspondant à des arcs de M_Δ. Par exemple, il existe une exploration de l'espace de modèles partant du modèle MA et allant au modèle MB en passant par le modèle MC.

3.1.2 Exploration d'ensemble de modèles

Dans un problème réel, seul un sous-ensemble de tous les modèles possibles est pertinent. En effet, la seule conformité à un métamodèle n'est pas suffisante pour assurer qu'un modèle

soit correct. Pour cela, il faut, en plus du métamodèle, un ensemble de *règles*, ou *contraintes*, définissant précisément la notion de validité d'un modèle. Ainsi un modèle valide est un modèle conforme à son métamodèle et respectant un ensemble de contraintes d'intégrité. Cette notion de validité est à faire correspondre avec la *well-formedness* utilisée dans [SV17].

Par exemple, dans le cas d'un diagramme, les formes géométriques qui le composent ne peuvent pas être placées n'importe où. Il faut respecter un ensemble de règles de constructions spécifique à chaque diagramme. De cette manière un diagramme valide est un diagramme ayant des éléments géométriques bien formés (conformité au métamodèle), mais aussi correctement placés (respect des règles de construction).

Définition 3.1.3. *On définit un* ensemble de modèles *comme un sous-ensemble de M_V de $M_\bullet$ ne contenant que des modèles valides.*

Comme pour un espace de modèles, il est possible d'explorer un ensemble de modèles en appliquant successivement des mises à jour de M_Δ. À la différence d'un espace de modèles, après l'application d'un delta sur un modèle appartenant à M_V il n'y a aucune assurance que le nouveau modèle soit valide (appartienne à M_V).

Définition 3.1.4. *L'exploration d'un ensemble de modèles est dite correcte si modèle initial m_s et terminal m_t de l'exploration appartiennent à M_V, $m_s \in M_V$ et $m_t \in M_V$.*

C'est pourquoi tous les chemins d'exploration d'un espace de modèles ne sont pas nécessairement aussi des chemins d'exploration d'un ensemble de modèles.

La Figure 3.2 illustre cette notion d'exploration. Les points correspondent à des modèles et les flèches à des deltas. La grande ellipse correspond à un espace des modèles, celle en pointillés correspond à un ensemble de modèles. On peut voir par exemple qu'il est possible de naviguer du modèle `a` au modèle `e` en suivant les deltas de `a` à `b`, puis de `b` à `d` et de `d` à `e`. Comme les modèles `a` et `e` sont valides le trajet de `a` à `b` est considéré comme une exploration valide.

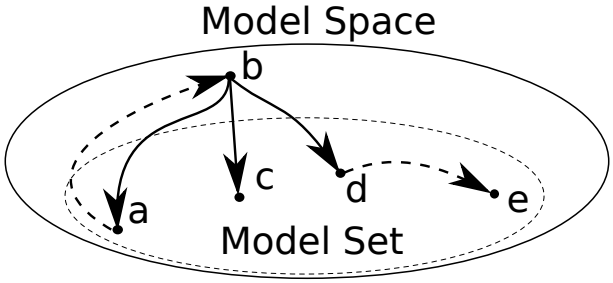


FIGURE 3.2 – Exemple d'exploration d'un ensemble de modèles.

Dans l'exemple précédent, on remarque qu'après la première mise à jour, le modèle `b` n'est pas valide. Notre objectif est de toujours afficher à l'utilisateur des modèles valides. Par conséquent, il faut donc le *réparer*. Revenons à la Figure 3.2, on remarque qu'il y a deux

types de flèches. Les flèches pointillées représentent des actions effectuées par l'utilisateur. Comme nous l'avons vu, les actions de l'utilisateur peuvent rendre le modèle invalide. Il est alors nécessaire de le réparer, ces actions de réparation sont indiquées par les flèches pleines.

Donc , après une mise à jour de l'utilisateur, le modèle peut être dans deux états possibles :

1. **Valide** : l'utilisateur a modifié le modèle et toutes les règles sont respectées (par exemple le modèle `e` dans la Figure 3.2). Comme le modèle est déjà valide, il n'y a pas de réparation à faire.
2. **Invalide** : l'utilisateur a modifié le modèle et certaines règles sont enfreintes (par exemple le modèle `b` dans la Figure 3.2). Le nouveau modèle est conforme à son métamodèle, mais ne respecte pas toutes les contraintes de validité. Dans ce cas il est nécessaire de réparer le modèle afin d'obtenir un nouveau modèle valide. Pour ce faire il faut suivre un delta amenant à un nouveau modèle appartenant à M_V (les flèches pleines dans la Figure 3.2). Il y a souvent plusieurs deltas possibles (par exemple dans la Figure 3.2, pour réparer `b` il est possible de suivre trois deltas différents). La réparation la plus simple et toujours possible consiste à annuler le changement effectué par l'utilisateur (cela correspond à l'arc allant de `b` à `a` dans la Figure 3.2). D'autres réparations résultant en d'autres modèles valides peuvent exister suivant les règles et le modèle considéré.

Lors de la réparation d'un modèle invalide, il existe de nombreux deltas applicables et tous ne résultent pas en un modèle valide. Par conséquent, déterminer une suite de deltas à appliquer pour obtenir un modèle valide est une tâche complexe. Ce genre de problème fortement combinatoire nécessite des méthodes de résolution spécifiques. Suivant la méthode utilisée, la suite de deltas à appliquer peut ne pas être identique.

On nomme *comportement* la stratégie de résolution déterminant quelle suite de deltas préférer aux autres.

Définition 3.1.5. *Un comportement est une fonction* $C : E(M_\Delta) \to E(M_\Delta)$ *prenant une exploration et retournant une exploration.*

Il existe de nombreuses stratégies plus ou moins pertinentes. Par exemple, le comportement consistant à systématiquement annuler le changement de l'utilisateur n'est pas satisfaisant. Ce comportement correspond à la fonction $annule(e) = \{\}$. On préférera utiliser une stratégie prenant en compte les changements de l'utilisateur. Cette stratégie devrait sélectionner une suite de deltas résultant en un modèle *proche* du modèle invalide donné par l'utilisateur. Cependant, il faudra s'accorder sur la notion de proximité à utiliser.

3.2 Modélisation de l'exploration d'ensembles de modèles avec des contraintes

Comme nous l'avons vu dans la sous-section 3.1.1, l'objectif de notre approche est de permettre à un utilisateur de naviguer de solutions valides en solutions valides. L'application

successive de deltas permet à l'utilisateur d'améliorer le modèle qui lui est présenté. Cependant, ces petits changements peuvent parfois rendre le modèle invalide. Dans ces cas, il faut le réparer. Cette étape de réparation peut être effectuée par un solveur de contraintes. Par exemple, considérons la règle suivante appliquée à un rectangle r, $r.largeur >= 100$. Il y a une infinité de valeurs possibles pour cette largeur, tant qu'elle est supérieure à 100.

Pour permettre l'exploration d'ensembles de modèles, nous proposons une méthode liant les propriétés des modèles à des variables d'un solveur de contraintes. Résoudre le problème de contraintes assure ainsi la validité du modèle.

3.2.1 Vue d'ensemble

Un ensemble de modèles peut être arbitrairement large, voir infini. C'est pourquoi il est nécessaire d'avoir un mécanisme pour les définir en intention. Nous proposons d'utiliser des contraintes pour définir en intention les modèles valides.

Pour cela nous étendons les modèles avec des contraintes. Ainsi, il est possible d'utiliser un solveur de contraintes pour vérifier et réparer au besoin les modèles après une modification de l'utilisateur. Nous proposons donc d'utiliser un solveur de contraintes pour calculer les réparations présentées dans la section 3.1. Pour fonctionner correctement, une synchronisation entre le modèle et le CSP que le solveur résout est nécessaire. Par exemple, l'ajout de nouveaux éléments au modèle doit entraîner l'ajout de nouvelles contraintes. Pour assurer cette synchronisation, nous introduisons le concept de *variable pont* (ou *bridge variable*).

3.2.2 Variable pont

Une variable pont synchronise une propriété d'un élément du modèle avec une variable de décision du problème. Chaque propriété de chaque élément du modèle utilisé dans les contraintes correspond à une et une seule variable pont. Il en va de même pour les variables de décision utilisées par le solveur et les variables ponts. La variable pont garde la variable de décision et la propriété du modèle synchronisé. Si l'une des deux est modifiée alors la variable pont propage le changement à l'autre.

La Figure 3.3 détaille comment la variable pont transmet l'information d'un changement sur une propriété au solveur qui recalcule alors une solution. Après que l'utilisateur ait effectué un changement sur une propriété d'un élément du modèle, la variable point attachée à cette propriété est notifiée du changement et le propage à la variable de décision correspondante. Ensuite, le solveur peut recalculer une solution au besoin et notifier les variables modifiées des changements. D'une façon similaire à la propagation des changements effectuées par l'utilisateur, les changements appliqués aux variables de décisions par le solveur sont transmis aux variables ponts attachées à ces variables qui à leurs tours notifient les propriétés correspondantes de leurs nouvelles valeurs. De par la nature de la résolution de problèmes de

contraintes, il est probable qu'un changement sur une variable nécessite des changements sur de nombreuses autres variables.

Dans la Figure 3.3 nous ne considérons qu'une modification atomique d'une propriété du modèle. De plus, si l'utilisateur modifie plusieurs propriétés dans une même interaction nous traitons les changements les uns après les autres. Ce comportement est suffisant pour une grande majorité des problèmes, mais pas toujours suffisant. En effet, il se peut que lors d'une modification de l'utilisateur il soit nécessaire d'avoir la totalité des changements pour obtenir un modèle valide (comme des changements de position, où les propriétés x et y sont souvent liées). Ainsi, si le premier changement est considéré seul, le solveur peut préférer l'annuler, car les solutions conservant le changement sont trop distantes. Pour remédier à ces cas problématiques, il est possible de ne pas demander au solveur de recalculer une nouvelle solution à chaque changement d'une de ses variables. Cela permet aussi d'appeler moins souvent des solveurs n'offrant pas des performances suffisantes pour résoudre les problèmes de contraintes à chaque modification de l'utilisateur.

La façon dont le solveur calcule les nouvelles valeurs des propriétés correspond à la notion de réparation que nous avons abordée dans la sous-section 3.1.2. Suivant le formalisme utilisé pour encoder le problème de contraintes (CSP, COP ou HCSP) la stratégie de réparation peut être encodée de diverses façons. Dans un HCSP, on associera un poids plus faible aux contraintes affectant des propriétés jugées moins importantes. Par exemple, on peut préférer conserver la taille d'un rectangle plutôt que sa position.

3.2.3 Encodage des propriétés en lecture seule

Dans certains cas particuliers, associer une variable de décision à une propriété du modèle n'est pas approprié. C'est notamment le cas lorsque l'on traite avec des propriétés accessibles en lecture seule, comme les propriétés dérivées. Par exemple, la propriété `width` d'un `Text` n'est accessible qu'en lecture, en effet, cette valeur est calculée à partir du `text` et de la taille de la police utilisée. Dans d'autres cas, l'utilisateur peut ne pas vouloir que le solveur puisse changer la valeur de la propriété. Ceci permet notamment de passer outre les limitations de certains solveurs. Certains solveurs ne permettent pas la résolution de problèmes non linéaires. Transformer une propriété en constante permet donc de linéariser certaines contraintes. Il y a deux grandes méthodes pour créer une constante :

1. Utiliser une variable de décision, mais réduire son domaine à une seule valeur. La transformant donc en constante pour le solveur.
2. Ne pas utiliser de variable de décision et encoder la propriété comme une constante numérique classique.

Les deux solutions permettent la création de constantes, mais ne sont pas équivalentes pour le solveur. Suivant le solveur utilisé, une solution peut être préférée à l'autre.

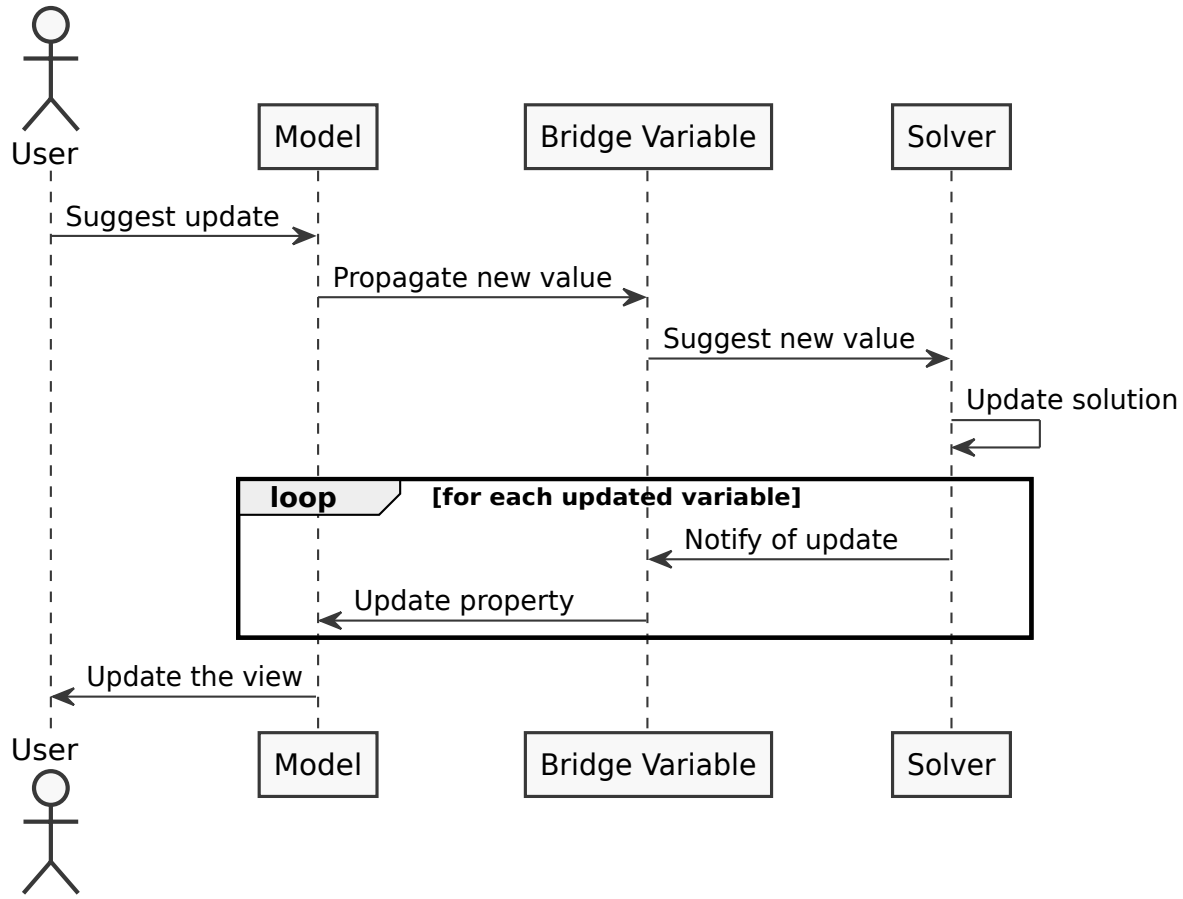


FIGURE 3.3 – Propagation d'un changement par les variables ponts

3.2.4 Encodage des relations

La méthode de synchronisation présentée dans la sous-section 3.2.2 fonctionne bien pour les propriétés numériques et plus généralement les types de variables supportées nativement par les solveurs. Cependant, toutes les propriétés des modèles ne sont pas de type numérique, il est donc nécessaire d'encoder les propriétés non supportées par les solveurs si l'on veut pouvoir raisonner dessus. Les relations font partie de ces types non supportés, cependant, celles-ci sont très utilisées dans les modèles. Ne pas être capable de traiter avec les relations dans les contraintes réduit considérablement l'expressivité des contraintes pouvant être écrites. Heureusement, il existe de nombreuses approches visant à encoder les relations en valeurs numériques.

Une approche simple consiste à assigner un identifiant unique à chaque objet pouvant être utilisé dans une relation et à utiliser des prédicats pour encoder la présence ou l'absence de relation entre les objets [SBP07]. D'autres approches exploitent des techniques de filtrage par motif (*pattern matching*) sur des graphes [HV09 ; SNV18]. Pour notre utilisation nous avons dé-

cidé d'utiliser une approche similaire à, [SBP07] mais en l'adaptant aux limitations des solveurs que nous utilisons. On peut imaginer plusieurs encodages suivant le type de relations.

Par exemple, imaginons deux classes `A` et `B`. La classe `A` possède une propriété `b` faisant référence à un élément du modèle de la classe `B`. On note a un élément instance de la classe `A`. De plus, on notera $ids(X)$ l'ensemble des identifiants valides d'une classe `X` et $id(X, i)$ l'élément de classe `X` ayant l'identifiant i.

Un premier encodage est de donner à chaque instance des classes `A` et `B` un identifiant unique, puis d'encoder la relation de la façon suivante :

$$a.b \in ids(B)$$

Cet encodage a l'avantage d'être simple à mettre en place, mais ne permet d'encoder que les relations de multiplicité 1.

Un second encodage un peu plus complexe permet quant à lui d'encoder les relations de multiplicité n :

$$\forall i \in ids(B) \quad a.b = id(B, i) \implies var_i$$

$$\sum_{i \in ids(B)} var_i = n$$

Cette seconde version est plus complexe et requiert un solveur capable de faire de la réification de contraintes. En effet, cette version génère une contrainte par valeur possible par propriété considérée.

Il est possible de trouver des encodages pour tous les types de relations existant dans les modèles utilisés en IDM.

3.3 Définition du comportement par des contraintes

Dans la sous-section 3.1.2 nous avons présenté la notion de comportement comme étant la stratégie de réparation utilisée lors de la réparation d'un modèle invalide. Par exemple, nous avons présenté le comportement le plus simple qui consiste à annuler tout changement effectué par l'utilisateur.

D'un point de vue théorique un comportement est une fonction qui, à partir d'un delta appliqué à un modèle, calcule une série de deltas à appliquer. Dans notre contexte, nous ne considérons que les explorations d'ensembles de modèles valides. Il faut donc que notre comportement génère une exploration d'ensemble de modèles correcte, c'est à dire une Exploration dont le modèle terminal appartient à l'ensemble des modèles.

Avant de proposer une façon d'encoder ces comportements sous la forme de contraintes, nous illustrerons les différents comportements souhaités avec un exemple.

3.3.1 Le problème du contenant et du contenu

Le problème du contenant et du contenu est un problème récurrant, apparaissant dans de nombreux diagrammes avec lesquels l'utilisateur peut interagir. Il est courant, dans les diagrammes, d'associer à la notion de contenance d'un élément graphique dans un autre un sens dépendant du diagramme.

Par exemple, considérons deux classes `A` et `B` et un package `P` auquel la classe `A` appartient. La Figure 3.4 donne une représentation sous la forme d'un diagramme de classes de ces classes et ce package. L'appartenance de la classe `A` au package `P` est encodé dans le diagramme par le fait que la représentation visuelle de la classe `A` soit contenue dans la représentation visuelle du package `P`. De plus, le fait que la représentation visuelle de la classeă`B` ne soit contenue dans rien indique que celle-ci n'appartient à aucun package.

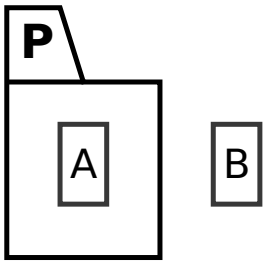


FIGURE 3.4 – Exemple de diagramme de classes

Toujours sur cet exemple du package contenant les classes lui appartenant, imaginons que ce diagramme puisse être modifié par l'utilisateur. On imagine facilement qu'il est possible de déplacer librement les classes tant que ces déplacements n'altèrent la signification du diagramme. Par exemple, déplacer la classe `B` vers la droite ne change pas les classes ni le package. Le diagramme de classes est valide par rapport au modèle de classe qu'il représente.

Cependant, si l'utilisateur déplace la classe `A` hors du package `P` alors la signification du diagramme change. Ainsi, le diagramme n'est plus un diagramme valide par rapport au modèle de classes original. Il faut alors réparer, au choix, le modèle de classes (en modifiant la classe `A` de façon à ce qu'elle n'appartienne plus au package `P`) ou le diagramme. Pour cet exemple considérons que la première solution (modifier le modèle de classes) ne soit pas possible. Il faut donc réparer le diagramme.

Cependant il existe de multiple façon de réparer le diagramme de façon à ce que celui-ci redevienne valide. Voici quelques exemple de réparation possibles :

1. Déplacement du package `P` de façon à ce que la classe `A` soit contenue dans celui-ci.
2. Agrandissement du package `P` de façon à ce que la classe `A` soit contenue dans celui-ci.

3. Réduction de la taille de la classe `A` pour qu'elle reste dans les limites du package `P`.
4. Déplacement de la classe `A` pour qu'elle reste dans les limites du package `P`.

3.3.2 Modélisation d'un comportement par des contraintes

Dans le formalisme vu dans les sections précédentes, le diagramme appartient à un espace de modèles. Les interactions de l'utilisateur sur le diagramme (déplacement des éléments graphiques) correspondent à des explorations de cet espace de modèles. Enfin, les règles de construction d'un diagramme de classes correspondant au modèle de classe présenté plus tôt forment les règles de validation nous permettant de créer un ensemble de modèles. Cet ensemble contient donc tous les diagrammes de classes représentant ce modèle de classe.

La seconde interaction de l'utilisateur, sortant la classe `A` du package `P`, correspond à une exploration de modèle dont le modèle initial appartient à l'ensemble de modèles mais dont le modèle terminal n'appartient pas à cet ensemble de modèles.

Nous avons vu qu'il est possible de définir un ensemble de modèles par des contraintes. Un solveur est capable de calculer un modèle respectant ces contraintes (si une solution au problème de contraintes existe) et appartenant donc à l'espace de modèles. Nous avons également vu qu'un solveur de contraintes peut être utilisé pour réparer un modèle rendu invalide par une interaction de l'utilisateur. Par exemple ici, de déplacement de la classe `A` en dehors du package `P`.

Cependant, sans indication, le solveur calculera une nouvelle solution sans prendre en compte les changements effectués par l'utilisateur. En effet, pour lui le modèle de contraintes n'a pas changé. C'est pourquoi nous avons repris les idées de contraintes `stay` et de suggestion de valeurs présentes dans les solveurs de HCSP.

La contrainte `stay` est une contrainte spéciale dans le sens où elle minimise les changements d'une variable d'une résolution à l'autre. Si on applique la contrainte `stay` à une variable x et que l'on note $prev(x)$ la valeur de la variable x dans la solution précédente. Alors, en interne le solveur crée une contrainte $x - prev(x) = 0$. Lors de la résolution, les solveurs de HCSP minimisent l'erreur pondérée des contraintes. Une contrainte ayant un poids plus élevé aura une erreur plus faible.

La suggestion de valeur à une variable correspond à l'ajout temporaire d'une contrainte. Par exemple, si l'on suggère la valeur 5 à une variable x, le solveur ajoute la contrainte $x = 5$ au problème. Il recalcule une solution et enlève ensuite la contrainte $x = 5$.

Avec ces deux contraintes ainsi que les poids il est possible de définir des préférences que le solveur devra prendre en compte lors de la résolution. Par exemple, pour reprendre le cas de la classe et du package, voici une partie des contraintes définissant un diagramme de classes valide :

Contrainte 1 : Le nom de la classe doit être positionné au centre du rectangle correspondant à la classe.

Contrainte 2 : Une classe doit être assez grande pour que son nom ne dépasse pas.

Contrainte 3 : Une classe doit conserver, sauf modification de l'utilisateur, sa position.

Contrainte 4 : Le rectangle d'une classe doit être contenu dans le rectangle correspondant au package auquel elle appartient.

Contrainte 5 : Un package doit être assez grand pour contenir ses classes.

Contrainte 6 : Un package doit conserver, sauf modification de l'utilisateur, sa position.

Ces contraintes sont suffisantes pour construire un diagramme valide, cependant elles ne décrivent pas comment réparer un modèle invalide. On remarquera que les contraintes 3 et 6, concernant les positions des classes et packages, correspondent à des contraintes `stay` et qu'elles portent un poids faible. Il ne faut pas modifier ces valeurs sauf si une autre contrainte le nécessite. On notera w_i le poids associé à la contrainte i et $w_{suggest}$ le poids associé à la contrainte *suggest* utilisée pour suggérer une nouvelle valeur. Par exemple, w_3 correspond au poids associé à la contrainte 3.

Avec ces contraintes, lorsque l'utilisateur déplace une classe hors du package, celui-ci poura, suivant la formulation exacte des contraintes 3, 5 et 6 :

1. **Déplacer le package** si $w_6 < w_5$ et $w_{suggest} > w_6$.
2. **Agrandir le package** si $w_6 > w_5$ et $w_{suggest} > w_5$.
3. **Replacer la classe dans le package** si $w_{suggest} > w_5$ et $w_{suggest} > w_6$.

Par un contrôle fin des contraintes ainsi que de leurs poids il est possible de spécifier des comportements. Grace à ces indices, l'utilisateur peut indiquer au solveur quelles contraintes sont plus importantes et ainsi, lesquelles doivent être violées en premier.

3.4 Les couches d'abstraction

Nous avons vu que les contraintes permettent la spécification et l'exploration d'ensembles de modèles. De plus, par une définition précise des contraintes et des poids de celles-ci il est possible de définir le comportement que devra suivre le solveur lorsqu'il réparera les modèles invalides.

Cependant, il est parfois préférable de proposer à l'utilisateur un langage plus adapté dans lequel il pourrait exprimer plus naturellement ses besoins. Les contraintes permettent d'exprimer et de résoudre de nombreux problèmes, cependant il peut être difficile pour un utilisateur néophyte d'exprimer correctement ses besoins.

Une façon simple d'aider l'utilisateur est de lui proposer un langage simplifié dans lequel il manipulera directement les concepts pertinents pour la tâche qu'il souhaite effectuer. Dans le cas de notre exemple, placer des éléments géométriques sur un plan. Plus généralement, on nomme ces langages des **D**omain **S**pecific **L**anguage (DSL). Il est courrament admis [MHS05] que les DSL avec leurs expressivités limitées permettent à l'utilisateur néophyte d'atteindre

plus rapidement un meilleur niveau de productivité que des langages généralistes. Parmi les DSL courrament utilisés on peut citer, LaTeX, Make ou SQL.

Dans notre cas, les couches d'abstraction peuvent être assimilées à des DSL simplifiant l'écriture de contraintes en apportant des abstractions proches du domaine métier de l'utilisateur. Par exemple, plutôt que de spécifier des contraintes arithmétiques sur les coordonnées et les dimensions des éléments graphiques d'un diagramme, il est plus simple de raisonner directement en terme de concepts géométriques sur ces formes. Par exemple, spécifier qu'un élément contient un autre élément ou que les coordonnées d'un élément doivent être calculées à partir des coordonnées d'un autre élément et d'une translation.

Dans le cas des contraintes, il existe de nombreux formalismes adaptés à la modélisation de problèmes spécifiques. Par exemple les **T**emporal **C**onstraint **S**atisfaction **P**roblem (TCSP) [DMP91] permettent de modéliser des problèmes dans lesquels les variables sont des points dans le temps et les relations entre celles-ci encodent des information temporelles. On peut aussi citer les logiques decriptives spatiales [RCC92] (RCC8) permettant de spécifier des relations spatiales entre des éléments. Enfin, les **C**onstraint **H**andling **R**ules (CHR) [Frü95 ; Frü98] permettent à l'utilisateur d'ajouter ses propres contraintes par un mécanisme de réécriture de contraintes.

CHAPITRE 4

GESTION DES CYCLES

Ce chapitre est basé sur des travaux [Le +18c] présentés au 18ème *International Workshop on OCL and Textual Modeling (OCL)*. Dans la section 4.1 nous présentons les opérations actives. Puis, dans la section 4.2, nous abordons les problèmes liés à la propagation de changement dans les expressions formées par des opérations actives ainsi qu'une formalisation permettant de détecter ces problèmes. Enfin, dans la section 4.3, nous présentons comment les concepts présentés dans la section 4.2 peuvent être appliqués aux transformations interagissant avec un ou plusieurs solveurs de contraintes.

4.1 Définition des opérations actives

Les opérations actives [Bea+10] sont un ensemble d'opérations opérant sur des collections appelées *boxes*. Ces opération sont inspirées des languages fonctionnels et d'OCL. On y retrouve, par exemple :

— `collect`($\lambda : S \to T$) : applique une fonction λ aux éléments de type S de la collection source pour calculer une collection cible d'éléments de type T (équivalent du `map` des langages fonctionels).

— `select`($pred : S \to \{0, 1\}$) : ne conserve que les éléments e pour lesquels $pred(e) = 1$ (équivalent de `filter`).

— `concat(Box end)` : calcule la collection contenant les éléments de la collection source suivis des éléments de la collection `end`.

— `distinct()` : supprime tous les éléments doublons de la collection.

— `first()` : retourne le premier élément de la collection.

Chacune de ces opérations est composée d'algorithmes capturant les changement sur la ou les collections sources de l'opération, et calculant les changements à appliquer sur la collection cible. Ainsi, chaque opération est capable de calculer les modifications à appliquer à son résultat lorsqu'un changement sur sa ou ses sources est détecté. Par composition, il est alors possible de construire des expressions dont le résultat est mis à jour de façon incrémental et automatique lors des changements de ses sources. La propagation des changements d'opérations en opérations permet de ne pas recalculer la totalité des collections lorsqu'un élément de source change.

Par exemple, soit l'expression `f` suivante :

```
def : f(a) =
    let b = a->distinct() in
    let c = b->first() in
    c
```

Avec `a = [2,1,2]` une liste d'entiers, après le calcul initial de l'expression nous avons, `b = [2,1]` et `c = [2]`. Enlevons le premier élément de `a`, nous avons `a = [1,2]`, l'opération `distinct` est notifiée du changement d'`a` et met à jour `b`. Bous avons donc `b = [1,2]`. Ensuite, pareillement, l'opération `first` est notifiée du changement et met à jour `c`, qui devient `c = [1]`.

4.2 Modélisation de la propagation des opérations actives

La propagation des changements d'une opération à l'autre est simple et facile à visualiser. Cependant, lorsque ces opérations sont composées en expressions, la visualisation de la propagation des changements devient complexe. C'est pourquoi nous introduisons la notion de graphe de propagation. Ce graphe permet de visualiser et de raisonner sur la propagation des changements entre les opérations composant une expression.

4.2.1 Graphe de propagation

Chaque opération active s'enregistre auprès des opérations actives lui servant de source. Ainsi, dès qu'une source change, elle peut notifier toutes les opérations suivantes de la nouvelle valeur. Une fois connectées les unes aux autres, les opérations forment un graphe orienté. On appelle ce graphe le *graphe de propagation* (ou *propagation graph* en anglais).

Dans l'**A**ctive **O**peration **F**ramework (AOF), en plus des opérations vues plus tôt il existe la notion de *box* qui sert à encapsuler les propriétés et valeurs simples, leur permettant ainsi d'être connectées à des opérations. Les *boxes* servent d'interfaces entre les opérations actives et le reste du programme. On peut considérer une *box* comme une opération identité, qui transmet les changements qu'elle reçoit sans les modifier. Dans la suite de ce chapitre, les *boxes* seront donc ignorées.

Par exemple, le graphe visible dans la Figure 4.1 correspond à l'expression suivante (utilisée dans [Jou+18]) :

```
def : f(a) =
    let b = a->collect(λ1) in
    let c = a->collect(λ2) in
    let d = b->zip(c) in
    d
```

Un autre exemple plus complexe avec plusieurs boxes sources est visible en Figure 4.2. Voici son expression équivalente :

```
def : g(a, b) =
    let v1 = a->A() in
```

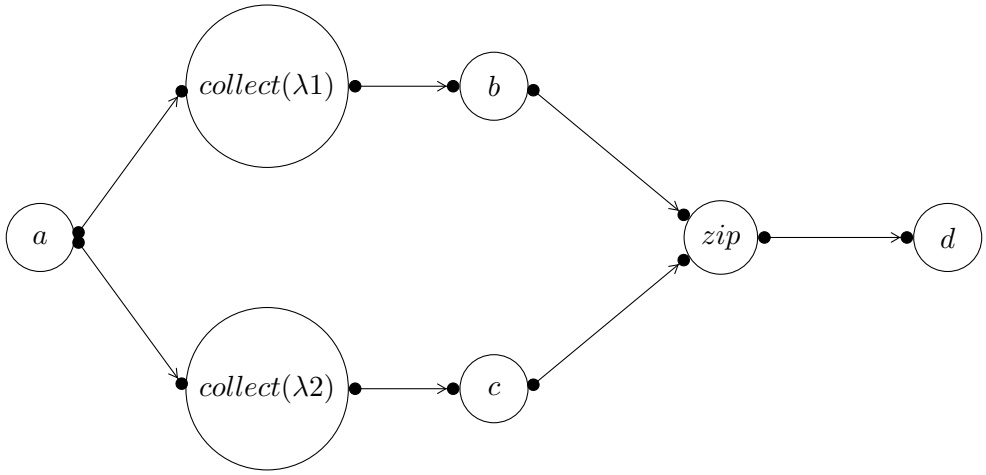


FIGURE 4.1 – Exemple de propagation problématique

```
let v2 = a->B(b) in
let v3 = v1->C(v2) in
let v4 = v2->D() in
let c = v3->E(v4) in
c
```

Dans cet exemple, les noms des opérations ont été remplacés par de faux noms d'opérations pour simplifier l'identification et réduire la taille des nœuds. Ces modifications ne posent pas de problèmes, car les problèmes de propagation que nous résolvons ne sont pas directement liés aux opérations utilisées dans le graphe de propagation, mais à sa topologie. De plus, nous supprimons les boxes intermédiaires pour simplifier la représentation visuelle car celles-ci sont assimilables à des opérations identités.

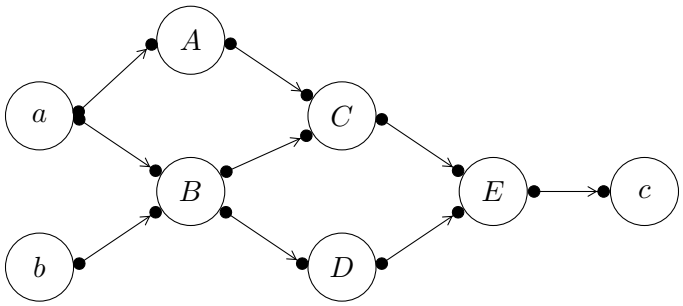


FIGURE 4.2 – Un autre exemple de situation problématique

Dans [Jou+18], il a été montré que l'approche basée sur l'observation que nous utilisons pour AOF ne permet pas de correctement gérer les cas où plusieurs paramètres d'entrée d'une opération dépendent d'une même box, directement ou non. Plusieurs solutions pour contourner le problème ont été proposées.

Ce problème apparaît lorsqu'il existe plusieurs chemins distincts entre une box et une opération. En effet, l'implémentation actuelle notifie les opérations des changements selon un parcours de l'arbre de propagation qui correspond à un parcours en profondeur. S'il est possible d'atteindre un nœud par plusieurs chemins depuis la racine, alors les descendants de ce nœud seront parcourus plusieurs fois.

Par exemple, sur l'expression illustrée en Figure 4.1, l'opération `zip` est notifiée deux fois, et par conséquent, l'opération `d` est aussi notifiée deux fois. Ce comportement n'est pas correct, car lorsque `zip` est notifiée pour la première fois, seulement la moitié des données nécessaires au nouveau calcul est disponible. `zip` calcule donc une valeur incohérente et la propage à `d`. Dans [Jou+18] une correction consistant à ajouter une option aux opérations binaires a été proposée. Cette option permet d'ignorer les notifications venant d'une entrée. C'est la solution qui a été retenue pour l'implémentation AOF que nous utilisons actuellement. Cependant, cette solution n'est pas satisfaisante. En effet, pour être correctement utilisée cette solution nécessite une compréhension précise du graphe de propagation et des algorithmes utilisés.

Une autre approche possible (présentée plus en détail dans [Jou+18]) consiste à concevoir le graphe de propagation directement et non plus à en créer un dynamiquement lors de la création des expressions. Ainsi, avec un graphe de propagation explicite il est possible de l'analyser et de proposer d'autres parcours ne posant pas de problèmes. Par exemple, un parcours en largeur du graphe de propagation est suffisant pour assurer une notification correcte des opérations.

En nous basant sur le graphe de propagation explicite présenté dans [Jou+18], nous proposons une approche complémentaire basée sur l'algèbre des processus permettant d'autres analyses.

4.2.2 Modèle relationnel des opérations actives

Les opérations actives sont des opérations (telles que `bind`, `collect`, `select`, `union` ou `zip`) sur les collections (singleton, ensembles, ordonnés ou non, listes, ordonnées ou non). Après une phase d'initialisation, assurant que les entrées et les sorties soient dans un état cohérent, les changements appliqués aux entrées sont transformés en changement sur les sorties de façon à ce que les entrées et les sorties restent cohérentes. Il est possible de construire des expressions complexes en composant ces opérations.

Par exemple, soit $a = [1, 2, 3]$ une liste d'entiers, alors l'expression suivante multiplie chaque valeur de a par 2 et sélectionne les multiples de 3. On a donc $h(a) = [6]$.

```
def : h(a) =
  let b = a->collect(e | e * 2)->filter(e | (e mod 3) = 0) in
    b
```

En nous basant sur les expressions précédentes, on peut dériver la notation relationnelle suivante.

Considérons les ensembles suivants :

— O un ensemble d'opérations (actives), telles que `collect` dans h.

— Π un ensemble de ports représentant les entrées et les sorties des opérations.

Pour un O et un Π donnés, une transformation est définie par les relations suivantes :

— $\Pi_{in} \subseteq O \times \Pi$ regroupe les ports d'entrée des opérations,

— $\Pi_{out} \subseteq O \times \Pi$ regroupe les ports de sortie des opérations,

— $L \subseteq \Pi \times \Pi$ représente les liens connectant les ports d'entrée et de sortie des opérations impliquées dans la transformation.

La condition suivante doit être vérifiée :

— $\forall (p, p') \in L, (\exists o \in O, (o, p) \in \Pi_{out}) \wedge (\exists o' \in O, (o', p') \in \Pi_{in})$ (les éléments de L sont utilisées pour connecter les sorties d'opérations aux entrées d'autres opérations.)

Une transformation est un tuple $Trans = (O, \Pi_{in}, \Pi_{out}, L)$. Cependant, ce tuple seul n'assure pas que la transformation soit correcte. C'est pourquoi une spécification des opérations existantes est nécessaire.

4.2.3 Spécification des opérations

Considérons $\Sigma = \{\sigma_1, \sigma_2, \ldots\}$ un ensemble de sortes (types) et un ensemble $P = \{p_1, p_2, \ldots\}$ de variables de ports. Pour une opération $o \in O$ donnée, considérons les deux fonctions suivantes :

— $P_{in} : O \to 2^{(P \times \Sigma)}$

— $P_{out} : O \to 2^{(P \times \Sigma)}$

Tels que $P_{in}(o)$ (respectivement $P_{out}(o)$) défini l'ensemble identifié des ports d'entrée (respectivement de sortie) de o.

Par exemple, avec l'opération `zip`, nous avons $P_{in}(\texttt{zip}) = \{\texttt{left} : l, \texttt{right} : r\}$ et $P_{out}(\texttt{zip}) = \{\texttt{out} : (l, r)\}$. Dans cette exemple et les suivants, $\texttt{left} : l$ dénote la paire contenant `left` et l. Remarquons que nous notons les pairs (p, s) du produit cartésien $(P \times \Sigma)$ par $p : s$ et que `zip` devrait normalement être défini pour tous les types de $s \in \Sigma$.

Une spécification d'opération est donc un tuple $Spec = (O, P_{in}, P_{out}, \Sigma)$.

Voici les spécifications de quelques opérations :

— `collect` : $P_{in}(\texttt{collect}) = \{\texttt{in} : i\}$, $P_{out}(\texttt{collect}) = \{\texttt{out} : o\}$

— `select` : $P_{in}(\texttt{select}) = \{\texttt{in} : i\}$, $P_{out}(\texttt{select}) = \{\texttt{out} : i\}$

— `concat` : $P_{in}(\texttt{concat}) = \{\texttt{in}_1 : s, \texttt{in}_2 : s\}$, $P_{out}(\texttt{concat}) = \{\texttt{out} : s\}$

Cette liste n'est pas complète et peut être généralisée. Soit o_1 une opération d'arité 1, elle est définie comme $P_{in}(\texttt{o}_1) = \{\texttt{in} : i\}$, $P_{out}(\texttt{out} : o)$. Pour une opération binaire o_2 nous avons $P_{in}(\texttt{o}_2) = \{\texttt{in}_1 : i_1, \texttt{in}_2 : i_2\}$, $P_{out}(\texttt{out} : o)$, et ainsi de suite pour les arités supérieures à 2.

Certaines opérations spéciales peuvent avoir des noms différents pour des raisons sémantiques comme l'opération `zip` (ici `left` et `right`).

4.2.4 Validation d'une transformation

Soit une spécification d'opérations $Spec = (O, P_{in}, P_{out}, \Sigma)$ et une transformation $Trans = (O, \Pi_{in}, \Pi_{out}, L)$, comment vérifier que $Trans$ soit valide par rapport à $Spec$? Une application de transformation de $Trans$ à $Spec$ est définie par les deux transformations suivantes :

— $\chi_{ports} : \Pi_{in} \cup \Pi_{out} \rightarrow P_{in} \cup P_{out}$

— $\chi_{sorts} : \Pi_{in} \cup \Pi_{out} \rightarrow \Sigma$

Une application de transformation $\chi_{Trans,Spec}$ est donc une paire $(\chi_{ports}, \chi_{sorts})$.

Une application de transformation est donc valide si les conditions suivantes sont vérifiées :

— $\forall o \in O, \forall p \in \Pi, (O, p) \in \Pi_{in} \Rightarrow \chi_{ports}(p) : \chi_{sorts}(p) \in P_{in}(o)$

— $\forall o \in O, \forall p \in \Pi, (O, p) \in \Pi_{out} \Rightarrow \ \chi_{ports}(p) : \chi_{sorts}(p) \in P_{out}(o)$

Ces conditions assurent seulement que les spécifications d'entrés/sortis des opérations soient respectées.

4.2.5 Notation graphique du modèle relationnel

Pour aider à la visualisation des propagations dans les expressions nous proposons une notation graphique. Le graphe de propagation est représenté par un graphe de la façon suivante :

— **Un cercle** : représente une opération, le nom de l'opération est noté dans le cercle.

— **Un point noir** : représente les entrées/sorties des opérations, les entrées sont situées sur la gauche du cercle tandis que les sorties sont sur la droite. Certaines opérations possèdent plusieurs entrées ou plusieurs sorties, donc il peut y avoir plusieurs points noirs d'un même côté. Il est possible de nommer les entrées/sorties pour lever les ambiguïtés.

— **Une flèche** : connecte une sortie d'une opération à une entrée d'une autre opération. Elle représente le flux de notification.

On notera que les opérations peuvent avoir plusieurs entrées, mais que celles-ci ne peuvent être reliées qu'à une seule sortie à la fois. Cependant, une sortie peut être connectée à plusieurs entrées.

La Figure 4.3 correspond à l'expression visible dans la sous-section 4.2.2.

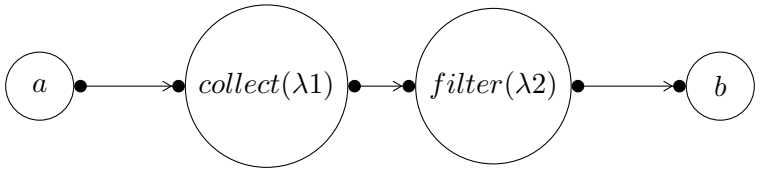


FIGURE 4.3 – Graphe de propagation correspond à $h(a)$

Le formalisme présenté plus tôt nous permet de représenter une instance de transformation ayant une spécification précise, définie par la notation relationnelle présentée dans la sous-section 4.2.3. Par exemple, l'opération `zip` peut être représentée par le graphe visible en Figure 4.4. Il faut noter que les entrées ont des types distincts, dont est déduit le type de sortie.

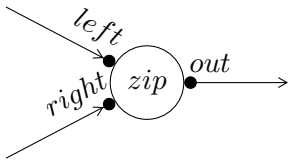


FIGURE 4.4 – Représentation graphique de l'opération `zip`.

4.2.6 Algèbre des processus

Les bases de l'algèbre des processus ont été posées par Milner [Mil80] et Hoare [Hoa78]. L'algèbre des processus forme un formalisme de raisonnement liant données et traitement. Elle est particulièrement adaptée pour représenter les processus pouvant s'exécuter en parallèle. Une fois modélisé, un processus peut être analysé et de nombreuses propriétés peuvent être dérivées. Un système est représenté par un ensemble de processus reliés entre eux par des opérateurs. Ces processus peuvent communiquer pour échanger des données et se synchroniser.

Nous rappelons ici les fondements de l'algèbre des processus nécessaires à la compréhension de la modélisation et des traitements que nous présenterons dans la suite de cette section. Pour plus de détails se référer à [BPS01]. Nous utiliserons une variante de l'algèbre des processus appelée **A**lgebra for **C**ommunicating **P**rocesses (ACP) présentée dans [BK84].

— **Actions** : nous considérons des processus composés d'actions atomiques $a, b, c, \cdots$, aussi appelées étapes (*steps*). Soit $A = \{a, b, c, \cdots\}$ un ensemble d'actions atomiques.

— **Opérateurs de base** : les actions atomiques peuvent être combinées en utilisant des opérateurs permettant ainsi de créer des processus plus complexes. L'opérateur de séquence, noté "." (peut être ignoré quand il n'y a pas d'ambiguïtés), permet de noter l'exécution séquentielle de deux actions. L'opérateur d'alternative, noté $+$, permet quant à lui d'exprimer des choix d'actions. Par exemple, $a.(b + c)$ est un processus qui exécute d'abord l'action a puis exécute b ou c et s'arrête. Ces deux opérateurs sont les blocs de bases pour d'autres opérateurs plus complexes.

— **Deadlock** : il existe une action spéciale, notée δ, appelée *deadlock*. δ représente l'échec, dès que δ est atteint le processus est bloqué. Dans l'exemple précédent, après que le

processus ait exécuté b ou c celui-ci se termine correctement. Cependant, si l'on considère $a.b.\delta$, après que a et b aient été exécutés, le processus échoue, car il reste dans un état bloqué.

— **Exécution parallèle** : il est possible d'exécuter des processus en parallèle grâce à l'opérateur de composition parallèle. On note $x \| y$ le processus commençant par x ou y et exécute ensuite l'action restante. On introduit aussi un opérateur auxiliaire, $\between$, tel que $x \between y$ se comporte comme $x \| y$, mais force l'exécution de x avant celle de y. On a donc $x \| y = x \between y + y \between x$.

Bien que suffisant pour exprimer la parallélisation simple, $\|$ ne permet pas de décrire les situations où deux processus doivent communiquer.

Pour modéliser la communication nous définissons une fonction binaire $\gamma : A \times A \to A$, telle que $\gamma(a, b)$ représente le résultat de la communication entre a et b. Si $\gamma(a, b)$ n'est pas définie alors cela signifie que les processus a et b ne sont pas autorisés à communiquer. Cette fonction permet de définir un nouvel opérateur de composition parallèle $|$ tel que $a|b = \delta$ si $\gamma(a, b)$ n'est pas définie et $a|b = \gamma(a, b)$ sinon. En utilisant ce nouvel opérateur nous avons : $x \| y = x \between y + y \between x + x|y$.

Enfin, nous utiliserons l'opération d'encapsulation notée $\partial_H, H \subseteq A$. Cet opérateur remplace toutes les actions de H par des δ. Par exemple, $\partial_H(a + b)$, avec $H = \{a, c, d\}$ est équivalent à $(\delta + b)$. Cet opérateur est utilisé pour forcer la communication entre processus. Ci-dessous, une liste des axiomes de l'ACP :

A1 $x + y = y + x$	CM4 $(x + y) \between z = x \between z + y \between z$	D1 $\partial_H(a) = a$ if $a \notin H$
A2 $(x + y) + z = x + (y + z)$	CM5 $ax\|b = (a\|b)x$	D2 $\partial_H(a) = \delta$ if $a \in H$
A3 $x + x = x$	CM6 $a\|bx = (a\|b)x$	D3 $\partial_H(x+y)=\partial_H(x)+\partial_H(y)$
A4 $(s + y)z = xz + yz$	CM7 $ax\|by = (a\|b)(x\|y)$	D4 $\partial_H(xy) = \partial_H(x).\partial_H(y)$
A5 $(xy)z = x(yz)$	CM8 $(x + y)\|z = x\|z + y\|z$	SC1 $x\|y = y\|x$
A6 $x + \delta = x$	CM9 $x\|(y + z) = x\|y + x\|z$	SC2 $x \| y = y \| x$
A7 $\delta x = \delta$	C1 $a\|b = b\|a$	SC3 $x\|(y\|z) = (x\|y)\|z$
CM1 $x \| y = x \between y + y \between x + x\|y$	C2 $(a\|b)\|c = a\|(b\|c)$	SC4 $(x \between y) \between z = x \between (y \| z)$
CM2 $a \between x = ax$	C3 $\delta\|a = \delta$	SC5 $(x\|ay) \between z = x\|(ay \between z)$
CM3 $ax \between y = a(x \| y)$	HA $x\|y\|z = \delta$	SC6 $x \| (y \| z) = (x \| y) \| z$

4.2.7 Traduction du graphe de propagation en algèbre des processus

Pour générer une formule ACP correspondante à un graphe de propagation, nous devons premièrement enrichir le modèle relationnel.

Isolation des sous-graphes de propagation

Premièrement, nous définissons $Start = \{o \in O | P_{in}(o) = \emptyset\}$ comme l'ensemble des opérations d'entrée d'une transformation.

Pour générer une formule ACP équivalente à la transformation, il faut, dans un premier temps, séparer le graphe de propagation en plusieurs sous-graphes, un pour chaque point d'entrée de la transformation. Pour cela, nous introduisons la notion de chemin (*path*) entre deux opérations.

Définition 4.2.1. *Il existe un chemin entre deux opérations a et b, noté $Path(a, b)$ si une de ces conditions est vérifiée :*

— $\exists (p, p') \in L, (\exists (a, p) \in \Pi_{out}) \wedge (\exists (b, p') \in \Pi_{in})$

— $\exists o \in O, Path(a, o) \wedge Path(o, b)$

Définition 4.2.2. *Le sous-graphe de propagation d'une transformation $t = \{O, \Pi_{in}, \Pi_{out}, L\}$ pour un point d'entrée s, $PTrans(t, s) = \{O', \Pi'_{in}, \Pi'_{out}, L'\}$, est dérivé de t par :*

— $O' = \{s\} \cup \{o \in O | Path(s, o)\}$

— $\Pi'_{in} = \{(o, p) \in \Pi_{in} | o \in O'\}$

— $\Pi'_{out} = \{(o, p) \in \Pi_{out} | o \in O'\}$

— $L' = \{(p, p') \in L | (p \in \Pi'_{in}) \wedge (p' \in \Pi'_{out})\}$

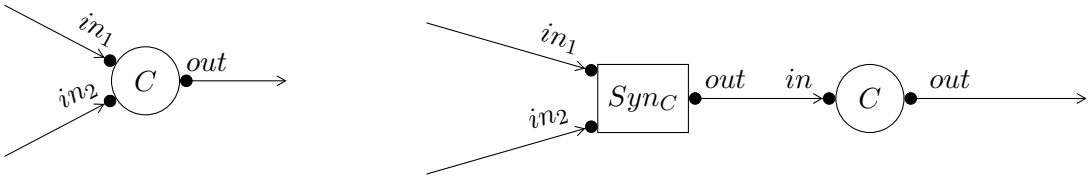
Le sous-graphe de propagation d'une entrée ne conserve que les opérations, ports et liens accessibles à partir de cette opération d'entrée. Ce sous-graphe de propagation d'un point d'entrée correspond à la partie du graphe de propagation qui peut être visitée lors de la propagation d'un changement démarrant au point d'entrée considéré.

Ajout des opérations de synchronisation

Maintenant que le graphe de propagation est séparé en plusieurs sous-graphes, un pour chaque opération d'entrée, il est nécessaire d'ajouter des opérations de synchronisation. Elles sont nécessaires quand il existe plusieurs chemins distincts entre un point d'entrée et une opération. Une opération o requiert une synchronisation si $\exists o' \in O, \exists o'' \in O, o' \neq o'', (Path(s, o') \wedge Path(o', o)) \wedge (Path(s, o'') \wedge Path(o'', o))$.

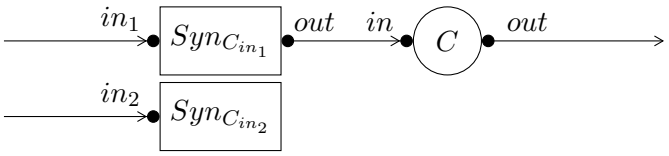
L'ajout de synchronisation se fait par l'ajout d'une opération de synchronisation spécifique avant l'opération o.

L'opération de synchronisation, notée Syn_o, prend les entrées de l'opération o, $P_{in}(Syn_o) = P_{in}(o)$ et ne possède qu'une sortie. On modifie o pour qu'elle ne prenne plus qu'une entrée, la sortie de Syn_o. Les Figures 4.5a et 4.5b illustrent cette étape. Remarque : cette transformation n'est nécessaire que pour l'analyse statique du graphe de propagation, en pratique il n'est pas nécessaire de modifier les opérations. Ainsi, il n'est pas nécessaire de savoir comment les flux d'entrée de l'opération sont fusionnés en une sortie.



(a) Avant l'ajout de synchronisations

(b) Après l'ajout de synchronisation



(c) Après la séparation des nœuds de synchronisation

FIGURE 4.5 – Représentation graphique des opérations liées à la gestion des nœuds de synchronisation

Séparation des nœuds de synchronisation

Considérons maintenant les graphes de propagation avec une seule opération d'entrée et avec des opérations de synchronisation là où elles sont nécessaires. Pour les traitements suivants, il est nécessaire de passer d'un graphe orienté acyclique (***D**irected **A**cyclic **G**raph (DAG)* en anglais) à un arbre.

Pour cela, chaque opération de synchronisation doit être séparée en plusieurs opérations, chacune n'ayant plus qu'une entrée. Soit s une opération de synchronisation.

— Pour chaque $i \in P_{in}(s)$, on créé une opération s_i avec $P_{in}(s_i) = \{i\}$ et $P_{out}(s_i) = \emptyset$. On nomme $Split(s)$ l'ensemble des opérations ajoutées lors de cette étape de séparation du nœud de synchronisation s, $Slipt(s) = \{s_i | \forall i \in P_{out}(s)\}$.

— Une des nouvelles opérations est sélectionnée, notée s', pour recevoir le port de sortie de s, $P_{out}(s') = P_{out}(s)$ et s est supprimé.

— On définit la fonction de communication γ pour les paires d'opérations de synchronisation introduites dans cette étape (c'est-à-dire, on définit $\gamma(o, o') \forall o, o' \in Split(s), o \neq o'$). Ceci permet aux opérations de synchronisation de communiquer afin de se synchroniser.

— Enfin, on note l'ensemble des opérations qui ont été ajoutées $Sync(o)$, o étant le point d'entrée de sous-graphe de propagation.

Cette étape est illustrée dans la Figure 4.5c, dans cet exemple, $Syn_{C_{in_1}}$ est l'opération qui a été sélectionnée pour recevoir le port de sortie.

Construction de la formule de l'algèbre des processus

Après l'étape de séparation des nœuds de synchronisation, le DAG est maintenant un arbre qui peut facilement être traduit en formule ACP. Avant de pouvoir définir la méthode de construction de la formule, il est nécessaire de définir la fonction $Next : O \to O$. Cette fonction retourne l'ensemble des opérations directement connectées à la sortie d'une opération. Soit o une opération. $Next(o)$ est défini par :

$$Next(o) = \{o' \in O | \forall p \in P_{out}(o) \wedge \exists p' \in P_{in}(o') \wedge (p, p') \in L\}$$

Finalement, on peut définir $R2ACP(o)$, une fonction prenant une opération o en paramètre et générant une formule ACP correspondante.

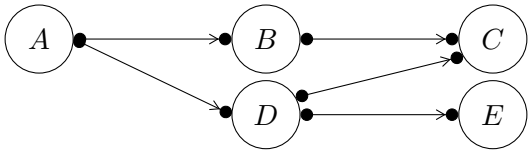
$$R2ACP(o) = \begin{cases} Next(o) = \emptyset, & o \\ Next(o) = o', & o \,.\, R2ACP(o') \\ Next(o) = \{o_1, ..., o_n\}, & o \,.\, \Big(R2ACP(o_1) \parallel ... \parallel R2ACP(o_n)\Big) \end{cases}$$

Cette fonction retourne une formule ACP correspondant à une opération et ses descendants. La formule complète d'une transformation peut donc être calculée en combinant les formules générées pour chaque opération d'entrée avec l'opérateur d'alternative. Après cette étape de traduction, il faut ajouter des opérateurs d'encapsulation pour forcer les opérations de synchronisation à communiquer et donc à se synchroniser.

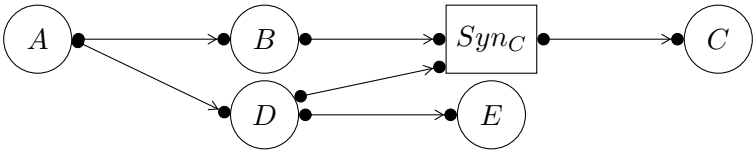
$$\sum_{o \in Start} \partial_{\{s \in Syncs(o)\}}(R2ACP(o))$$

Cette méthode peut être vue comme un parcours d'arbre avec la composition séquentielle des opérations lors de la descente dans l'arbre et la composition parallèle entre les enfants d'un nœud.

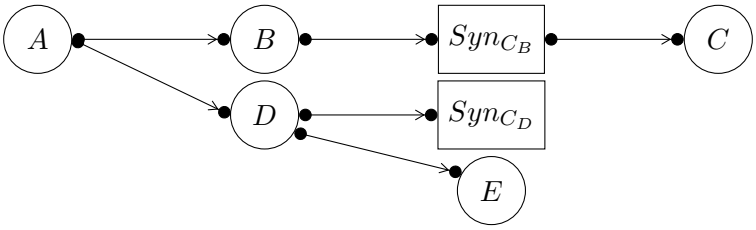
La Figure 4.6 montre un exemple complet de transformation du graphe de propagation en formule ACP. Les noms des ports ont été enlevés pour faciliter la lecture. Le graphe dans la Figure 4.6a ne possède qu'un point d'entrée, on peut donc ignorer l'étape de génération des sous-graphes de propagation. Dans la Figure 4.6b on ajoute l'opération de synchronisation $Sync_C$ avant l'opération C parce qu'elle peut être atteinte par B et D depuis A. Puis, dans la Figure 4.6c on sépare l'opération de synchronisation $Sync_C$ en deux nouvelles opérations de synchronisation $Sync_{C_B}$ et $Sync_{C_D}$. On affecte la sortie de $Sync_C$ à $Sync_{C_B}$. Enfin, dans la Figure 4.6d on applique la fonction $R2ACP$ à A pour générer la formule F. Les paires d'actions autorisées à communiquer (ici $(Sync_{C_B}, Sync_{C_D})$) sont listées sous la formule.



(a) Graphe de propagation



(b) Ajout des nœuds de synchronisation



(c) Séparation des nœuds de synchronisation

$$F = \partial_{\{Syn_{C_B},\, Syn_{C_D}\}}\Big(A \,.\, \Big((B \,.\, Syn_{C_B} \,.\, C) \parallel (D \,.\, (Syn_{C_D} \parallel E))\Big)\Big)$$

$$Syn_{C_B}|Syn_{C_D} = S$$

(d) Formule ACP correspondante

FIGURE 4.6 – Exemple de transformation de graphe de propagation en formule ACP

4.2.8 Analyses possibles

Une fois le graphe de propagation transformé en arbre, il est possible de le traduire en formule ACP. À partir de celle-ci, il est possible d'analyser la propagation.

La formule générée définit l'ensemble des ordonnancements d'opérations valides de la transformation. Il est donc possible d'utiliser la formule pour vérifier la validité des ordonnancements générés par d'autres approches (comme le parcours en largeur du graphe). Elle peut aussi servir à générer des ordonnancements valides.

Durant les différentes phases de la transformation, des opérations de synchronisation explicites sont ajoutées là où elles sont nécessaires. L'ajout de ces opérations permet l'identification et la gestion des cas où il était auparavant nécessaire de connaître la propagation pour décider quelle entrée ignorer pour les opérations problématiques. C'est donc un pas dans la bonne direction pour faciliter l'utilisation d'AOF par des personnes non expertes. De plus, l'ajout de la synchronisation permet de découpler la gestion de la synchronisation des opérations ayant

plusieurs entrées. Ainsi, `zip` et `zipWith` n'ont plus besoin de gérer d'une façon spécifique les problèmes de synchronisation qu'elles pouvaient avoir.

Avec la méthode basée sur l'ACP, les graphes de propagation sont séparés en plusieurs sous-graphes, un par entrée, et une formule est générée par sous-graphe. Ainsi, seuls les nœuds pertinents à la propagation sont intégrés à la formule. En résulte une formule spécialisée pour chaque point d'entrée de la transformation.

Contrairement au tri topologique et au parcours en profondeur (proposés dans [Jou+18]), les formules générées listent explicitement les opérations pouvant être effectuées en parallèle. Chaque $\|$ assure que les opérations peuvent être exécutées en parallèle [1]. Par exemple, avec la formule de la Figure 4.6d, il est possible d'inférer que les opérations B et D peuvent être exécutées en parallèle (c'est aussi le cas pour C et E). Exploiter ces sections parallèles ne serait pas impossible pour un algorithme centralisé chargé de contrôler l'exécution des opérations comme proposé dans [Jou+18].

Enfin, avec un graphe de transformation explicite et une formule correspondante, il est possible de détecter des motifs dans la transformation. Ces motifs pourraient être utilisés pour détecter les portions de la transformation où les caches sont inutiles. Par exemple, l'opération `zip` garde en cache ses entrées pour pouvoir facilement recalculer une sortie si une seule de ses entrées change. Mais dans certains cas, de par la configuration de la transformation, il est possible de prouver qu'il n'est pas possible d'avoir de changement sur une seule entrée. Ainsi, le cache n'est pas nécessaire, car il y a une assurance que les données seront toujours disponibles. Dans l'implémentation actuelle, il n'est pas possible de détecter ces situations (comme dans la Figure 4.1). On dit que des notifications sont alignées si un changement sur une box source entraîne des changements dans les deux entrées d'un `zip` (ou `zipWith`) et si les deux changements s'appliquent au même indice (par exemple, les deux changements ajoutent un élément à l'indice 3). Dans ce cas, l'opération n'a pas besoin de conserver un cache de ses sources, car la totalité de l'information nécessaire au calcul du résultat est disponible lorsque les deux entrées ont notifié l'opération.

4.3 Collaboration entre solveurs

Dans la sous-section 3.2.2 nous avons présenté comment il est possible d'utiliser un solveur de contraintes pour permettre l'exploration d'ensembles de modèles. Pour cela nous utilisons la variable pont qui sert de lien entre les propriétés du modèle et les variables d'un problème de contraintes. Utiliser un unique solveur peut limiter les problèmes pouvant être résolus.

En effet, la majorité des solveurs sont spécialisés sur un type de problème donné (par exemple, Choco [Pru+17] pour les problèmes discrets et Cassowary [BBS01] pour les problèmes continus linéaires). Pour résoudre certains problèmes, il est donc préférable de les séparer en sous-problèmes qui pourront être résolus indépendamment par plusieurs solveurs.

1. Si les fonction anonymes utilisées par les opérations n'ont pas d'effets de bord.

Ainsi, chaque solveur résout la partie du problème adaptée à ses capacités. De cette manière, chaque solveur résout ainsi des problèmes plus petits et donc plus simples. En effet, calculer une solution à un CSP est, dans le cas général, un problème NP-complet, réduire la taille du problème résulte donc en pratique en un temps de calcul plus faible. La collaboration entre solveurs est un sujet discuté depuis de nombreuses années [Ben96] et de nombreuses stratégies existent pour faire collaborer efficacement des solveurs. Ce problème n'est pas limité au domaine de la résolution de contraintes, par exemple, [NO79] présente un cadre permettant de combiner différentes théories.

Cependant, dans notre cas nous ne considérons que les problèmes indépendants. Considérer des problèmes indépendants facilite grandement la collaboration entre solveurs. Des problèmes sont dits indépendants s'ils ne partagent pas de variables de décision, ou dans notre cas de propriétés du modèle. Si deux problèmes sont indépendants, alors une mise à jour d'un problème n'impacte pas le second. Dans ce cas idéal, chaque solveur peut alors agir librement sans autres mécanismes de contrôle.

De façon plus générale, lorsque plusieurs problèmes ne sont pas indépendants, mais ne partagent que peu de variables entre eux il est possible de partager des propriétés entre plusieurs solveurs, mais il est alors nécessaire d'appliquer des règles strictes concernant l'accès aux propriétés.

Une solution simple de contrôle est de ne laisser l'accès en écriture à la propriété qu'à un solveur. Les autres solveurs n'ont accès à la propriété qu'en lecture (une constante du modèle CSP). Pour fonctionner correctement, cette stratégie nécessite que toutes les variables partagées entre plusieurs solveurs ne soient accessibles en écriture que par un solveur. En effet, si ce n'est pas le cas il est possible de créer des cycles entre solveurs. Modifier une propriété entraîne une mise à jour de la solution calculée par le premier solveur, qui est propagée au second solveur qui à son tour peut modifier des propriétés utilisées par le premier solveur. Ce mécanisme de contrôle d'accès aux propriétés peut être modélisé par un graphe orienté. Les nœuds correspondent aux variables et les arcs aux contraintes reliant ces variables. Un exemple avec deux problèmes non indépendants est montré en Figure 4.7. Dans cet exemple, les variables du problème 1 sont accessibles en lecture et écriture (doubles flèches). Les variables $f,\ g,\ h$ du problème 2 sont elles aussi accessibles en lecture et écriture, tandis que les variables b et e ne sont accessibles qu'en lecture pour le problème 2 (flèches simples). Cet exemple fonctionne correctement, car toutes les flèches entre le problème 1 et 2 vont dans la même direction. Il est possible d'étendre cette stratégie aux situations à plus de deux solveurs. Pour cela, il suffit de déterminer les droits d'accès aux variables de façon à ce qu'il n'y ait pas de cycles entre plusieurs sous problèmes.

La Figure 4.8 montre une variante de la Figure 3.3 adaptée au cas avec deux solveurs. Le solveur 1 a accès en écriture aux variables partagées, c'est pourquoi il est notifié des changements en premiers. Puis, les nouvelles valeurs sont propagées au solveur 2 comme des mises à jour de constantes. Le solveur 2 peut alors recalculer une solution.

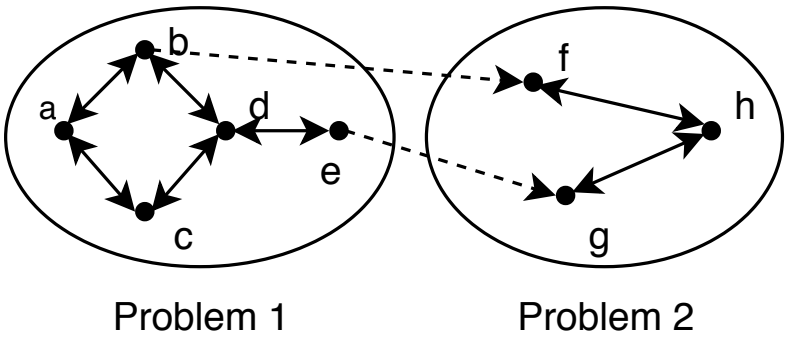


FIGURE 4.7 – Problèmes partageant deux variables.

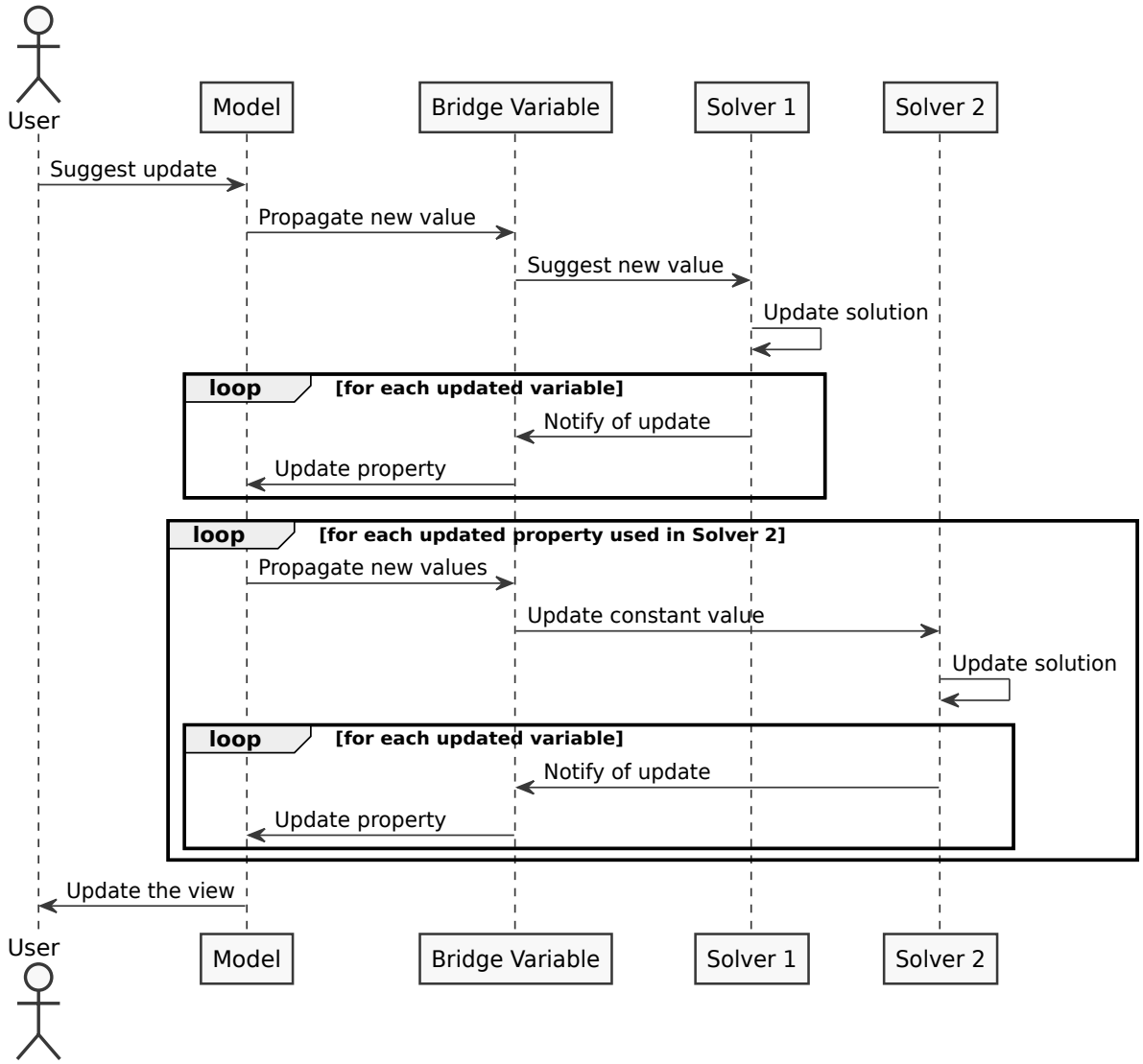


FIGURE 4.8 – Mise à jour d'une propriété dans un scénario avec plusieurs solveurs

CHAPITRE 5

ATL^C : UN LANGAGE POUR GÉNÉRER DES MODÈLES EXPLORABLES

5.1 Une spécification d'ATL^C

Dans le chapitre 3 nous avons présenté une approche permettant d'explorer des ensembles de modèles grâce à l'utilisation de contraintes. Ces contraintes utilisent des variables ponts pour assurer une synchronisation permanente entre les propriétés du modèle et les variables de décision correspondantes dans le solveur. Ce système permet l'exploration d'ensembles de modèles définis en intension par des contraintes.

Seulement, pour fonctionner cette approche nécessite qu'un ensemble de structures soient présentes et correctement initialisées. Cette mise ne place est complexe et fastidieuse. Il est donc nécessaire de proposer un moyen pour automatiser la mise en place de ces structures.

5.1.1 Aperçu de l'approche

Les approches classiques de transformation de modèles ne génèrent habituellement qu'un seul modèle pour une source donnée. Cependant, dans de nombreuses transformations il existe plus d'un modèle cible correspondant à une source. La Figure 5.1 montre un aperçu de notre approche basée sur les ensembles de modèles. Les deux ellipses aux bords pleins représentent les espaces de modèles sources et cibles. Les points représentent chacun un modèle particulier appartenant aux $M_\bullet$ des espaces correspondants. L'ellipse avec un contour pointillé représente l'ensemble des modèles cibles généré par notre approche. Avec notre approche, à partir d'un unique modèle source, nous générons un modèle cible ainsi que des contraintes. Les deux, combinés avec un solveur, permettent la définition d'un ensemble de modèles. Tandis que les approches classiques ne génèrent qu'un unique modèle cible à partir d'une source.

La Figure 5.2 détaille grossièrement les étapes que nous avons mises en place pour générer automatiquement les structures nécessaires à l'exploration d'ensembles de modèles. Dans un premier temps, le modèle source est donné à une transformation qui génère le modèle cible ainsi que les contraintes qui s'y appliquent. De plus, les liens de synchronisation entre le modèle cible et le modèle de contraintes sont mis en place. Ce lien correspond à la notion de *variable pont* présentée dans la sous-section 3.2.2. Ensuite, le modèle de contraintes généré

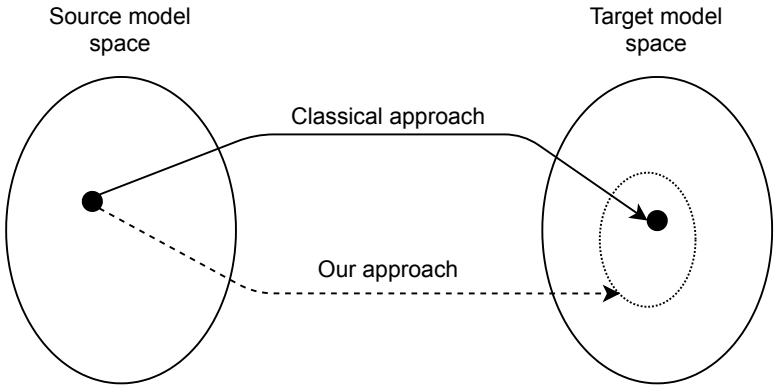


FIGURE 5.1 – Différence entre notre approche et les approches classiques

est traité afin d'affecter les contraintes aux bons solveurs. Pour pouvoir connecter des solveurs à notre modèle de contraintes, nous avons mis en place un système de *wrapper* autour des solveurs. Les *wrappers* prennent le modèle de contraintes et le transforment en contraintes adaptées au solveur. Dans le *wrapper* présenté plus tôt cette transformation nécessite plusieurs étapes qui peuvent ne pas être nécessaires pour d'autres solveurs. Enfin, une fois ces transformations effectuées, synchronisations entre les variables de décision des solveurs et les variables du modèle de contraintes sont mises en place. Ces différentes transformations seront présentées plus en détail dans la section 5.3.

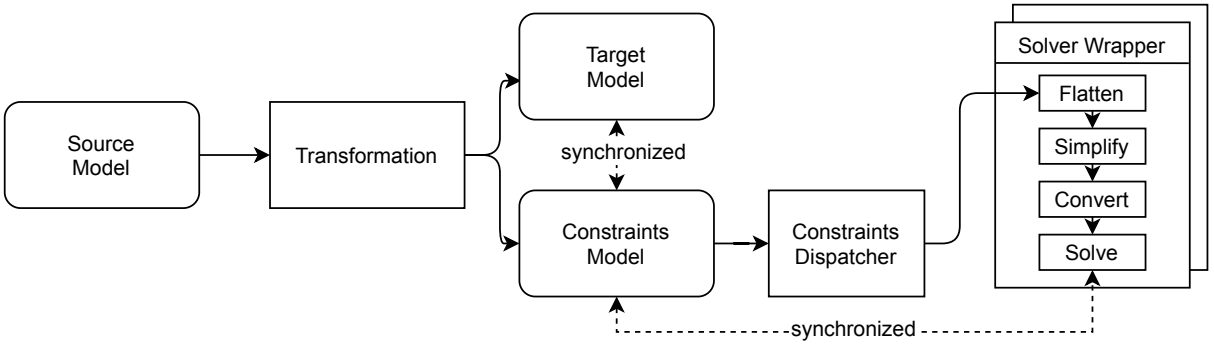


FIGURE 5.2 – Structure générale de l'approche de génération automatique d'ensembles de modèles explorables

Comme vu plus tôt, dans notre approche, les contraintes sont utilisées pour définir des ensembles de modèles en intention. Combiner l'étape de spécification de la transformation et de déclaration des contraintes permet de créer un langage unifié. En une étape, le développeur peut écrire une transformation décrivant comment créer le modèle de sortie ainsi que les règles nécessaires à la validation de ce modèle. Donnons donc une spécification du langage de transformation pouvant répondre à nos besoins.

5.1.2 Prérequis du langage

À partir des propriétés que nous avons présenté dans les sections précédentes, nous pouvons donner la liste de spécifications suivantes pour notre langage :

Prérequis 1 : Langage déclaratif à base de règles. Les langages de contraintes classiques sont séparés en deux grandes familles, les langages dédiés [BLP16 ; Net+07] et les bibliothèques [BBS01 ; Lau18 ; Pru+17]. Qu'ils soient dédiés ou intégrés comme des bibliothèques dans des langages impératifs, les langages de modélisation de contraintes sont déclaratifs. L'utilisateur spécifie un ensemble de contraintes qu'un solveur tente de résoudre. De même, il existe une multitude de langages de transformation, dont une partie est déclarative [JB16 ; JK06 ; Leb+14 ; Var+16] et utilise le concept de règles. Une règle de transformation est une construction, le plus souvent déclarative, encapsulant :

(a) les éléments sources ainsi que des gardes optionnelles servant de prédicats pour vérifier l'applicabilité de la règle,

(b) les éléments cibles qui seront créés par la règle,

(c) les *bindings* permettant de calculer les propriétés des éléments cibles à partir des propriétés des éléments sources.

Dans notre cas, le concept de règles est très intéressant, en effet, c'est un moyen naturel d'encapsuler un (ou plusieurs) élément(s) source(s) avec un (ou plusieurs) élément(s) cible(s) ainsi que les contraintes les concernant.

Prérequis 2 : Déclaration des contraintes. Pour efficacement définir des ensembles de modèles, il est nécessaire que le développeur soit libre de définir des contraintes s'appliquant aux propriétés des éléments de sortie. Selon le type d'application, les contraintes ne seront pas exprimées de la même façon. Par exemple, pour des problèmes de visualisation utilisant JavaFX, il est naturel d'utiliser des contraintes géométriques (par exemple, qu'un élément doit être contenu dans un autre, qu'un élément soit au-dessus d'un autre, ou que deux éléments soient toujours espacés d'un nombre fixe d'unités). Pour d'autres problèmes, il peut être plus simple de représenter le problème par un ensemble d'inéquations et une fonction objectif. Ainsi, suivant le problème, de nombreuses modélisations peuvent être utilisées. Idéalement, il faudrait que le langage de contraintes proposé au développeur permette de représenter toutes ces modélisations.

Prérequis 3 : Définition du comportement de réparation. Dans la section 3.3 nous avons présenté la notion de comportement et de stratégie de réparation. Nous avons vu que cette notion de stratégie de réparation est essentielle pour permettre une bonne exploration de l'ensemble des modèles générés. En effet, sans contrôle, le solveur peut réparer le modèle de façons non souhaitables (par exemple, en modifiant les positions de tous les éléments graphiques affichés). C'est pourquoi il est nécessaire que le langage de contraintes permette de définir avec précision comment le solveur peut réparer une solution non valide. Par exemple,

ceci peut être implémenté en utilisant le concept de priorités utilisé dans les **H**ierarchical **C**onstraint **S**olving **P**roblem.

Prérequis 4 : Transformation incrémentale. Idéalement, la transformation devrait être incrémentale. C'est-à-dire qu'un changement sur le modèle source (par exemple, l'ajout d'un nouvel élément) doit être propagé sur la cible sans tout recalculer. En effet, plaçons-nous encore dans le cas où la cible est une vue présentée à l'utilisateur, il est préférable de ne pas changer des éléments qui n'ont pas besoin de l'être. De plus, l'incrémentalité de la transformation permet de visualiser des modèles évoluant dans le temps et enrichie les interactions possibles. Par exemple, l'utilisateur peut ajouter de nouveaux éléments au modèle source ce qui entraîne la mise à jour de la vue pour ajouter les nouveaux éléments correspondants.

Prérequis 5 : Collaboration entre solveurs. Chaque solveur est spécialisé dans la résolution de certaines contraintes. Il pourrait être intéressant de permettre à l'utilisateur d'utiliser plusieurs solveurs pour résoudre différentes parties du problème de contraintes. Par exemple, résoudre les *bindings* simples par de la transformation de modèles classique, les relations plus complexes sur des nombres réels par un solveur de contraintes et les contraintes sur les entiers par un autre solveur de contraintes. Le formalisme basé sur l'algèbre des processus présentés dans le chapitre 4 pourrait servir de base pour assurer une exécution correcte dans un environnement composé de plusieurs solveurs agissant indépendamment.

En considérant ces prérequis et le fait que nous avons travaillé sur un moteur d'exécution de transformations ATL incrémentales nous avons décidé d'étendre ATL pour supporter les contraintes nécessaires à la définition d'ensembles de modèles.

On notera que certains choix présentés plus haut ont été influencés par nos cas d'étude principalement graphiques (présentés en détail dans le chapitre 7). Par exemple, la flexibilité dans la modélisation des contraintes ou l'incrémentalité de la transformation ne sont pas nécessaires pour toutes les applications.

5.2 Un métamodèle des contraintes

La variable pont permet de synchroniser une propriété du modèle avec une variable de décision d'un solveur. Cependant, seule elle n'est pas utile, dans cette section nous présenterons l'ensemble du modèle de contraintes que nous utilisons pour explorer des ensembles de modèles. Il existe de nombreux langages de modélisation de contraintes, qu'ils soient intégrés dans un autre langage par l'utilisation d'une bibliothèque (par exemple, Choco [Pru+17], Cassowary [BBS01] ou Alloy [Jac02]) ou dans un langage dédié (par exemple, XCSP3 [BLP16], MiniZinc [Net+07] ou AMPL [FGK90]). MiniZinc est un langage de modélisation de haut niveau supportant de nombreux types de données et contraintes. Ces contraintes de haut niveaux sont compilées en FlatZinc, un autre langage de modélisation de contraintes de plus bas niveau, compris par de nombreux solveurs (comme Gecode [Tea05], Choco [Pru+17],

ECLiPSe [WNS97] ou OR-Tools [Tea10]). **A M**athematical **P**rogramming **L**anguage (AMPL) est un autre langage de modélisation dédié pouvant être utilisé par de nombreux solveurs (comme CPLEX, Gecode ou Gurobi).

Bien qu'adaptés à la modélisation, ces langages s'intègrent mal aux technologies de l'IDM (par exemple EMF) et ne permettent pas de représenter nos variables ponts. C'est pourquoi nous avons créé notre propre métamodèle de contraintes, inspiré des métamodèles existant. De plus, ce métamodèle intermédiaire nous sert de métamodèle pivot qui est ensuite transformé en modèles adaptés à chaque solveur.

Notre métamodèle est construit autour de la variable pont et de l'idée qu'il doit être assez générique pour pouvoir représenter autant de domaines que possible. La Figure 5.3 présente une version simplifiée de ce métamodèle [1]. La `BridgeVariable` est responsable de la synchronisation entre une propriété du modèle et une variable de décision du solveur, c'est pour cela qu'elle possède deux attributs y faisant référence. Pour rester le plus générique possible, la classe `CompositeExpression` n'est composée que d'un opérateur et d'une liste d'arguments. C'est lors de la traduction du modèle de contraintes en contraintes spécifiques au solveur [2] que le nombre d'arguments des opérateurs est vérifié. Ainsi, un même opérateur peut être traduit par différentes contraintes suivant les capacités du solveur utilisé. Par exemple, un solveur géométrique pourrait traiter directement des opérateurs comme `auDessus` ou `contient` tandis qu'il faudrait les traduire en expressions arithmétiques pour un solveur classique. La `Constraint` est abstraite et deux classes concrètes en héritent. La `SimpleConstraint` représente les contraintes classiques, avec un opérateur (ou prédicat), des arguments, un poids et une force. La `ConstraintsGroup` représente un groupe de `Constraint`s et possède une propriété `solveur` permettant de déterminer le solveur devant utiliser les `Constraint`s qu'elle possède.

Pour être utilisées par les solveurs, les contraintes de notre métamodèle intermédiaire doivent être transformées en contraintes du solveur cible. Cette transformation peut nécessiter plusieurs transformations chacune modifiant le modèle de contraintes. Une de ces transformations peut réécrire des contraintes non supportées par le solveur cible en d'autres contraintes qui le sont. Par exemple, on peut convertir les contraintes géométriques (comme `contient` ou `auDessus`) en contraintes arithmétiques sur les coordonnées des formes géométriques. C'est aussi à ce moment que l'on va réécrire les contraintes impliquant des relations pour les adapter aux solveurs (voir sous-section 3.2.4). De plus, l'utilisation de ce métamodèle intermédiaire nous permet de facilement ajouter de nouveaux solveurs cibles.

Dans le métamodèle de contraintes, Figure 5.3, on remarque que les `Constraint`s possèdent un `weight` (poids) et une `strength` (force ou importance). Ces attributs sont utilisés

1. Le métamodèle utilisée dans le prototype d'ATL^C n'utilise pas tout à fait ce métamodèle pour des raisons historiques et de changement empiriques lors du développement.

2. On utilisera le terme de *contraintes spéficiques* pour faire référence aux modèles de contraintes utilisées par les différents solveurs de contraintes. Le terme *modèle de contraintes* seul fera référence au métamodèle contraintes présenté dans cette section.

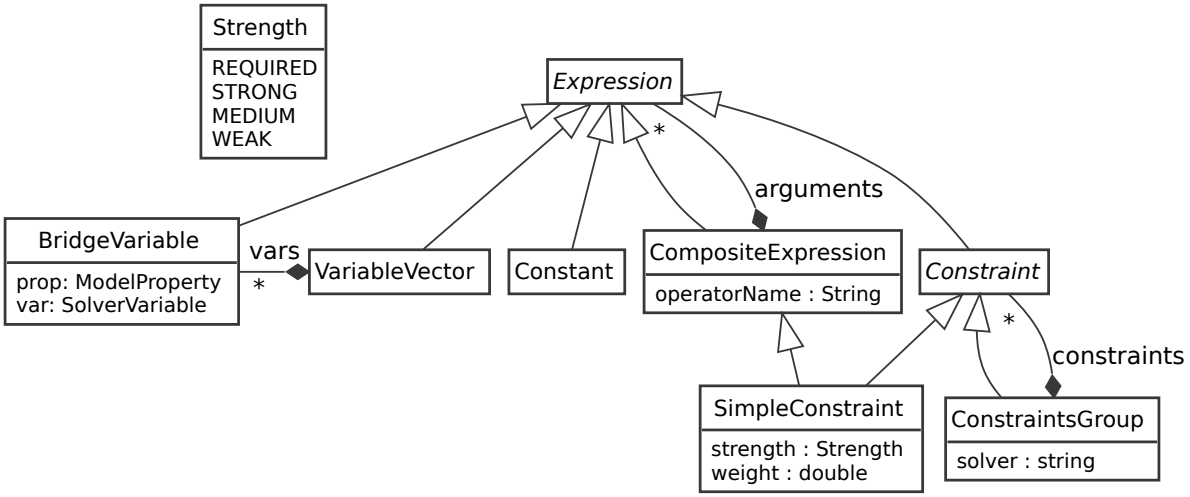


FIGURE 5.3 – Métamodèle de contraintes utilisé

pour représenter des problèmes comme les HCSP ou les WCSP. Si le solveur cible ne permet pas de modéliser ces problèmes, il est possible de réécrire une partie des contraintes de façon à transformer ces problèmes comme des problèmes d'optimisation (COP).

Enfin, la classe `VariableVector` permet d'utiliser des collections de variables. Ces `VariableVector`s regroupent plusieurs variables dans un même conteneur pouvant être utilisé comme une variable classique. Une contrainte contenant un `VariableVector` peut donc correspondre à plusieurs contraintes pour le solveur. Par exemple, imaginons la contrainte suivante :

$$\{a, b, c\} + 3 < y$$

Nous avons quatre variables a, b, c et y. De plus, les variables a, b et c sont regroupées dans un unique `VariableVector`. Une fois traduite, cette contrainte laisse place à trois nouvelles contraintes :

$$a + 3 < y$$
$$b + 3 < y$$
$$c + 3 < y$$

Avoir un métamodèle pour représenter nos contraintes nous permet d'exprimer cette série de réécriture de contraintes comme une série de transformations de modèles endogènes. Ainsi, grâce à la séparation de ces étapes en plusieurs transformations simples il est possible de les réutiliser pour plusieurs solveurs. Par exemple, l'étape de mise à plat des `VariableVector`s est commune à tous les solveurs.

Pour fonctionner de façon optimale avec notre approche, un solveur doit présenter plusieurs

caractéristiques. L'absence de ces caractéristiques n'empêche pas l'utilisation du solveur par notre approche, mais entraîne des complications au niveau du *wrapper* et potentiellement des problèmes de performances pour le solveur. Voici une liste de ces caractéristiques :

Capacité 1 : Ajout de variables et contraintes. Lorsque l'utilisateur modifie une propriété p du modèle, la nouvelle valeur val est propagée au solveur. Cette propagation entraîne la suggestion d'une nouvelle valeur pour la variable var correspondante. En pratique, cette suggestion correspond à l'introduction d'une contrainte $var = val$.

Capacité 2 : Suppression de variables et contraintes. De même, lorsque l'utilisateur propose une nouvelle valeur, il est nécessaire de supprimer les propositions précédentes si elles existent. Sans cela le problème de contraintes devient nécessairement incohérent.

Capacité 3 : Limitation des changements de valeurs des variables. Les techniques de résolution des solveurs ne sont pas toutes déterministes. C'est-à-dire qu'il n'y a pas toujours de garantie que pour un même problème la même solution soit retournée. Ce comportement n'est pas désirable dans le cas où l'on souhaite éviter les changements entre plusieurs solutions calculées successivement. Par exemple, lors que l'on considère un modèle présenté à l'utilisateur, il est préférable de ne pas modifier toutes les positions à chaque suggestion de changement par l'utilisateur. Il est donc préférable que le solveur propose un système permettant d'indiquer que certaines variables ne doivent, si possible, pas être modifiées entre deux solutions successives.

Capacité 4 : Résolution de problèmes d'optimisations. De plus, il est préférable que le solveur puisse résoudre des problèmes d'optimisation. Lorsque l'utilisateur suggère une valeur à une variable, le problème devient naturellement un problème d'optimisation. Le solveur cherche une solution vérifiant les contraintes tout en prenant en compte les suggestions de l'utilisateur et les variables devant garder leurs valeurs fixes.

Cependant, tous les solveurs ne présentent pas ces caractéristiques. Heureusement, il est souvent possible de réécrire les problèmes de contraintes de façon à contourner ces limitations. Par exemple, si un solveur ne permet pas l'ajout ou la suppression de contraintes il est possible de passer outre en créant et résolvant un nouveau problème de contraintes. Il faudra cependant ajouter des contraintes pour assurer que les anciennes valeurs des variables soient conservées. De plus, il est facile d'imaginer que résoudre de nouveaux problèmes à chaque suggestion de l'utilisateur peut être extrêmement coûteux. C'est pourquoi, bien qu'il soit possible de passer outre les limitations de certains solveurs, il est préférable d'utiliser des solveurs possédant les capacités listées plus haut.

Comme précisé dans la Capacité 3, lorsque l'on considère un problème avec lequel l'utilisateur peut interagir par le biais de suggestion de valeur pour les variables il est nécessaire de considérer un système où le solveur va recevoir des suggestions de modifications qu'il prendra en compte pour le calcul d'une nouvelle solution et ainsi de suite. Cependant, dans un tel système il est préférable de mettre en place des mécanismes pour que le solveur prenne

en compte qu'il est préférable que certaines variables ne changent pas de valeur. En effet, si l'utilisateur suggère une valeur pour une variable puis suggère une autre valeur pour une autre variable il est préférable que le solveur se souvienne du choix de l'utilisateur sur la première variable pour ne pas la remplacer par une valeur totalement différente. Dans un contexte de problème géométrique ceci correspond au fait qu'il est préférable que les formes géométriques conservent, tant que possible, leurs positions et dimensions. Un diagramme dont les positions des éléments changent à chaque interaction avec l'utilisateur n'est pas agréable à utiliser.

Certains solveurs comme Cassowary [BBS01] ont une contrainte spécifique, `stay`. Cette contrainte force une variable à conserver une valeur proche de celle qu'elle avait dans la solution précédente. Pour Cassowary, cela revient à minimiser une erreur e avec $e = |v - v'|$, où v l'ancienne valeur de la variable et v' la nouvelle valeur. Si cette contrainte `stay` n'est pas supportée par un solveur il est tout à fait possible de réécrire la contrainte en une contrainte similaire à celle utilisée par Cassowary. Le solveur n'a donc pas besoin d'explicitement supporter la contrainte `stay`, il faut cependant que le solveur puisse résoudre des problèmes d'optimisations. En effet, comme Cassowary, pour assurer que la solution trouvée soit la plus *proche* de la solution précédente (en ne prenant en compte que les variables sur lesquels l'utilisateur a appliqué la contrainte `stay`), le solveur cherche à optimiser la somme (potentiellement pondérée) des erreurs.

Grâce à ces réécritures, il est possible de passer outre certaines des capacités nécessaires à un solveur optimal pour notre approche. Cependant, ces réécritures peuvent être coûteuses et nécessitent le plus souvent de recalculer des solutions entières, là où des solveurs spécialisés sont capables de calculer une nouvelle solution à partir d'une solution existante.

5.3 Du modèle intermédiaire aux solveurs de contraintes

Nous avons, dans la section 5.2, introduit un métamodèle de contraintes adapté à notre approche. Celui-ci intègre la variable pont présentée en sous-section 3.2.2 tout en permettant de modéliser de nombreux types de problèmes. Cependant, ce métamodèle intermédiaire ne peut pas être utilisé directement dans les solveurs cibles. En effet, ceux-ci ont leurs propres métamodèles qu'il faut utiliser pour pouvoir interagir avec ces solveurs. Dans la Figure 5.2 nous avons montré les différentes étapes de notre approche. Dans cette section nous nous attarderons sur les *wrappers* mis en place autour des solveurs ainsi que les transformations (ou réécritures de contraintes) qui les composent.

5.3.1 Mise à plat des contraintes

Dans le métamodèle de contraintes présenté en Figure 5.3 on remarque que la `Constraint` est abstraite et que deux classes concrètes en héritent, `SimpleConstraint` et `ConstraintsGroup`. La première, `SimpleConstraint`, représente une contrainte classique, avec son opé-

rateur, ses arguments, son poids et sa force. La seconde, `ConstraintsGroup`, regroupe plusieurs `SimpleConstraint`s en une seule contrainte. De plus, elle possède un attribut `solveur` utilisé lors de la sélection du solveur cible pour déterminer pour quel solveur sont destinées les contraintes le contenant.

Cette classe `ConstraintsGroup` permet de regrouper toutes les contraintes d'une règle en un seul objet et de faciliter la rédaction des règles de transformation.

Cependant, les étapes suivantes de la transformation attendent une liste de `SimpleConstraint`s et non une liste de `ConstraintsGroup`s contenant les `SimpleConstraint`s. Le but de cette étape de *mise à plat* est de transformer cette imbrication de `ConstraintsGroup`s et `SimpleConstraint`s en une liste de `SimpleConstraint`s. Cette étape est similaire à l'opération `flatten :: Tree a -> [a]` en Haskell, mais avec une profondeur de mise à plat arbitrairement grande.

5.3.2 Expansion des contraintes

Une fois le modèle de contraintes débarrassé des `ConstraintsGroup`s il ne reste plus que des `SimpleConstraint`s . Cependant, ces `SimpleConstraint`s peuvent contenir des `BridgeVariable`s, mais aussi des `VariableVector`s. Peu de solveurs supportent les collections de variables, il est donc nécessaire d'appliquer l'étape d'*expansion des contraintes*. Celle-ci consiste à réécrire toutes les contraintes contenant des `VariableVector`s en contraintes ne contenant plus que des `BridgeVariable`s.

Par exemple, reprenons l'exemple utilisé dans la section 5.2 :

$$\{a, b, c\} + 3 < y$$

Dans cet exemple, les variables a, b et c sont regroupées dans un `VariableVector` (représenté par des $\{\}$). Pour être utilisée par un solveur classique il faut éclater la `VariableVector`. Pour cette contrainte, cela génère les trois contraintes suivantes :

$$a + 3 < y$$
$$b + 3 < y$$
$$c + 3 < y$$

L'expansion de contraintes est assez simple lorsque les contraintes ne possèdent qu'un `VariableVector`, il suffit de générer une nouvelle contrainte par variable du `VariableVector`. Nous avons donc, pour $vs = \{v_1, v_2, \cdots, v_n\}$ un `VariableVector` et la contrainte suivante :

$$vs \star e \sim e'$$

Avec $\star$ un opérateur binaire, e et e' deux expressions quelconques ne contenant pas de `Va-`

`riableVector` et $\sim$ un opérateur de comparaison binaire. L'expansion de cette contrainte est la suivante :

$$\bigwedge_{v \in vs} v \star e \sim e'$$

Cependant, ceci n'est qu'un cas particulier, dans le cas général il faut considérer les contraintes possédant plusieurs `VariableVector`s. Considérons la contrainte suivante :

$$\{a, b, c\} * -\{x, y\} < 0$$

La Figure 5.4 donne le diagramme d'objets de la contrainte précédente. Par souci de simplicité, certaines propriétés non importantes pour l'expansion ont été ignorées.

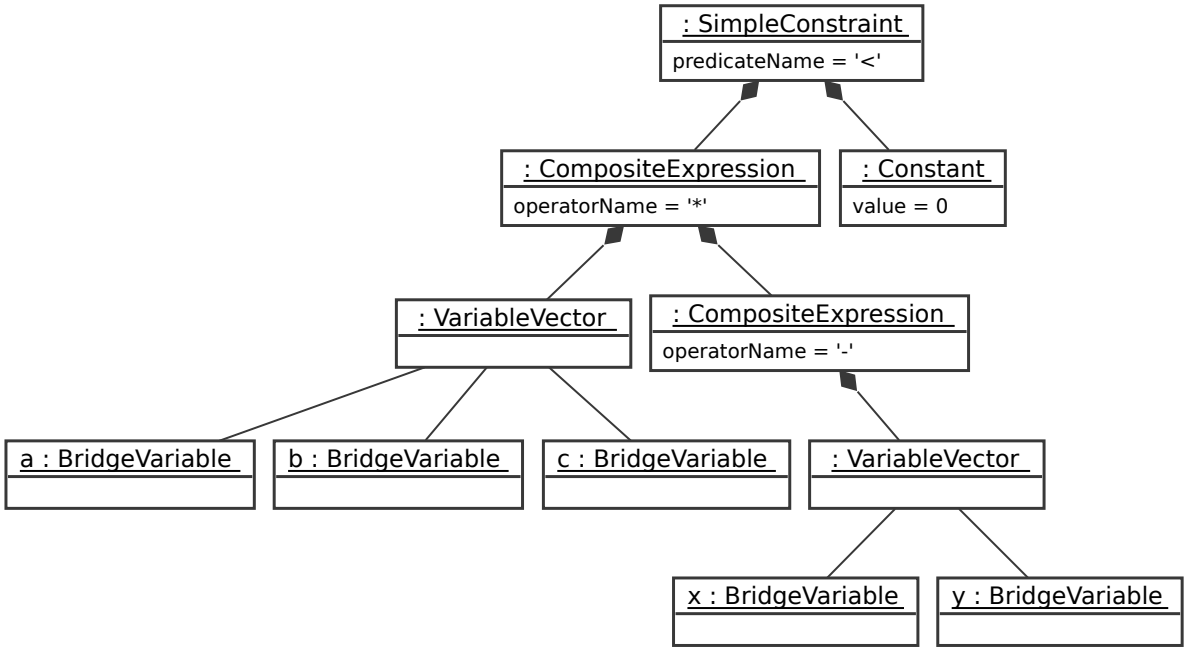


FIGURE 5.4 – Diagramme d'objets correspondant à la contrainte $\{a, b, c\} * -\{x, y\} < 0$

L'expansion d'une contrainte se fait de façon récursive en utilisant l'Algorithme 3. Dans cet algorithme :

— Les listes ordonnées (vecteurs) sont notées entre $\langle\rangle$: $L \leftarrow \langle 35, 76, 42\rangle$ représente une liste L contenant les entiers $35, 76, 42$.

— L'accès à l'élément à l'indice i d'une liste est noté par les $[]$: $L[2]$ correspond à 76.

— La concaténation de deux listes se note $\oplus$: $L \oplus L$ correspond à $\langle 35, 76, 42, 35, 76, 42\rangle$.

— L'instanciation d'objet est notée par le nom de la classe suivant, entre $\{\}$, des propriétés de l'objet : `Constant`$\{valeur : 0\}$ créer une nouvelle instance de la classe `Constant` avec la propriété valeur à 0.

Celui-ci est composé d'une seule fonction, `Expansion` prenant une `Expression` E en paramètre et retournant une liste d'`Expression`s. Les `Constante`s et `BridgeVariable`s sont déjà des éléments simples et donc ne nécessitent pas de traitement particulier. Ces éléments sont empaquetés dans une liste et retournés (ligne 3). Le traitement est similaire pour les `VariableVector`s, la fonction retourne les `BridgeVariable`s contenues dans le `VariableVector` (ligne 5). Le traitement des `CompositeExpression`s est plus compliqué, celles-ci possèdent plusieurs arguments, il faut donc les combiner. Dans un premier temps, nous calculons l'expansion de tous les arguments de la `CompositeExpression` (ligne 8). Ceci est fait en appliquant récursivement la fonction `Expansion` aux `Expression`s arguments de la `CompositeExpression`. Puis nous générons les nouvelles expressions, pour cela, nous proposons deux stratégies de combinaison. La première (lignes 9 à 13) correspond à un produit cartésien des arguments. La seconde (lignes 13 à 18) correspond à une version étendue du produit de matrices d'Hadamard [Hal47 ; Sty73].

Algorithme 3 Algorithme d'expansion

```
1 : fonction EXPANSION(E : Expression)
2 :     si E est une Constant ou une BridgeVariable alors
3 :         retourne ⟨E⟩
4 :     sinon si E est un VariableVector alors
5 :         retourne E.variables
6 :     sinon                                   # E est une CompositeExpression
7 :         ret ← ⟨⟩
8 :         args ← E.arguments.map(EXPANSION)   # Applique Expansion aux arguments
9 :         si E est une opération cartésienne alors
10 :            pour (a_1, a_2, ··· , a_n) ∈ args[1] × args[2] × ··· × args[n] faire
11 :                ret ← ret ⊕ ⟨CompositeExpression{
                                operatorName : E.operatorName,
                                arguments : ⟨a_1, a_2, ··· , a_n⟩}⟩
12 :            fin pour
13 :        sinon
14 :            m ← args.map(taille).min        # Taille du plus petit argument après expansion
15 :            pour 1 ≤ i ≤ m faire
16 :                ret ← ret ⊕ ⟨CompositeExpression{
                                operatorName : R.operatorName,
                                arguments : ⟨args[1][i], args[2][i], ··· , args[n][i]⟩}⟩
17 :            fin pour
18 :        fin si
19 :        retourne ret
20 :    fin si
21 : fin fonction
```

Par exemple, en appliquant l'Algorithme 3 à la contrainte $\{a, b, c\} * -\{x, y\} < 0$ en sélec-

tionnant l'expansion scalaire nous obtenons la contrainte suivante :

$$a * -x < 0$$

Détaillons le processus menant à cette contrainte. Dans un premier temps, appliquons l'algorithme d'expansion à la `CompositeExpression` ayant l'opérateur $-$. C'est une opération unaire donc la méthode d'expansion n'est pas importante, une nouvelle `CompositeExpression` est généré pour chaque `Expression` générée par `Expansion` pour l'argument de la `CompositeExpression`. Il y a donc deux `CompositeExpression`s générées, une ayant x en variable et l'autre ayant y.

Ensuite, considérons la `CompositeExpression` avec l'opérateur $*$. L'expansion de l'`Expression` à droite retourne trois nouvelles `Expression`s, une pour a, b et c. Tandis qu'à gauche, nous avons les deux `CompositeExpression`s présentées plus tôt. Lors de la combinaison des deux en mode scalaire on ne génère que deux nouvelles `Expression`s, car il n'y a que deux `Expression`s à gauche. On a donc deux `CompositeExpression`s représentant $a * -x$ et $b * -y$.

Enfin, considérons la `SimpleConstraint` ayant l'opérateur $<$. À gauche, nous avons les deux `Expression` précédentes et à droite une seule `Expression` pour la `Constant` 0. Donc, la combinaison des arguments ne donne qu'une `Expression` correspondant à $a * -x < 0$. La Figure 5.5 montre le diagramme d'objets correspondant.

En appliquant l'expansion cartésienne à la contrainte $\{a, b, c\} * -\{x, y\} < 0$, nous obtenons six contraintes :

$$\begin{array}{cc} a * -x < 0 & a * -y < 0 \\ b * -x < 0 & b * -y < 0 \\ c * -x < 0 & c * -y < 0 \end{array}$$

Deux des 6 diagrammes d'objets des contraintes sont visibles dans la Figure 5.6 ; deux autres sont visibles dans la Figure 5.5.

On notera que dans les exemples précédents nous avons considéré qu'il n'était possible de contrôler le comportement de l'expansion (scalaire ou cartésienne) qu'au niveau global. Dans la pratique, la distinction se fait au niveau de l'opérateur. Ainsi, les opérateurs classiques $+, -, *, /$ sont considérés comme des opérateurs cartésiens. La version scalaire d'un opérateur classique $\star$ est noté $.\star$, par extension, tous les opérateurs commençant par un point sont considérés comme des opérateurs scalaires. Par exemple, $+$ est un opérateur cartésien tandis que $.+$ est un opérateur scalaire.

5.3.3 Encodage des relations

C'est après la mise à plat des contraintes et l'extension des `VariableVector`s qu'il est possible d'encoder les relations en variables. Cette étape est nécessaire pour représenter les

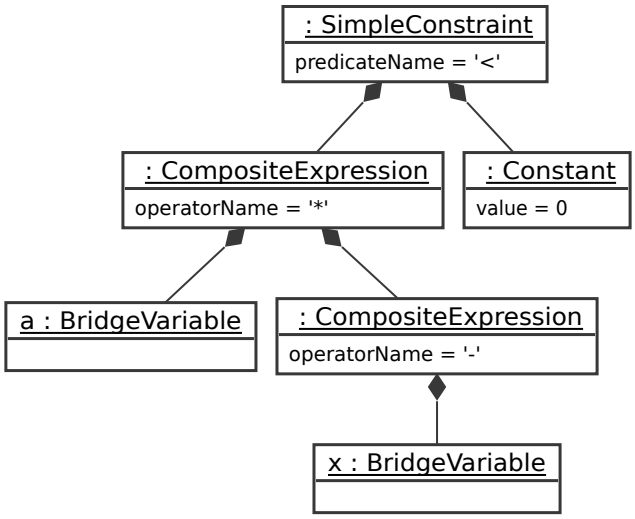


FIGURE 5.5 – Diagrammes d'objets de la contrainte générée par l'expansion scalaire

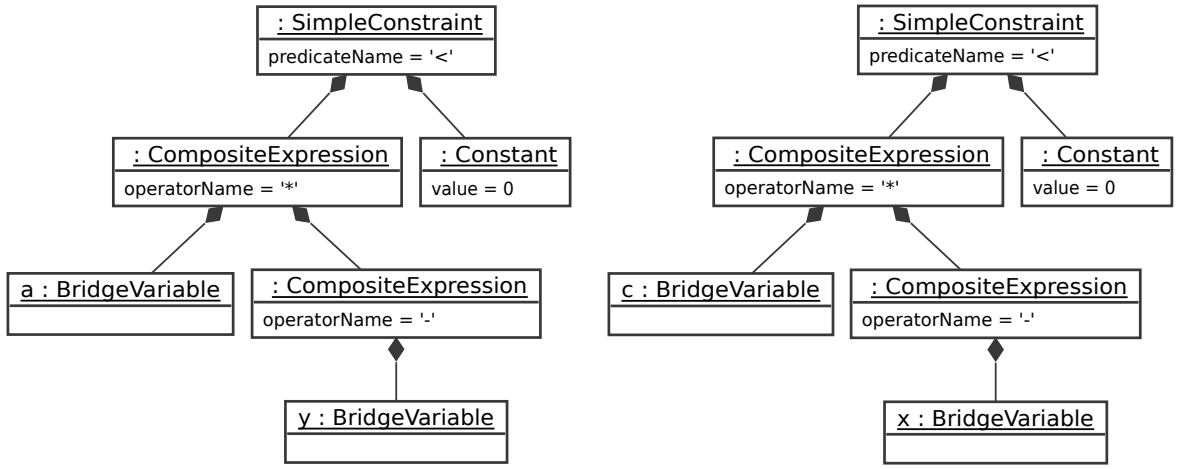
relations entre objets sous la forme d'entiers. La méthode consiste à parcourir le modèle de contraintes, comme à l'étape d'expansion, et à réécrire les contraintes portant sur des relations en contraintes sur des entiers. Cette transformation est décrite dans la sous-section 3.2.4.

Par exemple, la Figure 5.7a donne un métamodèle contenant deux classes, `A` et `B`. La Figure 5.7b donne trois objets `a`, `b1` et `b2`. Maintenant, imaginons la contrainte suivante $a.\{b\}.v = a.v$ où $\{\}$ dénote la relation variable. Soient $id(\texttt{b1})$ et $id(\texttt{b2})$ les identifiants associés aux éléments de modèle `b1` et `b2` et $a.v_{id}$ la variable recevant l'identifiant de l'élément affecté. La contrainte précédente est réécrite en l'ensemble de contraintes suivant :

$$a.v_{id} = id(\texttt{b1}) \implies a.v = b1.v$$
$$a.v_{id} = id(\texttt{b2}) \implies a.v = b2.v$$

5.3.4 Gestion des domaines d'abstraction

Suivant le problème, certaines modélisations de contraintes sont préférées. Par exemple, pour des problèmes de positionnements géométriques il peut être plus naturel d'exprimer les contraintes dans un formalisme adapté plutôt que de les exprimer par des contraintes sur des nombres réels représentant les coordonnées et dimensions des différents éléments. Ce type de contraintes est notamment utilisé dans les application de **C**onception **A**ssistée par **O**rdinateur (CAO) [BH11 ; DR94 ; SIN91]. Certains solveurs spécialisés peuvent résoudre ces contraintes qualitatives, mais ils sont assez rares. Cependant, il est possible d'encoder ces contraintes qualitatives (par exemple d'un élément soit contenu dans un autre) par des contraintes quantitatives utilisant des réels ou des entiers [Con+06 ; PTS06 ; WW09].



(a) Diagramme d'objets de la première contrainte (b) Diagramme d'objets de la seconde contrainte

FIGURE 5.6 – Diagrammes d'objets de deux des 6 contraintes générées par l'expansion cartésienne

(a) Diagramme de classes

(b) Diagramme d'objets

FIGURE 5.7 – Diagrammes de classes et d'objets utilisés par l'exemple d'encodage de relations

Avec notre approche, il est possible d'utiliser des solveurs capables de résoudre ces problèmes de contraintes qualitatives si ceux-ci exposent les capacités présentées dans la section 5.2 ou permettent de mettre en place des mécanismes d'émulations. Si ceci n'est pas possible, il faut encoder les contraintes qualitatives. Cet encodage se fait par l'application de transformations au modèle de contraintes pour l'adapter au solveur utilisé.

CHAPITRE 6

IMPLÉMENTATIONS

Dans le chapitre précédent, nous avons présenté un ensemble de techniques permettant d'implémenter l'approche d'exploration d'ensembles de modèles. Dans ce chapitre, nous détaillerons les caractéristiques des différentes implémentations.

Pour concrétiser l'approche présentée dans le chapitre 5 nous avons mis en place plusieurs prototypes. Parmi ces prototypes, nous présenterons les deux principaux. Le premier est un langage dédié (**D**omain **S**pecific **L**anguage (DSL)) interne à Xtend. Cette implémentation a été utilisée pour décrire le cas d'étude de l'article suivant [Le +19a].

Le second langage, appelé ATL^{C}, est une extension d'ATL, un langage de transformation dédié. Cette implémentation est basée sur ATOL, un compilateur ATL incrémental à base d'opérations actives, présenté dans [Le +19c]. ATL^{C} a été utilisé pour décrire les cas d'étude des articles suivants [Le +19b ; Le +19d]. Une version préliminaire de ce langage a été utilisée pour décrire le cas d'étude présenté dans l'article [Le +18b].

6.1 Première implémentation : langage dédié interne

La première implémentation utilisable de notre approche est un DSL interne développé dans Xtend combiné avec une **A**pplication **P**rogramming **I**nterface (API) permettant de créer la structure de synchronisation entre les contraintes, les propriétés du modèle et les solveurs. On nommera ce langage DSL Xtend dans la suite de ce document. La Figure 6.1 montre le processus d'exécution d'une transformation écrite dans ce DSL Xtend. Le processus global est proche de celui décrit dans la sous-section 5.1.1, on notera cependant que la partie concernant le traitement des contraintes est spécifique à Cassowary et que les différentes transformations nécessaires à la conversion des contraintes intermédiaires en contraintes spécifiques à Cassowary ne sont pas isolées, mais regroupées en une seule transformation écrite en Xtend (non incrémental).

6.1.1 Langage de transformation

Comme de nombreux langages de transformation, ce DSL est construit autour de la notion de règle. Une règle prend en entrée un élément et génère un ou plusieurs éléments en sortie. Le Listing 3 montre un exemple de règle de transformation, `UneRegle`, prenant en entrée une

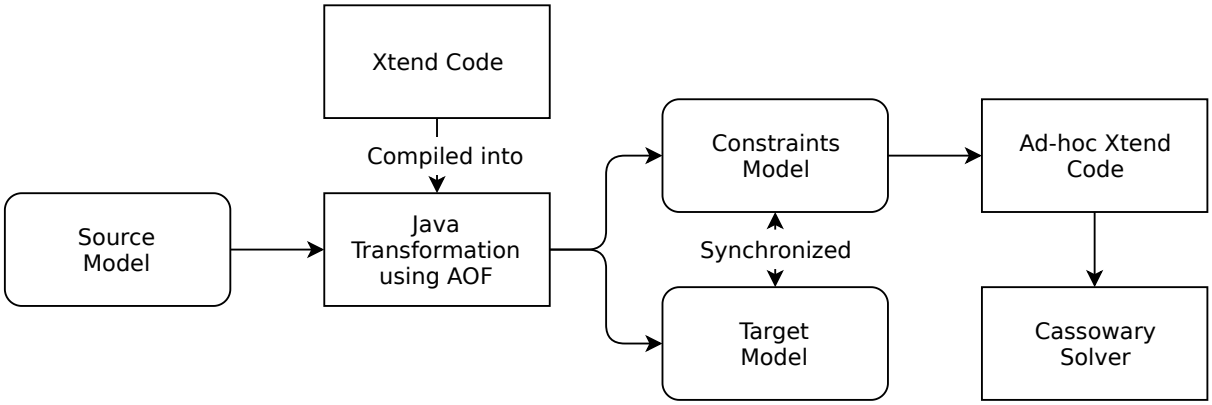


FIGURE 6.1 – Exécution d'une transformation avec le DSL Xtend

classe `Source` et générant deux éléments de sortie de classes `Cible1` et `Cible2` (ligne 1). La Figure 6.2 donne le diagramme de classes simplifié des règles de ce DSL. Dans ce DSL, les `Rule`s sont typées grâce à des types passés en paramètres. De plus, il existe une spécialisation de la classe `Rule` par nombre d'éléments de sortie. Par exemple, la règle du Listing 3 est une `Rule2`, car elle possède un élément d'entrée et deux éléments de sortie. Bien que techniquement possible, ce DSL ne permet pas actuellement la création de règles ayant plus d'un élément d'entrée.

```
1  private val Rule2<Source,Cible1,Cible2> UneRegle =
2  new Rule2<Source,Cible1,Cible2>(
3    to(Cible1Metaclass
4      ,cible1Prop <=> sourceProp1
5    ),
6    to(Cible2Metaclass,
7      cible2Prop <=> sourcePropEntier.collect[x | x * x],
8      cible2Prop2 <=> sourceProp2.collectTo(UneAutreRegle).a
9    )
10 )
```

Listing 3 – Règle de transformation avec contraintes

Les `Rule`s prennent en paramètre un `TargetPatternElement` par élément créé. Chaque `TargetPatternElement` est responsable de la création de l'élément de sortie ainsi que la définition des `Binding`s associés à l'élément créé. Dans le code, la création d'un `TargetPatternElement` se fait grâce à la méthode `to` qui en automatise la création. Celle-ci prend en paramètre une `Metaclass` permettant l'instanciation d'éléments et une liste de `Binding`s. Par exemple, dans le Listing 3 il y a deux sections `to`, une pour l'élément cible `Cible1` (lignes 3 à 5) et une pour l'élément cible `Cible2` (lignes 6 à 9).

Les `Binding`s sont composés d'une `TargetProperty` référençant une propriété de l'élément cible généré par le `TargetPatternElement`, et une `SourceExpression` permettant de calculer d'une valeur d'un type compatible avec la `TargetProperty` à partir d'une (ou plusieurs) proprié-

tés de l'élément source. Dans le DSL Xtend, l'opérateur binaire <=> est utilisé pour représenter un `Binding`. La `TargetProperty` se trouve à gauche du symbole et la `SourceExpression` à droite. Par exemple, dans le Listing 3 il y a trois `Binding`s (lignes 4, 7 et 8) illustrant les différentes possibilités des expressions utilisables dans les `SourceExpression`s.

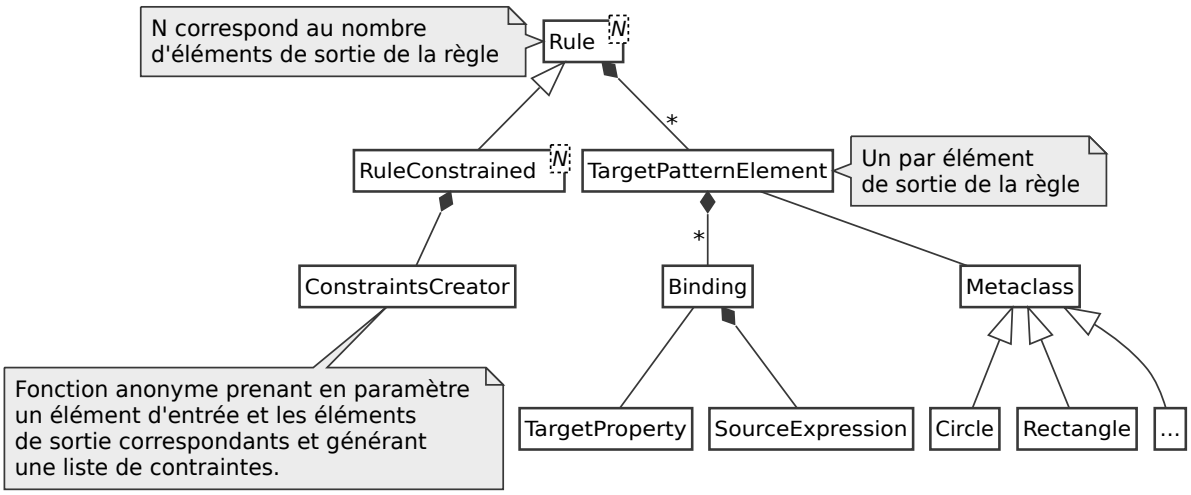


FIGURE 6.2 – Diagramme de classes simplifié d'une `Rule` Xtend

La `TargetProperty` est responsable de l'encapsulation de la propriété de l'objet concerné dans une `Box` AOF. De même, la `SourceExpression` récupère une propriété du modèle source et permet, par l'utilisation de méthodes comme `collect` ou `select`, de créer une expression calculant la valeur pour la `TargetProperty`. Par exemple, dans le Listing 3, le `Binding` à la ligne 4 est un *binding* simple liant directement les propriétés sources et cibles. Le second `Binding`, ligne 7, calcule la propriété `cible2Prop` de `Cible2` à partir du carré de la propriété `sourcePropEntier` de la classe `Source`. De nombreuses opérations (comme `collect`, `select` ou `zipWith`) prennent en paramètre une fonction anonyme effectuant l'action de l'opération (par exemple, la fonction à appliquer pour un `collect` ou un `zipWith` ou le prédicat servant à filtrer dans un `select`). Cependant, ces fonctions anonymes sont écrites en Xtend classique, elles peuvent donc avoir des effets de bords qui peuvent interférer avec les algorithmes de propagation des opérations actives. Le troisième `Binding`, ligne 8, utilise l'opération `collectTo` dans sa `SourceExpression`. Cette opération permet d'appliquer une règle à un ou plusieurs éléments. Dans cet exemple, on applique la règle `UneAutreRegle` aux éléments de la propriété `sourceProp2` de `Source`. Après l'utilisation de `collectTo` il est possible de référencer un élément de sortie de la règle en particulier avec les opérations `a`, `b`, `c` et `d`. Ces opérations référencent respectivement le premier, le second, le troisième et le quatrième élément de sortie de la règle.

Une fois instanciée, cette structure de `Rule`s, `TargetPatternElement`s, `SourceExpression`s et `Binding`s sert de cadre instanciant les opérations actives permettant le calcul incrémental de la transformation. Il est possible d'appliquer une règle à un élément source en

utilisant l'opération `collectTo`. Une fois la règle appliquée, les éléments cibles sont correctement générés et sont mis à jour dès qu'un changement est détecté sur l'élément source.

6.1.2 Langage de contraintes

La seconde partie du DSL Xtend concerne la déclaration de contraintes. En effet, pour l'instant nous n'avons présenté que la structure permettant la construction d'opérations actives calculant les cibles des règles.

Reprenons l'exemple de règle présentée plus tôt et ajoutons-y des contraintes. Le Listing 4 montre la règle modifiée.

La structure générale de la règle est conservée, on retrouve le même système de typage de la règle (ligne 2), on notera cependant l'utilisation de `Rule2Constrained` à la place de `Rule2`. Ensuite, les sections `to` sont identiques. L'ajout principal se situe après la section principale contenant les `to` (lignes 10 à 14).

```
1  private val Rule2<Source,Cible1,Cible2> UneRegle =
2  new Rule2Constrained<Source,Cible1,Cible2>(
3    to(Cible1Metaclass
4      ,cible1Prop <=> sourceProp1
5    ),
6    to(Cible2Metaclass,
7      cible2Prop <=> sourcePropEntier.collect[x | x * x],
8      cible2Prop2 <=> sourceProp2.collectTo(UneAutreRegle).a
9    )
10 ).constraints[s,c1,c2| #[
11   c1.x.stay(Strength.WEAK),
12   c1.x <= s.x,
13   c2.y >= s.y / 4
14 ]]
```

Listing 4 – Règle de transformation avec contraintes

Le diagramme de classes représentant les contraintes est donné en Figure 6.3. Il est assez proche du métamodèle intermédiaire de contraintes présenté dans la section 5.2. Cependant, nous pouvons noter plusieurs différences. Les `Constraint`s ne sont pas des `Expression`s, le DSL Xtend ne prenait en charge qu'un solveur, Cassowary, qui ne supporte pas la réification de contraintes. En conséquence, il n'était pas nécessaire de permettre la construction d'`Expression`s contenant plusieurs `Constraint`s. De plus, les opérateurs utilisables dans les `Constraint`s, `BinaryExpression`s et `UnaryExpression`s, sont limités par une énumération et non pas exprimés par des chaînes de caractères. Les méthodes `generate` de la `Constraint` et de l'`Expression` retournent respectivement des `CConstraint`s et `CExpression`s. Ces deux classes (`CConstraint` et `CExpression`) symbolisent les classes internes du solveur Cassowary représentant les concepts de contraintes et d'expression.

La méthode `constraints` (appelée à la ligne 10 du Listing 4) sert à spécifier le générateur de contraintes utilisé par la règle. Ce générateur de contraintes est une fonction anonyme

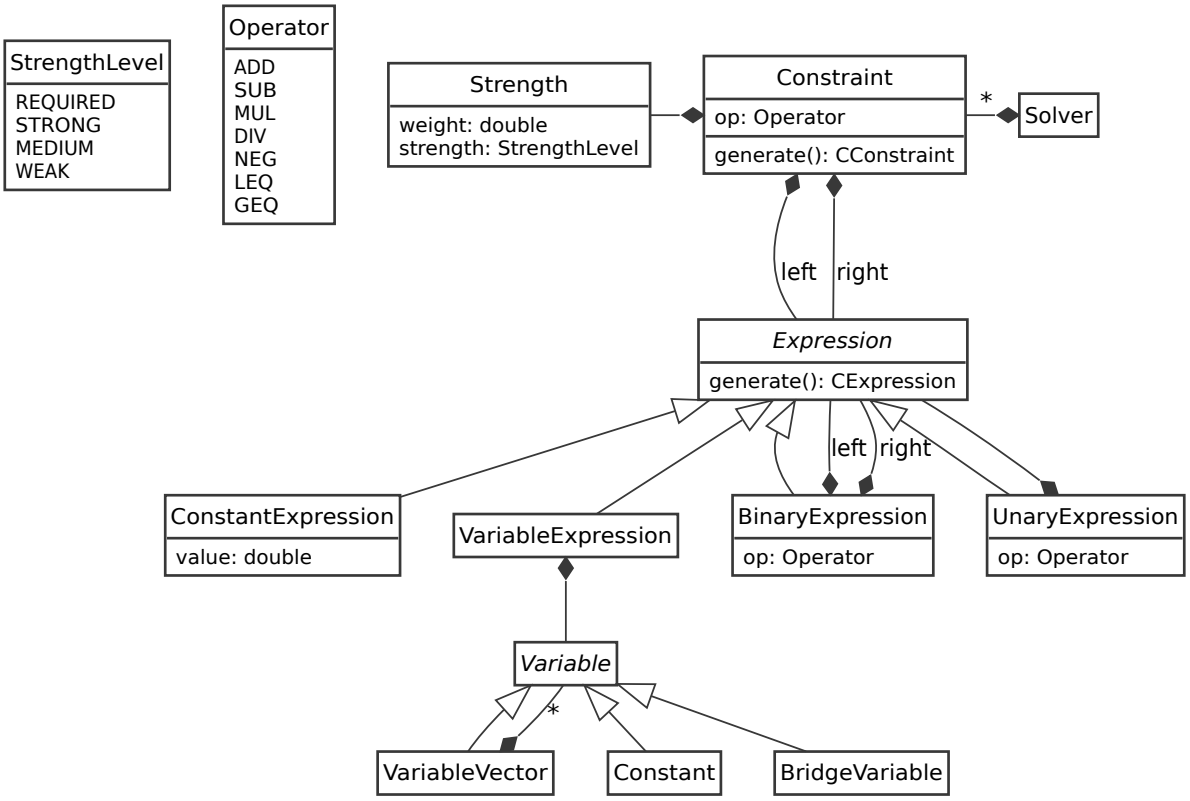


FIGURE 6.3 – Diagramme de classes simplifié des contraintes utilisées par le DSL Xtend

Xtend qui prend en paramètres l'élément source ainsi que les éléments cibles et génère une liste de `Constraint`s. Cette méthode est appelée chaque fois que la règle est appliquée à un nouvel élément source. La liste de `Constraint`s générée est donnée à un `Solver`. Le `Solver` crée ensuite les contraintes spécifiques à Cassowary grâce aux méthodes `generate`.

Dans le chapitre 5, le métamodèle de contraintes est générique et est transformé par chaque *wrapper* pour générer les contraintes spécifiques au solveur utilisé. Ici, le métamodèle de contraintes est déjà spécifique aux contraintes utilisables par `Cassowary` (par exemple, pas d'opérateur non supporté par Cassowary et contraintes sur les `double`s uniquement). Par conséquent, la majorité des étapes de transformation décrites dans la section 5.3 ne sont pas nécessaires. Le *wrapper* autour de Cassowary (la classe `Solveur`) se contente donc d'instancier les contraintes Cassowary en ne leur appliquant que peu de changements. La seule étape nécessaire est l'étape d'expansion des variables ponts qui est effectuée lors de l'appel à la méthode `generate`. Cette méthode regroupe donc l'expansion et la traduction des contraintes en une étape.

De plus, le code Xtend assurant la transformation du modèle de contraintes intermédiaires en contraintes Cassowary n'est pas incrémental. Par conséquent, ces transformations manuelles sont complexes et peu robustes.

6.1.3 Couche d'abstraction géométrique

Malgré ses limitations en termes de gestion de contraintes, ce DSL Xtend permet à l'utilisateur plusieurs façons d'exprimer ses contraintes. La première, présentée précédemment, consiste à écrire directement les contraintes arithmétiques envoyées à Cassowary.

Écrire ces contraintes peut se révéler complexe pour les utilisateurs n'ayant pas l'habitude de la programmation par contraintes. C'est pourquoi, pour faciliter l'utilisation du langage, nous avons ajouté une couche d'abstraction géométrique.

Cette couche fonctionne par réécriture des contraintes écrites par l'utilisateur en contraintes arithmétiques équivalentes (ou adaptées pour rester linéaires). La Figure 6.4 étend la Figure 6.1 en y ajoutant le traitement des abstractions géométriques. La Figure 6.5 montre l'ensemble des classes utilisées par la couche d'abstraction géométrique ainsi qu'une partie des méthodes qu'elles exposent.

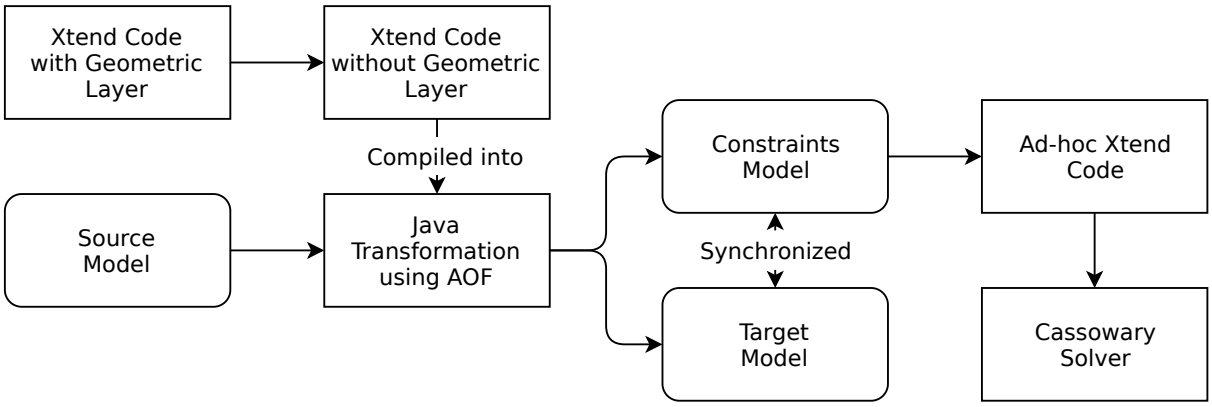


FIGURE 6.4 – Compilation et exécution d'une transformation utilisant l'abstraction géométrique

Le traitement des abstractions géométriques se fait au moment de la compilation des contraintes écrites en Xtend. La couche d'abstraction géométrique est composée d'un ensemble de classes regroupant les concepts géométriques souhaités. Ces classes exposent des méthodes permettant de facilement calculer des éléments géométriques (comme un `Point` ou une `Line`) à partir d'une forme géométrique. La `Value` encapsule le concept de `Variable`. Il est possible de lui appliquer des contraintes `stay` ou de lui suggérer une valeur (`suggest`). De plus, les `Value`s peuvent former des expressions arithmétiques complexes (grâce aux méthodes `+`, `-`, `*` et `/`). Il est aussi possible de construire des contraintes à partir de ces `Value`s en utilisant les opérateurs `<`, `>`, `>=`, `<=` et `==`, ou encore de construire une `Value` à partir d'une propriété JavaFX, d'une `Box` AOF ou d'un `double`. La classe `Shape` regroupe les opérations utilisables sur les différentes formes concrètes. Ces opérations permettent d'exprimer facilement des contraintes géométriques adaptées au positionnement d'éléments de diagrammes. Par exemple, il est possible de contraindre une `Shape` à être en dessous d'une autre `Shape`. Pour cela, les méthodes `getTopLeft` et `getBottomRight` sont utilisées pour calculer le rec-

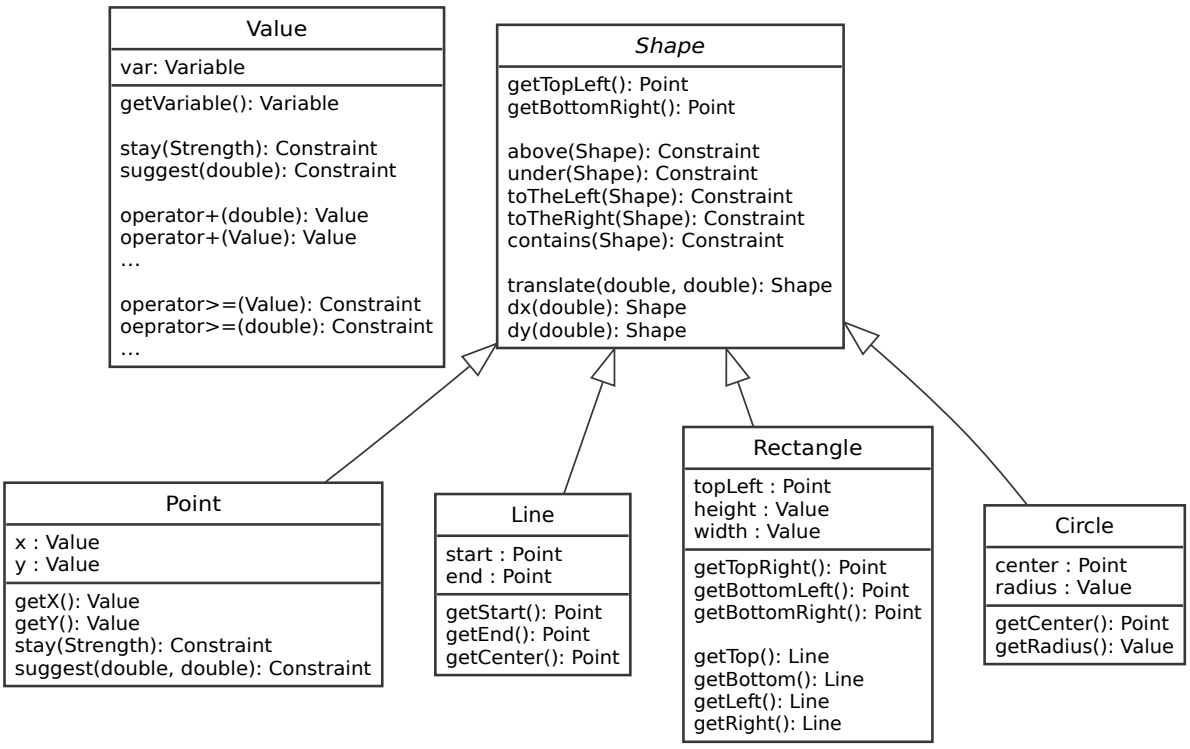


FIGURE 6.5 – Diagramme des classes utilisées par la couche d'abstraction géométrique

tangle contenant la `Shape`. L'abstraction géométrique propose quatre formes simples, le `Point`, la `Line`, le `Rectangle` et le `Circle`.

Ce sont les mêmes formes simples que celles utilisées en JavaFX. De plus, l'abstraction géométrique offre des méthodes permettant de facilement construire ces formes à partir de leurs équivalents JavaFX. Ainsi, il est possible d'utiliser la méthode `rectangle()` sur un `Rectangle` JavaFX pour obtenir un `Rectangle` de l'abstraction géométrique.

L'utilisation de ces méthodes entraîne la réécriture automatique des concepts géométriques en expressions arithmétiques. Par exemple, la contrainte suivante centrant un `Text t` 5 unités sous le bord supérieur d'un `Rectangle r`

```
t.rectangle.top.center.dy(5) == r.rectangle.top.center
```

est réécrite en deux contraintes arithmétiques, $t.x + (t.width/2) = r.x + (r.width/2)$ et $t.y + 5 = r.y$.

De même, la contrainte `r.rectangle.contains(c.circle.center)` contraignant le centre d'un `Cercle c` à être contenu dans un `Rectangle r` est réécrite en quatre contraintes, $c.x \geq r.x$, $c.x \leq r.x + r.width$, $c.y \geq r.y$ et $c.y \leq r.y + r.height$.

6.2 Seconde implémentation : extension d'un langage dédié avec compilateur

La seconde implémentation de notre approche conserve l'approche générale présentée dans la Figure 5.2 de la sous-section 5.1.1. La Figure 6.6 montre le processus général de génération de la transformation et son exécution. On nomme cette seconde implémentation ATL^C, pour ATL sous contraintes. Contrairement à l'implémentation précédente, ATL^C étend un langage de transformation dédié, ATL [JK06].

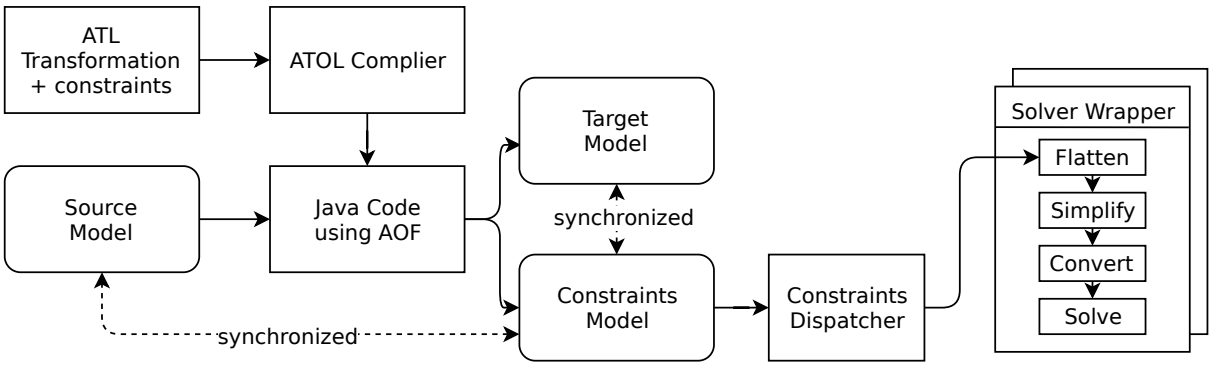


FIGURE 6.6 – Compilation et exécution d'une transformation en ATL^C

6.2.1 L'ATLAS Transformation Language

ATL est un langage de transformation hybride basé sur le concept de règle de transformation et utilisant une syntaxe basée sur OCL. La majorité du code ATL est déclaratif, cependant, il est possible d'utiliser du code impératif par moment. À l'origine, ATL est un langage de transformation unidirectionnel et non incrémental.

Les transformations ATL sont organisées en modules. Chaque module regroupe un ensemble de règles. Le Listing 5 montre un module ATL appelé `A2B` transformant des éléments d'un métamodèle `A` en éléments d'un métamodèle `B`. Dans le cas général, `A` et `B` sont différents, cependant, il est possible d'écrire des transformations ATL ayant le même métamodèle d'entrée et de sortie. Ces transformations particulières sont dites en mode *refining*.

```
1  module A2B;
2  create OUT : B from IN : A;
```

Listing 5 – Entête d'une transformation ATL

Le Listing 6 reprend la règle en DSL Xtend donnée dans la sous-section précédente. Cette règle prend une `Source` et génère une `Cible1` ainsi qu'une `Cible2`.

```
1  unique lazy rule UneRegle {
2    from
3      s : A!Source  (s.sourceProp1 > 2)
4    to
5      t1 : B!Cible1 (
6        cible1Prop <- s.sourceProp1
7      ),
8      t2 : B!Cible2 (
9        cible2Prop <- s.sourcePropEntier->collect(x | x * x)
10       cible2Prop2 <- thisModule.UneAutreRegle(s.sourceProp2).t
11     )
12 }
```

Listing 6 – Règle de transformation en ATL

Les règles ATL sont composées de trois parties. La première, ligne 1, donne le nom de la règle, son type et optionnellement une règle dont elle hérite.

Les différents types possibles sont :

— *rule* : ces règles (dites *classiques*) sont appliquées une fois par élément source compatible avec la règle.

— *lazy rule* : la règle doit être appelée explicitement par une autre règle. Plusieurs appels à une *lazy rule* sur une même source retournent des cibles distinctes.

— *unique lazy rule* : comme la *lazy rule*, la règle doit être appelée par une autre règle. Cependant, plusieurs appels à une même *unique lazy rule* avec une même source retournent les mêmes éléments cibles.

La deuxième partie d'une règle, lignes ligne 2 et ligne 3, spécifie les éléments sources de la règle. La section `from` liste les éléments source nécessaires à l'application de la règle. En plus de nommer et typer ces éléments, il est possible d'ajouter une garde prenant la forme d'une condition devant être satisfaite pour que la règle puisse être appliquée (ligne 3). Contrairement au DSL Xtend, ATL permet l'écriture de règles possédant plus d'un élément d'entrée. Lors de l'exécution d'une transformation ATL, le moteur de transformation essaye d'appliquer automatiquement les règles classiques aux éléments sources.

Enfin, la troisième partie (lignes 4 à 12) est composée des éléments cibles ainsi que de *bindings* permettant de calculer les valeurs de leurs propriétés. Ici, la règle possède deux éléments cible, `t1` de type `Cible1` (lignes 5 à 7) et `t2` de type `Cible2` (lignes 8 à 11). La propriété `cible1Prop` de `Cible1` est directement *bindée* à la propriété `sourceProp1` de `Source` (ligne 6). La propriété `cible2Prop` de `Cible2` est calculée comme le carré de la propriété `sourcePropEntier` de `Source` par une expression OCL (ligne 9). Enfin, la propriété `cible2Prop2` est calculée à partir du résultat de l'application de `UneAutreRegle` à `sourceProp2` de `Source`. Contrairement au DSL Xtend, il n'est pas possible de référencer un élément précis de la règle, c'est donc par défaut le premier élément qui est sélectionné.

L'héritage de règle permet le partage de code entre différentes règles. Une règle fille (ou sous-règle) doit être plus restrictive que la règle mère. La règle fille peut ajouter de nouveaux

bindings, modifier les *bindings* de la règle mère ou ajouter de nouveaux éléments de sortie.

Outre les règles ATL, il est possible d'écrire des *helpers* permettant de calculer des valeurs sous la forme d'un attribut (*attribute helper*) ou d'une opération (*operation helper*). Ces *helpers* permettent de créer des fonctions permettant de réutiliser des calculs ou encore de définir des calculs impossibles dans les règles. En effet, les *helpers* peuvent être récursifs.

6.2.2 ATOL, un compilateur ATL pour transformations incrémentales

Pour étendre ATL avec les contraintes dont nous avons besoin pour notre approche nous avons développé un compilateur ATL générant du code Java utilisant AOF, l'implémentation Java des opérations actives. Le compilateur ATOL [Le +19c] ne supporte qu'un sous-ensemble du langage ATL. Cependant, parce que le code généré utilise AOF, ATOL supporte les transformations incrémentales. C'est-à-dire, lorsqu'un élément du modèle source est modifié, AOF propagera le changement au modèle cible en ne recalculant que les éléments impactés par la mise à jour. La Figure 6.7 montre le processus global de compilation d'une transformation ATL en code Java utilisant AOF.

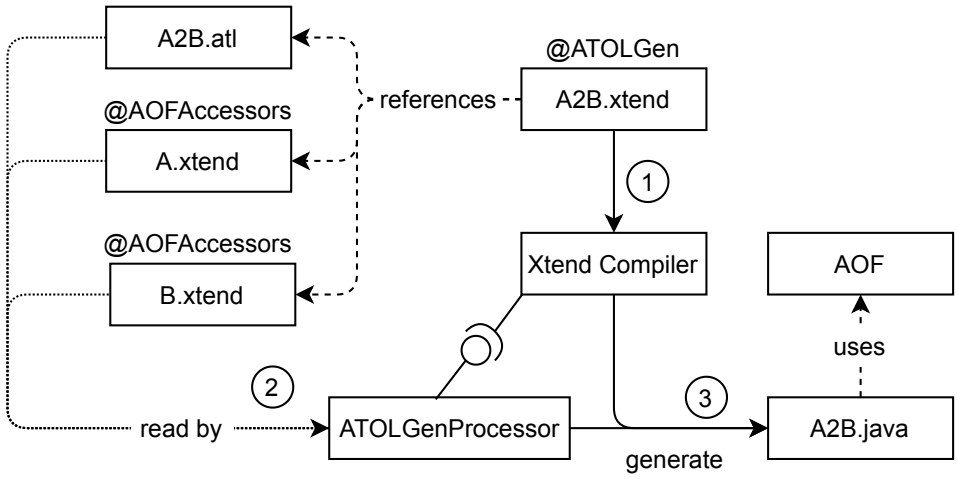


FIGURE 6.7 – Aperçu de l'architecture d'ATOL

Le compilateur ATOL prend la forme d'un plug-in au compilateur Xtend (plus précisément une annotation active). Ce plug-in ajoute plusieurs annotations générant du code Java lors du processus de compilation Xtend.

```
1 @AOFAccessors(APackage)
2 class A {
3 }
```

Listing 7 – Fichier Xtend générant les méthodes nécessaires à ATOL pour le métamodèle `A`

Pour utiliser le compilateur ATOL, l'utilisateur crée, en plus de son fichier ATL, un fichier

```
@ATOLGen(transformation="A2B.atl",metamodels=#[
  @Metamodel(name="A", impl=A),
  @Metamodel(name="B", impl=B)
])
class A2B {
}
```

Listing 8 – Exemple de fichier Xtend avec l'appel au compilateur ATOL

Xtend pour la transformation et un fichier Xtend par métamodèle utilisé par la transformation. Pour fonctionner, ATOL requiert la présence d'un ensemble de méthodes et de classes permettant la création de `Box`es à partir d'éléments du métamodèle. Ces méthodes peuvent être définies manuellement, mais elles peuvent aussi être automatiquement générées (pour les modèles utilisant EMF) en utilisant l'annotation `@AOFAccessors`. Par exemple, le Listing 7, montre la classe annotée qui contiendra le code permettant l'utilisation du métamodèle `A` par ATOL.

Une autre annotation (`@ATOLGen`) permet elle de générer le code correspondant à une transformation ATL passée en paramètre. Le Listing 8 montre l'utilisation de cette annotation pour générer une classe `A2B` contenant le code Java nécessaire à l'exécution de la transformation entre les métamodèles `A` et `B` décrite dans `A2B.atl`.

Parce qu'ATOL génère du code Java utilisant AOF pour calculer les *bindings* ATL ceux-ci sont calculés incrémentalement. Un changement sur le modèle source est propagé au modèle cible par le graphe de propagation d'AOF.

Cependant, toutes les constructions du langage ATL ne sont pas compatibles avec une exécution incrémentale de la transformation. Par ailleurs, ATOL est encore au stade de prototype, c'est pourquoi tous les concepts compatibles avec l'exécution incrémentale ne sont pas supportés. Et notamment, les sections impératives d'ATL ne sont pas supportées.

En effet, lors de l'exécution d'une transformation ATL classique, le moment auquel ces portions impératives doivent être exécutées est bien défini. Cependant, lors de l'exécution incrémentale il est plus complexe de prévoir ces moments. C'est pourquoi nous avons décidé de ne pas gérer les portions impératives d'ATL.

De plus, l'implémentation actuelle ne supporte que les *unique lazy rules*. Par souci de simplicité, l'algorithme de *matching* n'a pas été ajouté. Par conséquent, la gestion des gardes des règles n'est pas nécessaire, car les règles de transformation sont appelées explicitement par d'autres règles. Il n'y a donc pas besoin de ce mécanisme de contrôle d'application des règles.

6.2.3 ATL^{C}, une extension d'ATOL

Nous avons vu qu'ATOL est un compilateur ATL développé en Xtend. En plus de la possibilité d'exécution incrémentale des transformations, ATOL a été conçu avec l'idée de plug-ins permettant l'ajout de fonctionnalités à ATL. Dans notre contexte, un plug-in est utilisé pour modifier le comportement du compilateur lorsqu'il rencontre un type d'élément de sortie spéci-

```
1 @ATOLGen(transformation="A2B.atl",metamodels=#[
2   @Metamodel(name="A", impl=A),
3   @Metamodel(name="B", impl=B),
4   @Metamodel(name="Constraints", impl=Constraints)
5 ], extensions = #[Constraints])
6 class A2B {
7 }
```

Listing 9 – Exemple de fichier Xtend avec l'appel au compilateur ATOL avec les ajouts nécessaires à la compilation de contraintes

fique.

Plus précisément, pour la Figure 6.7, le plug-in gérant les contraintes intervient au niveau d'`ATOLGenProcessor`. Lors de la compilation des *bindings* du fichier ATL, le compilateur vérifie si le *binding* peut être compilé par une des extensions enregistrées. Si c'est le cas, alors l'extension est utilisée pour compiler le *binding* plutôt que le compilateur classique. Ici, seuls les *bindings* `constraints` des cibles appartenant au métamodèle `Constraint` sont ciblés.

Le Listing 9 montre le code nécessaire à la compilation d'une transformation `A2B.atl` utilisant les contraintes. Pour cela, le métamodèle de contraintes est ajouté aux métamodèles utilisés (ligne 4) ainsi que le plug-in de compilation des contraintes (ligne 5).

Le Listing 10 montre la règle ATL du Listing 6 (sans la garde sur l'élément source) augmentée avec des contraintes. On remarque que l'ajout de contraintes se fait par l'ajout d'un nouvel élément cible de type `ConstraintGroup`. Cet élément contient une liste de `Constraint`s (`constraints`). C'est ici que le plug-in de gestion des contraintes remplace le compilateur ATL classique. Sans ce plug-in, le compilateur ATOL évaluerait les contraintes et ne générerait qu'une liste de booléens.

```
1  unique lazy rule UneRegle {
2    from
3      s : A!Source
4    to
5      t1: B!Cible1 (
6        cible1Prop <- s.sourceProp1
7      ),
8      t2: B!Cible2 (
9        cible2Prop <- s.sourcePropEntier->collect(x | x * x)
10       cible2Prop2 <- thisModule.UneAutreRegle(s.sourceProp2).t
11     ),
12     cstr: Constraint!ConstraintGroup(
13       solver <- 'Cassowary',
14       constraints <- Sequence {
15         t1.x.stay('WEAK'),
16         t1.x <= s.x,
17         t2.y >= s.y / 4
18       }
19     )
20 }
```

Listing 10 – Règle de transformation ATL contenant des contraintes

Le plug-in ne va pas interpréter les contraintes, mais générer les contraintes équivalentes conforment au métamodèle de contraintes présenté dans la section 5.2. Ces contraintes pourront ensuite être données aux *wrappers* correspondant aux solveurs cibles (ici Cassowary, ligne 13). D'une certaine façon, le plug-in de compilation des contraintes n'est qu'un sucre syntaxique évitant à l'utilisateur de devoir instancier manuellement l'arbre de contraintes. De cette façon, l'utilisateur peut utiliser un langage plus proche des langages de contraintes classiques. Le code Java généré par le plug-in consiste en un ensemble de méthodes servant à instancier dynamiquement le modèle de contraintes correspondant aux contraintes décrites dans la transformation ATL.

L'inconvénient principal de cette approche est la limitation du langage de contraintes par la syntaxe reconnue par le parseur ATL utilisé par le compilateur ATOL. Par souci de simplicité, nous avons décidé de réutiliser le parseur du moteur d'exécution de référence d'ATL, EMFTVM [1]. Par conséquent, il n'est pas possible d'ajouter de nouveaux éléments de syntaxe pour faciliter l'écriture des contraintes. C'est pourquoi les contraintes doivent suivre la même syntaxe que les expressions OCL. Cependant, cette syntaxe est assez souple et proche de la syntaxe des contraintes, cette limitation n'apparaît donc pas lors de la spécification de la majorité des contraintes.

Une fois générées, contrairement au DSL Xtend, les contraintes peuvent être utilisées comme n'importe quel élément de sortie des règles. Il est par exemple possible de sérialiser ces contraintes au format XMI ou XCSP3 [BLP16] ou de les utiliser dans des solveurs de contraintes. Pour les utiliser, il est nécessaire d'appliquer les méthodes de réécritures décrites dans la section 5.3.

Au moment de l'écriture de ce manuscrit, ATL^C possède plusieurs *wrappers* autour de solveurs offrant des capacités variées.

— Cassowary [BBS01] : solveur dynamique de contraintes linéaires hiérarchiques sur nombres flottants.

— Choco [Pru+17] : solveur de contraintes généraliste sur variables discrètes.

— MiniCP [Lau18] : solveur de contraintes généraliste sur variables discrètes.

— XCSP3 [BLP16] : format de modélisation de contraintes basé sur le format E**x**tensible **M**arkup **L**anguage (XML).

Chaque *wrapper* intègre une partie des réécritures décrites dans la section 5.3 pour adapter les contraintes brutes en contraintes du solveur cible. Par exemple, lorsque l'utilisateur utilise des contraintes `stay` dans un problème destiné à Choco, il est nécessaire d'ajouter des contraintes pour émuler le comportement des contraintes `stay` de Cassowary. De plus, on passe d'un problème de satisfaction de contraintes à un problème d'optimisation. Toujours pour Choco, si l'utilisateur crée des variables à partir de propriétés utilisant des nombres réels alors

1. `https://wiki.eclipse.org/ATL/EMFTVM`

chaque propriété est convertie en une variable entière et l'utilisateur est averti de la conversion. Ces transformations sont écrites, pour l'instant, en Xtend et utilisent AOF.

Enfin, actuellement ATL^C ne possède pas de couche d'abstraction géométrique. Nous avons pour projet de proposer de nouvelles couches d'abstractions pour faciliter l'utilisation d'ATL^C pour des utilisateurs n'ayant pas l'habitude d'utiliser des solveurs de contraintes. Cependant, nous souhaitons par là même mettre en place un système pouvant être appliqué pour d'autres abstractions, comme l'abstraction temporelle. C'est pourquoi nous avons préféré, dans un premier temps, nous assurer du bon fonctionnement d'ATL^C avant d'introduire de nouvelles extensions.

Nous considérons aussi l'intégration d'outils spécialisés dans l'affichage de graphe comme l'**E**clipse **L**ayout **K**ernel (ELK) [2], ou d'autres approches [Gut+03 ; JM04]. Avec un outil comme ELK, il serait possible d'utiliser des contraintes pour placer les éléments critiques (par exemple, le nom d'une classe dans un diagramme de classes) et ELK pour donner des positions initiales aux éléments non critiques (par exemple, les positions des classes).

6.2.4 Fonctions d'un *wrapper* autour d'un solveur

On appel *wrapper*, l'interface entre notre modèle de contraintes généré par les transformations ATL^C et le solveur de contraintes. Ceux-ci remplissent les cinq rôles suivants :

1. **Réécriture des contraintes** : les contraintes générées par la transformation ne sont pas nécessairement dans un format adapté au solveur de contraintes. Il faut alors les réécrire pour qu'elles puissent être utilisées par le solveur. Par exemple, les contraintes contenant des `VariableVector`s doivent, le plus souvent, être mises à plat (cette étape correspond à la suppression des `ConstraintsGroup`s décrite dans la sous-section 5.3.1).
2. **Génération des contraintes spécifiques au solveur** : une fois traitées, les contraintes générées par la transformation sont exprimables dans le formalisme de modélisation du solveur cible. Cependant, il faut générer, à partir de notre modèle de contraintes, les contraintes équivalentes dans le formalisme du solveur.
3. **Synchronisation des propriétés et des variables** : une fois les contraintes spécifiques au solveur générées, il faut mettre en place des mécanismes de synchronisation entre les propriétés du modèle et les variables du solveur. La synchronisation des variables du solveur aux propriétés du modèle peut varier grandement d'un solveur à l'autre.
4. **Synchronisation du modèle de contraintes et des contraintes spécifiques** : lorsque le modèle source change, le modèle cible est mis à jour, il en va de même pour les contraintes correspondantes. Il faut alors modifier les contraintes spécifiques au solveur de façon à refléter les changements du modèle de contraintes. Par exemple, l'ajout d'un élément au modèle source peut entraîner l'ajout de plusieurs contraintes dans le modèle de contraintes, qui doivent être ajoutées aux contraintes spécifiques du solveur.

2. `https://www.eclipse.org/elk/`

5. **Interface de suggestion** : dans la version actuelle d'ATL^C, les suggestions de modifications de variables se font manuellement par un appel au *wrapper*. Cet appel permet un meilleur contrôle lorsque la suggestion n'est pas réalisable et que le solveur ne peut trouver de solution. Dans de telles situations, il est nécessaire d'effectuer un *rollback*, ce que la version actuelle d'AOF ne supporte pas.

Tous ces différents rôles ne sont pas nécessaires pour tous les *wrappers*. Par exemple, actuellement, le *wrapper* autour d'XCSP3 ne prend en charge que les deux premiers (réécriture de conversion des contraintes), car ce *wrapper* est utilisé pour sérialiser les contraintes et non les résoudre. Par conséquent, ce *wrapper* n'a pas besoin de mécanismes de synchronisation entre le modèle de contraintes et le modèle XCSP3. Il est plus simple de reconstruire un modèle XCSP3 complet à partir du nouveau modèle de contraintes.

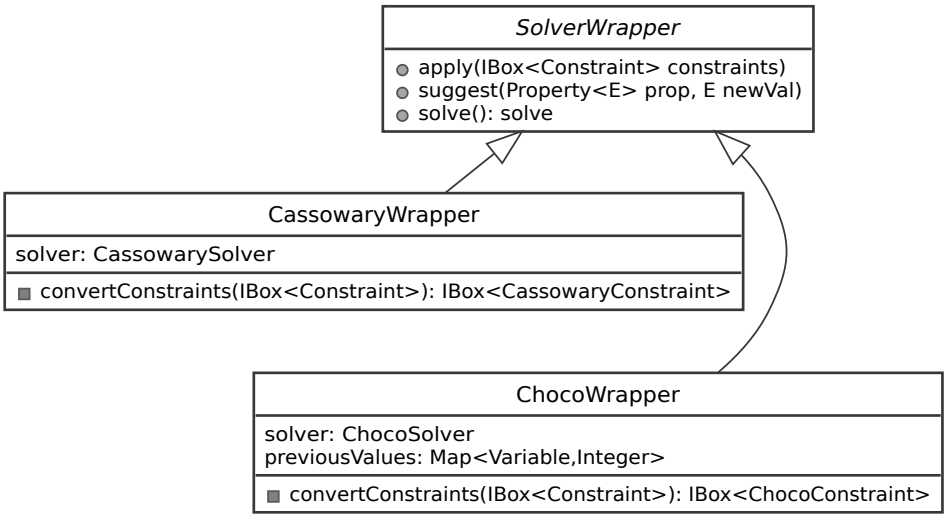


FIGURE 6.8 – Diagramme de classes d'une partie des *wrappers* utilisables avec ATL^C

La Figure 6.8 montre le diagramme de classes simplifié des *wrappers* autour de Cassowary et de Choco. La classe abstraite `SolverWrapper` regroupe les méthodes partagées par les différents *wrappers*. La première, `apply(IBox<Constraint>)`, donne les contraintes du modèle de contraintes que le *wrapper* devra traiter et soumettre au solveur. On notera l'utilisation d'une `IBox` permettant d'être notifié des changements des contraintes sources. La seconde, `suggest(Property<E> prop, E newVal)`, sert à suggérer une nouvelle valeur (`newVal`) à la variable correspondant à la propriété (`prop`). Suivant le solveur, la suggestion d'une nouvelle valeur peut entraîner une mise à jour automatique de la solution. Si aucune solution n'est trouvée, `suggest` peut lancer une exception pour indiquer le problème. Enfin, `solve()`, indique au *wrapper* qu'il faut recalculer une solution. Cette méthode est principalement utilisée avec les solveurs non dynamiques. Avec ces solveurs, le calcul d'une nouvelle solution peut être couteux, il est donc préférable de limiter le nombre d'appels au solveur.

Les différents *wrappers* autour des solveurs étendent `SolverWrapper`. Ces *wrappers* implémentent les différentes méthodes vues plus tôt et surtout implémentent un ensemble de méthodes pour convertir le modèle de contraintes en contraintes spécifiques (ici les méthodes `convertConstraints(IBox<Constraint>)` des classes `CassowaryWrapper` et `ChocoWrapper`).

Suivant les capacités du solveur et du *wrapper*, il est parfois nécessaire d'ajouter des fonctionnalités au *wrapper*. Par exemple, `ChocoWrapper` supporte la contrainte `stay`, mais Choco ne la supporte pas. C'est pourquoi, `ChocoWrapper` possède la map `previousValues` servant à ajouter les contraintes `stay` à chaque résolution.

6.3 Comparaison entre le DSL Xtend et ATL^C

Pour conclure, les deux implémentations, le DSL Xtend et ATL^C, permettent le développement de transformations générant des ensembles de modèles explorables. Les deux DSL partagent des concepts similaires dans l'approche, mais varient significativement dans l'implémentation. Il en va de même pour les capacités de ces deux implémentations. Le Tableau 6.1 compare les deux implémentations sur différents points.

	DSL Xtend	ATL^C
Type de DSL	Interne	Externe
Implémentation	API	Compilateur
Modèle de contraintes	Spécialisé	Générique
Solveurs disponibles	Cassowary	Cassowary, Choco, MiniCP, xcsp3
Collaboration entre solveurs	Non	Manuelle
Définition des interacteurs	Impérative	Impérative
Couches d'abstraction	Géométrique	Non [3]
Réécritures de contraintes	Manuelle	Manuelle / Transformation de modèles
Couplage cible / contraintes	Faible	Fort

TABLE 6.1 – Tableau comparatif des caractéristiques des deux implémentations

ATL^C a été conçu après le DSL Xtend, c'est pourquoi il ajoute globalement de nouvelles fonctionnalités telles que le support de plusieurs solveurs. La gestion des contraintes est améliorée avec ATL^C. En effet, avec le DSL Xtend, il n'est pas possible de récupérer les liens de traçabilité entre un élément source et les contraintes correspondantes. Par conséquent, il est difficile, lors de la suppression d'un élément source, de supprimer les contraintes correspondantes. Alors qu'en ATL^C les contraintes sont complètement intégrées dans la transformation et sont des éléments cibles comme les autres. La traçabilité est alors automatiquement gérée par la transformation et la suppression des contraintes liées à un élément source est possible.

Le seul avantage du DSL Xtend sur ATL^C est la possibilité d'utiliser une couche d'abstraction géométrique facilitant ainsi l'expression des contraintes pour des utilisateurs non ex-

3. Abstraction géométrique en projet.

périmentés. C'est pourquoi nous avons prévu d'ajouter des couches d'abstractions à ATL^C. De plus, nous souhaitons mettre en place une structure générique pour pouvoir, par la suite, intégrer facilement d'autres couches d'abstraction.

CHAPITRE 7

CAS D'ÉTUDE

Dans les chapitres précédents, nous avons présenté notre approche permettant l'exploration d'ensembles de modèles. Nous avons notamment présenté la méthode que nous avons mise en place pour résoudre ce problème et les langages que nous avons développés pour mettre en place cette méthode automatiquement.

Dans les sections suivantes, nous présenterons plusieurs cas d'étude développés durant la thèse pour explorer différents aspects des langages de transformation proposés. Avant de présenter les différents cas d'étude, nous introduirons, dans la section 7.1, le métamodèle de diagrammes utilisé dans toutes les transformations suivantes. Ensuite, dans la section 7.3, nous présenterons un visualiseur de diagrammes de séquence développé avec le DSL interne à Xtend. Puis, dans la section 7.2, nous présenterons trois visualiseurs de diagrammes UML développés, toujours avec le DSL Xtend, par quatre étudiants de l'ESEO dans le cadre de leur projet de fin d'études. Dans la section 7.4, nous détaillerons un visualiseur de planning, développé en ATL^C, dans lequel plusieurs solveurs collaborent. Enfin, dans la section 7.5 nous présenterons un éditeur permettant de manipuler un diagramme de classes et d'objets dans une même fenêtre. Cet éditeur sert de cas d'étude d'un article d'atelier international [Le +19d].

7.1 L'exploration de modèles appliquée aux diagrammes

Les diagrammes que nous allons présenter sont composés de formes géométriques simples. Par souci de simplicité, nous ne considérons que les formes simples telles que le rectangle, la ligne, la flèche (avec différentes têtes), le texte ou le cercle. La Figure 7.1 donne une version simplifiée du métamodèle de diagramme que nous utilisons. Ce métamodèle simplifié de vue est inspiré de JavaFX[1], une bibliothèque Java permettant de développer des interfaces graphiques. Comme JavaFX, ce métamodèle contient des éléments graphiques simples pouvant être affichés sur un écran.

Il y a plusieurs classes représentant des formes primitives, telles que le `Rectangle`, le `Circle` ou la `Line`. D'autres représentent différents types de flèches, `Arrow`, `DiamondArrow`, `GeneralizationArrow` et `AggregationArrow`. La classe `Text` permet d'afficher du texte. Enfin, `ActorFX` représente une personne stylisée, cet élément est notamment utilisé dans le diagramme de cas d'utilisation. Cette dernière forme pourrait être composée d'un cercle et de

1. Documentation accessible à l'adresse suivante : `https://docs.oracle.com/javase/8/javafx/api/`

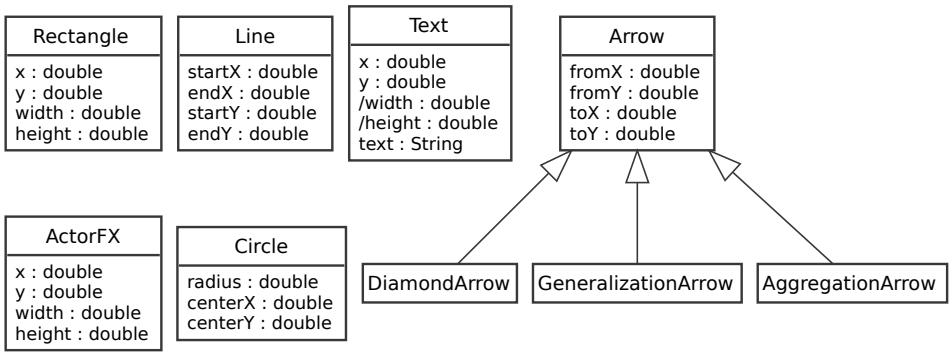


FIGURE 7.1 – Métamodèle de diagrammes simplifié

lignes mais sert ici à montrer comment une forme définie par l'utilisateur peut être réutilisée.

Ces classes se regroupent en plusieurs catégories.

Les classes `Rectangle`, `Text` et `ActorFX` se comportent toutes comme des rectangles au sens géométrique du terme. Elles possèdent quatre propriétés `x`, `y`, `width` et `height` décrivant respectivement la position et les dimensions du rectangle. Pour l'`ActorFX`, ce rectangle correspond à un rectangle invisible dans lequel la figure humaine est contenue. Pour le `Text`, les propriétés `width` et `height` ne sont accessibles qu'en lecture seule, car elles sont calculées à partir des dimensions du `text` à afficher et de la police utilisée.

Le `Circle` représente un cercle et possède trois propriétés : un `radius` pour son rayon et deux coordonnées `x` et `y` pour son centre. La `Line` représente une ligne et possède quatre propriétés, deux coordonnées pour chaque extrémité de la ligne (`startX`, `startY`, `endX` et `endY`). Les classes `Arrow`, `DiamondArrow`, `GeneralizationArrow` et `AggregationArrow` possèdent, elles aussi, quatre propriétés, deux coordonnées pour chaque extrémité de la flèche (`fromX`, `fromY`, `toX` et `toY`).

Ce métamodèle minimaliste n'est pas spécifique à un diagramme en particulier, mais peut servir pour en représenter une multitude. De plus, lorsque les diagrammes le nécessitent, il est possible d'ajouter des éléments dédiés (l'`ActorFX` par exemple).

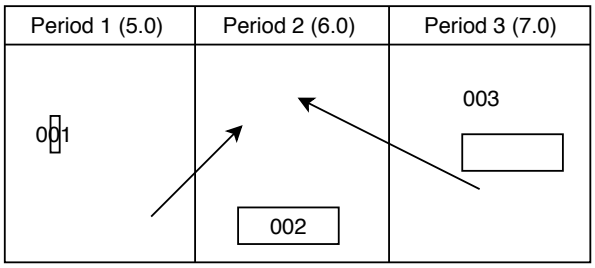


FIGURE 7.2 – Un diagramme conforme au métamodèle montré dans la Figure 7.1

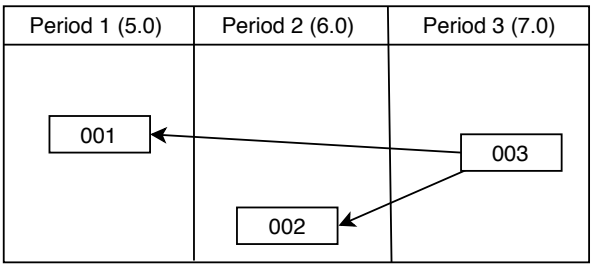


FIGURE 7.3 – Un diagramme conforme et valide

Ce métamodèle permet de représenter des figures géométriques, mais ne permet pas, par construction, d'assurer la validité d'un diagramme. En effet, chaque diagramme possède ses propres règles de construction qui spécifient notamment les positions possibles des éléments les uns par rapport aux autres. Par exemple, considérons un diagramme représentant un planning, des tâches doivent être affectées à des périodes. La Figure 7.3 montre un diagramme valide, chaque tâche est représentée par un rectangle contenant son identifiant, les rectangles correspondant aux tâches sont contenus dans les rectangles correspondant aux périodes et les liens de dépendance entre les tâches sont représentés par des flèches. Dans la Figure 7.2, les formes présentes ne forment pas un diagramme valide. Bien que les deux figures soient composées d'éléments graphiques respectant le métamodèle présenté dans la Figure 7.1 une seule des deux est valide quand on la considère comme un diagramme.

Pour être valide, un diagramme doit, en plus d'être conforme à un métamodèle de formes géométriques, respecter un ensemble de règles de construction imposées par la syntaxe graphique du diagramme en question. Ces règles sont spécifiques à chaque type de diagramme. Par exemple, le diagramme de la Figure 7.3 est valide, tandis que celui de la Figure 7.2 ne l'est pas.

Ainsi, les diagrammes valides forment un sous-ensemble des images composées de figures géométriques, ou, en d'autres termes, les diagrammes valides forment un ensemble de modèles, quand on considère l'espace des modèles formé par toutes les figures géométriques conformes au métamodèle présenté dans la Figure 7.1 et les règles de construction (ou contraintes) imposées par la syntaxe graphique du diagramme considéré. Il est donc possible d'utiliser notre approche pour :

1. Spécifier déclarativement la transformation permettant la visualisation, sous la forme d'un diagramme, d'un métamodèle quelconque.
2. Explorer l'ensemble des diagrammes valides correspondant à un modèle conforme au métamodèle source.

Dans le cas des diagrammes, l'exploration de l'ensemble des modèles peut impliquer plusieurs types de réparations. Le premier type de réparation, le plus simple, n'impacte que la vue, par exemple, sur le diagramme de la Figure 7.3, l'utilisateur déplace le rectangle correspondant à la tâche `003` dans le rectangle correspondant à la période 3. Dans ce cas, le solveur se contente de calculer les nouvelles positions des extrémités des flèches partant de la tâche `003` de façon à ce qu'elles soient toujours contenues dans le rectangle correspondant à la tâche `003`.

Le second type de réparation envisageable implique une mise à jour du modèle source. Par exemple, toujours sur le diagramme de la Figure 7.3, imaginons que l'utilisateur déplace le rectangle correspondant à la tâche `002` dans le rectangle correspondant à la période 1. Dans ce cas là, le changement est appliqué au modèle source, ici, changer l'affectation de la tâche `002` de la période 2 à la période 1. Ensuite, le moteur d'exécution de transformation de modèles incrémentale propage le changement jusqu'à la vue.

On définit un interacteur comme le système responsable de la capture des évènements, clavier et souris, produits par l'utilisateur et de l'application des modifications si nécessaire. Par exemple, dans le diagramme de la Figure 7.3, lorsque l'utilisateur clique sur un rectangle correspondant à une tâche et le déplace, c'est l'interacteur qui analyse le déplacement de la figure (simple déplacement du rectangle ou changement de période ?) et applique les changements nécessaires (mise à jour des coordonnées du rectangle pour un simple déplacement, ou changement de période sinon). Bien qu'intéressants, ces interacteurs sont au-delà du cadre de cette thèse et nous avons décidé de ne pas les intégrer aux langages développés. En effet, bien que pratiques pour définir comment l'utilisateur peut interagir avec une interface graphique, ces interacteurs ne sont pas directement liés à notre approche. C'est pourquoi, dans les différents exemples de visualiseurs de diagrammes suivants, le code des interacteurs est écrit en Xtend impératif.

De plus, ces démonstrateurs ont été implémentés dans l'objectif de montrer la pertinence de l'approche et non dans le but de concevoir des outils capables de remplacer des éditeurs de diagrammes professionnels.

7.2 Travaux étudiants

Les trois visualiseurs de diagrammes suivants ont été réalisés par quatre étudiants de l'ESEO dans le cadre de leur projet de fin d'études. Je remercie donc Grégoire CHABOT, Florian NERRIÈRE, Marion PIOU et Robin STRAEBLER pour leur investissement et leurs retours qui nous ont été précieux lors de la conception d'ATL^C. Dans la sous-section 7.2.1 nous présenterons le premier projet, un visualiseur de diagrammes de classes. Puis, dans la sous-section 7.2.2 nous présenterons un visualiseur de diagrammes d'activité. Enfin, dans la sous-section 7.2.3 nous présenterons un visualiseur de diagrammes de cas d'utilisation.

7.2.1 Diagramme de classes

Dans ce cas d'étude, l'objectif est, à partir d'un modèle UML, de générer un diagramme de classes avec lequel l'utilisateur peut interagir. La Figure 7.5 montre un exemple de diagramme de classes généré par le cas d'étude. Le diagramme généré par cet exemple utilise le métamodèle d'éléments graphiques de la Figure 7.1. La Figure 7.4 montre la partie du métamodèle UML utilisé par la transformation. Ce métamodèle, en plus d'être un sous-ensemble du métamodèle UML, a aussi été simplifié, les classes intermédiaires et les propriétés non utilisées ont été ignorées.

Ce métamodèle UML simplifié est composé de `NamedElement`s possédant un `name` et se déclinant en plusieurs sous-classes : `Association`, `Class`, `Operation`, `Package` et `Property`. Un `Package` est un `PackageableElement` et possède un ensemble de `PackageableElement`s (`packagedElement`). La `Class` est un `Type` qui est à son tour un `PackageableElement`. Une

`Class` est composée d'un ensemble d'`Operation`s (`ownedOperation`) et de `Property`ies (`ownedAttribute`). Les `Generalization`s encodent les liens d'héritage entre `Class`es : la super `Class` (`general`) et la `Class` fille (`specific`). Une `Association` référence plusieurs `Property`ies (`memberEnd`). Les `Property`ies correspondent à une propriété d'une `Class`, leur type est donné par la propriété `type` référençant un `Type`.

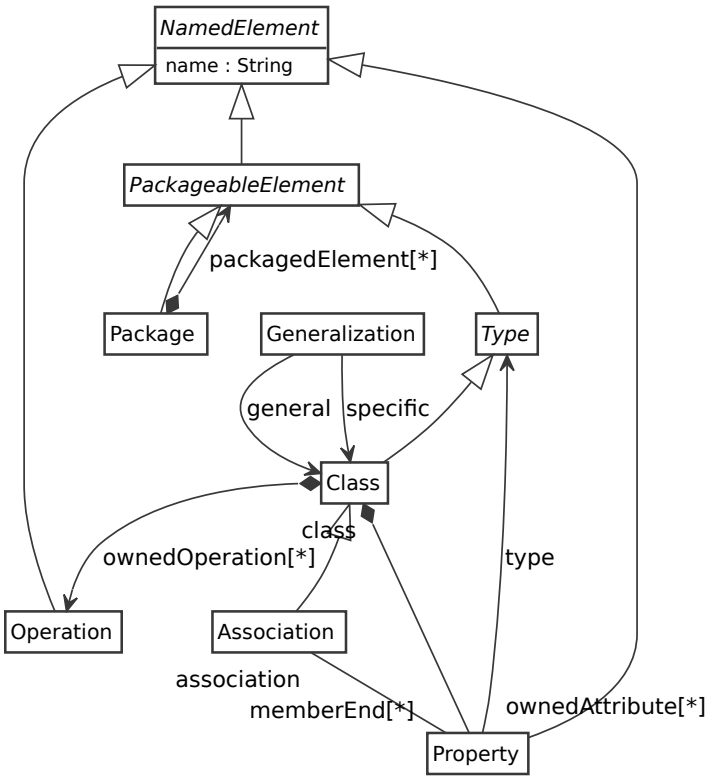
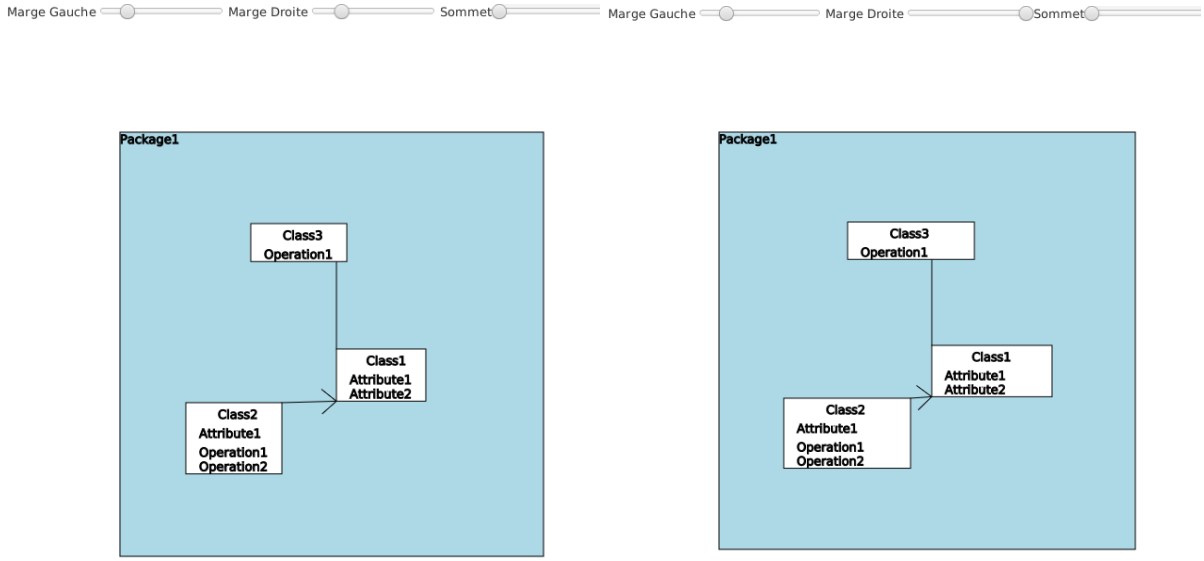


FIGURE 7.4 – Partie du métamodèle UML utilisée par le diagramme de classes

Les interactions avec l'utilisateur sont volontairement limitées aux interactions simples présentées dans la section 7.1. C'est-à-dire les interactions qui ne modifient que la vue sans nécessiter de changement sur le modèle source. Dans notre cas, nous n'autorisons que les déplacements d'éléments graphiques dans la limite de la sémantique d'un diagramme de classes. Par exemple, il n'est pas possible de sortir le `Rectangle` correspondant à une `Class` du `Rectangle` correspondant à son `Package`.

De plus, dans cet exemple, l'utilisateur a la possibilité de modifier certaines constantes utilisées lors de la construction du diagramme (par exemple, la marge à droite et à gauche des attributs d'une classe, ou la distance entre les attributs et les opérations d'une classe). Lorsque l'utilisateur modifie ces constantes (grâce à des *sliders* visibles en haut de la fenêtre), les nouvelles valeurs sont propagées au solveur qui recalcule une solution prenant en compte les nouvelles constantes. Par exemple, dans la Figure 7.5a, le diagramme a une marge à

droite des attributs de taille normale, tandis que dans la Figure 7.5b, la marge à droite a été augmentée au maximum.



(a) Petites marges à droite

(b) Grandes marges à droite

FIGURE 7.5 – Captures d'écran d'un diagramme de classes

Ce visualiseur utilise le DSL interne à Xtend. Une des règles de transformation est donnée dans le Listing 11. Cette règle transforme une `Class` en un `Rectangle` et un `Text`. Elle possède deux sections `to`, une pour chaque élément de sortie. L'initialisation du `Rectangle` se limite à sa couleur de fond (`fill`, ligne 4), son contour (`stroke`, ligne 5) et l'ajout d'interacteurs (`onDrag()` pour le déplacement et `onPress()` pour le clic, ligne 5). On notera que les interacteurs ne sont pas détaillés ici. Ils sont écrits en Xtend classique utilisant l'API JavaFX. On notera aussi que les propriétés `fill` et `stroke` sont statiques, mais calculées en utilisant une opération active (`name.collect[...]`) parce que cette version du DSL ne permettait pas d'établir de lien entre une propriété et une constante. Une façon de contourner cette limitation est de faire comme dans la règle et d'utiliser une propriété du modèle source pour calculer la valeur constante (ligne 4). Les propriétés `mouseTransparent` (ligne 9), `origin` (ligne 9) et `stroke` (ligne 8) du `Text` sont calculées de façon similaire. À la ligne 8, on associe la propriété `text` du `Text` à la propriété `name` de la `Class`.

Ces deux sections définissent les éléments de la vue qui sont créés lorsqu'une `Class` est affichée. Cependant, comme on peut le voir, les positions du `Rectangle` et du `Text` ne sont pas affectées par cette partie de la règle de transformation. C'est dans la section suivante que sont définies les contraintes qui assureront la cohérence visuelle des éléments générés.

Les contraintes sont définies dans la seconde partie de la règle (lignes 11 à 41). Cette section est en fait une fonction anonyme Xtend prenant autant de paramètres que la règle a

```
1  public val Rule2<Class, Rectangle, Text> Class2Rectangle =
2    new Rule2Constrained<Class, Rectangle, Text>(
3      to(JFX.Rectangle,
4        fill <=> name.collect[Paint.valueOf("white")]
5        ,stroke <=> name.collect[Paint.valueOf("black")], onDrag() ,onPress()
6      ),
7      to(JFX.Text,
8        stroke <=> name.collect[Paint.valueOf("black")], text <=> name,
9        origin <=> name.collect[VPos.TOP],mouseTransparent <=> name.collect[true]
10     )
11   ).constraints[c, r, t| #[
12     r.topLeft.stay(Strength.MEDIUM)
13     ,r.topLeft.suggest(350,350)
14     ,r.height2.minimize(Strength.MEDIUM)
15     ,r.width2 >= t.width + CLASSE_NAME_OFFSET * 2
16     ,r.bottomRight.x >= MARGE_HORIZONTALE_DROITE +
17     c._ownedAttribute.collectTo(Attribute2Text).rectangle_.bottomRight.x
18     ,r.bottomRight.x >= MARGE_HORIZONTALE_DROITE +
19       c._ownedOperation.collectTo(Operation2Text).a.rectangle_.bottomRight.x
20     ,r.width2.minimize(Strength.MEDIUM)
21     ,t.rectangle.bottomLeft.x >= r.topLeft.x + CLASSE_NAME_OFFSET
22     ,t.rectangle.bottomLeft.x >= r.topLeft.x + (r.width2 - t.width)/2
23     ,t.rectangle.topLeft.y == r.topLeft.y + MARGE_CLASSE
24     ,r.bottomRight.y >= t.rectangle.bottomRight.y
25     ,r.bottomRight.y >=
26       c._ownedAttribute.collectTo(Attribute2Text).a.rectangle_.bottomRight.y
27     ,r.bottomRight.y >=
28       c._ownedOperation.collectTo(Operation2Text).a.rectangle_.bottomRight.y
29     ,c._ownedAttribute.collectTo(Attribute2Text).a.rectangle_.topLeft.x ==
30       r.topLeft.x + MARGE_HORIZONTALE_GAUCHE
31     ,c._ownedAttribute.collectTo(Attribute2Text).a.rectangle_.topLeft.y >=
32       t.rectangle.bottomRight.y+MARGE_CLASSE_ATTRIBUTS
33     ,c._ownedOperation.collectTo(Operation2Text).a.rectangle_.topLeft.x ==
34       r.topLeft.x + MARGE_HORIZONTALE_GAUCHE
35     ,c._ownedOperation.collectTo(Operation2Text).a.rectangle_.topLeft.y >=
36       c._ownedAttribute.collectTo(Attribute2Text).a.last.rectangle_.bottomLeft.y
37       + MARGE_ATTRIBUTS_OPERATIONS
38     ,c._ownedOperation.collectTo(Operation2Text).a.rectangle_.topLeft.y >=
39       t.rectangle.bottomRight.y + MARGE_CLASSE_OPERATIONS
40   ]
41 ]
```

Listing 11 – Règle de transformation responsable de la génération du `Rectangle` et du `Text` correspondant à une `Class`

de paramètres et de résultats. Par exemple, ici `c` correspond à la `Class` source et `r` et `t` aux `Rectangle` et au `Text` de la cible.

On peut noter plusieurs types de contraintes parmi celles présentes dans cette règle. La première, ligne 12, est une contrainte *stay* comme décrite dans la section 5.2. Cette contrainte assure que les variables auxquelles elle est appliquée ne varient pas trop d'une résolution à l'autre. Pour un diagramme, cela limite les changements de position et dimension non désirés. La seconde contrainte, ligne 13, n'est pas réellement une contrainte, mais plutôt une façon simplifiée de suggérer une valeur initiale pour une variable. Elle est utilisée ici pour suggérer une position initiale au `Rectangle` correspondant à la `Class`. Vient ensuite, ligne 14, une contrainte de minimisation de variable, cette contrainte est réécrite en `r.height2 = 0`. On notera qu'une

force `Strength.MEDIUM` est associée à cette contrainte. Cette force est utilisée par le solveur (Cassowary) lorsqu'il n'existe pas de solutions vérifiant toutes les contraintes. Dans ce cas, le solveur préférera violer des contraintes ayant une force plus faible. Par défaut, les contraintes de notre langage sont requises : elles ne doivent pas être violées.

La contrainte suivante, ligne 15, est une contrainte plus classique : elle contraint le `Rectangle` correspondant à la `Class` à être assez large pour contenir son `Text`. Dans cette contrainte, on notera l'utilisation d'une constante externe, `CLASSE_NAME_OFFSET`. Cette constante est liée à un des *sliders* visibles au-dessus du diagramme. Enfin, on notera l'utilisation de l'opération `collectTo()` dans les expressions formant les contraintes (par exemple, ligne 17). Cette opération particulière permet de faire référence aux résultats de l'application d'une règle de transformation (dont le nom est passé en paramètre). Par exemple,
`c._ownedAttribute.collectTo(Attribute2Text).a` fait référence au premier élément de sortie de la règle de transformation (`b` fait référence au second et ainsi de suite) `Attribute2Text` appliquée aux éléments de `c._ownedAttribute`.

Les lignes 21 à 24 placent le `Text` correspondant au `name` de la `Class` en haut au centre du `Rectangle` correspondant à la `Class`. Le reste de la transformation (lignes 25 à 39) place les éléments correspondant aux `Attribut`s et `Operation`s de la `Class`.

7.2.2 Diagramme d'activité

L'objectif de ce cas d'étude est de générer un diagramme d'activité. Comme pour le visualiseur de diagrammes de classes cette transformation utilise une partie du métamodèle UML, détaillée dans la Figure 7.6, et génère un diagramme conforme au métamodèle de formes géométriques présenté dans la Figure 7.1. La Figure 7.7 montre une capture d'écran d'un diagramme d'activité généré par l'outil.

Une `Activity` est un `NamedElement` possédant des `ActivityEdge`s (`edge`) et des `ActivityNode`s (`ownedNode`). Un `ActivityEdge` est une classe abstraite et aussi un `NamedElement` et référence deux `ActivityNode`s (`source` et `target`). Le `ControlFlow` est une concrétisation d'un `ActivityEdge`. Un `ActivityNode` est une classe abstraite et aussi un `NamedElement`. Il existe 7 concrétisations d'un `ActivityNode` représentant différents nœuds du diagramme d'activité, `ActivityFinalNode`, `DecisionNode`, `ForkNode`, `InitialNode`, `JoinNode`, `MergeNode` et `OpaqueNode`.

Le Listing 12 montre une des règles de transformation permettant de générer ce diagramme. Cette règle transforme une `Activity` en un `Rectangle` et un `Text` auxquels sont ajoutées des contraintes. Comme pour la règle de transformation précédente (Listing 11), celle-ci est composée de deux parties. La première est responsable de la spécification des éléments de sortie (le `Rectangle` et le `Text`) et la seconde est responsable des contraintes.

Comme pour la règle précédente, la première section est succincte, on y retrouve les mêmes relations. Par exemple, on calcule le `fill` du `Rectangle` de la même façon (ligne 4). Il en va de même pour le `Text`.

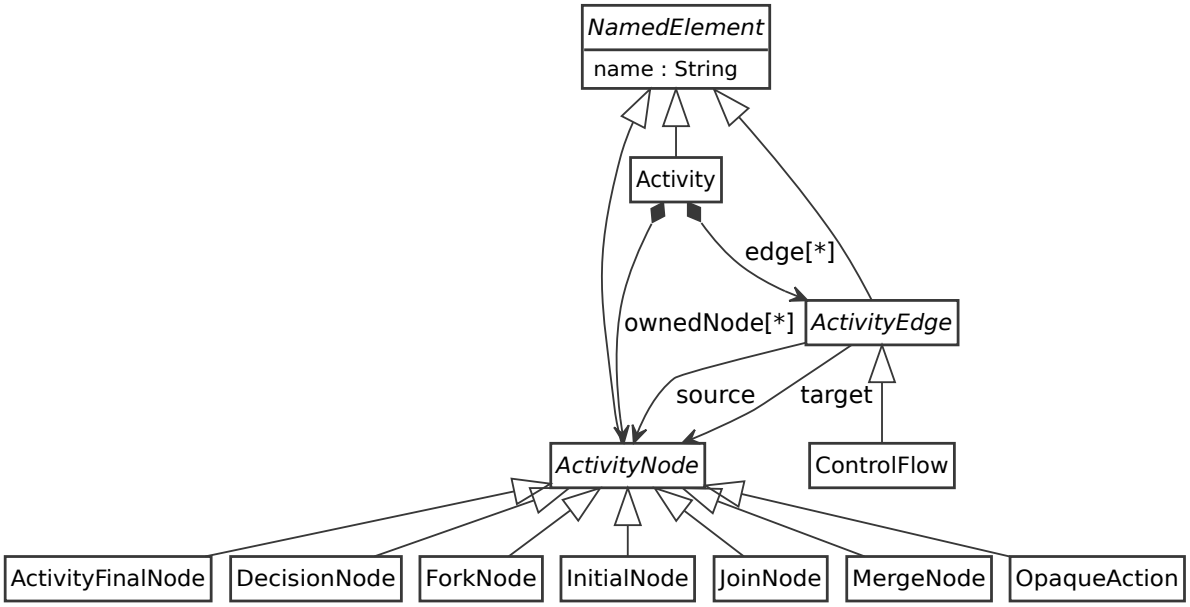


FIGURE 7.6 – Partie du métamodèle UML utilisée pour le diagramme d'activités

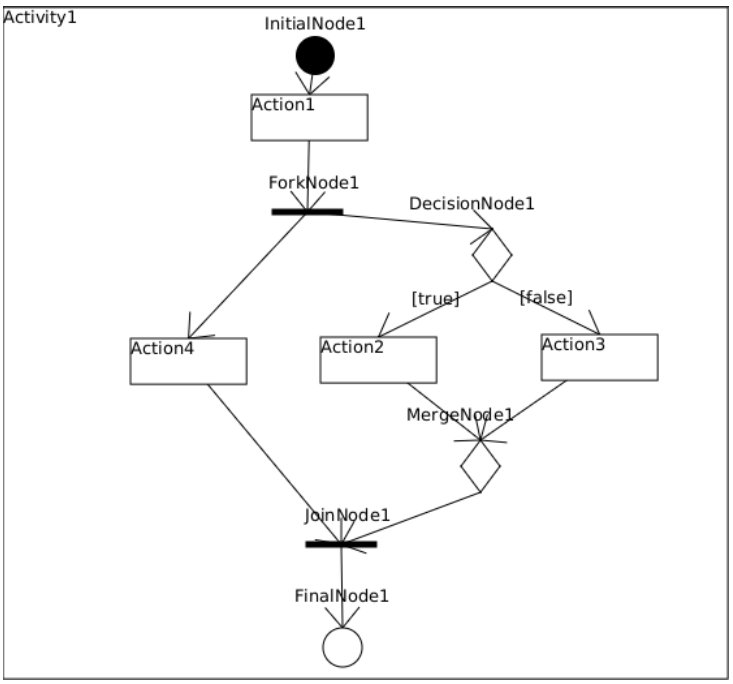


FIGURE 7.7 – Capture d'écran d'un diagramme d'activité

La seconde section est plus intéressante. Contrairement à la règle précédente, elle est ici séparée en deux parties. La seconde partie est similaire à celle de la première règle, c'est une liste de contraintes. Cependant, la partie avant cette liste de contraintes n'était pas présente

```
1  public val Rule2<Activity, Rectangle, Text> Activity2Rect =
2  new Rule2Constrained<Activity, Rectangle, Text>(
3    to(JFX.Rectangle,
4      fill <=> name.collect[Paint.valueOf("white")]
5      ,stroke <=> name.collect[Paint.valueOf("black")]
6      ,onPress
7      ,onDrag
8    ),
9    to(JFX.Text,
10     text <=> name
11     ,origin <=> name.collect[VPos.TOP]
12   )
13 ).constraints[a, r, t|
14   val circles = a._ownedNode.select(InitialNode).
15           collectTo(InitialNode2Circle).a.circle
16   val activityFinalNodeCircles = a._ownedNode.select(ActivityFinalNode).
17           collectTo(ActivityFinalNode2Circle).a.circle
18   val decisionNodeDiamonds = a._ownedNode.select(DecisionNode).
19           collectTo(DecisionNode2Diamond).a.rectangle3
20   val mergeNodeDiamonds = a._ownedNode.select(MergeNode).
21           collectTo(MergeNode2Diamond).a.rectangle3
22   val forkNodeLines = a._ownedNode.select(ForkNode).
23           collectTo(ForkNode2Line).a._line
24   val joinNodeLines = a._ownedNode.select(JoinNode).
25           collectTo(JoinNode2Line).a._line
26   val opaqueActionRects = a._ownedNode.select(OpaqueAction).
27           collectTo(OpaqueAction2Rect).a.rectangle_
28 #[
29   r.topLeft.stay(Strength.MEDIUM),
30   t.topLeft == r.topLeft,
31   r.width2 >= t.width + 500,
32   r.height2 >= t.height + 500,
33   r.contains(circles),
34   r.contains(activityFinalNodeCircles),
35   r.contains(decisionNodeDiamonds),
36   r.contains(mergeNodeDiamonds),
37   r.contains(opaqueActionRects),
38   (opaqueActionRects.topLeft - r.topLeft).stay(Strength.WEAK)
39 ]]
```

Listing 12 – Règle transformant une `Activity` en un `Rectangle` et un `Text`

dans la première règle de transformation.

La section contraintes des règles de transformation dans ce DSL Xtend est en fait une fonction anonyme retournant une liste de contraintes. La liste est notée entre `#[` et `]` en Xtend, par exemple `#[1,2,3]` correspond à la liste contenant les entiers $1, 2$ et 3. Il est donc possible d'exécuter du code Xtend avant de retourner la liste de contraintes. C'est ce qui est fait ici pour déclarer des variables (Xtend) afin de simplifier l'écriture et la lecture des contraintes suivantes (lignes 14 à 27).

Les contraintes suivantes sont similaires à celles trouvées dans la règle `Class2Rectangle`. On trouve une contrainte *stay* à la ligne 29 suivie de quatre contraintes positionnant le `Text` en haut à gauche du `Rectangle` correspondant à une `Activity` (lignes 30 à 32). Puis, cinq contraintes *contains* (lignes 33 à 37). Ces contraintes *contains* font partie de la couche d'abstraction géométrique présente dans le DSL Xtend. Elle permettent de contraindre un objet

géométrique (par exemple, un rectangle, une ligne ou un point) à être contenu dans un autre objet géométrique assimilable à un rectangle. Dans ce cas, on peut voir que cette contrainte est utilisée pour assurer que les différents éléments d'une activité restent contenus dans celle-ci. Enfin, ligne 38, on trouve une autre contrainte *stay*. Elle n'est pas directement appliquée à un point ou à une variable, mais à un point calculé à partir de deux autres points. Utiliser une contrainte *stay* de cette façon permet de contrainte la position relative d'un point par rapport à un autre. Ainsi, lorsque le `Rectangle` correspondant à l'`Activity` bouge le `Rectangle` correspondant à l'`OpaqueAction` se déplacera de façon à ce que sa position relative par rapport au `Rectangle` de l'`Activity` ne change pas.

7.2.3 Diagramme de cas d'utilisation

L'objectif de ce visualiseur est de créer un diagramme de cas d'utilisation à partir d'une partie du métamodèle UML. Comme pour les diagrammes précédents, la Figure 7.8 présente la partie du métamodèle UML utilisée. Le métamodèle cible, Figure 7.1, est le même que pour les diagrammes précédents. La Figure 7.9 montre une capture d'écran d'un diagramme de cas d'utilisation généré par le visualiseur.

Le métamodèle utilisé dans le diagramme de cas d'utilisation recouvre en partie celui utilisé dans le diagramme de classes. Par exemple, les classes `Association`, `Generalization`, `Package`, `PackageableElement`, `NamedElement`, `Property` et `Type` y sont aussi présentes. On notera l'absence de la classe `Class`, remplacée par sa classe mère abstraite `Classifier`, auquel la `Generalization` se ratache. En plus de l'`Association`, le `Classifier` possède deux autres concrétisations, l'`Actor` et le `UseCase`.

Comme pour les diagrammes précédents, ce visualiseur utilise le DSL interne Xtend, les règles du Listing 13 sont donc composées de deux parties, la première pour la gestion des éléments de sortie et leurs propriétés et la seconde pour les contraintes. La première règle `ActorRule` transforme un `Actor` en un `ActorFX` et un `Text` donnant le nom de l'`Actor`. La seconde, `AssociationRule`, génère une `Line` à partir d'une `Association`. De plus, après les règles, une fonction `useCaseOrActorCenter(IBox<Type> b)` permet de facilement calculer le centre d'une figure associée à un `Actor` ou à un `UseCase`.

Les premières sections des deux règles de transformation sont classiques et ressemblent en tout point aux règles précédentes. On notera l'utilisation de la classe `ActorFX` pour la première règle. Il est relativement facile d'ajouter de nouvelles formes aux diagrammes. Cela se fait de la même façon qu'en JavaFX. Il est ainsi possible de construire des formes qui ne pourraient pas être aisément réalisées avec les formes primitives mises à disposition. Dans ce cas, la figure humaine aurait pu être créée directement dans le DSL, mais sa présence ici permet d'illustrer l'utilisation de formes spécifiques.

Les contraintes de la règle `ActorRule` sont similaires aux règles précédentes. On notera l'utilisation de l'abstraction géométrique pour placer le `Text` sous l'`ActorFX`. Dans ces contraintes, l'`ActorFX` est considéré comme un `Rectangle`, ce qui permet d'utiliser l'abstraction

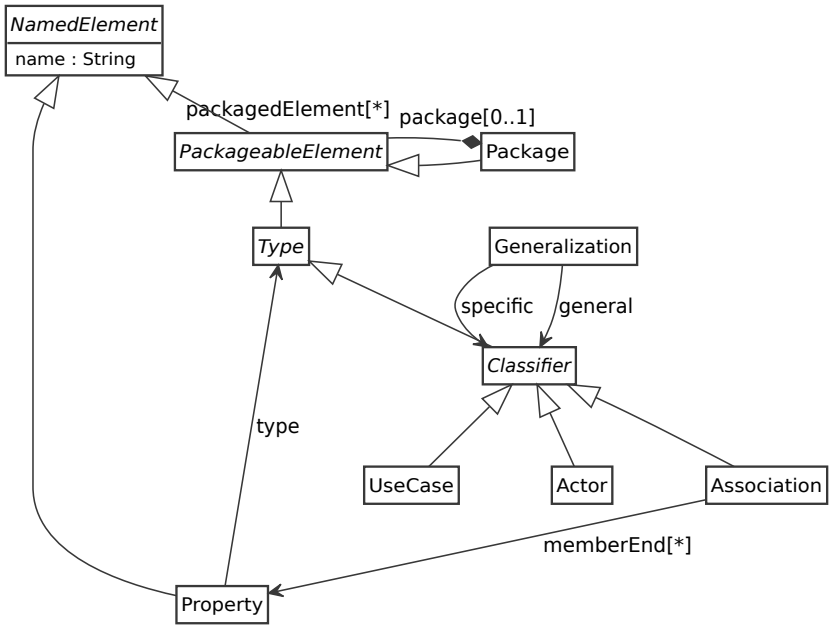


FIGURE 7.8 – Partie du métamodèle UML utilisée par le diagramme de cas d'utilisation

géométrique (comme l'utilisation de `af.rectangle.topLeft` ou `af.rectangle.bottom.middle`).

Les contraintes de la seconde règle sont aussi assez classiques. Les extrémités de la `Line` doivent coïncider avec les centres des `Rectangle`s correspondant aux `Actor`s et `UseCase`s que l'`Association` relie. Cependant, dans les expressions utilisées, on remarque plusieurs opérations particulières. Les premières, `first` et `second` sont deux opérations qui retournent respectivement le premier ou le deuxième élément d'une collection. La seconde `useCaseOrActorCenter` retourne le centre d'un `Actor` ou d'un `UseCase`. On trouve sa définition aux lignes 36 à 44. Parce que ce DSL est construit autour d'un langage généraliste, il est possible d'ajouter des méthodes écrites avec du code impératif aux transformations. Ici, la méthode `useCaseOrActorCenter` applique différentes règles suivant le type d'objet dans une collection.

7.3 Diagramme de séquence

Pour ce dernier cas d'étude utilisant le DSL Xtend, l'objectif est de générer des diagrammes de séquence à partir du métamodèle UML. Comme pour les exemples précédents, nous utilisons le métamodèle de formes géométriques inspiré de JavaFX présenté plus tôt (Figure 7.1). Comme pour les diagrammes précédents, nous ne considérons qu'une version simplifiée du métamodèle UML. Le sous-ensemble du métamodèle UML que nous considérons est montré dans Figure 7.10.

Une `Interaction` correspond, dans notre cas, à une suite de `Message`s entre une ou plusieurs `Lifeline`s. Une `Interaction` possède des `Lifeline`s (`lifelines`), des `Message`s

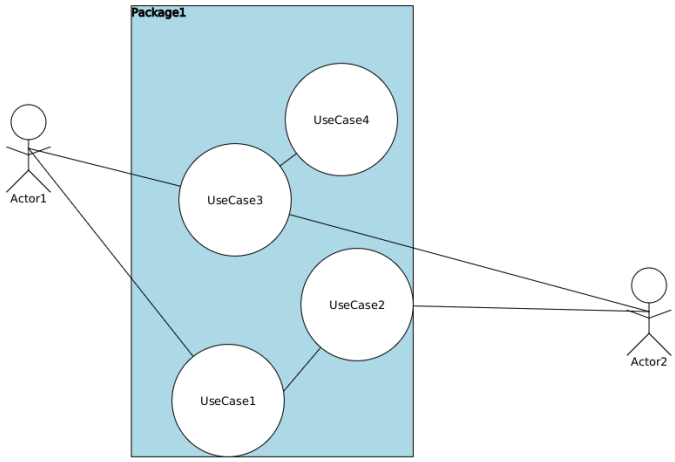


FIGURE 7.9 – Capture d'écran d'un diagramme d'utilisation

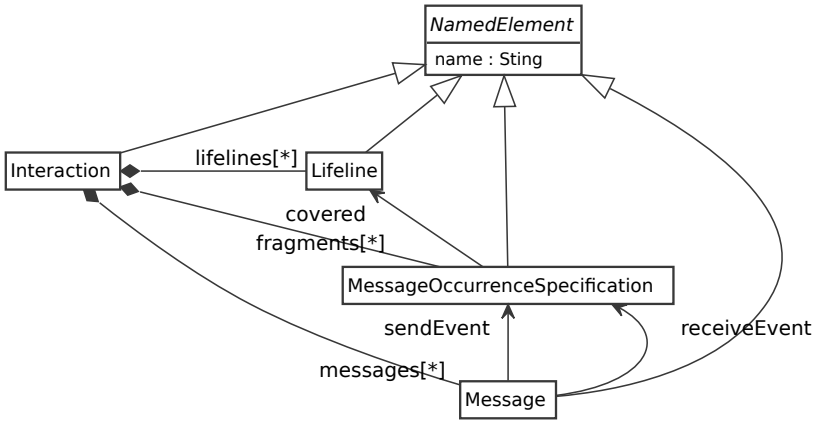


FIGURE 7.10 – Partie du métamodèle UML utilisée dans le diagramme de séquence

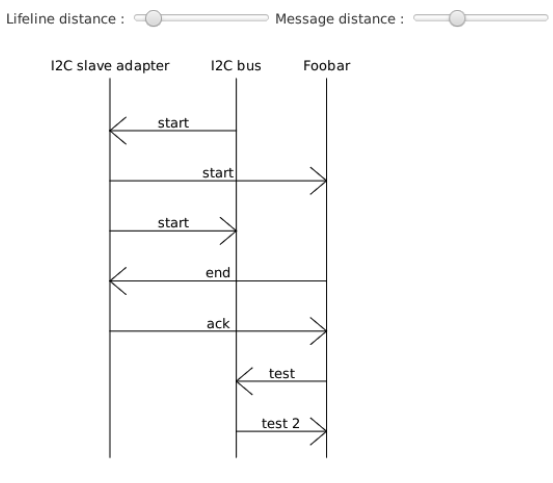
(`messages`) et des `MessageOccurenceSpecification` (`fragments`). Une `Lifeline` possède un `name` et représente un participant de l'`Interaction`. Un `Message` possède lui aussi un `name` et deux propriétés `receiveEvent` et `sendEvent` référençant deux `MessageOccurenceSpecification`. Un `Message` représente une interaction entre deux participants de l'`Interaction`. Enfin, une `MessageOccurenceSpecification` possède un `name` et une propriété `covered` référençant une `Lifeline`. Une `MessageOccurenceSpecification` permet de relier les `Message`s aux `Lifeline`s qu'ils relient.

On peut voir, dans la Figure 7.11, trois captures d'écran d'un diagramme de séquence généré. Comme pour le visualiseur de diagramme de classes, ce diagramme possède plusieurs *sliders* permettant de régler différentes constantes comme la distance entre les `Lifeline`s ou les `Message`s. Par exemple, dans la Figure 7.11a, la distance entre les `Lifeline`s est faible, tandis que dans la Figure 7.11b la distance a été augmentée. De plus, par une touche du

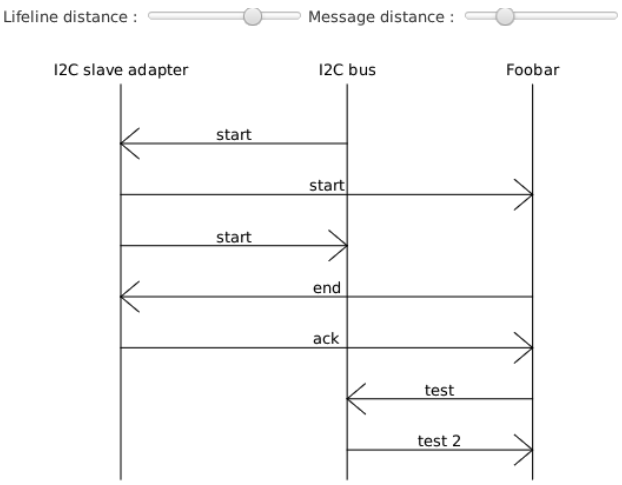
```
public val Rule2<Actor, ActorFX, Text> ActorRule =
new Rule2Constrained<Actor, ActorFX, Text>(
  to(
    JFX.ActorFX,
    onPress,
    onDrag
  ),
  to(
    JFX.Text,
    text <=> name,
    origin <=> name.collect[VPos.TOP]
  )
).constraints [ a, af, t |
  #[
    af.rectangle.topLeft.stay(Strength.WEAK),
    t.top.middle == af.rectangle.bottom.middle,
    af.rectangle.topLeft.suggest(200, 200),
    af.rectangle.width.suggest(50),
    af.rectangle.height.suggest(100)
  ]
]

public val Rule1<Association, Line> AssociationRule =
new Rule1Constrained<Association, Line>(
  to(
    JFX.Line,
    stroke <=> name.collect[Paint.valueOf("black")]
  )
).constraints [ a, l |
  #[
    l.start == a._memberEnd.first.type.useCaseOrActorCenter,
    l.end == a._memberEnd.second.type.useCaseOrActorCenter
  ]
]

def useCaseOrActorCenter(IBox<Type> b) {
  if (b.get(0) instanceof Actor) {
    b.select(Actor).collectTo(ActorRule).a.rectangle2.center as Point
  } else if (b.get(0) instanceof UseCase) {
    b.select(UseCase).collectTo(UseCaseRule).a.circle.center
  } else {
    (0.0).point(0.0) as Point
  }
}
```

Listing 13 – Règles transformant un `Actor` en un `Rectangle` et un `Text` et règles transformant une `Association` en `Line`

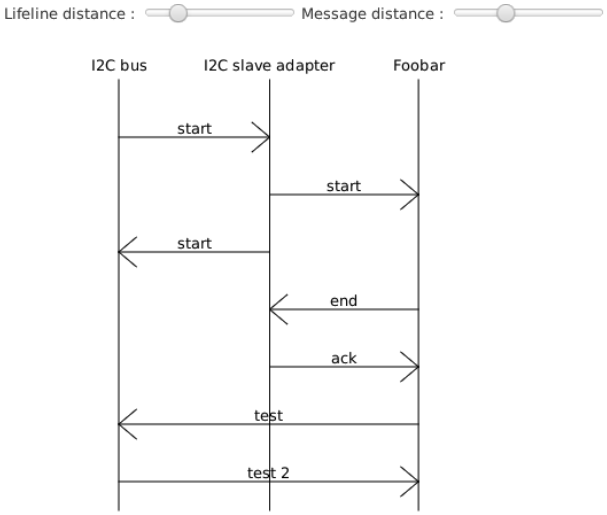
clavier, il est possible de permuter aléatoirement l'ordre des `Lifeline`s du diagramme. Presser la touche espace entraîne la mise à jour du modèle source, laquelle entraîne, par propagation grâce à la transformation incrémentale, la mise à jour de la vue et des contraintes assurant sa cohérence (Figure 7.11c). Ces interactions ne sont pas destinées à un utilisateur final, mais permettent de montrer que les contraintes fonctionnent comme attendu et que la mise à jour du modèle entraîne bien la mise à jour du diagramme. La possibilité de configurer les distances entre `Lifeline`s et entre `Message`s pourrait cependant être offerte à un utilisateur via une page de configuration.



(a) Petits espaces entre les `Lifeline`s



(b) Grands espaces entre les `Lifeline`s



(c) Changement de l'ordre des `Lifeline`s

FIGURE 7.11 – Captures d'écran d'un diagramme de séquence

Comme les visualiseurs précédents, cette règle de transformation est composée de deux parties, la première spécifiant les éléments à créer ainsi que les expressions pour calculer leurs propriétés et la seconde contenant les contraintes. Cependant ici, la première contient la définition d'un interacteur (lignes 7 à 15). Cet interacteur permet l'édition de la propriété `name` d'une `Lifeline`. On voit qu'un interacteur est défini comme une fonction anonyme écrite en Xtend prenant en paramètre, `e` l'évènement (ici un clic), `s` l'élément source (ici une `Lifeline`) et `t` l'élément cible (ici un `Text`). Un des avantages d'utiliser un DSL interne à Xtend est la possibilité d'utiliser du code impératif au sein de la transformation.

Dans ce cas d'étude, les `Lifeline`s sont représentées par des `Line`s verticales au sommet

```
1  public val Rule2<Lifeline, Line, Text> Lifeline =
2  new Rule2Constrained<Lifeline, Line, Text>(
3    to(JFX.Line),
4    to(JFX.Text,
5      text <=> name
6      ,origin <=> name.collect[VPos.TOP]
7      ,JFXInteractions.onClick[e, s, t|{
8        if (e.button == MouseButton.PRIMARY && e.clickCount == 2) {
9          val l = t as Text
10         val ll = s as Lifeline
11         val dialog = new TextInputDialog(l.text)
12         dialog.title = "New label"
13         dialog.showAndWait.ifPresent[ll.name = it]
14       }
15     ]{
16   )
17 ).constraints[ll, l, t | #[
18   l.start.x >= 100{
19   ,l.start.x == ll.previous.line.start.x{
20             + ll.previous.text.width / 2
21             + t.width / 2
22             + LIFELINE_DISTANCE
23   ,l.start.y == 50
24   ,l.end.x == l.start.x
25   ,l.end.y == ll.previous.line.end.y{
26   ,t.bottom.middle.dy(5) == l.start{
27   ]
28 ]
```

Listing 14 – Règle transformant une `Lifeline` en une `Line` et un `Text`

desquelles sont placés les `name`s des `Lifeline`s. Entre ces `Lifeline`s sont placées des `Arrow`s représentant les `Message`s. Les `MessageOccurenceSpecification`s deviennent des points (une `Line` dont les deux extrémités sont confondues) servant à placer les `Arrow`s correspondant aux `Message`s.

La section contrainte est similaire aux règles de transformation précédentes. On notera que comme il n'est pas possible de déplacer les `Lifeline`s à la souris (l'interacteur onDrag n'est pas présent) il n'est pas nécessaire d'ajouter des contraintes *stay*. Comme pour le diagramme de classes, l'utilisation de constantes permet à l'utilisateur de modifier des paramètres de construction du diagramme (ligne 19). Le `Text` est placé au-dessus de la `Line` en utilisant la couche d'abstraction géométrique. L'expression `t.bottom.middle.dy(5)` calcule un point situé 5 unités sous le centre du segment du bas du `Text`.

Dans un diagramme de séquence, les `Lifeline` et `Message`s sont ordonnés. Il est donc plus simple de les positionner les unes par rapport aux autres. Par exemple, ligne 25 contraint une `Line` à finir au même niveau que la `Line` correspondant à la `Lifeline` précédente.

On notera l'utilisation de l'opération `previous` qui est une opération active particulière. Elle fonctionne en deux temps. Dans un premier temps, elle prend une collection ordonnée d'éléments qu'elle stocke. Puis, dans un second temps, il est possible d'utiliser cette opération pour retourner l'élément précédent (s'il existe). Comme l'opération est incrémentale, un changement sur la collection peut entraîner le changement des éléments précédents calculés.

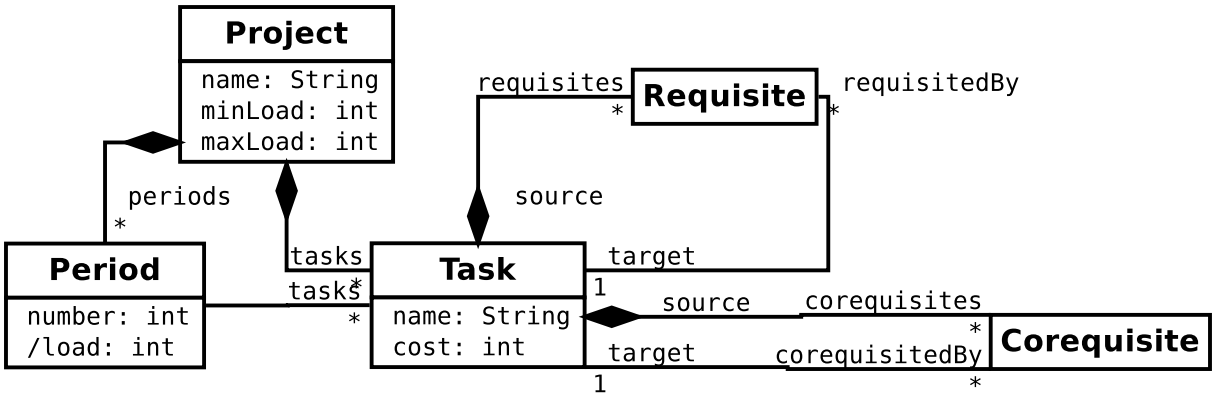


FIGURE 7.12 – Métamodèle de planification

Les `Lifeline`s sont placées les unes après les autres selon l'axe `x`. Deux contraintes sont suffisantes pour cela. La contrainte ligne 18 place la première `Lifeline` à 100 unités à gauche de l'écran. Ceci est suffisant, car, par défaut, Cassowary minimise les valeurs de toutes les variables de décision. Ensuite, ligne 19, cette contrainte place la `Line` correspondant à la `Lifeline` relativement à la `Line` correspondant à la `Lifeline` précédente.

7.4 Conception et visualisation de tâches

Pour ce cas d'étude, nous considérons un exemple un peu plus complexe. En effet, ce cas d'étude se compose de deux sous problèmes. Le premier, consiste à affecter des tâches à des périodes de façon à ce que l'affectation respecte un ensemble de contraintes métier. Bien que très simplifié dans cet exemple, ce type de problème apparaît dans de nombreuses situations telles que la répartition de ressources lors de la planification d'un développement logiciel.

Pour modéliser ce problème, nous proposons le métamodèle suivant (Figure 7.12). Un `Project` contient un ensemble de `Period`s et de `Task`s. Chaque `Task` est identifiée par un `name` et un possède un `cost`. En plus de son `name` et de son `cost`, une `Task` peut avoir une liste `requisites` de `Requisite`s et une liste `corequisites` de `Corequisite`s. Les `Requisite`s et `Corequisite`s font référence à deux `Task`s, une `source` et une `target`. La `Task` cible d'un `Requisite` (respectivement d'un `Corequisite`) doit être affectée à une `Period` ayant lieu après la (respectivement au cours de la même) `Period` à laquelle la `Task` `source` est affectée.

Chaque `Period` est identifiée par un `number` aussi utilisé pour indiquer l'ordre temporel des `Period`s. Un nombre plus petit indique une `Period` plus ancienne. Donc pour deux `Period`s données, celle ayant le `number` le plus faible est antérieure à l'autre. Chaque `Period` possède aussi une propriété dérivée, appelée `load`, calculée à partir de la somme des `cost`s des `Task`s affectées à la `Period`. Enfin, un `Project` possède, en plus d'un `name`, deux propriétés `minLoad` et `maxLoad` donnant une limite basse et haute de `load` pour les `Period`s qui lui sont affectées.

7.4.1 Génération d'un planning valide

Bien que relativement simple, ce métamodèle est suffisamment complexe pour permettre l'introduction de contraintes pertinentes. On appelle *planning* d'un `Project` une affectation de toutes ses `Task`s à ses `Period`s. Il existe de nombreux plannings possibles pour un `Project` donné. Cependant, de nombreux plannings ne sont pas pertinents. Par exemple, un planning dans lequel toutes les `Task`s seraient affectées à la première `Period` ne l'est pas. Il existe un nombre de contraintes à respecter. Par exemple, il faut que la `load` de toutes les `Period`s soit comprise entre les `minLoad` et `maxLoad` définis par le `Project`. Pour être pertinent, un planning doit, en plus d'être conforme à son métamodèle, vérifier un ensemble de contraintes. À partir du métamodèle de projet défini plus tôt nous définissons les contraintes de validité suivantes :

Règle 1 : Charge appropriée. La `load` de chaque `Period` doit être comprise entre `minLoad` et `maxLoad` définis dans le `Project`.

Règle 2 : Respects des précédences. Tous les `requisites` d'une `Task` doivent être affectés à des `Period`s précédentes.

Règle 3 : Respect des simultanéités. Une `Task` et ses `corequisite`s doivent être affectés à la même `Period`.

Il est possible d'encoder ces règles en contraintes et ainsi faire résoudre ce problème de planification par un solveur. Cependant, il faut convertir les relations des `Task`s aux `Period`s en variables utilisables par le solveur.

Après que le solveur ait trouvé un planning initial valide, l'utilisateur peut explorer l'ensemble des plannings valides en le modifiant. Après chaque mise à jour, le solveur est sollicité pour recalculer un nouveau planning prenant en compte les modifications et vérifiant toutes les contraintes.

7.4.2 Visualisation d'un planning

La seconde partie de ce cas d'étude s'intéresse à la visualisation et l'édition d'un planning généré comme décrit dans la sous-section précédente. La Figure 7.13 montre une capture d'écran du diagramme généré pour un `Project` ne contenant que trois `Period`s et `Task`s. Ces trois `Task`s sont nommées `000`, `001` et `002`. La `Task 002` requiert les `Task`s `000` et `001`. Ces trois `Task`s doivent être affectées à trois `Period`s nommées `Period 1`, `Period 2` et `Period 3`. On notera `P1`, `P2` et `P3` les périodes `Period 1`, `Period 2` et `Period 3`. Dans le planning de la Figure 7.13, `000` est affectée à `P1`, `001` à `P2` et `002` à `P3`. Ce planning est valide, car `000` et `001` précèdent `002`.

Dans la sous-section 7.4.1, nous avons vu un ensemble de contraintes métiers permettant de construire des plannings *valides*. Une fois calculé, un tel planning peut être affiché sous la forme d'un diagramme. Pour être valide, ce diagramme doit lui aussi respecter un ensemble de règles de construction. En voici quelques-unes :

Règle 1 : Placement du code d'une `Task`. Le `Text` affichant le `name` d'une `Task` doit être contenu par le `Rectangle` correspondant à la `Task`. Ou encore, le coin supérieur gauche du `Text` doit correspondre avec le coin supérieur gauche du `Rectangle` et le `Rectangle` doit être assez large et haut pour que le `Text` ne dépasse pas.

Règle 2 : Placement des `Task`s. Le `Rectangle` correspondant à une `Task` doit toujours être contenu dans le `Rectangle` correspondant à la `Period` à laquelle la `Task` est affectée.

Règle 3 : Placement des `Period`s. Les `Rectangle`s correspondant aux `Period`s doivent être placés les uns après les autres, de façon à ce que le `Rectangle` correspondant à une `Period` ayant le `number` n soit situé directement à la droite du `Rectangle` correspondant à la `Period` ayant le `number` $n-1$ et directement à gauche du `Rectangle` correspondant à la `Period` ayant le `number` $n+1$ (si ces deux `Period`s existent).

Ces contraintes géométriques permettent de construire un diagramme cohérent duquel il est possible d'extraire les informations nécessaires pour comprendre le planning. Comme pour les diagrammes précédents, l'utilisateur peut interagir avec le diagramme. Il peut par exemple déplacer les `Task`s au sein d'une même `Period`. Ce type d'interaction correspond à des interactions simples que nous avons décrites précédemment. Ces interactions ne modifient pas le planning, elles ne modifient que la visualisation.

Cependant, l'utilisateur peut aussi vouloir modifier le planning. Dans ce cas d'étude, nous permettons à l'utilisateur de déplacer une `Task` au-delà des limites de la `Period` à laquelle la `Task` est affectée. Lorsque ce comportement est détecté, un interacteur différent est utilisé. Cet interacteur va suggérer un changement au modèle source (i.e., le modèle de projet dont le métamodèle a été donné à la Figure 7.12). Cette suggestion est envoyée au premier solveur, qui va recalculer un nouveau planning à partir du planning courant et de la suggestion. Puis ce nouveau planning est utilisé par la seconde transformation qui est responsable de la génération du diagramme.

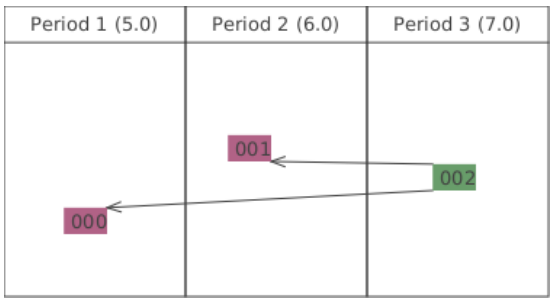


FIGURE 7.13 – Capture d'écran d'un planning valide

7.4.3 Problèmes de contraintes sous-jacents

Dans les sous-sections 7.4.1 et 7.4.2 nous avons présenté deux problèmes liés. Le premier consiste à créer des plannings valides tandis que le second consiste à afficher ces plannings sous la forme de diagrammes. Cependant, dans ces sous-sections, les descriptions de la validité des modèles considérés sont exprimées en langue naturelle. Dans cette sous-section, nous proposerons une traduction de ces contraintes dans un formalisme plus proche des contraintes utilisées par les solveurs. Par souci de lisibilité, nous avons ignoré les forces appliquées aux contraintes. De plus, les variables sont nommées d'après la propriété qu'elles représentent. Par exemple, la variable `p1.r.x` correspond à la propriété `x` et de l'élément `p1.r`.

```
p1.r.y = 0, p1.r.x = 0.0*p1.r.width, p1.l.startX = p1.r.x,
p1.l.endX = p1.r.x + p1.r.width, p1.l.endY = p1.l.startY,
p1.l.startY = p1.t.y + 15.1 + 5,
p1.r.width = 125, p1.r.height = 200,
p1.t.x = p1.r.x + p1.r.width/2 - 88.3/2, p1.t.y = p1.r.y+5,

t000.r.x >= 0,
t000.r.width = 1.2 * 24.8,
t000.r.height = 1.2 * 15.1,
t000.t.x = t000.r.x + 5, tt000.t.y = t000.r.y + 2,
t000.r.x >= p1.r.x, t000.r.y >= p1.l.startY,
t000.r.x + t000.r.width <= p1.r.x + p1.r.width,
t000.r.y + t000.r.height <= p1.r.y + p1.r.height,

rq_t002_t000.l.fromX = 0, rq_t002_t000.l.fromY = 0,
rq_t002_t000.l.fromX - rq_t002_t000.l.toX = 0,
rq_t002_t000.l.fromY - rq_t002_t000.l.toY = 0,
rq_t002_t000.l.fromX >= t002.r.x,
rq_t002_t000.l.fromX <= t002.r.x + t002.r.width,
rq_t002_t000.l.fromY >= t002.r.y,
rq_t002_t000.l.fromY <= t002.r.y + t002.r.height,
rq_t002_t000.l.toX >= t000.r.x,
rq_t002_t000.l.toX <= t000.r.x + t000.r.width,
rq_t002_t000.l.toY >= t000.r.y,
rq_t002_t000.l.toY <= t000.r.y + t000.r.height

...
```

Listing 15 – Partie des contraintes graphiques de la Figure 7.13

Ce cas d'étude est développé en ATL^C, par conséquent il est possible de faire appel à différents solveurs. Pour la partie graphique, nous avons utilisé Cassowary [BBS01], solveur de contraintes hiérarchiques sur des variables réelles. Cassowary est dynamique, il peut recalculer une nouvelle solution à partir d'une ancienne solution. Il est donc particulièrement adapté dans des interfaces graphiques. Cependant, il est limité aux contraintes linéaires. Le Listing 15 donne une partie des contraintes utilisées pour l'affichage du diagramme. Dans la sous-section 7.4.2 nous avons défini nos contraintes de haut niveau dans un formalisme utilisant des concepts géométriques. Comme nos diagrammes sont composés de formes simples (rectangle et lignes par exemple), la majorité de ces concepts géométriques peuvent être exprimés par des contraintes linéaires entre les coordonnées et les dimensions des différents

```
1  000.{period}.number < 3, 000.{period}.number >= 0,
2  001.{period}.number < 3, 001.{period}.number >= 0,
3  002.{period}.number < 3, 002.{period}.number >= 0,
4  002.{period}.number > 000.{period}.number,
5  002.{period}.number > 001.{period}.number,
6
7  sum(5*reify(000.period=0),6*reify(001.period=0),7*reify(002.period=0))>=0,
8  sum(5*reify(000.period=0),6*reify(001.period=0),7*reify(002.period=0))<=15,
9  sum(5*reify(000.period=1),6*reify(001.period=1),7*reify(002.period=1))>=0,
10 sum(5*reify(000.period=1),6*reify(001.period=1),7*reify(002.period=1))<=15,
11 sum(5*reify(000.period=2),6*reify(001.period=2),7*reify(002.period=2))>=0,
12 sum(5*reify(000.period=2),6*reify(001.period=2),7*reify(002.period=2))<=15
```

Listing 16 – Contraintes responsables de la génération du planning de la Figure 7.13

éléments. Par exemple, la Règle 1 qui place le coin supérieur gauche d'un `Text` relativement par rapport à son `Rectangle` devient deux contraintes, une pour chaque coordonnée (ligne 10).

Le Listing 16 liste la totalité des contraintes permettant la génération de plannings valides. Pour résoudre le problème de génération de planning, nous avons utilisé Choco [Pru+17], un solveur adapté à la résolution de problèmes sur des variables entières. Nous utilisons la même notation pour les contraintes que dans le Listing 15. Nous ajoutons de nouveaux opérateurs (`sum` et `reify`) et nous notons entre accolades les variables de relations dans les expressions (lignes 1 à 3). L'opérateur `sum` calcule la somme des expressions passées en paramètres. L'opérateur `reify` crée une variable booléenne dont la valeur est égale à la valeur de vérité de la contrainte passée en paramètre. Par exemple, avec $x = \texttt{reify}(y < 3)$, $x = 0$ si $y \geq 3$ et $x = 1$ sinon. La Règle 1, qui assure qu'une `Period` n'est pas trop chargée, correspond à deux contraintes (par exemple, lignes 7 à 8).

On peut observer des motifs dans les contraintes : on remarque que les mêmes contraintes sont appliquées à différentes figures par exemple. Chaque `Period` a deux contraintes, une pour assurer que sa `load` est supérieure à `minLoad` et une autre pour assurer qu'elle soit inférieure à `maxLoad` (par exemple, pour la `Period` `P1` ces contraintes sont lignes 7 et 8). De même, pour les `Task`s, on retrouve trois contraintes récurrentes : deux pour indiquer l'intervalle des valeurs valides pour encoder les `Period`s (lignes 1, 2 et 3), et une pour encoder les relations de `requisites` entre les `Task`s (lignes 4 et 5). Une autre contrainte, non présente ici, serait nécessaire pour encoder les relations de `corequisites` entre `Task`s . Cependant, on remarquera que ces contraintes ne sont pas définies pour toutes les `Task`s. En effet, certaines sont contextuelles et dépendent de valeur de certaines propriétés (comme les contraintes de `requisites` et `corequisites` qui ne s'appliquent que si la `Task` a des `Task`s dont elle dépend).

La contrainte de contenance utilisée pour décrire le placement des éléments dans la sous-section 7.4.2 peut être réécrite en quatre inéquations linéaires [2]. Par exemple, les quatre contraintes encodant le fait que la `Task` `000` est affectée à la `Period` `P1` sont lignes 11 à 13 du Listing 15. Similairement, un motif apparaît pour les contraintes correspondant aux `Arrow`s

2. Par souci de simplicité, nous considérons que toutes les formes géométriques peuvent être assimilées à des rectangles alignés sur les axes x et y du plan.

représentant les `Requisite`s. Les deux extrémités des `Arrow`s doivent être contenues dans le `Rectangle` correspondant à la `Task` ciblée (lignes 18 à 21 et lignes 22 à 25)

7.4.4 Règles de transformation

Ce cas détude est séparé en deux problèmes distincts, le premier, décrit dans la sous-section 7.4.1, calcule un planning valide, tandis que le second, décrit dans la sous-section 7.4.2, génère un diagramme permettant de visualiser et d'interagir avec le planning calculé.

```
1  rule Period {
2    from
3      s : Scheduling!Period
4    to
5      constraints : Constraints!ConstraintGroup (
6        solver <- 'choco',
7        constraints <- Sequence {
8          (s.project.tasks.cost.toConstant() *
9            (s.project.tasks.period."="(s.toConstant())).reify()
10         ).sum() >= s.project.minLoad.toConstant(),
11         (s.project.tasks.cost.toConstant() *
12           (s.project.tasks.period."="(s.toConstant())).reify()
13         ).sum() <= s.project.maxLoad.toConstant()
14       }
15     )
16 }
```

Listing 17 – Règle de transformation permettant de calculer un planning valide

Contrairement aux cas d'étude précédents, nous avons utilisé ATL^C pour celui-ci. Les règles de transformations sont donc écrites en ATL augmenté de contraintes.

Le Listing 17 contient une règle ATL responsable de la génération d'un planning valide. À partir d'une `Period` (ligne 3) la règle génère un `ConstraintGroup` (lignes 5 à 15) contenant les deux contraintes relatives à la `load` de chaque `Period`. Cette règle de transformation est particulière, car elle tire parti d'un mode particulier d'ATL, le mode *refining* (ou raffinement). Dans ce mode, une règle peut modifier les éléments d'entrée. On remarque que le solveur cible est indiqué dans la règle au niveau de la définition du `ConstraintGroup` (ligne 6). Ces deux contraintes assurent que la somme des `cost`s des `Task`s affectées à une `Period` est comprise entre `minLoad` et `maxLoad`. La vérification de l'appartenance d'une `Task` à une `Period` se fait grâce à la réification de contraintes (lignes 9 et 12). Celles-ci correspondent aux contraintes lignes 7-12 du Listing 16. On notera que les contraintes lignes 9 et 12 correspondent à plusieurs contraintes présentes dans le Listing 16. C'est parce que `s.project.task` est une collection et qu'un `VariableVector` est donc utilisé pour la représenter. Durant les transformations adaptant le modèle de contraintes à Choco, les contraintes contenant des `VariableVector`s peuvent être étendues en plusieurs contraintes simples (voir sous-section 5.3.2).

Le Listing 18 contient une règle générant les figures géométriques représentant une `Task`. Une `Task` devient un `Rectangle` et un `Text` auxquels s'ajoute un `ConstraintGroup`. Contraire-

```
1  unique lazy rule Task {
2    from
3      c : Scheduling!Task
4    to
5      r  : JFX!Rectangle (
6        movable <- true
7      ),
8      t  : JFX!Text (
9        text <- c.code, textOrigin <- #TOP,
10       mouseTransparent <- true
11     ),
12     constraints : Constraints!ConstraintGroup (
13       solver <- 'cassowary',
14       constraints <- Sequence {
15         r.width = 1.2 * t.width.toConstant()
16         ,r.height = 1.2 * t.height.toConstant()
17         ,r.x.stay('MEDIUM'), r.y.stay('MEDIUM')
18         ,t.x = r.x + 5, t.y = r.y + 2
19         ,r.x >= 0, r.y >= c.period->collectTo('Period').l.startY
20         ,r.x >= c.period->collectTo('Period').r.x -- strong
21         ,r.x + r.width <= c.period->collectTo('Period').r.x
22          + c.period->collectTo('Period').r.width --strong
23         ,r.y+r.height <= c.period->collectTo('Period').r.y
24                 + c.period->collectTo('Period').r.height
25       }
26     )
27 }
```

Listing 18 – Règle de transformation simplifiée générant la vue et les contraintes liées à une `Period`

ment à la règle précédente, celle-ci possède des éléments de sortie autres que les contraintes (lignes 5-11). Ces éléments sont définis comme en ATL classiques. La propriété `movable` du `Rectangle` (ligne 6) n'est pas présente dans le métamodèle de figures (Figure 7.1) et est dynamiquement ajouté par ATL^C pour faciliter la mise en place des interacteurs. Contrairement à la règle précédente, les contraintes de celle-ci sont destinées à Cassowary (ligne 13). Les deux premières contraintes spécifient la dimention du `Rectangle` (lignes 15 et 16). On notera l'utilisation explicite de l'opération `toConstant()` permettant la création d'une constante liée plutôt qu'une variable. Les contraintes correspondantes se situent lignes 8 à 9 du Listing 15. D'autres contraintes spécifient la position du `Text` (ligne 18) et la position du `Rectangle` (lignes 19 à 24). Certaines contraintes ont un poids (lignes 20 et 22).

Les contraintes `stay` (ligne 17) indiquent que la position des `Rectangle`s ne doit pas changer d'une résolution à l'autre sauf si d'autres contraintes plus prioritaires nécessitent un changement. Sans ces contraintes, les positions des `Rectangle`s pourraient être modifiées par l'utilisateur, mais retourneraient à leur position par défaut à chaque nouvelle solution calculée par Cassowary.

Remarque : le code ATL des Listings 17 et 18 est légèrement simplifié par rapport aux règles initiales. La différence principale est dans la règle du Listing 18 où un élément de sortie, `Figure`, regroupant les contraintes et les éléments géométriques est ignoré. Cet élément sup-

plémentaire est utilisé pour regrouper les contraintes, les figures géométriques et les figures des objets dépendants dans un seul objet afin de faciliter la gestion des éléments de sortie dans le code liant le résultat de la transformation à JavaFX. Cet élément est purement technique et n'est pas pertinent pour la compréhension de la règle ni de la logique de la transformation.

7.5 Visualiseur et éditeur de diagrammes de classes et d'objets

Ce dernier cas d'étude, publié dans le *1st International Workshop on View-Oriented Software Engineering (VoSE)* [Le +19d], reprend l'idée du premier cas d'étude (visualiseur de diagrammes de classes) et l'étend en ajoutant :

1. Des interacteurs permettant l'édition du diagramme de classes,
2. Une seconde vue permettant la visualisation et l'édition de diagrammes d'objets.

La Figure 7.14 montre un exemple de diagrammes générés. En plus de ces ajouts, ce second cas d'étude sur les diagrammes de classes a été entièrement redéveloppé en ATL^C.

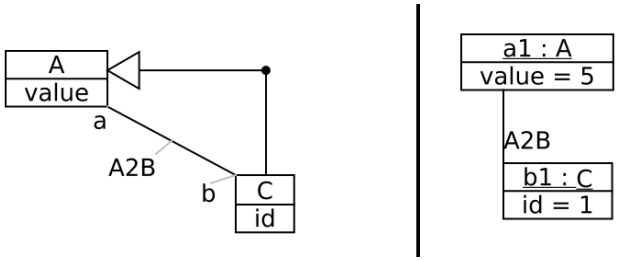


FIGURE 7.14 – Exemple de diagramme généré

7.5.1 Métamodèles source et cible

Comme les diagrammes précédents, celui-ci utilise une partie du métamodèle UML comme source. La Figure 7.15 détaille une version simplifiée du métamodèle UML considéré dans ce cas d'étude.

Celui-ci se compose d'une `Class`, qui est un `Classifier` et possède des `ownedAttribute`s, lesquels sont des `Property`s. Une `InstanceSpecification` (une instance d'une `Class`) a un `classifer` identifiant son type et des `slots`. Chaque `Slot` spécifie une `value` pour une `definingFeature` (`Property`). La `Class`, la `Property` et l'`InstanceSpecification` possèdent un `name` hérité de `NamedElement`.

Le métamodèle utilisé par les vues est très proche du métamodèle de diagrammes présenté dans la Figure 7.1. Comme celui-ci, il est basé sur le métamodèle de JavaFX. On notera l'ajout de classes (`Figure`, `Polygon`) et de propriétés (`movable` sur le `Node`, `width`, `height` et `editable` sur le `Text`).

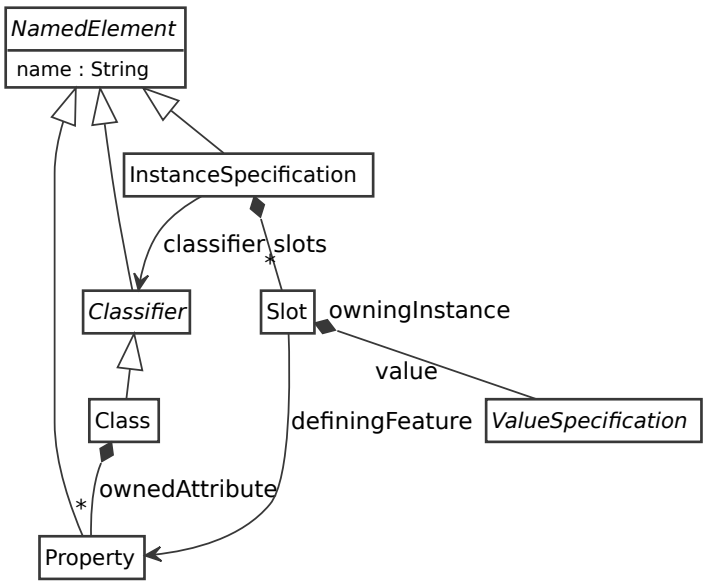


FIGURE 7.15 – Partie du métamodèle UML utilisé dans cette transformation

La classe `Node` sert de racine aux éléments graphiques de JavaFX. Elle possède un attribut (extension) `movable` nous indiquant s'il faut que les interacteurs liés à la gestion des mouvements s'appliquent au `Node`. Dans nos diagrammes nous ne considérons que les `Shape`s, c'est à dire des formes géométriques affichables. Ces formes sont définies par deux propriétés, `stroke` décrit le contour de la `Shape`, tandis que `fill` en décrit l'intérieur. Pour ce diagramme, nous ne considérons que quatre formes simples, le `Text`, le `Rectangle`, la `Line` et le `Polygon`. Le `Text` possède une propriété `text` contenant le texte à afficher ainsi que deux coordonnées `x` et `y`. Par souci de simplicité, nous avons ajouté deux propriétés dérivées `width` et `height`, ainsi qu'une propriété `editable` utilisée par les interacteurs. Le `Rectangle` possède, lui aussi, deux coordonnées `x` et `y`, ainsi qu'une `width` et une `height`. La `Line` comporte deux jeux de coordonnées `startX`, `startY` et `endX`, `endY` donnant les positions des deux extrémités de la `Line`. Enfin, le `Polygon` possède une liste `points` de valeurs flottantes doubles contenant les coordonnées des différents points composant le `Polygon`. Nous avons aussi ajouté une classe `Figure` nous permettant de regrouper les différents éléments graphiques sous la forme d'un arbre, facilitant ainsi la gestion des éléments graphiques.

7.5.2 Définitions des vues

Les Listings 19 et 20 montrent deux règles ATL générant les formes géométriques correspondant respectivement à une `Class` et une `Property`. Le Listing 19 transforme une `Class` et une `Figure` contenant un `Rectangle`, un `Text`, une `Line` et référençant des éléments fils construits par l'application de la règle `Property` sur les `ownedAttribute`s de la `Class` (lignes 7

```
unique lazy rule Class {
  from
    s : UML!Class
  to
    t : JFX!Figure (
      nodes <- Sequence {r, txt, l},
      children <-
        s.ownedAttribute->collect(e |
          thisModule.Property(e).t
        )
    ),
    r : JFX!Rectangle (
      movable <- true,
      fill <- thisModule.transparentWhite,
      stroke <- thisModule.opaqueBlack
    ),
    txt : JFX!Text (
      text <- s.name,
      fill <- thisModule.transparentWhite,
      stroke <- thisModule.opaqueBlack,
      editable <- true,
      movable <- true
    ),
    l : JFX!Line (
      stroke <- thisModule.opaqueBlack
    )
}
```

Listing 19 – Règle définissant les éléments graphiques d'une `Class` UML

```
unique lazy rule Property {
  from
    s : UML!Property
  to
    t : JFX!Figure (
      nodes <- Sequence {txt},
      children <- Sequence {}
    ),
    txt : JFX!Text (
      text <- s.name,
      fill <- thisModule.transparentWhite,
      stroke <- thisModule.opaqueBlack,
      editable <- true,
      movable <- true
    )
}
```

Listing 20 – Règle définissant les éléments graphiques d'une `Property` UML

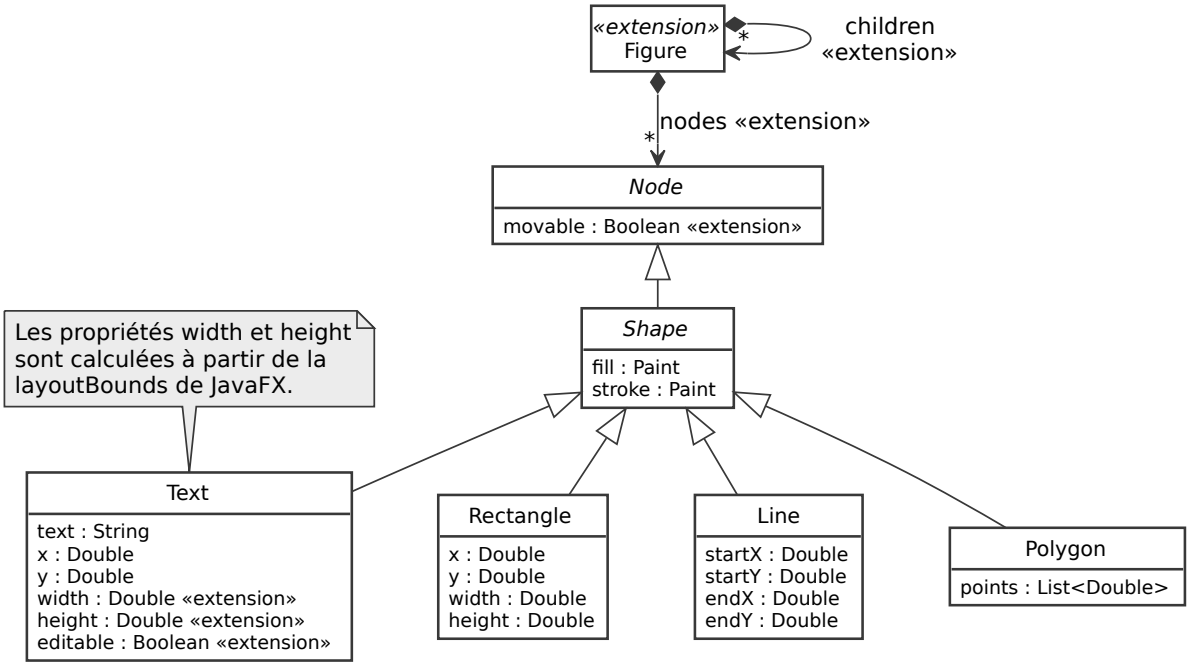


FIGURE 7.16 – Diagramme de classes des éléments utilisés par les vues

à 10). La seconde règle (Listing 20) transforme une `Property` en une `Figure` contenant un `Text`. Contrairement au cas d'étude précédent, la totalité de la partie ATL de la règle est montrée. On y retrouve donc les mêmes éléments que précédemment.

On notera l'utilisation de la `Figure` qui sert de conteneur regroupant tous les éléments graphiques correspondant à la règle ainsi que les éléments générés par les règles qui sont appelées explicitement. Grâce à cette `Figure`, il est possible de récupérer tous les éléments graphiques générés par l'application d'une règle. De plus, contrairement aux exemples précédents, les constantes sont nommées pour faciliter la lecture de la règle. Par exemple, `thisModule.transparentWhite` fait référence à un *helper* retournant une chaîne de caractères correspondant à un blanc transparent :
`helper def: transparentWhite : String = '0xffffff00';` Enfin, l'opération `collectTo()` (non standard en ATL) est remplacée par l'appel sous la forme `thisModule.Règle()` (exemple ligne 9), qui est standard en ATL pour les règles *lazy*.

7.5.3 Contraintes d'affichage

Comme nous avons pu le voir dans les exemples précédents, générer des figures géométriques n'est pas suffisant pour créer des diagrammes ayant du sens. Pour cela, il faut ajouter des contraintes spécifiant les positions des éléments les uns par rapport aux autres. En effet, sans ces contraintes, les figures géométriques ne portent plus aucun sens. Par exemple, la Fi-

gure 7.17 montre un diagramme dans lequel les contraintes de placement ont été désactivées.

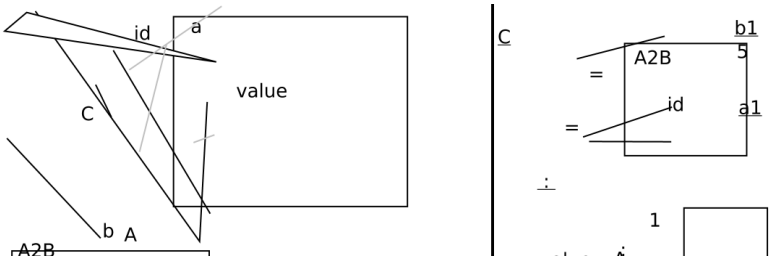


FIGURE 7.17 – Exemple de diagramme sans contraintes géométriques

Le Listing 21 montre comment la règle du Listing 19 est étendue pour gérer les contraintes graphiques. Celles-ci :

— Appliquent des contraintes `stay` aux coordonnées des `Rectangle` et `Text` pour que leurs positions conservent leurs valeurs autant que possible.

— Minimisent la taille du `Rectangle` (lignes 25 à 26).

— Indiquent que le `Rectangle` doit rester dans la portion visible de la fenêtre (ligne 27).

— Positionnent le `Text` correspondant au `name` de la `Class` en haut au centre de son `Rectangle` (lignes 29 à 30). On notera l'utilisation de variables ATL intermédiaires (lignes 15 à 20) pour simplifier la lecture de la contrainte.

— Dimensionnent le `Rectangle` de façon à ce qu'il soit assez large (ligne 32) et haut (ligne 34) pour contenir le `Text` représentant le `name` de la `Class`.

— Positionnent la `Line` de façon à ce qu'elle soit à l'horizontale au-dessous du `Text` (lignes 36 à 38).

La `Figure` créée par la règle est aussi modifiée afin qu'elle contienne la liste des contraintes associées aux éléments générés.

Similairement, la règle `Property` du Listing 20 est aussi étendue. Le Listing 22 montre ces modifications. Comme pour la règle précédente, ces modifications consistent en l'ajout d'un `ConstraintGroup` ainsi que l'enregistrement de celui-ci dans la `Figure`. Plus précisément, ces contraintes :

— Appliquent des contraintes `stay` aux coordonnées du `Text` (ligne 20).

— Placent le `Text` relativement à sa `Class` en `x` (ligne 21), ainsi qu'en `y` (lignes 23 à 25). Le positionnement en `y` se fait en deux temps, d'abord le `Text` est placé sous la `Line` séparant le `name` de la `Class` des `Properties` (ligne 23). Puis le `Text` est placé sous le `Text` correspondant à la `Property` précédente si elle existe (lignes 24 et 25).

— Dimensionnent le `Rectangle` correspondant à la `Class` à laquelle la `Property` appartient. D'abord en `width` (ligne 27), puis en `height` (ligne 30).

```
unique lazy rule Class {
  from
    s : UML!Class
  to
    t : JFX!Figure(
        ...,
        constraints <- Sequence {constraints}
    ),
    r : JFX!Rectangle (...),
    txt : JFX!Text (...),
    l : JFX!Line (...),
    constraints : Constraints!ConstraintGroup (
      solver <- 'cassowary',
      constraints <-
        let txtCenterX : Real =
            txt.x + txt.width.toConstant() / 2
        in
        let rCenterX : Real =
            r.x + r.width / 2
        in
        Sequence {
          r.x.stay('MEDIUM'), r.y.stay('MEDIUM'),
          txt.x.stay('MEDIUM'), txt.y.stay('MEDIUM'),

          r.width = 0 -- STRONG
          ,r.height = 0 -- STRONG
          ,r.x >= 0, r.y >= 0,

          rCenterX = txtCenterX,
          r.y = txt.y,

          r.width >= txt.width.toConstant() +
              thisModule.MARGIN * 2,
          r.height >= txt.height.toConstant(),

          l.startX = r.x, l.endX = r.x + r.width,
          l.startY = l.endY,
          l.startY = txt.y + txt.height.toConstant()
        }
    )
}
```

Listing 21 – Définition des contraintes liées à une `Class`

```
1  unique lazy rule Property {
2    from
3      s : UML!Property
4    to
5      t : JFX!Figure (
6          ...,
7        constraints <- Sequence {constraints},
8      ),
9      txt : JFX!Text (...),
10     constraints : Constraints!ConstraintGroup (
11       solver <- 'cassowary',
12       constraints <-
13         let class : OclAny =
14             thisModule.Class(s.class)
15         in
16         let prev : OclAny =
17             thisModule.Property(s.prev)
18         in
19         Sequence {
20           txt.x.stay('WEAK'), txt.y.stay('WEAK'),
21           txt.x = class.r.x + thisModule.MARGIN,
22
23           txt.y >= class.l.startY,
24           txt.y >= prev.txt.y +
25               prev.txt.height.toConstant(),
26
27           class.r.width >= txt.width.toConstant() +
28               thisModule.MARGIN * 2,
29           class.r.y + class.r.height >= txt.y +
30               txt.height.toConstant()
31         }
32     )
33 }
```

Listing 22 – Définition des contraintes liées à une `Property`

7.5.4 Application de la théorie de la gestion des cycles

Dans le chapitre 4 nous avons présenté une notation permettant de détecter les cycles dans les graphes de propagation créés par les opérations actives. L'approche n'a pas pu être ajoutée à AOF qui n'est pas conçu pour. Il n'est donc pas possible de détecter dynamiquement les graphes de propagation problématiques. Cependant, l'analyse de graphes de propagation a mis en évidence un ensemble de règles à respecter pour construire des graphes dans lesquels les notifications peuvent se propager sans erreurs. La règle principale est l'absence de cycles.

De plus, la notation et le travail de détections de synchronisation ne portaient que sur les opérations actives, mais ils peuvent facilement être adaptés aux graphes de propagation dans lesquels des solveurs de contraintes seraient utilisés. En effet, du point de vue de la propagation, un solveur de contraintes peut être assimilé à une opération active possédant de nombreuses entrées et sorties (une sortie par variable). La différence principale entre une opération active et un solveur est que lorsque le solveur reçoit une notification de changement sur une de ses variables, celle-ci est considérée comme une suggestion. Le solveur essaye de résoudre le problème en prenant en compte la suggestion, mais ne garantit pas que la nouvelle solution conservera la suggestion.

De ce point de vue, un solveur de contraintes peut être assimilé à une opération active et donc intégré au graphe de propagation. Il devient donc possible, en respectant les contraintes de cycles et de convergences du graphe de propagation, de concevoir des expressions calculées par une séquence d'opérations actives puis d'utiliser un solveur de contraintes et, à partir du résultat calculé par le solveur, de faire appel à d'autres opérations actives.

C'est notamment le cas des flèches utilisées dans ce cas d'étude. Le Listing 23 montre la règle générant les flèches représentant les `Generalization`s. Contrairement aux premiers cas d'étude, les flèches sont créées entièrement en ATL.

Les contraintes de cette règle sont similaires aux règles précédentes. On peut les regrouper en trois contraintes géométriques :

— Minimisation de la taille de la flèche (lignes 46 et 47). On notera que la minimisation de la taille de la flèche utilise la norme L1, car Cassowary est un solveur linéaire. Il n'est donc pas possible de calculer la distance en norme L2.

— L'extrémité `start` est contenue dans le `Rectangle` correspondant à la `Class` référencée par la propriété `specific` (lignes 49 à 52).

— L'extrémité `end` est contenue dans le `Rectangle` correspondant à la `Class` référencée par la propriété `general` (lignes 54 à 57).

La partie contenant les *bindings* ATL est elle aussi assez classique : la règle génère une `Figure` contenant le `ConstraintGroup`, le `Text`, la `Line` et le `Polygon`. On notera dans les *bindings* du `Polygon` l'utilisation de nombreux `let` permettant de calculer les différents points formant la tête de la flèche (lignes 16 à 33).

```
unique lazy rule GeneralizationSimpleLine {
  from
    s : UML!Generalization
  to
    t : JFX!Figure (
      nodes <- Sequence {l, p},
      constraints <- Sequence {constraints},
      children <- Sequence {}
    ),
    l : JFX!Line (
      stroke <- thisModule.opaqueBlack
    ),
    p : JFX!Polygon (
      stroke <- thisModule.opaqueBlack,
      fill <- thisModule.opaqueWhite,
      points <-
        let theta : Real = (l.endY - l.startY).atan2(l.endX - l.startX) in
        let barb : Real = 20 in
        let thirtyDeg : Real = 0.523599 in
        let phi : Real = thirtyDeg in
        (
          l.endX.oclAsType(Real)
        ).concat(
          l.endY.oclAsType(Real)
        ).concat(
          l.endX - barb * (theta + phi).cos()
        ).concat(
          l.endY - barb * (theta + phi).sin()
        ).concat(
          l.endX - barb * (theta - phi).cos()
        ).concat(
          l.endY - barb * (theta - phi).sin()
        ),
      movable <- true
    ),
    constraints : Constraints!ConstraintGroup (
      solver <- 'cassowary',
      constraints <-
        let start : OclAny = thisModule.Class(
          s.specific.oclAsType(UML!Class)
        ) in
        let end : OclAny = thisModule.Class(
          s.general.oclAsType(UML!Class)
        ) in
        Sequence {
          l.endX - l.startX = 0 -- weak
          ,l.endY - l.startY = 0 -- weak

          ,l.startX >= start.r.x,
          l.startX <= start.r.x + start.r.width,
          l.startY >= start.r.y,
          l.startY <= start.r.y + start.r.height,

          l.endX >= end.r.x,
          l.endX <= end.r.x + end.r.width,
          l.endY >= end.r.y,
          l.endY <= end.r.y + end.r.height
        }
    )
}
```

Listing 23 – Règle de transformation générant une `Generalization`

Plus précisément, ces points sont calculés à partir de différents paramètres constants (`barb` donnant la longueur de la tête de la flèche et `phi` donnant l'angle d'ouverture), mais aussi à partir de paramètres calculés (`theta` donnant l'orientation globale de la flèche, calculée à partir des coordonnées de la `Line` par exemple). Les coordonnées des extrémités de la `Line` sont utilisées dans le calcul des coordonnées des points du `Polygon`. Cependant, les coordonnées des extrémités de la flèche sont calculées par le solveur de contraintes.

Ainsi, lorsque l'utilisateur modifie le `name` d'une `Class`, l'interacteur détecte l'interaction de l'utilisateur et modifie la propriété correspondante. Ensuite, AOF propage le changement à JavaFX qui met à jour le `Text` correspondant, recalculant notamment les dimensions `width` et `height`. Ces nouvelles dimensions sont utilisées par le solveur pour calculer une nouvelle solution. Si cette nouvelle solution implique le changement de coordonnées des extrémités de la `Line` correspondant à une `Generalization`, alors les opérations actives calculant les positions des points du `Polygon` sont notifiées et les nouveaux points sont calculés.

Cette propagation est valide, car les points d'un `Polygon` ne sont pas utilisés par d'autres expressions pouvant modifier des propriétés du modèle UML source.

CONCLUSION

Rappel des problèmes considérés

Avec l'IDM, les modèles ont pris une place prépondérante dans le cycle de développement des systèmes complexes. Cependant, pour être facilement manipulés, ces modèles doivent être représentés de façon adaptée. Le plus souvent, cette représentation adaptée à l'être humain est un diagramme. Cependant, développer des outils de modélisation adaptés à chaque type de diagramme est une tâche complexe et coûteuse.

Certains outils facilitant la définition de syntaxes visuelles existent, mais ceux-ci sont limités. En effet, la majorité d'entre eux ne permettent que de générer des diagrammes similaires aux diagrammes de classes.

Solutions proposées

Notre approche, basée sur la transformation de modèles et la programmation par contraintes nous permet de spécifier déclarativement des transformations générant des visualiseurs ou éditeurs de diagrammes.

Plus généralement, nous avons formalisé ce problème sous la forme d'exploration d'ensembles de modèles. Un ensemble de modèles regroupe tous les modèles conformes à un métamodèle et vérifiant un ensemble de règles. L'exploration de ces ensembles est rendue possible par l'utilisation d'interacteurs, permettant à l'utilisateur de modifier le modèle qui est présenté, et de solveurs de contraintes restaurant la validité du modèle après chaque modification si nécessaire.

Ce système est générique et peut être adapté à de nombreux problèmes autres que la visualisation et l'édition de diagrammes. Par exemple, dans le cas d'étude de visualiseur de planning, nous vérifions des contraintes métiers d'un modèle pour générer un planning valide.

De plus, nous avons présenté deux implémentations de cette approche. La première est un DSL interne à Xtend, un langage de programmation généraliste. La seconde est une extension d'ATL, un langage de transformation dédié.

Enfin, nous avons proposé une formalisation de la notion de graphe de propagation utilisé par les opérations actives. Ce graphe de propagation sert de canevas sur lequel les opérations communiquent les changements à appliquer. Avec notre approche basée sur l'algèbre des processus, il est possible d'analyser ces graphes de propagation. Leur analyse permet la création de plans d'exécution ne comportant pas de problèmes de synchronisation.

Cette formalisation en algèbre des processus permet de nouvelles analyses telles que la détection des opérations pouvant être exécutées en parallèle.

Perspectives

En l'état, aucune implémentation du formalisme des graphes de propagation n'a été réalisée. En effet, la version actuelle d'AOF ne possède pas de graphe de propagation explicite, la propagation se fait d'opération en opération. Par conséquent, il est compliqué d'intégrer un outil de supervision de la propagation. C'est pourquoi ces travaux n'ont pas pu être implémentés dans la version actuelle d'AOF et sont restés à l'état de projet pour une version future d'AOF.

Le compilateur ATOL et le plug-in ATL^C sont fonctionnels, mais encore incomplets. En effet, ATOL ne supporte qu'un petit sous-ensemble d'ATL classique. Par exemple, celui-ci ne supporte pour l'instant que les règles *unique lazy*.

De même, ATL^C est encore limité par le manque de solveurs et de couches d'abstraction. L'ajout de nouveaux solveurs pourrait permettre une meilleure gestion des relations par exemple. Bien que performant et flexible, le solveur Choco n'est pas le plus adapté pour représenter des problèmes portant sur les relations d'un modèle. C'est pourquoi nous avons pour projet d'ajouter de nouveaux solveurs plus adaptés, ainsi que de nouveaux prédicats permettant de manipuler ces relations plus simplement.

Nous avons montré que les *wrappers* autour des solveurs doivent réécrire les contraintes du modèle de contraintes en contraintes spécifiques au solveur. Cependant, la version actuelle d'ATL^C utilise un ensemble de transformations écrites manuellement avec AOF. Ces transformations sont complexes à développer et à maintenir, c'est pourquoi nous avons lancé une réécriture de ces transformations en ATL incrémental avec ATOL.

Nous avons présenté des couches d'abstraction avec l'implémentation basée sur Xtend. Cependant nous ne les avons pas encore intégrées dans ATL^C. Nous avons pour projet d'ajouter ces couches d'abstraction, car celles-ci simplifient grandement l'utilisation du langage pour des utilisateurs non experts en programmation par contraintes.

Enfin, le langage de contraintes utilisé est basé sur OCL, mais il en diffère sur certains points. Cependant, pour des utilisateurs experts d'ATL, il serait préférable d'autoriser l'écriture de contraintes en suivant les structures classiques d'OCL plutôt que de forcer l'utilisation d'un langage de contraintes.

Listes des publications liées à la thèse

Conférences internationales à comité de lecture

— Théo LE CALVAR, Fabien CHHEL, Frédéric JOUAULT et Frédéric SAUBION, « Toward a Declarative Language to Generate Explorable Sets of Models », in : *34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*, Limassol, Cyprus, avr. 2019, p. 1837-1844, DOI : 10.1145/3297280.3297461

— Théo LE CALVAR, Frédéric JOUAULT, Fabien CHHEL et Mickael CLAVREUL, « Efficient ATL Incremental Transformations. », in : *The Journal of Object Technology* 18.*3* (2019), 2 :1, DOI : 10.5381/jot.2019.18.3.a2

Conférences nationales à comité de lecture

— Théo LE CALVAR, Fabien CHHEL, Frédéric JOUAULT et Frédéric SAUBION, « Transformation de Modèles et Contraintes pour l'Ingénierie Dirigée par les Modèles », in : *JFPC 2018 - Actes des 14es Journées Francophones de Programmation par Contraintes*, Amiens, France, juin 2018, p. 93-102, URL : https://home.mis.u-picardie.fr/~evenement/JFPC2018/actesJFPC2018.pdf#section.11

— Théo LE CALVAR, Fabien CHHEL, Frédéric JOUAULT et Frédéric SAUBION, « Transformation de modèles et programmation par contraintes avec ATLC », in : *JFPC 2019 - Actes des 15es Journées Francophones de Programmation par Contraintes*, Albi, France, juin 2019, p. 117-120, URL : https://hal-mines-albi.archives-ouvertes.fr/hal-02159866

Atelier de conférences internationales à comité de lecture

— Valentin BESNARD, Frédéric JOUAULT, Théo LE CALVAR et Massimo TISI, « The TTC 2018 Social Media Case, by ATL and AOF », in : *11th Transformation Tool Contest, co-located with the 2018 Software Technologies : Applications and Foundations (STAF 2018)*, Toulouse, France, juin 2018, URL : http://ceur-ws.org/Vol-2310/

— Théo LE CALVAR, Fabien CHHEL, Frédéric JOUAULT et Frédéric SAUBION, « Using process algebra to statically analyze incremental propagation graphs », in : *18th International Workshop in OCL and Textual Modeling*, Copenhague, Denmark, oct. 2018, p. 160-173, URL : http://ceur-ws.org/Vol-2245/

— Dennis WAGELAAR, Théo LE CALVAR et Frédéric JOUAULT, « Truth Tables to Binary Decision Diagrams in Modern ATL », in : *Proceedings of the 12th Transformation Tool Contest*, à paraître, Eindhoven, The Netherlands, juil. 2019

— Théo LE CALVAR, Frédéric JOUAULT, Fabien CHHEL, Frédéric SAUBION et Mickael CLAVREUL, « Intensional View Definition with Constrained Incremental Transformation Rules », in : *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, Munich, Germany : IEEE, sept. 2019, DOI : 10.1109/models-c.2019.00061

Poster de conférence internationale à comité de lecture

— Théo LE CALVAR, Fabien CHHEL, Frédéric JOUAULT et Frédéric SAUBION, *Static analysis of incremental propagation graphs with process algebra*, 18, Poster presentation related to OCL workshop 2018, Copenhague, Danemark, oct. 2018

BIBLIOGRAPHIE

[All90] James F. ALLEN, « Maintaining Knowledge about Temporal Intervals », in : *Readings in Qualitative Reasoning About Physical Systems*, Elsevier, 1990, p. 361-372, DOI : 10.1016/b978-1-4832-1447-4.50033-x.

[BBS01] Greg J BADROS, Alan BORNING et Peter J STUCKEY, « The Cassowary linear arithmetic constraint solving algorithm », in : *ACM Transactions on Computer-Human Interaction (TOCHI)* 8.*4* (2001), p. 267-306.

[Bea+10] Olivier BEAUDOUX, Arnaud BLOUIN, Olivier BARAIS et Jean-Marc JÉZÉQUEL, « Active Operations on Collections », in : *Model Driven Engineering Languages and Systems*, Springer Berlin Heidelberg, 2010, p. 91-105, DOI : 10.1007/978-3-642-16145-2_7.

[Ben96] Frédéric BENHAMOU, « Heterogeneous constraint solving », in : *Algebraic and Logic Programming*, sous la dir. de Michael HANUS et Mario RODRÍGUEZ-ARTALEJO, Springer Berlin Heidelberg, 1996, p. 62-76, ISBN : 978-3-540-70672-4.

[BK84] J.A. BERGSTRA et J.W. KLOP, « Process algebra for synchronous communication », in : *Information and Control* 60.*1-3* (jan. 1984), p. 109-137, DOI : 10.1016/s0019-9958(84)80025-x.

[BPS01] Jan A BERGSTRA, Alban PONSE et Scott A SMOLKA, *Handbook of Process Algebra*, Elsevier, 2001, DOI : 10.1016/b978-0-444-82830-9.x5017-6.

[Bes+18] Valentin BESNARD, Frédéric JOUAULT, Théo LE CALVAR et Massimo TISI, « The TTC 2018 Social Media Case, by ATL and AOF », in : *11th Transformation Tool Contest, co-located with the 2018 Software Technologies : Applications and Foundations (STAF 2018)*, Toulouse, France, juin 2018, URL : http://ceur-ws.org/Vol-2310/.

[BH11] Bernhard BETTIG et Christoph M. HOFFMANN, « Geometric Constraint Solving in Parametric Computer-Aided Design », in : *Journal of Computing and Information Science in Engineering* 11.*2* (juin 2011), DOI : 10.1115/1.3593408.

[Béz+05] Jean BÉZIVIN, Frédéric JOUAULT, Peter ROSENTHAL et Patrick VALDURIEZ, « Modeling in the Large and Modeling in the Small », in : *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2005, p. 33-46, DOI : 10.1007/11538097_3.

[Bis04] Stefano BISTARELLI, *Semirings for Soft Constraint Solving and Programming*, Springer Berlin Heidelberg, 2004, DOI : 10.1007/b95712.

[Boh+08] Aaron BOHANNON, J. Nathan FOSTER, Benjamin C. PIERCE, Alexandre PILKIEWICZ et Alan SCHMITT, « Boomerang », in : *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '08*, ACM Press, 2008, DOI : 10.1145/1328438.1328487.

[Bor18] Artur BORONAT, « Expressive and Efficient Model Transformation with an Internal DSL of Xtend », in : *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems - MODELS '18*, ACM Press, 2018, DOI : 10.1145/3239372.3239386.

[BSA17] Ilhem BOUSSAÏD, Patrick SIARRY et Mohamed AHMED-NACER, « A survey on search-based model-driven engineering », in : *Automated Software Engineering* 24.*2* (juin 2017), p. 233-294, ISSN : 1573-7535, DOI : 10.1007/s10515-017-0215-4.

[BLP16] Frédéric BOUSSEMART, Christophe LECOUTRE et Cédric PIETTE, « XCSP3 : An Integrated Format for Benchmarking Combinatorial Constrained Problems », in : *CoRR* abs/1611.03398 (2016), arXiv : 1611.03398.

[CCR07] Jordi CABOT, Robert CLARISÓ et Daniel RIERA, « UMLtoCSP : a tool for the formal verification of UML/OCL models using constraint programming », in : *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ACM, 2007, p. 547-548.

[Cic+11] Antonio CICCHETTI, Davide DI RUSCIO, Romina ERAMO et Alfonso PIERANTONIO, « JTL : A Bidirectional and Change Propagating Transformation Language », in : *Software Language Engineering*, sous la dir. de Brian MALLOY, Steffen STAAB et Mark van den BRAND, 2011, p. 183-202, ISBN : 978-3-642-19440-5.

[CRE06] Antonio CICCHETTI, Davide Di RUSCIO et Romina ERAMO, « Towards Propagation of Changes by Model Approximations », in : *2006 10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW'06)*, IEEE, oct. 2006, DOI : 10.1109/edocw.2006.68.

[Con+06] Jean-Francois CONDOTTA, Dominique DALMEIDA, Christophe LECOUTRE et Lahkdar SAIS, « From Qualitative to Discrete Constraint Networks », in : *Workshop on Qualitative Constraint Calculi held with KI 2006*, KI'2006, juin 2006, p. 54-64.

[DMP91] Rina DECHTER, Itay MEIRI et Judea PEARL, « Temporal constraint networks », in : *Artificial Intelligence* 49.*1-3* (mai 1991), p. 61-95, DOI : 10.1016/0004-3702(91)90006-6.

[Dis+14] Zinovy DISKIN, Arif WIDER, Hamid GHOLIZADEH et Krzysztof CZARNECKI, « Towards a rational taxonomy for increasingly symmetric model synchronization », in : *ICMT 2014*, 2014, p. 57-73.

[DXC10] Zinovy DISKIN, Yingfei XIONG et Krzysztof CZARNECKI, « From State- to Delta-Based Bidirectional Model Transformations », in : *Theory and Practice of Model Transformations*, Springer Berlin Heidelberg, 2010, p. 61-76, DOI : 10.1007/978-3-642-13688-7_5.

[DMC96] M. DORIGO, V. MANIEZZO et A. COLORNI, « Ant system : optimization by a colony of cooperating agents », in : *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)* 26.*1* (1996), p. 29-41, DOI : 10.1109/3477.484436.

[DR94] Chun DU et Manfred ROSENDAHL, « Constructive Geometric Modelling with Object-Oriented Methodology », in : *4th Eurographics Workshop on Object Oriented Graphics, Sintra, Portugal*, t. 9, 11.5, 1994, p. 94.

[FL93] Hélène FARGIER et Jérôme LANG, « Uncertainty in constraint satisfaction problems : A probabilistic approach », in : *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, Springer-Verlag, 1993, p. 97-104, DOI : 10.1007/bfb0028188.

[For89] Charles L. FORGY, « Rete : A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem », in : *Readings in Artificial Intelligence and Databases*, Elsevier, 1989, p. 547-559, DOI : 10.1016/b978-0-934613-53-8.50041-8.

[FGK90] Robert FOURER, David M. GAY et Brian W. KERNIGHAN, « A Modeling Language for Mathematical Programming », in : *Management Science* 36.*5* (mai 1990), p. 519-554, DOI : 10.1287/mnsc.36.5.519.

[Frü95] Thom FRÜHWIRTH, « Constraint handling rules », in : *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 1995, p. 90-107, DOI : 10.1007/3-540-59155-9_6.

[Frü98] Thom FRÜHWIRTH, « Theory and practice of constraint handling rules », in : *The Journal of Logic Programming* 37.*1-3* (oct. 1998), p. 95-138, DOI : 10.1016/s0743-1066(98)10005-5.

[GL88] Michael GELFOND et Vladimir LIFSCHITZ, « The Stable Model Semantics for Logic Programming », in : *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, 1988, p. 1070-1080.

[GW06] Holger GIESE et Robert WAGNER, « Incremental Model Synchronization with Triple Graph Grammars », in : *Model Driven Engineering Languages and Systems*, Springer Berlin Heidelberg, 2006, p. 543-557, DOI : 10.1007/11880240_38.

[Glo86] Fred GLOVER, « Future paths for integer programming and links to artificial intelligence », in : *Computers & Operations Research* 13.*5* (jan. 1986), p. 533-549, DOI : 10.1016/0305-0548(86)90048-1.

[GBR07] Martin GOGOLLA, Fabian BÜTTNER et Mark RICHTERS, « USE : A UML-based specification environment for validating UML and OCL », in : *Science of Computer Programming* 69.*1-3* (déc. 2007), p. 27-34, DOI : 10.1016/j.scico.2007.01.013.

[Gut+03] Carsten GUTWENGER, Michael JÜNGER, Karsten KLEIN, Joachim KUPKE, Sebastian LEIPERT et Petra MUTZEL, « A new approach for visualizing UML class diagrams », in : *Proceedings of the 2003 ACM symposium on Software visualization - SoftVis '03*, ACM Press, 2003, DOI : 10.1145/774833.774859.

[Hal47] Paul R. HALMOS, *Finite Dimensional Vector Spaces. (AM-7), Volume 7*, Princeton University Press, 21 jan. 1947, 208 p., ISBN : 0691090955, URL : https://www.ebook.de/de/product/3678850/paul_r_halmos_finite_dimensional_vector_spaces_am_7_volume_7.html.

[HJ01] Mark HARMAN et Bryan F JONES, « Search-based software engineering », in : *Information and Software Technology* 43.*14* (déc. 2001), p. 833-839, DOI : 10.1016/s0950-5849(01)00189-6.

[HMZ12] Mark HARMAN, S. Afshin MANSOURI et Yuanyuan ZHANG, « Search-based Software Engineering : Trends, Techniques and Applications », in : *ACM Comput. Surv.* 45.*1* (déc. 2012), 11 :1-11 :61, ISSN : 0360-0300, DOI : 10.1145/2379776.2379787.

[Hid+15] Soichiro HIDAKA, Massimo TISI, Jordi CABOT et Zhenjiang HU, « Feature-based classification of bidirectional transformation approaches », in : *Software & Systems Modeling* 15.*3* (jan. 2015), p. 907-928, DOI : 10.1007/s10270-014-0450-0.

[Hil+13] Stephan HILDEBRANDT, Leen LAMBERS, Holger GIESE, Jan RIEKE, Joel GREENYER, Wilhelm SCHÄFER, Marius LAUDER, Anthony ANJORIN et Andy SCHÜRR, « A Survey of Triple Graph Grammar Tools », en, in : *Electronic Communications of the EASST* 57 : Bidirectional Transformations 2013 (2013), DOI : 10.14279/tuj.eceasst.57.865.

[Hoa78] C. A. R. HOARE, « Communicating sequential processes », in : *Communications of the ACM* 21.*8* (août 1978), p. 666-677, DOI : 10.1145/359576.359585.

[HPW11] Martin HOFMANN, Benjamin PIERCE et Daniel WAGNER, « Symmetric lenses », in : *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '11*, ACM Press, 2011, DOI : 10.1145/1926385.1926428.

[Hol92] John H. HOLLAND, *Adaptation in Natural and Artificial Systems*, The MIT Press, 1992, DOI : 10.7551/mitpress/1090.001.0001.

[HV09] Ákos HORVÁTH et Dániel VARRÓ, « CSP(M) : Constraint Satisfaction Problem over Models », in : *Model Driven Engineering Languages and Systems*, sous la dir. d'Andy SCHÜRR et Bran SELIC, Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, p. 107-121, ISBN : 978-3-642-04425-0.

[HV12] Ákos HORVÁTH et Dániel VARRÓ, « Dynamic constraint satisfaction problems over models », in : *Software & Systems Modeling* 11.*3* (2012), p. 385-408.

[ISO96] ISO/IEC 14977 : 1996, *Information technology Syntactic metalanguage Extended BNF*, Available at https ://www.cl.cam.ac.uk/ mgk25/iso-14977.pdf, déc. 1996.

[Jac02] Daniel JACKSON, « Alloy : a lightweight object modelling notation », in : *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11.*2* (2002), p. 256-290.

[Jou+08] Frédéric JOUAULT, Freddy ALLILAIRE, Jean BÉZIVIN et Ivan KURTEV, « ATL : A model transformation tool », in : *Science of Computer Programming* 72.*1-2* (juin 2008), p. 31-39, DOI : 10.1016/j.scico.2007.08.002.

[JB16] Frédéric JOUAULT et Olivier BEAUDOUX, « Efficient OCL-based Incremental Transformations », in : *16th International Workshop in OCL and Textual Modeling*, 2016, p. 121-136.

[Jou+18] Frédéric JOUAULT, Olivier BEAUDOUX, Matthias BRUN, Fabien CHHEL et Mickaël CLAVREUL, « Improving Incremental and Bidirectional Evaluation with an Explicit Propagation Graph », in : *Software Technologies : Applications and Foundations*, Springer International Publishing, 2018, p. 302-316, DOI : 10.1007/978-3-319-74730-9_27.

[JK06] Frédéric JOUAULT et Ivan KURTEV, « Transforming Models with ATL », in : *Satellite Events at the MoDELS 2005 Conference*, sous la dir. de Jean-Michel BRUEL, Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, p. 128-138, ISBN : 978-3-540-31781-4.

[JT10] Frédéric JOUAULT et Massimo TISI, « Towards Incremental Execution of ATL Transformations », in : *Theory and Practice of Model Transformations*, sous la dir. de Laurence TRATT et Martin GOGOLLA, Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, p. 123-137, ISBN : 978-3-642-13688-7.

[JM04] Michael JÜNGER et Petra MUTZEL, éd., *Graph Drawing Software*, Springer Berlin Heidelberg, 2004, DOI : 10.1007/978-3-642-18638-7.

[Kah+18] Nafiseh KAHANI, Mojtaba BAGHERZADEH, James R. CORDY, Juergen DINGEL et Daniel VARRÓ, « Survey and classification of model transformation tools », in : *Software & Systems Modeling* (mar. 2018), ISSN : 1619-1374, DOI : 10.1007/s10270-018-0665-6, URL : https://doi.org/10.1007/s10270-018-0665-6.

[KJS11] Eunsuk KANG, Ethan JACKSON et Wolfram SCHULTE, « An Approach for Effective Design Space Exploration », in : *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, Springer Berlin Heidelberg, 2011, p. 33-54, DOI : 10.1007/978-3-642-21292-5_3.

[KE75] James KENNEDY et Russell C. EBERHART, « Particle swarm optimization », in : *Proceedings of ICNN'95 - International Conference on Neural Networks*, IEEE, 1975, DOI : 10.1109/icnn.1995.488968.

[KGV83] S. KIRKPATRICK, C. D. GELATT et M. P. VECCHI, « Optimization by Simulated Annealing », in : *Science* 220.*4598* (mai 1983), p. 671-680, DOI : 10.1126/science.220.4598.671.

[Kön05] Alexander KÖNIGS, « Model Transformation with Triple Graph Grammars », in : *IN MODEL TRANSFORMATIONS IN PRACTICE SATELLITE WORKSHOP OF MODELS 2005, MONTEGO*, 2005.

[KBA02] Ivan KURTEV, Jean BÉZIVIN et Mehmet AKSIT, « Technological spaces : An initial appraisal », in : *CoopIS, DOA2002 Federated Conferences, Industrial track*, 2002.

[Kus+13] Angelika KUSEL, Juergen ETZLSTORFER, Elisabeth KAPSAMMER, Philip LANGER, Werner RETSCHITZEGGER, Johannes SCHOENBOECK, Wieland SCHWINGER et Manuel WIMMER, « A survey on incremental model transformation approaches », in : *ME 2013–Models and Evolution Workshop Proceedings*, 2013, p. 4.

[Lau18] LAURENT MICHEL, PIERRE SCHAUS, PASCAL VAN HENTENRYCK, *MiniCP : A Lightweight Solver for Constraint Programming*, 2018, URL : https://minicp.bitbucket.io.

[Le +18a] Théo LE CALVAR, Fabien CHHEL, Frédéric JOUAULT et Frédéric SAUBION, *Static analysis of incremental propagation graphs with process algebra*, 18, Poster presentation related to OCL workshop 2018, Copenhague, Danemark, oct. 2018.

[Le +18b] Théo LE CALVAR, Fabien CHHEL, Frédéric JOUAULT et Frédéric SAUBION, « Transformation de Modèles et Contraintes pour l'Ingénierie Dirigée par les Modèles », in : *JFPC 2018 - Actes des 14es Journées Francophones de Programmation par Contraintes*, Amiens, France, juin 2018, p. 93-102, URL : https://home.mis.u-picardie.fr/~evenement/JFPC2018/actesJFPC2018.pdf#section.11.

[Le +18c] Théo LE CALVAR, Fabien CHHEL, Frédéric JOUAULT et Frédéric SAUBION, « Using process algebra to statically analyze incremental propagation graphs », in : *18th International Workshop in OCL and Textual Modeling*, Copenhague, Denmark, oct. 2018, p. 160-173, URL : http://ceur-ws.org/Vol-2245/.

[Le +19a] Théo LE CALVAR, Fabien CHHEL, Frédéric JOUAULT et Frédéric SAUBION, « Toward a Declarative Language to Generate Explorable Sets of Models », in : *34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*, Limassol, Cyprus, avr. 2019, p. 1837-1844, DOI : 10.1145/3297280.3297461.

[Le +19b] Théo LE CALVAR, Fabien CHHEL, Frédéric JOUAULT et Frédéric SAUBION, « Transformation de modèles et programmation par contraintes avec ATLC », in : *JFPC 2019 - Actes des 15es Journées Francophones de Programmation par Contraintes*, Albi, France, juin 2019, p. 117-120, URL : https://hal-mines-albi.archives-ouvertes.fr/hal-02159866.

[Le +19c] Théo LE CALVAR, Frédéric JOUAULT, Fabien CHHEL et Mickael CLAVREUL, « Efficient ATL Incremental Transformations. », in : *The Journal of Object Technology* 18.*3* (2019), 2 :1, DOI : 10.5381/jot.2019.18.3.a2.

[Le +19d] Théo LE CALVAR, Frédéric JOUAULT, Fabien CHHEL, Frédéric SAUBION et Mickael CLAVREUL, « Intensional View Definition with Constrained Incremental Transformation Rules », in : *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, Munich, Germany : IEEE, sept. 2019, DOI : 10.1109/models-c.2019.00061.

[Leb+14] Erhan LEBLEBICI, Anthony ANJORIN, Andy SCHÜRR, Stephan HILDEBRANDT, Jan RIEKE et Joel GREENYER, « A Comparison of Incremental Triple Graph Grammar Tools », en, in : *Electronic Communications of the EASST* Volume 67 : Proc. of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2014) (2014), DOI : 10.14279/tuj.eceasst.67.939.

[Lec09] Christophe LECOUTRE, *Constraint Networks : Techniques and Algorithms*, ISTE Ltd., 10 juil. 2009, 586 p., ISBN : 1848211066, URL : https://www.ebook.de/de/product/9344896/christophe_lecoutre_constraint_networks.html.

[MC16] Nuno MACEDO et Alcino CUNHA, « Least-change bidirectional model transformation with QVT-R and ATL », in : *Software & Systems Modeling* 15.*3* (juil. 2016), p. 783-810, ISSN : 1619-1374, DOI : 10.1007/s10270-014-0437-x.

[MGC13] Nuno MACEDO, Tiago GUIMARAES et Alcino CUNHA, « Model repair and transformation with Echo », in : *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, nov. 2013, DOI : 10.1109/ase.2013.6693135.

[MTD17] Salvador MARTINEZ, Massimo TISI et Rémi DOUENCE, « Reactive Model Transformation with ATL », in : *Science of Computer Programming* 136 (mar. 2017), p. 1-16, DOI : 10.1016/j.scico.2016.08.006, URL : https://hal.inria.fr/hal-01627991.

[MBC93] Francisco MENEZES, Pedro BARAHONA et Philippe CODOGNET, « An Incremental Hierarchical Constraint Solver », in : *PPCP*, t. 93, 1993, p. 190-199.

[MHS05] Marjan MERNIK, Jan HEERING et Anthony M. SLOANE, « When and how to develop domain-specific languages », in : *ACM Computing Surveys* 37.*4* (déc. 2005), p. 316-344, DOI : 10.1145/1118890.1118892.

[Mil80] Robin MILNER, éd., *A Calculus of Communicating Systems*, Springer Berlin Heidelberg, 1980, DOI : 10.1007/3-540-10235-3.

[NO79] Greg NELSON et Derek C. OPPEN, « Simplification by Cooperating Decision Procedures », in : *ACM Transactions on Programming Languages and Systems* 1.*2* (oct. 1979), p. 245-257, DOI : 10.1145/357073.357079.

[Net+07] Nicholas NETHERCOTE, Peter J STUCKEY, Ralph BECKET, Sebastian BRAND, Gregory J DUCK et Guido TACK, « MiniZinc : Towards a standard CP modelling language », in : *CP 2007*, 2007, p. 529-543.

[PTS06] Duc Nghia PHAM, John THORNTON et Abdul SATTAR, « Towards an Efficient SAT Encoding for Temporal Reasoning », in : *Principles and Practice of Constraint Programming - CP 2006*, Springer Berlin Heidelberg, 2006, p. 421-436, DOI : 10.1007/11889205_31.

[Pru+17] Charles PRUD'HOMME et al., *Choco Documentation*, TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017, URL : http://www.choco-solver.org.

[RCC92] David A. RANDELL, Zhan CUI et Anthony G. COHN, « A Spatial Logic Based on Regions and Connection », in : *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, KR'92, Cambridge, MA : Morgan Kaufmann Publishers Inc., 1992, p. 165-176, ISBN : 1-55860-262-3, URL : http://dl.acm.org/citation.cfm?id=3087223.3087240.

[Rut94] Zsófia RUTTKAY, « Fuzzy constraint satisfaction », in : *Proceedings of 1994 IEEE 3rd International Fuzzy Systems Conference*, IEEE, juin 1994, DOI : 10.1109/fuzzy.1994.343640.

[Sch95] Andy SCHÜRR, « Specification of graph translators with triple graph grammars », in : *Graph-Theoretic Concepts in Computer Science*, sous la dir. d'Ernst W. MAYR, Gunther SCHMIDT et Gottfried TINHOFER, Berlin, Heidelberg : Springer Berlin Heidelberg, 1995, p. 151-163, ISBN : 978-3-540-49183-5.

[SNV18] Oszkár SEMERÁTH, András Szabolcs NAGY et Dániel VARRÓ, « A Graph Solver for the Automated Generation of Consistent Domain-specific Models », in : *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, Gothenburg, Sweden : ACM, 2018, p. 969-980, ISBN : 978-1-4503-5638-1, DOI : 10.1145/3180155.3180186.

[SV17] Oszkár SEMERÁTH et Dániel VARRÓ, « Graph Constraint Evaluation over Partial Models by Constraint Rewriting », in : *Theory and Practice of Model Transformation*, sous la dir. d'Esther GUERRA et Mark van den BRAND, Cham : Springer International Publishing, 2017, p. 138-154, ISBN : 978-3-319-61473-1.

[SBM09] Sagar SEN, Benoit BAUDRY et Jean-Marie MOTTU, « Automatic Model Generation Strategies for Model Transformation Testing », in : *ICMT 2009*, sous la dir. de Richard F. PAIGE, 2009, p. 148-164, ISBN : 978-3-642-02408-5.

[SBP07] Sagar SEN, Benoit BAUDRY et Doina PRECUP, « Partial model completion in model driven engineering using constraint logic programming », in : *17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and 21st Workshop on (Constraint)*, 2007, p. 59.

[SIN91] S. SHIMIZU, K. INOUE et M. NUMAO, « An ATMS-based geometric constraint solver for 3D CAD », in : *Third International Conference on Tools for Artificial Intelligence - TAI 91*, IEEE Comput. Soc. Press, nov. 1991, DOI : 10.1109/tai.1991.167106.

[Soe+10] Mathias SOEKEN, Robert WILLE, Mirco KUHLMANN, Martin GOGOLLA et Rolf DRECHSLER, « Verifying UML/OCL Models Using Boolean Satisfiability », in : *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, Dresden, Germany : European Design et Automation Association, 2010, p. 1341-1344, ISBN : 978-3-9810801-6-2, URL : http://dl.acm.org/citation.cfm?id=1870926.1871248.

[ST09] Friedrich STEIMANN et Andreas THIES, « From Public to Private to Absent : Refactoring Java Programs under Constrained Accessibility », in : *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2009, p. 419-443, DOI : 10.1007/978-3-642-03013-0_19.

[Ste08] Perdita STEVENS, « A Landscape of Bidirectional Model Transformations », in : *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2008, p. 408-424, DOI : 10.1007/978-3-540-88643-3_10.

[Sty73] George P.H. STYAN, « Hadamard products and multivariate statistical analysis », in : *Linear Algebra and its Applications* 6 (1973), p. 217-240, DOI : 10.1016/0024-3795(73)90023-2.

[Tea05] Gecode TEAM, *Gecode : Generic constraint development environment*, 2005, URL : https://www.gecode.org/.

[Tea10] OR-Tools TEAM, *OR-Tools*, 2010, URL : https://github.com/google/or-tools.

[Var+16] Dániel VARRÓ et al., « Road to a reactive and incremental model transformation platform : three generations of the VIATRA framework », in : *Software & Systems Modeling* 15.*3* (juil. 2016), p. 609-629, ISSN : 1619-1374, DOI : 10.1007/s10270-016-0530-4.

[VB07] Dániel VARRÓ et András BALOGH, « The model transformation language of the VIATRA2 framework », in : *Science of Computer Programming* 68.*3* (2007), Special Issue on Model Transformation, p. 214-234, ISSN : 0167-6423, DOI : https://doi.org/10.1016/j.scico.2007.05.004, URL : http://www.sciencedirect.com/science/article/pii/S016764230700127X.

[von+13] Jens VON PILGRIM, Bastian ULKE, Andreas THIES et Friedrich STEIMANN, « Model/code co-refactoring : An MDE approach », in : *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, nov. 2013, p. 682-687, DOI : 10.1109/ASE.2013.6693133.

[WLJ19] Dennis WAGELAAR, Théo LE CALVAR et Frédéric JOUAULT, « Truth Tables to Binary Decision Diagrams in Modern ATL », in : *Proceedings of the 12th Transformation Tool Contest*, à paraître, Eindhoven, The Netherlands, juil. 2019.

[WNS97] Mark WALLACE, Stefano NOVELLO et Joachim SCHIMPF, « ECLiPSe : A platform for constraint logic programming », in : *ICL Systems Journal* 12.*1* (1997), p. 159-200.

[WK98] Jos B. WARMER et Anneke G. KLEPPE, *The Object Constraint Language : Precise Modeling With Uml*, Addison-Wesley, 1998, ISBN : 0-201-37940-6.

[WW09] Matthias WESTPHAL et Stefan WÖLFL, « Qualitative CSP, Finite CSP, and SAT : Comparing Methods for Qualitative Constraint-based Reasoning », in : *Twenty-First International Joint Conference on Artificial Intelligence*, 2009, URL : https://www.aaai.org/ocs/index.php/IJCAI/IJCAI-09/paper/view/435/725.

[WB93] Molly WILSON et Alan BORNING, « Hierarchical constraint logic programming », in : *The Journal of Logic Programming* 16.*3* (1993), p. 277-318, ISSN : 0743-1066, DOI : 10.1016/0743-1066(93)90046-J, URL : http://www.sciencedirect.com/science/article/pii/074310669390046J.

LISTE DES ACRONYMES

ACP **A**lgebra for **C**ommunicating **P**rocesses 63
AOF **A**ctive **O**peration **F**ramework 29
AMPL **A** **M**athematical **P**rogramming **L**anguage 77
API **A**pplication **P**rogramming **I**nterface 87
ASP **A**nswer **S**et **P**rogramming 42
ATL **A**TLAS **T**ransformation **L**anguage 10
CAO **C**onception **A**ssistée par **O**rdinateur 85
COP **C**onstraint **O**ptimization **P**roblem 34
CHR **C**onstraint **H**andling **R**ules 56
CSP **C**onstraint **S**atisfaction **P**roblem 32
DAG **D**irected **A**cyclic **G**raph 66
DSL **D**omain **S**pecific **L**anguage 11
EBNF **E**xtended **B**ackus-**N**aur **F**orm 15
ELK **E**clipse **L**ayout **K**ernel 100
EMF **E**clipse **M**odeling **F**ramework 21
GMF **G**raphical **M**odeling **F**ramework 21
HCSP **H**ierarchical **C**onstraint **S**olving **P**roblem 35
IDM **I**ngénierie **D**irigée par les **M**odèles 9
JTL **J**anus **T**ransformation **L**anguage 42
MDE **M**odel-**D**riven **E**ngineering 13
MOF **M**eta-**O**bject **F**acility 14
MTBE **M**odel **T**ransformation **B**y **E**xample 44
OCL **O**bject **C**onstraint **L**anguage 17
OMG **O**bject **M**anagement **G**roup 13
POO **P**rogrammation **O**rientée **O**bjet 13
RCC8 **R**egion **C**onnection **C**alculus 8 37
SAT **B**oolean **S**atisfiability **P**roblem 36
SBSE **S**earch-**B**ased **S**oftware **E**ngineering 42
SCSP **S**oft **C**onstraint **S**atisfaction **P**roblem 36
SGBD **S**ystème de **G**estion de **B**ase de **D**onnées 31

TCSP **T**emporal **C**onstraint **S**atisfaction **P**roblem 56
TGG **T**riple **G**raph **G**rammar 26
UML **U**nified **M**odeling **L**anguage 14
WCSP **W**eighted **C**onstraint **S**atisfaction **P**roblem 36
XMI **X**ML **M**etadata **I**nterchange 19
XML E**x**tensible **M**arkup **L**anguage 99
YAMTL **Y**et **A**nother **M**odel **T**ransformation **L**anguage 28

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

université angers

Titre : Exploration d'ensembles de modèles

Mot clés : transformation de modèles, programmation par contraintes, ingénierie dirigée par les modèles

Resumé : La transformation de modèles a prouvé qu'elle était un moyen efficace pour produire des modèles cibles à partir de modèles sources le tout en raisonnant en terme de métamodèle. La majorité des techniques de transformation de modèles se concentrent sur la génération d'un modèle cible pour une source donnée. Cependant, il existe des situations dans lesquelles il est préférable de considérer une transformation générant un ensemble de modèles cibles. Un tel ensemble pourra ensuite être exploré par l'utilisateur afin de sélectionner un modèle ayant des propriétés spécifiques.

Dans ce manuscrit, nous proposons une solution alliant transformation de modèles et programmation par contraintes pour permettre l'expression et l'exploration de ces ensembles de modèles. Nous proposons deux implémentations fonctionnelles de cette approche ainsi que plusieurs cas d'étude créés avec ces deux implémentations.

Title : Model Sets Exploration

Keywords : model transformation, constraints programming, model driven engineering

Abstract : Model transformation has proven to be an effective technique to produce target models from source models. Most transformation approaches focus on generating a single target model from a given source model. However there are situations where a collection of possible target models is preferred over a single one. Such situations arise when some choices cannot be encoded in the transformation. Then, search techniques can be used to help select a target model having specific properties.

In this thesis, we present an approach combining model transformation with constraint solving to generate and explore these model sets. Moreover, we present two implementations of this approach along with multiple case studies showcasing these implementations and there usefulness.