Une approche déclarative pour la génération de modèles

Adel Ferdjoukh

Université de Montpellier

Adel Ferdjoukh

Une approche déclarative pour la génération de modèles

Thèse de doctorat - 2016

Collège Doctoral du Languedoc-Roussillon

THÈSE

Pour obtenir le grade de Docteur

Délivré par l'**Université de Montpellier**

Préparée au sein de l'école doctorale **I2S**
Et de l'unité de recherche **LIRMM**

Spécialité: **Informatique**

Présentée par **Adel Ferdjoukh**

Une Approche Déclarative pour la Génération de Modèles

Soutenue le 20 octobre 2016 devant le jury composé de

M. Jean-Michel Bruel	Pr	Univ. Toulouse 2	Rapporteur
M. Michel Rueher	Pr	Univ. Sophia Antipolis	Rapporteur
M. Franck Barbier	Pr	Univ. de Pau	Examinateur
Mme Carmen Gervet	Pr	Univ. de Montpellier	Examinatrice, Présidente
Mme Marianne Huchard	Pr	Univ. de Montpellier	Directrice de thèse
Mme Clémentine Nebut	MdC	Univ. de Montpellier	Encadrante de thèse
M. Éric Bourreau	MdC	Univ. de Montpellier	Invité
Mme Annie Chateau	MdC	Univ. de Montpellier	Invitée

Remerciements

Devenir docteur est sans aucun doute un rêve et une grande fierté mais c'est surtout beaucoup, beaucoup de travail et du stress incessant. Trois catégories de personnes participent et aident à l'accomplissement d'un tel projet : Encadreurs et personnel du laboratoire, compagnons de galère et soutiens psychologiques.

J'ai eu la chance d'être entouré par un encadrement de thèse exceptionnel, les meilleurs du monde tout simplement ! Je tiens à les remercier infiniment d'avoir partagé ces années en ma compagnie et de m'avoir appris le métier de chercheur (et même celui d'enseignant). Il y a d'abord Annie, la personne la plus organisée et efficace à ma connaissance. La polyvalence d'Eric vous rend tout plus clair et plus simple. Clémentine est la boss dont tout le monde rêve. Elle est attentive, compréhensive et possède un sens du détail hors du commun. Enfin, il y a Marianne. C'est la personne qui arrête tout ce qu'elle fait pour vous écouter et s'occuper de tout, même des prises de courant à ce qu'il paraît !

Je remercie Anne-Élisabeth, Rémi, Rodolphe, Vincent, Roland, Christophe, Chouki, Jamel, Hind, Christelle et Sylvain pour tout ce que j'ai appris à leur contact.

Sans Laurie, Nico, Caroline, Virginie et Élisabeth rien de ce qui concerne l'administration n'est possible.

Je tiens également à remercier mes compagnons de galère que sont Julien, Emmanuel, Guillerme, Vincent, Pauline et Myriam, ou encore Iago, Seza, Zak, Sabrina, Yannick, Valentin, et bien d'autres. Nos déjeuners, séminaires, semindoc, débats, discussions et sorties ont rendu ces trois ans et demi plus courts et plus agréables. Vous verrez, un jour viendra où l'informatique et le salaire à vie rendront nos vies beaucoup plus cool !

Dans la catégorie des soutiens psychologiques, il y a évidement mes parents, petit frère, la famille et mes amis (Nassim, Kahled, Massi, Toufik et Mazigh pour ne citer que ceux-la).

Je tiens tout particulièrement à remercier mon frère Bibouh pour les blagues échangées par réseau social interposé et toutes les vacances passées ensemble.

Je dédie cette thèse aux petits Amine et Maya.
Franchement, c'était le pied !

I vava ak d yemma

Un message n'atteint réellement sa cible que lorsqu'il est écrit dans la langue que la personne ciblée comprend le mieux. M'adressant à mes parents, je rédige dans la toute première langue qu'ils m'ont enseigné. *Le paragraphe qui va suivre est donc écrit en kabyle. Je m'en excuse auprès des personnes qui ne pourront pas le lire.*

Deg taddart n talwaqaḍi γer usunti d doctora deg tsdawit n Montpellier, abrid iḍul, amecwar γezif. εedan attas isseggasen n leqraya : 6 deg uγerbaz n ssuq aqdim, 3 deg uγerbaz n ukemel n ssuq ufella, 3 deg tesnawit n ḥimadiyin deg Bgayet, 4 deg tesdawit n abderaḥman u mira n Bgayet ak d 4 deg tesdawit n Montpellier.

Maεna tura dayen leqraya tswḍed ar taggara s lemεwna-nwen. Deg zik kounwi tdemrem-aγ isnin ar leqraya, γas aken rabεa iseggasen aki ig fouken tesεad-aten zdefir n skype mačči udem ar wudem. Dacu tezgam d aseqsi ak d udemer aken an-ṣaweḍ ar wanecta.

Ihi bγiγ ad-iniγ tanmirt-nwen tameqrant ak d teγzi lεemr-nwen. Incalah att ḥeḍrem kter w kter.

إلا سحر

إنني متأكد أن الشكر لا يصل إلا قلب شخص إلآ إذا كتب بلغته الأولى، لهذا إنني سوف أكتب هذه الفقرة بالغة العربية.

إذا في الأخير أريد أن أشكر شخصا كان موجودا في السراء و في الضراء، طيلة العام الأخير من هذا المشوار الصعب و الطويل. لم تكن السنة المنتهية سهلة، على العكس. لكن بفظلكي، بوجودكي و مساندكي تمكن لي ختم شهادة الدكتورا بنجاح. أود إن أشكرك و أتمتى لكي التوفيق في الأشهر الأخيرة المتبقية لختم شهادتكي،

Table des matières

Table des figures

Liste des tableaux

Listings

Liste des acronymes

$\mathcal{G}$RIMM $\mathcal{G}$eneRating Instances of Meta-Models

$\mathcal{G}$ЯRIMM $\mathcal{G}$enerating ЯandomIzed and Relevant Instances of Meta-Models

CP Constraint Programming (Programmation par Contraintes)

CSP Constraints Satisfaction Problem (Problème de Satisfaction de Contraintes)

DSML Domain Specific Modelling Language

EMF Eclipse Modelling Framework

IDM Ingénirie Dirigée par les Modèles

MDA Model Driven Architecture

MDE Model Driven Engineering (IDM)

MOF Meta-Object Facility

OCL Object Constraint Language

OMG Object Management Group

PL Programmation Linéaire

SAT SATisfiability

SMT Satisfiability Modulo Theory

UML Unified Modelling Language

Xcsp Xml constraint satisfaction problem

Xmi XML meta-data interchange

Ne dites pas : “j’ai trouvé la vérité”, mais plutôt : “j’ai trouvé une vérité”.

Gibran Khalil Gibran

If you steal from one author it’s plagiarism ; if you steal from many it’s research.

Wilson Mizner

Chapitre 1

Introduction

Yettnadi γef iẓuran n tagut (Il cherche les racines du brouillard)

Proverbe Kabyle

Synopsis

Préambule

Ce chapitre présente le contexte de la thèse et sa problématique, à savoir la génération automatique de modèles (instances) définis par des méta-modèles (section 1.1). Les contributions et les réalisations qui ont été menées à leur terme au cours de cette thèse sont listées et introduites à la section 1.2. Enfin le plan de ce manuscrit est donné à la section 1.3.

1.1 Contexte et problématique

Quel est le point commun entre un nouveau compilateur d'un langage de programmation académique tel que Nit [83], une approche d'observation de modèles sociaux-environnementaux [86] et la reconstruction de génomes par scaffolding en bioinformatique [24] ? À première vue, il n'existe aucune similitude entre ces travaux issus de trois domaines très différents. Cependant, lorsque nous nous intéressons aux protocoles expérimentaux de validation et plus précisément aux données qu'ils utilisent, nous nous apercevons que les trois travaux possèdent au moins deux points communs : leurs données se modélisent sous forme d'un modèle logiciel à haut niveau d'abstraction, aussi appelé méta-modèle et les trois approches souffrent d'un manque sévère de données de tests ; elles sont rares ou indisponibles, ou bien elles sont coûteuses à obtenir et nécessitent respectivement, des programmeurs en grand nombre, des observations sur le terrain dans les forêts de Madagascar et du séquençage d'ADN basé sur une réaction en chaîne par polymérase.

Comme les exemples précédents l'illustrent, la problématique de disposer de données structurées en grande quantité est primordiale pour beaucoup de domaines. De manière plus spécifique en génie logiciel, on s'intéresse à cette problématique pour répondre à deux questions majeures : comment savoir si les modèles logiciels conçus capturent bien le domaine qu'ils prétendent modéliser d'une part, et quelles données utiliser pour tester les programmes qui manipulent ces mêmes modèles d'autre part. La difficulté réside dans la structure complexe et la syntaxe propre des modèles ce qui rend ardue la tâche de les écrire à la main, notamment, lorsqu'une grande quantité et des tailles importantes de ces modèles sont nécessaires à une utilisation donnée.

La solution présentée dans cette thèse est la génération automatique de ces données, que nous allons, par la suite, désigner par "modèle" ou "instance". Les données générées sont spécifiées par un modèle abstrait qui est nommé "méta-modèle". Nous proposons donc d'instancier les méta-modèles pour générer automatiquement des modèles qui respectent les spécifications des méta-modèles. Les modèles générés automatiquement seront ensuite utilisés pour répondre à deux objectifs principaux :

1. Aider les experts d'un domaine à valider ou corriger les spécifications de leurs méta-modèles à travers des modèles ou des instances générés dont certaines caractéristiques se rapprochent des modèles réels.

2. Tester les programmes qui transforment ces modèles avec des données de tailles, de structures et de caractéristiques diverses et variées.

Ces deux objectifs principaux de la génération automatique de modèles poussent à se poser une question d'une grande importance pour la suite : comment valider une approche de génération de modèles, c'est-à-dire, quels sont les critères de qualité qui permettront de répondre aux deux objectifs précédents ? Nous retenons cinq critères qu'un processus de génération doit assurer :

— **Validité.** Les modèles doivent respecter toutes les spécifications de leur méta-modèle et toutes les contraintes qui l'accompagnent. Le langage qui nous servira à écrire ces contraintes est OCL (Object Constraint Language).

— **Passage à l'échelle.** L'approche doit être capable de générer des instances de grandes tailles pour des méta-modèles de grandes tailles également, dans le but de tester le passage à l'échelle des programmes manipulant ces instances.

— **Vraisemblance.** Les modèles doivent posséder des caractéristiques aussi proches que possible des instances réelles. Cela permettra à un expert de les utiliser pour valider son méta-modèle.

— **Diversité.** Les instances générées doivent offrir une large diversité et couvrir de la meilleure manière possible l'espace des solutions.

— **Automatisation.** Le processus doit être automatique pour être accessible à des experts de domaines aussi variés que ceux cités plus haut.

Plusieurs approches de génération de modèles existent [13, 15, 19, 31, 74, 94, 106]. Divers paradigmes informatiques sont utilisés pour répondre à cette problématique. Cependant, l'étude de ces approches a montré qu'aucune d'entre elles ne répond aux cinq critères en même temps. Dans cette thèse, nous proposons une nouvelle approche de génération de modèles qui se conforme à tous les critères de qualité d'un processus de génération de modèles en même temps.

1.2 Contributions et réalisations

L'approche que nous proposons génère des modèles en se basant sur la programmation par contraintes (CSP). Les éléments d'un méta-modèle donné et ses contraintes sont modélisés en CSP. La résolution de ce dernier par un solveur de contraintes permet de calculer des instances valides. Les contributions principales de la thèse sont :

— Une modélisation efficace d'un méta-modèle en CSP qui offre un bon passage à l'échelle. Une formalisation d'une partie des constructions du langage de contraintes OCL en CSP assure la validité des modèles générés. Ces modélisations en CSP prennent soin d'utiliser au mieux les outils qu'offre la programmation par contraintes (telles les contraintes globales) et de les appliquer efficacement au génie logiciel.

— Une technique pour améliorer le réalisme, la pertinence et la vraisemblance des instances générées. Nous utilisons des métriques spécifiques à un domaine dans le but d'inférer des lois de probabilités usuelles. La simulation de ces dernières et leur intégration au CSP améliorent sensiblement la vraisemblance des solutions.

— Des métriques de distances qui mesurent la similarité entre les modèles deux à deux. Nous avons étudié puis adapté aux modèles logiciels des distances mathématiques ou de graphes. Des techniques de clustering sont ensuite utilisées pour sélectionner

les modèles les plus éloignés parmi un ensemble de modèles donné ou généré. Enfin, l'algorithmique génétique permet d'améliorer la diversité de l'ensemble de modèles.

— Un outil de génération de modèles, nommé $\mathcal{G}$RIMM, regroupe toutes ces contributions.

Le schéma de la figure 7.1 illustre le cheminement de la thèse. Il associe les contributions aux chapitres du manuscrit.

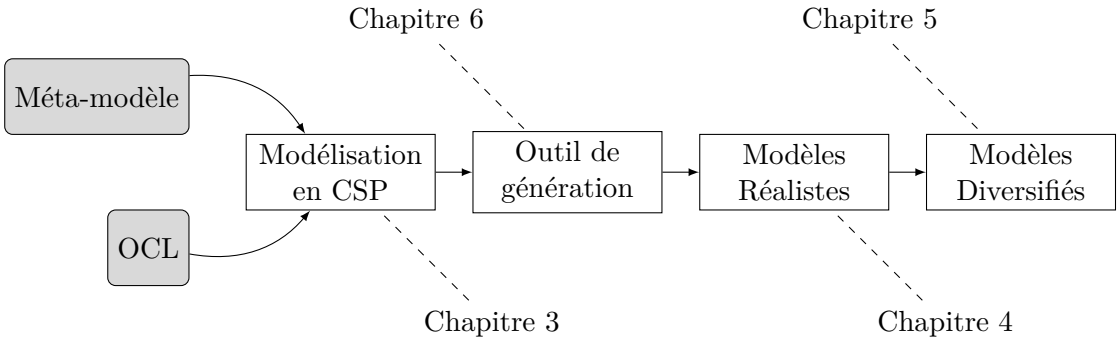


FIGURE 1.1 – Schématisation du plan de la thèse.

1.3 Plan du manuscrit

La suite du manuscrit se structure de la façon suivante. **Le chapitre 2** est consacré à la présentation des domaines qui constituent le contexte de la thèse, à savoir l'ingénierie dirigée par les modèles et la programmation par contraintes. L'état de l'art détaillé des outils de génération de modèles et des approches apparentées ou équivalentes est dressé par la suite. Les approches sont comparées par catégorie d'appartenance et selon divers critères de performance, d'expressivité, d'accessibilité, ou encore, de qualité.

Les chapitres qui suivent (3→6) détaillent les contributions et les réalisations de la thèse.

Le chapitre 3 présente la modélisation en programmation par contraintes d'un méta-modèle et de ses contraintes OCL. Plusieurs expérimentations sont menées pour montrer le bon passage à l'échelle de la solution proposée. Ces contributions ont permis la publication de deux articles en conférences internationales et la soumission d'un article de revue.

— A. Ferdjoukh, A.-E. Baert, A. Chateau, R. Coletta, and C. Nebut. A CSP Approach for Metamodel Instantiation. In *ICTAI, IEEE International Conference on Tools with Artificial Intelligence*, pages 1044–1051, 2013

— A. Ferdjoukh, A.-E. Baert, E. Bourreau, A. Chateau, R. Coletta, and C. Nebut. Instantiation of Meta-models Constrained with OCL: a CSP Approach. In *MODELSWARD, International Conference on Model-Driven Engineering and Software Development*, pages 213–222, 2015

Les modèles générés à l'aide de la programmation par contraintes uniquement, ne répondent pas à toutes les exigences en terme de pertinence et de vraisemblance des solutions, notamment pour les rendre utiles. Pour remédier à cela, **le chapitre 4** s'intéresse à cette problématique et propose une solution basée sur les métriques spécifiques aux domaines et la simulation de lois de probabilités usuelles pour améliorer la pertinence des instances générées. Deux études de cas issues de deux domaines différents (programmation objet et bioinformatique) sont menées pour montrer la faisabilité de l'approche. Ce travail est publié dans un article de conférence internationale.

— A. Ferdjoukh, E. Bourreau, A. Chateau, and C. Nebut. A Model-Driven Approach to Generate Relevant and Realistic Datasets. In *SEKE, International Conference on Software Engineering & Knowledge Engineering*, pages 105–109, 2016

Le chapitre 5 se penche sur la question de la diversité des modèles en général et de ceux que nous générons en particulier. Ainsi, il tente de répondre à la question "*Qu'est ce qui fait que deux modèles sont différents et comment quantifier cette différence ?*" par la proposition de trois métriques de distances entre modèles issues de distances mathématiques et de graphes. Des techniques de clustering sont ensuite utilisées pour identifier et sélectionner les modèles les plus diversifiés parmi un ensemble de modèles donné. Par ailleurs, nous avons implémenté une technique pour améliorer la diversité d'un ensemble de modèles. Une partie de ces travaux a donné lieu à la publication d'un article en conférence internationale.

— F. Galinier, E. Bourreau, A. Chateau, A. Ferdjoukh, and C. Nebut. Genetic Algorithm to Improve Diversity in MDE. In *META, International Conference on Metaheuristics and Nature Inspired Computing (To appear)*, 2016

Toutes les contributions de cette thèse sont implantées dans un outil de génération de modèles nommé $\mathcal{G}$RIMM. Il est décrit et testé au **chapitre 6**. $\mathcal{G}$RIMM a été présenté dans un poster aux journées du GDR GPL.

— A. Ferdjoukh. $\mathcal{G}$RIMM: un assistant à la conception de méta-modèles par génération d'instances. Poster, GDR GPL, 2015

Pour finir, une conclusion générale et des pistes d'améliorations ou d'applications sont données au **chapitre 7**.

Chapitre 2

Contexte et État de l'Art

> La physique donne le combien, la métaphysique le comment.
>
> Georges-Louis Leclerc de Buffon

Synopsis

Préambule

Ce chapitre s'intéressera, d'abord, à l'introduction de l'ingénierie dirigée par les modèles, contexte de ce travail de thèse (section 2.1). Un état de l'art détaillé des approches de génération de modèles (section 2.2) et celles d'assistance à la création de méta-modèles (ou apparentées comme telles, section 2.3) sera présenté par la suite. Enfin, nous introduirons et nous motiverons le choix de la programmation par contraintes, outil principal utilisé pour les contributions de la thèse (section 2.4).

2.1 Ingénierie Dirigée par les Modèles

Dans la section qui va suivre l'utilisation du terme modèle est omniprésente. Il s'agit du thème central de tout ce qui va être dit. Le modèle sera défini à la section 2.1.2.1. Néanmoins, pour une meilleure compréhension de cette section, nous donnons une brève définition de ce terme. Ainsi, parmi la dizaine de définitions du mot modèle dans le dictionnaire de la langue française, la plus proche de l'idée qu'on se fait du modèle en génie logiciel est :

> *Objet type à partir duquel on reproduit des objets de même sorte à de multiples exemplaires.*

Quand on consulte le dictionnaire de la langue anglaise, deux définitions retiennent notre attention :

> *A three-dimensional representation of a person or thing or of a proposed structure, typically on a smaller scale than the original.*

> *A simplified description, especially a mathematical one, of a system or process, to assist calculations and predictions.*

La notion de modèle en génie logiciel regroupe toutes ces caractéristiques à la fois. D'abord, il simplifie la reproduction de systèmes à de multiples exemplaires. Ensuite, il est de plus petite taille que le système qu'il décrit, pour qu'il soit plus simple de manipuler le modèle que directement le système lui-même. Enfin, il aide à effectuer des traitements sur le système sans prendre le risque d'altérer l'intégrité de ce dernier.

2.1.1 Historique de la modélisation en informatique

L'augmentation rapide de la taille et de la complexité des logiciels a poussé les informaticiens à la recherche d'outils et langages pour représenter de manière abstraite certains de leurs aspects. Cette représentation a pour but de faciliter la conception, la réalisation et la manipulation de ces systèmes. Depuis les tous premiers qui sont apparus dans années 1960, une multitude de langages de modélisation a été adoptée. Certains de ces langages ne sont plus d'actualité ou alors ont fusionné pour en créer de nouveaux, quant à d'autres, ils sont toujours très utilisés. La figure 2.1 s'inspire du travail de Tolvanen [102] pour retracer la chronologie des langages de modélisation qui ont le plus marqué l'histoire de l'informatique. Cette liste n'est néanmoins pas exhaustive.

Réseaux de Petri

Les réseaux de Petri ont été inventés par Carl Adam Petri [26],[79]. Un réseau de Petri est un outil mathématique utilisé pour la modélisation de systèmes à processus concurrents. Il est représenté par un graphe biparti contenant deux types de noeuds : les places et les transitions. Une place est marquée par un nombre entier (nombre de jetons). Il s'exécute

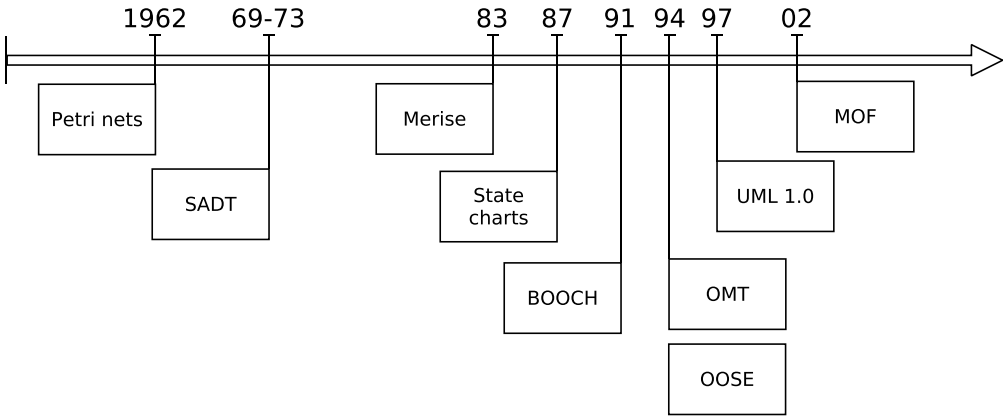


FIGURE 2.1 – Chronologie des principaux langages de modélisation utilisés en informatique.

par le franchissement des transitions. Après chaque étape d'exécution, des jetons d'une place sont déplacés dans une place située une transition plus loin. La figure 2.2 montre un réseau de Petri à deux états différents. La transition `Réserver` est franchie et un jeton de la place `Places libres` est déplacé vers la place `Places réservées`.

Le fait d'avoir une représentation graphique rend le réseau de Petri accessible à des personnes avec un bagage mathématique moins conséquent que ce qui est nécessaire pour l'utilisation de modèles d'équations. Cette caractéristique leur a permis d'être adoptés dans différents domaines. En informatique, les réseaux de Petri sont notamment utilisés pour la modélisation de processus d'exécution ou des protocoles de communication.

SADT

SADT (Structured Analysis and Design Technique) est une méthode de conception et de description des systèmes complexes par analyse fonctionnelle descendante. Elle a été développée aux État-Unis par Douglas T. Ross à partir de 1969 [88].

Dans une conception SADT, un système est décomposé en sous-taches de plus en plus spécifiques (Analyse fonctionnelle descendante). Une boite SADT décrit une tâche donnée et possède des entrées (gauche), des sorties (droite), des mécanismes de réalisations (bas) et des flux de contrôles (haut). Des boites peuvent être assemblées de telle sorte que la sortie d'une boite sera l'entrée de la suivante. La figure 2.3 montre un exemple de diagramme SADT.

Merise

Merise est une méthode française de conception de systèmes d'information [101] basée sur le modèle relationnel de Edgar Frank Codd [27]. Elle a été utilisée principalement en

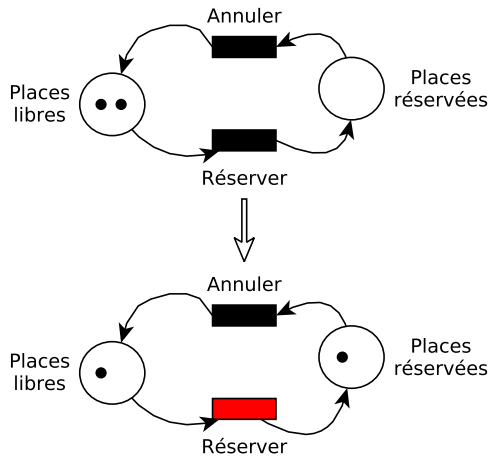


FIGURE 2.2 – Évolution d'un réseau de Petri simple.

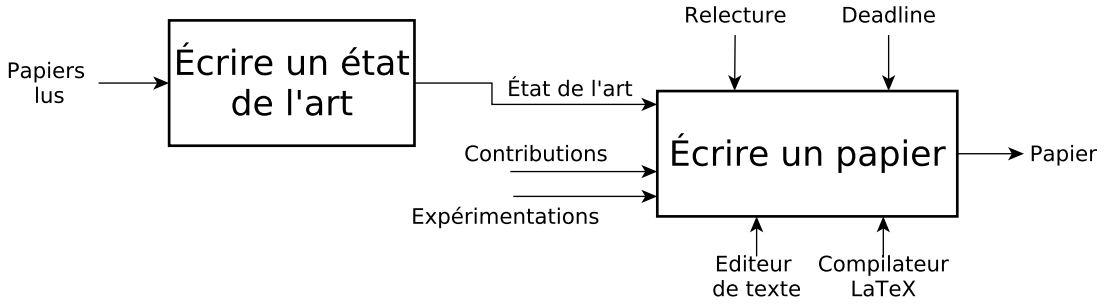


FIGURE 2.3 – Un diagramme SADT décrivant le processus de rédaction d'un papier.

France mais connaît des équivalents dans le monde anglo-saxon telles que SSADM [3].

La méthode s'appuie sur un certain nombre de diagrammes modélisant différents aspects du système. Par exemple, le MCD (Modèle Conceptuel des Données) pour les données, ou encore le MCT (Modèle Conceptuel des Traitements) pour les tâches.

State charts

Le state chart est utilisé pour la description du comportement des systèmes complexes à événements discrets. Il se base sur les automates à état finis et fonctionne donc sous la forme d'états et de transitions.

Il a, d'abord, été présenté en 1987 par David Harel dans [53]. Désormais, il fait partie des diagrammes UML spécifiés par l'Object Management Group (OMG) depuis 2000 et la version 1.3 d'UML [1].

Unified Modelling Language

Unified Modelling Language (UML) est un langage de modélisation graphique fondamentalement orienté objet [92]. Il fut normalisé par l'OMG en 1997. Il est issu de la fusion des 3 langages du début des années 1990 : Booch (par Grady Booch dans [9]), OMT (par James Rumbaugh [91]) et OOSE (par Ivar Jacobson [58]).

Dans sa version 2.5 de juin 2015, UML comporte 14 diagrammes parmi lesquels on peut citer : le diagramme de classes, le diagramme d'objets, le diagramme de cas d'utilisation, le diagramme d'activité, les machines à états.

Aujourd'hui, UML est l'un des langages de modélisation les plus connues et les plus utilisées pour la conception de logiciels.

Meta-Object Facility

Meta-Object Facility (MOF) est un standard de l'Object Management Group [2] apparu en 2002 [76]. Il s'agit d'un environnement où les modèles peuvent désormais être exportés à partir d'une application et importés par une autre, ou encore, transportés à travers un réseau ou stockés dans des dépôts prévus à cet effet. De plus, il est possible de les transformer et les utiliser directement pour générer du code source. MOF a par ailleurs influencé EMF (2004, [99]), le framework de modélisation d'Eclipse et Ecore, son langage de méta-modélisation.

MOF a pour particularité de se placer au dessus de tous les langages et méthodes de modélisation qu'on a vu jusqu'ici (en terme de niveaux d'abstraction). Nous allons notamment revenir avec plus de détails sur la relation entre MOF et UML dans la prochaine section [3].

1. UML, OMG Specification: `http://www.omg.org/spec/UML`
2. OMG, Organisation internationale qui fait la promotion des technologies objets.
3. MOF specification: `http://www.omg.org/spec/MOF/`

2.1.1.1 Discussion

L'historique présenté ci-dessus montre que la modélisation logicielle a connu beaucoup d'évolution et de changement au cours de ses cinquante années d'existence. Une multitude de paradigmes (donc de langages) a été utilisée pour parvenir à modéliser des programmes informatiques : le calcul matriciel avec les réseaux de Petri, l'algèbre relationnelle avec Merise et les automates à états pour state chart. À partir des années 1980, avec l'émergence du paradigme orienté objet, plusieurs langages de modélisation orientés objet ont vu le jour. Le plus connu d'entre eux est UML.

Le point commun entre tous ces langages de modélisation est que le modèle conçu s'avère être en dehors du programme lui-même. Ainsi, on peut utiliser un réseau de Petri pour simuler le comportement des acteurs d'un protocole de communication sans pour autant s'en servir pour coder le protocole lui-même, ou encore ses acteurs. Il sert en général de passerelle de communication entre les différentes personnes impliquées dans le projet. Son rôle peut également être lié à la documentation de l'application dans le but, par exemple, de faciliter sa maintenance lorsqu'un changement d'équipe survient. Mêne lorsque les modèles décrivent des comportements complexes d'un système, le passage du modèle sur papier au code se fait systématiquement de façon manuelle.

Au début de ce siècle, une nouvelle méthode de développement logiciel, l'ingénierie dirigée par les modèles (IDM), est apparue. La nouveauté de l'IDM est que les modèles s'empilent sur plusieurs niveaux d'abstraction. Par ailleurs, le modèle est placé au centre du processus de développement logiciel. Les modèles sont structurés et manipulés par des programmes. Ainsi, il est possible d'exporter les modèles d'une application et de les importer par une autre. Ils peuvent être transformés et stockés. Leur rôle est maintenant de participer à la réalisation des applications. Ils servent par exemple à générer une partie du code source. Désormais, il est possible d'automatiser le passage du modèle vers le code.

2.1.2 Ingénierie Dirigée par les Modèles

L'ingénierie dirigée par les modèles est une branche du génie logiciel dans laquelle les modèles sont l'artefact de base. Ils interviennent à toutes les étapes du développement logiciel. En IDM, les modèles subissent différentes opérations de transformation qui produiront d'autres modèles ou bien généreront du code.

La structuration et la manipulation des modèles par des programmes sont rendues possibles par l'empilement des modèles sur plusieurs niveaux d'abstraction. Ainsi, un modèle décrit un système réel, et au même temps le modèle est lui même défini par un modèle plus abstrait appelé méta-modèle. La figure 2.4 montre l'architecture à 4 niveaux de hiérarchie en IDM.

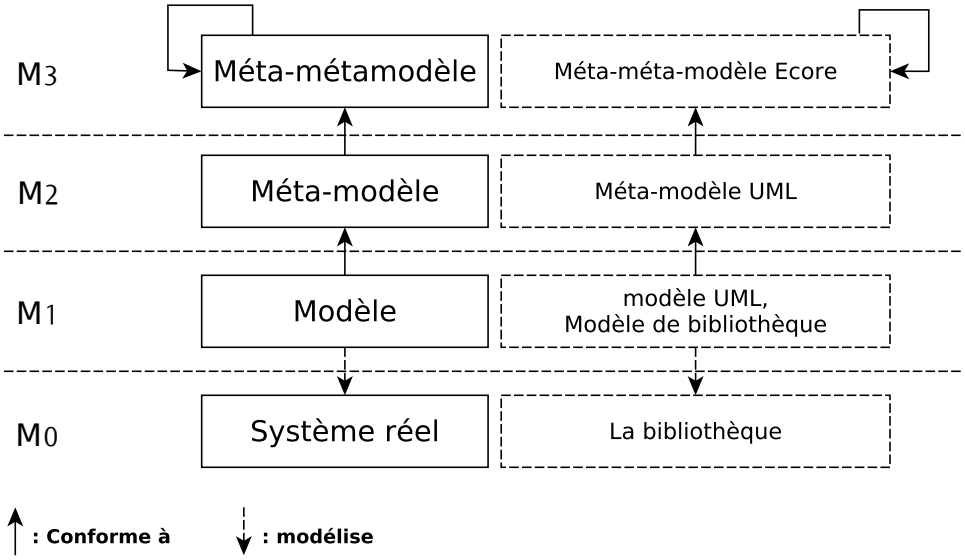


FIGURE 2.4 – Architecture de modèles à 4 niveaux en IDM. À droite : des exemples concrets.

2.1.2.1 Modèle

Il n'existe aucune définition universelle du modèle. Néanmoins, beaucoup de travaux en Ingénierie dirigée par les modèles s'accordent sur une compréhension commune. Les travaux de Jézéquel et al. [59], eux-mêmes inspirés de ceux de Muller et al. [75] recensent une dizaine de définitions. Nous avons choisi la définition suivante. Elle reprend les points importants et communs à toutes les définitions rencontrées.

Definition 1 *(***Modèle***) la représentation abstraite d'un certain aspect d'un système. Il a pour but la conception, la réalisation ou la validation de cet aspect du système.*

Pour expliquer les différents concepts présentés à cette section, nous prendrons comme exemple la création d'une application de gestion d'une bibliothèque. La figure 2.5 est un extrait d'un diagramme de classe UML (un modèle) modélisant la structure d'une bibliothèque.

2.1.2.2 Méta-modèle

Pour qu'un programme puisse manipuler le modèle de bibliothèque précédent, un niveau d'abstraction plus haut est nécessaire. Il faut donc définir le modèle du modèle (ou le méta-modèle).

Definition 2 *(***Méta-modèle***) un modèle permettant la spécification d'un langage de modélisation.*

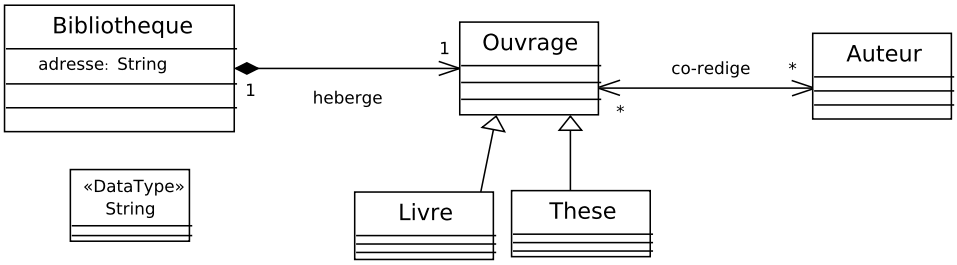


FIGURE 2.5 – Extrait d'un diagramme de classe UML modélisant une bibliothèque.

La figure 2.6 est un extrait du méta-modèle UML décrivant quelques éléments du diagramme de classe. Tout élément du diagramme de classe modélisant la bibliothèque est instance d'un élément du méta-modèle UML. On dira donc que le diagramme de la bibliothèque est conforme au méta-modèle UML.

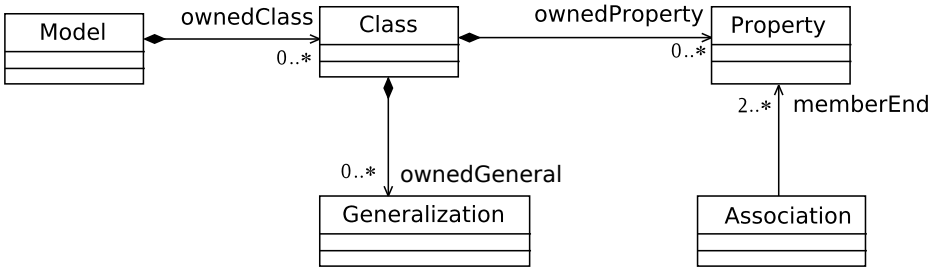


FIGURE 2.6 – Extrait du méta-modèle UML décrivant quelques éléments du diagramme de classe. Il est écrit conformément au méta-métamodèle Ecore.

La figure 2.7 montre la relation qui existe entre les éléments du modèle et ceux du méta-modèle. On voit que chaque élément du modèle est instance d'un élément de son méta-modèle.

Definition 3 *(***Méta-métamodèle***) un modèle permettant la spécification d'un langage de méta-modélisation.*

Dans le but d'éviter l'empilement infini des couches de modélisation, il est convenu que tout méta-métamodèle se définit lui-même.

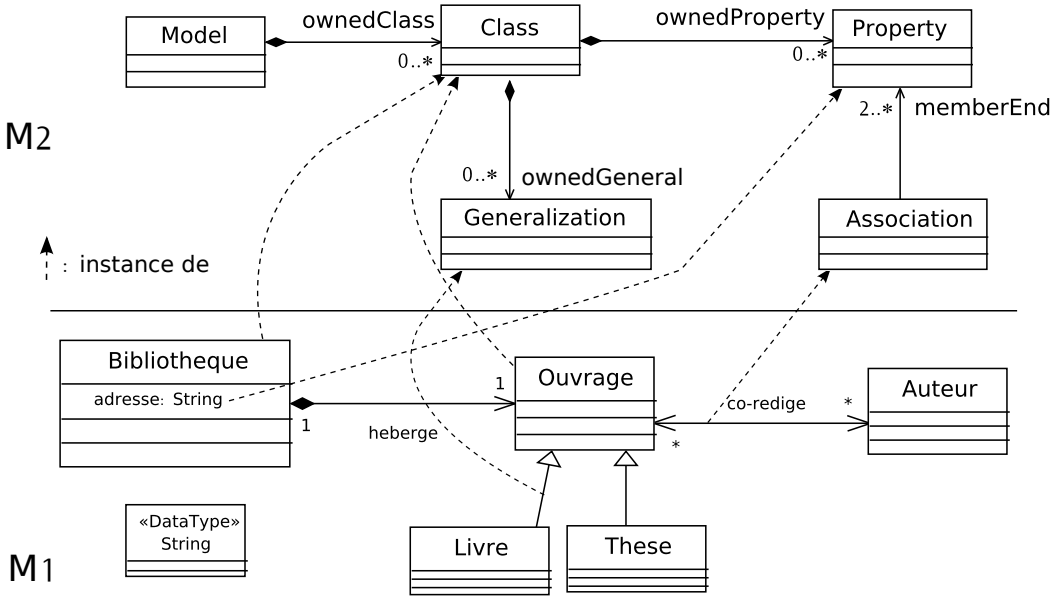


FIGURE 2.7 – La relation qui existe entre un modèle (M1) et son méta-modèle (M2).

2.1.2.3 Transformation de modèles

Maintenant que nous disposons de modèles conformes à des méta-modèles et qui peuvent être manipulés par des programmes, la prochaine étape d'un processus IDM est justement d'écrire ces programmes appelés transformations de modèles. La transformation de modèle est l'opération clé en IDM, elle permet la génération de code, le refactoring, la migration de logiciel, etc.

Ainsi, en partant d'un diagramme de classe UML indépendant des langages de programmation, nous pourrons générer une partie du code de l'application bibliothèque dans le langage de programmation voulu, à condition de disposer également de son méta-modèle. Pour cela, il faut procéder à la transformation du modèle UML en un modèle de code conforme au méta-modèle d'un langage de programmation (Figure 2.8).

Dans la pratique, il existe de nombreux environnement supports à l'IDM. Le plus connu d'entre eux est le framework EMF (Eclipse Modelling Framework) [99]. Nous l'avons d'ailleurs utilisé pour la mise en œuvre des contributions de cette thèse.

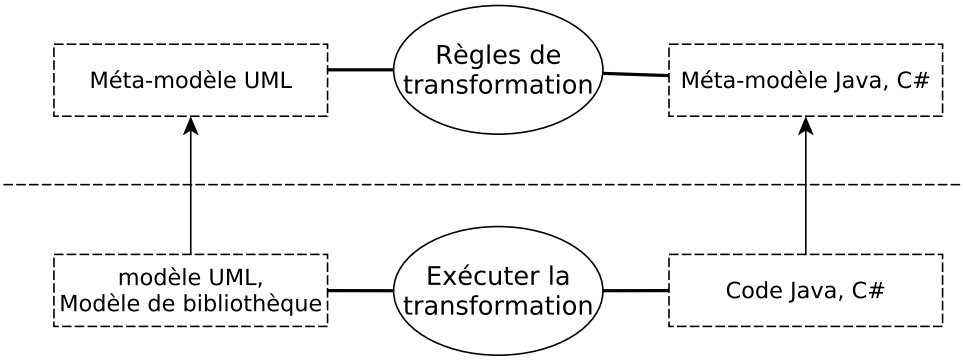


FIGURE 2.8 – Schéma de transformation d'un modèle UML en modèle de code Java, C♯.

2.1.3 Object Constraint Language

La structure d'un méta-modèle n'est pas suffisamment fine pour décrire tous les aspects du langage qu'il spécifie. Il y a besoin d'écrire des contraintes additionnelles qui éclaircissent certaines ambiguïtés ou ajoutent des informations sur les éléments des modèles. L'OMG préconise d'écrire ces contraintes en OCL (Object Constraint Language) [77].

OCL [4] est un langage de spécification de contraintes sur les élements d'un modèle (ou d'un méta-modèle). Il possède l'avantage d'être formel sans qu'il y ait besoin d'un bagage mathématique important pour l'utiliser. Par ailleurs, une contrainte OCL n'a pas d'effet de bord. Son évaluation retourne une valeur mais ne modifie pas l'état de l'élément qu'elle concerne.

Il est notamment utilisé pour spécifier des invariants sur les classes et les types d'un modèle ou d'un méta-modèle et des pré- et des post-conditions d'opérations ou méthodes.

Une expression OCL s'écrit dans le contexte d'un type donné. Le mot réservé `self` se réfère aux instances du type contexte de l'expression.

OCL ▷
```
Context These inv :
self.author→size() = 1
```

La contrainte OCL précédente est écrite dans le contexte de la classe `These` du diagramme de classes de la figure 2.5. Elle stipule qu'une thèse possède un et un seul auteur. La figure 2.9 montre deux extraits de diagrammes d'instances de bibliothèque. Le diagramme à droite viole la contrainte OCL introduite car il définit une thèse `t1` qui possède deux co-auteurs `Alan` et `Denis`, ce qui est évidemment en contradiction avec la contrainte.

4. OCL, OMG Specification: `http://www.omg.org/spec/OCL`

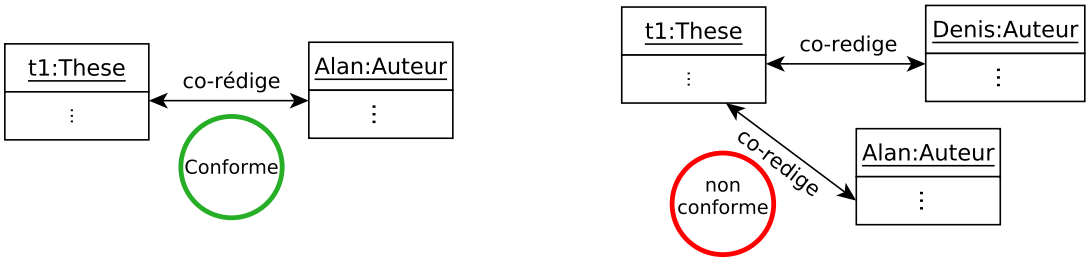


FIGURE 2.9 – Deux extraits de diagramme d'instances de bibliothèque. à gauche : il respecte la contrainte OCL, à droite : il viole la contrainte.

2.2 Approches de Génération de modèles

Dans ce qui suit, nous faisons une étude comparative des principales approches de génération de modèles et des principaux paradigmes utilisés pour répondre à cette problématique depuis plusieurs années.

Nous avons étudié les approches de génération de modèles décrites dans la littérature et pour chaque approche, nous nous sommes intéressés aux points suivants :

— Objectif : objectif de l'approche ou utilité des modèles générés.

— Paradigmes : paradigmes ou technologies auxquels l'approche fait appel.

— Historique : courte liste des approches qui l'ont inspiré (si elle existe).

— Description : processus de génération, entrées, sorties.

— Discussion : points forts et points faibles.

La discussion d'une approche concerne tout ou partie des cinq caractéristiques principales que le processus et les modèles générés doivent remplir : (1) Automatisation, (2) Validité (OCL), (3) Passage à l'échelle, (4) Pertinence et (5) Diversité.

2.2.1 Approche basée sur les arbres aléatoires

Objectif de l'approche : l'objectif est de générer des modèles de très grande taille pour tester des transformations de modèles.

Paradigmes : la génération d'arbres aléatoires et la Méthode Boltzmann (*voir glossaire*).

Description : Mougenot et al. décrivent dans [74] une approche de génération d'instances de méta-modèles de très grande taille. Le but de cette approche est de générer de manière aléatoire et uniforme des modèles contenant jusqu'à des millions d'éléments. Une grammaire à contexte libre est construite à partir de l'arbre représentant le méta-modèle. Il est à

noter qu'au moment de construire l'arbre, seules les relations de compositions sont prises en compte.

Un générateur de mots du langage de la grammaire est ensuite écrit. Chaque mot représente un squelette de modèle (car contenant uniquement les relations de composition). Les paramètres du générateur, c'est à dire, les probabilités de choisir telle ou telle règle de la grammaire d'abord sont, soit choisis manuellement, ou bien calculés. Ces probabilités ont un impact très important sur la vraisemblance des modèles générés.

Discussion : Le principal avantage de cette approche est qu'elle génère des structures dont la taille peut atteindre plusieurs millions d'éléments. Cependant, deux principaux inconvénients apparaissent. Premièrement, seule la structure arborescente du méta-modèle est prise en compte, c'est-à-dire, les relations de composition seulement. Ceci produit des squelettes de modèles et non des modèles en bonne et due forme. Deuxièmement, les contraintes OCL du méta-modèle ne sont en aucun cas prises en compte lors de la génération.

Les auteurs on voulu se pencher sur la pertinence des modèles générés. Ainsi, ils ont constaté que les modèles générés uniformément n'avaient pas les caractéristiques désirées en terme de distribution des éléments, car la méthode Boltzmann ne permet pas de prendre en compte des lois de probabilités autres que l'uniforme. Pour essayer de passer outre cette limitation de la méthode, ils proposent que l'utilisateur spécifient des pondérations de façon à améliorer la distribution des éléments du modèle.

2.2.2 Approche basée sur les grammaires de graphes

Objectif de l'approche : le but affiché de l'approche est de générer un nombre suffisant de modèles pour tester certains aspects des transformations de modèles, comme par exemple le passage à l'échelle.

Paradigmes : les grammaires de graphes et le concept de Graphe typé étiqueté borné avec héritage (*voir glossaire*).

Description : Ehrig et al. proposent une approche de génération de modèles basée sur les grammaires de graphes [31],[32]. Il s'agit de construire des grammaires de graphes à partir des méta-modèles dans le but d'instancier ces derniers. Les grammaires de graphes sont un ensemble de transformations que l'on peut appliquer sur un graphe pour en construire un autre. Un graphe (dans ce cas un modèle conforme) est construit par l'application successive de ces règles. Les grammaires de graphes permettent de définir des langages avec une forte base formelle. L'approche propose de dériver une grammaire de graphes à partir du méta-modèle afin de rendre opérationnelle la génération de modèles. La grammaire obtenue doit pouvoir générer toutes les instances possibles du méta-modèle et ne doit générer aucun modèle qui ne soit pas instance du méta-modèle. Le concept de Graphe typé étiqueté borné

avec héritage est utilisé ici également pour dériver une grammaire de graphes à partir du méta-modèle.

Discussion : D'abord, la dérivation d'une grammaire de graphes à partir d'un méta-modèle est une opération fastidieuse. Elle n'est d'ailleurs pas entièrement automatisée. Ensuite, cette méthode passe difficilement à l'échelle. En effet, les auteurs ont montré uniquement des petits exemples jouets de méta-modèles pour lesquels l'approche peut générer des instances conformes. Par ailleurs, la grammaire de graphe ne concerne que la structure du méta-modèle, les contraintes OCL ne peuvent y être intégrées. Enfin, concernant la diversité et la vraisemblance des solutions générées, aucune solution n'est proposée par les auteurs.

2.2.3 Approche basée sur la programmation par contraintes

Objectif de l'approche : elle vise à générer des modèles conformes à des méta-modèles dans le but de valider ces derniers.

Paradigmes : l'approche utilise la programmation par contraintes (Problème de Satisfaction de Contraintes, *voir glossaire*).

Source d'inspiration : Ces travaux s'inspirent des travaux de Cadoli et al [21] et de Malgouyres et al. [69] qui ont présenté deux premières approches basées sur la programmation par contraintes pour vérifier la satisfiabilité des diagrammes UML.

Description : Cabot et al. [15],[16] s'intéressent à la vérification de diagrammes UML annotés par des contraintes OCL. Ils considèrent qu'un modèle UML est valide s'il existe une instanciation conforme à ce modèle, respectant ses contraintes OCL. Pour ce faire, ils traduisent les modèles UML et leurs contraintes en CSP. La satisfaction du CSP obtenu leur garantit la validité du modèle.

Les mêmes auteurs ont étendu leurs précédents travaux aux méta-modèles conformes à Ecore dans [51]. Cette fois-ci un modèle, conforme à un méta-modèle donné, est généré automatiquement pour s'assurer de la validité du méta-modèle. Dans cette approche, tous les éléments d'un méta-modèle et ses contraintes OCL sont traduits en CSP. Le solveur ECL^{i}PS^e résout le CSP et toute solution est un modèle conforme au méta-modèle.

Discussion : La nouveauté ici par rapport aux approches précédentes basées sur les CSP est la prise en compte des contraintes OCL. Les contraintes OCL permettent l'obtention d'instances valides.

Le passage à l'échelle de la solution proposée est limitée par une modélisation en CSP peu efficace. En effet, le nombre important de variables, la quasi absence de contraintes

globales et l'utilisation de contraintes qui ne propagent pas très bien ralentissent considérablement l'approche et ne lui permettent pas de bien passer à l'échelle. Par ailleurs, les auteurs ne se sont intéressés pas à la pertinence ou à la diversité des modèles générés.

Note

*L*a section 2.4 aborde la programmation par contraintes avec plus de détails.

2.2.4 Approche basée sur les contraintes Alloy/SAT

Objectif de l'approche : elle vise à tester des transformations de modèles en leur fournissant des données en entrée, qui sont des modèles générés.

Paradigmes : l'approche utilise les contraintes Alloy. Le solveur Alloy opère par réduction vers SAT (*voir glossaire*).

Description : Sen et al. [94, 95] ont proposé une méthode de génération d'instances de méta-modèles basée sur une traduction en Alloy. Alloy est défini, dans le livre de référence [57], comme étant un langage de description de structures et un outil pour les analyser. Un modèle Alloy est une collection de contraintes qui décrivent un ensemble de structures équivalentes. Le solveur Alloy est capable, à partir de cette collection de contraintes et par réduction vers SAT, de trouver des instanciations qui les satisfont.

Discussion : Cette approche est automatisée pour ce qui concerne la traduction du méta-modèle en Alloy. Par contre, l'intervention de l'utilisateur est nécessaire pour encoder les contraintes OCL. De plus, toutes les constructions du langage OCL ne sont pas prises en compte.

Par ailleurs, les auteurs ne se sont pas penchés sur les problématiques de diversité et des vraisemblance des solutions générées.

2.2.5 Approche basée sur les fragments de méta-modèle

Objectif de l'approche : elle vise à générer des modèles conformes à des méta-modèles dans le but de tester des transformations de modèles.

Paradigmes : Fragments de méta-modèle (voir glossaire).

Source d'inspiration : Fleurey et al. proposent dans [38] une technique de partitionnement de méta-modèle permettant le découpage de ce dernier en fragments. L'assemblage des fragments permet la construction d'instances du méta-modèle.

Description : Dans [13], Brottier et *al.* proposent un algorithme itératif qui permet de générer des modèles en partant d'un méta-modèle et de son ensemble de fragments. Un fragment est une petite partie du méta-modèle produit par le partitionnement de celui-ci [38]. L'algorithme présenté permet de générer un ensemble de modèles couvrant tous les fragments.

Discussion : Le processus de génération est automatisé et permet d'obtenir un ensemble de modèles différents et couvrant l'espace des solutions par la couverture de tous les fragments. Néanmoins, les contraintes OCL ne sont pas étudiées. Il en va de même pour la pertinence des solutions générées.

2.2.6 Approche basée sur le recuit simulé

Objectif de l'approche : elle génère un ensemble de modèles conformes à des méta-modèles pour tester les transformations de modèles.

Paradigmes : Recuit simulé (*voir glossaire*) et fragments de méta-modèles.

Source d'inspiration : Cette approche reprend la technique de fragmentation de méta-modèle [38] dans le but de maximiser la diversité des modèles générés.

Description : Cadavid et al. [19] ont utilisé la technique du recuit simulé (Simulated Annealing) [62],[22] pour la génération de modèles de test conformes à un méta-modèle.

Le recuit simulé est une méta-heuristique qui s'inspire d'un processus analogue en métallurgie. En optimisation, le but est d'atteindre l'optimum d'une fonction donnée par l'alternance de cycles d'amélioration puis de dégradation de la solution.

Les auteurs s'intéressent, par ailleurs, à la diversité et ont pour cela utilisé des métriques de couverture de l'espace des solutions basées sur les fragments de méta-modèles. Par exemple, lorsqu'un ensemble de modèles est généré, ils s'assurent de l'apparition de tous les fragments du méta-modèle parmi l'ensemble de modèles générés.

Discussion : Les auteurs sont les seuls à s'être intéressés à la diversité des modèles générés et avoir répondu à cette problématique. L'utilisation des fragments de méta-modèles et la mise au point de métriques dans ce sens permettent de couvrir l'espace des solutions et donc d'améliorer la diversité.

Cependant, la prise en compte des contraintes OCL est rendue impossible par la technique du recuit simulé et la pertinence des modèles générés n'est pas étudiée. Par ailleurs, le passage à l'échelle de la solution semble mauvais (évaluation limitée de cet aspect).

2.2.7 Approche basée sur les contraintes SMT

Objectif de l'approche : son but est de générer des modèles dans l'optique de vérifier la validité des méta-modèles et leurs contraintes OCL, et de tester des transformations de modèles.

Paradigmes : l'approche utilise les contraintes SMT (Satisfiability Modulo Theory, *voir glossaire*) ..

Description : Wu et al. [106] transforment un méta-modèle en une instance de SMT (SAT Modulo Theory) dans le but de générer des modèles. Au cours de l'opération de traduction, une représentation basée sur un Graphe typé étiqueté borné avec héritage a été adoptée. Cette représentation est très utile pour représenter des modèles du génie logiciel. Un solveur SMT [29] se compose de deux choses, un solveur SAT classique pour résoudre la partie SAT du problème et des solveurs spécifiques qui permettent le support d'autres types de calculs (par exemple, arithmétique, simplex). Les auteurs se sont également intéressés au traitement des contraintes OCL du méta-modèle pour lesquelles ils décrivent un certain nombre de règles de transformation en SMT, et à la vraisemblance des modèles générés.

Discussion : Cette approche est entièrement automatique et traite une partie non négligeable du langage OCL. Les auteurs se sont également intéressés à la pertinence des solutions générées. Ils ont utilisé des mesures basées sur les graphes pour répondre à cette problématique. Ils vérifient que les modèles générés respectent certains propriétés de graphes comme par exemple l'acyclicité.

Par ailleurs, les auteurs ne se sont pas intéressés à la diversité des solutions produites.

2.2.8 Discussion générale

Nous avons identifié cinq critères de comparaison entre les différentes approches décrites plus haut :

Automatisation La plupart des approches évaluées sont automatisées. Il subsiste, néanmoins, deux approches qui ne le sont pas entièrement. Ainsi, dans l'approche par grammaires de graphes, le processus de dérivation de la grammaire de graphes à partir du méta-modèle nécessite l'intervention de l'utilisateur, et la transformation de l'OCL doit se faire de manière manuelle pour l'approche basée sur les contraintes Alloy.

Validité et prise en compte de l'OCL Il est important pour une approche de génération de modèles de produire des instances conformes aux méta-modèles et respectant leurs contraintes OCL. On peut remarquer de grandes disparités dans le traitement de l'OCL par les approches existantes. D'abord, il y a les approches qui ne traitent pas du tout l'OCL car le paradigme utilisé les en empêche. C'est le cas des arbres aléatoires, des grammaires de graphes, des fragments de méta-modèles et du recuit simulé. Ensuite, dans l'approche basée sur les contraintes Alloy, des petites parties du langage OCL sont prises en compte. Enfin, les approches par CSP et SMT encodent une plus grande partie du langage OCL.

Passage à l'échelle La possibilité de générer des grands modèles conformes à des méta-modèles de grande taille est un autre critère important pour une approche de génération de modèles. L'approche qui génère les plus gros modèles est celle des arbres aléatoires. Elle peut générer des modèles contenant jusqu'à plusieurs millions d'éléments. Néanmoins, ces modèles ne sont en fait que des squelettes de modèles et ne sont donc pas valides. Certaines approches comme celles des fragments de méta-modèle et du recuit simulé ne donnent aucune information sur la taille des modèles qui peuvent être générés.

Pertinence des modèles générés Seulement deux approches se sont intéressées à la pertinence et la vraisemblance des modèles qu'elles produisent.

Dans l'approche par arbres aléatoires, les auteurs ont observé sur des exemples que les modèles générés grâce à la distribution uniforme intrinsèque à la méthode Boltzmann possèdent des caractéristiques éloignées des modèles réels. Pour répondre à cette problématique, ils proposent que l'utilisateur spécifie des pondérations sur les fréquences d'apparition des éléments. Par exemple, en UML un package contient n classes, où n est donné par l'utilisateur.

L'approche basée sur les contraintes par SMT, quant à elle, génère des modèles respectant des métriques basées sur certaines propriétés de graphes. Par exemple, l'approche vérifie que le graphe généré est acyclique ou qu'un nœud du graphe n'est pointé que par autre nœud. Ceci par exemple assurera l'acyclicité de l'héritage en UML et évitera qu'un attribut appartienne à deux classes différentes.

Diversité des modèles générés Seules les deux approches basées sur les fragments de méta-modèle se sont penchées sur ce problème. Elles définissent des fonctions objectifs qui s'assurent de la couverture des fragments d'un méta-modèle par un ensemble de modèles conformes générés.

Ces fonctions sont paramétrées pour vérifier, par exemple, la couverture des n fragments une fois chacun, ou encore, de leur couverture m fois chacun. La diversité de la distribution des fragments assure ainsi la diversité des modèles générés.

La table 2.1 résume la comparaison entre les différentes approches selon les 5 critères. Les pictogrammes indiquent pour chaque critère s'il est : bien respecté : ⊕, pas respecté : ⊖, plutôt respecté : ⊕, plutôt pas respecté : ⊖ ou s'il n'existe aucune indication à son sujet : ?.

Auteurs	**Paradigme**	**Critères de comparaison**				
		Auto	**OCL**	**Échelle**	**Pertinence**	**Diversité**
Mougenot [74]	Arbre aléatoire	⊕	⊖	⊕	⊕	⊖
Ehrig [31, 32]	Grammaire de graphe	⊖	⊖	⊕	⊖	⊖
Cadavid [19]	Recuit simulé	⊕	⊖	?	⊖	⊕
Cabot [16, 15]	CSP	⊕	⊕	⊖	⊖	⊖
Sen [94, 95]	Alloy	⊖	⊕	⊖	⊖	⊖
Brottier [13]	Fragments	⊕	⊖	?	⊖	⊕
Wu [106]	SMT	⊕	⊕	⊖	⊕	⊖

TABLE 2.1 – Comparaison entre les principales approches de génération de modèles selon 5 critères.

2.2.9 Conclusion

La comparaison montre qu'aucune des approches étudiées ne répond à toutes les exigences en même temps. Pour certaines approches, il est même impossible de prendre en compte certaines caractéristiques du fait du paradigme employé, c'est le cas généralement pour les contraintes OCL, ou encore les contraintes sur la pertinence et la diversité des modèles générés.

Il y a néanmoins deux paradigmes qui sont susceptibles de gérer des contraintes en lien avec tous les critères à satisfaire. Il s'agit de la programmation par contraintes (CSP) et Sat Modulo Theory (SMT).

Nous avons choisi de développer une approche de génération de modèles basée sur la programmation par contraintes. Les raisons qui ont motivé ce choix sont listées en section 2.4.5.

2.3 Outils d'assistance à la modélisation

Cette section dresse un état de l'art de deux catégories d'outils. Chacune d'entre elles est traitée dans une sous-section :

— section 2.3.1, outils d'aide au design et à la validation des modèles (ou des méta-modèles) : Dans cette partie nous présentons les différents outils -générant ou non des modèles- dont le rôle est d'aider les concepteurs de méta-modèles dans leur démarche.

Cela peut se présenter sous différentes formes, donner un retour sur les instances possibles, ou encore détecter des erreurs de modélisation ou de syntaxe.

— section 2.3.2, outils de création de syntaxes concrètes : Cette sous-section aborde les environnements de création de syntaxe concrète pour les langages de modélisation spécifiques. Cela permet une meilleure visualisation des instances d'un tel langage.

Nous ajouterons que les environnements de création de syntaxes concrètes nous ont intéressé, bien qu'ils ne génèrent pas des modèles à proprement parler. En effet, ils peuvent être utilisés en complément d'une approche de génération de modèles dans le but de visualiser les modèles produits, graphiquement et dans la syntaxe graphique du DSML (Domain Specific Modelling Language, *voir glossaire*). Par ailleurs, la création de syntaxes graphiques pour les méta-modèles est un bon moyen de les vérifier. Les auteurs ont ainsi montré dans [81] que beaucoup de corrections pouvaient être apportées aux méta-modèles par la création d'une syntaxe graphique.

2.3.1 Design et validation de modèles

Lightning

Gammaitoni et al. présentent l'outil *Lightning* [44]. En réalité, il s'agit d'un workbench basé sur le langage Alloy. Il est intégré à Eclipse et permet de créer des langages de modélisation (DSML), que ce soit la syntaxe concrète ou la syntaxe abstraite. L'approche a pour but de faciliter la détection d'erreurs de modélisation au niveau des méta-modèles. Néanmoins, l'utilisation de l'éditeur Alloy est relativement non intuitive, notamment pour des personnes habituées à l'environnement Eclipse (langages OCL, UML, etc).

USE

Gogolla et al. [48],[49] présentent USE, un outil d'instanciation de modèles UML contraints par OCL. Cet outil se base sur un langage de script que les auteurs ont développé, ASSL (A Snapshot Sequence Language). L'utilisateur voulant instancier un modèle UML écrit en ASSL les spécifications du diagramme d'objet qu'il veut générer. Par exemple, il indiquera le nombre d'instances par classe, les valeurs à affecter aux attributs et quels objets relier entre eux. L'outil se chargera de compiler toutes ces données pour construire le diagramme d'objets correspondant si les contraintes OCL le permettent. L'utilisateur peut ainsi corriger son modèle UML ou les contraintes OCL en fonction de l'instance qu'il voit (syntaxe graphique d'un diagramme d'objets). Cependant, un biais d'utilisation subsiste. En effet, l'utilisateur choisit le nombre d'instances, les valeurs des attributs et les objets à relier par les associations ; tandis que l'outil ne fait aucun choix, dans la mesure où il se contente de vérifier la cohérence des choix avec les contraintes OCL définies précédemment.

Outil de Boufares et al.

Boufares et al. [10],[11] utilisent la programmation linéaire pour vérifier la consistance des cardinalités, d'abord, dans les modèles à entités-relations puis, dans un deuxième temps pour les modèles UML. L'approche prend en paramètre un diagramme de classes UML et traduit en PL (Programmation Linéaire) ses cardinalités. Le programme linéaire obtenu est ensuite résolu pour conclure sur la consistance ou non des cardinalités du diagramme de classe UML.

EMFtoCSP

Cabot et al. [17] présentent une approche et un outil dont l'objectif est de donner un retour à l'utilisateur pour lui permettre de corriger son modèle UML et/ou ses contraintes OCL. Un modèle UML et ses contraintes OCL sont transformées en CSP. Ce dernier est résolu pour fournir à l'utilisateur un diagramme d'objets visualisé graphiquement si une solution existe bel et bien. Les auteurs considèrent que l'utilisateur pourra ainsi corriger d'éventuels problèmes sur le modèle en regardant les diagrammes d'instances générés automatiquement.

2.3.1.1 Discussion

Nous comparons ces différentes méthodes d'assistance à la méta-modélisation en nous basant sur les critères suivants (la table 2.2 résume cette comparaison) :

Génération d'instances Certaines approches de génération de modèles le font dans le but d'assister les utilisateurs dans leur tâche de méta-modélisation. C'est notamment le cas de Cabot et al.. Les approches de Boufares et al. et de Gammaitoni et al. ne génèrent pas de modèles.

Éditeur de méta-modèles (ou modèles) Les approches de Gammaitoni et al. et de Gogolla et al. disposent d'éditeurs pour la création de méta-modèles ou de modèles dans un langage dédié. Les deux autres approches, quant à elles, se basent sur des environnements existants, comme Eclipse.

Intervention de l'utilisateur Nous remarquons, suivant les approches, qu'il existe une grande disparité dans le rôle donné à l'utilisateur au cours du processus de méta-modélisation. Pour Gammaitoni et al., l'utilisateur édite son méta-modèle dans un framework dédié et l'outil vérifie sa cohérence. Pour les autres approches, l'utilisateur a seulement besoin de paramétrer un générateur. En ce qui concerne l'outil USE, l'utilisateur édite son méta-modèle dans un langage dédié pour ensuite paramétrer le constructeur d'instances.

Détection d'erreur Seules deux approches détectent automatiquement les erreurs dans le méta-modèle voulu. Boufares et al. ne le font, néanmoins, que pour les cardinalités des associations. Les approches de Gogolla et al. et de Cabot et al. laissent l'utilisateur juger si le modèle qui est généré contient ou non des erreurs.

Auteur	**Outil**	**Critères de comparaison**								
		Génération			**Éditeur**		**Utilisateur**		**Détection**	
		auto	manu	non	oui	non	édite	param.	auto	manu
Gammaitoni [44]	Lightning			●	●		●		●	
Gogolla [48, 49]	USE		●		●		●	●		●
Boufares [10, 11]	?			●		●		●	●	
Cabot [17]	EMFtoCSP	●				●		●		●

Table 2.2 – Comparaison entre les principales approches d'assistance à la méta-modélisation.

2.3.2 Environnements de création de syntaxe concrète

Diagraph

Pfister et al. présentent dans [80],[81] une méthode et un outil nommé *Diagraph* pour la création de syntaxes concrètes (graphiques) pour les langages de modélisation spécifiques aux domaines (DSML). Diagraph prend en entrée des méta-modèles Ecore. La syntaxe graphique d'un élément est défini par une simple annotation. Ceci rend la tâche de création de syntaxe concrète très facile à réaliser car tout se fait au moment de l'édition du méta-modèle.

Eugenia

Eugenia [5] est un outil de construction de syntaxe concrète pour les méta-modèle Ecore. Il a été présenté par Kolovos et al. dans [63]. Il se base sur des annotations à ajouter aux éléments du méta-modèle pour décrire leur syntaxe graphique en choisissant parmi un ensemble de formes et de styles disponibles.

Obeo designer

Obeo designer [6] [60] est un outil commercial de création de langages de modélisation spécifiques au métier créé par l'entreprise du même nom. Il utilise une représentation arborescente du méta-modèle (relations de composition) et permet de définir un style pour les éléments du méta-modèle dans le but de leur construire une syntaxe concrète.

2.3.2.1 Discussion

Nous comparons ces outils de création de syntaxes graphiques selon trois critères :

Intégration à Eclipse les trois outils sont intégrés à l'environnement de modélisation d'Ecplise ce qui facilite grandement leur utilisation.

Distribution Diagraph et Eugenia sont des outils académiques réalisés par des chercheurs. Il sont accessibles gratuitement. Obeo designer est, quant à lui, un outil commercial, payant pour sa version professionnelle.

Méthode d'édition Obeo offre un éditeur arborescent de syntaxe graphique complètement indépendant du modèle. Il offre plus de possibilités de personnalisation. Diagraph et Eugenia sont plus faciles d'utilisation. En effet, des annotations sur les éléments du méta-modèles suffisent à générer leur syntaxe graphique.

5. Eugenia: `http://www.eclipse.org/epsilon/doc/eugenia/`
6. Obeo designer: `http://www.obeodesigner.com`

La table 2.3 récapitule la comparaison selon les critères présentés.

Auteur	**Outil**	**Critères de comparaison**							
		Eclipse/EMF		**Open-source**		**Édit. graphique**		**Édit. textuel**	
		oui	non	oui	non	oui	non	oui	non
Pfister [80],[81]	Diagraph	●		●			●	●	
Kolovos et al. [63]	Eugenia	●		●			●	●	
Juliot et Benois [60]	Obeo	●			●	●			●

Table 2.3 – Comparaison entre les principaux outils de création de syntaxe concrète.

2.4 Programmation par Contraintes

Tout le monde sait ce que sont les contraintes et en gère quotidiennement : contraintes d'horaires, contraintes administratives, contraintes de budget, etc. La programmation par contraintes ([41],[89]) est le paradigme qui permet de traduire tous ces différents types de contraintes sous forme informatique dans un formalisme dédié. Son but est de trouver des solutions qui satisfont ces contraintes. L'utilisation des contraintes couvre, d'ores et déjà, une multitude de domaines : gestion des emploi du temps, affectation de personnels, planification des tâches, optimisation des ressources, etc. Les contraintes traitent généralement des problèmes à forte combinatoire.

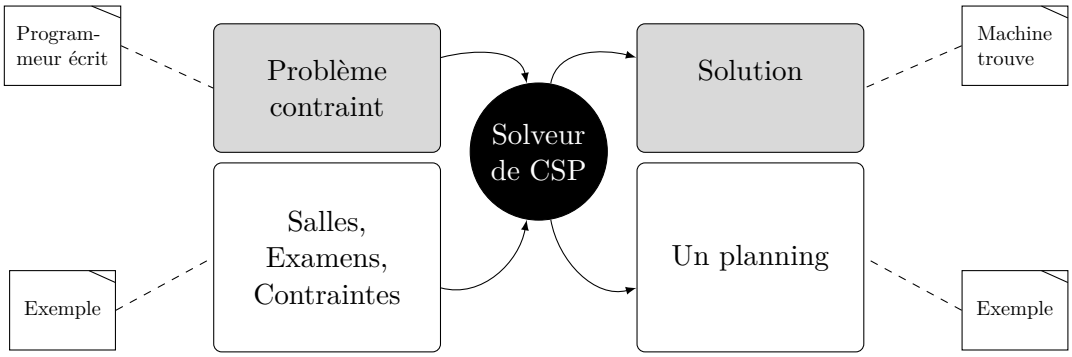


FIGURE 2.10 – Schématisation du principe de la programmation par contraintes. Exemple : un solveur se voit donné en entrée des salles, des examens et leurs contraintes (capacité des salles et horaires imposés) et a pour objectif de trouver un planning qui satisfera toutes les contraintes.

Le principe de fonctionnement de la programmation par contraintes (décrit à la figure 2.10) est le suivant : un programmeur a pour tâche de décrire (modéliser) une situation sous forme d'un problème de satisfaction de contraintes (CSP) et l'ordinateur, ou plus exactement le solveur de contraintes trouve une (ou des) solution(s) respectant le problème défini c'est à dire que chaque contrainte s'assure d'être satisfaite.

2.4.1 Définitions

Mackworth définit les CSP dans [68] comme suit : "*We are given a set of variables, a domain of possible values for each variable, and a conjunction of constraints. Each constraint is a relation defined over a subset of the variables, limiting the combination of values that the variables in this subset can take. The goal is to find a consistent assignment of values from the domains to the variables so that all the constraints are satisfied simultaneously.*".

Definition 4 *(**CSP**) Un problème de Satisfaction de Contraintes se définit comme un triplet* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *où :*

— $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ *est un ensemble de* n *variables.*

— *$\mathcal{D}$ est une application donnant le domaine de chaque variable :*

$$\begin{aligned} \mathcal{D}: \quad & \mathcal{X} \to P(\mathbb{Z}) \\ & x_i \mapsto \mathcal{D}(x_i) \subset \mathbb{Z} \end{aligned}$$

toute variable x_i possède un domaine fini $\mathcal{D}(x_i)$.

— *$\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ est un ensemble de m contraintes entre les variables.*

Definition 5 *(***Contrainte***) $C_j \in \mathcal{C}$ est une expression booléenne incluant un sous-ensemble de variables $\mathcal{X}(C_j) = \{x_1^j, x_2^j, \ldots, x_m^j\}$ appelé sa portée (scope). L'expression est définie sur $\mathbb{Z}^m$. Un tuple $t \in \mathbb{Z}^m$ satisfait C_j ssi $C_j(t)$ est vrai.*

Definition 6 *(***Solution***) Soit un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$:*

— *L'instanciation I d'un sous-ensemble de variables $Y = \{x_1, x_2, \ldots, x_k\} \subset \mathcal{X}$ est l'affectation des valeurs $v_1, v_2, \ldots, v_k$ aux variables $x_1, x_2, \ldots, x_k$ tel que $v_i \in \mathcal{D}(x_i)$.*

— *Une solution d'un CSP est une instanciation de toutes les variables de $\mathcal{X}$ qui satisfait toutes les contraintes au même temps.*

2.4.1.1 Contraintes globales

Cette partie introduit les contraintes globales et en cite quelques unes des plus utilisées et celles qui nous seront utiles par la suite. Elle ne prétend en aucun cas être une étude exhaustive de ce vaste domaine des contraintes globales. Pour avoir une vue plus globale de ce domaine, referez-vous à [8] ou bien au chapitre sur le sujet (numéro 7) de [89].

Une contrainte est dite Contrainte Globale (*voir glossaire*) si elle relie un nombre k de variables non-fixé à l'avance à l'inverse d'une contrainte unaire (1) ou binaire (2). L'exemple le plus connu est la contrainte `alldiff`$(x_1, x_2, \ldots, x_n)$ qui stipule que n variables doivent avoir des valeurs différentes les unes des autres. Sémantiquement, toute contrainte globale peut être remplacée par une conjonction de contraintes plus petites (plus petite arité). Cette opération est appelée décomposition. Ainsi, la contrainte `alldiff` se remplace par la clique d'inégalités : $x_1 \neq x_2$, $x_1 \neq x_3$, $\ldots$, $x_{n-1} \neq x_n$.

Les contraintes globales présentent l'avantage de capturer des motifs complexes et récurrents en une seule contrainte, comme par exemple, assigner des valeurs différentes à un ensemble de variables, borner le nombre d'occurrences d'un ensemble de valeurs à la fois, etc. Cette caractéristique les rend très utiles car elles sont plus intuitives que leurs décompositions en petites contraintes. Par ailleurs, elle permettent souvent d'obtenir un CSP plus compact et donc plus facile à lire et à corriger en cas d'erreurs.

Au niveau de la résolution, ces contraintes possèdent des algorithmes de filtrage et de propagation dédiés et relativement plus efficaces que les algorithmes par défaut. Ceci implique que dans la plupart des cas, un CSP contenant des contraintes globales est résolu plus rapidement que sa version ou les contraintes globales sont décomposées en petites

◁ *cf. sec. 2.4.2*

contraintes. Même s'il existe des cas extrêmes où le CSP sans contrainte globale est résolu plus efficacement.

Dans ce qui suit, nous donnons une liste non exhaustive des contraintes globales les plus importantes (du moins celles dont on se servira dans la suite de ce manuscrit). Il existe beaucoup d'autres contraintes globales listées dans la catalogue de contraintes globales [7].

Alldiff

CSP ▷ `alldiff`($x_1, x_2, \ldots, x_n$)

Elle stipule que les valeurs affectées à chacune des variables x_i seront toutes différentes les unes des autres. C'est sans doute la contrainte globale la plus connue et la plus utilisée. L'algorithme de filtrage le plus efficace a été proposé par Jean-Charles Régin [87].

Atleast

CSP ▷ `atleast`($N, \{x_1, x_2, \ldots, x_n\}, V$)

Elle stipule que la valeur V possède au moins N occurrences parmi l'ensemble des variables $x_1, x_2, \ldots, x_n$.

Atmost

CSP ▷ `atmost`($N, \{x_1, x_2, \ldots, x_n\}, V$)

Elle stipule que la valeur V possède au plus N occurrences parmi l'ensemble des variables $x_1, x_2, \ldots, x_n$.

Global cardinality (gcc)

CSP ▷ `gcc`($\{x_1, x_2, \ldots, x_n\}, \{V_1, \ldots, V_m\}, \{I_1, \ldots, I_m\}, \{S_1, \ldots, S_m\}$)

Cette contrainte est la généralisation des deux précédentes. Elle borne le nombre d'occurrences de chaque valeur V_i parmi un ensemble de variables $\{x_1, x_2, \ldots, x_n\}$: chaque valeur V_i doit posséder un nombre d'occurrence inférieur I_i et un nombre d'occurrences supérieur S_i : $I_i \leq occu(V_i) \leq S_i$.

Element

CSP ▷ `element`($V, \{x_1, x_2, \ldots, x_n\}, I$)

Elle stipule que la variable $V = x[I]$.

7. Catalogue des contraintes globales: `http://sofdem.github.io/gccat/`

Among

`among`(n, [Varis], [Vals]) ◁ CSP

Elle stipule que n variables de la collection Vars prennent leurs valeurs parmi la collection Vals.

2.4.2 Résolution

La résolution des CSP fait appel à des algorithmes appartenant à deux larges familles : inférence et recherche. Ainsi, la plupart des systèmes de résolution marient des algorithmes issus de ces deux grandes catégories, de la manière la plus efficace possible pour un processus de résolution rapide.

— Inférence : l'objectif des algorithmes d'inférence est de réduire, a priori, la taille des domaines afin d'éliminer des pans entiers de l'espace des possibilités (*i.e.* l'union des domaines).

— Recherche : ce type d'algorithme explore systématiquement tout l'espace et élimine seulement les choix qui ont conduit à des échecs.

Les deux catégories sont très liées et travaillent généralement ensemble. En effet, pour que la recherche soit guidée vers les zones les plus efficaces possibles, il est vital d'avoir une étape d'inférence qui réduira la taille des domaines de façon conséquente.

2.4.2.1 Propagation et filtrage

Les contraintes restreignent les valeurs que prennent les variables. Prenons l'exemple d'une variable x_1 dont le domaine est $\mathcal{D}(x_1) = \{1, 2, \ldots, 10\}$ et qui est présente dans une contrainte $c_1 : x_1 > 5$. Nous déduisons que les valeurs $\{1, 2, 3, 4, 5\}$ sont impossibles pour x_1. L'opération de réduction du domaine d'une variable grâce à une contrainte se nomme filtrage. Elle permet de réduire de façon substantielle l'espace des solutions possibles.

Par ailleurs, il arrive que la variable x_1 soit impliquée dans une autre contrainte c_2 : $x_1 > x_2$, où x_2 est une variable dont le domaine est $\mathcal{D}(x_2) = \{9, 10, 11, 12\}$. Dans ce cas, nous concluons que les valeurs $\{10, 11, 12\}$ sont impossibles pour x_2 car elles sont toutes supérieures ou égales ($\geq$) à la valeur maximale de x_1. La valeur 9 est donc la seule possibilité pour x_2. Ce processus de communication de valeurs entre contraintes se nomme propagation.

Les deux techniques de filtrage et de propagation permettent de réduire la taille de l'espace des possibilités et ainsi de résoudre les CSP plus efficacement.

2.4.2.2 Backtracking et Énumération

La recherche de solutions en CSP se base sur une fouille arborescente. À chaque niveau de l'arbre des possibilités (voir figure 2.11), on procède à l'affectation de valeurs à une variable et chaque nœud représente une valeur possible pour ladite variable. Une solution

est atteinte en arrivant aux feuilles. Lorsqu'à un niveau de l'arbre, toutes les branches sont épuisées, c'est-à-dire, qu'il n'y a plus aucune valeur possible pour une variable donnée, un retour-arrière (backtracking) est opéré pour tenter de modifier un choix effectué une étape plus haut dans l'arbre.

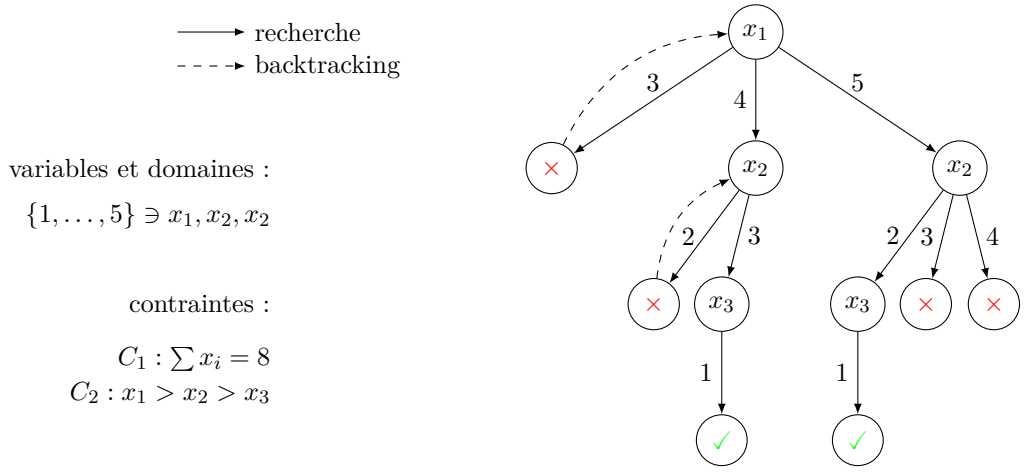


FIGURE 2.11 – Recherche arborescente montrant le mécanisme de backtracking.

Le choix d'affecter telle ou telle variable ou valeur en premier est primordial et peut avoir une incidence importante sur le nombre de nœuds à visiter et donc sur le temps de résolution. Ces choix sont dictés par ce qu'on appelle une stratégie d'énumération. La stratégie la plus triviale est l'ordre de déclaration des variables et des valeurs. Néanmoins, d'autres stratégies bien plus malines peuvent être envisagées, comme par exemple la déclaration d'un ordre lexicographique ou le choix de variables suivant des critères tels que la taille de leur domaine (plus petit domaine d'abord), ou encore le nombre de contraintes dans lesquelles elle apparaissent (variable la plus populaire d'abord). On peut également choisir la prochaine variable ou valeur à traiter de façon aléatoire. Les différentes stratégies d'énumération de variables sont étudiées par Gent et al. dans [46].

2.4.2.3 Solveur de contraintes

Un solveur de contraintes est un programme dont la fonction est de résoudre les CSP. Concrètement, sa tâche principale est d'affecter des valeurs à toutes les variables tout en respectant toutes les contraintes simultanément.

Chaque solveur possède un catalogue de contraintes qu'il peut traiter, la plupart des solveurs traitent les contraintes les plus connues et les plus utilisées. Ils s'appuient sur des algorithmes de filtrage et de propagation et sur des stratégies de recherche et d'énumération pour résoudre les CSP. Tout solveur permet de choisir parmi un certain nombre d'algorithmes et de stratégies.

Ainsi, les différences majeures entre les solveurs résident dans leur catalogues de contraintes, et de stratégies de recherche et d'énumération ou encore au niveau de la représentation des domaines en mémoire. En pratique, il arrive aussi que le choix d'un solveur ou d'un autre soit motivé par l'utilisation d'un format de fichiers d'instances CSP donné (par exemple, le format xcsp [66]).

Parmi les solveurs les plus connus : Abscon [72] qui est écrit en Java et supporte le format d'entrée xcsp ; choco [25] qui est écrit en Java et dispose d'une API très complète ; Eclipse [2] et JaCop [64] qui sont basés sur la programmation logique(Constraint Logic Programming) ; et sugar [100] qui est basé sur SAT. La table 2.4 donne quelques informations sur ces différents solveurs.

Solveur	**lien**	**langage de prog.**	**commentaire**
Abscon [72]		java	Support du format xcsp
Choco [25]		java	API très complète
ECL^{i}PS^e [2]		C et Prolog	Basé sur la programmation logique (CLP)
JaCop [64]		java	Basé sur la programmation logique (CLP)
Sugar [100]		java et perl	Basé sur SAT
Chip [96]		C++ et Prolog	Basé sur Prolog
Ilog [85]		C++	Basé sur CLP

Table 2.4 – Informations pratiques sur quelques solveurs connus.

2.4.3 Bonnes pratiques de modélisation

La programmation par contraintes est un paradigme de haut niveau permettant d'exprimer de manière assez simple des problèmes complexes (par exemple, ayant une combinatoire très forte) et de les résoudre relativement efficacement. De plus, il existe une séparation entre la modélisation et le processus résolution (solveurs faciles à utiliser sans connaissance en résolution des CSP).

Un aspect non négligeable des CSP est le niveau requis pour le programmeur voulant réaliser une application en programmation par contraintes. En effet, il faut porter un soin particulier à l'étape de modélisation de laquelle dépend fortement la viabilité de l'application. Ainsi, d'une modélisation à un autre, un facteur 100 voire 1000 peut être observé. Ceci est dû à la forte combinatoire et complexité des problèmes rencontrés en contraintes. Saisir les bonnes propriétés du problème à résoudre et les spécifier de la meilleure des manières aidera le solveur à tester moins de possibilités et à faire moins de choix et de ce fait à améliorer ses performances. Par ailleurs, la force des contraintes réside dans le fait d'être un langage proche du problème ; ce qui aide à en capturer les propriétés et les

caractéristiques plus facilement.

Il existe un certain nombre d'astuces et de conseils à suivre pour obtenir un CSP efficace. Dans ce qui suit, nous donnons une liste non exhaustive de directives à suivre lorsqu'on veut modéliser en CSP :

— **Réduire le nombre de variables :** toute introduction de variable implique des choix supplémentaires pour le système de résolution. Il est important que le nombre de variables soit le plus petit possible.

— **Optimiser leurs domaines :** plus les domaines augmentent en tailles, plus nombreux seront les choix et les possibilités. En CSP, il est conseillé d'introduire des domaines expressifs et optimaux et d'éviter les domaines par défaut.

— **Utiliser des contraintes efficaces :** contrairement aux variables et aux domaines ou moins on en définit, mieux c'est, pour ce qui des contraintes c'est beaucoup plus nuancé. Ainsi, introduire peu de contraintes impliquera peu de filtrage des domaines et plus d'énumérations, ce qui peut conduire à beaucoup de backtracks et potentiellement beaucoup d'échecs. Néanmoins, si les contraintes ne propagent pas bien, le même phénomène se produira. Il en sera de même si le nombre de contraintes est très grand. En résumé, le nombre de contraintes dépend fortement du problème et il est nécessaire d'en introduire juste ce qu'il faut.

— **Privilégier les contraintes globales :** le fait d'avoir des algorithmes dédiés pour chacune d'entre elles rend les contraintes globales très efficaces. Par ailleurs, elles facilitent beaucoup la modélisation et permettent d'obtenir des CSP compacts et facile à lire et corriger.

2.4.4 Exemple : Sudoku

Le sudoku est un jeu sous forme de grille (généralement 9×9) partiellement remplie. Le but est de compléter la grille de façon à obtenir une et une seule occurrence de chaque chiffre (1..9) sur toutes les lignes, colonnes et sous-boîtes (3×3).

Nous allons donner le modèle CSP qui permet de compléter une grille (9×9) pré-remplie (Figure 2.12). Dans [97] l'auteur présente une étude plus complète sur la modélisation et la résolution du sudoku en programmation par contraintes. Elle inclut notamment une généralisation aux grilles ($n^2 \times n^2$), avec $n > 3$.

Pour modéliser le sudoku de la figure 2.12 nous introduisons 81 **variables** (une variable par case du sudoku) : $x_{i,j}$, avec $0 \leq i \leq 8$ et $0 \leq j \leq 8$.

Les **domaines** des variables sont définis comme suit :

$$\mathcal{D}(x_{i,j}) = \begin{cases} \{v\}, & \text{si la case vaut } v. \\ \{1, \ldots, 9\}, & \text{si la case est vide.} \end{cases}$$

CSP ▷

Ainsi, le domaine de la case située à l'intersection de la première ligne et de la deuxième colonne est donné par : $\mathcal{D}(x_{1,6}) = \{6\}$.

Des contraintes globales `alldiff` sont également nécessaires, une contrainte pour chaque ligne, colonne et boite :

`pour` $i = 0 \ldots 8$
  `alldiff`($x_{i,0}, x_{i,1}, \ldots, x_{i,8}$) ◁ CSP

`pour` $i = 0 \ldots 8$
  `alldiff`($x_{0,j}, x_{1,j}, \ldots, x_{8,j}$) ◁ CSP

`pour` $i = 0 \ldots 8$, `pas`$= 3$
 `pour` $j = 0 \ldots 8$, `pas`$= 3$
  `alldiff`($x_{i,j}, \ldots, x_{i+2,j+2}, x_{i,j-1}, \ldots, x_{i+2,j+1}, x_{i-1,j}, \ldots, x_{i+1,j+2}$) ◁ CSP

Le CSP qui vient d'être construit est illustré par la figure 2.12.

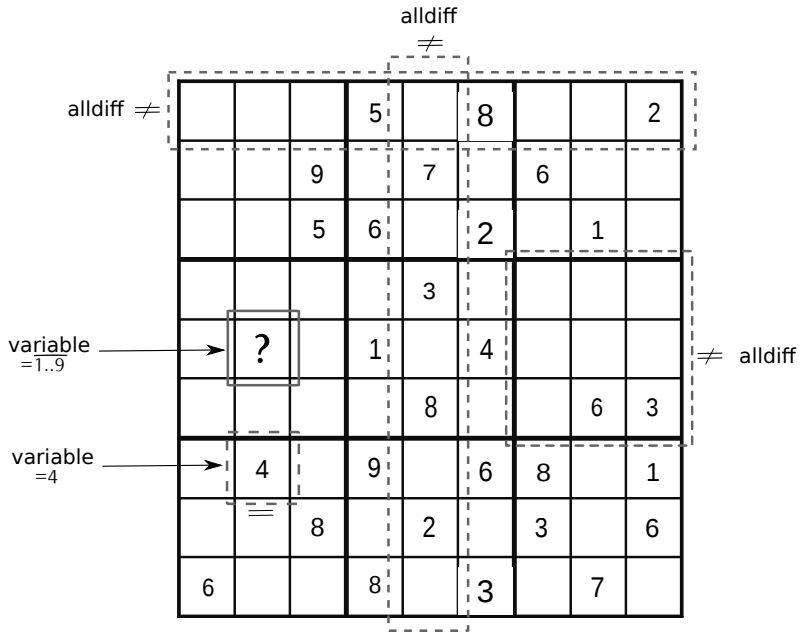


FIGURE 2.12 – Modélisation d'une grille de sudoku en CSP.

2.4.5 Pourquoi une approche de génération de modèles basée sur la programmation par contraintes ?

La caractéristique principale de la programmation par contraintes réside dans le fait que le programmeur décrit son problème et non pas les algorithmes qui vont le résoudre et trouver une solution. Ce sont des solveurs écrits par des experts en CSP qui sont utilisés pour résoudre les problèmes décrits en contraintes. Les solveurs effectuent une recherche

exhaustive mais néanmoins hautement optimisée dans le but de trouver une solution (voir plusieurs solutions). Ainsi, comme nous l'avons vu plus haut avec l'exemple du sudoku, seules les données et les règles élémentaires du jeu sont écrites en CSP ; la solution quant à elle est du ressort des algorithmes de résolution du solveur. Le programmeur n'a pas à s'en soucier. Par conséquent, la possibilité d'encoder un méta-modèle en CSP et de laisser le solveur trouver des solutions permet d'automatiser la tâche d'instanciation des méta-modèles. Plus haut dans le manuscrit (section 2.4.3), nous disions que l'efficacité d'une solution basée sur la programmation par contraintes dépend fortement de l'étape de modélisation en contraintes. Donc, travailler sur un CSP le plus optimisé possible permettra d'obtenir une approche qui passe à l'échelle.

Par ailleurs, une contrainte globale permet de capturer en une seule contrainte, facile à écrire une structure complexe venant du problème à résoudre. Il se trouve qu'il existe justement une correspondance entre le langage OCL et le catalogue de contraintes globales.

Enfin, la programmation par contraintes offre de la souplesse et un important choix de paramétrisation. Il est, en effet, possible d'ajouter autant de variables et de contraintes que l'on veut. Ces contraintes peuvent être notamment liées à des caractéristiques améliorant la pertinence ou la diversité des modèles générées automatiquement.

En résumé, nous avons choisi la programmation par contraintes car elle possède les atouts pour permettre la couverture des cinq critères de qualité d'une approche de génération de modèles, que ce soit en terme d'automatisation, de passage à l'échelle, la prise en compte de l'OCL, ou encore la pertinence et la diversité des modèles.

2.5 Conclusion

Dans le chapitre qui s'achève, nous avons donné un historique des langages de modélisation logiciel qui ont le plus marqué l'histoire de la modélisation en informatique. Nous nous sommes, ensuite, intéressés à l'ingénierie dirigée par les modèles (IDM). Ainsi, ses concepts les plus importants ont été définis.

Dans la deuxième section du chapitre, nous avons étudié et discuté les approches de génération de modèles que nous avons également comparé selon différents critères. La même étude comparative a été effectuée pour les outils d'aide à la méta-modélisation et des outils de création de syntaxe concrète.

La dernière section du chapitre est consacrée à la programmation par contraintes. Ses principaux concepts ont été définis. Nous avons donné les pratiques à suivre pour une bonne modélisation en CSP et motivé notre choix d'une approche de génération de modèles basée sur les contraintes.

Chapitre 3

Instanciation de Méta-modèles

> Une théorie ne ressemble pas plus à un fait qu'une photographie ne ressemble à son modèle.
>
> Edgar Watson Howe

Synopsis

Préambule

Dans ce chapitre nous allons présenter notre approche de génération de modèles basée sur la programmation par contraintes. La section 3.1 donne l'encodage d'un méta-modèle conforme à Ecore en CSP. Nous nous intéressons à la validité des modèles générés, et pour cela nous transformons également les contraintes OCL en section 3.2. La dernière section s'intéresse aux différentes expérimentations menées pour évaluer l'approche.

L'approche que nous proposons génère des modèles conformes à des méta-modèles au format Ecore et respectant leurs contraintes OCL. Le processus de génération est basé sur les CSP. En effet, le méta-modèle et les contraintes sont modélisés puis encodés en CSP. La solution du programme contraint va ensuite être utilisée pour construire une instance conforme au méta-modèle de départ. Le fonctionnement de l'approche suit le schéma général de la figure 3.1.

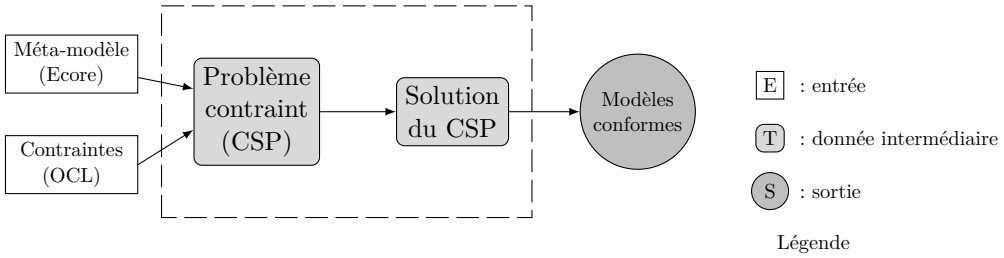


FIGURE 3.1 – Schéma général de notre approche de génération de modèles.

3.1 Modélisation d'un Méta-modèle en CSP

3.1.1 Quelles parties du méta-modèle prendre en compte

Les méta-modèles que nous encodons en CSP et pour lesquels nous voulons générer des instances sont conformes au méta-métamodèle Ecore (*voir glossaire*). Cependant seulement la partie statique des méta-modèles est prise en compte. Par conséquent, les opérations ne sont pas traitées par notre approche.

Par ailleurs, les méta-modèles que nous transformons doivent remplir les conditions énoncées ci-dessous :

— Être syntaxiquement correct.

— Contenir un seul package qui englobe toutes les méta-classes du méta-modèle.

— Disposer d'une classe dite classe racine (root class) qui possède la caractéristique d'avoir un chemin de composition vers toutes les autres classes, donc toutes les classes du méta-modèle doivent appartenir à la fermeture transitive de la relation de composition.

La Figure 3.2 montre un méta-modèle type contenant tous les éléments que notre encodage en CSP est capable de traiter. Le méta-modèle se compose de classes concrètes qui héritent des caractéristiques de leurs super-classes et d'un certain nombre d'attributs pour chaque classe. Il comporte également trois types de liens entre classes :

— référence unidirectionnelle : deux classes liées dans un seul sens (A → B).

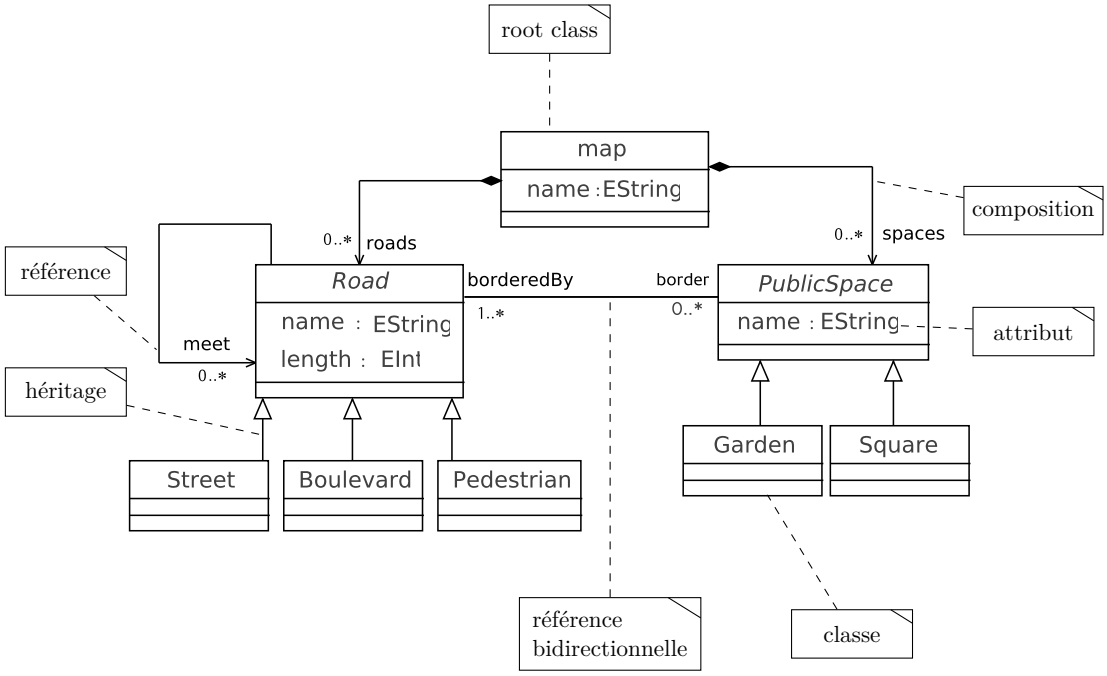


FIGURE 3.2 – Exemple type d'un méta-modèle conforme à Ecore et ses éléments à encoder en CSP.

— relation de composition : deux classes sont liées dans un seul sens, et où les instances de la première contiennent celles de la deuxième (A ◆→ B).

— références bidirectionnelles (ou Eopposite) : Quand deux références unidirectionnelles relient deux classes liées dans les deux sens (A → B et B → A), elles peuvent être regroupées pour former des références bidirectionnelles (A — B). On dit alors que l'une est l'opposée de l'autre et *vice-versa*.

Note

*D*ans la suite de ce manuscrit, les méta-classes d'un méta-modèle seront plus souvent désignées simplement par "classe", tandis que les classes du modèle seront désignées par "instance de classes". Quant au mot "référence", il désignera une référence unidirectionnelle.

Définition formelle d'un méta-modèle

Nous allons maintenant donner quelques définitions et notations qui seront utilisées par la suite pour décrire la modélisation en CSP des éléments d'un méta-modèle.

Un méta-modèle $\mathcal{M}$ est défini par un triplet ($\mathcal{C}l$,$\mathcal{F}$,$\mathcal{R}$). $\mathcal{C}l$=$\{C_1, C_2, \ldots, C_n\}$ est un ensemble de n Classes, $\mathcal{F}$=$\{a_1, a_2, \ldots, a_m\}$ est un ensemble de m attributs et $\mathcal{R}$=$\{a_1, a_2, \ldots, a_k\}$ est un ensemble de k références.

La table 3.1 donne quelques fonctions écrites par nos soins et utilisées pour récupérer certaines propriétés d'un méta-modèle. Elles seront utilisées pour expliquer l'encodage des éléments du méta-modèle en CSP.

fonction	**spécification informelle**
abstract(Class C) :boolean	indique si une classe est abstraite
superClasses(Class C) :Class[ ]	donne toutes les super-classes d'une classe
attributs(Class C) :Attribute[ ]	donne les attributs d'une classe (introduits et hérités)
references(Class C) :Reference[ ]	donne les références d'une classe
size(Class C) :int	indique le nombre d'instances d'une classe
type(Attribute a) :type	donne le type d'un attribut
type(Reference r) :Class	donne la classe destination d'une référence
upper(Reference r) :int	donne la cardinalité max d'une référence
lower(Reference r) :int	donne la cardinalité min d'une référence
opposite(Reference r) :Reference	retourne la référence opposée d'une référence

Table 3.1 – Quelques fonctions sur les éléments d'un méta-modèle Ecore.

3.1.2 Encodage des classes et attributs

Les prochaines sous-sections décriront l'encodage en CSP de chacun des éléments présentés ci-dessus. Pour chaque élément nous donnerons les variables, les domaines et les contraintes nécessaires à sa traduction en CSP. Le premier objectif de l'approche est d'être efficace pour pouvoir passer à l'échelle et générer des modèles de grande taille conformes à des méta-modèles de grande taille. Par conséquent, la modélisation en CSP qui est décrite ci-après est optimisée pour répondre à l'exigence d'efficacité de l'approche.

Classes concrètes

Une classe C possède dans le modèle généré automatiquement un nombre d'instances donné. Ce nombre est noté $size(C)$ et il est généralement choisi par l'utilisateur pour chaque classe. Par conséquent, ce choix est constant pour la classe et ne doit pas varier après la résolution. C'est pour cela que nous ne créons pas de variables pour modéliser les instances des classes, mais uniquement des domaines.

En effet, un intervalle de valeurs noté D_C est assigné à toute classe C suivant son nombre d'instances. Et pour éviter toute ambiguïté, nous affectons un identifiant unique pour toute instance de classe. Ainsi, les domaines seront contiguës pour ne pas partager de valeur commune. Le domaine D_{C_i} d'une classe C_i est calculé comme suit :

$D_{C_i} = \{M_{i-1} + 1, \ldots, M_i\}$, où $M_i = \sum_{j=1}^{i} size(C_j)$. ◁ CSP

Note

*L*a classe racine d'un méta-modèle possède une et une seule instance qui contiendra transitivement toutes les autres instances de classe du modèle généré.

Nous illustrons la modélisation des classes sur le méta-modèle de la figure 3.2. Il faut d'abord choisir un nombre d'instances pour chaque classe, par exemple : size(map)= 1, size(Street)= 3, size(Boulevard)= 5, etc. Ensuite, les domaines des classes pourront être déduits :

$D_{map} = \{1\}, D_{Street} = \{2, 3, 4\}, D_{Boulevard} = \{5, 6, 7, 8, 9\}$. ◁ CSP

Quand une valeur v (donc une instance de classe) est piochée aléatoirement, par exemple 8, il est très aisé de déduire à quelle classe elle correspond. En effet, il suffit de comparer v aux valeurs M_i qui ont permis de calculer les domaines :

$$4 = M_{Street} < 8 \leq M_{Boulevard} = 9 \Rightarrow 8 \text{ est une instance de Boulevard.}$$

Cette caractéristique est très utile pour construire le modèle après la résolution du CSP car elle permet d'identifier très facilement quelle classe et donc quel attribut et quelle référence instancier. De plus, elle sera également utile pour la modélisation des références entre classes.

Attributs

Une classe d'un méta-modèle peut avoir des attributs introduits ou hérités. Un attribut est modélisé en CSP par une variable. Par conséquent, une variable est créée pour toute instance de classe et pour tout attribut de la classe :

$\forall C \in \mathcal{C}l, \forall a \in attribut(C), \forall i \in D_C :$ déclarer une variable $F_{C,i,a}$. ◁ CSP

La classe `map` du méta-modèle 3.2 possède un attribut et une seule instance, donc une seule variable $F_{map,1,name}$ sera créée.

Les domaines des variables modélisant un attribut a dépendent de type(a). Néanmoins, les CSP ne prennent en compte que des domaines d'entiers. Pour modéliser les autres types de données, il faut trouver des correspondances entre le type d'origine et les entiers. Par

exemple, nous utilisons le codage ASCII pour les caractères, et nous générons des valeurs aléatoires pour les attributs de type chaîne de caractères.

Note

*I*l existe des solveurs CSP qui acceptent des variables de domaines réels comme par exemple CHOCO, ou encore des solveurs à contraintes continues tel IBEX [1]. Néanmoins, ce dernier est moins efficace sur les autres types de contraintes à valeurs entières.

Traitement de l'héritage

Lorsqu'une classe possède des super-classes, elle devra hériter de toutes leurs caractéristiques. Dans notre approche, les attributs, mais aussi les références des super-classes sont recopiés au moment de modéliser les sous-classes. Ainsi, au moment d'encoder en CSP les attributs et les références d'une classe, nous collectons ses attributs et ses références et ceux de toutes ses super-classes.

3.1.3 Encodage des références

La figure 3.3 contient un extrait de méta-modèle qui possède une référence d'une classe vers une autre. Le modèle conforme à droite montre que des instances de A sont liées à au moins deux instances de B et au plus trois. En CSP, tout lien probable sera représenté par une variable. Ainsi, pour chaque instance de A, upper$(r) = 3$ variables seront créées. Le domaine de ces variables est égal à D_B, le domaine de la classe B. De cette façon, toute valeur choisie dans ce domaine sera obligatoirement une instance de B.

Voici les variables CSP créées pour modéliser les références d'une classe A :

$\forall i \in D_A, \forall r \in \mathit{references}(A), \forall j \in \{1, \ldots, \mathit{upper}(r)\}$:

CSP ▷ déclarer j variables notées $Ref_{i,j}^{A,r}$.

Parmi les variables précédentes, nous distinguons deux types : les variables qui sont obligatoirement instanciées et celles qui le sont de manière optionnelle. Seules les $\mathit{lower}(r)$ premières variables sont obligatoires (2 pour l'exemple précédent), les autres peuvent être instanciées ou non. Le domaine des variables de référence est donc donné par :

CSP ▷
$$D(Ref_{i,j}^{A,r}) = \begin{cases} D_B, \text{ si } j \leq \mathit{lower}(r), \\ D_B \cup \mathit{jokers}, \text{ sinon.} \end{cases}$$

jokers est un ensemble de valeurs refuge qui ne sont instance d'aucune classe du méta-modèle. Elles servent à dire qu'une variable de référence ne pointe vers aucune instance de classe connue car elle est optionnelle.

1. Solveur IBEX: `http://www.ibex-lib.org`

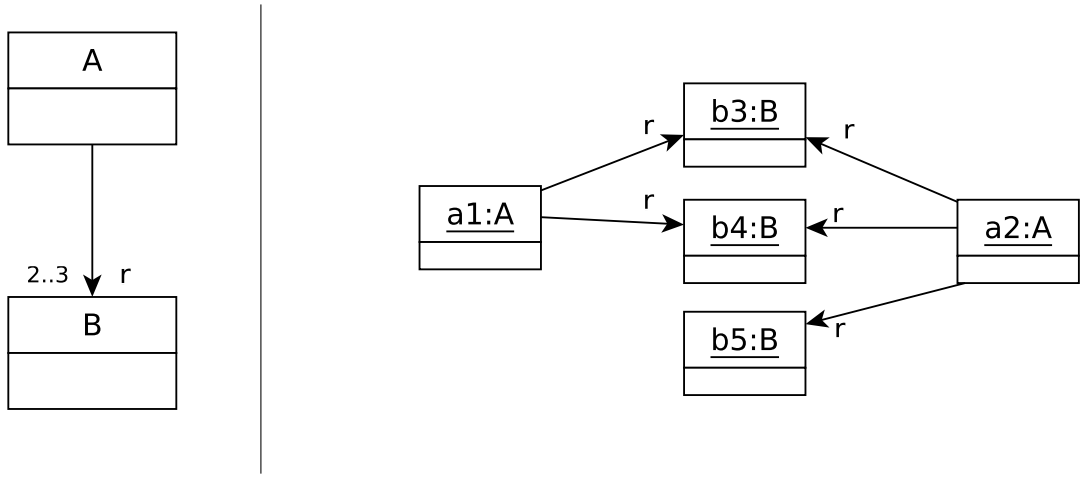


FIGURE 3.3 – Exemple d'une référence simple entre deux classes. À gauche : un extrait de méta-modèle. À droite : un extrait de modèle conforme.

La figure 3.4 montre l'instanciation des variables de références qui a permis de donner le modèle de la figure 3.3. La valeur de la variable $Ref_{i,j}^{A,r}$ indique l'instance de B qui sera pointée par la $i^{ème}$ instance de A. Par exemple, $(Ref_{2,1}^{A,r} = 3) \Rightarrow$ a2:A → b3:B . Les pointillés indiquent une variable optionnelle qui prend une valeur joker pour la première instance de la classe A.

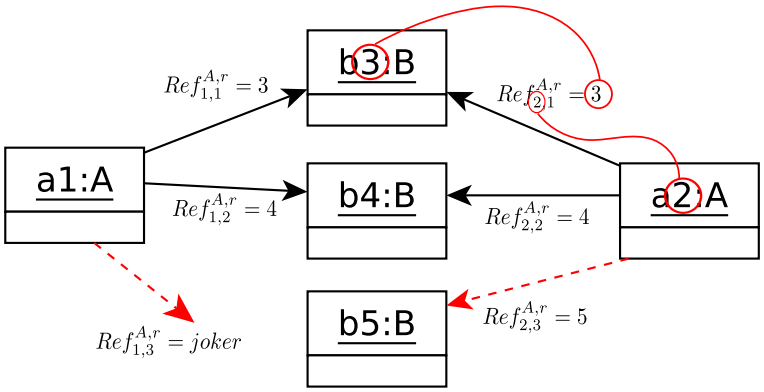


FIGURE 3.4 – Schéma montrant une instanciation des variables d'une référence. Les flèches rouges pointillées indiquent des variables optionnelles.

Note

*L*es compositions sont traitées de la même manière que pour les références. Les mêmes variables et domaines sont créés.

Références et héritage

Il arrive parfois que la classe destination d'une référence (son type) possède elle-même des sous-classes comme le montre la figure 3.5. Dans un tel cas de figure, une instance de la classe A peut référencer des instances de B ou des instances de chacune des sous-classes de B. Par conséquent, le domaine des variables qui vont encoder la référence doit inclure les domaines de toutes les classes de la hiérarchie d'héritage de B. Ainsi, les nouveaux domaines seront définis comme suit :

CSP ▷

$$D(Ref_{i,j}^{A,r}) = \begin{cases} D_{B_1} \cup D_{B_2} \cup \ldots \cup D_{B_n}, & \text{si } abstract(B), \\ D_B \cup D_{B_1} \cup D_{B_2} \cup \ldots \cup D_{B_n}, & \text{sinon.} \end{cases}$$

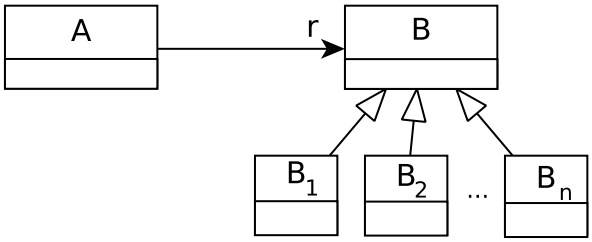


FIGURE 3.5 – Une hiérarchie d'héritage pour la classe destination d'une référence.

Références bidirectionnelles

Comme nous l'avons vu précédemment, la présence de références bidirectionnelles entre deux classes indique l'existence de deux références reliant les deux classes dans les deux sens. Chacune des références est dite l'opposée de l'autre. Pour qu'un modèle soit conforme à une telle configuration, il faut que les cardinalités des deux références soient respectées en même temps. Par exemple, la figure 3.6 montre un extrait de méta-modèle où deux classes sont reliées par des références bidirectionnelles et deux modèles. Le premier modèle est conforme car les cardinalités des deux références sont respectées. Tandis que pour le deuxième, l'instance numéro 4 de B est reliée à deux instances de A, violant ainsi les cardinalités de la référence f.

Si on applique l'encodage des références en CSP comme décrit plus haut, il faudra créer des variables CSP pour les deux références, dans les deux sens : $Ref_{i,j}^{A,r}$ et $Ref_{i,j}^{B,f}$. Cependant, nous constatons deux principaux problèmes liés à cette solution :

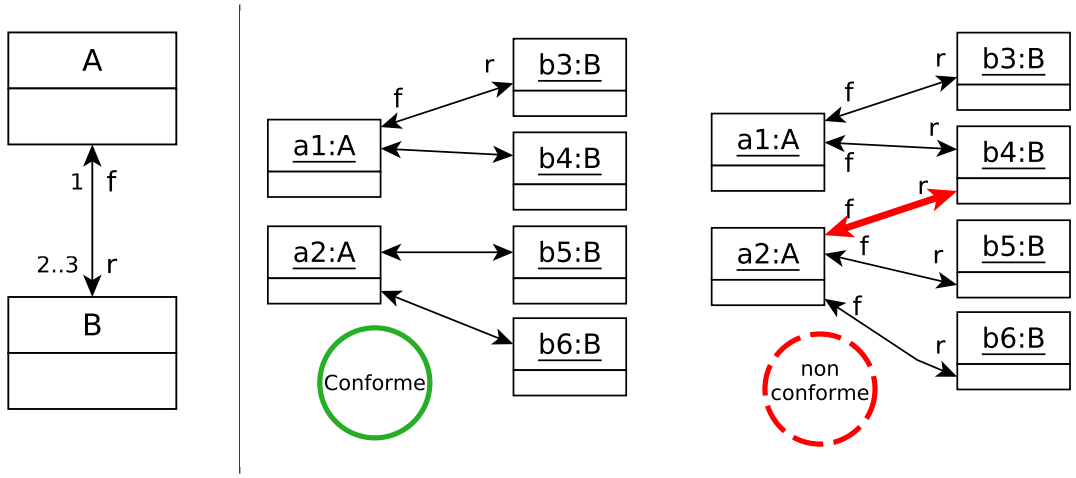


FIGURE 3.6 – Exemple d'une référence bidirectionnelle entre deux classes suivi de deux extraits de modèle, l'un est conforme à l'extrait de méta-modèle et l'autre est non conforme.

1. Il n'y a aucune garantie sur la conformité des modèles, car il n'y a aucune contrainte sur la non violation des cardinalités dans les deux sens en même temps.
2. Dans un modèle conforme à un méta-modèle Ecore, lorsqu'une référence est instanciée, son opposée est instanciée automatiquement. En d'autres termes, il n'est pas nécessaire d'avoir des variables CSP dans les deux sens. Un seul suffira.

Nous proposons donc une solution où il suffira de créer des variables dans un seul sens et une contrainte qui forcera la non violation des cardinalités dans l'autre sens. Pour créer le moins possible de variables, nous choisissons la référence avec la plus petite cardinalité maximale pour la création des variables et l'autre référence pour la contrainte.

Soient r une référence qui relie deux classes dans le sens A vers B et f une référence dans l'autre sens. Nous supposons que $upper(f) \leq upper(r)$. Des variables CSP sont créées pour modéliser la référence f. La contrainte globale suivante est ajoutée pour respecter les cardinalités de la référence r :

$$\texttt{gcc}(\ Ref_{i,j}^{B,f}, D_A, lower(r), upper(r)\ ) \qquad \triangleleft \text{ CSP}$$

Cette contrainte stipule que toutes les instances de A sont liées à au moins $lower(r)$ et au plus $upper(r)$ instances de B .

3.2 Formalisation des contraintes OCL en CSP

Cette section est consacrée à l'encodage des contraintes OCL d'un méta-modèle en CSP. Chaque construction importante en OCL est d'abord formalisée en CSP pour être prise en

compte. Nous donnerons donc les variables et contraintes CSP nécessaires à la traduction des contraintes OCL.

L'autre solution envisagée pour prendre en compte les contraintes OCL d'un méta-modèle est de générer et tester (*generate and test*). Cette solution consiste à générer des modèles conformes seulement à la structure du méta-modèle. Ensuite, un vérificateur de contraintes OCL se chargera de sélectionner uniquement les modèles qui respectent les contraintes OCL en plus de la structure. Cette solution présente un avantage important mais surtout un gros inconvénient :

— **Avantage** elle est facile et rapide à mettre en place car des vérificateurs de contraintes OCL existent (par exemple, Dresden OCL, Eclipse OCL). De plus, il n'y a pas de nécessité de formalisation des contraintes dans un autre langage tel que CSP.

— **Inconvénient** Il existe une probabilité faible, voire nulle, pour générer un modèle qui respecte toutes les contraintes OCL de son méta-modèle. Notamment quand le méta-modèle contient beaucoup de contraintes et que celles-ci sont complexes. Ainsi, nous avons mené une expérience au cours de laquelle nous générons des squelettes de programmes java qui doivent respecter 10 contraintes OCL pour pouvoir compiler correctement. Sur les 700 programmes générés sans prise en compte des contraintes OCL, il y avait 0% de programmes qui compilent alors que le taux est proche de 98% quand les contraintes sont ajoutées au CSP.

Finalement notre choix s'est porté sur l'approche qui transforme chaque contrainte en CSP. L'objectif est de maintenir l'efficacité en temps de résolution du CSP produit, même après l'ajout des contraintes OCL. C'est pour cela que nous avons pris soin d'optimiser l'encodage d'OCL pour qu'il soit le plus efficace à intégrer au CSP et surtout à résoudre. Un traitement spécifique est appliqué à chaque construction du langage OCL. Nous avons privilégié cette solution car nous remarquons une grande disparité dans les constructions du langage et car les CSP offrent beaucoup de possibilités d'optimisation. Par ailleurs, comme nous l'avons vu précédemment, programmer efficacement en CSP nécessite un grand soin lors de l'étape de modélisation.

voir sec. 2.4.3 ▷

3.2.1 Contraintes OCL sur les attributs

Cas d'un seul attribut

Les contraintes les plus simples en OCL portent sur un attribut d'une seule classe. Ces contraintes appliquent une expression booléenne sur l'attribut en question.

La contrainte suivante porte sur l'attribut a de la classe C de la figure 3.7. Elle applique l'expression booléenne *expr* sur a et stipule que pour toutes les instances de C, la valeur de a doit respecter l'expression.

OCL ▷ `Context` C `inv` : $expr(a)$

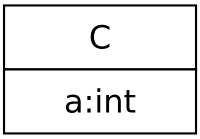


FIGURE 3.7 – Une classe d'un méta-modèle possédant un attribut.

Pour encoder une telle contrainte en CSP, nous modifions *a priori* le domaine des variables représentant a en appliquant l'expression à l'ancien domaine dans le but de déduire un nouveau domaine.

$D(F_{C,i,a}) = \{e \in type(a) |\ expr(e)\}$. ◁ CSP

De cette façon le domaine des variables représentant a est modifié ce qui implique que toutes les instances de C prendront obligatoirement des valeurs valides pour l'attribut a.

Note

expr est une expression logique incluant les opérateurs logiques et arithmétiques.

Cas de plusieurs attributs

Des contraintes OCL portent sur plusieurs attributs en même temps. Ces derniers peuvent appartenir à une seule classe seulement ou bien provenir de plusieurs classes. Nous donnons ici le cas d'une même classe. Quand les attributs appartiennent à des classes distinctes, le traitement n'est pas différent mais nécessite également une navigation de références (voir sec. 3.2.2).

La contrainte suivante porte sur les attributs a et b de la classe C en figure 3.8. Elle applique une expression booléenne sur les deux attributs en même temps.

`Context` C `inv` : $expr(a, b)$. ◁ OCL

La solution de modifier *a priori* les domaines des variables ne fonctionne pas dans tous les cas de figure. En effet, quand il y a interdépendance entre les variables, modifier les domaines peut s'avérer infaisable ou même dangereux. Ainsi, dans le cas de deux attributs appartenant à deux classes différentes, si on modifie les domaines, le risque est d'empêcher les attributs de prendre certaines valeurs lorsque leur instance de classe n'est pas liée par référence à une autre instance.

La solution choisie consiste donc à ajouter des contraintes (à deux variables dans cet exemple) dont la fonction est l'expression booléenne entre les attributs concernés par la

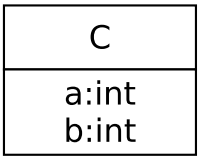


FIGURE 3.8 – Une classe d'un méta-modèle possédant deux attributs.

contrainte OCL. Pour toute instance i de C, nous introduisons une contrainte $Cons_i$, telle que :

CSP ▷ $Cons_i : expr(F_{C,i,a}, F_{C,i,b})$

3.2.2 Navigation de références

La navigation est l'une des constructions les plus importantes du langage OCL. Elle permet de parcourir le méta-modèle en se déplaçant d'une classe à une autre en empruntant les chemins de référencement. De cette façon, à partir des instances d'une classe A donnée, une contrainte peut par exemple être vérifiée sur les attributs des instances de la classe B que A référence.

Traiter la construction de navigation de références en CSP est une condition *sine qua non* pour écrire des contraintes plus complexes reliant plusieurs classes d'un méta-modèle.

Navigation d'une branche de n références

La contrainte suivante navigue n références consécutives à partir de la classe C_0 de la figure 3.9, puis applique une expression sur l'attribut a de la dernière classe :

OCL ▷ `Context` C_0 `inv` : $r_1.r_2. \ldots .r_n.expr(a)$

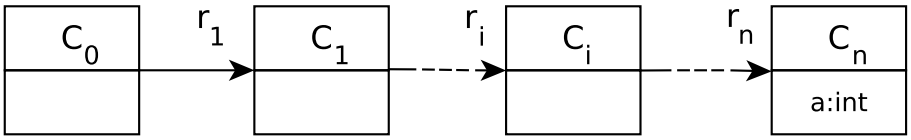


FIGURE 3.9 – Extrait d'un méta-modèle de $n + 1$ classes reliées par n références.

En CSP, appliquer une telle contrainte implique de sélectionner un sous-ensemble parmi les variables $F_{C_n,i,a}$ modélisant l'attribut a et de vérifier l'expression $expr$. Ce sous-ensemble doit comprendre toutes les variables d'attributs des instances de la classe C_n pointées par

des instances de C_{n-1}, et ainsi de suite jusqu'aux instances pointées par C_0. Il est donc impératif que la chaîne de référencement soit totalement respectée pour qu'à la fin la variable d'attribut soit concernée par l'application de l'expression.

L'encodage des contraintes de navigation en OCL est facilitée par l'encodage des références comme des variables pointeurs d'instances de classes vers d'autres instances. En effet, pour vérifier qu'une instance numéro i pointe vers une instance j, il suffit de vérifier l'égalité $Ref_{i,k}^{C,r} = j$ pour tout $k \in [lower(r), upper(r)]$.

Des contraintes CSP sous forme de conjonctions d'égalité sont créées pour encoder la navigation de références en OCL.

$\wedge_{m_i=1}^{Upper(r_i)}(Ref_{k_{i-1},m_i}^{C_{i-1},r_i} = k_i) \Rightarrow cons(F_{C_n,k_n,a}), \forall k_i \in D_{C_i}$, où $i \in \{1, \ldots, n\}$. ◁ CSP

Note

*D*ans notre cas, l'implication $a \Rightarrow b$ est transformée en $\neg a \vee b$ avant d'être encodée dans le CSP.

Exemple 1

Nous allons maintenant expliquer le déroulement du traitement d'une contrainte de navigation sur un exemple concret. La contrainte qui suit navigue une référence (figure 3.10) et applique une expression sur un attribut entier :

`Context` A `inv` : $r.a > 10$ ◁ OCL

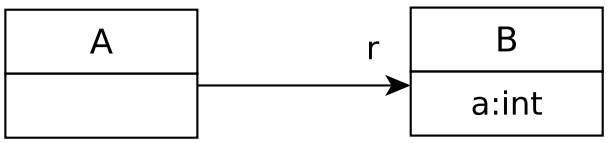


FIGURE 3.10 – Extrait d'un méta-modèle contenant 2 classes et une référence.

La contrainte stipule que pour toutes les instances de B pointées par A, la valeur de a doit être supérieure à 10. La figure 3.11 monte une instanciation possible du méta-modèle précédent après l'application de la contrainte OCL. On voit que l'instance $b5$ n'est pas concernée par la contrainte car elle n'est pas référencée par une instance de A. A contrario, les deux autres instances de B ont leurs attributs a contraints à prendre des valeurs supérieures à 10. La vérification de la navigation de la référence se fait s'il y a égalité entre les valeurs des variables de référence et les numéros des instances de classes (liens rouges sur la figure).

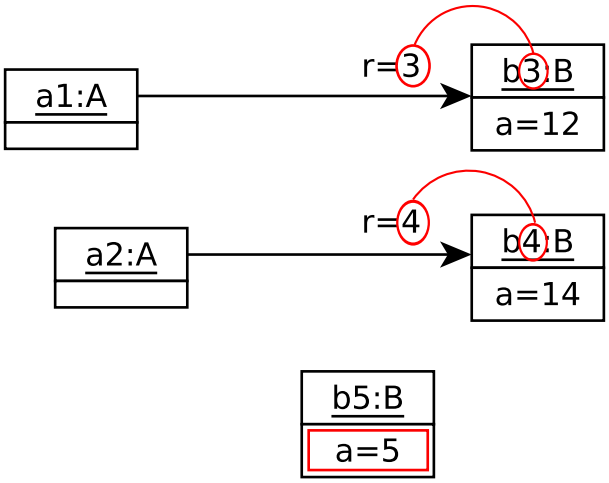


FIGURE 3.11 – Extrait d'un modèle conforme au méta-modèle figure 3.10.

Note

*E*n dépit du nombre important de contraintes CSP créées pour l'encodage de la navigation en OCL (nombre d'instances $*$ cardinalités des références $*$ nombre de références à naviguer), la solution reste efficace car dans les faits la navigation dépasse rarement 2 références successives. Ceci a été démontré dans l'étude de Juan Cadavid sur OCL pour les méta-modèles Ecore durant sa thèse [20].

Navigation de deux branches de n références

Certaines contraintes OCL appliquent une double navigation de deux branches de références en partant d'une classe donnée, comme le montre la figure 3.12. Voici une contrainte OCL naviguant deux branches de références et appliquant une expression sur les attributs des deux dernières classes visitées :

OCL ▷ `Context` C_0 `inv` : $expr(r_1.r_2. \ \ldots \ .r_n.a, \ r'_1.r'_2. \ \ldots \ .r'_m.a')$

Pour encoder ce type de contraintes OCL en CSP, nous introduisons une version étendue des conjonctions d'égalité déjà décrites. À cet escient, une conjonction de conjonctions est introduite.

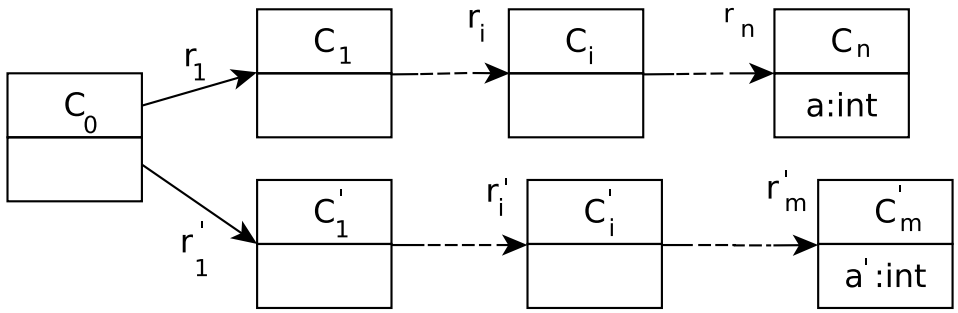


FIGURE 3.12 – Extrait d'un méta-modèle de $2 \times n + 1$ classes reliées par $2 \times n$ références.

$$\bigwedge_{m_i=1}^{Upper(r_i)} (Ref_{k_{i-1},m_i}^{C_{i-1},r_i} = k_i)$$

$$\wedge \bigwedge_{m'_j=1}^{Upper(r'_j)} (Ref_{k'_{j-1},m'_j}^{C'_{j-1},r'_j} = k'_j) \qquad \triangleleft \text{ CSP}$$

$$\Rightarrow \; cons(F_{C_n,k_n,a}, F_{C'_m,k'_m,a'}), \text{ où } i \in \{1,2,\ldots,n\} \text{ et } j \in \{1,2,\ldots,m\}$$

De cette façon, l'expression est appliquée uniquement aux couples d'attributs des instances de classes qui respectent les deux branches de référencement.

Note

*L'*encodage d'une expression dans une contrainte OCL contenant de la navigation est la même que celle des contraintes portant sur plusieurs attributs (sec. 3.2.1).

3.2.3 Opérations sur les collections

Une opération sur une collection s'applique sur un ensemble d'éléments du modèle. Dans notre modélisation en CSP, nous faisons en sorte d'identifier cet ensemble d'éléments (donc de variables CSP), puis d'introduire les bonnes contraintes CSP pour traiter l'opération. Pour plus d'efficacité nous privilégions les modélisations que ne créent par de contraintes, ou bien créent des contraintes globales.

Dans ce qui suit, nous décrivons avec précision la traduction en CSP de deux opérations OCL sur des collections, puis nous donnerons les contraintes CSP correspondant à plusieurs autres opérations.

forall

Nous considérons l'extrait de méta-modèle en figure 3.13 qui contient 2 classes reliées par une référence. La contrainte OCL suivante est définie dans le contexte de la classe C_0 :

OCL ▷ `Context` C_0 `inv` : $r_1 \rightarrow$`forall`$(c:\ C_1 \mid expr(c.a))$

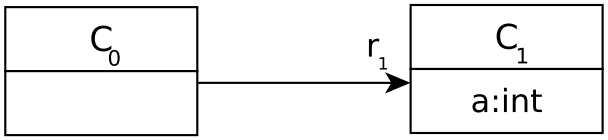


FIGURE 3.13 – Extrait d'un méta-modèle de 2 classes.

La contrainte précédente applique une expression sur les attributs des instances de la classe C_1 qui sont référencées par les instances de C_0. Son traitement se fait en deux étapes :

1. Navigation d'une seule référence.
2. Sélectionner les variables concernées et appliquer *expr*.

Ainsi, les contraintes CSP suivantes seront introduites :

CSP ▷ $\forall i \in D_{C_0}, j \in [1, Upper(r_1)], k \in D_{C_1}, (Ref_{i,j}^{C_0,r_1} = k) \wedge expr(F_{C_1,k,a})$

excludes et excludesAll

L'opération `excludes` (resp. `excludesAll`) permet de vérifier l'exclusion d'un objet (resp. d'une collection d'objets) d'une collection donnée.

Si une collection d'éléments du modèle exclut un objet ou une collection d'objets, cela implique que les domaines des variables représentant les éléments en question du modèle doivent également exclure les valeurs représentant ces mêmes objets. Ceci empêchera ces valeurs d'être assignées aux variables et donc d'apparaître dans les modèles générés. Cela implique que la contrainte OCL est respectée.

Une étape de pré-traitement est nécessaire pour la traduction de telles constructions du langage OCL en CSP. Durant ce pré-traitement, les domaines des variables concernées sont modifiés pour exclure les valeurs indésirables.

Exemple 2

Nous définissons la contrainte suivante contenant une opération `excludes` sur l'extrait de méta-modèle en figure 3.14. Elle stipule qu'une classe ne peut pas être sa propre sous-classe.

`Context` *Class* `inv` : `self`.subClasses→`excludes`(`self`). ◁ OCL

L'encodage de cette contrainte OCL se fait par pré-traitement du domaine de la référence subClasses pour chaque instance i et suppression de la valeur i de ce domaine.

$\forall i \in D_{Class}, D(subClasses_i) = D_{Class} \setminus \{i\}$. ◁ CSP

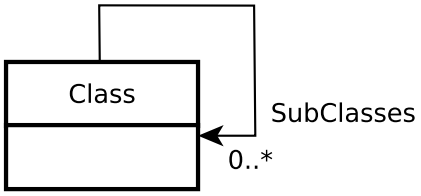


FIGURE 3.14 – Extrait de méta-modèle d'une classe.

D'autres opérations

D'autres opérations OCL sur des collections sont également traduisibles en CSP en utilisant notre approche et en exploitant la diversité des contraintes globales offertes par les CSP. En effet, nous avons remarqué beaucoup de correspondance entre les opérations sur les collections et les contraintes globales.

La table 3.2 donne quelques exemples de correspondances entre les opérations sur les collections en OCL et les contraintes globales.

Opération OCL	**Contrainte globale**
$\rightarrow sum() = res$	$sum(vars, res)$
$\rightarrow count(value) = res$	$count(value, vars, res)$
$\rightarrow size() = res$	$nvalue(vars, res)$
$\rightarrow exists(value)$	$atleast(1, vars, value)$
$\rightarrow includes(vals)$	$gcc(< vars >, < vals >, < 1 \ldots 1 >, < n \ldots n >)$
$\rightarrow isUnique(Value)$	$atleast(1, vars, value) \wedge atmost(1, vars, value)$

TABLE 3.2 – Quelques exemples de correspondance entre les opérations OCL et les contraintes globales.

Note

*P*our obtenir plus d'informations sur les contraintes globales citées dans la table 3.2, veuillez vous référer au catalogue de contraintes globales [2].

3.2.4 Opération de typage

Lorsqu'une hiérarchie d'héritage est présente dans un méta-modèle, des contraintes OCL peuvent porter sur les types des objets. Ceci peut avoir comme but de vérifier que les types des objets d'une collection sont tous différents les uns des autres.

Nous pouvons voir sur l'extrait de méta-modèle de la figure 3.15 une classe B reliée à une classe A_0 par n références différentes toutes de cardinalité 1. Les références peuvent pointer vers des objets de tous les types sous-classes de A_0. La contrainte OCL suivante veut s'assurer que les objets pointés sont de type tous différents les uns des autres :

OCL ▷ `Context` B `inv` : r_1`.oclType` <> r_2`.oclType` <> ... <> r_n`.oclType`

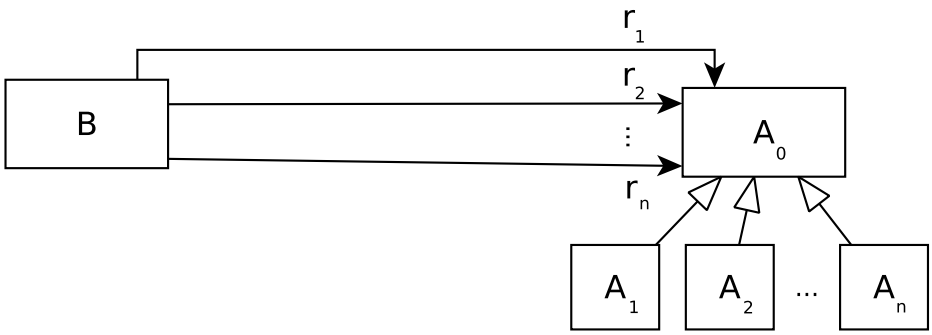


FIGURE 3.15 – Extrait d'un méta-modèle contenant une hiérarchie de n sous-classes et n références.

Quand un tel motif est détecté, la traduction suivante en CSP est opérée pour chaque instance $j \in D_B$ de B :

1. Introduire n variables temporaires U_i, où $i \in \{1, \ldots, n\}$ et $U_i \in D_{A_i}$. Une variable est créée pour chaque sous-classe A_i de A_0.
2. Introduire n variables V_i, où $i \in \{1, \ldots, n\}$ et $V_i \in \{1, \ldots, n\}$. Une variable est créée pour chaque référence r_i de B.
3. Déclarer n contraintes globales `element` : `element`$(Ref_{j,1}^{B,r_i}, [U_1, \ldots, U_n], V_i)$.
4. CSP ▷ Déclarer une contrainte globale `alldiff` : `alldiff`$(V_1, \ldots, V_n)$.

2. Catalogue de contraintes globales: `http://sofdem.github.io/gccat/`

La contrainte globale `alldiff` contraindra les variables V_i à prendre des valeurs différentes, ce qui par effet domino obligera les variables $Ref_{j,1}^{B,r_i}$ pour chaque référence à prendre des valeurs différentes parmi les valeurs des variables U_i. Vu que les variables U_i prennent leurs valeurs dans des ensembles disjoints représentant chacun une sous-classe A_i de A_0, alors les variables de références $Ref_{j,1}^{B,r_i}$ pointeront obligatoirement vers des instances de classes qui appartiennent à des types différents.

D'une façon plus générale, le type d'une instance de classe peut être déduit dans notre approche par les domaines D_C des classes du méta-modèle. En effet, vu que chaque instance possède un numéro unique, que chaque classe possède un domaine unique et que tous les domaines sont disjoints et contiguës, il est aisé de trouver le type d'une instance en vérifiant à quel domaine de classe appartient la valeur de cette instance. C'est exactement cette vérification que font les variables U_i introduites plus haut. Elles prennent leurs valeurs dans des ensembles disjoints, et, chacun d'entre eux représentant une classe donnée, ce qui permet d'obtenir des références pointant vers des instances dont les types sont tous différents les uns des autres.

3.3 Évaluations

La modélisation en CSP que nous venons de décrire est implémentée par un outil de génération de modèles nommé $\mathcal{G}$RIMM($\mathcal{G}$eneRating Instances of Meta-Models). Cet outil prend en paramètre des méta-modèles écrits en Ecore et leurs contraintes OCL et génère automatiquement des modèles conformes. Le chapitre 6 est consacré à sa description détaillée, à celles de ses options et différentes versions. La figure 3.16 montre les entrées/sorties et les principales étapes de la génération de modèles selon $\mathcal{G}$RIMM. Le processus est totalement automatisé : la génération de modèles conformes ne nécessite aucune intervention humaine. L'utilisateur a pour seul rôle de dimensionner les modèles désirés.

◁ *cf. chap. 6*

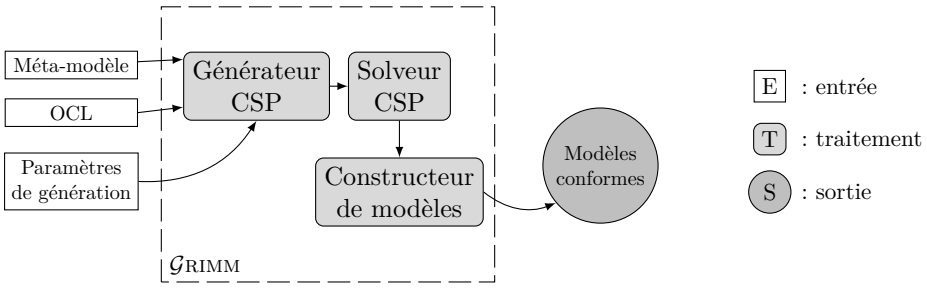


FIGURE 3.16 – Processus de l'outil $\mathcal{G}$RIMM.

Cette section rassemble trois expérimentations menées pour montrer le bon passage à l'échelle de l'approche et la prise en compte des contraintes OCL.

1. **Comparaison avec une approche de génération de modèles existante $\mathcal{G}$RIMM**

est comparé à un outil d'une approche de génération de modèles existante en ce qui concerne le temps de résolution.

2. **Test sur un benchmark de méta-modèles** Un ensemble de méta-modèles de tailles et d'origines diverses a été collecté. Des modèles sont ensuite générés et le temps de résolution est calculé pour chacun des méta-modèles.
3. **Influence de l'ajout de l'OCL sur l'efficacité** Des contraintes OCL sont ajoutées à certains méta-modèles. Le but est de montrer que la modélisation que nous proposons pour OCL est efficace et n'influe pas beaucoup sur le temps de résolution.

Le protocole expérimental détaillé est donné pour chacune des trois expérimentations et leurs résultats sont ensuite expliqués.

3.3.1 Comparaison avec une approche existante

Nous voulons dans cette section comparer notre approche de génération de modèles aux approches existantes en terme de passage à l'échelle. Malheureusement, il n'existe que trop peu d'outils disponibles et aucun n'est documenté. Par conséquent, l'utilisation des outils pour nous y comparer est impossible.

Nous nous sommes, ensuite, tournés du côté des publications pour exploiter les chiffres que les auteurs présentent dans leurs papiers. Là encore, très peu de données sont disponibles. En effet, seule l'approche de Cabot et al. a été évaluée du point de vue du passage à l'échelle. Nous décidons donc d'utiliser les données de [51] pour y comparer notre approche.

3.3.1.1 Protocole

Données dans [51], les auteurs s'intéressent au temps de génération de modèles pour le seul méta-modèle des entités relations (5 classes). Plus précisément, ils calculent le temps de résolution par le solveur de contraintes du CSP qui est produit à partir du méta-modèle. Le temps de résolution est donné en secondes.

Déroulement Nous avons évalué notre approche pour le même méta-modèle et en utilisant deux versions de notre approche :

1. La dernière version décrite plus haut dans ce chapitre et dans [35].
2. Une ancienne version [36] dans laquelle des variables CSP sont créées pour représenter les instances des classes et où le traitement des références bidirectionnelles n'est pas optimisé.

Pour chacune des deux versions de notre approche, nous générons des modèles conformes de même tailles que ceux présentés par l'approche existante. Le temps CPU est ensuite comparé sur des machines équivalentes et sur la même version de java.

Enfin, les courbes du temps de résolution en fonction du nombre d'instances par classe du méta-modèle sont tracées.

3.3.1.2 Résultats

Les courbes de la figure 3.17 montrent une comparaison entre les deux versions de notre outil et l'approche de Cabot et al. On y observe que notre approche de génération de modèles est plus rapide que l'approche existante pour ce méta-modèle des entités relations. Nous observons également que notre dernière version de la modélisation en CSP est nettement plus efficace que l'ancienne version.

Pour les valeurs les plus élevées, nous pouvons voir qu'il faut environ 20 secondes à l'approche de Cabot et al. pour générer un modèle contenant 60 instances de classe, tandis qu'il faut seulement 4 secondes (resp. 1.7 secondes) pour notre première (resp. deuxième) approche.

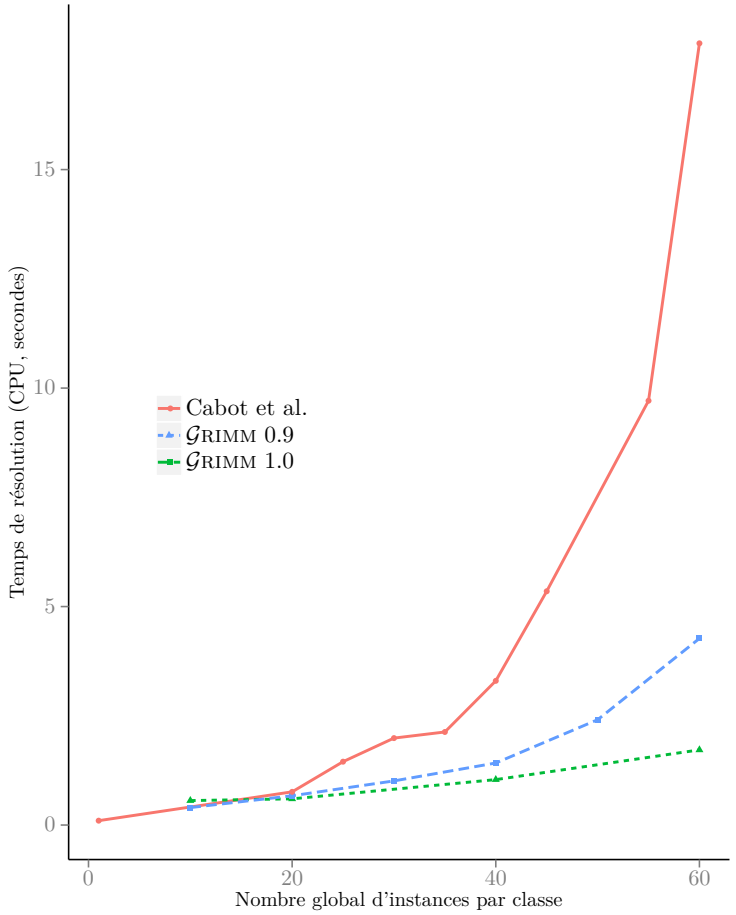


FIGURE 3.17 – Comparaison du temps de résolution entre notre méthode et l'approche de Cabot et al. [51] pour le méta-modèle Entités et Relations (5 classes).

3.3.1.3 Analyse des résultats

Les courbes précédentes montrent une nette amélioration du temps de résolution de notre solution par rapport à l'autre approche étudiée. L'amélioration observée peut-être analysée en deux étapes. Pour chacune des deux étapes nous allons donner les raisons qui ont abouti au gain de performances :

Amélioration par rapport à la solution existante La solution existante souffre d'un mauvais passage à l'échelle qui est du à une modélisation peu efficace. Voici les points les plus importants que notre modélisation corrige :

— Les instances des classes sont représentées par des domaines optimaux et contiguës. Cette solution simple est néanmoins très importante pour l'efficacité d'un solveur CSP. Elle évite l'ajout de contraintes inutiles pour distinguer les instances des différentes classes du méta-modèle.

— Le nombre de variables modélisant une référence est divisé par deux. En effet, la solution existante considère la relation de référencement comme un couple de variables, où la première référence la deuxième. Dans notre solution, le référencement est représenté par une seule variable. Cette variable est associée à l'instance qui référence. Seule l'instance référencée est représentée par la variable. Par conséquent, les référencements sont vus comme des variables pointeurs des instances qui référencent vers celles qui sont référencées.

— L'utilisation des contraintes globales est privilégiée. Ces contraintes possèdent des algorithmes et des structures de données dédiés qui les rendent plus efficaces que les contraintes plus simples dans la plupart des cas.

Amélioration par rapport à notre précédente modélisation Nous voyons également une amélioration importante entre les deux versions de notre approche. Cette amélioration est due aux points suivants :

— Les instances des classes ne sont plus représentées par des variables. En effet, le nombre d'instances pour chaque classe est un paramètre du générateur car il est choisi par l'utilisateur. Ceci implique que ce nombre restera constant et ne sera jamais modifié par le processus de résolution. C'est pour cela que nous avons fait le choix de supprimer toutes les variables qui représentent les instances des classes. Cette opération réduit drastiquement le nombre de variables et enlève quelques contraintes au passage.

— Les références bidirectionnelles sont traitées différemment. Elles ne sont plus considérées chacune séparément mais traitées comme un tout. Ceci permet de réduire fortement le nombre de variables nécessaires au traitement du couple de références, et donc de gagner en efficacité de résolution.

Note

*N*ous avons rencontré beaucoup de difficultés pour récolter des données concernant les autres approches de génération de modèles. Aucun outil de génération qui fonctionne n'est disponible. Nous nous sommes ensuite rabattus sur les données disponibles dans leurs articles publiés. Sauf que là encore, une seule approche avait des données sur le temps de génération et pour un seule méta-modèle uniquement.

3.3.2 Performance et Passage à l'échelle

Dans un second temps, nous voulons vérifier le bon passage à l'échelle de la solution en testant la génération de modèles sur beaucoup de méta-modèles de différentes tailles et origines.

3.3.2.1 Protocole

Données Nous avons rassemblé 15 méta-modèles de tailles et origines très diverses. Ces méta-modèles proviennent de la littérature ou bien sont des standards ou des langages connus.

— `Graph coloring`, Petri nets et maps sont des petits méta-modèles types que nous avons confectionnés. Ils contiennent la plupart des constructions que l'approche est capable de traiter.

— `Scaffold graph` est un méta-modèle que nous avons écrit pour représenter les graphes d'échafaudage qui est un type de graphe très utilisé en bioinformatique pour la reconstruction des génomes. ◁ *cf. chap.* 4

— `Jess, UML Class` sont extraits de [93]. Le premier représente des programmes Jess et le second une partie du diagramme de classes UML.

— `BMethod, Sad3, Business process` sont extraits de [18] et peuvent être trouvés sur le dépôt ReMoDD [3].

— `DiaGraph` est le méta-modèle de l'outil diagraph [80].

— `Feature` est extrait de [1]. Il représente des modèles de feature.

— `ER` est le méta-modèle des modèles entités et relations [51].

— `Royal&Loyal` est le méta-modèle type de Dresden OCL [4].

— `Ecore` est le méta-métamodèle d'Eclipse.

— BibTeX sert à représenter des modèles de bibliographie BibTeX [106].

3. Dépôt ReMoDD: `www.cs.colostate.edu/remodd/`
4. Desden OCL: `https://github.com/dresden-ocl`

Déroulement Pour chacun de ces méta-modèles, nous générons des modèles en faisant varier le nombre d'éléments du modèle produit. 30 modèles sont produit pour chaque jeu de données.

3.3.2.2 Résultats

		#instance de classe par modèle						
Méta-modèle	**#Classe**	**50**	**100**	**200**	**300**	**600**	**750**	**1000**
Graph Col.	2	0.33	0.39	0.41	0.47	0.60	0.67	0.81
Petri nets	5	0.47	0.69	1.25	2.25	2.58	5.76	5.85
ER	5	0.56	0.60	1.04	1.72	15.5	21.6	28.2
DiaGraph	5	0.49	0.55	0.71	0.96	3.10	4.28	7.71
Scaffold graphs	5	0.39	0.49	0.82	2.85	7.80	9.41	12.0
Maps	6	0.40	0.52	0.61	0.72	1.34	1.75	2.47
Jess	7	0.35	0.63	0.88	1.55	3.81	9.30	18.0
UML class	7	0.42	0.74	1.37	2.63	12.9	26.2	18.7
Feature	8	0.31	0.35	0.38	0.41	0.49	0.57	0.58
Ecore	17	0.42	0.66	0.83	1.84	2.05	2.86	4.18
Business process	25	0.41	0.45	0.63	0.69	1.09	1.29	1.84
BibTeX	28	0.43	0.47	0.59	0.67	0.87	1.04	1.62
Bmethod	33	0.37	0.56	0.83	1.26	3.97	5.31	11.8
Royal&Loyal	34	0.36	0.48	0.69	0.84	2.02	2.21	3.43
Sad3	39	0.43	0.48	0.51	0.57	0.77	0.87	1.00

Table 3.3 – Temps de résolution (donné en secondes) pour un benchmark de 15 méta-modèles de différentes tailles.

La table 3.3 donne la moyenne des temps de résolution pour les 15 méta-modèles. Nous observons que jusqu'à 300 instances de classes par modèle, il y a peu de disparités entre les différents méta-modèles et ce quelle que soit leur taille. Le temps de résolution ne dépasse par les 3 secondes quel que soit le méta-modèle.

Ensuite, nous constatons de nettes différences entre 2 groupes de méta-modèles. Le temps de résolution augmente de façon assez similaire entre les méta-modèles d'un même groupe.

— petits méta-modèles : on observe que la progression est plus rapide pour cette catégorie de méta-modèles. Par exemple, il faut en moyenne 28 secondes pour générer un modèle conforme au méta-modèle ER (5 classes) et contenant 1000 éléments.

— grands méta-modèles : le temps de résolution pour ce groupe de méta-modèles est

moins élevé que pour le précédent groupe. Par exemple, il faut seulement 1 seconde pour générer un modèle de 1000 éléments conforme au méta-modèle sad3 (39 classes).

Enfin, certains méta-modèles font figure d'exception et vont à l'encontre de leur groupe naturel. Voici les deux exemples les plus frappants :

— Maps (6 classes) est dans la catégorie des petits méta-modèle mais possède un comportement proches de l'autre groupe. Sa progression est très lente. En effet, un modèle de 1000 instances est généré en moyenne après seulement 2 secondes.

— BMethod (33 classes) est dans la catégorie des grands méta-modèles mais son comportement est plus proche des petits méta-modèles. Ainsi, 11 secondes sont nécessaires pour générer un modèle de 1000 instances qui lui est conforme.

3.3.2.3 Analyse des résultats

Nous donnons maintenant les raisons qui, selon nous, donnent les résultats précédents et qui expliquent aussi les exceptions observées.

Petits méta-modèles vs. grands méta-modèles Pour des raisons de clarté, l'expérimentation menée s'intéresse à la taille globale des modèles générés. En effet, elle donne le nombre global d'instances de classe du modèle en question. Par ailleurs, la modélisation que nous avons décrite aux sections précédentes montre que le nombre de variables de références dépend du nombre d'instances de classes. Plus il y a d'instances de classes, plus nous créons de variables pour modéliser leurs références. Ainsi, le nombre de variables augmente en fonction du nombre d'instances de classes.

Il parait évident qu'à nombre d'instances équivalent, il y aura nettement plus d'instances par classe pour un petit méta-modèle que pour un grand méta-modèle. Par exemple, pour obtenir un modèle de 100 éléments, il faudra 5 instances par classe pour un méta-modèle de 20 classes, alors qu'il sera nécessaire d'en avoir 20 par classe si le méta-modèle est petit et compte seulement 5 classes.

Par conséquent, il y a toujours plus de variables dans le CSP pour les petit méta-modèles si on les compare avec les plus grands. Le nombre de variables complexifie le CSP, alors il est normal qu'il soit plus facile à résoudre quand il y a peu de variables que dans le cas des grands méta-modèles.

Petits méta-modèles : les exceptions Nous avons observé que certains petits méta-modèle ne suivent pas la même règle que les autres méta-modèles de même taille. Ceci est dû à leur structure et aux optimisations apportées par notre modélisation. Par exemple, le méta-modèle maps contient des références bidirectionnelles où l'une des deux références possède une cardinalité maximale de 1 (*voir figure 3.18*). Comme, notre modélisation privilégie le traitement des références à plus petite cardinalité maximale, il y aura moins de variables de références créées et donc la résolution sera plus rapide.

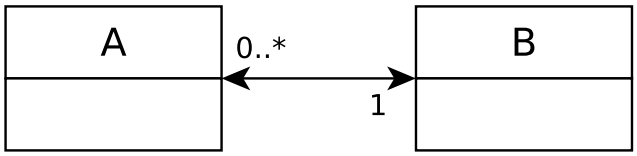


FIGURE 3.18 – Références bidirectionnelles avec cardinalités 0..∗ et 1.

Grands méta-modèles : les exceptions Inversement certains grand méta-modèles comme Bmethod ne comptent que des références dont la cardinalité maximale est ∗ même quand ce sont des références bidirectionnelles (*voir figure 3.19*). Dans ce cas, l'optimisation du traitement de ces références n'apporte pas de gain significatif en terme de nombre de variables (réduit de moitié seulement) et donc pas non plus de gain sur le temps de résolution.

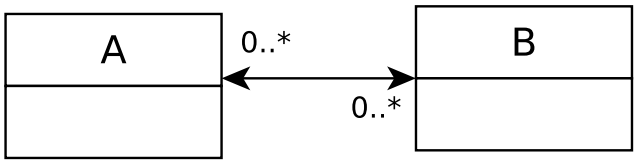


FIGURE 3.19 – Références bidirectionnelles avec cardinalités 0..∗ dans les deux sens.

3.3.3 Prise en compte de l'OCL

L'ajout des contraintes OCL est une étape importante pour obtenir des modèles valides. Lorsqu'un méta-modèle est accompagné par des contraintes OCL, il est impératif de les traduire en CSP. C'est en effet le seul et le meilleur moyen de générer des modèles conformes et respectant ces contraintes. Nous évaluons ici l'efficacité de notre solution après l'ajout des contraintes OCL accompagnant les méta-modèles.

3.3.3.1 Protocole

Données Seuls certains méta-modèles possèdent des contraintes OCL. Nous n'avons durant cette expérimentation récolté que 4 méta-modèles et environ 4 à 5 contraintes OCL pour chacun des méta-modèles. Néanmoins, ces contraintes couvrent la plupart des constructions OCL que nous modélisons et que nous avons implémentées.

Par exemple, nous donnons les contraintes OCL du méta-modèle des réseaux de Petri (figure 3.20).

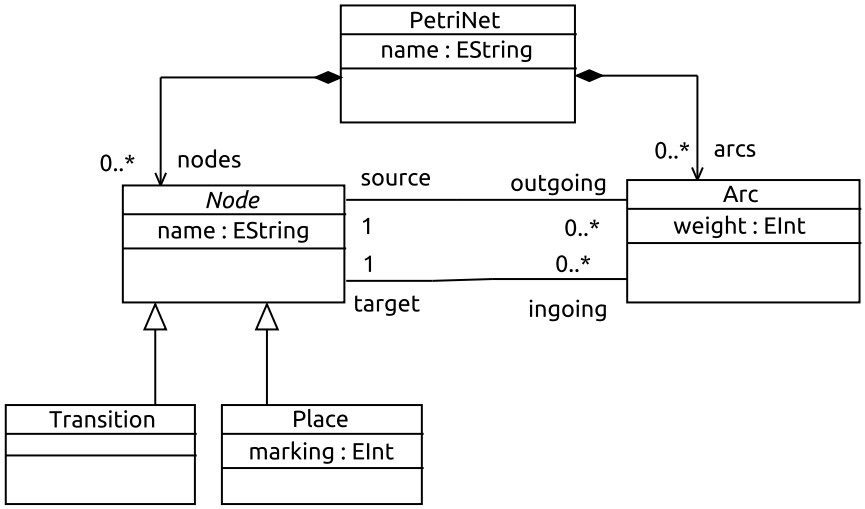


FIGURE 3.20 – Méta-modèle des réseaux de Petri.

1. `Context` Arc `inv` :
   `self`.weight >= 1 ◁ OCL

   Elle stipule que l'attribut weight de la classe Arc doit avoir une valeur strictement positive.

2. `Context` Place `inv` :
   `self`.marking >= 0 ◁ OCL

   Elle stipule que la valeur de l'attibut marking de la classe Place doit être positive.

3. `Context` PetriNet `inv` :
   `self`.nodes→`forall`(n1, n2 | n1<>n2 `implies` n1.name <> n2.name) ◁ OCL

   Elle indique que les attributs name des Node d'un réseau de Petri doivent tous être différents.

4. `Context` Arc `inv` :
   source.`oclType`() <> target.`oclType`() ◁ OCL

   Elle stipule que les types ocl source et target d'un Arc doivent être différents. Dans un réseau de Petri, un arc ne doit pas relier deux places ou deux transitions.

Déroulement Pour chacun des méta-modèles, nous faisons varier la taille des modèles générés de 50 à 600 instances de classe générés au total. Ensuite, 30 modèles sont générés pour chaque jeu de données avec deux versions du CSP :

1. Une version contenant les éléments de méta-modèle seulement.

2. Une version traduisant aussi les contraintes OCL.

La moyenne des temps de résolution par le solveur est donnée en secondes. De plus, la moyenne des différences entre les deux versions avec et sans OCL pour chaque méta-modèle est calculée.

3.3.3.2 Résultats

La table 3.4 compile les résultats et les différences pour les 4 méta-modèles étudiés. Nous observons une augmentation du temps de résolution à peu près équivalente avant et après l'ajout des contraintes OCL. Ceci laisse penser que l'ajout des contraintes avec la modélisation en CSP que nous proposons n'a pas une grande incidence négative sur les performances. En effet, les chiffres des différences entre l'ajout ou non de l'OCL pour les 4 méta-modèles sont bas voire très bas. Ainsi, pour PetriNet, il n'y a que 0.01 secondes de différence en moyenne entre les deux versions. La différence est de 0.21 secondes pour ER mais avec des temps de résolution nettement plus élevés.

méta-modèle	**Petri nets**		**ER**		**Feature**		**Graph Colour**	
#instances	**OCL ?**		**OCL ?**		**OCL ?**		**OCL ?**	
	oui	**non**	**oui**	**non**	**oui**	**non**	**oui**	**non**
50	0.46	0.47	0.58	0.56	0.31	0.31	0.36	0.33
100	0.70	0.69	0.67	0.60	0.36	0.35	0.43	0.39
300	2.26	2.25	1.88	1.72	0.42	0.41	0.62	0.47
600	2.57	2.58	16.1	15.5	0.47	0.49	0.85	0.60
différence	0.01		0.21		0.01		0.11	
%**différence**	1.1%		6.5%		%2.35		17.8%	

Table 3.4 – Temps de résolution (en secondes) pour 4 méta-modèles leurs contraintes OCL.

3.3.3.3 Analyse des résultats

Dans cette sous-section, nous analysons les précédents résultats. Nous donnons les explications sur la modélisation en CSP qui conduisent à de tels chiffres.

— **Contraintes OCL sur les attributs** Ces contraintes sont traitées par la réduction de leur domaine. Réduire un domaine n'est pas coûteux pour la résolution car cela est fait en dehors du solveur et il est même bénéfique pour les solveurs car ça implique moins de possibilités à tester.

— **Contraintes OCL sur les opérations de collections** Certaines des opérations sur les collections sont également traitées par réduction d'un domaine comme par exemple l'opération `excludes`. Ceci n'est pas coûteux et même plutôt bon pour l'accélération du processus de résolution.

— **Contraintes OCL sur les opérations de collections** D'autres contraintes OCL sur les opérations de collections sont traduites par des contraintes globales en CSP. Les contraintes globales sont connues pour leur efficacité (à expressivité équivalente) à cause des algorithmes qui leur sont dédiés.

3.3.4 Contraintes OCL imbriquées et satisfiabilité

Nous allons présenter dans cette section une situation dans laquelle notre approche permet de formaliser en CSP des contraintes OCL complexes contenant de l'imbrication d'opérations. Le but de cette expérience est de détecter l'incohérence des contraintes OCL et donc l'impossibilité de les satisfaire.

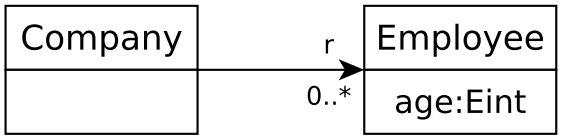


FIGURE 3.21 – Un petit méta-modèle Ecore de deux classes.

Pour cela, nous donnons le petit méta-modèle de la figure 3.21 qui contient deux classes et une référence. Ce méta-modèle est accompagné des contraintes OCL suivantes :

1. `Context` Company `inv` :
   `self`.r→`collect`(e Employee | e.age <= 18)→`AsSet`()→`size`()=3. ◁ OCL

2. `Context` Company `inv` :
   `self`.r→`includes`({20..25}) ◁ OCL

3. `Context` Company `inv` :
   `self`.r→`collect`(e Employee | e.age > 0 `and` e.age < 40)→`isEmpty`() ◁ OCL

La formalisation en CSP des contraintes OCL précédentes nécessite la création des contraintes globales suivantes :

1. `among`(3, [Variables d'age], [1..18]) ◁ CSP

   Elle stipule que 3 valeurs différentes parmi les valeurs 1 à 18 seront affectées aux variables modélisant l'attribut age.

2. `gcc`([Variables d'age], [20..25], [1..1], [5..5]) ◁ CSP

   Elle stipule que les valeurs de 20 à 25 devront apparaître au moins une fois chacune et au plus 5 fois (car il y a 5 valeurs).

CSP ▷ 3. `gcc`([Variables d'age], [0..40], [0..0], [0..0])

   Elle stipule que les valeurs 0 à 40 ne doivent pas être assignées aux variables d'age.

La formalisation précédente montre qu'une seule contrainte globale efficace suffit à transformer une succession d'opérations OCL imbriquées vers CSP.

Il faut uniquement 1 seconde au solveur abscon pour vérifier que les contraintes OCL sont incohérentes et qu'il ne peut donc pas trouver de solution. En effet, la première contrainte force l'apparition des valeurs 1 à 18 alors que la dernière interdit l'assignation des valeurs 0 à 40. Cela est évidemment incorrect.

3.3.5 Menaces à la validité

En dépit des bons résultats des expérimentations qui viennent d'être décrites, nous estimons que leur validité peut être menacée pour plusieurs raisons.

D'abord, la comparaison s'est faite avec une seule approche existante (Cabot et al.) et sur un corpus de données assez réduit. Néanmoins, ceci est dû à l'absence d'autres outils de génération de modèles fonctionnels.

Ensuite, le benchmark de méta-modèles choisi contient des données de tailles et d'origines diverses. Il se peut que ces données ne soient pas aussi représentatives de tous les méta-modèles qu'on le voudrait.

Enfin, concernant la prise en compte des contraintes OCL des méta-modèles, nous avons implémenté nombre de constructions importantes du langage OCL mais d'autres constructions que nous n'avons pas implémenté sont susceptibles d'être rencontrées. Néanmoins, nous avons implémentées toutes les constructions rencontrées durant notre collecte de contraintes OCL. Par ailleurs, Juan Cadavid a montré durant sa thèse [20] que les patterns les plus fréquents de contraintes OCL dans les méta-modèle Ecore sont simples. Par exemple, on ne retrouve que très peu d'imbrication et la navigation ne concerne que une à deux références. La partie du langage OCL que nous prenons en compte suit la totalité des patterns donnés dans cette thèse et d'autres constructions encore.

3.4 Conclusion

Ce chapitre a présenté la première contribution de cette thèse. Il s'agit d'une nouvelle approche de génération de modèles conformes à des méta-modèles Ecore. Nous utilisons une traduction en CSP des éléments du méta-modèle et de ses contraintes OCL. Une évaluation de l'approche en trois temps est menée pour prouver son efficacité et son respect des deux

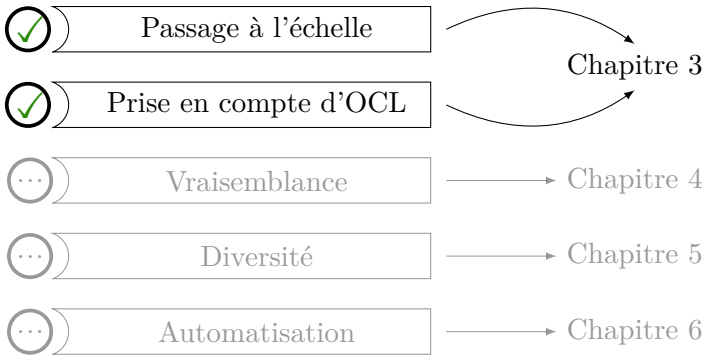


FIGURE 3.22 – Caractéristiques de la génération de modèles abordées par le chapitre 3

premières caractéristiques d'une bonne approche de génération de modèles : (1) Passage à l'échelle et (2) prise en compte des contraintes OCL (voir figure 3.22).

La première section du chapitre est consacrée à la modélisation des éléments composant un méta-modèle en CSP. Pour chaque élément, nous donnons les variables, domaines et contraintes CSP nécessaires. Plusieurs optimisations améliorant l'efficacité de l'approche sont par ailleurs décrites.

Ensuite, une section présente la traduction d'un certain nombre de constructions du langage OCL en CSP. Nous nous efforçons d'utiliser les contraintes CSP les plus efficaces telles que les contraintes globales pour un meilleur passage à l'échelle. Les constructions du langage OCL les plus importantes et les plus fréquemment rencontrées sont prise en compte.

La dernière section mène des expérimentations qui évaluent notre approche selon trois axes :

1. Comparaison avec une approche de génération de modèles existante.
2. Génération de modèles conformes à un ensemble de méta-modèles de tailles et d'origines diverses.
3. Test de l'influence de l'ajout des contraintes OCL sur les temps de résolution.

Pour chacune des expérimentations menées, le protocole expérimental, les données et une analyse détaillée sont donnés pour expliquer les résultats. Enfin, les menaces à la validité de l'évaluation sont également énoncées.

Des points d'amélioration subsistent, particulièrement en ce qui concerne la satisfaction des CSP obtenus à partir des méta-modèles et la prise en compte des contraintes OCL.

Lorsqu'une solution n'est pas trouvée par le solveur, il est intéressant d'identifier la partie du CSP qui a conduit à un échec et donc l'élément du méta-modèle ou des paramètres de configuration à corriger. En programmation par contraintes, il existe une variante des CSP appelée Max-CSP [65] qui permet de trouver le plus grand sous-ensemble de contraintes pouvant être satisfaite pour un problème donné.

Par ailleurs, nous avons remarqué que l'incapacité à satisfaire les CSP produits est dû dans notre cas à de mauvaises cardinalités de références ou bien à l'impossibilité d'assigner des liens quand le nombre d'instances pour une classe donnée est trop petit. Des travaux permettant de tester la consistance des cardinalités d'un modèle UML ont été menés dans [11]. Nous pouvons les adapter aux méta-modèles Ecore pour guider l'utilisateur vers le bon choix de cardianlités et de nombre d'instances par classe.

Bien que notre approche fasse partie de celles qui prennent en compte le plus de constructions du langage OCL, nous pensons que l'automatisation de toutes les contraintes OCL possibles est souhaitable. Cela permettra aux utilisateurs de notre outil de transformer toutes les contraintes qu'ils désirent, notamment les contraintes imbriquées. L'une des pistes que nous envisageons est le développement d'un compilateur complet d'OCL vers CSP. C'est un travail qui a déjà été entamé durant cette thèse et qui permet la prise en compte de contraintes complexes, même si une étude menée par Juan Cadavid [20] sur les contraintes OCL des méta-modèles Ecore a montré que les constructions les plus fréquentes sont simples et qu'il existe peu d'imbrication.

Chapitre 4

Génération d'Instances Pertinentes, Réalistes et Vraisemblables

Soyez réalistes, demandez l'impossible.

Ernesto Guevara

Synopsis

Préambule

Ce chapitre s'intéresse à la pertinence et au réalisme des modèles générés automatiquement. Une technique améliorant les caractéristiques des modèles issue des CSP est présentée en section 4.1. Une évaluation comparant les modèles générés à des modèles réels est présentée à la section 4.2. Une approche basée sur l'utilisation de lois de probabilités usuelles pour améliorer la pertinence des modèles est décrite en section 4.3. Enfin, deux études de cas validant cette approche sont données en sections 4.4 et 4.5.

Force est de constater que les modèles générés par notre approche, décrite au chapitre 3, ne possèdent pas les caractéristiques attendues, qu'elles soient juste visuelles, graphiques ou de diversité des éléments du modèle. Lorsque nous observons un modèle généré automatiquement (exemple figure 4.1), il apparaît que les points suivants sont à améliorer :

— Présence d'un grand nombre d'instances libres qui ne sont reliées à aucune instance autre que celle de la classe racine (lien gris). Nous cherchons donc à améliorer la connexité du modèle, si on fait abstraction de la racine.

— Présence d'un certain nombre de composantes connexes de tailles hétérogènes. En effet, il existe une grosse composante réunissant la plupart des instances de classe et des composantes toutes petites à coté d'elle. Nous allons donc chercher à ré-équilibrer les composantes connexes tout en conservant une certaine variété dans leurs tailles.

— Absence de diversité dans les degrés des instances. Ainsi, une instance est soit libre ou bien est toujours liée au même nombre d'éléments. Nous cherchons à distribuer ces degrés de façon moins binaire.

— Absence de diversité des valeurs des attributs. Par défaut, les mêmes attributs d'une classe portent la même valeur d'une instance à l'autre. Nous cherchons donc à instiller une part d'aléatoire dans le choix des valeurs des attributs.

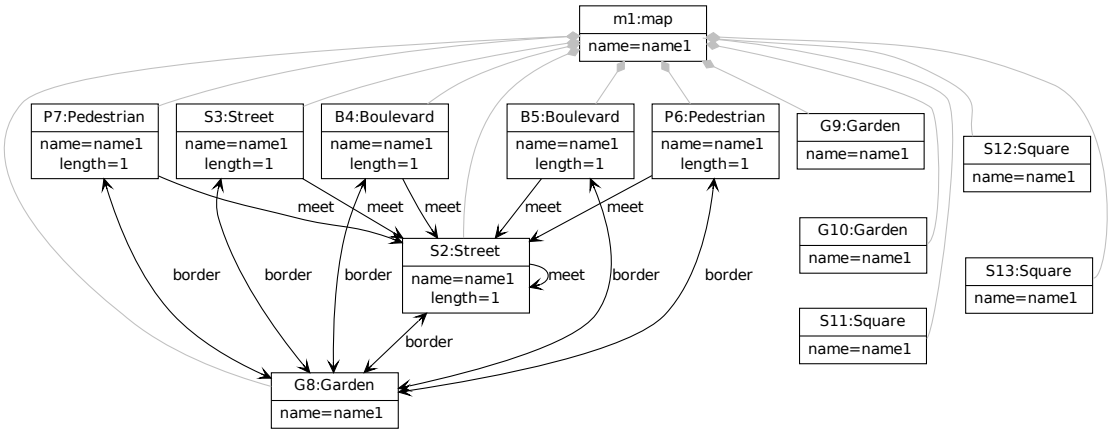


FIGURE 4.1 – Modèle généré automatiquement conforme au méta-modèle maps

Dans le but de générer des modèles utilisables, nous devons améliorer ces caractéristiques et répondre à la problématique de générer des modèles réalistes, vraisemblables et pertinents. Voici les définitions par le dictionnaire Le Robert 2016 de ces trois termes.

Définition.

Réaliste Caractère de ce qui est réel, de ce qui existe effectivement.

Définition.

Vraisemblable Qui peut être considéré comme vrai ; qui semble vrai. *synon.* plausible, envisageable.

Définition.

Pertinent Qui convient exactement, dénote du bon sens. *synon.* approprié, adéquat.

Un modèle réaliste doit exprimer fidèlement la réalité et donc s'apparenter aux modèles réels conçus par un expert ou issus d'une utilisation concrète. Les modèles doivent donc avoir des caractéristiques équivalentes ou proches des modèles réels. Ces caractéristiques peuvent être visuelles, des mesures de graphes ou encore des métriques d'un domaine donné. Ce chapitre présente deux contributions dont l'objectif est de générer des modèles pertinents, réalistes et vraisemblables.

La première consiste à casser les symétries qui peuvent apparaître dans les modélisations CSP. Elle sera évaluée en comparant selon certaines mesures de graphe (connectivité, degré des nœuds) les modèles générés et des modèles réels collectés.

La deuxième quant à elle, exploite les métriques d'un domaine donné pour en déduire des lois de probabilités usuelles. Ces dernières seront ensuite utilisées pour améliorer la vraisemblance des modèles générés. Deux cas d'études issus de deux domaines différents (code orienté objet et bioinformatique) serviront à l'illustration et la validation de cette approche.

Pour rendre les modèles pertinents ou appropriés, un soin est porté aux choix de l'utilisateur. En effet, ce dernier peut exprimer le besoin de générer des modèles éloignés de la réalité mais qui répondent à des critères liés à une utilisation précise de ces modèles.

4.1 Casser les symétries

Il est fréquent que les programmes contraints introduisent des symétries. Une symétrie dans un CSP veut dire que de nouvelles solutions peuvent être obtenues juste en permutant les valeurs de certaines variables d'autres solutions. Par exemple, si $\{x_1 = 1, x_2 = 3, x_3 = 2\}$

est une solution d'un problème contraint et que la permutation n'a pas de sémantique particulière pour ce problème, alors $\{x_1 = 1, x_2 = 2, x_3 = 3\}$ est probablement une autre solution. Il en sera de même pour toutes les permutations de $\{1, 2, 3\}$. On dit donc que les symétries partitionnent l'espace des solutions possibles d'un CSP en classes d'équivalence [47]. Par conséquent, trouver toutes les solutions revient à identifier le représentant de chaque classe d'équivalence. Cela a pour but d'éviter l'exploration de pans redondants de l'espace de recherche.

4.1.1 Techniques pour casser les symétries

Il existe dans la littérature une multitude de techniques pour casser les symétries en programmation par contraintes.

Gent et Smith décrivent dans [47] une technique pour casser les symétries durant la recherche (Symmetry Breaking During Search, SBDS). Dans cette méthode, le programmeur identifie les symétries mais c'est le solveur qui se charge de rajouter des contraintes pour casser ces symétries durant la phase de recherche.

Crawford et al. proposent dans [28] d'ajouter à la modélisation en CSP des contraintes pour casser les symétries. Ces contraintes ont la particularité d'être satisfaites uniquement par un représentant de chaque classe d'équivalence induite par la symétrie. Cette technique se veut hautement automatisée car c'est le solveur qui détecte les symétries. Cependant, les auteurs font remarquer que la détection de symétrie n'est pas chose aisée. Ils ajoutent, néanmoins, que dans la pratique, les algorithmes de détection s'en sortent quand même assez bien et que de toute façon on peut se contenter de casser les symétries ne serait-ce que partiellement.

Brown et al. proposent dans [14] d'adapter les algorithmes de recherche et de backtracking à la présence de symétries. Le solveur évite de parcourir des branches symétriquement équivalentes. Les auteurs affirment que leur méthode permet un gain de temps important pour des problèmes de tailles modérées.

Jean-François Puget décrit dans [84] une technique qui se base sur l'ajout par le programmeur en CSP de contraintes à sa modélisation dans le but de casser les symétries. Vu que les symétries ne sont en réalité que des permutations de variables CSP conduisant à des solutions équivalentes, la déclaration d'un ordre lexicographique sur ces variables est une des solutions.

Plusieurs autres techniques existent pour casser les symétries d'un problème contraint. Focacci et Milano les discutent dans [39].

4.1.2 Dans notre approche

Notre modélisation en CSP introduit des symétries entre les variables modélisant les références de chaque instance de classe. Par exemple, la figure 4.2 montre un extrait de méta-modèle contenant deux classes reliées par une référence et deux extraits de modèles

conformes. Les deux extraits de modèles sont équivalents. En effet, même si les valeurs des trois variables modélisant la référence r pour l'instance $a1$ ne sont pas identiques entre les deux modèles, il n'en demeure pas moins que dans les deux cas, l'instance $a1$ de la classe A est liée exactement aux mêmes instances $b3$, $b4$ et $b5$ de la classe B.

Par conséquent, il existe une classe d'équivalence entre les variables modélisant chaque référence d'une instance de classe donnée. Un seul représentant pour chaque classe d'équivalence doit être pris en compte pour éviter les solutions redondantes et symétriquement équivalentes.

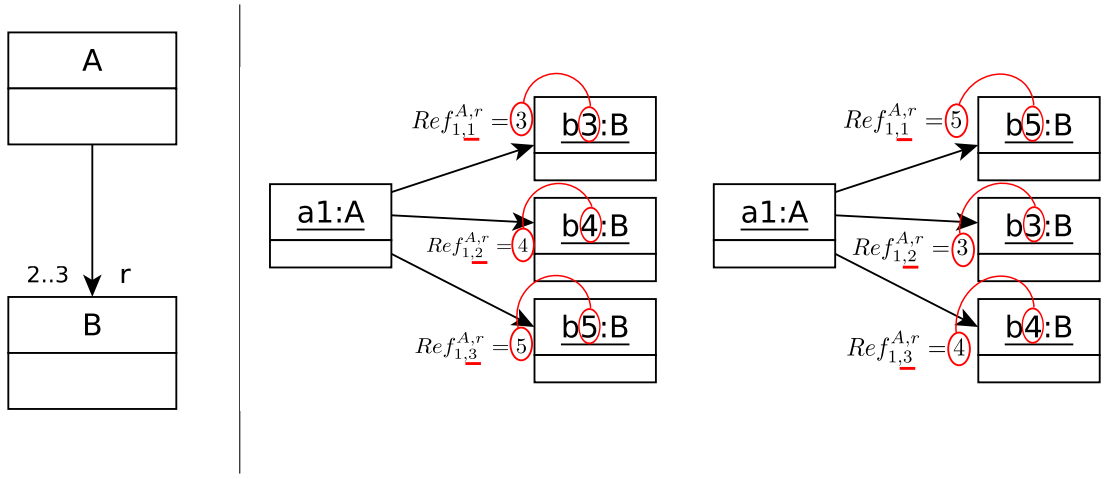


FIGURE 4.2 – Extrait d'un méta-modèle à gauche, suivi de deux extraits de modèle symétriquement équivalents.

Nous choisissons d'ajouter des contraintes d'ordre lexicographique à notre modélisation en CSP pour casser les symétries induites. Ce choix est motivé par la facilité de mise en place sans changer de solveur et sans devoir re-coder des parties de celui-ci. Pour toute classe C, et pour toute référence r de C, nous créons les contraintes suivantes :

$$\forall i \in [1, D_C], Ref_{i,1}^{C,r} < Ref_{i,2}^{C,r} < \ldots < Ref_{i,size(r)}^{C,r}. \qquad \triangleleft \text{ CSP}$$

Les contraintes précédentes ordonnent les variables modélisant la référence r de chaque instance i de la classe C. De cette manière, seul le modèle à gauche dans l'exemple de la figure 4.2 sera pris en compte pour représenter tous ceux qui lui sont équivalents.

4.1.3 Améliorer la connectivité des modèles

Dans l'implémentation de notre approche, c'est à l'utilisateur de choisir les paramètres de taille du modèle à générer (nombre d'instances par classe, borne pour les références et domaines des attributs). Par conséquent, la qualité des modèles générés peut être impactée voir chap. 6

par un mauvais paramétrage. Lorsque des valeurs inadéquates sont choisies, deux effets sont observés :

Beaucoup plus d'instances de classe que de variables de références Dans ce cas, beaucoup d'instances ne seront pas référencées ce qui augmente le nombre d'instances libres et donc celui des composantes connexes. De plus les composantes connexes sont de tailles déséquilibrées.

Beaucoup plus de variables de références que d'instances de classe Dans ce cas, le solveur créera beaucoup de liens redondants (plusieurs variables de références qui ont la même valeur). Ces liens redondants ne sont pris en compte qu'une seule fois au moment de construire le modèle ce qui réduit la connectivité du modèle (les degrés des instances de classe).

Les contraintes lexicographiques réduisent donc le nombre d'instances libres et augmentent le degré des instances et donc la connectivité des modèles. De plus, il s'opère un rééquilibrage des tailles des composantes connexes. En outre, elles permettent de réduire le nombre de liens redondants et donc d'augmenter les degrés des instances de classe. Toutefois, elles permettent uniquement de prévenir d'éventuels problèmes de résolution liés aux choix de l'utilisateur et non de corriger ceux-ci.

Par ailleurs, la technique que nous avons choisie pour casser les symétries permet également de réduire dans une certaine mesure le biais qui existe dans le choix des valeurs à assigner par le solveur. En effet, par défaut le solveur choisira toujours les premières valeurs des domaines. Ceci a pour effet d'empêcher les autres valeurs d'être assignées à des variables et donc réduit la diversité.

4.2 Évaluation

Le but de cette évaluation est de mesurer l'impact de casser les symétries sur la vraisemblance des modèles générés. Cette vraisemblance se mesure par certaines caractéristiques de graphe (composantes connexes, nœuds isolés, degré des nœuds). Ces mesures sont ensuite comparées à des mesures effectuées sur des modèles réels collectés.

4.2.1 Protocole

Données Nous avons récolté 240 modèles réels conformes à 10 méta-modèles différents. Il y a au moins 5 modèles par méta-modèle et au plus 49. Puis, nous générons à l'aide de deux versions de notre approche (avec et sans casser les symétries) 30 modèles par version et par méta-modèle.

Protocole Une première étape consiste à transformer les modèles collectés en graphes et à prendre 3 mesures différentes : nombre de composantes connexes, nombre d'instances (ou nœuds) libres et moyenne des degrés des nœuds.

Dans un deuxième temps, deux corpus de modèles conformes sont générés pour chaque méta-modèle :

1. Générer 30 modèles conformes avec la version de la modélisation en CSP présentée au chapitre 3.
2. Générer 30 modèles conformes avec la version de la modélisation en CSP plus les contraintes lexicographiques qui cassent les symétries sur les variables des références.
3. Construire les graphes correspondants aux modèles sans prendre en compte l'instance de la racine sinon le nombre de composantes connexes et le nombre d'instances libres seront constants et égaux respectivement à 1 et 0.
4. Mesurer les 3 caractéristiques précédentes sur les graphes obtenus.

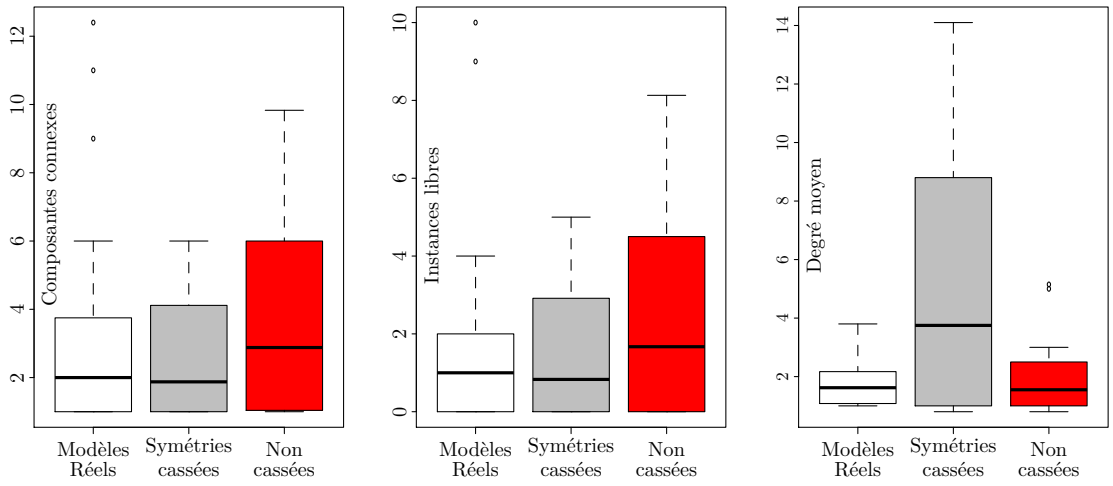
4.2.2 Résultats

La table 4.1 montre les résultats des mesures des 3 caractéristiques prises sur le corpus de modèles réels.

La figure 4.3 montre trois boites à moustaches, chacune correspondant à une des 3 caractéristiques. Elles comparent les modèles réels aux modèles générés avec et sans casser les symétries. Une boite à moustache se lit de la façon suivante : la ligne la plus haute (resp. la plus basse) correspond à 95% (resp. 5%) de l'échantillon, la petite boite s'échelonne de 25% à 75% de l'échantillon, tandis que la ligne en gras au centre correspond à la médiane (50% de l'échantillon).

Méta-modèle	**#Modèles**	**Composantes Connexes**	**Instances libres**	**Degré moyen**
Petri nets	49	1.00	0.00	1.00
BibTeX	44	12.4	1.20	1.08
Graph	40	1.08	0.08	3.80
Ecore	40	1.25	0.20	2.08
ER	20	1.00	0.00	1.00
Feature	20	3.40	1.85	2.09
UMLClass	10	1.00	0.00	2.17
DiaGraph	4	1.00	0.00	2.31
Maps	5	1.00	0.00	4.05
Jess	5	3.50	0.00	1.60

Table 4.1 – Mesures prises sur les modèles réels.



(a) #Composantes connexes. (b) #Instances libres (c) Degré moyen

FIGURE 4.3 – Utilisation des boites à moustaches pour la comparaisons des modèles générés et des modèles réels. Les pas représentent de bas en haut 5%, 25%, 50% ; 75% et 95% de l'échantillon. Les points isolés sont des valeurs extrêmes.

4.2.3 Discussion

Les mesures prises sur les modèles réels montrent que ces derniers possèdent les caractéristiques suivantes :

— Ils sont généralement fortement connectés, c'est-à-dire que le nombre de composantes connexes est égal à ou proche de 1 à l'exception de quelques méta-modèles contenant des constructions particulières. Ainsi, le méta-modèle BIBTEX contient une classe (Misc) qui n'est reliée à aucune autre et le méta-modèle Feature contient des ensembles de features liées entre elles mais indépendantes des autres ensembles de features.

— Ils ont très peu d'instances libres. Généralement le nombre d'instances libres se situe entre 1 et 2 maximum.

— Leur degré moyen est un peu plus variable d'un méta-modèle à un autre mais il est quand même assez petit, entre 1 et 4 maximum.

La comparaison avec les modèles générés montre des résultats plutôt bons. Nous observons une proximité entre les caractéristiques des modèles réels et des modèles générés par l'ajout des contraintes qui cassent les symétries et ce pour 2 des 3 mesures. Nous allons maintenant commenter chacune des mesures séparément.

— **Composantes connexes :** le nombre de composantes connexes des modèles générés avec symétries cassées correspond quasi-parfaitement à celui des modèles réels. Par contre, on voit qu'il est assez élevé lorsque les symétries du problème contraints ne sont pas traitées.

— **Instances libres :** Le nombre d'instances libres est aussi très proche dans les deux cas des modèles réels et des modèles générés après l'ajout des contraintes pour casser les symétries. Tandis que la solution basique induit la présence de beaucoup plus d'instances libres.

— **Degré moyen :** Pour le degré moyen, nous observons des résultats plus réalistes quand les symétries ne sont pas cassées. Ceci est dû au biais statistique introduit par la prise en compte de la moyenne des degrés par modèle. En effet, il existe un grand déséquilibre entre les degrés des nœuds mais la moyenne est basse car rapportée au nombre de nœuds. Casser les symétries augmente fortement le degré moyen mais de façon homogène entre tous les nœuds et les composantes connexes. Ce biais pourrait être outrepassé en s'intéressant à la distribution des degrés pour chaque modèles séparément.

4.2.4 Menaces à la validité

Nous pouvons identifier deux menaces à la validité de cette évaluation :

— Nous avons choisi de comparer les modèles selon 3 caractéristiques de graphes (nombre de composantes connexes, nombre d'instances libres et degré moyen des instances de classes). Il existe d'autres mesures possibles et ces 3 peuvent être biaisées ou encore non représentatives.

— Un corpus de modèles réels conformes à 10 méta-modèles différents a été collecté. Des interrogations peuvent être formulées sur l'homogénéité de ces données ou bien leur représentativité de toutes les instances de ces méta-modèles.

4.3 Des lois de probabilités pour améliorer la vraisemblance

Améliorer la vraisemblance et la pertinence des modèles générés aléatoirement implique de leur ajouter l'aléatoire qu'on retrouve dans les modèles réels. En effet, les modèles réels possèdent un côté aléatoire que n'ont pas les algorithmes de résolution des CSP qui eux sont intrinsèquement déterministes. Nous retrouvons cet aléatoire notamment sur les liens entre objets. Prenons l'exemple de la figure 4.5 où deux modèles conformes au méta-modèle de la figure 4.4 sont constitués d'objets noirs et d'objets blancs reliés entre eux. Le modèle gauche est qualifié de non vraisemblable car il y a une constance dans les degrés sortants des objets noirs et des degrés entrants des objets blancs. Un pattern où un objet noir est lié à deux blancs apparaît constamment. Par contre le modèle droit est dit vraisemblable

car on ne retrouve plus cette constance que possède le premier modèle. En effet, il y a une meilleure diversité dans les degrés des deux types d'objets.

Note

*E*n théorie algorithmique de l'information on dira que le modèle de gauche est moins complexe et contient moins d'informations que celui de droite car on peut le résumer très facilement par "six composantes constitués de trois éléments : un noir pointant vers deux blancs". Le modèle de droite est plus proche de ce que nous attendons d'un modèle, c'est-à-dire, qu'il soit plus complexe et contienne plus d'informations.

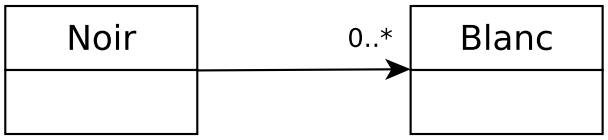


FIGURE 4.4 – Extrait de méta-modèle de deux classes.

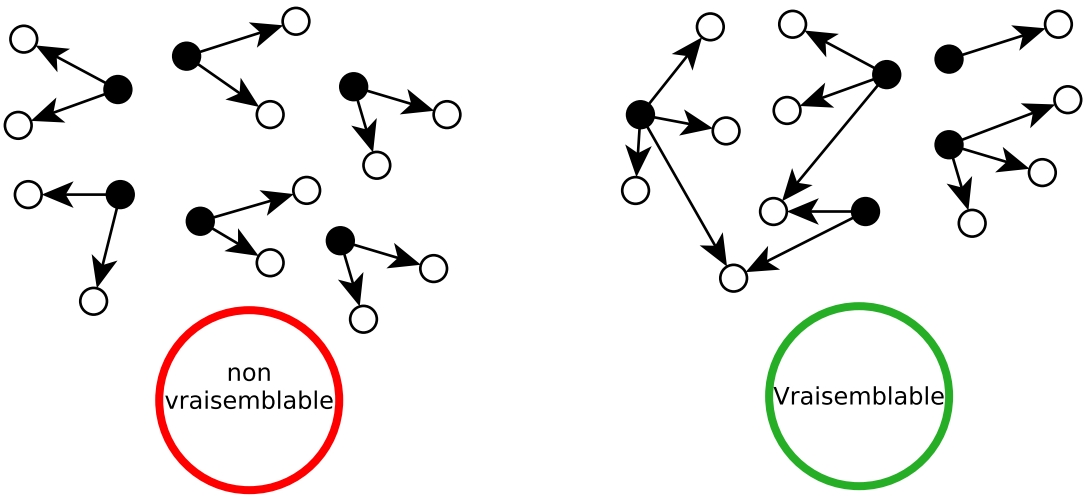


FIGURE 4.5 – Deux modèles conformes à l'extrait de méta-modèle de la figure 4.4. L'un est pertinent et vraisemblable et l'autre non.

Dans la réalité, il est plus intéressant de générer automatiquement des modèles avec des caractéristiques proches du modèle de droite que de celui de gauche. Ainsi, si nous générons des diagrammes de classes UML contenant 3 packages et 9 classes, nous préférerons un package contenant 2 classes, un autre en contenant 4 et un dernier de 3 classes, plutôt que 3 packages de 3 classes chacun.

Cette section décrit une approche se basant sur la simulation de lois de probabilités usuelles pour injecter de l'aléatoire dans les modèles générés et donc d'améliorer leur vraisemblance.

4.3.1 Simulation de lois de probabilités usuelles

Voici les deux raisons principales qui nous poussent à exprimer la vraisemblance des modèles en utilisant des lois de probabilités :

— Les algorithmes de résolution des CSP sont déterministes et ne nous permettent pas d'obtenir des solutions vraisemblables. Même quand on va très loin dans l'arbre de résolution, la millionième solution reste très insatisfaisante car il sera toujours plus facile et plus rapide pour les solveurs de contraintes de choisir la solution la plus naïve et qui ne répond donc pas à nos exigences de pertinence.

— Il n'existe dans le monde de l'IDM aucun langage pour exprimer la diversité intrinsèque des modèles. Ainsi, en OCL il est possible d'écrire des contraintes sur le degré des instances d'une classe (`size`) mais il est par exemple impossible de spécifier des moyennes ou des distributions (dire qu'en UML un package contient en moyenne $2,5$ classes).

4.3.1.1 Écrire un simulateur

Une loi de probabilité X est caractérisée par deux fonctions importantes : la fonction de densité f_X et la fonction de répartition F_X (voir figure 4.6).

Toutes les méthodes de simulation présentées à cette section [73] supposent l'existence d'un générateur d'échantillons uniformes sur $[0,1]$:

$$U \rightsquigarrow \mathcal{U}_{[0,1]}.$$

L'existence d'un générateur de variables aléatoires uniformes sur $[0,1]$ est une condition *sine qua non* pour toute simulation de loi de probabilités.

Les simulations que nous présentons se basent toutes sur l'inversion de la fonction de répartition pour générer un échantillon d'une loi. En effet, une fonction de répartition $F : \mathbb{R} \mapsto [0,1]$ prend ses valeurs dans l'intervalle $[0,1]$ et donc son inverse $F^{-1} : [0,1] \mapsto \mathbb{R}$ permet à partir d'échantillons suivant la loi uniforme sur $[0,1]$ de trouver des échantillons de la loi qu'elle caractérise. La figure 4.7 illustre cette opération.

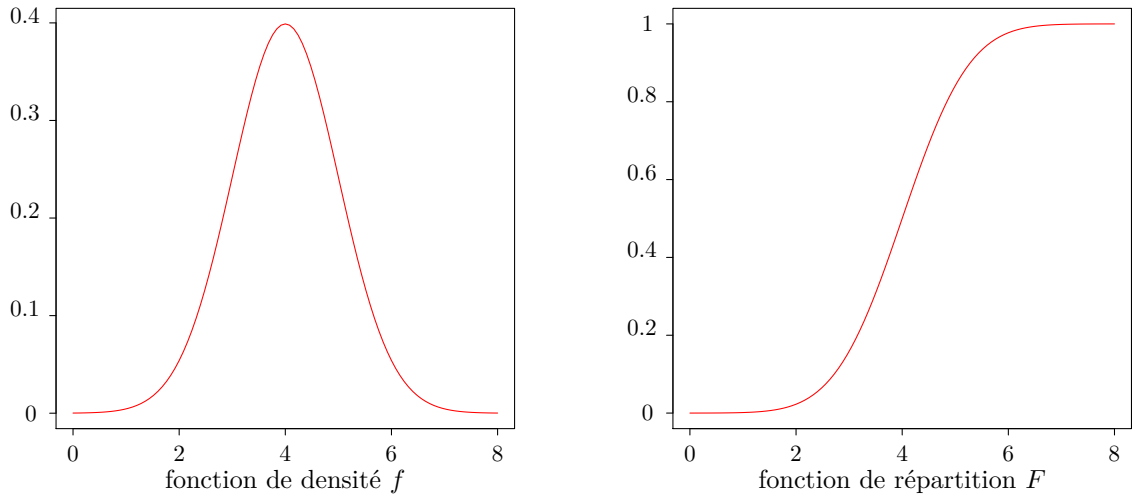


FIGURE 4.6 – Les fonctions de densité (gauche) et de répartition (droite) d'une loi normale.

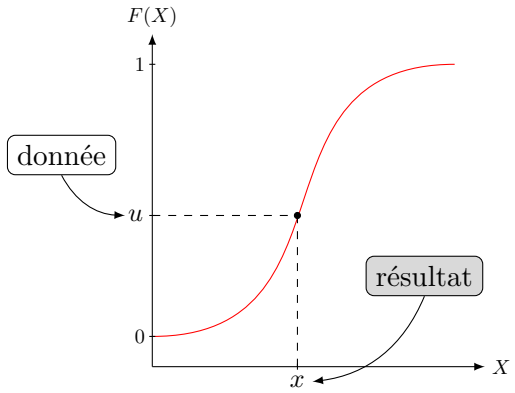


FIGURE 4.7 – Simulation d'un échantillon x d'une loi de probabilité X étant donné un échantillon uniforme u, une fonction de répartition F et son inverse F^{-1}.

Nous allons maintenant introduire quelques lois parmi les plus connues et surtout celles que nous avons le plus souvent rencontrées lors de notre collecte de données sur les modèles réels.

4.3.1.2 Loi discrète sur un ensemble fini

Soit X une loi de probabilité à valeurs dans $\{1, 2, \ldots, K\}$. On note p_k la probabilité que $X = K$ ($p_k = \sum_{k=1}^{K} p_k = 1$).

Le cumul des probabilités p_k est donné par : $P_k = \sum_{j \leq k} p_j$, $P_0 = 0$.

On tire un uniforme u : $P_{k-1} < u \leq P_k \rightarrow x = k$. Cela revient à décomposer l'intervalle $[0, 1]$ en k morceaux et d'en choisir un en fonction de la valeur de u (figure 4.8).

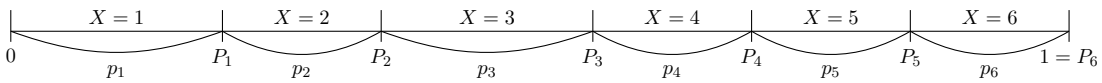


FIGURE 4.8 – Simulation d'une loi de probabilité discrète sur un ensemble fini.

Nous avons observé que cette loi concerne le plus souvent les valeurs des attributs des modèles. Par exemple, si un attribut est de type énumération, alors nous donnons une probabilité à chacune des valeurs de l'énumération.

4.3.1.3 Loi de Bernoulli

Pour simuler une loi de Bernoulli $X \rightsquigarrow \mathcal{B}(p)$ on tire un uniforme u :

$$x = 1, \text{ si } u \leq p$$
$$x = 0, \text{ sinon.}$$

La loi de Bernoulli n'est pas rencontrée dans notre cas. Même si elle peut servir pour les attributs booléens. Ici, elle sert surtout à introduire la loi binomiale.

4.3.1.4 Loi binomiale

Une variable binomiale $X \rightsquigarrow \mathcal{B}(n, p)$ (figure 4.9) est la somme de n variables de Bernoulli $Y_i \rightsquigarrow \mathcal{B}(p)$. Pour chaque valeur de X, il est nécessaire de tirer uniformément n nombres u et de faire la somme des n valeurs trouvées.

$$X = \sum_{i=1}^{n} 1\{U_i \leq p\}$$

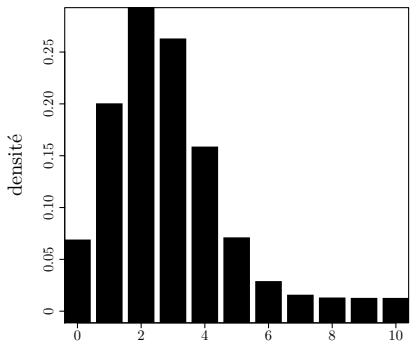


FIGURE 4.9 – Histogramme d'une loi binomiale $\mathcal{B}(n = 10, p = 0.25)$.

4.3.1.5 Loi exponentielle

La simulation d'une loi exponentielle (figure 4.10) se base sur l'inverse de sa fonction de répartition.

Soit une variable aléatoire $X \rightsquigarrow \varepsilon(\lambda)$ qui suit une loi exponentielle de paramètre λ, la simulation de X se fait par le tirage d'un uniforme u et le calcul de $x = F^{-1}(u)$ car :

$$F(x) = 1 - e^{-\lambda x} \Leftrightarrow F^{-1}(u) = -\frac{\ln(u)}{\lambda}$$

La méthode de simulation par fonction de répartition inverse peut néanmoins être utilisée pour toute loi de probabilité dont l'inverse de la fonction de répartition est facile à calculer.

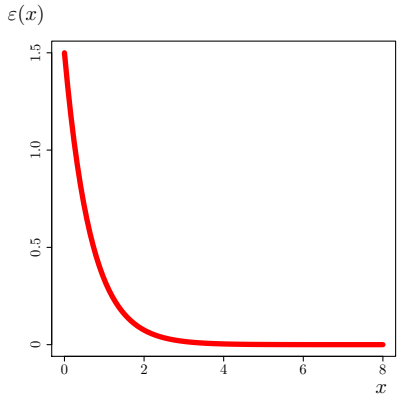


FIGURE 4.10 – Fonction de densité de d'une loi exponentielle $\varepsilon(\lambda = 1.5)$.

4.3.1.6 Loi normale

La loi normale $\mathcal{N}(\mu, \sigma)$ (figure 4.11) ne possède pas d'inverse connue de sa fonction de répartition. Pour simuler une variable aléatoire qui suit une loi normale, nous utilisons l'algorithme de Box-Muller [12]. Cette méthode se base sur une transformation en coordonnées polaires pour simuler une loi uniforme centrée réduite. Ensuite une simple transformation permet de simuler n'importe quelle loi normale $X \rightsquigarrow \mathcal{N}(\mu, \sigma)$ à partir d'une loi normale centrée réduite $Z \rightsquigarrow \mathcal{N}(0, 1)$:

$$X = \mu + \sigma \times Z.$$

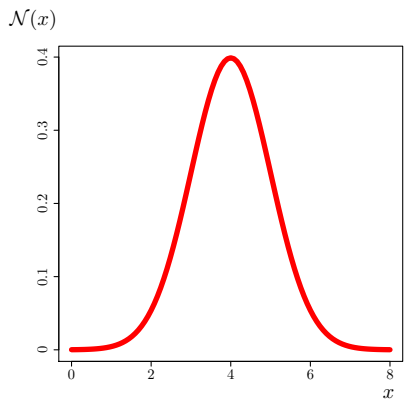


FIGURE 4.11 – Fonction de densité d'une loi normale $\mathcal{N}(\mu = 4, \sigma = 1)$.

4.3.1.7 Loi log-normale

Une variable aléatoire X suit une loi Log-Normale $\mathcal{N}(\mu, \sigma)$ (figure 4.12) si la variable $Y = \ln(X)$ suit une loi normale d'espérance μ et de variance σ.

La simulation d'une telle loi se base également sur l'algorithme de Box-Muller pour la version centrée réduite $Z(\mu = 0, \sigma = 1)$ et puis une simple transformation généralise à toutes les valeurs de μ et de σ :

$$X = exp(\mu + \sigma \times Z)$$

Les lois binomiale, exponentielle, normale et log-normale sont celles que nous observons le plus fréquemment sur les modèles. Elles concernent généralement les distributions des degrés des instances de classes (donc les références).

4.3.2 Intégration à l'approche de génération de modèles

Des échantillons de lois de probabilités usuelles connues sur un méta-modèle donné sont injectés au processus de génération de modèles pour améliorer la pertinence et la

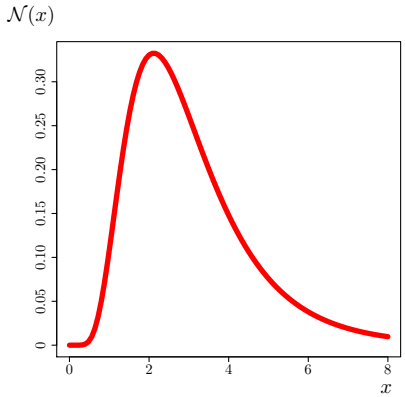


FIGURE 4.12 – Fonction de densité de d'une loi log-normale $\mathcal{N}(\mu = 1, \sigma = 0.5)$.

vraisemblance des modèles générés automatiquement. Nous avons dans un premier temps intégré cet aspect probabiliste à deux caractéristiques des méta-modèles.

1. **Cardinalités des références.** En effet, comme nous le disons plus haut, générer des modèles où les instances d'une classe sont toutes liées au même nombre d'instances d'une autre classe n'est pas très intéressant d'un point de vue pertinence du modèle. Le but de l'ajout de lois de probabilité sur les cardinalités des références est de diversifier les degrés des instances de classes des modèles.
2. **Valeurs d'attributs.** Certains attributs sont particulièrement importants pour le modèle. Leur valeur ne doit pas être choisie de manière arbitraire. Par exemple, les visibilités des attributs ou des méthodes sont très importantes dans un langage de programmation orienté objets. Des lois de distribution sont utilisées pour guider l'instanciation de certains attributs des méta-modèles.

4.3.2.1 Choix de ces lois

Les lois de probabilités usuelles utilisées par notre approche sont données par l'utilisateur et peuvent avoir deux origines différentes :

1. **Connaissance théorique d'un domaine.** Elles peuvent être liées à une connaissance théorique d'un domaine donné. Par exemple, dans l'étude de cas en section 4.5, des lois de probabilités théoriques concernant les degrés des nœuds des graphes générés sont données par l'expert. Le but ici est de générer automatiquement un corpus de modèles respectant des caractéristiques théoriques d'un domaine car il y a un manque de données réels.
2. **Connaissance empirique d'un domaine.** Quand beaucoup de modèles réels existent, ils peuvent être utilisés pour déduire des lois de probabilités liées à des métriques du domaine. Par exemple, dans la section 4.4 nous utilisons des programmes java réels

pour déduire des lois de probabilités en relation avec des métriques de programmation orientée objets. Dans ce cas, le but est de générer des modèles réalistes (proches des modèles réels) car l'acquisition de nouveaux modèles par un autre moyen est d'une manière ou d'une autre coûteuse.

3. **Souhait de l'expert.** Les lois de probabilités peuvent aussi dépendre d'une utilisation qui sera faite des modèles générés ; comme par exemple vouloir tester des caractéristiques qu'on ne retrouve pas dans les modèles réels. Dans ce cas, l'expert choisira lui même les lois à injecter au générateur.

4.3.2.2 À la recherche de la distribution théorique adéquate

Lorsqu'un échantillon est obtenu de façon empirique, il est primordial de trouver la loi de probabilité qui s'en approche le plus possible et surtout les paramètres adéquats. Il existe beaucoup de travaux sur l'approximation de lois de probabilités théoriques. Nous pouvons par exemple citer le livre de *Karian et Dudewicz* sur la question [61].

Nous proposons ici une méthode semi-automatique utilisant une fonction prédéfinie dans le langage R [1] pour inférer les lois de probabilités correspondant à des échantillons empiriques et leurs paramètres.

1. **Choisir la bonne distribution.** Nous devons d'abord choisir parmi les distributions les plus connues celle dont la forme se rapproche le plus de l'échantillon à analyser. Par exemple, il est aisé d'observer que la forme de l'histogramme de la figure 4.14 se rapproche d'une loi log-normale. La figure 4.13 montre certaines des lois de probabilités les connues.
2. **Déduire les bons paramètres.** Une fois la distribution adéquate choisie, il est nécessaire d'utiliser un outil d'approximation pour inférer les paramètres de la distribution théorique. Nous choisissons d'utiliser la fonction `fitdistr()` de R [61]. Cette fonction prend en paramètre l'échantillon à analyser et la loi de probabilité choisie et calcule les paramètres de cette loi qui donnent la meilleure adéquation. La courbe de la figure 4.14 dessine la loi log-normale calculée par R ($\mathcal{N}(\mu = 0.744, \sigma = 0.476)$).

4.3.2.3 Ajout à l'approche

La génération de modèles pertinents basée sur la simulation de lois de probabilités nécessite une étape supplémentaire par rapport à l'approche de génération de base. La figure 4.15 montre les étapes principales de $\mathcal{G}$ЯIMM($\mathcal{G}$enerating ЯandomIzed and Relevant Instances of Meta-Models). Les distributions de certains liens et certains attributs doivent être données. Elles servent de données d'entrée au simulateur de probabilités. Ce dernier fournit des échantillons concernant les liens qui seront ajoutés au CSP avant sa résolution. Après la résolution du CSP par le solveur de contraintes, les valeurs de certains attributs sont affectées suivant les distributions données en entrée.

1. Projet R: `https://www.r-project.org`

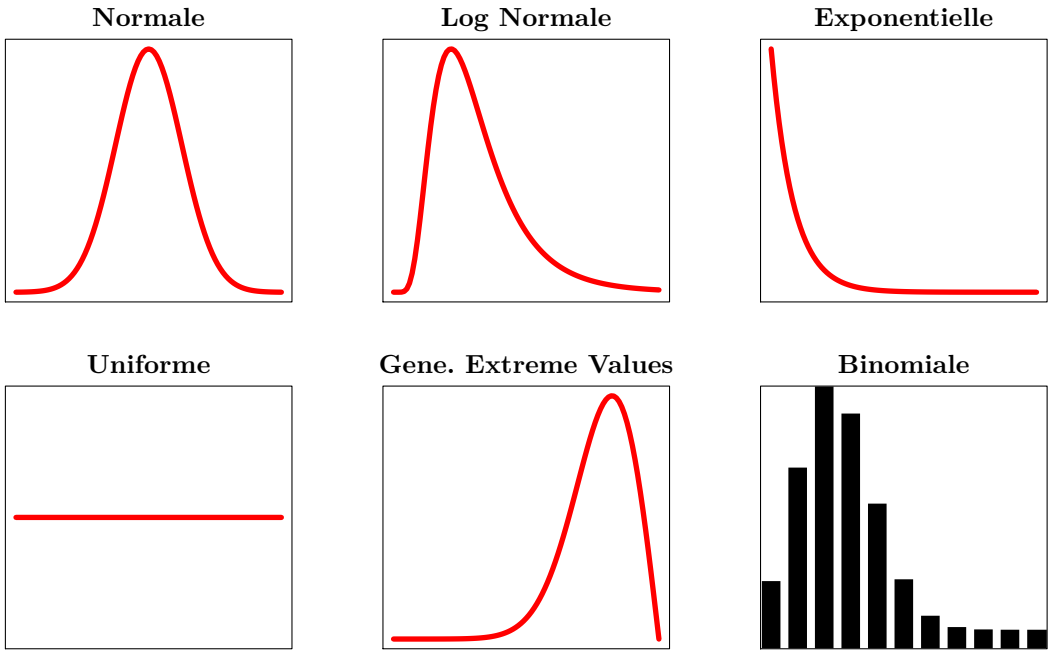


FIGURE 4.13 – Les courbes et histogrammes de quelques lois de probabilités usuelles.

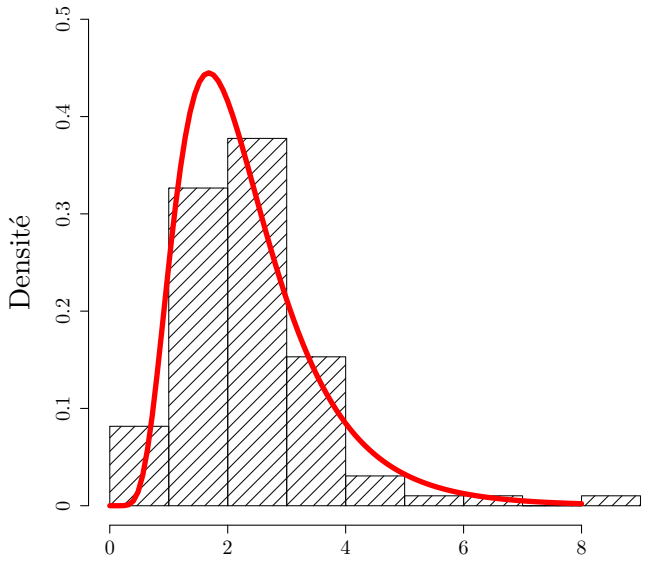


FIGURE 4.14 – Superposition d'un histogramme obtenu empiriquement et de la loi log-Normale théorique correspondante inférée (courbe rouge) par la fonction `fitdistr()` de R.

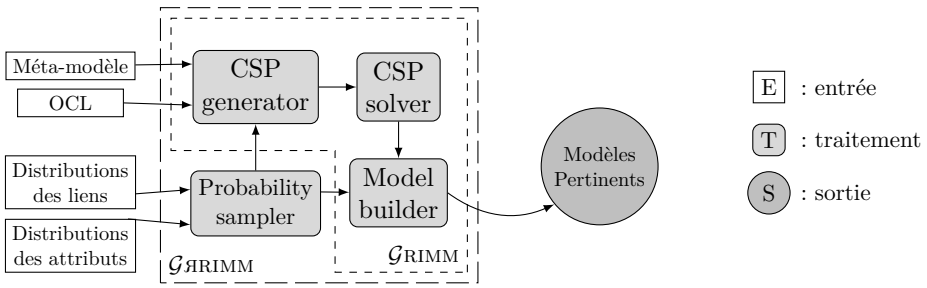


FIGURE 4.15 – Processus de l'outil GRIMM.

4.4 Étude de cas : génération de programmes JAVA

4.4.1 Introduction

Dans cette section nous appliquons l'approche de génération de modèles pertinents par simulation de lois de probabilités usuelles dans le but de générer automatiquement des squelettes de programmes java. L'objectif de cette étude de cas est double :

1. Dans un premier temps, valider l'approche de génération de modèles pertinents par simulation de lois de probabilités sur un cas concret. Ceci se fera par la comparaison des modèles générés avec des projets java réels récoltés puis analysés.
2. De manière plus générale, monter qu'il est possible, avec notre approche, de générer des modèles respectant n'importe quelle métrique donnée par l'utilisateur à condition qu'elle puisse être décrite par une loi de probabilité usuelle. Ces modèles peuvent ensuite servir à tester des constructions qu'il est difficile ou impossible à trouver dans les corpus de modèles réels.

Cette étude se structure comme suit. D'abord, nous récoltons et analysons des projets java réels selon des métriques de code connues. Ensuite, nous donnons le protocole qui permet de générer des modèles ayant les mêmes caractéristiques métriques que les projets réels. Enfin une comparaison entre les deux corpus de projets selon ces mêmes métriques et la discussion de cette comparaison sont données.

Note

*N*ous avons choisi java pour la facilité à trouver des projets réels de tailles diverses et surtout pour la facilité à trouver des outils d'analyse de métriques.

4.4.2 Étude de projets java réels

Choix des projets

Nous avons récolté 200 projets java de deux origines différentes :

— 100 projets de grande taille venant d'un corpus connu (Qualitas corpus [2]).

— 100 projets de taille moyenne venant de github.

Les diagrammes à moustaches de la figure 4.16 montrent le nombre de classes par projet des deux jeu de données de projets java utilisés. Nous pouvons remarquer que *Qualitas corpus* regroupe des projets de très grande tailles. Ils contiennent 1019 classes en moyenne et jusqu'à 7026 classes pour les plus grands projets. Par contre, les projets issus de github ne contiennent en moyenne que 119 classes et au maximum 1098.

2. Qualitas corpus: `http://qualitascorpus.com/docs/catalogue/20130901/index.html`

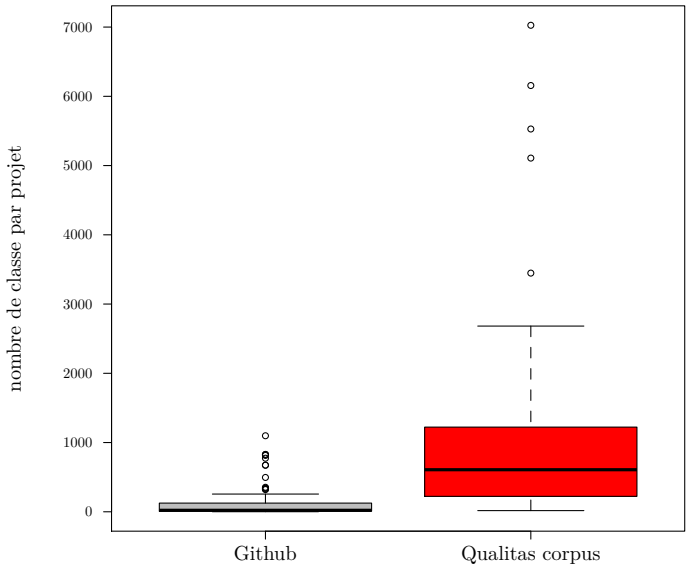


FIGURE 4.16 – Diagrammes à moustaches donnant le nombre de classes par projet des deux becnhmarks de projets java utilisés.

Choix des métriques

Pour analyser ces projets, nous avons choisi un ensemble de métriques de code objet connues. Ce métriques sont pour la plupart issues du livre de Henderson-Sellers sur les métriques [55]. Il s'agit par exemple du :

— Nombre de packages par projet.

— Nombre de classes par package.

— Nombre d'interfaces par package.

— Nombre de sous-classe par classe

— Nombre d'attributs par classe.

— Nombre de méthodes par classe.

— Nombre de constructeurs par classe.

— Nombre de paramètres par méthode.

En plus de ces métriques, nous avons également analysé les fréquences d'apparition des différentes visibilités (public, private, package et protected) pour les attributs et les méthodes.

Récolte des métriques

Nous avons, ensuite, utilisé l'outil open-source `metrics`[3] pour calculer les métriques sur les 200 projets réels récoltés.

Pour chaque projet, l'outil compte entre autres le nombre de packages, de classe, d'attributs, d'interfaces. mais également les visibilités des attributs et des méthodes.

Lois de probabilités inférées

Ces métriques sont utilisées pour inférer les lois de probabilités que suivent certains éléments d'un projet java. Pour cela, nous dessinons avec R[4] un histogramme pour chacune des métriques. Ensuite, la technique d'inférence de lois de probabilités est utilisée pour estimer les bons paramètres pour chaque métrique.

La table 4.2 donne la loi de probabilité correspondant à chaque métrique. Pour les visibilités des attributs et méthodes, les arguments donnés sont les pourcentages d'attributs ou méthodes de visibilité public, private, protected et package (dans l'ordre).

3. Metrics tool: `http://metrics.sourceforge.net`
4. R softaware: `https://www.r-project.org/`

Métrique	$\rightsquigarrow$	**Arguments**	**Loi**
Class/Package	$\rightsquigarrow$	$\varepsilon(\frac{1}{8.387})$	Exponentielle
Méthodes/Type	$\rightsquigarrow$	$\mathcal{N}(1.810, 0.323)$	Log-normale
Attributs/Type	$\rightsquigarrow$	$\mathcal{N}(0.744, 0.476)$	Log-Normale
Constructeurs/Type	$\rightsquigarrow$	$\mathcal{N}(0.73, 0.26)$	Normale
Sous-Classes/Classe	$\rightsquigarrow$	$\varepsilon(\frac{1}{0.22})$	Exponentielle
Interface/Package	$\rightsquigarrow$	$\varepsilon(\frac{1}{8.001})$	Exponentielle
Paramètres/Méthodes	$\rightsquigarrow$	$\mathcal{N}(0.87, 0.25)$	Normale
Visibilité attributs	$\rightsquigarrow$	$(20.41\%, 54.56\%, 14.46\%, 10.66\%)$	Loi discrète
Visibilité méthodes	$\rightsquigarrow$	$(77.15\%, 10.42\%, 6.06\%, 6.35\%)$	Loi discrète

TABLE 4.2 – Les lois de probabilités correspondant aux métriques de code choisies

4.4.3 Génération des projets

La figure 4.17 montre le méta-modèle que nous avons conçu pour construire des squelettes de programmes java. Nous avons conçu ce méta-modèle allégé pour ne générer uniquement les constructions du langage java pour lesquelles nous disposons de métriques.Voici la liste des constructions possibles offertes par ce méta-modèle :

— Un projet java se compose de packages.

— Le projet intègre des types primitifs.

— Chaque package peut contenir un certain nombre de classes et d'interfaces.

— Une classe peut hériter d'une classe et implémenter plusieurs interfaces.

— Les classes et les interfaces contiennent des méthodes et des attributs (Field).

— Une méthode possède des paramètres et un type de retour.

— Un attribut possède un type (Objet ou bien primitif).

— Les attributs et les méthodes ont une visibilité.

Ce méta-modèle est accompagné d'un certain nombre de contraintes OCL. Voici certaines de ces contraintes :

— Les packages du projet doivent porter des noms différents (de même pour tous les autres éléments).

> `Context` Project `inv` : ownedpackages→`forall`(p_1, p_2 Package | $p_1 \neq p_2$ `implies` p_1.name $\neq p_2$.name) ◁ OCL

— Une classe ne peut pas hériter d'elle-même.

> `Context` Class `inv` : super $\neq$ `self` ◁ OCL

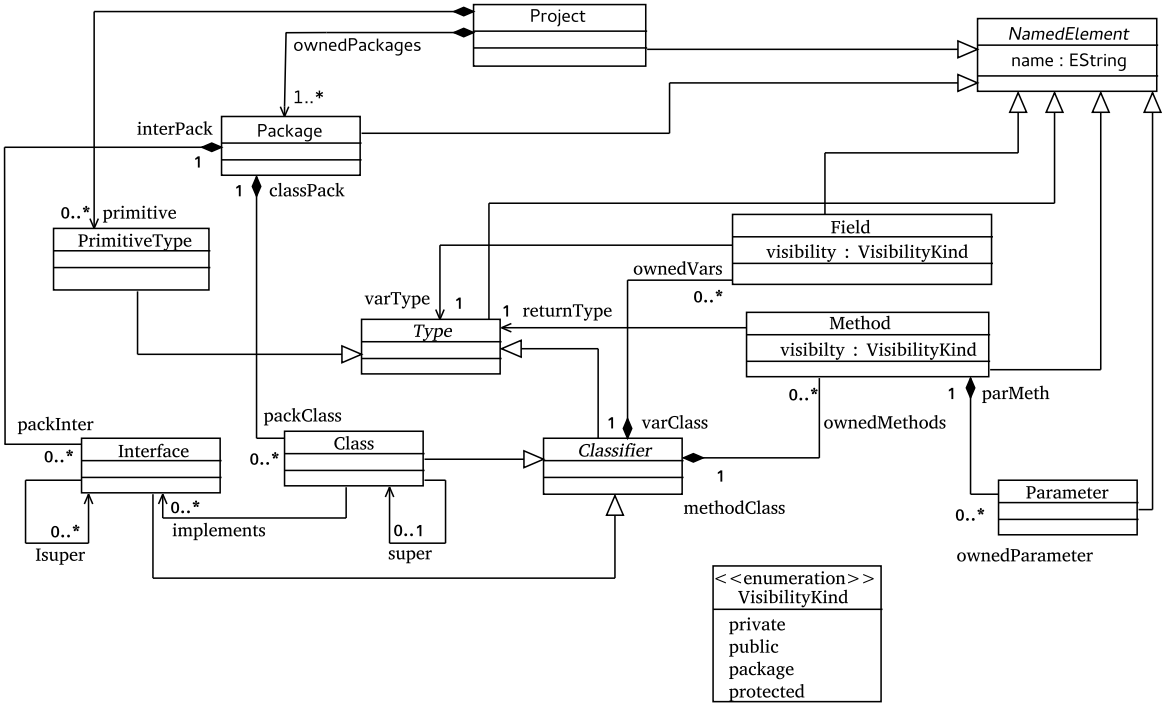


FIGURE 4.17 – Méta-modèle pour la construction de squelettes de programmes java.

— Une interface ne peut pas s'implémenter elle-même.

`Context` Interface `inv` : implements→`excludes(self)` ◁ OCL

Les lois de probabilités inférées plus haut sont ensuite injectées au CSP au même titre que l'encodage du méta-modèle et des contraintes OCL pour générer des projets java automatiquement. 300 projets java de différentes tailles sont ainsi générés avec trois configurations différentes :

1. Une version où seul le méta-modèle est pris en compte au moment de la génération des projets.
2. Une version où les contraintes OCL sont également encodées dans le CSP.
3. une version dans laquelle les lois de probabilités sont aussi prises en compte (+OCL).

4.4.4 Résultats

L'évaluation de l'approche se fait par la comparaison des distributions des métriques mesurées sur les projets réels et celles obtenues sur les projets générés automatiquement. La figure 4.18 montre le protocole suivi pour cette comparaison. Les figures 4.19 et 4.20 montrent des histogrammes de distributions de probabilité pour les quatre corpus. Cela concerne le nombre de constructeurs par classe et les visibilités des attributs des classes.

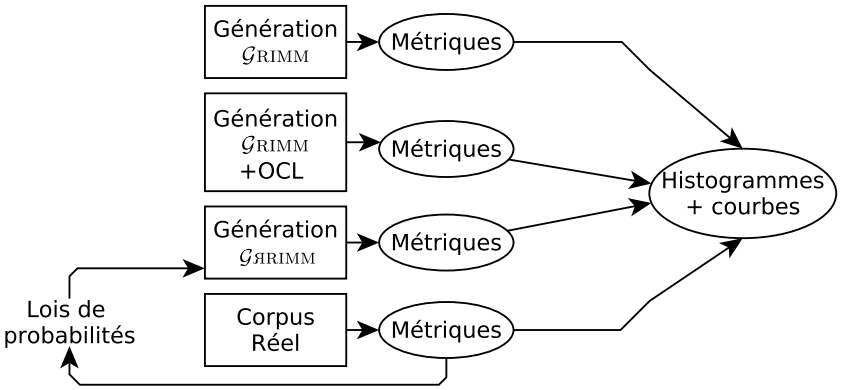


FIGURE 4.18 – Processus de comparaison des projets java générés et des projets réels.

4.4.5 Discussion

Lorsque les modèles sont générés par les versions de $\mathcal{G}$RIMM prenant en compte uniquement le méta-modèle et/ou ses contraintes OCL, les distributions sont très quelconques et

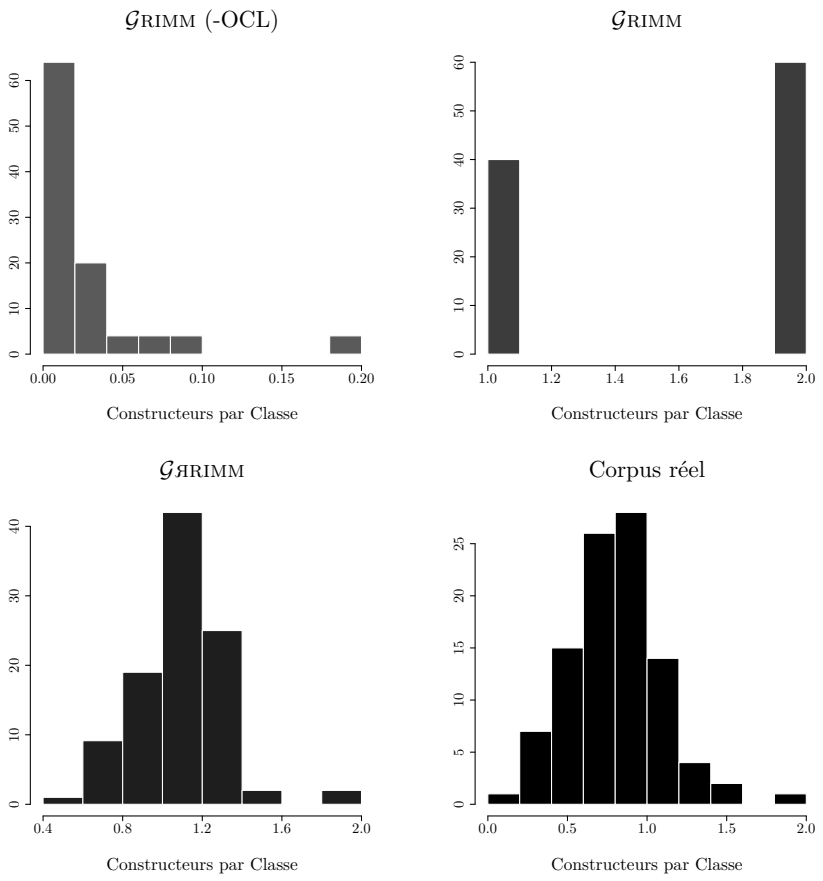


FIGURE 4.19 – Comparaison des distributions du nombre de constructeurs par classe entre les projets java générés et réels.

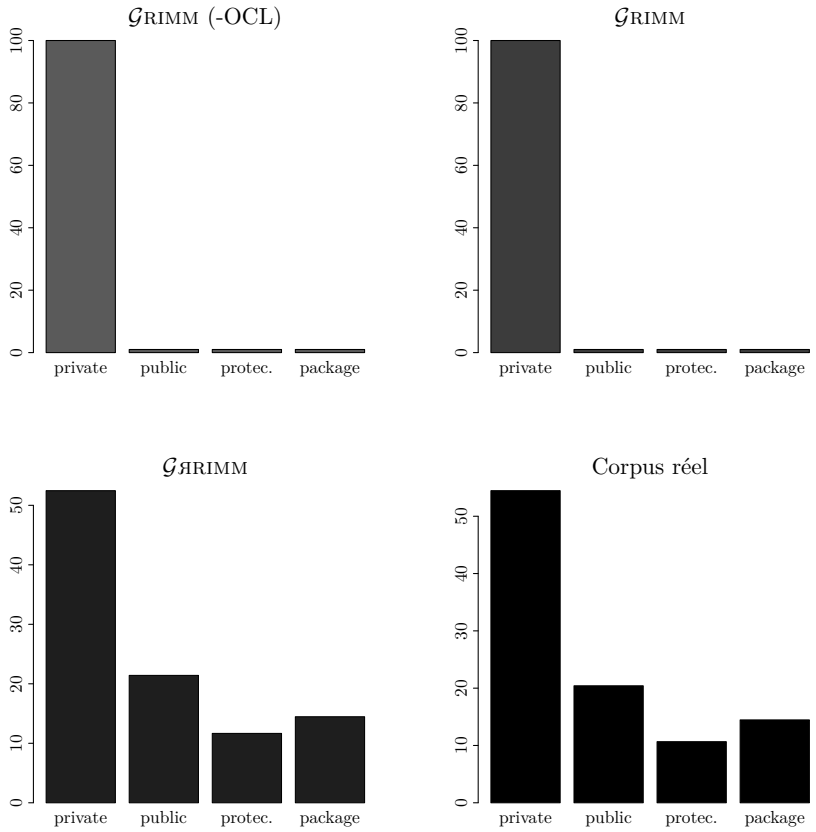


FIGURE 4.20 – Comparaison des distributions des visibilités des attributs.

ne ressemblent à aucune loi usuelle. Elles sont également très éloignées des distributions des projets réels. Ceci est valable pour toutes les métriques choisies.

Par contre, injecter des lois de probabilités en amont et en aval du processus de résolution du CSP augmente considérablement la vraisemblance des modèles générés. En effet, les histogrammes dans ce cas sont très proches des projets java réels.

La distribution obtenue pour le nombre de constructeurs par classe correspond à une loi normale de mêmes espérance et écart-type que la distribution réelle. Il en est de même pour les visibilités des attributs. Ainsi, les fréquences observées pour chaque visibilité sont égales aux pourcentages relevés sur les projets réels.

Concernant les autres métriques de code, les projets générés approchent dans tous les cas le bon type de distribution. Néanmoins, un faible écart est parfois observé sur les paramètres de la loi.

4.5 Étude de cas : génération de Graphes de Scaffold

4.5.1 Introduction

En biologie, le séquençage puis l'assemblage du génome d'une espèce produit une grande quantité de petites séquences (de quelques dizaines à plusieurs millions de séquences). Les techniques de reconstruction consistent ensuite à assembler ces séquences appelées contigs dans un ordre précis pour construire le génome complet de l'espèce.

Une des étapes s'appelle le scaffolding. Elle consiste à modéliser les séquences d'ADN (maintenant appelées contigs) sous la forme d'un graphe de scaffold. La linéarisation de ce graphe produira le génome complet [23, 24, 105].

Un graphe de scaffold se compose d'un certain nombre de contigs. Chaque contig comporte lui-même deux nœuds et une arête de contig reliant ses deux nœuds. D'autres arêtes, dites de scaffolding, relient les nœuds de différents contigs. Les arêtes de scaffolding sont pondérées par des entiers. La figure 4.21 montre un graphe de scaffold simple composé de 6 contigs et de 8 arêtes de scaffolding.

Tester les algorithmes de scaffolding nécessite en entrée des graphes de scaffold de différentes tailles et caractéristiques. La génération automatique de ces données a été envisagée car :

— L'obtention des graphes réels est coûteuse en temps et en argent. Elle nécessite un séquençage effectué dans un laboratoire de biologie si on s'intéresse à de nouvelles espèces. L'obtention de graphes à partir de données existantes nécessite un long calcul car les données sont de tailles conséquentes (plusieurs Go).

— Deux éléments importants d'un graphe de scaffold suivent des lois de probabilités usuelles dont les paramètres sont facilement calculables. Il s'agit des distributions des degrés des nœuds et des poids des arcs de scaffolding.

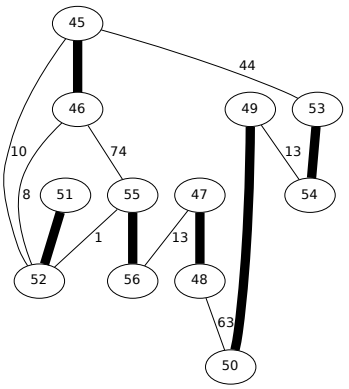


FIGURE 4.21 – Un graphe de scaffold contenant 6 contigs. Les arêtes de contigs sont en gras et les arêtes de scaffolding sont pondérées.

4.5.2 Génération de graphes

Dans la suite de cette section, nous allons décrire une approche dirigée par les modèles pour la génération de graphes de scaffold. La première étape de cette approche consiste à écrire un méta-modèle pour représenter les graphes de scaffold.

La figure 4.22 montre un méta-modèle servant à la construction de graphes de scaffold. Voici les caractéristiques d'un graphe de scaffold telles que définies par ce méta-modèle :

— Un graphe de scaffold se compose de contigs et d'arêtes.

— Les arêtes portent un poids.

— Un contig contient deux nœuds et une arête de contig.

— L'arête de contig relie entre eux les deux nœuds du contig.

— Les nœuds appartenant à deux contigs différents peuvent être liés par une arête de scaffolding.

Deux lois de probabilités usuelles concernant les graphes de scaffold sont ensuite injectées à notre outil pour améliorer la pertinence et la vraisemblance des graphes générés automatiquement.

1. **Distribution des degrés des nœuds.** Le degré des nœuds des graphes de scaffold suit une loi exponentielle. La moyenne de cette loi est soit directement donnée par l'utilisateur, ou bien calculée en fonction de la densité désirée du graphe à générer ou du nombre de nœuds et d'arcs que ce graphe contiendra.
2. **Les poids des arcs de scaffolding.** Les poids assignés aux arêtes de scaffolding suivent une loi uniforme dont les paramètres sont donnés par l'utilisateur.

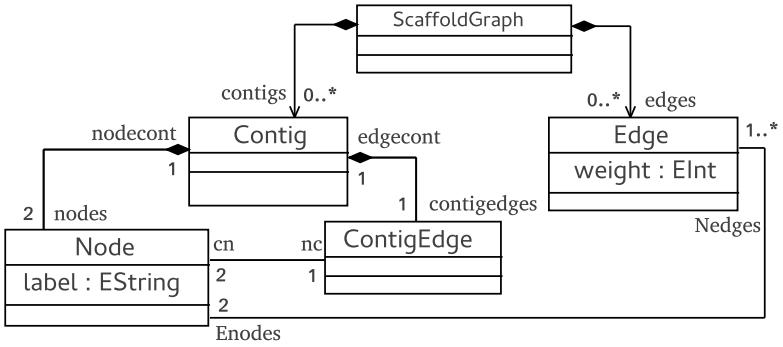


FIGURE 4.22 – Méta-modèle pour la constructions de graphes de scaffold.

Les auteurs dans [105] présentent des graphes de scaffold réels ainsi que des mesures prises sur ceux-ci pour les caractériser. Nous reprenons ces données et générons avec notre outil des graphes équivalents (en taille) aux graphes réels décrits dans [105]. En effet, pour chaque graphe de scaffold réel, 60 graphes de même taille sont générés avec deux versions de notre outil :

1. Une version de l'outil où seul le méta-modèle est encodé en CSP ($\mathcal{G}$RIMM).
2. Une deuxième version où les lois de probabilités sont injectées au générateur ($\mathcal{G}\mathcal{R}$RIMM).

Une fois les modèles (graphes de scaffold) générés, un outil d'analyse de graphes de scafolld développé par Weller et al. dans [105] est utilisé pour les analyser.

4.5.3 Comparaison & Résultats

La figure 4.23 montre 3 graphes de scaffold, l'un est réel et les deux autre sont générés respectivement par les deux versions de notre outil. Ces graphes sont tous les trois de même taille (nombre de nœuds).

La figure 4.24 montre les distributions des degrés des nœuds de trois graphes de scaffold. Un graphe réel et deux graphes générés.

La table 4.3 montre les résultats des mesures de graphe prises sur les graphes générés comparées avec celles prises sur les graphes de scaffold réels. h-index= v indique qu'il existe v nœuds avec un degré supérieur ou égal à v.

4.5.4 Discussion

D'abord sur la forme générale des graphes, nous observons une proximité entre le graphe réel et les graphes générés par injection de lois de probabilités. En effet, les deux graphes possèdent un nombre de nœuds isolés proche de 0. De plus, ils possèdent tous les deux une grande diversité dans leurs degrés des nœuds. Par contre, le graphe généré par la version ne prenant en compte que le méta-modèle présente quelques problèmes :

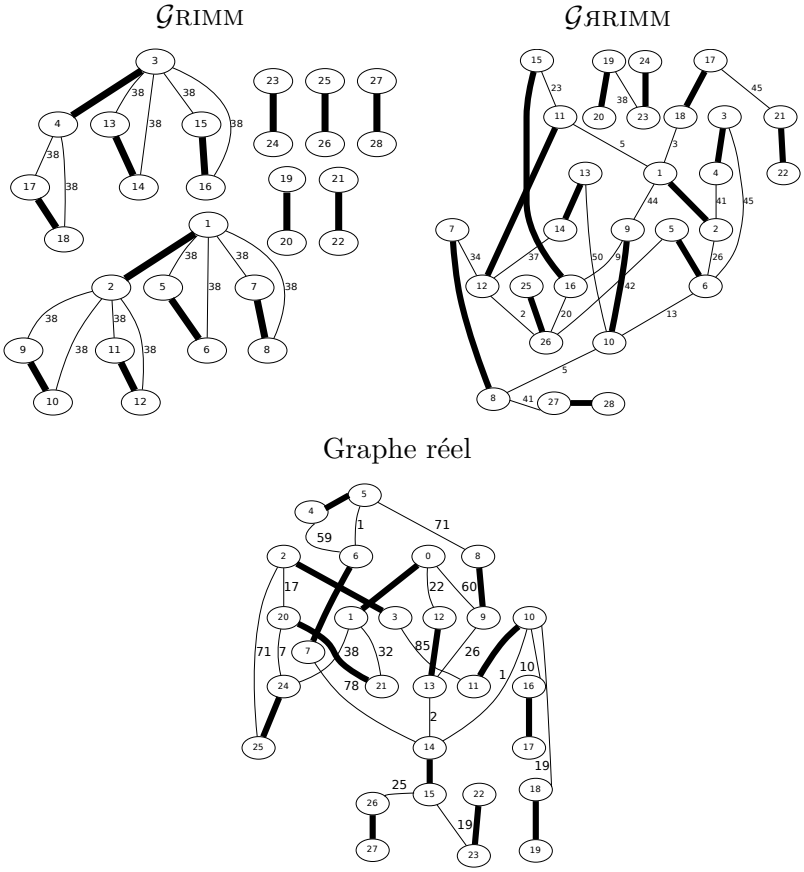


FIGURE 4.23 – Trois graphes de scaffold correspondant à la même espèce (génome mitochondrial du papillon monarque). Les arêtes en gras représentent les arêtes de contigs.

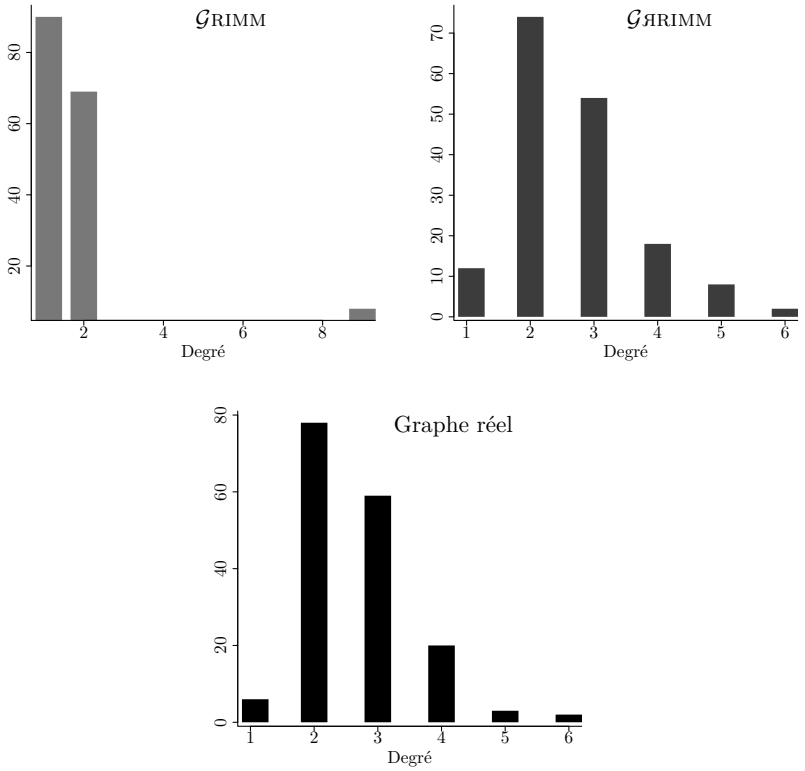


FIGURE 4.24 – Comparaison de la distribution des degrés des nœuds entre un graphe réel et ses équivalents générés (168 nœuds et 223 arcs).

Graphe			Métriques					
			Génération GRIMM		Génération GRRIMM		Graphes réels	
Espèce	nœuds	arêtes	degré min/max	h-index	degré min/max	h-index	degré min/max	h-index
monarch	28	33	1/9	3	1/ 4.6	4.06	1/ 4	4
ebola	34	43	1/9	3	1/ 4.83	4.60	1/ 5	4
rice	168	223	1/9	8	1/ 6.03	5.93	1/ 6	5
sacchr3	592	823	1/9	10	1/ 7	6.76	1/ 7	6
sacchr12	1778	2411	–	–	1/ 7.53	7	1/10	7
lactobacillus	3796	5233	–	–	1/ 8.06	7.8	1/12	8
pandora	4092	6722	–	–	1/ 8.23	7.96	1/ 7	7
anthrax	8110	11013	–	–	1/ 8.3	8.03	1/ 7	7
gloeobacter	9034	12402	–	–	1/ 8.46	8	1/12	8
pseudomonas	10496	14334	–	–	1/ 8.43	8	1/ 9	8
anopheles	84090	113497	–	–	1/ 8.96	9	1/ 51	12

TABLE 4.3 – Comparaison entre 60 graphes de scaffold générés et les graphes réels correspondant à chaque espèce selon quelques métriques de graphes.

— Un nombre de noeuds isolés important.
— Peu de diversité des degrés des nœuds.
— Des paquets de nœuds reliés entre eux de la même manière (motifs récurrents).

La distribution des degrés du graphe généré avec $\mathcal{G}$ЯRIMM coïncide parfaitement avec la distribution du graphe réel, tandis que la distribution du troisième graphe est plus quelconque et éloignée des deux autres. Ceci est le cas pour tous les graphes générés. Les graphes générés par $\mathcal{G}$ЯRIMM possèdent tous une distribution identique aux graphes réels.

Note

*N*ous avons choisi de tracer la distribution d'un seul graphe à la fois car elle est plus précise et expressive que la moyenne des distributions. La moyenne aurait faussé la bonne perception des résultats. Néanmoins, la table 4.3 montre les moyennes des degrés minimaux et maximaux et nous pouvons aussi voir qu'ils sont très proches des graphes réels dans le cas de l'injection des probabilités.

Les moyennes des degrés min/max et h-index des graphes générés par injection de lois de probabilités usuelles sont proches des valeurs mesurées sur les graphes de scaffold réels. Seul le dernier graphe (84.090 nœuds) possède des chiffres qui sont éloignés, voire très éloignés des valeurs de ces équivalents générés. Ceci est dû à la taille du graphe et à son degré max très élevé et qui ne correspond pas à la valeur de la moyenne donnée par sa densité.

4.6 Conclusion

Ce chapitre a abordé le volet de la pertinence et de la vraisemblance des modèles générés automatiquement (figure 4.25). D'abord, nous avons présenté une technique issue de la programmation par contraintes pour casser les symétries et améliorer la vraisemblance des modèles. Casser les symétries permet d'obtenir des modèles plus connectés et qui contiennent moins d'éléments isolés. Les modèles générés en utilisant cette technique sont évalués par comparaison avec des modèles réels que nous avons récoltés et selon quelques mesures de graphes.

Ensuite, nous avons décrit une approche pour améliorer la pertinence des modèles. Elle se base sur la simulation de lois de probabilités usuelles. Les lois concernent certains éléments du méta-modèle et sont soit données par l'utilisateur ou bien inférées empiriquement.

Cette approche est évaluée par deux études de cas. La première est issue de la programmation orientée objet et concerne la génération automatique de squelettes de programmes java selon quelques métriques de code. La deuxième étude vient de bioinformatique. Elle consiste à générer des graphes de scaffold qui sont utilisés pour la production des génomes.

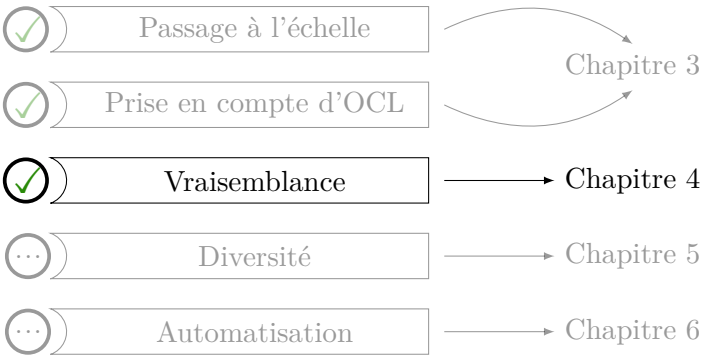


FIGURE 4.25 – Caractéristiques de la génération de modèles abordées par le chapitre 4.

Ces données peuvent être utiles pour tester les limites des algorithmes et des transformations de modèles. Par exemple, il est intéressant de générer des programmes avec des caractéristiques extrêmes : structures arborescentes horizontale ou bien verticale pour l'héritage ou les contenus des packages, ou encore, des caractéristiques difficiles à retrouver dans les données réelles comme des graphes de scaffold de grande densité.

Chapitre 5

Distances entre modèles et diversité des solutions

> L'abondance ne serait un progrès que si elle accompagnait de la diversité et de la qualité. Or les programmes tendent de plus en plus à se ressembler. Abondance est progressivement synonyme d'uniformité.
>
> Jacques Toubon

Synopsis

Préambule

Ce chapitre aborde le volet de la diversité des modèles générés automatiquement. À la section 5.2, nous présentons plusieurs distances qui s'inspirent de distances connues et que nous avons adaptées aux modèles. La section 5.3 utilise des techniques de clustering de matrices de distances pour sélectionner les modèles les plus éloignés parmi un ensemble de départ. Enfin, la section 5.4 montre une technique basée sur l'algorithmique génétique pour améliorer la diversité d'un échantillon de modèles en utilisant les distances entre modèles et le clustering.

5.1 Introduction

Lorsqu'on génère automatiquement des ensembles contenant beaucoup de modèles conformes au même méta-modèle avec notre approche, nous nous apercevons que certains d'entre eux se ressemblent. Pour des raisons pratiques et d'utilité, il n'est donc ni nécessaire ni souhaitable de garder deux modèles qui se ressemblent. Le but est de choisir les modèles les plus représentatifs de l'ensemble total de modèles générés. Comme nous pouvons le voir sur le schéma de la figure 5.1, la comparaison des modèles montre qu'ils se regroupent en paquets tel que les modèles de chaque paquet sont proches les uns des autres. Un élément sera pioché dans chaque paquet pour ne garder que les modèles les plus représentatifs de l'ensemble total.

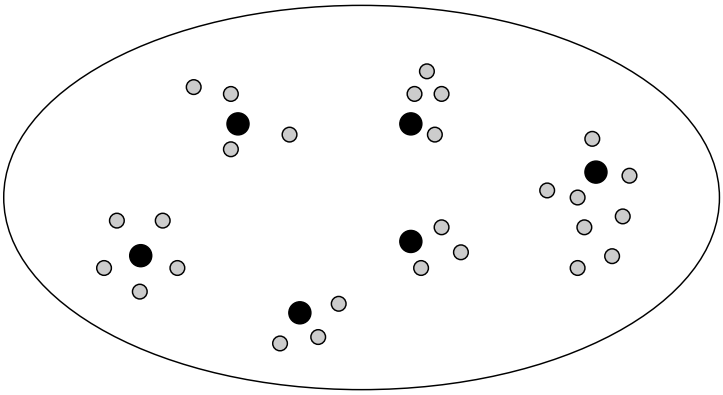


FIGURE 5.1 – Schématisation du principe de sélection de modèles les plus diversifiés. Les points représentent des modèles et deux points sont proches si les modèles qu'ils représentent le sont également. Les points noirs sont les modèles sélectionnés les plus représentatifs.

Mener à bien un tel processus requiert deux conditions nécessaires :

1. Trouver un moyen de comparaison entre deux modèles qui permettrait de mesurer la proximité entre les modèles d'un ensemble.
2. Mettre en place un système pour sélectionner des éléments représentatifs de l'ensemble en fonction du critère de comparaison précédent.

D'abord, nous comparerons deux modèles en utilisant des distances mathématiques et de graphes connues que nous avons adaptées aux modèles. Ces différentes distances seront présentées, étudiées et comparées entre elles selon différents critères notamment la complexité en temps, pour des raisons de passage à l'échelle. En effet, lorsque les modèles d'un ensemble sont comparés entre eux, les distances doivent être calculées pour tous les modèles deux à deux. Une matrice symétrique est ainsi obtenue (table 5.2).

distance	m_1	m_2	m_3	m_4	m_5	m_6
m_1	0	10	3	2	4	12
m_2	10	0	7	2	1	3
m_3	3	7	0	14	3	3
m_4	2	2	14	0	5	6
m_5	4	1	3	5	0	10
m_6	12	3	3	6	10	0

TABLE 5.1 – Matrice de distances entre 6 modèles.

Ensuite, nous utiliserons des techniques de clustering de matrices de distances pour évaluer l'éloignement des modèles de tout l'ensemble en même temps et pour découper cet ensemble en clusters. Enfin, l'élément le plus représentatif de chaque cluster est sélectionné.

5.2 Des distances de graphes adaptées aux modèles

5.2.1 Comparaison de modèles

Comparer des modèles logiciels et mesurer leur similarité ou leur différence est une problématique récurrente et beaucoup étudiée. Il existe plusieurs approches ayant traité la question. Kondrad Voigt a réalisé un comparatif où sont détaillées les approches les plus connues dans sa thèse [104]. Voici deux exemples d'approches de comparaison de méta-modèles dans le but de les apparier :

— Voigt et Heinze proposent dans [103] une approche d'appariement entre deux méta-modèles basée sur la distance d'édition de graphes planaires. Le but de l'approche est de comparer la structure de deux méta-modèles (l'entrée et la sortie d'une transformation de modèles) via la distance d'édition de graphe pour trouver le meilleur appariement entre les deux. La finalité de l'approche est d'assister le développement de la transformation de modèle impliquant les deux méta-modèles précédents.

— Falleri et al. présentent dans [33] une approche d'appariement entre deux méta-modèles basée sur l'algorithme Similarity Flooding [71]. Le but de l'approche est semblable à la précédente. Il s'agit de trouver un mapping entre les éléments de deux méta-modèles, l'un est le méta-modèle d'entrée d'une transformation de modèle et l'autre le méta-modèle de sortie.

D'autres approches et outils avec des objectifs similaires existent.

Il est à noter que toutes ces approches comparent deux méta-modèles uniquement, dans le but de trouver un appariement entre eux sans mesurer leur similarité ou leur différence. Dans notre cas le but est de trouver des métriques pour comparer un ensemble contenant beaucoup de modèles et de mesurer cette différence, pour ensuite utiliser cette mesure dans le but de sélectionner les modèles les plus diversifiés. De plus, les métriques que avons

adaptées aux modèles se focalisent en priorité sur la comparaison structurelle des modèles et non sur de l'appariement sémantique des labels (noms des classes et des attributs).

5.2.2 Distance de graphes

Voici la définition de la distance telle que donnée par le Petit Robert 2016 :

Définition.

Distance Longueur qui sépare une chose d'une autre.

Une distance est pour ainsi dire la mesure du degré d'éloignement d'un objet par rapport à un autre. Autrement dit, c'est la quantité de déplacement ou de modification qu'il faut apporter à l'un des deux objets pour atteindre ou obtenir l'autre.

Au sens mathématique une distance est définie par Adolf Lindenbaum dans [67] par :

Definition 7 *(***Distance***) une application d qui associe à tout couple d'éléments d'un ensemble E un réel positif.*

$$\begin{aligned} d: \quad & E \times E \to \mathbb{R}^+ \\ & (a,b) \mapsto d(a,b) \end{aligned}$$

Une distance doit également vérifier les trois propriétés suivantes :

— $\forall (a,b) \in E^2, d(a,b) = 0 \Rightarrow a = b$ (séparation).

— $\forall (a,b) \in E^2, d(a,b) = d(b,a)$ (symétrie).

— $\forall (a,b,c) \in E^3, d(a,c) \leq d(a,b) + d(b,c)$ (inégalité triangulaire).

Les distances de graphes les plus connues et notamment celles que nous avons utilisées et adaptées aux modèles respectent toutes la définition de distance mathématique et ses trois propriétés.

5.2.2.1 Matrice de distances

Comparer un ensemble de graphes et donc de modèles nécessite le calcul des distances pour tous les couples d'éléments de l'ensemble. Ces calculs donnent lieu à la création de matrices de distances (voir table 5.2). Les traitements qui seront apportés à cette matrice permettront par exemple dans notre cas de sélectionner les modèles les plus éloignés parmi un ensemble de modèles.

La matrice de distances est carrée, symétrique et à diagonale nulle. Cela aide à vérifier empiriquement qu'une distance vérifie bien les propriétés précédentes.

	m_1	m_2	$\dots$	m_i	$\dots$	m_n
m_1	0	$d(m_1, m_2)$	$\dots$	$d(m_1, m_i)$	$\dots$	$d(m_1, m_n)$
m_2	$d(m_1, m_2)$	0	$\dots$	$d(m_2, m_i)$	$\dots$	$d(m_2, m_n)$
$\vdots$	$\vdots$	$\dots$	$\ddots$	$\dots$	$\dots$	$\vdots$
m_i	$d(m_1, m_i)$	$d(m_2, m_i)$	$\dots$	0	$\dots$	$d(m_i, m_n)$
$\vdots$	$\vdots$	$\dots$	$\dots$	$\dots$	$\ddots$	$\vdots$
m_n	$d(m_1, m_n)$	$d(m_2, m_n)$	$\dots$	$\dots$	$d(m_i, m_n)$	0

TABLE 5.2 – Matrice de distances entre modèles.

Dans la suite de ce chapitre, nous allons, dans un premier temps, présenter 3 distances entre modèles que nous avons développées en nous inspirant de distances mathématiques ou de graphe connues. Les distances originelles sont présentées, puis les transformations nécessaires aux modèles et les modifications apportées à chaque distance pour s'adapter à ceux-ci sont décrites et détaillées. Par la suite, les trois distances sont comparées selon différents critères.

5.2.3 Distance de Hamming

La distance de Hamming est une distance mathématique entre deux vecteurs. Elle a été introduite par Richard Hamming en 1950 dans [52]. Elle est à l'origine utilisée pour la détection d'erreurs puis la correction de codes binaires.

Definition 8 (Distance de Hamming) *Le nombre de coefficients qui diffèrent entre deux vecteurs x et y.*

Version adaptée aux modèles

Nous utilisons une version adaptée de cette distance pour comparer deux modèles, puis nous calculons la matrice de distance pour un ensemble de modèles. La première étape consiste à trouver une représentation vectorielle d'un modèle dans le but d'appliquer une version modifiée de la distance de Hamming et comparer deux modèles.

La représentation vectorielle d'un modèle que nous proposons comporte les liens entre instance de classe et les valeurs des attributs. Pour chaque instance de classe, nous prenons les numéros des instances de classe qu'elle référence et les valeurs de ses attributs. Voici par exemple les vecteurs représentant les deux modèles donnés à la figure 5.2 :

$$a = (\overbrace{\underbrace{5,4,0,}_{liens}\ \underbrace{2,}_{attributs}}^{instance\ a1}\ \overbrace{\underbrace{4,3,6,}_{liens}\ \underbrace{1}_{attributs}}^{instance\ a2})$$

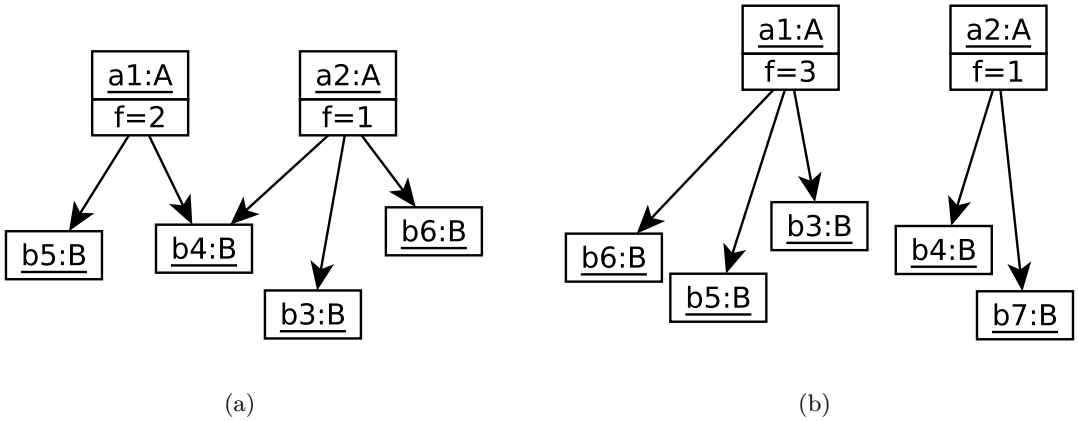


FIGURE 5.2 – Deux modèles que nous comparerons selon la distance de Hamming.

$$b = (\ \overbrace{\underbrace{6,5,3,}_{liens}\ \underbrace{3,}_{attributs}}^{instance\ a1}\ \overbrace{\underbrace{4,7,0,}_{liens}\ \underbrace{1}_{attributs}}^{instance\ a2}\ )$$

La formule de distance telle que donnée par Richard Hamming donne le résultat suivant :

$$\begin{array}{rcccccccc}
\text{a} & = & (5, & 4, & 0, & 2, & 4, & 3, & 6, & 1) \\
\text{b} & = & (6, & 5, & 3, & 3, & 4, & 7, & 0, & 1) \\
\hline
\text{d(a,b)} & = & 1+ & 1+ & 1+ & 1+ & 0+ & 1+ & 1+ & 0 \\
 & = & \frac{6}{8} & & & & & & & \\
 & = & 0.75 & & & & & & &
\end{array}$$

Nous remarquons que la formule d'origine de la distance de hamming donne des résultats assez élevés. Les modèles sont plus proches en réalité que ce que prétend le chiffre de distance. En effet, nous pouvons remarquer que l'instance a1 est par exemple liée dans les deux modèles à l'instance b5 mais ceci est compté comme une différence par la formule de distance car la position du chiffre 5 n'est pas la même entre les deux vecteurs. Autrement dit, la formule de distance est sensible aux permutations ; sachant que notre modèle introduit des symétries liées aux permutations au niveau des liens entre classes (comme expliqué à la section 4.1 sur les symétries).

Pour améliorer les résultats, nous opérons deux modifications importantes au calcul de la distance de hamming : (1) nous modifions la façon de déduire un vecteur à partir d'un modèle, (2) nous changeons de méthode de calcul de la distance.

1. Dans un premier temps les valeurs des coefficients représentant les liens entre classes sont triés par ordre croissant pour chaque instance de classe.

$$a = (\overbrace{\underbrace{5,4,0,}_{liens}\ \underbrace{2,}_{attributs}}^{instance\ a1}\overbrace{\underbrace{4,3,6,}_{liens}\ \underbrace{1}_{attributs}}^{instance\ a2})$$

$$a\ trié = (\overbrace{\underbrace{0,4,5,}_{liens}\ \underbrace{2,}_{attributs}}^{instance\ a1}\overbrace{\underbrace{3,4,6,}_{liens}\ \underbrace{1}_{attributs}}^{instance\ a2})$$

2. Dans un second temps la manière de calculer ne se fait pas par une comparaison coefficient par coefficient entre les deux vecteurs mais par une recherche de l'existence de chaque valeur de lien effectuée pour chaque instance de classe.

Les exemples de la figure 5.3 montrent les améliorations de la valeur calculée pour des extraits de modèles quand nous cassons les permutations, puis quand nous modifions la méthode de calcul.

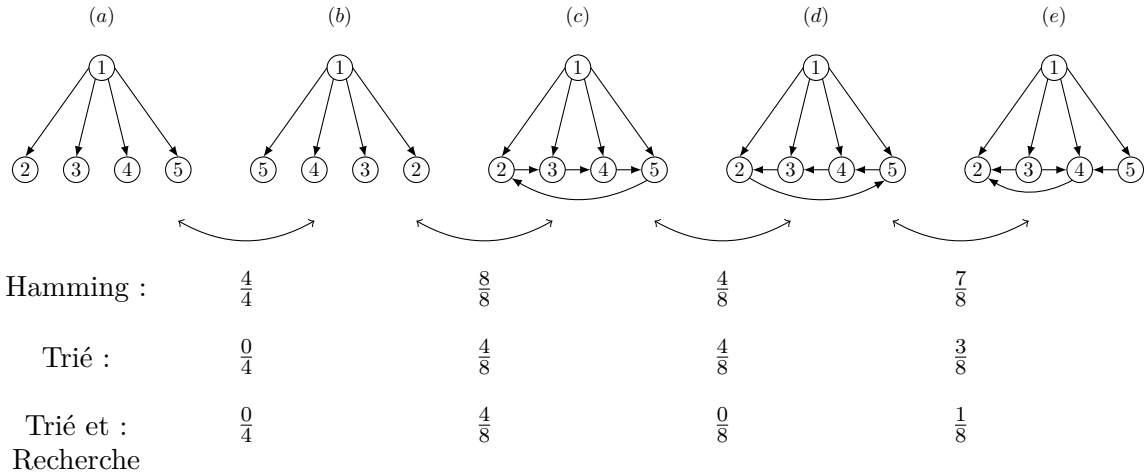


FIGURE 5.3 – Trois versions de la distance de Hamming pour des exemples simples. Les fractions donnent les distances entre deux modèles.

Nous observons que les améliorations apportées au calcul de la distance de Hamming donnent des résultats très intéressants. En effet, la distance n'est plus sensible aux permutations introduites par les liens entre instances de classe. Par exemple, la distance $d(a, b) = 0$ car l'instance 1 est liée aux mêmes instances permutées dans les deux modèles.

Distance Cosinus

D'autres distances assez proches de la distance de Hamming existent. Nous pouvons notamment citer la distance Cosinus. Elle permet également de comparer deux vecteurs. Autrement dit la transformation effectuée sur un modèle pour s'adapter à la distance de Hamming suffit au calcul de la distance Cosinus [98].

5.2.4 Distances basées sur les centralités

5.2.4.1 Centralité

La notion de centralité est un concept très étudié et appliqué à divers domaines depuis la fin des années 1940 [40]. Les premiers travaux sur la question ont appliqué la centralité à des domaines divers et variés. Ils concernaient par exemple l'étude des interactions sociales à l'intérieur de groupes spécifiques. En 1958, McKim Marriott [70] a utilisé la centralité pour analyser le système politique en Inde et au Pakistan. L'étude s'interrogeait sur la difficulté de gouvernance d'un pays aussi peuplé et hétérogène, sachant le système de castes qui était en vigueur à l'époque. D'autres études des années 1950 et 1960 se sont intéressées aux réseaux de transport urbains. Ainsi, Forrest Pitts [82] a réussi à expliquer la place centrale qu'occupe la ville de Moscou en modélisant et en reconstruisant le réseau de transport fluvial du XIIème siècle en Russie.

En théorie des graphes, la centralité est une mesure qui désigne l'importance de la place qu'occupe un nœud dans un graphe comparativement aux autres nœuds. Par exemple il paraît tout à fait intuitif que dans un graphe en étoile (figure 5.4), le nœud au centre du graphe est important car son voisinage comporte plus de nœuds que celui de ces voisins. Il a donc une plus grande valeur de centralité.

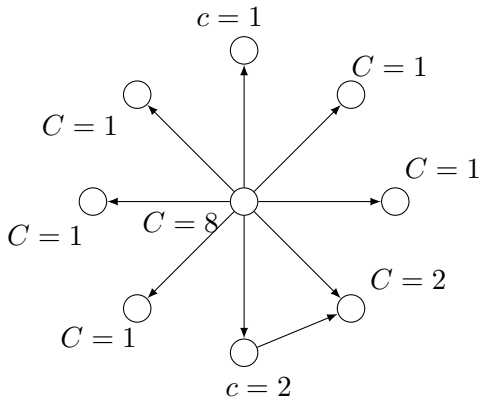


FIGURE 5.4 – Un graphe ayant une topologie en étoile et valeurs de centralité simple basée sur le degré.

Les degrés des nœuds sont affectés comme une valeur de centralité. Il s'agit de l'exemple

le plus simple et le plus intuitif de centralité.

Definition 9 (Centralité) *Une fonction qui applique à chaque nœud d'un graphe un réel positif pour quantifier sa place dans le graphe.*

$$\begin{aligned} C: \quad & E \to \mathbb{R}^+ \\ & v \mapsto C(v) \end{aligned}$$

Il existe plusieurs mesures de centralité connues. Nous n'allons pas toutes les détailler ici. Pour développer notre version adaptée aux modèles nous nous sommes inspirés de la centralité utilisée par la première version de l'algorithme `PageRank` de Google [78]. L'idée est en somme toute simple. Lorsqu'on modélise le web comme un graphe où les pages web sont les nœuds et les liens hypertextes les arêtes, plus une page est référencée, plus elle est populaire et mieux elle sera classée par les algorithmes des moteurs de recherche. Par conséquent, une page populaire possédera une valeur de contralité élevée.

La centralité d'un nœud dans ce cas n'est plus uniquement une valeur statique tel son degré, mais elle dépend des centralités de ses voisins et ainsi de suite. Les valeurs de centralité sont ainsi propagées des pages web les moins populaires vers les pages qu'elles référencent (les plus populaires).

5.2.4.2 Version adaptée aux modèles

La fonction de centralité que nous proposons est adaptée aux modèles dans le sens où elle prend en compte les concepts suivants :

— Instances de classe et leurs attributs.

— Références entre classes (liens entrants ou sortants).

— Type de lien (Référence simple, références bidirectionnelles ou lien de composition).

Voici la formule de centralité pour un nœud v d'un graph G :

$$C(v) = \sum_{u \in N^+(v)} \frac{C(u)}{deg(u)} \times w(v,u).$$

Notez que $w(v,u)$ est une fonction qui donne le poids de l'arête (v,u). Le poids d'une arête change en fonction du type de lien dont il s'agit (attribut, référence ou composition).

Transformation d'un modèle en graphe Avant de pouvoir calculer les centralités des éléments d'un modèle, une transformation de ce modèle en graphe est nécessaire.

Soient $(c,r,t) \in \mathbb{R}^3$ trois réels. Pour tout modèle à transformer, il faut appliquer les traitements suivants :

— Créer un nœud pour chaque instance de classe.

— Créer un nœud feuille pour chaque attribut.
— Créer un arc de poids t de chaque instance vers ses attributs.
— Créer un arc de poids r s'il existe un lien de référencement entre deux instances de classe.
— Créer deux arcs de poids r s'il existe un lien bidirectionnel entre deux instances.
— Créer un arc de poids c pour tout lien de composition.

Les tables 5.3, 5.4 résument et schématisent ces transformations. Tandis que les figures 5.5, 5.6 montrent un modèle contenant 3 instances de classe et le graphe pondéré lui correspondant après transformation.

Partie du modèle	**Nœud du graphe**
:a f=value	:a
:a f=val	f=val

TABLE 5.3 – Nœuds créés lors de la transformation d'un modèle en un graphe.

Partie du modèle	**Arc du graphe**
:a f=value	:a —t→ f=val
:a → :b	:a —r→ :b
:a — :b	:a ⇄ :b (r, r)
:a ◆— :b	:a —c→ :b

TABLE 5.4 – Arcs créés lors de la transformation d'un modèle en un graphe.

Méthode de calcul Le but ici est de calculer le vecteur de centralité pour tous les nœuds du graphe. L'application de la formule de centralité que nous proposons produira des équations à n inconnues :

$$C(v_i) = c_1C(v_1) + c_2C(v_2) + \ldots + c_iC(v_i) + \ldots + c_nC(v_n)$$

Calculer C le vecteur de centralité revient à résoudre ce système d'équation et donc à trouver le vecteur propre d'une matrice Adont les valeurs sont les coefficients des équations précédentes : $C = AC$. A est construite comme suit : $A_{ij} = 0$ s'il n'existe pas d'arête

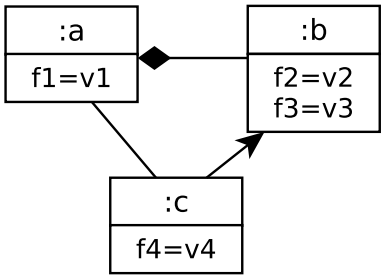


FIGURE 5.5 – Un modèle contenant 3 instances de classe à transformer en graphe.

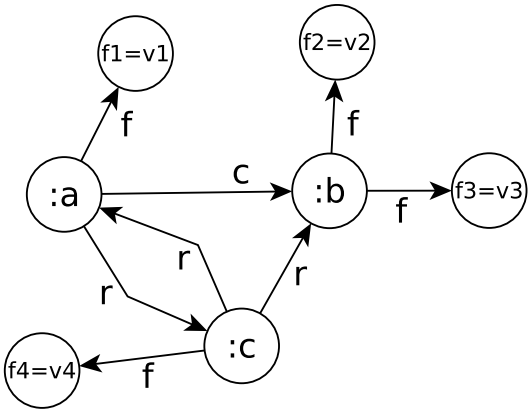


FIGURE 5.6 – Le graphe pondéré obtenu à partir du modèle à transformer.

	a	b	c	f1	f2	f3	f4
a	0	$\frac{c}{4}$	$\frac{r}{4}$	$\frac{f}{4}$	0	0	0
b	0	0	0	0	$\frac{f}{4}$	$\frac{f}{4}$	0
c	$\frac{r}{4}$	$\frac{r}{4}$	0	0	0	0	$\frac{f}{4}$
f1	0	0	0	0	0	0	0
f2	0	0	0	0	0	0	0
f3	0	0	0	0	0	0	0
f4	0	0	0	0	0	0	0

TABLE 5.5 – Matrice A issue du graphe en figure 5.6 qui servira au calcul des centralités.

(v_i, v_j) et $A_{ij} = \frac{w(v_i,v_j)}{N(v_i)}$ sinon. La table 5.5 donne la matrice A obtenue pour le graphe de la figure 5.6.

Nous utilisons la méthode de la puissance itérée pour calculer le vecteur dominant de centralité C (algorithme 1). La méthode commence par l'initialisation du vecteur $C^{(0)}$. Par exemple, un vecteur normalisé uniforme $(\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n})$ est utilisé. La première itération aura comme but le calcul de $C^{(1)}$. En effet, $Z^{(1)} = AC^{(0)}$ et la normalisation de $Z^{(1)}$ donne $C^{(1)}$: $C^{(1)} = Z^{(1)}/||Z^{(1)}||_1$.

L'opération est répétée jusqu'à convergence. L'algorithme est arrêté lorsqu'une précision, fixée à l'avance, est atteinte : $\delta = ||C^{(k)} - C^{(k-1)}|| < \epsilon$.

Data: A, ϵ
Result: C
$C \leftarrow \{1/n, 1/n, \ldots, 1/n\}$;
repeat
  $C_0 \leftarrow C$;
  $Z \leftarrow multMatVect(A, C)$;
  $C \leftarrow normalize(Z)$;
  $\delta \leftarrow normOne(minus(C, C_0))$;
until $(\delta > \epsilon)$;

Algorithm 1: Caclul du vecteur de centralité

La figure 5.7 donne le vecteur de centralité calculé sur le modèle d'exemple (figure 5.6).

Pour connaître les centralités des éléments d'un modèle, il suffit de cumuler la centralité de chaque instance de classe et les centralité de ses attributs (figure 5.8).

5.2.4.3 Normes basées sur la centralité

Nous pouvons désormais comparer deux modèles en utilisant leurs deux vecteurs de centralités. Roy et al. ont montré dans [90] qu'il était possible de spécifier une distance de graphe basée sur la centralité. Nous proposons donc des métriques de distance entre modèles basée sur notre version de la centralité.

Avant de donner les métriques que nous proposons, un post-traitement sera appliqué sur le vecteur de centralité obtenu pour une meilleure expressivité. En effet, lorsqu'un modèle contient beaucoup d'instances, le vecteur de centralité sera très grand. Par ailleurs, nous souhaitons comparer deux modèles conformes au même méta-modèle mais qui ne seront pas forcément de mêmes tailles et qui n'ont donc pas deux vecteurs de centralités de mêmes tailles.

Nous proposons une solution pour, d'un coté réduire la taille des vecteurs à comparer et de l'autre, avoir des vecteurs de mêmes tailles pour deux modèles conformes au même méta-modèle. Nous regroupons les centralités par classe du méta-modèle. Ainsi la centralité d'une classe sera la somme des centralités de ses instances.

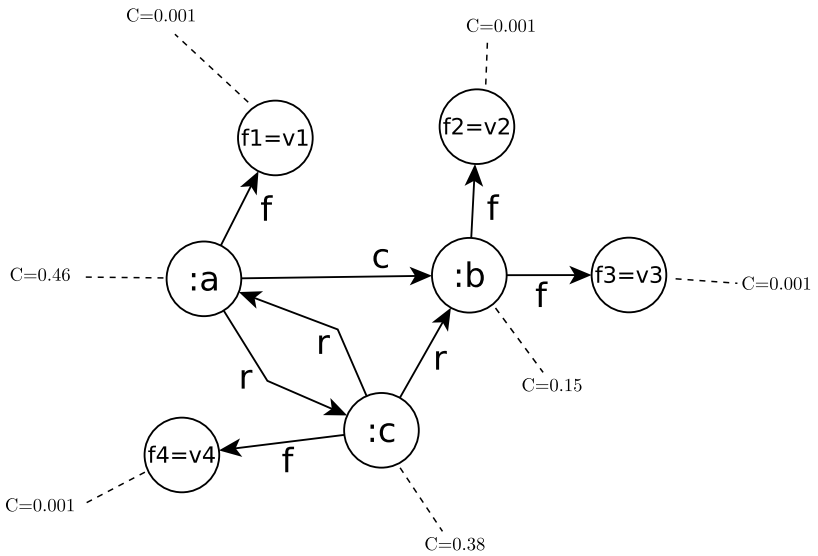


FIGURE 5.7 – Centralités normalisées calculées pour une précision ($\epsilon = 0.01$) et avec les poids suivants : composition ($c = 1.5$), référence ($r = 1$) et attribut ($t = 0.5$).

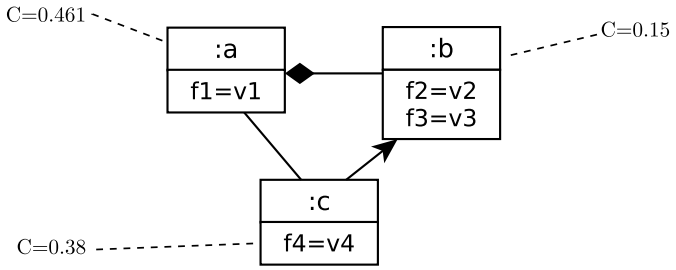


FIGURE 5.8 – Centralités normalisées calculées pour les éléments du modèle (5.5).

Distance Manhattan. Distance associée à la norme 1. Sa formule est donnée comme suit pour deux vecteurs de centralité A et B de tailles n :

$$d(A, B) = ||A - B||_1 = |A_1 - B_1| + |A_2 - B_2| + \ldots + |A_i - B_i| + \ldots + |A_n - B_n|.$$

Distance Euclidienne. Distance associée à la norme 2. Sa formule est donnée comme suit pour deux vecteurs de centralité A et B de tailles n :

$$d(A, B) = ||A - B||_2 = \sqrt{|A_1 - B_1|^2 + |A_2 - B_2|^2 + \ldots + |A_i - B_i|^2 + \ldots + |A_n - B_n|^2}.$$

5.2.5 Distance d'édition de graphes

Un graphe peut être transformé en un autre graphe avec un certain nombre d'opérations d'édition (généralement : l'ajout, la suppression et la substitution). Compter le nombre d'opérations après leur avoir assigné des poids est un moyen de calculer une distance entre deux graphes. Différentes méthodes et algorithmes ont été présentés pour répondre à cette problématique (voir les détails dans cette étude sur la question [45]).

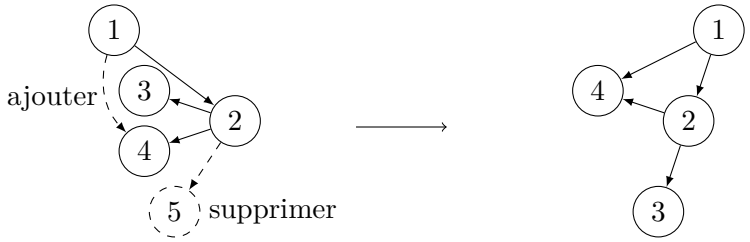


FIGURE 5.9 – Illustration d'une distance d'édition de graphe sur un exemple simple.

La figure 5.9 illustre la distance d'édition de graphe à travers un exemple très simple. Les arcs en pointillées sont à ajouter ou à supprimer pour passer du graphe de gauche au graphe de droite. La distance dans ce cas est de 2 opérations (un ajout d'arc et une suppression de nœud).

Une de ces méthodes nous intéresse tout particulièrement car elle compare deux arbres étiquetés de manière assez efficace [107]. Nous allons ici introduire cette approche et son fonctionnement puis présenter la version adaptée aux modèles que nous proposons.

5.2.5.1 Distance d'édition d'arbre

Zhang et Sasha [107] proposent un algorithme utilisant la programmation dynamique efficace de comparaison de deux arbres étiquetés. Cet algorithme se base sur trois opérations d'édition (illustrées par les figures 5.10, 5.11 et 5.12) :

— Substitution d'un nœud v : changer l'étiquette de v.
— Ajout d'un nœud v : Insérer v et faire de lui le parent des fils de son nouveau parent.
— Suppression d'un nœud v : Faire que les fils de v deviennent les fils du parent de v, puis supprimer v.

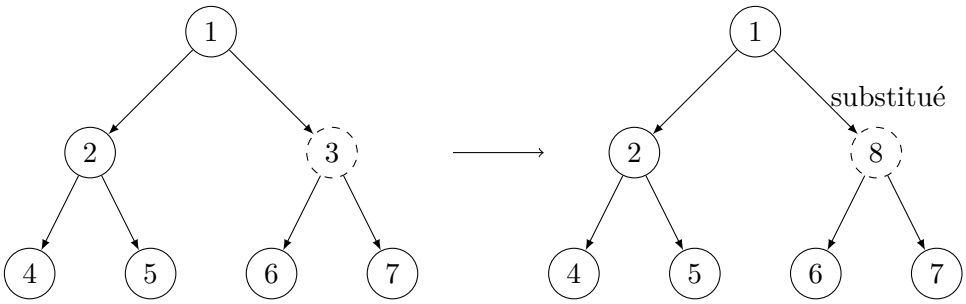


FIGURE 5.10 – Illustration de l'opération de substitution.

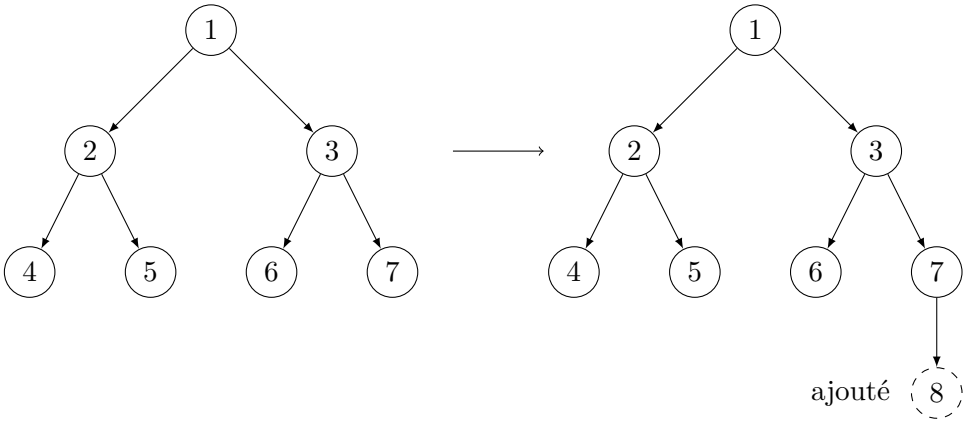


FIGURE 5.11 – Illustration de l'opération d'ajout.

Un poids est ensuite assigné à chaque opération d'édition. Par conséquent la distance entre deux arbres est donnée par :

$$d(A, B) = cout_{sub} \times |Sub| + cout_{ajout} \times |Ajout| + cout_{su} \times |Supp|.$$

5.2.5.2 Version adaptée aux modèles

Les modèles conformes aux méta-modèles Ecore admettent une structure arborescente lorsque seuls les liens de composition sont pris en compte. Néanmoins, pour une distance plus expressive, il ne suffit pas de prendre en compte uniquement les liens de composition car d'autres liens qui rendent les modèles cycliques existent également.

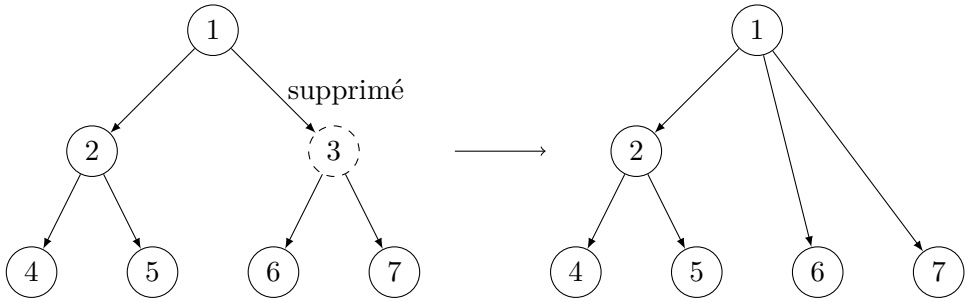


FIGURE 5.12 – Illustration de l'opération de suppression.

Deux problèmes se posent lorsqu'on désire utiliser l'algorithme de Zhang et Sasha pour comparer deux modèles : (1) trouver une structure arborescente d'un modèle et (2) améliorer la méthode de Zhang et Sasha pour qu'elle prenne en compte les attributs et tous les éléments d'un modèle.

1. Les liens de composition décrivent une arborescence partant de la racine du modèle et comprenant toutes les autres instances de classe.
2. Pour la prise en compte du reste des éléments d'un modèle (attributs et références), nous définissons une structure de données particulière. Ainsi, chaque nœud de notre arborescence est défini par un quadruplet : type, ae =nombre d'arcs entrants, as =nombre d'arcs sortants et $liste$ =liste des attributs avec leurs valeurs.

$$\begin{bmatrix} type \\ ae \\ as \\ liste \end{bmatrix}$$

Les figures 5.13 et 5.14 montrent un modèle simple et sa transformation en arbre pour le calcul de la distance d'édition de graphe.

La dernière étape consiste à définir les fonctions de coûts adéquates. Les coûts de l'ajout et de la suppression sont constants et égaux à 1 car seul le nœud est supprimé ou bien ajouté. Tandis que le coût de la substitution d'un nœud par un autre dépend également des valeurs des attributs et des arcs entrants et sortants des deux nœuds. Le coût total de la substitution est égal à la différence du nombre d'arcs sortants et entrants pour les deux nœuds plus le nombre d'attributs ayant des valeurs différentes. Par ailleurs, il est impossible de substituer un nœud par un nœud de type différent.

5.2.6 Comparaison des distances entre modèles

Dans cette section, nous dressons un comparatif détaillé des trois métriques de distance entre modèles que nous proposons. Cette comparaison s'effectuera selon des critères liés à

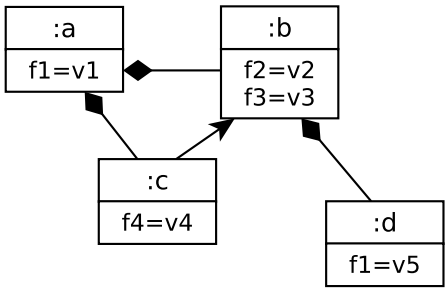


FIGURE 5.13 – Un modèle à transformer pour la distance d'édition.

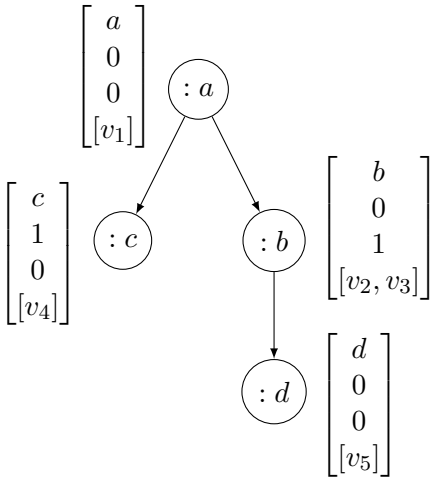


FIGURE 5.14 – Un arbre en entrée de l'algorithme de Zhang et Sasha. Le vecteur pour chaque nœud donne dans l'ordre : son type, ses arcs sortants, ses arcs entrants et les valeurs de ses attributs.

la complexité en temps (et à la vitesse d'exécution), à l'expressivité de la distance et à la structure de graphes que chacune des distances capture.

5.2.6.1 Complexité et vitesse d'exécution

La table 5.6 compare les complexités en temps des 3 différentes distances que nous avons développées. La distance d'édition de graphe basée sur l'algorithme de Zhang et Sasha possède une complexité très importante et des temps d'exécution très élevés. Ainsi, il ne faut que quelques centaines de millisecondes pour comparer deux modèles de tailles importantes (1000 éléments) avec la distance de hamming améliorée ou bien pour les distances basées sur le calcul des centralités. Tandis que plusieurs minutes sont nécessaires pour calculer la distance d'édition de graphe pour deux modèles de tailles modestes (moins de 50 éléments).

Il est à noter que le coût de transformation d'un modèle est à peu près semblable pour les trois distances.

5.2.6.2 Expressivité

Contrairement à la comparaison des complexités qui est quantifiable, la comparaison des trois distances selon le critère d'expressivité repose sur des considérations qualitatives.

Éléments du méta-modèle pris en compte

Distance	**Complexité**			**Temps d'exécution**	
	Transformation	**Calcul**	**Comparaison**	50	1000
Hamming amélioré	$O(n)$	-	$O(n^2)$	$0.1s$	$0.5s$
Centralité	$O(n)$	$O(n^2)$	$O(\|\mathcal{C}l\|)$	$0.1s$	$0.6s$
Édition de graphe	$O(n\log(n))$	-	$O(n \times m \times k_1 \times k_2)$	$15m$	–

TABLE 5.6 – Comparaison de la complexité des trois métriques. n, m : nombre d'instances de classe et d'attributs des modèles. $|\mathcal{C}l|$: nombre de classes du méta-modèle, $|\mathcal{C}l| \ll n \simeq m$, $k_i = min(depth(M_i), leaves(M_i))$.

— Hamming : la distance prenant en compte le moins d'éléments du méta-modèle. Seuls les liens entre classes et les attributs sont pris en compte. Le type de lien ne peut pas être traité.

— Centralité : dans ce cas tous les éléments du modèle peuvent être pris en compte. Cela comprend les instances de classe, les attributs et tous les types de liens.

— Édition de graphe : comme pour la précédente, tous les éléments du modèle sont traités.

Distance	**Éléments du méta-modèle**			**Pondération**
	Instances	**Attributs**	**Liens**	
Hamming amélioré	+	+	+	−
Centralité	+	+	+	+
Édition de graphe	+	+	+	+

TABLE 5.7 – Comparaison des éléments pris en compte par les trois métriques.

Possibilité de paramétrage et de pondération

— Hamming : elle n'offre que peu de possibilités de paramétrage et de pondération pour s'adapter aux modèles.

— Centralité : des poids peuvent être assignés aux attributs et aux différents types de liens.

— Édition de graphe : les fonctions de coûts permettent d'affecter des poids aux attributs et aux types de liens.

La table 5.7 résume la comparaison des distances selon les éléments du méta-modèle que prend en compte chacune d'elles. Les pictogrammes indiquent : ⊖ : critère non satisfait, ⊕ : satisfait, ⊕ :satisfait à moitié.

5.2.6.3 Structure des graphes

Nous allons maintenant nous intéresser aux structures de graphe qui peuvent être comparées en utilisant chacune des trois métriques de distances.

— Hamming : dans ce cas un graphe est transformé en vecteur pour être comparé avec un autre. Cela fait que seules des permutations de liens entre instances de classes sont détectées. Lorsqu'une structure plus importante est redondante, la distance de hamming n'est capable de le voir que dans certains cas très précis et simples. Néanmoins, les permutations de liens permettent à la distance de hamming de s'en sortir assez bien pour les modèles à topologie différente.

— Centralité : un score de centralité est associé à chaque instance de classe. Le fait de sommer les scores pour obtenir un seul score par classe du méta-modèle permet de comparer des modèles avec des sous-graphes redondants et ce quelles que soient les permutations. Néanmoins, des topologies très différentes peuvent donner les mêmes scores de centralité.

— Édition de graphe : cette distance est la plus à même de comparer les topologies différentes. Cette métrique est capable de détecter n'importe quel sous-graphe commun entre deux graphes à comparer. Il faut néanmoins garder en tête que montrer que deux graphes sont isomorphes est un problème extrêmement difficile et donc très coûteux.

La table 5.8 compare les structures de graphe détectées par les différentes distances. Les pictogrammes indiquent : ⊕ : parfaitement détecté, ⊕ : assez bien détecté, ⊖ : mal ou assez mal détecté.

Distance	**Sensibilité aux labels**	**Permutations de liens**	**Topologies différentes**	**Sous-graphes**
Hamming amélioré	⊖	⊕	⊕	⊖
Centralité	⊕	⊕	⊕	⊕
Édition de graphe	⊕	⊕	⊕	⊕

Table 5.8 – Comparaison des structures de graphe.

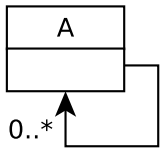


FIGURE 5.15 – Extrait de méta-modèle d'une classe et d'une référence.

5.2.6.4 Illustrations par des exemples

Étant donnés les trois modèles de la figure 5.16 et conformes au petit méta-modèle de la figure 5.15, allons calculer les distances entre ces trois modèles avec chacune des distances que nous proposons dans le but de comparer et d'analyser ces résultats.

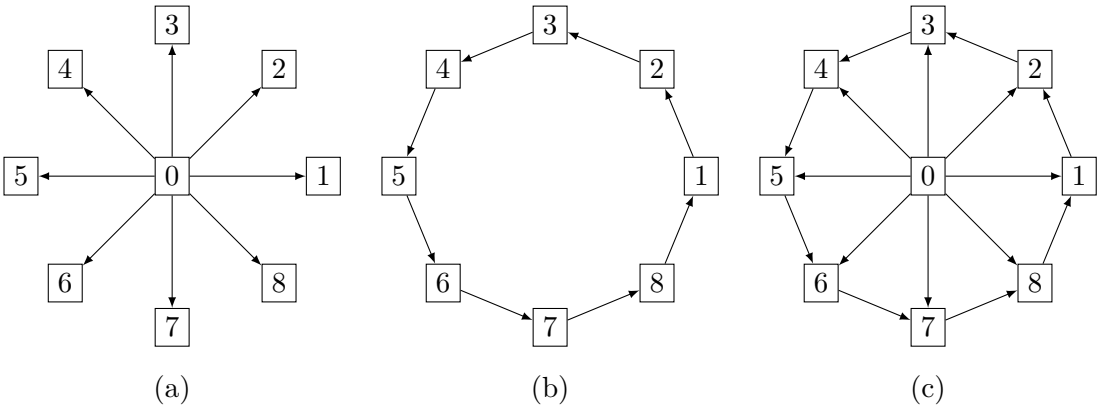


FIGURE 5.16 – Trois modèles de topologies types.

La figure 5.17 donne les résultats du calcul des distances entre les trois modèles précédents deux à deux. Nous pouvons faire différentes remarques pour analyser ces résultats :

— Pour ces exemples, toutes les distances vont dans le même sens. Elles capturent bien la structure de chaque modèle et les différences de topologies qu'il peut y avoir entre deux modèles.

— La distance basée sur la centralité s'avère être la plus précise car elle opère un calcul complexe et assigne un score réel à chaque élément et non des fonctions de coûts entières comme les deux autres. De plus on voit que c'est la seule qui donne des résultats différents pour (a) vs. (c) et (b) vs (c).

— Si on décide de modifier les labels des nœuds pour par exemple faire une rotation, seules les distances basée sur la centralité et d'édition de graphe vont pouvoir le

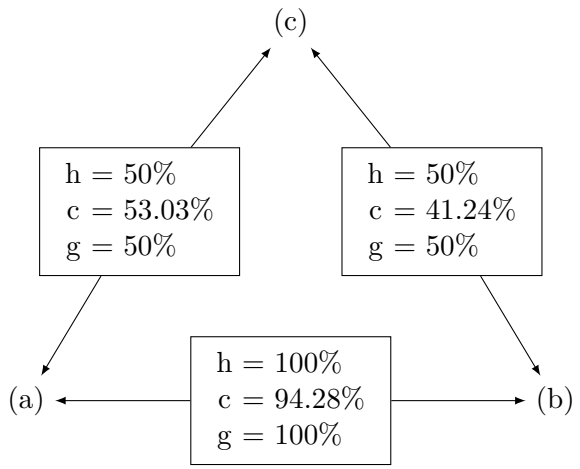


FIGURE 5.17 – Calcul des distances. h : hamming, c : euclidienne à base de centralité, g : édition de graphe.

détecter. La distance de hamming qui se base sur la comparaison des labels échouera à trouver la bonne distance.

5.2.6.5 Conclusion

En conclusion de cette comparaison, nous pouvons dire qu'il n'existe aucune nette dominance d'une distance par rapport aux deux autres. Le seul critère très discriminent et quantifiable reste la complexité, car elle met en avant deux des trois distances. Pour ce qui est de l'expressivité, nous remarquons qu'il existe des ressemblances, avec certes à petit désavantage pour hamming. Même si cette dernière se rattrape bien avec nos versions améliorées.

Donc, en lieu et place d'un choix radical d'une seule distance, nous pensons que la meilleure stratégie est d'opter pour des objectifs multiples à base de deux distances ou plus. C'est le seul moyen d'avoir une certitude sur la distance entre deux modèles.

5.3 Sélectionner des modèles diversifiés

Maintenant que des distances comparant des modèles sont au point, et que des matrices de distances peuvent être produites, nous pouvons procéder à l'étape de sélection des modèles les plus éloignés parmi un ensemble de modèles. Cette sélection peut se faire de deux manières différentes :

1. L'utilisateur formule un souhait sous forme d'un nombre de modèles à choisir et un algorithme sélectionne automatiquement les modèles les plus éloignés les uns des autres.

2. Un rendu graphique de la matrice de distance est produit automatiquement et l'utilisateur peut faire lui-même sa sélection.

Les deux sous-sections qui suivent présentent les deux manières que nous proposons pour sélectionner des modèles différents parmi un ensemble de modèles. Des exemples sont également donnés.

5.3.1 Sélection de modèles : Technique de Clustering

Étant donnée une matrice de distance entre modèles (carrée, symétrique et à diagonale nulle). Nous utilisons des techniques de clustering pour identifier les sous-ensembles de modèles les plus éloignés. La technique de clustering que nous avons choisi est K-means [54]. C'est l'une des techniques de clustering les plus connues et les plus utilisées et elle est disponible sur R. Par exemple, la table 5.10 donne les 4 différents clusters obtenus par l'utilisation de K-means (script, voir Annexe A) pour la matrice de distance de la table 5.9. Il est à noter que le nombre de clusters à trouver est choisi par l'utilisateur. Pour trouver 4 modèles différents et les plus éloignés, il suffit de prendre un seul modèle au hasard par cluster. Dans ce cas il y a 24 combinaisons différentes.

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
m1	0	12	27	27	27	26	46	44	45	39
m2	12	0	27	26	27	27	45	45	43	40
m3	27	27	0	18	17	16	46	45	46	39
m4	27	26	18	0	18	18	45	44	45	40
m5	27	27	17	18	0	18	45	43	44	38
m6	26	27	16	18	18	0	45	44	46	40
m7	46	45	46	45	45	45	0	36	36	41
m8	44	45	45	44	43	44	36	0	34	37
m9	45	43	46	45	44	46	36	34	0	39
m10	39	40	39	40	38	40	41	37	39	0

Table 5.9 – Matrice de distance (% de différence) pour 10 graphes de Scaffold comparés par Hamming.

5.3.2 Visualisation : Diagramme de Voronoi

La deuxième solution consiste à utiliser des diagrammes de Voronoi [4] pour représenter graphiquement l'éloignement des modèles les uns par rapport aux autres. Un diagramme

cluster #	Modèles contenus par le cluster
1	m7, m8, m9
2	m1, m2
3	m10
4	m3, m4, m5, m6

TABLE 5.10 – Clusters et les modèles qu'ils contiennent, pour 10 graphes de Scaffold comparés par Hamming.

de Voronoi représente un échantillon en se basant sur un critère de comparaison entre les éléments. Dans notre cas, le critère est la distance comparant deux modèles. Les figures 5.18 et 5.19 montrent deux diagrammes de Voronoi produits par R sur deux échantillons de modèles (des graphes de Scaffold d'un côté et des réseaux de Petri de l'autre). Les diagrammes de Voronoi opèrent également un clustering de l'échantillon de modèles et reproduisent aussi fidèlement que possible en 2D les distances entre les points.

5.4 Plus de diversité : Algorithmique génétique

Dans la section précédente, nous comparons des modèles grâce à des métriques de distance et cela permet la sélection des modèles les plus diversifiés parmi un plus grand ensemble de modèles. La limite d'une telle approche réside dans la supposition d'une grande diversité inhérente à l'ensemble de départ. Si la diversité de départ est faible, la solution que nous proposons ne fera rien pour l'améliorer. Imaginons maintenant un système itératif ou à chaque étape de nouveaux modèles sont créés. L'ensemble de modèles qui améliore la diversité est sélectionné pour passer à l'étape suivante. Ceci est la définition de l'algorithmique génétique (GA) [30].

Les algorithmes génétiques [56, 50] sont des méthodes méta-heuristiques inspirées du principe darwiniste de la théorie de l'évolution. Dans ce qui suit nous résumons brièvement quelques uns des termes les plus importants du vocabulaire associé à l'algorithmique génétique.

Definition 10 *(***Chromosome***) (ou individu) représentation d'une solution possible de l'espace des solutions, constitué d'un ensemble de gènes.*

Definition 11 *(***Gène***) élément atomique permettant le codage de la représentation du chromosome (e.g. un bit dans une chaîne de bits).*

Definition 12 *(***Population***) Ensemble des solutions considérées à un moment donné.*

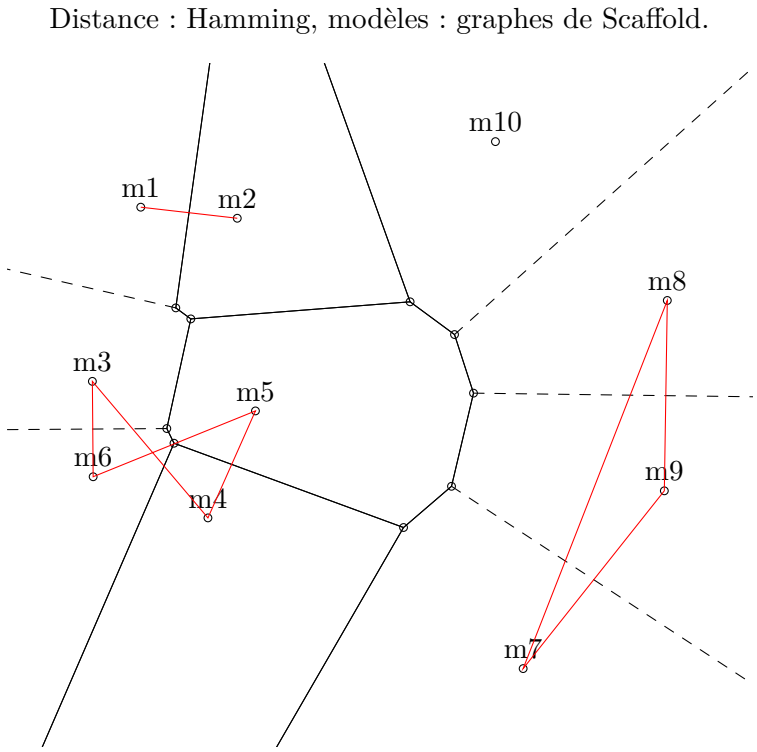


FIGURE 5.18 – Diagramme de Voronoi pour 10 graphes de Scaffold comparés par Hamming et basé sur la matrice de la table 5.9. Les clusters sont illustrés par les lignes rouge.

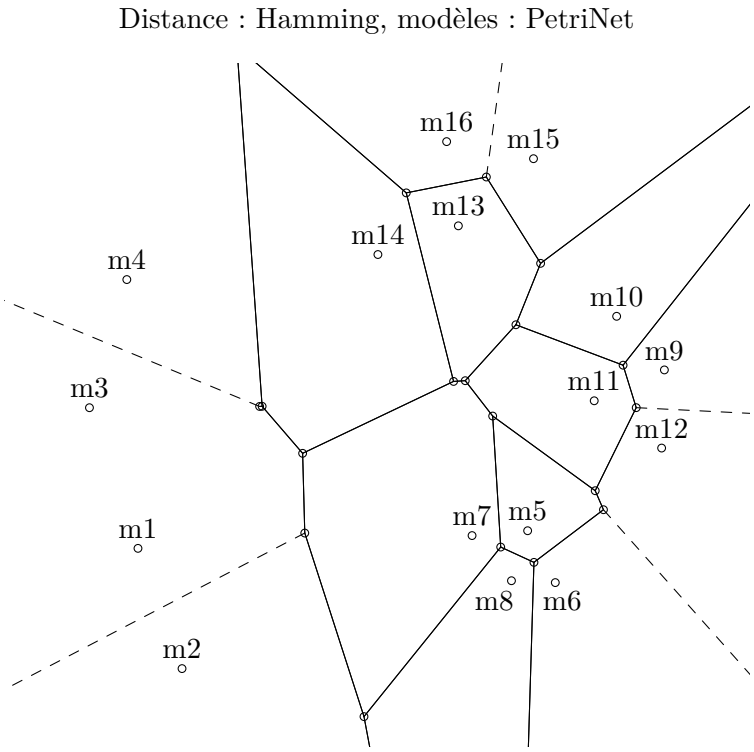


FIGURE 5.19 – Diagramme de Voronoi pour 16 modèles conformes au méta-modèle des réseaux de Petri comparés par Hamming.

Le principe de l'algorithme génétique est d'explorer l'espace des solutions à l'aide de réarrangements génomiques pour faire évoluer la population actuelle vers une nouvelle population qui améliore les scores des fonctions objectifs. L'évolution de la population repose sur deux opérations de base : le croisement et la mutation. Le croisement consiste à découper des morceaux de deux individus pour en fabriquer un nouveau. La mutation complète le croisement en faisant varier aléatoirement des petits morceaux des individus ainsi créés (figure 5.20).

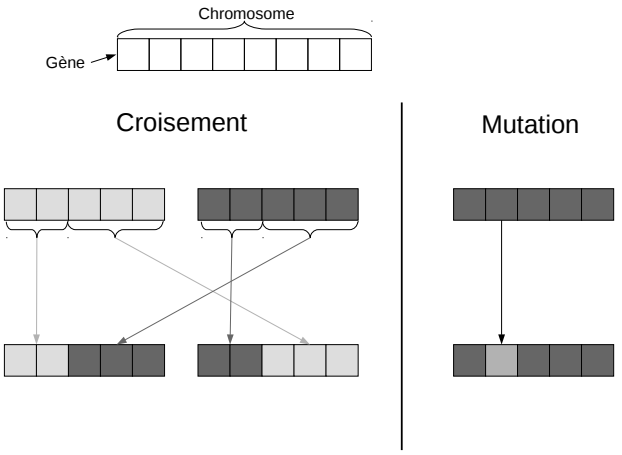


FIGURE 5.20 – Exemple de croisement et de mutation sur des chromosomes. À gauche, croisement en un point de deux chromosomes ; à droite, mutation d'un chromosome.

A chaque étape de l'algorithme génétique, un ensemble de nouveaux individus est généré par des croisements et des mutations. Ensuite, les meilleurs individus sont sélectionnés pour faire partie de la nouvelle population. La sélection des meilleurs individus se fera en optimisant une ou plusieurs fonctions objectifs.

chap. 3 ▷ Dans notre approche, nous choisissons de représenter un modèle par son vecteur de solution donnée par le CSP généré. La sélection se fait en choisissant les modèles les plus diversifiés parmi une population donnée, donc en se basant sur les distances entre modèles

sec. 5.3 ▷ et le clustering de matrices de distances.

Nous avons mené une évaluation où un ensemble de 100 modèles constitue la population de départ. Le but est d'appliquer l'algorithmique génétique pour diversifier cet ensemble. L'évaluation suit les étape suivantes :

1. Appliquer des croisements et des mutations sur la population de départ pour générer une nouvelle population plus grande.
2. Vérifier la validité des modèles générés en se servant du solveur de contraintes.
3. Calculer la matrice de distances sur la population obtenue avec les distances entre modèles définies plus haut.

4. Appliquer l'algorithme de clustering précédent pour sélectionner les 100 modèles les plus représentatifs.
5. Faire de ces 100 modèles la nouvelle population.
6. Répéter le processus 500 fois.

Les résultats de l'évaluation montre une diversification accrue et une bonne et rapide convergence de la méthode. En effet, les courbes de la figure 5.21 montre l'accroissement de la distance moyenne pour les deux métriques testées et une convergence assez rapide notamment pour la distance de Hamming.

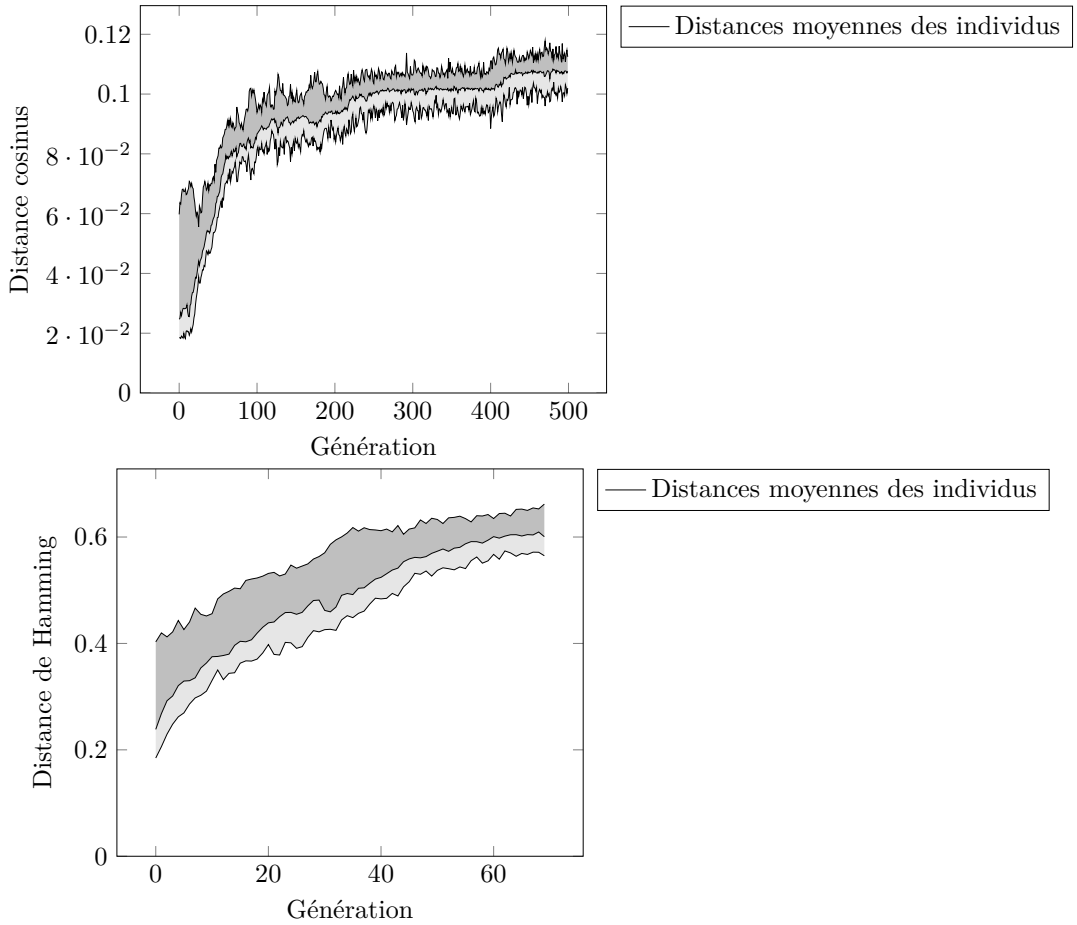


FIGURE 5.21 – Évolution des distances moyennes des modèles sur une population de 100 individus.

La figure 5.22 montre deux diagrammes de voronoi, mis à l'échelle selon la distance maximale de l'échantillon, représentant la répartition de deux populations de 100 modèles, celle de départ et celle de la dernière génération. Deux enseignements peuvent être tirés de

ces diagrammes :

1. Augmentation de la distance maximale et moyenne de la population. La taille du deuxième diagramme par rapport au premier le montre parfaitement.
2. Amélioration de la répartition dans l'espace des modèles générés par l'algorithmique génétique.

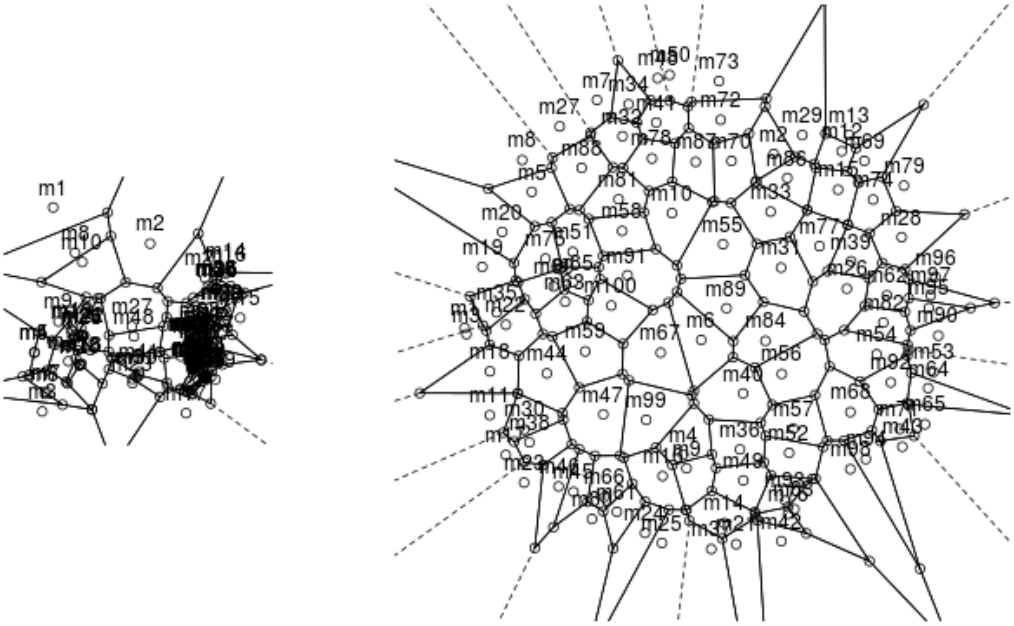


FIGURE 5.22 – Diagrammes de Voronoï (mis à l'échelle selon la distance maximale) représentant une population de modèles, à gauche : population de départ, à droite : 500ème génération génétique.

5.5 Conclusion

Le chapitre qui s'achève est consacré à la comparaison des modèles dans le but de sélectionner les plus éloignés parmi un ensemble. Nous avons d'abord introduit la diversité des modèles et son intérêt, et proposé l'idée de comparer des modèles en utilisant des métriques de distance. La section 5.2 présente les trois métriques de distances que nous proposons. Ces distances sont inspirées de distances mathématiques et de graphes connues et sont adaptées à l'IDM. Chaque distance est présentée dans sa version d'origine et dans sa version adaptée aux modèles. Les transformations nécessaires à un modèle pour s'adapter à chaque métrique sont également explicitées. Par ailleurs, un comparatif détaillé de la complexité et de l'expressivité des trois distances est dressé.

La section 5.3 montre comment l'utilisation des distances présentées plus haut et des techniques de clustering permet de sélectionner les modèles les plus éloignés représentatifs d'un ensemble de modèles donné.

La section 5.4 relate l'utilisation des algorithmes génétiques pour améliorer et maximiser, avec succès, la diversité d'un ensemble de modèles. Les résultats de cette section ont été publié par Florian Galinier [42] durant son stage que nous avons encadré.

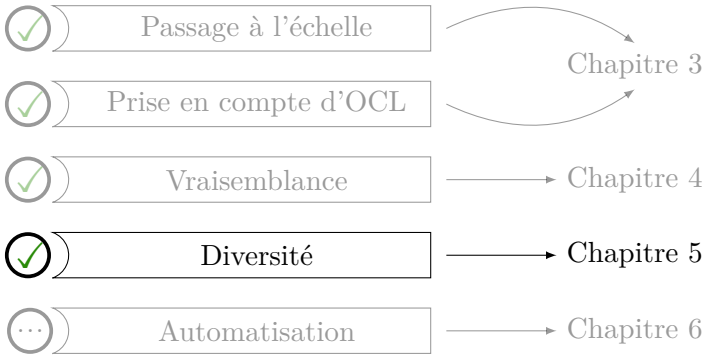


FIGURE 5.23 – Caractéristiques de la génération de modèles abordées par le chapitre 5.

Chapitre 6

Outil $\mathcal{G}$rimm

> In theory there is no difference between theory and practice, but in practice there is.
>
> Jan L. A. van de Snepscheut ou Yogi Berra

Synopsis

Préambule

Ce chapitre présente l'outil $\mathcal{G}$rimm ($\mathcal{G}$eneRating Instances of Meta-Models) dans lequel sont implémentées toutes les contributions de ce manuscrit. D'abord, nous expliquons l'outil et ses objectifs (section 6.1). Ensuite, nous abordons les détails d'implémentation et les technologies utilisées et donnons les paramètres obligatoires pour faire tourner $\mathcal{G}$rimm, ses différentes options et versions (section 6.2). Enfin, nous détaillons une expérience menée avec des utilisateurs de l'outil (section 6.3).

6.1 Outil $\mathcal{G}$rimm

Les contributions à la recherche présentées aux trois chapitres précédents ont donné lieu au développement d'un outil de génération de modèles nommé $\mathcal{G}$RIMM ($\mathcal{G}$eneRating Instances of Meta-Models). Le but de cet outil est d'instancier tout type de méta-modèles (au format Ecore) pour générer automatiquement des instances valides. Ainsi, il implémente toutes les techniques décrites plus haut dans le manuscrit.

Dans la suite de cette section, nous allons décrire son principe de fonctionnement, son paramétrage et présenter ses différentes versions et variantes et leurs options ainsi que des exemples de méta-modèles et de modèles conformes.

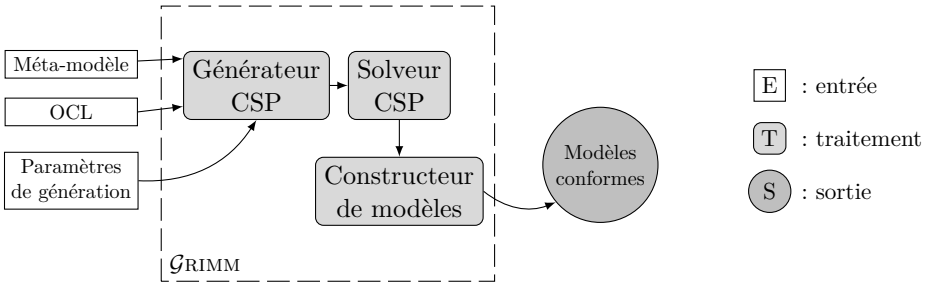


FIGURE 6.1 – Processus de l'outil $\mathcal{G}$RIMM.

La figure 6.1 montre les entrées, sorties et les étapes de génération d'instances en utilisant l'outil $\mathcal{G}$RIMM. Nous allons expliquer chacun des éléments qui composent cette figure. Nous donnons également toutes les commandes utiles pour utiliser $\mathcal{G}$RIMM.

6.1.1 Entrée & sorties

6.1.1.1 Entrées

L'outil $\mathcal{G}$RIMM prend en paramètre un méta-modèle, un fichier de contraintes OCL et un certain nombre de paramètres de configuration et génère des modèles conformes au méta-modèle de départ.

Méta-modèle L'outil génère des instances conformes uniquement à des méta-modèles au format Ecore. De plus ces méta-modèles doivent remplir les conditions suivantes :

— Être syntaxiquement corrects.

— Disposer d'une classe racine. Elle possède la caractéristique d'être directement ou indirectement liée par lien de composition à toutes les autres classes du méta-modèle.

Listing 6.1 – Spécification d'un méta-modèle et de sa classe racine

```
java -jar grimm.jar -mm=*.ecore -root=class
```

Contraintes OCL S'il le désire, l'utilisateur peut accompagner son méta-modèle d'un fichier contenant des contraintes OCL que devront satisfaire les modèles générés. Les contraintes supportées sont toutes celles décrites à la section 3.2.

Listing 6.2 – Spécification d'un fichier OCL

```
java -jar grimm.jar -mm=*.ecore -root=class -ocl=*.ocl
```

Paramètres de génération La génération nécessite que l'utilisateur paramètre le générateur en indiquant le nombre d'instances par classe et en donnant des bornes aux références non bornées (celles ayant $*$ comme cardinalité maximale). Pour plus de détails se référer à la section 6.1.2.

6.1.1.2 Sortie

La sortie produite est un modèle conforme au méta-modèle d'entrée. Ce modèle produit peut se présenter sous deux formes différentes :

1. Fichier XMI (*voir glossaire*) : le modèle est représenté comme un fichier xml.
2. Fichier Dot (*voir glossaire*) : le modèle est représenté graphiquement comme un diagramme d'instance du méta-modèle. Un visuel graphique (pdf, png, ...) peut également être produit en utilisant l'outil GraphViz.

Listing 6.3 – Génération d'un modèle au format XMI ou dot

```
java -jar grimm.jar -mm=*.ecore -root=class ... -xmi (ou -dot)
```

6.1.2 Paramétrage et configuration

L'outil offre deux modes de configuration pour les paramètres de taille du modèle à générer. L'un est rapide, directement accessible en ligne de commande mais peu détaillé, tandis que l'autre est plus détaillé et nécessite l'écriture d'un fichier de configuration.

6.1.2.1 Paramètres en ligne de commande

Ce mode requiert l'introduction par l'utilisateur de trois valeurs obligatoires :

— `-lb` : donne le nombre minimum d'instances par classe.

— `-ub` : donne le nombre maximum d'instances par classe.

— `-rb` : donne la cardinalité pour les références non bornées.

Le nombre exact d'instances pour chaque classe est ensuite choisi en générant aléatoirement un nombre compris entre `lb` et `ub`. Concernant les attributs, l'outil donnera des domaines arbitraires. Ce mode rapide est préféré pour les méta-modèles de grande taille.

Listing 6.4 – Paramètres de configuration en ligne de commande

```
java -jar grimm.jar -mm=*.ecore -root=class -xmi -lb=3 -ub=10 -rb=5
```

6.1.2.2 Fichier de configuration

Pour garder le contrôle total sur les paramètres de taille, il est nécessaire de se servir d'un fichier de configuration (.grimm). Ce fichier spécifie le nombre exact d'instances pour chaque classe et le domaine voulu pour chaque attribut. Pour faciliter son utilisation, $\mathcal{G}$RIMM génère des fichiers de configuration pré-remplis à compléter.

Listing 6.5 – Génération d'un fichier de configuration pré-rempli par l'outil

```
java -jar grimm.jar -mm=*.ecore -root=class
```

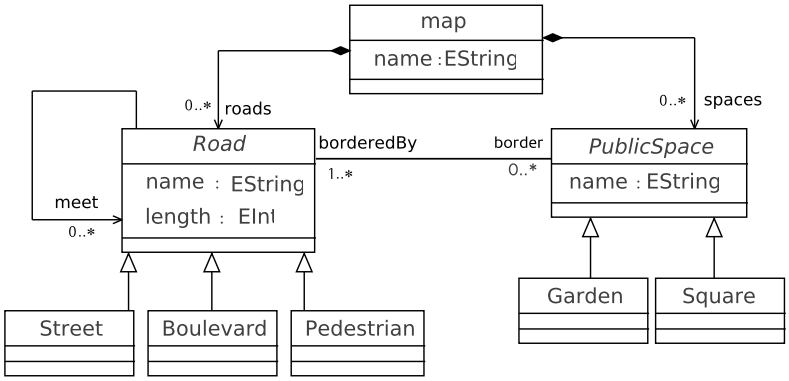


FIGURE 6.2 – Méta-modèle pour la définition d'extraits de cartes

Listing 6.6 – Contenu d'un fichier de configuration .grimm du méta-modèle en figure 6.2 généré pré-rempli puis complété à la main

```
% -- -- -- -- -- -- -- -- -- -- -- -- --
%Number of instances for Classes
Street=1
Boulevard=2
Pedestrian=1
Garden=2
Square=1
% -- -- -- -- -- -- -- -- -- -- -- -- --
%Domains of attributes
map/name=Montpellier
Street/name=Verdun Victor_Hugo
Street/length=200 400 500 150
Boulevard/name=Jeu_de_paume Jean_Jaures
Boulevard/length=1000 2000 3000 4000
Pedestrian/name=Leon_Blum Nelson_Mandela
```

```
Pedestrian/length=600 700 900
Garden/name=Tabalizt Akermus
Square/name=Lekhmis
% -- -- -- -- -- -- -- -- -- -- -- -- --
%Some others
RefsBound=3
AttributesBound=2
```

Une fois le fichier de configuration rempli, il peut être utilisé pour générer des modèles conformes (dot ou XMI).

Listing 6.7 – Utilisation d'un fichier de configuration .grimm

```
java -jar grimm.jar -mm=*.ecore -root=class -cfg=*.grimm -dot
```

6.1.3 Processus de fonctionnement & implémentation

6.1.3.1 Générateur de CSP

Cette étape consiste à générer un CSP en partant des entrées (Méta-modèle, et OCL) et des paramètres de configuration. Chaque triplet Méta-modèle, contraintes OCL et paramètres de configuration donne lieu à un fichier d'instance CSP.

Le générateur de CSP est écrit en Java. Il utilise le framework EMF (Eclipse Modelling Framework) pour lire le méta-modèle et Eclipse OCL pour manipuler l'arbre syntaxique d'une contrainte OCL avant de l'encoder en CSP.

Une fois l'encodage du méta-modèle et de ses contraintes OCL achevé, un fichier xcsp (xml csp) est obtenu et peut être résolu. ◁ sec. 2.4

Listing 6.8 – Extrait d'un fichier xcsp du méta-modèle en figure 6.2 et du fichier de configuration du listing 6.6

```
<?xml version="1.0" encoding="UTF-8"?>
<instance>
  <presentation name="?" maxConstraintArity="2" format="XCSP␣2.0" />
  <domains nbDomains="35">
    <domain name="DC1" nbValues="1">1</domain>
    <domain name="DC1_2" nbValues="2">0 1</domain>
    <domain name="DC_map" nbValues="1">1</domain>
    <domain name="DC3" nbValues="1">3</domain>
    <domain name="DC4" nbValues="1">5</domain>
  <variables nbVariables="42">
    <variable domain="DF1_1" name="F_map_1_name" />
    <variable domain="DCRJ2_1_1" name="Id_map_1_roads_1" />
    <variable domain="DCRJ2_1_1" name="Id_map_1_roads_2" />
    <variable domain="DCRJ2_1_1" name="Id_map_1_roads_3" />
    <variable domain="DCRJ2_2_1" name="Id_map_1_spaces_1" />
    <variable domain="DCRJ2_2_1" name="Id_map_1_spaces_2" />
    <variable domain="DCRJ2_2_1" name="Id_map_1_spaces_3" />
    <predicate name="inf">
```

```
</constraint>
```

6.1.3.2 Solveur de CSP

Le CSP généré à l'étape précédente est résolu par un solveur de programmes contraints. $\mathcal{G}$RIMM réutilise un solveur existant qui prend en entrée des fichiers au format xcsp [66]. Il s'agit du solveur Abscon[1] [72].

Si le CSP est satisfait, le solveur retourne une solution où chaque variable se voit assigner une valeur de son domaine.

6.1.3.3 Constructeur de Modèles

La dernière étape du processus de génération de modèles consiste à construire des instances valides en se basant sur les valeurs assignées aux variables par le solveur de contraintes. Cette partie est également codée en java. Elle utilise EMF pour la génération de fichiers XMI. Pour construire des diagrammes d'instances visible graphiquement, $\mathcal{G}$RIMM génère des graphes au format dot et fait appel à GraphViz pour produire un pdf ou une image.

Listing 6.9 – Contenu d'un fichier dot représentant une instance du méta-modèle en figure 6.2

```
Graph g{
#Class instances
#
struct1 [shape=record,label="{m1:map|name=Montpellier\n}"];
struct2 [shape=record,label="{S2:Street|name=Verdun\nlength=150\n}"];
struct3 [shape=record,label="{B3:Boulevard|name=Jeu-de_paume\nlength=4000\n}"];
struct4 [shape=record,label="{B4:Boulevard|name=Jean_Jaures\nlength=3000\n}"];
struct5 [shape=record,label="{P5:Pedestrian|name=Nelson_Mandela\nlength=900\n}"];
struct6 [shape=record,label="{G6:Garden|name=Tabalizt\n}"];
struct7 [shape=record,label="{G7:Garden|name=Akermus\n}"];
struct8 [shape=record,label="{S8:Square|name=Lekhmis\n}"];
#Root class references
#
struct1--struct2 [arrowtail=diamond,arrowhead=none,dir=both];
struct1--struct3 [arrowtail=diamond,arrowhead=none,dir=both];
struct1--struct4 [arrowtail=diamond,arrowhead=none,dir=both];
struct1--struct5 [arrowtail=diamond,arrowhead=none,dir=both];
struct1--struct6 [arrowtail=diamond,arrowhead=none,dir=both];
struct1--struct7 [arrowtail=diamond,arrowhead=none,dir=both];
struct1--struct8 [arrowtail=diamond,arrowhead=none,dir=both];
#Other references
#
struct2 -- struct6 [arrowhead=open,dir=both,label="border"];
struct2 -- struct7 [arrowhead=open,dir=both,label="border"];
```

1. Solveur Absocn: http://www.cril.univ-artois.fr/~lecoutre/software.html

```
struct2 -- struct8 [arrowhead=open,dir=both,label="border"];
struct2 -- struct3 [arrowhead=open,dir=forward,label="meet"];
struct2 -- struct4 [arrowhead=open,dir=forward,label="meet"];
struct3 -- struct6 [arrowhead=open,dir=both,label="border"];
struct3 -- struct7 [arrowhead=open,dir=both,label="border"];
struct3 -- struct8 [arrowhead=open,dir=both,label="border"];
struct3 -- struct2 [arrowhead=open,dir=forward,label="meet"];
struct3 -- struct4 [arrowhead=open,dir=forward,label="meet"];
struct4 -- struct6 [arrowhead=open,dir=both,label="border"];
struct4 -- struct7 [arrowhead=open,dir=both,label="border"];
struct4 -- struct8 [arrowhead=open,dir=both,label="border"];
struct4 -- struct2 [arrowhead=open,dir=forward,label="meet"];
struct4 -- struct3 [arrowhead=open,dir=forward,label="meet"];
struct5 -- struct6 [arrowhead=open,dir=both,label="border"];
struct5 -- struct7 [arrowhead=open,dir=both,label="border"];
struct5 -- struct8 [arrowhead=open,dir=both,label="border"];
struct5 -- struct2 [arrowhead=open,dir=forward,label="meet"];
struct5 -- struct3 [arrowhead=open,dir=forward,label="meet"];
struct5 -- struct4 [arrowhead=open,dir=forward,label="meet"];
}
```

La figure 6.3 montre un modèle conforme au méta-modèle de la figure 6.2. Ce modèle a été généré automatiquement en respectant les spécifications du fichier de configuration du listing 6.6.

6.1.4 Quelques Options

Il existe quelques autres options qu'offre $\mathcal{G}$RIMM.

— Casser les symétries : optimise les symétries du problème. La valeur par défaut est 0. ◁ sec. 4.1

Listing 6.10 – Casser les symétries du problème

```
java -jar grimm.jar -mm=*.ecore -root=class ... -sym=1
```

— Ajouter un fichier OCL.

— Générer la $i^{ème}$ solution : demande la solution numéro i trouvée par le solveur. La valeur par défaut est 1.

Listing 6.11 – Génération de la $i^{ème}$ solution

```
java -jar grimm.jar -mm=*.ecore -root=class ... -sol=i
```

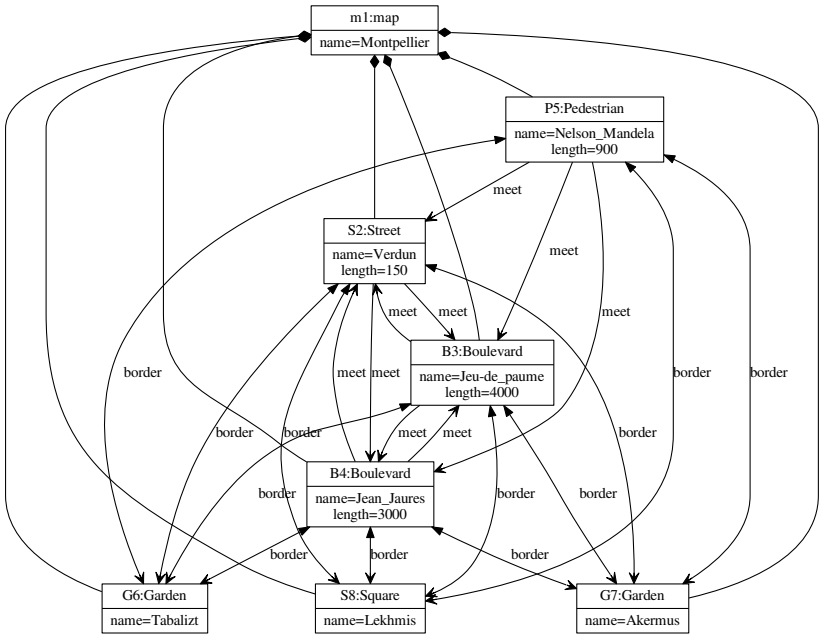


FIGURE 6.3 – Modèle conforme au méta-modèle en figure 6.2

6.1.5 Accessibilité et ergonomie

6.1.5.1 $\mathcal{G}$rimm sur le web

L'outil $\mathcal{G}$RIMM est disponible et directement utilisable en ligne [2]. Vous pouvez choisir un méta-modèle dans une liste fournie et générer des modèles conformes aux formats dot/pdf ou XMI. La figure 6.4 est une capture d'écran de la page web de l'outil disponible en ligne.

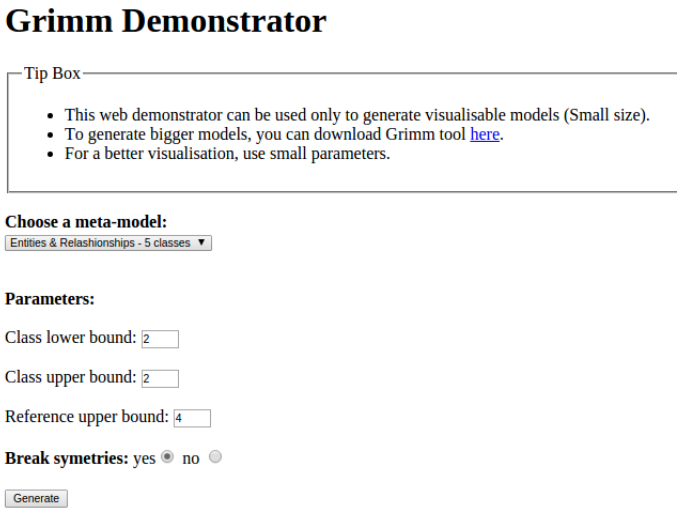


FIGURE 6.4 – Capture d'écran du démonstrateur $\mathcal{G}$RIMM disponible sur le web

6.1.5.2 Plugin $\mathcal{G}$rimm : intégration à Eclipse

Pour améliorer l'ergonomie de $\mathcal{G}$RIMM, nous avons développé un plugin intégré à l'environnement Eclipse. Ce dernier fonctionne exactement de la même manière que la version en lignes de commande et offre les mêmes options.

6.2 Les variantes de $\mathcal{G}$rimm

Les deux études de cas du chapitre 4 (Génération d'Instances Pertinentes, Réalistes et Vraisemblables) ont donné lieu chacune au développement d'une variante de $\mathcal{G}$RIMM. La particularité de ces deux variantes est leur capacité à prendre en paramètres des lois de probabilités liées à des métriques spécifiques à un domaine.

Le processus de l'outil évolue pour prendre en compte ce nouveau paramètre. La figure 6.7 montre les étapes de $\mathcal{G}$ЯRIMM ($\mathcal{G}$enerating ЯandomizedandRelevantInstances of

2. $\mathcal{G}$RIMM online: `http://info-demo.lirmm.fr/grimm/`

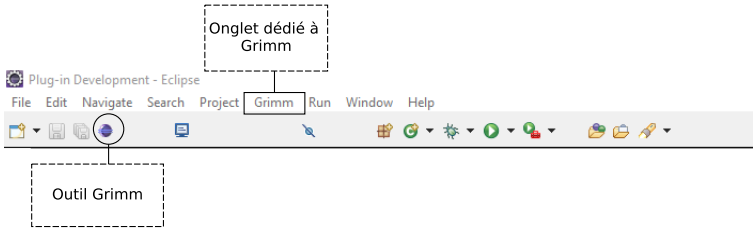


FIGURE 6.5 – Intégration de $\mathcal{G}$RIMM à la barre d'outil de Eclipse

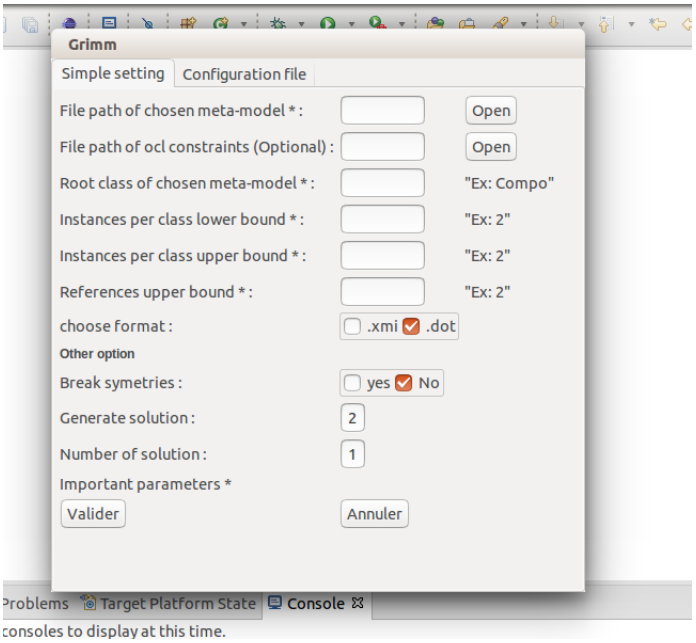


FIGURE 6.6 – Capture d'écran du plugin

Meta-Models). Nous voyons l'apparition de deux nouveaux paramètres et d'une étape de simulation de lois de probabilité.

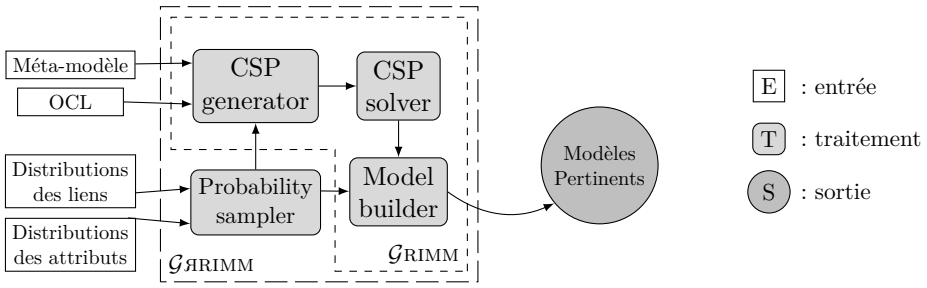


FIGURE 6.7 – Processus de l'outil $\mathcal{G}$ЯRIMM ($\mathcal{G}$enerating ЯandomizedandRelevant Instances of Meta-Models).

6.2.1 $\mathcal{G}$rimm pour la génération de programmes java

Dans cette variante nous générons des squelettes de projets java conformes à un méta-modèle que nous fournissons. Les contraintes OCL de ce méta-modèle sont directement intégrées à l'outil. L'utilisateur peut générer des fichiers de configuration et spécifier des lois de probabilités parmi le catalogue de lois supportées par l'outil.

Listing 6.12 – Génération d'un fichier de configuration pré-rempli pour java

```
java -jar grimm4java.jar -mm=MyJava.ecore -root=Project
```

Listing 6.13 – Extrait d'un fichier de configuration pour la génération de programmes java.

```
% -- -- -- -- -- -- -- -- -- -- -- -- --
% Number of instances for Classes
Package=20
PrimitiveType=8
Class=100
...
Interface=60
Variable=200
% -- -- -- -- -- -- -- -- -- -- -- -- --
% Domains of attributes
Variable/name=-13..13
Variable/visibility=1..4
...
Method/name=-13..13
Method/visibility=1..4
% -- -- -- -- -- -- -- -- -- -- -- -- --
% Distributions of references
Package/packClass->Expo(8.38)
Package/packInter->Expo(8.08)
```

```
...
Class/ownedVars->Norm(3.46,2.09)
Class/ownedMethods->Expo(7.6)
...
```

6.2.2 $\mathcal{G}$rimm pour la génération de graphes de Scaffold

Une deuxième variante concerne la génération de graphes de Scaffold. Là encore le méta-modèle est fourni avec l'outil et l'utilisateur a juste à choisir le nombre de nœuds (`-n`) et d'arcs (`-e`) de son graphe ou bien donner la distribution des degrés.

La figure 6.8 et le listing 6.15 montrent un graphe de Scaffold généré automatiquement.

Listing 6.14 – Génération d'un graphe de Scaffold

```
java -jar grimm4scaffold.jar -n=12 -e=13
```

Listing 6.15 – Contenu d'un fichier dot représentant un graphe de Scaffold généré.

```
Graph g{
#Nodes
#
1;
2;
3;
4;
5;
6;
7;
8;
9;
10;
11;
12;
#Contig edges
#
1--2 [penwidth=10];
3--4 [penwidth=10];
5--6 [penwidth=10];
7--8 [penwidth=10];
9--10 [penwidth=10];
11--12 [penwidth=10];
#Other edges
#
1--11 [label="11"] ;
2--5 [label="13"] ;
3--5 [label="2"] ;
4--5 [label="14"] ;
5--8 [label="34"] ;
6--9 [label="9"] ;
```

```
7--11 [label="31"] ;
}
```

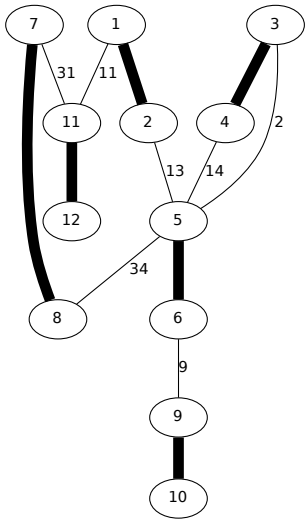


FIGURE 6.8 – Un graphe de Scaffold généré automatiquement

6.3 Assistance au design de méta-modèles

Nous rappelons que l'outil $\mathcal{G}$RIMM est développé pour générer des modèles dont le rôle est de remplir deux objectifs principaux :

1. Servir de données de tests réalistes pour les transformations de modèles. Cet objectif a été mis en œuvre dans les études de cas du chapitre 4, particulièrement pour la génération de graphes de scaffold respectant les métriques de leur domaine.
2. Assister les concepteurs de méta-modèles durant leur tâche. En effet, il est très intéressant pour le concepteur, de générer des instances au cours du processus de design du méta-modèle, car elles aident à détecter très tôt dans un processus dirigée par les modèles d'éventuelles erreurs de modélisation.

La génération entièrement automatisée de notre outil, la prise en main rapide et la visualisation graphique des modèles permettent au designer de méta-modèles de générer des instances pour essayer de détecter des erreurs, puis de les corriger. La figure 6.9 illustre le processus de conception de méta-modèles assistée par $\mathcal{G}$RIMM.

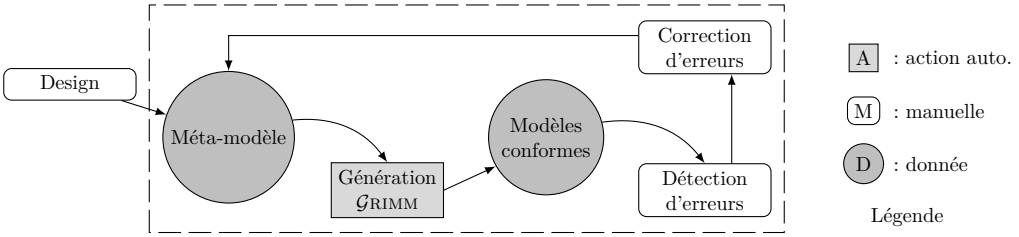


FIGURE 6.9 – Processus de conception de méta-modèles assistée par $\mathcal{G}$RIMM

Un groupe d'étudiants s'est vu donner pour tâche de concevoir des méta-modèles Ecore en se servant de $\mathcal{G}$RIMM pour les assister. Le but de cette expérience est de vérifier l'utilisabilité et l'ergonomie de l'outil, mais surtout de lui apporter des améliorations en se basant sur les commentaires de ces utilisateurs "volontaires".

6.3.1 Utilisateurs

Les utilisateurs étaient les Master 2 en génie logiciel de l'université de Montpellier (promotion 2015).

— Nombre d'utilisateurs : 30 personnes.
— Durée du test : 3 heures d'affilée.
— Date du l'expérience : décembre 2014.

6.3.2 Protocole

Les utilisateurs devaient suivre les étapes suivantes :

1. Créer une première version d'un méta-modèle en choisissant un thème parmi une liste proposée (architecture de maison, organigramme d'entreprise, ...), ou bien proposer son propre thème.
2. Utiliser $\mathcal{G}$RIMM pour générer un modèle conforme à la version courante du méta-modèle.
3. Vérifier si le modèle permet de trouver une erreur de modélisation, alors dire que l'étape a été utile puis aller à l'étape 4, sinon aller en 2 ou 5.
4. Modifier le méta-modèle de façon à corriger l'erreur détectée puis aller à l'étape 2.
5. Répondre à un questionnaire sur l'utilisation de l'outil et son ergonomie.

Le questionnaire porte sur les points suivants :

— Pour chaque étape, dire si elle est utile ou pas.
— Si une erreur a été détectée, préciser la partie du méta-modèle qu'elle concerne.
— Donner des remarques ou suggestions pour améliorer l'ergonomie de l'outil.
— Dire si l'outil vous a satisfait.

6.3.3 Résultats et analyse

Les figures 6.10 et 6.11 compilent les résultats des réponses des utilisateurs au questionnaire. Nous pouvons observer qu'ils étaient globalement satisfaits de leur utilisation de $\mathcal{G}$RIMM. Les étapes de génération ont permis de détecter beaucoup d'erreurs car ils considèrent majoritairement que les modèles générés étaient utiles pour la correction des méta-modèles.

Les erreurs détectées sont liées aux points suivants :

— Mauvais nommage des éléments.

— Typage incorrect d'attributs.

— Hiérarchie d'héritage manquante ou inadéquate.

— Oubli d'attributs ou de classes.

— Mauvais choix de cardinalités pour les références.

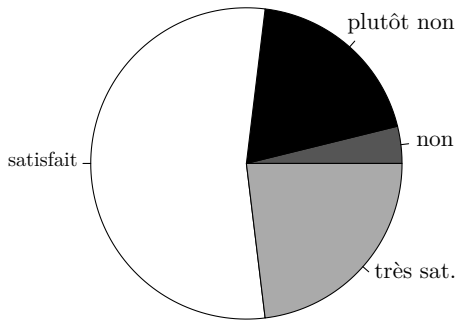


FIGURE 6.10 – Degré de satisfaction.

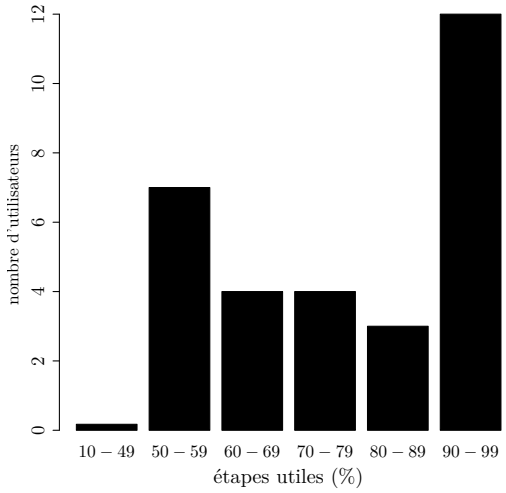


FIGURE 6.11 – Étapes utiles.

6.3.4 Menaces à la validité

Ces résultats sont évidemment à prendre avec prudence car le groupe d'utilisateurs est constitué exclusivement d'étudiants en Master génie logiciel à l'université de Montpellier ayant suivi le cours sur l'IDM. On peut penser qu'ils ne sont pas toujours parfaitement honnêtes et objectifs quand on leur demande de juger le travail de leur professeur !

Par ailleurs, les éléments les plus bons commencent l'expérience avec un méta-modèle plutôt correct et donc n'ont pas besoin de beaucoup d'étapes pour finir de le corriger. À l'inverse, les méta-modèles les moins bons nécessitent beaucoup d'étapes car il y a beaucoup d'éléments à revoir. Ceci peut biaiser les résultats relevés.

6.3.5 Amélioration de l'outil

Malgré les menaces à la validité de ces résultats, l'expérience a été bénéfique pour l'ergonomie de l'outil. Elle a permis d'apporter beaucoup de nouveautés et d'améliorations.

— Ajout des fichiers de configuration pour plus de contrôle et de précision sur le nombre d'instances par classe et les domaines des attributs.

— Développement d'un plugin intégré à l'environnement Eclipse en parallèle de l'interface en ligne de commande.

— Correction du traitement des références bidirectionnelles.

6.4 Conclusion

Ce chapitre a présenté l'outil $\mathcal{G}$RIMM ($\mathcal{G}$eneRating Instances of Meta-Models). Il s'agit de l'outil de génération de modèles développé pour implémenter les contributions de cette thèse. Les entrées, sorties et étapes de fonctionnement de l'outil ont été décrites. Les commandes, les options et des exemples de modèles générés ont été donnés.

En plus de la version générique de l'outil, nous avons introduit $\mathcal{G}$ЯRIMM ($\mathcal{G}$enerating ЯandomizedandRelevant Instances of Meta-Models). En effet, les contributions et les deux études de cas du chapitre 4 ont permis le développement d'une importante amélioration de $\mathcal{G}$RIMM. Cette dernière se présente sous la forme de deux versions qui prennent en paramètre -en plus de ceux de base- des lois de probabilités liées à des métriques des deux domaines étudiés.

Une expérience avec un groupe d'utilisateurs de $\mathcal{G}$RIMM a été menée dans le but de montrer l'utilisabilité et l'ergonomie de l'outil. L'analyse des résultats et des retours du groupe d'utilisateurs a permis d'apporter plusieurs améliorations à $\mathcal{G}$RIMM.

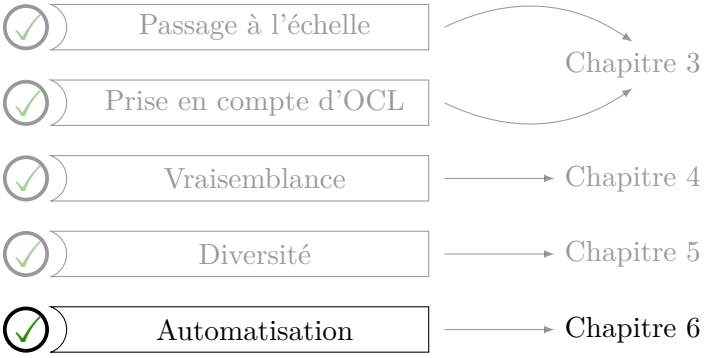


FIGURE 6.12 – Caractéristiques de la génération de modèles abordées par le chapitre 6

Chapitre 7

Conclusion

> Toutes les histoires ont une fin, mais chaque fin est le début de quelque chose de nouveau.
>
> Anonyme

Synopsis

Préambule

Ce chapitre conclut notre manuscrit et ouvre la voie à de nouvelles aventures scientifiques. Une conclusion générale qui rappelle les principales contributions et réalisations de la thèse est donnée par la section 7.1. Ensuite, la section 7.2 liste les pistes d'amélioration et les travaux futurs dont certains ont d'ores et déjà été entamés.

Dans ce manuscrit, nous avons abordé la problématique de génération automatique d'instances de méta-modèles. En Ingénierie Dirigée par les Modèles, générer des instances est une solution qui tente de répondre à deux questions primordiales :

1. Comment garantir la validité des méta-modèles ou les modèles logiciels ? Et surtout comment s'assurer qu'ils capturent bien le domaine qu'ils modélisent, ni plus ni moins ?
2. Où trouver des données de test pour les transformations de modèles, sachant que les challenges sont de se procurer ces données en grande quantité et qu'elles soient vraisemblables, pertinentes et diversifiées ?

Nous avons développé une approche de génération de modèles conformes à des méta-modèles basée sur la programmation par contraintes. L'approche proposée respecte les critères suivants :

— **Passage à l'échelle** Capacité de l'approche à générer des modèles de grandes tailles conformes à des méta-modèles de grandes tailles.

— **Contraintes OCL des méta-modèles** Prise en compte et respect des contraintes OCL accompagnant un méta-modèle lors du processus de génération.

— **Vraisemblance et pertinence des solutions** Qualités que doivent posséder les modèles pour se rapprocher des caractéristiques des modèles réels.

— **Diversité des modèles générés** Exigence de génération de modèles différents les uns des autres et diversifiés.

— **Automatisation de l'approche** Outil entièrement automatique qui génère des modèles avec très peu d'intervention humaine.

Dans le deuxième chapitre de ce manuscrit de thèse, nous avons dressé un état de l'art des approches et outils de génération de modèles. Ces différents travaux ont été décrits, analysés, puis comparés selon les critères précédents et selon d'autres critères, liés à l'expressivité, à la facilité d'accès et d'utilisation.

7.1 Résumé des contributions

Les chapitres 3 à 7 ont exposé les contributions apportées par ce manuscrit. La figure 7.1 résume par un schéma les différentes contributions en les associant à leurs chapitres respectifs.

Premièrement, nous avons présenté un nouveau modèle pour la formalisation des méta-modèles Ecore et de leurs contraintes OCL en programmation par contraintes (CSP). La modélisation que nous proposons est efficace car elle utilise des bonnes pratiques de modélisation en CSP qui permettent d'obtenir des CSP efficaces. Parmi ces bonnes pratiques, nous pouvons citer : la réduction ou la limitation drastiques du nombre de variables, l'utilisation de domaines expressifs et optimaux en tailles, la non création de contraintes lorsque celles-ci

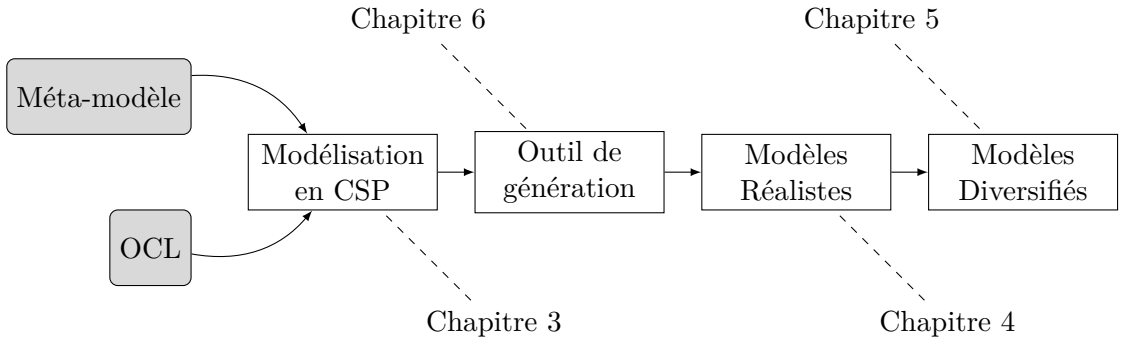


FIGURE 7.1 – Schématisation du plan de la thèse.

peuvent être exprimées par les domaines, et l'utilisation massive des contraintes globales notamment pour la modélisation des contraintes OCL. Par la suite, nous avons mené des expérimentations dans le but de montrer l'efficacité de notre approche par rapport à une approche existante en CSP, puis en utilisant un benchmark de plusieurs méta-modèles et de contraintes OCL.

Deuxièmement, nous avons proposé une approche pour l'amélioration de la pertinence et la vraisemblance des modèles générés. À cet escient, des métriques spécifiques à un méta-modèle ou à un domaine sont utilisées pour inférer des lois de probabilités usuelles. Leur simulation permet ensuite de générer des modèles dont les éléments suivent ces lois. L'utilisateur peut contrôler le processus de génération en proposant des lois de probabilités qui sont liées à une utilisation des données générées. Deux études de cas viennent étayer notre propos et valider l'approche. D'un côté, nous générons des squelettes de programmes java en utilisant des métriques de code orienté objet, et d'un autre, nous produisons des graphes de Scaffold (utilisés en bioinformatique) en nous basant sur l'observation des degrés et des pondérations liés à ces graphes.

Troisièmement, nous nous sommes intéressés à la diversité des modèles que nous générons. Pour cela, nous avons mis au point une nouvelle méthode pour la comparaison de deux modèles basée sur des métriques de distances. Nous avons proposé trois métriques basées sur des distances mathématiques et de graphes que nous avons adaptées aux modèles. Pour chacune des distances, nous donnons les transformations nécessaires à deux modèles pour être comparés. Par ailleurs, les distances sont comparées selon leur complexité théorique et leur expressivité. Le second challenge était de pouvoir comparer non pas deux, mais tout un ensemble de modèles, dans le but d'en sélectionner les meilleurs, c'est-à-dire, ceux qui donnent la meilleure diversité. Comparer un ensemble de modèles deux à deux donne lieu à une matrice de distance. Nous nous sommes servi de techniques de clustering de matrices pour identifier puis sélectionner les modèles qui offrent la meilleure diversité parmi l'ensemble. Enfin, nous avons expérimenté l'algorithmique génétique dans le but d'améliorer la diversité d'un ensemble de modèles généré.

Quatrièmement, toutes les contributions de la thèse décrites plus haut ont été implantées dans un outil de génération de modèles nommé $\mathcal{G}$RIMM [1](ou bien $\mathcal{G}$ЯRIMM pour sa version basée sur les lois de probabilités). Cet outil offre différents modes d'utilisation (ligne de commande, utilisation sur le web et plugin Eclipse) et différentes options. Une expérience avec un groupe d'utilisateurs a été décrite. Elle a permis d'apporter des améliorations substantielles à $\mathcal{G}$RIMM.

7.2 Perspectives

Les contributions de la thèse et les limites observées ouvrent la voie à diverses améliorations et perspectives. Dans ce qui suit, nous donnons les différentes pistes que nous envisageons pour le futur (elle sont illustrées par la carte mentale de la figure 7.2).

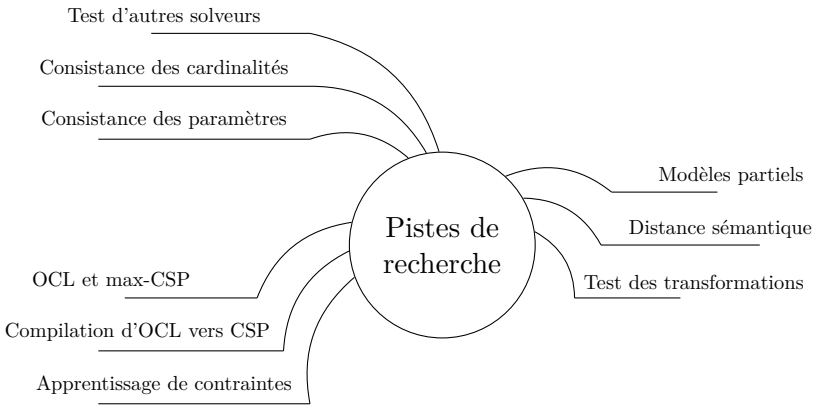


FIGURE 7.2 – Carte mentale des pistes de recherche envisagées

7.2.1 Court terme

Test d'autres solveurs

La formalisation d'un méta-modèle et de ses contraintes OCL en CSP est écrite pour produire des fichiers au format xcsp pris en charge par le solveur que nous utilisons, en l'occurrence *abscon*. Néanmoins, d'autres solveurs existent (voir section 2.4). Il serait intéressant pour nous de tester l'efficacité de notre approche avec d'autres solveurs. Ceci nous permettra d'un côté d'accéder à des types de contraintes que n'offre pas *abscon* ce qui augmentera les parties du langage OCL que nous traitons, et d'un autre côté pourrait améliorer la performance. La difficulté de cette tâche est liée uniquement au temps

1. Utiliser $\mathcal{G}$RIMM en ligne.: `http://info-demo.lirmm.fr/grimm/`

de développement. En effet, chaque solveur utilise son propre format d'entrée et le tester nécessitera l'écriture d'un traducteur vers le format désiré.

Inconsistance des contraintes OCL

Lorsque le solveur n'arrive pas à instancier le méta-modèle et que ce n'est pas à cause des cardinalités ou des paramètres de l'utilisateur, on en déduit que la cause est sans doute une ou plusieurs contraintes OCL mal exprimée ou inadaptée. Pour que l'utilisateur puisse les corriger il faudrait que l'outil soit capable d'isoler la ou les contraintes OCL qui provoquent l'échec de la résolution. Pour ce faire, nous estimons que la meilleure solution est de passer par une modélisation en Max-CSP [65] qui permet la violation d'un certain nombre de contraintes dites "molles" et tente de maximiser le nombre de contraintes satisfaites. Ainsi, les contraintes OCL qui posent problème sont identifiées et peuvent être corrigées.

Génération de solutions partielles

L'un des objectifs principaux de notre approche est d'assister les concepteurs de méta-modèles dans leur tâche. Pour cela, nous générons des instances et un expert corrige son méta-modèle et les contraintes OCL qui l'accompagnent jusqu'à se conformer à son souhait de départ. Malheureusement, lorsque le CSP généré n'est pas satisfait, aucune instance du méta-modèle n'est générée. Ceci implique que l'expert n'a aucune matière pour entamer son travail de correction du méta-modèle et donc sa tâche se complique. Une idée intéressante est de générer des modèles partiels conformes aux parties valides et instanciables du méta-modèle. Tout comme pour le point précédent, l'utilisation d'une modélisation par Max-CSP peut être envisagée.

7.2.2 Moyen terme

Inconsistance des cardinalités et des paramètres utilisateurs

L'impossibilité d'instancier un méta-modèle est dû, la plupart des cas, à une mauvaise combinaison des cardinalités de référence et des choix de paramétrage par l'utilisateur. Par exemple, nous désirons instancier une classe `maison` qui nécessite obligatoirement 4 instances de `mur`, mais en réalité nous ne disposons que de 3 instances de `mur`. Des travaux qui vérifient l'inconsistance des cardinalités d'un schéma relationnel ou d'un diagramme de classe UML utilisant la programmation linéaire existent déjà [10],[11]. Dans notre approche, on pourrait imaginer un système équivalent adapté aux méta-modèles Ecore qui aurait les objectifs suivants :

— Détecter l'inconsistance des cardinalités ou des paramètres très tôt dans le processus sans générer le CSP et sans faire appel au solveur de contraintes.

— Aider l'utilisateur à faire le bon choix en l'orientant vers les intervalles de valeurs correctes ou directement en corrigeant ses choix.

Test de transformations de modèles

Développer des approches automatiques pour le test des transformations de modèles nécessite, selon Baudry et al. [5], trois étapes : (1) générer des données de test, c'est-à-dire, des modèles. (2) définir des critères de test. (3) Construire un oracle capable de certifier que le programme est correct pour telle ou telle donnée.

Notre approche de génération automatique de modèles interviendra principalement à la première et à la deuxième étape. Elle aura pour objectif de fournir des modèles de test qui respecteront les critères définis. Autrement dit, les modèles générés devront être diversifiés pour une meilleure couverture de l'espace des solutions, d'un côté, mais aussi être pertinents et en adéquation avec des contraintes définies par la transformation à tester.

Extension du traitement de OCL

Concernant OCL, nous avons montré dans la section 3.2 que les briques pour formaliser la plupart des constructions du langage en CSP sont disponibles et que leur combinaison est possible. $\mathcal{G}$RIMM intègre déjà une bonne partie de ces constructions et des expérimentations ont montré que l'approche reste assez efficace malgré l'ajout de l'OCL. Néanmoins, la prise en compte des contraintes OCL des méta-modèles n'est pas complète et nécessite un important travail de développement logiciel. Nous projetons d'écrire un compilateur de OCL vers CSP permettant de traiter n'importe quelle contrainte OCL aussi complexe soit-elle. Le compilateur inclura les opérations arithmétiques et logiques parenthésées complexes et imbriquées et les opérations sur les collections imbriquées ou chaînées. Pour les autres constructions telle la navigation de références, le chaînage est d'ores et déjà possible.

7.2.3 Long terme

Prise en compte de la sémantique par les distances

Les distances que nous avons présentées au chapitre 5 s'intéressent prioritairement à l'aspect structurel des modèles. Néanmoins, nous avons montré qu'il existe également des techniques de comparaison de modèles logiciels basées sur la sémantique des éléments. Elles comparent par exemple les valeurs des attributs, les noms des éléments, etc. Nous pensons que nos distances et ce type de techniques peuvent très bien se conjuguer et offrir deux points de vue différents pour la comparaison des modèles. On pourrait par exemple imaginer un algorithme génétique à objectif multiple : distances structurelles et comparaison sémantique.

Apprentissage de contraintes OCL

À long terme, nous souhaitons nous intéresser à l'apprentissage de contraintes OCL. En effet, actuellement lorsque l'utilisateur génère des modèles dans le but de valider son méta-modèle, $\mathcal{G}$RIMM génère des instances et l'utilisateur s'occupe de la correction du

méta-modèle, de l'ajout ou de la suppression de contraintes OCL. L'idée est qu'à terme, le processus de validation des méta-modèles devienne entièrement automatique. La seule tâche de l'utilisateur sera d'identifier les parties des modèles générés qui sont incorrectes ou incomplètes. Par exemple, lorsqu'on génère des graphes de Scaffold l'utilisateur va identifier tous les arcs dont le poids est inférieur à 0 et la machine va déduire qu'une contrainte OCL qui stipule que le poids d'un arc est positif doit être ajoutée au méta-modèle.

Des travaux traitant de l'apprentissage de programmes contraints existent `Quacq` [7], `Modelseeker` [6]. Ils permettent à la machine d'apprendre des contraintes CSP à partir d'exemples d'instanciation positifs et négatifs ou bien à travers un processus interactif entre la machine qui soumet des bouts de solutions (requêtes) que doit approuver l'utilisateur.

Le frein à l'application d'une telle approche est qu'une contrainte OCL nécessite un ensemble de contraintes CSP complexes (plusieurs contraintes, contraintes globales). À l'heure actuelle, les approches d'apprentissage de contraintes se limitent à des contraintes assez simples (addition, multiplication, test logiques). Une autre difficulté réside dans la retranscription des contraintes apprises vers des contraintes "sémantiques", donc ayant du sens pour l'utilisateur.

Annexes

Annexe A

Analyse et valorisation de données en R

Synopsis

Préambule

Cette annexe décrit quelques unes des fonctions R qui nous ont servi dans les chapitres précédents. Les scripts donnés ici concernent des techniques que nous avons utilisées pour : (1) répondre à une problématique de recherche, comme par exemple, le clustering de matrices de distances, (2) ou bien dans le but de représenter des données expérimentales.

A.1 Déduction de lois de probabilités théoriques

La déduction de lois de probabilités à partir d'échantillons empiriques utilise la fonction `fitdistr`(). Voici un script R (A.1) qui prend en entrée un échantillon et qui trouve et dessine la loi Log-normale qu'il approche le plus. Le résultat est visible à la figure A.1.

Listing A.1 – script R utilisant la fonction `fitdistr`().

```
#Paquets necessaires
require(tikzDevice)
require(MASS)

#Echantillon a analyser
g=AVGATTR

#Methode fitDistr()
fit=fitdistr(g,"lognormal")

#Parametres de la loi
t1 <- paste("mu=",round(fit$estimate[1],digits=3),sep="")
t2 <- paste("sigma=",round(fit$estimate[2],digits=3),sep="")
t <- paste(t1,t2,sep=", ")
titre <- paste("Log-normale: ",t,sep="")

#Superposition des deux echantillons
#
#Sortie en tikz
tikz('fitAVGATTR.tex',standAlone=FALSE)
h<-hist(g, breaks=10, density=10, col="lightgray",
        ylim=c(0,0.5),xlab="",ylab="Densite", main=titre,prob=T)
curve(dlnorm(x,fit$estimate[1],fit$estimate[2]),col="red",
        from=0,to=8,lwd=5,add=T)
dev.off()

#Sortie en pdf
pdf('fitAVGATTR.pdf')
h<-hist(g, breaks=10, density=10, col="lightgray",
        ylim=c(0,0.5),xlab="",ylab="Densite", main=titre,prob=T)
curve(dlnorm(x,fit$estimate[1],fit$estimate[2]),col="red",
        from=0,to=8,lwd=5,add=T)
dev.off()
```

A.2 Clustering de matrice de distances

Le listing A.2 montre une fonction R qui à partir d'une matrice de distances entre modèles trouve le nombre de clusters de la matrice et retourne une liste contenant les modèles les plus représentatifs de la matrice.

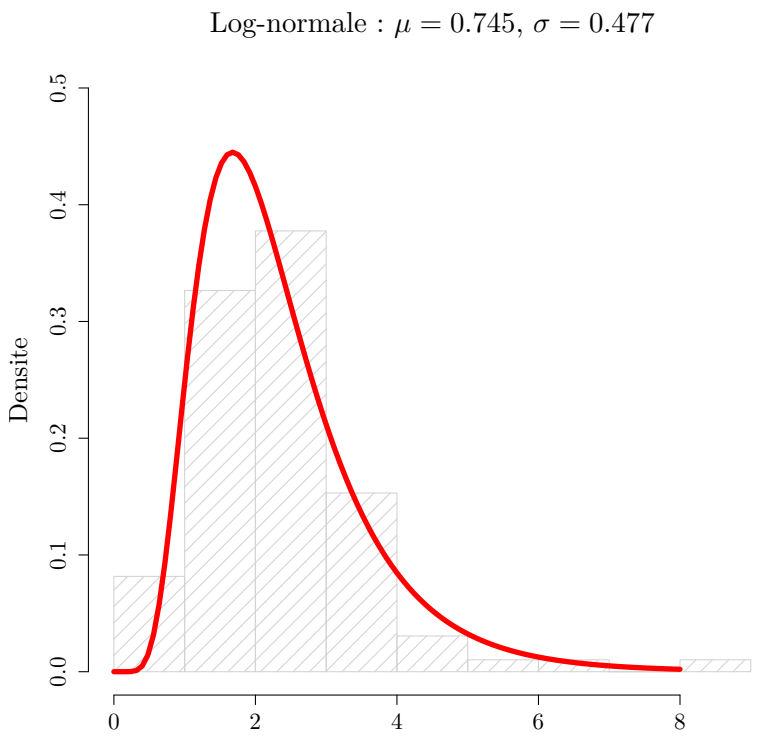


FIGURE A.1 – Résultat du script précédent

Listing A.2 – Une fonction R pour le clustering d'une matrice de distances.

```
#Two Parameters
# data: matrice de distance .csv
# nbClus: nombre de clusters
giveMeClusters <- function(data,nbClus){
    matrix1 <-read.table(data, header=FALSE, sep=",")
    m <- as.matrix(matrix1)

    #Clustering avec K-means
    foo <- kmeans(m,nbClus,iter.max=200)
    print(foo)
    j <-1

    vec <- vector(mode="integer", length=nbClus)
    now <- vector(mode="integer", length=nbClus)

    for(i in 1:length(res$cluster)){

        if (! res$cluster[i] %in% now)
        {
            vec[j] <- i
            j <- j+1
        }

        now[i]=res$cluster[i]
    }
    return(vec)
}
```

Listing A.3 – script bash pour trouver les clusters d'une matrice.

```
#!/bin/bash

if [ -z $1 -a -z $2 ]; then

  echo "Requires two arguments
        1: a distance matrix .csv file
        2: number of clusters = number of result models"
else
        R --slave --no-save <<EOF
        source('kmeans.r')

        bar=giveMeClusters("$1",$2)
        print(bar)
EOF
fi
```

A.3 Diagrammes de Voronoi

Le listing A.4 montre une fonction que nous avons écrit. Elle dessine des diagrammes de Voronoi à partir d'une matrice de distances entre modèles. Une script bash qui prend en paramètre un fichier csv contenant la matrice et qui produit un fichier pdf ou tikz en sortie (figure A.2) est montré par le listing A.5.

Listing A.4 – Une fonction R pour le dessin de diagrammes de Voronoi.

```
#Paquets requis
library("tripack")
require(tikzDevice)
library(MASS)

#Three Parameters
# data: matrice de distance .csv
# filepdf: sortie .pdf
# filetex: sortie .tex
voronoiIt <- function(data,filepdf,filetex){

#Lire les donnees
matrix1 <-read.table(data, header=FALSE, sep=",")
m <- as.matrix(matrix1)
d <- m
r <- sammon(d)

# Calculer la tessellation de Voronoi
x <- r$points
dt <- data.frame(x)
names(dt) <- c('x', 'y')
dt$labels <- paste0('m', 1:nrow(x))

#Dessiner le diagramme
#Pdf
pdf(filepdf)
plot(voronoi.mosaic(x[,1], x[,2]),axis=TRUE,
        main="",xlab="",ylab="",title="")
points(x,pch=1)
text(dt$x,dt$y,dt$labels, pos=3)
dev.off()

#Tikz
tikz(filetex)
plot(voronoi.mosaic(x[,1], x[,2]),axis=TRUE,
        main="",xlab="",ylab="",title="")
points(x,pch=1)
text(dt$x,dt$y,dt$labels, pos=3)
dev.off()
}
```

Listing A.5 – script bash pour dessiner le diagramme de Voronoi d'une matrice de distances.

```
#!/bin/bash
if [ -z $1 -a -z $2 ]; then

  echo "Requires three arguments
        1: a distance matrix .csv file
        2: an output pdf file
        3: an output tex file"
else
        R --slave --no-save <<EOF
        source('voronoi.r')
        foo=voronoiIt("$1","$2","$3")
        print(foo)
EOF
fi
```

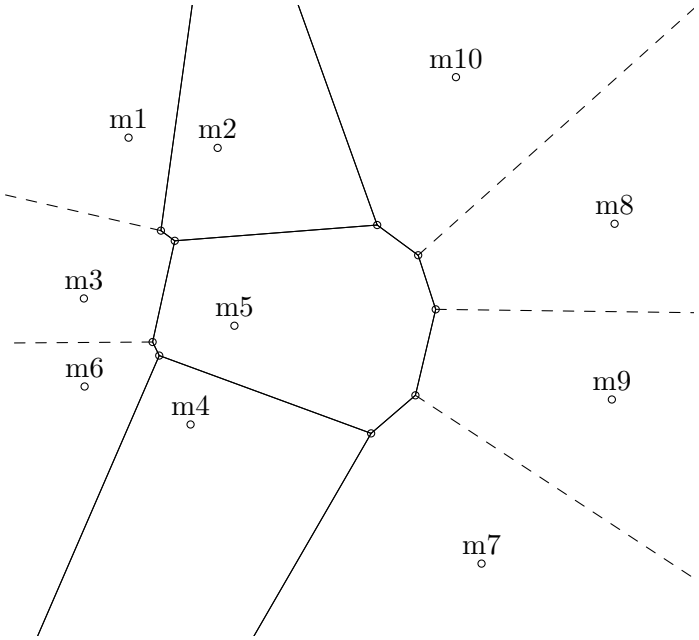


FIGURE A.2 – Diagramme de Voronoi pour une matrice de réseaux de Petri.

A.4 Superposition de Boîtes à moustaches

Le script du listing A.6 permet de superposer deux boites à moustaches (Box plot) sur le même plan. Il prend en paramètre un fichier de données contenant 2 échantillons (généralisable à n) comme illustrée au listing A.7. Le résultat du script est visible à la figure A.3.

Listing A.6 – Un Script R permettant de superposer deux boites à moustaches.

```
#Paquet requis
require(tikzDevice)

#Lire les donnees
classes <- read.csv("classQG.csv",sep=",",header=FALSE)

#Attacher les donnees dans la base
attach(classes)

#Faire coincider les legendes et les echantillons
boxes <- factor(V1,levels=1:2,labels=c("Github","Qualitas"))

#Sortie
#Pdf
pdf('supperposerClassesBoxPlot.pdf')
par(mfrow=c(1,1))
boxplot(V2~boxes,col=c('grey','red'),ylab="Nombre de classes")
dev.off()

#Tikz
tikz('supperposerClassesBoxPlot.tex',standAlone=FALSE)
par(mfrow=c(1,1))
boxplot(V2~boxes,col=c('grey','red'),ylab="Nombre de classes")
dev.off()
```

Listing A.7 – Un fichiers csv contenant deux échantillons à superposer.

```
echantillon,valeur
1,15
1,49
1,673
1,1098
1,14
1,9
1,5
...
2,427
2,830
2,1657
2,1173
2,503
```

```
2,669
...
```

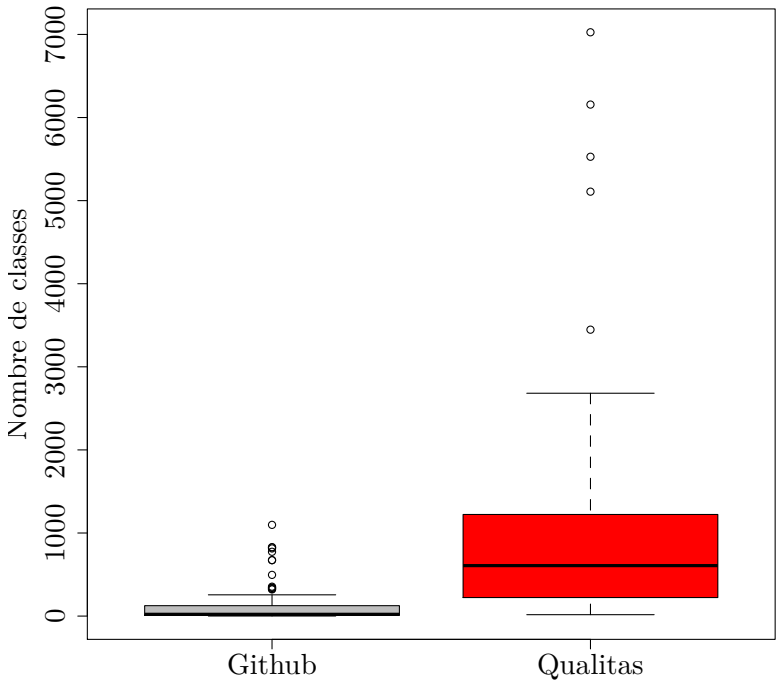


FIGURE A.3 – Deux boites à moustaches superposées.

Bibliographie

[1] R. AL-Msie'Deen. *Reverse Engineering Feature Models from Software Variants to Build Software Product Lines*. PhD thesis, University of Montpellier, 2014.

[2] K. Apt and M. Wallace. *Constraint logic programming using Eclipse*. Cambridge University Press, 2007.

[3] C. Ashworth and M. Goodland. *SSADM : A practical approach*. McGraw-Hill Book Company Limited, 1990.

[4] F. Aurenhammer. Voronoi diagrams — a survey of a fundamental geometric data structure. *CSUR, ACM Computing Surveys*, 23(3) :345–405, 1991.

[5] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6) :139–143, 2010.

[6] N. Beldiceanu and H. Simonis. A Model Seeker : Extracting Global Constraint Models from Positive Examples. In *CP, International Conference on Principles and Practice of Constraint Programming*, pages 141–157, 2012.

[7] C. Bessiere, R. Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C.-G. Quimper, and T. Walsh. Constraint Acquisition via Partial Queries. In *IJCAI, International Joint Conference on Artificial Intelligence*, pages 475–481, 2013.

[8] C. Bessière and P. Hentenryck. To Be or Not to Be ... a Global Constraint. In *CP, International Conference on Principles and Practice of Constraint Programming*, pages 789–794, 2003.

[9] G. Booch. *Object-oriented Analysis and Design with Applications*. Benjamin-Cummings Publishing, 1994.

[10] F. Boufares and H. Bennaceur. Consistency Problems in ER Schemas for Database Systems. *Information Sciences*, 163(4) :263–274, 2004.

[11] F. Boufarès, H. Bennaceur, and A. Osmani. On the Consistency of Cardinality Constraints in UML Modelling. In *ISPE, International Conference on Enhanced Interoperable Systems*, pages 287–292, 2003.

[12] G. Box and M. Muller. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, 29(2) :610–611, 1958.

[13] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon. Metamodel-based Test Generation for Model Transformations : an Algorithm and a Tool. In *ISSRE, International Symposium on Software Reliability Engineering*, pages 85–94, 2006.

[14] C. A. Brown, L. Finkelstein, and P. W. Purdom Jr. Backtrack Searching in the Presence of Symmetry. In *AAECC, International Symposium on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 99–110. 1989.

[15] J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL Class Diagrams using Constraint Programming. In *ICSTW, IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 73–80, 2008.

[16] J. Cabot, R. Clarisó, and D. Riera. Verifying UML/OCL Operation Contracts. In *IFM, International Conference on Integrated Formal Methods*, pages 40–55, 2009.

[17] J. Cabot, R. Clarisó, and D. Riera. On the Verification of UML/OCL Class Diagrams using Constraint Programming. *Journal of Systems and Software*, 93(0) :1–23, 2014.

[18] J. Cadavid, B. Baudry, and B. Combemale. Empirical evaluation of the conjunct use of MOF and OCL. In *EESSMod, Experiences and Empirical Studies in Software Modelling*, 2011.

[19] J. Cadavid, B. Baudry, and H. Sahraoui. Searching the Boundaries of a Modeling Space to Test Metamodels. In *ICST, IEEE Conference on Software Testing, Verification and Validation*, pages 131–140, 2012.

[20] J. J. Cadavid. *Assisting precise Metamodeling.* PhD thesis, Université de Rennes 1, 2012.

[21] M. Cadoli, D. Calvanese, G. De Giacomo, and T. Mancini. Finite Model Reasoning on UML Class Diagrams via Constraint Programming. In *AI IA : Italian Association for Artificial Intelligence Congress*, pages 36–47, 2007.

[22] V. Černỳ. Thermodynamical Approach to the Traveling Salesman Problem : An Efficient Simulation Algorithm. *Journal of Optimization Theory and Applications*, 45(1) :41–51, 1985.

[23] A. Chateau and R. Giroudeau. Complexity and Polynomial-Time Approximation Algorithms around the Scaffolding Problem. In *AlCoB, International Conference on Algorithms for Computational Biology*, pages 47–58, 2014.

[24] A. Chateau and R. Giroudeau. A complexity and approximation framework for the maximization scaffolding problem. *Theorical Computer Science.*, 595 :92–106, 2015.

[25] CHOCO Team. CHOCO : an Open Source Java Constraint Programming Library. Research report, École des Mines de Nantes, 2010.

[26] A. Choquet Geniet. *Les réseaux de Petri : Un Outil de Modélisation Cours et Exercices Corrigés*. Dunod, 2006.

[27] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communication of ACM*, 13(6) :377–387, 1970.

[28] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-Breaking Predicates for Search Problems. In *KR, International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159, 1996.

[29] L. M. De Moura and N. Bjørner. Satisfiability Modulo Theories : Introduction and Applications. *Communications of the ACM Journal*, 54(9) :69–77, 2011.

[30] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm : Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2) :182–197, 2002.

[31] K. Ehrig, J. M. Kister, G. Taentzer, and J. Winkelmann. Generating Instance Models from Meta Models. In *FMOODS, Formal Methods for Open Object-Based Distributed Systems*, pages 156–170, 2006.

[32] K. Ehrig, J. Küster, and G. Taentzer. Generating Instance Models from Meta models. *SoSyM, Software and Systems Modeling*, pages 479–500, 2009.

[33] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel matching for automatic model transformation generation. In *Model Driven Engineering Languages and Systems*, pages 326–340. 2008.

[34] A. Ferdjoukh. $\mathcal{G}$RIMM : un assistant à la conception de méta-modèles par génération d'instances. Poster, GDR GPL, 2015.

[35] A. Ferdjoukh, A.-E. Baert, E. Bourreau, A. Chateau, R. Coletta, and C. Nebut. Instantiation of Meta-models Constrained with OCL : a CSP Approach. In *MODELSWARD, International Conference on Model-Driven Engineering and Software Development*, pages 213–222, 2015.

[36] A. Ferdjoukh, A.-E. Baert, A. Chateau, R. Coletta, and C. Nebut. A CSP Approach for Metamodel Instantiation. In *ICTAI, IEEE International Conference on Tools with Artificial Intelligence*, pages 1044–1051, 2013.

[37] A. Ferdjoukh, E. Bourreau, A. Chateau, and C. Nebut. A Model-Driven Approach to Generate Relevant and Realistic Datasets. In *SEKE, International Conference on Software Engineering & Knowledge Engineering*, pages 105–109, 2016.

[38] F. Fleurey, B. Baudry, P.-A. Muller, and Y. Traon. Qualifying input test data for model transformations. *SoSyM, Software & Systems Modeling*, 8(2) :185–203, 2009.

[39] F. Focacci and M. Milano. Global Cut Framework for Removing Symmetries. In *CP, International Conference on Principles and Practice of Constraint Programming*, pages 77–92, 2001.

[40] L. C. Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3) :215–239, 1978.

[41] A. Fron. *Programmation par Contraintes*. Vuibert, 1994.

[42] F. Galinier. Des algorithmes génétiques pour générer des modèles diversifiés. Master's thesis, Université de Montpellier, France, 2016.

[43] F. Galinier, E. Bourreau, A. Chateau, A. Ferdjoukh, and C. Nebut. Genetic Algorithm to Improve Diversity in MDE. In *META, International Conference on Metaheuristics and Nature Inspired Computing (To appear)*, 2016.

[44] L. Gammaitoni, P. Kelsen, and F. Mathey. Verifying Modelling Languages using Lightning : a Case Study. In *MoDeVVa@MODELS, Workshop on Model-Driven Engineering, Verification and Validation co-located with International Conference on Model Driven Engineering Languages and Systems*, pages 19–28, 2014.

[45] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1) :113–129, 2010.

[46] I. Gent, E. MacIntyre, P. Presser, B. M. Smith, and T. Walsh. *An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem*, pages 179–193. 1996.

[47] I. P. Gent and B. M. Smith. Symmetry Breaking in Constraint Programming. In *ECAI, European Conference on Artificial Intelligence*, pages 599–603, 2000.

[48] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *SoSyM, Software & Systems Modeling*, 4(4) :386–398, 2005.

[49] M. Gogolla, F. Büttner, and M. Richters. USE : A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1–3) :27–34, 2007.

[50] D. E. Goldberg and J. H. Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2) :95–99, 1988.

[51] C. A. González Pérez, F. Buettner, R. Clarisó, and J. Cabot. EMFtoCSP : A Tool for the Lightweight Verification of EMF Models. In *FormSERA, Formal Methods in Software Engineering*, pages 44–50, 2012.

[52] R. W. Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2) :147–160, 1950.

[53] D. Harel. Statecharts : A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3) :231–274, 1987.

[54] J. A. Hartigan. *Clustering algorithms.* Wiley, 1975.

[55] B. Henderson-Sellers. *Object-Oriented Metrics : Measures of Complexity.* Prentice Hall, 1996.

[56] J. H. Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence.* University of Michigan Press, 1975.

[57] D. Jackson. *Software Abstractions : logic, language, and analysis.* MIT Press, 2012.

[58] I. Jacobson. *Object-oriented Software Engineering : a Use Case Driven Approach.* ACM Press, 1992.

[59] J.-M. Jézéquel, B. Combemale, and D. Vojtisek. *Ingénierie Dirigée par les Modèles : des concepts à la pratique.* Ellipses, 2012.

[60] E. Juliot and J. Benois. Viewpoints creation using Obeo Designer or how to build Eclipse DSM without being an expert developer. *Obeo Designer Whitepaper*, 2010.

[61] Z. A. Karian and E. J. Dudewicz. *Handbook of Fitting Statistical Distributions with R.* CRC Press, 2012.

[62] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598) :671–680, 1983.

[63] D. S. Kolovos, L. M. Rose, S. B. Abid, R. F. Paige, F. A. C. Polack, and G. Botterweck. *Taming EMF and GMF Using Model Transformation*, pages 211–225. 2010.

[64] K. Kuchcinski and R. Szymanek. *JaCoP Library User's Guide, version 4.4*, 2015.

[65] J. Larrosa and P. Meseguer. Exploiting the use of DAC in Max-CSP. In *CP, International Conference on Principles and Practice of Constraint Programming*, pages 308–322, 1996.

[66] C. Lecoutre and O. Roussel. XML Representation of Constraint Networks : Format XCSP 2.1. *Computing Research Repository ACM Journal*, 9(2) :2362–2370, 2009.

[67] A. Lindebaum. Contributions à l'étude de l'espace métrique i. *Fundamenta Mathematicae*, 8(1) :209–222, 1926.

[68] A. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence Journal*, 8(1) :99–118, 1977.

[69] H. Malgouyres and G. Motet. An UML Model Consistency Verification Approach Based on Meta-modeling Formalization. In *SAC, ACM Symposium on Applied Computing*, pages 1804–1809, 2006.

[70] M. Marriott. Caste ranking and community structure in five regions of India and Pakistan. *Bulletin of the Deccan College Research Institute*, 19(1/2) :31–105, 1958.

[71] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding : A versatile graph matching algorithm and its application to schema matching. In *ICDE, International Conference on Data Engineering*, pages 117–128, 2002.

[72] S. Merchez, C. Lecoutre, and F. Boussemart. AbsCon : A Prototype to Solve CSPs with Abstraction. In *CP, International Conference on Principles and Practice of Constraint Programming*, pages 730–744, 2001.

[73] M. Mitzenmacher and E. Upfal. *Probability and Computing : Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[74] A. Mougenot, A. Darrasse, X. Blanc, and M. Soria. Uniform Random Generation of Huge Metamodel Instances. In *ECMDA, European Conference on Model-Driven Architecture Foundations and Applications*, pages 130–145, 2009.

[75] P. Muller, F. Fondement, B. Baudry, and B. Combemale. Modeling modeling modeling. *SoSyM, Software and System Modeling*, 11(3) :347–359, 2012.

[76] OMG. *Meta-Object Facility Specification*, 2002. Available at : http ://www.omg.org/spec/MOF/.

[77] OMG, Object Management Group. Object Constraint Language Specification, Version 2.4. Official Specification, 2014. `http://www.omg.org/spec/OCL/`.

[78] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking : bringing order to the web. 1999.

[79] C. A. Petri. Communication with Automata. Technical report, Defense Technical Information Center, 1966.

[80] F. Pfister, V. Chapurlat, M. Huchard, and C. Nebut. A Light-Weight Annotation-based solution to design Domain Specific Graphical Modeling Languages. ECMFA Demos and Posters, European Conference on Modelling Foundations and Applications , 2013.

[81] F. Pfister, M. Huchard, and C. Nebut. A Framework for Concurrent Design of Metamodels and Diagrams - Towards an Agile Method for the Synthesis of Domain Specific Graphical Modeling languages. In *ICEIS, International Conference on Enterprise Information Systems*, pages 298–306, 2014.

[82] F. R. Pitts. A graph theoretic approach to historical geography. *The Professional Geographer*, 17(5) :15–20, 1965.

[83] J. Privat. A concise reference of the nit langage. Technical report, UQAM, Université du Québec à Montréal, 2011.

[84] J.-F. Puget. On the Satisfiability of Symmetrical Constrained Satisfaction Problems. In *Methodologies for Intelligent Systems*, pages 350–361. 1993.

[85] J.-F. Puget. A c++ implementation of clp. Technical report, 1994.

[86] H. L. Rakotonirainy, J.-P. Müller, and B. O. Ramamonjisoa. Towards a Generic Framework for the Initialization and the Observation of Socio-environmental Models. In *Model and Data Engineering*, pages 45–52. 2014.

[87] J.-C. Régin. A filtering algorithm for constraints of difference in csps. In *AAAI, Association for the Advancement of Artificial Intelligence*, pages 362–367, 1994.

[88] D. T. Ross. Structured Analysis (SA) : A Language for Communicating Ideas. *IEEE Transactions on Software Engineering*, (1) :16–34, 1977.

[89] F. Rossi, P. Van Beek, and T. Walsh, editors. *Handbook of Constraint Programming.* Elsevier Science Publishers, 2006.

[90] M. Roy, S. Schmid, and G. Trédan. Modeling and Measuring Graph Similarity : the Case for Centrality Distance. In *FOMC, ACM International Workshop on Foundations of Mobile Computing*, pages 47–52, 2014.

[91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design.* Prentice-Hall, 1991.

[92] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[93] H. Saada, M. Huchard, C. Nebut, and H. Sahraoui. Recovering Model Transformation traces using Multi-Objective Optimization. In *ASE, IEEE/ACM International Conference on Automated Software Engineering*, pages 688–693, 2013.

[94] S. Sen, B. Baudry, and J.-M. Mottu. On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing. In *ICST, IEEE International Conference on Software Testing, Verification and Validation*, pages 328–337, 2008.

[95] S. Sen, B. Baudry, and J.-M. Mottu. Automatic Model Generation Strategies for Model Transformation Testing. In *ICMT, International Conference on Model Transformation*, pages 148–164, 2009.

[96] H. Simonis. The chip system and its applications. In *CP, International Conference on Principles and Practice of Constraint Programming*, pages 643–646, 1995.

[97] H. Simonis. Sudoku as a Constraint Problem. In *CP Workshop, on Modeling and Reformulating Constraint Satisfaction Problems*, pages 13–27, 2005.

[98] A. Singhal. Modern information retrieval : A brief overview. *IEEE Data Engineering Bulletin*, 24(4) :35–43, 2001.

[99] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF : Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[100] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2) :254–272.

[101] H. Tardieu, A. Rochfeld, and R. Colletti. *La méthode MERISE : Principes et Outils*. Les Editions d'Organisation, 1984.

[102] J.-P. Tolvanen. The Business Cases for Modeling and Generators. Code Generation Conference, 2014. Aivalable at : http ://www.infoq.com/presentations/modeling-language-generator.

[103] K. Voigt and T. Heinze. Metamodel matching based on planar graph edit distance. In *Theory and Practice of Model Transformations*, pages 245–259. 2010.

[104] Voigt, Konrad. *Structural Graph-based Metamodel Matching*. PhD thesis, Dresden University, 2011.

[105] M. Weller, A. Chateau, and R. Giroudeau. Exact approaches for scaffolding. *BMC Bioinformatics*, 16(14) :1471–2105, 2015.

[106] H. Wu, R. Monahan, and J. F. Power. Exploiting Attributed Type Graphs to Generate Metamodel Instances Using an SMT Solver. In *TASE, International Symposium on Theoretical Aspects of Software Engineering*, pages 175–182, 2013.

[107] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6) :1245–1262, 1989.

Index

Index des noms propres

Glossaire

XMI (XML Metadata Interchange) Un format de fichier spécifié par l'OMG pour la représentation de modèles, `http://www.omg.org/spec/XMI/`. 137

Alloy Un langage déclaratif utilisé pour exprimer des structures de contraintes complexes. Il offre un outil de modélisation simple, basé sur la logique du premier ordre. Sa syntaxe est fortement inspiré du langage ***Z***. L'outil d'analyse Alloy fonctionne par réduction vers SAT. 20

Constraint Logic Programming (CLP) Une forme de programmation par contraintes basée sur la programmation logique. 35

Contrainte Globale Un type de contrainte très connu en Programmation par Contraintes. Elle offre plusieurs avantages: définir une seule contraintes expressive au lieu de plusieurs plusieurs contraintes et la possibilité de concevoir des algorithmes de filtrage efficaces. 31

Dot Un format de fichier pour la représentation de tout type de graphes, `http://www.graphviz.org/Documentation/dotguide.pdf`. 137

Ecore Le Méta-métamodèle EMF permettant de définir des concepts manipulables dans EMF. Ses concepts sont tous préfixés par un "E". 136

EMF (Eclipse Modeling Framework) Un framework de modélisation, permettant la génération de code afin de faciliter la construction d'outils et d'applications via des processus IDM. Par exemple, a partir d'un modèle écrit en XMI, l'environnement génère des classes JAVA pour manipuler des instances du modèle. Il contient aussi des outils pour l'édition des modèles et contraintes. 15, 139

Fragments de méta-modèle (Meta-model Fragments) Les parties atomiques d'un méta-modèle. 20

Graphe typé étiqueté borné avec héritage (Bounded attributed type graph with inheritance) Un graphe utilisé pour formaliser mathématiquement les modèles logiciels et leurs concepts (Objet, Type, Attribut, Cardinalité et Héritage). 18, 22

GraphViz Un outil open source pour la visualisation de graphes, `http://www.graphviz.org`. 137

Méthode Boltzmann Une technique de génération de structures aléatoires basée sur les générateurs du même nom. 17

OCL (Object Constraint Language) Un langage de spécification de contraintes sur les éléments des modèles. 16

OMG (Object Management Group) Une organisation internationale à but non lucratif créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes. L'OMG s'occupe notamment de la spécification des standard UML, OCL et XMI. 11, 16

Problème de Satisfaction de Contraintes (Constraint Satisfaction Problem, CSP) Un paradigme informatique d'expression de problèmes contraints puis résolus par des algorithmes de recherche arborescentes. 19

Recuit simulé (Simulated Annealing) Une technique probabiliste pour l'approximation de l'optimum d'une fonction donnée. 21

SAT Un problème de décision qui pour une formule logique donnée détermine s'il existe une assignation qui la satisfait. 20, 35

Satisfiability Modulo Theory (SMT) Un paradigme informatique permettant l'expression de problèmes de décision en logique classique couplée à d'autres théories (Arithmétiques, PL, etc). 22

xcsp Un format de spécification de programmes contraints supporté par plusieurs solveurs. 35, 139

Résumé

Disposer de modèles dans le but de valider ou tester une approche ou un concept est d'une importance primordiale dans beaucoup de domaines différents. Malheureusement, ces modèles ne sont pas toujours disponibles, sont coûteux à obtenir, ou bien ne répondent pas à certaines exigences de qualité ce qui les rend inutiles dans certains cas de figure.
Un générateur automatique de modèles est un bon moyen pour obtenir facilement et rapidement des modèles valides, de différentes tailles, pertinents et diversifiés.
Dans cette thèse nous proposons une nouvelle approche déclarative et basée sur la programmation par contraintes pour la génération de modèles. Ces derniers sont modélisés sous la forme d'un méta-modèle qui est ensuite formalisé efficacement en contraintes pour trouver des solutions. Dans l'optique d'obtenir des modèles utiles, nous nous intéressons à leur vraisemblance qui est obtenue via des lois de probabilités et des métriques spécifiques aux domaines et à leur diversité assurée par des distances comparant ces modèles.

Mots clefs : Ingénierie dirigée par les modèles, Programmation par contraintes, génération de modèles, lois de probabilités usuelles, distances entre modèles.

Abstract

Owning models is useful in many different fields. Models can be used to test and to validate approaches, algorithms and concepts. Unfortunately, models are rarely available, are cost to obtain, or are not adapted to most of cases due to a lack of quality.
An automated model generator is a good way to generate quickly and easily models that are valid, in different sizes, likelihood and diverse.
In this thesis, we propose a novel and declarative model driven approach, based on constraint programming for automated model generation. Models are modelled as a meta-model. This later is then encoded in constraint programming to find solutions. For more useful models, we are concerned by their likelihood, which is obtained by simulating probability distributions related to domain-specific metrics and also by their diversity through model comparison distances.

Keywords : Model Driven Engineering, Constraint Programming, Model Generation, Probability distributions, Model distances.

Laboratoire Lirmm — 161, rue Ada, 34095 Montpellier, France

Résumé

Disposer de modèles dans le but de valider ou tester une approche ou un concept est d'une importance primordiale dans beaucoup de domaines différents. Malheureusement, ces modèles ne sont pas toujours disponibles, sont coûteux à obtenir, ou bien ne répondent pas à certaines exigences de qualité ce qui les rend inutiles dans certains cas de figure.
Un générateur automatique de modèles est un bon moyen pour obtenir facilement et rapidement des modèles valides, de différentes tailles, pertinents et diversifiés.
Dans cette thèse nous proposons une nouvelle approche déclarative et basée sur la programmation par contraintes pour la génération de modèles. Ces derniers sont modélisés sous la forme d'un méta-modèle qui est ensuite formalisé efficacement en contraintes pour trouver des solutions. Dans l'optique d'obtenir des modèles utiles, nous nous intéressons à leur vraisemblance qui est obtenue via des lois de probabilités et des métriques spécifiques aux domaines et à leur diversité assurée par des distances comparant ces modèles.

Mots clefs : Ingénierie dirigée par les modèles, Programmation par contraintes, génération de modèles, lois de probabilités usuelles, distances entre modèles.

Abstract

Owning models is useful in many different fields. Models can be used to test and to validate approaches, algorithms and concepts. Unfortunately, models are rarely available, are cost to obtain, or are not adapted to most of cases due to a lack of quality.
An automated model generator is a good way to generate quickly and easily models that are valid, in different sizes, likelihood and diverse.
In this thesis, we propose a novel and declarative model driven approach, based on constraint programming for automated model generation. Models are modelled as a meta-model. This later is then encoded in constraint programming to find solutions. For more useful models, we are concerned by their likelihood, which is obtained by simulating probability distributions related to domain-specific metrics and also by their diversity through model comparison distances.

Keywords : Model Driven Engineering, Constraint Programming, Model Generation, Probability distributions, Model distances.

Laboratoire Lirmm — 161, rue Ada, 34095 Montpellier, France