Université de Montréal

Une approche heuristique pour l'apprentissage de transformations de modèles complexes à partir d'exemples

par
Islem Baki

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Décembre, 2014

RÉSUMÉ

L'ingénierie dirigée par les modèles (IDM) est un paradigme d'ingénierie du logiciel bien établi, qui préconise l'utilisation de modèles comme artéfacts de premier ordre dans les activités de développement et de maintenance du logiciel. La manipulation de plusieurs modèles durant le cycle de vie du logiciel motive l'usage de transformations de modèles (TM) afin d'automatiser les opérations de génération et de mise à jour des modèles lorsque cela est possible.

L'écriture de transformations de modèles demeure cependant une tâche ardue, qui requiert à la fois beaucoup de connaissances et d'efforts, remettant ainsi en question les avantages apportés par l'IDM. Afin de faire face à cette problématique, de nombreux travaux de recherche se sont intéressés à l'automatisation des TM. L'apprentissage de transformations de modèles par l'exemple (TMPE) constitue, à cet égard, une approche prometteuse. La TMPE a pour objectif d'apprendre des programmes de transformation de modèles à partir d'un ensemble de paires de modèles sources et cibles fournis en guise d'exemples.

Dans ce travail, nous proposons un processus d'apprentissage de transformations de modèles par l'exemple. Ce dernier vise à apprendre des transformations de modèles complexes en s'attaquant à trois exigences constatées, à savoir, l'exploration du contexte dans le modèle source, la vérification de valeurs d'attributs sources et la dérivation d'attributs cibles complexes. Nous validons notre approche de manière expérimentale sur 7 cas de transformations de modèles. Trois des sept transformations apprises permettent d'obtenir des modèles cibles parfaits. De plus, une précision et un rappel supérieurs à 90% sont enregistrés au niveau des modèles cibles obtenus par les quatre transformations restantes.

Mots clés: Apprentissage de transformations de modèles, transformation de modèles par l'exemple, ingénierie dirigée par les modèles, génie logiciel basé sur la recherche.

ABSTRACT

Model-driven engineering (MDE) is a well-established software engineering paradigm that promotes models as main artifacts in software development and maintenance activities. As several models may be manipulated during the software life-cycle, model transformations (MT) ensure their coherence by automating model generation and update tasks when possible.

However, writing model transformations remains a difficult task that requires much knowledge and effort that detract from the benefits brought by the MDE paradigm. To address this issue, much research effort has been directed toward MT automation. Model Transformation by Example (MTBE) is, in this regard, a promising approach. MTBE aims to learn transformation programs starting from a set of source and target model pairs supplied as examples.

In this work, we propose a process to learn model transformations from examples. Our process aims to learn complex MT by tackling three observed requirements, namely, context exploration of the source model, source attribute value testing, and complex target attribute derivation. We experimentally evaluate our approach on seven model transformation problems. The learned transformation programs are able to produce perfect target models in three transformation cases, whereas, precision and recall higher than 90% are recorded for the four remaining ones.

Keywords: Model transformation learning, model transformation by example, model-driven engineering, search-based software engineering.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES

AE	Algorithmes évolutionnaires
AG	Algorithmes génétiques
IDM	Ingénierie dirigée par les modèles
LD	Langages dédiés
MC	Modèle cible
MMC	Méta-modèle cible
MMS	Méta-modèle source
MS	Modèle source
PG	Programmation génétique
POO	Paradigme orienté objet
PT	Programme de transformation
RS	Recuit simulé
RT	Règle de transformation
TM	Transformation de modèles
TMPD	Transformation de modèles par démonstration
TMPE	Transformation de modèles par l'exemple
TR	Trace de transformation

REMERCIEMENTS

La réalisation de ce travail n'aurait pas été possible sans le soutien et l'aide précieuse de plusieurs personnes. Je tiens à leur témoigner ma reconnaissance.

Je tiens, en premier lieu, à exprimer ma gratitude à mon directeur de mémoire Monsieur Houari Sahraoui, pour avoir été présent tout au long de la réalisation de ce travail. Je le remercie pour son encadrement, ses conseils et sa bienveillance.

Je tiens également à adresser mes remerciements à tous les membres du laboratoire de génie logiciel pour l'environnement de travail convivial dans lequel j'ai été accueilli, et qui fut certes, très favorable à l'aboutissement de ce projet.

Enfin, je remercie ma famille pour le soutien moral et affectif qu'elle m'a apporté tout au long de cette expérience.

CHAPITRE 1

INTRODUCTION

1.1 Contexte

Durant les dernières décennies, l'industrie du logiciel s'est rapidement étendue pour constituer, aujourd'hui, l'un des secteurs les plus prolifiques et rentables à l'échelle mondiale. Selon une étude publiée par Gartner[1], le marché mondial du logiciel a généré plus de 400 milliards de dollars de revenus en 2013, enregistrant une évolution annuelle de 4,8%. Des chiffres qui ne surprennent pas en considérant le rôle névralgique qu'occupent les logiciels dans la modernisation de notre société. Attirées par ce secteur aux besoins inépuisables, beaucoup d'entreprises spécialisées en technologie tentent de progresser dans ce marché, qui est devenu par la même occasion hautement concurrentiel. De nombreux professionnels et chercheurs du domaine se sont donc naturellement penchés sur des moyens qui permettraient d'augmenter la productivité et l'efficience du développement et de la maintenance de logiciels qui se font de plus en plus complexes. La modélisation présente, à ce titre, un potentiel important qui reste très peu exploité.

À l'instar des autres disciplines, l'ingénierie du logiciel eut recourt à des modèles assez tôt dans son histoire. En effet, le besoin d'abstraction, afin d'adresser la complexité croissante des logiciels et de leurs plateformes, a conduit à l'apparition de méthodes qui font usage de modèles et de représentations graphiques dès les années 80, telles que le génie logiciel assisté par ordinateur (CASE) [51] ou les langages de programmation de 4ème génération (4GL) [56]. Le succès du paradigme orienté objet (OO) a ensuite permis à UML *(Unified Modeling Language)* [61] de s'imposer dans l'industrie comme langage de modélisation.

En dépit de l'omniprésence de la modélisation dans les projets logiciels, la majorité des praticiens maintiennent des processus de développement centrés sur le code [26]. L'utilisation de modèles se limite ainsi souvent à des fins d'analyse, de conception et

[1] http ://www.gartner.com/document/2695617

de documentation sans être formellement liée à l'implémentation du logiciel. Cette approche fut qualifiée par Stahl et Völter [78] d'ingénierie basée sur les modèles. Plusieurs outils tentent aujourd'hui d'offrir une meilleure intégration de la modélisation dans le processus de développement par une génération automatique de code à partir de modèles et vice versa. Ils restent cependant très peu flexibles et assez superficiels. Les modifications apportées aux programmes doivent souvent être répercutées sur différents modèles afin de les maintenir à jour. L'élaboration de modèles est souvent vue comme un travail additionnel, dont le fruit prend la forme d'une documentation plutôt que d'une partie du produit final.

L'ingénierie dirigée par les modèles (IDM) est un paradigme qui vient pallier cette situation. Plutôt que d'adopter une approche basée sur les modèles, le terme « dirigée » souligne le fait que les modèles se retrouvent au premier plan des différentes activités d'ingénierie du logiciel. La modélisation est vue alors comme étant l'activité principale de l'implémentation, dans le sens où un logiciel consiste en un ensemble de modèles qui peuvent être exploités par une interprétation directe ou par une génération automatique de code.

Selon Schmidt [65], l'IDM se compose de deux éléments essentiels : (1) des langages de modélisation dédiés qui permettent de capturer des aspects spécifiques du problème et de la solution, et (2) des transformations qui permettent de générer automatiquement à partir des modèles disponibles d'autres artéfacts (modèles, code source, suites de tests, etc.). Ces deux concepts fondamentaux marquent la transition de programmes exprimés en termes d'algorithmes manipulant des structures de données à des logiciels construits entièrement à partir de modèles et de transformations [9]. Considérée comme le successeur du développement de logiciels tel que connu actuellement, l'IDM a suscité beaucoup d'intérêt durant ces dernières années. Beaucoup de recherches et d'efforts de standardisation se font en ce sens. Le *Model Driven Architecture (MDA)*, proposé par l'*Object Management Group (OMG)*, s'impose déjà comme un standard de l'industrie.

1.2 Problématique

La transformation de modèles (TM) est un concept essentiel à la concrétisation de la vision dirigée par les modèles. Elle apporte l'automatisation nécessaire aux activités dirigées par les modèles en permettant, non seulement, de maintenir la cohérence des différents modèles manipulés tout au long des processus de production de logiciels, mais aussi, de leur appliquer différents traitements, tels que la fusion, le raffinement, la refactorisation et la génération de modèles de plus bas niveau ou de code.

Le sujet des transformations de modèles demeure l'objet de beaucoup de recherche. De nombreux langages, outils et environnements d'écriture de transformations ont été proposés. L'utilisation des langages de transformation pour l'implémentation des transformations, tout comme pour leur spécification, a conduit à de nombreuses initiatives qui ont permis d'élever le niveau d'abstraction des langages de transformation en adoptant un style déclaratif où la transformation peut être décrite selon un ensemble de règles. L'OMG a notamment proposé le standard *Query/View/Transformation* (QVT) qui définit trois langages de transformation de modèles : un premier déclaratif de haut niveau (*QVT Relationnal*), un deuxième déclaratif de bas niveau (*QVT Core*), et un troisième impératif (*QVT Operationnal*).

Cependant, tel que soulevé par Lano et al. [49], la majorité de ces contributions se concentrent sur des aspects d'implémentation. L'écriture des transformations demeure ainsi une tâche ardue qui requiert des connaissances du domaine du problème (les langages de modélisation) ainsi que celui de la solution (le langage de transformation). Dans le premier domaine (le problème), il est nécessaire d'avoir une connaissance approfondie des langages de modélisation des domaines sources et cibles, en termes de syntaxe concrète (notation graphique du modèle), et abstraite (représentation interne du modèle). S'ajoute à cela la nécessité de récolter et d'exprimer l'équivalence sémantique entre ces deux domaines, une étape dont l'aisance dépend fortement de la popularité des langages manipulés. Concernant le domaine de la solution, une maitrise du langage de transformation utilisé et de son environnement est primordiale. En outre, le processus d'écriture de transformations étant similaire à celui du développement de code, celui-ci doit inévi-

tablement se faire de manière itérative. Les règles de transformations doivent ainsi être écrites, exécutées, déboguées et corrigées jusqu'à produire le résultat escompté.

De nombreux travaux ont été proposés afin d'augmenter la productivité du processus de développement de TM et d'exploiter au mieux les différents langages de transformation disponibles [14, 31, 72]. Cependant, la récolte et la spécification de l'équivalence sémantique entre les domaines source et cible demeurent ardues, notamment pour des domaines peu connus. Partant de ce constat, plusieurs contributions ont exploré l'idée d'améliorer l'automatisation de la TM en apprenant automatiquement certaines transformations lorsque les connaissances nécessaires à leur écriture sont insuffisantes ou difficile à obtenir. Parmi ces contributions, la transformation de modèles par l'exemple (TMPE) a suscité, en particulier, beaucoup d'intérêt. Cette idée, proposée initialement par Varró [76] vise à apprendre automatiquement les transformations convoitées à partir d'un ensemble de paires de modèles sources et cibles fournis en guise d'exemples. Les paires d'exemples sont généralement accompagnées de traces de transformations capturant les fragments sources et cibles correspondants. Un mécanisme d'apprentissage est ensuite utilisé pour dériver des règles de transformation qui peuvent être sujettes à un affinement manuel avant d'être exécutées.

En dehors de l'algorithme d'apprentissage sous-jacent, la majorité des contributions portant sur l'apprentissage de TMPE adoptent le processus décrit plus haut. Malgré des progrès considérables en termes de degré d'automatisation et de règles pouvant être dérivées, l'apprentissage de transformations complexes est resté hors de portée jusqu'à présent. En effet, certains problèmes de transformation nécessitent des transformations structurelles sophistiquées. Elles peuvent également présenter des dépendances par rapport aux valeurs que prennent les attributs du modèle source, ou bien encore, requérir des dérivations complexes de certains attributs du modèle cible.

1.3 Contribution

Ce mémoire a pour but de proposer un processus d'apprentissage de transformations de modèles à partir d'exemples. L'approche proposée est capable de considérer un plus

grand éventail de cas de transformations de modèles que l'état de l'art. Elle permet, en particulier, d'apprendre des transformations de modèles complexes qui requièrent des transformations structurelles sophistiquées, l'identification de dépendances par rapport aux valeurs des attributs du modèle source ou des dérivations complexes d'attributs du modèle cible.

À l'instar du commun des travaux existants, notre approche prend en entrée un ensemble de pairs d'exemples de modèles sources et cibles accompagnés de traces de transformations capturant la correspondance entre les différents fragments sources et cibles. Le processus d'apprentissage proposé permet alors de dériver un programme de transformation en trois étapes principales. Dans un premier temps, les traces fournies sont complétées et analysées afin de construire des *pools* d'exemples cohérents. Ensuite, un ensemble de règles est dérivé pour chaque *pool* en utilisant un programme génétique. Durant cette étape, les règles sont également traitées pour éliminer d'éventuelles erreurs ou incohérences. Enfin, la troisième et dernière étape consiste à fusionner les groupes de règles apprises en un programme de transformation qui est affiné en utilisant la méthode du recuit simulé.

Notre processus d'apprentissage est validé sur sept (7) cas de transformations présentant divers caractéristiques et degrés de complexité. Les résultats obtenus indiquent que les transformations les plus communes sont parfaitement apprises. Par ailleurs, l'apprentissage des cas de transformations les plus complexes permet de générer des modèles cibles très similaires à ceux attendus.

1.4 Structure du mémoire

La suite de ce mémoire est organisée de la manière suivante : le chapitre 2 est divisé en deux parties. Dans un premier temps, des généralités sur la modélisation, le paradigme dirigé par les modèles et sur la transformation de modèles sont présentées. La deuxième partie du chapitre se concentre sur les contributions qui s'inscrivent dans le cadre de l'apprentissage des transformations de modèles. Le chapitre 3 constitue le cœur de ce mémoire. Après un aperçu du processus d'apprentissage, il décrit, en détail, les

trois étapes de l'approche proposée. Le chapitre 4 rapporte les éléments de la validation effectuée en abordant en premier lieu le cadre de l'expérimentation effectuée ainsi que les cas de transformation choisis. Par la suite, les résultats obtenus sont présentés et discutés. Enfin, le 5e et dernier chapitre conclut le présent mémoire en identifiant notamment certains axes d'amélioration et travaux futurs.

CHAPITRE 2

ÉTAT DE L'ART

L'objectif de ce chapitre est double. Il permet d'introduire, d'une part, les différents concepts qui entourent l'ingénierie dirigée par les modèles, et d'explorer de l'autre, les travaux qui traitent de l'apprentissage de transformations dans le cadre de l'IDM. Ce chapitre est organisé en quatre parties. Nous introduirons, dans la première, le paradigme d'ingénierie dirigée par les modèles ainsi que différentes notions liées à ce dernier. Nous aborderons ainsi les concepts de modèle, de méta-modèle, et en particulier, celui de transformation de modèles. Les deux parties qui suivront seront, quant à elles, consacrées à l'apprentissage des transformations de modèles et nous permettront de passer en revue les différentes contributions qui s'inscrivent dans ce cadre. Nous verrons deux familles d'approches d'apprentissage existantes dans la littérature, à savoir, les approches d'apprentissage par l'exemple et celles par démonstration. Enfin, la dernière partie de ce chapitre prend la forme d'une synthèse qui résume les caractéristiques clés et les limites des contributions identifiées.

2.1 Notions de base

2.1.1 Ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles est un nouveau paradigme d'ingénierie du logiciel qui encourage l'usage de modèles au premier plan des processus de développement et de maintenance du logiciel. L'objectif principal de l'IDM est d'élever le degré d'abstraction du développement logiciel en passant de processus centrés sur le code à des processus centrés sur les modèles. Ce besoin d'abstraction est essentiellement motivé par la nécessité de faire face à une complexité croissante des systèmes [65].

2.1.1.1 Vision globale

L'IDM aspire à offrir une vision complète du développement d'un système en utilisant des modèles à différents niveaux d'abstraction. Afin de comprendre les principes qui régissent ce paradigme, nous utiliserons la schématisation proposée par Brambilla et al. [9] qui considèrent qu'une solution dirigée par les modèles est représentée selon deux dimensions orthogonales (fig. 2.1), la conceptualisation et la réalisation. Ces deux axes sont représentés respectivement en colonnes (conceptualisation) et en lignes (réalisation).

L'axe de conceptualisation permet de définir des modèles conceptuels qui servent à décrire une réalité. Cette conceptualisation existe sur trois niveaux :

- Le niveau application : les modèles construits à ce niveau servent à décrire l'application en cours de développement. Des transformations sont appliquées à ces modèles pour générer différentes composantes exécutables du système.

- Le niveau du domaine de l'application : ce niveau permet de définir les langages de modélisation du domaine, les transformations et la plateforme sur laquelle seront exécutées les composantes .

- Le niveau méta : dans ce niveau, on définit des langages de transformation et de méta-modélisation qui permettent de conceptualiser les langages de modélisation et les transformations exprimées au niveau du domaine de l'application.

L'axe de réalisation vise à établir des correspondances entre les modèles exprimés et le système futur. À l'instar de l'axe de conceptualisation, il se compose de trois éléments clés :

- Le niveau de modélisation : à ce niveau, les modèles qui décrivent le système existant ou futur sont conçus. Les modèles abordent différents aspects du système et peuvent être exprimés à différents degrés d'abstraction.

- Le niveau d'automatisation : ce niveau comporte les mécanismes (les transformations) qui permettent de générer, de manière automatique, divers artéfacts à partir des modèles créés.

- Le niveau réalisation : il comporte les artéfacts utilisés lors de l'exécution du système. Bien que ce soit présentement du code, aucune supposition n'est faite quant à la nature de ces artéfacts. Ces derniers pourraient être des modèles exploités par des techniques d'interprétation ou de simulation.

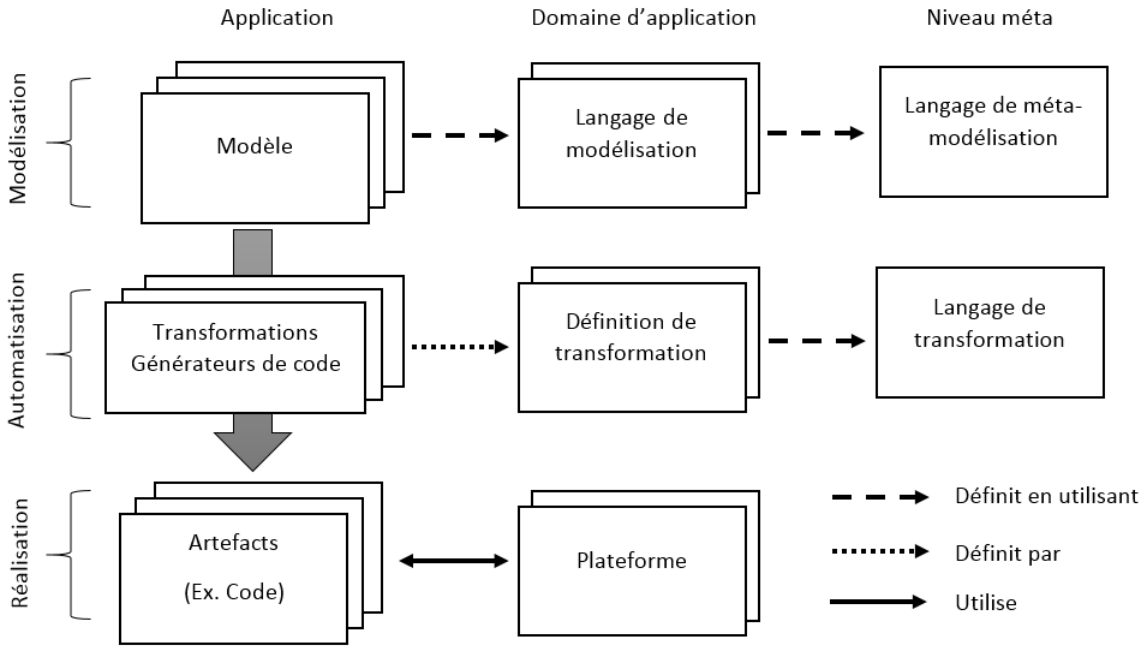

Figure 2.1 – Aperçu du paradigme d'ingénierie dirigée par les modèles

2.1.1.2 Gains potentiels

À partir de la description fournie plus haut, il devient possible d'entrevoir les avantages qui découleraient de la concrétisation de l'IDM. En effet, alors que l'usage de modèles permet de maitriser la complexité du système par plusieurs vues possibles sur plusieurs niveaux d'abstraction, les transformations de modèles permettent, quant à elles, d'augmenter la productivité en automatisant le passage des modèles d'application aux artéfacts de réalisation. Les modèles sont également exprimés en utilisant des notations spécifiques au domaine, ce qui encourage la réutilisation. Les retombées potentielles du paradigme peuvent être résumées dans les points suivants [78] :

– Une meilleure gestion de la complexité grâce au concept d'abstraction.

– Une augmentation de la productivité grâce à la modélisation des domaines et à l'automatisation des processus de mise à jour de modèles et de génération de code à partir de ces derniers.

– Une amélioration de la qualité du logiciel en découplant l'architecture de ce dernier de son implémentation.

– Une meilleure communication entre les intervenants attribuée à l'usage des modèles qui constituent un très bon support à la communication, et qui peuvent être appréhendés par les experts du domaine.

– Des produits plus facilement maintenables et une meilleure gestion des changements technologiques à travers le respect du principe de séparation des préoccupations [47].

2.1.1.3 Composantes principales

Les modèles et les transformations de modèles forment le cœur de l'ingénierie dirigée par les modèles. Schmidt [65] affirme en 2006 que l'IDM combine deux éléments : (1) des langages dédiés (*Domain Specific Language*), et (2) des moteurs de transformations et des générateurs. Brambilla et al. [9] vont encore plus loin, ils transforment l'équation de Niklaus Wirth [82]

$$Algorithmes + Structure\ de\ données = Programmes$$

en

$$Modèles + Transformations = Logiciels.$$

Nous explorons dans ce qui suit ces deux concepts fondamentaux.

2.1.2 Modèles et méta-modèles

Le concept de modèle n'est pas spécifique à l'ingénierie du logiciel. De nombreuses disciplines font usage de modèles depuis maintenant bien longtemps. Bien qu'étant com-

mun, le concept de modèle ne dispose pas d'une définition claire et précise, il demeure en fait sujet à beaucoup de désaccords [53]. Dans le contexte du génie logiciel, un modèle peut être défini comme une représentation abstraite de la structure, d'une fonction, ou du comportement d'un système [78].

2.1.2.1 Caractéristiques

Selon Stachowiak [67], trois critères permettent de distinguer un modèle des autres artéfacts :

Correspondance : le modèle correspond à un objet ou à un phénomène d'origine. L'élément d'origine n'existe pas forcément, il peut être planifié, suspecté ou tout simplement fictif.

Réduction : Le modèle constitue une représentation réduite de l'objet ou du phénomène d'origine. Certaines propriétés de ce dernier sont alors omises.

Pragmatisme : Le modèle peut remplacer l'objet d'origine à certaines fins. Sa création doit donc être justifiée par un objectif.

Il est important de noter que la qualité « réduite » du modèle n'est pas un défaut de ce dernier, au contraire, elle permet souvent de manipuler le modèle dans des situations ou l'élément d'origine ne peut l'être. Cette réduction est souvent effectuée par une activité qui est appelée « l'abstraction » [46]. Compte tenu de la définition ci-haut, le code source, étant une abstraction du langage machine, est un modèle également [57].

Dans le contexte du génie logiciel, un modèle peut être descriptif ou prescriptif. Un modèle est dit descriptif s'il reflète un existant. Les modèles descriptifs sont généralement crées lors de l'analyse pour décrire le problème. Au contraire, un modèle est dit prescriptif dans le cas où il est utilisé comme une spécification d'un système futur.

Contrairement à certains domaines, le développement de logiciel n'utilise pas de modèles physiques. Les modèles manipulés sont exclusivement linguistiques [46]. Il est important cependant de distinguer la modélisation du dessin. Alors qu'il est vrai que la modélisation implique généralement des graphiques qui se conforment à des règles syntactiques, les modèles disposent d'une sémantique implicite bien définie et sont exprimés en utilisant des langages de modélisation.

2.1.2.2 Langages de modélisation

Les langages de modélisation permettent aux experts de spécifier une représentation concrète des modèles conceptuels exprimés. Cette représentation peut être graphique, textuelle, ou une combinaison des deux. Il existe deux classes principales de langages de modélisation [9], (1) les langages de modélisation dédiés (LD) qui sont créés spécifiquement pour décrire un certain domaine ou un secteur d'activité et (2) les langages de modélisation généralistes, tels qu'UML et les réseaux de Petri, qui peuvent être utilisés dans n'importe quel domaine. Les langages de modélisation, qu'ils soient dédiés ou généralistes, doivent comporter au moins deux composantes [42].

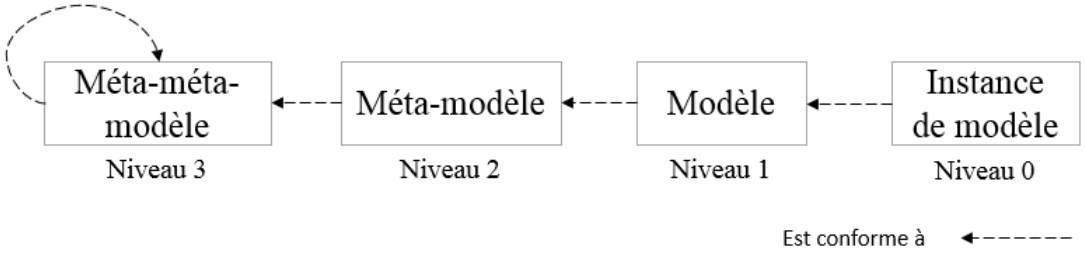

Figure 2.2 – Une pile de méta-modélisation

Une syntaxe abstraite. La philosophie de l'IDM sous-entend que toute chose peut être exprimée par un modèle. Il est alors logique de définir les modèles eux même grâce à d'autres modèles. Cette activité est appelée la méta-modélisation. Un méta-modèle est une seconde abstraction qui permet de définir la grammaire d'un langage de modélisation. Il est possible de définir un modèle qui décrit le méta-modèle exprimé (c.à-d. un méta-méta-modèle) et de continuer ainsi de manière récursive. Plutôt que de définir infiniment des méta-modèles, les modèles de 3ème ou de 4ème niveau sont généralement conçus de manière à s'auto-définir (fig. 2.2).

Une syntaxe concrète. Elle comporte dans la majorité des cas des éléments graphiques (rectangles, ellipses, étiquettes, etc.). Les éléments de la syntaxe concrète sont mis en correspondance avec des éléments de la syntaxe abstraite. Il est possible de définir plusieurs syntaxes concrètes pour un même langage de modélisation [9], ex. une notation textuelle et une autre graphique. Les correspondances entre les syntaxes concrètes

et abstraites sont généralement agrémentées par des contraintes spécifiant dans quelles situations une correspondance est possible. Dans *UML* par exemple, ces contraintes sont spécifiées en utilisant le langage *OCL* [80].

2.1.3 Transformation de modèles

Au même titre que les modèles, les transformations de modèles (TM) sont un ingrédient clé de l'IDM. Kleppe et al. [43] définissent une transformation de modèles de la manière suivante : *la* ***transformation de modèles*** *est la génération automatique d'un modèle cible à partir d'un modèle source, selon une définition de transformation.* ***La définition d'une transformation*** *est un ensemble de règles de transformations qui, ensemble, décrivent comment peut être transformé un modèle exprimé dans le langage source en un modèle exprimé dans le langage cible.* ***Une règle de transformation*** *est une description de comment peut être transformés un ou plusieurs éléments du modèle source en un ou plusieurs éléments du modèle cible*.

La définition du concept de transformation de modèles a évolué au fil des années. Mens et al. [57] étendent la définition fournie par Kleppe et al. en considérant une TM applicable à plusieurs modèles sources en entrée et/ou produisant plusieurs modèles cibles en sortie. Plus récemment, Syriani [71] prend en considération la multitude de cas d'utilisation de TM qui sont apparus. Il généralise ainsi le concept par la définition suivante : *Une transformation est la manipulation automatique d'un modèle avec une intention spécifique*.

Ces définitions considèrent toutes que la manipulation du modèle source est « automatique ». En effet, la transformation de modèles est définie à un niveau plus haut que celui des modèles (fig. 7). Elle est ainsi étroitement liée à la transformation de programmes [16], du fait que les deux sont une forme de méta-programmation, où les objets manipulés sont des méta-données (données qui représentent des artéfacts du logiciel sous la forme de données) tels que les schémas, les interfaces, les programmes ou les modèles.

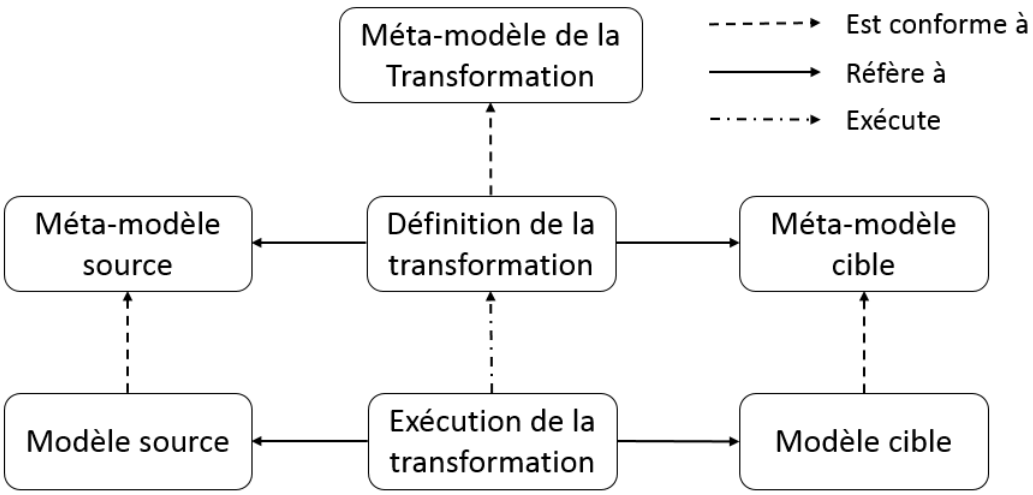

Figure 2.3 – Transformation de modèles

2.1.3.1 Cas d'utilisation

La transformation de modèles est cruciale au succès de l'IDM, dans la mesure où elle permet d'automatiser les activités d'évolution, de synchronisation et de génération de modèles. Elle peut opérer verticalement (des modèles à différents niveaux d'abstraction) ou horizontalement (modèles au même niveau d'abstraction décrivant différents aspects du système). Amrani et al. proposent dans [2, 52] un catalogue de transformations qui regroupe les cas d'utilisation suivants :

- Manipulation : elle regroupe les opérations de base telles que l'ajout, la suppression et la modification d'éléments.
- Interrogation : cette transformation consiste à extraire certaines informations à partir du modèle manipulé.
- Raffinement : elle permet de se déplacer vers un modèle d'un niveau de spécification plus bas que le modèle manipulé.
- Abstraction : à l'inverse du raffinement, cette transformation permet d'augmenter le degré d'abstraction.

- Synthèse : ce type d'opération permet de produire un langage bien défini qui peut être stocké, un exemple de synthèse est la génération de code.
- Rétro-ingénierie : contrairement à la synthèse, elle permet d'extraire un modèle à partir d'artéfacts de bas niveau.
- Approximation : cette opération est un raffinement dans lequel le modèle source est une idéalisation du modèle cible.
- Translation de la sémantique : elle permet de traduire la sémantique d'un langage dans un autre formalisme.
- Analyse : cette transformation fait correspondre le modèle à un formalisme qui peut être analysé plus facilement.
- Simulation : elle définit la sémantique opérationnelle du langage de modélisation en mettant à jour l'état du modèle.
- Normalisation : cette opération réduit la complexité syntactique du modèle en transformant ce dernier dans une forme canonique.
- Rendu : elle permet de faire correspondre une ou plusieurs représentations concrètes d'une même syntaxe abstraite.
- Génération d'instance : la génération produit automatiquement des instances (modèle) d'un méta-modèle en particulier.
- Migration : transforme les modèles vers un autre langage de modélisation en maintenant le niveau d'abstraction de chacun.
- Optimisation : cette catégorie regroupe les transformations qui visent à améliorer les qualités opérationnelles du modèle.
- Restructuration : cette opération restructure le modèle afin d'améliorer certaines de ses caractéristiques sans changer pour autant son comportement observable.

- Composition : elle permet d'intégrer des modèles produits séparément dans un modèle composé. Elle regroupe la fusion et le tissage de modèles.

- Synchronisation : cette transformation intègre des modèles ayant évolué séparément en propageant les changements nécessaires au maintien de leur consistance globale.

2.1.3.2 Taxonomie

Il est possible de catégoriser les transformations de modèles existantes en fonction des modèles manipulés et produits, ainsi qu'en fonction des caractéristiques inhérentes aux transformations elles-mêmes. Nous présentons dans ce qui suit une partie de la taxonomie proposée par Mens et al. [57]. Cette taxonomie nous permettra de définir plusieurs termes s'apprêtant à la TM, qui seront utilisés dans la suite de ce mémoire.

Transformations endogènes/exogènes : une transformation est dite *endogène* si les modèles source et cible sont des instances du même méta-modèle. Inversement, elle est dite *exogène* si les deux modèles sont conformes à deux méta-modèles différents.

Transformations *in-place*/*out-place* : une transformation est dite *in-place*, lorsque la manipulation permettant d'obtenir le modèle cible est effectuée directement sur le modèle source. la transformation est *out-place* si le modèle cible est construit à partir des propriétés d'un autre modèle.

Transformations horizontales/verticales : une transformation est dite *horizontale* lorsque le modèle produit par cette dernière se situe au même niveau d'abstraction que le modèle source. Elle est dite *verticale* quand les modèles source et cible sont à des niveaux d'abstraction différents.

Transformations syntactiques/sémantiques : les transformations qui transforment la syntaxe uniquement sont dites *syntactiques*. Les transformations, plus sophistiquées, qui prennent en considération la sémantique du modèle source lors de la manipulation sont dites *sémantiques*.

Il est possible de classer les différentes transformations présentées dans la sous-section 2.1.3.1 en fonction de ces catégories. Par exemple, les transformations de mi-

gration sont exogènes et horizontales tandis que les opérations de raffinement et d'approximation sont endogènes et verticales.

2.1.3.3 Caractéristiques

Compte tenu du rôle capital que jouent les transformations de modèles dans l'IDM, beaucoup d'efforts ont été consacrés à la mise en avant d'approches, de langages et d'outils de transformations. Afin de comparer et de catégoriser les différentes approches de TM, Czarnecki et Helsen [16] proposent un modèle de caractéristiques (*feature model*) dont le niveau le plus haut est illustré dans la figure 2.4. Nous décrivons brièvement dans ce qui suit, les éléments de ce premier niveau.

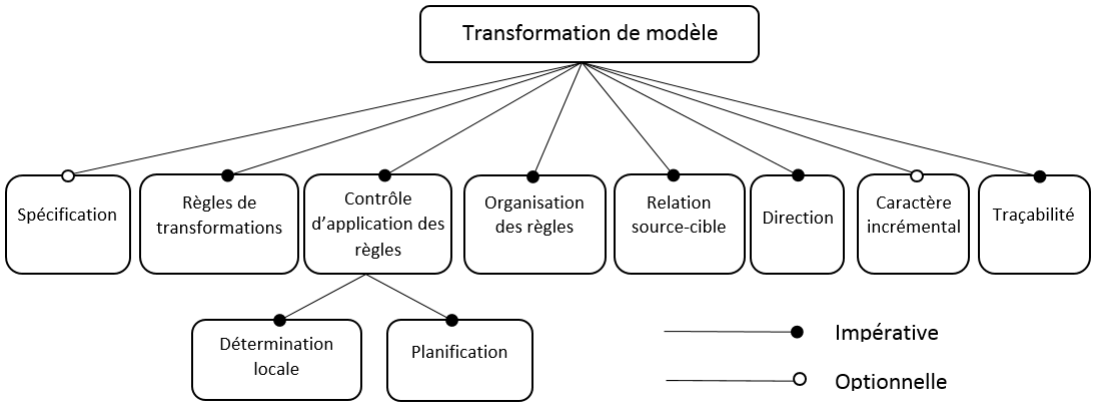

Figure 2.4 – Diagramme de caractéristiques des TM au niveau le plus haut [16]

Spécification : certaines approches de transformation permettent d'ajouter par exemple des pré et post-conditions aux règles de la transformation développée. Ces spécifications sont généralement exprimées en utilisant *OCL*.

Règles de transformation : en fonction du langage de transformation, il est possible que les règles de transformation soient explicitement exprimées sous une forme déclarative, où qu'elles prennent la forme de fonctions ou de procédures. Les gabarits (*templates*) sont également considérés comme une forme de règles dégénérées.

Contrôle d'application des règles : cette caractéristique englobe deux éléments : la détermination locale et la planification. La première consiste à définir, lorsqu'une règle

est applicable dans plusieurs endroits dans le modèle, l'ordre des fragments sur lesquelles la règle est appliquée. La planification permet de déterminer l'ordre dans lequel sont exécutées plusieurs règles dont les conditions se trouvent satisfaites simultanément.

Organisation des règles : elle permet de distinguer les mécanismes d'organisation et de réutilisation mis en œuvre par l'approche tels que les modules, les paquetages, la composition et l'héritage de règles, etc.

Relation source-cible : cet aspect permet de prendre en considération certains éléments de la taxonomie présentée dans 2.1.3.2, notamment, le caractère endogène/exogène de la transformation ainsi que, dans le cas endogène, l'aspect *in-place* ou *out-place* de la transformation.

Caractère incrémental : il fait référence à la capacité de l'approche à propager les changements ayant lieu dans le modèle source au modèle cible. La préservation des modifications apportées au niveau du modèle cible lors de cette propagation est également un aspect particulièrement important dans certaines activités, telle que la synchronisation de modèles.

Direction : certaines transformations sont unidirectionnelles, c'est-à-dire qu'elles peuvent s'exécuter dans une seule direction seulement. Les transformations multidirectionnelles peuvent, quant à elles, s'exécuter dans plusieurs directions. Dans ce dernier cas, les modèles sources peuvent devenir les modèles cibles et vice versa.

Traçabilité : concerne la capacité de l'approche à enregistrer les détails de l'exécution de la transformation, en maintenant des liens entre les éléments cibles et sources. Cette fonctionnalité est utilisée, en particulier, durant les opérations d'analyse d'impact et de débogage de transformations.

2.1.3.4 Approches de transformations

Selon Syriani [71], il existe actuellement dans la littérature plus d'une trentaine d'approches de transformation de modèles. Czarnecki et Helsen ont identifié dans leurs travaux 7 catégories majeures. La majorité des approches existantes de TM appartiennent à l'une de ces catégories.

Manipulation directe : ces approches offrent généralement une interface de pro-

grammation (*API*) pour manipuler le modèle. Les fonctionnalités de traçabilité, de planification, etc. ne sont généralement pas disponibles et doivent plutôt être implémentées par l'utilisateur. Des approches qui s'inscrivent dans cette catégorie sont : *Jambda* [10] et *Builder Object Network*.

Dirigées par la structure : ce type d'approche possède deux phases distinctes. La première phase permet de créer la structure hiérarchique du modèle cible, alors que la deuxième consiste en la création des attributs et des références de ce dernier. OptimalJ [58] est un exemple d'approche dirigée par la structure.

Opérationnelles : cette catégorie regroupe les approches de transformation de modèles à manipulation directe telles que *QVT Operational* [20], *Kermeta* [34], *MTL* [77]. Dans ces approches, les formalismes définis au niveau des méta-modèles (ex. OCL) sont généralement enrichis par des constructions impératives.

Basées sur les gabarits : dans ces approches, des gabarits (*templates*) sont exprimés dans la syntaxe concrète du modèle cible. Ils sont agrémentés par du méta-code pour accéder au modèle source. Une approche basée sur les gabarits est donnée par Czarnecki et Antkiewicz [15].

Relationnelles : ce groupe d'approches est fondé sur les relations mathématiques. À l'instar de la programmation logique, ces approches permettent d'exprimer la relation entre les éléments sources et cibles en utilisant des contraintes auxquelles est rajoutée une sémantique exécutable. On retrouve dans cette catégorie *QVT Relationnal*, *Tefkat* [50].

Basées sur les graphes : plusieurs approches de TM sont fondées sur les travaux théoriques portant sur la transformation de graphes. Ces transformations sont généralement exprimées par des règles où, chaque règle, est composée d'une partie gauche (G) filtrant des fragments du modèle source et d'une partie droite (D) construisant des fragments du modèle cible. Parmi les approches basées sur les graphes, on retrouve *AGG* [74], *ATOM3* [17], *VIATRA* [75], *GReAt* [6], *BOTL* [55] et *MOLA* [36].

Hybrides : cette catégorie comprend les approches qui combinent différentes techniques des catégories identifiées précédemment. Par exemple une transformation exprimée en *ATL* [35] peut être déclarative, impérative ou hybride. QVT est également hy-

bride compte tenu qu'il comporte trois composantes *QVT Operational*, *QVT Relational* et *QVT Core*.

2.1.3.5 Jess

Nous utilisons dans ce mémoire Jess [27] en tant qu'environnement de transformation de modèles. Jess est un moteur de règles qui permet de stocker en mémoire des connaissances exprimées sous la forme de faits (*facts*). Il est possible ensuite de raisonner sur ces connaissances à l'aide de règles déclaratives. Jess applique les règles sur la base de faits en utilisant un filtrage par motifs basé sur l'algorithme *Rete* [25].

Il est possible d'associer la notion de fait en Jess au concept d'objet dans le paradigme orienté objet (POO). Chaque fait peut contenir plusieurs attributs, appelés « *slots* ». Jess offre également la possibilité de définir des gabarits de faits (*template*) qui correspondent à la notion de classe dans le POO.

Afin d'utiliser Jess pour la transformation de modèles, nous exprimons chaque modèle sous la forme d'un ensemble de faits. Chaque élément du modèle est un fait dont les *slots* sont des attributs ou des références vers d'autres éléments. Les méta-modèles sont exprimés, quant à eux, sous la forme de gabarits de faits. Le listing 2.1 présente un exemple trivial, où sont représentés, un méta-modèle et un modèle comportant un seul élément.

Listing 2.1 – Exemple d'une règle exprimée en Jess

```
(deftemplate  class (slot name(type STRING)) )

(assert (class (name Personne)))
```

Une transformation de modèles est représentée sous la forme d'un ensemble de règles Jess. À l'instar des transformations basées sur les graphes, chaque règle comporte une partie gauche (G) qui exprime les conditions nécessaires au déclenchement de la règle, ainsi que d'une partie droite (D) qui consiste en l'assertion d'éléments cibles. L'initialisation des attributs cibles par des attributs sources se fait en utilisant le même nom de variable. Le listing 2.1 présente une règle qui transforme des éléments « Classe » en des

éléments « Table » du même nom.

Listing 2.2 – Exemple d'une règle exprimée en Jess

```
(defrule Class-2-Table
    (class (name ?c))
                =>
    (assert (table (name ?c)))
)
```

Outre les faits et règles, Jess fournit un ensemble de fonctions mathématiques et de manipulation de chaînes. Cet ensemble peut être étendu par des fonctions définies par l'utilisateur. De plus, il est possible de créer en Jess des primitives de navigation qui parcourent les faits à la recherche de relations particulières.

2.2 Transformation de modèles par l'exemple

Comme souligné dans l'introduction de ce mémoire, l'écriture de transformations de modèles est une tâche difficile qui nécessite parfois beaucoup d'efforts et de temps. Dans ce sens, la transformation de modèle par l'exemple (TMPE) semble être une solution appropriée à cette problématique. De manière analogue à la programmation par l'exemple [30] et à l'interrogation par l'exemple [86], les approches de TMPE ont pour but d'apprendre automatiquement un programme de transformation à partir d'artéfacts fournis en guise d'exemples.

Il existe deux axes de recherche portant sur l'apprentissage de transformations de modèles par l'exemple : les approches par l'exemple proprement dites (TMPE), ainsi que celles par démonstration (TMPD). Nous explorons dans cette section la première catégorie, tandis que nous aborderons la seconde dans la section suivante.

Les travaux de TMPE œuvrent à dériver automatiquement un programme de transformation à partir d'un ensemble d'exemples fournis en entrée. Chaque exemple est une paire de modèles constituée d'un modèle source et d'un modèle cible. En plus des exemples, la majorité des approches de TMPE exploitent des traces fines de transformation. Ces traces sont des liens, de plusieurs-à-plusieurs, qui associent un groupe de

n éléments sources à un groupe de *m* éléments cibles. La figure 2.5 montre trois traces. Dans chaque trace, le fragment cible (en bas de la figure) est associé au fragment source (en haut de la figure). Les traces de transformation sont généralement définies par des experts du domaine.

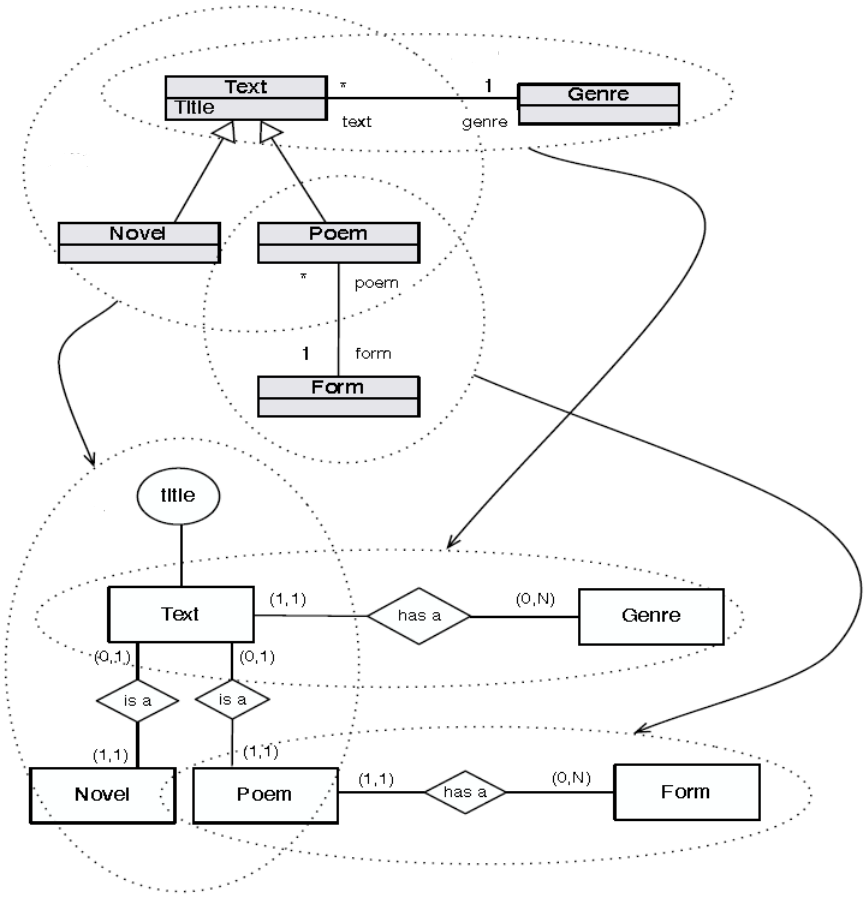

Figure 2.5 – Un exemple de traces entre un diagramme de classes et un modèle entité-relation [64]

En 2006, Varró [76] propose une première approche pour l'apprentissage de transformations à partir de paires d'exemples en utilisant un algorithme ad-hoc basé sur les graphes. L'algorithme prend en entrée des paires de modèles interreliés (accompagnés de traces) et dérive un ensemble de règles de transformation de type $1-1$. Le processus de dérivation est itératif et interactif, à chaque itération, les règles produites sont raffinées

par l'utilisateur. La contribution est ensuite automatisée davantage par Balogh et al. [7] en utilisant la programmation logique inductive (PLI). Cette nouvelle approche, bien que toujours itérative et incrémentale, permet de dériver des règles de transformation plus complexes (de type $n-m$).

Durant la même période Wimmer et al. [81] propose une approche similaire pour dériver des transformations sous la forme de règles de type $1-1$ exprimées en ATL. La contribution en question est également itérative et incrémentale, elle diffère cependant des deux contributions précédentes sur deux aspects. D'abord, les traces de transformation exploitées sont spécifiées dans une syntaxe concrète plutôt qu'abstraite. Ensuite, le processus de dérivation se fait selon une approche orientée objet, contrairement à celui de Varró où des graphes sont utilisés. La contribution de Wimmer est également améliorée dans des contributions subséquentes, par Strommer et al. [68, 69], où un langage pour la définition de correspondances de type $n-m$ est présenté, jumelé à un algorithme de raisonnement pour la dérivation de règles ($n-m$ également) à partir des correspondances exprimées. L'usage d'opérateurs de chaînes tel que la concaténation est également brièvement mentionné dans l'une des deux contributions.

Un autre travail qui s'inscrit dans le contexte des transformations de modèles par l'exemple est celui de Garcia-Magarino et al.[28] qui propose un algorithme permettant de générer des règles de transformation de type $n-m$, vraisemblablement dans plusieurs langages de transformation, à partir d'un ensemble de paires de modèles sources et cibles interconnectés (agrémentés par des traces). À l'instar de [76], l'approche utilise des graphes. Afin de pallier le fait que certains langages de transformations ne permettent pas d'écrire des règles de plusieurs à plusieurs, les règles sont d'abord dérivées dans un langage de transformation générique avant d'être transformées dans le langage de transformation souhaité (ATL dans l'article).

Kessentini et al. [39, 40] utilise des analogies pour effectuer une transformation. Contrairement aux contributions citées plus haut, cette approche ne produit pas de règles de transformation, mais dérive plutôt le modèle cible directement à partir du modèle source en considérant la transformation de modèles comme un problème d'optimisation. Le problème tel que posé est abordé en utilisant l'optimisation par essaims particulaires

(OEP) dans la première contribution et une combinaison de OEP et du recuit simulé (RS) dans la deuxième. L'approche d'apprentissage est ensuite améliorée dans [41] où des règles de transformation sont produites à partir de modèles sources et cibles qui ne sont pas accompagnés de traces de transformation. L'approche est validée sur une transformation de modèles comportementaux (diagramme de séquence vers réseau de Petri colorés).

Une autre contribution de TMPE qui ne produisait pas de règles de transformation initialement est celle de Dolques et al. [19]. Ce travail se base sur l'analyse relationnelle de concepts (ARC), une variante de l'analyse formelle de concepts, pour classifier les éléments sources et cibles ainsi que les correspondances fournies en entrée. Les patrons identifiés sont organisés dans des treillis partiellement ordonnés et sont ensuite analysés pour filtrer les plus pertinents. L'approche est également étendue par Saada et al. [62, 63] où des règles exécutables sont produites à partir des patrons filtrés. Dans cette dernière contribution, les règles sont exprimées en Jess.

Les travaux les plus récents dans le contexte de la TMPE sont ceux de Faunes et al. [23, 24] dans lesquels la programmation génétique (PG) est utilisée pour faire évoluer une population de transformations sur plusieurs générations jusqu'à produire la transformation attendue. Le processus de dérivation prend en entrée des paires d'exemples de modèles sources et cibles uniquement (sans traces de transformation) et produit en sortie des règles exécutables de type $n - m$. L'approche est ensuite améliorée par la contribution de Baki et al.[5] où le programme génétique tente d'apprendre simultanément les règles de transformation ainsi que le contrôle d'exécution qui doit être exercé sur celles-ci pour former un programme de transformation correct. Cette seconde version permet également de dériver des règles de transformations plus complexes en incluant la négation de conditions, l'usage de primitives de navigation ainsi que la considération des types et des domaines de définition lors de la construction du modèle cible.

2.3 Transformation de modèles par démonstration

La transformation de modèles par démonstration (TMPD) constitue le second axe œuvrant à l'apprentissage de TM par l'exemple. Elle forme un cas particulier des TMPE dont les différents travaux tentent d'apprendre les règles d'une transformation en exploitant les informations fournies par le déroulement d'une transformation qui est effectuée par un utilisateur.

Dans [11, 12], Brosch et al. proposent une approche pour dériver semi-automatiquement la spécification d'une transformation endogène à partir des opérations de modification effectuées par l'utilisateur sur un exemple concret. Le processus de dérivation se compose de deux phases. Dans la première phase, toutes les opérations atomiques effectuées par l'utilisateur sont récoltées. Au lieu d'enregistrer les actions de l'utilisateur en surveillant son environnement de modélisation, la méthode proposée se veut indépendante de l'outil utilisé. Pour cela, les auteurs utilisent le modèle initial, le modèle final ainsi que les différences qui existent entre eux. Ils contournent l'imprécision des heuristiques de comparaison de modèles en étiquetant préalablement chaque élément avec un identifiant unique. Durant la deuxième phase, les opérations sont agrégées dans un modèle de différences. Une version initiale des pré et post-conditions de chaque opération est également proposée à l'utilisateur qui peut la raffiner en spécifiant notamment comment des attributs complexes sont dérivés. Ces deux éléments (le modèle de différences et l'ensemble de pré et post-conditions) sont ensuite utilisés pour générer la transformation recherchée.

Sun et al. [70] proposent une approche similaire pour la dérivation de transformations endogènes. Contrairement à la contribution précédente, Sun et al. utilisent un module sous-jacent à l'environnement de modélisation, qui enregistre toutes les actions d'édition effectuées par l'utilisateur. Les opérations enregistrées sont analysées et filtrées afin de supprimer les opérations incohérentes ou inutiles. Un mécanisme d'inférence est ensuite utilisé afin de dériver les intentions de l'utilisateur sous la forme de patrons de transformation réutilisables décrivant les préconditions et les actions de chaque transformation. L'approche proposée est entièrement automatisée et ne requiert pas que la transforma-

tion dérivée soit affinée par l'utilisateur. À l'instar de [12] l'approche prend en charge certaines dérivations complexes d'attributs (ex. opérations arithmétiques). Cependant, contrairement ces opérations n'ont pas à être ajoutées lors d'une étape de raffinement, mais sont plutôt démontrées par les actions de l'utilisateur.

À ce jour, les derniers travaux portant sur la TMBD sont ceux de Langer et al. [48]. Dans leur contribution Langer et al. étendent le travail de Brosch et al. [12] en utilisant les opérations effectuées par un utilisateur afin de dériver des transformations exogènes en trois phases. Semblablement au travail de Brosch et al. la première phase invite l'utilisateur à transformer un modèle source dans l'éditeur de son choix. L'utilisateur rajoute des éléments sources et leurs correspondants cibles au fur et à mesure. Il peut également précéder certaines actions par la sélection d'un contexte pour indiquer la dépendance de la transformation qu'il s'apprête à effectuer par rapport à d'autres déjà illustrées. Durant la seconde phase, le scénario de transformation est généralisé dans des gabarits qui, dans les cas de dépendances, utilisent les mécanismes de comparaison introduits dans [12] pour isoler les éléments cibles nouvellement créés. Les gabarits dérivés peuvent être raffinés manuellement. Ils sont ensuite utilisés, durant une troisième et dernière phase, pour générer des règles grâce à une transformation d'ordre supérieur.

Comme souligné par [37], les approches d'apprentissage de transformations de modèles par l'exemple se concentrent exclusivement sur les transformations exogènes, tandis qu'à l'exception de [48], toutes les approches par démonstration se focalisent, elles, sur des transformations endogènes.

2.4 Synthèse

Le tableau 2.I résume les contributions qui s'inscrivent dans le cadre de l'apprentissage de transformation de modèles par l'exemple. On peut constater que ces contributions ne prennent pas en charge la dérivation complexe d'attributs cibles, exceptée [69] où la concaténation de chaînes de caractères est brièvement mentionnée. D'autre part, bien que les approches par démonstration permettent de rajouter ou de démontrer certaines dérivations complexes. Ces travaux demeurent, comme souligné par Sun et al. [70],

plus appropriés pour des transformations aux structures et attributs explicites, plutôt que celles aux relations et aux traitements implicites. L'usage de fonctions max ou min sur les attributs d'un ensemble d'éléments est, par exemple, difficile à démontrer.

Excepté [5], tous les travaux de TMPE et de TMPD actuels se limitent à la dérivation de conditions portant sur la présence d'éléments sources. Ils ne permettent donc pas de dériver des conditions qui explorent le contexte global dans lequel se trouvent ces éléments. Cette limite d'expressivité exclut également la possibilité d'apprendre de nombreuses transformations qui requièrent, par exemple, la vérification des valeurs de certains attributs sources.

Enfin, une limite commune de ces travaux réside dans la validation des approches proposées. En effet, la plupart des contributions présentées plus haut illustrent leurs approches respectives par des problèmes de transformation classiques, accompagnés d'exemples simplifiés, qui ne montrent pas clairement la complexité et l'exactitude des transformations pouvant être apprises.

Tableau 2.I – Approches de transformation de modèles par l'exemple

Approche	Algorithme	Entrée	Sortie	Règles N-M	Contrôle	Contexte	Contraintes de valeurs	Dérivations complexes
Approches par l'exemple								
Varró [76]	Ad-hoc	Exemples et Traces	Règles	1-1	Non	Non	Non	Non
Wimmer et al. [81]	Ad-hoc	Exemples et traces	Règles	1-1	Non	Non	Non	Non
Strommer et al. [68, 69]	Filtrage par motif	Exemples et Traces	Règles	N-M	Non	Non	Non	Mentionnées seulement
Kassentini et al. [39, 40]	OEP/OEP-RS	Exemples et Traces	MC	-	Non	Non	Non	Non
Balogh et al. [7]	PLI	Exemples et traces	Règles	N-M	Non	Non	Non	Non
Garcia et al. [28]	Algorithme ad-hoc	Exemples et Traces	Règles	N-M	Non	Non	Non	Non
Kassentini et al. [41]	OEP-RS	Exemples	Règles	1-M	Non	Non	Non	Non
Deloques et al. [19]	ARC	Exemples et traces	Patrons	-	Non	Non	Non	Non
Saada et al. [62, 63]	ARC	Exemples et traces	Règles	1-M	Non	Non	Non	Non
Faunes et al. [23, 24]	PG	Exemples	Règles	N-M	Non	Non	Non	Non
Baki et al. [5]	PG	Exemples	Règles	N-M	Oui	Oui	Non	Non
Approches par la démonstration								
Brosch et al. [11, 12]	Filtrage par motif	Actions de l'utilisateur	Règles	-	-	Non	Non	Ajoutées manuellement
Sun et al. [70]	Filtrage par motif	Actions de l'utilisateur	Règles	-	-	Non	Non	Démontrées par l'utilisateur
Langer et al. [48]	Filtrage par motif	Actions de l'utilisateur	Règles	-	Oui	Non	Non	Ajoutées manuellement

CHAPITRE 3

PROCESSUS D'APPRENTISSAGE

Dans l'approche d'apprentissage que nous proposons, nous remédions à l'explosion de l'espace de recherche ayant lieu lors de l'apprentissage de transformations de modèles complexes. Nous proposons un processus d'apprentissage composé de trois phases, fondé sur l'usage d'heuristiques. Nous adaptons également les heuristiques utilisées à la progression de la recherche, afin d'assurer un bon compromis entre leurs performances (la solution proche de l'optimal trouvée) et leur puissance d'exploration.

Ce chapitre est divisé en cinq parties. La première partie motive notre travail à travers un exemple qui expose certaines particularités qui caractérisent des transformations considérées comme étant complexes. Suite à cela, nous donnerons dans la seconde partie un aperçu de l'approche que nous proposons et qui se compose de trois phases. Enfin, les trois dernières parties du chapitre serviront à détailler chacune des trois phases du processus d'apprentissage proposé.

3.1 Objectifs de l'approche

L'approche que nous proposons s'intéresse aux transformations exogènes de modèle à modèle. Selon les définitions fournies dans la section 2.1.3, nous excluons ainsi les transformations portant sur des modèles sources et cibles appartenant aux mêmes méta-modèles, ainsi que celles dont le modèle cible est un artéfact autre qu'un modèle. L'objectif de notre approche est d'apprendre, à partir d'exemples, des transformations considérées comme étant complexes et qui ne sont pas prises en charge par les contributions identifiées dans les sections 2.2 et 2.3.

Afin de mettre en évidence les défis rencontrés lors de l'apprentissage de certaines TM, considérons le problème de transformation d'un diagramme de classes UML en un schéma relationnel (DC-à-SR), dont l'objectif est de générer automatiquement un schéma relationnel à partir d'un diagramme de classes UML.

Dans cette transformation, plusieurs éléments du modèle source sont transformés de différentes façons, selon le contexte global dans lequel ils apparaissent. Par exemple, lorsqu'une association $n - m$ possède une classe d'association, elle est transformée en une paire de clés étrangères dans la table correspondante à ladite classe d'association. Cependant, lorsque cette même association ne possède pas de classe d'association, une table possédant le nom de l'association doit être créée.

Un autre exemple d'une dépendance au contexte concerne les associations $1 - 1$. Une pratique répondue consiste à fusionner les classes reliées à travers une association $1 - 1$ dans une seule table, lorsqu'une des deux classes ne possède pas d'autres liens (c.-à-d., qu'elle n'est pas impliquée dans une autre association ou relation d'héritage). Si ce n'est pas le cas, une table est créée pour chaque classe. Les deux situations sont illustrées par l'exemple de la figure 3.1. Dans la trace $TR1$, la classe *Adresse* ne possède pas de lien autre que celui avec la classe *Personne*, par conséquent, une seule table est créée. Inversement, dans la trace $TR2$, les deux classes *Employe* et *Poste* possèdent au moins un autre lien. Ainsi, deux tables doivent être créées et reliées par des clés étrangères. Cet exemple illustre le fait que pour certains scénarios de transformation, un contexte plus grand que celui fourni par les traces doit être exploré. Ce cas de figure requiert l'apprentissage de règles complexes comportant des conditions sophistiquées.

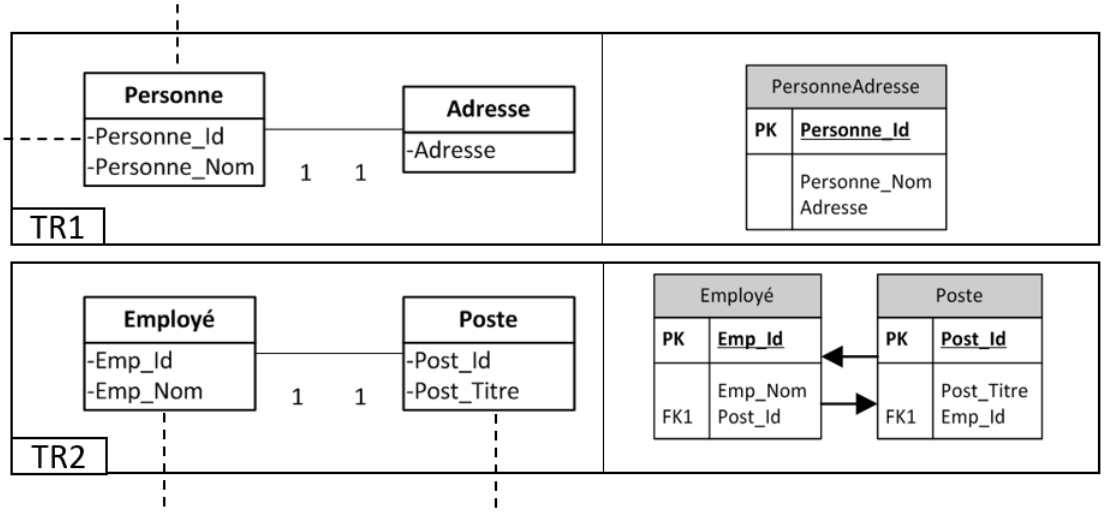

Figure 3.1 – Deux fragments d'un modèle source transformés différemment

En plus du problème de la dépendance au contexte, il existe de nombreuses transformations où des attributs d'éléments cibles ne correspondent pas directement à ceux des éléments sources, mais sont plutôt le résultat d'un calcul appliqué sur des nombres (addition, soustraction, maximum, etc.), ou d'un traitement appliqué sur des chaînes de caractères (vers minuscule, concaténation, etc.). Par exemple, lors de la fusion de deux classes reliées par une association $1-1$, il serait raisonnable de requérir que le nom de la table résultante de cette transformation soit une concaténation des deux noms de classes (voir fig. 3.1 - T1).

D'autres cas de transformations nécessitant un traitement pour dériver des attributs cibles sont ceux dont le modèle cible est une représentation graphique [13, 59] du modèle source. Dans ces transformations, des opérateurs ainsi que des fonctions numériques sont souvent indispensables pour réaliser certaines manipulations, en particulier, pour les attributs de coordonnées et de tailles.

Enfin, il est aussi courant de rencontrer des problèmes de transformation pour lesquels un élément du modèle source peut être transformé en différents éléments cibles, en fonction de son stéréotype ou de la valeur que prend l'un de ses attributs. Dans ce dernier cas, ceci est généralement vrai pour des éléments sources possédant des attributs de type énumération où l'ensemble des valeurs possibles que pourrait prendre l'attribut est fini.

Ces trois exigences, à savoir (1) l'exploration du contexte, (2) l'évaluation des valeurs d'attributs sources et (3) les dérivations complexes d'attributs cibles doivent être considérées lors de l'apprentissage de TM complexes. Alors que la taille de l'espace de recherche justifie, dans ce genre ce cas, l'usage d'heuristiques, deux considérations doivent être prises en compte lors de l'apprentissage. La première est l'explosion de l'espace de recherche qui a lieu suite à l'introduction des opérateurs numériques et de traitement de chaînes. La seconde est que, les algorithmes de recherche, notamment les algorithmes évolutionnaires, peuvent être sujet à une convergence sous-optimale *(local optimum convergence)*, aussi connu sous le nom de convergence prématurée [3]. Dans la suite de ce chapitre, nous décrivons le processus d'apprentissage que nous proposons pour dériver des TM complexes tout en faisant face à ces deux problèmes.

3.2 Aperçu de l'approche

Comme illustré dans la section 3.1, l'apprentissage de transformations de modèles complexes, en utilisant un algorithme de recherche, requiert dans certains cas l'exploration d'un espace de recherche d'une très grande taille qui ne peut pas être réalisée dans un temps raisonnable. Afin de réduire la taille de cet espace de recherche, nous exploitons des traces de transformation dans le but d'éviter de prendre en considération simultanément les trois caractéristiques de transformation identifiées précédemment, à savoir, (1) l'exploration du contexte dans le modèle source, (2) l'évaluation des valeurs d'attributs sources et (3) les dérivations complexes d'attributs cibles.

Notre processus d'apprentissage prend en entrée un ensemble de modèles sources et cibles agrémentés par leurs traces et produit en sortie un programme de transformation déclaratif et exécutable. Afin de simplifier la description de notre approche, nous décrirons cette dernière en considérant en entrée une seule paire d'exemples. Le processus proposé peut-être généralisé avec succès à plusieurs paires d'exemples. Les trois phases du processus d'apprentissage sont illustrées dans la figure 3.2.

- ***Analyse des traces de transformation*** : l'objectif de cette phase est d'analyser les correspondances entre les fragments sources et cibles de la paire d'exemples fournie en entrée. Ces traces peuvent être fournies par des experts du domaine ou récupérées de manière (semi-)automatique en utilisant, par exemple, une approche telle que celle proposée par Saada et al. [64], ou encore, l'approche de Grammel et al. [29]. Durant l'analyse, les traces sont séparées dans des *pools*, comme illustré dans la figure 3.2 - 1. Les traces sont séparées, de façon à ce que soient regroupés dans chaque *pool*, les fragments sources du même type (voir paragraphe 3.3 pour une définition) qui sont transformés de manière identique. Une version révisée de la paire d'exemples initiale est également ajoutée à chaque *pool*. Cette dernière exclut les éléments des modèles source et cible qui ne sont pas présents dans les traces ajoutées au *pool* en question.

- ***Apprentissage des règles de transformation*** : durant cette phase, nous utilisons les traces, ainsi que la paire d'exemples de chaque *pool*, pour dériver un ensemble

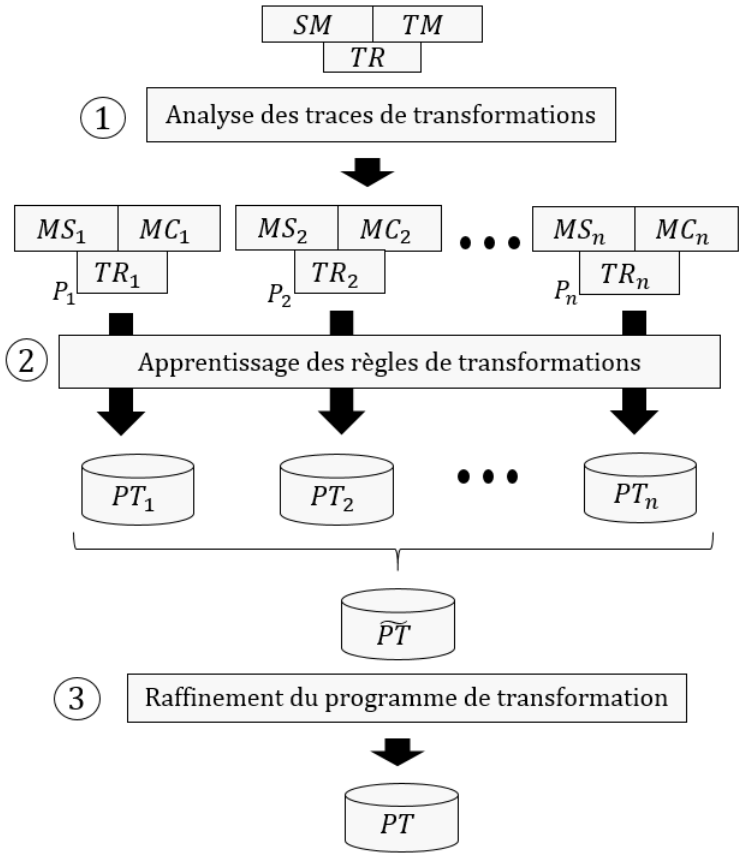

Figure 3.2 – Processus d'apprentissage d'une transformation de modèles

de règles en utilisant un programme génétique (PG). Nous nous sommes assurés (grâce à l'étape précédente) que toutes les traces ayant un même type de fragment source sont transformées de la même manière. De ce fait, chaque exécution de PG est dispensée, non seulement d'explorer un contexte plus large que celui des traces, mais aussi, d'évaluer les valeurs d'attributs sources. Ceci permet de réduire la taille de l'espace de recherche, et par conséquent, d'introduire des dérivations complexes d'attributs cibles dans le processus d'apprentissage. Cette phase produit un programme de transformation pour chaque *pool* (fig. 3.2 - 2). Chaque programme est ensuite automatiquement traité avant d'entamer la dernière phase. Ce post-traitement a pour but d'éliminer d'éventuelles règles incorrectes ou redondantes.

- ***Raffinement du programme de transformation*** : pendant la troisième phase, les programmes de transformation dérivés dans chaque *pool* sont fusionnés dans un seul programme. Ce dernier est ensuite utilisé comme une solution initiale à raffiner par un algorithme du recuit simulé (fig. 3.2 - 3). Alors que la deuxième phase se concentre sur la dérivation de règles qui transforme correctement les fragments sources en fragments cibles, l'objectif de cette phase est de raffiner ces règles afin de garantir qu'elles soient correctes dans un contexte plus général, c.-à-d, la globalité du modèle source. Ceci est réalisé en enrichissant les règles dérivées par des conditions plus sophistiquées qui pourraient explorer un contexte plus grand, tester des références ou des attributs du modèle source, conditionner les règles par l'absence de certains patrons (négation). Cette phase permet également de rajouter un contrôle implicite [5] au programme de transformation, en permettant de tester des patrons cibles au niveau des conditions des règles.

3.3 Analyse des traces de transformation

Soit une paire d'exemples de modèles source et cible (SM, TM). Une trace de transformation est un ensemble de correspondances $TR = \{T_1, T_2, ..., T_m\}$, où chaque correspondance est une paire de fragments source et cible $T_i = (FS_i, FC_i)$, $i \in \{1..m\}$. Un fragment source (resp. cible) est un ensemble d'éléments sources (resp. cibles). Chaque fragment source contient un élément principal qui est l'élément transformé. Formellement, $T_i = (\{es_{i0}..es_{ip}\}, \{ec_{i0}..ec_{iq}\})$ $p > 0, q > 0$ où es_{i0} est l'élément principal de FS_i. On dit alors qu'une trace est de type t, notée T_i^t, si son élément principal est de type $t \in MMS$.

Compte tenu du fait que nous utilisons un algorithme de recherche, il n'est pas nécessaire que les traces soient précises. Par exemple, le fragment cible identifié comme étant correspondant à une *association* $1-n$ peut inclure la table référencée en plus de la table qui référence et de sa clé étrangère. Dans un cadre réel, il est plus facile pour des experts d'identifier des fragments correspondants plutôt que des éléments cibles bien spécifiques.

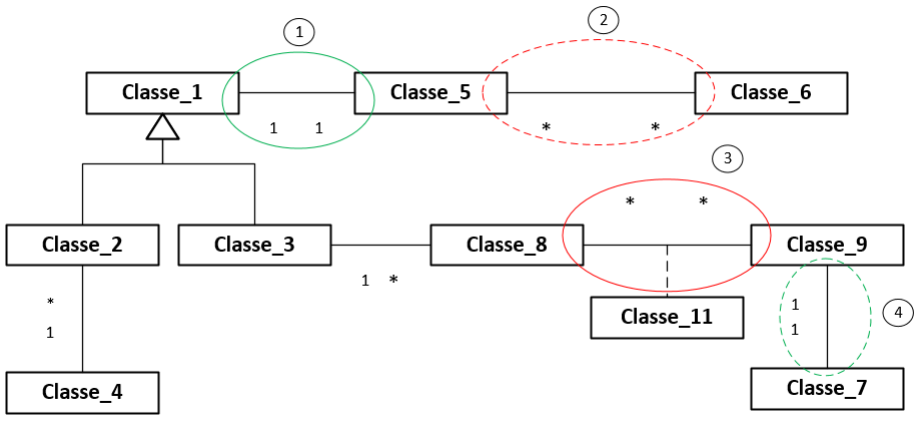

Figure 3.3 – L'identification de traces conflictuelles

Après avoir récolté l'ensemble des traces associées à la paire d'exemples fournie en entrée, chaque trace est automatiquement complétée en agrémentant son fragment source par les éléments du modèle source qui sont référencés par son élément principal, mais qui sont absents de la trace en question. Les éléments référencés par un élément principal sont susceptibles d'être utilisés pour transformer ce dernier. Dans notre exemple, si l'expert définit une correspondance entre une *association* $1-1$ et une table à travers une trace dont le fragment source ne contient que l'association elle-même, alors le fragment source de cette trace sera complété avec les deux classes qui sont référencées par l'association en question.

Les traces de transformation sont ensuite analysées et celles en conflit sont séparées dans des *pools* distincts. On dit que deux traces sont en conflit si leurs fragments sources respectifs sont du même type, mais qu'ils sont transformés différemment. Par exemple, considérons le modèle source représenté dans la figure 3.3. Les deux *associations* $1-1$ (fragments ① et ④) sont transformées différemment du fait que ① résultera en deux tables échangeant leurs clés, alors que le fragment ④ sera transformé en une seule table. La raison pour laquelle de telles traces sont séparées en plusieurs *pools*, réside dans le fait qu'essayer d'apprendre une transformation en utilisant des traces ayant des fragments sources similaires mais des fragments cibles différents, conduira à apprendre des règles de transformation contradictoires. Ces règles auront les mêmes conditions dans leurs

parties gauches, mais différentes actions dans leurs parties droites respectives.

Il est à noter que des traces conflictuelles de différents types (voir la définition de type de trace plus haut) peuvent être regroupées arbitrairement dans les mêmes *pools*. Dans le modèle source illustré dans la figure 3.3, les fragments ① et ② (resp. ③ et ④) auraient pu être groupés dans le même *pool* dans un autre scénario.

En fonction du nombre de conflits dans chaque sous-ensemble de traces du même type, *n pools* peuvent être construits (fig. 3.2-2). Chaque *pool* P_k, $k \in 1..n$ contiendra alors un ensemble de traces $TR_k = \{T_{k_1}, T_{k_2}, ..., T_{k_{mk}}\}$. Les traces d'un type lambda qui sont transformées de la même façon sont tout simplement dupliquées à travers tous les *pools*. Par conséquent, dans la figure 3.3, toutes les traces de type *class* et *heritage* sont directement copiées dans chaque *pool*. Nous avons donc : $TR = \bigcup_{k=1}^{n} TR_k$ et $\bigcap_{k=1}^{n} TR_k \neq \emptyset$.

Durant l'étape finale de cette phase, une version révisée de la paire de modèles initiale est ajoutée à chaque *pool*. La paire d'exemples de chaque *pool* (MS_k,MC_k) est ajustée en fonction des traces présentes dans ledit *pool*, en supprimant les éléments sources et cibles qui ne sont pas présents dans les traces du *pool* en question. Pour le modèle source de la figure 3.3, le premier *pool* contiendra le modèle source initial excepté les éléments dans ② et ④, alors que le modèle source du second *pool* exclura les éléments dans ① et ③.

3.4 Apprentissage des règles de transformation

L'objectif de cette phase est de dériver un programme de transformation PT_k pour chaque *pool* de traces P_k tel que $PT_k(MS_k) = (MC_k)$. La construction automatique de programmes est un des enjeux les plus importants de l'apprentissage machine. Dans notre processus d'apprentissage, nous utilisons à cette fin la programmation génétique (PG), une des techniques les plus prometteuses dans le domaine [79].

Cette section du mémoire décrit les détails de la 2e phase du processus d'apprentissage. Elle est divisée en trois parties. Nous exposons dans un premier temps quelques notions de base sur la programmation génétique. Nous détaillerons ensuite notre adap-

tation de la PG au problème d'apprentissage de règles de transformation. Enfin, nous décrirons l'étape de raffinement par laquelle se conclut cette phase.

3.4.1 Programmation génétique

La programmation génétique (PG) est une méthode systématique, permettant aux ordinateurs de résoudre un problème de manière automatique à partir d'un énoncé de haut niveau de la tâche à accomplir [45]. Elle forme une branche des algorithmes évolutionnaires (AE), un ensemble d'algorithmes destinés à l'optimisation et à l'apprentissage, qui imitent l'évolution naturelle des espèces qui s'adaptent au fil des générations à leur environnement. Cette famille d'algorithmes, incluant la PG, se caractérise par les points suivants [84] :

Basée sur les populations : les AE font évoluer un ensemble de solutions en parallèle, cet ensemble est appelé population.

Orientée vers la fitness : chaque solution est un membre de la population appelé individu. Afin d'imiter le concept de la sélection naturelle (survie des meilleurs), l'algorithme associe à chaque individu une mesure de sa performance appelée valeur fitness.

Dirigée par la variation : chaque individu subit, à travers les générations, un ensemble d'opérations destinées à imiter les variations génétiques que subissent les différentes espèces.

3.4.1.1 Aperçu de la PG

La figure 3.4 illustre le déroulement d'une exécution d'un programme génétique. L'algorithme démarre par une population initiale de programmes, générés généralement de manière aléatoire. Chaque individu est ensuite évalué afin de mesurer sa capacité à résoudre le problème donné. L'évaluation se fait en utilisant une fonction fitness qui associe à chaque programme un rang. Suite à cela, une nouvelle génération est dérivée à partir de la génération actuelle, en utilisant trois opérateurs génétiques, à savoir, la reproduction, le croisement et la mutation.

Le processus de dérivation est conçu de manière à favoriser la survie et la reproduc-

tion des meilleurs individus, tout en laissant une chance aux moins bons de s'améliorer. Une fois la nouvelle génération obtenue, les individus de cette dernière sont évalués à leur tour. Le processus itère ainsi, améliorant la qualité de la population de génération en génération.

L'exécution du programme génétique se poursuit jusqu'à satisfaire la condition d'arrêt spécifiée par l'utilisateur. Le meilleur programme est alors retourné.

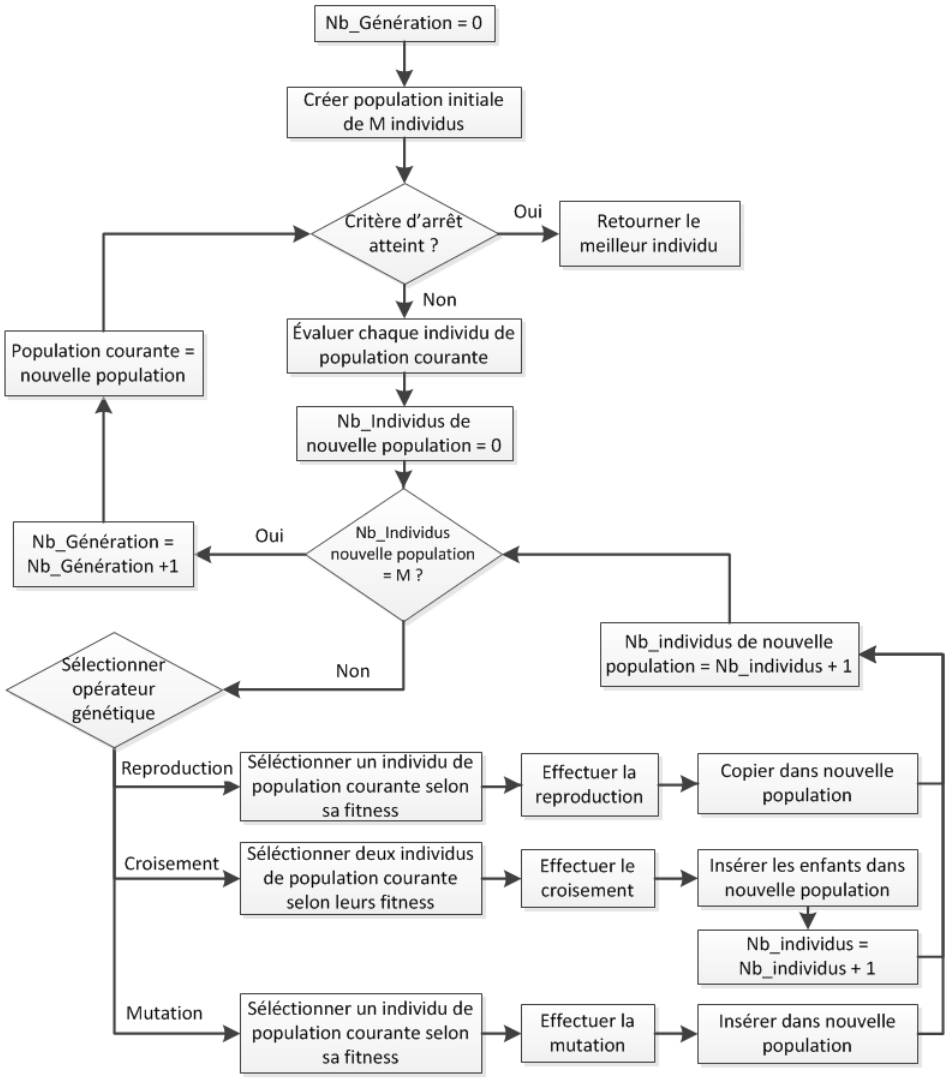

Figure 3.4 – Algorigramme de la programmation génétique

3.4.1.2 Spécification du problème

L'objectif de la programmation génétique est de dériver, suite à plusieurs itérations, un programme qui permettrait de réaliser la tâche spécifiée. Afin d'appliquer cette approche sur un problème en particulier, il est nécessaire de définir quatre éléments majeurs (Fig 3.5).

- Les terminaux et fonctions : ces deux ensembles délimitent l'espace de recherche. Ils constituent les gènes de l'individu, c.-à-d., les briques dont peut être composé le code de chaque programme construit par l'algorithme. Les terminaux sont les variables et les constantes utilisées par le programme. L'ensemble des fonctions peut, quant à lui, s'étendre de simples opérateurs arithmétiques à des fonctions, plus complexes, spécifiques au domaine d'application.

- La fonction de fitness : cette composante est critique dans la mesure où elle désigne l'objectif de la recherche. La fonction de fitness constitue le moyen principal de communiquer la description de la tâche à accomplir au programme génétique. Elle permet de mesurer la capacité de chaque programme à résoudre le problème posé.

- Les paramètres de contrôle de l'exécution : ces éléments permettent de contrôler l'exécution du programme génétique. Parmi les paramètres d'exécution les plus importants, on retrouve la taille de la population, la taille maximale de chaque programme, ainsi que la probabilité d'appliquer chaque opérateur génétique.

- Le critère d'arrêt du programme : l'arrêt du programme génétique peut être conditionné par un nombre maximal de générations, ou par l'atteinte d'un certain résultat. Le meilleur individu de la génération est alors retourné à l'utilisateur.

3.4.1.3 Représentation des individus

Dans la programmation génétique, chaque individu de la population est un programme. Chaque programme est représenté sous la forme d'un arbre syntaxique, par exemple, l'expression $2+3x$ est représentée par l'arbre de la figure 3.6.

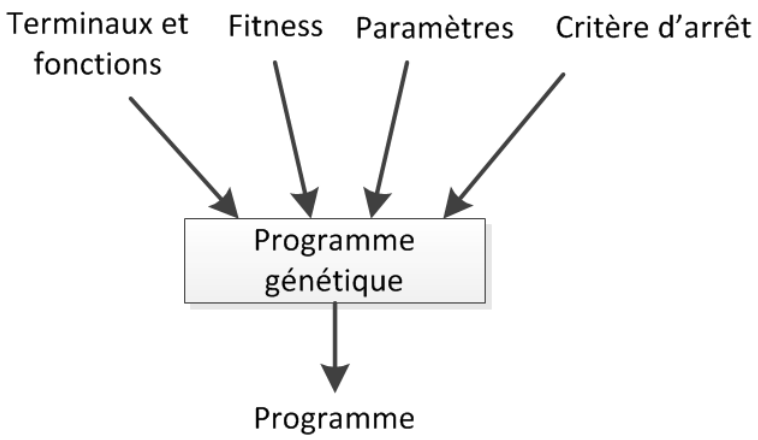

Figure 3.5 – Éléments à fournir au programme génétique

Les nœuds de l'arbre indiquent les instructions du programme à exécuter. Ses liens représentent, quant à eux, les arguments de ces instructions. Il est possible de distinguer ainsi deux types de gènes (nœuds), les fonctions et les terminaux. Les terminaux forment les feuilles de l'arbre, alors que les fonctions sont des nœuds qui possèdent des liens vers d'autres nœuds (donc des arguments).

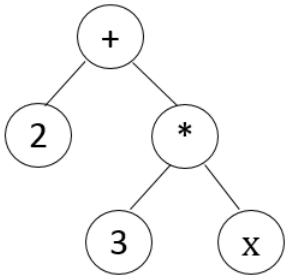

Figure 3.6 – Représentation en arbre d'un programme en PG

3.4.1.4 Génération de la population initiale

Une fois l'ensemble des terminaux défini, la première étape de l'algorithme consiste à créer une population initiale de programmes générés aléatoirement. La taille de la population initiale et des populations subséquentes est un paramètre fourni par l'utilisateur.

Koza [45] définit trois méthodes de génération possibles *Grow, Full, Ramped half-and-hal*. Indépendamment de la méthode utilisée, il recommande que cette première génération n'ait pas d'individus identiques. Étant donné que cette étape est une exploration aléatoire de l'espace de recherche, la génération initiale contiendra des individus probablement moins performants que l'algorithme fera évoluer. Il est néanmoins possible de définir un seuil de fitness à partir duquel les individus générés sont acceptés. Cette alternative permet d'assurer un minimum de qualité lors de la construction de la population initiale.

3.4.1.5 Évaluation des individus

À chaque fois qu'une génération est complétée, chaque programme est évalué par la fonction objectif définie. La qualité d'un individu peut être mesurée de plusieurs façons, selon le problème abordé. La fonction fitness peut mesurer, par exemple, l'écart entre le résultat produit par le programme et le résultat escompté, la capacité du programme à classifier des objets ou à maximiser/minimiser une valeur donnée. Souvent, la fitness de chaque individu est mesurée sur un ensemble d'entrées possibles.

3.4.1.6 Dérivation de nouveaux individus

La sélection

Cette phase s'inspire de la sélection naturelle des espèces. Pour dériver une nouvelle génération, les individus de la génération actuelle sont sélectionnés d'une manière probabiliste, afin de subir des opérations génétiques. Les meilleurs individus ont plus de chance de participer aux opérations génétiques, sans que cela exclue pour autant les éléments les plus mauvais de la population. Plusieurs procédures de sélection ont été décrites dans la littérature, nous présentons ci-dessous deux approches de sélection très répandues :

- **Sélection par roulette :** cette procédure de sélection est similaire à une roue de la fortune biaisée. Les individus sont représentés par des secteurs sur cette roue. La

roue est dite « biaisée » car la taille de chacun de ses secteurs est proportionnelle à la fitness de l'individu qu'il représente. Lorsque la roue est lancée, les meilleurs individus ont plus de chance d'être choisis, étant donné que leurs secteurs occupent plus d'espace sur la roue.

- **Sélection par tournoi :** cette procédure consiste à sélectionner de la population N individus au hasard, et à prendre le meilleur d'entre eux. Le nombre N représente la taille du tournoi, si $N = 2$ le tournoi est dit binaire. La taille du tournoi permet de contrôler le biais de la sélection envers les meilleurs individus. Lorsque N est grand, les individus les moins bons ont moins de chance d'être sélectionnés.

Tant que la nouvelle génération n'est pas complète, de nouveaux individus sont produits en appliquant des opérations génétiques sur les éléments sélectionnés. Les opérateurs génétiques les plus utilisés sont : la reproduction, le croisement et la mutation.

Reproduction

Elle consiste à copier certains individus dans la nouvelle population. L'objectif de cette opération est double. D'abord, elle permet de préserver les individus considérés comme prometteurs pour qu'ils ne soient pas altérés par les autres opérateurs génétiques, ensuite, elle permet de réduire les coûts associés à l'évaluation des individus, étant donné que les éléments reproduits n'ont pas à être évalués lors des générations subséquentes. La littérature suggère d'autoriser environ 10% de la population à se reproduire [79], cette proportion permet dans de nombreux cas d'accélérer la convergence de l'algorithme tout en préservant la diversité de la population.

Croisement

À l'instar de la reproduction sexuelle des organismes, l'opérateur de croisement permet de produire deux nouveaux individus (appelés enfants) en combinant les gènes de deux individus (appelés parents) sélectionnés de la population actuelle. La méthode de croisement la plus commune consiste à choisir un seul point de coupe aléatoirement

dans chaque arbre parent, et d'inverser au niveau des enfants produits, les sous arbres des parents qui ont pour racines le point de coupe sélectionné (voir figure 3.7).

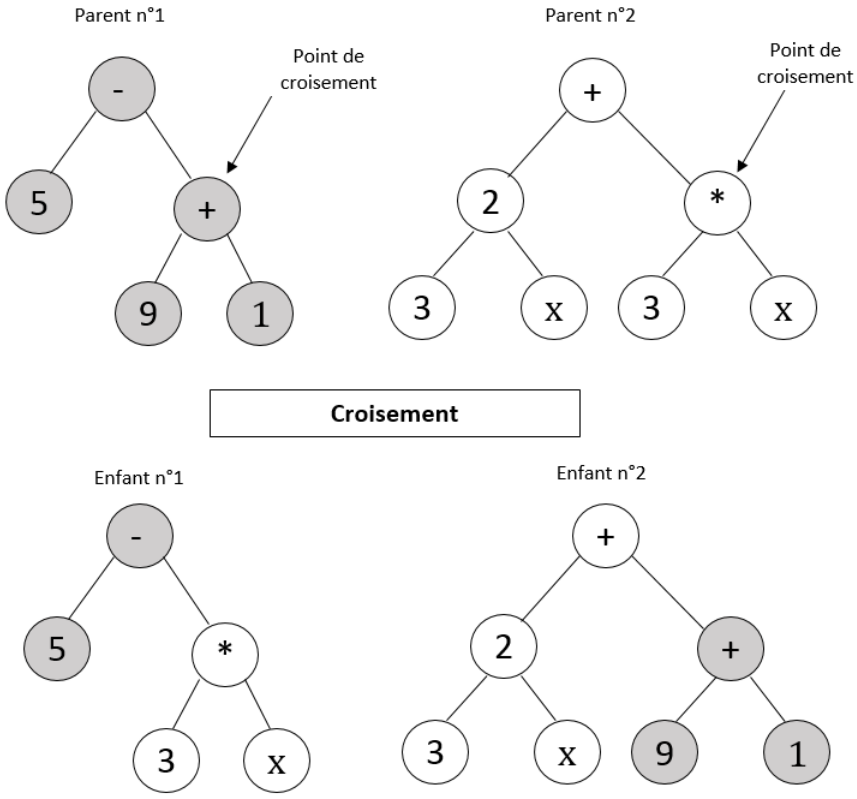

Figure 3.7 – Opération de croisement à un seul point

Outre la méthode à un seul point, il existe plusieurs autres stratégies de croisement telles que le croisement à deux points et le croisement uniforme. Le croisement est souvent l'opérateur génétique dominant dans les algorithmes génétiques [54], la probabilité de croisement utilisée est généralement très élevée (de l'ordre de 0.9).

Mutation

Cette opération permet d'introduire du nouveau matériel génétique dans la population. La procédure de mutation classique consiste à modifier aléatoirement un terminal ou un sous-arbre de l'individu choisi. Une deuxième forme de mutation, connue sous le

nom de *headless chicken crossover*, consiste à effectuer un croisement entre l'individu en question et un nouvel individu généré aléatoirement.

La mutation permet d'augmenter la diversité de la population. Contrairement à l'opérateur de croisement, la majorité des travaux qui utilisent la PG optent pour une probabilité de mutation assez basse (de l'ordre de 0.1).

3.4.2 Apprentissage de règles en utilisant la PG

Afin d'utiliser la programmation génétique pour dériver un programme de transformation, il est nécessaire de définir les éléments abordés dans les paragraphes précédents, à savoir, (1) l'encodage d'un programme, (2) la création de la population initiale, (3) l'évaluation des programmes de chaque génération et enfin, (4) la dérivation de nouvelles générations de transformations. Nous détaillons dans ce qui suit ces quatre aspects.

3.4.2.1 Encodage des programmes de transformation

Pour un *pool* P_k, nous définissons un programme candidat de transformation PT_k comme étant un ensemble de groupes de règles $PT_k = \{GR_k^t, t \in MMS\}$, où chaque groupe de règles $GR_k^t = \{R_{k_1}^t, R_{k_2}^t, ..., R_{k_x}^t\}$ transforme une trace de type $t \in MMS$.

Chaque règle de transformation est encodée comme un couple $R_{k_i}^t =< G_{k_i}^t, D_{k_i}^t >$, $i \in 1..x$. $G_{k_i}^t$ est la partie gauche de la règle et est composée de patrons à rechercher dans le modèle source MS_k. $D_{k_i}^t$ est la partie droite et comporte les patrons à instancier dans le modèle cible (voir listing 3.1).

$G_{k_i}^t$ détermine les conditions qui doivent être satisfaites par MS_k pour que la règle se déclenche. Elle se compose d'une ou plusieurs briques interconnectées. Une brique est un ensemble d'éléments sources interconnectés. Chaque brique contient un élément source principal et respecte les cardinalités minimales (définies dans le méta-modèle) de ce dernier ; par conséquent, une brique de type *class* contiendrait un seul élément alors qu'une brique de type *inheritance* contiendrait trois éléments, la relation d'héritage ainsi que les deux classes référencées par cette dernière.

On dit qu'une brique est de type t si son élément source principal est de type $t \in$

MMS. Par exemple, la partie G de la règle affichée dans le listing 3.1 contient deux briques. Une brique de type *inheritance* et une autre de type *attribute*. G_{kt_i} peut contenir un nombre arbitraire de briques, cependant, elle doit contenir au moins une brique de type t, le type de traces transformées par le groupe de règles auquel appartient la règle. De plus, G_{kt_i} ne peut pas contenir des éléments de type $t' \in MMS$ si t' n'apparaît pas dans le fragment source de TR_{kt}, l'ensemble de traces de type t présentes dans le *pool* k.

Les briques doivent être interconnectées pour être mises en correspondance avec un fragment concret du modèle source durant l'exécution de la transformation. L'interconnexion est effectuée à travers des éléments communs. Par exemple, dans le listing 3.1, la partie gauche de la règle contient deux briques, une de type *inheritance* et l'autre de type *attribute*. La première met en œuvre une classe et sa superclasse alors que la seconde implique un attribut et sa classe. Les deux briques sont reliées par le fait que la superclasse dans la première brique est la classe de l'attribut dans la deuxième brique.

Étant donné que l'objectif de cette phase est de transformer les fragments sources en fragments cibles, sans pour autant avoir une transformation complète, la partie G n'inclut pas des conditions négatives, des briques du modèle cible, des primitives de navigation ou l'évaluation des valeurs que prennent les attributs. L'objectif de ces restrictions est de réduire la taille de l'espace de recherche en se concentrant, dans un premier temps, uniquement sur certains aspects de la transformation.

Listing 3.1 – Exemple d'une règle exprimée en Jess

```
(defrule example_rule

(inheritance(class ?c00)(superclass ?c10))
(class(name ?c00))
(class(name ?c10))

(attribute(name ?a20)(class ?c10)(unique ?a22))
(class(name ?c10))
=>
(assert (fk(column ?a20)(table ?c00)(fktable ?c10))))
```

La partie droite d'une règle D contient les éléments cibles à créer si la règle se déclenche. Similairement à G_{kt_i}, D_{kt_i} ne peut pas contenir des éléments cibles de type $t' \in MMC$ si t' n'apparaît pas dans les fragments cibles de TR_{kt}, l'ensemble des traces de type t dans le *pool* k. Les attributs des éléments cibles peuvent être initialisés par trois types de terminaux : des variables de la partie gauche G, des constantes définies par des types énumération de MMC, ou des fonctions qui prennent en argument les deux types de terminaux précédents. Lors de l'établissement des liaisons d'attributs cibles, seuls les terminaux d'un type compatible sont considérés.

Concernant la liaison d'attributs cibles du type chaîne de caractères, un vecteur de similarité lexicale est construit en analysant les traces du *pool* sur lequel s'exécute l'algorithme. Ce vecteur permet de réduire le nombre de terminaux candidats et d'introduire plus efficacement les fonctions de chaînes. En effet, lorsque le vecteur de similarité lexical d'un attribut cible ne révèle aucune correspondance parfaite parmi les attributs sources et les constantes, l'ensemble des candidats de liaison pour cet attribut est réduit aux fonctions qui retournent une chaîne de caractères.

3.4.2.2 Génération de la population initiale

La dérivation d'un programme de transformation en utilisant la PG requiert la création d'une population initiale. La taille de la population initiale est déterminée par le paramètre *generation_size* fourni au démarrage de l'algorithme.

Lors de la création des individus, chaque programme de la population initiale possède autant de groupes de règles qu'il y a de types $t \in MMS$ (toutes les traces doivent être transformées). Le nombre de règles créées pour chaque groupe est choisi aléatoirement dans un intervalle fourni en paramètre. Chaque règle est ensuite créée en générant un nombre aléatoire de briques pour sa partie gauche G ainsi qu'un nombre, aléatoire également, d'éléments cibles dans sa partie droite D.

Tous les éléments sources et cibles d'une règle sont sélectionnés parmi les éléments présents dans le sous-ensemble de traces transformées par le groupe dont la règle fait partie. À la suite de cela, chaque attribut cible est lié à un terminal compatible, choisi aléatoirement parmi les trois types de terminaux possibles (variables de la partie G, constantes

et fonctions). L'algorithme de génération garantit évidemment que les règles générées soient syntaxiquement (par rapport à JESS) et sémantiquement (par rapport aux deux méta-modèles) correctes.

3.4.2.3 Évaluation des transformations candidates

Chaque programme candidat de transformation PT_{ki}, où $i \in \{1..generation\_size\}$, est exécuté avec MS_k en entrée. La fitness de PT_{ki} est ensuite évaluée en comparant le modèle MC_{ki} qu'il a produit avec le modèle escompté MC_k. La comparaison des deux modèles est pondérée par type d'élément cible produit afin d'éviter que l'évaluation soit biaisée vers des éléments fréquents d'un type en particulier. En posant T_{MC_k} comme étant l'ensemble des types définis par le méta-modèle cible, la fonction fitness est donc définie par la somme pondérée des résultats obtenus par la comparaison d'éléments de type $t \in MMC$ (équation 3.1).

$$f_i(MC_k, MC_{ki}) = \sum_{t \in T_{MC_k}} \frac{f_i^t(MC_k, MC_{ki})}{\left|T_{MC_k}\right|} \tag{3.1}$$

$$f_i^t(MC_k, MC_{ki}) = \alpha ct^t + \beta cp^t, \textit{ avec } \alpha + \beta = 1 \tag{3.2}$$

La comparaison entre les modèles MC_k et MC_{ki} pour un type $t \in MMC$ (équation 3.2) est effectuée en identifiant, en premier, les éléments qui correspondent totalement à ceux attendus. Deux éléments sont en correspondance totale, si tous leurs attributs et références sont identiques. Pour les éléments restants, des correspondances partielles sont identifiées parmi les éléments du même type qui n'ont pas été sélectionnés durant l'étape précédente. $f^t(MC_k, MC_{ki})$ est ensuite calculée selon l'équation 3.2, où ct^t (resp. cp^t), désigne le nombre de correspondances totales (resp. partielles).

Comme mentionné auparavant, un programme de transformation est un ensemble de groupes de règles, où chaque groupe GR_k^t contient des règles qui transforment des traces de type $t \in MMS$. Une évaluation alternative à celle que nous venons de décrire est d'évaluer chaque groupe GR_k^t sur des traces de son type, plutôt que d'évaluer l'ensemble des groupes sur le modèle source MS_k. Cependant, compte tenu du fait que nous ne re-

quérons pas que les traces soient précises, adopter une telle méthode d'évaluation amène le programme génétique à favoriser des transformations dont plusieurs règles créent les mêmes éléments plusieurs fois (les éléments qui existent dans plusieurs types de traces).

Par exemple, le fragment cible d'une trace de type *association* peut contenir deux tables ainsi que leurs clés étrangères respectives, évaluer les règles du groupe *association* sur les traces correspondantes conduira ces règles à évoluer pour produire non seulement la clé étrangère, mais aussi les éléments *table* impliqués, alors que ces éléments *table* seront probablement produits également par le groupe de règles responsable de la transformation du type *class*.

3.4.2.4 Dérivation de nouvelles transformations

Après l'évaluation de chaque programme candidat de la génération courante, une nouvelle génération de transformations est dérivée. D'abord, un opérateur de reproduction (élitisme) est utilisé pour insérer les x meilleurs individus dans la nouvelle génération. Ensuite, et jusqu'à ce que la nouvelle population soit complète, deux individus sont choisis à partir de la génération précédente en utilisant un tournoi de sélection binaire (voir paragraphe 3.4.1.6). Les individus peuvent alors être le sujet d'une opération de croisement avec une certaine probabilité. Les deux candidats enfants produits, qu'ils soient le résultat d'un croisement des parents ou des copies de ceux-ci, peuvent être à leur tour mutés avec une certaine probabilité, avant de rejoindre la nouvelle population. Ce processus de dérivation varie de la méthode classique décrite dans la figure 3.4 dans la mesure où les opérateurs de croisement et de mutation ne sont pas mutuellement exclusifs, mais sont plutôt appliqués successivement avec une certaine probabilité. Cette approche de dérivation s'est avérée efficace lors de contributions antérieures [5, 24]

- **Croisement** : l'opération de croisement que nous avons définie opère au niveau des groupes de règles avec une stratégie de croisement à un seul point. Elle consiste à produire deux programmes de transformation à partir de deux programmes existants en sélectionnant un point de croisement et en échangeant, dans les deux programmes enfants, les groupes de règles se trouvant après ce point.

Par exemple, considérons les deux transformations candidates PT_{k1}=$\{GR_{k1}^{t_1}, GR_{k1}^{t_2}, GR_{k1}^{t_3}, GR_{k1}^{t_4}\}$ et PT_{k2}=$\{GR_{k2}^{t_1}, GR_{k2}^{t_2}, GR_{k2}^{t_3}, GR_{k2}^{t_4}\}$. Étant donné que chaque enfant produit par le croisement se doit de maintenir ses groupes (chaque programme enfant doit transformer tous les types de traces), le même point de croisement est utilisé pour les deux parents. Supposons que le point 3 est choisi, les enfants obtenus sont alors les candidats : E_{k1}=$\{GR_{k1}^{t_1}, GR_{k1}^{t_2}, GR_{k1}^{t_3}, GR_{k2}^{t_4}\}$ et E_{k2}=$\{GR_{k2}^{t_1}, GR_{k2}^{t_2}, GR_{k2}^{t_3}, GR_{k1}^{t_4}\}$.

- **Mutation** : chaque programme candidat peut faire l'objet d'une mutation aléatoire avec une certaine probabilité. Les mutations que nous avons définies opèrent à deux niveaux distincts.

  - Au niveau du groupe : ces mutations permettent d'ajouter ou de supprimer une règle d'un groupe choisi aléatoirement. Ces deux mutations sont conditionnées, elles doivent prévenir l'apparition de groupes vides ainsi que le dépassement du nombre maximal de règles par groupe.

  - Au niveau de la règle : ce niveau regroupe au total 7 mutations possibles. Trois mutations concernent la partie *G* de la règle. Elles consistent respectivement en : l'ajout d'une nouvelle brique, la suppression d'une brique existante, ou la création d'une nouvelle partie gauche. Par exemple, la règle présentée dans le listing 3.2 subit une mutation dans sa partie *G* dans laquelle une nouvelle brique est ajoutée. La partie *D* de la règle peut, quant à elle, être l'objet de 4 mutations différentes qui sont : l'ajout d'un élément cible, la suppression d'un élément cible existant, la recréation de toute la partie droite, ou, la réinitialisation des attributs cibles par de nouveaux terminaux. Un exemple de cette dernière mutation est présenté dans le listing 3.3.

    Il est à noter également que les changements ayant lieu au niveau de la partie *G* doivent être propagés à la partie *D* pour éviter, par exemple, l'apparition de règles ayant des parties *D* qui font référence à des variables qui n'existent plus dans leurs parties *G*.

Listing 3.2 – Exemple d'une mutation gauche

```
(defrule R347261 //Regle avant la mutation
(associationnm (name ?s00)(classfr ?c00)(classto ?c10)(classas ?s03))
(class(name ?c00))
(class(name ?c10))
=>
(assert (table (name ?s00)(alternname nil))))


(defrule R347261 //Regle apres la mutation
(associationnm (name ?s00)(classfr ?c00)(classto ?c10)(classas ?s03))
(class(name ?c00))
(class(name ?c10))

(test (eq ?s03 nil))
=>
(assert (table (name ?s00)(alternname nil))))
```

Les mutations au niveau du programme de transformation en tant que tel (ajout et suppression de groupes) n'ont pas été implantées, car chaque programme de transformations doit transformer tous les types du modèle source et seulement eux (c.-à-d. qu'il doit, à tout moment, comporter autant de groupes de règles qu'il y a de types de traces à transformer).

Listing 3.3 – Exemple d'une mutation droite

```
(defrule R438783 //Regle avant la mutation
(association1n (classfr ?c00) (classto ?c10))
(class(name ?c00))
(class(name ?c10))

(column (name ?o20)(table ?c00)(pk 1))
(table (name ?c00)(alternname ?t21))
=>
```

```
(assert (fk (column ?o20) (table ?c00) (fktable ?c10))))

(defrule R438783 //Regle apres la mutation
(association1n (classfr ?c00) (classto ?c10))
(class(name ?c00))
(class(name ?c10))

(column (name ?o20)(table ?c00)(pk 1))
(table (name ?c00)(alternname ?t21))
=>
(assert (fk (column ?o20) (table ?c10) (fktable ?c00))))
```

3.4.2.5 Programmation génétique adaptative

Beaucoup d'algorithmes évolutionnaires font l'objet d'une convergence prématurée [3, 73]. Ceci arrive lorsque des parents, proches d'une zone d'optimum local, engendrent plusieurs progénitures similaires. Dans les algorithmes génétiques, ce problème est généralement contourné par l'augmentation de la puissance exploratoire de l'algorithme (par exemple, augmenter la taille de la population). Cependant, ceci a pour effet de produire une population qui atteint l'optimum global très lentement.

Beaucoup de contributions ont été proposées pour résoudre ce problème. Dans ce travail, nous tentons d'exploiter certains de ces travaux afin d'éviter de converger vers des programmes de transformations sous-optimaux, tout en améliorant les performances du programme génétique que nous utilisons.

Dans notre approche, nous considérons la mutation comme étant le mécanisme principal permettant d'explorer l'espace de recherche. Le croisement a toujours été l'opérateur le plus populaire dans les AG et la PG, principalement suite à l'hypothèse des blocs constructeurs (*building-block hypothesis [32]*). Cependant, plusieurs travaux de recherche mettent en doute la supériorité du croisement par rapport à la mutation dans le cas de la programmation génétique [54]. De plus, étant donné que l'évaluation de la population est une opération coûteuse dans notre cas (comparaison de modèles), notre

approche est fondée sur des populations de tailles relativement réduites pour lesquelles la mutation semble obtenir de meilleurs résultats que le croisement [54, 66].

Mutations sélectives

Yang et Uyar proposent dans [83] un mécanisme de mutation sélectif. Dans leur contribution, la probabilité de mutation de chaque locus de gènes est corrélée aux statistiques de la fitness et à la distribution de ces derniers. Dans notre approche, nous utilisons un mécanisme similaire. Une des principales difficultés rencontrées lors de l'implémentation d'une telle approche réside dans la difficulté d'évaluer les parties d'un programme individuellement. Pour notre part, nous tirons profit de notre capacité à évaluer chaque groupe de règles du programme sur les traces qu'il transforme.

Lorsque l'exécution d'un programme génétique produit une même valeur maximale de fitness durant plusieurs générations, le programme bascule de l'utilisation des mutations définies dans le paragraphe 3.4.2.4 vers des mutations sélectives. Durant les itérations à mutations sélectives, chaque programme candidat de transformation PT_{ki} fait l'objet de deux évaluations distinctes. Il est évalué d'une part sur l'exemple du *pool* (MS_k, MC_k) tel que décrit dans le paragraphe 3.4.2.3, de plus, chaque groupe de règles GR^t_{ki} est évalué sur l'ensemble des traces de type $t \in MMS$ qu'il est censé transformer.

Durant chaque mutation, un tournoi binaire est organisé pour élire le groupe qui fera l'objet d'une mutation. Deux groupes sont d'abord choisis aléatoirement, puis le groupe qui possède la fitness la plus basse gagne le tournoi. L'objectif de cette stratégie est de diriger l'exploration de l'espace de recherche vers les éléments sources qui ne sont pas transformés correctement afin d'assurer une convergence plus rapide vers un optimum global.

Les mutations sélectives ne sont pas utilisées dans les générations initiales afin d'éviter la question abordée dans le paragraphe 3.4.2.3, c'est à dire, que des groupes de règles génèrent des éléments cibles qu'elles ne sont pas supposées produire dû à la présence de traces imprécises (des traces comportant des éléments cibles additionnels).

Concernant le critère de début des mutations sélectives, nous avons opté pour un critère qui dépend de la progression de la recherche (les mutations sélectives sont en-

clenchées lorsque l'algorithme stagne) afin de ne pas avoir à ajuster un paramètre en fonction de chaque problème de transformation sur lequel l'approche est appliquée.

Mémoire collective

De nombreuses contributions dans le domaine de la PG se sont consacrées aux moyens qui permettraient de protéger les parties souhaitables d'une solution dans le génotype : *Automatically Defined Functions (ADF)* [44], *Module Acquisition (MA)* [4], *Adaptive Representation through Learning (ARL)* [60], et *Collective Memory* [8]. Ces mécanismes ont pour but d'assurer une convergence plus rapide en protégeant le matériel génétique intéressant des éventuelles perturbations causées par les opérations de croisement et de mutation.

Dans ce travail, nous nous inspirons de certains de ces travaux pour mettre en place un mécanisme de protection des segments de transformation considérés comme étant intéressants. Certaines règles apprises durant les premières générations sont stockées pour former une mémoire collective. Cette mémoire sera utilisée dans les générations subséquentes pour réintroduire les règles stockées. Encore une fois, un problème commun aux contributions mentionnées plus haut réside dans la difficulté d'évaluer des composantes de programmes d'une manière isolée. Nous pallions cela en effectuant une seconde évaluation des meilleurs individus. Ainsi, suite au calcul de fitness décrit dans le paragraphe 3.4.2.3, les meilleurs individus font l'objet d'une seconde évaluation durant laquelle on apprécie la qualité de chaque règle individuellement.

Afin d'implémenter ce mécanisme, nous enrichissons graduellement un dépôt de règles pendant les générations durant lesquelles le meilleur score de fitness augmente. Nous évaluons d'abord chaque groupe de règles $GR_k^t t \in MMS$ du meilleur individu sur l'ensemble des traces de son type TR_k^t. Un groupe est sélectionné s'il atteint un meilleur score que celui enregistré dans les générations antérieures. Si un groupe est sélectionné, chaque règle de ce groupe est évaluée individuellement sur le même ensemble de traces. Si la règle produit une correspondance totale, elle est insérée dans la mémoire collective.

À l'instar des programmes de transformation, les règles sont groupées dans la mémoire collective en fonction du type de fragment qu'elles transforment. Lorsque les mu-

tations sélectives sont lancées, un nouvel opérateur de mutation est ajouté aux deux mutations déjà disponibles au niveau des groupes. Cette mutation sélectionne la règle R_i^t, $t \in MMS$ du dépôt de mémoire collective et l'ajoute au groupe de règles GR_k^t qui transforme les traces de type t dans le programme de transformation candidat.

3.4.3 Post-traitement des programmes de transformation

Arrivé à cette étape, un programme de transformation est dérivé pour chaque *pool* par une exécution du PG. Avant de fusionner toutes les transformations dans un seul programme, nous procédons à une étape de nettoyage des transformations telles que dérivées.

D'abord, les règles qui produisent plus d'un élément cible sont divisées en plusieurs règles générant un seul élément cible chacune. Cette première opération vise à s'assurer que chaque élément cible est produit quand des conditions spécifiques sont remplies. Ces conditions seront apprises durant la troisième et dernière phase du processus d'apprentissage.

Les règles sont ensuite évaluées, dans leurs *pools*, sur leurs paires de modèles respectifs, afin de supprimer celles qui ne produisent pas de correspondances avec les éléments cibles attendus. Les règles restantes sont analysées pour effacer d'éventuelles conditions dupliquées. L'étape suivante consiste à supprimer les règles dupliquées. Celles-ci sont identifiées en analysant leurs parties D et G respectives. Enfin, les règles qui produisent les mêmes éléments sont analysées afin de vérifier si les conditions de la partie G d'une règle sont subsumées par celles d'une autre. À la fin de chaque opération de traitement, le programme de transformation est évalué sur la paire d'exemples du *pool* concerné afin de s'assurer que la valeur de sa fitness n'a pas décru.

Après avoir nettoyé et corrigé chaque programme, les règles de toutes les transformations produites (il y a autant de transformations que de *pools*) sont fusionnées dans un seul programme de transformation. Les règles qui appartiennent à des groupes conflictuels (c.-à-d. des groupes transformant des traces conflictuelles) sont toutes copiées dans le programme final. Les groupes non conflictuels sont, quant à eux, évalués un-à-un sur la paire d'exemples initiale. Les règles appartenant au groupe ayant réalisé le meilleur

score sont insérées dans le programme final.

Après avoir construit le programme final, ce dernier fait l'objet des mêmes traitements effectués sur le programme de transformation produit au niveau de chaque *pool*.

3.5 Raffinement des règles dérivées

La phase de raffinement a lieu uniquement lorsqu'une transformation de modèle a été apprise à travers plusieurs *pools*. Suite à la fusion de programmes, nous nous attendons à des règles de transformation possédant des parties *G* incomplètes, compte tenu du fait que les conditions censées distinguer les traces conflictuelles n'ont pas été apprises encore.

Dans le cas de *pools* multiples, le programme produit par la 2e phase possède une faible précision (c.-à-d. qu'il génère des éléments qui ne sont pas attendus) lorsqu'il est appliqué sur la paire de modèles initiale. L'objectif de cette phase est donc de raffiner et de compléter les parties *G* des règles. Nous utilisons à cette fin l'algorithme du recuit simulé.

3.5.1 Recuit simulé

Le recuit simulé (RS) est une approche heuristique, générique, utilisée pour les problèmes d'optimisation globale. L'approche s'inspire du domaine de la thermodynamique statistique [1].

Comme décrit par l'algorithme 1, le RS commence par une solution initiale de laquelle il se déplace vers une solution candidate voisine. Si la solution voisine est meilleure que la solution courante, elle est sélectionnée comme la solution courante. Sinon, elle peut comme même être sélectionnée avec une probabilité proportionnelle à la température du système et inversement proportionnelle à la dégradation subit (la différence entre la qualité de la solution courante et celle de la solution candidate).

Le RS se distingue des autres méthodes de recherche locale par la possibilité d'accepter des déplacements qui n'améliorent pas le résultat obtenu par la solution courante. Cette caractéristique permet à l'algorithme de s'échapper des optimums locaux. Afin

de garantir une amélioration globale des solutions trouvées, l'acceptation des solutions moins bonnes se fait, d'une manière contrôlée appelée le programme de recuit. Le programme de recuit réduit la probabilité d'accepter des solutions moins bonnes au fur et à mesure que la recherche progresse. Il est inspiré par la façon dont est réduite l'énergie thermodynamique d'une substance lors d'un refroidissement contrôlé.

Il est à noter que la méthode du RS garantit statiquement qu'une solution optimale est atteinte dans le cas où le programme de recuit est logarithmique [33].

Algorithm 1 Algorithme du recuit simulé

1: $PT_c \leftarrow \widetilde{PT}$
2: $PT_{meilleur} \leftarrow \widetilde{PT}$
3: $Temp \leftarrow Temp_0$
4: **while** $arret = faux$ **do** ▷ *arret* devient *vrai* lorsque le système est gelé
5: **for** $i = 0, Nb_{rep}$ **do**
6: $s_c = Voisinage(PT_c)$
7: $\sigma \leftarrow f(s_c) - f(PT_c)$
8: **if** $\sigma > 0$ **then**
9: $PT_c \leftarrow s_c$
10: **if** $f(PT_c) > f(PT_{meilleur})$ **then**
11: $PT_{meilleur} \leftarrow PT_c$
12: **end if**
13: **else**
14: $u \leftarrow U([0,1])$
15: **if** $u < \exp(\sigma/Temp)$ **then**
16: $PT_c \leftarrow s_c$
17: **end if**
18: **end if**
19: **end for**
20: $Temp \leftarrow Temp * \alpha$
21: **end while**

3.5.2 Raffinement de la transformation en utilisant RS

Soit un programme de transformation $\widetilde{TP}$, nous définissons la phase de raffinement comme étant un problème d'optimisation combinatoire (S, f) où S est l'espace fini de toutes les solutions possibles et f est la fonction fitness définie dans le para-

graphe 3.4.2.3. $f : S \rightarrow [0,1]$, pour $s \in S$. Notre objectif est de trouver $TP = s*$ tel que $f(s*) > f(s), \forall s \in S$.

3.5.2.1 Représentation et génération des solutions

À chaque itération une nouvelle solution candidate s_c est obtenue,en mutant la solution courante TP_c. Contrairement aux conditions initiales qui composent la partie G de chaque règle, les conditions supplémentaires apprises durant cette phase sont combinées en utilisant des opérateurs logiques *AND* et *OR*. Ces conditions sont donc organisées sous la forme d'un arbre binaire qui s'ajoute à la partie G initiale de la règle.

Étant donné que la dérivation d'une solution voisine n'inclut pas l'ajout, la duplication ou la suppression de règles, nous introduisons l'opérateur *OR* afin de prendre en charge les règles qui devraient se déclencher dans plusieurs situations différentes.

Un opérateur de mutation est utilisé pour dériver à chaque itération une solution voisine. Une règle de la solution courante est d'abord choisie, la mutation ajoute, modifie ou supprime une condition de l'arbre binaire de sa partie G. Les conditions qui composent cet arbre sont plus expressives que les conditions initiales. En plus des patrons sources, ces conditions peuvent exprimer des patrons cibles, des négations, la vérification des valeurs d'attributs et des références d'éléments sources, ainsi que l'utilisation de primitives de navigation qui explorent le modèle source.

3.5.2.2 Évaluation des solutions

Une solution candidate s_c est évaluée sur la paire d'exemples initiale (MS, MC) en utilisant la fonction fitness définie dans le paragraphe 3.4.2.3. La solution courante PT_c est remplacée par s_c si cette dernière possède une valeur de fitness supérieure ou si la condition exprimée dans la ligne 15 de l'algorithme 1 est satisfaite. La meilleure solution trouvée depuis le début de l'exécution de l'algorithme est sauvegardée dans $PT_{meilleur}$.

3.5.2.3 Programme du recuit

Le programme du recuit possède quatre paramètres qui doivent être définis : la température initiale, la fonction de décroissance de la température, le nombre d'itérations à chaque température, et enfin, le critère d'arrêt. Dans le cadre de notre approche, la température initiale est fixée à 1.0. Cette température est juste assez élevée pour permettre à l'algorithme de se déplacer vers n'importe quelle solution voisine, sans donner lieu, pour autant, à une recherche aléatoire.

Concernant la fonction de décroissance de la température, nous utilisons la fonction géométrique $Temp_t = Temp_0 * \alpha^t$, qui est très répandue dans la littérature, avec α entre 0.9 et 0.99. À chaque température, nous avons expérimenté avec plusieurs nombres d'itérations Nb_{reps}, allant de quelques dizaines à plusieurs milliers, afin d'identifier le meilleur compromis entre la fonction de décroissance de la température et le nombre d'itérations à chaque température.

L'algorithme de RS se termine lorsque le système est gelé, c'est à dire, lorsqu'aucune solution candidate n'est acceptée pendant plusieurs valeurs de température successives.

CHAPITRE 4

VALIDATION

Nous avons évalué notre approche sur 7 problèmes de transformation. L'objectif de notre validation est de répondre aux questions de recherche suivantes :

QR1 : Jusqu'à quel point, les modèles cibles produits par les règles résultantes du processus d'apprentissage sont-ils comparables à ceux attendus ?

QR2 : Les programmes de transformation dérivés sont-ils corrects ?

QR3 : Quels sont les gains obtenus par l'utilisation de techniques adaptatives dans la programmation génétique ?

Nous détaillerons dans un premier temps le cadre des expériences que nous avons conduites, en abordant notamment, les problèmes de transformation sélectionnés, le jeu de données utilisé, les paramètres des algorithmes de recherche, et enfin, notre méthode de validation. Dans la section 2, nous présenterons les résultats obtenus par l'approche et répondrons aux trois questions de recherche formulées. La section 3 permettra d'identifier les menaces qui pourraient affecter la validité de notre étude. Enfin, nous discuterons dans la 4e et dernière section les différentes observations que nous avons pu effectuer lors de la validation du processus d'apprentissage proposé dans ce mémoire.

4.1 Cadre d'expérimentation

4.1.1 Problèmes de transformation

Nous avons sélectionné 7 transformations de modèles exogènes. Chaque problème de transformation met en évidence diverses exigences. Excepté la transformation SPEM-à-BPMN [18], toutes les transformations ont été choisies de la base de données ATL [1]. Nous décrivons brièvement ci-dessous chacune de ces transformations.

[1] http ://www.eclipse.org/atl/atlTransformations/

Diagramme de classes UML vers schéma relationnel (DC-à-SR) : nous utilisons cette transformation de modèles afin d'évaluer la capacité de notre approche à faire face à des transformations structurelles complexes. La spécification que nous avons considérée pour ce cas inclut la transformation d'associations qui possèdent, ou non, des classes d'association, ainsi que, la fusion pour les associations de type $1-1$ lorsque cela est possible. Cette dernière exigence requiert l'exploration du contexte dans le modèle source, du fait que la fusion de classes ne devrait avoir lieu que lorsqu'une des deux classes n'est impliquée dans aucune autre association ou relation d'héritage. La spécification de cette transformation requiert également la dérivation d'attributs cibles complexes, car lesdites fusions nécessitent de concaténer les noms des classes impliquées dans la table résultante. Une description plus détaillée de cette transformation peut être trouvée dans [38].

***Domain Specific Language* vers *Kernel Metamodel* (DSL-à-KM3) :** cette transformation a été sélectionnée à partir de la chaîne de transformations proposée dans le cadre d'un pont entre le langage dédié de Microsoft *Domain Specific Language (DSL)* et le framework d'Eclipse *Eclipse Modeling Framework (EMF)*. À l'instar du cas précédent, ce problème comporte des transformations structurelles intéressantes. Les méta-modèles des deux domaines (DSL et KM3) sont assez similaires. Cependant, KM3 supporte l'héritage multiple alors que ce n'est pas le cas pour DSL. De plus, les relations de DSL peuvent posséder des attributs et des super-types, alors que ceci n'est pas vrai pour KM3. En conséquence, les relations DSL simples sont transformées en des paires de références au niveau de KM3, alors que celles qui sont complexes correspondront à des classes du modèle cible.

***Ant* vers *Maven* (Ant-à-Maven) :** cette transformation permet de spécifier comment générer un fichier *Maven* à partir d'un fichier *Ant*. La transformation est relativement simple avec plusieurs correspondances de type un-à-un. Nous avons sélectionné ce cas pour deux raisons principales. D'abord, les méta-modèles sources et cibles sont grands, ce qui donne lieu à une transformation assez importante (la version que nous avons écrite contient 26 règles). Ensuite, la plupart des éléments des deux modèles possèdent beaucoup d'attributs. Cette particularité nous permet d'évaluer les performances de notre approche face à la prolifération d'attributs, une caractéristique qui a entravé le succès

d'une contribution antérieure [5].

***Software Process Engineering Metamodel* vers *Business Process Managment Notation* (SPEM-à-BPMN) :** nous considérons pour cette transformation la spécification décrite dans [18]. L'objectif de cette transformation est d'automatiser la gestion des activités de développement logiciel, en passant d'activités modélisées avec *SPEM* en des sous-processus *BPMN*, qui peuvent ensuite être traduits vers un langage executable tel que *BPEL* [2]. Notre choix a porté sur ce problème de transformation, car plusieurs éléments sources du même type peuvent être transformés en différents types d'éléments cibles, dépendamment de la valeur de certains de leurs attributs ou références.

***Microsoft Excel* vers *Extensible Markup Language* (Excel-à-XML) :** cette transformation a pour but de générer un cahier *Excel* (un fichier XML bien formé) à partir d'un modèle *Excel* qui est conforme au méta-modèle *Simplified Spreadsheet-ML*. Notre objectif est d'apprendre uniquement la première partie de la spécification de ce problème étant donné qu'elle englobe, à elle seule, toute la logique de la transformation. La seconde partie de la spécification consiste simplement en une extraction XML (la sérialisation du modèle *XML* en un fichier *XML* valide). Il est intéressant d'appliquer notre approche sur cette transformation, car plusieurs attributs cibles prennent comme valeurs des constantes plutôt que les valeurs d'attributs sources.

Tableau vers Diagramme en barres *SVG* (TB-à-DB) : l'objectif de cette transformation est de générer un diagramme en barres, exprimé dans un modèle *Scalable Vector Graphics (SVG)* en fonction des données présentes dans une table fournie en entrée. Le modèle *SVG* inclut des éléments de type graphe, rectangle et texte. Ces éléments possèdent, entre autres, des attributs qui expriment leurs dimensions ou leurs coordonnées. La spécification de cette transformation requiert l'usage d'opérations arithmétiques et de fonctions afin de calculer ces attributs lors de la génération du modèle *SVG* cible.

Modèle *UML* vers *AMBLE* (UML-à-Amble) : ce cas décrit une transformation à partir d'un ensemble de modèles *UML* interreliés, qui expriment différents aspects d'un programme distribué, vers une implémentation de ce dernier dans le langage Amble[3].

[2] Business Process Execution Language

[3] Un langage de programmation basé sur OCaml

Les modèles *UML* consistent en plusieurs diagrammes d'états. Chaque diagramme représente un type de processus utilisé dans le programme Amble. L'ensemble des modèles sources en entrée inclut également un diagramme de classes qui décrit les différents processus impliqués (en tant que classes), ainsi que les canaux de communication qui existent entre eux (en tant qu'associations). La spécification de cette transformation requiert l'exploration du contexte dans le modèle source, la vérification de valeurs d'attributs et de références sources, ainsi que la dérivation de certains attributs cibles complexes.

4.1.2 Jeu de données

Comme mentionné dans le paragraphe 2.1.3.5, nous avons utilisé lors de notre validation Jess [27]. Les 14 méta-modèles ont donc été représentés par des gabarits de faits, alors que les modèles sont représentés par des faits. Après avoir défini tous les méta-modèles dans Jess, nous avons conçu deux paires de modèles sources et cibles pour chaque problème de transformation. Nous avons également écrit les programmes de transformation associés à chaque cas de transformation pour nous servir de référence lors de l'évaluation des transformations apprises. Nous avons testé chaque programme sur les deux paires de modèles en inspectant le modèle cible produit afin de nous assurer que les programmes sont corrects vis-à-vis de la spécification définie. Le tableau 4.I donne le nombre d'éléments de chaque modèle des paires d'exemples ainsi que la taille des programmes écrits.

Durant la validation de notre approche sur chaque problème de transformation, nous avons utilisé la première paire d'exemples afin d'apprendre le programme recherché et la seconde pour l'évaluer. Pour chaque cas, nous avons construit les traces associées uniquement à la paire source-cible d'apprentissage. Comme mentionné dans la section 3.3, nous avons tâché de simuler un scénario d'utilisation réel en évitant de fournir des traces précises, où chaque fragment source, est accompagné d'un fragment cible qui comporte seulement les éléments qui seraient produits par l'application des règles de transformation correspondantes.

Tableau 4.I – Taille des modèles utilisés lors de l'expérimentation

Transformation	Taille des modèles d'apprentissage		Taille des modèles de validation		Programme écrit
	Source	Cible	Source	Cible	Nombre de règles
DC-à-SR	30	42	42	59	20
DSL-à-KM3	31	38	46	60	9
Ant-à-Maven	26	29	33	36	26
SPEM-à-BPMN	14	14	22	23	9
Excel-à-XML	18	22	31	47	10
TB-à-DB	13	42	17	54	12
UML-à-Amble	41	33	62	66	14

4.1.3 Paramétrage des algorithmes

Le processus d'apprentissage a été lancé une fois pour chaque problème de transformation en ayant comme entrée la paire d'apprentissage agrémentée par ses traces. Pour chaque cas de transformation, nous avons évidemment autant d'exécution du programme génétique que de *pools* produits par la première phase. Les exécutions du PG ont toutes consisté en 200 générations de 100 programmes. Les probabilités de croisement et de mutation ont toutes les deux été fixées à 0.9. Dans chaque génération, le groupe de solutions à être reproduites se compose des 5 meilleurs programmes de transformation. Concernant l'algorithme du recuit simulé, nous avons opté pour une décroissance lente de température, jumelée avec un petit nombre d'itérations. Le paramètre α du programme de recuit a été fixé à 0.99 avec seulement 10 itérations à chaque valeur de température. Tel que mentionné dans le paragraphe 3.5.2.3, l'exécution de l'algorithme s'arrête lorsque le système est gelé.

Nous avons fourni aux deux algorithmes de recherche des fonctions et des constantes spécifiques aux domaines source et cible. Afin d'éviter l'injection de connaissances à propos de la transformation, plusieurs fonctions ont été ajoutées même si elles ne sont pas utilisées par la transformation. Par exemple, le cas TB-à-DB nécessite la dérivation du nombre total de lignes ainsi que la valeur maximale des cellules. Nous avons alors fourni aux algorithmes des fonctions, telles que *sum, min, max, count, etc.*, qui peuvent être appliquées à n'importe quel attribut numérique. Outre les terminaux spécifiques au

domaine, nous avons également ajouté, dans les 7 problèmes traités, des opérateurs de chaînes et d'entiers (concaténation de chaînes, addition, soustraction, etc.).

Le processus d'apprentissage, incluant l'étape de post-traitement menée à la fin de la 2e phase, est totalement automatisé et ne requiert aucune intervention humaine.

4.1.4 Méthode de validation

Pour répondre à la première question de recherche **QR1**, nous évaluons chaque programme de transformation dérivé en comparant le modèle produit au modèle cible attendu. La comparaison est effectuée automatiquement en utilisant les mesures de précision et de rappel.

Nous définissons le rappel comme étant le ratio entre le nombre d'éléments attendus qui sont produits et le nombre total d'éléments attendus. La précision est définie par le ratio entre le nombre d'éléments attendus qui sont produits et le nombre total d'éléments produits. Pour EP et EA, désignant respectivement l'ensemble d'éléments produits et l'ensemble d'éléments attendus, les deux métriques sont calculées de la manière suivante :

$$Precision_{MC} = \frac{|EA \cap EP|}{|EP|}, Rappel_{MC} = \frac{|EA \cap EP|}{|EA|} \tag{4.1}$$

Il est à noter qu'un élément attendu est considéré comme produit, si ce dernier correspond exactement à l'élément attendu. Il doit posséder le même nom, ainsi que les mêmes attributs et références. Contrairement à la fonction fitness définie, les mesures utilisées excluent les correspondances partielles. Un élément est donc considéré comme étant incorrect même si seulement un de ses attributs est incorrectement dérivé.

Nous répondons à la seconde question de recherche **QR2** de deux façons. D'abord, nous exécutons les transformations dérivées sur la seconde paire d'exemples (paire non vue par l'algorithme) et évaluons la précision et le rappel des modèles cibles produits par chaque transformation. En second lieu, nous inspectons les programmes de transformation. Au niveau de chaque programme, nous vérifions que les éléments cibles sont produits par les groupes de règles censés les produire. Au niveau des groupes de règles,

nous évaluons d'une part, la complétude et la correction des conditions qui doivent être satisfaites au niveau de chaque règle (inspection des parties *G*) et le nombre d'attributs cibles correctement dérivés par la règle (inspection des parties *D*).

Concernant la troisième et dernière question de recherche **QR3**, nous exécutons le programme génétique à plusieurs reprises, avec et sans ses mécanismes adaptatifs (mutation sélective et mémoire collective), afin d'apprécier la profitabilité des stratégies de mutation proposées.

4.2 Présentation des résultats

Le tableau 4.II montre les résultats réalisés sur la paire d'apprentissage à chaque étape du processus. Quatre des sept problèmes sélectionnés contiennent des traces conflictuelles et ont donc été traités en utilisant plusieurs *pools*. Pour ces cas-là, la fitness obtenue dans chaque *pool* est assez élevée ; elle varie entre 93% et 95%. Lorsque les programmes de transformation issus de chaque *pool* sont fusionnés et traités ($\widetilde{TP}$), un rappel parfait est atteint dans 3 des 4 cas. Cependant, comme prévu, la précision de chaque programme a chuté à suite de cette fusion, ce qui a entrainé une fitness relativement basse allant de 70% à 80% pour les trois cas DC-à-SR, DSL-à-KM3 et UML-à-Amble. Après le déroulement de la phase finale (le recuit simulé), la qualité de chaque transformation est considérablement améliorée avec une fitness supérieure à 95% observée sur les quatre cas de transformation possédant des traces conflictuelles.

Concernant les transformations apprises dans un seul *pool*, notre approche a produit un score parfait pour deux des trois cas, à savoir, Ant-à-Maven et Excel-à-XML. Pour le cas TB-à-DB, bien que la précision du modèle produit fut de 100%, seulement 90% des éléments attendus ont été produits correctement. Il est à noter que pour ces trois cas de transformation, la phase de raffinement de la transformation (la 3e phase) n'a pas eu lieu étant donné que toutes les règles ont été apprises à partir d'un seul *pool*. Le programme final de chacun de ces cas correspond donc à la version post-traitée de la transformation produite par le programme génétique.

Tableau 4.II – Fitness, rappel et précision à chaque étape du processus d'apprentissage

Transformation	Nb. *pools*	PT_k (Moyenne)			$\widetilde{PT}$			PT			
		Fitness	Rappel	Précision	Fitness	Rappel	Précision	Fitness	Rappel	Précision	Nb. Règles
DC-à-SR	2	95.2%	100%	92.6%	82.13%	100%	69.9%	97.6%	98.4%	96.9%	22
DSL-à-KM3	2	98.3	100%	96.9%	73.5%	100%	64.9%	100%	100%	100%	9
Ant-à-Maven	1	100%	100%	100%	100%	100%	100%	100%	100%	100%	28
SPEM-à-BPMN	2	94.9%	97.3%	93.7%	92.4%	100%	88.0%	100%	100%	100%	9
Excel-à-XML	1	100%	100%	100%	100%	100%	100%	100%	100%	100%	10
TB-à-DB	1	93.3%	90.0%	100%	93.3%	90.0%	100%	93.3%	90.0%	100%	12
UML-à-Amble	4	93.5%	94.4%	95.1%	78.5%	96.9%	72.9%	95.5%	95.8%	97.1%	16

4.2.1 Correction des modèles cibles (QR1)

Il y a deux problèmes de transformation où le rappel obtenu au niveau de $\widetilde{TP}$, bien que très élevé, ne fut pas parfait : TB-à-DB et UML-à-Amble. Lors de l'inspection des règles de chaque programme, nous avons constaté qu'il y avait, pour le cas TB-à-DB, deux situations qui n'ont pas pu être prises en charge par notre processus d'apprentissage. D'abord, certains éléments cibles possèdent un attribut qui est le résultat d'une addition entre un attribut source et une fonction (cas de fonctions imbriquées). Un exemple de cas est présenté dans le listage 4.1.

Listing 4.1 – Règle utilisant des fonctions imbriquées

```
(deffunction rowsNumberCount ()
        (bind ?result (count-query-results getAllRows))
        (return ?result) )

(defrule r6
(table (name ?a)  (title ?t) (posx ?x) (posy ?y))
=>
(assert (abscoord (ref ?t) (y (+ ?y (rowsNumberCount))) (x ?x))))
```

La deuxième situation TB-à-DB concerne les règles qui doivent générer des éléments qui ne contiennent pas un, mais plusieurs attributs nécessitant des dérivations complexes. Le listage 4.2 illustre ce problème. Dans ce listage, la règle $R11$ est apprise par notre algorithme, alors que ce n'est pas le cas pour $R12$. Nous attribuons l'incapacité de notre approche face à ce genre de cas, au fait que notre fonction fitness n'évalue pas les attributs des éléments produits indépendamment. Si un attribut complexe est dérivé correctement alors qu'un autre ne l'est pas, l'élément comportant ces deux attributs sera classé comme étant partiellement correct. La transformation qui produit cet élément ne sera donc pas favorisée vis à vis des autres transformations qui produisent des correspondances partielles moins bonnes (par exemple, celles qui dérivent correctement un attribut simple seulement).

Concernant le cas UML2Amble, notre algorithme n'a pas pu dériver deux attributs d'un élément en particulier. Ces attributs nécessitent de construire une collection ordonnée d'éléments et de naviguer dans cette collection pour calculer des valeurs. À l'instar des fonctions imbriquées, notre processus est incapable de dériver des règles de cette complexité.

Listing 4.2 – Dérivation de plusieurs attributs complexes

```
(defrule R11
(row (name ?r) (value ?v) (number ?nb) (table ?t))
(table (name ?t) (posx ?x) (posy ?y))
=>
(assert (abscoord (ref ?r) (x ?x) (y  (+ ?y ?nb)))))

(defrule R12
(row (name ?r) (value ?txt) (number ?nb) (table ?t))
(table (name ?t) (posx ?x) (posy ?y))
(cell (name ?n) (value ?v) (ref ?r))
=>
(assert (abscoord (ref ?txt) (y (+ ?y ?nb)) (x (+ ?x ?v)))))
```

Il est important de souligner que le score atteint par chaque *pool* ne doit pas forcément être parfait pour produire au final un programme de transformation correct. Lors de la construction de la paire d'exemples de chaque *pool*, certains éléments peuvent être ajoutés en dépit du fait qu'ils ne soient pas transformés dans le *pool* en question. Ceci est notamment le cas des éléments référencés par un élément principal d'une trace du *pool*. Par exemple, dans SPEM-à-BPMN, les moyennes de rappel et de précision atteintes au niveau des *pools* sont de respectivement 97.3% et 93.7%. Lors de la fusion des transformations, le programme atteint un rappel de 100% sur la paire de modèles initiale ainsi qu'une précision parfaite après la phase de raffinement.

Concernant les problèmes de transformation DC-à-SR et UML-à-Amble, nous avons obtenu un score parfait (resp. une précision parfaite) pour le programme final TP dans un cadre d'expérimentation alternatif, où nous avons augmenté la taille maximale autorisée

au niveau de l'arbre de conditions ajouté à la partie *G* des règles durant la phase de raffinement. Toutefois, lors de l'application des programmes dérivés sur la paire de modèles utilisée pour la validation, une très grande variabilité a été observée dans les résultats. Autoriser des arbres de conditions complexes en utilisant une seule paire d'exemples lors de l'apprentissage, conduit l'algorithme du recuit simulé à apprendre, parfois, des conditions spécifiques à l'exemple. Ces conditions qui se sont avérées ensuite incorrectes pour la paire de validation.

4.2.2 Qualité des règles apprises (QR2)

Le tableau 4.III détaille les résultats obtenus par l'exécution des programmes finaux de transformation TP sur les paires source et cible de validation. Les transformations apprises dans un seul *pool* ont obtenu des scores identiques à ceux réalisés sur la paire d'apprentissage. Quant aux quatre autres cas, les scores réalisés sont légèrement plus bas, excepté SPEM-à-BPMN, où le modèle produit est identique également au modèle cible de validation attendu. Ces résultats confirment que les transformations dérivées ne sont pas spécifiques aux modèles d'apprentissage et qu'elles peuvent ainsi être appliquées avec succès sur de nouveaux modèles.

Concernant la qualité des règles apprises durant la phase de la PG, nous avons noté que les éléments cibles sont toujours produits par les groupes de règles qui sont censés les produire. Néanmoins, nous avons constaté également que certains groupes produisent en plus, des éléments cibles qu'ils ne sont pas censés générer. Ce phénomène, qui se

Tableau 4.III – Fitness, rappel et précision obtenus sur les paires de validation

Transformation	PT		
	Fitness	Rappel	Précision
DC-à-SR	94.7%	96.4%	93.1%
DSL-à-KM3	98.4%	97.9%	98.9%
Ant-à-Maven	100%	100%	100%
SPEM-à-BPMN	100%	100%	100%
Excel-à-XML	100%	100%	100%
TB-à-DB	93.3%	90.0%	100%
UML-à-Amble	90.3%	92.1%	92.7%

manifeste par la présence de doublons dans le modèle cible, est dû au fait que nous fournissons des traces imprécises à l'algorithme. Ces cas ont toutefois été automatiquement gérés par l'étape de post-traitement par laquelle se conclut la seconde phase. Ainsi, dans l'exemple du listage 4.3, la seconde règle est automatiquement supprimée du programme de transformation, car ses conditions sont subsumées par celles de la première règle.

Listing 4.3 – Suppression d'une règle lors du post-traitement

```
(defrule R_649407
(MAIN::class(name ?c00))
=>
(assert (MAIN::table(name ?c00)(alternname nil))))

(defrule R_649408
(MAIN::attribute(name ?a00)(class ?c00)(unico 0))
(MAIN::class(name ?c00))
=>
(assert (MAIN::table(name ?c00)(alternname nil))))
```

Outre le fait que certaines conditions ne sont pas complètes, la qualité des règles apprises durant cette phase est très élevée.

Des conditions complexes ont également été correctement apprises durant la phase de raffinement. Par exemple, dans la règle présentée dans le listage 4.4, un rôle dans DSL est transformé en une référence vers la classe correspondante dans KM3, seulement si la relation du modèle DSL n'est pas complexe. Dans le cas opposé, le rôle est transformé en une classe. La règle du listage filtre les relations DSL simples en vérifiant qu'une relation ne possède pas de propriété (attribut) et qu'elle n'est impliquée dans aucune relation d'héritage, c.-à-d. qu'elle n'est ni une super-classe, ni une sous-classe.

Listing 4.4 – Une règle complexe correctement dérivée

```
(defrule R_514281
(role(name ?ro00)(source ?c00)(rtype ?c10)
(relationship ?re00)(min ?ro04)(max ?ro05)(isordered ?ro06))
(class(name ?c00)(supertype ?c01))
```

```
(class(name ?c10)(supertype ?c11))
(relationship(name ?re00)(supertype ?re01)(isembedding ?re02))

; Conditions apprises durant le raffinement
(and
(and
(not (relationship(name ?re1800)(supertype ?re00)(isembedding ?re1802)) )
(not (valueproperty(name ?vp800)(vtype ?vp801)(owner ?re00))))
(not (relationship(name ?re01)(supertype ?re3121)(isembedding ?re3122))))
=>
(assert (reference(name ?c00)(owner ?c10)(rtype ?c00)
 (lower ?ro04)(upper ?ro05)(iscontainer ?re02)(opposite ?c10))))
```

Dans cet exemple, les conditions dérivées par l'algorithme du recuit simulé correspondent exactement à celles que nous avions écrites. Ceci n'est cependant pas toujours le cas. Certaines règles dérivées durant cette étape comportaient des conditions qui n'étaient correctes que pour la paire d'exemples utilisée lors de l'apprentissage. En fait, nous avons constaté qu'il était toujours possible d'atteindre une précision parfaite lors de la dernière phase d'apprentissage, en augmentant notamment la taille maximale de l'arbre des conditions. Cela se fait toutefois au prix de la généralité des règles obtenues.

4.2.3 Apport des techniques adaptatives (QR3)

Nous avons exécuté l'approche proposée dans ce mémoire plusieurs fois avec et sans les deux mécanismes adaptatifs décrits dans le paragraphe 3.4.2.5. Nous avons observé que pour des transformations complexes, qui requièrent plus d'exploration durant la deuxième phase, de meilleures valeurs de fitness sont atteintes rapidement lors de l'usage de techniques adaptatives.

La figure 2.1 présente l'évolution de la fonction fitness pour deux exécutions différentes dans un *pool* de la transformation CD-à-RS. La courbe de couleur verte (resp. rouge) représente une exécution avec (resp. sans) les mécanismes adaptatifs. L'exécution de la version de PG adaptative surpasse celle de la version standard et converge plus rapi-

dement. Le programme adaptatif atteint un score parfait lors de la 145e génération, alors que le programme standard se trouve bloqué dans un optimum local avec une fitness de 96.7%.

Il est à noter cependant que cette supériorité de la PG adaptative n'a pas été observée sur tous les problèmes de transformation. En effet, lors de nos expériences, nous avons utilisé les mêmes paramètres de PG sur les sept cas de transformations. Ces paramètres, notamment la taille de la population et le nombre de générations, se sont avérés suffisants pour assurer une convergence rapide et optimale au niveau des problèmes de transformation les plus simples.

4.3 Menaces à la validité

L'approche d'apprentissage proposée dans ce mémoire a été appliquée avec succès sur 7 transformations de modèles. Cependant, une menace à la validité de notre expérimentation réside dans la généralisation des résultats obtenus sur toutes les transformations de modèles exogènes. Alors qu'il est vrai que l'échantillon que nous avons sélectionné n'inclut pas tous les problèmes de transformation, nous estimons qu'il n'en demeure pas moins représentatif d'un large éventail de scénarios de transformation rencontrés dans les activités d'ingénierie dirigée par les modèles.

Concernant l'applicabilité de notre approche à d'autres environnements de TM, nous pensons qu'il est tout à fait possible d'adapter le processus d'apprentissage proposé à la plupart des langages de TM existants, puisque nous utilisons un langage entièrement déclaratif sans constructions sophistiquées. La seule particularité de Jess qu'il faut considérer lors de son utilisation en tant que langage de transformation est la gestion des doublons. Étant donné qu'en Jess, créer un élément revient à asserter un fait, créer le même élément plusieurs fois ne résulte pas en plusieurs éléments dans le modèle cible. Nous mitigeons cette menace à travers l'étape de post-traitement de la deuxième phase, en nous assurons que chaque programme dérivé ne produise pas le même fait plus d'une fois.

Une autre menace à la validité de nos résultats se rapporte à l'usage d'une seule

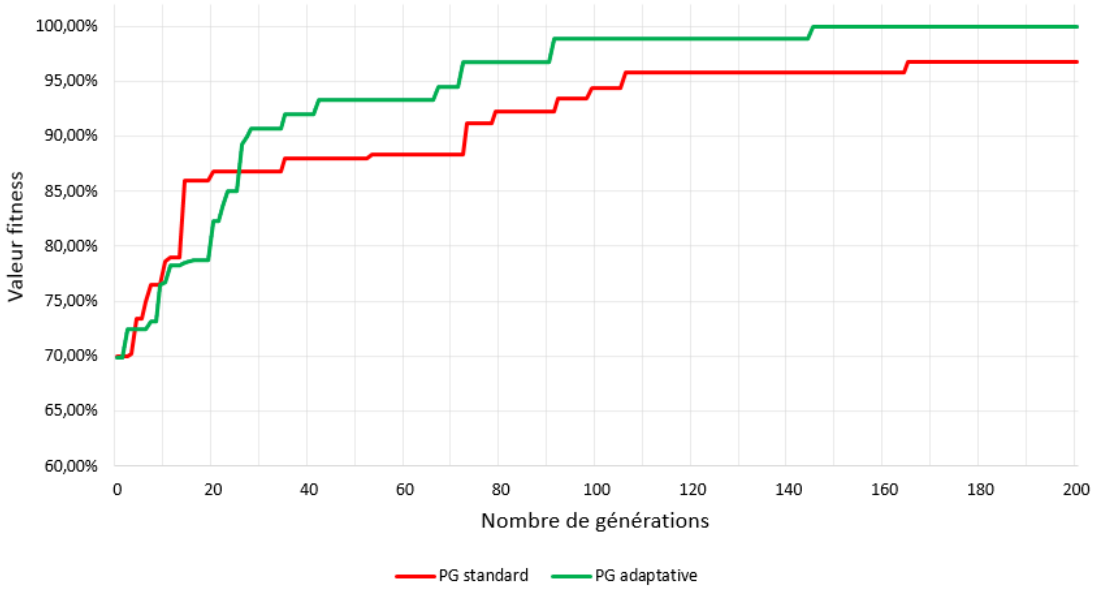

Figure 4.1 – Évolution de la fonction fitness pour une exécution standard et adaptative de la PG

paire d'exemples lors de l'apprentissage de chaque cas de transformation. Nous avons opté pour cette alternative, plutôt que de collecter plusieurs exemples, afin de simplifier notre expérimentation. Il serait raisonnable de s'attendre à ce que l'approche fonctionne aussi bien dans le cas où plusieurs paires d'exemples sont fournies, sous réserve qu'elles couvrent la spécification de la transformation.

Un dernier aspect à considérer en vue des résultats obtenus par l'algorithme d'apprentissage est la qualité des traces fournies. Dans un autre cadre d'utilisation, il est tout à fait possible que certaines traces fournies en entrée soient incorrectes, l'algorithme pourrait alors séparer, par erreur, des traces cohérentes dans des *pools* distincts. Ce type de phénomène est automatiquement géré par le post-traitement qui précède la phase de raffinement et qui permet d'éliminer les règles dupliquées. De plus, étant donné que nous utilisons une approche basée sur la recherche, nous conjecturons que notre processus d'apprentissage est robuste face à de telles imperfections lorsque leur proportion est faible.

4.4 Discussion

4.4.1 Traces de transformation

Contrairement à certaines contributions discutées dans la section 2.2, l'approche que nous proposons dans ce mémoire nécessite que des traces de transformation soient fournies conjointement à chaque paire d'exemples (à l'instar de la majorité des approches de TMPE). Bien que cette exigence pourrait être vue comme une limite de l'approche, elle s'avère nécessaire pour réduire la taille de l'espace de recherche lors de l'apprentissage de transformations de modèles complexes.

Dans notre contribution, nous mitigeons la surcharge causée par la nécessité de fournir des traces en acceptant des traces imprécises. Nous pensons également que le travail supplémentaire requis pour définir les traces peut être sensiblement réduit en assistant l'expert par un outil. En particulier, lors de la manipulation de grands modèles, l'expert pourrait alors définir la correspondance d'un seul type d'éléments sources, l'outil analyserait ensuite la correspondance et suggèrerait des correspondances pour les éléments du même type qui se trouvent dans des contextes similaires.

4.4.2 Règles complexes

Lors de la validation de notre approche, nous avons constaté que notre algorithme d'apprentissage était incapable de produire tous les éléments attendus ($rappel < 100\%$ dans $\widetilde{TP}$) pour deux transformations, à savoir, TB-à-DB et UML-à-Amble.

Dans le cas du TB-à-DB, des fonctions imbriquées et une dérivation simultanée de plusieurs attributs complexes étaient requises. Cette dernière caractéristique peut être prise en charge par notre processus, en définissant, par exemple, une fonction fitness qui évalue à un niveau plus fin les programmes de transformation. Ainsi, il serait judicieux de classer les correspondances partielles produites par une transformation en fonction de leurs ratios d'attributs correctement dérivés. Implémenter une telle idée requiert, toutefois, une comparaison de modèles bien plus sophistiquée et donc coûteuse.

Lors de l'apprentissage de la transformation UML-à-Amble, l'algorithme n'a pas pu apprendre une règle en particulier qui nécessitait d'effectuer des itérations pour calcu-

ler des valeurs sur une collection ordonnée d'éléments sources. Afin de permettre au programme génétique d'apprendre ce genre de règles, il est nécessaire d'ajouter des opérateurs d'ensembles, en particulier, les opérations de tri et d'itération. À l'instar de la composition de fonctions, de tels apports augmenteraient considérablement la taille de l'espace de recherche.

4.4.3 Généralisation de la transformation

Durant les expériences que nous avons conduites, nous avons été capables d'atteindre une précision parfaite après la phase de raffinement en augmentant la taille maximale des arbres de conditions. L'approche a cependant tendance, dans ces cas-là, à apprendre des conditions qui sont spécifiques aux exemples fournis.

Il existe généralement, pour un ensemble d'entrées/sorties, plusieurs programmes qui permettent d'obtenir les sorties escomptées à partir des entrées fournies. Afin de produire la transformation recherchée, il est donc impératif d'augmenter le nombre ou la taille et la complexité des modèles utilisés lors de l'apprentissage. Dans le cadre de l'approche que nous proposons, il est possible d'adopter un style de développement piloté par les tests [37]. Le processus d'apprentissage est exécuté de manière itérative à chaque fois que la base d'exemples est enrichie par une nouvelle paire de modèles. Si la transformation n'est pas satisfaisante, l'expert pourra réviser les exemples et les correspondances fournis à l'algorithme, ou corriger directement le programme de transformation dérivé.

4.4.4 Qualité des règles

Un des objectifs de la phase de raffinement des règles est d'apprendre le contrôle requis pour former un programme de transformation correct. Dans notre approche ce contrôle est implicite [5]. Il se manifeste par des règles conditionnées par les éléments cibles que produiraient d'autres règles (c.-à-d. des parties G portant sur des éléments cibles). Dans certains cas, ceci a pour effet de produire des règles qui sont plus difficiles à comprendre et à déboguer.

Une solution possible à ce problème serait de rajouter une mesure de qualité à la

fonction fitness utilisée durant l'algorithme du RS. Cette fonction pourrait ainsi favoriser les règles qui portent sur des patrons du modèle source. Une analyse plus approfondie est cependant nécessaire afin de déterminer comment cela pourrait être réalisé sans affecter l'apprentissage du contrôle implicite.

4.4.5 Passage à l'échelle de l'approche

Lors de la validation de l'approche proposée, l'exécution du processus d'apprentissage pour chaque problème de transformation a duré environ une heure. Nous avons opté au cours de nos expériences pour une stratégie de *tuning* des algorithmes. Les paramètres de chaque algorithme ont été donc déterminés en testant plusieurs combinaisons de paramètres possibles sur les sept transformations. Nous avons ensuite choisi une seule configuration pour chaque algorithme que nous avons maintenue pour tous les problèmes de transformation.

En raison de notre choix d'uniformiser les paramètres d'exécution, plusieurs transformations telles que qu'Excel-à-XML et Ant-à-Maven ont été apprises totalement avant que l'algorithme atteigne les dernières générations. De plus, pour certaines transformations, un plus grand nombre de générations était requis dans la phase de la PG, alors que pour d'autres, telle que DSL-à-KM3, c'est la phase du RS qui fut chargée d'apprendre les conditions les plus complexes pour compléter les règles dérivées.

Une approche plus sophistiquée pour déterminer les paramètres de chaque algorithme de recherche permettrait d'augmenter l'efficacité de notre approche. Il serait en particulier intéressant de considérer les travaux d'Eiben et al. [21, 22] sur le contrôle des paramètres des algorithmes de recherche.

CHAPITRE 5

CONCLUSION

Nous avons proposé à travers ce mémoire un processus d'apprentissage de transformations de modèles exogènes. L'approche décrite est basée sur la recherche ; elle prend en entrée des paires de modèles sources et cibles agrémentées par des traces de transformation. Les exemples fournis sont alors utilisés pour dériver, de manière automatique, la transformation de modèles escomptée.

Le processus d'apprentissage proposé se compose de trois phases. Durant la première phase, les traces d'exemples sont analysées afin de construire des *pools*. Chaque *pool* contient un sous-ensemble de traces ainsi qu'une version réduite des paires de modèles initiales. La 2e phase du processus utilise un algorithme inspiré de la programmation génétique adaptative au niveau de chaque *pool*, pour dériver un programme de transformation. L'algorithme exploite les informations fournies par les traces afin de réduire la taille de l'espace de recherche. Les transformations obtenues sont ensuite traitées et combinées dans un seul programme de transformation. La dernière phase consiste à raffiner la transformation en utilisant un algorithme du recuit simulé qui œuvre à parfaire les règles apprises dans les différents *pools* en complétant leurs parties conditionnelles.

Nous avons évalué notre approche sur sept cas de transformation. Notre validation démontre que le processus d'apprentissage proposé est capable de dériver des transformations pour des spécifications complexes qui nécessitent, entre autres, l'exploration du modèle source, la vérification de valeurs d'attributs sources et la dérivation d'attributs cibles complexes. Les transformations apprises produisent des modèles cibles très similaires à ceux attendus. De plus, l'inspection des programmes dérivés a révélé que plusieurs règles sont identiques à ce qu'écrirait un expert.

Malgré ces résultats très encourageants, il existe de nombreuses pistes d'amélioration qui méritent d'être explorées. Dans des travaux futurs, nous comptons (1) explorer l'applicabilité de notre approche dans un environnement de modélisation plus commun, tel que l'*Eclipse Modeling Framework*, (2) développer un outil graphique qui permettrait

d'assister l'expert (par exemple, à travers des suggestions interactives) dans la définition des traces entre les domaines source et cible, et (3) évaluer à quel degré contribue notre processus d'apprentissage à la réduction des efforts de développement des transformations de modèles.

Concernant l'approche proposée, nous comptons remédier à ses limites actuelles en modifiant l'algorithme pour permettre l'apprentissage des éléments cibles qui possèdent plusieurs attributs complexes. Deux options méritent d'être explorées en ce sens : la définition d'une fonction fitness qui permet une comparaison plus fine, mais efficiente des modèles, et l'usage d'une mémoire pour stocker les fonctions complexes (imbriquées) qui ont été correctement dérivées. Une autre amélioration possible serait d'introduire des structures qui permettraient de manipuler des collections. Ceci requiert, cependant, la mise en place conjointe de mécanismes réduisant la taille de l'espace de recherche. Enfin, il serait intéressant de tirer profit des travaux portant sur le contrôle des paramètres des algorithmes de recherches [21, 85] afin de gérer, de manière efficace, la singularité de l'espace des solutions de chaque problème de transformation.

BIBLIOGRAPHIE

[1] Emile Aarts et Jan Korst. *Simulated annealing and Boltzmann machines : a stochastic approach to combinatorial optimization and neural computing*. Wiley, 1988.

[2] Moussa Amrani, Jürgen Dingel, Leen Lambers, Levi Lúcio, Rick Salay, Gehan Selim, Eugene Syriani et Manuel Wimmer. Towards a model transformation intent catalog. Dans *Proceedings of the First Workshop on the Analysis of Model Transformations*, pages 3–8. ACM, 2012.

[3] Jerome Andre, Patrick Siarry et Thomas Dognon. An improvement of the standard genetic algorithm fighting premature convergence in continuous optimization. *Advances in engineering software*, 32(1):49–60, 2001.

[4] Peter Angeline et Jordan Pollack. Evolutionary module acquisition. Dans *Proceedings of the second annual conference on evolutionary programming*, pages 154–163. Citeseer, 1993.

[5] Islem Baki, Houari Sahraoui, Quentin Cobbaert, Philippe Masson et Martin Faunes. Learning implicit and explicit control in model transformations by example. Dans *Model-Driven Engineering Languages and Systems*, pages 636–652. Springer, 2014.

[6] Daniel Balasubramanian, Anantha Narayanan, Christopher van Buskirk et Gabor Karsai. The graph rewriting and transformation language : Great. *Electronic Communications of the EASST*, 1, 2007.

[7] Zoltan Balogh et Dániel Varró. Model transformation by example using inductive logic programming. *Software and Systems Modeling*, 8:347–364, 2009. ISSN 1619-1366.

[8] Keith Bearpark et Andy J Keane. The use of collective memory in genetic pro-

gramming. Dans *Knowledge Incorporation in Evolutionary Computation*, pages 15–36. Springer, 2005.

[9] Marco Brambilla, Jordi Cabot et Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[10] Daniel Brolund. Jambda, 2009. URL `http://sourceforge.net/projects/jamda`.

[11] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger et Wieland Schwinger. An example is worth a thousand words : Composite operation modeling by-example. Dans *Model Driven Engineering Languages and Systems*, pages 271–285. Springer, 2009.

[12] Petra Brosch, Philip Langer, Martina Seidl et Manuel Wimmer. Towards end-user adaptable model versioning : The by-example operation recorder. Dans *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 55–60. IEEE Computer Society, 2009.

[13] Sabine Buckl, Alexander M Ernst, Josef Lankes, Christian M Schweda et André Wittenburg. Generating visualizations of enterprise architectures using model transformations. Dans *EMISA*, volume 2007, pages 33–46, 2007.

[14] Jesús Sánchez Cuadrado. Towards a family of model transformation languages. Dans *Theory and Practice of Model Transformations*, pages 176–191. Springer, 2012.

[15] Krzysztof Czarnecki et Michał Antkiewicz. Mapping features to models : A template approach based on superimposed variants. Dans *Generative Programming and Component Engineering*, pages 422–437. Springer, 2005.

[16] Krzysztof Czarnecki et Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[17] Juan de Lara et Gabriele Taentzer. Automated model transformation and its validation using atom3 and agg. Dans *Diagrammatic Representation and Inference*, pages 182–198. Springer, 2004.

[18] Narayan Debnath, Fabio Zorzan, German Montejano et Daniel Riesco. Transformation of bpmn subprocesses based in spem using qvt. Dans *Electro/Information Technology, 2007 IEEE International Conference on*, pages 146–151. IEEE, 2007.

[19] Xavier Dolques, Marianne Huchard, Clémentine Nebut et Philippe Reitz. Learning transformation rules from transformation examples : An approach based on relational concept analysis. Dans *International Conference on Enterprise Distributed Object Computing Workshops*, pages 27 –32, 2010.

[20] Radomil Dvorak. Model transformation with operational qvt. *Borland Software Corporation*, 19, 2008.

[21] Agoston Eiben, Zbigniew Michalewicz, Marc Schoenauer et James Smith. Parameter control in evolutionary algorithms. Dans *Parameter setting in evolutionary algorithms*, pages 19–46. Springer, 2007.

[22] Agoston Endre Eiben, Robert Hinterding et Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 3(2):124–141, 1999.

[23] Martin Faunes, Houari Sahraoui et Mounir Boukadoum. Generating model transformation rules from examples using an evolutionary algorithm. Dans *Automated Software Engineering*, pages 1–4, 2012.

[24] Martin Faunes, Houari Sahraoui et Mounir Boukadoum. Genetic-programming approach to learn model transformation rules from examples. Dans *Theory and Practice of Model Transformations*, pages 17–32. Springer, 2013.

[25] Charles Forgy. Rete : A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982.

[26] Andrew Forward, O Badreddin et TC Lethbridge. Perceptions of software modeling : a survey of software practitioners. Dans *5th workshop from code centric to model centric : evaluating the effectiveness of MDD (C2M : EEMDD)*, 2010.

[27] Ernest Friedman-Hill. *JESS in Action*. Manning Greenwich, CT, 2003.

[28] Iván García-Magariño, Jorge J. Gómez-Sanz et Rubén Fuentes-Fernández. Model transformation by-example : An algorithm for generating many-to-many transformation rules in several model transformation languages. Dans *Proceedings of the International Conference on Theory and Practice of Model Transformations*, pages 52–66, 2009.

[29] Birgit Grammel, Stefan Kastenholz et Konrad Voigt. *Model matching for trace link generation in model-driven software development*. Springer, 2012.

[30] Daniel Conrad Halbert. *Programming by example*. Thèse de doctorat, University of California, Berkeley, 1984.

[31] Florian Heidenreich, Jan Kopcsek et Uwe Aßmann. Safe composition of transformations. Dans *Theory and Practice of Model Transformations*, pages 108–122. Springer, 2010.

[32] John H Holland. *Adaptation in natural and artificial systems : An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[33] Lester Ingber. Simulated annealing : Practice versus theory. *Mathematical and computer modelling*, 18(11):29–57, 1993.

[34] Jean-Marc Jézéquel, Olivier Barais et Franck Fleurey. Model driven language engineering with kermeta. Dans *Generative and Transformational Techniques in Software Engineering III*, pages 201–221. Springer, 2011.

[35] Frédéric Jouault et Ivan Kurtev. Transforming models with atl. Dans *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.

[36] Audris Kalnins, Janis Barzdins et Edgars Celms. Model transformation language mola. Dans *Model Driven Architecture*, pages 62–76. Springer, 2005.

[37] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger et Manuel Wimmer. Model transformation by-example : a survey of the first wave. Dans *Conceptual Modelling and Its Theoretical Foundations*, pages 197–215. Springer, 2012.

[38] Marouane Kessentini. *Transformation by example*. Thèse de doctorat, University of Montreal, Canada, 2010.

[39] Marouane Kessentini, Houari Sahraoui et Mounir Boukadoum. Model transformation as an optimization problem. Dans *Model Driven Engineering Languages and Systems*, pages 159–173. Springer, 2008.

[40] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum et Omar Ben Omar. Search-based model transformation by example. *Software and System Modeling*, 11(2):209–226, 2012.

[41] Marouane Kessentini, Manuel Wimmer, Houari Sahraoui et Mounir Boukadoum. Generating transformation rules from examples for behavioral models. Dans *Proc. of the 2nd Int. Workshop on Behaviour Modelling : Foundation and Applications*, pages 2 :1–2 :7, 2010.

[42] Anneke G Kleppe. *Software language engineering : creating domain-specific languages using metamodels*. Pearson Education, 2008.

[43] Anneke G Kleppe, Jos B Warmer et Wim Bast. *MDA explained : the model driven architecture : practice and promise*. Addison-Wesley Professional, 2003.

[44] John Koza, David Andre, Forrest Bennett III et Martin Keane. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis with genetic programming. Dans *Proceedings of the First Annual Conference on Genetic Programming*, pages 132–140. MIT Press, 1996.

[45] John R Koza. *Genetic programming III : Darwinian invention and problem solving*, volume 3. Morgan Kaufmann, 1999.

[46] Thomas Kühne. *What is a Model ?* Internat. Begegnungs-und Forschungszentrum für Informatik, 2005.

[47] Ramnivas Laddad. *AspectJ in action : practical aspect-oriented programming*. Dreamtech Press, 2003.

[48] Philip Langer, Manuel Wimmer et Gerti Kappel. Model-to-model transformations by demonstration. Dans *Theory and Practice of Model Transformations*, pages 153–167. Springer, 2010.

[49] Kevin Lano et Shekoufeh Kolahdouz-Rahimi. Model-driven development of model transformations. Dans *Theory and practice of model transformations*, pages 47–61. Springer, 2011.

[50] Michael Lawley et Jim Steel. Practical declarative model transformation with tefkat. Dans *Satellite Events at the MoDELS 2005 Conference*, pages 139–150. Springer, 2006.

[51] Theodore Gyle Lewis. *CASE : computer-aided software engineering*. Van Nostrand Reinhold Co., 1990.

[52] Levi Lúcio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan Selim, Eugene Syriani et Manuel Wimmer. Model transformation intents and their properties. *Software & Systems Modeling*, 2014.

[53] Jochen Ludewig. Models in software engineering–an introduction. *Software and Systems Modeling*, 2(1):5–14, 2003.

[54] Sean Luke et Lee Spector. A revised comparison of crossover and mutation in genetic programming. *Genetic Programming*, 98(55):208–213, 1998.

[55] Frank Marschall et Peter Braun. Botl-the bidirectional object oriented transformation language. *Instituto de Informática, Universidad Técnica de Munich. Munich*, 2003.

[56] James Martin. *Application development without programmers*. Prentice Hall PTR, 1982.

[57] Tom Mens et Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[58] Compuware OptimalJ. Model-driven development for java, 2001.

[59] François Pfister, Vincent Chapurlat, Marianne Huchard et Clémentine Nebut. A proposed tool and process to design domain specific modeling languages, 2012.

[60] Justinian P Rosca. Genetic programming exploratory power and the discovery of functions. Dans *Evolutionary Programming*, pages 719–736. Citeseer, 1995.

[61] James Rumbaugh, Ivar Jacobson et Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[62] Hajer Saada, Xavier Dolques, Marianne Huchard, Clémentine Nebut et Houari Sahraoui. Generation of operational transformation rules from examples of model transformations. Dans *International Conference on Model Driven Engineering Languages and Systems*, 2012.

[63] Hajer Saada, Xavier Dolques, Marianne Huchard, Clémentine Nebut et Houari Sahraoui. Learning model transformations from examples using fca : One for all or all for one ? Dans *CLA'2012 : 9th International Conference on Concept Lattices and Applications*, pages 45–56, 2012.

[64] Hajer Saada, Marianne Huchard, Clémentine Nebut et Houari Sahraoui. Recovering model transformation traces using multi-objective optimization. Dans *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference*, pages 688–693. IEEE, 2013.

[65] Douglas C Schmidt. Model-driven engineering. *IEEE Computer Society*, 39(2):25, 2006.

[66] Mandavilli Srinivas et Lalit Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *Systems, Man and Cybernetics, IEEE Transactions on*, 24(4):656–667, 1994.

[67] Herbert Stachowiak. Allgemeine modelltheorie. *Stachowiak H. : Models'[w :] Scientific Thought. Some Underlying Concepts*, 1973.

[68] Michael Strommer, Marion Murzek et Manuel Wimmer. Applying model transformation by-example on business process modeling languages. Dans *Advances in Conceptual Modeling–Foundations and Applications*, pages 116–125. Springer, 2007.

[69] Michael Strommer et Manuel Wimmer. A framework for model transformation by-example : Concepts and tool support. Dans *Objects, Components, Models and Patterns*, Lecture Notes in Business Information Processing, pages 372–391. Springer, 2008.

[70] Yu Sun, Jules White et Jeff Gray. Model transformation by demonstration. Dans *Model Driven Engineering Languages and Systems*, pages 712–726. Springer, 2009.

[71] Eugene Syriani. *A multi-paradigm foundation for model transformation language engineering*. Thèse de doctorat, McGill University, 2011.

[72] Eugene Syriani, Hans Vangheluwe et Brian LaShomb. T-core : a framework for custom-built model transformation engines. *Software & Systems Modeling*, pages 1–29, 2013.

[73] Helena Syrjakow, Matthias Szczerbicka et Michael Becker. Genetic algorithms : a tool for modelling, simulation, and optimization of complex systems. *Cybernetics & Systems*, 29:639–659, 1998.

[74] Gabriele Taentzer. Agg : A graph transformation environment for modeling and validation of software. Dans *Applications of Graph Transformations with Industrial Relevance*, pages 446–453. Springer, 2004.

[75] Dániel Varró. Automated program generation for and by model transformation systems. *Applied Graph Transformation (AGT'02)*, pages 161–174, 2002.

[76] Daniel Varró. Model transformation by example. Dans *International Conference on Model Driven Engineering Languages and Systems*, pages 410–424. Springer, 2006.

[77] Didier Vojtisek, Jean-Marc Jézéquel et al. Mtl and umlaut ng-engine and framework for model transformation. *ERCIM News 58*, 58, 2004.

[78] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase et Simon Helsen. *Model-driven software development : technology, engineering, management*. John Wiley & Sons, 2013.

[79] Matthew Walker. Introduction to genetic programming. Rapport technique, University of Montana, 2001. URL `http://www.cs.montana.edu/~bwall/cs580/introduction_to_gp.pdf`.

[80] Jos B Warmer et Anneke G Kleppe. *The Object Constraint Language : Precise Modeling With Uml (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 1998.

[81] Manuel Wimmer, Michael Strommer, Horst Kargl et Gerhard Kramler. Towards model transformation generation by-example. Dans *Annual Hawaii International Conference on System Sciences*, pages 285–295, 2007.

[82] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, 1978.

[83] Shengxiang Yang et Şima Uyar. Adaptive mutation with fitness and allele distribution correlation for genetic algorithms. Dans *Proceedings of the 2006 ACM symposium on Applied computing*, pages 940–944. ACM, 2006.

[84] Xinjie Yu et Mitsuo Gen. *Introduction to evolutionary algorithms*. Springer, 2010.

[85] Liang Zhang et Ling Wang. Optimal parameters selection for simulated annealing with limited computational effort. Dans *Proceedings of the 2003 International Conference on Neural Networks and Signal Processing*, volume 1, pages 412–415. IEEE, 2003.

[86] Moshé M Zloof. Query by example. Dans *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 431–438. ACM, 1975.