MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITÉ ABDELHAMID IBN BADIS DE MOSTAGANEM

Faculté des Sciences Exactes et d'Informatique
Département de Mathématiques et d'Informatique
Filière Informatique

MÉMOIRE DE FIN D'ÉTUDES
Pour l'Obtention du Diplôme de Master en Informatique
Option : Ingénierie des Systèmes d'Information

Une Approche Heuristique pour l'Apprentissage du Refactoring des Diagrammes de Classes à partir d'Exemples

Étudiant :

MOKADDEM Chihab Eddine Mohamed Omar.

Encadrant :

Mr. HENNI Fouad.

Co-Encadrant :

Pr. SAHRAOUI Houari – Université de Montréal, Canada.

Année Universitaire 2014/2015

RÉSUMÉ

L'ingénierie dirigée par les modèles ou « Model Driven Engineering » (MDE) est un paradigme qui promet de réduire la complexité du logiciel par l'utilisation intensive de modèles et des transformations automatiques entre modèles (TMs). L'automatisation des transformations de modèles demeure cependant une tâche ardue.

La transformation de modèles à partir des exemples ou « Model Transformation By Examples » (MTBE) s'est avérée efficace et prometteuse pour apprendre automatiquement les TMs et extraire les bonnes connaissances sur ces TMs sous forme de règles opérationnelles qu'on appelle « Règles de Transformation » (RTs).

L'état de l'art des approches existantes du MTBE montre le manque de travaux portant sur la transformation endogène. Ces approches souffrent de limitations telles que la difficulté de les mettre en œuvre, leur applicabilité limitée, et les règles, souvent triviales et peu riches, qu'elles produisent.

Dans le cadre de ce projet, nous présentons une approche qui prend en charge la transformation endogène la plus répandue dans le processus du développement et de maintenance des logiciels : le refactoring des diagrammes de classes UML. L'approche s'inspire de la programmation génétique (GP) pour dériver itérativement des règles de transformation de qualité, à partir des exemples de refactorings préétablis. Il s'agit d'une approche prometteuse qui ouvre des perspectives sur la qualité des résultats obtenus.

Mots clés : Apprentissage des transformations de modèles, transformation de modèles Par l'exemple, ingénierie dirigée par les modèles, génie logiciel basé sur la recherche, programmation génétique, refactoring des diagrammes de classes.

ABSTRACT

Model Driven Engineering (MDE) is a paradigm that promises to reduce software complexity by the intensive use of models and automatic model transformations (MTs). However, model transformation automation, remains a hard task.

Model Transformation By Examples (MTBE) has been found efficient and promising to automatically learn MTs and extract good knowledge about these MTs, in the form of operational rules called «Transformation Rules» (TRs).

The state of the art of existing MTBE approaches shows the lack of works on endogenous transformation. These approaches suffer from limitations such as the difficulty of their implementation, their limited applicability, and rules, often trivial and not rich, they produce.

The approach we propose in this project comes to handle the most common endogenous transformation in the software development and maintenance process: the UML class diagram refactoring. The approach is based on Genetic Programming (GP) in order to; iteratively; derive transformation rules of high quality from examples of pre-established refactorings. This is a promising approach that opens perspectives on the quality of obtained results.

Keywords : Model transformation learning, model transformation by example, model driven engineering, search-based software engineering, genetic programming, class diagrams refactoring.

SOMMAIRE

TABLE DES FIGURES

LISTE DES ABRÉVIATIONS

AE	Algorithmes Evolutionnaires
GP	Genetic Programming
IDM	Ingénierie Dirigée par les Modèles
JESS	Java Expert System Shell
MDE	Model Driven Engineering
MTBD	Model Transformation By Demonstration
MTBE	Model Transformation By Examples
n-m	Many-to-many
1-n	One-to-many
OMG	Object Management Group
POO	Paradigme Orienté Objet
RT	Règle de Transformation
TM	Transformation de Modèles
UML	Unified Modeling Language
LHS	Left-Hand Side
RHS	Right-Hand Side
AP	Assert Pattern
MP	Modify Pattern
RP	Retract Pattern

REMERCIEMENTS

Avant tout, nous remercions ***Allah Azza wa Jalla*** le tout puissant de nous avoir donné le courage et la patience pour réaliser ce projet.

La réalisation de ce travail n'aurait pas été possible sans le soutien et l'aide précieuse de plusieurs personnes. Nous tenons à leur témoigner notre gratitude.

En premier lieu, nous adressons notre sincère reconnaissance à ***Monsieur Fouad Henni*** qui nous a encadrés et soutenus avec ses conseils et suggestions précieuses durant toutes les étapes de ce projet.

Nous exprimons nos plus vifs remerciements à notre co-encadreur ***Monsieur Houari Sahraoui***, de l'université de Montréal à Canada, qui a initié et suivi le thème du présent mémoire. Ses conseils avisés, ses critiques constructives, son suivi permanent, et sa patience furent certes, très favorables à l'aboutissement de ce projet.

Nous remercions les membres du jury pour avoir accepté de juger et d'évaluer ce travail, ainsi que pour l'attention qu'ils nous ont accordée.

Toute notre gratitude et reconnaissance à nos très chers parents pour le soutien moral et affectif qu'ils nous ont apporté tout au long de cette expérience. Nous leur dédicaçons ce travail.

Enfin, nous remercions chaleureusement tous nos amis, ainsi que nos familles pour nous avoir encouragés durant la réalisation de ce projet.

INTRODUCTION GÉNÉRALE

L'adoption de nouvelles technologies suit généralement un cycle récurrent décrit par Moore dans [Moo 02]. Dans ce cycle, les différentes catégories d'utilisateurs adoptent une technologie selon leurs profiles et la maturité de cette technologie. Pour toute technologie, Moore identifie le passage de la catégorie « récemment adoptée » à la catégorie « mature » ou « largement adoptée » comme le fossé le plus difficile à traverser et dans lequel de nombreuses technologies passent une longue durée ou échouent définitivement. Etant une nouvelle technologie qui change considérablement la façon dont nous développons des logiciels, l'ingénierie dirigée par les modèles ou « Model Driven Engineering » (MDE) n'échappe pas, elle aussi, à cette observation.

Le MDE fait référence à une série d'approches où les modèles jouent un rôle central dans le développement de logiciels. La modélisation favorise le raisonnement à un niveau d'abstraction plus élevé, réduisant ainsi la complexité du développement de logiciels, tout en cachant les détails de bas niveau inutiles aux étapes appropriées, et promouvant la communication entre les parties intervenantes dans le processus de développement.

Le MDE a reçu beaucoup d'attention ces dernières années en raison de sa promesse de réduire la complexité de la mise au point et de la maintenance des applications logicielles. Cependant, et malgré les réussites rapportées au cours de cette dernière décennie, le MDE est encore à un stade non mature [Moh 10]. Selic stipule qu'en plus des facteurs économiques et culturels, les facteurs techniques, en particulier la difficulté d'automatisation, représentent des obstacles majeurs à l'adoption du MDE.

L'automatisation est un principe clé et fondateur du paradigme MDE. Selon Schmidt, les technologies MDE combinent des langages de modélisation dédiés à des domaines spécifiques, avec des moteurs et des générateurs de transformation pour produire divers artéfacts logiciels [Sch 06]. En automatisant les transformations modèle-à-modèle et modèle-à-code, le MDE franchit le fossé conceptuel entre le code source et les modèles et assure que les modèles obtenus soient à jour vis-à-vis du code et des autres modèles.

L'automatisation d'une activité engendre toujours un coût. Tout d'abord, il faut que la bonne connaissance de la façon dont l'activité doit être effectuée soit recueillie. De plus, cette connaissance doit être convertie en algorithmes cohérents et évolutifs. Enfin, les algorithmes doivent être validés globalement sur un ensemble de tests qui considère la plupart des cas courants mais aussi des situations particulières. Par conséquent, améliorer le degré d'automatisation consiste à remplacer les tâches de MDE, qui sont effectuées manuellement, par des processus automatiques (par ordinateur). Par contre, l'amélioration de l'automatisation des tâches déjà automatisées, consiste à les remplacer par de nouveaux processus qui nécessitent moins d'interactions humaines.

En ce qui concerne la transformation de modèles, en tant qu'activité centrale du MDE, les connaissances sur la façon de transformer un modèle source à un autre modèle cible sont souvent incomplètes ou non disponibles. La difficulté de la génération des règles de transformation est la principale motivation derrière la recherche sur l'apprentissage des règles de transformation à partir d'exemples. Bien que l'idée remonte au début des années 90,

le premier travail concret sur la transformation de modèles à partir des exemples « Model Transformation By Examples » (MTBE) a été proposé par Varró en 2006 [Var 06]. L'objectif du MTBE est de dériver des programmes de transformation, en appliquant un apprentissage automatique supervisé sur un ensemble d'exemples de modèles prototypes sources et cibles. Depuis lors, de nombreuses approches ont été proposées pour dériver des règles de transformation ou pour transformer un modèle par analogie à des exemples transformés [Kes 08]. Toutefois, les approches existantes du MTBE ne résolvent que partiellement, le problème de dérivation des règles. La majorité d'entre elles souffrent de quelques limitations qui les rendent difficiles à exploiter, et qui diminuent ainsi leur applicabilité [Saad 12].

Dans un travail précédent, une approche a été proposée pour la dérivation de règles complexes, riches, et exécutables à partir d'exemples sans la nécessité des traces de transformation. L'approche utilisée s'inspire de la programmation génétique (GP) et exploite la capacité de la GP pour faire évoluer des programmes afin d'améliorer leur aptitude à approximer un comportement défini par un ensemble de paires valides d'entrées/sorties [Fau 13b]. L'approche que nous venons de citer traite uniquement le cas de transformation exogène et, dans le cadre de ce projet, nous tentons d'appliquer les mêmes techniques à l'une des transformations endogènes les plus répandues. Il s'agit du refactoring des modèles, en particulier, le refactoring des diagrammes de classes UML, qui joue un rôle prépondérant en MDE durant tout le processus de développement, et dont les techniques et les travaux sont peu nombreuses dans l'état de l'art. Nous soulevons également dans ce travail, les problèmes rencontrés lors de notre exploit. Il y a aussi lieu de proposer des améliorations et des alternatives en fonction des résultats obtenus.

Ce mémoire sera organisé comme suit : dans un 1er chapitre, les notions et les concepts de base permettant de réaliser une démarche dirigée par les modèles, sont introduits. Ensuite, dans le 2ème chapitre, nous mettons l'accent sur l'importance de l'automatisation en MDE, et nous dressons un état de l'art des approches de transformation de modèles par l'exemple, avant d'aborder la transformation endogène en général. La suite du chapitre sera consacrée au cas particulier du refactoring des modèles que nous explorons, à travers le présent projet, dans le cadre plus spécifique du refactoring des diagrammes de classes UML. Dans le 3ème chapitre, nous détaillons notre approche proposée pour résoudre le problème de dérivation des règles de transformation opérationnelles permettant le refactoring d'un diagramme de classes et ceci, en exploitant la programmation génétique combinée aux exemples concrets de tels refactorings. Le dernier chapitre sera consacré à la mise en œuvre de notre solution. Il aborde dans un premier temps, les outils utilisés, ainsi que leurs configurations nécessaires. Puis, il inclut une validation de notre approche sur un exemple concret d'un diagramme de classes mal construit. Le chapitre se termine par une évaluation quantitative et qualitative des règles de transformation obtenues. Enfin, une conclusion et des perspectives clôturent ce travail.

CHAPITRE I : INGÉNIERIE DIRIGÉE PAR LES MODÈLES

I.1. INTRODUCTION

Au début de cette décennie, la taille et la complexité des logiciels développés augmentent de plus en plus rapidement. Les technologies de composants et de l'orienté objet semblent insuffisantes pour fournir des solutions satisfaisantes en vue d'assurer convenablement le développement et la maintenance des logiciels. De nouvelles techniques de génie logiciel sont nécessaires afin de remédier à la complexité et d'assurer la qualité du logiciel produit. Ces techniques sont regroupées sous l'intitulé « Ingénierie Dirigée par les Modèles » (IDM) ou « Model Driven Engineering » (MDE). Ainsi, le MDE est un paradigme qui promet de réduire la complexité du logiciel par l'usage intensif de modèles et de leurs transformations automatiques (TMs).

Une TM est un programme qui prend en entrée un modèle source conforme à un méta-modèle source et produit en sortie un autre modèle cible conforme à un méta-modèle cible. La réalisation de cette transformation exige une connaissance profonde et solide des modèles (source, cible), des méta-modèles (source, cible), et des techniques (algorithmes, heuristiques) de transformations.

Les experts du domaine ne disposent, généralement, pas de suffisamment de compétences en MDE, leurs connaissances ne sont que partielles. Ainsi, regrouper ces connaissances devient une tâche primordiale. Une approche novatrice, appelée « Transformation de modèles à partir des exemples » ou « Model Transformation By Examples » (MTBE), est proposée. Cette approche consiste à observer un expert pendant sa transformation manuelle de modèles et tenter d'extraire cette expertise par des techniques d'apprentissage automatique. Elle fait donc usage d'un ensemble initial d'exemples (base des exemples), l'exemple étant une transformation de modèle.

Ce premier chapitre présente les concepts de base permettant la réalisation, voire l'amélioration, du processus MTBE, et qui sont nécessaires à une démarche dirigée par les modèles. Pour cela, nous commencerons par introduire les notions du MDE avec toutes ses caractéristiques, modèles, méta-modèles, et les transformations de modèles existantes. Ensuite, nous entamons les principes de transformation de modèles par l'exemple.

I.2. L'INGÉNIERIE DIRIGÉE PAR LES MODÈLES

I.2.1. Définition du MDE

Le MDE est un paradigme qui promet de réduire la complexité d'un logiciel par usage intensif de modèles et de leurs transformations automatiques. L'idée fondamentale du MDE est que le développement de logiciel peut être simplifié par usage de modèles de hauts niveaux d'abstraction (modèles de domaine) représentant un système et par application de transformations automatiques produisant des modèles de bas niveaux d'abstraction (implémentation de modèles) tels que les codes sources et leurs ensembles de tests. On suppose, à ce stade, que procéder avec des modèles de domaine est considérablement plus simple que de procéder avec des artéfacts d'implémentation [Fau 13b].

I.2.2. Principe et concepts fondamentaux

Le MDE est encore un domaine de recherche récent, dynamique et en pleine évolution. Cela se traduit par une pluralité d'idées, de concepts et de terminologies compétitives qui tendent à créer une confusion dans ce domaine. À titre d'exemple, le terme « Ingénierie Dirigée par les Modèles » (IDM) possède plusieurs synonymes : « Model Driven Engineering » (MDE), « Model Driven Development » (MDD), « Model Driven Software Development » (MDSD), etc. [Bon 06].

I.2.2.1. Principe du MDE

L'idée d'utiliser des modèles pour maîtriser la complexité des logiciels existe depuis des décennies, mais dans la pratique, son application n'a été que partielle dans les processus de développement. En l'occurrence, elle a été mise en œuvre pour la structuration et la composition pendant la phase de conception, et pour la vérification pendant les phases de test.

Le principe du MDE consiste à utiliser intensivement et systématiquement les modèles tout au long du processus de développement logiciel. Les modèles devront désormais non seulement être au cœur même du processus, mais également devenir des entités interprétables par les machines. Par analogie au « *tout est objet* » des années 80, on pourrait dire que le principe de base du MDE consiste à dire que « *tout est modèle* ».

Un système peut être vu sous plusieurs angles ou points de vue. Les modèles offrent la possibilité d'exprimer chacun de ces points de vue indépendamment des autres. Cette séparation des différents aspects du système permet non seulement de se dégager de certains détails, mais permet également de réaliser un système complexe par petits blocs plus simples et facilement maîtrisables. Ainsi, on pourrait utiliser des modèles pour exprimer les concepts spécifiques à un domaine d'application, des modèles pour décrire des aspects technologiques, etc., chacun de ces modèles étant exprimé dans la notation et le formalisme les plus appropriés [Bon 06].

I.2.2.2. Modèles et Systèmes

La première notion importante est la notion de modèle. Quelle que soit la discipline scientifique considérée, un modèle est une abstraction d'un système construite dans un but précis. On dit alors que le modèle représente le système. Un modèle est une abstraction dans la mesure où il contient un ensemble restreint d'informations sur un système. Il est construit dans un but précis et les informations qu'il contient sont choisies pour être pertinentes vis-à-vis de l'utilisation qui sera faite du modèle [Mul 06].

I.2.2.3. Caractéristiques des modèles

Selon Stachowiak [Bak 14], 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 fictif.
- **Réduction** : Le modèle constitue une représentation réduite de l'objet ou du phénomène d'origine. Certaines propriétés de ce dernier sont alors omises. 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 où l'élément d'origine ne peut l'être.

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

I.2.2.4. 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, (1) les langages de modélisation dédiés 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 Pétri, qui peuvent être utilisés dans n'importe quel domaine [Bak 14].

I.2.2.5. Modèles contemplatifs et Modèles productifs

Un modèle contemplatif est un modèle interprétable et manipulable par un être humain. Par contre, un modèle productif est un modèle interprétable et manipulable par une machine. L'utilisation de modèles contemplatifs a eu un impact limité sur la production logicielle. L'émergence du MDE offre une nouvelle manière de considérer le modèle dans le cadre d'un développement logiciel, le but n'étant plus de voir le code comme l'élément central de production mais de s'en abstraire au profit des modèles tout en restant productif. Il n'est plus alors question d'un modèle contemplatif mais d'un modèle productif [Pir 14].

I.2.2.6. Méta-modèle

La notion de modèle dans le MDE fait explicitement référence à la définition des langages utilisés pour les construire. En effet, pour qu'un modèle soit productif, il doit pouvoir être manipulé par une machine. Le langage dans lequel ce modèle est exprimé doit donc être explicitement défini. De manière naturelle, la définition d'un langage de modélisation a pris la forme d'un modèle, appelé méta-modèle [Jéz 12].

D'après Miller et Mukerji, en 2003, un méta-modèle est un modèle de modèles [Mon 08]. Il définit, aussi, le langage d'expression de modèles comme un langage de modélisation, dans la mesure où il englobe un ensemble de concepts nécessaires à la description d'une famille de modèles donnée [Jéz 12]. À titre d'exemple, UML est un méta-modèle qui offre des concepts permettant de décrire les différents modèles (Diagramme de classes, Diagramme de cas d'utilisation, …) d'un système [Dia 09].

I.2.2.7. Méta-méta-modèle

S'il est possible de voir le méta-modèle comme un modèle, alors ce modèle peut posséder lui aussi un méta-modèle. Ce méta-modèle du méta-modèle se nomme alors le méta-méta-modèle. Lors de l'émergence de la notion de modélisation et du « *tout est modèle* », il était nécessaire d'établir une base théorique pour la méta-modélisation afin de fédérer la création de méta-modèles et ainsi éviter la prolifération de méta-modèles incompatibles. Afin de limiter les niveaux d'abstraction possibles, la structure hiérarchique d'une modélisation s'organise historiquement autour d'une architecture de 4 niveaux d'abstraction (ou de modélisation) :

- **1er niveau (M0) :** Le système : tel qu'il existe dans le monde réel. Il est représenté par le modèle défini au niveau M1.

- **2ème niveau (M1) :** Le modèle : la représentation du système dans le langage de modélisation donné. Il doit être conforme au méta-modèle défini en M2.
- **3ème niveau (M2) :** Le méta-modèle : la définition du langage de modélisation. Il doit être conforme au méta-méta-modèle défini en M3.
- **4ème niveau (M3) :** Le méta-méta-modèle : il définit l'architecture du méta-modèle. Il a la propriété de se définir lui-même, et c'est l'élément avec le plus haut niveau d'abstraction. En conséquence, il doit être conforme à lui-même [Pir 14]. La représentation la plus classique de cette architecture est donnée dans la figure I.1. La figure I.2 quant à elle, explicite les relations entre un système, un modèle, un langage, et un méta-modèle.

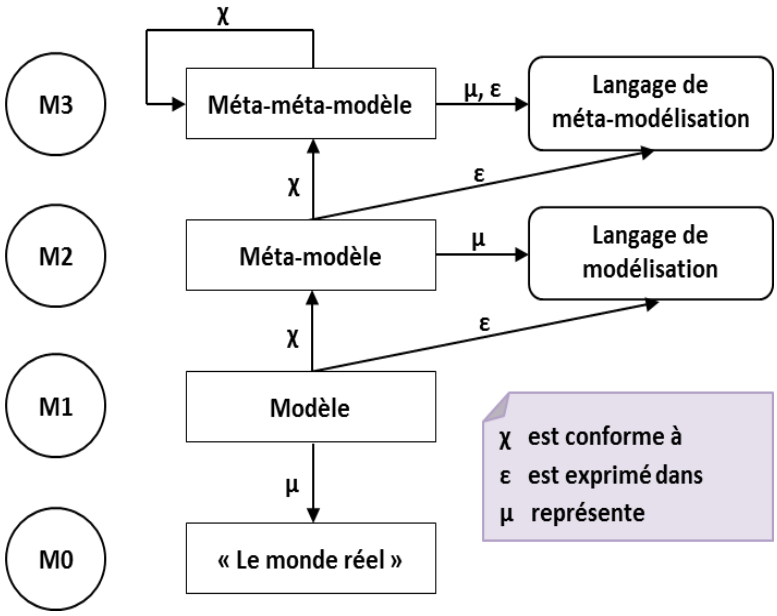


Fig.I.1. Niveaux de modélisation [Tem 12]

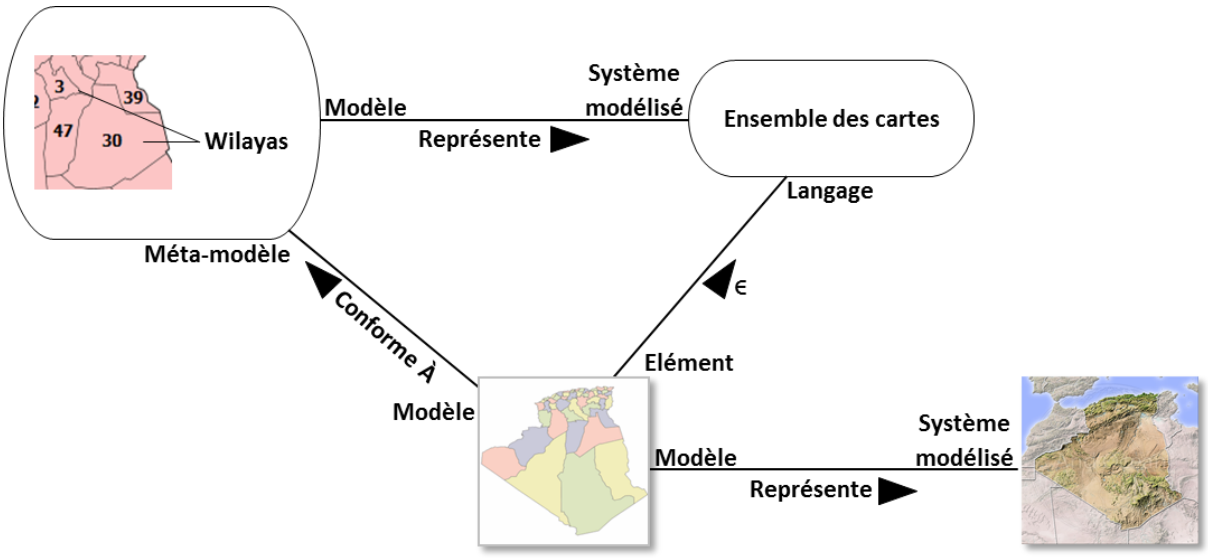


Fig.I.2. Relations entre système, modèle, langage et méta-modèle [Kub 06]

I.3. L'ARCHITECTURE DIRIGÉE PAR LES MODÈLES

Bien que cette structure hiérarchique ne soit pas propre au MDE, le MDE a toutefois été marqué par une approche basée sur ce type de structures. Depuis les années 2000, la prolifération des middlewares dans l'industrie devient problématique. Il était devenu difficile pour une entreprise de se reposer sur une seule plate-forme middleware. Et même lorsqu'une entreprise était capable de s'appuyer sur une unique plate-forme en interne, le besoin d'interopérabilité avec des entreprises partenaires imposait le plus souvent de travailler avec une technologie différente. Afin de pouvoir maîtriser et gérer cette multiplicité de plates-formes middleware, et plus précisément ce besoin d'interopérabilité, l'OMG (Object Management Group), une association industrielle pour la normalisation en génie logicielle, a proposé une approche qui applique les techniques de modélisation et qui s'appelle : « Model Driven Architecture » (MDA). L'objectif du MDA est de pouvoir permettre la création de modèles productifs d'applications, pérennes, stables et indépendants des plates-formes. Il est alors possible de tirer profit de ces modèles afin d'obtenir du code pour des plates-formes ciblées, plateformes qui évoluent continuellement. Le cœur de l'approche MDA se base sur un ensemble de standards créés par l'OMG. Ainsi, son architecture de quatre niveaux se base sur un méta-méta-modèle nommé « Meta Object Facility » (MOF), qui est un standard pour la méta-modélisation. L'approche s'appuie également sur le langage de modélisation UML, dont le méta-modèle est basé sur le MOF, afin de pouvoir exprimer des modèles d'applications indépendants des plates-formes tels que des modèles de conception, et des modèles sur les parties dépendantes des plates-formes, c'est à dire des modèles liés au code. La version finale de cette approche a été publiée en 2003. Elle est toujours d'actualité et il existe de nombreux retours de réussite de son application dans l'industrie [Pir 14].

I.4. DOMAINES D'APPLICATION

La modélisation est présente dans de nombreux domaines (Télécommunication, Aéronautique, Automobile, Robotique, Industrie Spatiale, Architecture, ..). Chaque domaine utilise des modèles et des langages qui lui sont propres, à des fins qui varient grandement : simulation, design, décision, administration, preuve, etc. Puisque la modélisation est au cœur du MDE, il a bien marqué sa présence dans ces domaines, qui font souvent intervenir les systèmes embarqués et les systèmes d'informations.

Le MDE a donné naissance à de grands projets basés sur des collaborations à la fois d'acteurs académiques tels que l'Institut de Recherche en Informatique de Toulouse (IRIT) et l'Institut National de Recherche en Informatique et en Automatique (INRIA), et d'industriels tels qu'Airbus, Thales, Continental, et bien d'autres [Pir 14]. L'INRIA a exploité le MDE dans de nombreux secteurs tels que l'industrie aérospatiale et automobile, ses ingénieurs s'appuyaient sur les langages de modélisation dédiés afin de résoudre les problèmes posés par le développement de logiciels critiques [Inria 10]. Une équipe de chercheurs à l'INRIA a proposé, également, le projet « Gaspard » (et son successeur « Gaspard2 ») afin d'utiliser les avancées méthodologiques du MDE, dans la conception des systèmes embarqués, en général, et des applications de traitement de signal intensif, en particulier [Bon 06]. L'Institut Supérieur de l'Aéronautique et de l'Espace (ISAE), à l'aide du MDE, a construit sa Plateforme pour la Recherche en Ingénierie des Systèmes Embarqués (PRISE) pour simuler ses applications spatiales et avioniques [Hug 14]. De même, Philips s'est bien servie du MDE

dans le domaine médical [Jéz 13]. Thales à son tour emploie largement le MDE pour réaliser ses applications portant sur des domaines critiques tels que le transport terrestre, la sécurité, l'espace et la défense [Le Noir 13]. Dans le domaine des jeux, le MDE a encore fait un grand succès. À titre d'exemple, le jeu « Angry Bird » de Rovio, qui dépasse les 500 Millions $ de revenues [Jéz 13].

I.5. TRANSFORMATION DE MODÈLES

Cette partie présente les concepts de base permettant de réaliser une transformation de modèles, nous nous intéressons, donc, à tous ses principes.

I.5.1. Définitions de la TM

Les transformations de modèles sont au cœur de l'ingénierie dirigée par les modèles. La TM a déjà fait l'objet d'une littérature conséquente, il s'agit d'un concept clé d'une démarche MDE [Tru 11]. D'un point de vue formel, la TM a été définie par D'Antonio, en 2005, de la manière suivante : soit A un modèle et t : MOD → MOD une fonction (MOD désigne l'ensemble des modèles), t est alors une fonction de TM. B = t(A) est donc le modèle A transformé par t (mathématiquement, B est l'image de A par la fonction t) [Tru 11].

La TM est un domaine jeune où il y'a plusieurs concurrents, mais qui se chevauchent partiellement les définitions des termes. Tratt [Bie 10] définit la TM très largement comme « un programme qui mute un modèle en un autre ». L'OMG la définit dans le contexte du MDA (Model Driven Architecture) comme « le processus de conversion d'un modèle en un autre modèle du même système ». Kleppe et al. [Bie 10] la définissent comme « la génération automatique d'un modèle cible à partir d'un modèle source, selon une description de transformation ». Mens et al. [Bie 10] étendent cette définition à plusieurs modèles en entrée ou en sortie et définissent la TM comme « la génération automatique d'un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources, selon une description de transformation ». La définition la plus générale et qui fait l'unanimité au sein de la communauté MDE, est celle de Bézivin en 2004, et qui dit qu'une TM est la génération d'un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources [Bie 10].

Dans l'approche par modélisation, cette transformation se fait par l'intermédiaire de règles de transformation (RTs) qui décrivent la correspondance entre les entités du modèle source et celles du modèle cible. En réalité, la transformation se situe entre les méta-modèles source et cible. Le processus de TM prend en entrée un ou plusieurs modèles sources et crée en sortie un ou plusieurs modèles cibles. La figure I.3, illustre le mécanisme de cette transformation. Cette transformation met en jeu deux étapes. La 1ère étape permet d'identifier les correspondances entre les concepts des modèles source et cible au niveau de leurs méta-modèles, ce qui induit l'existence d'une fonction de transformation applicable à toutes les instances du méta-modèle source. La 2ème étape consiste à appliquer la transformation du modèle source afin de générer automatiquement le modèle cible par un programme appelé moteur de transformation ou d'exécution [Dia 09].

I.5.2. Règles de Transformation

Les règles de transformation (RTs) ont une place prépondérante dans le mécanisme de TM. D'après Levendovski et al. en 2002, ces RTs sont assimilées à la notion de « mapping ». En 2008, Deguil propose quatre types de mappings de modèles : les mappings de type « 1-1 »

utilisent des relations qui associent un élément d'un modèle source à un et un seul élément d'un modèle cible. Les mappings de type « 1-m » qui peuvent associer un élément du modèle source à plusieurs éléments du modèle cible. Les mappings de type « n-1 » peuvent associer plusieurs éléments du modèle source à un seul élément du modèle cible. Les mappings de type « n-m » associent plusieurs éléments du modèle source à plusieurs éléments du modèle cible [Tru 11]. Ces types de mappings sont exposés à travers la figure I.4.

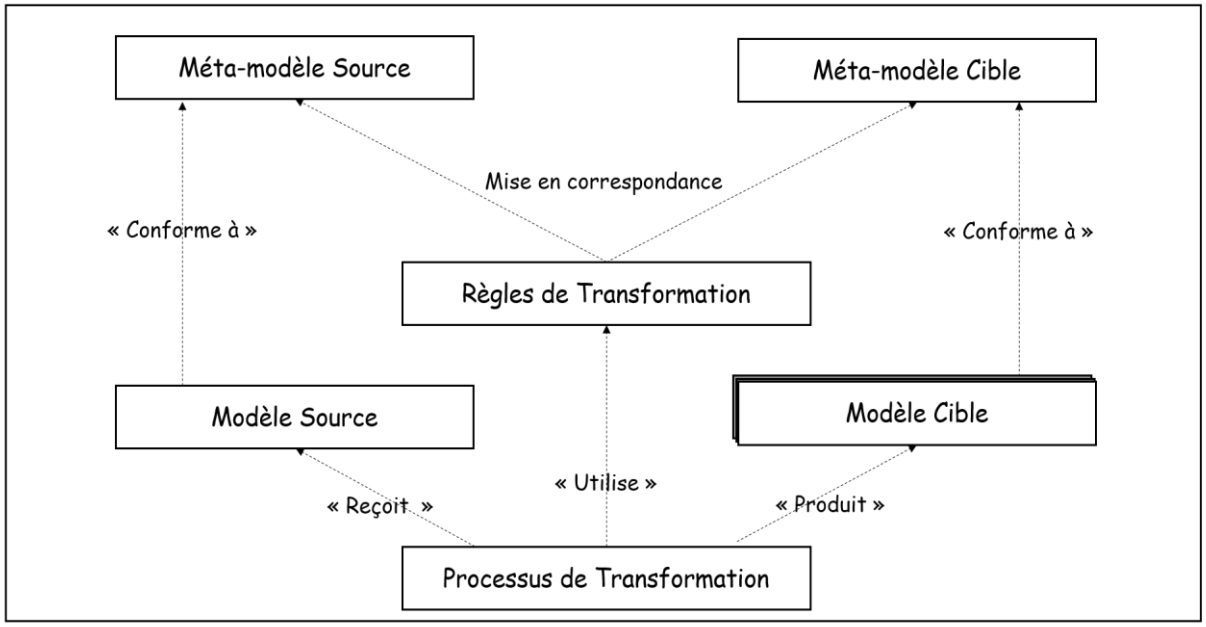


Fig.I.3. Processus de transformation de modèles [Tru 11]

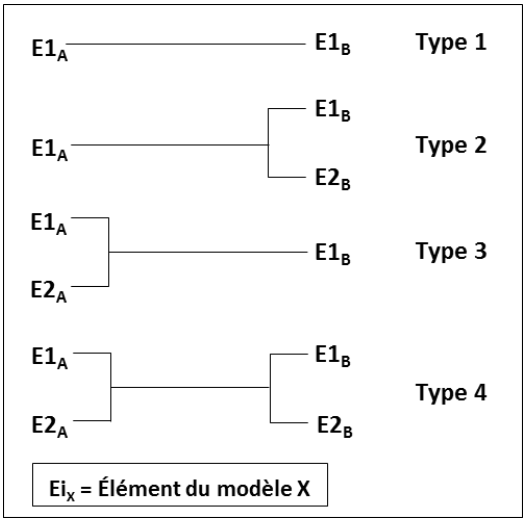


Fig.I.4. Différents types de mappings (Selon Deguil, en 2008) [Tru 11]

I.5.3. Objectifs de la TM

Les deux originalités du MDE sont la représentation des systèmes par des modèles et les TMs. Un modèle a pour but d'une part d'éditer des vues et d'autre part de générer du code ou de la documentation. De plus, en réalité, on a souvent besoin d'analyser des modèles édités. Et pourtant, il y a beaucoup de points de vue lors du développement, c'est-à-dire, qu'il peut y avoir beaucoup de modèles qui représentent un système. Par exemple, un modèle peut être

représenté sous forme d'un modèle de cas d'utilisation, d'un modèle de composants, d'un modèle de diagramme d'activité, etc. Pour que les modèles soient bien exploitables par des machines, il est nécessaire d'avoir des programmes de TM qui formalisent des liens explicites entre les divers modèles pour assurer la cohérence entre ces modèles. En partant des besoins de production des systèmes dirigés par les modèles ou de support de l'interopérabilité entre les systèmes, on a besoin de TMs telles que le raffinement, le refactoring ou la projection des modèles [Thi 10].

I.5.4. Taxonomie des Transformations de modèles

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 taxonomie qui permet de définir plusieurs termes s'apprêtant à la TM, qui seront utilisés dans la suite de ce mémoire.

I.5.4.1. Transformation horizontale de modèles

Si les modèles source et cible appartiennent au même niveau d'abstraction, on parle de TM horizontale. Un exemple de ce type est le refactoring, où le modèle cible, par rapport au modèle source, change dans sa structure interne sans changer de comportement [Rag 11]. Un autre exemple, est le renommage d'un élément dans le modèle, ce qui ne change rien à son comportement (voir la figure I.5).

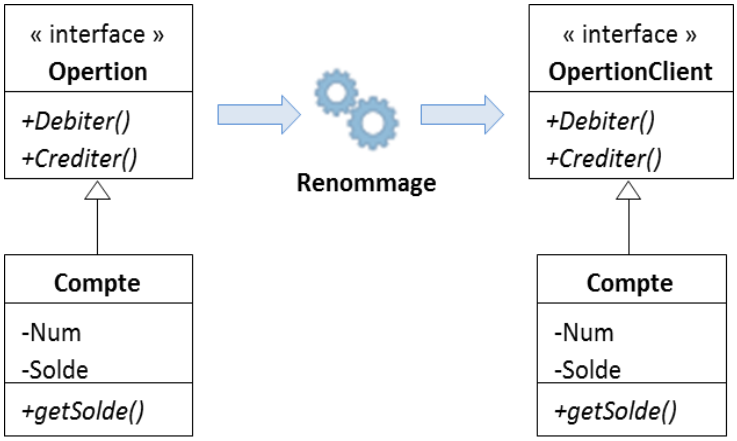


Fig.I.5. Renommage de l'interface *« Operation »* [Rag 11]

I.5.4.2. Transformation verticale de modèles

Contrairement à la TM horizontale, différents niveaux d'abstraction sont utilisés dans le modèle source et cible. Un exemple de cette transformation est le raffinement [Rag 11].

I.5.4.3. Transformation endogène de modèles

Une TM endogène affecte les modèles exprimés dans le même langage. Les deux modèles source et cible sont conformes au même méta-modèle. L'optimisation telle que la fusion ou l'élimination du code mort ou encore le refactoring, en sont des exemples [Rag 11].

I.5.4.4. Transformation exogène de modèles

Contrairement aux TMs endogènes, les TMs exogènes sont exprimées entre des modèles conformes à différents méta-modèles. Les transformations exogènes sont également appelées « translation ». Un exemple de transformations exogènes est la génération du code

ou de la documentation, ou le reverse engineering par décompilation. Par exemple, un diagramme de classes UML peut être traduit en code Java. La traduction d'un code Java en diagramme de classes UML est un exemple pour le reverse engineering [Rag 11]. Ces familles de transformations sont résumées dans la figure I.6.

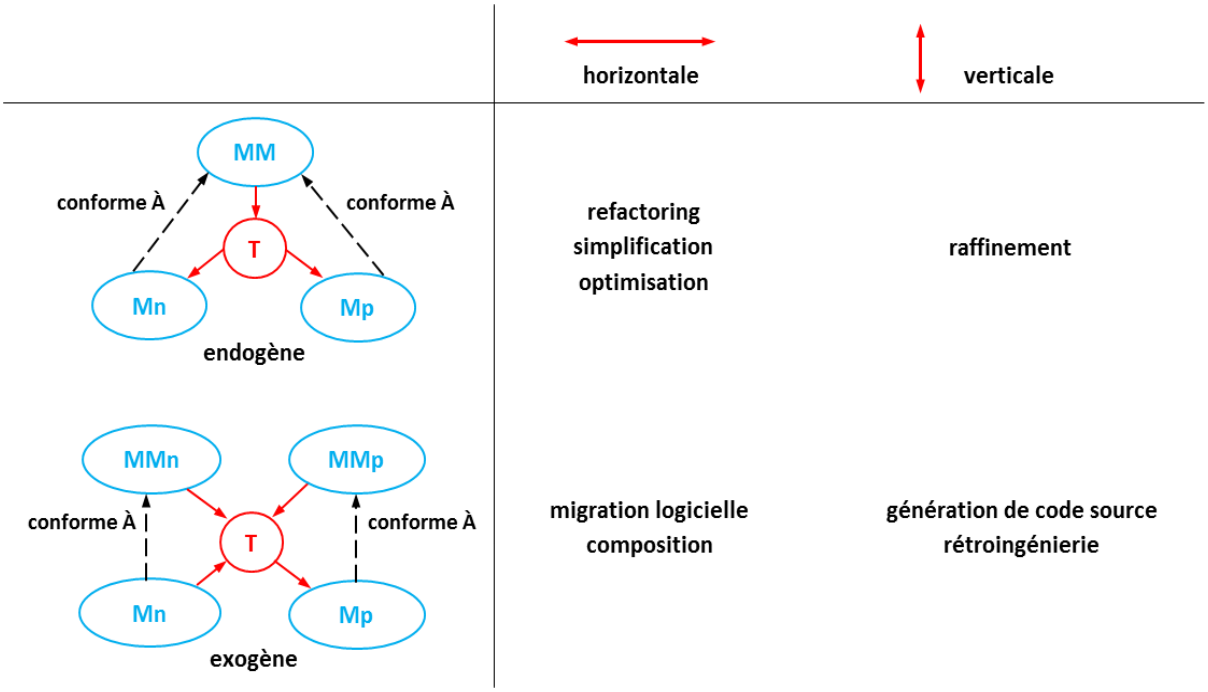


Fig.I.6. Taxonomie des transformations de modèles [Pal 12]

I.5.4.5. 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 [Bak 14].

I.5.5. Langages pour la TM

De nombreux langages sont disponibles pour décrire les TMs :

- **ATLAS Transformation Language (ATL)** est un langage hybride (déclaratif et impératif) qui permet de définir une TM sous la forme d'un ensemble de règles. Il est défini par un modèle MOF (Meta Object Facility) pour sa syntaxe abstraite et possède une syntaxe concrète textuelle. Pour accéder aux éléments d'un modèle, ATL utilise des requêtes sous forme d'expressions OCL (Object Constraint Language) [Rag 11].

- **SmartQVT** est une implémentation du langage standardisé QVT-Operational qui permet la TM à l'aide de requêtes. SmartQVT est un langage impératif de transformation. Il permet d'expliciter le détail des opérations de transformation [Rag 11].

- **Kermeta** est un langage open-source pour la modélisation et la méta-modélisation, développé par l'équipe Triskell à l'Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA). Il a été conçu comme extension d'EMOF (Essential MOF), en ajoutant notamment la possibilité d'exprimer la sémantique et le comportement des méta-modèles avec un langage d'action impératif et orienté-objet. Son approche impérative pour modéliser les résultats de transformation est différente par rapport à ATL et SmartQVT, qui suivent

principalement le paradigme déclaratif. Au lieu des règles, Kermeta utilise des opérations, qui sont fondamentalement très similaires à des opérations ou méthodes dans les langages de programmation orientés objet, tels que Java [Rag 11].

I.6. TRANSFORMATION DE MODÈLES PAR L'EXEMPLE

Nous divisons la TM par l'exemple en MTBE et MTBD, qui sont similaires à part le fait que l'utilisateur doit intervenir dans la 2ème pour démontrer comment les transformations auront lieu.

I.6.1. MTBE

Le MTBE définit une TM par le biais d'un ensemble d'exemples de cette TM. Les exemples sont donnés sous forme de paires où chacune a un modèle en entrée et son modèle transformé correspondant avec les traces de cette transformation. Les RTs sont alors automatiquement extraites à partir de ces exemples [Saad 12]. Au lieu d'écrire les RTs manuellement, le MTBE permet aux utilisateurs de définir un ensemble prototype de mappings entre des instances de modèles sources et cibles en relation. À partir de ces mappings, les RTs peuvent être générées d'une manière semi-automatique.

Varró [Var 06] propose l'usage de la programmation logique inductive comme méthode pratique et efficace pour l'implémentation de MTBE. L'idée de base consiste à représenter les mappings initiaux sous forme de clauses et d'inférer ensuite les RTs par usage d'un moteur d'inférence dédié. D'une manière similaire, Strommer et Wimmer [Str 08] implémentent sous Eclipse un prototype de génération de RTs à partir de mappings sémantiques entre les modèles du domaine. Au lieu d'utiliser la programmation logique, leur inférence et leur raisonnement sont basés sur un appariement de patterns entre modèles sources et cibles.

Afin d'illustrer le MTBE, considérons le cas bien connu des transformations des diagrammes de classes UML en schémas relationnels. Les exemples sont sous forme de modèle UML en entrée et le modèle relationnel correspondant en sortie. Les transformations indiquent les liens entre les éléments du diagramme UML et les éléments du modèle relationnel. Par exemple, un lien est donné pour spécifier qu'une classe « Client » est transformée en table « Client » et respectivement pour chaque champ. Un lien est équivalent à une trace d'exécution de la transformation appropriée. Deux éléments sont liés par un lien de transformation si l'information contenue dans le 1er est nécessaire à la construction du 2ème élément [Saad 12].

I.6.2. MTBD

Au lieu de l'inférence de règles du MTBE, le MTBD demande aux utilisateurs de démontrer comment la TM doit se faire par une édition directe (ajout, suppression, mise à jour, etc.) de l'instance du modèle pour simuler le processus de la TM pas à pas. Un moteur d'inférence et d'enregistrement capture toutes les opérations des utilisateurs dans une tâche de transformation afin de déduire l'intention d'un utilisateur. Un pattern de transformation est généré de cette inférence pour associer une pré-condition à la séquence d'opérations nécessaire à la réalisation de la transformation. Ce pattern peut être réutilisé pour faire correspondre automatiquement une pré-condition à une nouvelle instance de modèle et refaire les mêmes opérations pour simuler le processus de transformation. Malheureusement, le

MTBD exige un grand nombre de patterns pour produire des résultats cohérents. En plus, il ne peut pas être utile pour transformer entièrement un modèle source [Kes 10].

I.6.3. Principes du MTBE

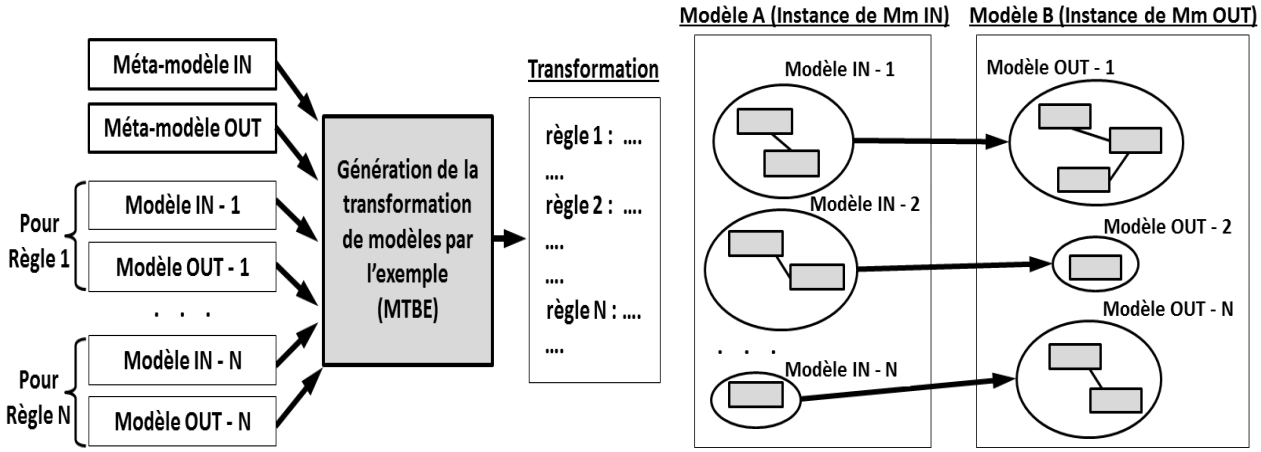
Les étapes courantes de cette transformation sont comme suit :

1. Initialisation manuelle de mappings entre modèles sources et cibles : Le concepteur de TMs doit rassembler un ensemble initial de paires de modèles (sources**,** cibles) respectifs et en correspondance.

2. Dérivation automatique des RTs : Basée sur les mappings de l'étape 1, la plateforme doit synthétiser l'ensemble des RTs (Figure I.7 (a)). Ces règles devront transformer correctement, au moins les modèles prototypes sources en modèles cibles équivalents (Figure I.7 (b)).

3. Raffinement manuel des RTs : Le concepteur peut raffiner manuellement les RTs à tout moment. Toutefois, le MTBE recommande que ces modifications doivent être incluses dans les paires de modèles, de sorte que les altérations ne soient pas réécrites la prochaine fois que la TM sera générée.

4. Exécution automatique des RTs : Le concepteur valide la fiabilité des RTs synthétisées en les exécutant sur un ensemble additionnel de paires de modèles (sources, cibles) pris comme ensemble de test [Gar 08].



(a) Les entrées et les sorties du MTBE **(b) Le comportement de la transformation générée**

Fig.I.7. Description du processus de MTBE [Gar 08]

I.7. CONCLUSION

Ce chapitre a permis d'expliciter en quoi consiste le paradigme MDE à travers ses principes et concepts de base, tout en faisant le tour d'horizon des domaines où il a bien marqué sa présence. En mettant le point sur la TM qui est au cœur du MDE, une taxonomie a été présentée afin de distinguer entre la transformation exogène et endogène, horizontale et verticale. Les types de RTs ont été également passés en revue. Une partie du chapitre a été consacrée pour illustrer le processus de TM par l'exemple, et pour bien distinguer entre MTBE et MTBD. La génération automatique des RTs par le MTBE suit des principes communs représentés, vers la fin, par une suite d'étapes qui englobe ce processus.

CHAPITRE II : TRANSFORMATION DE MODÈLES PAR L'EXEMPLE

II.1. INTRODUCTION

L'automatisation des TMs n'est pas une tâche facile. En effet, pour écrire les transformations, le spécialiste en MDE doit avoir une connaissance approfondie des méta-modèles source et cible ainsi que l'équivalence sémantique entre eux. L'idée principale derrière les approches guidées par les exemples, est de procéder à un apprentissage supervisé des TMs afin d'aboutir à une génération automatique de la solution à partir du modèle source.

Une des questions importantes dans la maintenance des logiciels est de proposer des outils automatisés qui améliorent la qualité de ces logiciels, d'où l'objectif principal des transformations endogènes.

Le refactoring des modèles, étant la transformation endogène la plus connue, et l'activité la plus importante durant tout le processus de maintenance, constitue l'une des TMs dont l'automatisation est le plus potentiellement bénéfique.

Ce chapitre est divisé en quatre parties. D'abord, nous discutons la relation entre l'automatisation en MDE et les approches guidées par les exemples. Ensuite nous présentons l'état de l'art des approches existantes en MTBE. Puis, nous nous intéressons aux transformations endogènes, et enfin, nous focalisons sur le refactoring des modèles, qui fait l'objet de ce projet, en introduisant quelques-unes de ses notions fondamentales et caractéristiques. Nous dressons, également, un état de l'art des travaux existants dans ce domaine, tout en profitant de cette occasion, pour mettre l'accent sur le langage UML et, en particulier, les diagrammes de classes. Ces derniers sont souvent les plus concernés par le refactoring, et auxquels, nous nous intéressons dans le cadre de ce travail.

II.2. L'AUTOMATISATION EN MDE

II.2.1. L'Automatisation en génie logiciel

Les systèmes logiciels d'aujourd'hui sont considérablement grands, complexes et critiques. Ces systèmes ne peuvent pas être développés et ne peuvent pas évoluer d'une manière économique et en temps optimal sans automatisation. Le génie logiciel automatisé applique des calculs à ses activités. Le but est d'automatiser partiellement ou totalement ces activités, augmentant ainsi considérablement la qualité et la productivité [Kes 10].

II.2.2. L'amélioration de l'automatisation en MDE

L'automatisation est la technique, la méthode ou le système d'exploitation ou de contrôle d'un processus par des moyens hautement automatiques, comme par le biais de dispositifs électroniques, réduisant l'intervention humaine au minimum. Ainsi l'automatisation a lieu lorsque les activités humaines (en particulier les activités productives) sont remplacées par des dispositifs électroniques contrôlés automatiquement.

Selon Sheridan [She 92], le degré d'automatisation, est la relation entre les tâches d'un processus qui sont attribuées à des machines et celles attribuées aux humains. Wei et al. [Wei 98] étendent cette définition pour considérer la difficulté de la tâche. Pour ces auteurs, les tâches peuvent être le processus effectué pour transformer une entrée en une sortie ou les activités de contrôle nécessaires pour assurer le bon fonctionnement du système [Fau 13].

Par conséquent, dans le cas de MDE, améliorer le degré d'automatisation consiste à remplacer les tâches du MDE, qui sont effectuées manuellement, par les processus automatiques (par ordinateur). Dans le cas de tâches MDE qui sont déjà effectuées automatiquement, l'amélioration de l'automatisation consiste à les remplacer par de nouveaux processus qui nécessitent moins d'interactions humaines.

MDE est un paradigme qui promet de réduire la complexité du logiciel par l'utilisation intensive de modèles et des transformations automatiques entre modèles (TMs). D'une façon simplifiée, dans la vision du MDE, les spécialistes utilisent plusieurs modèles pour représenter un logiciel, et ils produisent le code source en transformant automatiquement ces modèles. En effet, les développeurs de logiciels devront produire manuellement les modèles de domaine et en dériver manuellement les artefacts d'implémentation. Ils auront aussi à les garder manuellement à jour et à gérer la cohérence entre eux après toute modification. L'expérience a montré que ces tâches manuelles entraînent un écart important entre les représentations de haut niveau du logiciel et leurs correspondants de bas niveau. Conséquemment, l'automatisation est un facteur clé et un principe fondateur de MDE [Fau 13].

Bien sûr, MDE n'est pas censé être une approche entièrement automatisée. Les tâches manuelles sont impossibles à éviter. À un certain moment, un être humain doit valider le résultat produit automatiquement ou organiser l'ensemble du processus. Enfin, c'est à l'humain de définir, par exemple, quelles sont les TMs intéressantes à effectuer ou les systèmes intéressants à modéliser. Dans ce sens, l'automatisation en MDE ne prétend pas remplacer le rôle de l'humain, mais éviter d'effectuer manuellement toutes ces tâches qui, en raison de leurs natures (complexité, coût, sujettes à l'erreur et répétitives), sont mieux faites automatiquement [Fau 13].

Toutefois, l'automatisation d'une activité engendre toujours un coût. Tout d'abord, la bonne connaissance de la façon dont l'activité devrait être effectuée doit être recueillie. De plus, cette connaissance doit être convertie en algorithmes cohérents et évolutifs. Finalement, ces algorithmes doivent être globalement validés sur un ensemble de tests qui considère la plupart des cas communs, mais aussi des cas particuliers.

II.2.3. Techniques d'automatisation et les approches guidées par les exemples

Pour améliorer l'automatisation, des techniques appropriées doivent être choisies. Ce choix dépend du contexte. Si l'algorithme d'automatisation est connu et performant, un programme l'implémentant suffit, comme dans le cas de tri. Mais il y'a des situations où l'algorithme est inconnu ou non évolutif. Dans ces cas, des techniques comme les méta-heuristiques, les algorithmes de recherche, et les algorithmes d'apprentissage machine peuvent être utilisées.

Ces techniques peuvent être combinées avec des exemples pour résoudre des problèmes spécifiques par analogie avec les exemples ou, même, pour apprendre une bonne solution générale et optimale à partir des exemples. Cette idée a été largement appliquée en apprentissage machine pour automatiser la classification d'objets.

Les exemples ont également été utilisés en MDE pour apprendre les TMs. Certainement, pour certains types spécifiques de modèles, l'algorithme de transformation est

bien connu (ex : transformation de diagrammes de séquence aux modèles de réseaux de Petri). Mais, en général, l'algorithme pour transformer un modèle d'un type arbitraire à son modèle correspondant d'un autre type arbitraire est inconnu. Pour ce cas général, des approches utilisant des modèles exemples (paires de modèles sources et cibles) ont été proposées.

Que l'approche soit destinée à résoudre un problème spécifique suivant des exemples ou à abstraire une solution générale à partir d'exemples, un mécanisme pour traiter ces exemples et en obtenir des informations ou des connaissances, est nécessaire [Fau 13].

II.3. ÉTAT DE L'ART

Dans cette section, nous présentons l'état de l'art des approches de transformation de modèles par l'exemple. Nous divisons les travaux existant entre les approches MTBE et les approches MTBD. Pour les approches MTBE, nous les divisons entre celles qui génèrent des RTs opérationnelles et celles qui dérivent des mappings abstraits ou qui transforment directement un modèle mais ne produisent aucun type de connaissances sur les transformations.

II.3.1. MTBE

II.3.1.1. Règles de Transformation (RTs)

En 2006, Varró [Var 06] a proposé une première approche de MTBE. Dans son travail, il dérive des RTs à partir d'un ensemble d'exemples prototypiques de transformations avec des modèles inter-reliés. Les exemples sont fournis par l'utilisateur. Ce processus semi-automatique et itératif commence par analyser les mappings entre les modèles sources et cibles des exemples avec leurs méta-modèles respectifs. Les RTs sont finalement produites à l'aide d'un algorithme ad-hoc. L'approche de Varró peut dériver seulement des RTs 1-1, ce qui est une limitation importante. Les RTs dérivées par cette approche, peuvent tester la présence ou l'absence d'éléments dans les modèles sources. Ce niveau d'expressivité est cependant insuffisant pour de nombreux problèmes de transformation courants (ex : transformation de diagramme de classes UML au diagramme relationnel). Bien que les modèles sources et cibles des exemples peuvent être recueillis dans les transformations manuelles passées, fournir l'équivalence sémantique fine entre les constructions contenues dans ces modèles est difficile à garantir. Enfin, Varró n'a pas publié les données de validation. Par conséquent, on ne sait pas comment l'approche se comporte dans un scénario réaliste.

En 2007, Wimmer et al. ont proposé dans [Wim 07] une approche qui produit des RTs 1-1 basée sur des exemples de transformations et sur leurs traces. Le processus de dérivation est similaire à celui proposé par Varró dans [Var 06], à la différence que Wimmer produit des RTs exécutables écrites en ATL (ATLAS Transformation Language**)**. La production des RTs 1-1 est une limitation importante. Là encore, l'absence d'une étape de validation ne permet pas d'évaluer comment l'approche se comporte dans des situations réalistes.

En 2008, Strommer et al. [Str 08] étendent l'approche de Wimmer en permettant des RTs 2-1 dans un processus de dérivation basé sur les correspondances des patterns. Comme les approches précédentes, cette contribution n'a pas été validée dans des contextes concrets.

En 2009, Balogh et al. [Bal 09], améliorent le travail original de Varró en utilisant la programmation logique inductive (ILP) à la place de l'heuristique ad-hoc originale. Cependant, comme pour l'approche de Varró, l'idée principale est de dériver des RTs en

utilisant les mappings de transformations. Cette approche produit des RTs n-m à partir de clauses Prolog qui sont obtenues par un processus semi-automatique dans lequel l'utilisateur doit ajouter des assertions logiques jusqu'à ce que le moteur d'inférence ILP puisse produire la RT souhaitée. En plus de la nécessité de donner les mappings détaillés, le fait que l'utilisateur doit interagir avec le moteur d'inférence ILP est une limitation de cette approche. Comme Varró, Balogh et al. produisent également des conditions de RTs de faible expressivité (seulement capables de tester la présence et l'absence d'éléments sources). Balogh et al. prétendent être en mesure de produire l'ensemble des RTs nécessaires pour transformer un diagramme de classes UML en un schéma relationnel. Malheureusement, l'ensemble des RTs obtenues n'est pas publié. Le document manque également une étape de validation, ce qui rend très difficile d'évaluer l'utilité de l'approche proposée.

En 2009, Garcia-Magariño et al. [Gar 09] proposent un algorithme capable de créer des RTs n-m à partir des exemples de modèles inter-reliés et leurs méta-modèles. Les RTs sont créées dans un langage de TM générique, et elles sont, plus tard, converties en ATL (ATLAS Transformation Language**)** à des fins d'évaluation. Le papier présente une étape de validation mais il n'évalue pas la qualité des RTs produites. La nécessité de traces de transformations limite l'applicabilité de cette approche.

En 2010, Kessentini et al. [Kes 10b] ont proposé une approche pour dériver des RTs 1-m à partir des exemples de transformations. Le processus de dérivation de RTs est établi par la production de mappings 1-m entre les éléments du méta-modèle source et les éléments du méta-modèle cible. Cette contribution n'utilise pas de traces et produit réellement les RTs, mais l'absence de RTs n-m reste toujours une limitation. L'approche est basée sur une recherche méta-heuristique et hybride qui utilise l'optimisation d'un essaim de particules (PSO) combinée à la technique de recuit simulé. Les auteurs effectuent également une étape de validation, mais elle a les mêmes problèmes de validité que l'approche précédente.

En 2012, Saada et al. [Saad 12], étendent le travail de Dolques et al. [Dol 10] en proposant un processus de dérivation de RTs de deux étapes. Dans la première étape, l'approche de Dolques est utilisée pour apprendre les patterns de transformations à partir des exemples et des traces de transformations. Dans la deuxième étape, les patterns appris sont analysés et ceux considérés comme pertinents sont sélectionnés. Les patterns sélectionnés sont alors traduits en RTs sous JESS (Java Expert System Shell). Comme dans [Dol 10], les RTs produites sont de type n-m. L'approche est finalement testée dans un cadre expérimental de dix groupes (ten-fold) où la précision et le rappel sont calculés.

En 2013, Faunes et al. [Fau 13b] ont proposé une approche basée sur la programmation génétique pour apprendre automatiquement les règles à partir de transformations préalables de paires de modèles source-cible utilisées comme exemples. L'approche peut produire des RTs n-m, ce qui est une amélioration remarquable par rapport à la plupart des autres contributions. Ces RTs sont exécutables, ce qui n'est pas le cas de l'approche de Balogh et al. Les RTs dérivées, écrites en JESS, recherchent des patterns non triviaux dans les modèles sources et instancient des patterns non triviaux dans les modèles cibles. Contrairement à l'approche de Garcia-Magariño et al., celle-ci n'a pas besoin de traces fines de transformations. Elle est donc applicable à un large spectre de problèmes de transformation. Cette approche a été évaluée quantitativement et qualitativement sur des transformations de modèles structurels et à

contraintes de temps. Les résultats ont montré que les RTs produites sont opérationnelles et correctes.

II.3.1.2. Mappings et autres formes de transformations

En 2008, Kessentini et al. [Kes 08] proposent une autre approche de TMs qui, à partir d'exemples de transformations et leurs traces, transforme un modèle source en un modèle cible. Cette contribution diffère des précédentes car elle ne produit pas de RTs. Au lieu de ça, elle dérive directement le modèle cible requis par analogie avec les exemples existants. Le processus de transformation est effectué en produisant des mappings n-m entre les éléments du modèle source et les éléments du modèle cible. Ces mappings sont dérivés par une recherche heuristique et hybride qui trouve d'abord une solution initiale en effectuant une recherche globale, puis améliore cette solution en effectuant une recherche locale (en utilisant le recuit simulé). Ces recherches heuristiques sont guidées par une fonction « Objectif » qui mesure la similarité entre la solution et la base d'exemples. Le fait que cette approche ne produit pas de RTs pourrait être une limitation. Lors de l'obtention d'une transformation incomplète pour un modèle donné, l'utilisateur doit corriger manuellement et compléter le modèle cible. Bien que cette approche ait été validée sur des données industrielles, le processus d'évaluation présente, quand même, des problèmes de validité.

En 2010, Dolques et al. [Dol 10], proposent une approche de MTBE basée sur l'analyse relationnelle de concepts (RCA). Le processus de dérivation à base de RCA analyse un exemple de transformation unique et ses mappings, ainsi que les méta-modèles sources et cibles. Il produit des ensembles de mappings récurrents n-m organisés en un treillis de Gallois. Lorsque les mappings ne sont pas disponibles, Dolques et al. proposent une technique pour produire un alignement entre les modèles sources et cibles des exemples. Cet alignement est basé sur les identifiants, ce qui compromet la qualité des mappings résultants, comme mentionné par les auteurs. Puisque cette approche de dérivation de RTs n'a pas été validée sur des données concrètes, il est difficile de déterminer à quel point elle est efficace. En outre, la nécessité de traces de transformations constitue également une limite.

II.3.2. MTBD

En 2009, Sun et al. [Sun 09] ont proposé une nouvelle perspective pour la dérivation des transformations, à savoir, le MTBD. Dans ce travail, l'objectif est de généraliser les cas de TMs. Cependant, au lieu d'utiliser les exemples, les utilisateurs sont invités à démontrer comment la TM devrait être faite. Les actions d'édition de l'utilisateur sont enregistrées et servent de correspondances et de patterns de transformation qui peuvent être appliqués ultérieurement sur un modèle similaire (ou le même) en effectuant un processus d'appariement de patterns. Du point de vue de l'expressivité, les patterns de transformation produits peuvent contenir des actions de transformation sophistiquées comme la concaténation des chaînes de caractères et les opérations mathématiques sur les propriétés numériques ce qui constitue une amélioration des approches précédentes. Cependant, le fait que cette approche vise à effectuer uniquement une transformation endogène, représente une limite importante. Une autre limitation primordiale est que les patterns de transformation générés ne sont pas réellement des RTs. Puisque Sun et al. n'ont pas effectué de validation, on n'a pas suffisamment d'éléments pour juger la performance de leur approche sur un scénario réaliste.

En 2010, Langer et al. [Lan 10] proposent une approche de MTBD, très similaire à celle de Sun et al. avec l'amélioration de prise en charge des transformations exogènes. La limitation principale de cette approche est qu'elle ne produit pas de RTs, mais plutôt des patterns de transformation. Comme pour de nombreuses approches, l'absence d'une validation rigoureuse, rend difficile d'évaluer la performance de cette approche dans des scénarios réalistes.

Les approches de transformation de modèles par l'exemple, que nous avons citées dans cet état de l'art, sont résumées dans la figure II.1.

II.3.3. Discussion

Les approches de MTBE existantes ne résolvent le problème de dérivation de RTs que partiellement. La plupart d'entre elles nécessitent des mappings détaillés (traces de transformations) entre les modèles sources et cibles des exemples, qui sont difficiles à fournir dans certaines situations. D'autres ne peuvent guère dériver des RTs qui testent de nombreuses constructions dans le modèle source et/ou produisent de nombreuses constructions dans le modèle cible. Or les RTs n-m, aussi complexes soient elles, sont une nécessité dans les problèmes de transformation complexes. Une troisième limitation est l'incapacité de certaines approches de produire automatiquement des conditions complexes de RTs pour définir des patterns précis à rechercher dans le modèle source. Enfin, certaines approches produisent, des RTs non-exécutables et abstraites qui doivent être complétées et translatées manuellement à un langage exécutable.

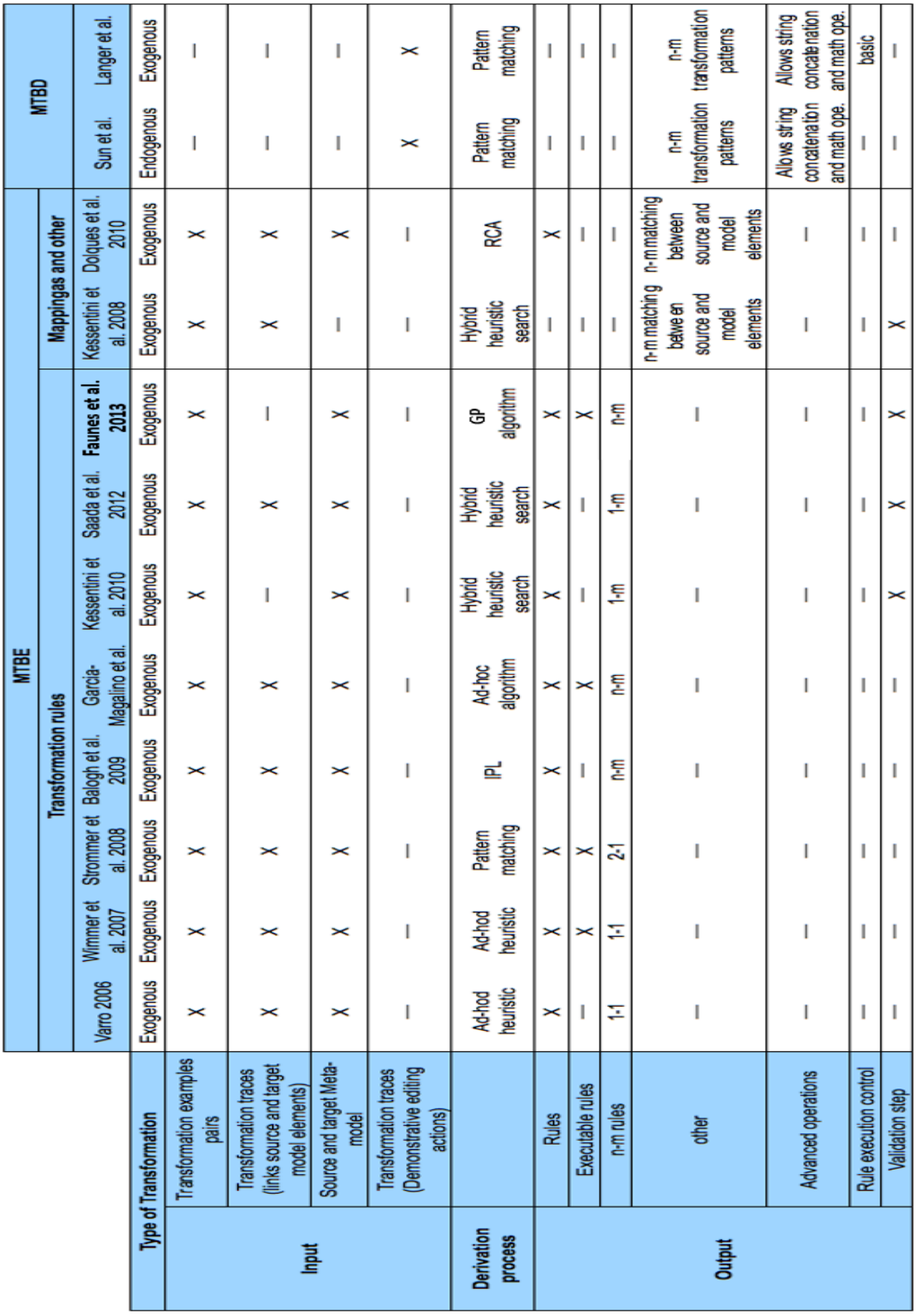
En proposant une approche basée sur la GP, Faunes et al. [Fau 13b] ont réussi à remédier à ces limitations. Contrairement aux approches courantes, leur approche n'a pas besoin de traces de transformations fines pour produire des RTs n-m. Elle est donc applicable à un large éventail de problèmes de transformation. Puisque les RTs apprises sont produites directement dans un langage de transformation proprement dit, elles peuvent être facilement testées, améliorées et réutilisées. L'approche a été évaluée avec succès sur des problèmes bien connus de transformation, ce qui n'est pas le cas pour de nombreuses approches.

Faunes et al. [Fau 13b], dans leur travail, focalisent uniquement et surtout sur la transformation exogène. La transformation endogène semble être volontairement délaissée ce qui pourrait créer une limitation dans leur travail. Selon l'état de l'art précédent, à part les contributions de Sun et Langer, aucune autre contribution ne prétend prendre en charge la transformation endogène. Malheureusement, ces dernières ont beaucoup de limitations tel que mentionné précédemment, entre autres, la connaissance sur la façon dont la transformation est effectuée est incomplète ou non disponible, ce qui constitue la motivation de notre travail.

II.4. LA TRANSFORMATION ENDOGÈNE

II.4.1. Principe et avantages de la transformation endogène

Les transformations endogènes sont principalement liées aux activités de maintenance (refactoring, performances, etc.). Dans le développement logiciel moderne, la maintenance engendre la majorité du coût total et des efforts dans un projet logiciel.



Type of Transformation		MTBE												MTBD	
		Transformation rules									Mappings and other				
		Varro 2006	Wimmer et al. 2007	Strommer et al. 2008	Balogh et al. 2009	Garcia-Magalino et al.	Kessentini et al. 2010	Saada et al. 2012	Faunes et al. 2013		Kessentini et al. 2008	Dolques et al. 2010		Sun et al.	Langer et al.
Input	Transformation examples pairs	Exogenous	Exogenous	Exogenous	Exogenous	Exogenous	Exogenous	Exogenous	Exogenous		Exogenous	Exogenous		Endogenous	Exogenous
		X	X	X	X	X	X	X	X		X	X		—	—
	Transformation traces (links source and target model elements)	X	X	X	X	X	—	X	—		X	X		—	—
	Source and target Meta-model	X	X	X	X	X	X	X	X		—	X		—	—
	Transformation traces (Demonstrative editing actions)	—	—	—	—	—	—	—	—		—	—		X	X
Derivation process		Ad-hod heuristic	Ad-hod heuristic	Pattern matching	IPL	Ad-hoc algorithm	Hybrid heuristic search	Hybrid heuristic search	GP algorithm		Hybrid heuristic search	RCA		Pattern matching	Pattern matching
Output	Rules	X	X	X	X	X	X	X	X		—	X		—	—
	Executable rules	—	X	X	—	X	—	—	X		—	—		—	—
	n-m rules	1-1	1-1	2-1	n-m	n-m	1-m	1-m	n-m		—	—		—	—
	other	—	—	—	—	—	—	—	—		n-m matching betwe en source and model elements	n-m matching between source and model elements		n-m transformation patterns	n-m transformation patterns
	Advanced operations	—	—	—	—	—	—	—	—		—	—		Allows string concatenation and math ope.	Allows string concatenation and math ope.
	Rule execution control	—	—	—	—	—	—	—	—		—	—		—	basic
	Validation step	—	—	—	—	—	X	X	X		X	—		—	—

Fig.II.1. Les approches de transformation de modèles par l'exemple (Adaptée à partir de [Fau 13])

Particulièrement lourdes sont les tâches qui nécessitent d'appliquer une nouvelle technologie afin d'adapter une application aux changements d'exigences ou à un environnement différent. Le coût élevé de la maintenance du logiciel pourrait être réduit en améliorant automatiquement la conception de programmes orientés-objet sans altérer leur comportement.

L'avantage potentiel des outils automatisés et adaptifs de maintenance ne se limite pas à un seul domaine, mais s'étend sur un large spectre de développement moderne de logiciels. La principale préoccupation des développeurs est de produire du code très efficace et optimisé, capable de résoudre des problèmes scientifiques et techniques intenses dans un temps très réduit (Rapid Application Development). Une des questions importantes dans la maintenance automatisée du logiciel est de proposer des outils automatisés qui améliorent la qualité de ce logiciel. En effet, afin de limiter les coûts et améliorer la qualité de leurs systèmes logiciels, les entreprises tentent de faire respecter les bonnes pratiques de développement et de conception, et de même, éviter les mauvaises pratiques [Kes 10].

Kessentini et al. [Kes 10] distinguent deux étapes de transformations endogènes. La première étape est l'identification des éléments de modèle source (seulement quelques fragments de modèle) à transformer, et la seconde étape est la transformation elle-même. Dans la plupart des cas, les transformations endogènes correspondent au refactoring de modèles où les méta-modèles d'entrée et de sortie sont les mêmes. Dans ce cas, la première étape est la détection des opportunités de refactoring (ex : des défauts de conception) et la seconde est l'application des opérations de refactoring (la transformation).

Les transformations endogènes sont principalement liées aux activités de maintenance suivantes: 1) L'optimisation (une transformation visant à améliorer certaines qualités opérationnelles (ex : performance), tout en préservant la sémantique du logiciel); 2) Le refactoring (un changement à la structure interne du logiciel pour améliorer certaines de ses caractéristiques de qualité telles que la compréhensibilité, la modificabilité, la réutilisabilité, la modularité, l'adaptabilité, sans changer son comportement observable) [Kes 10].

II.4.2. Amélioration de la qualité des modèles

Un type spécifique d'évolution du modèle pour lequel les transformations sont particulièrement utiles est l'amélioration de la qualité du modèle. Les modèles peuvent avoir différents types de critères de qualité qui doivent être satisfaits, mais cette qualité a tendance à se dégrader au fil du temps en raison de nombreuses modifications apportées au modèle pendant sa durée de vie. Par conséquent, nous avons besoin de transformations qui nous permettent d'améliorer la qualité du modèle. En particulier, si nous voulons améliorer la qualité structurelle d'un modèle, nous pouvons faire usage de refactoring de modèle. En analogie avec les refactorings de programmes, il s'agit d'une TM horizontale et endogène qui permet d'améliorer la structure du modèle tout en préservant son comportement. Divers auteurs ont commencé à explorer le problème de refactoring de modèles UML, ce n'est donc qu'une question de temps avant que les environnements de modélisation UML commencent à supporter cette activité [Bab 10].

II.4.3. La gestion de l'incohérence des modèles

L'activité de gestion d'incohérence de modèles est également bien adaptée à être supportée par la TM. En raison du fait que les modèles sont généralement exprimés par de multiples points de vue [Gru 98], qu'ils sont en constante évolution, et qu'ils sont souvent développés dans un cadre collaboratif, les incohérences dans les modèles ne peuvent pas être évitées. Par conséquent, nous avons besoin de techniques basées sur la TM pour réparer ces incohérences [Bab 10].

II.5. LE REFACTORING DES MODÈLES

Les activités de conception logicielle ne se limitent pas à la création de nouvelles applications. D'une part, le concepteur est souvent amené à modifier et faire évoluer une application existante. D'autre part, une application ne se construit pas en une seule étape, et un développement en cours nécessite des réorganisations et des remises en cause [Ben 08].

L'activité de restructuration est bien connue en génie logiciel. Il s'agit de la transformation endogène d'une représentation à une autre du même niveau d'abstraction, tout en préservant le comportement externe du système (fonctionnalités et sémantique) [Van 05]. Elle vise à améliorer certains facteurs de qualité, dont la lisibilité, la compréhension, l'adaptabilité et, l'efficacité du modèle. Elle peut aussi être utilisée dans un souci de simplification. Dans le contexte orienté-objet, le terme « refactoring » est utilisé à la place de la restructuration [Ben 08].

En ingénierie dirigée par les modèles, les techniques de refactoring sont peu nombreuses. Hacene et al. proposent, en 2007, une analyse formelle des données relationnelles pour restructurer des modèles UML. Cette analyse formelle s'effectue sur les méta-modèles UML à l'aide de treillis permettant de dériver une hiérarchie d'abstractions à partir d'un ensemble d'entités. Markovic et Baar proposent, en 2008, quelques règles de refactoring de base pour des diagrammes de classes en tenant compte des contraintes OCL, inspirées des règles de refactoring proposées dans les langages orientés-objet [Ben 08].

La complexité inhérente des modèles continue à croître tant qu'ils évoluent. Pour contourner cette complexité croissante, nous avons besoin de techniques comme le refactoring des modèles qui améliore leur qualité et réduit leur complexité [Van 05]. On définit le refactoring des modèles comme suit : Un refactoring de modèle est une transformation d'un modèle à un modèle différent exprimé dans le même langage de modélisation afin d'améliorer la qualité des modèles d'un système, sans changer son comportement externe [Van 05].

Les refactorings des modèles jouent un rôle important en MDE et peuvent se situer à différents niveaux du MDE, par exemple, dans les cahiers de charges, les modèles de conception, ou le code source. Comme le processus de refactoring du code source, le processus de refactoring des modèles consiste en quelques activités distinctes :

1. L'identification des lieux où le modèle doit être refactorisé.
2. Déterminer quel refactoring(s) de modèle doit être appliqué dans les lieux identifiés.
3. Appliquer le refactoring(s) de modèle.
4. Garantir que le refactoring appliqué préserve le comportement.

5. Évaluer l'effet du refactoring sur les caractéristiques de qualité de la conception (telles que la complexité, la compréhensibilité, la maintenabilité).

6. Maintenir la consistance entre le modèle refactorisé et les autres artéfacts logiciels (les besoins, le code du programme, etc) [Van 05].

II.5.1. Préservation du comportement du modèle

Par définition, un refactoring ne devrait pas modifier le comportement ou les fonctionnalités d'un logiciel. La première définition de la conservation du comportement a été donnée par Opdyke en 1992. Il stipule que, pour les mêmes valeurs d'entrée, l'ensemble des valeurs de sortie résultant doit être le même avant et après l'application du refactoring. Mais ce n'est pas une contrainte assez forte dans certains domaines d'application. Par exemple, elle ne tient pas compte des autres critères comportementaux et importants tels que les contraintes de temps, de mémoire, de consommation d'énergie, de synchronisation entre processus distribués,…etc. Dans les applications où le traitement de données joue un rôle prépondérant, une autre contrainte essentielle est la préservation de la cohérence de ces données. En UML, certaines sortes de diagrammes décrivent le comportement (ex. les machines à états, les diagrammes d'interaction), alors que d'autres décrivent la structure (ex. les diagrammes de classes et les diagrammes de composants). Le défi est donc de trouver une bonne définition du comportement d'un modèle, afin de pouvoir garantir la conservation de ce comportement lors d'un refactoring. Une définition formelle et mathématique peut s'avérer nécessaire [Men 07].

II.5.2. Préservation de la cohérence du modèle

Le design d'un logiciel est souvent modélisé en utilisant différentes sortes de diagrammes. Par exemple, un modèle UML est typiquement constitué de diagrammes de classes, d'interaction, de cas d'utilisation, d'états, etc. À cause de cette variété de modèles utilisés lors de la modélisation, un problème apparaît lorsqu'on veut appliquer un refactoring. Le refactoring d'un diagramme peut engendrer des incohérences dans les autres diagrammes. Le défi est donc d'adapter tous les diagrammes associés afin de préserver leur cohérence avec le diagramme refactorisé [Men 07].

II.5.3. Exemple

Voici un exemple illustrant l'utilisation de deux refactorings (voir la figure II.2). Il fait intervenir les refactorings *Add Class* et *Pull Up Method*. *Add Class* permet d'ajouter une nouvelle classe à l'application tandis que *Pull Up Method* déplace des méthodes des sous-classes avec des signatures identiques vers une superclasse. Supposons une ville virtuelle dans laquelle des voitures et des bus peuvent se déplacer. On peut associer à cette ville un graphe qui représente les divers chemins que ces bus et voitures peuvent emprunter. Les classes *Car* et *Bus* contiennent chacune une méthode *calculateNextNode()* qui permet de calculer le nœud suivant vers lequel le véhicule doit se déplacer.

Cette méthode de calcul est identique pour les voitures et les bus. Il est préférable d'éviter ce code dupliqué car si l'on modifie l'algorithme de calcul à un endroit, il faudra le modifier de l'autre côté. De plus, le design de l'application oblige la duplication du code de cette méthode à chaque fois que l'on rajoutera de nouveaux véhicules utilisant cette méthode de calcul.

On peut éviter ceci en ajoutant une classe *Véhicule* comme superclasse de *Car* et *Bus* par le refactoring *Add Class* et ensuite, appliquer *Pull Up Method* sur *calculateNextNode()*. Il est trivial de voir dans cet exemple que l'application des refactorings conserve le comportement grâce à l'héritage [Sél 05].

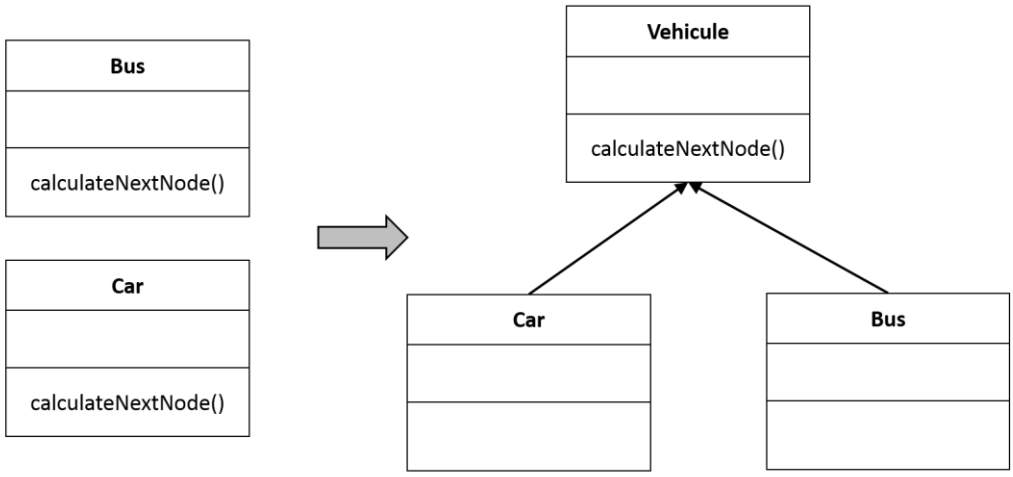


Fig.II.2. Application de « *Add Class* » et ensuite « *Pull Up Method* » sur « *calculateNextNode()* » [Sél 05]

II.5.4. État de l'art du refactoring des modèles

Selon Sunyé, Astels, et Bogeret, en 2002, les modèles UML sont principalement considérés comme des candidats appropriés pour le refactoring. En particulier, le refactoring des diagrammes de classes a été étudié par plusieurs chercheurs. Beaucoup de refactorings connus dans la programmation orientée-objet peuvent aussi être portés aux diagrammes de classes UML. D'après Bouden en 2006, d'autres techniques sont nécessaires afin d'assurer la traçabilité et la cohérence entre les diagrammes de classes et leurs codes sources correspondants, lors de l'application du refactoring des diagrammes de classes.

Lorsqu'il s'agit de raisonner sur les propriétés de conservation du comportement des modèles, nous devons nous appuyer sur les contraintes OCL, les modèles du comportement (par exemple : les diagrammes d'états, les diagrammes d'interactions ou les diagrammes d'activités), ou le code du programme. En ce qui concerne le refactoring des modèles du comportement, il n'existe pas assez de travaux disponibles à part quelques approches qui abordent le problème de refactoring des diagrammes d'états, et que les chercheurs essayent de démontrer formellement leurs propriétés de conservation du comportement.

Van Kempenet et al. ont utilisé, en 2005, le formalisme *CSP* (Constraint Satisfaction Problem) pour décrire les refactorings des diagrammes d'états-transitions, et ont montré comment ce formalisme peut être utilisé pour vérifier qu'un refactoring préserve effectivement le comportement.

Gheyiet et al. ont suggéré, en 2005, de spécifier le refactoring des modèles en Alloy, un langage de modélisation orienté-objet utilisé pour la spécification formelle. Il peut être utilisé pour démontrer la préservation des propriétés sémantiques des refactorings des modèles.

Différents formalismes ont été proposés pour comprendre et explorer le refactoring des modèles. La plupart de ces approches suggèrent l'expression de ces refactorings de manière déclarative.

Pretschner et Prenninger ont proposé, en 2006, une approche formelle pour le refactoring des machines à états, basée sur les prédicats et les tables logiques.

Van Der Straeten et D'Hondt ont utilisé, en 2006, un moteur de raisonnement logique à chaînage-avant, pour supporter les refactorings complexes des modèles.

Biermannet et al. en 2006, et Mens et al. en 2007, ont utilisé la théorie de la transformation des graphes pour spécifier le refactoring des modèles, et ils ont basé sur les propriétés formelles pour analyser et raisonner sur ces refactorings [Men 07].

II.5.5. Le refactoring des modèles UML

II.5.5.1. Le langage de modélisation UML

UML est une notation permettant de modéliser un programme de façon standard. En 1997, UML est devenu une norme OMG, désormais, la référence en termes de modélisation objet [Bou 06].

Un modèle est une abstraction de la réalité. Modéliser est le processus qui consiste à identifier les caractéristiques intéressantes d'une entité en vue d'une utilisation précise dans un contexte donné. UML est donc un moyen d'exprimer des modèles objet en faisant abstraction de leur implantation, c'est-à-dire que le modèle fourni par UML devrait être valable pour n'importe quel langage de programmation orienté-objet. En effet, il propose une multitude de concepts pour représenter diverses vues d'un système.

La force du langage de modélisation UML provient du fait qu'il s'appuie sur un méta-modèle, un modèle de plus haut niveau qui définit les éléments d'UML (les concepts utilisables) et leur sémantique (leur signification et leur mode d'utilisation). Le méta-modèle permet de se placer à un niveau d'abstraction supérieur car il est plus abstrait que le modèle qu'il permet de construire [Bou 06].

Un diagramme UML est une représentation graphique qui s'intéresse à une vue précise du modèle. La première vue permet de représenter le programme physiquement à l'aide des diagrammes d'objets, des diagrammes de classes, des diagrammes de cas d'utilisation, des diagrammes de composants et des diagrammes de déploiement. Quant à la seconde vue, elle permet de montrer le fonctionnement du programme à l'aide des diagrammes de séquence, des diagrammes de collaboration, des diagrammes d'états et les diagrammes d'activités [Bou 06].

Le diagramme de classes, cœur du langage de modélisation UML, est le diagramme le plus utilisé par les développeurs de programmes. Il permet de définir les composants d'un programme orienté-objet. En général, on constate sa disponibilité tôt dans le cycle de vie d'un programme et en plus, il est souvent la seule forme de documentation. Les diagrammes de classes expriment de manière générale la structure statique d'un système, en termes de classes et de relations entre ces classes [Bou 06].

Une classe est une description abstraite d'un ensemble d'objets du domaine d'application. Elle définit leur structure, leur comportement et leurs relations. Dans la notation UML, la classe est représentée par un rectangle divisé en trois parties. Le nom de la

classe, qui est unique dans un paquetage, se trouve dans la première partie. Dans la deuxième partie se trouvent les attributs de la classe. Dans la troisième partie prennent place toutes les méthodes (les comportements) de la classe. On distingue quatre types de relations : l'héritage, l'association, l'agrégation, et la composition. Dans un diagramme de classes, les paquetages servent à organiser les différentes classes en espaces de noms et en sous-systèmes [Bou 06].

II.5.5.2. Le rôle du langage UML en MDE

MDE ne nous impose pas UML. N'importe quel langage de modélisation peut être utilisé. Cependant, UML est devenu le plus répandu des langages de modélisation pour modéliser les systèmes logiciels. Les modèles exprimés en UML, capturent la connaissance sur un système à différents niveaux d'abstraction, en partant des modèles de besoins et d'analyse, et passant par les modèles de conception, jusqu'aux modèles utilisés comme spécifications de programmation. UML a plusieurs applications en MDE. Il peut être utilisé comme un langage de modélisation général ou comme base pour une extension spécifique à un domaine particulier, ou pour des fins de réutilisation [Van 05].

II.5.5.3. L'apport du refactoring des modèles UML

Le refactoring des modèles UML est utile pour les trois raisons principales suivantes :

1. Les concepteurs, les développeurs et spécifiquement les mainteneurs préfèrent visualiser leurs programmes sous forme de modèles pour simplifier la compréhension des programmes orientés-objet.

2. Manipuler directement le code à un niveau plus abstrait (c'est-à-dire méthodes, variables, et classes plutôt que des caractères) peut rendre le refactoring plus efficace.

3. Visualiser le code en utilisant des diagrammes de classes, spécifiquement le contenu des classes et les liens entre elles, peut aider dans la détection des mauvaises odeurs (*Bad smells*). Beck et Fowler ont décrit, en 1999, les mauvaises odeurs comme certaines structures dans un code qui suggèrent la possibilité d'appliquer un refactoring.

Astels a affirmé, en 2001, que le diagramme de classes et le diagramme de séquences prouvent particulièrement leur utilité pour le processus de refactoring de façon complémentaire puisque le diagramme de classes donne une vue statique du programme (les classes composant le système, leur contenu et leurs liens) alors que le diagramme de séquences donne une vue dynamique (l'appel des méthodes et l'accès aux champs et aux objets) [Bou 06].

II.6. CONCLUSION

Dans ce chapitre, nous avons mis en valeur l'importance de l'automatisation en MDE, et le rôle primordial que la transformation de modèles par l'exemple joue, ainsi que ses apports. Nous avons également survolé l'état de l'art des approches existantes de transformation de modèles à partir des exemples où nous avons mis en évidence leurs limitations, et les différences entre elles. Nous avons, donc, pu constater l'absence d'approches de transformation endogène qui produisent effectivement des RTs exécutables n-m, et qui nécessitent moins d'interaction avec l'utilisateur. La transformation endogène a été, ensuite, abordée avant de s'intéresser, vers la fin, au refactoring des modèles qui est au cœur de notre projet portant, spécifiquement, sur le refactoring des diagrammes de classes UML.

CHAPITRE III : ÉTUDE ET CONCEPTION DE L'APPROCHE PROPOSÉE

III.1. INTRODUCTION

Le refactoring des diagrammes de classes est une activité qui est au cœur du développement logiciel, dans la mesure où elle intervient fréquemment dans les phases de maintenance des produits logiciels, et contribue à l'amélioration de leur qualité. Notre objectif est de définir un processus de dérivation automatique des règles de transformation, qui s'applique au problème du refactoring des diagrammes de classes UML. Ce processus est basé sur un apprentissage à partir des exemples de refactorings préétablis, et exploite la programmation génétique (GP) afin de dériver, automatiquement, un ensemble de règles exécutables qui effectuent le refactoring d'un diagramme de classes.

Le refactoring est mieux adapté pour être abordé comme étant une transformation *« in-place »*, car il agit directement sur le même modèle et c'est ce qui est constaté, également, par les experts en génie logiciel. De ce fait, nous avons choisi de traiter le refactoring comme une transformation *« in-place »*.

Ce chapitre met en évidence les détails de la conception du processus de dérivation proposé. Tout d'abord, nous donnons une vue conceptuelle du langage et moteur de règles JESS que nous utilisons pour l'écriture et l'exécution des RTs dérivées. Ensuite, nous entamons l'approche proposée en commençant par introduire les notions et les étapes de la GP dans un cadre général. Puis, nous expliquons en détail l'adaptation de l'algorithme de GP à notre problème spécifique, en soulevant ses différentes particularités qui ont engendré des efforts considérables d'adaptation. Des améliorations du processus de dérivation sont exposées vers la fin du chapitre, dans le cadre de la GP adaptative. Ces améliorations ont prouvé leur grande utilité dans l'optimisation de la qualité et des performances de notre approche.

III.2. VUE CONCEPTUELLE DE JESS

JESS est un moteur de règle qui permet de stocker en mémoire des connaissances exprimées sous forme de faits (*facts*). Il est possible, ensuite, de raisonner sur ces connaissances à l'aide de règles déclaratives. Chaque fait peut contenir plusieurs attributs, appelés « *slots* ». JESS offre également la possibilité de définir des gabarits de faits (*template*). Afin d'utiliser JESS pour le refactoring des diagrammes de classes, nous exprimons chaque diagramme de classes sous forme d'un ensemble de faits. Chaque élément du diagramme 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 forme de gabarits de faits. La figure III.1 présente un exemple simple, où sont représentés un méta-modèle et un diagramme de classes. Une transformation de modèles est représentée sous forme d'un ensemble de règles JESS.

III.2.1. Problématique

La transformation « *in-place* » est effectuée directement sur le modèle lui-même. De ce fait, les règles de transformation dérivées par l'algorithme de GP filtrent et génèrent les mêmes types

de faits. Par exemple, les règles filtrent des faits de type classe et attribut et génèrent des faits de mêmes types. Ici, nous nous trouvons face à une problématique où les faits que nos règles filtrent et qui représentent le modèle source ne sont pas monotones, c'est-à-dire, qu'ils subissent des changements dans le temps. Cette nature dynamique des faits ne permet pas de déterminer, dès le début, les règles qui seront déclenchées, car les règles se déclenchent au fur et à mesure que d'autres génèrent de nouveaux faits.

```
(deftemplate class (slot name (type STRING)))
(deftemplate inheritance (slot class (type STRING)) (slot superclass (type STRING)))
(assert (class (name Personne)))
(assert (class (name Homme)))
(assert (inheritance (class Homme) (superclass Personne)))
```

Fig.III.1. Exemple d'un diagramme de classes et son méta-modèle exprimés en JESS

Pour prendre en charge cette situation, JESS utilise l'algorithme Rete (Rete est le mot Latin qui signifie réseau). Cet algorithme détermine de façon incrémentale l'ensemble des règles satisfaites durant le changement de la base de faits. Cette détermination est donc efficace car il n'y a pas d'itération sur l'ensemble des faits. Autrement dit, au lieu de tester tous les faits à chaque itération comme le font beaucoup d'autres moteurs de règles, seuls les nouveaux faits sont testés sur les parties gauches des règles (LHS : Left Hand Side) ou prémisses. D'autre part, les conditions communes à plusieurs règles, qui sont fréquentes, sont partagées de façon à ce qu'en testant seulement quelques conditions, on puisse déterminer la satisfiabilité de plusieurs règles.

III.2.2. Fonctionnement de l'algorithme Rete

Rete est un algorithme de recherche de patterns qui augmente sa rapidité et son efficacité en sacrifiant de l'espace mémoire. Dans Rete, les règles sont décrites sous forme de réseau de nœuds. Les faits ou les entrées passent par ce réseau et subissent des tests à chaque nœud. Si ces faits arrivent à une feuille du réseau Rete, la règle correspondante est ajoutée à l'agenda des règles qui seront déclenchées [Dje 10]. Rete fonctionne avec les modifications (assert, retract, modify) de la base des faits. À chaque fois qu'une modification de cette base a lieu, elle est propagée à travers le réseau Rete pour engendrer des modifications dans l'agenda [Cro 00].

L'algorithme Rete est implémenté par la construction d'un réseau de nœuds, dont chacun représente un ou plusieurs tests dans la partie LHS d'une règle. Les faits ajoutés, modifiés, ou supprimés de la base des faits sont traités par ce réseau de nœuds. En bas du réseau, se trouvent les nœuds feuilles qui représentent chacun une seule règle. Lorsqu'un ensemble de faits traverse tout le chemin descendant qui mène à une feuille, il aurait passé tous les tests de la partie LHS d'une règle, et cet ensemble devient ce qu'on appelle une activation. La partie droite (RHS : Right Hand Side) de la règle en question est alors exécutée (déclenchée) si l'activation n'est pas annulée avant, par la suppression d'un ou de plusieurs faits se son ensemble d'activation [Fri 08].

Le réseau Rete est composé de deux types de nœuds : les nœuds à une seule entrée (*one-input*), et les nœuds à deux entrées (*two-input*). Les nœuds *one-input* effectuent des tests sur des faits individuellement, tandis que les nœuds *two-input* font des tests sur plusieurs faits à la fois et procèdent à des groupements. Des sous-types de ces deux classes de nœuds sont également utilisés, et il existe encore des types auxiliaires tels que les nœuds terminaux (ou feuilles) mentionnés auparavant [Fri 08]. La compilation des deux règles suivantes : (defrule example-1 (x) (y) (z) =>) et (defrule example-2 (x) (y) =>) donne le réseau Rete de la figure III.2.

Les nœuds désignés par x?, y?, etc., séparent les faits en catégories selon leurs têtes, et testent si un fait est de la catégorie appropriée. Les nœuds désignés par + mémorisent tous les faits et se déclenchent à chaque fois qu'ils reçoivent les données de leurs entrées gauches et droites. Afin de lancer le réseau, JESS présente les nouveaux faits, comme ils sont ajoutés à la base de faits, à chaque nœud en haut du réseau.

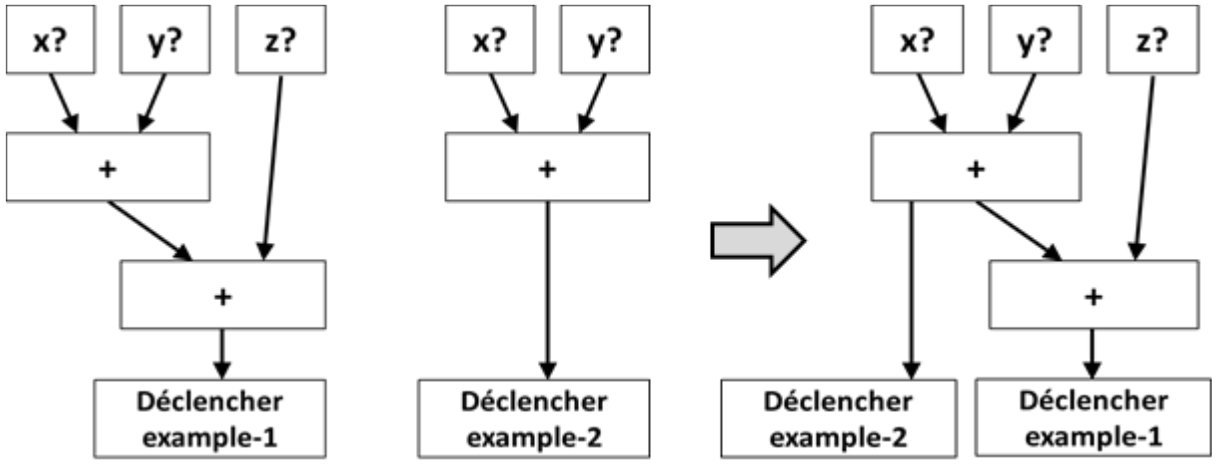


Fig.III.2. Réseau Rete des deux règles example-1 et example-2 avec leur réseau optimisé

Chaque nœud prend une entrée du haut et envoie sa sortie vers le bas. Un nœud *one-input* reçoit généralement un fait du haut, il le teste, et si le test réussit, il envoie ce fait au prochain nœud, sinon, il ne fait rien. Les nœuds *two-input* doivent intégrer des faits de leurs entrées gauche et droite, et pour y parvenir, leur comportement devrait être plus complexe. D'abord, tous les faits qui atteignent le haut d'un nœud *two-input* pourraient, potentiellement, contribuer à une activation parce qu'ils ont déjà réussi tous les tests qui s'appliquent à un seul fait. Les nœuds *two-input* doivent donc mémoriser tous les faits qui s'y présentent, et tenter de grouper les faits arrivant à leurs entrées gauches avec ceux arrivant à leurs entrées droites, afin de construire des ensembles d'activations complets. De ce fait, un nœud *two-input* est doté d'une mémoire gauche et d'une mémoire droite, et grâce à ce mécanisme, JESS évite l'inefficacité algorithmique des autres moteurs d'inférences en restant incrémental plutôt qu'itératif, mais en sacrifiant de l'espace [Fri 08].

III.2.3. Optimisation de l'algorithme Rete

Nous pouvons distinguer deux simples optimisations qui peuvent rendre Rete meilleur encore. La 1ère consiste à partager les nœuds *one-input* identiques. Dans le réseau gauche de la

figure III.2, Il y'a 5 nœuds en haut, cependant, 3 seulement sont distincts. On peut modifier le réseau de manière à partager ces nœuds entre les deux règles. Toutefois, ceci n'est pas toute la redondance dans le réseau original, car il y'a un nœud *two-input* qui effectue exactement la même fonction (intégrer les pairs de faits de types x et y) dans les deux règles, et on peut partager ça également (Le réseau à droite de la figure III.2). Ce genre de partage survient fréquemment dans les systèmes réels, et constitue un facteur significatif de performance [Fri 08].

III.3. L'APPRENTISSAGE DES RTs À PARTIR DES EXEMPLES

L'approche que nous proposons devrait fonctionner même si des traces fines de transformation ne sont pas disponibles. En outre, les contraintes sur la forme ou la taille des règles devraient être aussi limitées que possible. Enfin, les ensembles de règles produits doivent être exécutables sans étape de raffinement manuel. Les règles de transformation sont des programmes qui analysent certains aspects des modèles sources donnés en entrée et synthétisent les modèles cibles correspondants en sortie.

Apprendre des structures complexes et dynamiques telles que les programmes n'est pas une tâche facile [Ban 98]. Parmi les outils possibles qui peuvent être utilisés pour la génération automatique des programmes, la programmation génétique (GP) est un candidat puissant car il a été créé dans ce but [Koz 05]. Ceci a motivé notre choix du GP pour dériver automatiquement des ensembles de règles, c'est-à-dire, des programmes déclaratifs, en utilisant des exemples de transformations de modèles, à savoir, des entrées/sorties complexes.

III.4. PROGRAMMATION GÉNÉTIQUE

La programmation génétique (GP) est une méthode systématique, qui permet aux ordinateurs de résoudre automatiquement un problème à partir d'un énoncé de la tâche à accomplir [Koz 99]. Elle représente une branche d'une famille d'algorithmes destinés à l'optimisation et à l'apprentissage, qui s'inspirent de l'évolution naturelle des espèces qui s'adaptent au cours des générations à leur environnement. Cette famille d'algorithmes, qu'on appelle « les algorithmes évolutionnaires » (AEs), se caractérise par les points suivants [Yu 10] :

- Basée sur les populations : les AEs font évoluer un ensemble de solutions, cet ensemble est appelé population.
- Orientée vers la fitness : chaque solution est appelée individu. Afin d'imiter le concept du suivi des meilleurs de la sélection naturelle, l'algorithme attribue à chaque individu une mesure de sa performance appelée valeur de fitness.
- Guidée par la variation : chaque individu subit, à travers les générations, un ensemble d'opérations afin d'imiter les variations génétiques subies par les différentes espèces.

III.4.1. Aperçu de la GP

La figure III.3 illustre le processus d'une exécution d'un programme génétique. L'algorithme démarre par une population initiale de programmes, générés de manière aléatoire. Ensuite, chaque individu est évalué afin de déterminer sa capacité à résoudre le problème donné.

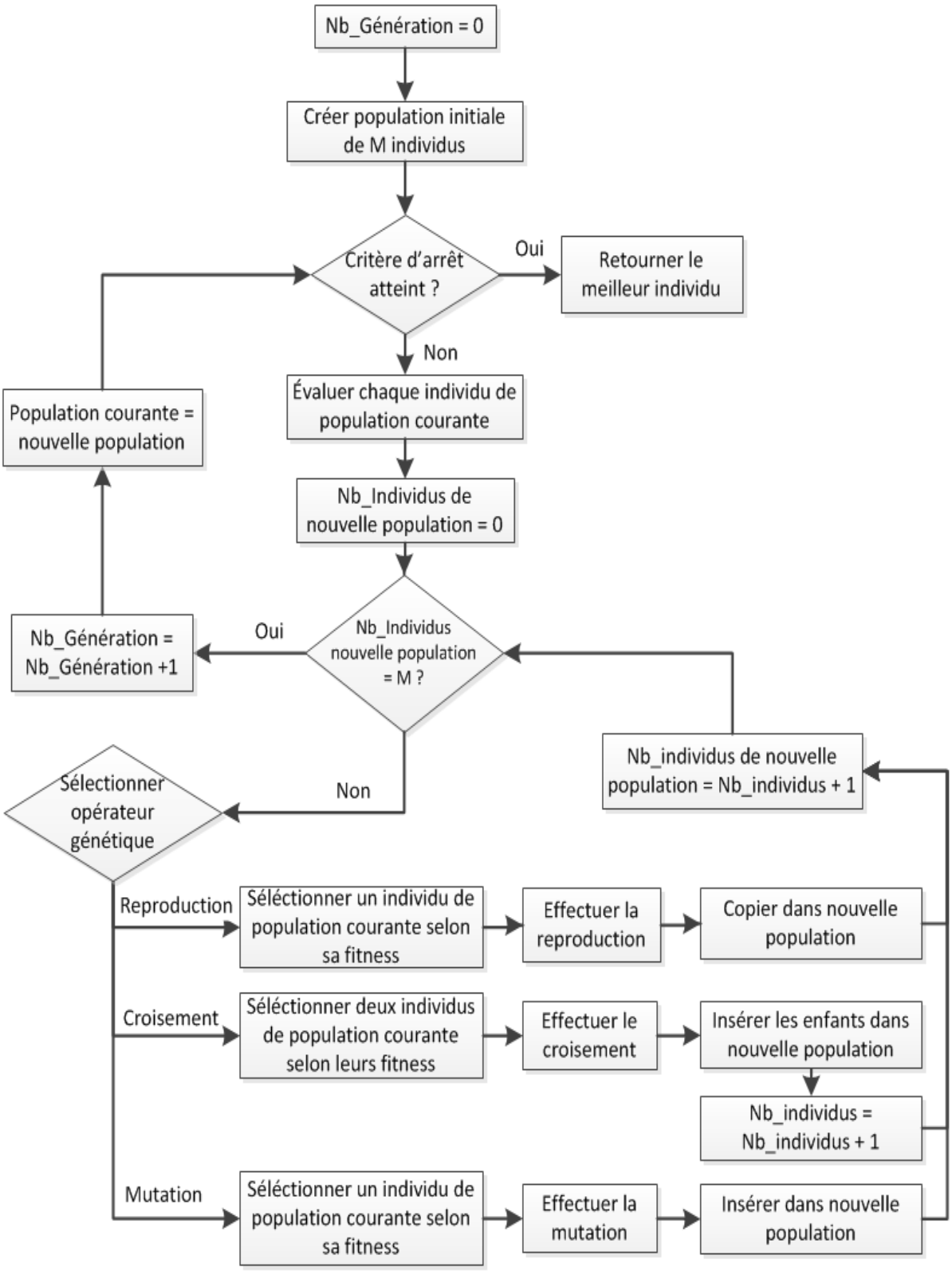


Fig.III.3. Organigramme de la programmation génétique [Bak 14]

L'évaluation se fait par usage d'une fonction fitness qui associe à chaque programme un rang. Puis, une nouvelle génération est dérivée à partir de la génération courante, en utilisant 3 opérateurs génétiques : la reproduction, le croisement et la mutation.

Le processus de dérivation est conçu de sorte que la survie et la reproduction des meilleurs individus soient favorisées, tout en laissant une chance aux moins bons de s'améliorer. Lorsque la nouvelle génération est obtenue, ses individus sont évalués à leur tour. Le processus itère ainsi, ce qui améliore la qualité de la population au fil des générations. L'exécution du programme génétique continue jusqu'à satisfaire la condition d'arrêt que l'utilisateur spécifie. Le meilleur programme est alors retourné [Bak 14].

III.4.2. Spécification du problème

La GP vise à dériver un programme qui permet de réaliser la tâche spécifiée. Pour obtenir un programme qui résout un problème particulier, 4 éléments nécessaires doivent être définis (voir la figure III.4) :

- Les terminaux et les fonctions : ces deux ensembles délimitent l'espace de recherche. Ils forment les briques dont peut être composé le code de chaque programme dérivé par l'algorithme. Les terminaux sont les variables et les constantes utilisées par le programme. L'ensemble des fonctions contient des opérateurs arithmétiques simples ou même des fonctions complexes dédiées au domaine d'application.
- La fonction de fitness : c'est le moyen principal de communiquer la description de la tâche à accomplir au programme génétique. Elle mesure la capacité de chaque programme à résoudre le problème posé.
- Les paramètres de contrôle de l'exécution : ils servent à contrôler l'exécution du programme génétique. Parmi les paramètres d'exécution les plus utilisés : la taille de la population, la taille maximale de chaque programme, et la probabilité d'application de chaque opérateur génétique.
- Le critère d'arrêt du programme : comme condition d'arrêt, on peut imposer au programme génétique de s'arrêter à un nombre maximal de générations, ou à l'atteinte d'un certain résultat.

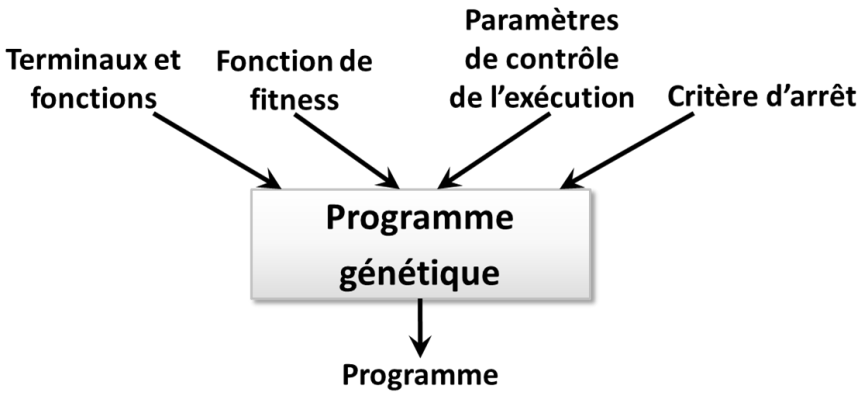


Fig.III.4. Éléments à fournir au programme génétique [Bak 14]

III.4.3. Représentation des individus

Dans la GP, chaque individu de la population est un programme. Chaque programme est représenté sous forme d'un arbre syntaxique, par exemple, l'expression ***5x – 6*** est représentée par l'arbre de la figure III.5. Les nœuds de l'arbre indiquent les instructions du programme dérivé, et ses liens représentent les arguments de ces instructions. Les terminaux sont les feuilles de l'arbre, alors que les fonctions sont représentées par des nœuds ayant des liens vers d'autres nœuds qui ne sont rien d'autres que des arguments.

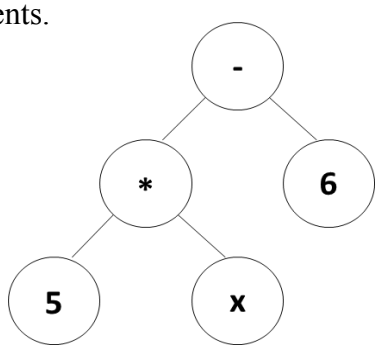


Fig.III.5. Représentation en arbre d'un programme en GP

III.4.4. Génération de la population initiale

Après avoir défini l'ensemble des terminaux, la première étape de l'algorithme est la création d'une population initiale de programmes générés aléatoirement. L'utilisateur doit spécifier comme paramètre, la taille de la population initiale et des populations subséquentes. Trois méthodes de génération possibles, à savoir, *Grow*, *Full*, et *Ramped half-and-hal* ont été définies dans [Koz 99]. Sans tenir compte de la méthode utilisée, il est recommandé que cette première génération n'ait pas d'individus identiques.

III.4.5. Evaluation des individus

Une fois qu'une génération est complétée, chacun de ses programmes est évalué par la fonction de fitness définie. La qualité d'un individu peut être mesurée différemment, selon le problème abordé. À titre d'exemple, la fonction fitness peut mesurer l'écart entre le résultat produit par le programme dérivé et le résultat attendu [Bak 14].

III.4.6. Dérivation de nouveaux individus

III.4.6.1. La sélection

Pour dériver une nouvelle génération, les individus de la génération courante sont sélectionnés selon une probabilité, afin d'y appliquer des opérateurs génétiques. Les meilleurs individus ont plus de chance de subir des opérations génétiques, et cela n'empêche pas que les éléments les plus mauvais participent également à ces opérations. Il existe dans la littérature plusieurs méthodes de sélection, nous en citons ci-dessous deux méthodes très répandues :

- Sélection par roulette : les individus sont représentés par des secteurs sur une roue qui ressemble à celle de la fortune biaisée car la taille de chacun de ses secteurs est proportionnelle à

la fitness de l'individu qu'il représente. Puisque leurs secteurs occupent plus d'espace sur la roue, les meilleurs individus ont plus de chance d'être choisis, lorsque la roue est lancée [Bak 14].

- Sélection par tournoi : N individus sont sélectionnés au hasard de la population, et on prend le meilleur d'entre eux. Le nombre N est la taille du tournoi. Plus N est grand, plus la chance des individus les moins bons d'être sélectionnés diminue. De nouveaux individus sont produits par application des opérations génétiques sur les éléments sélectionnés, et ceci, tant que la nouvelle génération n'est pas complète. Les opérateurs génétiques les plus utilisés sont : la reproduction, le croisement et la mutation [Bak 14].

III.4.6.2. Reproduction

Elle consiste à copier certains individus dans la nouvelle population. Cette opération préserve les meilleurs individus pour qu'ils ne soient pas modifiés par les autres opérateurs génétiques, puis, elle permet de réduire les coûts de l'évaluation des individus, car les éléments reproduits n'ont pas besoin d'être évalués lors des générations subséquentes.

III.4.6.3. Croisement

C'est l'opérateur génétique souvent dominant dans les algorithmes génétiques (avec une probabilité d'environ 0.9). Il permet de produire deux nouveaux individus (appelés fils) en combinant les gènes de deux individus (appelés parents) sélectionnés de la population courante.

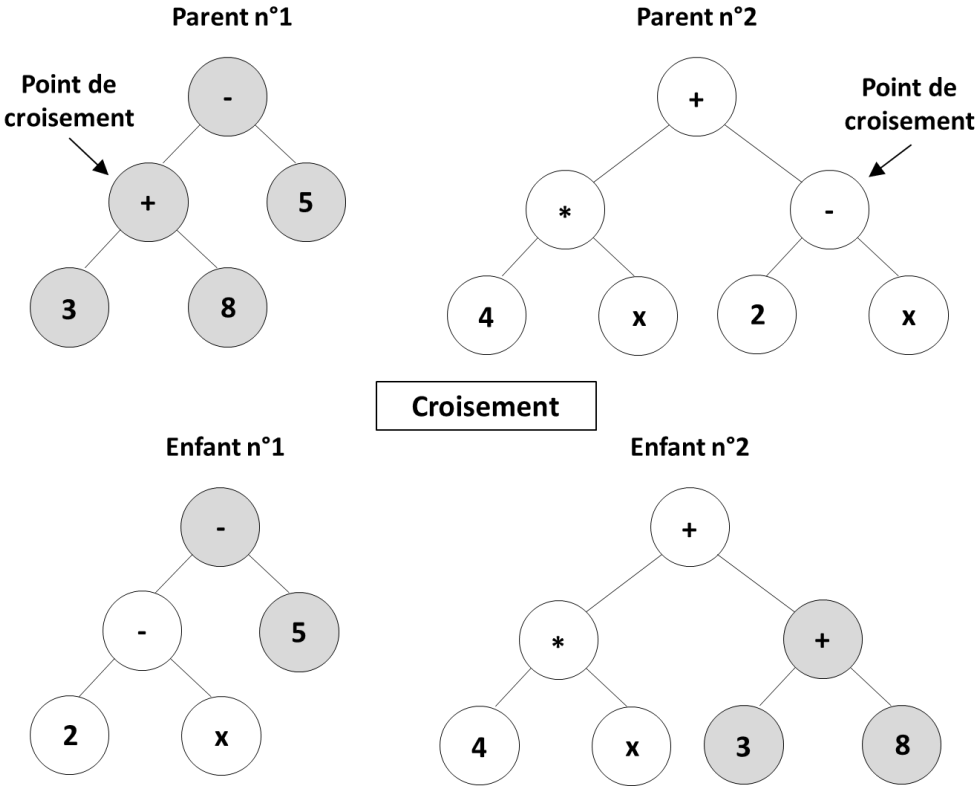


Fig.III.6. Opération de croisement à un seul point

La méthode de croisement la plus fréquente consiste à choisir un seul point de coupure aléatoirement dans chaque arbre parent, et à inverser au niveau des fils produits, les sous arbres des parents ayant pour racines les points de coupure sélectionnés [Luk 98]. Cette méthode est illustrée à travers la figure III.6.

III.4.6.4. 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 GP optent pour une probabilité de mutation assez basse (de l'ordre de 0.1) [Bak 14].

III.5. L'APPRENTISSAGE DES RTs EN UTILISANT LA GP

L'approche que nous proposons vise à dériver des RTs qui s'appliquent au refactoring des diagrammes de classes UML, et ceci, en utilisant un apprentissage à partir des exemples (MTBE). Pour appliquer la GP au problème du MTBE, nous devons considérer deux faits. Tout d'abord, les RTs ne sont pas des programmes impératifs et ne peuvent pas être codées en arbres comme souvent fait en GP [Koz 05]. De plus, les sorties des transformations sont des modèles qui ne sont pas faciles à comparer pour évaluer l'exactitude d'un programme. L'utilisation de la GP pour dériver un programme de transformation implique alors la définition des éléments abordés dans les paragraphes précédents, à savoir, l'encodage d'un ensemble de règles, la génération de la population initiale de programmes, l'évaluation des programmes de chaque génération, et enfin la dérivation de nouvelles générations de transformations. Dans ce qui suit, nous détaillons notre adaptation de l'algorithme de GP au problème spécifique du MTBE.

III.5.1. L'encodage des ensembles de règles

Comme nous l'avons mentionné dans la section III.2, le refactoring des diagrammes de classes est une transformation endogène « *in-place* » qui s'effectue directement sur le modèle source lui-même. Nous ne parlons donc plus de modèles sources et cibles proprement dits, mais nous utilisons plutôt la notion du modèle courant qui subira les différentes actions de transformation. Afin que nos ensembles de règles soient adaptée à ce cas de figure, nous avons décidé que leur structure sera comme suit :

Un programme *p* est donc codé comme un ensemble de RTs, $p = \{r_1, r_2, ..., r_n\}$. Chaque règle possède à son tour deux côtés : un côté gauche que nous appelons *« Left Hand Side » (LHS)* et un côté droit appelé *« Right Hand Side » (RHS).*

Le côté gauche de la règle contient le pattern courant *(Current Pattern)*. Il représente le pattern tel qu'il existe actuellement, avant l'application de la règle, et c'est le pattern à rechercher dans le modèle courant.

Le *pattern courant* est une paire *Current Pattern = (CGC, G)*, où *CGC (Current Generic Constructs)* est un ensemble de constructions génériques courantes et *G (Guard)* est un garde. Si nous prenons la règle de l'exemple ci-après, $CGC = \{C_1, C_2, A_1, A_2\}$ tels que *Ci* et *Ai* représentent respectivement, deux classes et deux attributs. *CGC* peut donc comprendre plus qu'une construction générique du même type. Chaque construction générique a les propriétés de son type de construction dans le méta-modèle.

```
Current Pattern : (LHS)
  // Current Generic Constructs
        Class C1, C2
        Attribute A1, A2
  // Guard : Join Condition
        (and (A1.class = C1.name)
             (A2.class = C2.name))
  // Guard : State Condition
        A1.name = A2.name
Actions : (RHS)
  // Modify Pattern
        //Modify Generic Constructs
              A1
        //Assignments
        //Join-Statements
              A1.class = C3.name
  // Assert Pattern
        //Assert Generic Constructs
              Class C3
              Inheritance I1, I2
        //Assignments
              C3.name := newC
        //Join-Statements
              I1.class = C1.name
              I1.superclass = C3.name
              I2.class = C2.name
              I2.superclass = C3.name
  // Retract Pattern
        // Retract Generic Constructs
              A2
```

Fig.III.7. Exemple d'encodage d'une règle

Le garde G contient deux types de conditions : les conditions de jointure et les conditions d'état. Les propriétés de jointure sont utilisées pour définir l'ensemble des conditions de jointure, qui permet de spécifier le pattern courant comme un fragment d'un modèle, à savoir, un ensemble de constructions inter-reliées selon le méta-modèle. Par exemple, la condition de jointure $A_1.class = C_3.name$ affirme que A_1 doit être un attribut dans la classe C_3.

Les conditions d'état quant à elles, portent sur les propriétés descriptives des constructions génériques courantes ainsi que leurs propriétés de jointure. Par exemple, la condition d'état $A_1.name = A_2.name$ porte sur la propriété descriptive des deux attributs A_1 et A_2 qui est le nom de chacun d'eux, et affirme qu'ils ont la même valeur de cette propriété (même nom).

Ces conditions sont codées comme un arbre binaire contenant des éléments de l'ensemble des terminaux (*T*) et de l'ensemble des primitifs (*I*). *T* est l'union des propriétés des constructions dans *CGC,* et un ensemble de constantes *C*. Pour la règle de la figure III.7, les propriétés sont $C_1.name$, $A_1.name$, etc. Comme les propriétés sont des nombres et des chaînes de caractères, les constantes numériques et chaînes de caractères telles que {*0, 1, Empty, ...*} sont ajoutées aux terminaux.

Puisque des conditions sont manipulées, les constantes booléennes *true* et *false* sont également ajoutées. L'ensemble des primitifs *I* est composé au minimum des opérateurs logiques et des comparateurs (*I = {and, or, not, =,>, <, ...}*). D'autres opérateurs, tels que les opérateurs arithmétiques ou ceux des chaines de caractères, pourraient être ajoutés pour tester les valeurs dérivées des propriétés de base. Comme ce travail utilise le langage de règles JESS, la distinction conceptuelle entre les conditions de jointure et celles d'état n'est pas reflétée dans le code. Les deux types de conditions forment l'arbre de condition avec les terminaux comme des nœuds feuilles et les primitifs comme des nœuds internes. Une règle sans aucune condition sera représentée par un arbre avec le seul nœud "*true*". Une règle est déclenchée pour toute combinaison d'instances pour laquelle l'arbre de condition est vrai.

Le côté droit de la règle représente les actions *(Actions)* à appliquer sur le pattern courant lors du déclenchement de la règle. Ces actions sont sous forme d'un triplet de patterns distincts *Actions = (MP, AP, RP)* où *MP*, *AP*, et *RP* représentent respectivement, le *« Modify Pattern »*, le *« Assert Pattern »*, et le *« Retract Pattern »*.

Le *« Modify Pattern »* est représenté sous forme d'un triplet *MP = (MGC, MPA, MPJ)*, et il correspond à une action de modification du pattern courant. *MGC (Modify Generic Constructs)* est l'ensemble des constructions génériques qui vont subir la modification, *MPA (Modify Pattern Assignments)* est l'ensemble des affectations qui attribuent aux propriétés descriptives des constructions génériques (telles que le nom d'une classe), leurs nouvelles valeurs après la modification. Donc cet ensemble modifie les propriétés descriptives des constructions sujettes de modifications. Enfin, *MPJ (Modify Pattern Join-statements)* est l'ensemble des déclarations de jointures de la construction générique en question après la modification, c'est-à-dire, les nouvelles valeurs que devraient prendre les propriétés de jointure (Par exemple, la classe à laquelle appartient un attribut) après leurs modifications. Par exemple, la déclaration $A_1.class = C_3.name$ indique la modification de la classe à laquelle appartient l'attribut A_1, elle devient désormais la classe C_3.

Noter que *MPA* et *MPJ* ne concernent que les propriétés descriptives et de jointure dont les valeurs subissent des modifications après le déclenchement de la règle.

Le *« Assert Pattern »* est représenté sous forme d'un triplet *AP = (AGC, APA, APJ)*, et il correspond à une action d'ajout de nouvelles constructions génériques au pattern courant. *AGC (Assert Generic Constructs)* est l'ensemble des constructions génériques qui seront ajoutées au pattern courant, *APA (Assert Pattern Assignments)* est l'ensemble des affectations qui attribuent aux propriétés descriptives des constructions génériques ajoutées, leurs valeurs initiales, et *APJ (Assert Pattern Join-statements)* est l'ensemble des déclarations de jointures des nouvelles constructions génériques, c'est-à-dire, les valeurs initiales que devraient prendre leurs propriétés de jointure. Par exemple, la déclaration $I_1.class = C_1.name$ indique que la relation d'héritage I_1 sera ajoutée au pattern courant avec une valeur initiale de sa propriété de jointure, qui est dans ce cas sa classe origine, égale au nom de la classe C_1.

Le *« Retract Pattern »* est un singleton *RP = (RGC)*, et il correspond à une action de suppression de constructions génériques du pattern courant. *RGC (Retract Generic Constructs)* est l'ensemble des constructions génériques à supprimer du pattern courant.

À ce stade, nous tenons à signaler qu'au niveau du *MP* et du *RP*, nous ne déclarons pas effectivement les constructions génériques à modifier ou à supprimer du pattern courant. Nous nous contentons uniquement de les référencer afin de les distinguer, ce qui n'est pas le cas du *AP*, là où nous créons et déclarons concrètement les nouvelles constructions à ajouter au pattern courant.

Cette structure de règles permettra d'obtenir une diversité de programmes qui effectuent les mêmes transformations mais de différentes manières, parfois meilleures et plus intéressantes que l'on puisse attendre ou imaginer. Par exemple, le déplacement d'un attribut commun des classes filles vers la classe mère (refactoring connu sous le nom de *« Pull up attribute »*) peut être effectué de différentes manières : une première est celle adoptée par la règle de la figure III.7, qui modifie la classe de l'un des deux attributs et supprime le deuxième. Nous pouvons avoir le même résultat avec une autre règle qui ajoute un nouvel attribut dans la classe mère avec le nom des deux attributs, et supprime ces derniers.

III.5.2. Création des ensembles de règles

Dériver les règles de transformation en utilisant la programmation génétique nécessite la création d'une population initiale d'ensembles de règles aléatoires. Chaque ensemble de règles doit être syntaxiquement correct par rapport au langage de règles (JESS dans notre travail).

En outre, un ensemble de règles devrait être compatible avec le méta-modèle. À cet égard, les règles devraient décrire des patterns courants et des actions valides. Pour la population initiale, un nombre d'ensembles de règles *population_size* est créé (*population_size* est un paramètre de l'approche). Ces ensembles de règles représentent nos programmes. Le nombre de règles à créer pour chaque programme est choisi aléatoirement dans un intervalle donné, par exemple, un intervalle de 1 à 4.

Pour chaque règle, nous utilisons une combinaison aléatoire de fragments élémentaires de modèles (blocs constructeurs) pour créer le pattern courant. La combinaison aléatoire des blocs

constructeurs vise à réduire la taille de l'espace de recherche en tenant compte des fragments de modèles connectés plutôt que des sous-ensembles arbitraires de constructions.

Un bloc constructeur est un fragment de modèle minimal qui est autonome, c'est-à-dire, que son existence ne dépend pas de l'existence d'autres constructions. Par exemple, une seule classe pourrait former un bloc constructeur. De même, une relation d'héritage forme avec deux classes (superclasse et sous-classe) un bloc constructeur. Cependant, un attribut devrait être associé à sa classe pour former un bloc. La détermination du bloc constructeur pour un méta-modèle donné ne dépend que de ce dernier et non pas de la transformation de ses modèles.

Pour créer le pattern AP, un fragment est aléatoirement choisi dans le pattern courant. Ensuite, un nouveau fragment est créé aléatoirement, contenant au moins une construction générique commune avec le fragment choisi du pattern courant. Cette construction sert à connecter le nouveau fragment avec celui choisi. Puisque ces deux fragments seront liés, il ne faut pas que les constructions déjà existantes soient dupliquées. La connexion se fait en considérant les deux constructions à connecter comme la même construction générique, et puisque cette construction existe déjà dans le fragment choisi, elle ne sera donc pas créée.

Imaginons que nous avons choisi aléatoirement dans le pattern courant, le fragment (*Class* C_1, *Attribute* A_1, A_1*.class =* C_1*.name*) contenant deux constructions génériques connectés C_1 et A_1. Soit (*Inheritance* I_1, *Class* C_3, C_4, I_1*.class =* C_3*.name*, I_1*.superclass =* C_4*.name*) le fragment aléatoire créé. L'une des deux possibilités de connexions ((C_1, C_3) ou (C_1, C_4)) est sélectionnée, disons (C_1, C_4). C_4 est alors remplacée par C_1 dans le fragment choisi y compris les liens.

Pour le pattern MP, un fragment est aléatoirement sélectionné dans le pattern courant. Ensuite, les constructions à modifier sont aléatoirement choisies dans ce fragment.

En ce qui concerne le pattern RP, un fragment est également sélectionné de manière aléatoire dans le pattern courant. Puis, les constructions à supprimer sont aléatoirement choisies dans ce fragment. Pour chacune de ces constructions, s'il y'a d'autres constructions qui dépendent d'elle (Par exemple, une classe qui possède des attributs, ces derniers dépendent alors de cette classe), l'une des stratégies suivantes est adoptée lors de la suppression :

1. Ne pas supprimer la construction en question.
2. Supprimer cette construction avec les constructions qui en dépendent.
3. Supprimer uniquement cette construction. Mais pour cette stratégie, il faut veiller à ce que la validité syntaxique des règles vis-à-vis du langage de transformation utilisé (JESS dans notre cas), soit assurée et maintenue.

Dans le cas contraire, où il n'y a pas des constructions qui dépendent d'elle, cette construction est supprimée directement sans aucun problème. Noter que le pattern courant tout entier peut être considéré comme un fragment, et par conséquent, il peut être utilisé tel qu'il est dans la création des patterns de la partie *Actions*, sans faire la sélection aléatoire d'un fragment.

La dernière étape vers la création des patterns est la génération aléatoire des conditions d'état pour les patterns courants, et des affectations et des déclarations de jointures pour les patterns MP et AP de la partie *Actions*.

Pour un pattern courant, un arbre est créé en mixant aléatoirement des éléments de l'ensemble des terminaux *T*, c'est- à-dire, les propriétés des constructions sélectionnées et les constantes compatibles avec leurs types, et des éléments de l'ensemble des primitifs *I* d'opérateurs. La création de l'arbre est réalisée en utilisant une variante de la méthode "*Grow*" définie dans [Koz 05].

Dans le cas du pattern AP, la génération des affectations se fait en choisissant aléatoirement l'une des techniques suivantes :

1. Sélectionner des propriétés descriptives aléatoires des constructions du pattern courant, et attribuer leurs valeurs aux propriétés descriptives des constructions créées.

2. Sélectionner des propriétés descriptives aléatoires des constructions du pattern AP lui-même, et attribuer leurs valeurs aux propriétés descriptives des constructions créées.

3. Sélectionner des propriétés descriptives aléatoires des constructions du pattern MP, et attribuer leurs valeurs aux propriétés descriptives des constructions créées.

4. Utiliser l'une des fonctions définies, pour générer les valeurs des constructions créées.

Afin de générer les déclarations de jointures du pattern AP, l'une des alternatives suivantes, est aléatoirement choisie :

1. Sélectionner, aléatoirement, des propriétés de jointure des constructions du pattern courant, et attribuer leurs valeurs aux propriétés de jointure des constructions créées.

2. Sélectionner, aléatoirement, des propriétés de jointure des constructions du pattern AP lui-même, et attribuer leurs valeurs aux propriétés de jointure des constructions créées.

3. Sélectionner, aléatoirement, des propriétés de jointure des constructions du pattern MP, et attribuer leurs valeurs aux propriétés de jointure des constructions créées.

Les affectations aléatoires *« propriété-valeur »* sont effectuées en respectant la compatibilité des types de propriétés. Cette compatibilité est vérifiée au niveau du méta-modèle.

Pour le pattern MP, les affectations sont générées en choisissant aléatoirement l'une des méthodes suivantes :

1. Sélectionner des propriétés descriptives aléatoires des constructions du pattern courant, et attribuer leurs valeurs aux propriétés descriptives des constructions à modifier.

2. Sélectionner des propriétés descriptives aléatoires des constructions du pattern AP, et attribuer leurs valeurs aux propriétés descriptives des constructions à modifier.

3. Sélectionner des propriétés descriptives aléatoires des constructions du pattern MP lui-même, et attribuer leurs valeurs aux propriétés descriptives des constructions à modifier.

4. Utiliser l'une des fonctions définies, pour générer les nouvelles valeurs des constructions à modifier.

Afin de générer les déclarations de jointures du pattern MP, l'une des techniques suivantes, est aléatoirement choisie :

1. Sélectionner, aléatoirement, des propriétés de jointure des constructions du pattern courant, et attribuer leurs valeurs aux propriétés de jointure des constructions à modifier.

2. Sélectionner, aléatoirement, des propriétés de jointure des constructions du pattern AP, et attribuer leurs valeurs aux propriétés de jointure des constructions à modifier.

3. Sélectionner, aléatoirement, des propriétés de jointure des constructions du pattern MP lui-même, et attribuer leurs valeurs aux propriétés de jointure des constructions à modifier.

Ici encore, les affectations aléatoires *« propriété-valeur »* sont effectuées en respectant la compatibilité des types de propriétés, qui est vérifiée au niveau du méta-modèle.

III.5.3. Dériver les nouveaux ensembles de règles

En GP, une population de programmes est évoluée et améliorée par l'application des opérateurs génétiques (croisement et mutation). Ces opérateurs sont spécifiques au problème à résoudre. Comme avec la création de la population initiale, les opérateurs génétiques doivent garantir que les ensembles de règles dérivés soient syntaxiquement et sémantiquement valides.

Avant d'appliquer les opérateurs génétiques croisement et mutation pour produire de nouveaux ensembles de règles, un opérateur de reproduction est utilisé pour insérer les *x* meilleurs ensembles de règles dans la nouvelle génération en fonction de leurs valeurs de fitness (stratégie élitiste). Pour la dérivation des ensembles de règles de transformation, la sélection par roulette est utilisée (voir la section III.4.6.1).

Croisement : Après avoir sélectionné deux ensembles de règles parents pour le croisement, deux nouveaux ensembles de règles sont créés en échangeant des parties des parents, c'est-à-dire, des sous-ensembles de règles.

Par exemple, considérons les deux ensembles de règles $p_1 = \{r_{11}, r_{12}, r_{13}, r_{14}\}$ ayant quatre règles et $p_2 = \{r_{21}, r_{22}, r_{23}, r_{24}, r_{25}\}$ avec cinq règles. Si deux points de coupure sont fixés aléatoirement à 2 pour p_1 et 3 pour p_2, les descendants obtenus sont les ensembles de règles $o_1 = \{r_{11}, r_{12}, r_{24}, r_{25}\}$ et $o_2 = \{r_{21}, r_{22}, r_{23}, r_{13}, r_{14}\}$. Puisque chaque règle est syntaxiquement et sémantiquement correcte avant le croisement, cette cohérence n'est pas altérée pour les descendants.

Mutation : La mutation pourrait se produire au niveau de l'ensemble des règles ou au niveau d'une seule règle. Chaque fois, un ensemble de règles est aléatoirement choisi pour subir une mutation, une stratégie de mutation est également sélectionnée de manière aléatoire.

Deux stratégies de mutation sont définies au niveau de l'ensemble de règles : (1) l'ajout d'une règle créée aléatoirement à l'ensemble des règles et (2) la suppression d'une règle choisie aléatoirement. Pour éviter les ensembles de règles vides, la suppression ne peut être effectuée si l'ensemble de règles est réduit à une seule règle.

Au niveau de la règle, de nombreuses stratégies sont possibles. Pour une règle choisie aléatoirement, une de ces stratégies pourrait ajouter ou supprimer une construction générique de l'un des quatre patterns de la règle : le pattern courant, *MP*, *AP*, ou *RP*. La stratégie à appliquer est choisie aléatoirement. En cas de suppression, le choix de la construction à supprimer se fait également de manière aléatoire, et pour le pattern courant comme pour les patterns AP et MP, il faut vérifier et assurer la cohérence de tous les patterns de la partie *Actions*. Par exemple, si une

construction est supprimée du pattern courant, il ne faut pas qu'elle apparaisse dans le pattern RP car à ce moment, elle serait déjà inexistante. Ceci conduit à une erreur syntaxique au niveau du langage de transformation.

Une autre stratégie pourrait également choisir un nombre aléatoire de propriétés dans l'un des deux patterns MP et AP et les lier de manière aléatoire à des propriétés dans le pattern courant et le pattern AP, et à des constantes. Cependant, les patterns de la partie *Actions* doivent être modifiés en conséquence pour éviter les erreurs sémantiques et syntaxiques.

III.5.4. Évaluation des ensembles de règles

Pour la population initiale et au cours de l'évolution, chaque ensemble de règles généré est évalué pour déterminer sa capacité d'effectuer des transformations correctes. Cette évaluation est faite en deux étapes : (1) l'exécution de l'ensemble des règles sur les exemples et (2) la comparaison des modèles cibles des exemples avec ceux produits. Les ensembles de règles sont traduits en JESS, et exécutés sur les exemples en utilisant le moteur de règles JESS.

Les méta-modèles sont représentés comme des ensembles de templates de faits et les modèles comme des ensembles de faits. La traduction de la règle est simple avec les particularités que la déclaration des constructions génériques, les déclarations de jointures, et les affectations sont fusionnées en clauses de faits. La figure III.8 montre la traduction en JESS de la règle de la figure III.7.

```
(defrule myRule

?fc1 <- (class (name ?c10))
?fc2 <- (class (name ?c20))
?fa1 <- (attribute (name ?a10) (class ?a11))
?fa2 <- (attribute (name ?a20) (class ?a21))

(test (and (eq ?a11 ?c10)
           (and (eq ?a21 ?c20) (eq ?a10 ?a20))))
=>

(bind ?v1 (new_name_class))

(modify ?fa1 (class ?v1))

(assert (class (name ?v1)))
(assert (inheritance (class ?c10) (superclass ?v1)))
(assert (inheritance (class ?c20) (superclass ?v1)))

(retract ?fa2)
)
```

Fig.III.8. Exemple d'une règle JESS

Notre fonction de fitness mesure la similarité entre les modèles produits par un ensemble de règles et les modèles cibles donnés dans les exemples. Considérez *E* l'ensemble des exemples e_i composés chacun d'une paire de modèles source et cible (ms_i, mt_i). La fitness *F(E, p)* d'un ensemble de règles *p* est définie comme la moyenne de l'exactitude de transformation $f\,(mt_i, p(ms_i))$ de tous les exemples e_i. L'exactitude de transformation $f\,(mt_i, p(ms_i))$ mesure à quel point le modèle produit $p(ms_i)$, obtenu en exécutant *p* sur le modèle source ms_i, est similaire au modèle cible mt_i de e_i.

La comparaison de deux modèles, c'est-à-dire, deux graphes avec des nœuds typés, est un problème difficile. Considérant que dans la dérivation de règles proposée, la fonction de fitness est évaluée pour chaque ensemble de règles, sur chaque exemple, et à chaque itération, cela ne peut pas permettre des comparaisons exhaustives de graphes. Au lieu de cela, une fonction simple et efficace *f* est utilisée. *f*, qui est une mesure de similarité de modèles, calcule la moyenne pondérée de l'exactitude de transformation par type de construction $t \in T_i$ (T_i est l'ensemble des types de constructions dans le modèle produit $p(ms_i)$ et le modèle cible mt_i).

Ceci a pour but d'éviter que l'évaluation soit biaisée vers des éléments fréquents d'un type en particulier, et par conséquence, donner la même importance à tous les types de constructions indépendamment de leurs fréquences. Formellement :

$$f(mt_i, p(ms_i)) = \sum_{t \in T_i} \frac{f_t(mt_i, p(ms_i))}{|T_i|}$$

f_t est définie comme la somme pondérée des pourcentages des constructions de type *t* qui sont, respectivement, en correspondance totale (fm_t), en correspondance partielle (pm_t), et pas en correspondance (nmt_t et nmp_t) :

$$f_t(\mathrm{mt_i}, \mathrm{p}(\mathrm{ms_i})) = \alpha \mathrm{fm_t} + \beta \mathrm{pm_t} + \gamma(1 - (\mathrm{nmt_t} + \mathrm{nmp_t})),\ \ \alpha + \beta + \gamma = 1$$

Pour chaque construction de type *t* dans le modèle cible, nous déterminons d'abord si elle est en correspondance totale avec une construction dans le modèle produit, c'est-à-dire qu'il existe dans le modèle produit une construction du même type qui a les mêmes valeurs des propriétés. Pour les constructions dans le modèle cible qui ne sont pas encore en correspondance totale, nous déterminons, dans une deuxième étape, si elles peuvent être en correspondance partielle.

Une construction est en correspondance partielle s'il existe dans le modèle produit une construction du même type qui n'est pas en correspondance totale dans la première étape. Finalement, la dernière étape consiste à classer toutes les constructions restantes comme pas en correspondance. Nous distinguons ici deux cas de figures : (1) les constructions existantes dans le modèle cible et qui n'apparaissent pas dans le modèle produit, leur pourcentage est désigné par (nmt_t), et (2) les constructions qui figurent sur le modèle produit et qui n'existaient pas dans le modèle cible, leur pourcentage est désigné par (nmp_t). Donc nmt_t représente le nombre de constructions qui devraient être générées dans le modèle produit alors qu'elles n'ont pas été générées, et inversement, nmp_t représente le nombre de constructions qui figurent dans le modèle produit, alors qu'elles ne le devraient pas, constituant ainsi des constructions supplémentaires et inutiles. Les quantités nmt_t et nmp_t reflètent le fait que la fitness d'un ensemble de règles est évaluée

également sur la base des constructions manquantes qu'il n'a pas générées pour nmt_t, et les constructions supplémentaires qu'il a créées au niveau du modèle produit pour nmp_t.

La somme ($nmt_t + nmp_t$) sert à exprimer la fitness en fonction de toutes les constructions qui ne sont pas en correspondance dans le modèle cible et le modèle produit, c'est-à-dire, les constructions manquantes avec les constructions supplémentaires. Enfin, (1- ($nmt_t + nmp_t$)) soustrait le nombre de toutes ces constructions de la fitness de l'ensemble de règles. Donc, si l'ensemble de règles en question ne génère aucune construction erronée, le terme γ(1- ($nmt_t + nmp_t$)) de sa fitness sera égal à la valeur complète de γ, et plus les valeurs de nmt_t et nmp_t augmentent, plus la fitness de l'ensemble de règles correspondant diminue.

Les coefficients α, β, et γ ont chacun un impact différent sur le processus de dérivation durant l'évolution. α, qui devrait avoir une valeur élevée (généralement 0,6), est utilisé pour favoriser les règles qui produisent correctement les constructions souhaitées. β, avec une valeur de 0,15, permet de donner des chances aux règles produisant les bons types des constructions souhaitées et aide à converger vers la solution optimale. Cependant, le refactoring en particulier, et les transformations endogènes en général, sont souvent appliquées sur des modèles contenant déjà beaucoup de constructions qui sont en correspondance partielle avec celles du modèle cible. Dans ce cas, il n'est pas intéressant d'avoir beaucoup d'ensembles de règles qui nous font, encore, tomber sur des constructions en correspondance partielle, et c'est la raison pour laquelle nous avons donné une petite valeur à β. Enfin, γ doit être fixé à une valeur moyenne de 0,25 pour donner plus de chances aux programmes qui effectuent beaucoup d'actions sur le modèle courant, en tenant toujours compte du fait que ces actions sont erronées.

Une bonne solution pourrait inclure des règles correctes qui génèrent les bonnes constructions, mais elle pourrait aussi contenir des règles redondantes ou des règles qui génèrent des constructions inutiles. Pour gérer cette situation, nous considérons la taille de l'ensemble des règles et le nombre de constructions génériques manipulées par les règles lors de la sélection de la meilleure solution. Par conséquent, même si une solution optimale est trouvée en termes d'exactitude, le processus d'évolution continue à rechercher des solutions aussi optimales, mais avec moins de règles et moins de constructions génériques.

III.6. PROGRAMMATION GÉNÉTIQUE ADAPTATIVE

Nous avons développé dans la section précédente, les idées principales et les démarches suivies dans notre approche afin d'adapter les étapes de la GP au problème d'apprentissage des RTs qui s'appliquent au refactoring des diagrammes de classes UML, à partir des exemples (MTBE). Toutefois, nous avons apporté des améliorations au niveau de l'algorithme de GP, qui minimisent sa complexité et optimisent considérablement le temps de convergence vers la solution optimale. Ces améliorations feront l'objet de cette section.

Tout d'abord, le processus de dérivation que nous proposons se distingue de la méthode classique de GP 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 méthode de dérivation s'est avérée efficace pour notre approche. Cependant, il y'a des cas

où on tombe sur des ensembles de règles qui sont déjà bons dans leurs grandes parties et seulement un de ces opérateurs suffit pour les améliorer, alors que l'application successive des deux diminue leur qualité et perturbe leur matériel génétique intéressant. Afin de prendre en charge cette situation, nous avons mis en place un mécanisme qui permet de sélectionner, aléatoirement, l'une des stratégies suivantes : (1) appliquer seulement l'opérateur de croisement avec une probabilité de 0.25, (2) appliquer uniquement l'opérateur de mutation avec une probabilité de 0.25 aussi, et (3) appliquer, successivement, le croisement et la mutation avec une probabilité de 0.5 pour cette dernière stratégie qui reste globalement plus efficace. Avec cette nouvelle politique, les ensembles de règles ont plus de chances de ne subir que l'opérateur (ou les opérateurs) génétique qui les améliorent, évitant ainsi la dégradation de leur qualité à cause d'un autre opérateur inapproprié.

Nous avons également établi une variante de l'opérateur de croisement qui choisit aléatoirement une seule règle de chacun des deux ensembles de règles sélectionnés pour subir cet opérateur. Ensuite les deux règles choisies sont échangées entre les deux ensembles de règles. Une alternative généralisée de cette variante, est le choix d'un nombre aléatoire et différent de règles dans chaque ensemble de règles sélectionné pour subir le croisement. Puis, échanger les règles choisies entre les deux ensembles de règles. Cette version du croisement s'avère, dans plusieurs cas, plus intéressante et efficace que celle décrite dans la section III.5.3.

Une autre situation pour laquelle une amélioration serait hautement bénéfique, est le chargement des différentes générations par des ensembles de règles ressemblants et proches de la solution optimale. Ce problème survient, par exemple, lorsque la fitness d'un ensemble de règles qui vient de subir une évolution devient très grande par rapport à celles des autres. À ce moment, les populations des générations suivantes seront saturées par ce genre d'ensembles de règles qui n'évoluent qu'entre eux, sans laisser de chance au nouveaux ensembles de règles, effectuant d'autres refactorings intéressants, d'apparaitre et de participer à l'évolution. Cela empêche la convergence vers la solution optimale globale. Afin de résoudre ce problème, nous proposons d'ajouter une certaine diversité aux ensembles de règles des différentes générations en introduisant dans chaque population, de nouveaux ensembles de règles créés aléatoirement. Le nombre de ces ensembles est égal à environ 1 ou 2% du nombre total des ensembles de la population.

La fonction de fitness peut, à son tour, être améliorée en introduisant un nouveau coefficient δ et en éclatant le terme $\gamma(1- (nmt_t + nmp_t))$, ce qui donne la fonction de fitness suivante :

$$f_t(\mathrm{mt_i}, \mathrm{p}(\mathrm{ms_i})) = \alpha \mathrm{fm_t} + \beta \mathrm{pm_t} + \gamma(1 - \mathrm{nmt_t}) + \delta(1 - \mathrm{nmp_t}),\ \alpha + \beta + \gamma + \delta = 1$$

Le coefficient δ doit être fixé à une valeur supérieure à celle de γ afin de favoriser les ensembles de règles qui produisent des constructions supplémentaires, plutôt que ceux qui ne génèrent pas les constructions souhaitées (constructions manquantes). Nous avons opté pour ce choix car nous considérons qu'un ensemble de règles qui produit les bonnes constructions avec d'autres supplémentaires, est meilleur qu'un autre ensemble qui laisse des constructions manquantes.

Dans notre approche, nous considérons la mutation comme étant le mécanisme principal permettant d'explorer l'espace de recherche et d'introduire un nouveau matériel génétique pendant

que la population évolue. Les mutations qui s'appliquent au niveau d'un ensemble de règles, et celles qui s'appliquent au niveau d'une règle (voir la section III.5.3) peuvent être sélectives, c'est-à-dire, que le choix de la mutation à appliquer sur un ensemble de règles dépendra des valeurs fm_t, pm_t, nmt_t, et nmp_t qui constituent sa fitness.

III.6.1. Mutations sélectives

L'idée de base de cette technique consiste à définir, au préalable, des seuils pour chacune des valeurs fm_t, pm_t, nmt_t, et nmp_t. Ensuite, nous testons ces valeurs pour l'ensemble de règles qui a été sélectionné pour subir une mutation. Selon les valeurs qui dépassent leurs seuils, un ensemble de mutations appropriées est choisi. Puis une mutation est sélectionnée aléatoirement de chaque ensemble et ajoutée à la liste des mutations candidates pour être appliquées à l'ensemble de règles concerné. Afin d'illustrer ce mécanisme, imaginons un ensemble de règles *e* désigné pour subir une mutation. Si, par exemple, la valeur nmp_t de la fonction fitness de *e* dépassent son seuil, alors ceci signifie que *e* génère un nombre relativement grand de constructions supplémentaires et inutiles dans le modèle produit. De ce fait, deux mutations sont convenables à son cas : la suppression d'une construction générique du pattern *AP* de l'une de ses règles, ou l'ajout d'une construction générique au pattern *RP*. L'une de ces mutations est sélectionnée aléatoirement et ajoutée à la liste des mutations candidates de *e*. Nous faisons de même pour chaque valeur ou combinaison de ces valeurs. Une fois la liste des mutations candidates est complétée, l'une d'elles est choisie aléatoirement pour être appliquée à l'ensemble de règles en question.

Une variante de cette technique est l'application de toute ou d'une partie aléatoire de la liste des mutations candidates. Cette variante est avantageuse dans la mesure où elle permet d'accélérer la convergence vers la solution optimale en procédant à toutes les mutations intéressantes de l'ensemble de règles concerné.

III.7. CONCLUSION

Ce chapitre a permis d'expliciter en détail, notre conception du processus de dérivation des RTs à partir des exemples, dédié au problème du refactoring des diagrammes de classes UML. La nature et les particularités de cette transformation ont impliqué un grand effort d'adaptation de la GP. Nous tenons également à signaler que JESS était un excellent choix pour notre approche, qui a bénéficié de son moteur d'inférence basé sur l'algorithme Rete et reconnu pour son efficacité dans la gestion des systèmes de règles non monotones. Les améliorations que nous avons apportées à notre processus de dérivation se sont avérées considérablement utiles, en particulier, lorsqu'il s'agit de l'optimisation du temps de convergence vers la solution optimale. La mise en œuvre de notre approche fera l'objet du chapitre suivant.

CHAPITRE IV : MISE EN ŒUVRE DE L'APPROCHE

IV.1. INTRODUCTION

La conception de notre approche a été décrite en détail, au niveau du chapitre précédent. Afin de réaliser cette approche, nous avons utilisé le langage Java sous l'environnement Eclipse, pour l'implémentation de notre version adaptée de l'algorithme de GP. Le langage et le moteur de règles JESS a été intégré dans Java grâce à son API spéciale, et ceci, dans le but d'écrire et d'exécuter les RTs dérivées.

Ce chapitre sera consacré à la description et la configuration nécessaire des outils choisis pour mettre en œuvre notre approche. Ensuite, nous abordons les possibilités de manipulation de JESS en ligne de commande, et via Java. Ensuite, nous validons notre approche sur un exemple concret d'un diagramme de classes mal construit. Enfin, nous faisons une évaluation quantitative et qualitative du processus de dérivation, à la base des RTs obtenues et qui font le refactoring du diagramme de l'exemple de validation.

IV.2. CHOIX ET CONFIGURATION DES OUTILS

Pour la réalisation de ce projet, nous avons choisi d'utiliser l'environnement Eclipse, et l'outil JESS pour l'écriture et l'exécution des RTs dérivées (figure IV.1). Dans ce qui suit, nous décrivons en détail ces outils ainsi que leurs configurations.

Fig.IV.1. Les outils choisis, Eclipse et JESS

IV.2.1. L'environnement Eclipse

Eclipse est un « Environnement de Développement Intégré » (EDI), ou « Integrated Development Environment » (IDE) en anglais. Il est libre, extensible, universel et polyvalent. Eclipse est principalement écrit en Java qui est également utilisé pour écrire des extensions. Donc, pour utiliser Eclipse, il est nécessaire d'avoir un environnement d'exécution Java « Java Runtime Environment » (JRE) installé sur la machine [Huo 08]. Eclipse est disponible pour les plates-formes Windows 32/64 bits, Mac 32/64 bits et Linux 32/64 bits, et la version officielle en cours, que nous utilisons dans ce travail, est Eclipse Luna 4.4 téléchargeable depuis le site officiel **www.eclipse.org**.

La spécificité d'Eclipse vient du fait que son architecture est totalement développée autour de la notion de « plug-in ». Cela signifie que toutes les fonctionnalités de celui-ci sont développées

en tant que plug-ins [Dou 08]. Autrement dit, si nous voulons ajouter des fonctionnalités à Eclipse, il suffit de télécharger et d'ajouter les plug-ins correspondants. Cette propriété de modularité fait d'Eclipse une boite à outils facilement modifiable et extensible. De plus, l'utilisation de l'environnement Eclipse est très répandue dans les laboratoires de recherche en informatique autour du monde. Tout cela a motivé notre choix d'Eclipse pour la réalisation de ce projet.

Eclipse ne nécessite pas d'installation proprement dite. Il faut d'abord vérifier la disponibilité du JRE sur la machine, et télécharger Eclipse sous forme d'une archive. Ensuite, il faut juste décompresser l'archive téléchargée et Eclipse devient opérationnel.

IV.2.2. JESS

JESS représente un moteur de règles écrit entièrement en Java au « laboratoires nationaux de Sandia »[1] au Canada. Il supporte le développement des systèmes experts à base de règles, qui peuvent être efficacement couplés à un langage de programmation puissant et portable tel que Java [Fri 08].

Nous utilisons dans ce projet JESS en tant qu'environnement de transformation de modèles. Ce choix est motivé par l'utilisation efficace et réussie de cet outil dans des travaux récents, en l'occurrence, celui de Faunes et al [Fau 13b]. De plus, JESS est plus facile à lier et à manipuler depuis Java que ses semblables (tels que Prolog), et ceci, grâce à une bibliothèque prédéfinie, gratuite, et développée à cette fin.

IV.2.3. Préparation de JESS

JESS est une bibliothèque logicielle qui sert d'interpréteur pour un langage, qu'on appelle généralement : le langage JESS. L'utilisation de cette bibliothèque nécessite l'installation d'une machine virtuelle Java (JVM) [Fri 03b]. La version de JESS que nous utilisons dans ce travail est la version 7.1 qui est compatible avec toutes les versions Java en partant du JDK 1.4 jusqu'à la dernière version qui existe aujourd'hui.

JESS est fourni en un seul fichier **.zip** qui contient tout ce dont nous avons besoin pour utiliser JESS sous Windows, UNIX, ou Macintosh (sauf la JVM que nous devons installer nous-même). Lorsque JESS est décompressé, nous devrions avoir un répertoire nommé *Jess71p2/*. Ce répertoire contient plusieurs fichiers et sous-répertoires. Nous en citons les plus importants parmi eux :

- ***bin*** : Un répertoire contenant un fichier batch de Windows (**jess.bat**) et un script de shell UNIX (**jess**) que nous pouvons utiliser pour démarrer l'invite de commandes JESS.
- ***lib*** : Un répertoire contenant JESS lui-même, sous forme d'une archive Java (Java Archive ou JAR). Ce répertoire contient également le JSR-94 dans le fichier **jsr94.jar**, qui n'est rien d'autre que l'API **javax.rules**.
- ***eclipse*** : C'est le JessDE, l'environnement de développement intégré de JESS, fourni en un ensemble de plugins pour Eclipse.

[1] http://www.sandia.gov/

IV.2.4. Intégration de JESS en Java

Afin de pouvoir utiliser la bibliothèque JESS à partir de notre application Java, une certaine configuration de l'environnement d'exécution Java (***JRE***) est nécessaire. Cette configuration consiste à installer une extension standard du JRE comme suit :

- Accéder au répertoire ***« Jess71p2/lib »*** de la distribution JESS.
- Copier l'archive Java ***« jess.jar »***.
- Aller dans le répertoire ***« jre/lib/ext »*** et coller l'archive ***« jess.jar »*** dedans (Le répertoire ***« jre/ »*** se trouve dans le répertoire ***« Java/ »*** où nous avons installé Java, il s'agit généralement du répertoire ***« C:/Program Files/Java/jre/ »***).
- Pour utiliser l'API ***javax.rules***, il faut faire de même que l'archive ***« jess.jar »*** avec l'archive ***« jsr94.jar »***.

IV.3. JESS EN LIGNE DE COMMANDE

JESS est doté d'une interface interactive pour l'utilisation en ligne de commande. La distribution inclut deux scripts que nous pouvons exécuter pour obtenir l'invite de commandes de JESS : ***« jess »*** pour UNIX et ***« jess.bat »*** pour Windows. Les deux se trouvent dans le répertoire « ***bin/ »***. Nous exécutons alors celui qui correspond à notre système d'exploitation (Windows dans notre cas). Pour exécuter un fichier du code JESS en ligne de commande, nous utilisons la commande ***« batch »***, comme illustré dans la figure IV.2.

```
Jess, the Rule Engine for the Java Platform
Copyright (C) 2008 Sandia Corporation
Jess Version 7.1p2 11/5/2008

Jess> (batch "nomDuFichier.clp")
```

Fig.IV.2. Lancement d'un programme JESS en ligne de commande

IV.4. MANIPULATION DE JESS VIA JAVA

Le cœur de la bibliothèque JESS est la classe *jess.Rete*. Une instance de cette classe est, dans un certain sens, une instance de JESS. Chaque objet de *jess.Rete* a sa propre mémoire de travail, sa liste de règles, et son ensemble de fonctions [Fri 03]. La classe *Rete* dispose de méthodes pour ajouter, trouver, et supprimer les faits, les règles, et les fonctions.

Le moyen le plus simple pour manipuler JESS depuis Java est l'utilisation de la méthode *executeCommand* de la classe *Rete* qui reçoit une chaine de caractères en paramètre, et retourne un objet de la classe *jess.Value*. La chaine de caractère est interprétée comme une expression du langage JESS et la valeur de retour est le résultat de l'évaluation de cette expression [Fri 03]. Par exemple, pour ajouter un nouveau fait à la mémoire de travail de JESS, et avoir un accès à l'objet de la classe *jess.Fact*, nous devons faire comme dans l'exemple de la figure IV.3.

Afin de résoudre la valeur de *jess.Value*, nous avons besoin d'un objet *jess.Context*. La méthode *getGlobalContext* de la classe *jess.Rete* est un moyen approprié pour en obtenir un. Les

commandes exécutées via *executeCommand* référencient des variables JESS, et elles sont interprétées dans le même contexte [Fri 03].

```
import jess.Rete;
import jess.JessException;
import jess.Value;
import jess.Fact;
// …
public class Engine {
        private Rete engine;
        private String command = "(assert (class (name Personne)))";
        // …
        public Engine() throws JessException {
                engine = new Rete();
        }
        // …
        public Value executeMyCommand(String command) {
                return this.engine.executeCommand (command);
        }
        public Fact getFactValue (Value v) {
                return v.factValue(this.engine.getGlobalContext());
        }
        // …
}
```

Fig.IV.3. Exemple de manipulation de JESS via Java

IV.5. VALIDATION

Notre approche a été testée et validée sur un ensemble de diagrammes de classes mal construits, et elle a donné de très bons résultats en dérivants les règles de transformations qui effectuent les refactorings appropriés sur chacun de ces diagrammes, améliorant ainsi leur qualité. Dans ce qui suit, nous donnons un exemple de l'un des diagrammes que nous avons utilisés dans la validation de notre approche. Nous mettons également en évidence, l'ensemble de règles final (meilleure solution) obtenu en exécutant notre processus de dérivation sur le diagramme en question.

Le diagramme de classes initial est celui de la figure IV.4, contenant plusieurs mauvaises odeurs (*Bad Smells*) qui nécessitent différents refactorings. Tout d'abord, l'association qui lie les classes « ClassD » et « ClassC » devrait être supprimée car la même association lie également leurs classes mères respectives « ClassB » et « ClassA». De même, l'association qui lie les classes « ClassG » et « ClassD », devrait à son tour être supprimée car une même association existe entre

leurs classes mères respectives « ClassF » et « ClassB ». Enfin, les deux associations qui lient les classes « ClassG » et « ClassH » à la classe « ClassB », devraient être supprimées parce que la même association existe entre leur classe mère « ClassF » et la classe « ClassB ».

L'attribut « Att1 » devrait être supprimé de la classe « ClassB », car il figure déjà dans la classe mère « ClassE ». Il est de même pour l'attribut « Att2 » de la classe « ClassH » qui existe déjà das la classe mère « ClassF ».

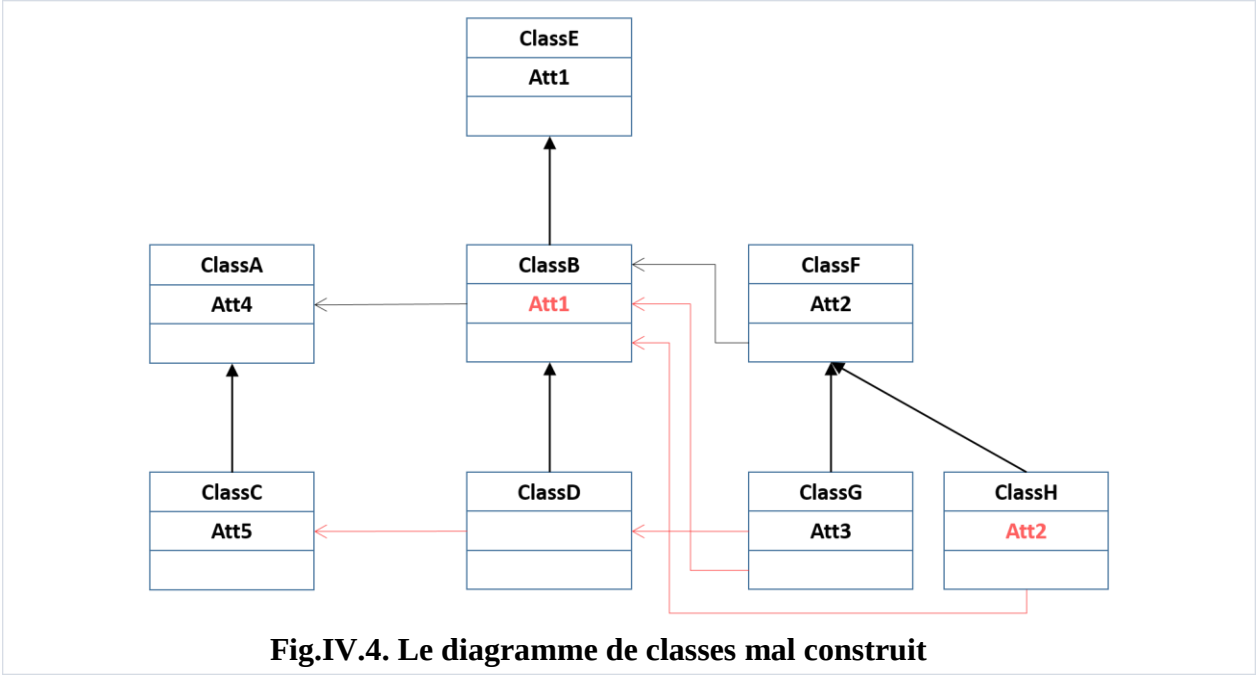


Fig.IV.4. Le diagramme de classes mal construit

Les règles obtenues après l'exécution de notre processus sur l'exemple ci-dessus, sont exposées dans la figure IV.5. Puisque les algorithmes basés sur la GP sont de nature probabiliste, et pour l'exécution de notre approche, nous avons fixé le nombre d'itérations à 3000, la taille de la population à 100, et l'élitisme à 10 programmes. La probabilité du croisement a été fixée à 0.9, et celle de la mutation à 0.9. Par opposition aux algorithmes génétiques classiques, avoir une probabilité élevée de mutation est très fréquent dans les algorithmes de GP. Les poids α, β, et γ de la fonction fitness ont été fixés à 0.6, 0.15, et 0.25, respectivement.

IV.6. ÉVALUATION

Nous évaluons notre approche de deux manières : quantitativement et qualitativement. L'évaluation quantitative permet de déterminer à quel point notre approche génère les règles qui transforment correctement l'ensemble donné des exemples. D'autre part, l'évaluation qualitative aide à savoir si les exemples sont correctement transformés et si les règles dérivées sont celles aide à savoir si les exemples sont correctement transformés et si les règles dérivées sont celles prévues.

```
(defrule R_13663
?fs0 <- (association1n (classfr ?s00)(classto ?s01))
?fs1 <- (association1n (classfr ?s10)(classto ?s11))
?fi0 <- (inheritance (class ?i00)(superclass ?i01))
?fc0 <- (class (name ?c00))
?fc1 <- (class (name ?c10))
?fc2 <- (class (name ?c20))
(test  (and (eq ?s00 ?c00)(and (eq ?s01 ?c20)(and (eq ?s10 ?c10)
          (and (eq ?s11 ?c20)(and (eq ?i00 ?c00)(eq ?i01 ?c10)))))))
=>
(retract ?fs0)
)

(defrule R_13664
?fa1 <- (attribute (name ?a10)(class ?a11))
?fa0 <- (attribute (name ?a00)(class ?a01))
?fi0 <- (inheritance (class ?i00)(superclass ?i01))
?fc1 <- (class (name ?c10))
?fc0 <- (class (name ?c00))
(test    (and  (and (eq ?a11 ?c00)(and (eq ?a01 ?c10)
                  (and (eq ?i00 ?c10)(eq ?i01 ?c00))))(eq ?a10 ?a00)))
=>
(modify ?fa0 (name ?a10)(class ?c00))
)

(defrule R_13666
?fs0 <- (association1n (classfr ?s00)(classto ?s01))
?fs1 <- (association1n (classfr ?s10)(classto ?s11))
?fi0 <- (inheritance (class ?i00)(superclass ?i01))
?fi1 <- (inheritance (class ?i10)(superclass ?i11))
?fc0 <- (class (name ?c00))
?fc1 <- (class (name ?c10))
?fc2 <- (class (name ?c20))
?fc3 <- (class (name ?c30))
(test    (and (eq ?s00 ?c00)(and (eq ?s01 ?c10)(and (eq ?s10 ?c20)
          (and (eq ?s11 ?c30)(and (eq ?i00 ?c00)(and (eq ?i01 ?c20)
          (and (eq ?i10 ?c10)(eq ?i11 ?c30)))))))))
=>
(retract ?fs0)
)
```

Fig.IV.5. L'ensemble des règles de la meilleure solution

IV.6.1. Évaluation Quantitative

Le graphe de la figure IV.6 illustre l'évolution de la solution pour le refactoring du diagramme de classes de la figure IV.4.

Ce graphe comporte trois courbes qui représentent, respectivement, la valeur de la fonction fitness (F), le nombre des constructions en correspondance totale (FM) (l'axe vertical à gauche), et le nombre de règles dans la solution (RN) au fil des générations (l'axe vertical à droite). Les courbes correspondent aux meilleurs ensembles de règles des générations identifiées sur l'axe horizontal.

Dans la génération initiale, qui est considérée comme une génération aléatoire de programmes de transformation, la plupart des constructions ont été correctement générées (FM = 0.80). La solution optimale en termes de FM a été trouvée à la 73ème génération, avec 3 règles. Une fois la solution avec FM = 1 est trouvée, le processus de dérivation poursuit son exécution. Donc, la solution courante est remplacée si une nouvelle solution avec moins de règles est trouvée. Depuis la 73ème génération, nous n'avons pas observé d'autres améliorations en termes de nombre de règles.

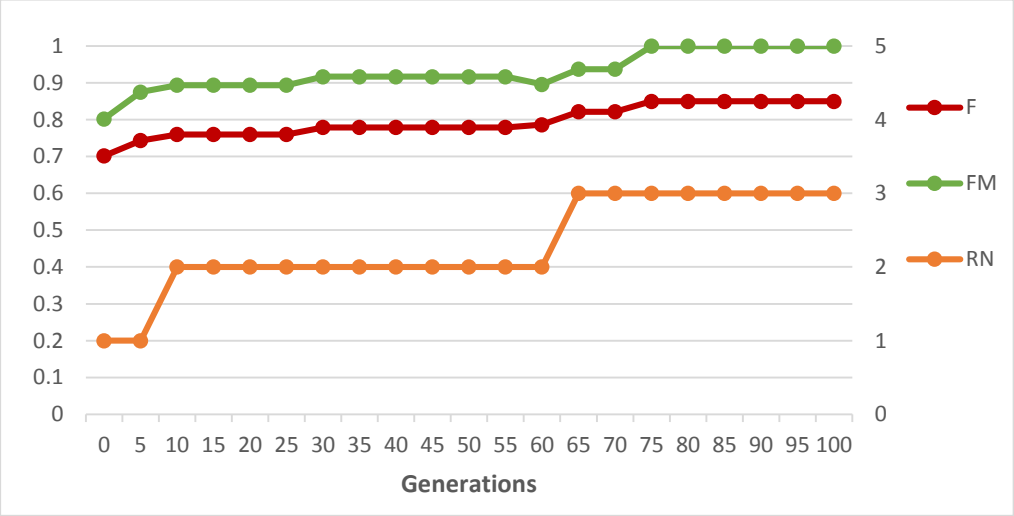


Fig.IV.6. L'évolution de la recherche de la meilleure solution

Notre processus de dérivation a été lancé sur une machine i5 CPU @ 2.50 GHz avec 6 Go de RAM. Cependant, le temps d'exécution est relativement long. Ceci est le résultat du nombre considérable de programmes que le moteur JESS exécute. De plus, ce temps n'est pas excessif, sachant que le processus de dérivation des RTs n'est pas censé être exécuté fréquemment.

IV.6.2. Évaluation Qualitative

L'obtention des transformations correctes des exemples ne signifie pas nécessairement que nous avons dérivé les règles prévues. Pour un échantillon limité de cas de tests, les mêmes valeurs de sortie pourraient être produites par différents programmes. Ainsi, pour évaluer nos résultats qualitativement, nous avons besoin de comparer les règles produites avec celles utilisées pour générer les exemples (celles prévues).

Nous estimons que l'ensemble des règles dérivées en testant notre approche sur l'exemple précédent (voir figure IV.4), est de bonne qualité. Cet ensemble contient trois règles dont deux sont exactement comme les règles prévues qui ont généré l'exemple. La troisième règle effectue correctement la transformation au même titre que celle prévue, mais d'une manière différente. Nous avons prévu qu'un attribut soit supprimé directement par un « retract », alors qu'il a été supprimé par un « modify ». En JESS, la modification d'un fait, qui donne un autre fait existant déjà dans la mémoire de travail signifie la suppression du fait modifié. Pour notre approche, ce cas de figure ne constitue pas un défaut de qualité, mais il reflète la flexibilité de notre structure de règles. Toutefois, si nous tenons à obtenir des règles écrites exactement comme celles prévues, nous pouvons inclure une petite étape de raffinement automatique qui prend en charge cette politique de JESS. L'étape de raffinement pourrait donc facilement remplacer cette construction de « modify » dans notre règle, par une construction de suppression de l'attribut courant, et à ce moment-là, nous aurons obtenu la même règle écrite de la même façon que celle prévue.

IV.7. CONCLUSION

En guise de conclusion, ce dernier chapitre a explicité, dans un premier temps, le choix et la configuration des outils de développement utilisés dans la réalisation de ce projet. La mise en œuvre de notre approche nous a permis, en particulier, de nous familiariser avec le langage et le moteur de règles puissant JESS. Nous avons également eu l'occasion d'explorer les possibilités offertes par ce langage, notamment, sa manipulation via Java.

L'exécution de notre processus de dérivation sur un exemple concret d'un diagramme de classes mal construit, a révélé l'efficacité de notre approche dans l'obtention des bonnes règles de transformation qui effectuent les refactorings appropriés de différentes manières. Ceci reflète la richesse et la diversité des solutions que notre approche peut générer.

CONCLUSION GÉNÉRALE ET PERSPECTIVES

Le MDE est un paradigme qui promet de réduire la complexité du logiciel par l'utilisation intensive de modèles et des transformations automatiques entre modèles (TMs). La TM est un concept essentiel à la concrétisation de la vision dirigée par les modèles. Elle apporte l'automatisation nécessaire aux activités du MDE en permettant de maintenir la cohérence des différents modèles manipulés durant les processus de développement, et de leur appliquer différents traitements tels que le raffinement et le refactoring, en vue d'améliorer leur qualité. La TM doit être automatique pour bénéficier des avantages du paradigme MDE. Cependant, l'automatisation implique la récolte des connaissances sur la façon dont la TM doit être effectuée, et ces connaissances sont souvent non disponibles ou difficiles à obtenir.

Les approches de TM par l'exemple viennent pallier cette situation, en particulier, le MTBE qui vise à apprendre automatiquement les TMs à partir d'un ensemble de paires de modèles sources et cibles fournis en guise d'exemples. Un mécanisme d'apprentissage est ensuite utilisé pour dériver les règles de transformation (RTs). Malheureusement, très peu de travaux en MTBE prennent en charge les TMs endogènes, et encore moins, le refactoring des modèles, qualifié de l'une des TMs les plus importantes dans le cycle de vie du logiciel.

Nous avons proposé dans le cadre de ce projet un processus d'apprentissage d'une transformation endogène connue sous le nom de « refactoring des modèles », et nous nous sommes intéressés au refactoring des diagrammes de classes UML. L'approche proposée est basée sur la programmation génétique (GP), guidée par des exemples de refactorings précédents, et ne nécessite pas des traces fines de transformation. Le type de TM que nous avons pris en charge, à savoir une TM *in-place*, nous a impliqué un effort considérable d'adaptation, en particulier, au niveau du choix de la structure et de l'encodage des RTs. En outre, les éléments à fournir à l'algorithme génétique ont été conçus et adaptés en fonction du problème de refactoring des diagrammes de classes, notamment, l'ensemble des fonctions et des terminaux utilisés, et les paramètres choisis de l'approche, de sorte que l'algorithme puisse produire la solution désirée. Les opérateurs génétiques qui améliorent itérativement la solution obtenue ont été également implémentés dans ce sens, et la fonction de fitness est définie pour évaluer la qualité des RTs générée sur la base de similarité entre les modèles produits et ceux décrits à travers les sorties des exemples.

Les règles dérivées sont de qualité, ce qui reflète la réussite de notre approche. Cependant, il existe de nombreuses pistes d'amélioration qui méritent d'être explorées. Bien que le temps de réponse soit relativement long à cause de l'exécution d'un grand nombre de programmes sous le moteur JESS, ceci n'est pas un sérieux défaut de notre approche car, dans la pratique, on n'effectue pas fréquemment le refactoring des modèles pour les grandes applications au sein des entreprises. En outre, Il serait énormément avantageux d'étendre cette approche pour permettre de dériver des règles avec des conditions complexes. Nous signalons ici, que cette étape est déjà entamée avec succès. Il serait également intéressant de développer un outil graphique pour mieux se servir de notre application et la rendre plus ergonomique pour ses futurs utilisateurs. Enfin, une perspective hautement souhaitable, serait de généraliser notre approche pour permettre le refactoring des autres diagrammes UML.

BIBLIOGRAPHIE

[Bab 10] J. Babau, M. Blay-Fornarino, J. Champeau, S. Robert, A. Sabetta. Model-Driven Engineering for Distributed Real-Time Systems, 2010.

[Bak 14] I. Baki. Une approche heuristique pour l'apprentissage de transformations de modèles complexes à partir d'exemples. Université de Montréal, 2014.

[Bal 09] Z. Balogh, D. Varró. Model transformation by example using inductive logic programming. Soft. and Syst. Modeling, 2009.

[Ban 98] W. Banzhaf. An Introduction on the Evolution of Computer Programs, 1998.

[Ben 08] B. Ben Ammar, M. Tahar Bhiri, J. Souquières. Schéma de refactoring de diagrammes de classes basé sur la notion de délégation. ERTSI, 2008.

[Bie 10] M. Biehl. Literature Study on Model Transformations. Stockholm, 2010.

[Bon 06] L. Bondé. Transformations de Modèles et Interopérabilité dans la Conception de Systèmes Hétérogènes sur Puce à Base d'IP. Université Lille, 2006.

[Bou 06] S. Bouden. Etude de la traçabilité entre refactorisations du modèle de classes et refactorisations du code. Université de Montréal, 2006.

[Cro 00] L. Crowley. Systèmes de productions : CLIPS 6.0. ENSIMAG, 2000.

[Dia 09] S. Diaw, R. Lbath, B. Coulette. Etat de l'art sur le développement logiciel dirigé par les modèles. Laboratoire IRIT, Université de Toulouse 2, 2009.

[Dje 10] S. Djerir, E. Zimanyi. Étude et implémentation d'un moteur d'inférence dans une architecture orientée événements. Université Libre de Bruxelles, 2010.

[Dol 10] X. Dolques, M. Huchard, C. Nebut, P. Reitz. Learning transformation rules from transformation examples: An approach based on relational concept analysis, 2010.

[Fau 13] M. Faunes. Improving Automation in Model-Driven Engineering using Examples. Université de Montréal, 2013.

[Fau 13b] M. Faunes, H. Sahraoui, M. Boukadoum. Genetic-Programming Approach to Learn Model Transformation Rules from Examples, 2013.

[Fri 03] E. Friedman-Hill. Rule-Based Systems in Java: JESS in action, 2003.

[Fri 08] E. Friedman-Hill. Jess The Rule Engine for the Java Platform. Sandia National Laboratories, 2008.

[Gar 08] I. García-Magariño, S. Rougemaille, R.Fernández, F. Migeon. A Tool for Generating Model Transformations By-Example in Multi-Agent Systems. Univ. Toulouse, 2008.

[Gar 09] I. García-Magariño, J. Gómez-Sanz. An algorithm for generating many-to-many transformation rules in several model transformation languages, 2009.

[Gru 98] J. Grundy, J. Hosking, W. Mugridge. Inconsistency Management for Multiple-View Software Development Environments. IEEE Trans. Software Engineering, 1998.

[Hil 03] E. Hill. Jess in Action: Java Rule-Based Systems, 2003.

[Hug 14] J. Hugues, P. Siron. MDE for Simulation: the PRISE platform. ISAE/DMIA, 2014.

[Huo 08] S. Huot. Java - Introduction à Eclipse. iUT ORSAY, 2008.

[Inria 10] INRIA. Model Driven Language Engineering using Kermeta. www.inria.fr, 2010.

[Jéz 12] J. Jézéquel, B. Combemale, D. Vojtisek. IDM : des concepts à la pratique, 2012.

[Jéz 13] J. Jézéquel. Introduction to Model-Driven Engineering. Univ. Rennes 1, INRIA, 2013.

[Kes 08] M. Kessentini, H. Sahraoui, M. Boukadoum. Model transformation as an optimization problem. Springer Heidelberg, 2008.

[Kes 10] M. Kessentini. Transformation by Example. Université de Montréal, 2010.
[Kes 10b] M. Kessentini, M. Wimmer, H. Sahraoui, M. Boukadoum. Generating transformation rules from examples for behavioral models, 2010.
[Koz 99] J. Koza. Genetic programming III: Darwinian invention and problem solving, 1999.
[Koz 05] J. Koza, R. Poli. Genetic programming. In: Search Methodologies, 2005.
[Kub 06] S. Kubicki. Assister la coordination flexible de l'activité de construction de bâtiments. Une approche par les modèles pour la proposition d'outils de visualisation du contexte de coopération. Université Henri Poincaré - Nancy I, 2006.
[Lan 10] P. Langer, M. Wimmer. Model-to-model transformations by demonstration, 2010.
[Le Noir 13] J. Le Noir. Les langages de modélisation en ingénierie système. THALES, 2013.
[Luk 98] S. Luke. A revised comparison of crossover and mutation in genetic programming, 1998.
[Men 07] T. Mens. Refactoring des modèles : concepts et défis. Univ. Mons-Hainaut, Belgique, 2007.
[Moh 10] P. Mohagheghi, W. Gilani, A. Stefanescu. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases, 2010.
[Mon 08] M. Monperrus. La mesure des modèles par les modèles. Université Rennes 1, 2008.
[Moo 02] G. Moore. Marketing and Selling Disruptive Products to Mainstream Customers, 2002.
[Mul 06] P. Muller. De la modélisation objet des logiciels à la métamodélisation des langages informatiques. Software Engineering. Université Rennes 1, 2006.
[Pal 12] M. Palyart-Lamarche. Une approche basée sur les modèles pour le développement d'applications de simulation numérique haute-performance. Université Toulouse, 2012.
[Pir 14] A. Pires Fernandes. Amélioration des processus de vérification de programmes par combinaison des méthodes formelles avec l'ingénierie dirigée par les modèles. ISAE, 2014.
[Rag 11] I. Ragoubi. Motifs de transformation de métamodèles. Software Engineering, 2011.
[Saad 12] H. Saada, X. Dolques, M. Huchard, C. Nebut, H. Sahraoui. Generation of operational transformation rules from examples of model transformations, 2012.
[Sch 06] D. Schmidt. Model-driven engineering. IEEE Computer, 2006.
[Sél 05] D. Sélim. Etude et automatisation de refactorings pour le langage, 2005.
[She 92] T. Sheridan. Telerobotics, automation and human supervisory control. MIT, 1992.
[Str 08] M. Strommer, M. Wimmer. A framework for model transformation by-example: Concepts and tool support. Springer Heidelberg, 2008.
[Sun 09] Y. Sun. Model Transformation by Demonstration. Univ. Alabama, USA, 2009.
[Tem 12] H. Temate. Des langages de modélisation dédiés aux environnements de méta-modélisation dédiés. Institut National Polytechnique de Toulouse, 2012.
[Thi 10] P. Thinga. Transformation générique de modèles. Univ. Bretagne Occidentale, 2010.
[Tru 11] S. Truptil. Etude de l'approche de l'interopérabilité par médiation dans le cadre d'une dynamique de collaboration. Institut National Polytechnique de Toulouse, 2011.
[Van 05] R. Van Straeten. Inconsistency Management in Model-Driven Engineering, 2005.
[Var 06] D. Varró. Model transformation by example. Springer Heidelberg, 2006.
[Wei 98] Z. Wei, A. Macwan, P. Wieringa. A quantitative measure for degree of automation and its relation to system performance and mental load, 1998.
[Wim 07] M. Wimmer, M. Strommer. Towards model transformation generation by-example, 2007.
[Yu 10] X. Yu, M. Gen. Introduction to evolutionary algorithms. Springer, 2010.