Université de Nantes

Mémoire de Master

Génération automatique de modèles de test pour les transformations de modèles en exploitant l'analyse de mutation

Auteur :
Thomas Degueule

Sous la direction de :
Jean-Marie Mottu
Gerson Sunyé

Mémoire soumis pour l'obtention du grade de Master en Informatique

27 juin 2013

Sommaire

Chapitre 1

Introduction

Ces dernières années, le développement de l'Internet, des applications réparties, embarquées ou autogérées a considérablement complexifié le développement des systèmes informatiques. Ceux-ci sont utilisés dans des environnements de plus en plus variés et contraints (terminaux mobiles, automobiles, robots) et dans des contextes de plus en plus critiques (aérospatial, médical, militaire, nucléaire). De plus, au cours de leur durée de vie, les logiciels sont soumis à des évolutions constantes pour satisfaire les besoins des utilisateurs et s'adapter à de nouvelles plateformes. Le monde académique et le monde industriel sont ainsi amenés à repenser les processus de production des logiciels afin de les adapter à ces nouveaux enjeux.

Une vision pragmatique pour s'adapter à ces changements est de permettre la production de logiciels de manière automatique afin de prendre en compte les évolutions inévitables dont ceux-ci font l'objet. Depuis toujours, les modèles ont été utilisés en informatique comme support de la réflexion. Cependant, ils ont souvent été relégués au rang de simple documentation. L'ingénierie dirigée par les modèles vise à placer les modèles au cœur du processus de développement pour en faire les éléments principaux par lesquels les applications sont générées. La prise en compte du changement au niveau des modèles et la production automatique de systèmes par génération depuis ces modèles semblent être une issue prometteuse pour répondre aux contraintes d'évolution.

Au cœur de l'ingénierie des modèles se situent les transformations de modèles. Elles réalisent l'automatisation des étapes essentielles de la production de logiciels : raffinement et composition de modèles, rétroingénierie, génération de code et de documentation, etc.. Ces transformations sont destinées à être utilisées de manière répétitive par les équipes de développement, par exemple au sein d'ateliers logiciels. Par conséquent, il est crucial qu'elles soit validées et testées de sorte que le développeur puisse avoir pleine confiance en elles.

Le test de logiciel est une discipline intensément étudiée dans la littérature et différentes techniques pour la génération automatique de données de test ont été définies. Les transformations de modèles sont des programmes classiques, mais présentent des spécificités qui empêchent l'utilisation directe des techniques de test traditionnelles. Parmi les techniques de test éprouvées, nous nous intéressons ici à l'analyse de mutation et proposons des méthodes pour l'adapter aux

spécificités des transformations de modèles. De cette manière, nous permettons la génération de modèles de test adéquats à la mutation. Nous montrons que notre approche permet de soulager le travail du testeur en lui fournissant toutes les informations nécessaires pour la génération de tels modèles, et, lorsque cela est possible, en générant directement ces modèles. L'algorithme présenté est expérimenté sur un cas d'étude complet qui montre son efficacité. Enfin, nous présentons différents outils développés mettant en œuvre ces concepts.

Nous présentons au chapitre 2 l'ingénierie des modèles et les transformations de modèles, ainsi qu'un état de l'art de l'analyse de mutation et du test des transformations de modèles. Le chapitre 3 détaille notre approche ainsi qu'un algorithme complet pour l'amélioration automatique du jeu de test des transformations de modèles. Des perspectives d'amélioration ainsi que différentes pistes pour de futures recherches sont tracées au chapitre 4. Enfin, nous concluons au chapitre 5.

Chapitre 2

État de l'art

2.1 Programmer, abstraire

Depuis que la notion de programme informatique a été introduite au milieu du XX^{e} siècle, les programmeurs ont toujours souhaité disposer de mécanismes d'abstraction leur permettant d'outrepasser les détails techniques d'implémentation liés au contexte d'exécution afin de se concentrer sur la logique du programme à réaliser.

Le seul langage directement interprétable par un processeur classique est le langage binaire consistant en une suite de *bits*. Il est extrêmement compliqué pour le cerveau humain d'exprimer un algorithme, même simple, avec ce langage. Aussi, le langage assembleur, qui utilise des mnémoniques pour représenter le langage machine sous une forme compréhensible par l'humain peut être vue comme la première forme d'abstraction proposée aux programmeurs.

Fortran fût le premier langage de programmation à introduire des structures de contrôle de plus haut niveau [1]. Le premier compilateur Fortran, développé à la fin des années 1950, constitue l'une des grandes étapes pour l'abstraction de la complexité des programmes en permettant au programmeur de décrire ce que le programme doit faire plutôt que la manière dont il doit le faire.

Depuis cette époque, les différentes générations de langages (et les différents langages de programmation) n'ont cessé de tendre vers une abstraction de plus en plus forte des problématiques techniques comme la gestion de la mémoire, la résolution d'adresse ou la gestion des types.

L'avènement de la programmation orientée objet (et des langages dits orientés objet [2]) constitue une étape importante dans ce processus d'abstraction. La notion même d'*objet*, en tant que structure de données abstraite, apporte un formalisme permettant d'exprimer directement les objets du monde réel tels que naturellement appréhendés par le cerveau humain. La programmation n'y est plus seulement vue que comme le déroulement d'un algorithme sur des données

1. Comme les sauts inconditionnels, les boucles, etc.
2. Simula et Smalltalk dans les années 1960-1970 et plus récemment Java, C++ ou encore Python

simples mais comme la mise en interaction de structures complexes possédant leurs caractéristiques propres. La programmation orientée objet constitue aujourd'hui le standard *de facto* dans le monde industriel pour la production de systèmes d'information complexes.

Programmer, c'est abstraire. L'informatique ne peut pas et n'a pas vocation à reproduire le monde réel, mais à le représenter, à l'abstraire. Des mécanismes d'abstraction doivent donc aussi être utilisés pour la représentation des objets du monde réel. La notion de modèle a été utilisée de tout temps, particulièrement dans le monde de l'ingénierie, pour représenter le monde réel. Nous introduisons dans la section suivante les problématiques liées à la modélisation et l'utilisation particulière des modèles dans le cadre de l'ingénierie dirigée par les modèles.

2.2 L'ingénierie dirigée par les modèles

Depuis les années 1980, le monde de l'ingénierie informatique est dominé par l'approche objet et le principe du « tout est objet ». Plus récemment, une nouvelle tendance émerge, s'orientant vers l'ingénierie dirigée par les modèles et le principe du « tout est modèle » [11, 75].

Les modèles ont depuis toujours été utilisés en informatique comme support du travail des ingénieurs et programmeurs. La représentation de systèmes à travers des modèles, et notamment leur représentation graphique, permet de cerner plus rapidement les différents concepts mis en jeu et leurs interactions.

Par ailleurs, l'expérience acquise sur les systèmes informatiques montre qu'ils sont soumis à de fortes contraintes de variabilité et d'évolution : les besoins des utilisateurs peuvent évoluer, de nouvelles contraintes non-fonctionnelles peuvent être introduites, une extension de leur domaine d'application peut être requise, etc.. Ces contraintes particulières amènent les ingénieurs à repenser les spécifications et modèles au cours du temps, et les programmeurs à faire évoluer les codes sources en conséquence. De plus, les nouveaux logiciels gagnent de plus en plus en complexité, notamment avec les logiciels embarqués, nomades ou fortement répartis. Une issue réaliste pour la prise en compte de ces problématiques est de permettre la production automatique de logiciels.

Or, historiquement, on constate que les modèles ont été essentiellement utilisés de manière consultative et souvent relégués au rang de simple documentation. L'ingénierie dirigée par les modèles [3] (IDM) vise à placer les modèles *au cœur du processus de développement* pour en faire les éléments centraux par lesquels les applications sont générées. L'IDM est une forme d'ingénierie générative dans laquelle tout ou partie d'une application est générée automatiquement à partir de modèles. En plaçant les modèles plutôt que la programmation classique au cœur du processus de développement, le niveau d'abstraction auquel celui-ci a lieu se voit relevé, et la prise en compte du changement est donc facilitée. Les modifications inévitables auxquelles doivent se plier les systèmes au cours du temps n'impactent qu'au niveau modèle, les codes sources pouvant être regénérés automatiquement à partir de ceux-ci.

3. Ou *Model-Driven Engineering*, MDE en anglais

Une des idées centrales de l'IDM est d'adapter les langages de modélisation au domaine spécifique du système étudié afin de permettre l'expression la plus précise et la plus simple possible des problématiques qui lui sont propres. Le concept de *métamodélisation* permet justement la définition de tels langages. De plus, afin de rendre les modèles ainsi produits dynamiques et directement opérationnels, l'IDM s'intéresse à la notion de *transformation de modèles*, support indispensable de la génération de code, de documentation, de tests etc. directement depuis ces modèles.

L'ingénierie dirigée par les modèles est un domaine en plein développement et, bien que les principes qui la fondent ne soient pas encore définis de manière suffisament satisfaisante [30], l'émergence de conférences internationales dédiées à l'IDM et le développement de nombreux outils montrent l'intérêt des communautés scientifiques et industrielles pour cette approche.

La section 2.2.1 introduit les concepts de modèles et métamodèles, les langages de modélisation spécifiques au domaine sont présentés en section 2.2.2 et les transformations de modèles en section 2.2.3. Enfin, nous présentons en section 2.2.4 l'une des technologies les plus utilisées mettant en œuvre les concepts de l'IDM : MDA.

2.2.1 Modéliser, métamodéliser

2.2.1.1 Modéliser pour décrire les systèmes

Un modèle est une abstraction d'un système dans un objectif précis. On dit qu'un modèle *représente* un système. Un modèle est une abstraction en ce sens qu'il omet une partie de la réalité du système étudié pour n'en conserver que les informations intéressantes pour l'objectif choisi. Un exemple souvent utilisé est celui de la carte routière : une carte routière *représente* un objet réel (un territoire) en ne conservant que les informations utiles au déplacement des personnes (chemins, routes, distances, dénivelés, etc.) et en rejetant les autres (climat, géopolitique, etc.). A l'inverse, une carte géopolitique représente aussi un territoire mais en se concentrant sur d'autres informations (données politiques, ethniques, militaires, etc.). Ces deux modèles correspondent à deux vues différentes d'un même système.

Les modèles ont toujours été utilisés en informatique pour permettre l'étude de systèmes. Leur représentation graphique permet une compréhension plus rapide des concepts qu'ils symbolisent là où une description textuelle, bien que plus précise, est moins abordable. Actuellement, le langage de modélisation le plus utilisé dans le monde de l'ingénierie informatique est UML (cf. section 2.2.1.6). Il est à noter que, en accord avec cette définition, un code source exprimé dans un langage particulier est lui-aussi un modèle puisqu'il est une abstraction du code machine appartenant au monde réel.

2.2.1.2 Métamodéliser pour décrire les modèles

Un certain formalisme, partagé par les personnes travaillant sur un certain type de modèles, est nécessaire à leur interprétation : la notion de *métamodèle*, au cœur de l'ingénierie dirigée par les

modèles (sans lui être exclusive), permet de définir des *langages de modélisation* avec lesquels sont écrits les modèles.

Un métamodèle définit les éléments et leurs relations au sein d'un certain type de modèles. En ce sens, il définit la syntaxe de ces modèles. La relation unissant un modèle à son métamodèle est une relation forte, on parle de *conformité* : un modèle est *conforme* à son métamodèle si et seulement s'il respecte les règles qui y sont établies. De plus, un métamodèle est lui-même un modèle et peut bénéficier des mêmes avantages.

En reprenant l'exemple des cartes routières, nous pouvons dire que quelque soit le système (le territoire) étudié, la carte associée respecte toujours les mêmes conventions qui sont définies par sa légende, que nous nommons métamodèle dans notre terminologie. A partir d'une certaine légende, il est possible de construire une infinité de cartes routières représentant des territoires différents. De la même manière, pour un métamodèle donné, il est possible d'écrire une infinité de modèles s'y conformant.

Puisque tous les modèles conformes à un métamodèle sont écrits de la même manière (dans un même *langage*, cf. section 2.2.1.5), il est possible de raisonner de manière automatique sur chacun d'entre eux en exploitant l'information portée par le métamodèle.

2.2.1.3 Méta-métamodéliser pour décrire les métamodèles

De la même manière que la métamodélisation est nécessaire pour raisonner de manière automatique sur un ensemble de modèles, il est nécessaire de disposer d'un métamodèle de métamodèle pour manipuler ces derniers : un tel métamodèle est appelé *méta-métamodèle*. Un méta-métamodèle décrit les éléments et leurs relations au sein d'un métamodèle. Il est en quelque sorte un *langage de métamodélisation*. Ce raisonnement pourrait s'appliquer à l'infini : nous pourrions vouloir écrire des méta-méta-métamodèles pour la manipulation de méta-métamodèles. Afin de ne pas tomber dans une montée en abstraction infinie, les méta-métamodèles sont conçus de manière à se définir eux-mêmes : un méta-métamodèle permet de définir des métamodèles, et peut donc se définir lui-même en tant que métamodèle de métamodèle. Un méta-métamodèle étant lui-même un métamodèle, il est par extension lui aussi un modèle.

2.2.1.4 L'architecture de modélisation en quatre couches

Le standard MOF [65, 68], ainsi que plusieurs autres standards OMG [4] portant sur la modélisation, utilisent le terme de « métacouche » pour représenter un certain niveau d'abstraction. Les concepts clés utilisés dans les standards OMG pour la modélisation sont *Classifier* et *Instance*, ainsi que la possibilité de naviguer d'une *Instance* vers son métaobjet (le *Classifier*). Une *Instance* est toujours située sur la couche immédiatement inférieure à son *Classifier*. De cette manière, il est possible de travailler sur un nombre infini de métacouches de manière récursive (au minimum 2).

4. L'*Object Management Group* est une association américaine à but non lucratif créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes. Ce consortium est l'auteur de nombreux standards mondialement répandus comme UML, CORBA, MOF, QVT ou l'IDL

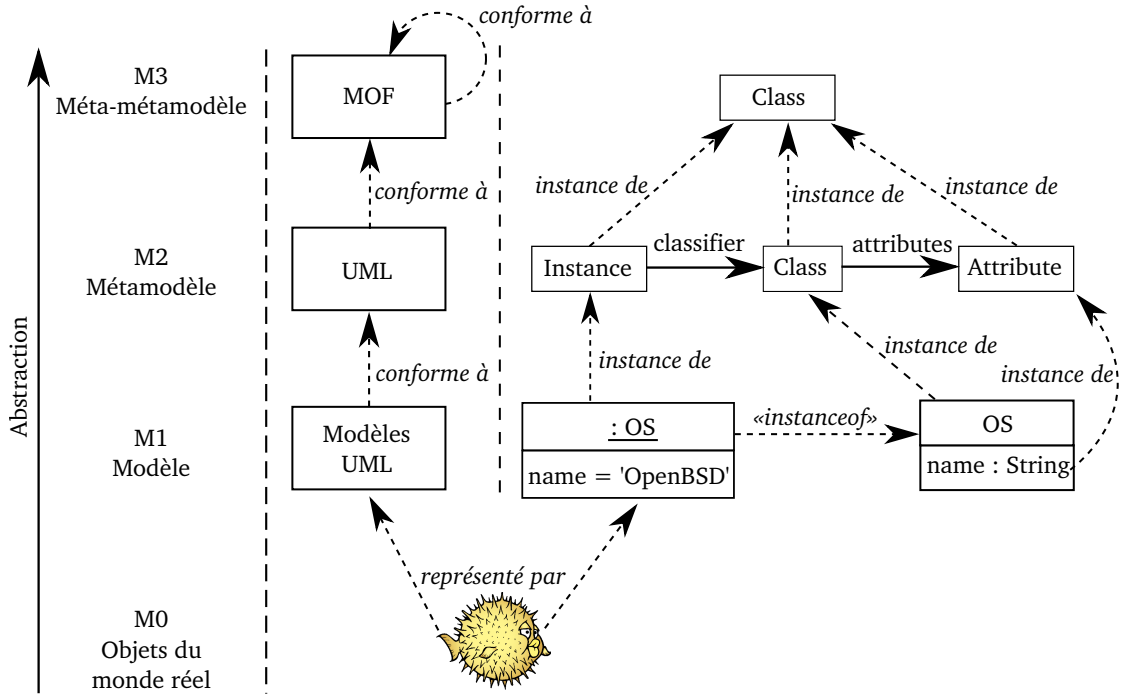


FIGURE 1: L'architecture en quatre couches du MOF

En pratique, l'approche souvent retenue dans l'IDM (par exemple, dans MDA), est une architecture de modélisation en quatre couches. Au niveau M0 se trouvent les objets étudiés du monde réel, au niveau M1 se trouvent les modèles de ces objets tels que décrits à la section 2.2.1.1, au niveau M2 les métamodèles décrivant ces modèles tels que décrits à la section 2.2.1.2 et au niveau M3 le méta-métamodèle utilisé pour l'écriture de métamodèles tel que décrit à la section 2.2.1.3. Dans la plupart des cas, le méta-métamodèle utilisé au niveau M3 est le MOF lui-même.

La relation de conformité « *est conforme à* » unit les couches $N-1$ aux couches N (sauf le niveau M3 qui est conforme à lui-même). Chacune de ces couches définit des concepts, que nous nommons classe au niveau M1, métaclasse au niveau M2 et méta-métaclasse au niveau M3. La relation qui unit les concepts d'une couche $N-1$ à ceux d'une couche N est une relation d'instanciation. Ainsi, un objet du monde réel est représenté par l'instance d'une classe de niveau modèle (M1). Cette classe est instance d'une métaclasse au niveau M2, elle-même instance d'une méta-métaclasse au niveau M3. La figure 1 présente un exemple de cette métamodélisation en quatre couches, où UML et MOF sont choisis aux niveaux M2 et M3. La relation d'instanciation qui relie l'instance d'OS à sa classe au niveau M1 est une relation interne à UML et ne doit pas être confondue avec les relations d'instanciation entre métacouches.

2.2.1.5 Parallèle avec la théorie des langages

Afin de faciliter la compréhension de la notion de modèle, métamodèle et méta-métamodèle, le parallèle avec la théorie des langages est souvent employé [30]. Un métamodèle peut être vu comme un *langage* permettant l'écriture de différent mots (ici, modèles) conformes à celui-ci. Au niveau supérieur, un méta-métamodèle peut être vu comme une *grammaire* permettant l'écriture des différents langages (ici, métamodèles) dans lequels sont décrits les modèles. Les

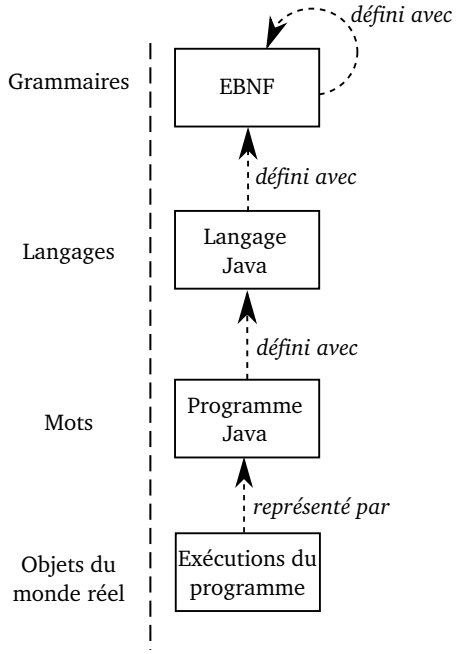


FIGURE 2: Parallèle entre l'architecture en quatre couches et la définition de langages

méta-métamodèles se décrivent eux-mêmes, de la même manière qu'une grammaire EBNF [5] peut s'écrire elle-même en grammaire EBNF. Cependant, les métalangages comme EBNF décrivent à la fois la syntaxe et la sémantique d'un langage, contrairement aux métamodèles utilisés dans l'IDM qui ne présentent que la syntaxe d'un langage de modélisation.

La figure 2 prend l'exemple de Java pour illustrer le parallèle avec la théorie des langages. Sont présents : EBNF au niveau M3 pour l'écriture de grammaires, la grammaire du langage Java au niveau M2, un programme Java particulier au niveau M1, et une exécution particulière de ce programme au niveau M0 [10].

2.2.1.6 Le standard UML

Le langage de modélisation le plus utilisé à l'heure actuelle pour la représentation de systèmes dans le domaine de l'ingénierie informatique est le *Unified Modeling Language* [67] (UML). Il a été créé en 1995 afin d'unifier les différentes méthodes d'analyse et de conception orientées objet en vigueur à l'époque, notamment la méthode Booch [14], l'OMT [74] de Rumbaugh et l'OOSE [42] de Jacobson.

UML a été standardisé par l'OMG en 1997 et est devenu une norme ISO en 2000. Il permet la description graphique de systèmes pour la production de logiciels orientés objet. UML propose plusieurs types de modèles, décrits par des diagrammes, et permettant la représentation d'un système sous différentes vues, statiques ou comportementales : diagrammes de classes, de déploiement, de séquence, de cas d'utilisation etc.. La figure 3 montre la hiérarchie complète des diagrammes proposés par UML.

5. EBNF [88] est une extension du métalangage *Backus-Naur Form* utilisée notamment pour décrire la syntaxe des langages de programmation

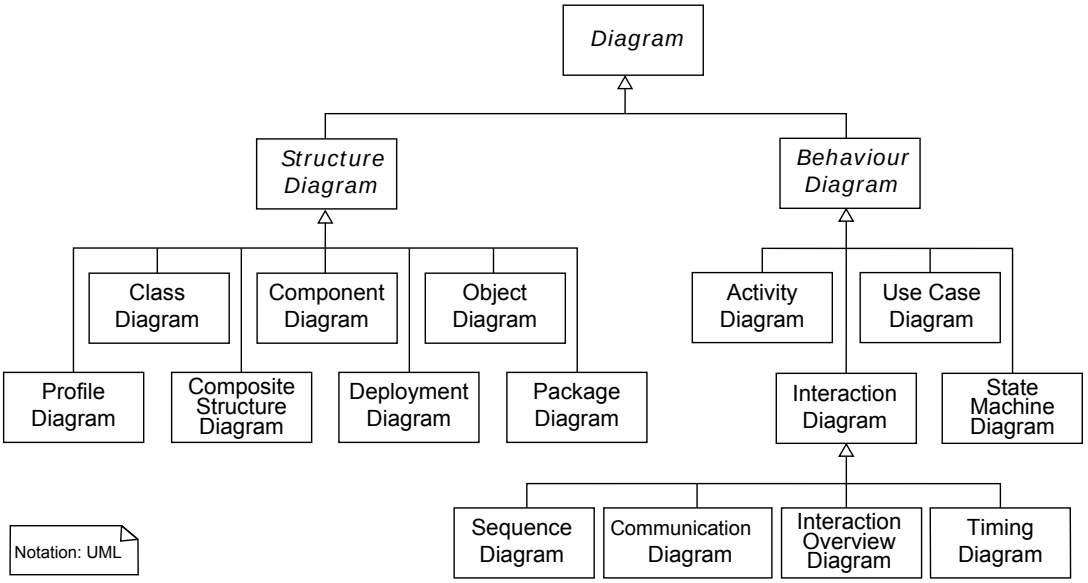


FIGURE 3: Hiérarchie des diagrammes UML

Puisqu'UML est un langage de modélisation, il est situé au niveau M2 de l'architecture MOF : c'est un métamodèle utilisé pour écrire des modèles de niveau M1 représentants des objets du monde réel au niveau M0. UML est lui-même conforme au méta-métamodèle MOF au niveau M3.

UML 2.0 propose plusieurs mécanismes d'extension dont l'écriture de profils utilisant des stéréotypes [36]. Un stéréotype permet de créer de nouveaux types d'éléments, dérivés d'éléments existants dans le métamodèle UML, mais possédant des propriétés spécifiques. Un profil UML contient un ensemble de stéréotypes dédiés à un domaine particulier : par exemple, MARTE [69] (*Modeling and Analysis of Real Time and Embedded systems*) est un profil UML pour la modélisation des systèmes temps réels embarqués. Cependant, la puissance des profils reste limitée dans la mesure où un profil doit toujours rester conforme au métamodèle UML. Les profils UML comme MARTE sont donc souvent vus comme des outils difficiles à maîtriser et contraignants. Afin de résoudre le manque de flexibilité d'UML, l'IDM propose la définition de langages de modélisation spécifiques au domaine, que nous introduisons dans la section suivante.

2.2.2 Les langages de modélisation spécifiques au domaine

La modélisation des systèmes est au cœur de l'ingénierie dirigée par les modèles. Celle-ci doit être à la fois la plus précise et la plus simple possible afin de faciliter le travail des ingénieurs. Or, modéliser un système quelconque à l'aide d'UML nécessite de traduire les concepts qu'il manipule dans le formalisme UML : ce travail de traduction d'un domaine propre vers le formalisme UML nécessite des experts en modélisation UML et peut s'avérer complexe.

Pour remédier à ce problème, la démarche IDM propose la définition de langages de modélisation spécifiques au domaine (*Domain-Specific Modeling Language* ou DSL). Un tel langage incorpore directement, pour un domaine d'application particulier, les concepts propres au domaine et utiles

à la résolution des problèmes de ce dernier. Une démarche similaire existe dans le monde des langages de programmation : conjointement aux langages de programmation dits « d'usage général » comme Java ou Python, on trouve des langages de programmation « spécialisés », spécifiques à un domaine d'application, tels que Mathematica pour les mathématiques ou VHDL pour la description de matériel informatique. De la même manière, dans le monde de la modélisation, UML peut être classé comme un langage de modélisation « d'usage général », alors que les DSL sont des langages de modélisation spécifiques à un domaine. Bien sûr, la frontière entre ces deux types d'approche n'est pas strictement marquée : chaque langage est plus ou moins spécifique à un domaine.

En proposant d'incorporer les concepts du domaine directement dans le langage de modélisation, l'approche IDM permet aux experts du domaine d'exprimer intuitivement et par eux-mêmes leur savoir dans la modélisation. Les DSL définissent à la fois une syntaxe abstraite (les concepts du domaine, leurs relations) et une syntaxe concrète (représentation graphique ou textuelle) utile pour la compréhension.

Dans l'IDM, la définition des DSL est faite par *métamodélisation* (cf. section 2.2.1.2) : un métamodèle définit les concepts propres à un domaine et leurs relations. Une fois ce métamodèle défini, des modèles s'y conformant peuvent être écrits par les experts du domaine dans leur propre langage (celui du métamodèle).

Bien sûr, le plus souvent, les DSL utilisent un formalisme très éloigné du formalisme des langages de programmation. Pourtant, les modèles doivent être en dernière instance traduits en code source pour produire une application réelle. Les transformations de modèles réalisent les étapes indispensables de traduction, raffinement et génération de modèles abstraits et spécifiques à un domaine vers du code source exécutable.

2.2.3 Les transformations de modèles

2.2.3.1 Principe

Puisque l'IDM vise à faire des modèles les entités principales du processus de développement de logiciel, ces modèles doivent pouvoir être traités de manière systématique. Les modèles tels que nous les avons décrits dans les sections précédentes doivent être considérés à la fois de manière horizontale (pour représenter différents aspects d'un système) et de manière verticale (pour faire varier le niveau d'abstraction) [79]. Le traitement automatique des modèles doit pouvoir s'effectuer sur ces deux plans. Quelques exemples de manipulations de modèles qui peuvent bénéficier d'une automatisation sont :

Raffinement Le passage de modèles abstraits (spécification) à des modèles extrêmement précis (code source) est effectué par étapes de raffinement successives : à chaque étape, des détails (par exemple liés à la plateforme d'exécution) sont insérés dans les modèles afin de se rapprocher du langage informatique et de permettre leur exécution. Par exemple, une étape de raffinement peut, pour chaque attribut public d'un diagramme de classe, le rendre privé et générer automatiquement son accesseur et mutateur.

Rétroingénierie A l'inverse du raffinement, il peut être utile de générer automatiquement une version plus abstraite d'un modèle pour pouvoir travailler à un niveau d'abstraction plus élevé. Typiquement, la génération d'un diagramme de classe à partir d'un code source est un exemple de rétroingénierie.

Génération de nouvelles vues A partir d'un modèle donné, différents modèles exposant différents points de vues peuvent être générés. Par exemple, un modèle représentant l'ensemble des composants d'une application peut être filtré de sorte à générer un modèle de communication de l'application où seuls les composants utilisant des appels distants sont retenus.

Application de patrons de conception Les patrons de conception permettent de répondre à des problèmes classiques de conception souvent rencontrés dans le développement de logiciels. L'application d'un patron de conception à une classe d'un modèle peut impacter sa hiérarchie et ses relations et devrait être automatisée.

Réusinage Des changement successifs et mal coordonnés sur un modèle peuvent le rendre difficilement compréhensible. Une opération de réusinage peut, par exemple, découper des composants complexes en plus petits composants afin d'améliorer sa lisibilité et sa maintenabilité tout en conservant sa sémantique générale.

La relation de conformité qui relie un modèle à son métamodèle fournit une base structurelle pour le traitement automatique des modèles. Les transformations de modèles [55, 79], quant à elles, décrivent les relations entre deux modèles ou plus. Bien que les termes utilisés soient parfois confondus dans la littérature, nous utilisons la définition suivante, inspirée de [33, 55, 79] :

Definition. Une *transformation de modèle* (TM) exprime la relation entre deux modèles ou plus. Elle joue le rôle de spécification pour un *programme de transformation de modèle* (PTM) qui réalise effectivement la transformation en s'assurant que les éléments des modèles manipulés respecte la spécification de la TM. Les PTM sont exprimés dans un *langage de transformation de modèles* (LTM).

Les transformations de modèles manipulent des données complexes : des modèles conformes à des métamodèles. Elles peuvent prendre en entrée et produire en sortie un nombre quelconque de modèles, se conformant à un nombre quelconque de métamodèles, être unidirectionnelles ou bidirectionnelles, etc. [55]. De plus, une transformation peut imposer en entrée un certain nombre de préconditions sur les modèles : afin de pouvoir être transformé, un modèle doit alors à la fois être conforme au métamodèle d'entrée de la transformation et aux préconditions de celles-ci. Si le modèle remplit ce contrat, un certain nombre de postconditions sont assurées par la transformation sur le modèle de sortie. Il en résulte qu'une transformation peut n'accepter qu'un sous-ensemble de modèles parmi l'ensemble des modèles conformes au métamodèle d'entrée, et peut ne produire qu'un sous-ensemble de modèles parmi l'ensemble des modèles conformes au métamodèle de sortie. Ces concepts sont décrits à la figure 4 dans le cas d'une transformation unidirectionnelle prenant un modèle en entrée et produisant un modèle en sortie.

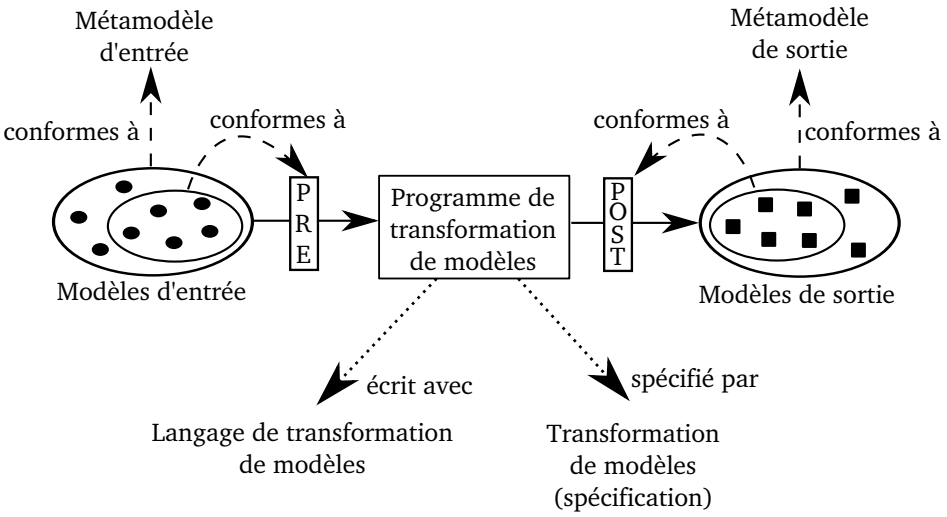


FIGURE 4: Transformation de modèles

2.2.3.2 Mise en œuvre

Comme nous venons de le voir, la mise en œuvre des transformations de modèles nécessite un langage de transformation de modèles. De tels langages, comme MTL [86], Kermeta [43] ou ATL [47] existent déjà, et possèdent chacun leurs caractéristiques propres. Quelques caractéristiques importantes des langages de transformations sont :

Paradigme de programmation Les langages soumis en réponse à la RFP OMG/QVT [66] se scindent en deux principaux groupes : langages de style impératif ou déclaratif. L'approche déclarative définit un ensemble de règles de transformation en spécifiant des préconditions sur le modèle source et les postconditions associées sur le modèle de sortie. Un processus de *mapping* permet ainsi de mettre en relation les éléments en entrée et en sortie. L'approche impérative utilise des structures de contrôle plus proches de celles des langages de programmation classiques, laissant ainsi aux développeurs une liberté totale dans l'écriture des transformations. Les langages purement déclaratifs permettent difficilement l'écriture précise de transformations complexes, ainsi une approche prometteuse se situe à mi-chemin des deux paradigmes [12].

Métamodélisation du langage L'approche suggérée dans [12] recommande d'appliquer aux langages de transformation de modèles les principes mêmes de l'IDM. Ainsi, la création d'une transformation de modèles passe par la définition d'une transformation indépendante de toute plateforme (PIT), raffinée ensuite en une transformation spécifique à une plateforme (PST). La transformation elle-même devient alors un modèle, conforme à son métamodèle. Les avantages de la modélisation y sont donc directement applicables : abstraction, généricité, réutilisatilité, etc.. De telles transformations peuvent à leur tour être prises en entrée d'autres transformations appelées transformations d'ordre supérieur.

2.2.4 L'approche MDA

La *Model-Driven Architecture* [54, 70, 84] (MDA) est une méthode pour le développement de logiciels mettant largement en œuvre les concepts de l'IDM et originellement proposée par l'OMG en 2001.

Dans MDA, les spécifications sont représentées par des modèles et sont au cœur du processus de développement. Les systèmes y sont représentés à différents niveaux d'abstraction et sous différents points de vue. L'idée principale est que les systèmes développés sont destinés à être déployés sur des plateformes hétérogènes ; les plateformes étant sujettes à évolution au cours du temps et à une vitesse différente de l'évolution des spécifications, il est intéressant de séparer ces deux vues d'un système afin de les faire évoluer indépendamment tout en maintenant leur cohérence.

Le comportement attendu d'un système y est décrit par un *Computation-Independent Model* (CIM). Le CIM exprime les problématiques du domaine dans un langage directement compréhensible par l'utilisateur. Les fonctionnalités du système sont décrites en utilisant un modèle indépendant de la plateforme (*Plateform-Independent Model*, PIM), écrit dans un langage de modélisation spécifique à un domaine (un métamodèle particulier, cf 2.2.2). Le PIM est ensuite traduit (raffiné) en un autre modèle spécifique à la plateforme (*Plateform-Specific Model*, PSM). Un PSM est un modèle écrit soit dans un langage de modélisation spécifique à la plateforme, soit dans un langage de programmation d'usage général comme Java ou C#. Bien sûr, le passage d'un modèle à un autre nécessite l'écriture de transformations de modèles telles que décrites en section 2.2.3. L'objectif final est d'arriver à un modèle suffisament détaillé pour la génération automatique du code dédié à une plateforme particulière. Les CIM/PIM évoluent avec les spécifications du système, les PSM avec les technologies liées aux plateformes. Les transformations de modèles permettent de maintenir la cohérence entre les différents niveaux en fonction des évolutions des modèles. La figure 5 résume ces concepts. Il est à noter que les différents modèles CIM/PIM/PSM ne doivent pas être confondus avec l'architecture en métacouches du MOF (cf. section 2.2.1.4) : ils prennent tous place au niveau modèle (M1) de cette architecture et peuvent être conformes à n'importe quels métamodèles.

Une série de standards de l'OMG est associée à la technologie MDA, parmi lesquels MOF pour l'écriture des métamodèles et modèles, UML (et les profils UML) comme métamodèles souvent utilisés, XMI pour l'échange standardisé de modèles, CWM pour les modèles de données, etc..

MDA étant un standard de l'OMG, il fait l'objet d'une attention toute particulière dans le monde industriel et scientifique. Il faut cependant noter qu'MDA n'est qu'une implémentation spécifique des concepts de l'IDM, ce dernier ne devant pas se résumer au premier.

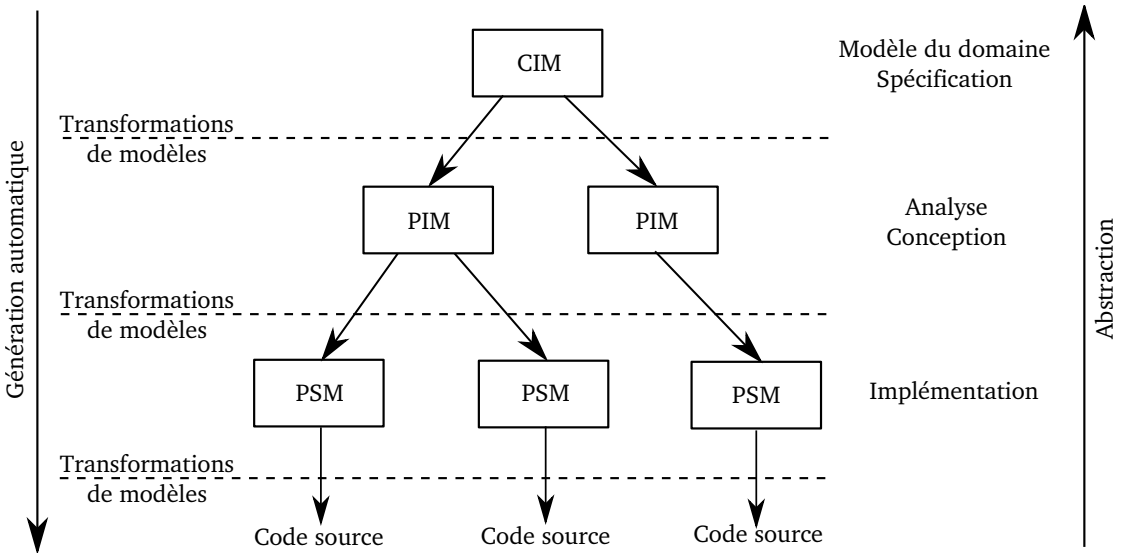


FIGURE 5: *Model-Driven Architecture*

2.3 L'analyse de mutation

2.3.1 Rappel sur le test logiciel

Le test logiciel [9] vise à vérifier le comportement d'un programme en lui soumettant des données en entrée, appelées *données de test*, et en observant les sorties qu'il produit. Ces sorties sont évaluées à l'aide d'un *oracle* qui doit déterminer si elles correspondent ou non au comportement attendu du programme. Chaque couple $(donnée\ de\ test, oracle)$ forme un *cas de test*, dont le rôle est de vérifier une partie d'un programme. L'ensemble des données de test écrites pour un programme forme son *jeu de test*.

Bien sûr, il n'est pas possible de vérifier le comportement d'un programme pour chacune des valeurs qu'il peut prendre en entrée, celles-ci pouvant être en nombre infini. Ainsi, les différentes techniques de test utilisent des *critères de test* permettant de déterminer si le programme sous test a été *suffisamment testé*. Ces critères se basent par exemple sur la couverture du code ou de la spécification.

2.3.2 Méthodologie

L'analyse de mutation est une technique permettant d'évaluer la qualité d'un jeu de test. Son origine remonte aux années 1970 et est attribuée aux travaux de DeMillo *et al.* [24] et Hamlet [40]. Plus précisément, l'analyse de mutation permet de qualifier un jeu de test en fonction de sa capacité à détecter des erreurs réelles dans un programme. Bien que cette technique soit principalement utilisée au niveau du test unitaire, elle peut s'appliquer au test d'intégration [21] ou de spécification [17]. Une étude récente [44] montre que l'analyse de mutation fait l'objet d'un intérêt grandissant de la part de la communauté scientifique depuis ses débuts, et que le nombre d'outils développés ainsi que l'intérêt de la communauté industrielle ne cessent d'augmenter.

2.3.2.1 Principe

L'analyse de mutation est une technique de test boîte blanche dans laquelle des versions volontairement erronées d'un programme, appelées *mutants*, sont créées afin de déterminer si le jeu de test du programme est capable de les détecter. Chaque mutant est créé à partir du programme sous test, en injectant *une et une seule erreur simple*[6]. Ces erreurs correspondent à un changement syntaxique simple et sont semblables à celles qu'un programmeur peut commettre.

Une fois les mutants créés, le jeu du test du programme est exécuté sur le programme original et sur ses mutants. Si le résultat de l'exécution d'un mutant est différent du résultat de l'exécution du programme original pour un test donné, alors l'erreur introduite a été détectée et le mutant est considéré comme *tué*. Si, au contraire, le résultat de l'exécution d'un des mutants est le même que le résultat du programme original pour l'ensemble du jeu de test, alors le mutant est considéré comme *en vie* : aucune des données de test n'a été en mesure de détecter l'erreur volontairement introduite dans le mutant.

L'analyse de mutation produit une métrique de test appelée *score de mutation* indiquant la capacité du jeu de test à détecter les erreurs volontairement introduites dans les mutants. Elle correspond au pourcentage de mutants tués (dont les fautes ont été détectées) par rapport au nombre total de mutants non-équivalents créés. Les mutants équivalents sont des mutants qui, bien que syntaxiquement différents du programme original, sont sémantiquement équivalents et ont donc exactement le même comportement quelque soit les données en entrée : ils ne peuvent pas être tués. Puisque le problème de la détection de l'équivalence de deux programmes est indécidable [16], il n'est pas possible de détecter automatiquement les mutants équivalents parmi l'ensemble des mutants [61]. Ce problème est abordé plus en détail en section 2.3.3.2.

Soit T un jeu de test, d le nombre de mutants tués, N le nombre total de mutants et eq le nombre de mutants équivalents, la formule pour le calcul du score de mutation du jeu de test T est la suivante :

$$ScoreMutation(T) = 100 \times \frac{d}{N - eq}$$

L'analyse de mutation vise à identifier les données de test pouvant détecter des erreurs réelles. Cependant, le nombre de ces erreurs, et donc de mutants, est extrêmement élevé : il n'est pas possible de tous les analyser. Afin de remédier à ce problème, l'analyse de mutation se concentre sur un sous-ensemble de ces erreurs, susceptible de simuler l'ensemble des erreurs possibles. Cette théorie est basée sur deux hypothèses fondamentales :

Competent Programmer Hypothesis Cette hypothèse stipule que les programmeurs sont compétents, et que les programmes qu'ils produisent sont *proches* de la version correcte [24]. Ainsi, bien que des erreurs puissent apparaître dans les programmes, nous considérons que ces erreurs sont simples et peuvent être corrigées par de simples changements syntaxiques. L'analyse de mutation se focalise donc sur l'insertion d'erreurs consistant en de simples changements syntaxiques.

6. L'analyse de mutation d'ordre supérieur [45], où plusieurs fautes sont insérées dans chaque mutant, n'est pas considérée dans ce document

Coupling Effect Hypothesis Cette hypothèse stipule que, si une donnée de test est capable de détecter toutes les erreurs simples insérées dans un programme, alors elle est aussi capable de détecter les erreurs plus complexes. Cette hypothèse est reformulée précisément par Offutt pour l'analyse de mutation dans [58] : « *Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will also detect a large percentage of the complex mutants* ». La *Coupling Effect Hypothesis* a depuis fait l'objet de nombreuses études théoriques et empiriques [44] qui tendent à prouver sa véracité.

De nombreux travaux ont été entrepris afin de comparer l'efficacité du critère de test introduit par l'analyse de mutation avec les autres critères de tests connus [44]. Il a notamment été montré que le critère de mutation subsume [7] le critère *all-uses* [34, 62], le critère *edge-pair* [19] et le critère de couverture d'instructions [1]. La capacité de l'analyse de mutation à simuler des erreurs complexes dans des programmes à grandes échelles a aussi été évaluée, notamment sur un programme de nucléaire civil [20] et sur TeX [80]. En conclusion, le critère de mutation ayant été évalué comme plus puissant que d'autres critères et capable de détecter des erreurs complexes réelles, son usage est recommandé par de nombreuses études [51, 82, 83].

2.3.2.2 Les opérateurs de mutation

Un opérateur de mutation est une règle de transformation qui, à partir d'un programme de base P, génère un mutant M : les opérateurs de mutation sont responsables de l'insertion des erreurs, et sont donc un point crucial de l'analyse de mutation. Leur objectif est de simuler des erreurs réelles qui pourrait être commises par un programmeur [22]. Puisque les erreurs introduites sont d'ordre syntaxique, elles dépendent de la syntaxe du langage utilisé par le programme. Ainsi, les opérateurs de mutation sont spécifiques à un langage particulier.

Historiquement, les premiers opérateurs de mutation ont été créés pour le langage FORTRAN et sont connus sous le nom des « 22 opérateurs Mothra » [49]. Le tableau 1 les détaille : on y trouve des opérateurs modifiant des constantes ou des connecteurs logiques, supprimant des instructions ou insérant des valeurs arbitraires, etc..

Le tableau 2 montre un exemple de l'application de l'opérateur de mutation AOR (*Arithmetic Operator Replacement*) sur une instruction Java simple [8]. Il apparaît que l'application d'un seul opérateur de mutation sur une seule instruction produit déjà 4 mutants : intuitivement, nous voyons déjà les problèmes de combinatoire qu'induit l'analyse de mutation (cf. section 2.3.3).

D'une manière générale, les opérateurs de mutation pour les langages de programmation impératifs s'inspirent très largement des opérateurs Mothra. Cependant, de nouveaux opérateurs ont dû être proposés pour pouvoir appliquer l'analyse de mutation aux langages orientés objet [52], aux diagrammes états-transitions [28], aux schémas XML [50] et à bien d'autres [44].

7. Un critère C_1 subsume un critère C_2 si un jeu de test adéquat pour C_1 l'est aussi pour C_2
8. L'opérateur AOR est directement utilisable sur le langage Java, bien qu'il ait été défini pour le langage FORTRAN

Opérateur	Description
AAR	Array reference for array reference replacement
ABS	Absolute value insertion
ACR	Array reference for constant replacement
AOR	Arithmetic operator replacement
ASR	Array reference for scalar variable replacement
CAR	Constant for array reference replacement
CNR	Comparable array name replacement
CRP	Constant replacement
CSR	Constant for scalar variable replacement
DER	DO statement alterations
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	Logical connector replacement
ROR	Relational operator replacement
RSR	RETURN statement replacement
SAN	Statement analysis
SAR	Scalar variable for array reference replacement
SCR	Scalar for constant replacement
SDL	Statement deletion
SRC	Source constant replacement
SVR	Scalar variable replacement
UOI	Unary operator insertion

Tableau 1: Les 22 opérateurs Mothra pour FORTRAN

Programme P	Mutants P'
`c = a + b;`	`c = a - b;`
	`c = a * b;`
	`c = a / b;`
	`c = a % b;`

Tableau 2: Application de l'opérateur AOR sur une instruction Java

2.3.2.3 Le processus d'analyse

Le processus général de l'analyse de mutation est décrit en figure 6.

L'étape préliminaire (a) vise à produire les données nécessaires à la suite du processus : à partir d'un programme sous test, un ensemble de mutants est produit de manière automatique par l'application d'opérateurs de mutation choisis en fonction du langage utilisé.

Le processus doit de plus disposer d'un jeu de test à évaluer : si un tel jeu de test existe, il est utilisé, sinon il doit être créé à cette étape. La présence d'oracles permettant de statuer sur la réussite des tests n'est pas obligatoire, l'analyse de mutation ne visant qu'à comparer les sorties respectives du programme sous test et de ses mutants. En fait, deux approches différentes existent (nous retenons la première dans ce document) :

– Seules des données de test sont présentes, auquel cas le processus d'analyse de mutation va simplement comparer le résultat (*i.e.* les sorties produites) de l'exécution du programme sous test avec celui de ses mutants : si les sorties sont différentes le mutant est tué, sinon il est toujours vivant.

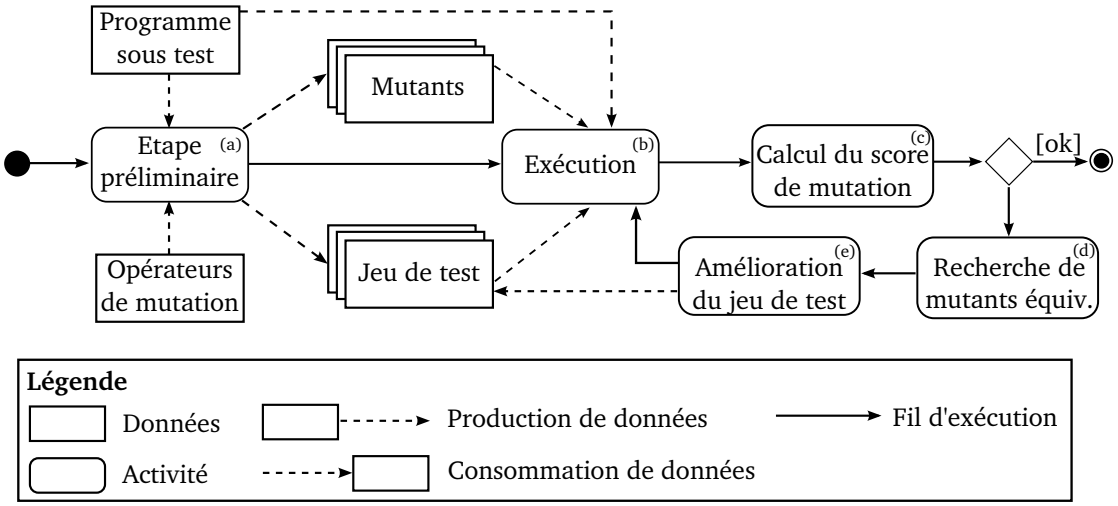


FIGURE 6: Processus de l'analyse de mutation

– Des cas de test comportant données de test et oracles sont présents, auquel cas le processus d'analyse de mutation va s'intéresser aux verdicts produits par ces cas de test. L'étape préliminaire (a) doit s'assurer que tous les cas de test réussissent sur le programme sous test, puis, pour chaque mutant, les cas de test sont exécutés : si l'un des cas de test échoue le mutant est tué, sinon il est toujours vivant.

Une fois un ensemble de mutants et un jeu de test produits, le programme sous test ainsi que l'ensemble de ses mutants sont exécutés sur chacune des données de test (activité (b)). Les sorties respectives sont collectées afin d'être comparées pour produire un verdict sur le statut des mutants (tués ou en vie).

L'étape suivante (c) calcule, à partir du statut des mutants, le score de mutation du jeu de test selon la formule présentée section 2.3.2.1. Cette formule recquiert d'avoir identifié les différents mutants équivalents, mais on peut en pratique les ignorer pour l'instant. En effet, nous savons qu'un mutant équivalent ne peut pas être tué ; il est donc plus pratique de rechercher ceux-ci parmi les mutants en vie que parmi l'ensemble des mutants. Le score de mutation calculé pour le jeu de test doit être évalué par le testeur afin de déterminer s'il est satisfaisant. Un jeu de test obtenant un score de 100 % est dit *adéquat à la mutation*. En pratique, il est assez rare de pouvoir atteindre ce score avec un budget limité : le testeur doit donc placer un seuil au-delà duquel le jeu de test est considéré comme satisfaisant. L'expérience montre que la plupart du temps, le jeu de test fourni par le testeur obtient un score allant de 50 % à 75 %, cependant il est souvent difficile d'atteindre un score de 95 % [25].

L'activité (e) vise à créer de nouvelles données de test afin d'améliorer le score de mutation global du jeu de test. Cette activité est particulièrement coûteuse, puisqu'elle recquiert l'expertise d'un testeur afin de déterminer pour chaque mutant *pourquoi* il n'a pas été tué et *comment* faire pour le tuer. Un certain nombre de travaux ont été menés afin d'automatiser au maximum cette étape, ils sont détaillés en section 2.3.4. Une fois les nouvelles données de test produites, le processus recommence à partir de l'étape (b), et ce jusqu'à ce que le critère de fin fixé par le testeur soit atteint.

2.3.3 Les problèmes de l'analyse de mutation

L'analyse de mutation est une technique efficace pour la détection d'erreurs complexes dans un programme sous test. Cependant, son adoption dans le monde industriel est limitée par un certain nombre de problèmes. Nous abordons dans cette partie les principaux d'entre-eux : les coûts de calcul, l'analyse des mutants équivalents et l'automatisation du processus.

Le problème de l'écriture des oracles [87] s'applique à l'analyse de mutation comme à toutes les autres formes de test et n'est pas considérée dans ce document.

2.3.3.1 Réduction des coûts de calcul

Les opérateurs de mutation sont définis de telle sorte que la majorité des instructions d'un programme est susceptible de muter, et que plusieurs mutants sont souvent créés pour une même instruction. Il en résulte que le nombre de mutants créés pour les programmes, même de petites tailles, est très important.

Puisque chaque mutant doit être complètement exécuté, et ce pour chaque donnée du jeu de test, le coût de l'analyse de mutation est extrêmement important. Afin de palier à ce problème, plusieurs techniques ont été developpées dans la littérature afin de réduire les coûts d'exécution. Elles sont principalement regroupées en trois catégories : « *Do fewer* », « *Do faster* » and « *Do smarter* ».

Do fewer Cette approche vise à réduire le nombre de mutants à étudier. Parmi les techniques permettant d'effectuer une telle sélection, on trouve des techniques de *clustering* de mutants, ou la sélection d'un sous-ensemble des opérateurs de mutation afin de réduire le nombre de mutants produits tout en gardant intacte leur efficacité. Par exemple, Offutt *et al.* ont réduit expérimentalement les opérateurs Mothra de 22 à 5 en conservant une efficacité similaire [59].

Do faster et Do smarter Ces approches visent à améliorer les performances de l'analyse de mutation. De nombreuses techniques ont été développées à cette fin [44], la plus utilisée dans la littérature étant la mutation faible telle qu'introduite en section 2.3.4.

2.3.3.2 Mutants équivalents

Le problème visant à montrer l'équivalence de deux programmes est indécidable [16]. Il n'est donc pas possible de savoir de manière automatique si l'on a créé un mutant équivalent : les mutants en vie à l'issue d'une itération du processus d'analyse doivent être étudiés manuellement afin de déterminer s'ils peuvent ou non être tués par l'insertion d'une nouvelle donnée de test.

Un exemple de mutant équivalent en langage C est listé au tableau 3. La mutation appliquée correspond au remplacement de l'opérateur relationnel `<` par `!=` et correspond à l'opérateur de mutation ROR du tableau 1. En supposant que le bloc d'instructions contenu dans la boucle ne modifie pas la variable de boucle i, le comportement des deux boucles est strictement identique :

P' est donc un mutant équivalent à P, il n'est pas possible de trouver une donnée de test en mesure de le tuer.

Il apparaît que s'il n'est pas possible de détecter les mutants équivalents, il n'est pas possible d'obtenir un score de mutation de 100 %. Malheureusement, des études empiriques montrent que 10 % à 40 % des mutants créés pour un programme sont équivalents [61]. Le problème des mutants équivalents est un problème central de l'analyse de mutation, et fait toujours l'objet de recherches. De nombreuses techniques ont été développées afin de contourner ce problème et de permettre la détection automatique d'une partie des mutants équivalents. Nous renvoyons le lecteur à [44] pour un état de l'art de ce sujet.

PROGRAM P	MUTANT ÉQUIVALENT P'

for(int i = 0 ; i < 5 ; i++)
{
(...i inchangé...)
}
	for(int i = 0 ; i != 5 ; i++)
	{
	(...i inchangé...)
	}

TABLEAU 3: Exemple de mutant équivalent en langage C

2.3.3.3 Automatisation du processus

Afin de faciliter au maximum le travail du testeur, il est nécessaire d'automatiser le plus possible le processus d'analyse de mutation tel que décrit figure 6. Cette section détaille les différentes activités et décrit leur degré d'automatisation.

Génération des mutants La génération des mutants est effectuée par l'application des opérateurs de mutation sur le code source du programme sous test. Cette tâche peut facilement être automatisée dès lors que l'on possède suffisament d'informations sur la nature du langage (*i.e.* sa grammaire) afin de générer des mutants compilables et exécutables.

Génération du jeu de test Plusieurs techniques ont été développées afin de générer automatiquement des données de test pour les programmes. Parmi elles figurent la génération aléatoire [72], la génération par couverture [73], etc.. N'importe quelle technique permettant de générer un jeu de test initial peut être utilisée : l'analyse de mutation se charge de son amélioration en vue de le rendre adéquat à la mutation.

Exécution L'exécution (ainsi que l'éventuelle compilation) des mutants et du programme sous test est une activité entièrement automatisable. Une plateforme d'exécution de l'analyse de mutation doit être en mesure (i) de compiler les mutants (ii) de les exécuter avec chacune des données de test (iii) de collecter les sorties de l'exécution (iv) de déterminer le statut des mutants par comparaison des sorties avec le programme de base.

Calcul du score de mutation Le calcul du score de mutation consiste en un simple calcul mathématique, il est bien sûr automatisable.

Recherche de mutants équivalents Comme précisé en section 2.3.3.2, il n'est pas possible de déterminer de manière automatique si un mutant est équivalent, bien que des heuristiques aient été trouvées pour s'en rapprocher. Cette activité est donc en grande partie laissée à la charge du testeur.

Amélioration du jeu de test Cette activité est considérée comme la plus coûteuse en terme de temps pour l'analyse de mutation et pour les méthodes de test en général. Un état de l'art de la génération automatique de données de test exploitant l'analyse de mutation est détaillé en section suivante.

2.3.4 Génération automatique de données de test

Le coût du test logiciel est souvent estimé à 50 % du coût total du processus de développement [9]. Parmi les activités composant le test de logiciel, la génération de données de test est vue, avec la création d'oracles, comme l'une des activités les plus coûteuses. Bénéficier d'une automatisation partielle ou totale du processus de création des données de test apporterait des gains significatifs aux testeurs. La recherche dans le domaine de la génération automatique de données de test est donc très active [27, 53].

La génération automatique de données de test est souvent guidée par un critère de test en particulier, comme par exemple la couverture de branche. Nous nous intéressons dans cette section à la génération de données de test satisfaisant le critère de mutation. Malheureusement, la littérature concernant la génération de données de test dans le but de tuer des mutants est assez pauvre. Nous détaillons ici la principale approche étudiée dans les travaux traitant de ce sujet : la génération par résolution de contraintes. Une autre approche étudiée est l'utilisation d'algorithmes génétiques [7, 15] ; puisque nos travaux n'utilisent pas directement cette méthode, nous ne la détaillons pas ici.

Dans [23], DeMillo et Offutt stipulent qu'une donnée de test créée dans le but de tuer un mutant doit avoir une caractéristique simple : elle doit faire apparaître une différence de comportement entre le programme original et le mutant. Les auteurs présentent la méthode *Constraint-Based Testing* (CBT) pour générer automatiquement de telles données.

Soit P le programme sous test, M un mutant de P dont l'instruction modifiée est S et T le jeu de test de P. Les auteurs définissent trois contraintes, les « *killing constraints* », nécessaires et suffisantes pour qu'une donnée de test tue un mutant donné :

Atteignabilité (*Reachability*) *L'instruction mutée S doit être exécutée par la donnée de test $t \in T$.* La seule différence entre le programme P et son mutant M se situant à l'instruction S, il est obligatoire que cette instruction soit exécutée par $t \in T$ pour espérer faire apparaître une différence de comportement.

Nécessité (*Necessity*) *Les états internes de P et M doivent être différents immédiatement après l'instruction mutée S pour la donnée de test $t \in T$.* Puisque P et M sont strictement équivalents hors de l'instruction S, si les états respectifs de P et M immédiatement après S ne diffèrent pas, ils ne diffèreront jamais.

Suffisance (*Sufficiency*) *La différence entre les états intermédiaires de P et M après S doit se propager afin que les états finaux de P et M diffèrent.* En effet, le but est de faire apparaître une différence dans les sorties respectives de P et M, ce qui implique que leurs états finaux doivent être différents. Puisque P et M sont strictement identiques après S, la

différence dans les états finaux ne peut se faire que par propagation des différences d'états intermédiaires après S.

La condition d'atteignabilité se résume à un problème de satisfaction de chemin [18] dont le but est de trouver pour quelles valeurs des variables en entrée du programme l'instruction S est exécutée. L'exécution symbolique [48] permet justement de construire des contraintes de chemin exprimant, en fonction des variables en entrée, le chemin que prendra l'exécution du programme, et donc les instructions qui seront exécutées. Typiquement, un graphe de flot de contrôle [9] est construit, l'instruction S y est localisée et un moteur d'exécution symbolique collecte les contraintes symboliques menant à ce chemin. Les contraintes ainsi obtenues sont ensuite réécrites en terme de variables d'entrée, puis résolues de manière à trouver une valeur pour chacune des variables en entrée telle que S soit exécutée.

La condition de nécessité stipule que les états internes de P et M immédiatement après S doivent être différents. Cette condition est réécrite sous la forme (*instruction mutée* $\neq$ *instruction originale*). Par exemple, en considérant le mutant qui remplace l'instruction `a > b` par l'instruction `a >= b`, on obtient la condition $(a > b) \neq (a >= b)$ qui est réécrite en terme de contraintes de la manière suivante : $((a > b) \wedge \neg(a >= b)) \vee (\neg(a > b) \wedge (a >= b))$ [90]. Ces contraintes sont combinées aux contraintes de chemin exprimées par la condition d'atteignabilité afin de dériver des données de test qui (i) atteignent l'instruction mutée S et (ii) induisent des états différents de P et M après S.

Un problème se pose avec la condition de suffisance : il n'est pas possible de connaître à l'avance le chemin complet que l'exécution d'un programme va suivre [63]. Par suite, on ne peut pas automatiquement trouver des valeurs pour les variables d'entrée qui satisfont cette contrainte. Afin de palier à ce problème, Howden *et al.* introduisent la notion de *mutation faible* [41], en opposition à la mutation « classique » renommée *mutation forte*. Une donnée de test tue faiblement un mutant si elle satisfait les critères de la mutation faible, et le tue fortement si elle respecte les critères de la mutation forte. Les définitions suivantes illustrent la différence entre ces deux approches :

Definition. Une donnée de test $t \in T$ tue faiblement un mutant M si et seulement si t satisfait les conditions d'atteignabilité et de nécessité de M

Definition. Une donnée de test $t \in T$ tue fortement un mutant M si et seulement si t satisfait les conditions d'atteignabilité, de nécessité et de suffisance de M

Différentes études, dont [60], montrent qu'une donnée de test tuant faiblement un mutant a de fortes chances de le tuer fortement. La plupart des travaux actuels utilisent donc la notion de mutation faible pour la génération de données de test destinées à tuer des mutants.

CBT souffre de nombreuses limitations : cette technique ne permet pas de gérer efficacement les boucles, tableaux ou pointeurs dynamiques. La méthode *Dynamic Domain Reduction* (DDR) a donc été proposée afin de résoudre un certain nombre de ses problèmes en utilisant des techniques de recherche plus sophistiquées [64].

9. Un graphe de flot de contrôle est une représentation sous la forme d'un graphe de tous les chemins que peut suivre un programme durant son exécution.

Plus récemment, l'exécution symbolique dynamique [38, 76, 90] a été utilisée pour résoudre les problèmes liés à l'utilisation de l'exécution symbolique pour la construction des contraintes de chemin. L'exécution symbolique dynamique met en œuvre l'exécution symbolique tout en exécutant réellement le programme lorsque cela est nécessaire pour lever les limites de l'exécution symbolique comme les boucles ou les structures dynamiques.

Chacune de ces méthodes diffère des autres dans sa réalisation et dans les optimisations utilisées, mais toutes se basent sur les « *killing constraints* » définies par DeMillo et Offutt dans [23]. Ces contraintes sont utilisées pour la génération par résolution de contraintes que nous venons de présenter, mais aussi pour la génération par l'utilisation d'algorithmes génétiques [15].

2.4 Le test des transformations de modèles

Comme nous l'avons montré en section 2.2, les transformations de modèles sont un point crucial de l'ingénierie dirigée par les modèles : l'automatisation des différents processus au cœur de l'IDM repose sur elles. Les différentes transformations de modèles intégrées par exemple dans des ateliers logiciels sont destinées à être génériques et réutilisées de nombreuses fois. Une erreur dans une transformation peut donc être à l'origine de nombreux modèles erronés, ces erreurs se répercutant au fil du processus de développement. Par conséquent, elles doivent être dignes de confiance afin d'assurer la qualité du processus global [33].

Les programmes de transformation de modèles sont des programmes « classiques », à ceci près qu'ils manipulent des données complexes : des modèles conformes à des métamodèles (cf. section 2.2.3). En fait, les contraintes auxquelles doivent se plier des modèles de test pour les transformations de modèles sont multiples :

Conformité au métamodèle Un modèle de test doit être conforme au métamodèle d'entrée de la transformation.

Invariants du métamodèle Souvent, l'écriture d'un métamodèle ne permet pas de stipuler toutes les contraintes implicites qu'il suppose. Ainsi, il est courant de spécifier un certain nombre de contraintes (le plus souvent en OCL [10]) sur celui-ci qui viennent s'y ajouter.

Préconditions de la transformation Une transformation de modèle ne peut traiter qu'un sous-ensemble de tous les modèles conformes à son métamodèle d'entrée. Un modèle de test correct doit donc aussi les respecter.

Intention de test En plus de toutes ces contraintes, un modèle de test doit correspondre à un objectif, une intention de test. Cette intension de test est spécifiée directement par le testeur en fonction de sa connaissance de la transformation. Les modèles de test générés doivent donc y correspondre.

La complexité des données manipulées rend l'écriture de transformations de modèles ardue et augmente le risque d'erreur. Adapter les techniques de test développées depuis des décennies

10. L'*Object Constraint Language* est un langage déclaratif permettant d'exprimer des propriétés sur des modèles. Initialement proposé pour complémenter UML et standardisé par l'OMG en 1997, il est aujourd'hui largement utilisé dans l'IDM pour spécifier des invariants sur les modèles ou des préconditions et postconditions sur les transformations de modèles

pour les programmes classiques est un bon moyen de s'assurer de leur correction. Cependant, les opérations particulières que proposent les transformations de modèles (*e.g.* filtrage de collections, navigation d'associations) ne sont pas utilisées dans la programmation traditionnelle, et les techniques de test doivent donc être adaptées [8]. Nous présentons ici plusieurs travaux portant sur le test des transformations de modèles.

Guerra [39] propose de générer automatiquement des modèles de test pour les transformations de modèles en se basant sur leurs spécifications. Les préconditions, postconditions et invariants des transformations sont définis dans un langage formel, et des techniques de résolution de contraintes sont utilisées pour générer des modèles de test couvrant les propriétés des spécifications.

Dans [2], Andrews *et al.* proposent une adaptation de l'analyse de partition [71] aux diagrammes de classes UML. Pour cela, trois critères de couverture sont définis (couverture des multiplicités, des attributs, et des hiérarchies de types). Les concepts de classe et d'association présents dans UML et MOF étant similaires, Fleurey *et al.* proposent dans [33] d'élargir ces critères de couverture aux métamodèles conformes à MOF. Les auteurs introduisent un nouveau critère de test pour les transformations de modèles reposant sur la couverture du métamodèle d'entrée de la transformation.

Dans [32], de nouveaux critères pour la couverture du métamodèle d'entrée sont définis. À chacun de ces critères est associé un ensemble de fragments de modèles qui doivent être couverts par le jeu de test d'une transformation. Un fragment de modèle correspond par exemple à la couverture des valeurs possibles d'une multiplicité d'association. De plus, un métamodèle est proposé pour formaliser les notions de partitions et de fragments de modèles. Puisque les partitions ne reposent que sur le métamodèle d'entrée de la transformation, la méthode est indépendante du langage de transformation utilisé. Ces concepts ont été intégrés à l'outil Pramana par Sen *et al.* [78].

Dans [57], une analyse statique de la transformation sous test est entreprise afin d'identifier les éléments du métamodèle d'entrée effectivement utilisés. Les éléments identifiés sont ensuite partitionnés afin de générer les fragments de modèles correspondants. Enfin, ces fragments sont réécrits en terme de contraintes afin de générer automatiquement des modèles de test les couvrant.

Les modèles de test générés par ces différentes méthodes ont été évalués grâce à l'analyse de mutation, mais aucune d'elles n'a pu atteindre un score de 100 %. Les différentes expérimentations décrites rapportent des scores de 70 % dans [39], 89,9 % dans [78] et 97,62 % dans [57]. De plus, ces méthodes de génération ont été évaluées sur le critère de mutation, mais n'ont pas pour objectif propre de générer des modèles de test adéquats à la mutation. Le cœur de notre travail est justement de combler ce manque en générant automatiquement des modèles de test dont l'objectif est d'atteindre un haut score de mutation.

Chapitre 3

Vers une automatisation de l'analyse de mutation des transformations de modèles

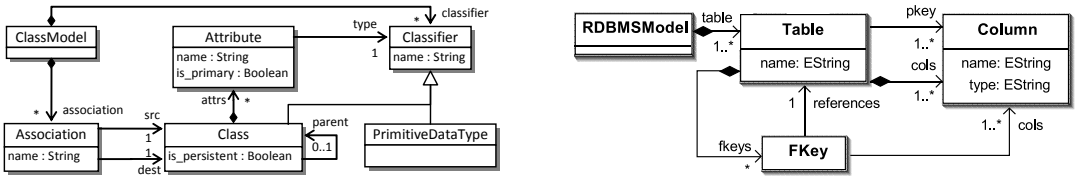
3.1 Introduction

Nous avons présenté au chapitre 2 le rôle des transformations de modèles au sein de l'IDM et différents travaux portant sur le test des transformations de modèles. Cependant, aucun d'entre-eux ne vise à produire de manière automatique des modèles de test adéquats à la mutation.

Nous introduisons ici une nouvelle approche pour la génération de tels modèles. En particulier, nous nous intéressons à la phase d'amélioration du jeu de test pour l'analyse de mutation. Dans ce contexte, les données de test sont des modèles. Les opérateurs de mutation pour les transformations de modèles que nous utilisons ont déjà été décrits par Mottu *et al.* dans [56].

L'approche décrite ici repose sur une représentation abstraite des opérateurs de mutation de [56] et sur un mécanisme de traçabilité adapté de [6] permettant de faire correspondre les éléments des modèles d'entrée aux éléments des modèles de sortie qu'ils produisent. Les principales contributions sont les suivantes :

– Une modélisation précise des opérateurs de mutation est réalisée et les modèles associés sont mis en relation avec les liens de trace produits par les différentes exécutions. Nous disposons ainsi d'un modèle abstrait où sont collectés les résultats des exécutions ; pour chacune d'entre-elles nous savons quel est le modèle de test mis en jeu, le mutant associé et son opérateur ainsi que le verdict produit par comparaison des modèles de sortie. Lorsqu'un mutant est encore en vie, la trace nous permet de savoir quels modèles et quelles parties de ces modèles doivent être modifiés pour le tuer.
– La seconde contribution consiste en l'identification de configurations connues (*patterns*) dans les modèles d'entrée qui, pour un opérateur de mutation donné, peuvent laisser un mutant



(a) Métamodèle de diagramme de classe simple

(b) Métamodèle de base de données relationnelle simple

FIGURE 7: Métamodèles d'entrée et de sortie de la transformation `class2rdbms`

en vie. Des heuristiques sont associées à chacun de ces patterns, indiquant des modifications simples à appliquer aux modèles afin que ceux-ci tuent le mutant sélectionné.

– L'approche est validée sur une transformation réalisant la mise à plat d'un automate fini. 8 modèles sont créés de manière semi-automatique et le score de mutation du jeu de test passe de 45 % à 100 %. Le travail du testeur est fortement facilité en réduisant de 87 % le nombre d'éléments à analyser dans les modèles d'entrée.

Les différents exemples utilisés au cours de ce chapitre se basent sur la transformation `class2rdbms` qui transforme un modèle de diagramme de classe simple en un modèle de base de données relationnelle. Cette transformation a été proposée comme *benchmark* au workshop MTIP de la conférence MoDELS 2005 [13] pour valider les différentes fonctionnalités des langages de transformation de modèles et est depuis prise en exemple de nombreux travaux sur le sujet. Les métamodèles d'entrée et de sortie de la transformation `class2rdbms` sont respectivement décrits par les figures 3.7(a) et 3.7(b).

Les travaux présentés ici ont conduit à l'écriture d'un article [4, 5] actuellement en cours de revue pour publication dans le journal *Software Testing, Validation and Reliability* (STVR). Par soucis de concision, nous ne pouvons pas détailler ici l'ensemble des métamodèles, patterns et recommandations créés pour les opérateurs de mutation et l'ensemble des étapes de l'expérimentation. Nous renvoyons le lecteur vers [4] pour une description exhaustive de ces éléments.

3.2 L'analyse de mutation des transformations de modèles

L'analyse de mutation est une technique permettant d'évaluer la qualité d'un jeu de test en fonction de sa capacité à détecter des erreurs réelles dans un programme sous test (cf. section 2.3). A partir d'un programme sous test, un ensemble de mutants est créé par l'insertion d'une et une seule erreur par mutant correspondant à celles qu'un programmeur peut commettre durant la phase de développement. L'insertion de ces erreurs est réalisée par des opérateurs de mutation (cf. section 2.3.2.2).

La plupart des travaux proposent l'écriture d'opérateurs de mutation en relation directe avec la syntaxe d'un langage particulier. Des opérateurs ont ainsi été écrits pour nombre de langages, parmi lesquels Java, C, Fortan, ou ADA. À l'inverse, certains travaux proposent de définir les opérateurs de mutation indépendamment d'une syntaxe particulière. Dans [81], Simão *et al.* proposent MuDeL, un langage dédié à l'écriture d'opérateurs de mutation indépendants de la syntaxe

du langage effectivement utilisé. Les opérateurs ainsi définis de manière générique peuvent ensuite être compilés afin d'être adaptés à la syntaxe d'un langage en particulier. D'autres travaux considèrent par exemple la généralisation des opérateurs de mutation au niveau d'un paradigme en particulier, comme par exemple pour la programmation orientée aspects [31]. Cependant, aucun de ces travaux ne considère en particulier les transformations de modèles et leurs spécificités.

Dans [35] et [85], les auteurs proposent de définir les opérateurs de mutation pour les transformations de modèles par des transformations d'ordre supérieur [1]. Tous les langages de transformations de modèles ne disposent cependant pas d'une modélisation des transformations sous forme de modèles : cette méthode ne peut donc être appliquée qu'aux langages proposant une métamodélisation des transformations comme Kermeta ou ATL, et doit être adaptée en fonction de celle-ci.

Dans [56], Mottu *et al.* définissent 10 opérateurs de mutation pour les transformations de modèles. Ces opérateurs sont directement liés aux opérations spécifiques que réalisent les transformations de modèles (*i.e.* la navigation dans les modèles, le filtrage de collections, la création d'éléments dans les modèles de sortie et la modification d'éléments existants dans les modèles d'entrée) et sont ainsi indépendants de la syntaxe particulière d'un langage de transformation de modèles. Ces opérateurs sont spécifiquement conçus pour simuler les erreurs réelles pouvant être commises par les programmeurs de transformations de modèles. L'analyse de mutation que nous détaillons ici exploite ces opérateurs.

Le processus global pour l'analyse de mutation des transformations de modèles est le même que celui de l'analyse de mutation classique tel que décrit par la figure 6. Toute analyse de mutation supposant l'existence préalable d'un jeu de test, celui-ci peut être généré automatiquement par l'utilisation des méthodes présentées à la section 2.4 comme la couverture par partionnement du métamodèle d'entrée. La génération des mutants est réalisée en appliquant les opérateurs de mutation spécifiques proposés dans [56] (cf. section 3.8.3). L'exécution et la comparaison des sorties des différents mutants peut être réalisée automatiquement à l'aide d'outils appropriés comme EMFCompare [26].

Le principal problème réside dans l'amélioration automatique du jeu de test : même si des méthodes ont été proposées pour créer automatiquement des données de test destinées à tuer des mutants (cf. section 2.3.4), aucune d'elles ne s'intéresse au cas spécifique des transformations de modèles.

Puisque nous ne disposons pas de méthodes pour générer automatiquement des modèles de test destinés à tuer des mutants, chaque mutant en vie doit être analysé de manière statique (par analyse statique) et de manière dynamique (il est nécessaire de réexécuter la transformation) afin de déterminer pourquoi aucun modèle de test ne l'a tué, et comment faire pour y arriver. L'approche que nous proposons ici repose sur l'hypothèse suivante : puisque les transformations de modèles manipulent des données complexes (des modèles conformes à des métamodèles), il peut être extrêmement compliqué de construire des modèles de test qui (i) sont conformes au métamodèle d'entrée (ii) respectent les préconditions de la transformation et (iii) permettent

1. Les *Higher-Order model Transformations* (HOT) sont des transformations manipulant d'autres transformations décrites par des modèles

de tuer un mutant. À l'inverse, réutiliser des modèles de test existants pour en construire de nouveaux semble beaucoup moins coûteux : des modifications simples au sein d'un modèle de test peuvent l'amener à tuer d'autres mutants. Les réponses aux trois questions formulées ci-après doivent nous donner suffisamment d'information pour savoir quels modèles de test modifier, et de quelle manière :

- Parmi les paires (*modèle de test*, *mutant*) existantes, lesquelles sont les plus intéressantes à étudier ?
- À quoi devrait ressembler le modèle de sortie pour que le mutant soit tué ? Quelle différence souhaite-t-on y faire apparaître ?
- Comment modifier le modèle de test de manière à produire la différence attendue dans le modèle de sortie, et ainsi tuer le mutant ?

Nous proposons dans ce document de répondre à ces trois questions de manière automatique, afin de pouvoir augmenter le niveau d'automatisation du processus d'amélioration du jeu de test pour l'analyse de mutation des transformations de modèles. La réponse à la première question peut être obtenue en exploitant les informations obtenues par la traçabilité des éléments lors d'une transformation et est détaillée en section 3.3. La réponse aux deux dernières questions est apportée par la détection de configurations connues au sein des modèles de test, et l'application d'heuristiques associées pour la génération de nouveaux modèles de test destinés à tuer des mutants.

3.3 Traçabilité pour les transformations de modèles

D'après le glossaire de l'IEEE, « *Traceability allows one to establish degrees of relationship between products of a development process, especially products bound by a predecessor-successor or master-subordinate relationship* » [37]. Plus précisément, dans le cadre des transformations de modèles, les liens de trace relient les éléments de différents modèles en spécifiant lesquels sont utiles à la génération des autres [6].

3.3.1 Traçabilité des éléments

Plusieurs mécanismes de traçabilité ont été proposés pour les transformations de modèles, mais ils sont le plus souvent dédiés à un langage spécifique comme Kermeta [29] ou ATL [46, 89]. Dans [3, 6], Aranega *et al.* proposent une approche pour la traçabilité indépendante des langages de transformations où les créations et modifications d'éléments sont représentées par des liens de trace. Chaque lien de trace spécifie (i) un ensemble d'éléments sources (ii) un ensemble d'éléments cibles et (iii) la règle de transformation qui a mené à cette création ou modification. La notion de *règle de transformation* peut être plus ou moins explicite en fonction du langage de transformation utilisé (déclaratif ou impératif). Chaque lien de trace est toujours associé à une règle de transformation. Chaque règle peut être appliquée plusieurs fois sur des éléments sources différents afin de produire différents éléments en sortie. La figure 8 présente des exemples de liens de trace créés lors d'une exécution de la transformation `class2rdbms`. Par exemple, le lien *Link3*

stipule que les attributs `Association.name = 'address'` et `Attribute.name = 'street'` du modèle d'entrée ont été consommés pour produire l'attribut `Column.name = 'address_street'` du modèle de sortie via la règle de transformation `createColumns`.

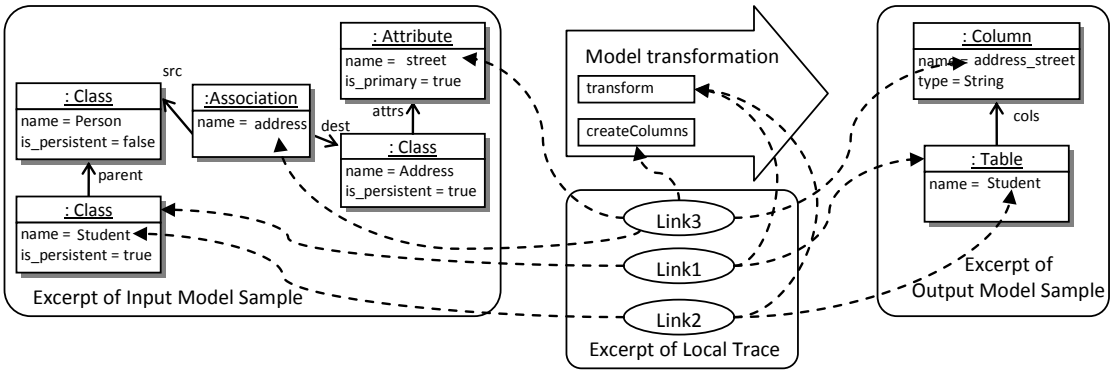


FIGURE 8: Exemple de trace locale pour `class2rdbms`

Dans [6], un métamodèle de trace locale est proposé pour recueillir l'ensemble des liens de trace créés lors d'une transformation. Les modèles conformes à ce métamodèle sont totalement découplés du langage de transformation utilisé et peuvent être analysés indépendamment. Cependant, la génération automatique du modèle de trace locale d'une transformation est elle dépendante du langage utilisé : ici, nous utilisons le langage Kermeta, et la génération du modèle de trace se fait par l'insertion d'appels de trace directement dans le code de la transformation [2] [29]. Nous avons par ailleurs apporté quelques améliorations au métamodèle de [6] afin de faciliter, au-delà des instances de classes, la traçabilité des attributs et des références (cf. section 3.8.4). En effet, les opérateurs de mutation utilisés affectent tous ces éléments. Nous avons donc apporté un incrément au métamodèle de [6] en définissant de nouvelles sous-classes de $ModelRef$ capables de tracer ces éléments. Le métamodèle finalement utilisé pour la traçabilité des transformations est présenté figure 9. La métaclasse *EObject* situé à l'extrêmité droite du diagramme est un concept défini par ECore [3] et permet de référencer n'importe quel élément au sein d'un modèle. Les différentes métaclasses *AttributeRef*, *ClassRef* et *ReferenceRef* sont utilisées pour tracer ces différents types d'éléments. Les liens de trace sont portés par la métaclasse *Link*, qui associe une règle à un ensemble d'éléments sources et cibles.

3.3.2 Matrice de mutation

Dans l'analyse de mutation, l'ensemble des mutants créés à partir de la transformation sous test sont exécutés sur l'ensemble des modèles contenus dans son jeu de test. Pour chacune de ces exécutions, un modèle de trace locale est produit contenant l'ensemble des liens de trace reliant éléments en entrée ou en sortie et règles de la transformation. Habituellement, les résultats de l'analyse de mutation sont collectés dans une matrice où les deux dimensions correspondent respectivement aux mutants et aux données de test, chaque cellule indiquant si la donnée de test a tué ou non le mutant. Ainsi, le statut final d'un mutant peut être automatiquement déduit à

2. D'autres travaux considèrent la création des liens de traçabilité sans altérer le code source de la transformation sous test, comme par exemple pour QVTo [3]
3. ECore est une technologie intégrée au sein du framework EMF d'Eclipse implémentant les concepts décrits par la norme EMOF de l'OMG

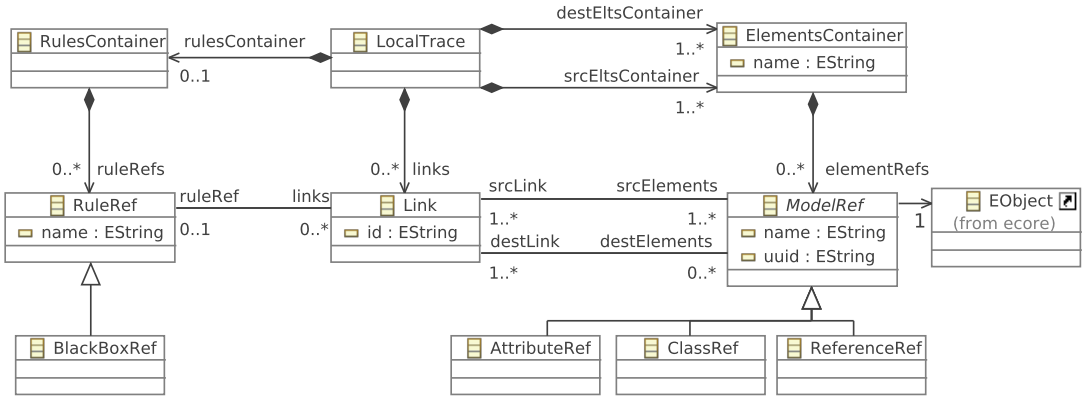


FIGURE 9: Métamodèle de trace locale

partir de l'ensemble des cellules constituant sa ligne (au moins l'une des données de test doit avoir tué le mutant).

Dans [6], les résultats de l'exécution des mutants sur les modèles de test ainsi que les modèles de trace locale produits sont collectés dans une *matrice de mutation*. Une cellule correspond à l'abstraction de l'exécution d'un mutant sur un modèle de test. En associant les modèles de trace à chacune des exécutions, la matrice devient un modèle pivot entre les différents niveaux : mutants, modèles de test et traces locales. Ces concepts sont illustrés par la figure 10 : les mutants ($T_0 ... T_n$) sont représentés en colonnes, les modèles de test ($m_0 ... m_m$) en lignes, et chaque cellule ($C_{00} ... C_{mn}$) contient le verdict (*tué* ou *en vie*) ainsi qu'un lien vers le modèle de trace locale ($lt_{00} ... lt_{mn}$) associé.

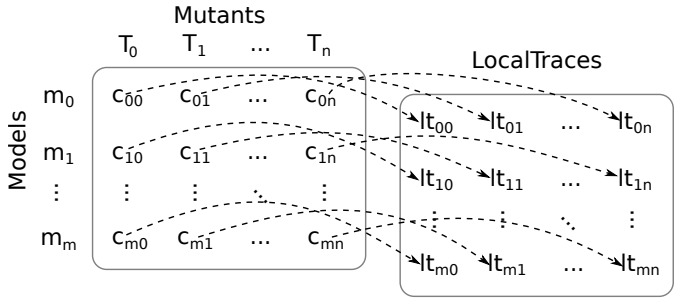


FIGURE 10: Matrice de mutation et modèles de trace locale

La figure 11 détaille les différentes activités liées à l'exécution des différents mutants pour prendre en compte la traçabilité et la construction de la matrice de mutation. Le processus décrit correspond au détail de l'activité (b) de la figure 6. Cette étape prend en entrée la transformation sous test T, les mutants T_i et l'ensemble des modèles de test. Pour chacune des transformations (originale ou mutée), une trace locale est produite lors de l'exécution et associée au couple $(modèle\ de\ test, mutant)$ correspondant. Les résultats des exécutions (*i.e.* les modèles de sortie) sont ensuite comparés afin de produire un verdict sur le statut du mutant. Enfin, le verdict ainsi que le modèle de trace locale associé sont stockés dans la cellule correspondante de la matrice de mutation.

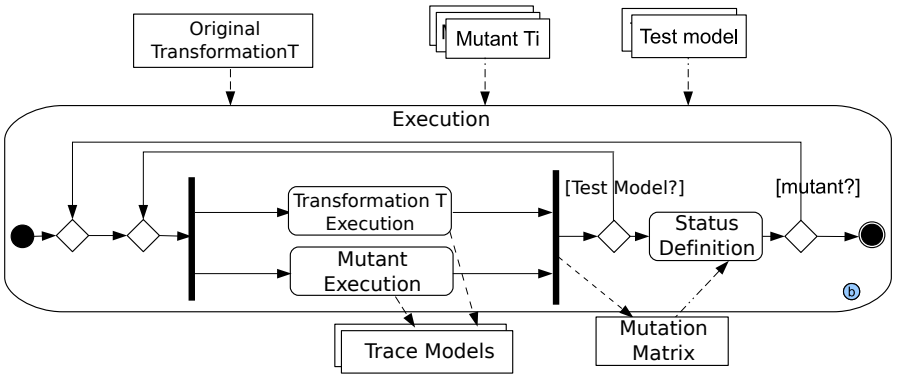


FIGURE 11: Génération des traces locales et de la matrice de mutation

La matrice de mutation étant elle-même un modèle, elle peut être automatiquement produite à l'aide d'outils dédiés [6]. La figure 12 présente le métamodèle de matrice de mutation utilisé. À chaque cellule sont associés : le mutant exécuté (avec notamment la règle qu'il modifie et l'opérateur de mutation utilisé), le modèle de test utilisé et la trace locale produite par l'exécution. On retrouve bien ici les différentes dimensions de la matrice telles que décrites figure 10.

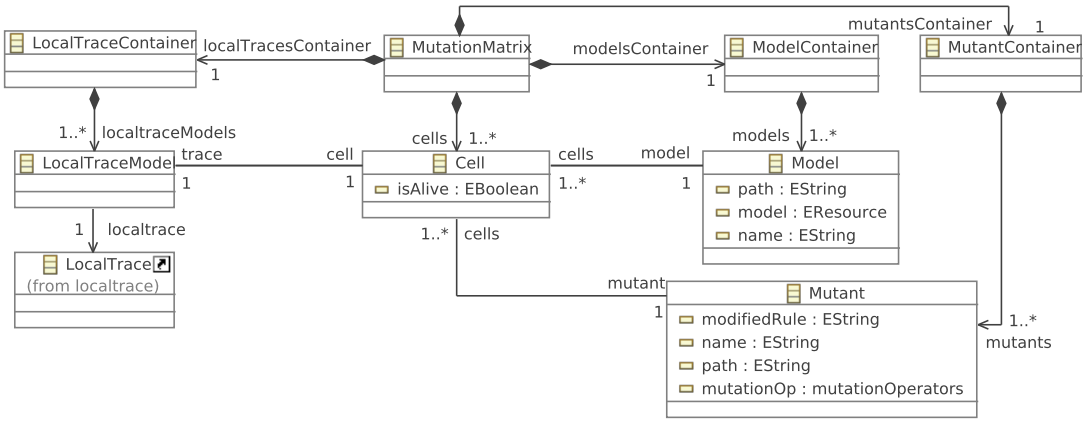


FIGURE 12: Métamodèle de la matrice de mutation

3.3.3 Identification des paires $(modèle, mutant)$ pertinentes

Nous tentons dans cette section d'apporter une réponse automatique à la question : « *Parmi les paires* $(modèle\ de\ test, mutant)$ *existantes, lesquelles sont les plus intéressantes à étudier ?* ». Pour chaque mutant en vie à l'issue des exécutions, les modèles de test ainsi que les modèles de trace locale associés à leurs exécutions sont collectés. Notre intuition est que les modèles les plus intéressants à étudier sont ceux pour lesquels la règle mutée a été traversée. En effet, pour tuer un mutant, il est nécessaire d'exécuter son intruction mutée. Les modèles de trace locale construits lors des exécutions associent les éléments en entrée et sortie aux règles qui les manipulent : il est donc possible d'y récupérer cette information en analysant les liens de trace.

Les figures 13 et 14 présentent en détail l'activité (e) de la figure 6 (p. 18) correspondant à l'étape d'amélioration du jeu de test. Le processus commence par la sélection d'un mutant parmi

ceux en vie : cette information peut être facilement récupérée en parcourant la colonne associée à un mutant dans la matrice de mutation (si aucune des cellules associées n'a marqué le mutant comme tué, alors il est en vie). Le processus se poursuit par l'identification des modèles de test pertinents. En parcourant les cellules associées au mutant, les modèles de trace locale sont analysés afin de déterminer s'ils ont déclenché la règle mutée. Si aucun des modèles de test n'a déclenché la règle mutée, un nouveau modèle de test doit être créé en prenant soin de satisfaire les préconditions de cette règle. Si un ou plusieurs modèles de test ont déclenché la règle mutée, l'algorithme les renvoie, en y associant pour chacun la liste des éléments mis en jeu en entrée et la liste des éléments produits en sortie. Ainsi, le testeur sait précisément quels éléments de quels modèles sont intéressants pour tuer le mutant : il n'a pas besoin de parcourir exhaustivement tous les éléments de tous les modèles de test.

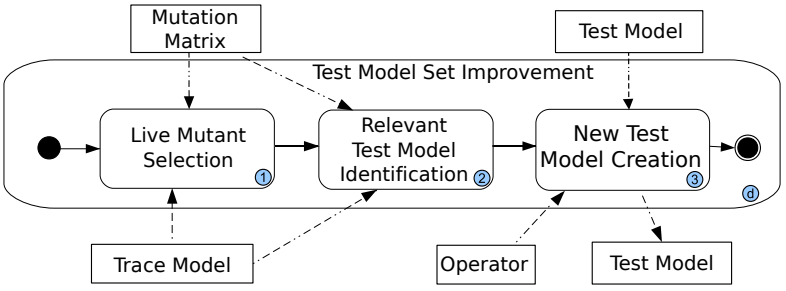


FIGURE 13: Processus d'amélioration du jeu de test

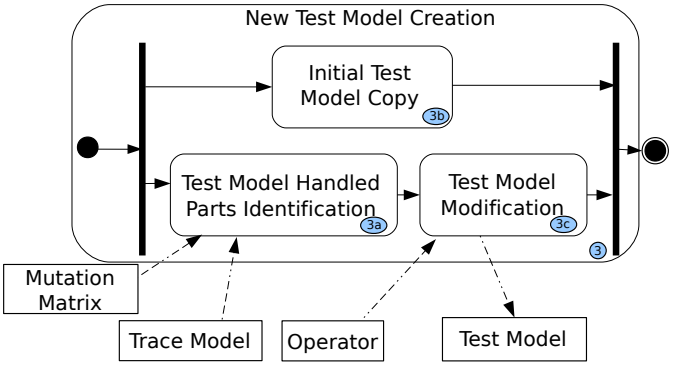


FIGURE 14: Processus de création d'un nouveau jeu de test

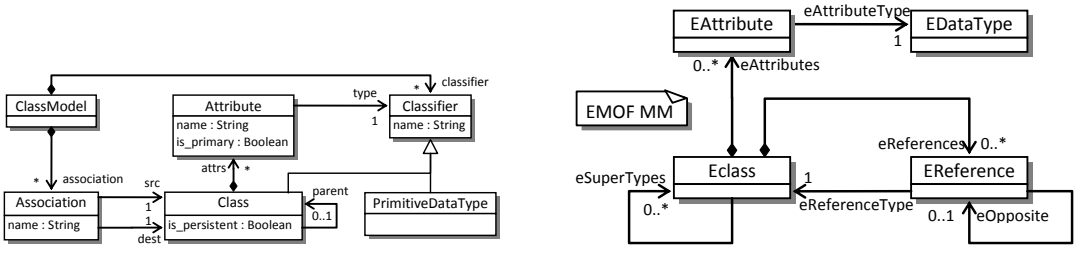
Afin de palier à un éventuel phénomène de régression, le modèle de test choisi pour modification est tout d'abord copié. Les éléments pertinents à l'intérieur de celui-ci sont ensuite identifiés automatiquement par l'algorithme à partir du modèle de trace locale, et le testeur peut produire un nouveau modèle de test qui est intégré au jeu de test de la transformation. Afin d'assister à la création de ce nouveau modèle de test, il est utile de connaître précisément la modification réalisée par l'application de l'opérateur de mutation à l'origine du mutant. Dans ce but, une modélisation précise desdits opérateurs est nécessaire : nous la présentons en section suivante.

3.4 Modélisation des opérateurs de mutation

3.4.1 Principe

Les opérateurs de mutation pour les transformations de modèles de [56] ne sont présentés qu'à l'aide de descriptions informelles. Pourtant, l'information qu'ils portent (*i.e.* la différence sémantique entre la transformation originale et l'un de ses mutants) doit pouvoir être exploitée de manière automatique afin de (i) faciliter la création de nouveaux modèles de test et (ii) générer automatiquement les mutants d'une transformation. Afin de permettre ce raisonnement automatique, nous proposons ici un métamodèle des opérateurs de mutation. Afin de rester totalement indépendant du langage de transformation utilisé, ce métamodèle s'attache à décrire les effets des opérateurs en terme de données manipulées et non en terme de syntaxe : il décrit l'effet des opérateurs sur la manipulation des métamodèles d'entrée et de sortie de la transformation. Les modèles créés à partir de ce métamodèle sont appelés *modèles de mutation*.

Afin de permettre l'exploitation des opérateurs de mutation indépendamment des métamodèles utilisés par la transformation, nous exploitons une caractéristique des métamodèles : ils sont décrits par un méta-métamodèle, dans notre cas EMOF. La figure 15 illustre la relation de conformité entre le métamodèle d'entrée de la transformation `class2rdbms` et son méta-métamodèle EMOF : par exemple, $Classifier$ est une $EClass$, $is\_primary$ est un $EAttribute$, $attrs$ est une $EReference$ et $Boolean$ est un $EDataType$. En décrivant les actions des opérateurs de mutation directement au niveau du méta-métamodèle EMOF, il est possible d'écrire des modèles de mutation pour n'importe quel métamodèle s'y conformant.



(a) Métamodèle de diagramme de classe UML simple

(b) Extrait du méta-métamodèle EMOF

FIGURE 15: Métamodèle de diagramme de classe simple et son méta-métamodèle EMOF

Les modèles de mutation ainsi créés doivent spécifier comment les éléments des métamodèles de la transformation peuvent être traités, et de quelle manière les mutants modifient ces traitements. En pratique, une transformation peut effectuer zéro, une, ou plusieurs fois un traitement (par exemple, naviguer une certaine référence). Par conséquent, chaque modèle de mutation peut mener à la création de zéro, un, ou plusieurs mutants. L'application des modèles de mutation à une transformation particulière dépend à la fois des traitements effectués par cette transformation et du langage de transformation utilisé : certains mutants créés peuvent par exemple ne pas compiler si le langage utilisé est fortement typé. De manière à maintenir un lien entre chaque mutant et le modèle de mutation qui décrit sa sémantique, chaque mutant $(T_0...T_n)$ est associé à son modèle $(Op_1...Op_k)$ dans la matrice de mutation ainsi que décrit par la figure 16.

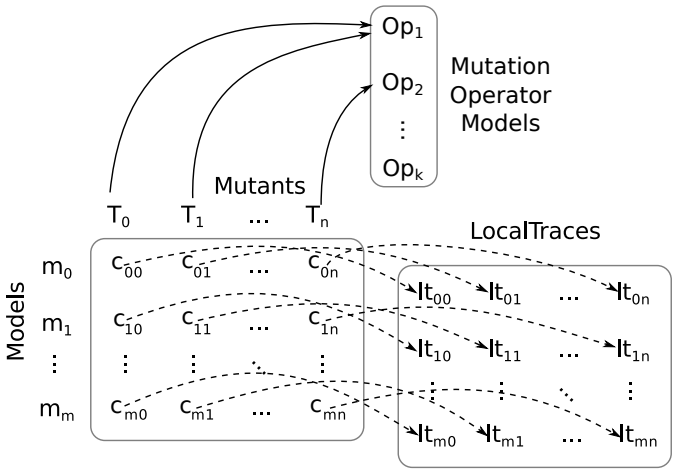


FIGURE 16: Association des modèles de mutation aux mutants dans la matrice de mutation

Afin d'illustrer nos propos, nous présentons dans la section suivante la métamodélisation d'un des dix opérateurs de mutation : *Relation to Same Class Change* (RSCC). L'intégralité des métamodèles des opérateurs de mutation de [56] ainsi que des exemples d'instanciation sont disponibles en annexe et dans [4].

3.4.2 Exemple : métamodélisation de l'opérateur RSCC

Conformément à la définition faite dans [56], l'opérateur RSCC remplace la navigation d'une association vers une classe par la navigation d'une autre association menant à la même classe.

Cet opérateur altère les opérations de navigation des transformations de modèles. Il peut être appliqué à la fois sur les métamodèles d'entrée et de sortie d'une transformation, à condition que ceux-ci présentent au moins deux associations différentes entre deux mêmes classes. En utilisant la terminologie du méta-métamodèle EMOF, cela revient à dire que le métamodèle contient au moins deux $ERef erence$ entre deux $EClass$ identiques. La figure 17 présente le métamodèle de cet opérateur en relation avec EMOF.

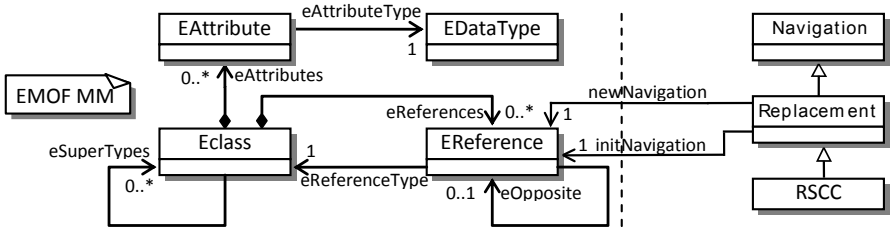


FIGURE 17: Métamodèle de l'opérateur RSCC

Chaque opérateur est représenté par une métaclasse (RSCC ici). Les différents opérateurs sont hiérarchisés au sein d'un ensemble de métaclasses : l'opérateur RSCC est un opérateur de navigation opérant un remplacement (à l'inverse par exemple de l'opérateur RSMA qui ajoute une nouvelle navigation). La métaclasse RSCC possède deux références vers la méta-métaclasse $ERef erence$ d'EMOF : $initNavigation$ correspond à l'association naviguée par la transformation originale, $newNavigation$ à l'association naviguée par le mutant. Nous disposons ainsi de

toute l'information nécessaire pour comprendre la différence sémantique entre la transformation originale et son mutant.

Le métamodèle présenté graphiquement ne contient cependant pas toute l'information nécessaire pour l'application correcte de l'opérateur RSCC. Par exemple, il n'est nul part indiqué que les deux références pointées doivent partir de la même métaclasse pour arriver à la même métaclasse. Afin d'ajouter la sémantique manquante au métamodèle, les contraintes OCL présentées au listing 3.1 lui sont adjointes. Le premier invariant permet de s'assurer que le mutant créé ne sera pas un mutant équivalent. Le second invariant précise que la référence naviguée par le mutant doit pointer vers la même métaclasse que la transformation originale. Enfin, le troisième invariant vérifie que les métaclasses à la source des références appartiennent à la même hiérarchie de type.

```
context RSCC
  inv differentRefs :
    self.initNavigation <> self.newNavigation
  inv sameTargets :
    self.initNavigation.eReferenceType = self.newNavigation.eReferenceType
  inv sameOwnersTypeHierarchy :
    not self.initNavigation.eContainingClass.eAllSuperTypes
    ->including(self.initNavigation.eContainingClass)
    ->intersection(
      self.newNavigation.eContainingClass.eAllSuperTypes
      ->including(self.newNavigation.eContainingClass)
    )->isEmpty()
```

Listing 3.1: Contraintes OCL sur l'opérateur RSCC

La figure 18 présente un exemple d'instanciation de l'opérateur RSCC sur le métamodèle d'entrée de la transformation `class2rdbms`. Ce modèle de mutation est appliqué à la transformation à chaque fois que celle-ci navigue la référence *Association.dest*, la remplaçant par la référence *Association.src* et créant à chaque fois un nouveau mutant. Le listing 3.2 présente un extrait du pseudo-code d'un mutant créé par application de ce modèle de mutation.

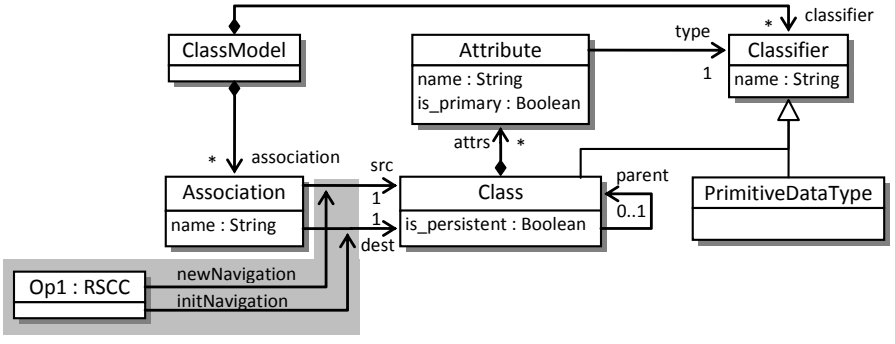


Figure 18: Exemple d'application de l'opérateur RSCC sur la transformation `class2rdbms`

```
Association assoc;
[...]
//assoc.dest := X // Navigation originale
assoc.src    := X // Navigation mutee
[...]
```

Listing 3.2: Mutant créé par application de l'opérateur RSCC

3.5 Création de nouveaux modèles de test par application de patterns

Nous tentons dans cette section d'apporter une réponse automatique aux deux questions : « *À quoi devrait ressembler le modèle de sortie pour que le mutant soit tué ? Quelle différence souhaite-t-on y faire apparaître ?* » et « *Comment modifier le modèle de test de manière à produire la différence attendue dans le modèle de sortie, et ainsi tuer le mutant ?* ».

Nous identifions, pour chacun des opérateurs de mutation, un ensemble de configurations spécifiques (*patterns*) dans les modèles traités par la transformation qui peuvent laisser un mutant en vie. Nous associons à chacun des patterns une ou plusieurs modifications automatiques qui, appliquées à ces modèles, doivent tuer un mutant donné. Ces patterns et recommandations sont définis de manière générique pour chacun des opérateurs de mutation : leur application effective dépend du jeu de test de la transformation ainsi que des modèles de mutation qui décrivent la sémantique entre une transformation et l'un de ses mutants.

Les patterns que nous présentons sont basés sur une représentation abstraite des opérateurs de mutation, nous ne pouvons donc pas être sûrs à 100 % que les recommandations associées tueront le mutant visé. Notre approche propose une détection automatique de configurations problématiques connues dans les modèles traités et des modifications automatiques à effectuer sur ces modèles ; dans la plupart des cas, ces modifications tuent le mutant visé.

Par soucis de concision, nous ne présentons ici que les patterns et recommandations identifiés pour l'opérateur RSCC. La description exhaustive des patterns et recommandations pour l'ensemble des opérateurs est disponible en annexe et dans [4].

3.5.1 Exemple : patterns pour l'opérateur RSCC

L'opérateur de mutation RSCC, tel que nous l'avons présenté à la section 3.4.2, remplace la navigation d'une association vers une classe par la navigation d'une autre association vers la même classe. Afin d'illustrer les patterns identifiés pour cet opérateur, nous prenons pour exemple une transformation qui navigue la séquence $self.a.ba.c$ et l'un de ses mutants RSCC qui navigue à la place la séquence $self.a.bb.c$. Les patterns suivants sont identifiés :

Pattern RSCC_P1 La séquence originale et la séquence mutée pointent vers la même instance.

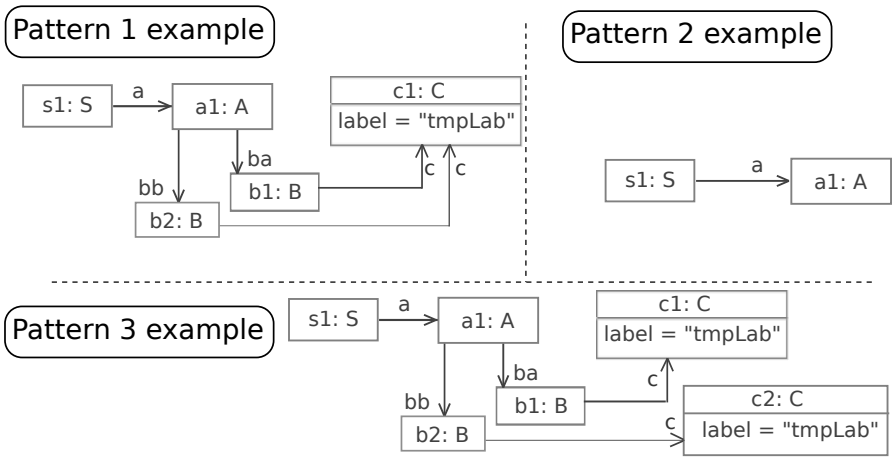


FIGURE 19: Patterns pour l'opérateur RSCC

Pattern RSCC_P2 La séquence originale et la séquence mutée pointent vers *null*.

Pattern RSCC_P3 Les valeurs des propriétés pointées par la séquence originale et la séquence mutée sont les mêmes.

La figure 19 présente trois modèles. Ces modèles exécutent la règle mutée, mais ne tuent pas le mutant : ils correspondent chacun à l'un des patterns identifiés. Nous détaillons ci-après la détection des patterns et les recommandations associées.

Pattern RSCC_P1 Ce pattern a lieu lorsque les instances pointées par les deux séquences de navigation sont les mêmes. Dans le premier modèle présenté par la figure 19, en partant de l'instance $s1 : S$, les deux séquences $self.a.ba.c$ et $self.a.bb.c$ arrivent à la même instance $c1 : C$. De manière à tuer le mutant associé, une nouvelle instance de C doit être créée, ajoutée au modèle, et l'une des deux références $B.c$ doit être mise à jour de manière à pointer vers la nouvelle instance. Ainsi, les deux séquences de navigation $self.a.ba.c$ et $self.a.bb.c$ pointeront vers deux instances différentes, renforçant ainsi les chances de tuer le mutant associé.

Recommandation pour le pattern RSCC_P1 L'idée générale est de créer une nouvelle instance complètement différente de celle pointée par les deux séquences de navigation. Plusieurs cas sont à prendre en compte, en fonction des cardinalités de la dernière référence de la séquence :

- Si au moins l'une des deux références a sa multiplicité supérieure à 1 (c'est une collection), la nouvelle instance créée est ajoutée à cette collection. Ainsi, les collections d'instances retournées par chacune des séquences ne contiennent pas les mêmes éléments et peuvent faire apparaître une différence entre la transformation originale et son mutant,
- Si au moins l'une des deux références a sa multiplicité inférieure ou égale à 0, l'instance pointée peut être supprimée. Ainsi, l'une des deux séquences retournera une instance, l'autre non, créant une différence entre la transformation originale et son mutant.

Pattern RSCC_P2 Ce pattern a lieu quand l'une des références intermédiaires des séquences ne peut être naviguée. Dans le second modèle présenté figure 19, les références $A.bb$ et $A.ba$ ne

peuvent être naviguées : les deux séquences retournent donc *null* et aucune différence n'apparaît entre la transformation originale et son mutant. De manière à tuer le mutant, une nouvelle instance du type attendu doit être créée et l'une des deux références doit être mise à jour de manière à pointer vers cette nouvelle instance.

Recommandation pour le pattern RSCC_P2 L'idée est la même que pour le pattern 1 : une nouvelle instance est créée, et l'une des deux références *A.bb* ou *A.ba* est mise à jour de manière à pointer vers la nouvelle instance. Ainsi, seule l'une des deux transformation (originale ou mutée) renverra une instance, l'autre renvoyant *null*.

Pattern RSCC_P3 Ce pattern a lieu lorsque les instances retournées par les deux séquences de navigation présentent des valeurs communes pour l'un de leurs attributs. Dans le troisième modèle présenté figure 19, les deux séquences de navigation retournent bien deux instances différentes, mais celles-ci présentent toutes deux un attribut *label* ayant pour valeur *tmpLab*. Afin de faire apparaître une différence entre la navigation originale et son mutant, la valeur de cet attribut doit être mise à jour dans l'une des deux instances.

Recommandation pour le pattern RSCC_P3 Cette fois, la valeur d'un des deux attributs doit être changée. Après avoir identifié les éléments concernés dans le modèle (*i.e.* les instances pointées par les deux séquences de navigation), les valeurs de leurs attributs sont changées afin de faire apparaître une différence.

Détection automatique des patterns Pour chacun des éléments identifiés par l'algorithme présenté en section 3.3.3 (les éléments mis en jeu par la règle mutée), les éléments pointés par la navigation des références *initNavigation* et *newNavigation* du modèle de mutation sont collectés. Les instances ainsi récupérées sont ensuite comparées afin de déterminer si elles sont identiques (Pattern RSCC_P1) ou *null* (Pattern RSCC_P2). Si les instances sont différentes, leurs attributs sont comparés deux à deux afin de détécter des valeurs communes (Pattern RSCC_P3). Selon le pattern identifié, la recommandation associée est appliquée au modèle afin de créer un nouveau modèle de test.

3.6 Expérience : la transformation `fsm2ffsm`

Afin de montrer la validité de notre approche, nous l'expérimentons sur la transformation `fsm2ffsm` réalisant la mise à plat d'un automate fini. Cette transformation prend en entrée un automate hiéarchique et produit en sortie un automate fini applati équivalent. Le métamodèle présenté figure 20 est utilisé pour spécifier les modèles d'entrée et de sortie : `fsm2ffsm` est une transformation endogène[4]. La figure 3.21(a) présente un exemple d'automate hiérarchique et la figure 3.21(b) son automate fini applati équivalent.

4. Une transformation endogène est une transformation entre des modèles exprimés dans le même langage, à l'inverse d'une transformation exogène [55]

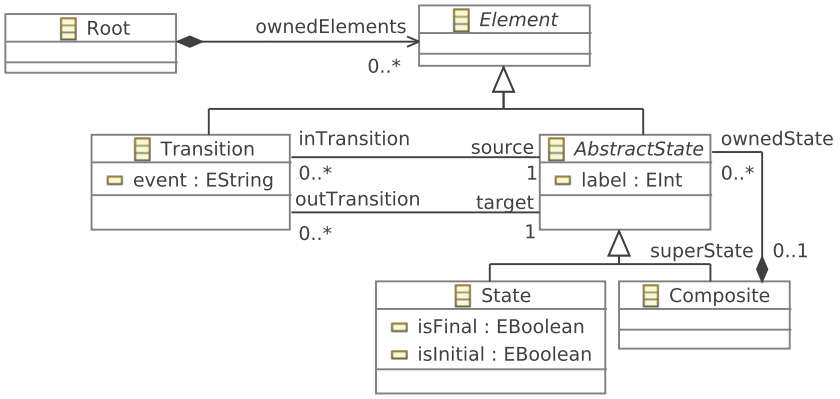
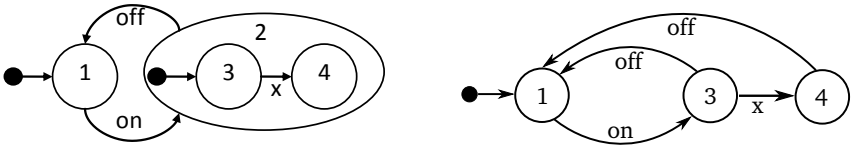


FIGURE 20: Métamodèle d'automate fini



(a) Exemple d'automate hiérarchique (b) Automate fini applati équivalent

FIGURE 21: Un automate hiérarchique avant et après mise à plat

La transformation `fsm2ffsm` est écrite en Kermeta 2 et comprend 169 lignes réparties en huit opérations où les appels de trace sont insérés manuellement. Deux préconditions doivent être respectées par les modèles d'entrée : chaque composite doit contenir un et un seul état initial et chaque état de l'automate doit avoir un label unique. D'autres contraintes sont exprimées en tant qu'invariants sur le métamodèle d'entrée ; par exemple, un composite ne peut se contenir lui-même.

Nous ne présentons ici que les grandes lignes de l'expérience : génération des mutants, du jeu de test initial et résultats obtenus. Un compte-rendu exhaustif des itérations successives de l'algorithme d'amélioration du jeu de test est disponible en annexe et dans [4].

3.6.1 Création des mutants

En utilisant la transformation de génération des modèles de mutation (cf. section 3.8.3), 148 modèles de mutation sont obtenus pour le métamodèle de la figure 20. Un exemple de modèle de mutation généré pour l'opérateur RSCC est présenté figure 22. Il spécifie qu'à chaque fois que la transformation originale navigue la référence *Transition.target*, un mutant est créé où la référence naviguée à la place est *Transition.source*. Nous n'avons pas développé d'outil pour la génération automatique des mutants, ainsi nous les créons manuellement en appliquant les différents modèles de mutation générés. Au final, nous obtenons 126 mutants ; le tableau 4 montre leur répartition par type d'opérateur de mutation.

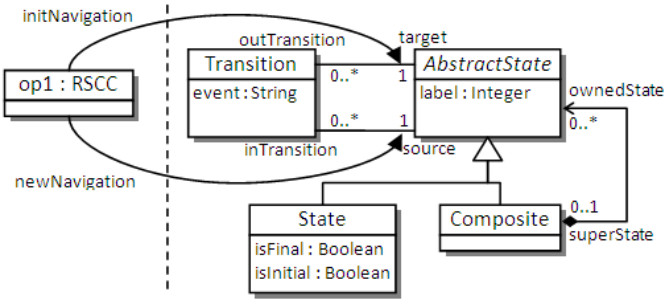


FIGURE 22: Exemple de modèle de mutation pour l'opérateur RSCC sur le métamodèle de la figure 20

CATÉGORIE	OPÉRATEUR	NOMBRE
Navigation	*RSCC* - Relation to the Same Class Change	16
	RSMA - Relation Sequence Modification with Addition	34
	RSMD - Relation Sequence Modification with Deletion	8
Filtrage	*CFCP* - Collection Filtering Change with Perturbation	31
	CFCD - Collection Filtering Change with Deletion	14
	CFCA - Collection Filtering Change with Addition	9
Création	*CACD* - Classes Association Creation Deletion	7
	CACA - Classes Association Creation Addition	7
	CCCR - Class Compatible Creation Replacement	–
Total		126

TABLEAU 4: Répartition des mutants créés pour la transformation `fsm2ffsm`

3.6.2 Jeu de test initial

Afin de pouvoir dérouler l'algorithme d'amélioration du jeu de test, un jeu de test initial doit être généré. Dans [32], les auteurs utilisent des critères de couverture du métamodèle d'entrée de la transformation pour qualifier un jeu de test. En particulier, le critère *AllPartitions* spécifie que la partition complète de chacune des propriétés du métamodèle d'entrée doit être couverte par au moins un modèle de test. En utilisant cette méthode, nous créons neuf modèles de test $M_1 ... M_9$. Le tableau 5 détaille leurs tailles respectives, calculées en fonction du nombre d'éléments, attributs et associations qu'ils contiennent.

NOM	$M1$	$M2$	$M3$	$M4$	$M5$	$M6$	$M7$	$M8$	$M9$	TOTAL
TAILLE	14	31	31	14	16	12	10	10	25	163

TABLEAU 5: Taille des modèles du jeu de test initial

3.6.3 Résultats et analyse

L'application de notre approche sur la transformation `fsm2ffsm` nous a permis de passer d'un score de mutation initial de 45 % à un score final de 100 % en 8 itérations successives. À chaque itération, un mutant en vie est étudié et un nouveau modèle de test est généré pour le tuer. Notre algorithme fournit à chaque étape la liste des modèles pertinents ainsi que les éléments

Iteration	Nombre de modèles	Score de mutation (%)	Mutants en vie	Taille du jeu de test	Mutant étudié	Taille des modèles étudiés	Tailles des éléments étudiés	Gain (%)	Pattern détecté	Modèle créé
0	9	45, 24	69	163	$CFCD\_4$	39	11	93, 25	$CFCD\_P2$	$N1$
1	10	48, 41	65	180	$CACA\_4$	90	36	80	–[5]	$N2$
2	11	67, 46	41	205	$CFCA\_10$	25	5	97, 56	$CFCA\_P2$	$N3$
3	12	69, 84	38	235	$CFCA\_8$	55	10	95, 74	–	$N4$
4	13	70, 63	37	260	$CFCD\_8$	80	15	94, 23	$CFCD\_P2$	$N5$
5	14	75, 40	31	288	$CACA\_6$	198	56	80, 56	–[5]	$N6$
6	15	91, 27	11	313	$CACD\_5$	223	60	80, 83	$CACD\_P2$	$N7$
7	16	99, 21	1	338	$CFCA\_7$	248	68	79, 88	$CFCA\_P1$	$N8$
Résultat	17	100	0	368	–	–	–	–	–	–

Tableau 6: Résultat final du processus d'amélioration du jeu de test sur la transformation `fsm2ffsm`

au sein de ces modèles ayant déclenché la règle mutée. La création des nouveaux modèles se fait par identification de patterns et application des recommandations associées. Dans la plupart des cas, le nouveau modèle créé tue d'autres mutants par effet de bord. Le tableau 6 présente un résumé du processus global d'amélioration du jeu de test. Nous tirons les conclusions suivantes de cette expérience :

– L'utilisation de la traçabilité et des modèles de trace locale permet de réduire considérablement le champ de recherche du testeur pour trouver les modèles et les éléments de ces modèles les plus pertinents pour la création de nouveaux modèles de test. Alors qu'un testeur doit potentiellement analyser l'ensemble des éléments des modèles du jeu de test, ici seuls 33 % des modèles et 27 % des éléments qu'ils contiennent ont dû être analysés. Le gain moyen en terme d'éléments à couvrir par le testeur est de 87 %.
– Les patterns identifiés pour chacun des opérateurs permettent de fournir au testeur des modifications simples à appliquer aux modèles de manière à tuer le mutant visé. Dans notre expérience, cinq des huit mutants visés ont été tués en appliquant simplement la recommandation associée. Les modèles de trace locale ont indiqués pour deux des mutants visés que leur instruction mutée n'était pas exécutée. Finalement, seul un des huit mutants étudiés a nécessité une analyse en profondeur.

5. Dans ces cas-là, aucun pattern n'est détecté mais les modèles de trace locale nous indiquent que l'instruction mutée n'a pas été exécutée. Ainsi, le testeur sait qu'il doit produire un nouveau modèle de test couvrant les préconditions de la règle mutée.

3.7 Conclusion

L'analyse de mutation est une technique efficace pour qualifier un jeu de test, mais elle souffre toujours d'un manque d'automatisation, notamment pour la phase d'amélioration du jeu de test. La méthode que nous venons de présenter constitue une étape importante vers une automatisation complète du processus d'analyse de mutation pour les transformations de modèles.

La notation de traçabilité a déjà été utilisée dans un article précédent [6] pour l'identification des éléments pertinents au sein des modèles de test. Cependant, ce travail ne s'intéressait pas à l'analyse des modèles de sortie. L'identification de configurations connues pouvant laisser un mutant en vie et la définition de modifications simples à appliquer à ces modèles dans de tels cas permet d'obtenir un meilleur niveau d'automatisation du processus de création de nouveaux modèles de test destinés à tuer des mutants.

L'approche retenue reste totalement indépendante de la transformation sous test, du langage de transformation de modèles et des métamodèles manipulés par la transformation. Cette indépendance résulte de l'utilisation du mécanisme de traçabilité de [6] et de la modélisation des opérateurs de mutation de [56].

L'expérience retenue pour évaluer notre méthode montre que l'algorithme présenté permet de réduire considérablement le nombre d'éléments à analyser par le testeur pour la production de nouveaux modèles de test. De plus, dans la plupart des cas, l'application des recommandations associées aux patterns détectés de manière automatique dans les modèles permet de tuer les mutants en vie. Ces résultats montrent combien le travail du testeur peut être facilité.

Nous ne pouvons actuellement pas appliquer à chaque fois et de manière automatique une modification aux modèles d'entrée pour tuer de nouveaux mutants. En effet, les contraintes qui s'appliquent aux modèles de test d'une transformation doivent être prises en compte lors de l'application des recommandations. L'enjeu de la génération automatique de modèles de test s'applique à notre méthode de même qu'aux autres travaux effectués dans ce domaine. L'effort doit être donc être poursuivi pour arriver à une automatisation complète de la génération de modèles de test.

3.8 Travail réalisé

L'approche décrite au chapitre 3 ainsi que son expérimentation sur la transformation `fsm2ffsm` ont mené à l'écriture d'un article de 30 pages « *Towards an Automation of the Mutation Analysis Dedicated to Model Transformation* » [5], actuellement en cours de revue pour publication dans le journal *Software Testing, Verification and Reliability*.

Dans ce cadre, j'ai réalisé la modélisation des 10 opérateurs de mutation de [56] et écrit différents patterns et recommandations associés. J'ai également réalisé un incrément sur le métamodèle de trace locale de [6] en lui permettant de tracer de manière efficace les attributs et références au sein des modèles manipulés par une transformation. J'ai ensuite intégré l'ensemble de ces concepts au sein d'une plateforme expérimentale réalisant de manière automatique le processus

d'analyse de mutation des transformations de modèles. Différents outils viennent compléter cette plateforme, notamment une transformation générant à partir des métamodèles manipulés par une transformation l'ensemble des modèles de mutation associés et un mécanisme de traçabilité pour Kermeta permettant la création automatique d'un modèle de trace locale associé à chaque exécution.

J'ai implémenté la transformation `fsm2ffsm` en utilisant les outils susnommés, ainsi que réalisé la collecte et l'exploitation des résultats obtenus. La transformation `fsm2ffsm`, le mécanisme de trace pour Kermeta ainsi que la transformation générant les modèles de mutation d'une transformation ont aussi été rendus publics dans [4] et joints à la soumission au journal STVR.

Enfin, j'ai rédigé une annexe de 33 pages décrivant en détail la modélisation des opérateurs de mutation ainsi que leurs contraintes, patterns et recommandations, la transformation générant les modèles de mutation et l'expérimentation complète sur la transformation `fsm2ffsm`. Les sections suivantes présentent plus précisément le travail effectué.

3.8.1 L'expérience `fsm2ffsm`

Un cas d'étude avait déjà été réalisé pour illustrer l'approche proposée sur une transformation classique réalisant la transformation depuis un diagramme de classes vers un schéma de base de données relationnelle. Il manquait à ce cas d'étude une description étape par étape de l'application de l'algorithme ainsi que des données quantitatives. Afin de pouvoir proposer une expérience complète, j'ai implémenté la transformation `fsm2ffsm`, écrit l'ensemble de ses mutants par application des opérateurs de mutation de [56] et déroulé l'algorithme d'amélioration du jeu de test afin d'obtenir un score de mutation de 100 %. La description étape par étape des itérations effectuées est retranscrite dans [4].

3.8.2 Plateforme expérimentale

L'analyse de mutation nécessite l'exécution de tous les mutants sur tous les modèles du jeu de test. Afin de ne pas avoir à réaliser manuellement toutes ses exécutions, et ce à chaque étape du processus d'amélioration du jeu de test, j'ai développé une plateforme expérimentale en Java pour automatiser (i) l'exécution des mutants sur les modèles de test (ii) la collecte des modèles de trace locale produits par ces exécutions (iii) la comparaison des modèles de sortie pour produire un verdict (iv) la construction du modèle de matrice de mutation global agrégeant les modèles de test, les mutants, les modèles de mutation et les verdicts associés et (v) la présentation des résultats (matrice et score de mutation) et des informations pertinentes tirées de la trace locale (éléments en entrée/sortie, règle mutée, etc.).

La plateforme expérimentale se veut générique, en ce sens qu'elle ne dépend pas d'une transformation en particulier et peut être facilement adaptée à n'importe quelle transformation, qu'elle soit écrite en Kermeta ou non. La comparaison des modèles de sortie est réalisée à l'aide de l'API *EMFCompare* [26], mais peut être facilement remplacée par un oracle quelconque vérifiant telle

ou telle propriété sur ces modèles. La construction et le parcours de la matrice de mutation sont réalisés à l'aide des différentes APIs proposées par la plateforme EMF.

La principale limite de la plateforme expérimentale est qu'elle ne suppose qu'un et un seul modèle en entrée, et un et un seul modèle en sortie. Là encore, les modifications nécessaires pour accepter un nombre quelconque de modèles peuvent être effectuées en un temps raisonnable.

3.8.3 Modélisation des opérateurs de mutation

En étroite collaboration avec Jean-Marie Mottu, j'ai réalisé la modélisation des opérateurs de mutation. Une première version de ces métamodèles avait déjà été réalisée par Vincent Aranega mais certains ne correspondaient pas aux descriptions précises des opérateurs telles que nous les avons retenues pour l'article. De plus, j'ai écrit l'ensemble des contraintes OCL qui leur sont associées afin de soulever les ambiguïtés liées à leur description graphique. Les différents métamodèles finalement produits sont définis en relation avec le méta-métamodèle EMOF. Il est ainsi possible d'écrire des modèles de mutation (conformes aux métamodèles de mutation) pour n'importe quel métamodèle conforme à EMOF afin de préserver l'indépendance de notre approche vis-à-vis des métamodèles utilisés par la transformation sous test.

Afin de produire automatiquement les modèles de mutation, j'ai écrit une transformation Kermeta prenant en entrée les métamodèles d'entrée et de sortie d'une transformation et produisant en sortie l'ensemble des modèles de mutation potentiellement applicables à cette transformation. Cette transformation se base sur les métamodèles de mutation décrits dans l'annexe [4] et sur les contraintes OCL associées.

Les modèles de mutation générés ne mènent pas nécessairement à la création d'un mutant. En effet, ils sont basés exclusivement sur les métamodèles d'entrée et de sortie, et ne prennent pas en compte la transformation en elle-même afin de rester indépendant d'un langage de transformation particulier. Puisqu'une transformation peut n'utiliser qu'une partie des métamodèles qu'elle manipule, ne naviguer qu'un sous-ensemble des références, ne filtrer qu'une partie des collections et ne créer que certains types d'éléments, il existe des modèles de mutation pour lesquels aucun mutant ne sera créé. A l'inverse, tout mutant créé sur une transformation est associé à l'un des modèles de mutation généré, et un même modèle de mutation peut mener à la création de plusieurs mutants si la transformation effectue plusieurs fois le même type de traitement.

3.8.4 Mécanisme de traçabilité pour Kermeta

Afin de permettre l'exploitation des informations de la trace, il est nécessaire de générer automatiquement des modèles de trace locale pour chaque exécution d'un couple $(modèle\ de\ test, mutant)$, tel que décrit en section 3.3. Bien que le métamodèle de trace locale soit utilisable indépendamment d'un langage de transformation particulier, la génération de tels modèles est elle spécifique à un langage. Dans notre cas, la création des modèles de trace locale est réalisée par l'insertion d'appels de trace directement dans le code de la transformation sous test. Concrètement, j'ai écrit une classe Kermeta *Tracer*, indépendante d'une transformation en particulier,

permettant la création de liens de trace depuis une transformation par l'appel de l'opération `Tracer.trace(<rule>, <srcElements>, <dstElements>)`. Cette opération peut prendre en argument un nombre quelconque d'éléments en entrée ou en sortie, ces éléments pouvant correspondre à des instances de classes, des attributs ou des références.

Au cours d'une transformation, les différents liens de trace sont construits et le modèle de trace locale ainsi défini est sérialisé à la fin de l'exécution. Le modèle de trace locale est ensuite récupéré par la plateforme expérimentale qui le lie à un modèle de test et un mutant particulier. Ensuite, les liens de trace qu'il contient sont exploités dans le cadre de l'algorithme proposé pour l'amélioration du jeu de test.

En outre, l'expérience `fsm2ffsm` a permis de soulever un certain nombre d'interrogations sur le mécanisme de traçabilité proposé dans [6], notamment concernant la traçabilité des attributs et des références (cf. section 3.3.1). Ces interrogations ont été transcrites dans un document de travail échangé avec les principaux auteurs. Les modifications que j'ai proposées ont été intégrées au métamodèle utilisé pour l'expérience afin de permettre le bon déroulement de l'algorithme d'amélioration du jeu de test. Enfin, j'ai mis en relation les modèles de mutation générés de manière automatique avec les modèles de matrice de mutation afin de pouvoir exploiter la différence sémantique qu'ils portent.

3.8.5 Rédaction

L'essentiel de mon travail de rédaction a consisté en l'écriture d'une annexe de 33 pages [4] décrivant les points suivants :

Modélisation des opérateurs de mutation Pour chacune des catégories d'opérateurs de mutation décrits dans [56], le métamodèle associé ainsi que les contraintes OCL qui s'y rapportent sont décrits en profondeur. Les différents opérateurs sont en outre illustrés par un exemple de modèle de mutation et un exemple d'instanciation sur le pseudo-code d'une transformation.

Patterns et recommandations Pour chacun des opérateurs, les patterns identifiés et les recommandations associées sont présentés. Certains patterns avaient déjà été identifiés par Vincent Aranega dans d'autres documents non publiés. Pour chacun des opérateurs, j'ai réétudié les configurations possibles et écrit de nouveaux patterns ainsi que leurs recommandations. Le document sur lequel je me suis basé présentait 11 patterns, la version finale de l'annexe en présente 23.

Génération automatique des modèles de mutation La deuxième section de l'annexe présente succintement le principe de la transformation Kermeta générant les modèles de mutation.

Expérience `fsm2ffsm` Enfin, la dernière section détaille étape par étape l'implémentation de le transformation `fsm2ffsm`, la génération d'un jeu de test initial, la création des mutants et le déroulement de l'algorithme d'amélioration du jeu de test pour passer d'un score de mutation initial de 45 % à un score final de 100 %. À chaque étape, une description

précise des éléments retournés par l'algorithme ainsi que les modifications effectuées sur les modèles d'entrée sont détaillées.

Dans le papier en lui-même, j'ai rédigé un paragraphe de l'état de l'art présentant mes dernières lectures sur la production automatique de données de test en exploitant l'analyse de mutation. J'ai également pris en compte les remarques manuscrites effectuées par Benoit Baudry lors de sa relecture des premières sections. Enfin, j'ai effectué une relecture complète de l'article avant sa publication et réalisé la correction d'erreurs. En parallèle, j'ai été sollicité pour apporter mon avis sur les réponses à donner à certaines questions soulevées par les correcteurs lors de leur première revue.

Chapitre 4

Perspectives et travaux futurs

L'approche présentée au chapitre 3 constitue une réelle contribution pour la génération semi-automatique de modèles de test adéquats à la mutation pour les transformations de modèles. Elle repose directement sur les spécificités des données manipulées par les transformations de modèles en associant des recommandations simples à appliquer lorsque certaines configurations particulières sont détectées dans les modèles d'entrée. L'expérience menée pour valider cette approche montre qu'elle est efficace dans la plupart des cas. Cependant, elle recquiert d'une part l'existence d'un jeu de test initial et d'autre part la mise en place d'un mécanisme de traçabilité sur le langage de transformation utilisé.

Les transformations de modèles étant des programmes classiques avec certaines spécificités sur les données manipulées et les opérations effectuées, nous pensons que certains travaux effectués sur la génération automatique de données de test adéquats à la mutation pour les programmes classiques peuvent être adaptés aux transformations de modèles.

En particulier, nous pensons que l'approche par résolution de contraintes telle qu'introduite en section 2.3.4 peut bénéficier aux transformations de modèles. Certains travaux exploitent déjà la génération par contraintes de modèles de test. Dans [78], Sen *et al.* utilisent Alloy pour générer de manière automatique des modèles de test satisfaisant les critères de partitionnement du métamodèle d'entrée des transformations tels que décrits dans [32]. Alloy est un langage de spécification déclaratif reposant sur la logique du premier ordre et permettant d'exprimer des contraintes sur des modèles. Il peut utiliser différents solveurs SAT qui, à partir d'une spécification Alloy, génèrent de manière automatique des modèles satisfaisant les contraintes de la spécification. Les auteurs ont développé un outil, Pramana [77], dont le but est d'agréger les différentes contraintes exprimées en différents formalismes que doit respecter un modèle de test afin de produire une spécification Alloy. Pramana permet de plus de retranscrire les modèles produits par les solveurs utilisés par Alloy en des modèles directement exploitables par le framework EMF d'Eclipse. Nous rappelons ci-dessous les différentes contraintes que doit respecter un modèle de test d'une transformation de modèle :

– Il doit être conforme au métamodèle d'entrée de la transformation
– Il doit respecter les invariants exprimés sur ce métamodèle

– Il doit respecter les préconditions de la transformation sous test
– Il doit correspondre à un objectif, une intention de test

Dans [78], l'intention de test est la couverture du métamodèle d'entrée de la transformation proposée dans [32]. Des fragments de modèles sont produits pour exprimer cette intention de test. Par exemple, un fragment de modèle peut exprimer la contrainte : « *Le modèle de test doit couvrir l'ensemble des partitions de la propriété X* ». Cette contrainte est traduite en un prédicat ALLOY intégré à la spécification. Puisque la spécification ALLOY contient toutes les contraintes exprimées ci-dessus, le solveur est en mesure de générer des modèles satisfaisant l'intention de test ainsi que les invariants et préconditions de la transformation et du métamodèle d'entrée. La figure 23 résume le processus de génération de modèles de test.

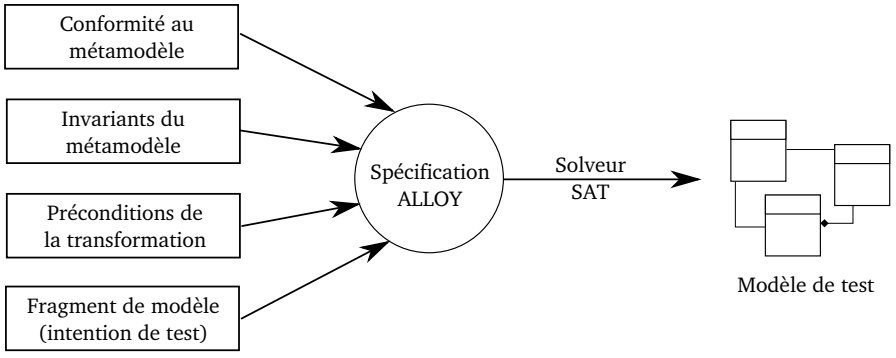


FIGURE 23: Utilisation d'ALLOY pour la génération de modèles de test

Nous travaillons actuellement sur une adaptation de ces concepts afin de générer des fragments de modèles dont l'intention de test est de tuer des mutants. Chaque fragment de modèle que nous souhaitons exprimer doit amener à la création d'un modèle de test destiné à tuer un mutant en particulier. Nous avons vu en section 2.3.4 que, pour tuer un mutant, une donnée de test doit respecter les contraintes d'atteignabilité, de nécessité et de suffisance établies par DeMillo dans [23]. Nous avons aussi vu que des problèmes se posent avec la condition de suffisance : nous nous limitons donc à l'écriture de contraintes pour tuer *faiblement* les mutants. Ainsi, les fragments de modèles que nous souhaitons créer expriment l'intention de test suivante : « *Le modèle de test doit satisfaire les conditions d'atteignabilité et de nécessité pour tuer faiblement le mutant X* ».

La condition d'atteignabilité se résume à un problème de satisfaction de chemins. Il est possible de générer le graphe de flot de contrôle d'une transformation de modèles et d'établir les contraintes à satisfaire pour qu'un modèle de test atteigne l'instruction mutée d'un mutant. Toutefois, les techniques pour générer de telles contraintes sont dépendantes du langage de transformation utilisé. De plus, un programme de transformation de modèles peut prendre en entrée des données provenant de diverses sources : modèles bien sûr, mais aussi fichiers de configuration ou arguments arbitraires. Si des données provenant d'autres sources que les modèles d'entrée sont utilisées dans les contraintes de chemin, cette approche ne peut aboutir. Une issue pragmatique à ce problème serait de réutiliser le mécanisme de traçabilité du chapitre 3 afin de trouver directement des modèles exécutant l'instruction mutée.

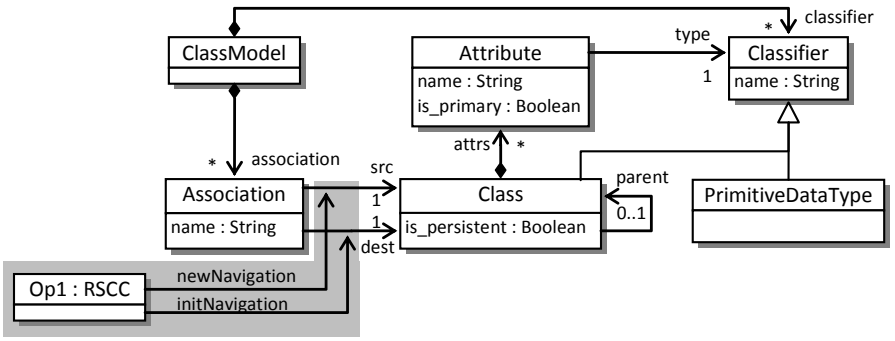


FIGURE 24: Exemple d'application de l'opérateur RSCC sur la transformation `class2rdbms`

La condition de nécessité stipule que les états respectifs de la transformation originale et de la transformation mutée juste après l'instruction mutée doivent différer. Nous avons vu que pour des programmes classiques, cette condition peut être simplement reformulée sous la forme (*instruction mutée* $\neq$ *instruction originale*). Nous proposons d'adapter ces conditions aux différents opérateurs de mutation de [56]. Ainsi, à partir du modèle de mutation d'un mutant quelconque, il est possible d'écrire une contrainte de nécessité pour ce mutant. Par exemple, pour tuer le mutant dont le modèle de mutation est présenté figure 24, un modèle de test doit respecter la condition suivante : « *Il existe une instance d'Association X telle que X.src pointe vers une instance de Class différente de celle pointée par X.dest* ». En ALLOY, cette contrainte peut être écrite sous la forme d'un prédicat de la façon suivante :

```
pred kill_RSCC_op1 {
  some a : Association | a.src != a.dest
}
```

LISTING 4.1: Prédicat ALLOY exprimant la contrainte de nécessité du mutant Op1

La combinaison des conditions d'atteignabilité et de nécessité pour chacun des mutants, exprimée en Alloy, permet de générer des modèles dont l'intention de test directe est de tuer des mutants. La génération de telles contraintes pourrait être à terme directement intégrée à l'outil PRAMANA. Bien sûr, nos travaux ne sont qu'à leurs débuts, mais nous espérons qu'ils poursuivront le chemin vers la génération automatique de modèles de test adéquats à la mutation.

Chapitre 5

Conclusion

En permettant la génération automatique de tout ou partie d'une application à partir de modèles de haut niveau, l'ingénierie dirigée par les modèles facilite le travail des équipes de développement en facilitant la prise en compte du changement et la variabilité des environnements d'exécution.

L'automatisation d'un tel processus repose sur les transformations de modèles qui réalisent les opérations de composition, de raffinement, de réusinage, etc.. Celles-ci étant destinées à être utilisées en boîte noire par les ingénieurs et développeurs, elles doivent être minutieusement testées afin d'éviter la propagation de modèles incorrects.

L'analyse de mutation a prouvé son efficacité pour qualifier les jeux de test des programmes. L'utilisation du critère de mutation pour guider la génération de données de test permet de produire des jeux de test efficaces, renforçant ainsi la confiance accordée au programme sous test. Néanmoins, l'analyse de mutation souffre toujours d'un trop fort coût en ressources et d'un manque d'automatisation.

En augmentant le niveau d'automatisation de l'analyse de mutation pour les transformations de modèles, l'approche décrite ici permet de soulager le travail du testeur en lui fournissant toutes les informations nécessaires à la génération de nouveaux modèles de test, et en générant de manière automatique ces nouveaux modèles lorsque cela est possible. De plus, notre approche reste totalement indépendante d'un langage de transformation de modèles, d'une transformation de modèles ou d'un métamodèle en particulier. Ainsi, elle peut être appliquée quelque soit la transformation sous test ou le langage utilisé.

Bien sûr, la création de modèles de test ne se suffit pas à elle-même et des méthodes doivent être développées pour générer de manière automatique les oracles associés. Les travaux dans ce domaine restent pour le moment limité, et le chemin à parcourir est encore long. Néanmoins, notre approche constitue une étape importante pour le test des transformations de modèles et l'adoption de l'ingénierie dirigée par les modèles.

Liste des figures

Liste des tableaux

Bibliographie

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[2] Anneliese Andrews, Robert France, Sudipto Ghosh, and Gerald Craig. Test Adequacy Criteria for UML Design Models. *Software Testing, Verification and Reliability*, 13(2) :95–127, 2003.

[3] Vincent Aranega, Anne Etien, and Jean-Luc Dekeyser. Using an Alternative Trace for QVT. In *Workshop on Multi-Paradigm Modeling*, volume 42, Olso, Norway, 2010.

[4] Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry, and Jean-Luc Dekeyser. Annexe and Experimentation Material. https://sites.google.com/site/mutationtesttransfo/, 2013.

[5] Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry, and Jean-Luc Dekeyser. Towards an Automation of the Mutation Analysis Dedicated to Model Tranformation. 2013.

[6] Vincent Aranega, Jean-Marie Mottu, Anne Etien, and Jean-Luc Dekeyser. Traceability for Mutation Analysis in Model Transformation. In *Proceedings of the 2010 international conference on Models in software engineering*, MODELS'10, 2011.

[7] Kamel Ayari, Salah Bouktif, and Giuliano Antoniol. Automatic mutation test input data generation via ant colony. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1074–1081. ACM, 2007.

[8] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to Systematic Model Transformation Testing. *Communications of the ACM*, 2009.

[9] Boris Beizer. *Software Testing Techniques*. Dreamtech Press, 2002.

[10] Jean Bézivin. From Object Composition to Model Transformation with the MDA. In *Proceedings of TOOLS*, volume 1, pages 350–354, 2001.

[11] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *Novatica Journal, Special Issue*, 5(2) :21–24, 2004.

[12] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. Reflective Model Driven Engineering. In *«UML» 2003-The Unified Modeling Language. Modeling Languages and Applications*, pages 175–189. Springer, 2003.

[13] Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt. Model Transformations in Practice Workshop. In *Satellite Events at the MoDELS 2005 Conference*, pages 120–127. Springer, 2006.

[14] Grady Booch. *Object Oriented Analysis & Design with Application*. Pearson Education India, 2006.

[15] Leonardo Bottaci. A genetic algorithm fitness function for mutation testing. In *Proceedings of the SEMINALL-workshop at the 23rd international conference on software engineering, Toronto, Canada*, 2001.

[16] Timothy A Budd and Dana Angluin. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica*, 18(1) :31–45, 1982.

[17] Timothy A Budd and Ajei S Gopal. Program Testing by Specification Mutation. *Computer languages*, 10(1) :63–73, 1985.

[18] Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *Software Engineering, IEEE Transactions on*, (3) :215–222, 1976.

[19] Lori A Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *Software Engineering, IEEE Transactions on*, 15(11) :1318–1332, 1989.

[20] Murial Daran and Pascale Thévenod-Fosse. Software Error Analysis : a Real Case Study Involving Real Faults and Mutations. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 158–171. ACM, 1996.

[21] Marcio Eduardo Delamaro, JC Maidonado, and Aditya P. Mathur. Interface Mutation : an Approach for Integration Testing. *Software Engineering, IEEE Transactions on*, 27(3) :228–247, 2001.

[22] R A DeMillo, R J Lipton, and F G Sayward. Hints on Test Data Selection : Help for the Practicing Programmer. *Computer*, 11(4), 1978.

[23] Richard A Demillo. Constraint-Based Automatic Test Data Generation. (September 1991) :1–18, 1997.

[24] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on Test Data Selection : Help for the Practicing Programmer. *Computer*, 11(4) :34–41, 1978.

[25] Richard A DeMillo and A Jefferson Offutt. Experimental Results from an Automatic Test Case Generator. *ACM Trans. Softw. Eng. Methodol.*, 1993.

[26] Eclipse. EMFCompare. http://www.eclipse.org/emft/projects/compare, 2013.

[27] Jon Edvardsson. A Survey on Automatic Test Data Generation. (x), 1999.

[28] Sandra Camargo Pinto Ferraz Fabbri, José Carlos Maldonado, Tatiana Sugeta, and Paulo Cesar Masiero. Mutation Testing Applied to Validate Specifications Based on Statecharts. In *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, pages 210–219. IEEE, 1999.

[29] Jean-Rémy Falleri, Marianne Huchard, and Clémentine Nebut. Towards a Traceability Framework for Model Transformations in Kermeta. In *ECMDA-TW Workshop*, 2006.

[30] Jean-Marie Favre. Towards a Basic Theory to Model Driven Engineering. In *3rd Workshop in Software Model Engineering, WiSME*. Citeseer, 2004.

[31] Fabiano Cutigi Ferrari, José Carlos Maldonado, and Awais Rashid. Mutation Testing for Aspect-Oriented Programs. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 52–61. IEEE, 2008.

[32] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, Yves Le Traon, and Yves Le Traon. Qualifying Input Test Data for Model Transformations. *{SoSyM} Journal*, 8(2) :185–203, November 2007.

[33] Franck Fleurey, Jim Steel, and Benoit Baudry. Validation in Model-Driven Engineering : Testing Model Transformations. In *Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on*, pages 29–40. IEEE, 2004.

[34] Phyllis G Frankl, Stewart N Weiss, and Cang Hu. All-Uses vs Mutation Testing : an Experimental Comparison of Effectiveness. *Journal of Systems and Software*, 38(3) :235–253, 1997.

[35] Piero Fraternali and Massimo Tisi. Mutation Analysis for Model Transformations in ATL. *Model Transformation with ATL*, page 145, 2009.

[36] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. An Introduction to UML Profiles. *UML and Model Engineering*, 2, 2004.

[37] Anne Geraci, Freny Katki, Louise McMonegal, Bennett Meyer, John Lane, Paul Wilson, Jane Radatz, Mary Yee, Hugh Porteous, and Fredrick Springsteel. IEEE Standard Computer Dictionary : Compilation of IEEE Standard Computer Glossaries. 1991.

[38] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART : Directed Automated Random Testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[39] Esther Guerra. Specification-Driven Test Generation for Model Transformations. In *Theory and Practice of Model Transformations*, pages 40–55. Springer, 2012.

[40] Richard G. Hamlet. Testing Programs with the Aid of a Compiler. *Software Engineering, IEEE Transactions on*, (4) :279–290, 1977.

[41] William E. Howden. Weak Mutation Testing and Completeness of Test Sets. *Software Engineering, IEEE Transactions on*, (4) :371–379, 1982.

[42] Ivar Jacobson. *Object-Oriented Software Engineering : a Use Case Driven Approach*. Pearson Education India, 1992.

[43] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model Driven Language Engineering with Kermeta. In *Generative and Transformational Techniques in Software Engineering III*, pages 201–221. Springer, 2011.

[44] Y Jia and M Harman. An Analysis and Survey of the Development of Mutation Testing. *Software Engineering, IEEE Transactions on*, 37(5) :649–678, 2010.

[45] Yue Jia and Mark Harman. Higher Order Mutation Testing. *Information and Software Technology*, 51(10) :1379–1393, 2009.

[46] Frédéric Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*, volume 91. Citeseer, 2005.

[47] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL : a QVT-Like Transformation Language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006.

[48] James C King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7) :385–394, 1976.

[49] Kim N King and A Jefferson Offutt. A Fortran Language System for Mutation-Based Software Testing. *Software : Practice and Experience*, 21(7) :685–718, 1991.

[50] Jian Bing Li and James Miller. Testing the Semantics of W3C XML Schema. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, volume 1, pages 443–448. IEEE, 2005.

[51] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An Experimental Comparison of Four Unit Test Criteria : Mutation, Edge-Pair, All-Uses and Prime Path Coverage. In *ICSTW '09 : Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, 2009.

[52] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-Class Mutation Operators for Java. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 352–363. IEEE, 2002.

[53] Phil McMinn. Search-Based Software Test Data Generation : a Survey. *Software Testing, Verification and Reliability*, 14(2) :105–156, 2004.

[54] Stephen J Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. Model-Driven Architecture. In *Advances in Object-Oriented Information Systems*, pages 290–297. Springer, 2002.

[55] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152 :125–142, 2006.

[56] Jean-marie Mottu, Benoit Baudry, and Yves Le˜Traon. Mutation Analysis Testing for Model Transformations. In *ECMDA 06*, Spain, 2006.

[57] Jean-marie Mottu, Sagar Sen, Massimo Tisi, and Jordi Cabot. Static Analysis of Model Transformations for Effective Test Generation. 2012.

[58] A Jefferson Offutt. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1) :5–20, 1992.

[59] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. An Experimental Determination of Sufficient Mutant Operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2), 1996.

[60] A. Jefferson Offutt and Stephen D. Lee. An Empirical Evaluation of Weak Mutation. *Software Engineering, IEEE Transactions on*, 20(5) :337–344, 1994.

[61] A Jefferson Offutt and Jie Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software Testing, Verification and Reliability*, 7(3) :165–192, 1997.

[62] A. Jefferson Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An Experimental Evaluation of Data Flow and Mutation Testing. *Software Practice and Experience*, 26(2) :165–176, 1996.

[63] AJ Offutt. An Integrated Automatic Test Data Generation System. 1(3) :1–26, 1996.

[64] AJ Offutt, Zhenyi Jin, and Jie Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. *Software- . . .*, pages 1–29, 1999.

[65] OMG. Meta Object Facility Core 1.4 (MOF) Specification. http://www.omg.org/spec/MOF/1.4, 2002.

[66] OMG. OMG MOF 2.0 Query / Views / Transformations RFP. http://www.omg.org/cgi-bin/doc?ad/2002-4-10, 2002.

[67] OMG. UML 2.0 Superstructure Specification. http://http://www.omg.org/cgi-bin/doc?formal/05-07-04, 2002.

[68] OMG. Meta Object Facility Core 2.4.1 (MOF) Specification. http://www.omg.org/spec/MOF/2.4.1, 2004.

[69] OMG. UML Profile for MARTE : Modeling and Analysis of Real-time Embedded Systems. http://www.omg.org/spec/MARTE/, 2011.

[70] OMG. Model-Driven Architecture Specifications. http://www.omg.org/mda/specs.htm, 2012.

[71] Thomas J. Ostrand and Marc J. Balcer. The Category-Partition Method for Specifying and Generating Fuctional Tests. *Communications of the ACM*, 31(6) :676–686, 1988.

[72] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 75–84. IEEE, 2007.

[73] Chittoor V Ramamoorthy, S-BF Ho, and WT Chen. On the Automated Generation of Program Test Data. *Software Engineering, IEEE Transactions on*, (4) :293–300, 1976.

[74] James R Rumbaugh, Michael R Blaha, William Lorensen, Frederick Eddy, and William Premerlani. Object-Oriented Modeling and Design. 1990.

[75] Douglas C Schmidt. Model-Driven Engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2) :25, 2006.

[76] Koushik Sen, Darko Marinov, and Gul Agha. *CUTE : a Concolic Unit Testing Engine for C*, volume 30. ACM, 2005.

[77] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing. In *ICST.*, Norway, 2008.

[78] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic Model Generation Strategies for Model Transformation Testing. In *International Conference on Model Transformation, ICMT09.*, Zurich, Switzerland, 2009.

[79] Shane Sendall and Wojtek Kozaczynski. Model Transformation : The Heart and Soul of Model-Driven Software Development. *Software, IEEE*, 20(5) :42–45, 2003.

[80] Purdue University. Software Engineering Research Center (SERC)., RA DeMillo, and AP Mathur. *On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Production Software*. 1991.

[81] Adenilso da Silva Simão and José Carlos Maldonado. MuDeL : A Language and a System for Describing and Generating Mutants. *Journal of the Brazilian Computer Society*, 8(1) :73–86, 2002.

[82] Ben H Smith and Laurie Williams. On Guiding the Augmentation of an Automated Test Suite via Mutation Analysis. *Empirical Softw. Engg.*, 14(3), 2009.

[83] Ben H Smith and Laurie Williams. Should Software Testers Use Mutation Analysis to Augment a Test Set ? *Journal of Systems and Software*, 82(11) :1819–1832, 2009.

[84] Richard Soley et al. Model-Driven Architecture. *OMG white paper*, 308 :308, 2000.

[85] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In *Model Driven Architecture-Foundations and Applications*, pages 18–33. Springer, 2009.

[86] Didier Vojtisek, Jean-Marc Jézéquel, et al. MTL and Umlaut NG-Engine and Framework for Model Transformation. *ERCIM News 58*, 58, 2004.

[87] Elaine J Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4) :465–470, 1982.

[88] Niklaus Wirth. What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions ? *Communications of the ACM*, 20(11) :822–823, 1977.

[89] Andrés Yie and Dennis Wagelaar. Advanced Traceability for ATL. In *1st International Workshop on Model Transformation with ATL*, pages 78–87, 2009.

[90] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan de Halleux, and Hong Mei. Test Generation via Dynamic Symbolic Execution for mutation testing. *2010 IEEE International Conference on Software Maintenance*, pages 1–10, September 2010.