# Test de Transformation de Modèles : Expression d'Oracles

Jean-Marie Mottu, Benoit Baudry, Yves Le Traon

# Test de Transformation de Modèles : expression d'Oracles

**Jean-Marie Mottu[*], Benoit Baudry[*], Yves Le Traon[**]**

** INRIA / IRISA,*
*Campus Universitaire de Beaulieu*
*35042 Rennes Cedex, France*
*{jmottu, bbaudry}@irisa.fr*

*** IT-TELECOM Bretagne,*
*2 rue de la Châtaigneraie CS 17607,*
*35576 Cesson Sévigné, France*
*yves.letraon@telecom-bretagne.eu*

RÉSUMÉ. *La définition de fonctions d'oracle pour le test de transformation de modèles est compliquée par la complexité des modèles produits par la transformation. En effet, valider l'exactitude d'un modèle de sortie nécessite la vérification d'un grand nombre de ses propriétés structurelles et sémantiques. Dans cet article, nous proposons six fonctions d'oracle que nous évaluons selon deux qualités: le risque d'erreur introduit par l'oracle et la réutilisation de l'oracle.*

ABSTRACT. *The definition of an oracle function for model transformation is challenging because of the very complex nature of models resulting from a transformation. Validating the correctness of an output model requires checking a large number of properties on the structure and semantics of this model. The oracle function can thus be very complex if it checks every property. In this paper, we propose two oracle qualities that should be evaluated when defining an oracle function for a transformation: the risk of error introduced by the oracle and the reusability of the oracle. We also propose 6 oracle functions and compare them according to several quality factors that we introduced.*

MOTS-CLÉS : *transformation de modèles, test, oracle, comparaison de modèles, contrats, assertions, réutilisation*

KEYWORDS: *model transformation, test, oracle, model comparison, contracts, assertions, reuse*

## 1. Introduction

Les transformations de modèles sont intensivement utilisées en Ingénierie Dirigée par les Modèles (IDM). Ces transformations automatisent des opérations critiques au cours du développement tel que les raffinements, les « refactoring », la génération de code. L'automatisation de ces opérations permet d'augmenter la réutilisation d'éléments d'un projet à un autre ce qui entraine des économies en terme d'effort et de temps. Cependant, cette automatisation peut aussi introduire un risque supplémentaire d'erreur à cause de transformations de modèles erronées. Ainsi, le test systématique de transformations de modèles est nécessaire pour assurer le succès d'un développement IDM.

Deux problèmes doivent être résolus lors du test de transformations de modèles : la sélection de modèles de test efficaces et la définition de fonctions d'oracle. Dans de précédents travaux (Fleurey et al., 2007a; Sen et al., 2008), nous avons étudié la génération et la sélection de modèles de test. Cet article[1] porte sur la fonction d'oracle qui vérifie l'exactitude des modèles transformés.

La définition d'une fonction d'oracle pour une transformation de modèles est une tâche difficile car les modèles retournés par la transformation sont complexes. Une transformation de modèles retourne des modèles conformes à un méta-modèle et qui sont manipulés comme un graphe d'objets instances des méta-classes.

Dans cet article, nous proposons d'étudier plusieurs fonctions d'oracle et de les évaluer suivant deux critères qualitatifs. Le premier critère est lié au risque d'erreur dans l'oracle. En effet, ces fonctions d'oracles évaluent des modèles complexes, ce qui implique un grand nombre de vérifications de propriétés et accroît le risque d'erreur dans la définition de l'oracle. Le second critère est relatif à la réutilisation d'un oracle dans différents cas de test et lorsque la transformation de modèles évolue. Dans le but de qualifier une fonction d'oracle selon ces deux qualités, nous évaluons différentes propriétés caractérisant chaque fonction d'oracle.

Dans la section 2, nous présentons la problématique de l'oracle pour le test de transformations de modèles, les qualités que nous voulons évaluer et les propriétés d'un oracle qui influencent sa qualification. Dans la section 3, nous proposons six fonctions d'oracles différentes qui exploitent différentes données d'oracle grâce à des techniques utilisées en IDM. Dans la section 4, nous comparons ces fonctions selon le risque d'erreur qu'elles présentent et leur possibilité de réutilisation. Dans la section 5, nous illustrons cette analyse.

## 2. La problématique de l'oracle dans le test de transformation de modèle

La phase de test nécessite la création de *cas de test* composés d'une *donnée de test* et d'un *oracle*. Le testeur doit générer et sélectionner des modèles d'entrée de la

[1] Ces travaux sont supportés par le projet RNTL Domino

transformation pour disposer de données de test et doit les associer à des oracles qui analysent les modèles de sortie correspondant.

Dans cette section, nous introduisons la problématique de la définition d'oracles pour le test de transformation de modèles. L'oracle vérifie l'exactitude du modèle de sortie renvoyé par la transformation d'un modèle de test conformément à sa spécification. C'est une fonction avec plusieurs paramètres. Le premier paramètre est le modèle de sortie qui doit être analysé ; il est retourné par la transformation et est conforme au méta-modèle de sortie. Le testeur fournit le second paramètre que nous appelons la « *donnée d'oracle* ». Cette donnée fournit un ensemble de détails utiles à l'oracle pour vérifier le modèle de sortie. Le *modèle de sortie attendu* du cas de test est un exemple de donnée d'oracle pouvant être utilisé. Les fonctions d'oracle que nous introduisons section 3.2 utilisent différents types de données d'oracle qui fournissent différemment les informations nécessaires à la vérification. Le modèle de test peut aussi faire partie de la donnée d'oracle si la fonction d'oracle a besoin d'en extraire des informations. Nous continuons à parler d'oracle dans le reste de l'article comme d'une fonction d'oracle avec des paramètres effectifs.

Dans cet article, nous présentons, qualifions et comparons différentes fonctions d'oracles. Nous portons une attention particulière aux données d'oracle car ce sont elles qui nécessitent un travail d'écriture de la part du testeur, et servent de base pour la qualification et la comparaison des différentes fonctions d'oracle.

### 2.1. *Qualifier l'oracle*

Nous mettons en avant deux qualités d'une fonction d'oracle que nous voulons évaluer. Leur valeur va dépendre de la fonction elle-même, de son contexte d'utilisation, et surtout de la donnée d'oracle qu'elle exploite.

Premièrement, nous voulons évaluer le *risque d'erreur* introduit par l'oracle. Puisque les fonctions d'oracles contrôlent l'exactitude de modèles complexes, un nombre important de vérifications est réalisé. Ces vérifications se font sur des objets attendus, des relations nécessaires entre objets, avec différentes cardinalités et sur des propriétés du modèle tout entier. Le testeur définit ces vérifications dont la complexité augmente en même temps que celle de la transformation testée et des modèles qu'elle manipule. Face à la complexité de ses traitements, le testeur peut commettre des erreurs. Les erreurs peuvent être de deux types différents:

– *faux-négatif* : la fonction d'oracle annonce une erreur qui n'en est pas une. Dans ce cas, une phase d'analyse est nécessaire pour comprendre l'erreur annoncée, détecter le code qui serait erroné, corriger l'erreur suspectée. Dans le meilleur des cas, les conséquences sont un gaspillage de temps, de moyens, d'énergie. Dans le pire des cas, une correction inutile peut devenir une erreur dans la transformation. Ce type d'erreur survient quand la donnée d'oracle utilisée est erronée.

– *faux-positif* : l'oracle annonce le succès d'un test alors qu'une erreur est présente. Ce type d'erreur survient quand la donnée d'oracle utilisée est erronée ou

que les vérifications qu'elle définit sont incomplètes à cause d'un *manque de vérification* d'une partie du modèle de sortie ou d'une partie de la spécification.

Pour augmenter la qualité d'un oracle, il faut qu'il détecte les vraies erreurs et qu'il procède à tous les contrôles nécessaires. Néanmoins, il n'est pas obligatoire qu'un seul oracle dans un unique cas de test réalise la totalité des vérifications, mais l'ensemble des oracles et des cas de test qu'ils composent le doit.

Deuxièmement, nous considérons la *réutilisabilité* d'un oracle. Les transformations de modèles dans un développement dirigé par les modèles sont sujettes à de nombreuses évolutions. Même si les transformations sont développées pour être réutilisées d'un projet à un autre, elles doivent généralement être adaptées au préalable. Les adaptations peuvent concerner des changements dans les méta-modèles sources et cibles, des modifications ou suppressions d'exigences de la spécification, l'ajout de nouvelles exigences. Un tel exemple de réutilisation de transformation de modèles est donné par Fleurey et al. (Fleurey et al., 2007b) qui présentent une série de transformations pour la migration d'application qui nécessitent d'importantes adaptations pour chaque nouveau projet de migration. Réutiliser un cas de test pour le test d'une nouvelle version peut nécessiter l'adaptation de son oracle si les vérifications qu'il effectue sont altérées par les changements de la spécification. L'adaptation d'oracle est une tâche pour le testeur qui peut réduire grandement l'intérêt de la réutilisation des tests.

D'une autre manière, il est possible que plusieurs oracles de différents cas de test puissent réutiliser les mêmes données d'oracle.

### 2.2. *Quatre propriétés d'oracle*

Nous identifions plusieurs propriétés d'oracle qui selon leurs valeurs influencent les qualités d'un oracle. Ces propriétés dépendent à la fois de la fonction d'oracle considérée ainsi que de la donnée d'oracle qu'elle utilise.

#### 2.2.1. *Complexité*

La complexité d'une donnée d'oracle augmente avec sa taille, le nombre et la complexité des différents concepts qu'elle contient.

#### 2.2.2. *Redondance*

Il peut y avoir de la redondance dans les données d'oracle. Selon le type de donnée, il peut s'agir de concepts qu'elles contiennent plusieurs fois, ou d'opérations qu'elles réalisent plusieurs fois.

#### 2.2.3. *Complétude*

La complétude d'un oracle mesure l'étendue des vérifications ou leur nombre. On distingue principalement trois classes de complétude :

– *totale* : lorsque l'oracle vérifie l'ensemble de la spécification quelque soit le cas de test et le modèle de test.

– *semi-totale* : lorsque l'oracle vérifie entièrement l'exactitude d'un modèle résultant de la transformation d'un modèle de test.

– *partielle* : lorsque l'oracle ne vérifie qu'un nombre limité de propriétés. Le verdict d'un tel oracle ne statut que sur le respect d'une partie de la spécification. Ainsi plusieurs cas de test seront nécessaires avec différents oracles pour vérifier l'exactitude de la transformation d'un même modèle de test.

2.2.4. *Généricité*

Une fonction d'oracle est générique si elle n'est pas dédiée à un modèle de test particulier et au cas de test qui le contient. Une fonction d'oracle générique utilise le modèle de test dans ses données d'oracle en tant que paramètre.

### 2.3. *Les propriétés d'un oracle comme facteurs de qualité*

Les propriétés de chaque fonction d'oracle prennent différentes valeurs, selon les vérifications qu'elle effectue sur la base de la donnée d'oracle qu'elle utilise. Ces propriétés sont des facteurs de qualité qui permettent de qualifier chaque oracle.

Tout d'abord, nous relevons que certaines propriétés d'un oracle ne sont pas orthogonales puisque leurs valeurs sont corrélées. Un oracle avec une complétude totale est forcément générique (pas forcément l'inverse) puisqu'un seul suffit pour vérifier toute la transformation et donc tous les modèles qui lui sont passés. Par ailleurs, cette unicité limite le nombre de redondances qui surviennent dans la décomposition en plusieurs cas de test des vérifications : chaque vérification ne peut pas forcément se faire seule dans un cas de test (selon la fonction d'oracle utilisée), donc certaines vont se recouvrir, entraînant des redondances. Nous observons aussi une corrélation entre la complétude et la complexité. Plus la complétude d'un oracle est totale, et plus l'importance des vérifications qu'il effectue est grande jusqu'à considérer la totalité de la spécification pour vérifier la transformation de n'importe quel modèle de test. Dans ce cas la complexité de l'oracle augmente. Il est en effet nécessaire de fournir des informations de plus en plus nombreuses, importantes et précises pour atteindre la complétude totale de l'oracle.

Plusieurs propriétés d'un oracle augmentent le *risque d'erreur*. Evidemment, l'augmentation de la complexité accroît ce risque. Puisque la complétude totale implique une plus forte complexité, elle aussi est source d'erreur. D'un autre côté, plus l'oracle est complet et moins le risque de *manque de vérification* est important. Les redondances inutiles sont aussi un point négatif puisqu'elles augmentent le nombre de vérifications et donc le risque de faire une erreur dans l'une d'entres elles. Comme un oracle générique analyse et connecte les modèles de test et de sortie correspondant, cela augmente son risque d'erreur par rapport à un oracle considérant uniquement le modèle de sortie.

Plusieurs critères réduisent les possibilités de *réutilisation* d'un oracle dans différentes versions de la transformation de modèles. Si un oracle est complexe, il sera compliqué pour le testeur de le réutiliser, spécialement s'il doit le modifier pour

l'adapter à la nouvelle spécification. Plus la complétude d'un oracle est totale et plus il vérifie d'exigences de la spécification alors plus il va être affecté par les modifications de la spécification. Les redondances compliquent la réutilisation de cas de test avec une version modifiée de la transformation de modèles car une modification d'une exigence de la spécification qui serait vérifiée plusieurs fois devrait être adaptée autant de fois.

La généricité affecte positivement la *réutilisation* d'un oracle dans différents cas de test d'une même version de la transformation de modèles sous test.

## 3. Mise en œuvre en différentes solutions

Dans cette section, nous présentons différentes fonctions d'oracle qui varie selon les techniques qu'elles emploient et les données d'oracle qu'elles exploitent.

### 3.1. *Trois techniques utilisées en IDM et exploitables dans des fonctions d'oracle*

Nous énumérons ici trois techniques utilisées en IDM que nous avons identifiées pour leur capacité à analyser des modèles et que nous pouvons exploiter pour implémenter des fonctions d'oracle de transformation de modèles.

#### 3.1.1. *Comparaison de modèles*.

Les dépositaires de modèles et les technologies utilisées actuellement dans l'IDM stockent et manipulent les modèles comme des graphes d'objets. La complexité de ces structures de données rend difficile la mise au point de techniques de comparaison de modèles efficaces et fiables. En effet, en considérant les modèles comme des graphes d'objets, leur comparaison atteint la complexité d'un algorithme NP-complet. L'approche de Alanen et Porres dans (Alanen et al., 2003) simplifie le problème en comparant deux versions d'un même modèle, elle profite ainsi des identifiants des objets. Leur méthode n'est pas utilisable dans une phase de test car les modèles comparés par l'oracle viennent de sources différentes et leurs objets n'ont donc pas les mêmes identifiants. La comparaison des modèles doit donc se baser sur leur structure et leurs attributs. D'autres algorithmes exploitent les méta-modèles et les structures qu'ils définissent. De tels algorithmes sont proposés dans (Xing et al., 2005; Lin et al., 2007). La comparaison de modèles commence ainsi à être outillée par des outils comme EMF Compare (EclipseFoundation, 2007).

Différentes fonctions d'oracle peuvent employer la comparaison de modèles. Elles peuvent comparer le modèle renvoyé par la transformation du modèle de test avec un modèle de référence qui peut être disponible ou obtenu par le testeur.

#### 3.1.2. *Utilisation des contrats*

Les pre et post-conditions forment les contrats d'une méthode. Ce sont des assertions qui sont évaluées avant et après l'exécution d'une méthode (Le Traon et al., 2006). Des contrats peuvent être définis pour une transformation de modèles.

Les pré-conditions contraignent l'ensemble des modèles en entrée et les post-conditions vérifient un ensemble de propriétés des modèles de sortie.

Les travaux présentés dans (Briand et al., 2003; Le Traon et al., 2006) montrent l'intérêt de l'utilisation des contrats comme fonction d'oracle dans des systèmes orientés objet. Les bénéfices de ces approches peuvent être transcrits dans l'ingénierie des modèles. Dans de précédents travaux (Mottu et al., 2006), nous proposons une méthode pour spécifier et implémenter l'oracle de transformation de modèles avec des contrats exprimés en OCL. De tels contrats peuvent être implémentés dans d'autres langages et différents outils, comme Kermeta (Muller et al., 2005) ou ATOM3 (de Lara et al., 2002).

3.1.3. *Utilisation du pattern matching*

Le pattern matching est une technique qui n'est pas propre à l'ingénierie des modèles mais qui y est largement employée. Elle permet d'identifier les ensembles d'objets d'un modèle qui correspondent au pattern. Le pattern peut être exprimé par des assertions OCL et des « model snippets » (Ramos et al., 2007) par exemple.

Les model snippets sont des modèles partiels écrits à partir d'une version relâchée d'un métamodèle. Ils contiennent des objets instances des méta-classes du méta-modèle mais qui ne sont pas contraints par les cardinalités et contraintes du méta-modèle. La section 5.2.5 donne des exemples. Le testeur peut utiliser les mêmes outils pour écrire ces patterns que ceux utilisés pour les modèles de test. Les assertions OCL expriment des contraintes sur le modèle de sortie. Dans ce sens, elles peuvent être considérées comme des post-conditions, mais contrairement aux contrats, chaque pattern ne porte que sur un modèle de sortie particulier.

Ainsi exprimés, les patterns peuvent être considérés comme des assertions qui doivent être vraies après la transformation d'un modèle de test particulier. Chaque assertion ou une conjonction d'assertions peut être associée à un cas de test comme donnée d'oracle d'une fonction d'oracle.

### 3.2. *Six fonctions d'oracle pour le test de transformation de modèles*

Les techniques énumérées dans la précédente section peuvent être utilisées de différentes manières. Nous énumérons ici six fonctions d'oracle qui exploitent les différentes techniques et les différentes données d'oracle utilisées.

1- Oracle utilisant une *version de référence* de la transformation sous test :

Une comparaison est effectuée entre le modèle de sortie `(mtout)` renvoyé par la transformation d'un modèle de test et un modèle de référence renvoyé par la transformation de référence.

Le testeur fournit une donnée d'oracle qui est une transformation de référence `(R)` de la transformation de modèles sous test. Cette transformation de référence produit le modèle de référence à partir du modèle de test `(mt)`. Cet oracle est la fonction $O_1$ telle que :

```
O1(mtout , (R,mt)) : Boolean is do
    result := compare(mtout , R(mt))
end
```

2- Oracle utilisant une *transformation inverse*

Une comparaison est effectuée entre le modèle de test mt et le modèle obtenu après deux transformations successives du modèle de test : avec la transformation sous test, puis avec la transformation inverse. Le testeur doit fournir cette transformation inverse `(I)` comme donnée d'oracle. Cela n'est possible que si la transformation est une application injective (ce qui n'est pas systématique), sinon la transformation ne pourrait pas être inversée. Cet oracle est la fonction $O_2$ telle que :

```
O2(mtout , (I,mt)) : Boolean is do
    result := compare(mt , I(mtout))
end
```

3- Oracle utilisant un *modèle de sortie attendu*

Une comparaison est effectuée entre le modèle de sortie `(mtout)` renvoyé par la transformation du modèle de test et le modèle attendu `(mtexpected)` fourni par le testeur. Cet oracle est la fonction $O_3$ telle que :

```
O3(mtout,mtexpected) : Boolean is do
    result := compare(mtout, mtexpected)
end
```

4- Oracle utilisant un *contrat générique*

L'oracle vérifie que le modèle de sortie satisfait le contrat générique en fonction du modèle de test correspondant `(mt)`. Dans une fonction d'oracle, un contrat générique est une post-condition de la transformation qui contraint n'importe quel modèle de sortie `(mtout)` en fonction du modèle de test `(mt)` correspondant. Ce contrat peut vérifier l'exactitude de ce modèle de sortie en fonction d'une partie plus ou moins grande de la spécification. Le testeur doit fournir ce contrat générique `(Cg)`. Cet oracle est la fonction $O_4$ telle que :

```
O4(mtout, (Cg,mt)) : Boolean is do
    result := (mtout,mt).satisfies(Cg)
end
```

5- Oracle utilisant une *assertion OCL*

L'oracle vérifie que le modèle de sortie satisfait l'assertion OCL. Le testeur doit fournir une assertion OCL `(cd)` qui est capable d'analyser le modèle de sortie `(mtout)`. Cette contrainte est dédiée à un cas de test et à son modèle de test, elle vérifie seulement la validité du modèle de sortie correspondant. Le testeur va exprimer dans cette contrainte les propriétés que le modèle de sortie doit avoir. Il n'est pas nécessaire d'exprimer toutes les propriétés à la fois. Cet oracle est la fonction $O_5$ telle que :

```
O5(mtout,cd) : Boolean is do
    result := mtout.satisfies(cd))
end
```

6- Oracle utilisant des *model snippets*

L'oracle vérifie que le modèle de sortie contient le pattern. Le testeur fournit un pattern sous la forme d'une liste de model snippets (ms), chacun étant associé à une cardinalité (n) et à un opérateur de comparaison (op). Ces deux derniers définissent combien de fois le modèle de sortie (mtout) doit contenir le model snippet. Cet oracle est la fonction $O_6$ telle que :

```
O6(mtout,list{(ms,n,op)}) : Boolean is do
    result := list.forAll(compare(nb_match (mtout,ms), n, op))
    //compare(x,y,op) compare deux nombres x et y en fonction d'un
    //opérateur de comparaison op et renvoie un booléen
end
```

## 4. Qualifier chaque fonction d'oracle

Les différentes fonctions d'oracle utilisent différentes données d'oracle de différents types (Tableau 1). Selon la fonction, sa donnée d'oracle principale et le type de donnée, nous identifions leurs propriétés et nous les qualifions.

| Fonction d'oracle | Donnée d'oracle | Type de donnée |
|---|---|---|
| $o_1$ | Transformation de référence | Transformation de modèles |
| $o_2$ | Transformation inverse | Transformation de modèles |
| $o_3$ | Modèle attendu | Modèle de sortie |
| $o_4$ | Contrat générique | Post-condition OCL |
| $o_5$ | Assertion OCL | Contrainte OCL |
| $o_6$ | Model snippet | Model snippet |

**Tableau 1 - Fonctions d'oracle avec leur donnée d'oracle principale et leur type**

### 4.1. *Propriétés de chaque fonction d'oracle*

Les transformations de référence et inverse sont compliquées à écrire puisque elles ont la même complexité que la transformation sous test ; à moins que la transformation soit simple. Ecrite une seule fois pour vérifier le respect de toute la spécification, elles ont une complétude totale, ce qui limite les redondances. Elles sont génériques puisqu'elles sont utilisables avec n'importe quel modèle de test ou de sortie respectivement.

Les contrats génériques peuvent avoir une complétude partielle, voire totale. Dans les deux cas, les redondances ne sont pas évitées car des navigations et sélections sont répétées, ainsi qu'à cause du découpage des vérifications (comme illustré section 5.2.4). Les contrats sont compliqués à écrire car ils analysent à la fois les modèles de test et de sortie, ce qui les rend génériques. Un seul contrat pourrait être utilisé avec une complétude totale pour la vérification de toute la spécification.

Il serait composé d'une conjonction de vérifications qui auraient pu être utilisées séparément dans différents cas de test avec une complétude partielle.

| Fonction d'oracle | Complexité | Complétude | Redondance | Généricité |
|---|---|---|---|---|
| $o_1$ (transfo ref) | C | Totale | A+ | Oui |
| $o_2$ (transfo inv) | C | Totale | A+ | Oui |
| $o_3$ (m attendu) | B | Semi-totale | C | Non |
| $o_4$ (contrat gen) | C | Totale/Partielle | B | Oui |
| $o_5$ (assert OCL) | A | Partielle | A- | Non |
| $o_6$ (m snippet) | A | Partielle | A- | Non |

**Tableau 2 - Fonctions d'oracle avec les valeurs de leurs propriétés**

L'écriture des modèles attendus est un travail difficile, ils ont la même complexité que les modèles de sortie qui peut être très importante. Leur complétude est semi-totale puisqu'ils considèrent l'entière validité de chaque modèle de sortie, mais uniquement en validant les parties de la spécification que les modèles de test correspondants sollicitent. Chaque modèle de test ne sollicite pas toute la spécification, mais plusieurs solliciteront les mêmes parties ce qui implique des redondances. Dédiés chacun à un modèle de test, ils ne sont pas génériques.

Les model snippets et les assertions OCL ne sont pas compliqués à écrire car ils peuvent être très réduits pour vérifier une seule partie de la spécification sur une partie d'un modèle de sortie. Ils ne sont donc pas génériques mais ils évitent les redondances avec une complétude partielle.

Dans le Tableau 2, nous résumons les valeurs des différentes propriétés de chaque fonction d'oracle en utilisant parfois des lettres (de A+ à C) selon leur importance. Ainsi la complexité des transformations de référence et inverse est la pire mais leurs redondances sont légèrement moins nombreuses que celle des assertions OCL et des model snippets.

### 4.2. *Qualité de chaque fonction d'oracle*

En fonction des propriétés de chaque fonction d'oracle, leurs qualités varient, comme illustrée dans le Tableau 3.

Le *risque d'erreur* est fortement influencé par la complexité et les redondances de l'oracle. Ainsi les transformations de référence et inverse impliquent un fort risque d'erreur (valeur C). Les assertions OCL et les model snippets ont le plus faible risque d'erreur car elles ne sont pas complexes. Ce risque est quelque peu augmenté (A-) par leur complétude partielle qui peut causer des *manques de vérifications*. Les modèles attendus ont un risque d'erreur limité dû à leur complexité et leurs redondances.

La *réutilisation* de ces oracles avec une même version dépend de leur généricité. Ainsi les transformations de référence et inverses, et les contrats génériques ont cette qualité. Les assertions OCL et les model snippets ont peu cette qualité mais leur concision leur permet de convenir à la vérification de différent modèles de sortie.

Les modèles attendus n'ont que très rarement cette qualité quand la transformation est surjective.

| Fonction d'oracle | Risque d'erreur | Réutilisation avec une même version | Réutilisation avec différentes versions |
|---|---|---|---|
| $o_1$ (transfo ref) | C | A | C |
| $o_2$ (transfo inv) | C | A | C |
| $o_3$ (m attendu) | B | C+ | C |
| $o_4$ (contrat gen) | B | A | B- |
| $o_5$ (assert OCL) | A- | B | A |
| $o_6$ (m snippet) | A- | B | A |

**Tableau 3 - Fonctions d'oracle avec leurs qualités**

La *réutilisation* de ces oracles avec une autre version est principalement influencée par leur complétude. Dans ce cas les assertions OCL et les model snippets ont le plus cette qualité. Un contrat générique n'a cette qualité que s'il a une complétude partielle et qu'il ne considère que des parties de la spécification non modifiées. Les modèles attendus ont trop de redondances de vérifications et une complétude semi-totale qui empêchent leur réutilisation sans modifier plusieurs modèles attendus à chaque modification de la spécification. Les transformations de référence et inverse sont pénalisées par leur complétude totale.

## 5. Evaluation sur un exemple

Dans cette section, nous illustrons l'implémentation d'oracles avec les six différentes fonctions d'oracle que nous proposons. Cela nous permet de démontrer leurs qualités en fonction de leurs propriétés.

### 5.1. *Illustration*

Nous utilisons une transformation qui à partir d'un modèle de Classe produit un modèle RDBMS. La spécification initiale que nous allons suivre correspond à l'appel à soumission du workshop MTIP'05 (Bézivin et al., 2005). Elle se compose d'un ensemble de règles qui forme une spécification complexe. L'implémentation d'un tel système nécessite l'emploi de traitements importants sur le modèle d'entrée avec de la récursivité, des navigations avec fermeture transitive, différentes passes. Différentes règles définissent comment les classes persistantes d'un modèle de Classe (comme celui de la figure 1), leurs attributs et associations sont transformés en tables, avec colonnes et clés. La transformation de modèles (`T`) doit ainsi produire le modèle RDBMS de la figure 2 à partir de `Mt1`.

Nous ne rentrons pas dans les détails de la spécification de la transformation (`T`) mais seulement d'une partie de ses règles que nous composons en une règle homogène `Ru`:

`Ru` : « Les classes persistantes, et uniquement celles là, sont transformées en tables dont les noms correspondent, sauf si elles héritent directement ou indirectement d'une autre classe persistante »

Par cette règle, la transformation produit dans le modèle de sortie (figure 2) la table nommée B et uniquement celle-là.

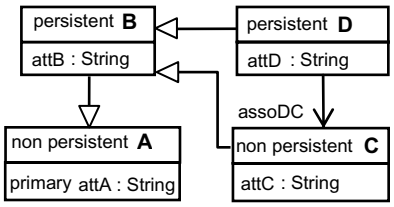


**Figure 1. Modèle d'entrée Mt1**

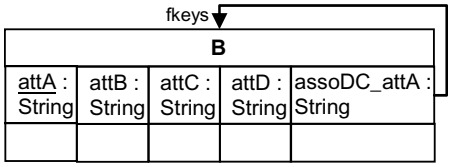


**Figure 2. Modèle de sortie T(Mt1)**

### 5.2. *Mise en œuvre dans des cas de test*

Dans un premier temps, le testeur dispose d'un ensemble de modèles de test qu'il a pu générer, sélectionner, ou écrire et qui forment la première partie de cas de test. Ces cas de test doivent être complétés par des oracles qui vérifient que les modèles de sorties produits sont conformes à la spécification. Dans cette section illustrative, nous considérons uniquement le modèle de test de la figure 1 et vérifions que le modèle de sortie produit par la transformation de modèles sous test respecte la règle `Ru`. Pour illustrer la réutilisabilité des différents oracles, nous créons une évolution `T'` de la transformation `T` par une modification de la règle `Ru` :

`Ru'` : « Les classes persistantes, et uniquement celles là, sont transformées en tables dont les noms correspondent »

Avec cette nouvelle spécification, la transformation `T'` devrait produire le modèle RDBMS de la figure 3 à partir du model de test `Mt1`.

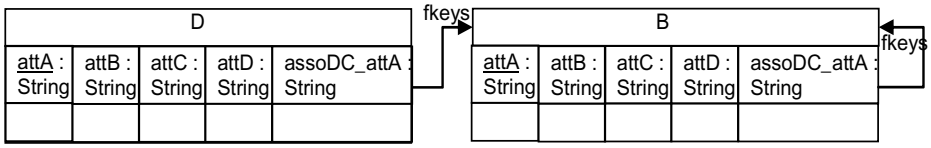


**Figure 3. Modèle de sortie T'(Mt1)**

#### 5.2.1. *Transformation de modèles de référence*

L'implémentation de la transformation sous test a été réalisée en Kermeta (Muller et al., 2005) et se compose de 113 lignes de code en 11 opérations. Le testeur peut utiliser une autre implémentation réalisée pour le workshop comme celle de Lawley et al. (Lawley et al., 2005). C'est un programme fonctionnel de 94 lignes de code, contenant 8 patterns et 5 règles. La complexité de cette seconde implémentation est importante, sa validité n'est pas plus assurée. De plus quand le

testeur va devoir l'adapter à la nouvelle spécification, il devra apprendre un nouveau langage, comprendre la mise en œuvre et la modifier correctement. Ainsi, même si le testeur peut tirer avantage d'une transformation de référence existante, sa réutilisation difficile avec une nouvelle version et le risque dû à sa complexité n'assurent pas la qualité de cet oracle. L'écriture d'une version de référence est trop complexe, c'est une tâche de développeur plutôt que de testeur.

5.2.2. *Transformation inverse*

La transformation étudiée n'est pas injective, les éléments non transformés du modèle, comme les classes A ou C, ne peuvent pas être déduits du modèle de sortie. Ainsi il n'est pas possible d'écrire de transformation inverse pour cet exemple, ce qui est souvent le cas d'après notre expérience.

5.2.3. *Modèle attendu*

Le testeur doit écrire les modèles attendus illustrés figure 2 et figure 3. Ces modèles sont évidemment aussi *complexes* que les modèles de sortie. La complétude de cet oracle est *semi-totale*, nous remarquons qu'il est nécessaire de créer aussi des colonnes et des clefs alors que nous ne considérons qu'une règle de la spécification portant sur la création des tables. Le *risque d'erreur* est ainsi augmenté par des vérifications inutiles. Par ailleurs, il est important de constater l'importance de l'adaptation nécessaire comparée à la simplicité de l'évolution de la spécification. Cet oracle double l'effort nécessaire pour écrire le nouveau modèle attendu, cela pour vérifier la création d'une seule nouvelle instance de Table. Notons que cet effort serait à reproduire dans tous les autres cas de test.

5.2.4. *Contrat générique*

Il est possible d'écrire ce contrat générique en OCL :

```
post table_correctly_created :
result.table.size = inputModel.classifier.select(cr|cr.oclIsTypeOf(Class))
 .select(cs | cs.oclAsType(Class).is_persistent)
 .select(csp | not csp.oclAsType(Class).parents.exists(p | p.is_persistent)
).size and //note: les classes ont des noms différents
inputModel.classifier.select(cr | cr.oclIsTypeOf(Class))
 .select(cs | cs.oclAsType(Class).is_persistent)
 .select(csp | not csp.oclAsType(Class).parents.exists(p | p.is_persistent))
 .forAll(csp | result.table.exists(t | t.name = csp.name))
```

Ce contrat n'est pas très complexe, mais la règle considérée est une des plus simples. Il est possible de l'écrire différemment en particulier avec un autre langage. Mais nous pouvons remarquer que les navigations et sélections sont répétitives. Pour tester toute la transformation, un ensemble de contrats génériques doit être écrit pour considérer toutes les exigences. Nous avons eu besoin de 14 contrats pour cela. Ce contrat peut être réutilisé pour la nouvelle version de la transformation en enlevant les deux select(…) en gras. Les vérifications qui sont réalisées dans les autres contrats devraient aussi considérer cette modification. Par exemple, les vérifications de correspondance entre les attributs et les colonnes prennent en compte les classes

et les tables qui les contiennent. Ainsi les navigations et filtrages du contrat considéré sont répétés dans plusieurs autres contrats et devront donc être modifiés.

5.2.5. Model snippets

La figure 4 représente cinq model snippets basés sur le métamodèle RDBMS. `MF1` à `MF4` définissent seulement une table nommée, et `MF5` une table sans nom. Ainsi, ces model snippets peuvent être utilisés dans des données d'oracle et leurs oracles quand le testeur veut vérifier la présence d'une table nommée `B` (avec `MF1`), ou `A` (avec `MF2`), ou `C` (avec `MF3`), ou `D` (avec `MF4`), ou simplement la présence d'une table sans considération de nom (avec `MF5`). Nous utilisons ces model snippets pour écrire plusieurs oracles qui vérifient `Ru` pour le modèle de test `Mt1` et que nous utilisons dans quatre cas de test :

```
o1: {(MF1 , 1 , =)}                          o3: {(MF4 , 0 , =)}
o2: {(MF2 , 0 , =) , (MF3 , 0 , =)}          o4: {(MF5 , 1 , =)}
```

L'oracle `o1` valide « Les classes persistantes […] sont transformées en tables dont les noms correspondent » : puisqu'il y a une classe persistante `B` dans le modèle de test, `o1` vérifie la présence d'une table `B` dans le modèle de sortie. `o2` valide « et uniquement celles là » : puisqu'il y a deux classes non persistantes dans `Mt1`, `o2` vérifie qu'il n'y a pas de table de même nom. `o3` valide « sauf si elles héritent directement ou indirectement d'une autre classe persistante » : puisqu'il y a une classe persistante `D` qui hérite d'une classe persistante, `o3` vérifie qu'il n'y a pas de table `D`. Finalement, `o4` valide aussi « et uniquement celles là » mais sans considérer spécifiquement l'attribut persistant : `o4` vérifie qu'une seule table est créée dans le modèle de sortie.

Ces model snippets et leurs oracles sont simples à écrire et à modulariser en fonction de la règle qu'ils considèrent. Ils sont facilement réutilisables avec la nouvelle version : `o1` et `o2` sont similaires. `o3` est adapté en changeant sa cardinalité à 1 puisqu'une classe persistante même héritant d'une classe persistante doit désormais être transformée et o4 prend la cardinalité 2 puisque le modèle de sortie doit contenir deux tables.


**Figure 4. Cinq model snippets de tables**

5.2.6. Assertions OCL

Les patterns des quatre précédents oracles peuvent être implémentés avec des assertions OCL. Par exemple le second pattern serait :

```
result.table.select(t|t.name=A).size()=0 and
result.table.select(t|t.name=C).size()=0
```

Ces assertions sont réutilisables facilement aussi. Elles sont un peu moins simple à écrire que les model snippets car, même si apprendre un langage tel qu'OCL n'est pas complexe, l'écriture de patterns de la même façon que les modèles (de test) est un avantage des model snippets.

## 6. Travaux relatifs

Les travaux portants sur le test de transformation de modèles s'intéressent principalement à la génération et la sélection des modèles de test. Fleurey et al. (Fleurey et al., 2007a) définissent plusieurs critères de test en adaptant une technique de partitionnement du domaine d'entrée aux méta-modèles d'entrée de la transformation de modèles. La génération automatique de modèles a été étudiée par Ehrig et al. (Ehrig et al., 2006) et dans (Sen et al., 2008) nous proposons une approche de sélection de modèles de test.

Concernant l'oracle du test de transformation de modèles, plusieurs travaux (Lin et al., 2007), (Heckel et al., 2003) ont identifié que le modèle attendu pouvait être comparé dans un oracle. Dans (Küster et al., 2006), les auteurs considèrent qu'en plus les contraintes de la spécification peuvent être utilisées. Kolovos et al. (Kolovos et al., 2006) présentent une variation dans l'utilisation de la comparaison de modèles dans une fonction d'oracle. Ils ne réalisent pas de comparaison avec un modèle de référence, mais ils définissent des règles entre le modèle de test et le modèle de sortie qui sont basées sur la comparaison des objets de ces modèles.

## 7. Conclusion

Nous avons présenté la problématique de l'oracle pour le test de transformation de modèles. La complexité des transformations de modèles et des modèles manipulés rend difficile l'écriture d'oracle. Nous introduisons six fonctions d'oracle qui exploitent les techniques de comparaison de modèles, de contrats et de pattern matching. Ces fonctions d'oracle exploitent différentes données d'oracle et se caractérisent par différentes propriétés qui nous permettent de les qualifier selon leur risque d'erreur et leur réutilisabilité. Lors d'évolution de la transformation, les oracles utilisant des transformations de références et inverses, des modèles attendues sont moins adaptés que l'utilisation de model snippets et d'assertions OCL. Ces derniers disposent en plus d'une complexité faible limitant le risque d'erreurs. Les contrats génériques peuvent avoir des avantages proches si leurs vérifications sont suffisamment décomposées. Transformations de références et inverses ne bénéficient que de leur réutilisation dans tous les cas de test d'une version.

## 8. Références

Alanen, M. and I. Porres. "Difference and Union of Models". *UML'03 (Unified Modeling Language)*, San Francisco, CA, USA, 2003.

Bézivin, J., B. Rumpe, A. Schürr and L. Tratt. (2005). "MTIP workshop CFP." from http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf.

Briand, L. C., Y. Labiche and H. Sun. "Investigating the Use of Analysis Contracts to Improve the Testability of Object Oriented Code." *Software Practice and Experience* 33(7), 2003.

de Lara, J. and H. Vangheluwe. "AToM3: A Tool for Multi-formalism and Meta-modelling". *FASE '02*, Grenoble, France, 2002.

EclipseFoundation. (2007). "EMF Compare." from www.eclipse.org/emft/projects/compare.

Ehrig, K., J. M. Küster, G. Taentzer and J. Winkelmann. "Generating Instance Models from Meta Models". *FMOODS'06 (Formal Methods for Open Object-Based Distributed Systems)*, Bologna, Italy, 2006.

Fleurey, F., B. Baudry, P.-A. Muller and Y. Le Traon. "Towards Dependable Model Transformations: Qualifying Input Test Data." *Software and Systems Modeling*, 2007a.

Fleurey, F., E. Breton, B. Baudry, A. Nicolas and J.-M. Jézéquel. "Model-Driven Engineering for Software Migration in a Large Industrial Context." *MoDELS/UML 2007 conference*, Nashville, USA, 2007b.

Heckel, R. and M. Lohmann. "Towards Model-Driven Testing." *Electronic Notes in Theoretical Computer Science* 82(6), 2003.

Kolovos, D. S., R. F. Paige and F. a. C. Polack. "Model Comparison: A Foundation for Model Composition and Model Transformation Testing". *workshop GaMMa'06*, Shangaï, China, 2006.

Küster, J. M. and M. Abd-El-Razik. "Validation of Model Transformations - First Experiences using a White Box Approach". *workshop MoDeV²a 2006, part of MoDELS'06*, Genova, Italy, 2006.

Lawley, M. and J. Steel. "Practical Declarative Model Transformation With Tefkat". *Model Transformation in Practice Workshop, part of MoDELS'05*, Montego Bay, Jamaica, 2005.

Le Traon, Y., B. Baudry and J.-M. Jézéquel. "Design by Contract to Improve Software Vigilance." *IEEE Transactions on Software Engineering* 32(8), 2006.

Lin, Y., J. Gray and F. Jouault. "DSMDiff: A Differentiation Tool for Domain-Specific Models." *European Journal of Information Systems, Special Issue on Model-Driven Systems Development*, 2007.

Mottu, J.-M., B. Baudry and Y. Le Traon. "Reusable MDA Components: A Testing-for-Trust Approach". *MoDELS'06*, Genova, Italy, 2006.

Muller, P.-A., et al. "On Executable Meta-Languages applied to Model Transformations". *Model Transformation in Practice Workshop, part of MoDELS'05*, Montego Bay, Jamaica, 2005.

Ramos, R., O. Barais and J.-M. Jézéquel. "Matching Model-Snippets". *MoDELS'07*, Nashville, USA, 2007.

Sen, S., B. Baudry and J.-M. Mottu. "On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing". *Proceedings of ICST'08*, Lillehammer, Norway, 2008.

Xing, Z. and E. Stroulia. "UMLDiff: An Algorithm for Object-Oriented Design Differencing". *Automated Software Engineering (ASE'05)*, USA, 2005.