Comparaison de modèles filtrée pour le test de transformations de modèles

Olivier Finot
LINA CNRS UMR 6241
Université de Nantes
2, rue de la Houssinière
F-44322 Nantes Cedex, France
olivier.finot@univ-nantes.fr

Jean-Marie Mottu
LINA CNRS UMR 6241
Université de Nantes
2, rue de la Houssinière
F-44322 Nantes Cedex, France
jean-marie.mottu@univ-nantes.fr

Gerson Sunyé
INRIA - Université de Nantes
Campus de Beaulieu
35000 Rennes
gerson.sunye@univ-nantes.fr

Christian Attiogbe
LINA CNRS UMR 6241 - Université de Nantes
2, rue de la Houssinière
F-44322 Nantes Cedex, France
christian.attiogbe@univ-nantes.fr

Résumé

Dans l'Ingénierie Dirigée par les Modèles, le traitement des modèles est réalisé automatiquement par des transformations. Nous nous intéressons au test de ces transformations de modèles et ici à la définition d'oracles qui analysent les modèles de sortie pour vérifier leur correction par rapport à la spécification. Une difficulté est que la spécification d'une transformation peut accepter plusieurs versions différentes du même modèle de sortie. Nous considérons ici l'oracle pour le test de ces transformations aux sorties polymorphes. Jusqu'à présent, il faut soit définir l'ensemble des modèles possibles en sortie de transformation, travail rendu difficile par la complexité de ces modèles, soit décomposer la vérification d'un modèle en vérifications de ses propriétés, communes ou non aux différentes variantes. Nous proposons dans cet article une méthode permettant de n'utiliser qu'un seul oracle. Elle consiste à comparer le modèle de sortie obtenu et un seul des modèles de sortie possibles, puis à filtrer le résultat de la comparaison. Par ce filtrage, nous obtenons un verdict en excluant du résultat de la comparaison les éléments non communs aux variantes du modèle. Finalement, nous étudions comment vérifier la partie non commune du modèle de sortie obtenu.

Mots-clé : Comparaison de modèles, transformation de modèles, test, oracle

Within Model Driven Engineering, models are automatically processed by transformations. We are interested in testing these model transformations and more precisely at defining oracles that analyse output models to ensure their correctness w.r.t. the transformation's specification. The fact that a transformation's specification can accept several versions of the same output model is a difficulty. We are considering the oracle for the test of these transformations with polymorphic outputs. So far, we have to either define the whole set of possible output models for a transformation, which is difficult because of these models' complexity, or we split the model's verification into verifying its properties, properties that may or not be common to all the variants of the output model. In this paper we propose a method that allows us to use only one oracle. Its principle is to compare the model obtained after execution of the transformation with only one of the possible output models ; the result of this comparison is then filtered. With this filtering, we obtain a verdict by excluding from the comparison result elements which do not belong to the common part of the model's variants. Finally we deal with the verification of the not common parts of the output model.

Keywords : Model comparison, model transformation, test, oracle

1 Introduction

Avec l'approche d'Ingénierie Dirigée par les Modèles (IDM), la modélisation est la base du développement logiciel. Les transformations de modèles au cœur de cette approche, ont vocation à être automatisées, pour permettre une grande réutilisation.

Un cycle de développement exploitant l'IDM utilise et réutilise des modèles et leurs transformations. Chaque développement enchaîne modélisation, transformation, réusinage manuel de modèles pour produire le code final d'une application, par exemple. La multiplication de ces étapes impose d'anticiper les vérifications pour éviter la propagation et la dissémination d'erreurs. Une erreur dans une transformation produira de nombreux modèles erronés. Une erreur d'un modèle sera transmise jusqu'au code final. Il est fastidieux de localiser la source d'une erreur quand elle n'est rencontrée que dans le code. C'est pourquoi il faut vérifier la correction des phases de transformation et de modélisation. Nous nous intéressons ici à la vérification des transformations de modèles, en utilisant des techniques de test.

Pour chaque test d'une transformation, nous associons un modèle d'entrée à un oracle. Notre étude porte sur cet oracle qui contrôle l'exactitude du modèle de sortie produit. Certaines transformations de modèles appliquées à un modèle d'entrée donné peuvent produire plusieurs variantes d'un même modèle de sortie. Plus précisément, la spécification de certaines transformations de modèles peut accepter plusieurs sorties différentes qui sont en général des variantes d'une même sortie. Ces transformations de modèles ont des *sorties polymorphes*. Par exemple, un modèle d'automate dont plusieurs transitions relient des états finaux distincts est une variante du même automate dont les transitions relient un même état final. Même si l'implantation de la transformation est déterministe et produit toujours la même variante d'un modèle, sa spécification peut autoriser des sorties polymorphes. Le testeur doit considérer ces variantes d'un modèle de sortie car la spécification impose généralement le sens des modèles de sorties sans en préciser la syntaxe. Quand ces variations ne concernent que des parties précises des modèles, le testeur doit pouvoir vérifier les parties qui sont communes aux différentes variantes sans que le polymorphisme empêche d'obtenir tout verdict.

Généralement, l'oracle du test d'une transformation de modèles repose sur l'utilisation d'un modèle de sortie de référence, comparé au modèle de sortie obtenu après exécution de la transformation de modèles. Dans le cas du test de transformations de modèles aux sorties polymorphes, il serait nécessaire de définir autant de modèles de référence qu'il existe de modèles de sortie valides. Notre objectif est de proposer une méthode plus efficace pour l'oracle du test de ces transformations.

Nous proposons d'améliorer l'oracle du test de transformations de modèles en séparant le contrôle de la partie commune de celui de la partie variable d'un modèle de sortie polymorphe. Notre principale contribution est de produire un verdict partiel du test de la partie commune d'un modèle polymorphe produit par une transformation de modèles. Un gain d'efficacité se fait en n'ayant à produire qu'un seul modèle de référence et à n'exécuter qu'une seule comparaison de modèles. L'effort pourra ainsi être reporté sur la vérification de la partie variable du modèle polymorphe.

Notre approche consiste à écrire un seul modèle de sortie de référence par modèle d'entrée. Le modèle d'entrée étant obtenu par d'autres techniques [SBM09], le modèle de sortie de référence est écrit par le testeur en interprétant la spécification. L'exécution de la transformation sur le modèle d'entrée produit un modèle de sortie que nous comparons avec notre modèle de sortie de référence. Cette comparaison renvoie les différences que nous filtrons pour ne conserver que celles qui concernent la partie commune du modèle polymorphe. Ce filtrage est paramétré par les éléments du métamodèle dont le testeur veut reporter le contrôle dans la partie variable du

modèle polymorphe. Nous produisons un verdict partiel pour ce test à partir du résultat filtré. Si des différences sont observées dans la partie commune, nous concluons que le test a échoué. Sinon le verdict est positif sur les parties du métamodèle considérées. Restera à analyser la partie non commune du modèle de sortie polymorphe.

Cet article s'articule de la manière suivante. Dans la section 2, nous détaillons la problématique qui nous intéresse et son contexte. Nous présentons notamment la création d'oracles pour le test de transformations de modèles et deux cas d'étude. Dans la section 3, nous détaillons notre approche. Nous montrons que nous pouvons utiliser une comparaison de modèles filtrée, pour tester une transformation de modèles aux sorties polymorphes. Dans la section 4, nous appliquons notre proposition à nos cas d'étude. Puis nous discuterons des résultats.

2 Oracle du test de transformations de modèles aux sorties polymorphes

Tester un système, un programme, c'est le faire exécuter des données d'entrées et contrôler que les résultats produits sont ceux attendus. Le programme que l'on teste est le *programme sous test*, les données d'entrée sont les *données de test*. Les résultats produits par l'exécution du programme sont examinés par l'*oracle* du test, qui produit un *verdict*. Si les résultats sont conformes, le verdict est positif et le test *passe*, sinon il *échoue*. Un *cas de test* regroupe essentiellement une initialisation, une donnée de test, un oracle.

2.1 Test de transformations aux sorties polymorphes

Tester une transformation de modèles consiste à transformer un ensemble de *modèles de test* et à contrôler avec des oracles que les modèles de sortie obtenus soient corrects, comme illustré figure 1. De cette manière, le testeur vérifie la correction de la transformation sous test vis-à-vis de sa spécification. Chaque modèle étant conforme à son métamodèle, on peut vérifier par exemple (i) qu'un objet conforme à une métaclasse d'entrée soit transformé en un objet conforme à une autre métaclasse de sortie (e.g. une classe transformée en table), (ii) qu'un objet soit préservé entre entrée et sortie (e.g. les états non finaux d'un automate ne changent pas si l'on transforme plusieurs états finaux en un seul), ou au contraire (iii) qu'un concept ne soit plus instancié en sortie de transformation (e.g. les états composites disparus après mise à plat d'un automate). Le testeur se base sur une spécification qui de manière générale est textuelle, et en tout cas non formelle dans le cadre de notre travail. Le workshop MTIP05 [**?**] avait par exemple spécifié textuellement plusieurs transformations pour comparer les différentes manières de les implanter.

La tâche du testeur est de produire des modèles de test, ce qui est automatisable [SBM09] et d'associer des oracles à ces modèles de test pour former les cas de test. Cette deuxième tâche nous intéresse ici.

Produire les oracles est difficile car le testeur ne peut pas automatiser cette tâche à partir d'une spécification non formelle. De plus, certaines transformations produisent des modèles de sortie polymorphes : plusieurs modèles de sortie différents peuvent être des résultats valides pour un modèle d'entrée donné. Toutes les variantes d'un modèle de sortie polymorphe devant être analysables par l'oracle, cela rend la tâche du testeur extrêmement coûteuse.

Un exemple de transformation de modèles aux sorties polymorphes est la mise à plat de machine à états. Il s'agit de supprimer les états composites comme illustré par la transformation de la machine à états de la figure 3 en celle de la figure 4(a). L'oracle de ce test devra par exemple

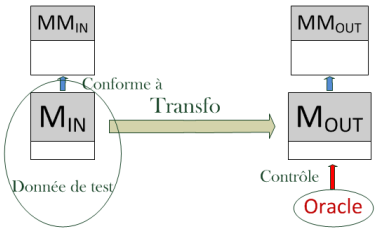


FIGURE 1 – Test d'une transformation de modèles

vérifier que l'état A relie B, que B et C sont toujours reliés, etc. Cette transformation produit des sorties polymorphes, ce qu'on voit dans la variante de la figure 4(b). La difficulté pour le testeur sera de faire les précédentes vérifications, tout en vérifiant que B et C sont liés à un état final, mais pas forcément le même. Nous redétaillerons cette transformation section 2.3.

Ces transformations de modèles aux sorties polymorphes sont fréquentes, en particulier quand le métamodèle (respectivement le langage) de sortie est riche. Dans de nombreux cas, la sortie exacte ne peut pas être anticipée par le testeur. Soit la spécification exprime directement que plusieurs variantes sont acceptables, soit le métamodèle de sortie permet d'exprimer une sémantique définie en laissant une liberté sur la syntaxe d'un modèle. L'équivalence sémantique [MS03] est un problème complexe à résoudre de manière statique et il faudrait que nos modèles soient exécutables pour pouvoir la vérifier de manière dynamique, ce qui n'est pas le cas à cette étape du développement. Par ailleurs si un testeur peut exploiter le code (boîte blanche) pour produire les modèles de test, il ne doit pas le faire pour l'oracle sous peine d'être influencé par les erreurs qu'il recherche. De plus, le polymorphisme peut aussi être introduit par la spécification du langage de transformation ou son implantation (interpréteur).

Nous illustrons dans la sous-section 2.3 deux cas d'étude de transformations de modèles aux sorties polymorphes de 1 vers 1 (un modèle d'entrée transformé en un modèle de sortie) mais la problématique est la même avec des transformationsde n vers m (n de 1 à p, m de 1 à q). Si un modèle parmi m est polymorphe, notre proposition est utile.

2.2 Oracle contrôlant des modèles polymorphes

Nous avons déjà étudié la problématique de la définition d'un oracle pour le test de transformations de modèles [MBLT08]. Nous avons proposé six fonctions d'oracle, que nous présentons dans un premier temps, dans un second temps, nous ferons le lien avec notre problématique :

- Utilisation d'un modèle de sortie attendu : c'est un modèle de référence que la fonction d'oracle compare avec le modèle de sortie obtenu par la transformation d'un modèle de test. Il est fourni par le testeur qui le définit à partir de la spécification.
- Utilisation d'une version de référence de la transformation : si elle existe, elle est similaire fonctionnellement à la transformation sous test. La fonction d'oracle compare les modèles de sortie respectifs de la transformation sous test et celle de référence.
- Utilisation d'une transformation inverse : si elle existe, son action est l'inverse de la transformation testée. Le modèle de test est traité par la transformation sous test puis le modèle de sortie est transformé par la transformation de référence. Le modèle d'entrée ainsi obtenu est comparé avec le modèle de test.
- Utilisation d'un contrat générique : la fonction d'oracle utilise un contrat générique défini sous la forme d'une post-condition de la transformation, généralement en OCL. Le contrat valable pour n'importe quel modèle de test peut être un extrait de la spécification, fourni

par le développeur de la transformation ou produit par le testeur.
- Utilisation d'une assertion : la fonction vérifie que le modèle de sortie satisfait une assertion définie par le testeur, généralement en OCL. Cette assertion impose des propriétés spécifiques au modèle de sortie issu de la transformation du modèle de test.
- Utilisation d'un *pattern matching* : la fonction d'oracle vérifie que le modèle de sortie contient un *pattern*. Le *pattern* permet d'exprimer des propriétés attendues dans le modèle de sortie. Sous la forme de model snippet [RBJ07], il s'agit de bouts de modèle.

Une fonction d'oracle adaptée au contrôle de modèles polymorphes, doit analyser un modèle de sortie en considérant ces différentes variantes (comme illustré figure 2). En particulier, il ne faut pas obtenir de verdict faux alors que la partie variable a été contrôlée à tort incorrecte, ni obtenir de verdict vrai alors que la partie commune était incorrecte. Les fonctions d'oracle existantes ne sont pas adaptées :
- Utilisation d'un modèle de sortie attendu : pour chaque test, le testeur devra définir manuellement tous les modèles variantes et les comparer au modèle de sortie. Il faut une et une seule égalité stricte pour obtenir un verdict positif au test.
- Utilisation d'une version de référence de la transformation : cette référence ne produira qu'une variante du modèle de sortie. Les autres restant à la charge du testeur.
- Utilisation d'une transformation inverse : cette fonction est limitée aux transformations injectives, peu fréquentes, en particulier avec des sorties polymorphes. Si une telle transformation existait elle reproduirait un modèle d'entrée lui-même polymorphe dont la comparaison avec le modèle de test serait aussi difficile que pour les oracles précédents.
- Utilisation d'un contrat générique : nous avions déjà expliqué dans [MBLT08] qu'il était difficile d'exprimer un contrat pour des modèles complexes, ce qui est souvent le cas des transformations aux sorties polymorphes dont le métamodèle de sortie est riche. Il est coûteux d'exprimer les variantes du modèle de sortie à l'aide d'un contrat. Celui-ci deviendrait une expression complexe à écrire et à maintenir.
- Utilisation d'assertion ou de *pattern matching* : ces deux fonctions d'oracle sont appropriées pour le contrôle de parties du modèle. De petites tailles, elles sont peu appropriées au contrôle de la partie commune d'un modèle polymorphe, qui représente la majorité du modèle. Nous discutons plus loin de leur utilisation pour contrôler les parties variables d'un modèle polymorphe.

Ces fonctions d'oracle ont le défaut de nécessiter toutes les variantes d'un modèle polymorphe (pour les trois premières), d'être trop complexes (pour la quatrième), ou par leur simplicité d'être seulement adaptés au contrôle de parties restreintes d'un modèle. Nous proposons dans la section 3 d'adapter la comparaison de modèle utilisé dans les trois premières fonctions d'oracle pour produire un verdict à partir d'une seule variante d'un modèle polymorphe.

2.3 Oracle pour le test de deux transformations aux sorties polymorphes

Nous illustrons cet article avec deux cas d'étude. Ces deux transformations de modèles aux sorties polymorphes transforment des modèles UML. La première **T1** réalise la mise à plat d'une machine à états hiérarchique. La seconde **T2** transforme un diagramme d'activités en un modèle représentant un programme CSP (Communicating Sequential Processes)[Hoa78].

Mise à plat d'une machine à états

Le diagramme d'états-transitions ou machine à états est un des diagrammes de la norme UML. Il est utilisé pour représenter les états d'un système, et les conditions de modification de

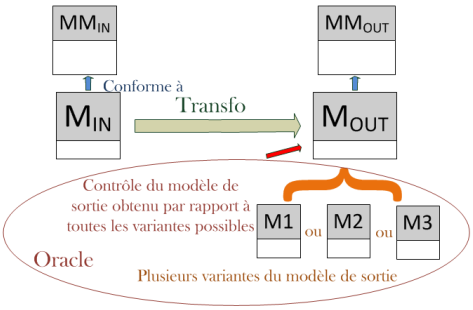


FIGURE 2 – Test d'une transformation de modèles aux sorties polymorphes

ces états.

La figure 3 présente un modèle M^{in}, une machine à états hiérarchique. La transformation de modèles **T1** met à plat une machine à états telle que celle-ci. **T1** supprime les états composites, pour les remplacer par des états simples. La machine obtenue après transformation doit être équivalente sémantiquement à celle d'entrée. Cette transformation de modèles est *endogène* : les méta-modèles d'entrée et de sortie sont les mêmes.

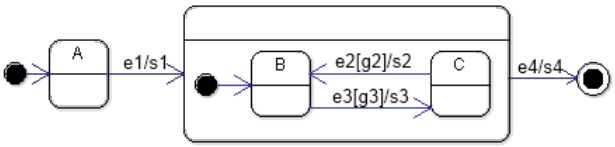
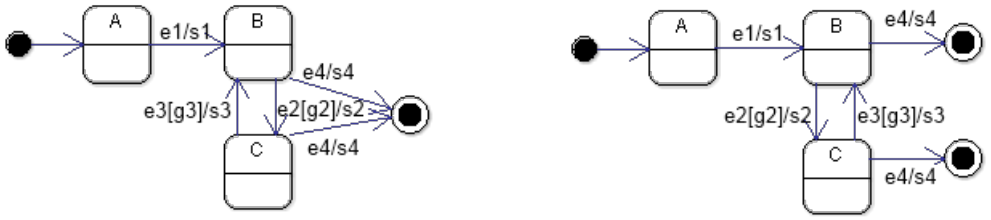


FIGURE 3 – Exemple M^{in}, de machine à états hiérarchique

Cette transformation produit des modèles de sorties polymorphes, plusieurs automates différents peuvent correspondre à la mise à plat d'un même modèle d'entrée. Par exemple, les figures 4(a) et 4(b) présentent les modèles M_1^{out} et M_2^{out}, deux résultats possibles de l'application de la transformation de modèles **T1** sur M^{in}. L'état composite a été remplacé par les états simples de la sous-machine qu'il contenait. La transition quittant A rejoint B, qui était lié à l'état initial de la sous-machine. Les états B et C deviennent les nouvelles sources de la transition sortant de l'état composite rejoignant l'état final. Cette transition est donc dupliquée. Comme la norme UML [UML11] ne précise pas qu'une machine à états ne doit avoir qu'un seul état final, les modèles M_1^{out} et M_2^{out} diffèrent par leur nombre d'états finaux.

Une machine à états hiérarchique peut être bien plus complexe que cet exemple. Un état composite peut contenir plusieurs sous-machines parallèles et de multiples sorties. La transformation proposée par [HAB09] mettant à plat d'une telle machine peut entraîner une explosion combinatoire du nombre d'états dans le modèle résultat.

Pour définir l'oracle du test de la transformation de modèles **T1** pour la donnée de test M^{in}, le testeur devra définir les deux modèles résultats présentés figures 4(a) et 4(b). L'oracle sera alors en charge de contrôler que le modèle de sortie obtenu soit identique à l'un d'eux, en les comparant. Avec notre proposition, le testeur n'aura besoin que de définir un seul modèle de sortie de référence.



(a) Variante M_1^{out} avec un seul état final (b) Variante M_2^{out} avec deux états finaux

FIGURE 4 – Résultats possibles de l'application de **T1** sur M^{in}

Diagramme d'activités vers code CSP

Le diagramme d'activités de la norme UML, est utilisé pour représenter graphiquement le comportement d'une méthode. CSP est une algèbre de processus permettant de modéliser l'interaction de systèmes. La transformation de modèles **T2** proposée par Bisztray et al. [BEH07] transforme un diagramme d'activités en un modèle représentant un programme décrit en CSP.

Nous avons identifié deux règles de la transformation **T2** produisant des variations du modèle de sortie (qui en devient polymorphe) :

1. Un branchement conditionnel devient une condition n-aire. Les différents opérandes peuvent être permutées et différents branchements peuvent se combiner de différentes manières. Les auteurs précisent que d'après la norme UML, si les conditions des gardes sont disjointes, différentes organisations syntaxiques sont sémantiquement équivalentes.
2. Une barre de synchronisation devient une combinaison d'opérateurs de concurrence : l'opérateur de concurrence indique que plusieurs processus sont parallélisés. L'ordre dans lequel ces processus parallèles sont présentés ne change pas la sémantique du programme.

La figure 5 présente un exemple de diagramme d'activités possédant un branchement conditionnel avec trois choix différents, et une barre de synchronisation. La transformation du branchement conditionnel recevant l'arc S4, peut produire deux résultats différents :

- `S4 = M1` $\not<$*nohelp*$\not>$[1] `(D2` $\not<$*askhelp*$\not>$ `D1a)` ou
- `S4 = D2` $\not<$*askhelp*$\not>$ `(M1` $\not<$*nohelp*$\not>$ `D1a)`

La transformation de la barre de synchronisation se transforme en trois processus parallèles `(F1, F2, F3)`, qui peuvent s'organiser de six manières différentes :

- `D3 = F1 ‖ F2 ‖ F3`
- `D3 = F1 ‖ F3 ‖ F2`
- `D3 = F2 ‖ F1 ‖ F3`
- `D3 = F2 ‖ F3 ‖ F1`
- `D3 = F3 ‖ F1 ‖ F2`
- `D3 = F3 ‖ F2 ‖ F1`

Dans cet exemple, deux éléments du modèle d'entrée introduisent des variations possibles en sortie. Pour définir les modèles de sortie possibles, il faut combiner ces deux possibilités de variations. La transformation du branchement conditionnel peut produire deux résultats, et celle de la barre de synchronisation six, ce qui porte à douze le nombre de modèles de sortie valides. Pour définir l'oracle du test de cette transformation de modèles pour ce modèle d'entrée, le testeur devra définir les douze modèles de référence valides. L'oracle ainsi défini devra contrôler

1. Ici *nohelp* est l'expression d'une condition ayant comme opérandes M1 et une autre condition

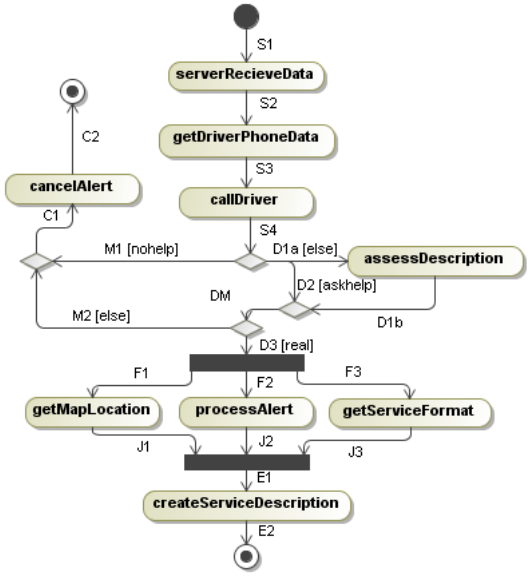


FIGURE 5 – Exemple de diagramme d'activités

si le modèle de sortie obtenu fait bien partie de ces douze variantes, en le comparant avec chacune.

Ces cas d'étude montrent que le test d'une transformation de modèles peut nécessiter de définir plusieurs modèles de sortie de référence pour une unique donnée de test.

3 Utilisation de la comparaison de modèles filtrée

Nous introduisons une nouvelle méthode pour définir un oracle pour le test d'une transformation de modèles aux sorties polymorphes. Nous proposons d'utiliser une comparaison de modèles filtrée dont l'avantage est de n'utiliser qu'un seul modèle de référence par test.

Dans cette section, nous présentons le principe de notre méthode pour contrôler la partie commune d'un modèle polymorphe, puis nous présentons comment des *patterns* permettent de contrôler la partie variable de l'approche.

3.1 Principe de la comparaison de modèles filtrée

Notre hypothèse est qu'un modèle polymorphe comporte une partie commune à toutes les variantes. C'est le cas des deux cas d'étude présentés section 2.3. Dans la mise à plat de l'automate nous avons identifié qu'en dehors des états finaux le reste des modèles de sortie est commun, dans la transformation d'UML vers CSP, à part les conditionnelles et les synchronisations le reste des modèles CSP est commun. Si les variantes d'un modèle de sortie impliquent des variations dans l'ensemble du modèle, nous revenons à l'utilisation des oracles présentés section 2.2.

Une fonction d'oracle ne considérant que cette partie commune produit un *verdict partiel*. Le verdict ne dira pas si la totalité du modèle est exacte mais seulement d'une partie et par conséquent seule une partie de la transformation est vérifiée. Un verdict négatif indique bien

FIGURE 6 – Élément de filtrage pour la comparaison partielle d'un oracle de la mise à plat d'une machine à états

la présence d'une erreur qu'il faudra diagnostiquer. Un verdict positif n'indique pas l'absence d'erreur mais annonce qu'aucune erreur n'a été observée dans la partie considérée. Pris unitairement chaque test peut paraître incomplet mais c'est l'ensemble des tests qui importe. Comme d'une manière générale le test ne peut être exhaustif, ces verdicts partiels positifs apportent quand même de l'information utile.

Nous restreignons le contrôle d'un oracle sur la partie commune d'un modèle polymorphe en proposant une fonction de comparaison filtrée :

$$\texttt{filteredComparison}(M^{out}, M^{ref}, \texttt{filteringElements}) : \texttt{boolean}$$

Elle prend trois paramètres qui pour l'oracle du test de transformation de modèles sont :
- M^{out} : le modèle obtenu par la transformation du modèle de test.
- M^{ref} : le modèle de référence attendu pour la transformation du modèle de test.
- `filteringElements` : les éléments de filtrage exprimés sous la forme d'un fragment du métamodèle de sortie de la transformation de modèles.

Elle renvoie un résultat de la comparaison binaire : égal ou différent. L'information supplémentaire sous-jacente est égal/différent *modulo le filtrage effectué*.

L'exécution de la fonction `filteredComparison`(M^{out}, M^{ref}, `filteringElements`) se déroule en trois étapes : (i) M^{ref} est comparé avec M^{out} pour obtenir leurs différences, et pas uniquement un résultat binaire (égal/différent), (ii) ces différences sont filtrées pour retirer les éléments de résultat qui ne concernent pas la partie commune, (iii) le verdict est produit : positif si les différences filtrées sont vides, négatif sinon.

Pour paramétrer le filtrage, le testeur doit définir quels éléments des modèles de sortie ne doivent pas être pris en compte. Dans cette proposition, nous ne souhaitons pas définir ces éléments rejetés au niveau du modèle, car cela nécessiterait de les identifier pour chaque test, correspondant à chaque modèle de test. Nous proposons au testeur de définir les éléments à rejeter en fonction du métamodèle de sortie de la transformation. Le paramètre `filteringElements` est un fragment du métamodèle de sortie, contenant les méta-classes dont les instances appartiennent à la partie non commune des modèles de sortie valides. C'est le testeur qui exprime ces `filteringElements` en fonction de son interprétation de la spécification.

Dans l'exemple de la mise à plat d'une machine à état, la partie non commune est constituée des états finaux ainsi que des transitions dont ils sont la cible. Ici, l'élément de filtrage sera la méta-classe Transition reliée par l'association target à la méta-classe FinalState, comme le présente la figure 6. De cette manière les états finaux et leurs transitions n'influencent plus le résultat de la comparaison. Le testeur sait si la partie commune des modèles de sortie est exacte ou pas, il a obtenu un verdict partiel pour le test de la transformation de M^{in}.

3.2 Complétion du verdict partiel avec des *model snippets*

Pour compléter le verdict, il faut que l'oracle contrôle également la partie variable d'un modèle polymorphe. Nous proposons d'utiliser la fonction d'oracle qui repose sur la technique

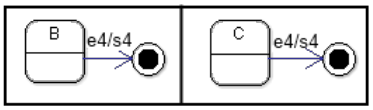


FIGURE 7 – Expression de la partie non commune des modèles de sortie de notre cas d'étude par des *model snippets*

du *pattern matching* présentée dans [MBLT10] pour le contrôle de la seule partie variable du modèle de sortie.

Cette oracle est paramétré par le modèle de sortie à contrôler et par des *patterns* faits de *model snippets*. Introduits par Ramos et al. [RBJ07], les model snippets sont des modèles partiels au sens où ils sont partiellement conformes à un métamodèle. Ils sont conformes à une version relâchée du métamodèle dont les contraintes sont supprimées : les cardinalités 1 sont changées en 0..1, les 1..* en 0..*, les invariants et valeurs par défaut des propriétés sont supprimés. Le framework associé permet de générer ce métamodèle relâché et de vérifier comment les model snippets correspondent avec des modèles. Le pattern définit quels attributs des models snippets peuvent servir de variables pour consulter les résultats de l'opération de pattern matching [MBLT10]. Pour l'oracle de transformation de modèles, le métamodèle relâché est le métamodèle de sortie et les modèles devant correspondre avec les patterns sont les modèles de sorties à contrôler.

Cette fonction d'oracle peut être utilisée en complément de celle que l'on propose. Le testeur peut représenter dans un pattern la partie non commune d'un modèle de sortie polymorphe. En fonction de son interprétation de la spécification, le testeur peut définir un pattern et donc un oracle pour chaque variante, ou définir (s'il existe) un pattern pour toutes les variantes. Dans la transformation de mise à plat d'une machine à états, nous pouvons caractériser la partie non commune des modèles de sortie présentés figures 4(a) et 4(b) de la manière suivante : ≪*Les états B et C doivent être liés à un état final quel qu'il soit*≫. La figure 7 présente cette caractérisation exprimée à l'aide de *model snippets*.

4 Implantation et mise en pratique de notre proposition

Dans cette section, nous présentons la mise en application de notre méthode de définition d'un oracle partiel pour le test d'une transformation de modèles aux sorties polymorphes. Nous détaillons dans un premier temps la manière dont nous l'avons implantée, pour ensuite revenir sur l'application à nos cas d'étude.

4.1 Mise en œuvre de notre approche

Pour mettre en œuvre notre approche, nous avons utilisé l'environnement EMF d'Eclipse. Nous avons fait ce choix car cet environnement est couramment utilisé dans le monde académique et il offre un grand nombre d'outils. Parmi les outils de comparaison de modèles disponibles, nous avons utilisé EMF Compare [BP08], qui est intégré à l'environnement EMF, et correspond aux principes exposés par Cicchetti et al. [CDRP07]. Il fourni le résultat de la comparaison sous la forme d'un modèle que nous pouvons exploiter selon nos besoins. Il ne permet pas de restreindre la comparaison à seulement certains concepts du métamodèle, ce qui nous conduit à proposer une approche par filtrage de la différence. Pour mettre en œuvre la transformation

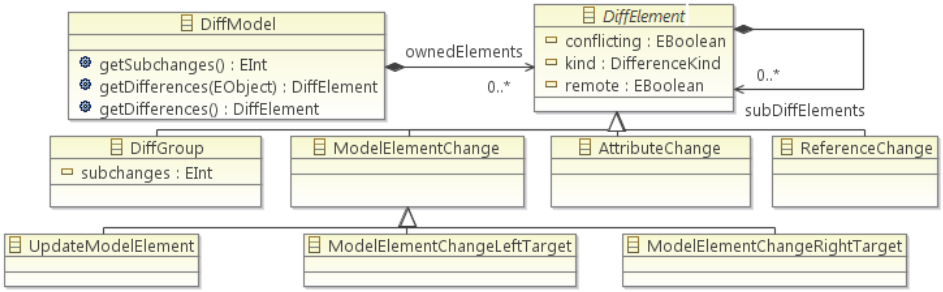


FIGURE 8 – Une partie du métamodèle de différence d'EMF Compare

de modèles de mise à plat d'une machine à état nous avons utilisé UML2 [2], une implantation du méta-modèle UML basée sur l'environnement EMF. Pour utiliser EMFCompare et mettre en œuvre le filtrage du résultat de la comparaison de modèles nous avons utilisé Java.

Mise en œuvre du filtrage

EMF Compare fournit le résultat d'une comparaison de modèles sous la forme de deux modèles, un modèle de correspondances (`MatchModel`) et un modèle de différences (`DiffModel`). Dans le cadre de notre utilisation pour l'oracle de tests de transformations de modèles, nous n'avons pas besoin de connaître le détail des points communs entre les modèles comparés, nous nous intéressons au détail des différences observées, s'il y en a. Pour établir le verdict partiel à l'issue de la comparaison de modèles filtrée, nous vérifions s'il existe des différences entre les deux modèles sur la partie commune aux modèles de sortie valides.

La figure 8 présente une partie du métamodèle de différence d'EMF Compare. Un modèle de différences est constitué d'un ensemble d'éléments de différence (`DiffElement`). Un élément de différence peut simplement regrouper plusieurs éléments correspondant à une même unité (même package, même modèle, même état composite ou autre). Il peut aussi représenter :

- un changement au niveau d'un élément des modèles (élément modifié ou présent uniquement d'un côté).
- la modification d'un attribut d'un élément des modèles comparés.
- la modification d'une référence d'un élément à un autre, par exemple la cible d'une transition modifiée d'un modèle à l'autre.

Notre fonction `filteredComparison(`M^{out}`,` M^{ref}`, filteringElements)` fait appel à EMF-Compare, pour comparer M^{out} et M^{ref}. Elle réalise ensuite le filtrage du modèle de différences résultant de la comparaison de modèles de M^{out} et M^{ref}, nous détaillons ce processus ci-dessous. A l'issue de ce filtrage nous vérifions si des différences sont observées sur la partie commune des modèles de sortie. Dans ce cas, l'oracle produit un verdict négatif, positif sinon.

Le principe de notre méthode de filtrage du résultat d'une comparaison de modèles est identique quelle que soit la transformation de modèles que nous testons. Nous avons donc écrit une classe Java `Filter` dans laquelle nous avons implanté notre méthode. Elle possède un attribut `elements` qui correspond aux éléments de filtrage, exprimés sous la forme d'un ensemble de méta-classes. Un élément du résultat de la comparaison de modèles concernant un élément du modèle doit être filtré si cet élément correspond à l'un des éléments de filtrage. Le résultat du filtrage est un ensemble de `DiffElements`, qui contient les différences observées sur la partie commune des modèles de sortie valides.

2. http ://www.eclipse.org/uml2/

Pour effectuer le filtrage, nous parcourons les éléments du résultat de la comparaison de modèles. Les `DiffElement` étant organisés de façon hiérarchique, le parcours est récursif. Ils sont ensuite traités de la manière suivante :

case (DiffElement) in
`DiffGroup` → parcourir les éléments de différence qu'il contient.
`ModelElementChange` → vérifier si l'élément du modèle concerné doit être filtré.
`AttributeChange` → vérifier si l'élément du modèle qui contient cet attribut doit être filtré.
`ReferenceChange` → vérifier si l'élément du modèle qui contient cette référence doit être filtré.
fin case

Nous n'ajoutons le `DiffElement` traité au résultat que dans les trois derniers cas, si la réponse est négative.

Environnement d'exécution

Pour procéder à nos expérimentations, il nous a fallu mettre en place un environnement pour l'exécution des transformations étudiées et celle de leur tests. Notre objectif a été de proposer une application simple à utiliser. Nous avons défini nos propres méthodes pour interfacer EMFCompare, ainsi le testeur n'a pas à se préoccuper de la manière dont EMFCompare fonctionne.

Nous avons créé une classe Java `Comparator`, pour prendre en charge l'utilisation d'EMFCompare pour réaliser la comparaison entre deux modèles. Cette classe est le point d'entrée de notre solution, elle possède plusieurs attributs dont l'emplacement des modèles à comparer mais aussi une instance de `Filter` utilisée pour filtrer le résultat de la comparaison de modèles. Par défaut, une instance de `Comparator` ne permet de comparer que des métamodèles au format `ecore`. Pour comparer d'autres modèles il est nécessaire d'enregistrer manuellement le métamodèle auquel ils correspondent. Cet enregistrement est effectué grâce à la méthode `registerPackage(package)`. La fonction `filteredComparison()` réalise la comparaison puis le filtrage du résultat. Elle retourne vrai s'il n'y a plus de différence observée après filtrage, faux sinon.

4.2 Application

Nous avons appliqué notre proposition aux cas d'étude de la section 2.3.

Mise à plat d'une machine à état UML

Nous avons utilisé l'implantation de la transformation de modèles **T1** de mise à plat d'une machine à états UML réalisée par Hölt et al. [HAB09]. Les modèles ont été créés à l'aide de UML2Tools [3], un outil permettant l'édition graphique de modèles UML.

Nous avons spécifié les éléments à filtrer : les états finaux et les transitions qui en ont un pour cible. Nous avons utilisé le cas de test suivant :

- Le modèle de test est celui présenté figure 3.
- L'oracle utilise comme paramètre de la fonction filteredComparison() le modèle de référence de la figure 4(b) et le pattern fait de model snippets de la figure 6.

3. http ://www.eclipse.org/modeling/mdt/ ?project=uml2tools

Le modèle de sortie issu de l'exécution de la transformation de modèles est celui de la figure 4(a). Avant filtrage deux différences étaient identifiées : l'état final supplémentaire, et le fait que la transition sortant de C ait une cible différente. Après filtrage, plus aucune différence n'était observée, le test n'a pas échoué.

Transformation d'un diagramme d'activités en un programme CSP

Nous avons également mis en pratique notre proposition sur notre second cas d'étude. N'ayant pas accès à une implantation de la transformation, nous avons simulé son fonctionnement. Nous avons créé le métamodèle de CSP proposé dans [BEH07], au format ecore. Puis, nous avons créé un modèle de sortie correspondant à la transformation de l'exemple figure 5, ainsi que des variations valides ou non de celui-ci. Nous avons défini les éléments à filtrer : les `Concurrency`, les `Condition` et les `ProcessExpression` qui leur sont associés.

Comme nous l'avons précisé, pour pouvoir comparer deux modèles, il est nécessaire d'enregistrer manuellement leur métamodèle. Plus précisément, le package de leur métamodèle doit être ajouté au registre des packages. Pour procéder à cet enregistrement, nous devons disposer d'une instanciation Java du métamodèle. Si pour le métamodèle UML elle existe déjà, nous avons dû la créer pour le métamodèle de CSP. Après avoir créé le métamodèle, nous avons défini un fichier `genmodel` qui s'est chargé de générer une implantation.

Nous avons défini plusieurs cas de test : Pour le premier, le modèle de sortie contrôlé et celui de référence sont deux variantes valides, qui diffèrent au niveau des deux points de variation. Dans ce cas, quatre différences étaient observées avant filtrage :

- Inversion des noms de deux processus parallèles (2 différences) ;
- Inversion des conditions (2 différences).

Après filtrage, plus aucune différence n'était observée, le test n'a pas échoué.

Pour le second cas de test, nous avons modifié le nom d'un processus non parallèle qui n'intervient pas dans une condition. Avant filtrage, nous observions deux différences supplémentaires, qui étaient toujours présentes après. Le test a échoué.

5 Travaux connexes

Plusieurs travaux portent sur la vérification de la correction de transformations de modèles, elle peut se faire de différentes manières, dont le test. Là où nous nous intéressons à la définition d'oracles pour le test d'une transformation de modèles, d'autres étudient la sélection de données d'entrée pour les cas de test. Küster et al. [KAER07], proposent trois techniques en boîte blanche pour construire les cas de test. Une transformation de modèles peut être vue comme une transformation de graphes, dont Darabos et al. [DPV08] ont étudié le test. Ils ont notamment identifié et catégorisé les fautes généralement présentes dans des transformations erronées.

Nous avons précédemment proposé l'utilisation de contrats génériques pour l'oracle. Pour vérifier une transformation de modèles,Cariou et al. [CBBD09] utilisent aussi des contrats qui comportent des contraintes (i) sur le modèle d'entrée, (ii) sur le modèle de sortie, (iii) sur l'évolution des éléments entre le modèle d'entrée et le modèle de sortie. Braga et al. [BMC+], représentent la spécification d'une transformation de modèles à l'aide d'un métamodèle, dont une instance est une exécution particulière de cette transformation. Pour vérifier cette transformation ils utilisent des contrats exprimés sous la forme d'un métamodèle de transformation et un ensemble de propriétés sur celui-ci. De façon similaire, Büttner et al. [BCG11] proposent de modéliser une transformation de modèles ATL, pour la valider. Ils regroupent tous les éléments

des métamodèles et de l'implantation ATL de la transformation en un modèle qu'ils valident ensuite.

Narayanan et al. [NK08a] considèrent que si le modèle de sortie obtenu après application de la transformation est correct, il existe une correspondance vérifiable entre le modèle d'entrée et le modèle de sortie. Pour l'établir et la vérifier il est nécessaire de pouvoir spécifier la correction du modèle de sortie. Ils ont ainsi dressé le cahier des charges pour un langage permettant de spécifier cette correction [NK08b].

D'autres ont étudié la vérification formelle d'une transformation de modèles. C'est le cas de Giese et al. [GGL+06], qui spécifient des transformations de modèles à l'aide d'une grammaire de graphes, puis les valident par des techniques de *model checking*. Ces techniques sont aussi utilisées par Rensink et al. [RSV04] qui comparent deux approches de la vérification d'une transformation de graphes.

Enfin, des techniques de preuve peuvent également être utilisées pour vérifier formellement une transformation de modèles. Schätz [Sch10] cherchent à vérifier une transformation de modèles à l'aide d'Isabelle [4], un assistant de preuve. Dans leur approche, les modèles sont formalisé et l'implantation de la transformation est effectuée avec des règles Prolog.

6 Conclusion

Nous avons identifié la problématique de la construction d'oracles pour le test de transformations de modèles aux sorties polymorphes. Les approches actuelles ne permettent pas de tester efficacement de telles transformations de modèles. Nous avons introduit une nouvelle approche qui considère séparément la partie commune à toutes les variantes d'un modèle de sortie polymorphe et la partie non commune. Nous proposons une comparaison de modèles filtrée, pour contrôler la correction de la partie commune d'un modèle de sortie obtenu. A l'issue de cette étape, nous fournissons un verdict partiel pour le test. Nous proposons de réutiliser dans un deuxième temps un oracle utilisant du *pattern matching* pour contrôler la partie non commune. Pour la suite de nos travaux notre premier objectif est de tester complètement nos cas d'étude et de mesurer le gain, que nous avons expliqué ici, en terme d'efficacité vis-à-vis d'autres oracles. Cette comparaison pourra être faite en mesurant le taux de détection d'erreurs, en encore le nombre d'oracles nécessaires, leurs tailles. Nous souhaiterions ensuite intégrer les contrôles des parties communes et variables dans une fonction d'oracle globale. Cela permettrait au testeur d'écrire un seul oracle et d'obtenir un verdict complet. Une autre piste de recherche envisagée est l'assistance du testeur dans l'écriture de ces oracles, uniquement basé aujourd'hui sur sa connaissance de la transformation

Références

[BCG11] Fabian Büttner, Jordi Cabot, and Martin Gogolla. On validation of atl transformation rules by transformation models. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa, pages 9 :1–9 :8, 2011.

[BEH07] D. Bisztray, K. Ehrig, and R. Heckel. Case study : Uml to csp transformation. *Applications of Graph Transformation with Industrial Relevance (AGTIVE)*, 2007.

[BMC+] Christiano Braga, Roberto Menezes, Thiago Comicio, Cassio Santos, and Edson Landim. On the specification, verification and implementation of model transformations with transformation contracts. In *Formal Methods, Foundations and Applications*, Lecture Notes in Computer Science.

4. http ://www.cl.cam.ac.uk/research/hvg/isabelle/

[BP08] C. Brun and A. Pierantonio. Model differences in the eclipse modelling framework. *UPGRADE, The European J. for the Informatics Professional*, 9, 2008.

[CBBD09] Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam. Ocl contracts for the verification of model transformations. *ECEASST*, 24, 2009.

[CDRP07] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9) :165–185, 2007.

[DPV08] Andrea Darabos, András Pataricza, and Dániel Varró. Towards testing the implementation of graph transformations. *Electron. Notes Theor. Comput. Sci.*, 211 :75–85, April 2008.

[GGL+06] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner. Towards verified model transformations. *MoDeV²a : Model Development, Validation and Verification.*, page 78, 2006.

[HAB09] Nina Elisabeth Holt, Erik Arisholm, and Lionel Briand. Technical report 2009-06 : An eclipse plug-in for the flattening of concurrency and hierarchy in uml state machines. Technical Report 2009-06, 2009.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. volume 21, pages 666–677, New York, NY, USA, August 1978. ACM.

[KAER07] Jochen Küster and Mohamed Abd-El-Razik. Validation of model transformations – first experiences using a white box approach. In Thomas Kühne, editor, *Models in Software Engineering*, volume 4364 of *Lecture Notes in Computer Science*, pages 193–204. Springer Berlin / Heidelberg, 2007.

[MBLT08] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Model transformation testing : oracle issue. In *MoDeVVa workshop colocated with ICST'08.*, pages 105–112, Lillehammer, Norway, April 2008.

[MBLT10] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Construction de tests qualifiés de transformation de modèles. *Technique et Science Informatiques, special issue L'Ingénierie dirigée par les modèles*, 29 :537–569, 2010.

[MS03] Faron Moller and Scott A. Smolka. On the computational complexity of bisimulation, redux. In *Proceedings of the Paris C. Kanellakis memorial workshop on Principles of computing & knowledge : Paris C. Kanellakis memorial workshop on the occasion of his 50th birthday*, PCK50, pages 55–59, 2003.

[NK08a] A. Narayanan and G. Karsai. Verifying model transformations by structural correspondence. *Electronic Communications of the EASST*, 10(0), 2008.

[NK08b] Anantha Narayanan and Gabor Karsai. Specifying the correctness properties of model transformations. In *Proceedings of the third international workshop on Graph and model transformations*, GRaMoT '08, pages 45–52, New York, NY, USA, 2008. ACM.

[RBJ07] Rodrigo Ramos, Olivier Barais, and Jean-Marc Jézéquel. Matching model-snippets. In *In Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07)*, pages 121–135, Nashville, TN, USA, October 2007.

[RSV04] Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations : A comparison of two approaches. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, pages 219–222. Springer Berlin / Heidelberg, 2004.

[SBM09] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic model generation strategies for model transformation testing. In *International Conference on Model Transformation, ICMT'09.*, pages 148–164, Zurich, Switzerland, July 2009.

[Sch10] B. Schätz. Verification of model transformations. *Electronic Communications of the EASST*, 29(0), 2010.

[UML11] OMG UML. 2.0 specification. *URL http ://www.omg.org/spec/UML/*, 2011.