

Nom : Antoine Toulmé
Responsable pédagogique UTT :
Marc Lemercier

Spécialité : Master RACOR
Année : 2006

Développement d'un moteur de comparaison de modèles

En informatique, les modèles sont très utilisés pour représenter des concepts. Ceux-ci sont implémentés au moyen de langages de programmation. Les utilisateurs de modeleurs ont besoin des mêmes outils que les développeurs lors de développements de modèles. Ce rapport présente les travaux réalisés pour les outils de comparaison de modèles. Il discutera les choix faits par les moteurs existants. Il présentera leur mode de fonctionnement actuel et proposera une alternative, basée sur la théorie de l'information. Cette alternative sera développée en s'appuyant sur la sémantique du modèle pour définir les types de différence pouvant intervenir entre deux éléments d'un modèle. Il présentera les résultats de cette approche en s'appuyant sur des tests et une analyse qualitative. Le rapport est conclué par une discussion des améliorations à apporter au dispositif pour certains modèles.

Entreprise : Intalio

Lieu : Paris, France

Responsable : Ismael Chang Ghalimi

Recherche appliquée
Services marchands
Génie logiciel
Analyse des données



I. Introduction

Un modèle est une représentation d'un objet. Il permet de conceptualiser ce dernier, en en faisant ressortir certaines caractéristiques. Les modèles sont très utilisés dans le monde de l'informatique pour construire des applications. On distingue des modèles de données, statiques, et des modèles de processus, dynamiques, représentant des activités.

Les modèles sont des ressources critiques en informatique, et sont souvent soumis à des modifications par différents intervenants. Dans une perspective d'exploitation industrielle, les utilisateurs ont besoin d'identifier les changements entre modèles, éventuellement de les modifier, d'accéder à l'historique de leurs changements. L'ensemble de ces opérations peut être effectué au moyen d'un moteur de comparaison entre modèles.

Les comparateurs textuels permettent déjà de comparer deux documents. Les comparateurs de modèles sont encore aux balbutiements. Une première approche de la comparaison de modèles UML existe, elle compare efficacement les éléments en se basant sur leur identifiant.

Ce rapport propose une approche différente, basée sur une comparaison de tous les attributs d'un élément. Il présentera les bases sur lesquelles elle s'appuie, et discutera les approches précédentes.

Il décrira ensuite le processus complet de la comparaison de modèles en utilisant cette approche. Il présentera notamment la comparaison entre deux éléments, et le système de notation qui calculera la distance les séparant.

Ce rapport sera conclué par un tour d'horizon du travail réalisé, et présentera brièvement des idées pour consolider cette approche.

II. État de l'art

Le modèle et l'approche orientée modèle

Un modèle est défini¹ comme une « vue de l'esprit » analytique et algorithmique représentant des phénomènes et leurs relations. Il vise à la compréhension et au diagnostic. Les systèmes informatiques sont représentés par des modèles.

Le développement orienté modèle est brièvement abordé par l'article de Mark Kofman[KOF03]. Ce type de développement est centré sur le développement et la gestion des modèles. L'OMG [OMG] a présenté l'idée du MDA (Model Driven Architecture) [MDA06], qui consiste à développer un modèle indépendant des contraintes liées à la plateforme d'implémentation, ce qui permet d'avoir une plus grande stabilité et une maintenance plus aisée du système.

L'OMG est une fondation dont le but est la promotion des approches objet dans l'industrie. Cet organisme a défini le MOF (Meta-Object Facility) [MOF00]. Cette approche définit un méta-modèle commun à tous les méta-modèles. Dans une architecture à quatre niveaux, le MOF se décrit comme le niveau méta-méta, qui permet de décrire les niveau méta (par exemple, le méta-modèle UML). Ceux-ci définissent les modèles UML. Ceux-ci décrivent eux-même une quantité d'information.

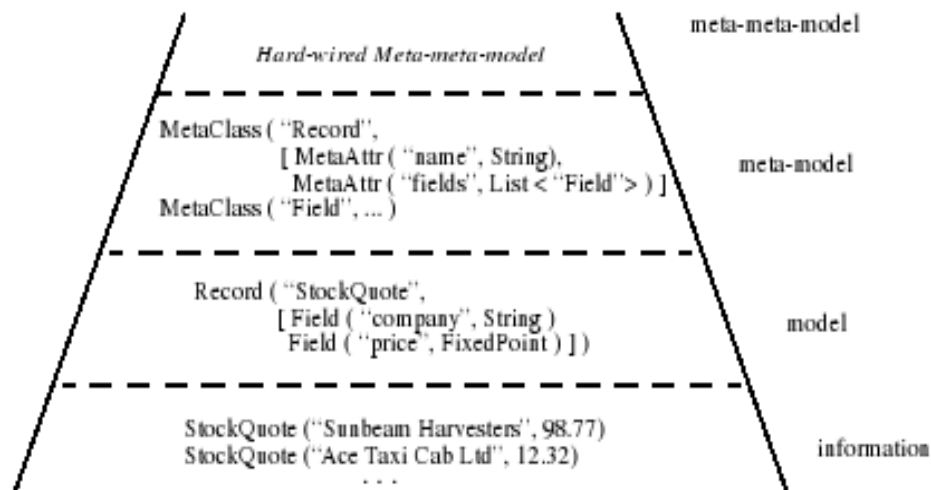


Illustration 1: L'architecture du MOF

Le MOF est donc une plateforme qui permet de décrire n'importe quel méta-

1 Source : Wikipedia

modèle. Une de ses implémentations est EMF (Eclipse Modeling Framework) [EMF], que nous utiliserons au cours de notre expérimentation.

Le MOF s'appuie sur la spécification XMI, (XML Metadata Interchange) [XMI05] pour permettre l'échange d'informations et le stockage des modèles sous forme sérialisée. Chaque élément de chaque modèle est sérialisé en lui attribuant un identifiant.

Caractéristiques pour le management des modèles

Kofman [KOF03] fait remarquer que les modèles ne sont pas subdivisables en unités de sens comme le sont les artefacts créés par les langages de programmation. Une grande quantité d'information est contenue dans une seule unité de modélisation.

Cette concentration d'information pose problème. Il est nécessaire de comparer certains fragments du modèle, de les réviser au besoin, de fusionner des modèles ensemble.

Les modèles sont représentés en utilisant des notations graphiques ; les différences graphiques ne sont pas nécessairement des différences du modèle. Contrairement à un texte, un modèle peut changer l'ordre de ses éléments, sans que cela crée forcément une différence sémantique. L'ordre des éléments doit être pris en compte comme une différence possible entre les deux éléments.

Les modèles sont constitués d'éléments. Les éléments sont reliés entre eux par des liens de contenance, d'évènement, etc. Les modèles issus du MOF sont organisés pour pouvoir être parcourus en arbres.

Les éléments sont composés d'attributs. Les attributs sont des objets contenant une clé unique (« name », « isAbstract ») assimilée au nom de l'attribut et une valeur. La valeur d'un attribut est typée, elle peut être une chaîne de caractères, un nombre, un booléen.

Marcus Alanen [ALA03] décrit les différents types de différences pouvant être relevés entre modèles. Il distingue d'une part les différences affectant l'élément tout entier, d'autre part les différences s'appliquant aux attributs.

Les éléments peuvent être supprimés ou ajoutés.

Un élément est considéré comme modifié quand les attributs qui le composent ont changé. La différence « set » décrit la différence entre les valeurs de deux attributs, les différences « insert » et « remove » décrivent l'ajout et la suppression d'un attribut. Enfin, les différences « InsertAt » et « RemoveAt » sont associés à l'ajout ou la suppression d'un attribut si l'élément ordonne les attributs qui le composent.

L'usage d'un moteur de comparaison

Le moteur de comparaison textuel de la plateforme Eclipse [ECL] est utilisé pour deux usages :

Comparer deux modèles : le moteur de comparaison doit pouvoir comparer deux modèles, et établir les différences entre ces deux modèles.

Comparer deux modèles avec leur ancêtre commun : le moteur de comparaison compare les deux modèles, en utilisant un ancêtre commun pour repérer les changements qui ont été effectués de part et d'autre. Les changements ayant affectés le même élément sont déclarés conflictuels, ils peuvent éventuellement être déclarés comme pseudo-conflictuels si les valeurs des deux membres sont les mêmes. (par exemple, un élément présent dans l'ancêtre commun a été supprimé dans les deux membres de la comparaison).

Discussion

Un ensemble de modèles hermétiques

Les moteurs de comparaison d'Alanen [ALA03] et de Letkeman [LET05] s'appuient sur la comparaison des identifiants des éléments pour les relier, puis calculent les différences entre ces éléments.

Plusieurs scénarios montrent que cette approche est invalide, quoique certainement la plus simple à mettre en oeuvre.

Un premier scénario consiste à ajouter à deux copies du même modèle deux éléments en tous points semblables à l'aide des éditeurs associés au moteur. Ces éditeurs vont générer leurs identifiants. Les comparateurs mis en avant par Letkeman et Alanen déclarent alors que deux différences sont relevés, correspondant à l'ajout de chaque élément. Les identifiants des éléments sont les seules données différentes.

Un deuxième scénario consiste à comparer deux modèles développés indépendamment. Comme les identifiants des deux modèles sont différents, le comparateur rapporte une erreur et la comparaison ne peut avoir lieu. Ce scénario est bien sur un cas particulier du premier scénario.

Ces problèmes représentent de grandes limitations, les modèles ne peuvent être comparés qu'entre eux. Ils forment des ensembles hermétiques, dont l'exportation et la réutilisation sont malaisées.

Une approche trop éloignée du modèle logique

Le modèle d'Alanen avance des types de différences qui tiennent compte de l'ordre des attributs qui forment un élément, mais pas de l'ordre des éléments

eux-mêmes. Cette approche est maladroite, en cela que l'ordre des attributs formant un élément varie peu et a peu de signification sémantique, tandis qu'il convient d'accorder de l'importance à l'ordre des éléments entre eux, surtout dans un modèle ordonné.

Mieux s'approcher de la sémantique du modèle

Opérations de refactoring

Le refactoring est un processus emprunté à la programmation, qui consiste à renommer un élément sans changer son comportement².

Le refactoring est utilisé pendant le développement orienté modèle pour restructurer un diagramme. Les opérations de refactoring ont des impacts sur les objets utilisant les éléments renommés, comme type ou comme parent.

Les comparateurs abordés utilisent les identifiants pour éviter de relever des différences invalides. Il faut proposer un mécanisme de prise en charge des opérations de refactoring qui prenne en compte les changements dans les objets dépendant de l'objet renommé.

Appuyer la démarche sur la théorie de l'information

Pour distinguer un élément d'un autre, il faut trouver les composantes sémantiques véhiculant un maximum d'information, et permettant ainsi d'identifier l'élément.

La théorie de l'information a été fondée en grande partie par Claude Shannon [SHA48]. Elle propose de calculer la quantité d'information liée à un mot m par la formule :

$$I(m) = \log_2[p(m)]$$

où $p(m)$ est la probabilité d'apparition du mot m . Plus un mot apparaît, plus il est redondant, moins la quantité d'information qu'il transporte est grande.

² Source : Wikipedia

III. Un moteur de comparaison totale

Hypothèse de comparaison

Nous avons vu que les solutions pour la comparaison de modèles existant sont limitées et hermétiques. Nous désirons implémenter une méthode de comparaison qui montre véritablement les changements logiques du modèle et compare efficacement n'importe quels modèles, sans se baser sur les identifiants des éléments.

Nous proposons donc de recourir à une comparaison totale des attributs des éléments entre eux. Si cette comparaison des attributs deux à deux montre que les valeurs des attributs sont proches, alors on pourra déclarer les éléments proches.

Types de différences

Nous proposons d'utiliser sept types de différences :

Les différences propres aux éléments :

- Création : l'ajout d'un élément
- Suppression : la suppression d'un élément
- Ordre : l'ordre d'un élément est changé
- Renommer : le nom d'un élément a changé. Cette différence est un cas particulier de la différence set. Elle est particulière en cela qu'elle a des répercussions sur les éléments utilisant l'élément renommé comme un type, ou sur les éléments contenus par l'élément renommé.

Les différences propres aux attributs des éléments :

- Set : Cette différence représente le changement de valeur d'un attribut.
- Insert : cette différence représente l'ajout d'un attribut.
- Remove : cette différence représente la suppression d'un attribut.

Les différences Ordre et Renommer ont été ajoutées par rapport aux différences d'Alanen [ALA03]. Les différences InsertAt et RemoveAt ont été supprimées.

Représentation graphique des différences

Dans son article sur MetaDiff [KOF03], Kofman conclue en présentant les perspectives de ces travaux. Parmi celles-ci, il aborde la difficulté à représenter une différence. Le comparateur textuel fourni par la plateforme Eclipse donne un bon point de départ. Pour ce comparateur, une différence s'applique sur une ligne. Le comparateur définit un code de couleurs pour

décrire le type et l'origine de la différence.

Le comparateur textuel présente un arbre de ces différences, avec le même code de couleurs, en utilisant des objets graphiques représentant une addition, une suppression, ou un changement.

L'éditeur de comparaison à construire devra définir ce qu'est une différence pour la représenter. Il ne s'agira plus de lignes mais d'éléments du modèle.

Comparaison d'attributs

Calcul de la distance entre attributs

On appelle distance la différence entre les valeurs de deux attributs. Selon le type de la valeur, cette distance est calculée différemment. La distance a une valeur comprise entre 0 (les attributs ont une valeur égale) et 1 (les attributs ont des valeurs totalement différentes).

Comparaison de booléens

Si les deux booléens ont une valeur identique, la distance vaut 0. Dans le cas contraire, elle vaut 1.

Comparaison de chaînes de caractères

Nous utilisons l'algorithme de calcul de la distance de Levenshtein pour calculer la distance entre les deux chaînes de caractères. L'algorithme renvoie le nombre de manipulations nécessaires pour transformer la chaîne de caractères en celle à laquelle elle est comparée.

Une manipulation de la chaîne de caractères est une des opérations suivantes:

- suppression d'un caractère
- ajout d'un caractère
- modification d'un caractère

La distance est calculée par cette opération :

$$distance(c1, c2) = \frac{(\text{nombre de manipulations de la chaîne de caractères})}{(\max(\text{longueur}(c1, c2)))}$$

Si les deux chaînes de caractères sont complètement dissemblables, la distance maximale est égale à la plus grande longueur des deux chaînes de caractères.

Comparaison de nombres

Les nombres ont une sémantique large. Deux dates ne se comparent pas comme deux entiers. Les nombres sont manipulés par l'utilisateur comme des chaînes de caractères. De ce fait, nous avons préféré les traiter en tant que telles, et appliquer sur elles l'algorithme ci-dessus.

Poids des attributs

Un attribut a un poids dépendant du volume d'informations qu'il transporte (voir plus haut le paragraphe concernant la théorie de l'information).

Le volume d'informations transporté par un attribut est estimé par deux facteurs :

-le nombre de fois où l'attribut est rencontré. Plus un attribut est présent, moins il est important.

-le nombre de valeurs prises par l'attribut par rapport au nombre de fois où il a été rencontré. Si un attribut a toujours la même valeur, alors il n'a pas d'intérêt. Un attribut dont la valeur est différente pour chaque élément identifie l'élément.

Les informations relatives aux attributs sont collectées en parcourant récursivement le modèle. La formule suivante donne le poids de l'attribut :

$$Poids(A) = \frac{1}{\left[\frac{(\text{nombre d'apparitions})}{(\text{nombre total d'éléments})} \right]} \times \frac{(\text{nombre de valeurs prises})}{(\text{nombre d'apparitions})}$$

C'est la multiplication de l'inverse de la probabilité d'apparition de l'attribut par le nombre de valeurs prises en fonction du nombre d'apparitions.

Comparaison de deux éléments**Algorithme de comparaison**

Deux éléments sont comparés en comparant leurs attributs, au moyen de cet algorithme :

```

1*****
2*Function match
3*Input : element, comparedEl t
4*Output : di ff
5*****
6 Body :
7 Di ff di ff <- new Di ff();
8 // we select the two elements' attributes and compare them.
9
10
11 AttributeResul tLi st is a new Li st
12 while (more attributs in element)
13   if (comparedAttribute exists with the same name)
14     int i <- compare_values(attribute, comparedattribute, di ff)
15     attributeResul tLi st.add(attribute,i); // the attribute existed in the
other element and we compared them
16   else
17     attributeResul tLi st.add(attribute,0); // the attribute didn't exist in the
other element, we have a matching rate of 0 on it.
18     di ff <- mi ssi ng_el ement;
19   end i f
20 end whi le
21
22 // finally we watch the other element to see if he has attributes that our

```

```

element hasn' t.
23
24 while (more comparedAtt in comparedElement)
25   if (!attributeResultList.contains(comparedAtt))
26     AttributeResultList.add(comparedAtt, 0);
27   diff <- new_element;
28   end if
29 end while
30
31 // now we just calculate the total matching rate
32 total <- 0;
33 while (more attributeResult in AttributeResultList)
34   attribute_weight <-
35   getAttributeWeightForName(attributeResult.name) // get the weight for the
attribute
36
37   total <- total + attributeResult.value * attribute_weight;
38 end while
39
40 if (rate < 0.2)
41
42 // the rate symbolises the difference between the two elements, it must be
under 20% to pass.
43   diff.potentialmatch is set to true
44 else
45   diff.potentialmatch is set to false
46 end if
47
48 return diff;

```

Illustration 2 : Algorithme de comparaison de deux éléments

Le taux de différence appelé « rate » dans l'algorithme correspond à la distance entre les éléments. Si la distance est inférieure à un seuil dont la valeur est définie arbitrairement, les deux éléments sont considérés comme un couple potentiel.

Poids locaux

Lorsque les deux éléments sont comparés l'un à l'autre, la distance finale doit être comprise entre 0 et 1. Pour cela, les distances entre valeurs des attributs sont conscrites entre 0 et 1, et les poids des attributs sont divisés par la somme des poids des attributs des deux éléments.

L'intérêt de cette manœuvre est de circonscrire l'ensemble des résultats à un petit intervalle, ce qui permet de comparer entre eux les valeurs obtenues et de le transformer aisément en pourcentages.

Comparaison de deux modèles

```

1*****
2*Functi on arrangeCouples
3*Input : children, comparedChildren
4*Output : couples
5*****
6

```

```
7 Body :
8 couples is a new list
9 potential couples is a new list
10 while (more child in children)
11   while (more comparedChild in comparedChildren)
12     Diff diff = match(child, comparedChild);
13     if (diff.potentialmatch is true)
14       // we place the couple and diff in the potential
15       couples list
16       potential couples.add(child, comparedChild, diff);
17     end if
18   end while
19 end while
20
21 // we have all the potential matching couples
22 // we reorder them by the matching rating in increasing order
23
24 sort_on_rating_in_increasing_order(potential couples)
25
26 while(more potential in potential couples)
27   if (!one_member_in(couples))
28     couples.add(potential)
29     differencesList.add(potential.diff) // add the
30     differences found for the couple
31   end if
32 end while
33
34 // we finally take in account the remaining singles
35
36 while (more child in children)
37   if (!in couples)
38     differencesList.add(new Diff(missing_element));
39   end if
40 end while
41
42 while (more child in comparedChildren)
43   if (!in couples)
44     differencesList.add(new Diff(new_element));
45   end if
46 end while
```

Illustration 3 : Algorithme de comparaison de deux modèles

Les modèles sont comparés récursivement. Les éléments de chaque niveau sont comparés avec tous les éléments du même niveau, en utilisant les algorithmes cités plus haut.

Une fois les éléments comparés, ils sont ordonnés suivant leur distance par ordre croissant. Le programme crée alors des couples. Les éléments dont les distances sont les plus petites sont sélectionnés en premier. Les couples potentiels dont un des deux éléments au moins est déjà utilisé par un couple sélectionné sont éliminés.

Les éléments célibataires restants sont ajoutés à la liste des couples en spécifiant que la différence entre les deux éléments est un ajout ou une suppression d'élément (selon si l'élément célibataire est dans le modèle ou dans le modèle comparé).

Les éléments contenus par chaque élément forment le niveau de récursivité

suisant. L'algorithme est répété jusqu'à ce qu'il ait parcouru les deux modèles.

Comparaison à trois

```

1*****
2 *Function threeWayMatch
3 *Input : left, right, ancestor
4 *Output : diff
5*****
6 Body : diff <- new diff();
7 // adapted algorithm from the Differencer class.
8 if (ancestor == null)
9   if (left == null)
10    if (right == null)
11     // nothing
12    else
13     diff <- added element on right;
14   endif
15  else
16   if (right == null)
17    diff <- added element on left;
18   else
19    if (left == right)
20     diff <- added element on left, added element on right, pseudo conflict;
21    else
22     diff <- added element on left, added element on right, conflict;
23   endif
24  endif
25 else
26  // ancestor != null
27  if (left == null)
28   if (right == null)
29    diff <- element deleted on left, element deleted on right,
pseudo_conflict;
30   else
31    if (right == ancestor)
32     // left is null and right == ancestor
33     diff <- element deleted on left;
34    else
35     // left is null and change on right
36     diff <- element deleted on left, match(ancestor, right);
37   endif
38  else
39   if (right == null)
40    // right is null and left is non null
41    if (perfectMatch(left, ancestor))
42     // right is null and left == ancestor
43     diff <- element deleted on right;
44   else
45    // right is null and change on left
46    diff <- element deleted on right, match(ancestor, left);
47   endif
48  else
49   boolean al <- match(ancestor, left) == 0;
50   boolean ar <- match(ancestor, right) == 0;
51   // left and right are non null
52   if (al && ar)
53    // the three EObjects are equal, there is nothing to do.
54   endif
55   if (al && !ar)
56    diff <- match(ancestor, right);

```

```
57     endi f
58     i f (!al && ar)
59         di ff <- match(ancestor, l eft);
60         i f (!al && !ar)
61             // conflict or pseudo conflict
62             leftdi ff<- match(ancestor, l eft);
63             rightdi ff <- match(ancestor, ri ght);
64             di rectdi ff <- match(l eft, ri ght);
65             di ff <- mergeAndTagPseudoConfl icts(l eftDi ff, ri ghtDi ff, di rectDi ff);
66         endi f
67     endi f
68 endi f
69 endi f
70 endi f
71 endi f
```

Illustration 4 : Algorithme de comparaison de modèles avec un ancêtre commun

La comparaison à trois consiste en une comparaison entre deux éléments en utilisant un ancêtre commun. Ce type de comparaison apparaît souvent lors de l'utilisation de systèmes de contrôles de version.

Les comparateurs existant se bornent à utiliser les identifiants pour retrouver les éléments de l'ancêtre, du modèle local, et du modèle comparé.

En s'inspirant de l'algorithme de la classe Differencer [DIF06], qui régit le fonctionnement du comparateur textuel de la plateforme Eclipse, nous avons tracé cet algorithme, qui prend appui sur l'algorithme de comparaison à deux.

Résultats et futurs travaux

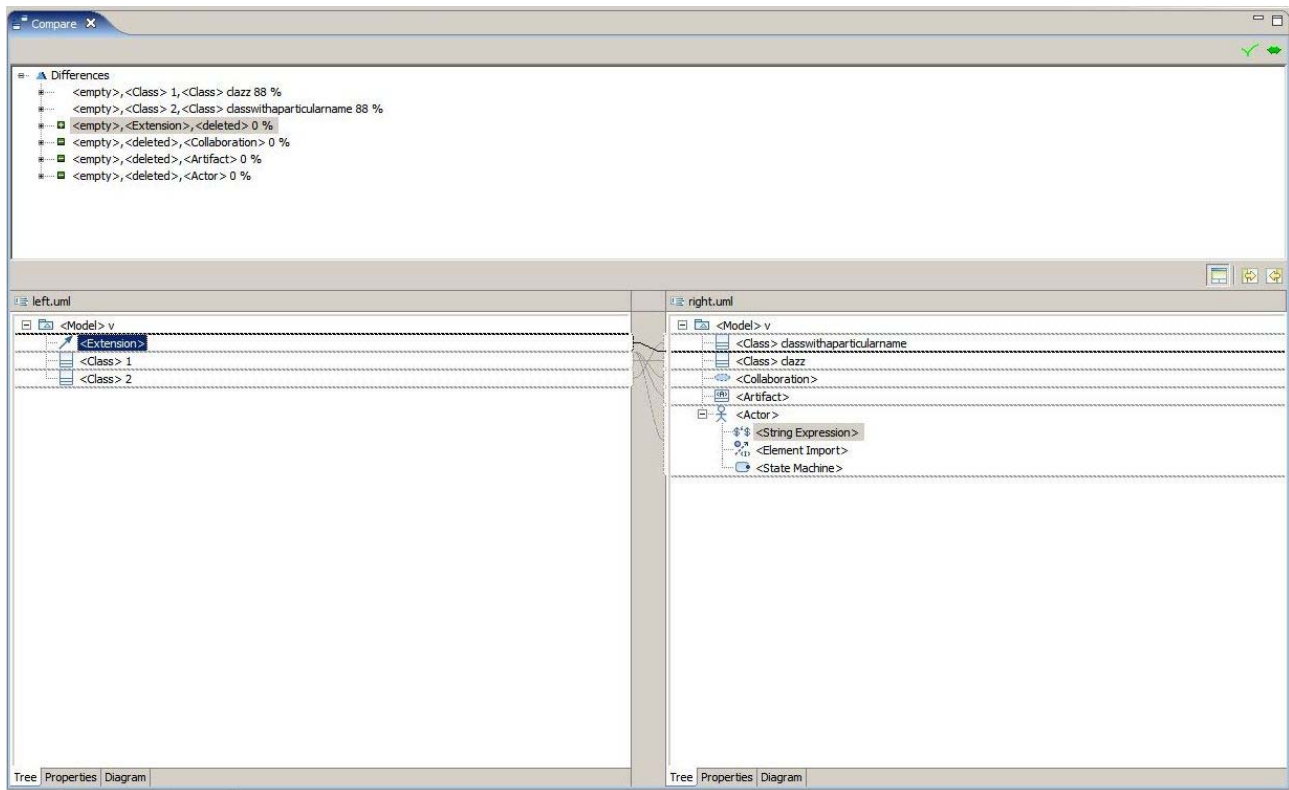


Illustration 1: L'éditeur de comparaison à deux

Illustration 5 : L'éditeur de comparaison de modèles

Les différences sont représentées dans l'arbre en haut. L'ancêtre commun aux deux éléments est représenté au milieu, les deux modèles comparés sont représentés en bas. Les différences sont mises en évidence en tracant des lignes entre les couples.

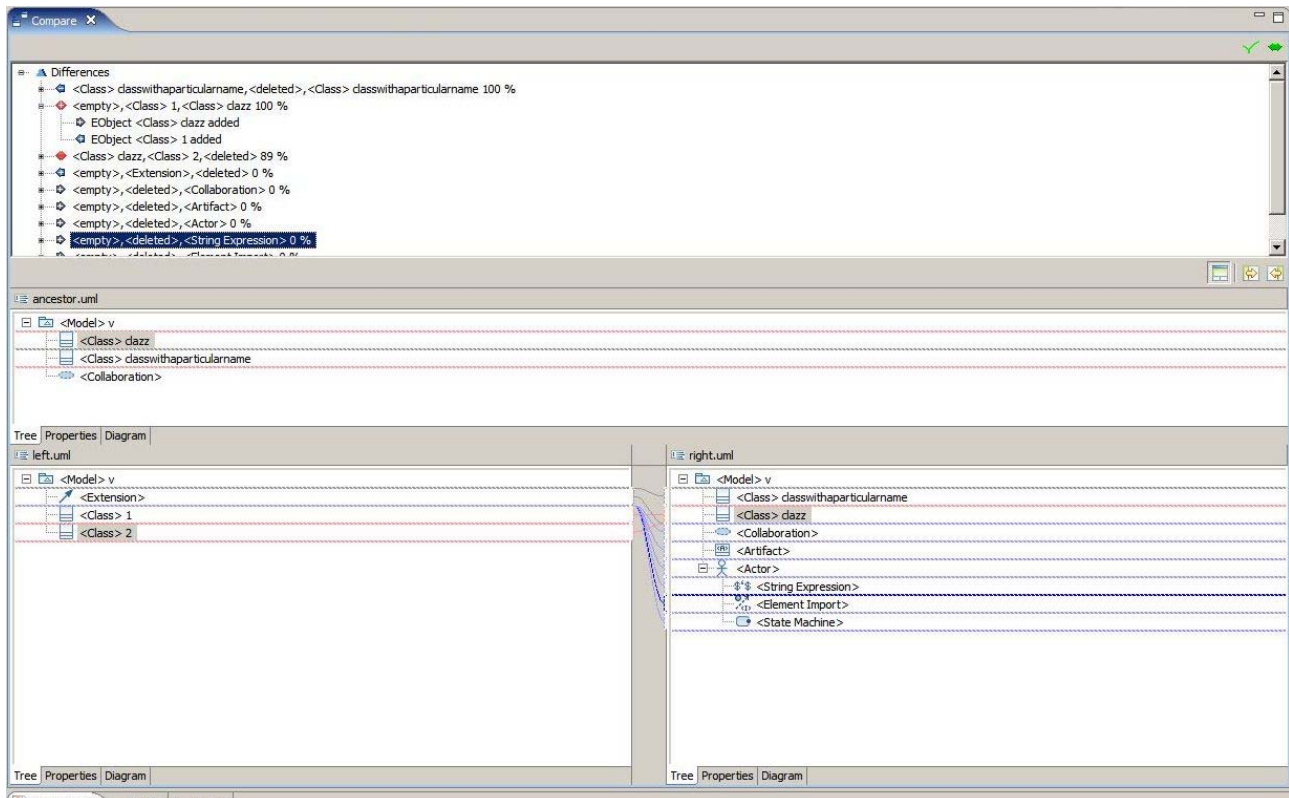


Illustration 2: L'éditeur de comparaison à trois

Illustration 6 : L'éditeur de comparaison avec un ancêtre commun

Analyse par tests unitaires

Le moteur de comparaison a été testé en utilisant des séries de tests unitaires. Chacun de ces tests était constitué de deux modèles qui se différenciaient par une seule différence. Chaque test testait la performance du moteur pour la détection d'un type de différence, parmi les types présentés plus haut.

Les tests ont montré que le moteur reconnaissait chaque type de différence.

Analyse qualitative

Sémantique

Les résultats de la comparaison sont satisfaisants et représentent les véritables différences sémantiques. Le moteur détecte tous les types de différences entre éléments.

Le problème de cette comparaison en arbre est qu'elle ne convient pas à certains modèles qui sont basés sur une approche graphique. Les diagrammes

BPMN [BPMN] sont un exemple de ce type de limitation. Ces modèles ne sont pas transformables en arbres sans perdre des données, et la sémantique d'un diagramme est différente (le modèle est ordonné, il a plusieurs entrées et plusieurs sorties, la suppression d'un lien entre deux éléments peut entraîner des changements en cascade).

Performance

Le moteur n'a pas été construit en réfléchissant en termes de performances, et le nombre de comparaisons entre éléments est très élevé. Les temps de calcul peuvent monter à plusieurs minutes pour deux modèles disposant de cent éléments.

Pour réduire les temps de calcul, nous proposons de retirer des listes des éléments à comparer les éléments qui ont été trouvés parfaitement uniques. De cette façon, nous

Conclusion

Le développement orienté modèle nécessite des outils pour gérer et développer les modèles. La comparaison entre modèles est un de ces outils.

Ce rapport présente une approche remettant en question les travaux sur la comparaison de modèles. Au lieu de se baser sur les identifiants des éléments, l'algorithme de comparaison utilise la théorie de l'information pour analyser la sémantique du modèle. Il établit par la suite une correspondance entre les éléments du modèle en procédant par récursivité.

Nous avons créé un moteur de comparaison de modèles qui implémente cette approche, et qui permet de visualiser les différences, leurs origines et leurs types.

Nos travaux vont se porter sur l'amélioration des performances du moteur, en le rendant apte à traiter de grandes quantités de données. Nous allons également réfléchir à une approche englobant la sémantique de modèles en diagramme, ne pouvant pas être représentés en arbres.

Bibliographie

- [ALA03] ALANEN, Markus and PORRES, Ivan, Difference And Union Of Models, 2003
- [BPMN] OMG, Business Process Modeling Notation Specification, 2004
- [DIF06] IBM, Class Differencer - Eclipse Platform API Specification, 2006, <http://help.eclipse.org/help32/index.jsp?topic=/or>
- [ECL] Eclipse Foundation, Eclipse Project, 2006, <http://www.eclipse.org>
- [EMF] Eclipse Foundation, Eclipse Tools - EMF Home, <http://www.eclipse.org/emf/>
- [KOF03] KOFMAN, Mark, MetaDiff - a Model Comparison Framework, 2003
- [LET05] LETKEMAN, Kim, Comparing and merging UML models in IBM
- [MDA06] OMG, Model Driven Architecture Specification, 2006, <http://www.omg.org/mda/> Rational Software Architect, 2005
- [OMG] Object Management Group, 2006, <http://www.omg.org>
- [SHA48] SHANNON, Claude, A Mathematical Theory of Communication, 1948
- [XMI05] OMG, XMI (XML Metadata Interchange) Specification, 2005