Génération de correctifs pour les modèles partiels d'AnimUML

Mickael Clavreul ESEO
Angers, France
mickael.clavreul@eseo.fr

Frédéric Jouault ESEO
Angers, France
frederic.jouault@eseo.fr

Maxime Méré
STMicroelectronics, INSA
Rennes Le Mans, France
maxime.mere@st.com

Matthias Brun ESEO Angers, France matthias.brun@eseo.fr

Théo Le Calvar IMT Atlantique, LS2N, UMR CNRS 6004, F-44307 Nantes, France theo.le-calvar@imt-atlantique.fr

Matthias Pasquier
ERTOSGENER Angers, France matthias.pasquier@ertosgener.com

Ciprian Teodorov ENSTA Bretagne, Lab-STICC CNRS UMR 6285 Brest, France
ciprian.teodorov@ensta-bretagne.fr

ABSTRACT
Fournir une analyse de modèle à un concepteur lui facilite la tâche de modélisation et permet de gagner en confiance sur la qualité des modèles produits. Cette analyse est d'autant plus pertinente lorsque l'outil est capable de suggérer des correctifs et des améliorations que le concepteur peut appliquer automatiquement. Cependant, le développement de ce type d'outil basé sur l'analyse statique de modèle est chronophage et sujet à l'erreur. Cet article présente une approche basée sur la rétro-propagation d'expressions booléennes inspirées du langage de contraintes OCL pour proposer des correctifs et suggestions correctes par construction. La proposition de correctifs utilise un mécanisme d'annotations pour calculer les choix proposés au concepteur de modèles. Le concept décrit dans l'article a été évalué sur l'outil de conception de modèles partiels AnimUML. Cette implémentation permet de construire automatiquement des messages issus de l'analyse statique d'un modèle, de renvoyer ces messages au concepteur de modèles et de lui proposer des correctifs à appliquer.

Author Keywords
OCL, analyse statique, modèles, suggestion.

INTRODUCTION

L'essor des langages de modélisation s'accompagne d'outils de modélisation textuels ou graphiques pour lesquels l'effort d'apprentissage est non négligeable. Selon cette étude [5], un des défis auxquels les concepteurs font face lorsqu'ils utilisent des outils de modélisation est de localiser, comprendre et corriger leurs erreurs. Les développeurs de ces outils doivent donc fournir des fonctionnalités d'analyse de modèle efficaces et proposer des opérations de correction ou de complétion semi-automatisées.

Cet article est extrait de [4], qui présente un mécanisme de rétro-propagation d'expressions OCL pour faciliter le calcul d'opérations de correction semi-automatisées. Il explique, sur un exemple différent, l'approche de génération de correctifs pour la conception de modèles partiels avec AnimUML. Avec cette approche, les développeurs peuvent définir des règles pour vérifier la construction de leurs modèles en utilisant des invariants OCL. L'écriture des règles permet de calculer automatiquement des options de correction pour les modèles. L'approche permet également de paramétrer la génération des correctifs avec des messages spécifiques et des options de correction fournies par les développeurs d'outils de modélisation. Ces mécanismes ont été évalués sur l'outil de conception de modèles partiels AnimUML [3].

CAS D'ÉTUDE : ANIMUML

Modèles partiels et conception

Le support des modèles partiels, voire incohérents, permet, dans les étapes préliminaires de conception logicielle, d'exécuter et donc de tester des conceptions incomplètes qui pourront être améliorées de manière incrémentale. Bien que cette approche soit répandue en programmation, elle n'est pas autant utilisée en modélisation.

AnimUML [3] est un outil de modélisation sur le Web qui permet de créer des modèles partiels. Alors que les outils UML classiques considèrent les modèles partiels et leurs incohérences comme des erreurs, AnimUML remonte des avertissements aux concepteurs pour les aider à améliorer leur modélisation. Dans ce processus incrémental, proposer des correctifs et suggestions est d'autant plus pertinent.

Méta-modèle

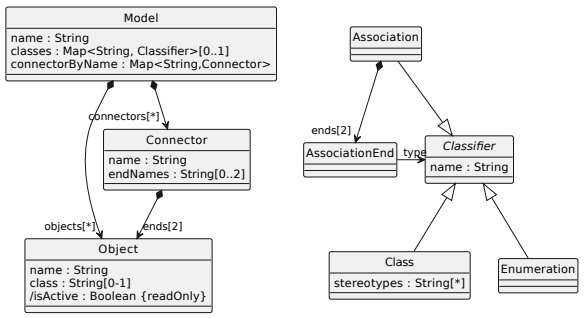

FIGURE 1. Extrait du méta-modèle d'AnimUML

La figure 1 présente un extrait du méta-modèle AnimUML nécessaire à la compréhension de l'exemple de contrainte présentée dans la suite. Ce méta-modèle est différent du standard UML pour notamment autoriser la définition de modèles incomplets. L'élément racine est un *Model*, qui contient une liste d'*Object*s "objects" et un dictionnaire de *Classifier*s "classes" indexé par leurs noms. Chaque *Object* a un nom (chaîne de caractères) et éventuellement un nom (chaîne de caractères) de *Class*. Dans un modèle partiel, la *Class* d'un *Object* peut ne pas être présente. Le nom de cette *Class* pourra être résolu plus tard grâce au dictionnaire "classes".
Un *Model* contient des *Connector*s dont les extrémités correspondent à des Objects. Un *Connector* est une représentation, entre deux *Object*s, d'une "instance" d'une *Association* entre deux *Class*es composée de deux extrémités (*AssociationEnd*).

Contraintes

Un exemple d'invariant OCL est présenté en Listing 1. Un seul invariant est défini pour le type *Connector* (lignes 1 à 7) à partir de la conjonction de trois expressions. Les deux premières expressions (lignes 4 et 5) vérifient que les extrémités du *Connector*, c'est-à-dire les objets, sont typées avec des classes présentes dans le modèle. Cette vérification utilise les deux variables OCL déclarées aux lignes 2 et 3. La troisième expression (lignes 6 à 7) vérifie qu'il existe une *Association* entre les classes correspondant aux objets situés aux extrémités du *Connector*.

```
1 context Connector inv:
2   let class_end0 : Classifier = model.classes.get(self.ends->get(0).class) in
3   let class_end1 : Classifier = model.classes.get(self.ends->get(1).class) in
4   not class_end0.oclIsUndefined()
5   and not class_end1.oclIsUndefined()
6   and model.classes->select((clname,cl) | cl.oclIsTypeOf(Association)
7    and (cl.ends->get(0).type = class_end0 and cl.ends->get(1).type = class_end1))
       ->notEmpty()
```

Listing 1. Invariant sur les connecteurs entre objets

EXPLICATIONS DE L'APPROCHE

La génération de correctifs utilise une technique de rétro-propagation similaire à [2] basée sur [1]. Lorsqu'un invariant est évalué à faux, nous identifions les changements nécessaires, à rebours, pour satisfaire les expressions constituant l'invariant. Certaines expressions nécessitent des informations supplémentaires qui sont fournies par le développeur. Plutôt que d'appliquer automatiquement les changements comme dans [2], nous proposons une liste des changements à faire pour satisfaire les expressions contenues dans l'invariant. Le développeur décide des correctifs à appliquer ou bien un algorithme automatique peut être exécuté.

La définition des correctifs présentés au concepteur de modèles est fournie par le développeur de l'invariant grâce à un mécanisme d'annotations incluses dans le code OCL (voir Listing 2). Le développeur peut définir la sévérité du problème et éventuellement, lorsque cela est possible, le correctif à appliquer. Ces informations sont utilisées pour générer les correctifs affichés au concepteur et pour définir l'action à effectuer lors de l'application du correctif.

```
1 context Connector inv:
2   let class_end0 : Classifier = model.classes.get(self.ends->get(0).class) in
3   let class_end1 : Classifier = model.classes.get(self.ends->get(1).class) in
4   not class_end0.oclIsUndefined()       -- @severity: warning
5   and not class_end1.oclIsUndefined()   -- @severity: warning
6   and model.classes->select((clname,cl) | cl.oclIsTypeOf(Association)
7    and (cl.ends->get(0).type = class_end0 and cl->ends.get(1).type = class_end1))
       ->notEmpty()   -- @severity: warning, @value: Association.newInstance().
       refSetValue('ends',Set{AssociationEnd.newInstance().refSetValue('type',
       class_end0},AssociationEnd.newInstance().refSetValue('type',class_end1))
```

Listing 2. Invariant avec définition des correctifs

VALIDATION PRÉLIMINAIRE

La validation de l'approche est proposée par l'implémentation d'opérateurs similaires à ceux d'OCL sous la forme de fonctions JavaScript opérant sur des modèles AnimUML pour un sous-ensemble de règles de construction de modèles. Les annotations sont passées en paramètres des fonctions JavaScript pour permettre la génération des correctifs. L'outil actuel est capable de générer des messages automatiquement à partir des expressions OCL, mais ceux-ci sont moins compréhensibles que ceux fournis par les développeurs (sous la forme d'annotations supplémentaires non présentes sur le Listing 2). Un extrait des correctifs suggérés au concepteur de modèles lorsque l'invariant n'est pas satisfait est présenté sur la figure 2 et la figure 3 après l'application des premiers correctifs.

- Static analysis (auto-fix)
  - Errors:
  - Warnings:
    1. Class Bank of object bank not found.
       - Create class Bank.
    2. Class ATM of object atm not found.
       - Create class ATM.
    3. No type is specified for connector connector_bank_atm between bank and atm.

FIGURE 2. Exemple de correctifs suggérés par AnimUML (Étape 1)

- Static analysis (auto-fix)
  - Errors:
  - Warnings:
    1. No type is specified for connector connector_bank_atm between bank and atm.
       - Create new association.

FIGURE 3. Exemple de correctifs suggérés par AnimUML (Étape 2)

La figure 4 illustre l'exemple de diagramme d'objets UML correspondant à la contrainte présentée en Listing 1 et la figure 5 présente le diagramme de classes incomplet du système avec l'association créée par le correctif.

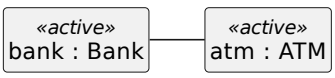

FIGURE 4. Diagramme d'objets initial avec un connecteur

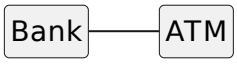

FIGURE 5. Diagramme de classes après application des correctifs

CONCLUSION

Cet article résume la génération de correctifs basée sur l'évaluation et la rétro-propagation d'invariants OCL présentée dans [4]. Elle utilise un mécanisme d'annotations pour aider les développeurs à implémenter cette fonctionnalité dans leurs outils de modélisation.

REFERENCES

[1] Olivier Beaudoux, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2010. Active operations on collections. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 91–105.

[2] Frédéric Jouault and Olivier Beaudoux. 2015. On the Use of Active Operations for Incremental Bidirectional Evaluation of OCL 15th International Workshop on OCL and Textual Modeling. (2015).

[3] Frédéric Jouault, Valentin Besnard, Théo Le Calvar, Ciprian Teodorov, Matthias Brun, and Jerome Delatour. 2020. Designing, animating, and verifying partial UML models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 211–217.

[4] Frédéric Jouault, Maxime Méré, Matthias Brun, Théo Le Calvar, Matthias Pasquier, and Ciprian Teodorov. 2022. From OCL-based model static analysis to quick fixes. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems : Companion Proceedings*. 889–893.

[5] Parsa Pourali. 2018. Tooling advances inspired to address observed challenges of developing uml-like models when using modelling tools. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems : Companion Proceedings*. 168–173.