Réalisation d'un module d'abstraction pour les GAL

Yann Thierry-Mieg, Yann.Thierry-Mieg@lip6.fr

Projet pour deux étudiants. Il est recommandé de postuler en binôme avec un étudiant avec qui vous vous entendez bien.

Introduction

L'objectif du stage est de réaliser un plugin Eclipse s'appuyant sur EMF et un langage (GAL – guarded Action Language) existant pour réaliser un refactoring ou transformation model-to-model qui produit un GAL plus abstrait ou au contraire raffiné par rapport à un prédicat.

L'objectif global est d'exploiter ce mécanisme d'abstraction dans une approche de vérification dite CEGAR de raffinement successifs guidés par un contre-exemple. Le stage a pour objectif de battre les performances de l'outil actuel (http://ddd.lip6.fr) sur un benchmark fourni.

Objectifs :

1. Construction de la transformation model-to-model en Java appuyé sur l'API EMF
2. Intégration dans le serveur de build de l'équipe, permettant la mise en ligne d'un update-site pour installer l'outil
3. Mise en place de tests intégrant la chaine : lecture du prédicat, transformation du modèle source, invocation de l'outil de model-checking. Selon l'avancement réalisation d'une boucle CEGAR pleinement fonctionnelle.
4. Documentation de l'outil

Exemple de source cible:

```
GAL :
GAL philo
{
   //variables
   fork[0]=0;
   fork[1]=0;
   phil_0.state=0;
   phil_1.state=0;
   //transitions
   transition t0  [ ( ( phil_0.state == 0 ) && ( fork[0] == 0 ) ) ]
      { phil_0.state = 1;
        fork[0] = 1;    }

   transition t1  [ ( ( phil_0.state == 1 ) && ( fork[1] == 0 ) ) ]
      { phil_0.state = 2;
        fork[1] = 1;    }
```

```
Abstrait par rapport à {fork[0],fork[1]} :
GAL philo
{
   //variables
   phil_0.state=0;
   phil_1.state=0;
   //transitions
   transition t0  [ ( ( phil_0.state == 0 ) && ( True) ) ]
      { phil_0.state = 1;
      }

   transition t1  [ ( ( phil_0.state == 1 ) && ( True ) ) ]
      { phil_0.state = 2;
      }
```

CEGAR par l'exemple :
On veut vérifier que p1 et p2 ne peuvent accéder à la ressource critique en même temps.
On commence par tout abstraire du modèle, sauf l'état de p1 et de p2, mentionnés dans la propriété.
Sur ce modèle abstrait on recherche des comportements qui mènent à la situation redoutée.
Supposons qu'on trouve un contre-exemple (par un outil existant), « p1getLeft, p1getRight, p2getLeft, p2getRight ».
Cette trace du système abstrait amène dans un état (abstrait) qui viole la propriété, mais est-ce une vraie trace du système concret (vu que le système abstrait permet PLUS de comportements que le système concret) ?
On essaie d'exécuter la séquence trouvée sur le modèle de départ (complexité faible, on explore un seul comportement).
On trouve une contradiction, p2getLeft n'est pas franchissable dans le système concret (vu que la fourchette à droite de p1 est déjà ramassée par p1getRight).
On cherche à identifier pourquoi, i.e. quelle variable a été abstraite mais était testée dans le franchissement de p2getLeft.
On trouve que c'est fork1, la fourchette à droite de p1 et à gauche de p2. Donc il ne fallait pas l'abstraire cette fourchette (variable).
On relance une abstraction sur le modèle concret où l'on abstrait tout sauf l'état de p1 et p2, et la fourchette fork1 (donc on un système raffiné vis-à-vis la première abstraction).
Il n'y a pas de contre-exemple dans le modèle abstrait, plus riche que le vrai modèle.
Donc le modèle initial est correct, vu que le sur-ensemble du comportement que l'on vient d'explorer valide le comportement souhaité.
Si à un moment on arrive au contraire à rejouer sur le modèle initial une séquence contre-exemple du modèle abstrait, on s'arrête aussi, le système ne vérifie pas la propriété (et on l'a vérifié sur une abstraction simplifiée du vrai système).
Au pire on finira par explorer un modèle ou rien n'est abstrait.

Pré-requis et apports du projet :

La connaissance de Java est nécessaire, une connaissance du point de vue utilisateur d'Eclipse est souhaitable.

Le stage permettra de découvrir comment définir un plugin Eclipse et manipuler un méta-modèle pour réaliser une opération de transformation. Pour la chaine complète CEGAR, on devra se familiariser avec les principes de base du model-checking. C'est donc une bonne introduction aux méthodes d'analyse formelle des systèmes concurrents.

L'outil sera développé en utilisant des outils classiques du développement collaboratif : gestion de versions (svn), serveur d'intégration continue (teamcity), conventions de codage (nommage, commentaires…), configurations de build (maven), métriques de qualité de code (Sonar) etc… et permettra donc de se familiariser avec ces outils. Un jeu d'exemples de Benchmark existe, on espère une amélioration des performances de l'outil de model-checking sur ces exemples.

Si la réalisation est de bonne qualité, le projet sera intégré dans la plateforme développée au sein de l'équipe Move, et distribué publiquement. Le stage sera encadré par une réunion hebdomadaire.

Références :

XText, un outil pour construire des DSL (domain specific languages) : www.**xtext**.org

Sonar, outil d'analyse de qualité du code : http://www.sonarsource.org/

Teamcity, serveur d'intégration continue : http://www.jetbrains.com/teamcity/

ITS et GAL : http://ddd.lip6.fr/