Premières expériences pour l'édition correcte par construction de modèles

Mounira Kezadri, Marc Pantel

IRIT/Université de Toulouse
ENSEEIHT, 2 rue Charles Camichel,
31071 TOULOUSE CEDEX, FRANCE
Prenom.Nom@enseeiht.fr
http://www.irit.fr/~Prenom.Nom

Résumé. L'objectif de nos travaux est le développement de transformations vérifiées pour la manipulation de modèles dans un atelier de développement qui combine ingénierie dirigée par les modèles et méthodes formelles. Cette contribution présente de premières expériences de spécification et vérification formelles de transformations élémentaires communes à une famille de langages de modélisation. Ces expériences concernent les modèles fonctionnels de type "flots de données" semblables à SIMULINK ou SCADE. Le langage et les transformations sont spécifiés et vérifiés dans l'assistant de preuve COQ de manière générique en prenant comme paramètre les parties de la sémantique spécifiques à chaque langage.

1 Introduction

La question de la transformation de modèles se situe au centre de l'approche IDM, see **?**. Au sein de cette approche, le processus de développement est vu comme une transformation progressive d'un modèle. Aujourd'hui, bien qu'on constate un grand savoir-faire méthodologique dans le domaine de l'IDM, des questions liées à la sûreté et à la correction de transformations restent posées. Nous présentons dans cet article notre expérience dans l'utilisation d'un assistant de preuve pour vérifier la correction de transformations. Les transformations considérées peuvent être exploitées dans toute phase de développement qui repose sur des langages de modélisation dont la sémantique peut s'exprimer dans le cadre générique proposé. Il s'agit de transformations relativement simples mais qui exigent d'effectuer à nouveau les activités de vérification des modèles concernés si elles ne sont pas elles-même vérifiées. La première étape de notre travail consiste à trouver un bon niveau d'abstraction pour représenter un modèle pour la famille des langages flots de données, see **??**, puis spécifier une transformation représentative des opérations d'édition de modèles hiérarchiques et enfin formaliser celui-ci dans l'assistant de preuve `Coq` pour vérifier la terminaison et prouver par la suite la correction de la transformation.

2 Abstraction des langages considérés

Les langages synchrones flot de données tels que LUSTRE **?** et SIGNAL **?** ont été introduits pour la spécification de systèmes temps-réel critiques. Ils ont été créés dans l'objectif de construire un langage de programmation proche des modèles mathématiques utilisés par exemple dans les systèmes embarqués tels les équations sur les flots de données. Le point commun entre ces langages est le fait qu'ils structurent le système sous forme d'un réseau de processus, autrement dit un ensemble de boites noires composées en parallèle et qui échangent les données via des connexions prédéfinies. Un processus est soit un processus élémentaire (générateur) ou bien un réseau construit à l'aide d'opérateurs de connexion appliqués aux processus existants. Nous nous sommes basés sur ceci pour construire notre abstraction du langage.

La syntaxe abstraite du langage est donnée par le type `bloc` :

B	: :=	$\mathrm{B}_1 \mid\mid \mathrm{B}_2$	(Parallele)
	|	var $\mathrm{x}_1\mathrm{x}_2...\mathrm{x}_\mathrm{n}$ in B	(BlocESV)
	|	b	(BlocES)

Cette abstraction est constituée de trois constructeurs :

`Parallele` est la composition de blocs. Nous ne considérons pas l'ordonnancement entre composants et nous supposons que tous les composants sont en parallèle.

`BlocESV` est un `bloc` hiérarchique avec entrées, sorties et variables locales. Il correspond à un ensemble de composants regroupés dans un seul composant. Ces composants sont inter-connectés entres eux et ces interactions représentent les variables locales du `bloc` englobant.

`BlocES` est un `bloc` spécifiant une contrainte entre ses entrées et ses sorties. Il correspond à un composant qui à partir de ses entrées fait un certain nombre de traitements et génère des sorties. C'est le composant le plus élémentaire.

La sémantique de ce langage est un ensemble de règles paramétrées par `BBSem` (la sémantique des blocs élémentaires). La notation `interp D Sem BBSem B` signifie que `Sem` est une interprétation du bloc `B` dans le domaine de valeurs `D` (ce paramètre de type est exploité dans le type des autres paramètres en COQ). La règle `interpBES` donne la sémantique d'un bloc élémentaire. `Sem` est une interprétation pour un bloc élémentaire si elle est équivalente à l'interprétation donnée par `BBSem` pour le même bloc dans le domaine de valeurs `D`. Cette forme générique a été choisie pour assurer la couverture d'un grand nombre de langage. Les transformations vérifiées pourront ensuite être appliquées sur tous les langages dont la sémantique pourra être exprimée sous cette forme.

$$\frac{BBSem\ name(b)\ Sem}{Interp\ D\ Sem\ BBSem\ b}\ interpBES$$

Les deux règles `interpBESVNil` et `interpBESVCons` permettent de traiter le cas d'un bloc avec des variables locales. `Sem` est une interprétation pour un `BlocESV` avec une liste vide de variables locales si `Sem` est une interprétation de son bloc `B` interne.

$$\frac{Interp\ D\ Sem\ BBSem\ B}{Interp\ D\ Sem\ BBSem\ (\mathrm{var}\ in\ B)}\ interpBESVNil$$

`Sem` est une interprétation pour un bloc `BlocESV` avec une liste non vide de variables locales, s'il existe une valeur que l'interprétation peut donner pour la première variable et qui rend l'interprétation valable pour le reste du bloc.

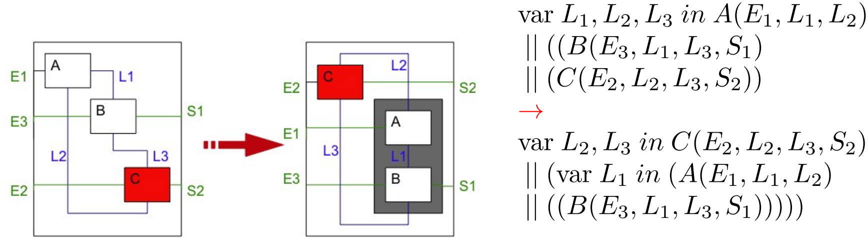
$$\frac{\exists v. Interp\ D\ (Sem(x_1 := v))\ BBSem\ (\text{var}\ x_2 ... x_n\ in\ B)}{Interp\ D\ Sem\ BBSem\ (\text{var}\ x_1 x_2 ... x_n\ in\ B)}\ interpBESVCons$$

Une interprétation `Sem` pour le premier et le deuxième paramètre d'un bloc parallèle est une interprétation pour le bloc `Parallele`.

$$\frac{Interp\ D\ Sem\ BBSem\ B1 \wedge Interp\ D\ Sem\ BBSem\ B2}{Interp\ D\ Sem\ BBSem\ (B1\ ||\ B2)}\ interpPar$$

3 Règles de transformation

Transformations utilisateurs Pour illustrer notre démarche, nous présentons l'exemple de la transformation "déplacement d'un `bloc`" qui consiste à extraire un composant particulier du système en le faisant sortir de son système englobant pour le placer dans un composant à part en conservant ses points d'interactions avec les autres composants. La Figure 3 montre la transformation `Move` du `bloc C`.



La transformation move C

Transformations élémentaires Nous définissons un ensemble de transformations élémentaires (`swap`, `ShiftLeft`, `ShiftRight`, `appl1`, `appl2`, `dropVar` et `keepVar`). Ces transformations de base peuvent être composées de différentes manières pour obtenir des transformations utilisateur.

La transformation `swap` permet d'inverser l'ordre des deux arguments d'un `bloc Parallele`.

swap $(B1\ ||\ B2) = (B2\ ||\ B1)$

Les deux transformations `shiftRight` et `shiftLeft` permettent de changer l'ordre dans un `bloc Parallele` contenant un autre `bloc Parallele` imbriqué.

shiftRight $(B1\ ||\ B2)\ ||\ B3 = B1\ ||\ (B2\ ||\ B3)$

shiftLeft $B1\ ||\ (B2\ ||\ B3) = (B1\ ||\ B2)\ ||\ B3$

Les deux fonctions `appl1` et `appl2` permettent dans l'ordre d'appliquer une fonction `f` sur le premier et le deuxième argument d'un `bloc Parallele`.

appl1 $f\ (B1\ ||\ B2) = (f\ B1\ ||\ B2)$

appl2 $f\ (B1\ ||\ B2) = (B1\ ||\ f\ B2)$

La fonction `dropVar` permet de descendre une variable x_1 si elle n'est pas dans les variables libres du `bloc B1` (si $x_1 \notin$ (Free-Vars B1)).
dropVar var $x_1 x_2 ... x_n$ in $(B1 \,||\, B2)$ $=$ var $x_2 ... x_n$ in $(B1 \,||$ var x_1 in $B2)$

La fonction `keepVar` permet de mettre x_1 à la dernière position si elle est dans les variables libres du `bloc B1` (si $x_1 \in$ (Free-Vars B1)).
keepVar var $x_1 x_2 ... x_n$ in $(B1 \,||\, B2)$ $=$ var $x_2 ... x_n x_1$ in $(B1 \,||\, B2)$

La fonction `applyInside` permet d'appliquer une fonction `f` sur le `bloc` interne.
applyInside f var $x_1 x_2 ... x_n$ in B $=$ var $x_1 x_2 ... x_n$ in f B

Nous avons cité l'ensemble des transformations élémentaires que nous avons utilisées pour notre exemple, mais cet ensemble peut être complété pour couvrir d'autres types de transformations.

4 Formalisation en COQ

La spécification, la vérification et les preuves de correction de transformation en COQ constituent plus de 1600 lignes de code. Nos sources COQ sont directement accessibles à l'adresse (`http://www.irit.fr/~Mounira.Kezadri/Code/`). L'abstraction du langage est formalisée en COQ sous forme d'un type inductif avec ses trois constructeurs `BlocES`, `BlocESV` et `Parallele`. Nous avons ainsi développé plusieurs transformations et vérifications qui ne sont pas présentées ici mais sont accessibles avec notre code COQ.

Correction et vérification En utilisant la sémantique déjà définie, nous introduisons la notion sémantique classique de transformation `correcte`. Une fonction de transformation est correcte si elle préserve la sémantique de la fonction. En d'autres termes, si n'importe quelle interprétation du bloc avant transformation est également une interprétation du bloc après transformation alors celle-ci est correcte.

$$\frac{(\text{Interp D Sem BBSem b}) \leftrightarrow (\text{Interp D Sem BBSem (f b)})}{\text{correct f}}\ correct$$

Nous avons utilisé la fonction `correct` pour démontrer que toutes nos fonctions de transformation sont correctes. Nous avons commencé par démontrer la correction des fonctions élémentaires. Nous avons ensuite défini et démontré un jeu de règles de déduction compositionnelles portant sur le prédicat correct afin de faciliter l'écriture des preuves de correction de transformations plus complexes. Citons par exemple la règle `comCorrect` qui exprime que la composition de deux fonctions `f1` et `f2` correctes l'est aussi.

$$\frac{(\text{correct f1}) \wedge (\text{correct f2})}{\text{correct } (f1 \circ f2)}\ comCorrect$$

Et enfin, nous avons démontré que la transformation utilisateur 3 est correcte par composition de transformations élémentaires prouvées correctes.

5 Conclusion

Nous avons prouvé avec COQ la terminaison et la correction sémantique de nos fonctions de transformations d'éditions. Nous avons présenté dans cet article un exemple de transformation. La prochaine étape est la vérification de la correction vis-à-vis de la spécification (vérifier si la transformation fait exactement ce quelle est censée faire). Le type de transformations d'édition considérées dans cette contribution sera ensuite exploitée pour donner une sémantique de substitution à un modèle de composant générique. Ceci s'inscrit dans la formalisation de composants et de leur vérification et permettra d'étudier les contraintes qui doivent être posées sur la sémantique par rapport aux possibilités de vérification compositionnelle.

6 Remerciements

Ces travaux ont été financés partiellement par l'Agence Nationale pour la Recherche, programme Arpège, Projet SPaCIFY. Ce travail était le fruit d'un stage du master qui se poursuit pour des travaux de thèse sur la prise en compte de la sémantique dans les composants et leur vérification. Le premier auteur adresse ses sincères remerciements à Mamoun Filali pour ses conseils de réflexion modulaire, Jean-Paul Bodeveix pour ses idées et son savoir faire incontournable en COQ et Martin Strecker pour sa disponibilité, ses idées et son soutien sans limite. Le rapport du master qui présente plus de détails est accessible à l'adresse `http://www.irit.fr/~Mounira.Kezadri/Publications/M2RRapport.html`.

Summary

This work targets the development of verified transformations for model edition in a workshop that combines model driven engineering and formal methods. It describes the first experiments conducted for the formal specification and verification of common elementary transformations that can be applied to a family of modeling languages (functional data-flow modeling languages related to SIMULINK or SCADE). The language and the transformations were specified and verified using the COQ proof assistant in a generic manner taking the semantics of each specific language as a parameter.