Spécification et vérification de patrons de propriétés pour des langages dédiés

Faiez Zalila, Xavier Crégut et Marc Pantel

IRIT - Université de Toulouse - France,
`Firstname.Lastname@enseeiht.fr`.

RÉSUMÉ. *L'ingénierie dirigée par les modèles joue un rôle essentiel dans la construction des systèmes critiques. Elle permet d'une part d'offrir aux différents experts intervenant dans le développement d'un système d'exploiter le langage dédié le plus adapté à son expertise, et d'autre part d'introduire des technologies de validation et de vérification le plus tôt possible dans le cycle de développement. Le fait de disposer d'un grand nombre de langages dédiés pose par contre un problème de coût de construction des outils d'exploitation (édition graphique et textuelle, V & V, génération de documentation ou de code, transformation de modèles, ...) de ces langages. Les techniques génératives associées à la métamodélisation permettent de réduire significativement ces coûts. Les travaux présentés concernent les outils de vérification exhaustive de la conformité du comportement d'un modèle aux exigences des utilisateurs. Nous proposons une approche pour définir des patrons de propriété au niveau du langage utilisateur et générer automatiquement les propriétés qui doivent être satisfaites par les modèles de vérification.*

MOTS-CLÉS : *Vérification de modèles, Métamodélisation, Patrons de propriétés*

1. Introduction

L'ingénierie dirigée par les modèles joue un rôle essentiel dans la construction de systèmes critiques. Son rôle est fondamental dans la construction des systèmes les plus complexes tels l'A380 ou le B787 : de tels systèmes ont été entièrement conçus, validés et vérifiés en s'appuyant sur des modèles, et connaissent des vols d'essais sans aucun accident grave, ce qui était inconcevable précédemment. L'IDM permet d'une part d'offrir aux différents experts intervenant dans les développement d'un système d'exploiter le langage dédié le plus adapté à leur expertise, et d'autre part d'introduire des technologies de validation et de vérification le plus tôt possible dans le cycle de développement.

En revanche, disposer d'un grand nombre de langages dédiés pose un problème de coût de construction des outils d'exploitation (édition graphique et textuelle, V & V, génération de documentation ou de code, transformation de modèle...) de ces langages. Les techniques génératives associées à la métamodélisation permettent de réduire significativement ces coûts.

Nous allons présenter une méthode de construction d'outils de vérification exhaustive de la conformité du comportement dynamique des modèles à partir d'exigences explicites des utilisateurs qui s'appuie sur les outils génératifs usuels de l'IDM.

Pour fixer les idées, nous prendrons comme exemple de DSML (Domain Specific Modelling Language) un langage très simplifié de description de processus que nous appelons SimplePDL (éléments en noir sur fig. 1). Il permet de décrire un processus (*Process*) composé d'activités (*WorkDefinition*) et de contraintes d'ordonnancement entre ces activités (*WorkSequence*).

2. Vérification de modèles

Nous souhaitons pouvoir vérifier les modèles construits à partir d'un DSML exécutable. Un DSML exécutable est un DSML dont les modèles pourront évoluer au cours du temps. Nous avons proposé un patron de métamodélisation pour structurer les informations nécessaires à la définition d'une sémantique d'exécution de tels DSML [COM 08a]. Nous nous limitons à une sémantique de type « événements discrets ». De manière schématique, il s'agit de décrire les informations statiques usuelles du DSML, les informations dynamiques qui seront gérées à l'exécution, ainsi que les interactions avec l'environnement représentées sous la forme d'événement pouvant être regroupés en scénarios. La Figure 1 présente en rouge les informations dynamiques à ajouter sur SimplePDL : l'état d'une activité, son taux d'achèvement, un seuil sur les dépendances. Les événements ne sont pas représentés. Ils correspondent à démarrer ou terminer une activité ou un processus ou à modifier le taux d'avancement d'une activité ou le seuil d'une dépendance.

Si le patron décrit les informations nécessaires pour exprimer une sémantique d'exécution, on a ensuite le choix entre l'exprimer sous la forme d'une sémantique opérationnelle (ceci revient à écrire une machine virtuelle pour exécuter les modèles

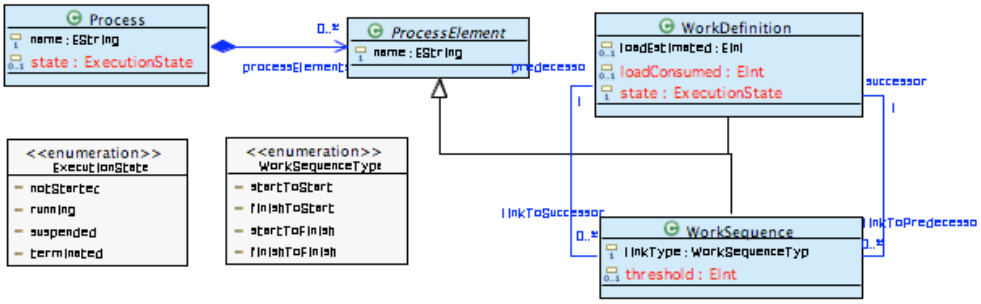

Figure 1. *Métamodèle de SimplePDL avec les informations statiques et dynamiques*

dont les événements constituent le langage d'action) ou de définir une traduction vers un langage formel, la traduction donnant donc la sémantique du modèle d'origine (sémantique de traduction).

Dans les deux cas, plusieurs sémantiques peuvent être définies qui nécessiteront des informations différentes. Par exemple, si l'objectif est simplement de vérifier qu'un processus peut terminer en tenant compte des contraintes de dépendances entre activités, il est inutile de modéliser le taux d'avancement d'une activité et les événement démarrer et terminer sont suffisants. La sémantique sera donc choisie et définie en fonction des questions auxquelles l'utilisateur souhaite apporter une réponse, et donc guidée par les propriétés qu'il souhaite vérifier [COM 08b]. C'est la formalisation de ces propriétés qui permettront d'identifier les informations dynamiques à définir et les événements à prendre en compte.

Le point de départ est donc la formulation des propriétés à vérifier, par exemple « le processus peut terminer », et son expression en tenant compte des éléments du modèle : « un processus peut terminer si toutes les activités qui le composent peuvent terminer ». Ces propriétés peuvent être formalisées avec une logique temporelle. En utilisant TOCL [ZIE 02], nous pouvons donc écrire :

context Process **inv**:
sometime activities −>**forall**(a | a. state = # finished);

Nous ajoutons donc l'attribut `state` sur les éléments WorkDefinition du métamodèle (informations dynamiques).

S'appuyer directement sur l'état choisi pour représenter l'état de l'activité n'est pas une bonne idée car il pourrait changer, par exemple si on adapte la sémantique. Si l'utilisateur souhaite connaître le degré d'avancement des activités, la propriété pourrait devenir `a.loadConsumed >= 100` (la charge est exprimée en pourcentage). Aussi, nous définissons de nouvelles requêtes sur les métaclasses du modèle. Elles correspondent aux informations pertinentes pour exprimer les propriétés. La propriété devient alors :

context Process **inv**:
sometime activities −>**forall**(a | a. isFinished ());

En utilisant EMF, nous avons défini le sous-ensemble de TOCL qui était utile pour exprimer les propriétés OCL de notre exemple. De manière à en faciliter l'utilisation, nous avons utilisé xText pour en définir une syntaxe concrète textuelle correspondant à la syntaxe ci-dessus.

Pour vérifier les propriétés nous avons choisi d'utiliser une sémantique de traduction et les réseaux de Petri comme langage formel, et plus précisément la boîte à outil Tina [BER 04]. Il s'agit alors de construire le réseau de Petri qui donne la sémantique souhaitée au réseau de Petri. Nous ne rappelons pas le schéma de traduction ici mais il peut être trouvé dans [BEN 07].

Vérifier une propriété sur le modèle de processus, par exemple la terminaison d'un processus, revient alors à traduire cette propriété sur le réseau de Petri correspondant. Nous utilisons pour cela LTL, la logique linéaire temporelle utilisée par le *model-checker* de Tina. La difficulté est que cette transformation doit tenir compte du schéma de traduction utilisé pour transformer le modèle de Processus en réseau de Petri.

La figure 2 explique comment nous avons mis en œuvre ces traductions. Partant d'un modèle de processus, SPD2PN.atl est une transformation modèle à modèle écrite en ATL qui transforme un modèle SimplePDL en un réseau de Petri correspondant. Une autre transformation ATL, modèle à texte cette fois, traduit un modèle de réseau de Petri dans la syntaxe textuelle utilisée par la boîte à outils Tina.

La partie supérieure de la figure explique la traduction des propriétés exprimées en TOCL en propriétés LTL utilisées par le model-checker de Tina. Le point important est la deuxième transformation notée T2 qui traduit une propriété TOCL en une propriété LTL. Cette propriété dépend des conventions choisies pour traduire le modèle SimpleDPL en réseau de Petri. Par exemple, une activité a est terminée si la place du réseau de Petri *a_finished* contient un jeton. Ces conventions de nommage des places sont décrites dans le module ATL `Utils.atl`. Ainsi une fonction (*helper* en ATL) *getFinishedStateString* définie sur *WorkDefinition* retourne le nom à donner à la place qui correspond à l'état terminé. Il contient également la manière d'implanter les requêtes définies sur le modèle SimplePDL. Par exemple, la requête *isFinished* de *WorkDefinition* appliquée sur a donnera l'expression LTL *a_finished*. Les requêtes sont définies sous la forme de *helpers*.

3. Conclusion

Nous avons présenté une approche systématique pour construire des outils de vérification exhaustive de propriétés métiers dans des DSML. Celle-ci s'appuie sur l'utilisation de TOCL au niveau du métamodèle étendu du DSML. Les propriétés TOCL sont traduites automatiquement en LTL en s'appuyant sur la transformation du DSML en réseau de Petri dédiée à une certaine abstraction adaptée à une famille de propriétés.

Ce travail s'inscrit dans une approche plus générale, dite orientée propriété. Nous préconisons en effet de commencer par identifier les propriétés qui intéressent l'utilisateur pour définir ensuite la sémantique et donc la traduction adaptées.

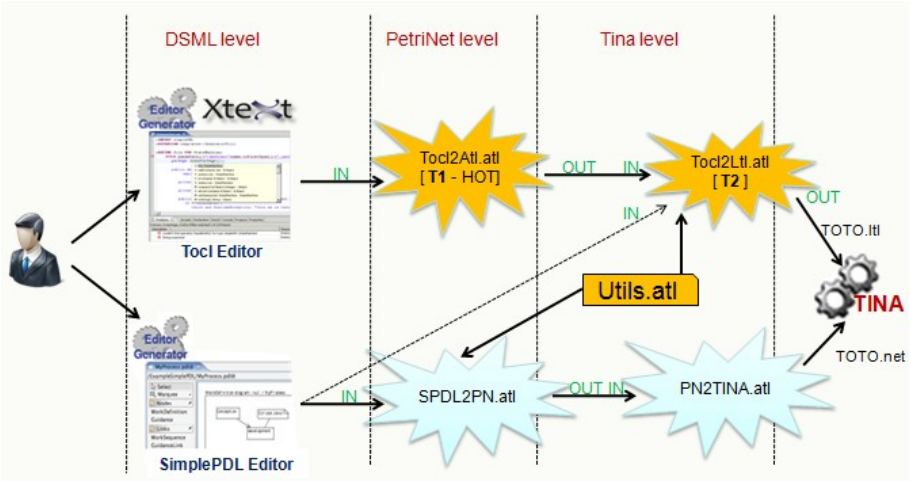

Figure 2. *Principales étapes pour la vérification d'un modèle conforme à un DSML*

Plusieurs travaux sont en cours. D'une part, nous étendons la validation à d'autres cas d'application. D'autre part, nous facilitons la gestion du retour des échecs de vérification, sous la forme de traces d'exécution. Pour cela, nous construisons explicitement les liens entre les modèles utilisateurs et les modèles de vérification dans la chaîne actuelle, et nous exploitons ces liens lors du retour pour construire une trace dans le langage utilisateur à partir d'une trace dans le langage de vérification.

4. Bibliographie

[BEN 07] BENDRAOU R., COMBEMALE B., CRÉGUT X., GERVAIS M.-P., « Definition of an eXecutable SPEM2.0 », *Proceedings of the 14th Asian-Pacific Software Engineering Conference (APSEC)*, Japan, décembre 2007, IEEE Computer Society, p. 390–397.

[BER 04] BERTHOMIEU B., RIBET P.-O., VERNADAT F., « The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets », *International Journal of Production Research*, vol. 42, n° 14, 2004, p. 2741–2756.

[COM 08a] COMBEMALE B., « Approche de métamodélisation pour la simulation et la vérification de modèle », Thèse de doctorat, Institut National Polytechnique de Toulouse, Toulouse, France, juillet 2008.

[COM 08b] COMBEMALE B., CRÉGUT X., GAROCHE P.-L., THIRIOUX X., VERNADAT F., « A Property-Driven Approach to Formal Verification of Process Models », CARDOSO J., CORDEIRO J., FILIPE J., PEDROSA V., Eds., *Enterprise Information System IX*, Springer-Verlag, 2008.

[ZIE 02] ZIEMANN P., GOGOLLA M., « An Extension of OCL with Temporal Logic », JÜRJENS J., CENGARLE M.-V., FERNANDEZ E., RUMPE B., SANDNER R., Eds., *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, vol. TUM-I0208, Université Technique de Munich, Institut d'Informatique, septembre 2002, p. 53–62.