Xmodeling Studio : Un outil pour définir des DSL exécutables

Léa Brunschwig[1], Eric Cariou[1], and Olivier Le Goaer[1]

Université de Pau et des Pays de l'Adour, Pau, LIUPPA, France
lea.brunschwig@etud.univ-pau.fr
eric.cariou@univ-pau.fr
olivier.legoaer@univ-pau.fr

Résumé

Cet article présente un plugin Eclipse appelé Xmodeling Studio. Cet outil permet d'assister, dans un premier temps, les ingénieurs de langages dans la conception et l'implémentation de DSL exécutables et de leur moteur d'exécution en prenant en compte les opérations métier et la gestion de leur flot de données, puis, dans un second temps, Xmodeling donne la possibilité aux ingénieurs logiciel d'intégrer le travail produit par les ingénieurs de langages dans l'environnement Java de leur choix.

Abstract

This article introduces an Eclipse plugin called Xmodeling Studio. This tool assists, initially, the language engineers for the design and implementation of executable DSL and their execution engine considering the business methods and the management of their data flow, then, secondly, Xmodeling is giving the possibility to the software engineers to integrate the language engineers' work in the environment of their choice.

1 Introduction

L'ingénierie des modèles (IDM) est une discipline qui base son processus d'ingénierie logicielle sur les modèles. Un modèle étant l'abstraction d'un système concret faite de manière à ce que l'on puisse facilement comprendre le système qui est modélisé et de façon à ce que ce modèle réponde aux questions que l'on se pose sur lui [4, 6]. Le but de l'IDM est de gagner en productivité, portabilité, maintenabilité et interopérabilité cependant, il semble d'après [7] que ces objectifs ne soient pas atteints dans de nombreuses situations. En effet la génération de code permet un gain en productivité mais l'intégration à du code ou des systèmes déjà existants semble poser problème.

Dans cet article, nous présentons un outils appelé « Xmodeling Studio » (pour *eXecutable modeling*) qui va dans le sens d'une intégration simple de modèles exécutables. De nombreuses recherches concernant la modélisation exécutable ont été menées, s'intéressant plus particulièrement à la simulation, vérification, validation ou génération de code [3, 2, 8, 1] mais laissant le plus souvent la partie métier de côté. Pour le développement d'une application utilisant les modèles exécutables, il est primordial de bien séparer la partie comportementale de celle métier [5]. Le modèle s'occupe du comportement du système et il évolue grâce au moteur d'exécution. La partie métier est, sauf exception, implémentée à part. Le lien entre les deux est fait en associant des opérations métier à des éléments de comportement. Cependant, un autre problème émerge, celui de la gestion du flot de données entre les opérations métier. Ces dernières ayant des paramètres et des valeurs de retour pouvant être communs entre elles, il faut une solution pour assurer la pérénnité de la valeur de ces variables. Par exemple, pour une machine à états avec deux états A et B, on peut associer à l'état A une opération métier

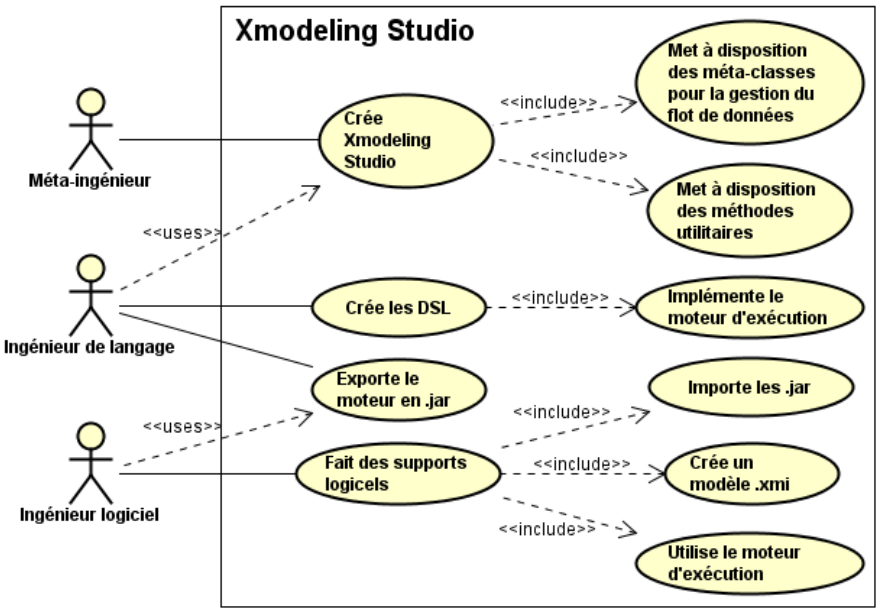

FIGURE 1 – Le processus sur trois niveaux d'Xmodeling Studio

op1 et à l'état *B* une opération métier *op2*. Quand on activera l'état *A*, le moteur d'exécution exécutera l'opération *op1*. De même pour *op2* en activant l'état *B*. C'est le moteur d'exécution qui gère ces appels d'opération et il n'existe pas de moyen de lui préciser que la valeur de retour de *op1* est à passer ensuite en paramètre de *op2*. Xmodeling Studio offre une solution générique pour la gestion des opérations métier.

Dans la suite de cet article, nous présentons la vue globale de l'architecture de l'outil tout en illustrant son fonctionnement avec la réalisation d'un DSL exécutable : PDL pour *Process Description Language*.

2 Outils et processus

Xmodeling Studio est un plugin pour EMF [9] qui sert à produire du code multi-plateforme et a été conçu pour répondre à un problème de neutralité technologique. Cette solution se veut concrète et pragmatique puisque le langage employé est Java, tout en restant lié aux concepts d'abstraction de l'IDM. La figure 1 présente le contexte dans lequel évolue cet outil et illustre le processus à 3 étages d'Xmodeling Studio. L'ingénieur logiciel est libre d'utiliser le moteur d'exécution produit par l'ingénieur de langage dans l'environnement qu'il souhaite en dehors d'EMF (par exemple Android Studio) à condition d'importer les librairies nécéssaires que l'on énoncera plus tard. Ce moteur d'exécution doit ainsi être générique pour qu'il puisse exécuter n'importe quel modèle sans en connaître le contenu métier. Pour cela, nous, les méta-ingénieurs, offrons un ensemble de méta-classes capables de gérer les opérations métier et leur flux de données qui sont ajoutées automatiquement dans le cadre d'un projet Xmodeling proposé par le plugin. Dans cette section, nous détaillons l'architecture de l'outil en distinguant la partie méta-modélisation fournie par Xmodeling Studio de la partie moteur d'exécution qui est du ressort de l'ingénieur de langage à destination de l'ingénieur logiciel. Afin d'illustrer nos propos

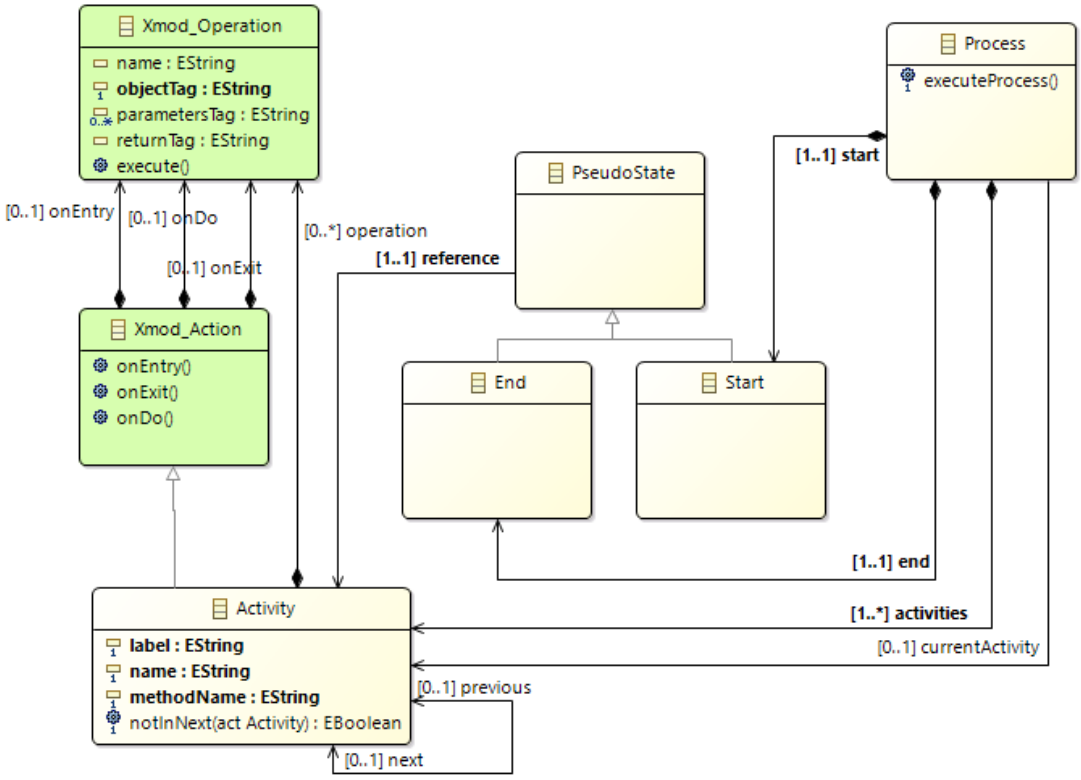

FIGURE 2 – Métamodèle de PDL (diagramme Ecore)

et de les rendre plus clairs, nous nous appuierons sur l'exemple de PDL qui permet une exécution séquentielle d'opérations métier définies par l'ingénieur logiciel.

La figure 2 représente en jaune le méta-modèle initial de PDL. Un processus contient un ensemble d'activités qui forment une séquence ordonnée via les références `next/previous`. Un processus possède une activité de début et une de fin. Le but ici est de rajouter à ce méta-modèle la possibilité de définir au niveau modèle des opérations métier que l'on associera aux activités et au niveau code de pouvoir les exécuter, tout cela de manière automatique et générique.

2.1 Partie méta-modélisation

Afin de gérer le complexe flux de données entre les opérations métier, nous utilisons des annotations Ecore qui sont placées sur certains éléments du méta-modèle : `Xmod_main` pour identifier la racine du méta-modèle et `Xmod_exec` pour les éléments exécutables à qui on veut associer des opérations métier. Cela va permettre d'offrir un cadre générique qui est un ensemble de méta-classes qui, en fonction de ces annotations, vont être automatiquement générées et ajoutées à ce méta-modèle source au travers d'une transformation méta-modèle vers méta-modèle (MM2MM). Ces méta-classes sont présentes en vert sur la figure 2. Au moment d'implémenter l'interpréteur, un ensemble de classes Java capables de prendre en charge ces méta-classes sont fournies.

Une opération métier est définie par la méta-classe `Xmod_Operation` qui est composée d'un

nom de méthode, d'un objet sur lequel est appelé la méthode, d'éventuels paramètres et d'une possible valeur de retour. A l'exception du nom de la méthode, tous ces méta-attributs textuels sont des tags qui vont servir de clé dans une map Java et dont la valeur sera l'instance de l'objet ou la valeur de la variable représentée. Cette map est sérialisable en XML et permet de garder en mémoire la valeur de chaque variable ou objet employé par les opérations métier. Elle est utilisée par le moteur d'exécution pour gérer le flot de données.

La deuxième méta-classe proposée est `Xmod_Action`, elle possède trois méthodes `onEntry()`, `onExit()` et `onDo()` accompagnées de trois références vers une opération portant les mêmes noms et qui permettent de mimer la sémantique des machines à état UML. Ainsi, cela donne la liberté à l'ingénieur de langage d'utiliser ou non le concept d'entrée et sortie d'un état pour le DSL qu'il aura méta-modélisé.

Enfin, lorsque l'on applique la transformation, les méta-classes décrites sont ajoutées au méta-modèle en liant les méta-classes annotées `Xmod_exec` avec un héritage depuis `Xmod_Action` ce qui permet donc de créer le lien entre la partie comportementale du modèle et les opérations métier. La méta-classe annotée `Xmod_main` est identifiée comme élément racine du méta-modèle et permet d'ajouter une méthode dans une classe java appelée `[nomProjet]XmodUtil.java` que l'on va décrire par la suite. Ainsi, dans le cadre de notre exemple, la méta-classe *Process* est annotée `Xmod_main` puisqu'il s'agit de la méta-classe racine et la méta-classe *Activity* est annotée `Xmod_exec`. Après que l'ingénieur de langage a enclenché la transformation, on obtient le méta-modèle completé de la partie verte.

2.2 Partie moteur d'exécution

Le plugin Xmodeling Studio propose la création de projet personnalisé Xmodeling vide ou bien à partir d'un fichier .ecore déjà existant. Ce type de projet fournit à la création les classes Java correspondant aux méta-classes décrites précédemment ainsi qu'une classe utilitaire aussi évoquée.

La partie principale de l'implémentation se trouve dans la méthode `execute()` de la classe `Xmod_Operation`. Cette méthode est chargée de gérer le flux des données entre les différentes opérations métier basé sur les tags. Dans un premier temps, elle récupère la map et cherche l'objet sur lequel l'opération métier s'exécute puis les paramètres sont récupérés. Ensuite, via une invocation dynamique, l'opération métier est exécutée et sa valeur de retour, si elle existe, est mise dans la map. Cependant, à l'implémentation du moteur d'exécution, l'ingénieur de langage n'appelle pas forcément directement cette méthode `execute()` mais plutôt les méthodes proposées par `Xmod_Action` à savoir `onEntry()`, `onDo()` et `onExit()`.

Pour PDL, le moteur d'exécution se trouve dans la méta-classe *Process* et, plus précisément, il s'agit de sa méthode `executeProcess()`. On peut écrire le code suivant qui est donc générique puisqu'il peut traiter les modèles conformes à son DSL sans connaître les opérations métier, ce code peut s'adapter facilement à de nombreux autres DSL :

```
public void executeProcess() {
  // on recupere la premiere activite du processus
  Activity act = this.getStart().getReference();
  do {
    // execution des operations de l'activite si elles sont definies
    act.onEntry();
    act.onDo();
    act.onExit();
    // on passe a l'activite suivante
    act = act.getNext();
    // fin de boucle si il n'y a plus d'activite
  } while (act != null);
```

```
}
```

Enfin, la classe utilitaire `XmodUtil` fournit des méthodes pour charger et sauvegarder un modèle XMI ou la map ainsi qu'une méthode pour accéder à l'élément racine du modèle et sont destinées à l'ingénieur logiciel afin qu'il puisse utiliser l'interpréteur produit par l'ingénieur de langage. Pour pouvoir utiliser le langage et le moteur d'exécution correspondant produit par l'ingénieur de langage, l'ingénieur logiciel devra dans un premier temps récupérer le JAR préalablement exporté par le premier. L'interpréteur n'ayant pas d'interface graphique, il est possible de l'exporter sous forme de JAR exécutable capable de prendre en entrée un fichier XMI dès l'instant où celui-ci est conforme au méta-modèle. Il faudra néanmoins l'accompagner de bibliothèques Java spécifiques à EMF.

L'ingénieur logiciel pourra ensuite se charger d'implémenter dans des classes à part les opérations métier correspondant à celles du modèle utilisé. Supposons qu'il ait modélisé un processus avec une opération de signature `returnVal m1(nb)` associée à une activité et pour une activité plus loin dans le processus, une opération de signature `void m2(returnVal)` et que ces deux opérations s'exécutent sur un même objet `metier`. Il pourra alors lancer le moteur d'exécution avec un code qui ressemblera au suivant :

```
// on cree le contenu initial de la map avec les objets sur
// lesquels s'appliquent les operations metiers
HashMap<String, Object> map = new HashMap<>();
Metier metier = new Metier(...);
map.put("metier", metier);
map.put("nb", new Integer(12));
// on charge le contenu de notre modele avec notre classe utilitaire generee
Process proc = PDLXmodUtil.loadProcess("modele.xmi");
// on affecte la map a notre classe utilitaire generee
PDLXmodUtil.setMap(map);
// on execute le processus
proc.executeProcess();
```

On peut noter l'insertion des objets dans la map qui est donc un couple tag/objet comme décrit auparavant. Les autres tags sont gérés automatiquement par le moteur notamment ici `returnVal` qui est la valeur de retour de la première opération métier et qui sera passée comme paramètre de la seconde opération métier. L'ingénieur logiciel devra préciser dans le fichier XMI les tags des différents paramètres, objets et valeur de retour.

3 Conclusion et perspectives

Xmodeling Studio permet la production d'un interpréteur autonome ayant la capacité de gérer le flot de données d'opérations métier. Il était ici présenté avec le langage PDL mais s'applique à tous les DSL exécutables.

Notre outil donne la possibilité de pouvoir exporter le moteurs d'exécution en un JAR exécutable ce qui permet une facilité d'intégration au sein de systèmes appartenant à l'écosystème Java mais il permet aussi de fournir des fonctionnalités aux ingénieurs de langages pour simplifier l'implémentation de moteur d'exécution en rendant générique la gestion des flots de données. En effet, l'ingénieur devait s'occuper lui même de la gestion du passage des variables entre différentes opérations métier qu'il ne peut pas connaître à l'avance tandis qu'avec Xmodeling Studio, le travail de l'ingénieur de langage est maintenant centré sur la création de méta-modèles et de leurs moteurs d'exécution respectifs intégrant le concept d'opérations métier.

Les perspectives directes de ce travail incluent la définition et la gestion des différents rôles joués par les méta-éléments. En effet certain sont considérés comme statiques et ne doivent pas être changés voire sont inaccessible par les différents ingénieurs quand d'autres peuvent l'être au cours de l'exécution. Il serait donc intéressant de pouvoir assurer un contrôle de la sémantique du code de l'interpréteur.

Acknowledgments

The presented work is part of the MegaM@RT2 project (Megamodeling at Runtime – Scalable Model-based Framework for Continuous Development and Runtime Validation of Complex Systems) which has received funding from the Electronic Component Systems for European Leadership Joint Undertaking (ECSEL-JU) under grant agreement No. 737494. This project receives support from the European Union's Horizon 2020 research and innovation program and from Sweden, Spain, Italy, Finland & Czech Republic.

Références

[1] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, October 2016.

[2] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. Omniscient Debugging for Executable DSLs. *Journal of Systems and Software*, 2018.

[3] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In *11th European Conference on Modelling Foundations and Applications (ECMFA 2015)*, volume 9153 of *LNCS*, pages 45–61. Springer, 2015.

[4] Erwan Breton and Jean Bézivin. Towards an understanding of model executability. In *Proceedings of the international conference on Formal Ontology in Information Systems (FOIS '01)*. ACM, 2001.

[5] Eric Cariou, Olivier Le Goaer, and Franck Barbier. On the Executable Nature of Models. In *The 2nd International Workshop on Executable Modeling at MODELS (EXE 2016)*, volume 1760. CEUR Workshop Proceedings, 2016.

[6] Benoît Combemale. Ingénierie Dirigée par les Modèles (IDM) – État de l'art. Working paper or preprint, 2008.

[7] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480. ACM, 2011.

[8] Dorian Leroy, Erwan Bousse, Manuel Wimmer, Benoit Combemale, and Wieland Schwinger. Create and Play your Pac-Man Game with the GEMOC Studio (Tool Demonstration). In *EXE 2017 - 3rd International Workshop on Executable Modeling*, pages 1–6, Austin, United States, September 2017.

[9] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework (2nd Edition)*. the Eclipe serie. Addison-Wesley.