Introduction à la construction d'un DSL sous Eclipse

Didier Vojtisek

▶ To cite this version:

Didier Vojtisek. Introduction à la construction d'un DSL sous Eclipse. Programmez !, 2009, 120, pp.70-72. inria-00468511

HAL Id: inria-00468511
https://inria.hal.science/inria-00468511

Submitted on 31 Mar 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INTRODUCTION À LA CONSTRUCTION D'UN DSL SOUS ECLIPSE

Créer un langage spécifique à un domaine permet de proposer à vos utilisateurs un environnement de travail adapté à ce domaine, c'est-à-dire manipulant directement les concepts de celui-ci. Nous verrons aujourd'hui comment l'Ingénierie Dirigée par les Modèles (IDM) va nous aider à construire un tel langage et son environnement.

Avec les outils disponibles aujourd'hui dans Eclipse, il est possible de choisir entre réutiliser et adapter un langage existant générique comme UML ou bien directement créer un langage dédié (ou Domain Specific Language). En choisissant cette seconde solution, l'un des avantages sera que l'utilisateur final sera naturellement guidé dans l'utilisation de ses modèles.

Pour construire un DSL, nous vous proposons de suivre un processus qui permet une boucle de prototypage entre chaque étape de construction. Cela permet donc d'expérimenter facilement le langage avant de l'outiller complètement. Suivant notre expérience, cela assure une meilleure progression et permet d'en améliorer la fiabilité.

PRÉSENTATION DE L'EXEMPLE

Pour illustrer nos propos, nous nous s'appuierons sur l'exemple du langage de la tortue Logo (que certains d'entre nous ont peut-être déjà connu sur leur MO5). Logo est un petit langage permettant de diriger une tortue équipée d'un crayon pour lui faire dessiner une figure là où elle est passée. Ce langage a d’abord été destiné à initier des enfants aux concepts de la programmation. Nous allons décrire comment l’IDM va nous aider à construire un environnement de programmation pour Logo, et au final l'appliquer concrètement pour effectivement piloter un petit robot construit en Lego Mindstorm.

UN PEU DE THÉORIE

Traditionnellement, quand on pense à concevoir un langage, on est tenté de réfléchir d'abord à la syntaxe concrète qu'on va donner à celui-ci, et donc en terme de grammaire et de parser. En IDM, la syntaxe concrète n'en est qu’un aspect secondaire. C'est d’abord sur les concepts du cœur du langage que l'on se base (la syntaxe abstraite). Ces concepts de base servent de fondations sur lesquels nous allons bâtir autant de vues que nécessaire.

Nous allons profiter ici des facilités offertes par un outil Eclipse comme Kermeta. Tout d'abord parce qu'étant un Langage Orienté Modèle, Kermeta offre nativement des fonctions qui en facilitent la manipulation, mais surtout parce qu'il nous permet de tisser et d'assembler les éléments nécessaire sans modifier le cœur du langage que l'on souhaite construire. Nous verrons par la suite que cette fonctionnalité est intéressante car elle nous permet de viser plusieurs usages et outils autour du langage. Cela nous permettrait même d'envisager plusieurs variantes de sémantiques si le cœur nous en disait!

DEFINITION DES CONCEPTS DU LANGAGE

Ainsi, dans le monde IDM, les DSL sont communément représentés avec ce que l'on appelle un métamodèle. En fait, c'est un diagramme de classe qui définit les concepts de notre langage : dans Eclipse, cela se fait avec un modèle Ecore.

Dans notre exemple Logo, les concepts seront typiquement les instructions que l'ont voudra faire exécuter à notre tortue. Par exemple, lever le crayon, avancer d'un certain nombre (entier) de pas ou tourner d'un certain angle (exprimé, pour simplifier, comme un nombre entier de degrés). Nous allons donc créer et relier les concepts correspondants : une classe PenUp, une classe Forward, et une classe Right Les instances de ces classes représenteront les instructions du programme Logo de l'utilisateur, par exemple « FORWARD 10 » ou « RIGHT 3*30 ». Les classes Forward, et Right étant paramétrés par une expression arithmétique, elles sont reliées à une classe « Expression », qui peut être par exemple soit une expression binaire (ie. qui compose deux expressions par un opérateur de type +, *, etc.), soit une constante entière, ce qui est représenté par de l'héritage.

L'éditeur arborescent de base fourni avec Ecore est certes fonctionnel, mais n'hésitez pas à lui adjoindre un diagramme de classe (fichier avec l'extension ecorediag) grâce à l'éditeur fourni par le projet Eclipse Ecore Tools.

Afin de profiter des avantages de génération des outils basés sur EMF, il faut penser à ajouter une notion de conteneur sur certains liens pour créer une hiérarchie. En effet, même si les concepts de notre langage sont bien là et forment un modèle valide, la plupart des générateurs s'appuient sur cette notion pour automatiser l'affichage de certaines vues. C'est le cas par exemple de la vue "outline". Pour rendre la tortue plus intéressante à piloter, nous compléterons ses instructions avec quelques structures de contrôle telles que Block, If ou Repeat …

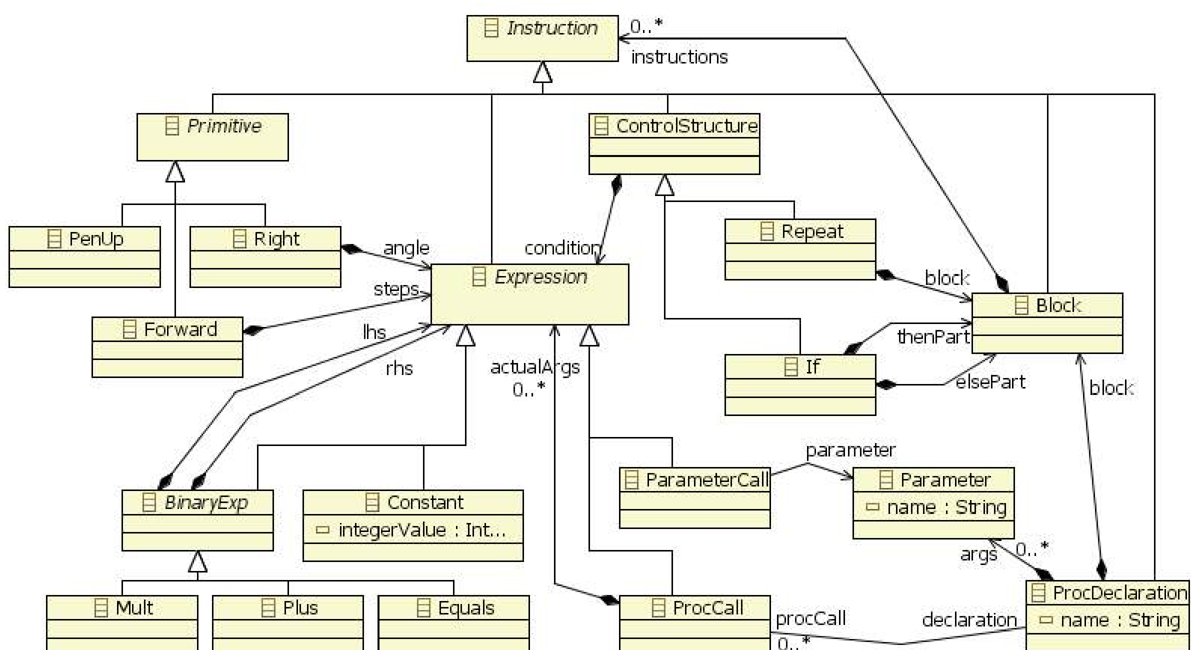

Figure 1. Version minimaliste du métamodèle du langage Logo

Celles-ci permettent de structurer et d'ordonner les séquences d'instructions. De même, l'instruction ProcCall donnera la possibilité d'appeler des blocks définis dans ProcDeclaration avec des paramètres.

CRÉATION DES PREMIERS MODÈLES LOGO AVEC L'ÉDITEUR RÉFLEXIF.

Pour débuter avec votre nouveau langage, le plus simple maintenant est d'utiliser l'éditeur arborescent réflexif. Depuis votre métamodèle logo.ecore, il vous suffit de sélectionner l'élément racine de votre langage et de créer une nouvelle instance. Vous pouvez maintenant ajouter des instructions pour créer vos premiers programmes Logo.

TISSER DES CONTRAINTES COMPLÉMENTAIRES

En expérimentant un peu votre langage, vous vous apercevrez qu'ecore ne vous permet pas d'exprimer certaines contraintes, comme par exemple que les paramètres formels d'une procédure doivent avoir des noms différents. C'est normal puisqu'il n'en défini que la structure. Puisque l'objectif d'un DSL est d'aider les utilisateurs du futur langage, nous allons ajouter des contraintes qui vont les aider à ne pas faire ces erreurs. Le langage Kermeta nous donne ici la possibilité de rouvrir les définitions ecore pour ajouter des éléments nécessaires au besoin courant. Par exemple, pour vérifier que les arguments des procédures logo ont des noms uniques, il suffit d'ajouter l'invariant suivant :

```
package kmLogo::ASM;
require kermeta
// importe les définitions du fichier logo.ecore
require "http://www.kermeta.org/kmLogo"
// réouvre la classe ProcDeclaration du métamodèle logo pour lui ajouter un invariant
aspect class ProcDeclaration{
        inv unique_names_for_formal_arguments is do
                args.forAll{ a1 | args.forAll{ a2 |
                        a1.name.equals(a2.name).implies(a1.equals(a2))}}
        end
}
```

Obtenir un vérificateur de modèle revient alors à charger un modèle logo et d'appeler la méthode checkAllInvariants sur les éléments à la racine du modèle.

Cet invariant aurait aussi pu être écrit en OCL, (le langage officiel de l'OMG) et importé de la même manière. D'ailleurs Kermeta en reprend les facilités de navigation dans les modèles existant en OCL. Néanmoins, Kermeta nous sert surtout de système d'assemblage et lui ajoute d'autres fonctions qui seront utiles pour les besoins des autres activités d'ingénierie des métamodèles.

EXTENSION DES CONCEPTS POUR FOURNIR UNE SIMULATION (SÉMANTIQUE OPÉRATIONNELLE)

Maintenant nous souhaitons rendre ce modèle plus "vivant" et voir notre tortue bouger. Pour en définir le comportement, le plus simple et le plus rapide est de fournir un simulateur.

La première étape consiste à définir une représentation du domaine d'application (une sorte de Machine Virtuelle) sur laquelle le langage va s'appliquer. Pour notre exemple, ce sera tout simplement une ... tortue et les traits qu'elle trace sur son terrain de jeu. Cela peut ici encore être représenté avec un diagramme ecore.

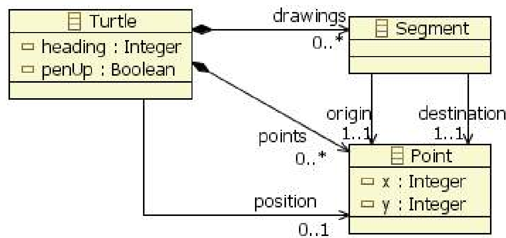

Figure 2. Machine Virtuelle pour le simulateur logo

Ensuite, nous allons donner un peu de comportement à notre tortue en lui fournissant des primitives telles que move ou rotate qui sont spécifiques à cette représentation. Pour cela, il suffit d'indiquer que l'on a besoin d'étendre les classes du domaine d'application et de leur ajouter directement les actions que l'on souhaite sous forme d'opérations.

Enfin, créer le simulateur pour le langage logo revient donc à ajouter des méthodes evaluate(context : Context) sur chacune des instructions du langage. Cette méthode fait appel au contexte pour avoir des interactions avec le domaine d'application et enchaîner sur l'évaluation des instructions suivantes. En fait, c'est tout simplement une variation du patron de conception "Visiteur" qui a été simplifiée grâce à l'utilisation du tissage d'aspect de Kermeta.

```
package kmLogo::ASM;
require kermeta
require "../1.MetaModel/ASMLogo.ecore"
require "../4.VirtualMachine/LogoVMSemantics.kmt"
...
	aspect class Block {
		method eval(context : Context) : Integer is do
			instructions.each{instruction  | result := instruction.eval(context)}
		end
	}
	aspect class Forward {
		method eval(context : Context) : Integer is do
			context.turtle.forward(steps.eval(context))
			result := void
		end
	}
...
```

Pour lancer une simulation, il suffit de charger un modèle logo et de lancer l'évaluation de la première instruction.

AMÉLIORATION DU SIMULATEUR

Le langage Kermeta étant principalement dédié à la manipulation de modèles, il ne propose pas par défaut de fonctions graphiques qui seraient trop spécifiques à un domaine et nuirait à sa compacité. Néanmoins, il offre la possibilité d'accéder à du code java. C'est ainsi que nous pouvons réaliser une petite interface par exemple en AWT pour compléter les sorties textes du simulateur logo.

Parfois, vous pouvez vouloir augmenter les performances de votre simulateur. Dans ce cas, plutôt que d'utiliser la version dynamique de Kermeta (qui interprète le code Kermeta), vous pouvez utiliser son compilateur pour obtenir le code java EMF de votre simulateur. Il pourra ainsi être déployé directement sans Kermeta lui même dans l'environnement de l'utilisateur final.

A ce stade de la mise au point de la sémantique du langage, le coût des évolutions reste raisonnable. Ceci ne sera probablement plus vrai une fois que vous aurez avancé dans les étapes suivantes et en particulier si vous avez construit un compilateur car leur coût de maintenance est généralement plus élevé.

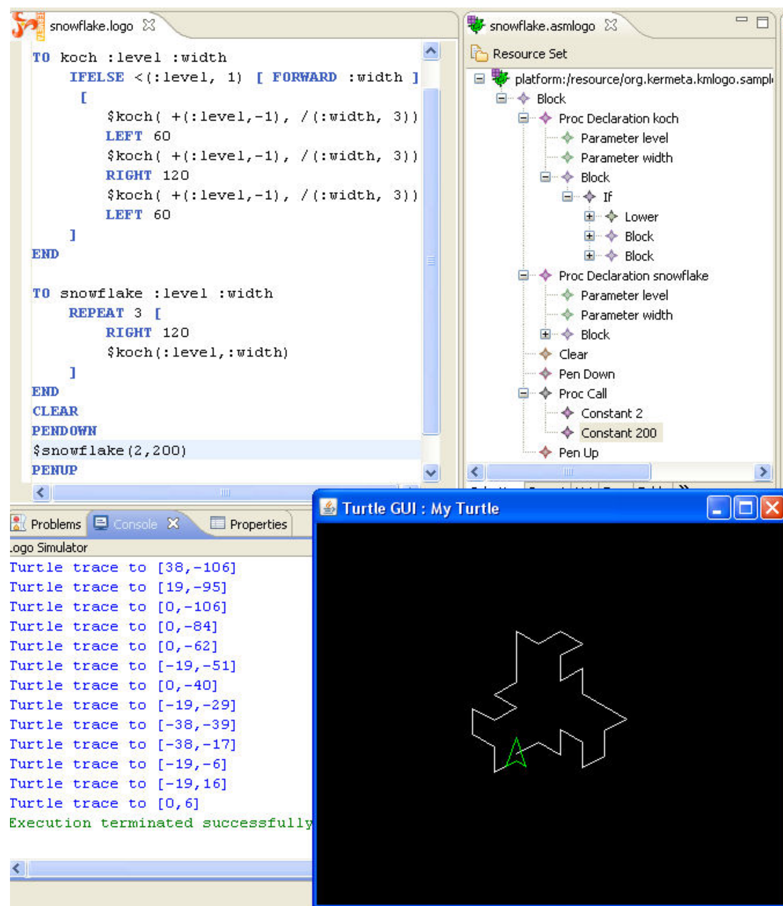

Figure 3. Exemple d'exécution un programme logo avec le simulateur

VOUS AVEZ DIT SYNTAXE CONCRÈTE ?

Maintenant que les concepts de notre langage sont stables, nous pouvons commencer à capitaliser sur son usage. En particulier pour le rendre plus agréable à utiliser, nous souhaitons en améliorer l'IHM, rien de mieux alors que d'en définir une syntaxe dédiée par exemple textuelle ou graphique.

Gérer des syntaxes implique d'utiliser une ou plusieurs techniques, chacune d'entre elles pouvant nécessiter de longues explications. Nous nous contenterons ici d'évoquer les pistes utilisant des outils de l'IDM visant à vous faciliter cette tâche.

- Générer et customiser l'éditeur arborescent par défaut. Cela ce fait grâce au genmodel d'EMF. En général, on commence par ça, cela permet très facilement d'avoir une extension de fichier qui est propre à notre modèle, d'utiliser des libellés et des icônes plus parlantes. C'est d'autant plus intéressant que ces informations seront reprises par d'autres éditeurs plus évolués (typiquement dans la vue "outline" qui leur sera associée)

- Générer un éditeur graphique. Cela se fait grâce à un modèle permettant de faire le lien entre le langage et éditeur graphique. L'outil, (GMF ou Topcased) génèrera alors la plupart du code pour fonctionner dans Eclipse.

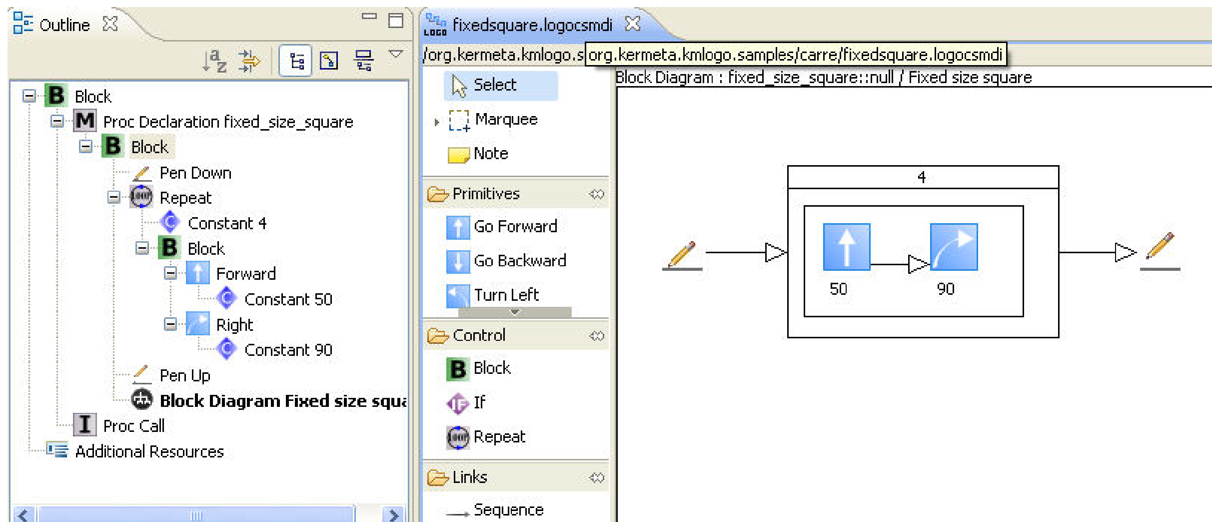

Figure 4. Exemple de modeleur généré avec Topcased

- Obtenir un éditeur textuel. Traditionnellement cela consiste à fournir un parser et un pretty printer pour notre langage (typiquement avec des outils comme sableCC ou antlr).Plus récemment des outils suivant la même philosophie que pour les éditeurs graphiques commencent à apparaitre pour obtenir des éditeurs plus intelligents à moindre coût, mais ceci est hors du spectre de cet article.

CONNECTONS-LE AU MONDE RÉEL

Bon, c'est bien joli tout ça, mais et si nous utilisions notre langage sur une vrai plateforme ? Nous avons choisi un robot construit avec des Lego Mindstrom. Il est équipé de trois moteurs indépendants : deux pour piloter les roues, un pour actionner le stylo. Il faudra que nous transformions nos modèles de programme logo en du code compréhensible par la machine.

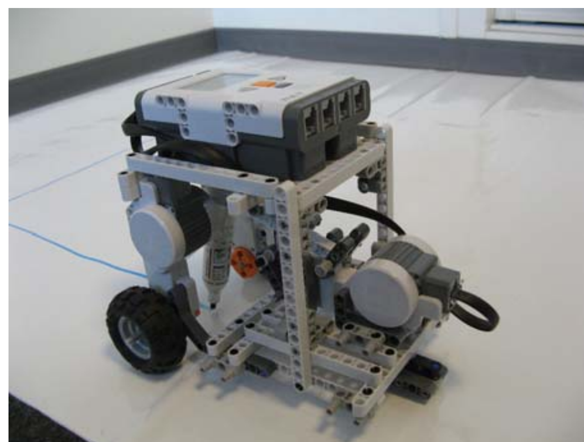

Figure 5. Exemple de plateforme concrète : Tortue robot en Lego

Pour cette tâche, il existe de nombreuses techniques et langages de transformations de modèle qui ont leurs avantages et inconvénients. Le choix dépend généralement de la complexité et du niveau de

maintenabilité voulus pour la transformation à développer. Notre robot peut être programmé en NXC (Not eXactly C). Cette fois encore, nous utilisons ici les capacités de tissage pour compléter notre métamodèle de langage avec un compilateur. En effet, pour une compilation simple comme celle dont nous avons besoin ici, il suffit de traverser ce modèle pour générer directement le code NXC correspondant. Si la transformation avait été plus complexe nous serions passés par un modèle intermédiaire spécifique au NXC avant de générer le code. Dans tous les cas, la structure de base du langage nous sert de trame directrice.

A notre avis, la construction d'un compilateur doit être faite après avoir fait un simulateur. Il y a bien évidement des exceptions mais la mise au point et la maintenance d'un compilateur est sensiblement plus coûteuse que celle d'un simulateur et doit donc être faite seulement quand le langage est suffisamment stable. De plus, votre simulateur vous servira de version de référence pour tester les différentes implémentations concrètes. Dans notre exemple, nous visons une plateforme Lego Mindstorm, mais déjà sans même parler d'un autre robot, nous pouvons envisager plusieurs dialectes et bibliothèque de programmation pour celui ci !

Ainsi il est possible de mettre au point des programmes dans notre nouveau langage tortue logo sans avoir à dépendre de la lenteur du robot réel ou de la santé des batteries qui le meuvent. On voit ici vraiment tout l'intérêt des modèles qui donnent une abstraction de la réalité.

CONCLUSION

Avec ce petit exemple, nous avons un aperçu de différentes technologies utiles pour créer un langage dédié en capitalisant sur le cœur de ses concepts. En particulier, le fait de spécifier sa sémantique de manière plus précise grâce aux modèles autorise une exploitation de ceux-ci pour de nombreux usages. On pourrait envisager par exemple de générer des tests ou d'assembler votre DLS avec d'autres pour générer une plateforme plus complexe. L'utilisation des aspects de Kermeta nous en a simplifié la tâche. Vous pouvez bien entendu appliquer ces techniques à vos langages existants (comme le standard UML par exemple) dès lors que vous disposez d'une définition Ecore pour celui-ci.

Didier Vojtisek
Ingénieur de recherche INRIA
www.irisa.fr/triskell
www.kermeta.org