# Une approche de MDE pour la résolution de problèmes de configuration : Une application à la plate-forme Eclipse

Valerio Cosentino, Guillaume Doux, Patrick Albert, Gabriel Barbier, Jordi Cabot, Marcos Didonet del Fabro, Scott Uk-Jin Lee

# Une approche de MDE pour la résolution de problèmes de configuration : Une application à la plate-forme Eclipse

**Valerio Cosentino[*/**] — Guillaume Doux [*] — Patrick Albert [**] — Gabriel Barbier [***] — Jordi Cabot [*] — Marcos Didonet Del Fabro [****] — Scott Uk-Jin Lee [*****]**

[*] *AtlanMod, INRIA & EMN,*
*4 rue Alfred Kaestler, 44307 Nantes*
*{Guillaume.Doux,Jordi.Cabot}@inria.fr*

[**] *IBM France,*
*9 rue de Verdun, 94253 Gentilly*
*{valerio.cosentino,AlbertPa}@fr.ibm.com*

[*****] *CEA-List,*
*CEA/Saclay, 91191 Gif-sur-Yvette*
*Scott.Lee@cea.fr*

[***] *Mia-software,*
*4 rue du chateau de l'Eraudière,*
*44324 Nantes*
*gbarbier@mia-software.com*

[****] *Universidade Federal do Paranà, Brésil*
*marcos.ddf@inf.ufpr.br*

RÉSUMÉ. *La recherche de la bonne configuration est souvent une tâche complexe nécessitant la gestion des nombreuses dépendances entre plug-ins. D'autant plus que la plupart des moteurs de configuration existants n'ont pas la flexibilité nécessaire permettant de s'adapter à différents scénarios. Dans cet article, nous proposons une approche fondée sur l'IDM permettant la résolution de problèmes de configuration, en les représentant comme des problèmes de satisfaction de contraintes. Un de nos partenaires industriels a utilisé cette approche pour la gestion des plug-ins dans le cadre d'Eclipse. Cette gestion est considérée comme un problème important pour tous les fournisseurs de solutions basées sur Eclipse.*

ABSTRACT. *Finding the right configuration is often a challenging task since one needs to deal with many dependencies between plug-ins and most of existing configuration engines are not flexible enough to work in different scenarios. In this paper we propose a MDE-based approach to solve configuration problems, considering them as constraints satisfaction problems. This approach has been applied by an industrial partner to the management of plug-ins in the Eclipse framework, a big issue for all the technology providers that distribute Eclipse-based tools.*

MOTS-CLÉS : *Configuration, IDM, Eclipse, Plug-in, CSP*

KEYWORDS: *Configuration, MDE, Eclipse, Plug-in, CSP*

[e] soumission à *7ièmes Journées sur l'Ingénierie Dirigée par les Modèles*, le 5 mai 2011.

## 1. Introduction

Les systèmes logiciels complexes sont construits par assemblage de composants (composants dans un sens large, i.e. COTS, bibliothèques, plug-ins, ...) provenant de différents dépôts. Cela simplifie le développement du système, mais introduit inévitablement une dimension de complexité supplémentaire due à la nécessité de gérer tous ces composants. Chaque composant peut évoluer de façon indépendante et des nouvelles versions peuvent introduire / casser des dépendances. Ce problème prend effectivement une importance particulière au sein de la communauté Eclipse où les nouveaux outils sont construits en utilisant les plug-ins disponibles sur la plate-forme. Ces plug-ins sont généralement disponibles dans plusieurs versions, mais chaque application peut requérir des plug-ins avec des versions spécifiques. Maintenant toutes les informations concernants les dépendances pour un outil sont fournites manuellement par les ingénieurs de configuration, qui révisent empiriquement que la definition du build créée soit valide. Ce processus que on vient de décrire n'est pas reproductible, adaptable et renouvelable.
Dans cet article, on se propose de automatiser et d'optimiser les définitions de builds pour les outils en utilisant l'Ingénierie dirigée par les Modèles et des techniques de la Programmation par Contraintes, afin d'éviter que les utilisateurs finaux ne soient confrontés à des problèmes de configuration. La partie 2 décrit la configuration avec une approche CSP, alors que la partie 3 présente la mise en œuvre, la partie 4 conclut cette étude.

## 2. La configuration comme un problème de satisfaction de contraintes

La programmation par contraintes [JAC 00] est un paradigme déclaratif de résolution de problèmes où le processus de programmation se limite à la définition de l'ensemble des contraintes. Ces contraintes sont prises en charge par un solveur ayant la tâche de trouver une solution satisfaisant ces constraintes.
Les problèmes adressés par la programmation par contraintes sont appelés problèmes de satisfaction de contraintes (CSP). Un CSP est représenté par le tuple CSP = <V,D,C> où V désigne l'ensemble fini des variables du CSP, D l'ensemble des domaines, un pour chaque variable, et C l'ensemble des contraintes sur les variables. Une solution d'un CSP consiste en une affectation de valeurs aux variables satisfaisant toutes les contraintes, en ayant chaque valeur dans le domaine de la variable correspondante.
La configuration des plug-ins Eclipse peut être représenté par un CSP et elle peut être considérée comme un cas particulier de l'approche de «recherche de modèle »[KDA 10]. Le problème peut, donc, être énoncé comme suit : étant donné un ensemble de plug-ins d'Eclipse partiellement connectés et un ensemble de contraintes qui doivent être satisfaites, trouver une distribution valable et exécutable d'un build (ou optimale, en fonction d'une propriété donnée).

## 3. L'implementation

La mise en œuvre a été realisée en utilisant IBM ILOG JRules, un système de gestion de règles métier, et IBM ILOG OPL-CPLEX, un système pour la resolution de problèmes d'optimisation.
OPL (Optimization Programming Language) [HEN 99] est un language utilisé pour le développement de problèmes de contraintes et des modèles d'optimisation. Les programmes OPL sont executés par le moteur d'optimisation IBM ILOG CP.
La résolution d'un problème de configuration de plug-ins Eclipse en termes de CSP est implémentée comme une chaîne d'opérations de transformation sur le modèle de plug-ins initial (Fig. 1).

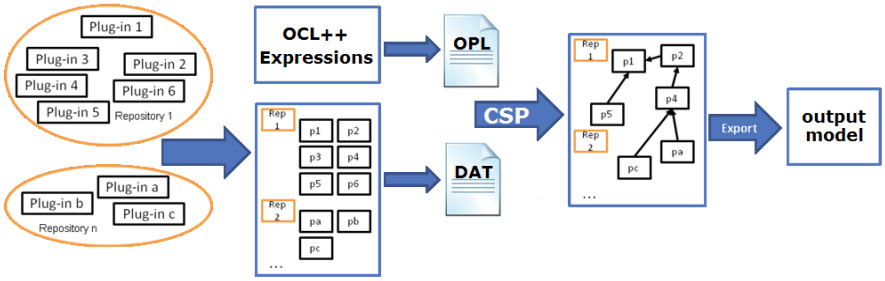


**Figure 1.** *Le processus d'une définitions de build*

Dans la première étape, toutes les informations utiles concernants les plug-ins d'input, comme par example les noms, les numeros de version, les dépendances, etc.. sont memorisées dans un modèle conforme au metamodèle en figure 3 et après traduites dans le format *OplDat*, qui représente les données d'entrées utilisées par OPL. Les données produites sont des tableaux et des matrices de constantes, utilisées comme des indices pour garder les associations parmi les elements d'entrée de la chaîne. Les contraintes OPL sont extraites à partir des expressions en OCL++, écrites sous forme de règle métier (Jrules). Les expressions en OCL++ ajoutent l'expressivité des problèmes de contraintes au language OCL.
OCL est un langage formel, basé sur la logique des prédicats du premier ordre, pour annoter les diagrammes UML en permettant l'expression de contraintes.

```
package "bundle" {

    context pd : PluginDependency, plg : Plugin inv :
        pd.plugin = plg implies
        Plugin.allInstances().exists(p | plg = p and p.id = pd.id
                                         and p.version >= pd.minimumVersion
                                             and p.version <= pd.maximumVersion);

}
```

**Figure 2.** *Un example d'une expression en OCL++*

L'expression en figure 2 permet de trouver des bonnes configurations, au cas oú existent, en assignant à chaque dépendance, un plugin tel que son numero de version soit compris entre les valeurs *maximum* et *minimum* de la dépendance même et que son *id* soit égal à l'*id* de la dépendance.
On peut identifier quatre types des contraintes dans le ficher *Opl* généré : des contraintes structurelles, utilisées pour garder des informations comme la cardinalité et la composition ; des contraintes référençant les éléments du modèle d'entrée ; des contraintes concernant l'initiliasation des données de sortie et des constraintes liées au domaine.
La génération des fichiers *Opl* et *OplDat* permette la traduction du problème de configuration dans le format OPL, en gardant la séparation entre les données d'entrée (booleans, integers, sets, strings, tuples, ...) et les variables de décision.
Dans l'étape suivante l'exécution du moteur CSP peut démarrer. Cette opération est appelée *recherche du modèle* et pendant cette phase, le modèle initial est étendu grâce aux solutions calculées par le moteur.
Le résultat produit par le moteur de contraintes est exprimé comme un ensemble d'entiers, de chaînes de caractères et de réels (le format de sortie OPL). Cependant le ficher de sortie étant difficile d'analyser on le retransforme en un modèle conforme au metamodèle de figure 3.

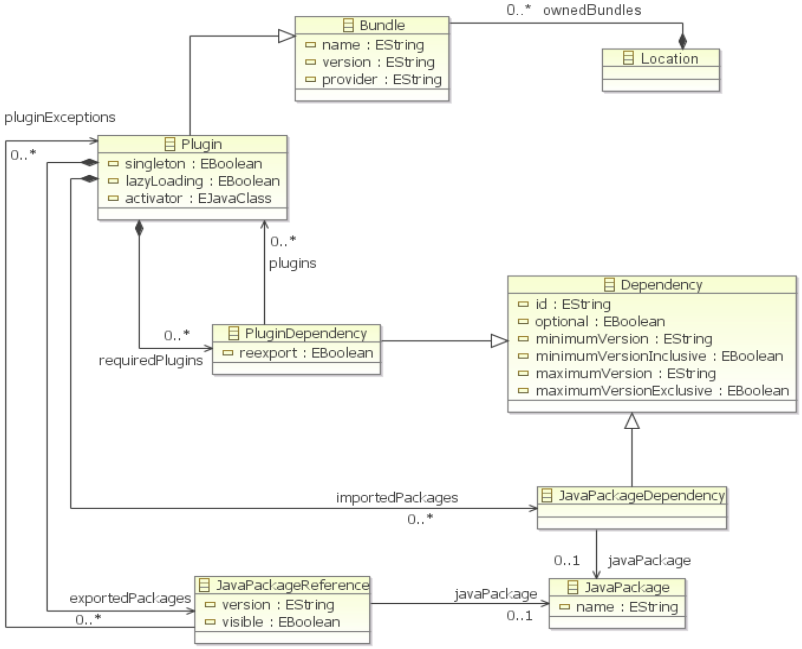


**Figure 3.** *Le métamodèle créé pour représenter les plug-ins*

Le metamodèle de figure 3 modélise les *Locations*, les *Plugins* et les *PluginDependency*. Une *Location* contient 0 ou plus *Bundle*. *Bundle* a été conçu comme une

classe abstraite pour permettre la modélisation des nouveaux éléments (par example les *Features* d'Eclipse). Un *Bundle* est representé par un nom, un numero de version et un provider. Un *Plugin* étend la classe *Bundle* et contient 0 ou plus dépendances. Une dépendance se rapporte à 0 ou plus *Plugin* et elle est representée par un id, un minimum et un maximum numero de version acceptés et des attributs booléens. *PluginDependency* étend la classe abstraite *Dependency*.

## 4. Conclusions

Cet article présente une solution aux problèmes de configuration en utilisant une combinaison des techniques de l'ingénierie dirigée par les modèles et de programmation par contraintes. Cette approche combine les avantages du CSP et ceux de l'expression du problème au niveau modèle (par exemple en écrivant les contraintes en OCL++). En outre, la chaîne de transformation rend le solveur de contraintes transparent pour l'utilisateur, qui fournit et reçoit des modèles en entrées / sorties du processus. La séparation du modèle d'entrée et de méta-modèles en deux transformations permet une indépendance entre la spécification du problème (métamodèles + contraintes) et les modèles d'entrée avec les données initiales pour démarrer le processus de résolution de contraintes. L'approche CSP utilisè est une bonne option pour la recherche d'une solution complètement automatique. En outre, la réalisation de ce projet a également montré les avantages de l'IDM lorsqu'il est utilisé comme une méthode d'unification de domaines hétérogènes, comme les domaines des configuration de plug-in Eclipse et de la programmation par contraintes. Cependant, notre approche souffre de l'absence de normes dans le domaine de la programmation par contraintes. Même si une partie de la chaîne de transformation est générique, les dernières étapes sont dépendantes du solveur et devront être réimplémentées si l'équipe de développement décide d'utiliser un solveur différent à l'avenir. Dans le futur, nous avons l'intention de créer un méta-modèle pour les CSP indépendant du solveur permettant l'abstraction des spécificités du solveur (dans notre cas OPL). On travaille aussi sur l'amélioration du language OCL++, afin de le rendre plus flexible pour faciliter sa traduction en OPL.

**Remerciements :** Cet article a été réalisé dans le cadre du projet ANR IDM++.

## 5. Bibliographie

[JAC 00] D. JACKSON, *Automating first-order relational logic In FSE* vol. 11(2), 2000, p. 130-139

[KDA 10] M. KLEINER, M.DIDONET DEL FABRO, P.ALBERT, « Model Search : Formalizing and Automating Constraint Solving in MDE Platforms », *In proceedings of ECMFA 2010 : (6th European conference on Modelling Foundations and Applications)*, 2010, p. 173-188.

[HEN 99] P.V. HENTENRYCK, *The Optimization Programming Language*, MIT Press, Cambridge, 1999.