MajecSTIC 2009
Avignon, France, du 16 au 18 novembre 2009

Expérimentation d'un modèle abstrait de syntaxe abstraite

Fabien Cadoret et Mickaël Kerboeuf

Université de Bretagne Occidentale, LISyC, 20 avenue Le Gorgeu - 29238 BREST Cedex 3
Contact : Fabien.Cadoret@etudiant.univ-brest.fr

Résumé
Dans le contexte de l'*ingénierie dirigée par les modèles*, un DSL est un langage dédié à l'expression et à la manipulation de données spécifiques à un certain domaine. Pour qu'un DSL soit exploitable, il doit être accompagné d'outils d'analyse et de transformation. Ces outils ne sont pas tous spécifiquement liés au DSL. Par exemple, les outils d'optimisation fondés sur l'analyse du flot de contrôle ne sont pas liés à un quelconque domaine et devraient donc pouvoir être appliqués à plusieurs DSL. Or, dans un développement dirigé par la syntaxe, ces outils sont difficiles à réutiliser car trop intimement liés au DSL pour lequel ils ont été conçus. Dans le but de favoriser la réutilisation, nous proposons une démarche qui consiste à créer en premier lieu un modèle abstrait des données requises par les outils partagés. Les DSL ciblés sont ensuite dérivés de ce modèle abstrait. Ils sont ainsi par construction compatibles avec les outils correspondants. Nous expérimentons le bénéfice de cette démarche sur deux DSL de domaines et de syntaxes très différents.

Abstract
In the context of *model-driven engineering*, a DSL is a language dedicated to a particular domain. Using a DSL implies to build analysis and processing tools that are not all specifically related to the DSL. For instance, optimization tools based on the analysis of control flow graphs are not specifically linked to any domain. They should be applied to different DSL. But in a syntax-driven development, these tools are not easily reusable because they are too tightly linked to the DSL for which they have been designed. In order to promote reusability, we suggest a model-driven approach whose first step is the definition of an abstract model of data that are required by the shared tools. DSLs are derived from this abstract model. By construction, they are consistent with the corresponding tools. In this paper, we investigate the benefit of this approach on two DSLs whose syntaxes and domains are very different.

Mots-clés : génie logiciel, modèles, IDM, DSL, syntaxe abstraite

Keywords: software engineering, models, MDE, DSL, abstract syntax

1. Introduction

La réutilisation de composants est un moyen fondamental de construire efficacement des logiciels fiables [6]. Dans le cas particulier des composants d'analyse et de transformation de données exprimées dans un langage spécifique, la réutilisation se heurte à certains écueils. Dans cette section, nous précisons le contexte général de la production logicielle, plus particulièrement dans le cadre de l'ingénierie dirigée par les modèles. Nous présentons ensuite le contexte particulier des langages à domaines spécifiques, et le problème de réutilisation auquel nous nous intéressons.

Production du logiciel

L'industrie du logiciel est confrontée à un état de crise chronique qui se traduit par des coûts et des délais de développement difficiles à maîtriser, une qualité difficile à garantir, et une très faible capacité pour un produit existant à être réutilisé dans différents contextes. Les phases de maintenance sont elles-mêmes difficiles à maîtriser en termes de coûts et de délais [4].
Cette situation est en partie due au caractère immatériel de la production. Une solution logicielle à un problème est notamment constituée de programmes, de données, de procédures et de documentations, dont on peut facilement sous-estimer la complexité. De plus, la constante et rapide

progression de la technologie de l'information s'accompagne d'une constante et rapide progression des besoins exprimés. L'industrie du logiciel doit donc en permanence s'adapter, d'une part à ces nouveaux besoins, et d'autre part à ce que la technologie offre de plus innovant sans qu'il y ait parfois suffisamment de recul pour l'évaluer complètement.

Ingénierie dirigée par les modèles
Pour pallier cet état de crise, un certain nombre de pratiques ont été proposées, étudiées et partiellement adoptées par l'industrie du logiciel. Parmi elles, l'*ingénierie dirigée par les modèles* [7,15] est un paradigme d'ingénierie dans lequel le *modèle* d'un logiciel est au cœur du processus de son développement. Ce modèle a pour objectif de structurer les données et les traitements qui caractérisent les entités conceptuelles que manipule le logiciel, ainsi que les flux d'informations entre ces entités. Parmi elles, on peut distinguer les entités purement techniques liées à l'architecture du logiciel (*e.g.* réseaux, bases de données), et les *entités métier* liées au domaine d'application du logiciel (*e.g.* la banque, la pêche). Dans un développement dirigé par les modèles d'un logiciel, il convient dans un premier temps de définir, ou de réutiliser, le modèle de son domaine d'application. Ce modèle, également appelé *métamodèle*, décrit des concepts métier, ainsi que leurs propriétés, leurs relations et leurs contraintes. Il décrit également la façon de les manipuler. Le premier but est de capitaliser la connaissance d'un domaine métier afin de la réutiliser dans différentes applications liées à ce même domaine. Dans une certaine mesure, via des *environnements* spécialisés [3,5,14], un métamodèle permet également de générer tout ou partie d'un logiciel, d'en vérifier les propriétés, de le documenter, et de maîtriser ainsi plus facilement son évolution.

Domain Specific Languages
Un langage spécifique à un domaine, ou DSL (*i.e. Domain Specific Language*), est un langage dédié à l'expression et à la manipulation des concepts d'un domaine métier particulier [11,16]. Par exemple, SQL peut être considéré comme un DSL spécifiquement dédié au domaine des bases de données relationnelles. Un langage de script shell peut être considéré comme un DSL spécifiquement dédié au domaine de l'administration des systèmes. L'usage d'un DSL est donc ciblé, par opposition à l'usage d'un langage généraliste comme Java ou UML.

Production de l'outillage d'un DSL
L'outillage d'un DSL est l'ensemble des outils d'analyse et de transformation qui manipulent les programmes de ce DSL. Par exemple, l'outillage de SQL pourra contenir un outil de mise en forme normale, ou un compilateur. La démarche qui consiste à développer conjointement un DSL et son outillage peut être considéré comme la généralisation d'un processus d'ingénierie dirigé par les modèles [10]. En effet, les modèles, et en particulier les modèles de domaines qui spécifient des entités métier, sont largement utilisés, et très tôt, dans le processus de développement d'une application centrée sur un DSL. L'importance des modèles intermédiaires qui permettent de transformer un programme source exprimé dans un DSL en une certaine cible (*e.g.* une entité du domaine, ou une propriété à valider) a été montré depuis plusieurs années (*e.g.* [2]).

Problématique et plan de l'article
Plusieurs DSL peuvent requérir un même outil d'analyse ou de transformation tout en étant très différents du point de vue de leur syntaxe et de leur domaine respectif. Il apparaît alors naturel de tenter de réutiliser cet outil dans l'outillage de chacun des langages. Or, on constate que dans une démarche traditionnelle de développement dirigée par la syntaxe, ces outils sont difficiles à réutiliser car trop intimement liés à la syntaxe des DSL pour lesquels ils ont été conçus. Afin de favoriser la réutilisation, nous proposons une démarche dirigée par les modèles qui vise à abstraire en premier lieu toute construction syntaxique. Dans cet article, nous détaillons cette proposition et nous justifions sa pertinence en l'expérimentant sur deux exemples concrets. Dans la prochaine section, nous exposons en détail le principe de notre proposition. Dans la section suivante, nous présentons son expérimentation. En conclusion, nous présentons les travaux connexes et futurs.

2. Modèle abstrait de syntaxe abstraite

La *syntaxe abstraite* d'un langage est un modèle statique dont les instances représentent des programmes du langage. Généralement, la syntaxe abstraite provient de la transformation d'une *syn-*

taxe concrète issue d'une grammaire non-contextuelle (typiquement exprimée en forme de Backus-Naur). La syntaxe concrète est structurée par des *règles de dérivation* qui portent sur des *vocabulaires* terminaux et non-terminaux. Les données conformes à ce type de syntaxes concrètes sont des arbres appelés *arbres de dérivation* ou *arbre de syntaxe concrète* [1]. La syntaxe abstraite est une simplification de la syntaxe concrète qui consiste typiquement à abstraire les notions de règles de dérivation, de terminaux et de non-terminaux. Au delà de ces premières simplifications, la syntaxe abstraite peut être plus ou moins proche de la syntaxe concrète selon les objectifs d'analyse et de transformation des données (*e.g.* compilation ou vérification).

2.1. Développement dirigé par la syntaxe

La transformation de la syntaxe concrète vers la syntaxe abstraite est traditionnellement programmée par les actions sémantiques qui décorent une grammaire fournie à un générateur d'analyseur syntaxique. Ce type d'analyseur produit des instances conformes à la syntaxe abstraite à partir d'un programme syntaxiquement correct.

L'*outillage* d'un DSL est constitué d'un analyseur syntaxique, mais aussi d'outils d'analyse et de transformation. Les données d'entrée de ces outils sont des instances conformes au modèle de syntaxe abstraite. Les outils d'analyse produisent des informations qualitatives (*e.g.* validité du typage statique). Les outils de transformation produisent des données conformes au même modèle (*e.g.* optimisation), ou conformes à d'autres modèles (*e.g.* production de code exécutable).

Lorsqu'on développe l'outillage de plusieurs DSL, on peut constater qu'un certain nombre d'outils réalisés séparément et spécifiquement pour chacun d'eux ont les mêmes finalités. Par exemple, les outils d'optimisation qui analysent le flot de contrôle pour éliminer les variables inutiles et le code mort (*i.e.* code jamais exécuté) peuvent être nécessaires pour des DSL très différents mais partageant un même caractère impératif. Il apparaît donc naturel de tenter de réutiliser ces composants pour les différents langages concernés. Or, on constate que la réutilisation d'un composant est loin d'être immédiate. En effet, comme les modèles de syntaxe abstraite des langages proviennent de leurs syntaxes concrètes, ils encapsulent étroitement des notions plus ou moins spécifiquement liées aux domaines avec des constructions purement liées aux langages.

2.2. Développement dirigé par les outils

Afin de favoriser la réutilisation, nous proposons une démarche dirigée par les modèles dont la première étape consiste à créer indépendamment d'une quelconque syntaxe le modèle des données qui doivent être traitées par les outils ciblés. Nous appelons ce modèle le *modèle abstrait de syntaxe abstraite*. Il s'agit bien d'un modèle de syntaxe abstraite puisqu'il décrit les données qu'au moins un DSL doit permettre d'exprimer. Nous le qualifions d'*abstrait* car il n'est pas issu de la syntaxe concrète d'un DSL particulier. La production de ce modèle abstrait consiste à mettre en place les structures de données les mieux adaptées aux outils. Il pourra s'agir par exemple d'un *binary decision diagram* pour un *model-checker* ou d'un *control data flow graph* pour un compilateur. Cette première phase nécessite des connaissances expertes du domaine d'application des outils, et une maîtrise des meilleures pratiques de conceptions de ce type d'outils (*e.g. design patterns*). Le modèle abstrait doit avoir pour vertue première d'être indépendant des constructions particulières d'un langage. Il doit pouvoir aussi bien s'appliquer à un langage existant qu'à un langage créé de toute pièce pour un besoin précis.

Dans une deuxième phase, les syntaxes abstraites des langages nécessitant ces outils sont dérivées du modèle abstrait. Sur la base de ces modèles dérivés, on peut alors achever l'outillage de chacun des DSL en développant d'une part les outils réellement spécifiques, et en invoquant d'autre part les outils réutilisables définis au niveau abstrait.

La figure 1 présente un exemple typique d'une série de traitements appliqués à des programmes d'un DSL procédural. Un programme conforme à une grammaire est syntaxiquement analysé par le *parser* qui produit des instances conformes au modèle de syntaxe abstraite. Un analyseur permet ensuite de vérifier le respect de certaines contraintes, notamment de typage. Ensuite, un optimiseur est chargé de produire un nouveau lot d'instances, toujours conforme au modèle de syntaxe abstraite, dans lequel on a par exemple éliminé les variables inutiles et le code mort. Enfin, un *pretty-printer* permet de récupérer le programme ainsi analysé et optimisé dans la syntaxe concrète d'origine. La syntaxe abstraite du DSL dérive d'un modèle abstrait sur lequel sont fondés des outils par construction réutilisables (*e.g.* analyse de flot de contrôle utilisée par l'optimiseur).

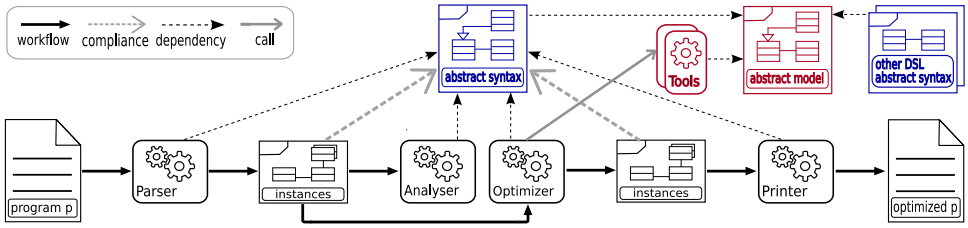

FIG. 1 – Workflow de DSL connecté à un modèle abstrait de syntaxe abstraite

3. Expérimentation

Dans cette section, nous présentons une expérimentation de cette approche sur deux langages. Les DSL sont généralement soit des langages existants, soit des langages créés pour un besoin spécifique. Nous avons retenu un langage dans chacune des catégories. Ils sont de syntaxes et de domaines très différents. Nous présentons d'abord un développement traditionnel dirigé par la syntaxe, et nous présentons ensuite notre proposition. Nous en mesurons le bénéfice par une série de métriques portant notamment sur la quantité de code réutilisé et le temps de développement. Le détail et les résultats complets de cette expérimentation sont disponibles dans [9].

3.1. Exemples de DSL : C-Script et PL/SQL

Pour illustrer la catégorie des DSL créés pour un besoin spécifique, nous introduisons C-Script, un DSL de type C-Shell dont le domaine est l'administration système. Il s'agit syntaxiquement d'un sous-ensemble très restreint du langage C. Il est possible de créer des fonctions locales, de déclarer des variables scalaires, et les seules fonctions primitives correspondent à des commandes système. Il est possible de rediriger l'entrée et la sortie standard des commandes depuis, ou vers, des fichiers. Dans l'exemple suivant, une variable est déclarée, typée, initialisée, et affichée. L'affichage est redirigé vers un fichier :

```
int x = 2;
echo x > resultat.txt;
```

Le deuxième DSL que nous utilisons dans le cadre de l'expérimentation est un langage existant. Il s'agit de PL/SQL [13], une extension procédurale de SQL. Nous le considérons ici comme un DSL dédié à la gestion des bases de données relationnelles. L'exemple suivant illustre la notion de *procédure*, qui se distingue dans le langage de la notion de *fonction* :

```
CREATE PROCEDURE update_price { _newprice NUMBER; _pkey products.pkey%TYPE; }
IS BEGIN UPDATE products SET price = _newprice WHERE pkey = _pkey; END;
```

3.2. Production dirigée par la syntaxe

La première étape de notre expérimentation consiste à développer des outils pour C-Script et PL/SQL en adoptant une démarche dirigée par la syntaxe.

La figure 2 représente une portion du modèle de syntaxe abstraite de C-Script. Le modèle complet est disponible dans le rapport [9]. Il s'agit d'un modèle typiquement produit par une démarche dirigée par la syntaxe. En effet, ce sont les constructions syntaxiques exprimées par la grammaire du langage qui ont déterminé la structure finale du modèle. Dans l'extrait présenté, une instruction (classe Statement) peut être la déclaration d'une fonction locale (classe Function), une conditionnelle (classe If), un bloc (classe Bloc), une affectation (classe Assignment), ou une simple expression (classe Expression). Une conditionnelle est associée à deux sous-ensembles d'instructions qui correspondent respectivement aux clauses *alors* et *sinon*. Elle est également associée à une expression correspondant à une condition. Parmi les expressions existantes, un *appel* (classe Call) est associé à un élément qui peut être *appelé* (classe Callable). Il s'agit soit d'une fonction locale, soit d'un opérateur du langage. Un appel est également associé à une série éventuellement vide d'expressions qui constitue ses paramètres effectifs.

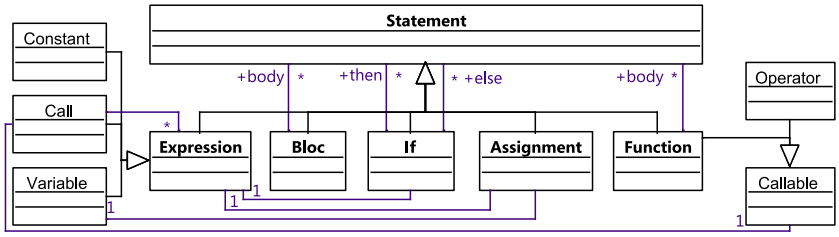

FIG. 2 – Extrait du modèle de syntaxe abstraite de C-Script

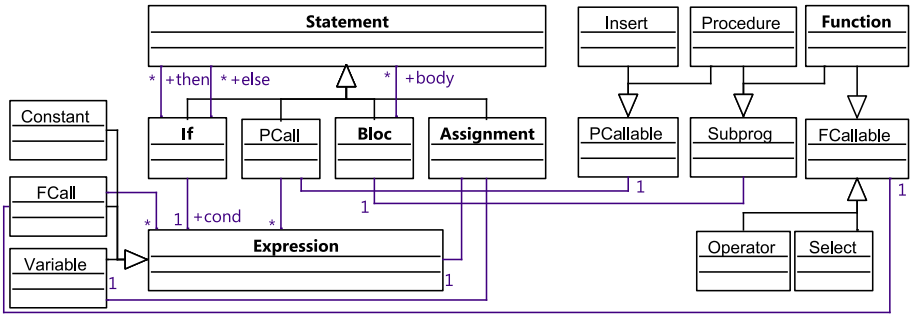

FIG. 3 – Extrait du modèle de syntaxe abstraite de PL/SQL

La figure 3 est un extrait du modèle de syntaxe abstraite de PL/SQL. Le modèle complet est disponible dans le rapport [9]. Comme pour C-Script, ce modèle est typiquement issu d'une grammaire. On y retrouve des notions déjà présentes pour C-Script (instructions, conditionnelles, blocs, affectations, expressions, variables, et constantes). Dans l'extrait présenté ici, nous mettons l'accent sur deux différences fondamentales : une expression *n'est pas* une instruction ; et les notions de *procédure* (classe PCallable) et de *fonctions* (classe FCallable) sont distinctes. L'appel d'une procédure (classe PCall) est une instruction. L'appel d'une fonction est une expression (classe FCall). On distingue ainsi le contexte d'utilisation des procédures (procédures locales et prédéfinies comme INSERT) de celui des fonctions (opérateurs ; fonctions locales et prédéfinies comme SELECT).
Les modèles présentés ont été développés et outillés en Java. Parmi les outils développés, certains sont spécifiques comme l'outil de vérification de la cohérence des requêtes d'insertion pour PL/SQL, ou les *pretty-printer* qui permettent de récupérer les programmes dans leur syntaxe concrète d'origine. Les autres outils développés ont des finalités partagées. Il s'agit d'outils d'optimisation fondés sur une analyse du flot de contrôle : suppression des variables inutiles, détection des variables non initialisées, et propagation de constantes.

3.3. Production dirigée par les outils

Le caractère impératif des deux langages explique l'existence d'outils à finalités communes fondés sur l'analyse de flot de contrôle. Pour autant, l'adaptation d'un composant PL/SQL à C-Script (ou inversement) est loin d'être immédiate. Par exemple, la distinction des concepts de fonctions et procédures en PL/SQL a un impact dans la syntaxe abstraite du langage qu'on ne retrouve pas dans C-Script. Or, du point de vue de l'analyse du flot de contrôle, cette distinction est inutile. Elle est pourtant prise en compte dans le composant d'analyse développé pour PL/SQL, ce qui le rend difficilement adaptable à C-Script.

Modèle abstrait pour l'outillage du flot de contrôle

Ce problème nous amène à définir un modèle dans lequel les données à analyser sont dans une forme optimale pour l'outil. Dans notre exemple, il s'agit d'un *graphe* adapté à l'analyse du flot de contrôle. Il apparaît partiellement en *gris* sur le modèle de la figure 4. Le modèle complet est

disponible dans le rapport [9]. Nous avons défini cette structure selon certains principes fondamentaux de conception objet (*e.g.* principe de *separation of concerns*, *pattern composite*). La structure ainsi construite n'est pas soumise au problème évoqué de distinction de fonctions et procédures.

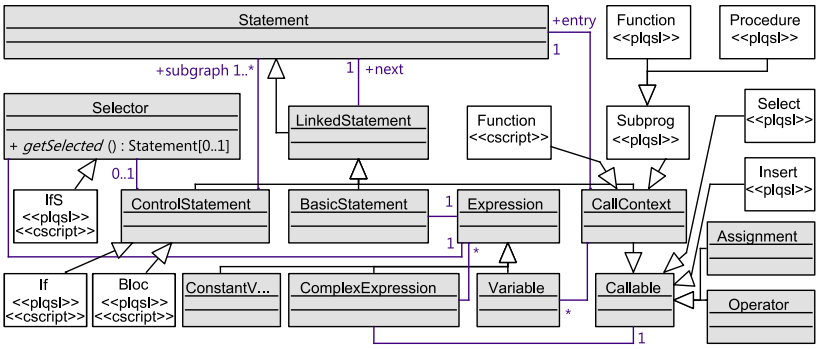

FIG. 4 – Extrait du modèle abstrait (en gris) et dérivations des syntaxes de C-Script et PL/SQL

Les instructions (classe Statement) constituent les sommets du graphe. Parmi les différents types de sommets existant, nous illustrons ici les sommets reliés à un successeur (classe LinkedStatement). Parmi ceux, on distingue les instructions de contrôle (classe ControlStatement), les instructions basiques (classe BasicStatement) et les instructions d'appel (classe CallContext), lesquelles sont liées à un point d'entrée dans un sous-graphe de flot de contrôle (rôle entry joué par une instance de la classe Statement). Les instructions de contrôle introduisent également un ou plusieurs sous-graphes, sélectionnés à l'exécution par un sélecteur (classe Selector) qui indique à partir d'une expression quel sous-graphe doit être parcouru. Enfin, une instruction basique a un successeur dans le graphe de flot de contrôle et encapsule une expression dont l'évaluation produit typiquement un effet de bord. À partir de ce modèle, nous avons recréé les outils de suppression de variables inutiles et de propagation de constantes.

Dérivation de syntaxes

Pour pouvoir réutiliser les outils créés au niveau abstrait, les syntaxes abstraites de PL/SQL et C-Script doivent être connectées au modèle abstrait commun. La connexion est réalisée par héritage, encapsulation, et par application de contraintes. L'encapsulation et l'héritage induisent une dépendance fonctionnelle vers le modèle abstrait, sans réciproque. Les contraintes portent sur le modèle ainsi dérivé et ne remettent pas en cause l'intégrité du modèle abstrait. Dans la figure 4, les classes en *blanc* sont extraites des modèles dérivées. Elles sont stéréotypées par le nom des modèles correspondants, ce qui permet de faire apparaître sur un même diagramme des extraits des syntaxes abstraites C-Script et PL/SQL. Dans les deux cas, les notions d'instructions de contrôle sont précisées (classes If, IfS et Bloc). Le seul contexte d'appel dans le cadre de C-Script est la *fonction* (classe Function). En revanche, le contexte d'appel pour PL/SQL est dénoté par la notion de *sous-programme* (classe Subprog) qui se décline en deux notions distinctes dans le langage : *fonctions* et *procédures*. Les modèles dérivés ainsi obtenus sont soumis à un certain nombre de contraintes. Elles permettent notamment de préciser que l'instance de type *Selector* associée à une instance de If est plus précisément une instance de la classe IfS qui met en œuvre la méthode getSelected conformément à la condition de l'instruction If. Concernant PL/SQL, des contraintes liées d'une part aux classes Procedure et Insert, et d'autre part aux classes Function et Select spécifient la différence qui est faite dans le langage entre les appels conformes à des *expressions* (*e.g.* appel de SELECT ou d'une fonction locale), et les appels conformes à des *instructions* (*e.g.* appel de INSERT ou d'une procédure locale).

Ces contraintes ont été codées en Java. Conformément au principe illustré dans la figure 1, sur la base du modèle commun des données utiles aux optimisations, nous avons ainsi construit un composant réutilisable et réellement intégré à la fois à l'outillage de C-Script et à celui de PL/SQL.

3.4. Comparaison des approches

Afin d'estimer plus précisément le bénéfice de notre proposition, nous avons comptabilisé pour chaque DSL le nombre de classes définies, le nombre de lignes de code et le taux de code réutilisé. La durée de développement est donnée à titre indicatif. Le tableau 1 rassemble ces résultats.

Approche	Outillage	Lignes de code	Classes	Durée	Code réutilisé
syntax-driven	C-Script	3000	47	4 jours	0%
	PL/SQL	3100	58	5 jours	0%
tool-driven	Modèle abstrait	3400	62	4 jours	0%
	C-Script	1500	22	2 jours	72%
	PL/SQL	2200	32	2 jours	68%

TAB. 1 – Comparaison des approches *dirigée par la syntaxe* et *dirigée par les outils*

La première partie du tableau indique le nombre de lignes de code, le nombre de classes et le temps indicatif de modélisation et de développement dans la première phase de l'expérimentation (approche dirigée par la syntaxe). A cause de l'impact des syntaxes concrètes dans la conception des composants, il n'a pas été possible (ou simple) de les réutiliser (d'où le taux nul). Néanmoins, nous aurions pu réutiliser les seules *structures* basiques communes aux deux DSL. Le taux de code commun aurait alors avoisiné les 5%.

La deuxième partie du tableau indique les résultats pour la deuxième phase de l'expérimentation (approche dirigée par les outils). Elle fait clairement apparaître la forte proportion de code réutilisé dans les deux langages. Le nombre de lignes de code mentionné dans cette deuxième partie pour C-Script et PL/SQL concerne le code *spécifique*. On obtient donc le nombre *total* de ligne en y rajoutant les 3400 lignes du modèle abstrait.

A première vue, même si notre proposition semble bien favoriser la réutilisation, elle implique également une inflation du code produit. Cependant, nous avons montré que nos outils pouvaient être réutilisés dans différents contextes. Ansi, plus ils sont réutilisés pour d'autres DSL, moins la part de code supplémentaire est significative proportionnellement. De plus, quand nous analysons en détail le code supplémentaire, nous constatons qu'il correspond essentiellement à la connexion des modèles, à la transformation des données, et à la vérification de contraintes. Ce type de code est précisément celui qui est susceptible d'être généré automatiquement par des procédés récents d'ingénierie dirigée par les modèles. Le code réutilisé est quant à lui le cœur des outils d'analyse. Il est souvent critique, et il requière des connaissances expertes du domaine. Il s'agit donc typiquement de code dont la capitalisation est essentielle.

La durée est indicative, mais elle corrobore nos arguments. En effet, le code supplémentaire n'a pas induit un temps proportiennel de développement supplémentaire. De plus, nos travaux en cours concernant la génération automatique de ce code tendent à réduire significativement cet écart de durée.

4. Conclusion

Dans une démarche traditionnelle de développement dirigée par la syntaxe, la grammaire qui formalise la syntaxe d'un DSL imprègne le modèle des données sur lequel sont fondés les outils d'analyse et de transformation. Ces outils sont donc difficilement exploitables pour d'autres langages. Le développement dirigé par les outils que nous proposons consiste à créer d'abord le modèle des données, puis à en dériver des syntaxes abstraites. Nous avons pu constater sur une expérience liée à deux DSL de syntaxe et de domaine très différents que cette approche permet effectivement de partager directement des outils, par construction communs.

Dans le contexte de l'ingénierie dirigée par les modèles, de nombreux travaux ont été menés dans le but de favoriser la réutilisation de composants, de transformations, ou de modèles, en particulier de syntaxes. Par exemple, dans [8], les auteurs proposent une technique de composition

logicielle qui s'applique à n'importe quel type de langage, décrit par une grammaire ou un métamodèle, grâce à un environnement dédié. Cette technique est un fondement d'EMFText, plug-in Eclipse qui permet de définir des syntaxes textuelles pour des langages décrits par des métamodèles Ecore. Dans l'état actuel des travaux, nous nous sommes focalisés sur les connexions entre modèles abstraits de syntaxes abstraites et modèles *concrets* de syntaxe abstraite (*i.e.* liés directement à des syntaxes concrètes). Nous n'avons pas traité le problème des connexions des syntaxes concrètes et abstraites. Par exemple, parmi les outils que nous avons développés, les *pretty-printers* qui régénèrent des programmes syntaxiquement conforme à une grammaire produisent actuellement directement des chaînes de caractères. Conformément aux principes développés dans [12], nous envisageons de produire à la place des instances d'un *métamodèle de syntaxe concrète*, qu'un *visiteur* pourrait ensuite parcourir dans le but de générer la syntaxe concrète. Ce principe permet de décoreller complètement ces outils d'une quelconque syntaxe, et de les rendre ainsi adaptables à différents langages.
Dans l'état actuel de l'expérimentation, le développement des modèles, des contraintes et du code est entièrement réalisé en Java. Afin de donner un caractère plus systématique (et en partie automatisable) à notre démarche, nous étudions la possibilité d'*outiller* les modèles dans un environnement de *métamodélisation* (*e.g.* EMF) et de *transformation* (*e.g.* ATL). Ces techniques permettraient notamment de guider la dérivation des syntaxes abstraites à partir d'un modèle abstrait.
Nous étudions également la généralisation de la démarche présentée ici afin d'établir une *hiérarchie de modèles abstraits* qui structurerait l'ensemble des outils réutilisables pour un ensemble important de DSL. Au final, cette hiérarchie permettrait d'établir une classification de DSL guidée par des besoins en outillage.

Bibliographie

1. Alfred Aho, Monica Lam, Ravi Sethi, et Jeffrey Ullman. *Compilers : Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, août 2006.
2. Jon Bentley. Programming pearls : little languages. *Commun. ACM*, 29(8) :711–721, 1986.
3. Steve Cook. Domain-Specific Modeling. The architecture journal (9), `http ://msdn.microsoft.com/en-us/arcjournal/bb245773.aspx`, 2006.
4. Edsger Dijkstra. The Humble Programmer. *Communications of the ACM*, 1972.
5. Eclipse Modeling Framework. http ://www.eclipse.org/modeling/emf.
6. William Frakes et Christopher Fox. Sixteen questions about software reuse. *Communications of the ACM*, 38(6) :75–ff., 1995.
7. Robert France et Bernhard Rumpe. Model-driven Development of Complex Software : A Research Roadmap. In *FOSE '07 : Future of Software Engineering*. IEEE Computer Society, 2007.
8. Jakob Henriksson, Florian Heidenreich, Jendrik Johannes, Steffen Zschaler, et Uwe Aßmann. Extending grammars and metamodels for reuse : the Reuseware approach. *IET Software*, 2008.
9. Mickael Kerboeuf et Fabien Cadoret. Modèle abstrait de syntaxe abstraite pour l'outillage réutilisable de DSL. Technical report, LISyC, numéro RR-2009.1.MKFC, `http ://www.lisyc.univ-brest.fr/pages_perso/kerboeuf/rced.pdf`, mai 2009.
10. Ivan Kurtev, Jean Bézivin, Frédéric Jouault, et Patrick Valduriez. Model-based DSL frameworks. In *OOPSLA '06 : Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. ACM, 2006.
11. Marjan Mernik, Jan Heering, et Anthony Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4) :316–344, 2005.
12. Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, et Jean-Marc Jézéquel. Model-Driven Analysis and Synthesis of Concrete Syntax. In *MoDELS*, pages 98–110, 2006.
13. Oracle Database 10g Documentation Library. *PL/SQL User's Guide and Reference*. `http ://download-uk.oracle.com/docs/cd/B14117_01/appdev.101/b10807.pdf`, 2006.
14. Platypus. Technical Summary and download. `http ://cassoulet.univ-brest.fr/mme`.
15. Douglas Schmidt. Model Driven Engineering. *IEEE Computer*, 39(2) :25–31, février 2006.
16. Arie van Deursen, Paul Klint, et Joost Visser. Domain-specific languages : an annotated bibliography. *SIGPLAN Not.*, 35(6) :26–36, juin 2000.