



Université Pierre et Marie Curie



Laboratoire d'Informatique de Paris 6

## RAPPORT DU PROJET

# Éditeur Xtext et Transformation Divine

*Auteurs :*

Mokrane KADRI

Tahar OUAZIB

Sarah DAHAB

*Responsable :*

Yann THIERRY-MIEG

Mai 2014

## Table des matières

Éditeur Xtext et Transformation Divine .....	1
1. Préambule .....	3
2. Objectifs du projet.....	3
3. Outils de développement.....	4
4. Etape et réalisation.....	6
I. Le langage Divine.....	6
1. Description informelle.....	6
2. Méta-Model.....	7
II. L'éditeur Divine .....	13
1. Les Règles De validation .....	13
2. Scoping .....	14
3. Outlines .....	14
4. Quickfix.....	15
III. Transformation vers GAL.....	16
1. Règles de transformation .....	17
5. Benchmark Divine VS GAL .....	21
6. Déploiement et intégration.....	22
7. Conclusion .....	24
8. Annexe.....	25

## 1. Préambule

Ce projet s'inscrit dans le cadre de l'UE (PSTL/PSAR) du deuxième semestre, proposé par L'équipe *MoVe* du Laboratoire Informatique de l'Université Pierre et Marie Curie et supervisé par Yann Thierry- Mieg.

C'est également un projet basé sur le plugin *GAL* (**G**uarded **A**ction **L**anguage), réalisé par une autre équipe d'étudiants lors d'un projet similaire durant l'année 2011-2012.

Le but de ce stage est de réaliser un plugin pour la plateforme *Eclipse* sous forme d'un éditeur de texte jouissant de la puissance de cet IDE. Notre éditeur permettra non seulement la création et l'édition de fichier écrit en *Divine*, mais offrira aussi la possibilité de réaliser une transformation sémantique d'un système *Divine* vers un système *GAL* équivalent.

## 2. Objectifs du projet

La réalisation du projet reposera dans un premier temps sur *Eclipse* et le Framework *Xtext* (cf. section description des langages) puis sur le plugin *GAL* qui offre un ensemble de fonctions indispensables lors de la phase de transformation.

Le stage s'est déroulé en plusieurs étapes. Dans cette section, on soulignera les parties majeures du travail effectué dans un ordre chronologique qui seront présentés plus en détail dans les chapitres qui suivront.

- **Grammaire Xtext :**

C'est la première étape du projet, elle consistait à écrire une grammaire *Xtext* pour le langage *Divine* tout en respectant la spécification et la syntaxe de ses éléments. Voir le site officiel <https://divine.fi.muni.cz/manual.html> ). On s'appuyant sur cette grammaire, *Xtext* est capable de générer un méta-modèle spécifique de chaque élément de notre langage cible (*Divine*).

- **L'éditeur de texte :**

Une fois notre grammaire opérationnelle, on a implémenté un ensemble de fonctionnalités afin d'améliorer notre éditeur de texte et le rendre plus riche. Parmi ces fonctionnalités on trouve :

- Les Règles de validation : un mécanisme qui s'exécute en temps réel et qui vérifie la conformité du programme que l'on a écrit dans le langage cible avec la grammaire que l'on a défini.
- Les Outlines : On a implémenté des méthodes susceptibles de générer automatiquement une vue sous forme d'arbre des éléments du programme, visible dans Eclipse.
- Les QuickFix : un ensemble de règles permettant de proposer des corrections aux erreurs potentielles lors l'écriture du programme.

Remarque : Pour les autres options de l'éditeur, à savoir la coloration syntaxique et l'auto-complétion, qui ne sont pas pertinentes pour ce projet, on s'est contenté de celles générées automatiquement par *Xtext* qui étaient suffisante.

- **La transformation :**

La deuxième partie du projet a pour objectif de réaliser une transformation sémantique des programmes écrits en *Divine* vers leurs équivalents en *GAL*.

- **Evaluation des performances :**

C'est la phase final de notre projet, elle consistait à réaliser une batterie de tests afin d'évaluer les performances de notre programme de conversion, **i.e.** le coût en mémoire, le temps d'exécution, ainsi que la concordance entre le nombre d'états du langage utilisé en entrée (*Divine*) et celui obtenu en sortie (*GAL*).

### 3. Outils de développement

Tout au long du projet on a été amené à manipuler un ensemble d'outils et technologies qui se diversifiaient au fur et à mesure que le projet avançait. Certaines de ces technologies nous ont été bien familières, d'autres beaucoup moins, vu que c'était la première fois qu'on les voyait. Parmi les utilitaires manipulés figures :

- **Eclipse** : C'est l'IDE sur lequel repose le projet vu que les plugins conçus tourneront sur ce dernier. On l'a enrichi avec plusieurs greffons tel que *Xtext* et *Xtend*, *Ecore Documentation* et *Coloane*.

- **Plugin GAL** : qui offre un ensemble de fonctionnalités sous forme d'une librairie sur laquelle on se base pour la transformation Divine vers GAL.
- **Xtext** : permet le développement de la grammaire d'un langage spécifique à un domaine (**DSL: Domain Specific Languages**). Il permet de générer automatiquement un méta-Modèle *Ecore*, un analyseur syntaxique (parser en anglais) basé sur le générateur **ANTLR** ou **JavaCC** à partir du modèle (grammaire) *Xtext* définie.  
Il est également étroitement lié à l'**Eclipse Modeling Framework (EMF)**.
- **EMF (Eclipse Modeling Framework)** : C'est le Framework d'Eclipse qui se charge de la génération du code **java** et/ou **Xtend** correspondant au modèle obtenu à partir de la spécification qu'on a défini pour notre langage cible (Divine).
- **Xtend** : est un langage orienté objet de haut niveau. Syntaxiquement et sémantiquement il tire ses racines du langage Java. Il est compilé en code Java et s'intègre ainsi parfaitement avec toutes les bibliothèques Java.  
Concernant notre projet, *Xtend* était indispensable pour éditer certaines classes générées par *Xtext* permettant ainsi d'implémenter un ensemble de fonctionnalités (Règle de validation, QuickFix, Outline...). Ces fonctionnalités seront détaillées dans les sections suivantes.
- **Maven** : un outil de gestion de projet (compilation, gestion des dépendances, etc...). Dans notre stage, on a utilisé POM (**P**roject **O**bject **M**odel) pour décrire notre projet au sens de *Maven*. Cette description est contenue dans le fichier **pom.xml** présent dans le répertoire de base de chaque plugin (projet).
- **SVN (SubVersion)** : L'utilitaire de gestion de version, indispensable pour assurer le bon déroulement du travail de l'équipe grâce à ces multiples fonctionnalités qui ont le mérite d'être à la fois efficace et intuitives.

Url du dépôt : <https://projets-systeme.lip6.fr/svn/research/thierry/PSTL/DIVINE>

Total de commit effectué : **110**.

- **Teamcity** : le système d'intégration continue qui permet de vérifier s'il y a régression de l'application au cours du développement après chaque édition des ressources partagées.

L'Update site est disponible à l'url suivante :

[http://teamcity-systeme.lip6.fr/guestAuth/repository/download/Pstl\\_Divine/.lastSuccessful/update-site](http://teamcity-systeme.lip6.fr/guestAuth/repository/download/Pstl_Divine/.lastSuccessful/update-site)

## 4. Etape et réalisation

### I. Le langage Divine

#### 1. Description informelle

*Divine* est un langage de Model Checking conçu pour la modélisation et la vérification des systèmes utilisant des composants qui communiquent via des canaux synchrones ou asynchrones.

Un système *Divine* est composé d'un ensemble de déclarations de variables et de processus pouvant communiquer entre eux.

Voici un aperçu des principaux éléments du langage :

- Variables et constantes:

Les variables en *Divine* peuvent être globales au système ou locales à un processus. Elles ont un nom unique et sont de type **byte** ou **int**.

Les constantes se déclarent de la même manière mais sont précédées par le mot clef **const**.

- Les processus :

En *Divine*, les Processus peuvent être considéré comme des sous-systèmes. Ils comportent à leur tour un ensemble de déclaration de variables, de constantes et d'états ainsi qu'un ensemble de transitions (**Trans**).

- Les transitions :

Les transitions sont toujours déclarées dans un processus, elles indiquent le(s) changement(s) d'état(s) du processus, ce qui est possible que lorsqu'une certaine

condition dite garde (**guard**) est respectée. Une transition peut aussi apporter un certain nombre d'effets (**Effect**) sous forme de simples affectations aux variables.

- les Channels & synchronisation :

On distingue deux catégories de **channel** : tamponné et non-tamponné. Ils peuvent être utilisés comme un moyen d'échange de données entre processus (envoi et/ou réception). De ce fait, les canaux pourraient être considérés comme un mécanisme de synchronisation (2 processus différents exécutent chacun une transition en même temps).

- Les Expressions :

Comme dans la majorité des langages de programmation, *Divine* permet de manipuler les expressions arithmétiques et expressions booléennes sous leurs deux formes, binaire et unaire.

La figure suivante montre un exemple de programme écrit en *Divine*.

```
// EXEMPLE DE PROGRAMME DIVINE

//variable globales
int c=1, x1, x2=5;
byte g;
// tableau
int tab [5] = {1,5,2,2,0};

//constantes
const int f =5;
const byte tt =0;
// processus
process a2 {
state Q, R, S;
init Q;
// declaration des transitions
trans
    //garde    //affectations
    Q -> R { guard c<100; effect x2 = c; },
    R -> S { effect x2 = x2 + c; },
    S -> Q { effect c = x2; };
}
```

## 2. Méta-Model

Nous allons maintenant voir comment est construite notre grammaire *Xtext* pour chacun des éléments du langage *Divine*. Nous utiliserons une approche basée sur la méta-modélisation pour appuyer les explications fournies.

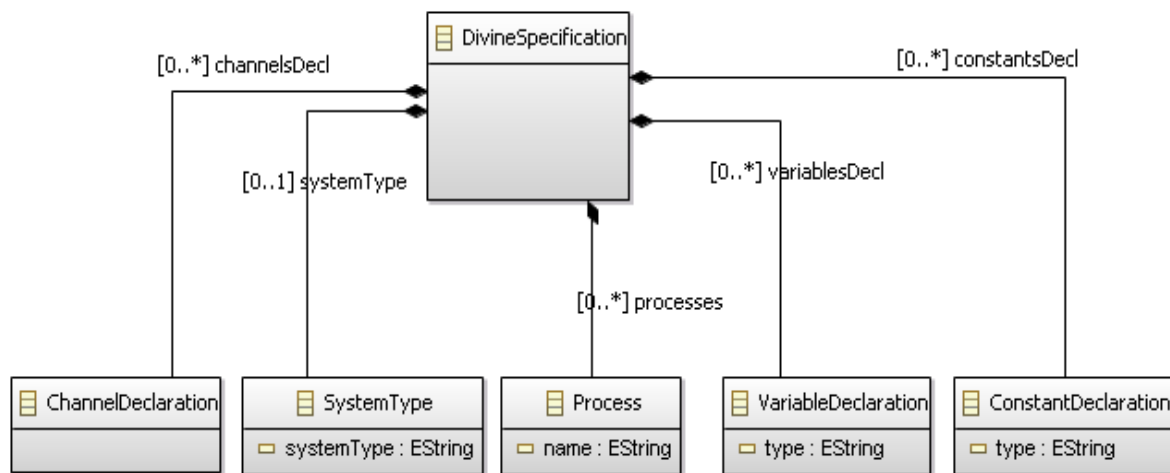
Les différents aspects des langages *Divine* et *Xtext* ont été présentés dans les sections précédentes. La documentation officielle est également disponible sur leurs sites respectifs :

*Xtext*: “<https://www.eclipse.org/Xtext/documentation.html>”

*Divine*: “<https://divine.fi.muni.cz/manual.html>”

#### a. Le système *Divine*:

Cette entité représente la structure globale d'un programme *Divine*, tous les programmes écrit en *Divine* devraient s'appuyer sur le méta-modèle ci-après :



La figure ci-dessus représente le méta-modèle construit à partir de la règle *DivineSpecification* de notre grammaire *Xtext* :

```

DivineSpecification :
(
    variablesDecl+=VariableDeclaration |
    constantsDecl+=ConstantDeclaration |
    channelsDecl+=ChannelDeclaration
)*
processes+=Process*
systemType=SystemType
;
  
```

Il montre bien que la méta-classe *DivineSpecification* englobe les définitions des différents type d'entité du système, tel que les processus (méta-classe **Process**) ainsi que

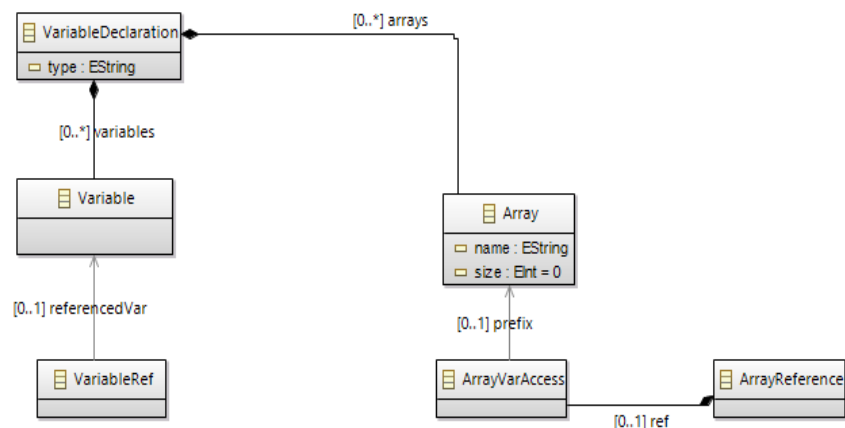


les différents type de données tel que les variables (Meta-classe **VariableDeclaration**), les constantes (Meta-classe **ConstantDeclaration**) ... .

### *b. Les structures de données*

*Divine* permet de manipuler plusieurs types de données comme les variables, les constantes et les tableaux.

#### 1. Les variables :

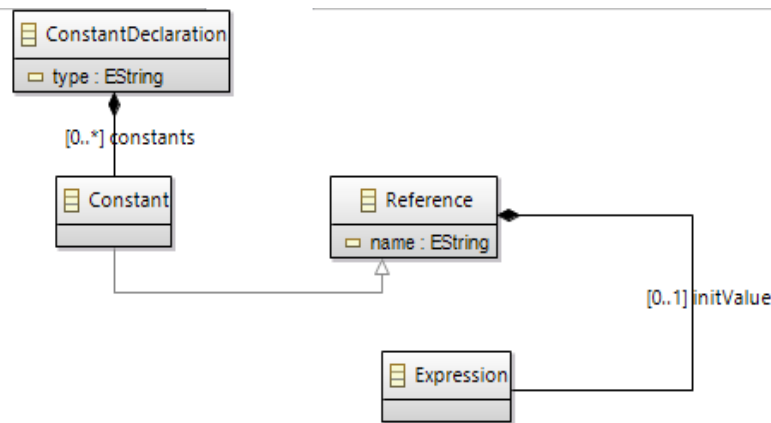


Comme l'indique le Modèle **VariableDeclaration** de la figure ci-dessus, la déclaration des variables se décline en un ensemble de :

- Tableau (Méta-classe **Array**), qui sera identifié par son nom (unique) ainsi que sa taille.
- Variable (Méta-classe **Variable**), des variables classiques portant un nom.

Ces deux déclarations sont typées soit par int ou byte.

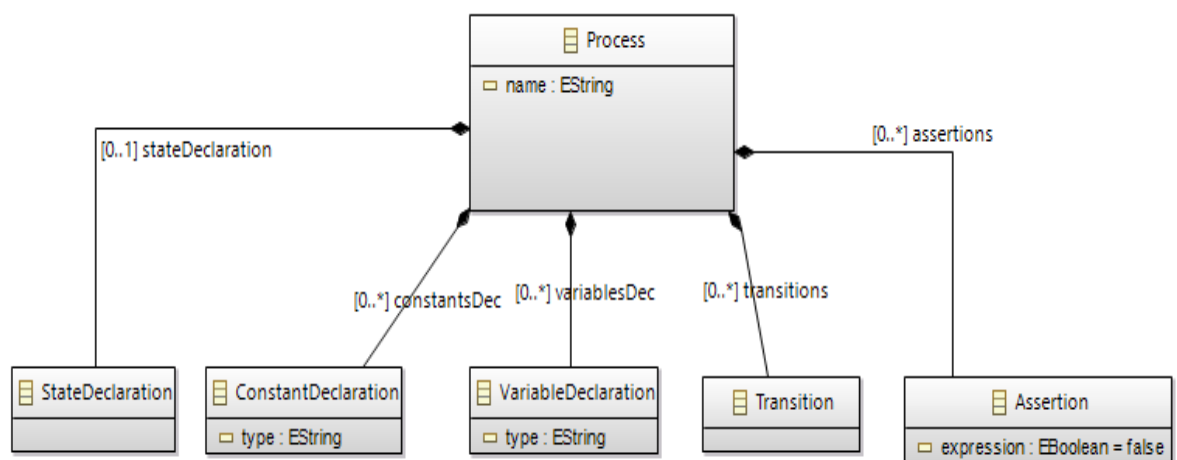
## 2. Les constantes



La méta-classe **ConstanteDeclaration** se décline en un ensemble de déclarations de constantes (classe **Constante**). On voit aussi qu'une constante est typée et porte un nom pour la référencer (**Reference**), ces constantes peuvent être initialisées soit par une valeur littérale soit par une expression évaluable.

### c. Les processus

Le diagramme qui suit illustre la structure d'un processus (méta-classe **Process**). Un processus comporte donc un ensemble de déclarations de variables, de constantes, ayant une portée locale, mais aussi une déclaration d'un ensemble d'états (méta-classe **StateDeclaration**). Il est caractérisé par son nom unique et un ensemble de transitions. (cf. Transition).



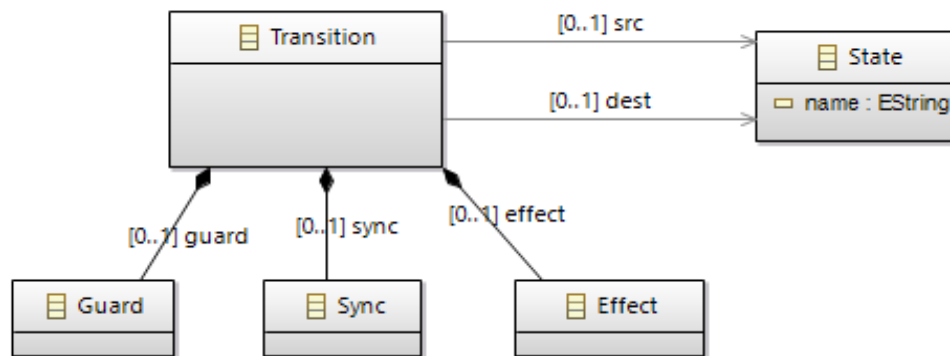
Remarque :

À noter que les variables déclarées dans le système *Divine*, à l'exception de celles déclarées dans les processus sont toutes considérées comme étant des variables globales.

#### d. Les transitions :

Une transition est responsable des changements opérés sur les états. Concrètement elle fait passer un processus d'un état (méta-classe **State**) « X » vers ( $\rightarrow$ ) un nouvel état « Y », tout en réalisant un ensemble d'effets de bord sous forme d'assignements (méta-classe **Effect**). Cependant ceci n'est possible que si la condition de la garde (classe **Guard**) est satisfaite.

Le modèle UML suivant représente une transition en *Divine*.



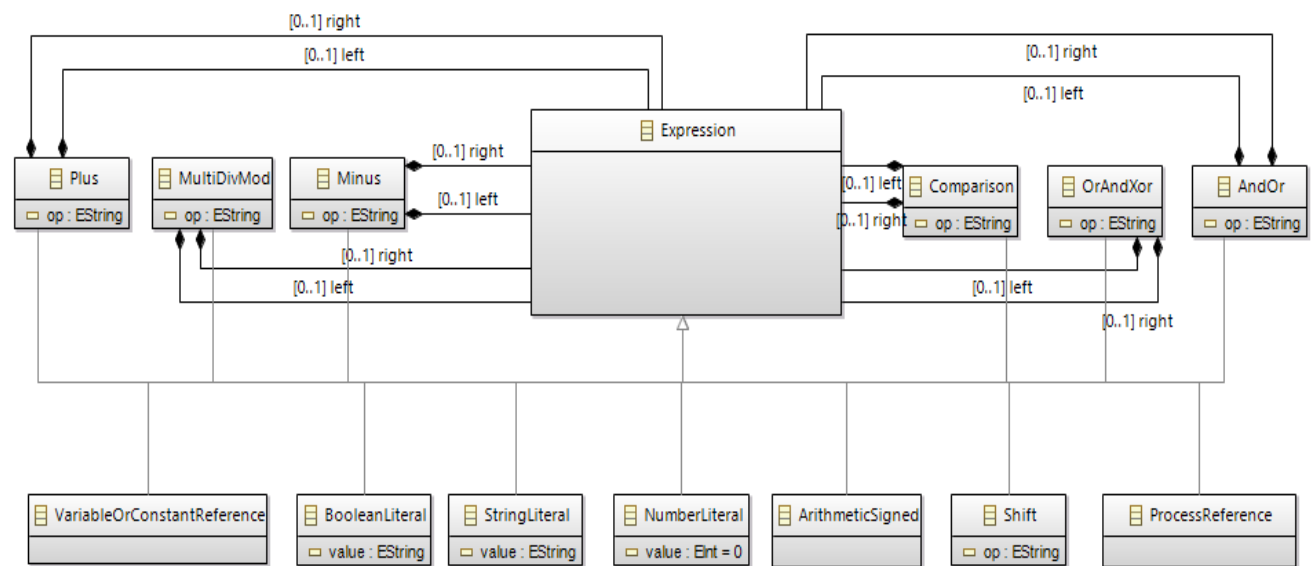
#### Remarque :

Les transitions ont un nom unique dans le système et sont uniquement déclarées dans un processus.

#### e. Les Expressions :

Les règles représentant les expressions étaient sans doute les règles les plus laborieuses qu'on a eu à définir. Il fallait gérer les priorités des opérations ainsi que leurs distributivités à gauche.

Le diagramme ci-après, montre la représentation qu'on a choisie pour définir les expressions Divine dans notre grammaire **Xtext**.



Comme le montre ce diagramme, il s'agit d'un pattern Composite. Cette modélisation nous permet ainsi de manipuler deux Grandes familles d'expressions :

### 1. Les expressions arithmétiques

Ce sont des expressions formées avec les opérateurs arithmétiques classiques.

Ce genre d'expression se divise en plusieurs sous types :

- Les expressions arithmétiques binaires comportent une opération (type String) ainsi que deux opérandes qui sont aussi des expressions. C'est le cas de l'addition, soustraction etc... (cf. Les méta-classe **Plus**, **Minus**, **MultiDivMod**...)
- Les expressions arithmétiques unaires, pour la représentation des entiers négatifs par exemple la Classe **ArithmeticSigned**.
- Les constantes qui ont une valeur(**NumberLiteral**).

### 2. Les expressions Booléennes:

Les expressions booléennes suivent le même pattern que les expressions entières, on y trouve :

- des expressions binaires comme la comparaison entre deux expressions entières (méta-classe **Comparaison**), mais aussi les opérateurs logique usuels tel que le «ET », le « OU », modélisés par **AndOr**.
- des expressions booléennes binaires
- Un type constant, représenté par la classe **BooleanLiteral**, qui contient un champ indiquant sa valeur (**true /false**).

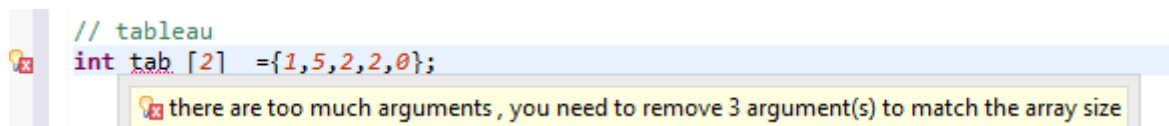
Remarque : Pour plus de détail sur la façon dont les modèles ci-dessus ont été générés, ainsi que le code *Xtext* correspondant il faut se référer aux règles correspondantes de la grammaire qui se situe dans le fichier «[fr.lip6.move.divine/src/fr/lip6/move/Divine.xtext](http://fr.lip6.move.divine/src/fr/lip6/move/Divine.xtext)».

## II. L'éditeur Divine

### 1. Les Règles De validation

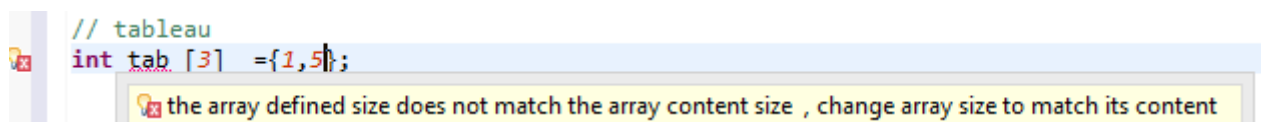
L'ensemble des règles de validation définies vise à assurer :

- Une cohérence entre la taille des tableaux et le cardinal de l'ensemble de leurs éléments lors de la déclaration et l'initialisation.



```
// tableau
int tab [2] = {1,5,2,2,0};
```

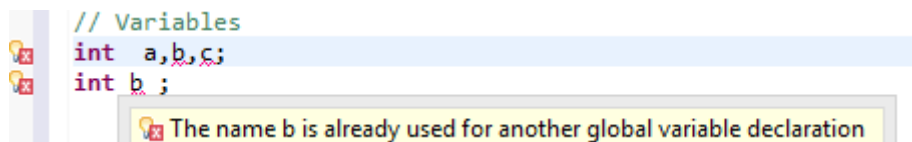
there are too much arguments , you need to remove 3 argument(s) to match the array size



```
// tableau
int tab [3] = {1,5};
```

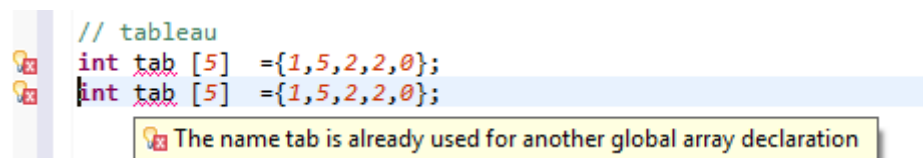
the array defined size does not match the array content size , change array size to match its content

- L'unicité des noms de variables, tableaux et constantes.



```
// Variables
int a,b,c;
int b ;
```

The name b is already used for another global variable declaration



```
// tableau
int tab [5] = {1,5,2,2,0};
int tab [5] = {1,5,2,2,0};
```

The name tab is already used for another global array declaration

L'ensemble de ces règles sont définies dans un fichier annexe ([DivineValidator.xtend](#)) fournie par *Xtext*. Voici un extrait de la méthode chargée de vérifier l'unicité de nom de variables.

Remarque : l'annotation `@Check` indique au Framework que la méthode qui suit est une règle de validation à vérifier.

```

// check if a variable name is duplicated and rise an error
@Check
def checkVarNameNotDuplicated(Variable v) {

    var fr.lip6.move.divine.divine.Process p = getProcess(v);
    if (p != null) {
        for (VariableDeclaration varr : p.variablesDec) {
            for (Variable vari : varr.variables) {
                if (v != vari && v.name == vari.name) {
                    error(
                        "The name " + v.name + " is already used for another local variable declaration ", /* Error Message */
                        v, /* Object Source of Error */
                        DivinePackage.Literals.REFERENCE_NAME, /* wrong Feature */
                        DUPLICATED_NAME
                    );
                }
            }
        }
    }
}

```

## 2. Scoping

*Xtext* fournit un mécanisme de référencement, le problème est que les références au niveau syntaxique sont vues comme des identifiants (au même titre que les noms de variables) et sont résolues après que l'arbre syntaxique de la grammaire ait été généré. Dans notre projet, on a eu recours au mécanisme de scope pour résoudre les problèmes d'accès aux états des processus dans le corps de ces derniers.

Voici un extrait du code :

```

class DivineScopeProvider extends AbstractDeclarativeScopeProvider {

    def scope_ProcessStateAccess_state(ProcessStateAccess psa, EReference ref) {
        scopeFor(psa.process.stateDeclaration.states)
    }
}

```

Les règles de scoping sont définies quant à elles dans un autre fichier annexe (**DivineScopeProvider.xtend**), généré automatiquement par *Xtext* aussi.

## 3. Outlines

Les *Outlines* permettent de visualiser la structure des programmes, c'est comme un sommaire. Il a pour vocation de faciliter la navigation dans le code. *Xtext* génère par défaut des *Outlines* qui portent le nom des labels (ID), des nœuds simples de la grammaire définie. Cependant pour les nœuds composés *Xtext* produit des balises <unnamed>.

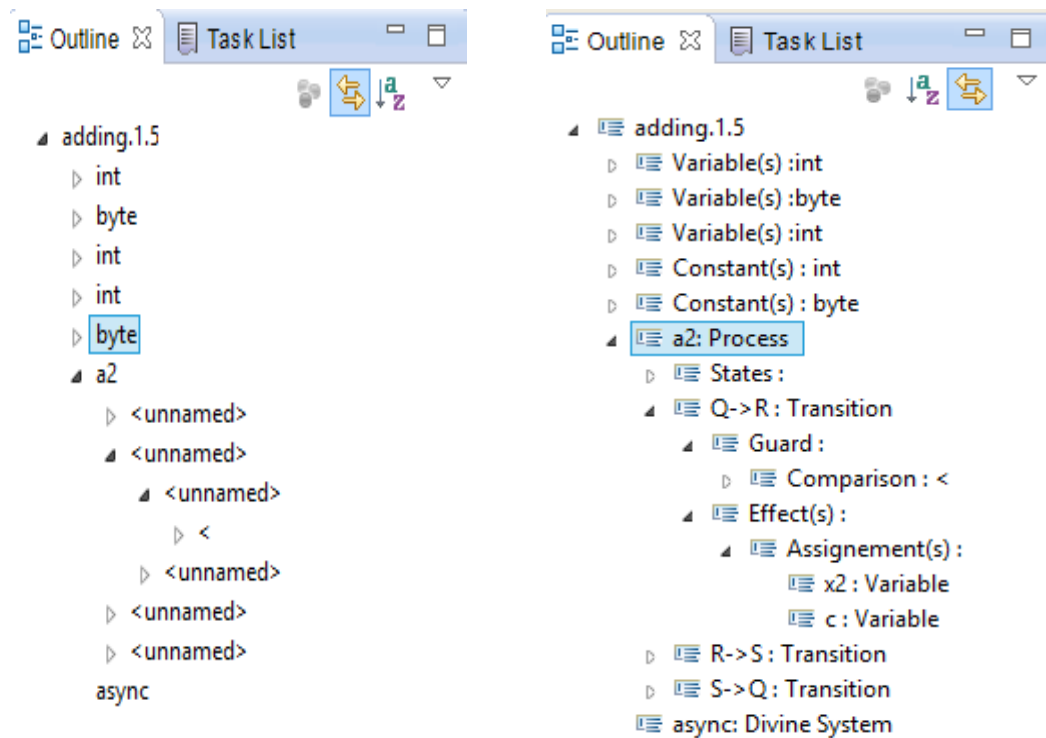
Dans cette optique, on définit un ensemble de méthodes qui ont pour but de guider *Xtext* à traiter tous les nœuds possibles de notre grammaire et générer ainsi des *Outlines* complets.

Ce fichier regroupe un ensemble de méthodes qui se charge de renvoyer une chaîne de caractère adéquate qui sera affichée dans les *Outlines*.

Pour illustrer, voici des captures d'écran correspondant à l'aperçu des *Outlines* avant et après la définition des règles de leur génération pour un même programme.

AVANT

APRES

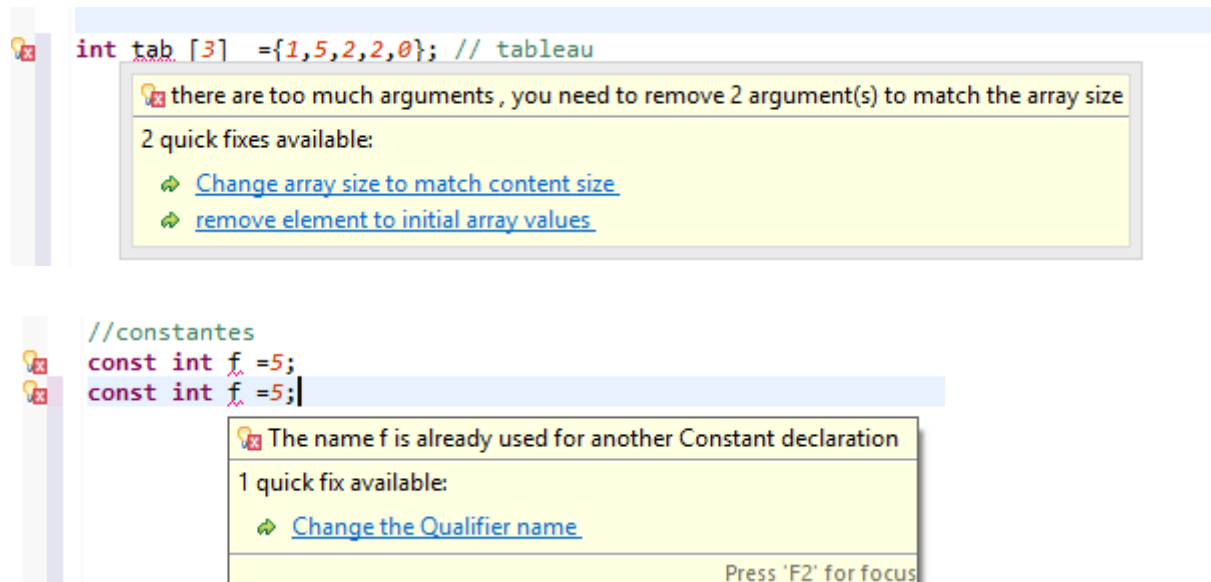


#### 4. Quickfix

Les *Quickfix* ont pour vocation de proposer une correction à certaines erreurs qui peuvent survenir lors de l'édition du programme.

Les deux figures suivantes illustrent deux exemples de *Quickfix* implémentés.

Le premier permet de corriger les erreurs liées à l'absence de cohérence entre la taille d'un tableau et le cardinal des éléments qui lui sont affectés, le second permet de renommer les éléments ayant le même nom.



Le code correspondant aux *Quickfix* se trouve dans le fichier

«**DivineQuickFixProvider.xtend**» du package « **fr.lip6.move.divine.ui.quickfix** »

### III. Transformation vers GAL

Le langage GAL (**G**uarded **A**ction **L**anguage) est un langage de programmation dédié à la vérification formelle de systèmes concurrents, toutefois *GAL* possède une syntaxe moins riche que celle de *Divine* (pas de processus, channel, variable locale).

Pour plus de précision sur la spécification de *GAL*, cf. :

<http://move.lip6.fr/software/DDD/files/gal.pdf>

Voici un exemple d'un système écrit en *GAL* :



```

gal system
{
    // Variables
    int v = 5;
    array[2] tab = (1, 2);

    // Transitions
    transition t1 [ var > 9 ] {
        tab[0] = tab[1] * (l.peek()+3);
        var = var * 5;
    }

    transition t2 [ l.peek() == 23 ] label a {
        tab[1] = 0;
    }

    TRANSIENT = False;
}

```

## 1. Règles de transformation

Cette partie consistait à réaliser une transformation sémantique du langage *Divine* vers le langage *GAL* en s'appuyant à la fois sur le Meta-Model qu'on a obtenu grâce à notre grammaire mais aussi sur un ensemble de fonctions offertes par *GAL* via le plugin *Coloanne*.

L'ensemble des règles des transformations sont définies dans le paquetage « **togal** », qui est composé de deux classes principales :

- « **Converter.java** » : ce fichier regroupe un ensemble de fonctions utilitaires, capable notamment de faire des conversions de certains types d'éléments *Divine* en leur équivalent en *GAL*, stocker les variables et vérifier le typage des éléments.
- « **DveToGalTransformer.java** » : ce fichier est composé d'un certain nombre de fonctions réalisant la transformation *Divine* vers *GAL*.

Comme on l'a déjà mentionné, la grammaire de *Divine* est beaucoup plus riche que celle de *GAL*, c'est pourquoi nous avons dû élaborer quelques règles de transformations en tenant comptes des contraintes imposées par chaque type du langage *GAL*. Pour mieux comprendre le processus de transformation, nous allons montrer pour chaque élément du langage *Divine* le résultat de sa transformation vers *GAL*.

### A. Variables:

La transformation des variables est réalisée en deux étapes :

- **buildGlobalVars()** : cette fonction agit sur les variables globales déclarées au tout début du programme *Divine*
- **buildLocalVars()** : qui comme son nom l'indique agit sur les variables locales (déclarées aux seins des processus).

les etapes de transformation d'une variable *Divine* vers *GAL* reste les memes pour les deux methodes et elle consiste a :

1. instancier une variable *GAL* a l'aide de la fabrique(factory) *GalFactory*.
2. definir le nom de la variable en le prefixant par « local\_ » dans le cas d'une variable local et « glob\_ » si c'est une variable globale.
3. si la variable en question porte une valeur (une expression), notre variable *GAL* portera a son tour le resultat de la conversion en expression entière *GAL* de notre expression en *Divine*.
4. Enfin on mappe la variable afin de pouvoir agir sur les references de celles-ci en ajoutant le couple <variable *Divine*, sa reference en *GAL*>.

### B. Constante

Ce type est manipulé d'une façon particulière en *GAL*, car toutes les constantes du programme sont passées en paramètre au système. De ce fait, la conversion d'une constante *Divine* vers *GAL* se traduit par :

1. L'instanciation d'un paramètre en *GAL* grâce à la factory **GalFactory** et lui définir un nom (le même que celui de la constante en *Divine*).
2. La valeur initiale du paramètre est la valeur résultant de l'interprétation du résultat de la conversion en expression entière de l'expression initial de *Divine*. Car contrairement à *Divine*, une constante (paramètre en *GAL*) doit être impérativement initialisé par un entier (ne supportant pas d'expression).
3. Pour finir on ajoute le paramètre obtenu à la liste des paramètres du programme.

### C. Transitions et Assertions :

À chaque transition *Divine*, on associe une transition *GAL* qui porte un nom spécifique qui indique à quel processus *Divine* elle appartient.

1. On définit une condition (expression booléenne) en réalisant une comparaison entre l'état actuel du processus auquel appartiennent la transition et l'état source de celle-ci.
2. Si la transition *Divine* a une garde : la garde de la transition *GAL* sera formée par la composition avec des conjonctions (type **AND** en *GAL*) de notre condition calculée avec les gardes de la transition *Divine*.
3. On met à jour l'état du processus courant en affectant l'état destination de la transition à son état courant (type **Assignement** en *GAL*).
4. À chaque affectation en *Divine* (**Effect**) on fait correspondre une **Action** en *GAL* sous forme d'un assignement (**Assignement**).

### D. Channels

Comme il n'y a pas de channel en *GAL*, on a été amené à trouver un substitut qui n'est rien d'autre que les transitions. Cette transformation est réalisée par la méthode **buildchannel** comme suit :

1. L'instanciation d'une transition en *GAL* grâce à la factory **GalFactory**. Cette transition portera le nom du channel préfixé par « **chan\_** »
2. Cette transition comporte une garde toujours passable (sa condition est toujours vraie) et un label spécifique pour pouvoir les différencier des autres vraies transitions.
3. Suivant le type du channel exprimé en *Divine*, on dote notre transition de deux actions *GAL* portant deux labels distincts, qui sont des appels (type **Call**) pour traiter les deux cas d'envoi et de réception de données.

### E. Processus

Un processus *Divine* est représenté par une variable *GAL* qui porte le même nom que celui du processus suivi du nom « **\_state** », elle porte comme valeur l'indice de l'état initial du processus. Les processus sont ensuite mappés sous forme de couple <Processus, variable *GAL* calculée>.

Les figures suivantes montrent un exemple de transformation *Divine* vers *GAL*.

```

int c=1, x1, x2;
const int c1 = 0;
const int c2 = 5;
int tab[4] = {1, 2, 3, 6};

process a1 {
state Q, R, S;
init Q;
trans
  Q -> R { guard c<100; effect x1 = c; },
  R -> S { effect x1 = x1 + c; },
  S -> Q { effect c = x1; };
}

process a2 {
state Q, R, S;
init Q;
trans
  Q -> R { guard c<100; effect x2 = c; },
  R -> S { effect x2 = x2 + c; },
  S -> Q { effect c = x2; };
}

system async;|

```

```

gal adding_1_5 ($c1 = 0, $c2 = 5) {
int glob_c = 1 ;
int glob_x1 = 0 ;
int glob_x2 = 0 ;
int a1_state = 0 ;
int a2_state = 0 ;
array [4] glob_tab = (1, 2, 3, 6) ;
transition a1_t1_Q_R [glob_c < 100 && a1_state == 0] {
  a1_state = 1 ;
  glob_x1 = glob_c ;
}
transition a1_t2_R_S [a1_state == 1] {
  a1_state = 2 ;
  glob_x1 = glob_x1 + glob_c ;
}
transition a1_t3_S_Q [a1_state == 2] {
  a1_state = 0 ;
  glob_c = glob_x1 ;
}
transition a2_t1_Q_R [glob_c < 100 && a2_state == 0] {
  a2_state = 1 ;
  glob_x2 = glob_c ;
}
transition a2_t2_R_S [a2_state == 1] {
  a2_state = 2 ;
  glob_x2 = glob_x2 + glob_c ;
}
transition a2_t3_S_Q [a2_state == 2] {
  a2_state = 0 ;
  glob_c = glob_x2 ;
}
}

```

## 5. Benchmark Divine VS GAL

Afin de vérifier le bon fonctionnement de notre transformation ainsi que les performances de celle-ci, on a réalisé un ensemble de tests sur des fichiers résultants de cette transformation. Ces tests ont été évalués sur trois critères principaux : le temps d'exécution, le coût en mémoire et le nombre d'états atteignables par chaque programme. Le tableau ci-dessous représente une comparaison des résultats obtenus par les fichiers source *Divine* (fournis) et les fichiers *GAL* résultants de la transformation de ces derniers.

### Remarque :

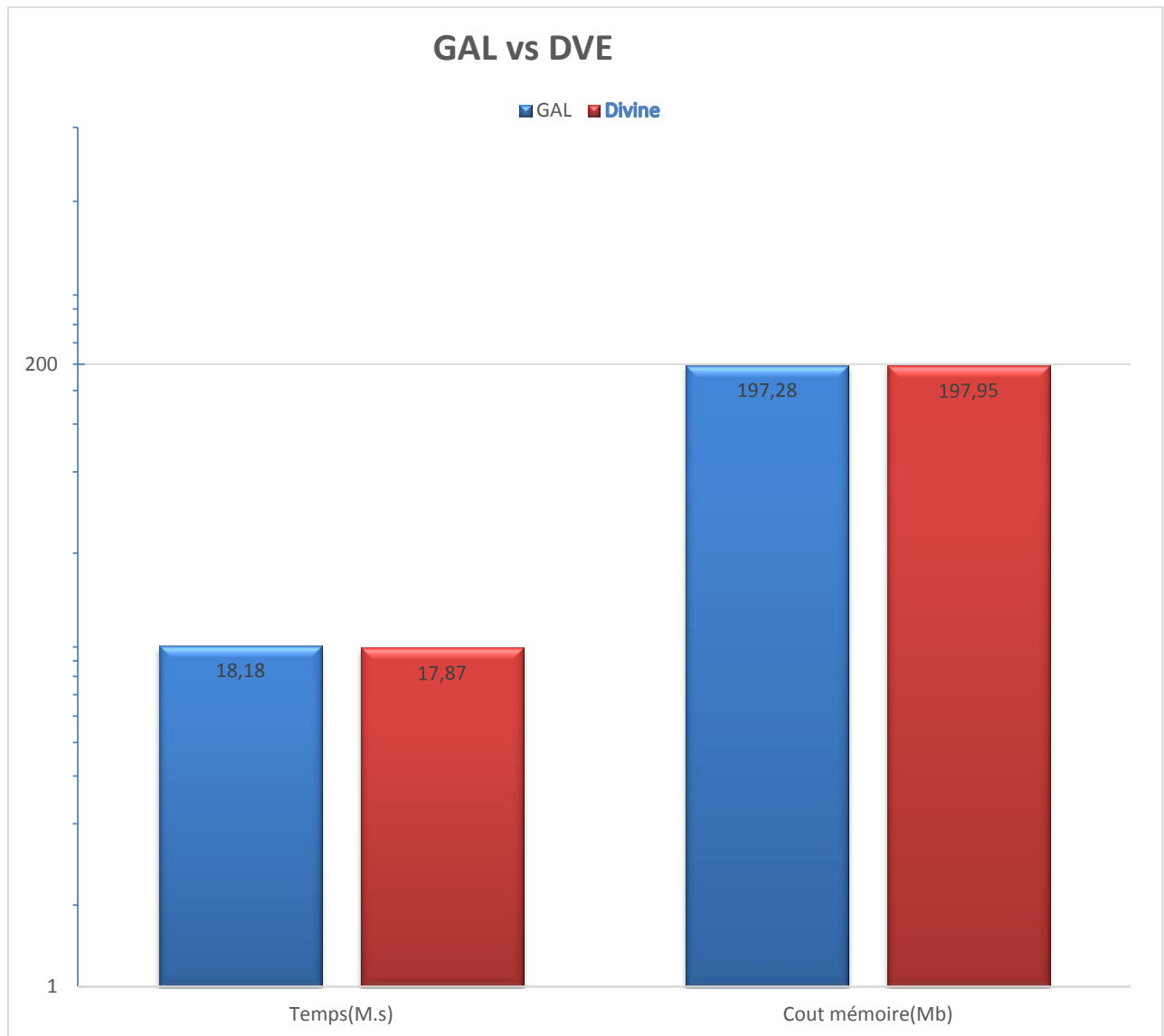
Les fichiers *GAL* employés sont des fichiers « flatten » i.e. simplifié grâce à une fonctionnalité offerte par le plugin *Coloane*.

<u>Programmes</u>	<u>Temps d'exécution(Ms)</u>	<u>Coût en mémoire(Kb)</u>	<u>Nombre d'états atteignables</u>
Adding1.5.dve	6.38	51732	206310
Adding1.5.flat.gal	6.45	51544	206310
Backery.4.dve	3.65	46216	157003
Backery.4.flat.gal	3.64	46080	157003
exit.1.dve	1.16	16588	3239334
exit.1.flat.gal	1.21	16340	3239334
Ficher2.dve	0.12	4585	21733
Ficher2.flat.gal	0.12	4256	21733
Lamporte.2.dve	1.53	19248	110920
Lamporte.2.flat.gal	1.54	19032	110920
Peterson.4.dve	5.03	64336	1119560
Peterson.4.flat.gal	5.22	64772	1119560

### Analyse :

Le tableau montre bien que le nombre d'états atteignables par le système représenté en *GAL* est le même que celui du système décrit en *Divine*, ce qui prouve que notre transformation est correcte.

On remarque aussi qu'il y a une légère différence au niveau du temps d'exécution, où les programmes *Divine* finissent légèrement plus vite que les programmes *GAL*. Cependant ces derniers sont moins coûteux en mémoire.

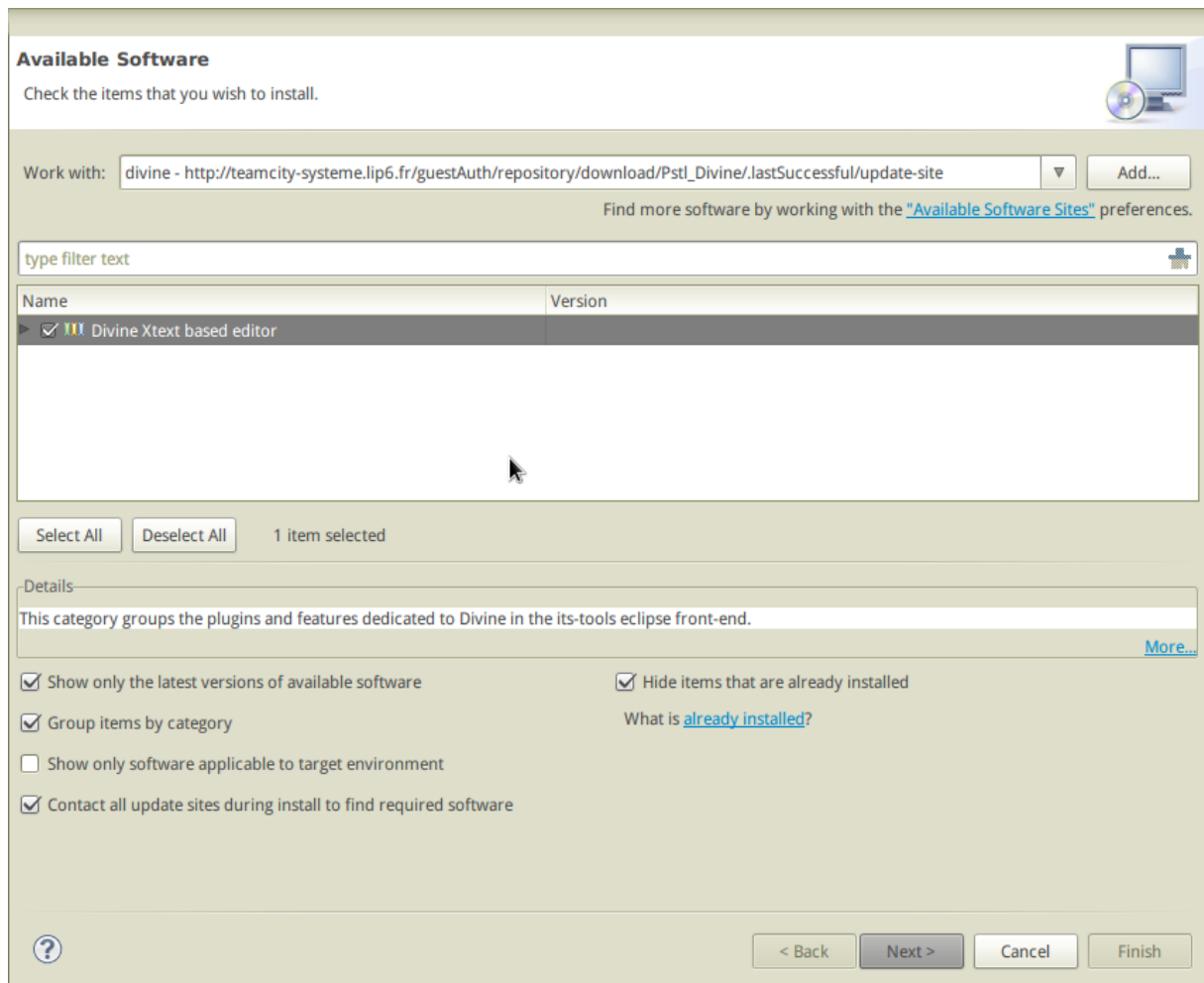


## 6. Déploiement et intégration

Grâce à notre encadrant, notre plugin est disponible en ligne via la plateforme *teamCity* et la notion d'update-site fourni par *Eclipse*. Celui-ci est ainsi automatiquement mis à jour à chaque commit.

L'intégration dans *Eclipse* se fait avec l'update-site depuis le menu « Help/install New Software » en renseignant l'url de dépôt suivante (cf. figure ci-dessous):

[http://teamcity-systeme.lip6.fr/guestAuth/repository/download/Pstl\\_Divine/.lastSuccessful/update-site](http://teamcity-systeme.lip6.fr/guestAuth/repository/download/Pstl_Divine/.lastSuccessful/update-site)



## 7. Conclusion

Le projet PSTL/PSAR a été une expérience unique en son genre car pour la première fois nous avons vraiment connu le travail en groupe, ce qui ne paraissait pas évident au début mais qui s'est finalement avéré très positifs.

De plus, cela nous a permis de prendre conscience de l'ampleur du travail que nécessite le développement d'un plugin qui à première vue nous paraissait pas aussi complexe que ça l'a été.

Ainsi par ces travaux nous avons découvert de nouveaux outils, des méthodologies de développements plus efficaces et professionnelles que nous avons pu mettre en œuvre et nous permettre de consolider nos acquis.

Grâce à l'aide et au soutien de notre encadrant Yann Thierry-Mieg, nous avons pu mener à bien ce projet au bout des objectifs fixés dans une ambiance agréable et positive.

Nous tenons à remercier notre encadrant Yann Thierry-Mieg pour sa présence et sa bonne humeur tout au long de ce projet.



## 8. Annexe

Les documentations des langages de programmation :

- La documentation Xtext:  
<http://www.eclipse.org/Xtext/documentation.html>
- La documentation Xtend:  
<http://www.eclipse.org/xtend/documentation/index.html>
- Java Doc:  
<http://docs.oracle.com/javase/7/docs/api/>
- GAL :  
<http://move.lip6.fr/software/DDD/gal.php>
- Divine et sa spécification :  
<https://divine.fi.muni.cz/manual.html#the-dve-specification-language>

Liens utiles :

- Les expressions en Xtext:  
<http://xsemantics.sourceforge.net/xsemantics-documentation/Expressions-example.html>
- Code source pour comprendre la grammaire:  
<http://divine.fi.muni.cz/trac/browser/divine/dve/>

Autre liens utiles :

- Le site du Laboratoire :  
<http://move.lip6.fr/software/DDD/>
- TeamCity :  
<http://teamcity-systeme.lip6.fr/login.html>
- Modèles Utiliser pour les tests :  
[https://projets-systeme.lip6.fr/svn/research/libddd/libits/trunk/tests/test\\_models](https://projets-systeme.lip6.fr/svn/research/libddd/libits/trunk/tests/test_models)  
[https://projets-systeme.lip6.fr/svn/research/libddd/libits/trunk/perfs/test\\_models](https://projets-systeme.lip6.fr/svn/research/libddd/libits/trunk/perfs/test_models)