

# Rapport du Projet STL SPIN, from Promela to Gal

Adrien Becchis, Fjorilda Gjermizi, Julia Wisniewski

May 16, 2014

*Projet de Master 1 STL UPMC*  
auprès de Mr Yann Thierry-Mieg.

# 1 Introduction

Le projet "Promela To Gal" a pour but de mettre en place une infrastructure pour pouvoir traduire des descriptions de processus concurrents écrits en **Promela** pour le simulateur Spin, vers un autre paradigme de description de système, avec **GAL** (Guarded Action Language), un langage relativement simple de modélisation. Ceci nous permettra de comparer les résultats des deux grandes familles de modèles checking, respectivement les méthodes explicites et celles symboliques.

Ce faisant, l'utilisation du framework de développement de langages **Xtext** nous a permis de réaliser un éditeur enrichi du langage Promela intégré à Eclipse.

Nous reviendrons dans ce rapport plus en détails sur le besoin et les langages sources, cibles, support, avant de passer en revue nos différentes contributions.

Nous avons regroupé nos contributions en **quatre catégories**:

- La première contribution a été de développer une grammaire et un métamodèle pour le langage Promela, et a servi de base pour toutes les autres contributions.  
Nous reviendrons sur les différents choix faits lors de l'élaboration de la grammaire, présenterons le Métamodèle résultant avant d'en discuter certaines limites.
- Une grammaire et un Métamodèle, donc un ensemble de règles syntaxiques, ne sont pas suffisants pour s'assurer qu'un programme est correct. C'est pour cela que nous reviendrons sur notre deuxième contribution qui est le développement d'outils pour vérifier les programmes. Pour cela une analyse du typage, ainsi qu'un certain nombre d'analyses statiques ont été réalisées pour s'assurer que le code Promela soit correct.
- S'étant appuyé sur le framework **Xtext**, nous avons parallèlement pu développer un éditeur eclipse doté des désormais indispensables colorisation syntaxique, auto-complétion, marquage d'erreur, etc. Nous avons personnalisé l'éditeur standard, mais aussi ajouté d'autres fonctionnalités ne venant pas par défaut avec **Xtext**.  
Nous reviendrons sur nos différentes actions afin de rendre cet éditeur à la fois plus fonctionnel et plaisant à utiliser.
- L'aboutissement du projet était la transformation de code **Promela** vers du code **Gal**. Dans un premier temps, nous nous pencherons sur l'infrastructure mise en place pour la traduction, ainsi que le "mapping", l'appariement entre les concepts des deux langages.

Un plug-in **eclipse** se liant à l'éditeur a aussi été mis au point, ainsi qu'une génération de graphes décrivant les automates déduits du code **Promela**, représentation intermédiaire nécessaire à la traduction.

Nous reviendrons ensuite sur les résultats concrets obtenus sur le "banc de test" Beem qui a servi de base de référence.

Une fois ces quatre contributions amplement détaillées, nous tirerons un bilan de l'état du projet, mais aussi de nos expériences personnelles respectives. Pour finir, nous aborderons les perspectives, c'est-à-dire les améliorations qui pourraient être apportées à nos contributions.

*Bonne lecture*

# Plan

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Présentation des technologies utilisées.</b>	<b>4</b>
2.1	Promela/Spin: le langage source . . . . .	4
2.2	GAL: le langage cible . . . . .	7
2.3	Xtext et Xtend : Langages et framework support . . . . .	7
<b>3</b>	<b>Méta-modèle et grammaire de Promela</b>	<b>10</b>
3.1	Elaboration de la grammaire . . . . .	10
3.2	Meta Modèle de Promela . . . . .	12
3.3	Limites du MM . . . . .	14
<b>4</b>	<b>Analyse Statique de Promela</b>	<b>15</b>
4.1	Typage . . . . .	15
4.2	Validation . . . . .	16
<b>5</b>	<b>Éditeur avancé Promela</b>	<b>17</b>
5.1	Quickfixes . . . . .	17
5.2	Outline and Labelling . . . . .	18
5.3	Templates . . . . .	18
5.4	Formatting . . . . .	19
<b>6</b>	<b>Transformation ToGal</b>	<b>20</b>
6.1	Objectif . . . . .	20
6.2	Mapping entre concepts Promela et Gal . . . . .	21
6.3	Architecture . . . . .	21
6.4	Visualisation du Graphe de flot de contrôle . . . . .	25
6.5	Résultats. . . . .	26
<b>7</b>	<b>Conclusion</b>	<b>28</b>
7.1	Bilan du projet . . . . .	28
7.2	Bilans Personnels . . . . .	28
7.3	Perspectives . . . . .	29
<b>8</b>	<b>Annexes</b>	<b>30</b>
8.1	Plugins Spin2Gal . . . . .	30
8.2	Exemples de Transformation . . . . .	30
8.3	Résultats Benchmark Beem . . . . .	34
8.4	Outils utilisés. . . . .	36
8.5	Références . . . . .	36

## 2 Présentation des technologies utilisées.

Dans cette partie, nous allons présenter les différents langages et technologies utilisées.

Nous décrivons dans un premier temps le langage source *Promela* et son outil *SPIN*, avant de parler du langage cible *GAL* vers lequel on va traduire. Ensuite nous présenterons le framework *Xtext* qui a été utilisé pour réaliser un MétaModèle du langage *Promela*; MM qui nous a servi ensuite à réaliser la transformation vers un MM *GAL*, préalablement développé dans un ancien *PSTL* aussi avec *Xtext*.

### 2.1 Promela/Spin: le langage source

#### 2.1.1 Intro

*Promela* (PROtocol MEta Language) est un langage pour décrire des processus communicants afin d'en vérifier les propriétés. Ce langage de spécification de systèmes asynchrones a été introduit par *Gerard J. Holzmann*, et les modèles décrits sont analysés à l'aide du vérificateur *SPIN* (Simple Promela Interpreter)

Cet outil a reçu le prestigieux *Software System Award* décerné par l'ACM (Association for Computing Machinery) en 2001. Il est présenté comme l'outil le plus populaire pour la vérification des systèmes concurrents et a été notamment utilisé dans des logiciels utilisés en téléphonie, médecine ou exploration spatiale.

#### 2.1.2 Spin

L'outil *Spin* va permettre d'analyser un modèle *promela*, et d'éventuellement établir sa correction. Il va donc notamment de vérifier l'absence de deadlock (états puits), la valeur de certains invariants ou autres propriétés décrites lors de la spécification. La vérification se fait grâce à la réalisation aléatoire d'exécution, ou l'exploration exhaustive de l'espace d'état du système.

L'outil ne réalise pas directement la vérification du modèle. Il va, à la place, générer du code C pour réaliser celle-ci, ce qui est censé permettre d'économiser de la mémoire, de bénéficier des optimisations du compilateur C, et d'incorporer directement du code C. (ce qui néanmoins retarde la détection de certaines erreurs).

De nombreuses options permettent d'accélérer et alléger le processus.

Cet outil est librement accessible depuis 1991, et en est à sa version 6 depuis 2010 (celui-ci évolue conjointement avec le langage puisqu'il en est en quelque sorte l'implantation de référence).

#### 2.1.3 Langage

*Promela* permet de spécifier des processus, qui représentent les entités en concurrence d'un système distribué, et un environnement d'exécution via variables et canaux. Le comportement des processus est décrit à l'aide d'instructions et de structures de contrôles. Nous allons présenter plus en détails les principaux concepts et constructions du langage.

**L'environnement d'exécution.** Le contexte d'exécution d'un modèle *Promela* repose sur deux types de constructions dont l'état va définir l'état global du système. Celles-ci peuvent être globales, ou avoir une portée restreinte à un processus (et ainsi modéliser des données internes à un processus).

C'est de cet environnement que sera déduit l'espace des états du système.

**Les variables Mémoires.** Il s'agit des variables classiques des langages de programmation impératif, et correspondent à l'équivalent d'une case mémoire.

Celles-ci sont typées et doivent être déclarées avant utilisation, si aucune valeur initiale est déclarée, ceux-ci valent zéro. La syntaxe de déclaration est la suivante `int answer = 42;`

Les types de données supportés par *promela* sont les entiers, avec différentes variantes selon le nombre de bits de l'emplacement mémoire, les tableaux, ou les structures à l'image du C.

**Les canaux** Les canaux sont déclarés par le mot `chan` suivi du nom de canal. Ils ont une capacité maximale et les messages sont typés. La déclaration se fait selon la syntaxe suivante: `chan qname = [ 16 ] of {short}`. Les canaux peuvent être locaux à un processus ou globaux. Il est éventuellement possible d'assurer l'exclusivité de l'émission ou réception à un processus particulier.

**Les Processus** Les processus représentent les entités concurrentes du système distribué que l'on cherche à modéliser.

Un processus et son comportement est défini à l'aide d'un bloc

`proctype`. Il définit éventuellement des variables locales et réalisera un ensemble d'instructions qui pourront modifier l'environnement via des affectations, ou échanges de messages sur un canal de communication.

Voici un exemple d'une déclaration d'un processus basique:

```
active[1] proctype inc() {
    int cpt = 2;
    cpt++;
    cpt++
}
```

Le mot clé `init` est réservé au bloc de la déclaration du processus initial. Celui-ci s'exécute à l'état initial du système, et est utilisé pour dynamiquement créer d'autres processus à l'aide de l'opérateur `run` (qui permet aussi de passer des paramètres au processus).

Le mot clé `active` a été introduit dans les versions récentes de Promela pour permettre de déclarer directement des processus sans passer par une succession de `run`

**Instructions de modifications de l'environnement** L'état global du système est modifiable avec trois opérations: l'affectation, l'émission et la réception de message.

**Affectation** Comme tout langage impératif, Promela est doté de l'affectation avec sa sémantique standard. On assigne à une variable mémoire une nouvelle valeur. Les instructions classiques d'incrémentement `++` et décrémentation `--` sont également disponibles.

**Le transfert des messages** Les canaux acceptent les deux opérations de base que sont l'émission et la réception.

L'émission (`send`) sur un canal est représentée par le point d'exclamation. Par exemple : `qname ! expr;` correspond à une émission standard, envoie la valeur de l'expression `expr` sur le canal nommé `qname` ajoutant le message en queue.

La réception est elle représentée par le point d'interrogation ?

Par exemple, avec l'instruction `qname ? msg;`, on récupère le message en tête du canal et on le stocke dans la variable `msg`.

Les opérations d'émission et réception sont bloquantes si jamais le canal est respectivement plein ou vide. De manière standard, les messages sont reçus dans l'ordre qu'ils sont émis mais il existe des variantes de ces instructions notamment avec la réception aléatoire `??` qui prendra au hasard un des messages disponibles. Ceci permettra notamment de simuler des canaux de communication non fiables.

Par défaut, les messages reçus sont consommés mais il est possible d'avoir une réception non destructive avec la syntaxe suivante: `keepchan ? < a >`

A noter qu'il est éventuellement possible de discriminer les messages à la réception, ainsi l'instruction `zerochan ? 0` sera bloquante tant que le prochain message à recevoir ne sera pas nul.

**Les structures de "flux de contrôle"** En promela, toute instruction est bloquante si elle n'est pas exécutable, tout comme les conditions, expressions booléennes, si celles-ci s'évaluent à faux.

Il existe aussi plusieurs structures de flux de contrôle: les blocs atomics (sections critiques), alternatives (selections), les boucles (iterations) et les sauts inconditionnels (goto).

**Les blocs Atomic** Il est possible de regrouper des instructions devant être réalisées de concert dans une construction spéciale. Le bloc `atomic` permet ainsi de définir un fragment de code qui doit être exécuté de façon indivisible.

```
atomic { /* swap the values of a and b */
    tmp = b;
    b = a;
    a = tmp
}
```

Si jamais une des instructions du bloc n'est pas exécutable, par exemple la réception sur canal vide, l'exécution du bloc sera interrompue le temps nécessaire, avant de reprendre son exécution.

Il existe une variante dite déterministe, `d_step` qui, quant à elle,

ne peut être interrompue et qui ne peut donc pas contenir des instructions bloquantes.

**Les sélections** La construction `if` représente une alternative. Celle-ci va contenir plusieurs options, des séquences précédées par `::`. La première instruction d'une séquence de contrôle est appelée une garde et détermine si l'option peut être choisie ou non. Si plusieurs gardes sont vraies, comme dans l'exemple suivant si `x` est compris entre `a` et `b`, alors le choix est effectué de manière non déterministe. Si aucune des gardes est vraie, alors soit le `if` est bloquant, soit il existe une option par défaut, une séquence précédée du mot clef `else`.

```
if
:: (A < x) -> option1;
:: (x < B) -> option2;
:: else -> option_par_defaut;
fi
```

**Les boucles** Une boucle `do` est une variante de la construction `if` à la différence qu'à la fin de l'option choisie, le programme revient à la position de départ. Le choix va donc être répété jusqu'à ce qu'une instruction `break`, ou un `goto` soit rencontré.

Une boucle `do` est exécutable si et seulement si, au moins une des gardes des séquences d'options est exécutable, et une option `else` est aussi admise.

La boucle suivante modélise un simple compteur, dont le programme sortira une fois que la variable `cpt` vaudra 12.

```
int cpt;
do
:: (cpt < 12) -> cpt++;
:: (cpt == 12) -> break;
od
```

**Les sauts inconditionnels** Toute instruction peut être nommée à l'aide d'un label qui va la précéder. `A:`  
`a++`

Ceci va permettre d'effectuer des sauts inconditionnels dans le flux de contrôle à l'aide de l'instruction `goto target_label` qui va identifier l'instruction cible. Cette instruction est toujours exécutable.

Si jamais l'instruction `goto` est seule dans une séquence, elle est exécutée comme un `skip`, l'instruction sans effet qui consomme un pas d'exécution.

## 2.2 GAL: le langage cible

### 2.2.1 Présentation

Le langage GAL (Guarded Action Language) est un langage de modélisation pour décrire des manipulations de données pour la vérification formelle de systèmes concurrents. C'est un langage de modélisation plutôt bas niveau sans notions de processus, structures ou canaux de communications. Ces derniers sont néanmoins exprimables dans ce simple langage ce qui en fait un langage intermédiaire dédié à la vérification symbolique.

Ce dernier a été développé par l'équipe Move au Lip6 en tant que backend pour d'autres notations plus "confortables" et ciblant un domaine particulier. Des transformations depuis des description de réseaux de Pétri, d'automates temporisés ont déjà été développés.

### 2.2.2 Langage

Les constructions de bases du langage GAL sont les transitions, les actions, les expressions et les variables, celles-ci étant regroupées dans des systèmes.

Les transitions regroupent un certain nombre d'actions qui, généralement, consistent juste en une succession d'affectations. Toutes les variables sont globales et de types entier ou booleen, les tableaux sont aussi supportés. Une garde booléenne est éventuellement associée à la transition pour éventuellement empêcher son exécution.

En plus de cette "syntaxe simpliste", la sémantique de GAL est assez simple, proche d'une structure de Kripke. Un état est associé à chaque valuation possible de l'ensemble des variables. Les transitions sont atomiques et à chaque instant une transition à la garde vraie est exécutée. Voici un petit aperçu simpliste d'un système GAL.

```
gal forLoop_inst {
  int var = 42;
  array [3] tab = (1, 2, 4) ;

  transition forExample [ var != 12 ] {
    tab [0] = 0 ;
    tab [1] = 1 ;
    tab [2] = 2 ;
  }
}
```

Pour plus de détails, consulter la [description du langage du le site officiel](#).

### 2.2.3 Outil

Les spécifications GAL sont analysable à l'aide de l'Outil ITS de model checking symbolique. Il s'appuie sur la représentation des Diagrammes de Decision (implanté à travers la librairie ddd) ITS signifie **Instantiable Transition System**, et est un formalisme abstrait permettant de combiner plusieurs formalismes dont GAL, les réseaux de pétri, ou automates temporisés en les unifiant sous la même interface. Il permet d'analyser l'espace d'état d'un modèle, et l'accessibilité de certains états à l'aide d'**its-reach**. Il offre aussi deux autres outils pour pouvoir vérifier des propriétés de logique temporelle exprimées à l'aide des formalismes LTL (Linear Temporal Logic) et CTL (computational tree logic). Cet outil développé par notre encadrant dans l'équipe *move* du *Lip6* et est disponible sur le [site dédié aux Diagrammes de Décision](#).

## 2.3 Xtext et Xtend : Langages et framework support

Afin de réaliser la transformation, et donc de produire toute l'infrastructure de compilation/traduction de Méta-modèle nous sommes basés sur le framework OpenSource de développement de langage **Xtext**, que ce soit de simples DSL (Domain Specific Languages) ou de nouveaux langages bien plus complexes. Nous allons introduire le framework, en présenter son utilisation, avant d'évoquer le langage **Xtend**, développé à l'aide d'**Xtext** et largement utilisé pour développer toute l'infrastructure de notre langage.

### 2.3.1 Le Framework Xtext

A partir d'une grammaire, Xtext va générer un parser, mais aussi un MétaModèle et les différentes classes pour l'arbre de syntaxe abstraite, ainsi qu'un éditeur Eclipse complet et configurable.

Pour ce faire, Xtext se base sur différentes autres technologies open sources existantes.

Le parser se base sur **Antlr** (ANother Tool for Language Recognition), framework de construction de compilateur, le MétaModèle sur **EMF** (Eclipse Modeling Framework), un framework de modélisation et génération de code, et bien d'autres comme **Guice** (Google Juice) pour l'injection de dépendances, ou **MWE** (Modeling Workflow Engine) pour gérer l'agencement des différentes phases de génération de code.

Pour le MétaModèle, Xtext s'appuie sur EMF comme représentation standard, et génère la description selon les formalismes **Ecore** et **Genmodel**. Le métamodèle Ecore contient l'information concernant les classes définies, leur attributs et leurs propriétés. Le Genmodel, quant à lui, contient l'information sur la génération du code. Ceci nous permet d'avoir à la fois une API de manipulation générique des Metamodèles, mais aussi spécifique à notre AST (Arbre de Syntaxe Abstraite), ainsi qu'un factory pour créer les différents éléments. Pour plus de détails sur EMF et son intégration avec Xtext voir cette [documentation sur le site officiel Xtext](#)

Parmi les fonctionnalités offertes de base, on trouve tout ce qui fait le confort des IDE actuels: à savoir: la coloration syntaxique, la complétion contextuelle, l'analyse statique, le "outline" et la navigation dans le code source, l'indexation, le refactoring et le quickfix, et bien d'autres.

Depuis 2008, Xtext est devenu un projet Eclipse, et son Moto est "Language Development Made Easy!", tout un programme. Il a reçu en 2010 le prix du projet eclipse le plus innovant et en est actuellement à sa version 2.5.

### 2.3.2 Développement de langage avec Xtext en pratique.

La première étape pour développer le langage est d'écrire la grammaire. La syntaxe permet en même temps de définir les différentes règles de parsing et les constructions du langage.

Extrait de la grammaire.

```
Iteration:
    'do'
    {Iteration}
    ( '::' options += Sequence )+
    ( '::' 'else' '->' else = Sequence )?
    'od'
;
```

La syntaxe utilise les opérateurs standards des expressions régulières pour exprimer la cardinalité. Dans cet exemple, on voit une règle nommée Iteration. Celle-ci commence avec le token **do**. Le **{Iteration}** entre crochet indique qu'une structure Iteration devra être instantiée. Ensuite, une ou plusieurs règles **Sequence** seront invoquées, et le résultat sera accumulé dans l'attribut **options**. L'éventuel **else ->** suivi d'une séquence sera stocké dans la variable **else**. La règle se termine par l'utilisation d'un point virgule.

La grammaire entière est définie avec un ensemble de règles plus ou moins complexes.

Une fois une première version écrite, l'infrastructure du langage doit être générée. L'ensemble des classes pour représenter le MM sera créé. De plus, lors de la première génération, toutes les implémentations par défaut des différents composants de l'infrastructure sont créées. Ce sont ces classes que l'on viendra compléter pour ajouter de nouvelles fonctionnalités, ou configurer celles existantes.

Plusieurs projets Eclipse sont créés dont les deux principaux:

**xtext** il s'agit du projet principal, dans lequel se trouve la grammaire (donc les classes ast générées), mais aussi tout ce qui fait le coeur du langage, c'est-à-dire le code de validation (pour vérifier statiquement nos programmes), celui pour gérer la portée, la mise en forme. . .

**ui** ce projet regroupe tout le code pour personnaliser l'éditeur. A savoir, les quickfixes, le labelling (les noms à attribuer aux différentes constructions du langage), l'outline (structure d'arbre résumant le programme), l'assistance contextuelle. Chacune de ces fonctionnalités est regroupée dans son propre package.



Parmi les autres projets, on trouve celui pour écrire les tests pour notre infrastructure de langage, un autre pour éventuellement développer un sdk (software development kit) ainsi que d'autres pour gérer l'update-site. (Le site où toute installation eclipse pourra récupérer notre plug-in.)

Xtext génère un ensemble de classes génériques, avec des définitions de bases que l'on viendra redéfinir pour donner à l'éditeur et au langage le comportement souhaité.

Des [tutoriels introductifs](#) sont aussi disponibles sur le site officiel, et n'offrent qu'un très bref aperçu des premières étapes du développement d'un langage. Il existe aussi [7 implémentation de langage](#), en tant que modèles de références.

### 2.3.3 Le langage Xtend

Le framework Xtext s'appuie sur un nouveau langage pour écrire nos nouveaux langages. Nous allons présenter les raisons, atouts et inconvénients de ce "GPL pour DSLs", Xtend.

**Pourquoi un langage de plus?** Le langage Xtend a été développé par les créateurs du framework Xtext afin de faciliter l'écriture du code pour supporter le langage, et de s'épargner la verbosité de Java.

De ce point de vue, il peut être considéré comme du super sucre syntaxique Java, en se compilant vers du code Java 5, et en fournissant une syntaxe et des fonctionnalités rendant le code plus compact à sémantique égale, tout en ayant accès à l'ensemble de l'API Java. Il s'agit donc bien d'un "General Purpose Language".

**Fonctionnalités intéressantes** Xtend aide à produire un code à lecture et écriture aisées, bien plus qu'avec du Java Standard.

Ceci est rendu possible grâce à l'introduction de nouvelles fonctionnalités, dont nous allons passer en revue une partie.

Tout d'abord, si Xtend utilise le même système de typage que Java (ce qui lui permet d'être totalement compatible), il utilise l'inférence de type pour épargner au programmeur de spécifier le type lorsque celui-ci est évident. On pourra toujours le spécifier pour clarifier le code, permettre une récursion, ou forcer l'utilisation d'un surtype.

Ceci permet d'alléger le code en supprimant les redondances javaesques du style `final Stack<PC> breakStack = new Stack<PC>();` pour `val breakStack = new Stack<PC>()`.

La syntaxe peut aussi être allégée vu que les `;` sont facultatifs, ainsi que les parenthèses des fonctions sans paramètres. De plus les `get` et `set` des accesseurs peuvent être sous entendus.

Ainsi, on pourra écrire : `o.corps.steps.size` en lieu et place de `corps.getCorps().getSteps().size();`

Xtend apporte aussi les Lambdas, et ceci sans attendre Java 8. Leur syntaxe légère, et l'itérateur implicite `it` s'avère extrêmement utiles couplés à des méthodes `forEach` des collections.

Ceci a été utilisé au niveau des classes de l'interface utilisateur, notamment dans les méthodes pour gérer l'outline. Ceci permet aussi d'alléger le code là où on devait faire une implémentation anonyme de l'unique méthode abstraite de l'interface cible comme dans les méthodes `addListener`, `run`, etc.

§sample

Une autre fonctionnalité très intéressante voire même vitale pour de la génération de code sont les templates multilignes. En bref, il s'agit de chaînes particulières permettant d'insérer à l'intérieur d'une chaîne, le résultat d'une expression, à l'image du `"abc#{expression}"` en Ruby ou `echo "Hello $USER"` en shell.

Ceci évite d'avoir à écrire soi-même les successions d'`append` sur des `StringBuffer`, ce qui a le don d'obstruer totalement l'intention du code et le schéma de compilation. Cependant, comme nous avons fait de la traduction via métamodèle, plutôt que d'écrire du code compilé, nous n'avons pas eu vraiment l'occasion d'en profiter.

§sample

La dernière caractéristique originale est ce que Xtend nomme les méthodes en extensions. En résumé, il s'agit d'une notation alternative où la méthode peut être invoquée par `arg1.meth(arg2,arg3)` plutôt que par `Classe.meth(arg1,arg2,arg3)`, ce qui permet notamment de chaîner les appels de fonctions et éviter des variables locales superfétatoires (pour plus de détails, se référer à la documentation).

`Xtend` permet aussi la surcharge des opérateurs, mais nous n'avons pas eu besoin de les utiliser. D'autres fonctionnalités intéressantes existent, telles que l'opérateur `elvis ?:` ou l'accesseur `null-safe ?.`, et l'ensemble d'entre elles pourront être découvertes en lisant la documentation officielle.

**Petits Bémols à suivre.** Si `Xtend` a l'air génial sur le papier, son utilisation vient néanmoins doucher un peu l'enthousiasme initial.

Le langage est encore jeune, et mériterait de gagner en maturité (par exemple, certaines fonctionnalités de java ont été perdues sans que l'on sache pourquoi: à l'image du `+=` et autres affectations composées (compound assignment:  $x+ = 2 \equiv x = x + 2$ ) en passe d'être incorporées dans les nouvelles versions du langage).

Cependant, le principal problème de `Xtend`, est que l'éditeur Eclipse associé est à ce jour encore peu fonctionnel. Son utilisation peut même être désagréable, et compenser tous les bienfaits du langage: Les messages d'erreur sont parfois cryptiques et mal localisés, le "control assist" (C-spc) laggue tellement qu'il est tout simplement inutilisable, les templates javadoc ne s'adaptent pas à la fonction commentée,...

A noter aussi qu'en production, une passe supplémentaire doit être effectuée avec une phase de génération de code java.

Il n'est probablement qu'une question de temps avant que ces problèmes ne soient réglés.

## 3 Méta-modèle et grammaire de Promela

### 3.1 Elaboration de la grammaire

Tout le MM du langage et l'éditeur `Xtext` dérive de la grammaire, nous allons présenter celle-ci. Nous allons présenter notre démarche avant de nous concentrer sur quelques coins particuliers.

#### 3.1.1 Démarche

Nous avons consacré les premières semaines sur cette étape initiale, ceci jusqu'à obtenir une grammaire assez stabilisée pour commencer la transformation. Nous sommes partis d'une première grammaire sommaire gérant les concepts de bases (processus, bloc atomic) avant d'ajouter progressivement de nouvelles instructions du langage. Dans un premier temps, nous avons permis le compilateur de backtracker afin d'avancer rapidement.

Une fois la plupart du langage pris en compte, afin d'avoir une grammaire LL, nous avons rétabli et dû refactorer une grande partie de la grammaire. Après cela, la grammaire étant considérée comme stable, nous avons pu attaquer sereinement la transformation.

Par la suite, la grammaire n'a été modifiée qu'à la marge pour résoudre quelques petits problèmes apparus lors de la construction de la transformation `pml2spin`. Quelques défauts mineurs ont aussi été notés et pourront être corrigés si une modification de moyenne envergure de la grammaire était à réaliser, afin de ne pas modifier pour des broutilles le code client (outline, togal).

#### 3.1.2 Le choix d'une grammaire LL et ses conséquences

Lors de la rédaction de la grammaire nous avons dû faire face à plusieurs choix, certains totalement anecdotiques, d'autres aux conséquences bien plus grandes.

De loin, le choix d'avoir une grammaire LL fut le plus important.

**Pourquoi une grammaire LL?** L'activation du backtracking (retour sur trace) lors du parsing va permettre d'écrire de manière bien plus simple, car ceci permet d'avoir une grammaire ambiguë. Face à un dilemme, le parseur optera pour une des options, et viendra essayer les suivantes si jamais celle-ci mène à une impasse. (Il s'agit de la même méthode utilisée dans `prolog`, appelée autrement méthode des essai et erreur.)

L'écriture de la grammaire est simplifiée, sans que la performance soit impactée de manière sensible. Par contre, ceci vient avec un coût: les erreurs de syntaxe seront mal localisées. Par conséquent, les messages d'erreur de syntaxe remontés à l'utilisateur seront assez cryptiques et risqueraient d'indiquer une mauvaise piste.

C'est pour cela que la désactivation du backtracking, malgré les complexités impliquées par la rédaction d'une grammaire LL, est préférable. C'est d'ailleurs considéré comme une bonne pratique dans la communauté `Xtext`.

**La grammaire xtext GAL comme inspiration** Notre langage cible étant GAL, qui a aussi été développé utilisant le framework Xtext, nous nous en sommes inspirés afin de faciliter la passe de traduction.

Cependant, comme les deux langages sont assez différents, nous n'avons pu réutiliser que les **Expressions**, et encore, de nombreuses modifications ont dû être réalisées. En effet, Gal fait la distinction entre les expressions Booléennes et Entières, au niveau de sa grammaire, alors que pour promela, inspiré du C, il n'y a pas de distinction de ces deux genres d'expressions. Les deux ont été fusionnés dans la grammaire.

**Rekursivité à gauche des expressions, et associativité des opérateurs.** La nature intrinsèquement récursive des expressions a aussi dû être gérée.

En effet, la plupart des expressions contiennent elles même d'autres expressions, et un grand nombre des règles de grammaire pour les expressions sont de la forme  $E = E \langle \text{op} \rangle E$ , ce qui est assez problématique, surtout pour une grammaire LL. De plus, l'associativité des opérateurs doit être gérée. En effet le parseur doit savoir comment construire l'AST lorsqu'il y a deux opérations infixes de priorités différentes. Par exemple  $a + b * c$  correspond à  $a + (b*c)$  et non  $(a+b)*c$ .

Pour réaliser ceci en Xtext, nous allons définir chaque règle de manière indépendante, mais chacune devra déclarer Expression comme son superType. Surtout ses appels récursifs à ses sous-expressions, plutôt que de préciser la règle Expression, elle appellera la règle qui la suit dans l'ordre d'associativité.

Ceci est illustré par ce fragment de grammaire:

```
Or returns Expression:
    And ( {Or.left = current} '||' right = And)* ;
And returns Expression:
    Not ( {And.left = current} '&&' right = Not)* ;
Not returns Expression:
    '!' {Not} value = Comparison
    | Comparison ;
```

L'expression entre crochet correspond à une réécriture de l'arbre, c'est-à-dire que l'expression qui est retournée par l'appel à la règle And (le current), est affectée à l'attribut left du Or si jamais un token || était rencontré.

L'associativité doit aussi être gérée pour des opérateurs de même priorité.

Par exemple, prenons une expression comme :  $a + b + c$ . En promela, l'associativité est gauche comme pour la plupart des langages, l'expression est donc interprétée comme  $(a + b) + c$  et évaluée de gauche à droite. Ceci est réalisé dans la règle xtext par la cardinalité \* du bloc ( {And.left = current} '&&' right = Not).

S'il s'agissait d'une associativité droite, il aurait fallu utiliser la cardinalité ? (1 au plus).

**Factorisation à gauche** Afin d'avoir une grammaire LL, nous avons dû factoriser à gauche notre grammaire (left factoring).

Ainsi, si deux règles commencent par le même pattern, celui-ci doit être regroupé dans une règle à part. Ainsi, le parseur n'a pas à revenir en arrière, ou à choisir entre deux règles compatibles, ce qu'il ne peut pas faire sans aller voir plus en aval lequel est le bon.

Il faut donc supprimer toute ambiguïté pour le parseur.

La left factorisation a rendu la grammaire moins lisible. Ceci s'est surtout ressenti au niveau de certaines expressions, celles-ci pouvant toutes commencer par un token ID (c'est-à-dire une chaîne de caractères quelconque correspondant à un identifiant).

En pratique nous avons regroupé toutes ces instructions commençant par un token ID dans une règle spéciale pour plus de lisibilité.

```
LLFactoredInstructions returns Instruction:
    /*@ LLFactoring: Expression > Send, Receive, Assignment
    Expression ( {Send.channel = current} (fifo ?= '!' | '!!') args = SendArgs
    | {Receive.channel = current} ( normal ?= '?' | '??' )
    ( keep ?= '<' args = ReceiveArgs '>' | args = ReceiveArgs )
    | {Assignment.var = current} kind = AssignmentOperators
    =>( newValue = Expression )
    | {Condition.cond = current} )
;
```

Le nom entre crochets `{Send}` permet de forcer la création d'un sous-type et ainsi, contrôler un peu mieux le Métamodèle qui sera déduit de la grammaire.

**Hierarchie des MetaClasses perturbée** En plus de venir complexifier les règles, la rédaction d'une grammaire LL en `Xtext` vient aussi perturber le metamodèle qui est déduit des règles de grammaire. En effet, comme on vient de le voir avec la factorisation gauche, la variable d'une Affectation ou le canal d'une Réception/Emission est une Expression, alors qu'on voudrait que ce soit une Reference. Quelques contrariétés du même type viennent perturber l'AST. De plus si l'on voulait que deux règles factorisées aient un super-type commun, il fallait définir un certain nombre de règles factices pour cela.

Il a été un temps envisagé de figer le `ecore` (désactiver la génération depuis la grammaire), et d'opérer les modifications pour avoir les classes voulues. Cependant, ceci impliquait indirectement de figer la grammaire ce qui n'était pas envisageable à ce stade du développement.

Finalement, nous avons simplifié les choses en factorisant et regroupant tous les différents types d'Affectation, Send, Receive sous les mêmes classes et en utilisant des attributs pour encoder leur variabilité. Pour les Expressions à la place des variables, rien n'a pu être fait à ce niveau: une validation statique a été mise en place, et des fonctions utilitaires pour récupérer directement l'attribut sous le bon type.

## 3.2 Meta Modèle de Promela

Le Méta Modèle final est généré par `Xtext` dans le fichier `Promela.ecore` à partir de l'ensemble des règles et annotations que l'on a mis en place dans la grammaire. Celui-ci est visualisable grâce au plug-in Eclipse [Ecore Tools](#), qui va générer un tel diagramme de classe modifiable.

Montrer notre MétaModèle dans son intégralité n'aurait aucun sens. Tout simplement car un tel diagramme serait illisible de part la complexité du langage. Dans sa version complète, seule une navigation interactive aurait du sens.

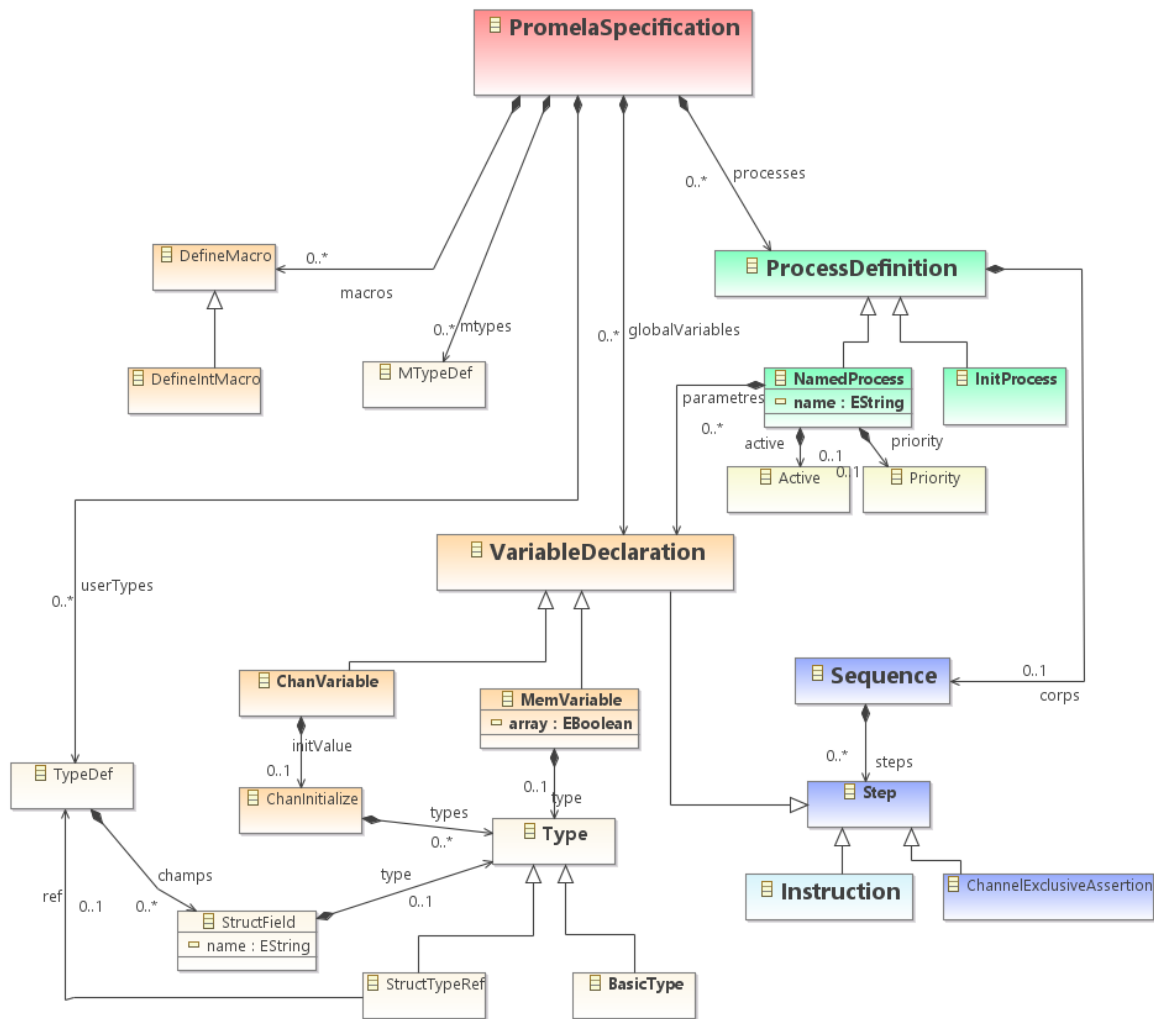
Pour avoir un aperçu nous avons donc préparé deux vues du MétaModèle à deux niveaux d'abstractions différents pour lesquels nous avons réalisé des diagrammes lisibles.

### 3.2.1 Spécification Promela

La première vue est celle d'ensemble, qui montre le lien entre les différentes constructions du langage sans entrer dans la diversité des expressions et des instructions.

Un code couleur a été utilisé pour départager des constructions de différentes natures.

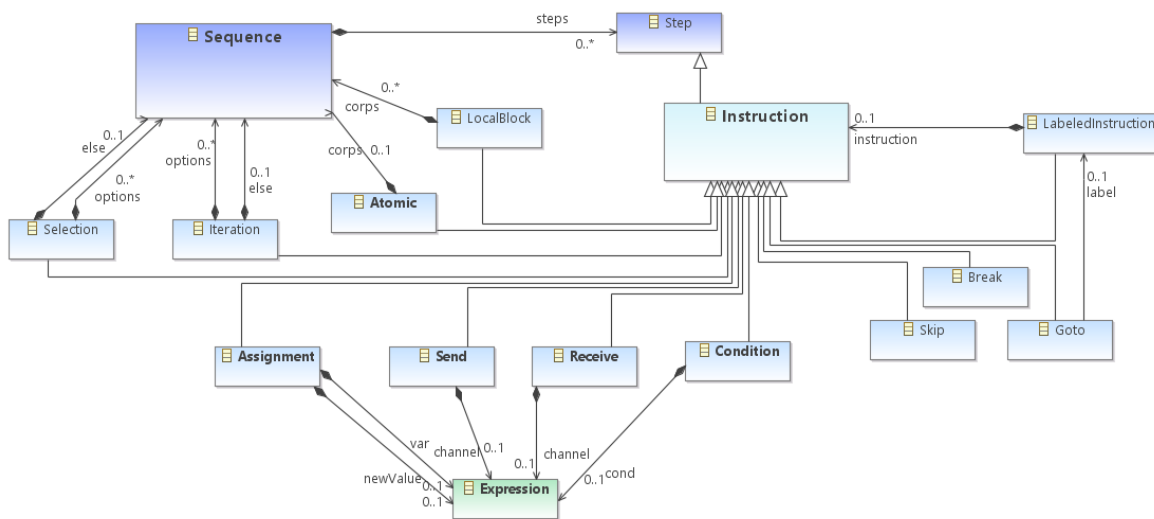
- En *rouge*, tout en haut, la Spécification Promela qui est la racine de notre arbre de syntaxe abstraite représentant tout programme Promela.
- En *vert clair* les Définitions de processus, éventuellement nommées et dotées de paramètres.
- En *bleu* les Séquences et les Instructions qui vont décrire les actions entreprises par un processus.
- En *blanc cassé*, on trouvera les constructions reliées au typage. Il y a donc les types de bases, les structures définies par les utilisateurs, ainsi que les "MTypes", c'est à dire des noms de messages afin de pouvoir utiliser `ACK SYNC` comme valeur dans le programme
- En *orange*, les différentes variables typées et macros du langage



### 3.2.2 Les Instructions

Ce diagramme vient illustrer les principales instructions du langage Promela.

Toute instruction fait partie d'une séquence, et certaines instructions contiennent une séquence donnant ainsi un structure récursive au langage que nous avons dû gérer pour la transformation.



### 3.2.3 Les Expressions

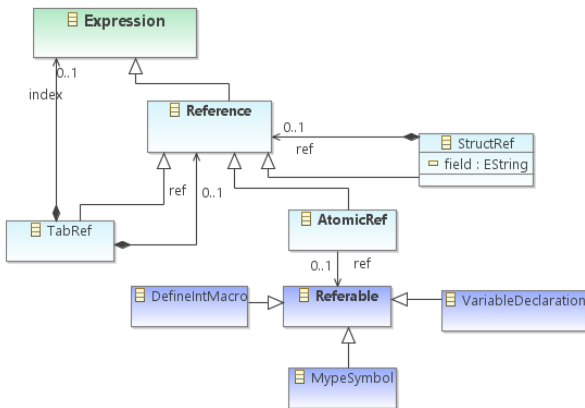
Les expressions sont la catégorie sémantique avec la plus grande diversité de construction, dont la plupart sont de nature récursive (c'est-à-dire qu'elles contiennent elles même d'autres expressions).

On y retrouve tous les opérateurs arithmétiques, booléens. Les principales feuilles des expressions sont les références à des variables, les constantes littérales et booléennes et certains opérateurs promela spécifiques.

Pour pouvoir gérer les références de structures et de tableaux, nous avons adopté une structure récursive inspirée du design pattern composite. Les Références (qui sont des expressions), sont soit une Référence de tableau, soit une référence de structure, ayant donc une référence associée à un index ou un nom de champs.

Soit une "Atomic Reference" qui va se référer à une construction "Referable". Cette dernière est soit une déclaration de variable, soit une macro ou un symbole de message.

Ceci est illustré dans le diagramme suivant:



On retrouvera en *bleu ciel* les références, *bleu foncé* les éléments référençables.

### 3.3 Limites du MM

En suivant le précepte "*Loose Grammar, Strict Validation*", la grammaire seule accepte grand nombre de programmes qui ne sont pas du tout valides.

Par exemple, si rien n'est fait pour vérifier, le code suivant serait considéré comme correct:

```
int c = 3; (4 + c) = 2 // Affectation Bidon
```

En effet, que ce soit pour la variable d'une affectation, ou le canal d'une réception/émission, on ne peut malheureusement pas mettre une référence bien que c'est ce que l'on souhaiterait.

Effectivement, pour éviter les forts appréciables erreurs `error(211): ../fr.lip6.move.promela.xtext/src-gen/fr/lip6 [fatal] rule ruleIDAmbiguous has non-LL(*) decision due to recursive rule invocations reachable from alts 2,4. Resolve by left-factoring or using syntactic predicates or using backtrack=true option.`, nous avons dû factoriser les règles comme préalablement raconté.

De ce fait, ceux-ci sont des expressions, alors que seules les références sont acceptables!

Pour autant, des vérifications auraient dû être faites même si notre grammaire était bien plus stricte.

En effet, une grammaire ne peut pas tout contenir, surtout avec un langage comprenant autant de constructions et un tel système de types.

Une validation du programme et son modèle associé est donc nécessaire.

En plus d'être inaccessible, une Grammaire stricte est bien plus complexe à écrire, beaucoup plus verbeuse et répétitive.

Par exemple, si on avait voulu contraindre la présence des `break` uniquement dans des `do` au niveau grammatical, il aurait fallu pour cela dupliquer les règles `Instructions`, `Step`, `Sequence` par des règles soeurs `[ISS] DansBoucle`. Ce qui, de tout évidence, n'est pas une bonne pratique car étant difficile à maintenir en cas de modification et, de plus, propice aux erreurs.

## 4 Analyse Statique de Promela

### 4.1 Typage

Le typage permet de vérifier la concordance des types des expressions et des instructions : par exemple une variable de type `short` ne peut pas contenir une valeur de type `int` mais le contraire est possible. De même, nous ne pouvons pas recevoir ni envoyer n'importe quel type de message dans un canal de type défini. Cette vérification ne peut être faite que lors de la validation du programme. Elle doit aussi être étendue sur tous les types définis dans Promela.

Voici la solution que nous proposons :

La stratégie que nous avons adoptée est de calculer d'abord les types dans la classe `PromelaTypeProvider` et de faire, ensuite, la validation des types dans `PromelaTypeValidator`.

Nous avons représenté chaque type de Promela par une classe Java.

Toutes ces classes Java implémentent l'interface `PromelaType`.

La classe `PBasicType` est en fait une énumération qui regroupe tous les types primitifs (`bit`, `bool`, `byte`, `short`, `int`).

A chacun est associé un nom (la chaîne de caractère du nom du type en question) et une taille (la taille en nombre de bits du type).

Notre typage repose sur le fait qu'un type englobe tous ceux de taille inférieure. Dans Promela, nous avons l'ordre suivant:

```
bit (1) < byte(8 bits) < short(16 bits) < int(32 bits)
```

Les classes `PArrayType`, `PChannelType` et `PUserType` permettent de représenter, respectivement, les types tableau, channel et type utilisateur de Promela.

Dans `PromelaTypeProvider`, nous allons retourner un type `PromelaType` pour chaque expression, variable et référence. Cela se fait par la surcharge d'une méthode `typeFor` qui prends en argument chaque entité de la grammaire pour laquelle nous voulons récupérer un type. Nous utilisons un `switch` pour les cas simples. Pour les expressions pouvant être typées directement, on écrit donc le code suivant:

```
def static dispatch PromelaType typeFor(Expression e) {
    switch (e) {
        True:
            PBasicType.BOOL
        // .. Others
        Comparison:
            PBasicType.BOOL
        //...
    }
}
```

Pour les expressions qui nécessitent plus de réflexion, il s'agit d'adapter le `typeFor` afin qu'il renvoie le type effectif recherché. Dans l'exemple suivant, nous mettons en place le `typeFor` pour les `ChanVariable`, c'est-à-dire les déclarations de canaux.

Il convient dans un premier temps, afin de pouvoir expliquer ensuite ce bout de code, d'évoquer la structure des canaux en Promela. Lorsque l'on déclare un canal, on lui donne les types des champs du message qu'il pourra contenir. Ainsi lorsque l'on déclare, par exemple, le canal suivant:

```
chan qname = [ 8 ] of { int, short, byte }
```

on indique que ce canal peut contenir un message comprenant trois champs (un de type `int`, un de type `short` et un de type `byte`).

Les messages sont donc des structures de données.

Dans le code suivant, nous ajoutons dans l'ordre les différents types des champs dans une liste. Ainsi, nous pourrions vérifier ensuite qu'ils correspondent bien (et dans cet ordre) aux champs du message auquel on fait référence dans un `send` ou un `receive`.

```
def static dispatch PChannelType typeFor(ChanVariable chan) {
    val size = (chan.initValue.size as LiteralConstant).value;
    val varL = new ArrayList<PromelaType>

    for (Type t : chan.initValue.types) {
```

```

    if (t instanceof BasicType) {
        varL.add(PBasicType.get(t as BasicType))
    } else if (t instanceof StructTypeRef) {
        var name = (t as StructTypeRef).ref.name
        var struct = new HashMap<String, PromelaType>();
        val f = (t as StructTypeRef).ref.champs
        for (StructField ty : f) {
            struct.put(ty.name, PBasicType.get(ty.type as BasicType))
        }
        varL.add(new PUserType(struct, name))
    }
}
return new PChannelType(varL, size);
}

```

Nous avons un validateur spécifique pour le typage: `PromelaTypeValidator.xtend`, également rédigé en Xtend.

Dans ce validateur, nous avons une méthode annotée par `@Check` pour chaque sorte d'expression excluant les expressions qui sont implicitement bien typées comme : les macros et les constantes.

Nos validations peuvent être regroupées en deux catégories : validation des types des expressions et validation des types des instructions.

Pour les expressions, nous procédons de manière récursive en typant chaque sous-expression qui les forment.

Les instructions concernées par la validation que nous effectuons sont l'affectation, ainsi que l'émission et réception sur un canal (`send` et `receive`)

La validation du typage de la règle de l'affectation consiste à récupérer le type de la variable référencée à gauche et le type de la nouvelle valeur assignée (l'expression à droite). On vérifie ensuite si le type de l'expression est compatible avec celui de la variable

Le `send` et le `receive` se ressemblent dans le sens que nous récupérons le type défini lors de la déclaration du canal auquel on fait référence, et vérifie que le type des arguments (messages) correspond.

Pour chaque cas où le typage n'est pas respecté, une erreur est signalée et un marqueur (et message associé) apparaît dans l'éditeur.

## 4.2 Validation

Après l'élaboration de la grammaire, nous avons procédé à l'ajout de contraintes là où les règles de la grammaire était trop permissives ou ambiguës.

L'ajout de ces contraintes dans les règles de grammaire était difficile voire impossible, d'où la nécessité de mettre en place un validateur statique: `PromelaValidator`, rédigé en XTend.

Ainsi nous gardons une grammaire propre, facile à manipuler et permissive, mais dotée d'une validation forte.

Nous vérifions donc que:

- les `send`, les `receive`, les `poll` se font sur des références à des canaux uniquement
- les affectations sont bien effectuées sur des références et que la nouvelle valeur assignée n'est pas nulle
- les blocs `atomic` contiennent uniquement des instructions
- les blocs déterministes ne contiennent pas de `send` ni de `receive` ( car ce ne sont pas des opérations atomiques, donc invalides dans ce cas)
- le nombre d'arguments d'un `print` est égal au nombre de valeurs attendues dans la chaîne de caractères (par l'intermédiaire de `"%d"`, `"%e"`, `"%c"`, `"%o"`, `"%u"`, `"%x"`)
- l'index de la référence à un tableau n'est pas négatif et ne dépasse pas la taille du tableau (pour la référence `tab[i]`, on a bien  $i \geq 0 \wedge i < \text{tab.len}$ )



Si une telle erreur est détectée, elle est signalée à l'utilisateur avec un marqueur et un message approprié.

Nous pouvons également afficher un avertissement commenté (warning) à l'utilisateur pour les cas qui n'entraînent pas d'erreur, mais qui peuvent être gênants, sont le signe d'une erreur de conception potentielle ou vont à l'encontre des conventions de style.

Nous l'avons mis en place notamment pour:

- les séquences vides dans les alternatives, les boucles, les processus, les blocs atomic
- le cas où deux variables de même portée ont le même nom
- le nom d'une macro n'est pas en majuscule
- le cas où l'on trouve du code après un goto (ce code ne sera jamais exécuté, c'est donc du code mort, ou une erreur)
- le cas où l'on trouve des variables non utilisées
- le cas où les instructions `break` et `skip` ne se trouvent pas dans une boucle (en remontant dans les containers pères, on ne trouve pas de boucle)

Dans ces cas-là, un quickfix est proposé. Nous allons en parler plus en détail dans la partie suivante.

La principale difficulté rencontrée lors de l'élaboration du validateur était de naviguer sur les règles de la grammaire afin de récupérer les entités à observer.

Nous avons dû également d'abord élaborer une liste des endroits où la grammaire était permissive afin de pouvoir mettre en place toutes les validations nécessaires.

## 5 Éditeur avancé Promela

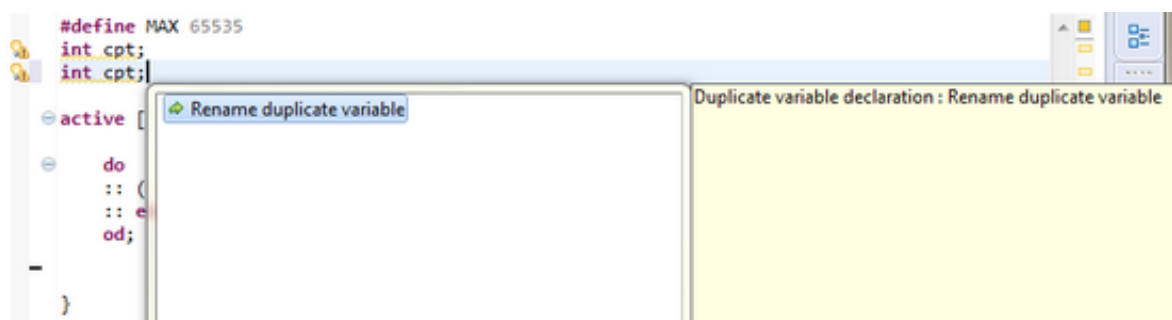
Bien que pouvant être amélioré davantage, l'éditeur Xtext généré par défaut s'avère déjà plus plaisant à utiliser et utile que le rudimentaire éditeur xspin. De plus certaines erreurs sont relevées à la volée ce qui n'est pas le cas avec les outils spin devant attendre la compilation du programme C généré. Nous avons décidé de personnaliser l'éditeur fourni par Xtext pour l'adapter au mieux possible aux besoins du langage Promela. Nous l'avons enrichi de différentes manières:

Ainsi, nous proposons des Quickfixes, une outline plus esthétique et navigable, des templates et la possibilité de formater le code.

### 5.1 Quickfixes

Comme expliqué dans la partie validation, nous nous sommes appuyés pour notre plugin sur une validation poussée des programmes Promela.

Ainsi, nous avons également pris soin de donner la possibilité de traiter les erreurs envoyées par des quickfixes. Comme vous pouvez le voir dans la capture d'écran ci-dessous, un quickfix est proposé à l'utilisateur de l'éditeur pour la plupart des warnings et erreurs détectée.



Ainsi, la possibilité est offerte de:

- renommer une variable au nom dupliqué
- supprimer du code mort après un goto
- mettre les noms des Macro en majuscule
- supprimer les variables inutilisées
- ajouter un skip dans les corps vides (il est préférable d'avoir une instruction plutôt que de laisser vide une séquence)

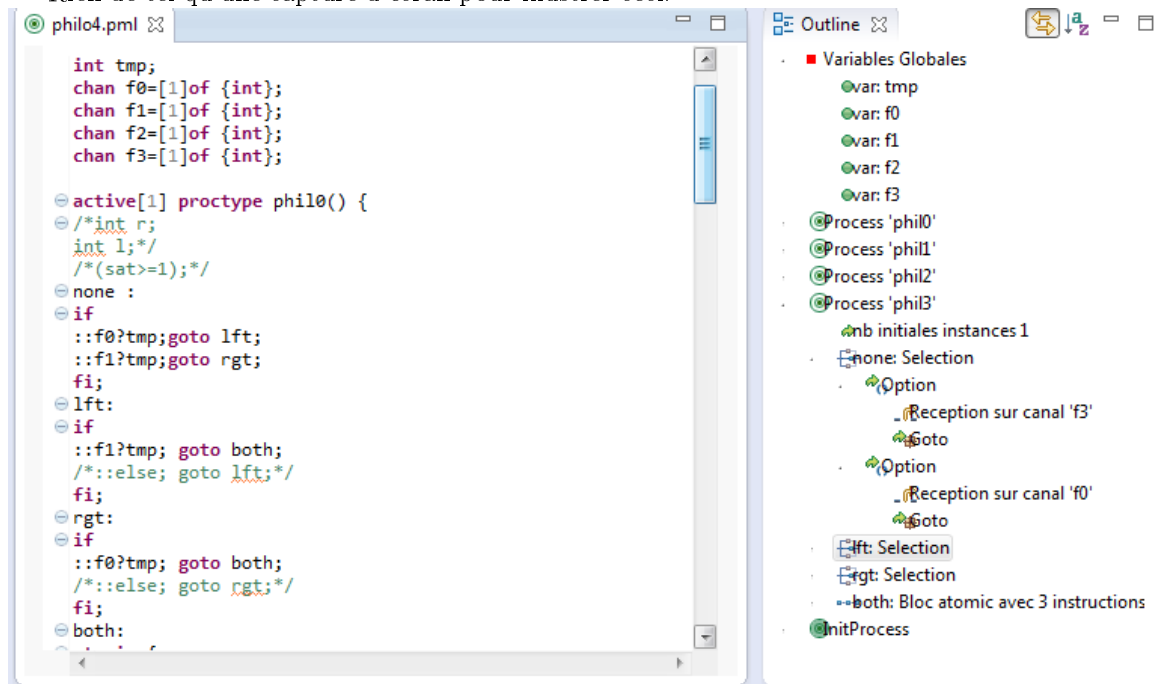
Les quickfixes se trouvent dans le package `quickfix` dans le project de L'interface graphique `fr.lip6.promela.xtext.ui`. Le langage utilisé est `xtext`. La principale difficulté rencontrée lors de leur élaboration était de récupérer ce qu'il fallait au niveau de la validation pour pouvoir ensuite le traiter simplement.

## 5.2 Outline and Labelling

Pour permettre une meilleure expérience utilisateur, l'outline et labelling de base fourni par la première génération de code Xtext a été entièrement personnalisé pour être à la fois visuellement attractif et permettre une navigation dans le code optimale.

L'outline correspond à une vue hiérarchique des différentes parties d'une code. Il s'agit juste d'une vue partielle de l'AST correspondant à la spécification en cours d'édition. Avec Xtend et ses switches aux gardes typées, certaines parties du code sont quasi déclaratives, où on associe chaque construction du langage à une image et un résumé textuel.

Rien de tel qu'une capture d'écran pour illustrer ceci.



## 5.3 Templates

Afin de donner un coup de pouce au programmeur PROMELA et de rendre notre éditeur de texte plus engageant, nous avons décidé de profiter des fonctionnalités offertes par Eclipse et de mettre en place des templates, comme ceux que l'IDE propose pour d'autres langages.

Ainsi, en un clic, l'utilisateur peut à présent récupérer le squelette pour un `proctype`, une boucle (`for` et `do`), une sélection (`if`), un `atomic` ou un `d_step`.

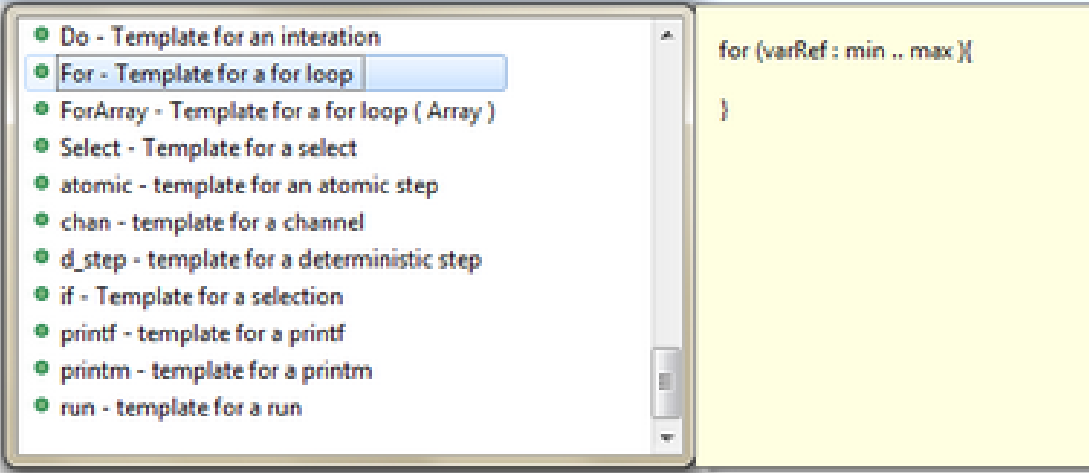
Ce service n'est pas proposé non plus, comme les précédents, par l'éditeur de l'outil SPIN.

Les templates sont définis dans le fichier `templates.xml`, se trouvant dans le dossier `templates` du package

fr.lip6.move.promela.xtext.ui. Il s'agit d'un fichier XML qui regroupe les templates que nous avons décidé de proposer. Chacun d'entre eux est défini dans un contexte précis et possède un identifiant unique. On y définit également la position du curseur après l'ajout du template en question. L'ajout du dossier templates dans les propriétés du build fait de sorte que les templates sont actifs par défaut au lancement de l'éditeur. Il n'y a donc plus besoin de les configurer dans les préférences.

```
#define MAX 65535
int cpt;

active [ 1 ] proctype inc ( ) {
  do
    :: ( cpt < MAX )-> cpt ++;
    :: else -> break;
  od;
}
```



The screenshot shows a code editor with a dropdown menu open. The menu lists several templates, with 'For - Template for a for loop' selected. The background code shows a 'for' loop structure: `for (varRef : min .. max){` followed by a blank line and a closing brace `}`.

## 5.4 Formatting

Une autre fonctionnalité intéressante que nous proposons est le formatage du contenu du fichier Promela. En appuyant simultanément sur les touches CTRL + SHIFT + F, l'utilisateur modifie l'indentation et les sauts de ligne dans le programme pour le rendre plus clair et plus lisible. Voici les captures d'écran du programme test.pml avant et après le formatting:

```

test1.pml
#define MAX 133
chan set = [ 8 ] of { byte }; bool b=0;
active proctype p ( ) {
  chan ret = [ 8 ] of { int };
  set = ret;
}
active proctype f ( ) { int i;
  for (i : 1 .. 10) {
    printf ( "i = %d\n", i )
  };
  chan ret = [ 8 ] of { int };
  set = ret;
  if:: if :: 8; 32;
    2;
    fi;
    :: set ? 5;
  fi;
  do:: 3
    :: b
    :: 5 od
}

test1.pml
#define MAX 133
chan set = [ 8 ] of { byte };
bool b = 0;
active proctype p ( ) {
  chan ret = [ 8 ] of { int };
  set = ret;
}
active proctype f ( ) {
  int i;
  for (i : 1 .. 10) {
    printf ( "i = %d\n", i )
  };
  chan ret = [ 8 ] of { int };
  set = ret;
  if
  :: if
  :: 8;
  32;
  2;
  fi;
  :: set ? 5;
  fi;
  do
  :: 3
  :: b
  :: 5
  od
}

```

Le formatting est défini dans le package `fr.lip6.move.promela.xtext` dans le package `formatting` des sources. Ce formateur est écrit en `xtext` dans un fichier appelé `PromelaFormatter.xtext`. Il s'est agit de définir les insertions de sauts de ligne, d'espace et autres caractères suivant les spécifications de notre langage, mais aussi suivant la convention usuelle.

## 6 Transformation ToGal

La réalisation d'une transformation `spin` vers `gal` a été l'aboutissement de notre projet. Nous présenterons, dans un premier temps, plus en détails la vocation d'une telle transformation avant de voir comment on a traduit les concepts `promela` vers `gal`. Ensuite, nous traiterons de l'architecture de notre traducteur et de la solution de visualisation graphique mise au point pour analyser les automates déduits des définitions de processus `promela`.

Pour finir nous reviendrons sur les résultats concrets que nous avons obtenus.

### 6.1 Objectif

Le but ultime et donc de pouvoir traduire des spécifications `Promela` vers une spécification `GAL`, que nous pourrions analyser avec l'outil `ITS` développé par le `Lip6`.

L'intérêt de ceci est que `ITS` et `SPIN` utilisent deux paradigmes opposés de `model checking` avec leurs avantages respectifs selon les modèles.

`Spin` se base sur une méthode explicite, alors qu'`ITS` se base sur une méthode symbolique qui peut s'avérer beaucoup plus performante et moins gourmande en mémoire.

Développer une telle traduction, nous permettra de faire tourner la même spécification mais avec deux outils différents et ainsi déterminer les méthodes les plus adaptées à un type de spécification précis, afin d'utiliser dans

un second temps l'outil le plus adapté.

Aussi, Promela étant un des langages de spécification de systèmes concurrents parmi les plus utilisés, ceci permet à l'outil ITS de toucher une plus large audience.

## 6.2 Mapping entre concepts Promela et Gal

La première étape a été de voir comment traduire les concepts Promela vers ceux Gal avant de s'attaquer à l'implémentation de ces idées (et donc toutes les complications pratiques qui peuvent émerger).

Comme nous l'avons vu en présentant le langage GAL, celui-ci ne contient que des concepts de très bas niveau, à savoir des variables, et des transitions. Il a donc fallu que nous voyions comment exprimer nos concepts d'assez haut niveau (processus, canal de communication, structures) avec ces constructions de base.

Derrière sa syntaxe, Promela modélise des algorithmes de systèmes asynchrones distribués comme des automates non déterministes. Pour réaliser la conversion, un processus a été associé à une machine à état, c'est-à-dire à un ensemble d'états et de transitions.

Pour ce faire nous avons recensé dans chaque processus l'ensemble des positions du pointeur de code, que nous avons représenté par un variable interne nommée PC (pour program counter à l'image de l'assembleur). La plupart des instructions promela correspondent à une action ayant lieu entre deux positions possibles du programme. Pour réaliser ceci en GAL, une variable d'état est associée à chaque processus, et contient sa position courante.

Chaque instruction sera traduite vers une transition Gal. Cette transition sera notamment gardée par la valeur du PC (ainsi que d'éventuelles autres conditions d'exécutabilité). Cette transition contiendra la traduction de l'instruction en actions gal, et la dernière instruction sera la mise à jour du PC.

La traduction des structures conditionnelles se fera vers autant de transitions que d'options, chacune étant gardée par la condition correspondante (en plus de la position courante du programme). Les goto et break correspondront eux à une simple mise à jour du PC.

Tout ceci est capturée dans un objet ProcessRepresentation qui sera utiliser pour instancier plusieurs processus, mais aussi générer une représentation graphique de l'automate.

Les variables et tableaux existant tous les deux dans les deux langages, la traduction est immédiate, il y a juste à s'assurer que les références vers une variable gal soient traduites en une référence vers la variable gal associée. Cette association sera maintenue dans une environnement de conversion.

Les canaux de communication sont représentés à l'aide de trois variables GAL: une variable contenant le nombre d'éléments maximal du tableau, le nombre courant, et un tableau contenant les différents messages transitant au moment considéré. Ce concept est capturé dans CanalRepresentation.

Ainsi une émission dans un canal consiste à rajouter notre message à la position dont l'index est le nombre d'éléments courant, avant d'augmenter cette valeur. Une Réception sera implémentée en récupérant la valeur à l'index 0 avant de réécrire le tableau en décalant tous les éléments d'un cran vers la gauche et de diminuer le nombre de messages courant.

Avec ce système un canal vide (respectivement plein) est détecté quand le nombre d'éléments courant est nul (égal au nombre maximum.)

## 6.3 Architecture

Maintenant que nous avons annoncé comment se fait la traduction en principe d'un modèle promela vers gal, on va voir comment ceci est réalisé concrètement.

L'opération de Traduction est gérée par une classe nommée PromelaToGALTranformer, dotée d'une méthode transformToGal qui prendra en entrée une PromelaSpecification et qui renverra une Specification gal.

C'est cette méthode qui sera invoquée par l'Action eclipse invoquable depuis l'interface graphique.

Lors de la transformation, un Converter, un Environment et diverses représentations "TransfoTime" des concepts promela seront utilisés. Le Converter aura la charge de traduire les expressions et instructions, et l'Environment contiendra les liaisons entre les variables des programmes promela sources et gal cibles.

A coté de cela, des classes utilitaires ont été introduites pour factoriser tout le code redondant.

Nous allons désormais rentrer un peu plus en détail sur le contenu de ces différentes classes.

### 6.3.1 Principales Classes

**Actions** Cette classe `PromelaToGalAction` est invoquée par Eclipse pour lancer la transformation. Cette classe étend la classe eclipse `IObjectActionDelegate`. Elle contient un ensemble de fichiers sur lesquels la transformation doit être réalisée. Cet ensemble de fichier est mis à jour lorsque la sélection change à l'aide de la méthode redéfinie `selectionChanged(IAction a, Iselection s)` qui va ajouter tous les fichiers `promela` (dont l'extension est `.pml`) à la liste des fichiers à traiter.

La méthode redéfinie `run` va, pour chaque fichier, l'ouvrir, créer une représentation mémoire `promela`, réaliser la transformation, sauvegarder le modèles `gal` (sous forme brute, et sous forme "applatie")

**Transformer** C'est la Classe qui va gérer la transformation de la spécification. Tout d'abord, elle va normaliser le modèle `Promela` et ainsi remplacer certains `null` par leur valeurs par défaut. Ceci nécessite un parcours complet de l'AST, du coup nous en profitons pour récolter l'ensemble des instructions `Run`. Grâce à cela, on pourra savoir combien de fois chaque définition de processus sera instantiée. Après cela l'environnement et le converteur sont instantiées, ainsi qu'une Spécification `Gal` vierge. Nous commençons par ajouter toutes les macros, définitions de types et messages, et variables globales. Une fois ceci fait, une représentation `transformation` va être créée pour chaque `Run` (création dynamique de processus), avant d'en créer une pour chaque définition du processus. Quand toutes les `ProcessRepresentation` auront été créées, celles-ci seront instantiées le nombre de fois requis.

**Convertir** Le `Convertir` contient toutes les méthodes de conversion d'instructions et expressions `Promela` vers `GAL`.

Pour chaque instruction, il renverra une liste d'actions `gal` correspondantes, pour les expressions, il pourra les renvoyer soit sous la forme d'une Expression Booléenne soit Entière vu que la distinction existe en `gal`. Il existe aussi une méthode pour obtenir la garde d'une instruction (expression indiquant si l'instruction est exécutable). Cette classe contient une référence vers l'environnement pour pouvoir récupérer les références sur les variables `gal`.

**Environnement** La gestion de l'environnement est assez complexe et a d'ailleurs tout un package dédié.

**Environment** La classe façade `Environment` va contenir un environnement `Local`, et un `Global`. A la résolution des variables, celle-ci regardera dans un premier temps dans l'environnement local (à un processus), puis dans le global en cas d'échec. Deux méthodes permettent de mettre en place un nouvel environnement local, ou de délier celui actuel. Ces méthodes seront invoquées au début et à la fin de l'instantiation de chaque processus.

Pour résumer, cette classe capture le mapping entre les variables `Promela`, et les variables `GAL`.

**Local/GlobalEnvironment** La classe `LocalEnvironment` représente un environnement local. Il contient les associations pour les variables simples (qu'on a appelé `atomic`), les tableaux et canaux vers leur représentation `gal`. Ces trois sont stockées dans 3 `maps` différents. A cela s'ajoute les méthodes pour ajouter une nouvelle liaison, ou récupérer la variable `gal` associée à celle `promela`. Pour finir, une méthode sert à rajouter les déclarations qu'elle contient dans la spécification `gal`. La classe `GlobalEnvironment` étend la version locale en y rajoutant les attributs méthodes associées aux macros et symboles de message.

Il s'agit juste d'une `Map` personnalisée.

**EnvironmentAdapter** Cette classe sert à créer les environnements, à bien les peupler. L'environnement contient juste les liaisons, cette classe va se charger pour chaque variable `promela` de créer celle `gal` qui y sera associée.

Pour chaque type de variables (`MemVar`, `Channel`, `Macro`, `Mtype`), une méthode `handle` va se charger de créer l'équivalent `gal`, et d'associer les deux dans l'environnement.

En réalité, il s'agit d'une classe abstraite. En effet, deux classes statiques internes viennent raffiner la classe pour distinguer la création de l'environnement `Global`, et local. Ces deux classes raffinent une méthode `makeName` qui va déterminer le nom de chaque variable. Ceci permet de préfixer les variables locales du nom de leur processus pour gérer la portée et éviter tout conflit de nom dans le `gal` généré. De plus la version locale va surcharger

les méthodes qui ne concernent que le Global pour lever une exception si jamais elles étaient invoquées.

### 6.3.2 Classes Représentation "Runtime"

Ces classes ont été créées afin de capturer une représentation de nos objets `promela` qui va contenir les différents éléments GAL qui vont réaliser ces concepts.

En plus de contenir dans un objet unique ces différentes données, des méthodes ont été produites pour directement produire les gardes ou actions que l'on associera aux instructions.

**ProcessRepresentation** On peut considérer `ProcessRepresentation` comme classe principale de la transformation, celle qui contient l'essentiel de la logique de traduction.

C'est dans cette classe, qui contient la représentation sous forme d'automate de processus, qu'on trouve tant les méthodes de fabrique statique qui va créer la représentation à partir d'une `ProcessDefinition`, que la méthode pour instancier cette représentation dans une spécification GAL.

**Classes Internes** Pour la réalisation, plusieurs classes internes ont été développées.

On a tout d'abord la classe `PC`, et ses raffinements `PCid` et `PClabel` qui vont représenter les positions possibles dans le code. Le `PClabel` sert à gérer les références vers un certain label dans le cadre du `goto`, et a été introduite car le vrai `PCid` associé au `Label` n'a peut être pas encore été résolu.

Les `Ptransitions` (avec `P` pour `promela`, et pour éviter une ambiguïté avec les transitions `gal`, vers lesquels elles seront traduites), correspondent à une instruction `promela` comprise entre deux positions dans le code. Une éventuelle garde (condition) peut y être associée.

**Création d'une représentation** Voici l'esprit de la procédure pour créer une `ProcessRepresentation`.

Tout d'abord les déclarations de variables sont collectées lors d'une première passe. Ensuite un `PC` initial est créé, puis chaque instruction du bloc va être traitée par une méthode `process`.

Cette méthode récursive va, selon le type de l'instruction, créer une transition correspondante (ex: `assignment`, `send`, `receive`), ou par exemple s'appeler récursivement sur ses sous instructions dans le cas d'une séquence.

Les `Iterations` (`do`), et alternatives (`if`) vont elles aboutir à la naissance de plusieurs transitions.

Diverses méthodes vont permettre de factoriser certains traitement, ainsi on trouvera les méthodes privée `processOpt`, `processElseOpt`, `processGoto`, `processBreak`, `processIntrList`

Un Système a été mis au point pour pouvoir gérer les valeur de `PC`. En résumé, un nouveau `Pc` est créé sauf si un `PC` final a été fourni à l'invocation de la méthode `process`. Ceci est notamment utilisé quand on traite une des options de la boucle, on sait que le dernier élément doit renvoyer à la position correspondant au début de la boucle.

Une fois toutes les instructions traitées, et donc les transitions collectées, une normalisation finale va se charger de remplacer tous les `PClabel` par le `PC` cible qui est désormais connu.

**Instantiation** Cette méthode va créer les variables et transitions représentant le processus `promela` dans la spécification `gal` qui lui sera passée. La méthode commence par créer une variable d'état, qui contiendra la valeur du `pc` courant. Ensuite un environnement local sera créé à l'aide d'un `Adaptateur Local`, et attaché à l'environnement du `converter`. Chaque transition sera ensuite instantiée. Celle-ci sera gardée par une éventuelle condition à laquelle s'ajoutera la vérification que la valeur de l'état courante est la bonne; et une action de mise à jour du `pc` est ajoutée en fin de transition.

A noter qu'un effort particulier a été fait pour commenter les différentes variables et expliquer leur vocation comme on peut le constater avec les exemples de codes générés disponibles en annexe.

Une variante de la méthode `instantiate` avec un flag `inactive` existe pour gérer les processus qui ne sont pas initialement activés, et qui le seront une fois le `run` correspondant rencontré.

**Autres** D'autres méthodes existent:

- Création représentation graphique, avec la méthode `todot`, ce qui sera décrit plus en détail ultérieurement.
- une création et représentation de processus à la volée `createInitProcess` qui sert uniquement pour les définitions de processus `init` qui sont uniques.

**Channel Representation** Comme cela a été décrit dans l'association entre les concepts des deux langages, cette représentation contient une référence sur les 3 variables évoquées.

A cela s'ajoute un ensemble de méthodes pour générer les opérations sur les canaux, `Send`, `Receive`, `Pool`, ainsi que le garde correspondantes.

**Run Representation** Les Runs, c'est-à-dire l'activation dynamique de processus, n'étaient pas gérés dans les premières versions de notre transformation. Ils ont cependant été introduits pour supporter l'ensemble des fichiers du benchmark `beem`.

La représentation contient la variable d'état correspondante à l'état, ainsi qu'une méthode pour mettre celui-ci à jour. Celle-ci devra être revue quand les paramètres de processus seront intégrés.

**UserType/StructRepresentation** Les Structures ne sont pas encore supportées. Cette classe aura pour vocation de représenter une instance de structure lors de la phase de traduction. L'information sera contenue dans un map entre les noms des différents champs, et les variables gal associées (ou éventuellement vers une autre représentation abstraite de structure en cas de structure imbriquée). Des méthodes permettront de récupérer la variable gal ciblée à partir d'une `StructRéférence`.

### 6.3.3 Classes utilitaires.

**GALUtils** Cette classe est une collection de fabriques statiques permettant d'obtenir un objet GAL correctement initialisé. En effet, ceci était nécessaire vu que la Factory EMF à notre disposition nous renvoie des objets avec leurs propriétés non initialisées.

Par exemple, la méthode `public static IntExpression makeGALInt(int i)` va se charger de créer un entier gal à partir d'une valeur `int` en créant les objets nécessaires, et en réglant leurs attributs aux bonnes valeurs. Cette fonction est un peu plus complexe qu'en apparence car il n'y a pas de constantes de valeurs négatives en Gal. Si l'entier demandé est négatif, il faut donc créer une constante avec sa valeur absolue, et l'enrober dans une `UnaryMinus`.

D'autres méthodes permettent de créer les références à une `Variable` à partir de la `Variable`, ou de combiner avec des `And`, ou `Or` deux ou plusieurs expressions booléennes.

**PromelaUtils** Cette classe contient grand nombre de méthodes utilitaires.

On y trouve l'équivalent Gal des fabriques, mais pour les constructions `promela`.

De plus, c'est ici que l'on trouve les méthodes utilitaires pour récupérer les références des `Assignments`, `Send`, `Receive`, comme évoqué dans la partie `grammaire`.

Une méthode `asInstruction(Step)` va aussi factoriser la vérification du typage, et le cast des `Step` vers les `Instructions` (les `steps` sont soit des instructions, soit des déclarations de variables, ou déclarations d'accès exclusifs aux canaux).

**TransfoUtil** Cette classe sert à factoriser la levée des exceptions lors de la conversion pour avertir de données illégales, invalides, ou de constructions du langage non encore soutenues. Ceci permet de pouvoir changer en un seul lieu le type d'exception envoyées (Pour l'instant une `UnsupportedOperationException`, qui est une `RuntimeException`), et le message associé.

Ces méthodes sont importées de manière statiques par le `converter` et les autres classes l'utilisant et les exceptions sont levées de la manière suivante: `throw illegal("This Rus as already been handled")` ou `throw unsupported("Struct are not supported for now")`



## 6.4 Visualisation du Graphe de flot de contrôle

### 6.4.1 Motivation

Afin d'avoir une meilleure vue des résultats de notre transformation et de la représentation des processus que l'on avait mise au point, nous avons mis un processus de génération de graphes qui s'est avéré grandement utilisé pour l'analyse de nos résultats et le débogage.

En effet, ceci était indispensable. Le code Gal produit est une présentation textuelle et aplatie. La structure ne saute pas au yeux, malgré l'effort apporté pour que la vocation des variables soit clairement identifiée et le code commenté. Une description graphique a donc été développée en utilisant le langage `Dot`, et l'outil `graphviz`.

### 6.4.2 Choix Technique

`Dot` a été choisi pour de nombreuses raisons au delà de l'expérience préalable. Ce langage plus très jeune mais super efficace est un des standards pour décrire des graphes et les outils associés sont open-sources. Une fois ceux-ci décrits dans une syntaxe de base très simple, un outil comme `dot` ou `neato` viendra générer une image après avoir agencé le graphe selon un algorithme de layout particulier.

`Dot` permet donc de produire rapidement une visualisation facilement personnalisable.

### 6.4.3 Génération de représentation textuelle

Pour obtenir nos graphes de flot de contrôle il a donc fallu générer une description de graphe au format `dot` à partir de nos `ProcessRepresentation`. La classe possédait déjà toutes les informations nécessaires pour générer la représentation textuelle du graphe.

Dans le graphe, les noeuds/node vont correspondre aux `PCs`, donc positions possibles dans le code et les `arrows/edge` vont correspondre aux différentes transitions.

Pour la génération, trois méthodes `toDot` ont donc été introduites:

- au niveau des `PC` afin de générer la description du noeud correspondant. Celui-ci sera personnalisé selon ces propriétés (en effet les noeuds correspondant à des labels, ou des entrées de boucle ont une forme et couleur particulières.
- au niveau des `PTransition` pour décrire l'`edge` correspondant.
- la principale dans `ProcessRepresentation` qui va se charger d'invoquer la méthode `toDot` sur l'ensemble des `PC` et `PTransition`.

De plus, un marquage de certain état à l'aide d'une `enum PCTAG` a été utilisé pour décorer de manière particulières les états correspondant à un label, ou l'entrée dans un `do` ou `if`

### 6.4.4 Action, intégration plugin

Une action pour produire les graphes a donc été introduite dans le plugin. Celle-ci va générer les différentes descriptions `dot` correspondante aux différentes définitions de processus. Ceux-ci seront ensuite sauvegardés sur le disque, avant qu'une image au format `svg` et `png` ne soient générés si l'outil `dot` est installé.

Tout ceci est déclenchable de la même manière que la transformation Gal.

### 6.4.5 Améliorations envisageables

Cette première version s'est déjà montrée d'une très grande valeur, mais on a tous les éléments pour pouvoir faire quelque chose de bien plus utile et pratique à utiliser.

En effet, il serait envisageable de développer une vue eclipse qui permettrait de voir à la volée les automates associées aux définitions de processus du programme courant. De la même manière, un build continu qui reproduirait les graphes au fur et à mesure, comme des fichiers dérivés/compilés peut s'avérer très pratique pour l'utilisateur.

Tout ce qu'il faut est présent, il suffit juste d'invoquer, combiner les fonctions et développer un frontend pour rajouter ces fonctionnalités.

On pourrait aussi envisager d’offrir une autre représentation du graphe de flot de contrôle en optant pour un graphe bipartites. Ceci nous permettrait d’avoir des descriptions plus précises des transitions en y faisant apparaître à la fois les actions et la garde associée à celles-ci.

La représentation actuelle quant a elle pourrait être améliorée en faisant apparaître l’imbrication des séquences, en les regroupant dans des sous-graphes dot.

Pour finir, la possibilité de customizer les couleurs, choisir le format d’image, pourrait être offerte à l’utilisateur en construisant une page eclipse de préférence à l’image de ce qu’il existe déjà dans le plug-in ITS.

## 6.5 Résultats.

### 6.5.1 La base de Données Beem

Afin d’évaluer les résultats de nos transformations, nous avons utilisé les modèles du jeu de test de la base de données BEEM.

Cette base de données est une collection de modèles (dont la vocation est de doter le domaine du model checking d’exemples standards afin de comparer les différentes approches. Elle a été établie par un groupe de recherche de l’université de Brno, en République Tchèque, mais leurs travaux ont été interrompus. Il y a en tout 57 modèles regroupés en 8 groupes (protocoles de communication, primitives de synchronisation, élections de leader, planification, etc). Chaque modèle possède plusieurs, plus ou moins importantes, comme le fameux repas des philosophes selon le nombre de convives.

Les résultats de l’outil spin sont disponible à l’adresse suivante: <http://spinroot.com/spin/beem.html> et nous ont servis de base de comparaison.

### 6.5.2 Nos résultats

Notre transformation ne permet pas encore de gérer l’ensemble des modèles, mais un nombre conséquent de modèles est déjà géré et la démarche peut déjà être qualifiée de concluante.

Nous gérons actuellement 30% des fichiers de la base `beem`, et nos résultats actuels sont consultables en annexe.

Sur nos fichiers `gal` générés, nous avons fait tourner l’outil de model checking `its-reach` et voici les premières conclusions. (l’ensemble des résultats est consultable en annexe)

Les résultats sont très concluants, et les performances souvent supérieures à `promela`.

Nous avons même réussi à gérer des modèles non pris en charges par `promela`, notamment les modèles `phils7` et `8` où `spin` ne peut suivre l’explosion exponentielle du nombre d’états.

Le modèle du dîner des philosophes (dont une sample de code est disponible en annexe) donne de meilleur bien meilleurs résultats en lors d’une analyse model checking basée sur les méthodes symboliques.

Par exemple malgré le fait que les premiers tests aient été réalisés sur un modeste netbook, les résultats sont saisissants. Par exemple, le repas avec 16 convives aura été traité en moins d’une seconde à l’aide d’`its` sur le `gal` que nous avons généré alors que celui-ci n’est pas géré par `spin` et `divine` qui respectivement mettent 359 et 3840.4 secondes (6 et 64 minutes) (sans toutefois arriver au bout de l’énumération).

On peut voir dans ce tableau les résultats respectifs des 3 outils (`its-reach` sur nos `gal` générés, et `beem` sur modèle `divine`, et `spin` sur `promela`)

Model	Nb Convives	#S	Mem(kb)	Time Gal	T. Spin	T. Beem
<code>phils<sub>1</sub></code>	4	80	2776	0.02	0	0.4
<code>phils<sub>2</sub></code>	5	581	2844	0.04	0	0.4
<code>phils<sub>3</sub></code>	6	729	2808	0.02	0	0.4
<code>phils<sub>4</sub></code>	9	340789	3092	0.26	2.72	9.5
<code>phils<sub>5</sub></code>	12	531440	2924	0.1	4.26	13.9
<code>phils<sub>6</sub></code>	15	1.43489 (+07)	3144	0.16	144	589.7
<code>phils<sub>7</sub></code>	13*	7.19348 (+07)	3428	0.32	568	3965.6
<code>phils<sub>8</sub></code>	16	4.30467 (+07)	3112	0.16	359	3840.4

`Phils 7` est une variante avec un compteur.

Cependant sur d'autres fichiers la différences et moins flagrantes, et sur certains modèles les méthodes symboliques sur les `gal` générés sont moins performantes que `spin` sur le `promela` original.

Ceci semble bien confirmer les résultats de l'étude [LTSMIN: Distributed and Symbolic Reachability](#) par [S Blom](#) qui avait mis en évidence la complémentarités des deux approches, les méthodes explicites marchant mieux sur des modèles au nombres d'états réduit.

## 7 Conclusion

### 7.1 Bilan du projet

En conclusion, nous pouvons affirmer que nous avons atteint les objectifs fixés, même si de nombreuses améliorations peuvent être apportées au fruit de notre travail (nous en parlerons plus en détails dans la partie Perspectives, consacrée à cet effet).

Nous avons développé une grammaire et un métamodèle pour le langage Promela. Parallèlement, nous avons développé les méthodes pour vérifier les programmes promela avec une analyse de typage et une validation.

Pour finir, nous avons réalisé l'objectif final du projet, la transformation d'une spécification Promela vers le formalisme GAL. Ce faisant, notre projet est assorti d'un éditeur xtext eclipse, personnalisé et enrichi de nouvelles fonctionnalités.

### 7.2 Bilans Personnels

#### 7.2.1 Adrien

Ce projet STL a été l'occasion de travailler sur un projet d'envergure. Il m'a permis de découvrir plusieurs "technos", principalement celles associées au framework Xtext, et au développement de plugin Eclipse, mais aussi de perfectionner mes compétences dans certaines autres connaissances déjà, notamment dot et Java.

J'ai aussi pu utiliser en pratiques les connaissances acquises cette année, je pense notamment aux cours orienté langage et génie logiciel à savoir ILP, APS et IL (respectivement Implantation Langages de Programmation, Analyse des Programmes et Sémantique et Ingénierie Logicielle).

Je tenais aussi à remercier Yann Thierry-Mieg notre encadrant et commanditaire pour ce qu'il m'a appris ce semestre tant en terme de connaissances théoriques et techniques que de méthodologies.

Ce projet fut donc globalement une expérience très enrichissante et formatrice à de nombreux égards.

#### 7.2.2 Fjorilda

Ce projet m'a tout d'abord attiré par son côté technique. En effet, les technologies qu'on y a abordées (à savoir: XText, Xtend, Promela, GAL) m'étaient inconnues, et, j'ai pu les découvrir et les utiliser dans un projet d'envergure. Bien que ce projet ait l'air abstrait au premier abord, il s'est avéré beaucoup plus concret dans sa finalité. Effectivement, Eclipse est un outil que j'utilise depuis longtemps et régulièrement, participer au développement d'un plug-in pour cet IDE m'a permis de connaître plus en profondeur son fonctionnement. D'autre part, savoir que ce projet serait utilisé par la suite, m'a beaucoup motivé. De plus, je peux ajouter que l'élaboration d'une grammaire m'a permis de mettre en pratique les connaissances en sémantique acquises lors de l'UE APS.

En conclusion, je peux affirmer que j'ai acquis de nouvelles compétences et que je suis satisfaite de la contribution réalisée par notre groupe.

#### 7.2.3 Julia

Ce projet de STL fut pour moi l'occasion d'aborder des technologies auxquelles je n'ai jamais eu à faire auparavant. En effet, j'ai choisi ce projet dans cette perspective d'autant plus que travailler plus en profondeur sur la grammaire d'un langage de programmation m'intéressait vraiment. Par le passé, je n'ai que très peu eu l'occasion de me pencher sur de telles problématiques, si ce n'est dans quelques enseignements en licence tels que la Théorie des Langages, mais ce n'était que très superficiel. Toutefois, cette année, j'ai pu en apprendre davantage, et d'un autre point de vue, dans le module APS et pu mettre cela en pratique. J'ai pu également découvrir comment se passait le développement d'un plugin Eclipse et approfondir la façon dont j'utilisais cet IDE jusqu'alors. Je suis satisfaite du travail que nous avons fourni et du résultat obtenu.

## 7.3 Perspectives

Les bases ont été posées, mais de nombreuses fonctionnalités, plus ou moins difficiles à mettre en place, peuvent être ajoutées au travail existant.

Ces perspectives peuvent se faire selon les trois axes que nous allons à présent aborder.

### 7.3.1 Support de nouvelles constructions du langage Promela.

Le langage Promela ayant été enrichi dans ses dernières versions, le support des nouvelles constructions de ce langage pourrait être mis en place.

En effet, l'on pourrait supporter des constructions telles que le LTL (Linear Temporal Logic).

De plus, Promela permet l'ajout de code C dans le programme, il serait donc possible de faire en sorte que ceci ne soit pas considéré comme une erreur syntaxique, au moins au niveau de l'éditeur.

Il faut aussi ajouter que la traduction vers GAL ne supporte pas encore toutes les constructions: nous nous sommes concentrés sur les constructions les moins exotiques, afin de supporter tout le banc de test Beem, laissant de côté certaines d'entre elles. Par exemple: les canaux multitypes, les réceptions aléatoires, les structures, ou les paramètres de processus. Certains d'entre eux ont déjà été ébauchés et aucun ne devrait demander de modification majeure qui nécessiterait une refonte de l'architecture déjà conçue. En ce qui concerne les structures, nous pouvons procéder de deux manières: soit faire une deuxième transformation Promela to Promela pour les aplatir en différentes variables, soit créer une `StructureRepresentation` pour lier l'identifiant de la structure vers les variables en GAL.

Actuellement, les blocs atomic sont considérés comme un `d_step` dans la transformation. Il faudrait donc permettre à un atomic de s'interrompre au milieu quand il contient une instruction bloquante. Nous avons réfléchi à une solution, mais celle-ci apporterait des changements conséquents dans la transformation.

### 7.3.2 Perfectionnement de l'éditeur

L'éditeur Promela que nous avons déjà bien amélioré pourrait l'être d'avantage encore.

En effet, nous pourrions revoir le formatage pour quelques points, comme rajouter des espaces/retour à la ligne où cela rendrait le code plus lisible (après chaque instruction conséquente par exemple, pour aérer le texte).

D'autres quickfixes pourraient être développés en se basant sur les erreurs les plus fréquentes. Par exemple, proposer la déclaration d'une variable lorsque l'on fait appel à la référence d'une variable qui n'existe pas encore. L'indication du typage au survol des entités pourrait également être mis en place et serait une fonctionnalité intéressante.

La coloration syntaxique par défaut peut aussi être changée pour rajouter à l'esthétique du code et à sa lisibilité. Celle-ci permettrait par exemple de mieux distinguer les structures de contrôles, les goto, l'indication des types, les canaux des variables mémoire, etc.

Une amélioration du système d'auto-complétion pourrait également être mise en place.

### 7.3.3 Intégration, et nouvelles fonctionnalités du plugin toGAL

Le plugin `toGal` pourrait encore être amélioré pour apporter à l'utilisateur encore plus de fonctionnalités. Une vue eclipse pourrait être développée pour que le graphe de flux de contrôle puissent être directement visualisé.

Il est également envisageable que la génération du code Gal, ainsi que celle des graphes automates de processus, se fasse en arrière plan lors de la transformation.

Une page de configuration du plug-in pourrait aussi être mise au point avec la possibilité de configurer différentes options tel le style graphique des automates, le nom du fichier généré, ou autres ayant plus à trait au fichier gal généré.

Le plug-in mériterait aussi d'être refactorisé afin de passer à la nouvelle architecture `eclipse` en dissociant les actions en une commandes et de multiples handlers, ce qui rendrait plus flexible la manière d'invoquer la transformation, actuellement liée au menu déroulant.

Un standalone invocable depuis la ligne de commande serait aussi des plus appropriés.

## 8 Annexes

### 8.1 Plugins Spin2Gal

Notre Editeur Promela ainsi que le module de transformation `spin2gal` pour convertir les modèles promela vers gal sont disponibles dans `eclipse`, dans le menu `Help > Install New Software` en renseignant l'update site suivant: [https://teamcity-systeme.lip6.fr/guestAuth/repository/download/Pstl\\_Spin/.lastSuccessful/update-site](https://teamcity-systeme.lip6.fr/guestAuth/repository/download/Pstl_Spin/.lastSuccessful/update-site)

### 8.2 Exemples de Transformation

Voici deux exemples de la transformation que notre outil `spin2gal` permet de réaliser. Pour chacun des modèles, on montrera la spécification Promela originale, la spécification Gal générée ainsi qu'une représentation sous forme graphique de l'automate associé au processus.

#### 8.2.1 Simple compteur

Ce modèle décrit un simple compteur.

##### Code Promela original

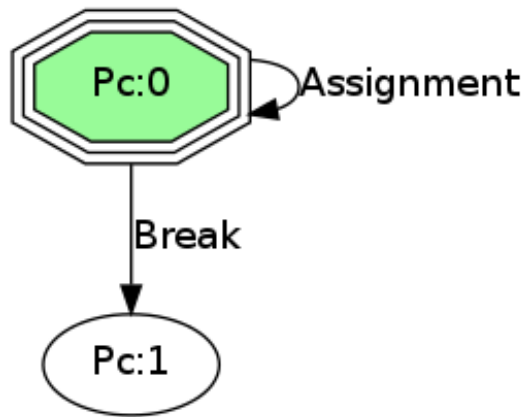
```
#define MAX 42
int cpt;

// Simple compteur
active[1] proctype inc() {
do
::(cpt<MAX)-> cpt++
::else-> break
od;
}
```

##### Code GAL Produit

```
gal firstsample_flat {
    int cpt = 0 ;
    /** @pcvar process inc_0 */
    int inc_0_pcVar_ = 0 ;
    /** @proctrans inc_0 0 -> 1 : Break */
    transition inc_0__t0__from_0_to_1 [inc_0_pcVar_ == 0 && ! cpt < 42] {
        /** @PCUpdate 1 */
        inc_0_pcVar_ = 1 ;
    }
    /** @proctrans inc_0 0 -> 0 : Assignment */
    transition inc_0__t1__from_0_to_0 [inc_0_pcVar_ == 0 && cpt < 42] {
        /** Assignment */
        cpt = 1 + cpt ;
        /** @PCUpdate 0 */
        inc_0_pcVar_ = 0 ;
    }
}
```

##### Illustration du Flot de Contrôle



Resultat

Code dot Voici le code source du graphe.

```

// Representation du processus inc
digraph inc {

// Liste des transitions
  PC_0 -> PC_1 [ label = "Break" ];
  PC_0 -> PC_0 [ label = "Assignment" ];

// Liste des etats
  PC_0[ label = "Pc:0", style = "filled", fillcolor = "palegreen:spinggreen", shape = "tripleoctagon"]
  PC_1[ label = "Pc:1"]
}
  
```

### 8.2.2 YAPE: Yet Another Philosopher Example

Cet exemple se base sur le "Oh combien classique" diner des philosophes de Dijkstra.

#### Code Promela original

```

byte fork[4];

active proctype phil_0() {
think: if
  :: d_step {fork[0]==0;fork[0] = 1;} goto one;
  fi;
one: if
  :: d_step {fork[1]==0;fork[1] = 1;} goto eat;
  fi;
eat: if
  :: fork[0] = 0; goto finish;
  fi;
finish: if
  :: fork[1] = 0; goto think;
  fi;
}

active proctype phil_1() {
think: if
  :: d_step {fork[1]==0;fork[1] = 1;} goto one;
  fi;
one: if
  :: d_step {fork[2]==0;fork[2] = 1;} goto eat;
  fi;
eat: if
  :: fork[1] = 0; goto finish;
  fi;
finish: if
  :: fork[2] = 0; goto think;
  fi;
}
  
```

```

active proctype phil_2() {
think: if
  :: d_step {fork[2]==0;fork[2] = 1;} goto one;
  fi;
one: if
  :: d_step {fork[3]==0;fork[3] = 1;} goto eat;
  fi;
eat: if
  :: fork[2] = 0; goto finish;
  fi;
finish: if
  :: fork[3] = 0; goto think;
  fi;
}

```

```

active proctype phil_3() {
think: if
  :: d_step {fork[3]==0;fork[3] = 1;} goto one;
  fi;
one: if
  :: d_step {fork[0]==0;fork[0] = 1;} goto eat;
  fi;
eat: if
  :: fork[3] = 0; goto finish;
  fi;
finish: if
  :: fork[0] = 0; goto think;
  fi;
}

```

## Code GAL Produit

```

gal phil_1_flat {
  /** @pcvar process phil_0_0 */
  int phil_0_0_pcVar_ = 0 ;
  /** @pcvar process phil_1_0 */
  int phil_1_0_pcVar_ = 0 ;
  /** @pcvar process phil_2_0 */
  int phil_2_0_pcVar_ = 0 ;
  /** @pcvar process phil_3_0 */
  int phil_3_0_pcVar_ = 0 ;
  array [4] fork = (0, 0, 0, 0) ;
  /** @proctrans phil_0_0 5 -> 0 : Assignment */
  transition phil_0_0_t0__from_5_to_0 [phil_0_0_pcVar_ == 5] {
    /** Assignment */
    fork [1] = 0 ;
    /** @PCUpdate 0 */
    phil_0_0_pcVar_ = 0 ;
  }
  /** @proctrans phil_0_0 1 -> 3 : Atomic */
  transition phil_0_0_t1__from_1_to_3 [phil_0_0_pcVar_ == 1 && fork [1] == 0] {
    /** Première instruction de l'atomic*/
    fork [1] = 1 ;
    /** @PCUpdate 3 */
    phil_0_0_pcVar_ = 3 ;
  }
  /** @proctrans phil_0_0 0 -> 1 : Atomic */
  transition phil_0_0_t2__from_0_to_1 [phil_0_0_pcVar_ == 0 && fork [0] == 0] {
    /** Première instruction de l'atomic*/
    fork [0] = 1 ;
    /** @PCUpdate 1 */
    phil_0_0_pcVar_ = 1 ;
  }
  /** @proctrans phil_0_0 3 -> 5 : Assignment */
  transition phil_0_0_t3__from_3_to_5 [phil_0_0_pcVar_ == 3] {
    /** Assignment */
    fork [0] = 0 ;
    /** @PCUpdate 5 */
    phil_0_0_pcVar_ = 5 ;
  }
  /** @proctrans phil_1_0 0 -> 1 : Atomic */
  transition phil_1_0_t0__from_0_to_1 [phil_1_0_pcVar_ == 0 && fork [1] == 0] {
    /** Première instruction de l'atomic*/
    fork [1] = 1 ;
    /** @PCUpdate 1 */
    phil_1_0_pcVar_ = 1 ;
  }
}

```



```

}
/** @proctrans phil_1_0 3 -> 5 : Assignment */
transition phil_1_0_t1__from_3_to_5 [phil_1_0_pcVar_ == 3] {
    /** Assignment */
    fork [1] = 0 ;
    /** @PCUupdate 5 */
    phil_1_0_pcVar_ = 5 ;
}
/** @proctrans phil_1_0 1 -> 3 : Atomic */
transition phil_1_0_t2__from_1_to_3 [phil_1_0_pcVar_ == 1 && fork [2] == 0] {
    /** Première instruction de l'atomic*/
    fork [2] = 1 ;
    /** @PCUupdate 3 */
    phil_1_0_pcVar_ = 3 ;
}
/** @proctrans phil_1_0 5 -> 0 : Assignment */
transition phil_1_0_t3__from_5_to_0 [phil_1_0_pcVar_ == 5] {
    /** Assignment */
    fork [2] = 0 ;
    /** @PCUupdate 0 */
    phil_1_0_pcVar_ = 0 ;
}
/** @proctrans phil_2_0 1 -> 3 : Atomic */
transition phil_2_0_t0__from_1_to_3 [phil_2_0_pcVar_ == 1 && fork [3] == 0] {
    /** Première instruction de l'atomic*/
    fork [3] = 1 ;
    /** @PCUupdate 3 */
    phil_2_0_pcVar_ = 3 ;
}
/** @proctrans phil_2_0 5 -> 0 : Assignment */
transition phil_2_0_t1__from_5_to_0 [phil_2_0_pcVar_ == 5] {
    /** Assignment */
    fork [3] = 0 ;
    /** @PCUupdate 0 */
    phil_2_0_pcVar_ = 0 ;
}
/** @proctrans phil_2_0 0 -> 1 : Atomic */
transition phil_2_0_t2__from_0_to_1 [phil_2_0_pcVar_ == 0 && fork [2] == 0] {
    /** Première instruction de l'atomic*/
    fork [2] = 1 ;
    /** @PCUupdate 1 */
    phil_2_0_pcVar_ = 1 ;
}
/** @proctrans phil_2_0 3 -> 5 : Assignment */
transition phil_2_0_t3__from_3_to_5 [phil_2_0_pcVar_ == 3] {
    /** Assignment */
    fork [2] = 0 ;
    /** @PCUupdate 5 */
    phil_2_0_pcVar_ = 5 ;
}
/** @proctrans phil_3_0 3 -> 5 : Assignment */
transition phil_3_0_t0__from_3_to_5 [phil_3_0_pcVar_ == 3] {
    /** Assignment */
    fork [3] = 0 ;
    /** @PCUupdate 5 */
    phil_3_0_pcVar_ = 5 ;
}
/** @proctrans phil_3_0 0 -> 1 : Atomic */
transition phil_3_0_t1__from_0_to_1 [phil_3_0_pcVar_ == 0 && fork [3] == 0] {
    /** Première instruction de l'atomic*/
    fork [3] = 1 ;
    /** @PCUupdate 1 */
    phil_3_0_pcVar_ = 1 ;
}
/** @proctrans phil_3_0 5 -> 0 : Assignment */
transition phil_3_0_t2__from_5_to_0 [phil_3_0_pcVar_ == 5] {
    /** Assignment */
    fork [0] = 0 ;
    /** @PCUupdate 0 */
    phil_3_0_pcVar_ = 0 ;
}
/** @proctrans phil_3_0 1 -> 3 : Atomic */
transition phil_3_0_t3__from_1_to_3 [phil_3_0_pcVar_ == 1 && fork [0] == 0] {
    /** Première instruction de l'atomic*/
    fork [0] = 1 ;
    /** @PCUupdate 3 */
    phil_3_0_pcVar_ = 3 ;
}

```

}  
}

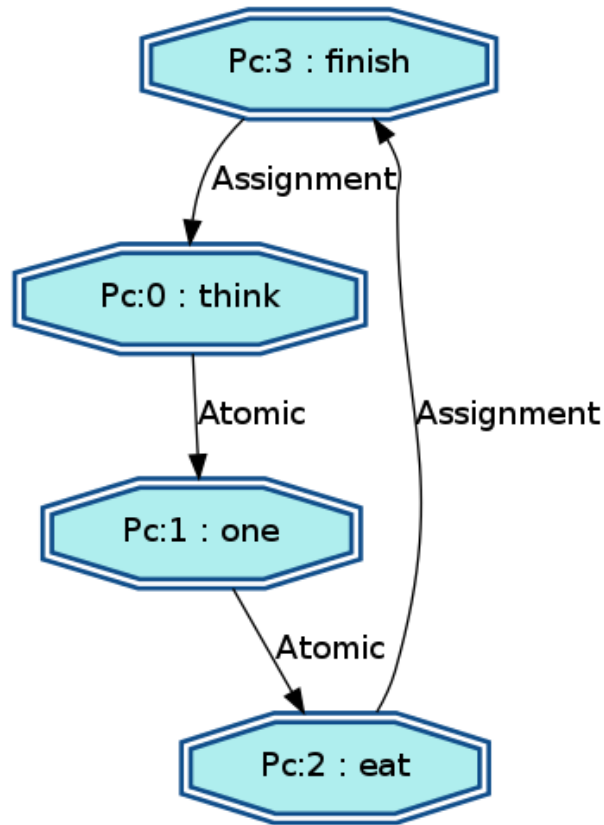


Illustration du Flot de Contrôle

### 8.3 Résultats Benchmark Beem

Actuellement **86 sur 252** sont gérés.  
Voici les résultats sur ces fichiers:

Model	#S	Time	Mem(kb)	fn. SDD	fn. DDD	peak SDD	peak DDD	SDD Hom	DDD Hom
adding <sub>1</sub>	7372	0.1	3868	2	243	5	9092	6	33
adding <sub>2</sub>	836838	19.3	176544	2	2554	5	452350	6	33
adding <sub>3</sub>	1.89438 (+06)	48.56	427964	2	3841	5	921842	6	33
at <sub>1</sub>	39356	0.24	6032	2	1071	5	27170	6	211
at <sub>2</sub>	49445	0.3	7292	2	1261	5	36929	6	211
at <sub>3</sub>	1.71162 (+06)	3.2	41292	2	4619	5	314443	6	276
at <sub>4</sub>	6.59725 (+06)	5.26	70908	2	8347	5	583029	6	341
at <sub>5</sub>	3.19994 (+07)	14.94	165552	2	18185	5	1.34688 (+06)	6	341
at <sub>6</sub>	1.6059 (+08)	57.68	612980	2	51222	5	4.92618 (+06)	6	341
at <sub>7</sub>	8.19244 (+08)	64.98	707544	2	38175	5	6.51877 (+06)	6	406
bakery <sub>1</sub>	1506	0.04	3216	2	350	5	4486	6	83
bakery <sub>2</sub>	1146	0.04	3032	2	214	5	2945	6	83
bakery <sub>3</sub>	32919	0.32	7508	2	2835	5	37835	6	123
bakery <sub>4</sub>	157003	1.16	18368	2	7157	5	124010	6	123
bakery <sub>5</sub>	7.8664 (+06)	55.42	640292	2	88402	5	5.86236 (+06)	6	163
bakery <sub>6</sub>	1.1845 (+07)	64.74	730300	2	86250	5	6.71888 (+06)	6	163
bakery <sub>7</sub>	2.90475 (+07)	226.24	1514804	2	280429	5	1.36659 (+07)	6	163
bakery <sub>8</sub>	2.53131 (+08)	649.58	2823152	2	772348	5	2.57929 (+07)	6	203
blocks <sub>2</sub>	7059	0.12	5868	2	1131	5	33658	6	175
blocks <sub>3</sub>	695420	7.08	125260	2	22130	5	1.2631 (+06)	6	227
blocks <sub>4</sub>	1.04907 (+08)	300.98	1999752	2	194087	5	2.20421 (+07)	6	275
elevator2 <sub>1</sub>	1728	0.02	2860	2	76	5	1743	6	55
elevator2 <sub>2</sub>	179200	0.02	3052	2	473	5	2139	6	103
elevator2 <sub>3</sub>	7.66771 (+06)	0.32	6916	2	343	5	27400	6	82
fischer <sub>1</sub>	636	0.02	2828	2	149	5	1724	6	98

Conti

Model	#S	Time	Mem(kb)	fn. SDD	fn. DDD	peak SDD	peak DDD	SDD Hom	DDD Hom
fischer <sub>2</sub>	21735	0.1	4324	2	224	5	14938	6	122
fischer <sub>3</sub>	2.89671 (+06)	0.44	11248	2	492	5	70885	6	176
fischer <sub>4</sub>	1.27226 (+06)	0.32	8908	2	1445	5	60419	6	210
fischer <sub>5</sub>	1.01028 (+08)	2.9	50372	2	857	5	414697	6	203
fischer <sub>6</sub>	8.32173 (+06)	0.52	11856	2	2071	5	87382	6	238
fischer <sub>7</sub>	3.86297 (+08)	4.6	72064	2	987	5	608286	6	230
frogs <sub>1</sub>	5096	0.34	10932	2	1712	5	117727	6	108
frogs <sub>2</sub>	18209	0.36	13184	2	3558	5	145077	6	104
frogs <sub>3</sub>	760791	7.36	137728	2	9848	5	1.994 (+06)	6	136
frogs <sub>4</sub>	1.74432 (+07)	147.68	1347296	2	219210	5	1.63948 (+07)	6	124
hanoi <sub>1</sub>	35	0	2796	2	235	5	1276	6	127
hanoi <sub>2</sub>	531443	117.68	1300460	2	88175	5	1.43246 (+07)	6	145
krebs <sub>1</sub>	74	0	2668	2	157	5	881	6	62
krebs <sub>2</sub>	74	0	2668	2	157	5	873	6	62
krebs <sub>3</sub>	74	0	2672	2	157	5	877	6	62
krebs <sub>4</sub>	74	0	2668	2	157	5	868	6	62
loyd <sub>1</sub>	722	0.04	3268	2	266	5	7321	6	59
loyd <sub>2</sub>	362882	3.42	66704	2	3011	5	646754	6	65
mcs <sub>1</sub>	7965	0.1	4120	2	550	5	11154	6	161
mcs <sub>2</sub>	1410	0.02	2928	2	267	5	1987	6	161
mcs <sub>3</sub>	571461	2.06	31420	2	5515	5	209199	6	211
mcs <sub>4</sub>	16386	0.04	3068	2	475	5	3678	6	211
mcs <sub>5</sub>	6.05565 (+07)	71.2	764012	2	63166	5	5.22026 (+06)	6	261
mcs <sub>6</sub>	332546	0.12	4684	2	1739	5	19361	6	261
msmie <sub>1</sub>	2336	0.04	3116	2	147	5	2190	6	283
msmie <sub>2</sub>	10560	0.06	3560	2	213	5	3999	6	569
msmie <sub>3</sub>	134846	0.12	4056	2	348	5	8410	6	765
msmie <sub>4</sub>	7.12544 (+06)	0.24	5316	2	518	5	18973	6	1304
peg <sub>solitaire</sub> 1	32183	2.5	44724	2	5937	5	360298	6	348
peg <sub>solitaire</sub> 2	X								
peg <sub>solitaire</sub> 3	X								
peg <sub>solitaire</sub> 4	X								
peg <sub>solitaire</sub> 5	X								
peg <sub>solitaire</sub> 6	X								
peterson <sub>1</sub>	12498	0.1	4448	2	620	5	16060	6	102
peterson <sub>2</sub>	124704	0.14	5552	2	954	5	27263	6	102
peterson <sub>3</sub>	170156	0.22	6124	2	1340	5	32497	6	102
peterson <sub>4</sub>	1.11956 (+06)	2.72	43836	2	8212	5	380165	6	135
peterson <sub>5</sub>	1.31065 (+08)	19.26	267032	2	28154	5	2.66836 (+06)	6	135
peterson <sub>6</sub>	1.74496 (+08)	24.5	294692	2	53033	5	2.87471 (+06)	6	135
peterson <sub>7</sub>	32	0	2832	2	31	5	99	6	178
phils <sub>1</sub>	80	0.02	2776	2	30	5	213	6	75
phils <sub>2</sub>	581	0	2740	2	83	5	803	6	130
phils <sub>3</sub>	729	0	2708	2	42	5	336	6	111
phils <sub>4</sub>	340789	0.02	2996	2	263	5	4018	6	230
phils <sub>5</sub>	531440	0	2824	2	109	5	803	6	219
phils <sub>6</sub>	1.43489 (+07)	0.04	3052	2	301	5	3861	6	273
phils <sub>7</sub>	7.19348 (+07)	0.04	3336	2	434	5	7592	6	330
phils <sub>8</sub>	4.30467 (+07)	0.02	3020	2	317	5	3630	6	291
sokoban <sub>1</sub>	91455	7.56	158084	2	9928	5	998345	6	187
sokoban <sub>2</sub>	761635	10.76	210228	2	30640	5	1.52767 (+06)	6	179
sokoban <sub>3</sub>	7.23805 (+07)	-1860.19	3024280	2	393512	5	2.95491 (+07)	6	477
sorter <sub>1</sub>	14	0	2956	2	89	5	235	6	221
sorter <sub>2</sub>	14	0.02	2956	2	89	5	235	6	221
sorter <sub>3</sub>	18	0.02	3120	2	115	5	291	6	300
sorter <sub>4</sub>	18	0.02	3120	2	142	5	349	6	301
sorter <sub>5</sub>	18	0.02	3120	2	115	5	291	6	300
szymanski <sub>1</sub>	20264	0.14	3884	2	699	5	7249	6	273
szymanski <sub>2</sub>	31875	0.18	4404	2	877	5	9587	6	273
szymanski <sub>3</sub>	1.12842 (+06)	2.1	26220	2	4375	5	182560	6	363
szymanski <sub>4</sub>	2.31386 (+06)	7.12	68828	2	8037	5	485275	6	363
szymanski <sub>5</sub>	7.95187 (+07)	26.26	214984	2	46900	5	1.29386 (+06)	6	453

Les modèles `peg_solitaire` à l'exception du premier n'ont pu être analysés correctement, une `Out of memory error` étant rencontrée par l'outil `its-reach`

## 8.4 Outils utilisés.

Le développement s'est bien entendu fait sur l'IDE Eclipse vu que le framework Xtext s'appuie sur celui-ci pour développer les DSL.

Comme gestionnaire de version, nous avons utilisé subversion([svn](#)) pour le code, mais [git](#) pour la rédaction du rapport, la décentralisation entraînant une plus grande souplesse.

Maven a été utilisé pour automatiser le build et gérer les multiples dépendances.

[Teamcity](#), un serveur de build continu à été utilisé tout au long du projet et s'est révélé d'une aide très précieuse. En effet, celui-ci était lié au dépôt SVN du projet, et tentait de construire les "artefacts" dès que le code était modifié. Ceci a permis de rattraper les erreurs le plus tôt possible et ainsi faciliter leur débogage. Les résultats étaient accessibles depuis l'adresse suivante [consultable depuis un navigateur web](#), même sans disposer de compte sur le serveur de build du Lip6.

## 8.5 Références

Voici les principales ressources utilisées lors de la réalisation de ce projet.

### 8.5.1 Bouquin Xtext-Xtend

- [Xtext Homepage](#)
- [Xtext Official Documentation](#)
- [Xtend Official Documentation](#)
- Implementing Domain-Specific Languages with Xtext and Xtend, Lorenzo Bettini, 2013 packtpub

### 8.5.2 Promela, typing

- [site de référence de spin](#) : description, grammaire, page manuels sur différentes constructions
- Type inference and strong static type checking for Promela, A.F Donaldson, S.J. Gay, Sciences of Computer Programming (2010)165-1191
- Cours d'Algorithmie Répartie du master SAR de C. Dutheillet
- Spin-Introduction, C. Dima, LACL, Université PARIS 12

### 8.5.3 Développement Eclipse

- Eclipse 4 Plug-in Development by Example, Dr Alex Blewitt, 2013 packtpub
- diverses autres ressources en ligne [[stackoverflow](#), [developper.com](#)]

### 8.5.4 Autres

- [Site du lip6 sur sa bibliothèque de manipulation de diagrammes de décisions, libddd](#)
- [LTSMIN: Distributed and Symbolic Reachability](#) par S Blom
- articles Wikipédia [[fr](#)+[en](#)]

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Présentation des technologies utilisées.</b>	<b>4</b>
2.1	Promela/Spin: le langage source	4
2.1.1	Intro	4
2.1.2	Spin	4
2.1.3	Langage	4
	L'environnement d'exécution.	4
	Les Processus	5
	Instructions de modifications de l'environnement	5
	Les structures de "flux de contrôle"	6
2.2	GAL: le langage cible	7
2.2.1	Présentation	7
2.2.2	Langage	7
2.2.3	Outil	7
2.3	Xtext et Xtext : Langages et framework support	7
2.3.1	Le Framework Xtext	8
2.3.2	Développement de langage avec Xtext en pratique.	8
2.3.3	Le langage Xtext	9
	Pourquoi un langage de plus?	9
	Fonctionnalités intéressantes	9
	Petits Bémols à suivre.	10
<b>3</b>	<b>Méta-modèle et grammaire de Promela</b>	<b>10</b>
3.1	Elaboration de la grammaire	10
3.1.1	Démarche	10
3.1.2	Le choix d'une grammaire LL et ses conséquences.	10
	Pourquoi une grammaire LL?	10
	La grammaire xtext GAL comme inspiration	11
	Recursive à gauche des expressions, et associativité des opérateurs.	11
	Factorisation à gauche	11
	Hierarchie des MetaClasses perturbée	12
3.2	Meta Modèle de Promela	12
3.2.1	Spécification Promela	12
3.2.2	Les Instructions	13
3.2.3	Les Expressions	14
3.3	Limites du MM	14
<b>4</b>	<b>Analyse Statique de Promela</b>	<b>15</b>
4.1	Typage	15
4.2	Validation	16
<b>5</b>	<b>Éditeur avancé Promela</b>	<b>17</b>
5.1	Quickfixes	17
5.2	Outline and Labelling	18
5.3	Templates	18
5.4	Formatting	19
<b>6</b>	<b>Transformation ToGal</b>	<b>20</b>
6.1	Objectif	20
6.2	Mapping entre concepts Promela et Gal	21
6.3	Architecture	21
6.3.1	Principales Classes	21
	Actions	22
	Transformer	22
	Convertir	22
	Environnement	22
6.3.2	Classes Représentation "Runtime"	23
	ProcessRepresentation	23
	Channel Representation	24

	Run Representation . . . . .	24
	UserType/StructRepresentation . . . . .	24
6.3.3	Classes utilitaires. . . . .	24
	GALUtils . . . . .	24
	PromelaUtils . . . . .	24
	TransfoUtil . . . . .	24
6.4	Visualisation du Graphe de flot de contrôle . . . . .	25
6.4.1	Motivation . . . . .	25
6.4.2	Choix Technique . . . . .	25
6.4.3	Génération de représentation textuelle . . . . .	25
6.4.4	Action, intégration plugin . . . . .	25
6.4.5	Améliorations envisageables . . . . .	25
6.5	Résultats. . . . .	26
6.5.1	La base de Données Beem . . . . .	26
6.5.2	Nos résultats . . . . .	26
<b>7</b>	<b>Conclusion</b> . . . . .	<b>28</b>
7.1	Bilan du projet . . . . .	28
7.2	Bilans Personnels . . . . .	28
7.2.1	Adrien . . . . .	28
7.2.2	Fjorilda . . . . .	28
7.2.3	Julia . . . . .	28
7.3	Perspectives . . . . .	29
7.3.1	Support de nouvelles constructions du langage Promela. . . . .	29
7.3.2	Perfectionnement de l'éditeur . . . . .	29
7.3.3	Intégration, et nouvelles fonctionnalités du plugin toGAL . . . . .	29
<b>8</b>	<b>Annexes</b> . . . . .	<b>30</b>
8.1	Plugins Spin2Gal . . . . .	30
8.2	Exemples de Transformation . . . . .	30
8.2.1	Simple compteur . . . . .	30
	Code Promela original . . . . .	30
	Code GAL Produit . . . . .	30
	Illustration du Flot de Contrôle . . . . .	30
8.2.2	YAPE: <i>Yet Another Philosopher Example</i> . . . . .	31
	Code Promela original . . . . .	31
	Code GAL Produit . . . . .	32
	Illustration du Flot de Contrôle . . . . .	34
8.3	Résultats Benchmark Beem . . . . .	34
8.4	Outils utilisés. . . . .	36
8.5	Références . . . . .	36
8.5.1	Bouquin Xtext-Xtend . . . . .	36
8.5.2	Promela, typing . . . . .	36
8.5.3	Développement Eclipse . . . . .	36
8.5.4	Autres . . . . .	36