Ecole des Mines de Nantes - FIL A1

Rapport de projet
Elascript

Etudiants :
Maël Naccache
Faouzi Chiheb
Xavier Huberdeau
Valentin Cocaud

Encadrant :
Simon Dupont

Année scolaire 2014 -2015

Table des matières

Introduction

Aujourd'hui, nous sommes capables d'utiliser des ressources informatiques que nous ne possédons pas, via le réseau : c'est le Cloud Computing. On peut augmenter la quantité de ressources comme la diminuer. C'est ce que l'on appelle l'élasticité. On parlera notamment de l'élasticité de l'infrastructure, c'est-à-dire la possibilité de configurer le nombre de machines virtuelles utilisées mais aussi la capacité de chacune d'entre-elles. Cependant, cette élasticité de l'infrastructure présente des limites, comme par exemple le fait d'être peu adaptative et d'être limitée en terme de ressources. Pour pallier ces problèmes, il convient d'introduire la couche logicielle. On parlera alors d'élasticité logicielle, définie comme étant la capacité de s'auto-adapter à la demande et de repousser les limites de l'élasticité de l'infrastructure. Pour ce faire, il faut écrire des stratégies décrivant des comportements autonomes utilisant ces deux élasticités. Le but de ce projet était de faciliter la mise en oeuvre de ces stratégies de gestion des ressources permettant l'élasticité dans le Cloud Computing. Pour cela, nous avons élaboré un Domain Specific Language (DSL) à l'aide de XText et créé un éditeur de diagramme grâce à Sirius. Ces outils sont destinés aux administrateurs du Cloud qui n'ont pas forcément de bagages en terme de développement informatique.

1 Le Contexte

1.1 Élasticité dans le cloud - Principe général

L'une des notions importantes du Cloud est la scalabilité : c'est-à-dire la capacité d'un système à s'adapter à une montée en charge. Cette notion est étroitement liée à l'élasticité.

Qu'est-ce que l'élasticité ? C'est le degré avec lequel un système est capable de s'adapter à une charge de travail (workload) en faisant fluctuer les ressources disponibles automatiquement. Cela permet d'ajuster les ressources du Cloud à la demande, même forte, et ce de manière instantanée, en allouant finement ses ressources sans compromettre les exigences attendues par le client en terme de qualité de service rendu, exigences que le fournisseur s'est engagé à respecter dans les SLAs [1] qu'il a signé avec son client. Le but est donc d'éviter le surdimensionnement qui provoquerait alors un gaspillage énergétique (donc économique) tout autant que le sous-dimensionnement, qui mettrait alors en péril les exigences du consommateur voire mènerait à la violation des SLAs.

Cette élasticité s'opère dans un environnement dynamique dans lequel l'intervention humaine devient de plus en plus complexe voire impossible. En effet, on parle ici d'architectures complexes possédant un nombre important de composants, de contraintes dont la calibration est primordiale pour assurer un fonctionnement optimisé du système (i.e. : répondant aux qualités décrites dans les SLAs sans dépasser les coûts). Il devient alors difficile d'imaginer qu'un humain puisse gérer une telle architecture en temps raisonnable, sans faire d'erreurs. C'est pourquoi l'utilisation de l'Autonomic Computing, qui est selon wikipédia "*l'approche consistant à munir les logiciels et les matériels de garde-fous internes ou externes leur permettant de restaurer automatiquement leur fonction en cas d'altération non planifiée*" est répandue aujourd'hui et permet notamment d'ajuster les ressources au niveau de l'infrastructure d'un système, dans la couche IaaS [2] du Cloud.

1.2 Élasticité de l'infrastructure

L'élasticité de l'infrastructure dans la couche IaaS est la capacité d'ajuster rapidement les ressources selon deux dimensions :

- dimension horizontale : il s'agit d'ajuster le nombre de machines virtuelles, en les ajoutant ou en les retirant de la "réserve" (pool) de ressources
- dimension verticale : il s'agit d'ajuster la quantité de ressources d'une instance de machine virtuelle (RAM, CPU, ...). Généralement, cela se fait en changeant d'offre (comme chez Amazon, Microsoft Azure, ...) en passant par exemple d'une offre "Small Instance" (1 coeur, 1 GB) à une offre "Large Instance" (8 coeurs, 8 GB).

1. Le Service Level Agreement est un accord entre un fournisseur et un client décrivant les qualités qu'un service doit posséder, en terme de performance par exemple.

2. L'Infrastruture As A Service est un modèle du Cloud Computing dans lequel un client va disposer d'une infrastucture informatique virtualisée présente physiquement chez un fournisseur. Le client paie un abonnement pour disposer de cette infrastructure qui peut-être composée d'espaces serveurs, de bande passante, de moyens de stokage, ...

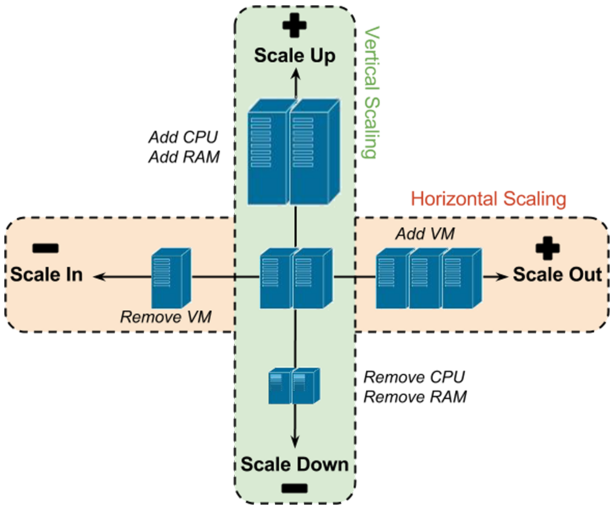

FIGURE 1 – Élasticité de l'infrastructure - Scaling Vertical et Horizontal [1]

Cette élasticité a des limites :

- Les ressources ne sont pas infinies, et il peut très bien arrivé que la demande soit bien trop forte et surpasse les ressources nécessaires.
- Le temps de lancement est très variable. Si on choisit d'ajouter une machine virtuelle, il faut prendre en compte son temps de démarrage, ainsi que le temps de démarrage des services qui la composent. On peut alors imaginer le scénario suivant : la demande est très forte, on demande l'ajout d'une machine virtuelle pour répondre à cette charge. Pendant le temps de lancememnt de cette machine, la demande redescend à un seuil suffisant pour rendre la dernière machine inutile. La machine virtuelle est prête, et elle ne sert à rien car le pic est passé. On a donc un effet de désynchronisation (effet "Ping-Pong") entre l'évolution de la demande et l'ajustement des ressources, ce qui provoque une baisse de la qualité de service par manque de réactivié et un coût inutile car la machine virtuelle nouvellement lancée n'a pas été utilisée.
- Le coût des utilisations partielles des ressources comme dans l'exemple précédent peut-être important à grande échelle, cela dépend en revanche du modèle de paiement.
- Le coût énergétique de telles infrastructures n'est pas négligeable.

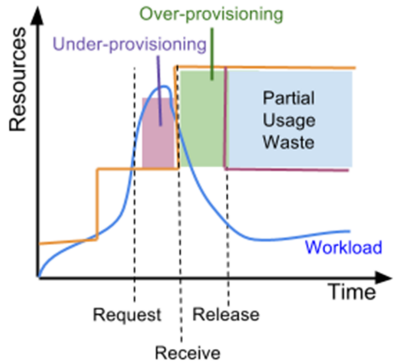

FIGURE 2 – Élasticité de l'infrastructure - Effet [1]

Pour dépasser ces limites, on peut envisager une élasticité du logiciel.

1.3 Élasticité Logiciel

L'élasticité du logiciel est la capacité pour un logiciel de s'auto-adapter (et si possible de manière autonome) pour répondre à la variation de la charge et/ou aux limitations de l'élasticité de l'infrastructure. Elle agit sur la couche SaaS [3]. Sur le même principe que l'élasticité de l'infrastructure, on distingue deux dimensions :

— dimension horizontale : il s'agit d'ajouter ou de retirer des composants à la volée.

— dimension verticale : il s'agit de faire varier des fonctionnalités. Un exemple : imaginons un service qui permet d'afficher de la publicité sur un logiciel, sous trois formes qui sont la vidéo, l'image et le texte. L'idée est de dégrader ou bien d'améliorer ce service en fonction de la charge : si un pic arrive, on affiche les publicités sous forme de texte, cela consomme moins de ressource que la vidéo. A l'inverse, si les ressources le permettent, on peut afficher les publicités sous forme d'images ou de vidéos.

Cela a plusieurs avantages :

— L'ajustement est presque instantané, de l'ordre de la miliseconde, et ne requiert pas le lancement d'un nouvel environnement virtuel.

— Cet ajustement offre une granularité plus fine que l'élasticité de l'infrastructure, ce qui permet d'alléger les ressources nécessaires lorsqu'un pic de charge survient : on atténue le pic en dégradant un service ou bien en suppriment temporairement un composant pour par exemple éviter d'avoir recours à l'élasticité de l'infrastructure lorsque cela n'est pas essentiel (on évite l'effet "Ping-Pong").

L'idée est donc d'utiliser conjointement ces deux types d'élasticité afin de gérer la demande à des échelles différentes. Tout d'abord, ce système apporterait une réactivité importante car il est plus flexible et rapide, c'est ce qui nous importe dans ce type d'environnement qui, nous le rappelons, est dynamique. En outre, l'élasticité du logiciel permet de minimiser le problème de réactivité de l'élasticité de l'infrastructure, en allégeant temporairament la charge dans des cas précis évoqués précédemment. Les deux types d'élasticité sont donc complémentaires.

Les architectures étant complexes, l'élasticité dans le Cloud est gérée par des règles à base de seuils permettant de définir un comportement en fonction du seuil de charge. L'enjeu de ce projet était donc d'allier ces deux types d'élasticité dans un même DSL afin de pouvoir écrire ces règles basées sur les deux élasticités complémentaires. Pour cela, nous avons tout d'abord créé une grammaire.

3. Le Software As A Service "*est un modèle d'exploitation commerciale des logiciels dans lequel ceux-ci sont installés sur des serveurs distants plutôt que sur la machine de l'utilisateur. Les clients ne paient pas de licence d'utilisation pour une version, mais utilisent généralement gratuitement le service en ligne ou payent un abonnement récurrent.*" - Source : Wikipedia

2 Conception du langage : Grammaire

2.1 Conception générale

Pour décrire le langage, il nous fallait écrire une grammaire. Pour qu'elle soit portable et compréhensible par tous, nous avons décidé de l'écrire avec le métalangage Extended Backus-Naur Form, qui est le formalisme le plus connu pour écrire des grammaires. Nous avons donc réaliser une grammaire simple, en nous basant sur ce que nous avons appris durant les cours sur la compilation. Les terminaux étaient les suivants :

«[»	«]»	«||»	«(»	«)»	« ;»
Début d'un bloc parallèle	Fin d'un bloc parallèle	Opérateur de parallélisme	Parenthèse ouvrante pour signifier un début de liste d'arguments pour une fonction	Parenthèse fermante pour signaler la fin d'une liste d'arguments	Opérateurs de séquentialité pour chaîner les opérations entres-elles

Et les règles :
Un Script est un ensemble de «Statement» :

```
Script ::= Statement+
```

Un Statement peut être une commande ou un bloc parallèle, cela permet de chaîner les deux, peu importe leurs types :

```
Statement ::= (Command | Parallelized)
```

Une commande est un simple appel de fonction, où une fonction est un nom et une liste, potentiellement vide, d'arguments séparés par des virgules. Les arguments ne peuvent être que des nombres entiers :

```
Command ::= FUNCTION '(' ( ')' | ParamList ')' ) ';'
FUNCTION ::= [a-Z]([a-Z]|[0-9])*
ParamList ::= NUMBER (',' NUMBER)*
NUMBER ::= [0-9]+
```

Un bloc parallèle est encadré par nos terminaux de début et fin de bloc et contient des statements (pour permettre l'imbrication de bloc parallèle) séparés par des opérateurs de parallélisme :

```
Parallelized ::= '[' Body ']'
Body ::= BodyPart ('||' BodyPart)+
BodyPart ::= Statement+
```

Suite à une rencontre avec notre product owner, Simon DUPONT, nous avons modifié la grammaire de façon à remplacer la définition de fonctions par une liste fixe de fonctions autorisées dans Elascript :

```
Command ::= (FUNCTION | FUNCTION ParamList) ';'
ParamList ::= '(' Param ( COMMA Param )* ')'
FUNCTION ::= (ScaleFunctions | WaitFunction)

ScaleFunctions:
        ScaleInInfra
        | ScaleOutInfra
            ...
```

```
        |ScaleDownSoft

WaitFunction    ::= "wait"

ScaleInInfra    ::= "scaleInInfra"
    ...
ScaleDownSoft   ::= "scaleDownSoft"
```

De plus, il fallait permettre de ne pas écrire les parenthèses si une fonction ne possède aucun argument, c'est-à-dire, pouvoir écrire $scaleDownInfra;$ plutôt que $scaleDownInfra();$:

```
Command ::= (FUNCTION | FUNCTION '(' ( ')' | ParamList ')' )) ';'
```

Pour des questions de temps et de compatibilité avec Sirius, c'est la seul fonctionnalité de la grammaire qui n'a pas encore été implémentée dans la grammaire xText.

Enfin, pour faciliter l'usage de Sirius, et à la demande de Simon DUPONT, nous avons rajouté des terminaux pour identifier le début et la fin d'un script :

«begin»	«end»
Marque le début d'un script	Marque la fin d'un script

La grammaire finale est disponible dans l'annexe A.1.

2.2 Illustrations

Notre grammaire étant non-contextuelle, nous pouvons représenter le résultat de l'analyse syntaxique sous forme d'arbre syntaxique. Voici quelques exemples de code Elascript et de leurs AST associé. (ok pour AST mais vous n'avez pas annoncé l'accronyme)

2.2.1 Exemple 1 :

Cet exemple nous permet de montrer un script Elascript minimal, entièrement séquentiel et n'utilisant que des fonctions.

```
begin
scaleUpInfra(); scaleDownInfra();
end
```

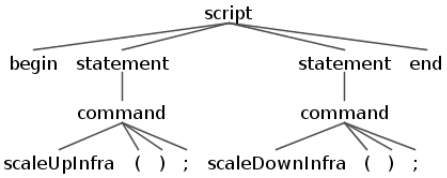

FIGURE 3 – Exemple 1

2.2.2 Exemple 2 :

Cet exemple présente l'utilisation du parallélisme dans Elascript, avec un simple bloc parallèle à deux branches. Il montre aussi l'utilisation d'arguments dans les fonctions.

```
begin
[
    scaleInInfra(); wait();
    ||
    scaleInSoft(42);
]
scaleDownInfra();
end
```

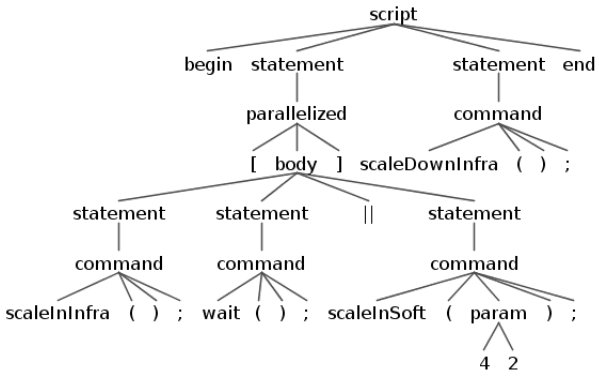

FIGURE 4 – Exemple 2

2.2.3 Exemple 3 :

Voici un exemple plus complet qui regroupe toute les fonctionnalités d'Elascript : bloc parallèles à $n > 2$ branches, imbrication de blocs et paramètres multiples dans les fonctions.

```
begin
wait();
[
        scaleOutInfra(5,2);
        ||
        scaleUpInfra();
        ||
        [
                scaleOutSoft();
                ||
                scaleDownSoft(56);
        ]
]
end
```

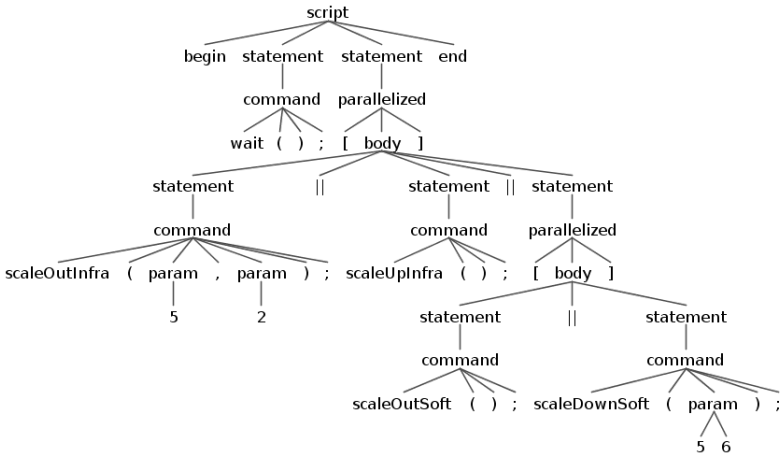

FIGURE 5 – Exemple 3

2.3 Formalisme graphique

Le formalisme graphique reprend les terminaux de la grammaire. Le script commence par un Begin et un End :

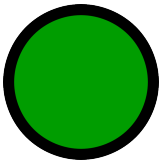

FIGURE 6 – Begin

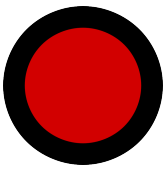

FIGURE 7 – End

Les fonctions dédiées à l'élasticité de l'infrastructure sont représentées par un carré tandis que les fonctions dédié à l'élasticité Logiciel sont représentées par des cercles.

Dans les deux cas, pour représenter le Vertical Scaling, l'icône est coupée par une ligne horizontale et un signe est placé en haut ou en bas. Á l'inverse pour l'Horizontal Scaling, l'icône est coupée par une ligne verticale et le signe est placé à gauche ou à droite.

Nous avons également représenté la fonction wait par une icône d'horloge

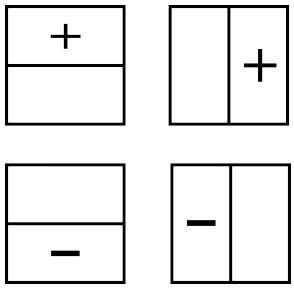

FIGURE 8 – Élasticité de l'infrastructure

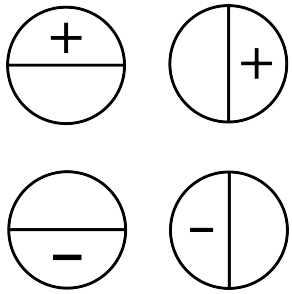

FIGURE 9 – Élasticité Logiciel

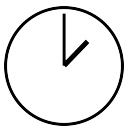

FIGURE 10 – Fonction wait

Le parallélisme est représenté comme dans la grammaire par un icône de début et de fin. Chaque séquence de fonctions est représenté par une branche reliant les deux icônes.

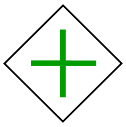

FIGURE 11 – Début d'un bloc parallèle

FIGURE 12 – Fin d'un bloc parallèle

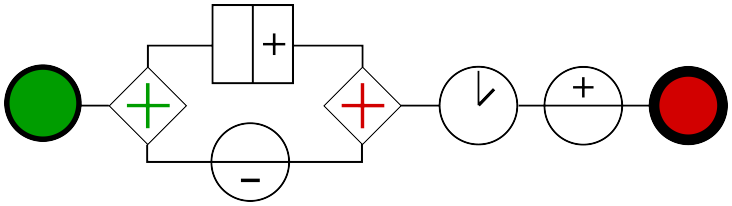

FIGURE 13 – Exemple

Ce formalisme s'est fortement inspiré de la norme BPMN.

3 XText : un framework de création de métamodèle

3.1 Présentation

Xtext est un Framework Eclipse permettant de créer des langages dédiés (DSL). Il permet de générer un analyseur syntaxique depuis une grammaire en utilisant ANTLR. Il permet également de créer un métamodèle de langage en se basant sur EMF. Enfin, il génère un plug-in Eclipse de reconnaissance de ce langage.

3.2 Langage dédié

Un langage dédié est un langage propre à un métier ou à un domaine particulier. Il s'oppose aux langages de programmation classique comme le Java ou le C# par exemple, qui eux peuvent être utilisés dans de nombreux domaines. Cette notion n'est pas propre à l'informatique, par exemple les électriciens ont leur propre langage de modélisation de circuit électrique. En informatique de nombreux langages dédiés existent :

- PHP pour créer des pages web dynamiques et gérer les bases de données
- Ruby pour créer des applications Web
- SQL pour interroger et manipuler une base de données
- Etc …

3.2.1 Avantages

- Cela permet de créer des langages simples pouvant être compris et utilisés par des gens sans connaissances de programmation dans les langages généralistes.
- Un langage dédié est par définition dédié à un domaine spécifique, il va être donc plus concis et l'implémentation sera plus rapide.

3.2.2 Inconvénients

- Le domaine d'application étant limité, la réutilisabilité est moindre
- La multiplication des langages dédiés peut entrainer des problèmes de portabilités
- Au niveau de la performance, les programmes développés dans des langages dédiés utilisent habituellement moins efficacement le temps processeur par rapport aux langages généralistes disposant généralement de compilateurs optimisant l'exécution ou de machines virtuelles très performantes

3.3 Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework (EMF) est un framework de modélisation. A partir d'une spécification de grammaire , EMF fournit des outils permettant de produire un modèle correspondant à cette grammaire. EMF va d'abord créer un métamodèle. Celui-ci va définir les classes, les relations entre ces

classes, les cardinalités... Ce métamodèle est contenu dans un fichier "ecore". EMF permet ensuite de créer un modèle respectant ce métamodèle et fourni un éditeur permettant d'adapter les éléments du modèle afin de pouvoir les visualiser, les éditer avec un système de commandes et les manipuler.

3.4 ANTLR

ANTLR, sigle de ANother Tool for Language Recognition, est un framework libre de construction d'analyseurs syntaxiques. Il prend en entrée une grammaire définissant un langage et produit le code reconnaissant ce langage.

3.5 Fonctionnement de la grammaire

La grammaire Xtext est différente d'une grammaire Extended Backus-Naur Form dans sa syntaxe et dans sa sémantique. Ces différences sont faites pour permettre à Xtext de créer les métamodel et plug-in Eclipse. La différence notable que nous avons été amené à utiliser est la possibilité de spécifié des attributs dans les règles qui seront ensuite utilisées dans les classes générées.

Par exemple, la règle suivante :

```
Param :
        value=INT
;
```

Créera une classe Java Param qui contiendra un attribut value de type int.

Il existe aussi certains attributs spéciaux, notamment l'attribut name qui sert à définir le nom de l'objet dans la représentation graphique de l'arbre d'Eclipse, par exemple :

```
BeginParallel :
        name="["
;
```

La classe aura pour nom «[» dans l'affichage Eclipse. Enfin, on peut créer des listes avec l'opérateur «+=» :

```
Parallel :
         BeginParallel=BeginParallel  (statementLists+=StatementList) (
            PARALLEL_SEPARATOR (statementLists+=StatementList))+ EndParallel=
            EndParallel
;
```

La grammaire complète de Xtext est située dans l'annexe A.2.

4 Sirius : un outil de modeling graphique

4.1 Présentation

Sirius est un projet permettant de créer facilement un modeleur graphique spécifique à un langage dédié. L'outil se base sur un métamodèle EMF. Le développement d'un modeleur graphique se fait en 3 étapes.

1. D'abord on crée une représentation graphique correspondant à chaque entité de notre métamodèle.
2. Ensuite on crée une représentation graphique pour chaque relation entre entités.
3. Enfin, on crée une boite à outil permettant de créer un modèle et de le modifier.

4.2 Représentation graphique d'une entité

On crée un projet Sirius : "ViewPoint Specification Model", c'est dans ce projet que la création du modeleur graphique est réalisée et plus précisément dans le fichier description/project.odesign .

On cherche ensuite à associer notre métamodèle au fichier odesign. Pour cela on crée un "Viewpoint" dans le fichier odesign.

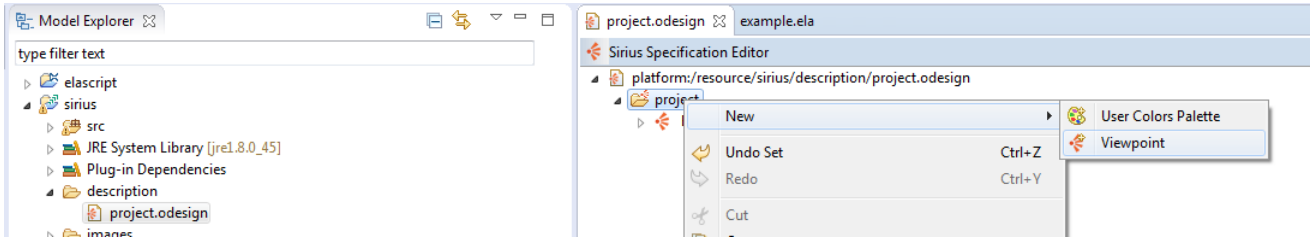

Figure 14 – Création de Viewpoint

On associe la représentation aux fichiers elascript dans la vue Eclipse "Properties".

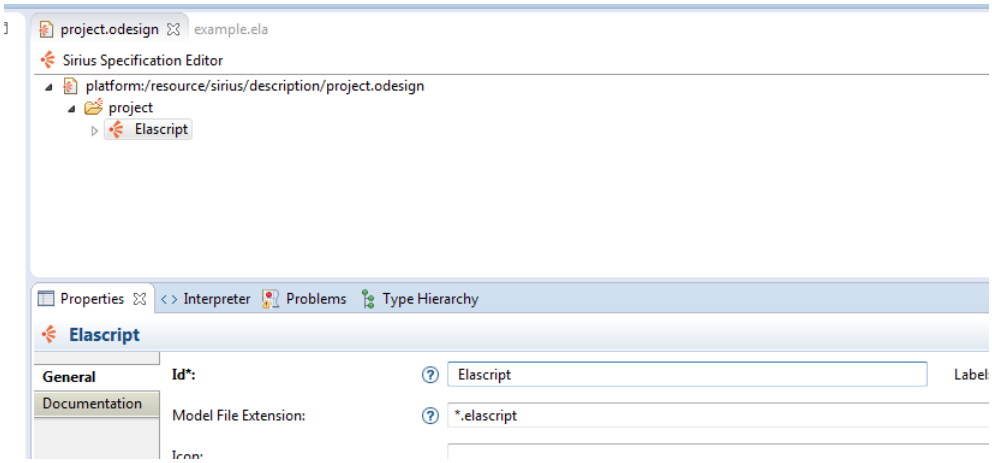

Figure 15 – Association aux fichiers Elascript

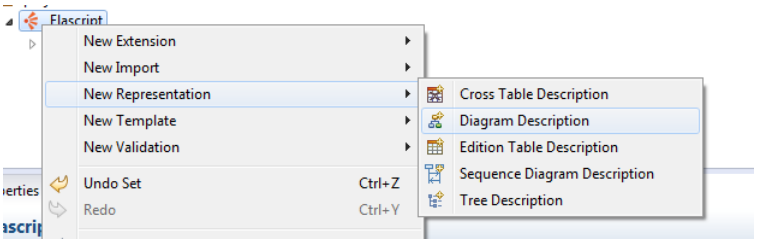

FIGURE 16 – Création de la représentation d'un script

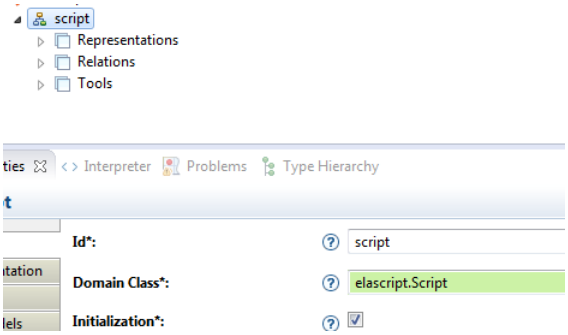

FIGURE 17 – Création de la représentation d'un script

On crée ensuite une nouvelle représentation pour la racine de notre modèle (ici elascript.Script)

Puis, pour chaque entité que l'on veut représenter graphiquement, on crée un nouveau noeud (Node) auquel on associe une entité et un style.

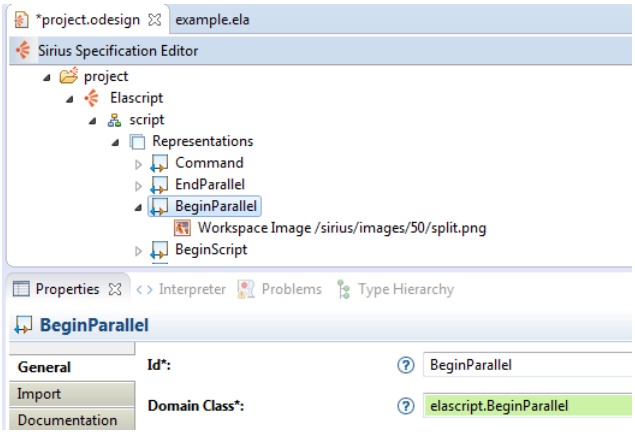

FIGURE 18 – Création d'un node

On recommence cette opération jusqu'à avoir toutes nos entités représentées.

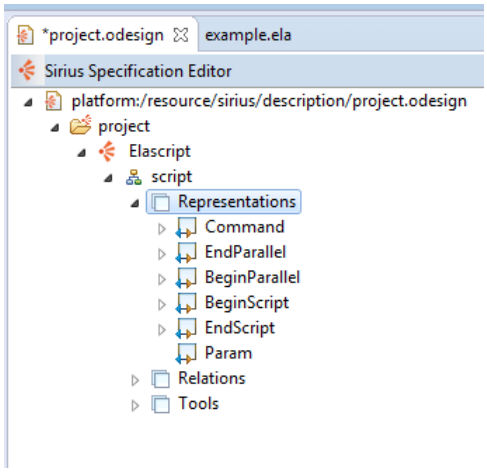

FIGURE 19 – Toutes les entités sont représentés

On voit que toutes les entités sont représentées, mais sans liens. On cherche maintenant à les relier entre-elles.

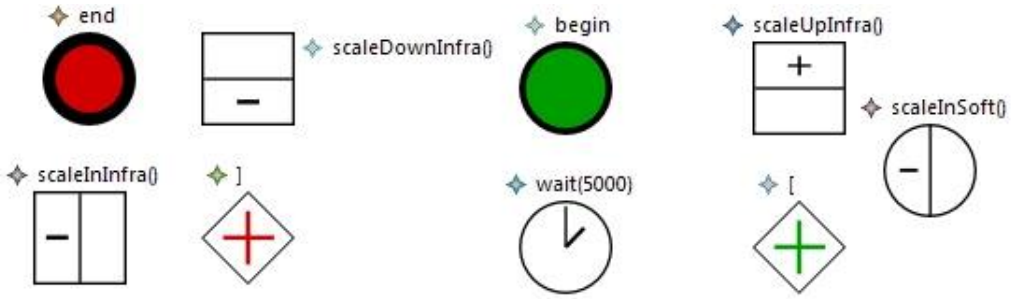

FIGURE 20 – Représentation graphique sans relation

4.3 Représentation d'une relation

La méthode est similaire à la précédente. On va simplement créer des "Relations Based Edge" au lieu de "Node". Par exemple, on veut relier les entités "Command" entre elles.

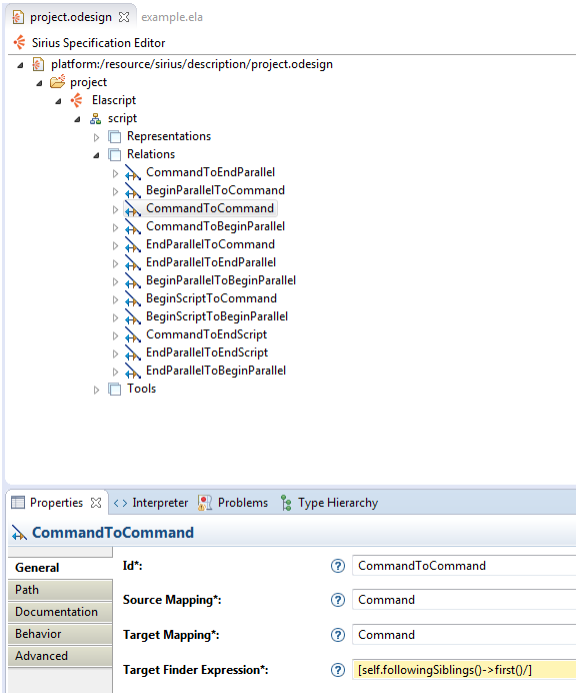

Figure 21 – Création d'un lien

Dans la vue Eclipse "Properties", on spécifie le nom de de la relation, la source ("Source") dans notre cas est "Command" et la cible ("Target") est également "Command". Afin de ne pas relier toutes les commandes entre elles, on précise une condition dans "Target Finder Expression". Ce champ utilise utilise le "langage" Acceleo qui permet de trouver la cible. Ici, "followingSibling()" permet de récupérer une collection d'éléments situés après l'objet courant. "first()" permet d'obtenir le premier objet de la collection. On réalise cela pour toutes les relations. On obtient la représentation suivante :

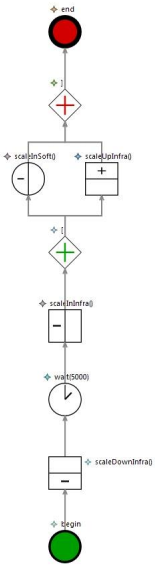

Figure 22 – Représentation graphique avec relations

La représentation graphique est terminée. On souhaite pouvoir créer graphiquement un script.

4.4 Boite à outils

On crée une "Section". Celle-ci correspond à un menu dans lequel va apparaitre toutes les possibilités de création. Dans cette section on ajoute un "Node Creation" qui permet de créer un outil de création de Noeud.

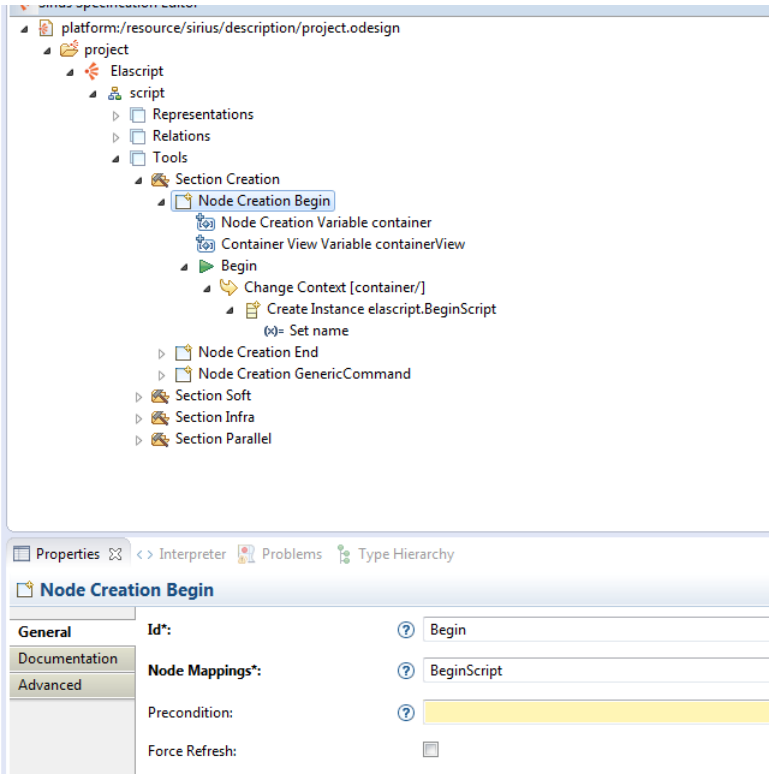

Figure 23 – Création d'un outil

Dans la section "Begin", on peut mettre un semble d'action à réaliser. Ici par exemple on se place dans "container" qui est la racine du Script puis on crée une instance de "BeginScript". On fait cela pour tous les outils que l'on veut créer. On a maintenant un modeleur graphique fonctionnel.

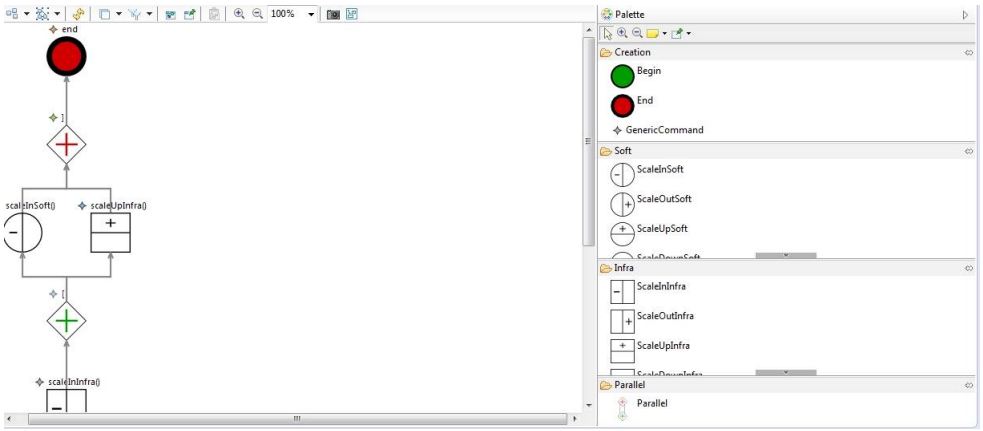

Figure 24 – Modeleur complet

5 Gestion de projet

5.1 Livrables

Le livrable du projet ce composent :

- De la grammaire EBNF
- Du formalisme graphique
- De la grammaire Xtext
- Du projet Sirius
- Du présent rapport

L'intégralité du projet, ainsi qu'un Eclipse pré-configurer est disponible sur le dépôt git (https://github.com/EmrysMyrddin/ElaScript).

Le projet Xtext peut s'utiliser sous deux formes : Exporter les plugins Elascript et les intégrer à Eclipse ou utiliser un Eclipse avec Xtext puis lancer un runtime Eclipse via le projet Elascript. Ces deux méthodes permettent d'avoir accès au support du langage Elascript dans Eclipse.

5.2 Outillage et méthode Agile

Pour pouvoir travailler en commun sur le code source, nous avons créer un dépôt GIT, ce dernier nous a aussi servi de livrable. Pour organiser le travail en groupe, nous avons mis en place un Trello (https://trello.com/b/eXo56Ghg/elascript) pour gérer et affecter les tâches à accomplir sur le projet. Nous avons aussi créer un diagramme de Gantt pour gérer le temps affecté aux tâches.

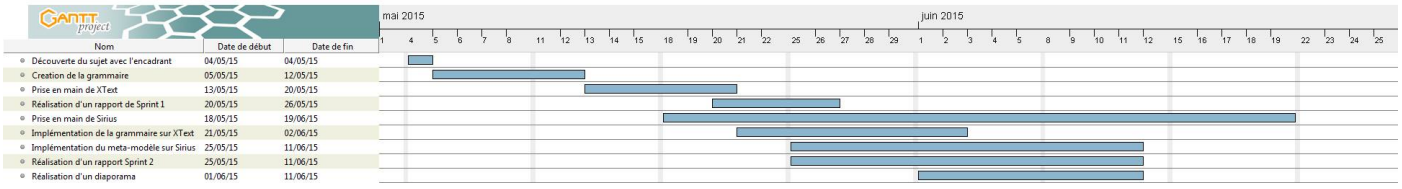

Figure 25 – Diagramme de Gantt du projet

6 Problèmes rencontrés

6.1 XText

Lors de la création de la grammaire Xtext, nous avons rencontré plusieurs difficultés. La première était le manque de documentation et d'exemples d'utilisation de Xtext, ce qui a rendu compliqué la compréhension de cet outil. De fait, la création d'un premier prototype de grammaire a été plus longue que prévu. Heureusement, nous avons reçu l'aide de Massimo TISI, enseignant-chercheur à l'école des Mines, qui nous a aidé à implémenter notre grammaire. Enfin, nous avons eu un problème certaines règles de la grammaire ne sont pas nommées dans l'AST affiché par Eclipse.

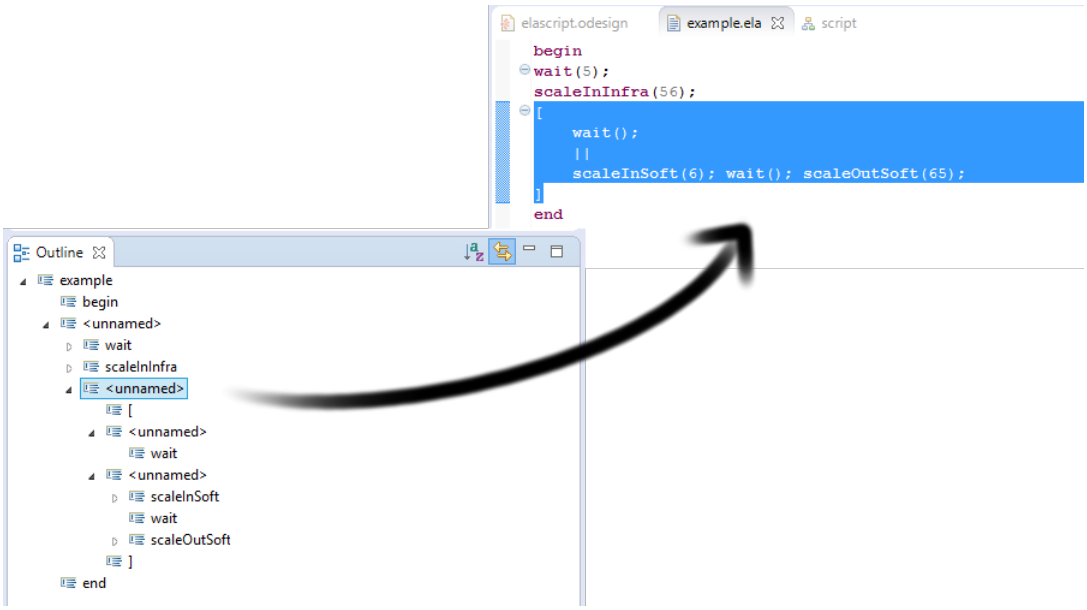

Figure 26 – Problème de nom

En effet, comme nous l'avons expliqué dans la section 3.5, pour nommer une règle dans Xtext, il faut lui donner un ou une attribut «name», or, cette attribut ne peut prendre en paramètre qu'une partie du terminal dans le cas d'un terminal,partie d'une règle dans le cas d'une règle. Cela exclue donc les règles ne servant qu'à grouper d'autre règle, typiquement, la règle Statement ne peut pas être nommé de cette façon car elle ne contient que $Statement : (Command|Parallel)$. Avec l'assistance de Massimo TISI, nous avons trouvé un moyens de contourner ce problème en ré-implémentant une fonction dans les classes généré par Xtext, toutefois, pour des raisons de temps, nous n'avons pas pu l'implémenter.

6.2 Sirius

Sirius est un outil très complet mais il manque cruellement de documentation. La prise en main est de ce fait très compliquée. Le peu de documentation trouvée se résumait en général à des fonctionnalités très basiques. Malgré l'aide de Massimo TISI, nous n'avons pas réussi à faire tout ce que nous souhaitions. Nous n'avons pas réussi à placer les éléments, via la boîte à outils, aux endroits souhaités dans la représentation graphique. Actuellement, nous ne pouvons ajouter des éléments qu'en fin de script, en avant dernière position. Cela induit l'impossibilité de créer des imbrications depuis le diagramme. Depuis le diagramme toujours, nous ne pouvons pas non plus déplacer d'éléments.

6.3 Projet

Lors de la réalisation du projet nous avons rencontré plusieurs problème d'organisation. Tout d'abord, ne connaissant pas d'avance l'organisation d'un projet Xtext et Sirius, il nous a été difficile d'aménager le dépôt et nous avons été amené à le restructurer durant tout le projet en fonction de notre progression dans l'apprentissage de Xtext et Sirius. Ces difficultés de gestion du git on souvent créer des problèmes de déphasage des versions du projet entre les membres du projets, couplé à de nombreux difficultés à configurer les solutions précedement nommés.

En 1 mois, nous n'avons pas eu le temps de prendre en main correctement les 2 outils précédents. Le manque de temps combiné à un manque de documentations et d'exemples a fait que nous n'avons pas pu réaliser ce que nous souhaitions. De plus, malgré notre volonté de bien faire, nous n'avons pas pu appliquer de méthodes agiles. Ce projet ne s'y prête tout simplement pas. Toute les tâches sont séquentielles. D'abord la création de la grammaire puis l'implémentation sous Xtext et enfin l'implémentation sur Sirius.

Conclusion

Ce projet était intéressant, il faisait appel à différents concepts que nous ne connaissions pas ou du moins vaguement. Les outils que nous avons dû appréhender sont complets mais complexe, et la durée du projet était trop courte pour poouvoir réaliser toutes les tâches.

Webographie et Références

➤ [1] Simon Dupont, Yousri Kouki, Frederico Alvares de Oliveira Jr., Jonathan Lejeune and Thomas Ledoux. Software Elasticity : When Infrastructure Elasticity becomes insufficient.

A Annexes

A.1 Annexe 1 : Grammaire finale

Cette grammaire est à daté du 10/06/2015, la dernière version peut être trouver sur : https://github.com/EmrysMyrddin/ElaScript/blob/master/conception/grammar.txt

```
//Rules
Script ::= BEGIN Statement+ END

Statement ::= Command | Parallelized

Parallelized ::= SPLIT Body JOIN

Body ::= BodyPart (PARALLELSEPARATOR BodyPart)+

BodyPart ::= Statement+

Command ::= (FUNCTION | FUNCTION ParamList) SEQUENCIALSEPARATOR

ParamList ::= LP Param ( COMMA Param )* RP

Param ::= NUMBER+

// Functions Names
FUNCTION ::= (ScaleFunctions | WaitFunction)

ScaleFunctions::=
        ScaleInInfra
        | ScaleOutInfra
        | ScaleUpInfra
        | ScaleDownInfra
        | ScaleInSoft
        | ScaleOutSoft
        | ScaleUpSoft
        |ScaleDownSoft

WaitFunction   ::= "wait"

ScaleInInfra   ::= "scaleInInfra"
ScaleOutInfra  ::= "scaleOutInfra"
ScaleUpInfra   ::= "scaleUpInfra"
ScaleDownInfra ::= "scaleDownInfra"
ScaleInSoft    ::= "scaleInSoft"
ScaleOutSoft   ::= "scaleOutSoft"
ScaleUpSoft    ::= "scaleUpSoft"
ScaleDownSoft  ::= "scaleDownSoft"

//Lexems
LP ::= '('
RP ::= ')'
LETTER ::= [a-zA-Z]
NUMBER ::= [0-9]
SEQUENCIALSEPARATOR ::= ';'
```

```
PARALLELSEPARATOR ::= '||'
COMMA ::= ','
SPLIT ::= '['
JOIN ::= ']'
BEGIN ::= 'begin'
END ::= 'end'
```

A.2 Annexe 2 : Grammaire Xtext

```
grammar emn.a1.elascript.Elascript with org.eclipse.xtext.common.Terminals

generate elascript "http://www.a1.emn/elascript/Elascript"

Script :
        BeginScript=BeginScript scriptStatement=StatementList EndScript=EndScript
;

BeginScript:
        name="begin"
;

EndScript:
        name="end"
;

Statement :
        Command | Parallel
;

Parallel :
         BeginParallel=BeginParallel  (statementLists+=StatementList) (
            PARALLEL_SEPARATOR (statementLists+=StatementList))+ EndParallel=
            EndParallel
;

BeginParallel :
        name="["
;

EndParallel :
        name="]"
;

StatementList:
        statements+=Statement+
;

Command:
        ( ScaleFunction | GenericFunction | WaitFunction )  LP ( RP | (params+=
           Param COMMA)* params+=Param RP)  SEQUENTIAL_SEPARATOR
;

ScaleFunction:
```

```
        ScaleInInfra | ScaleOutInfra | ScaleUpInfra | ScaleDownInfra | ScaleInSoft
            | ScaleOutSoft | ScaleUpSoft | ScaleDownSoft
;

WaitFunction:
        name="wait"
;

GenericFunction:
        name=ID
;
ScaleInInfra :
        name="scaleInInfra"
;
ScaleOutInfra :
        name="scaleOutInfra"
;
ScaleUpInfra :
        name="scaleUpInfra"
;
ScaleDownInfra :
        name="scaleDownInfra"
;

ScaleInSoft :
        name="scaleInSoft"
;
ScaleOutSoft :
        name="scaleOutSoft"
;
ScaleUpSoft :
        name="scaleUpSoft"
        ;
ScaleDownSoft :
        name="scaleDownSoft"
;
Param :
        value=INT
;
terminal PARALLEL_SEPARATOR : '||';
//Lexems
terminal LP : '(';
terminal RP : ')';
terminal fragment LETTER: '$' | 'A'..'Z' | 'a'..'z' | '_';
terminal fragment NUMBER: '0'..'9';
terminal SEQUENTIAL_SEPARATOR : ';';

terminal COMMA : ',';
```