ECOLE D'INGÉNIEUR ENSEEIHT

Génie des Logiciels et des Systèmes
Projet long Rapport

Lucien HAURAT
Florian GARIBAL
Guillaume HOTTIN
Paul LARATO

19 décembre 2016

Table des matières

Chapitre 1

Introduction

Ce présent document consiste en un rapport de projet de Génie des Logiciels et des Systèmes effectué au cours de la seconde année du parcours Informatique et Mathématiques Appliquées.

Celui-ci portait sur l'élaboration d'un méta-modèle permettant de représenter un jeu d'exploration et l'élaboration d'outils permettant de créer des modèles de jeu, les vérifier et les transformer en code exécutable dans un langage de programmation de notre choix.

Chapitre 2

Le méta-modèle du jeu

2.1 La syntaxe textuelle *XText*

Pour créer le méta-modèle permettant de représenter des modèles de jeu, nous avons rédigé une syntaxe textuelle *XText* permettant de spécifier les classes constituant un jeu. Cela permet ensuite de générer automatiquement un méta-modèle *Ecore*.

Le méta-modèle est constitué d'une classe principale *Game*, contenant tous les éléments suivants :
- *Player* : la classe permettant de représenter le joueur
- *Item* : la classe permettant de représenter un objet dans le jeu
- *Knowledge* : la classe permettant de représenter une connaissance que le joueur peut apprendre
- *Place* : la classe permettant de représenter un lieu du jeu
- *People* : la classe permettant de représenter une personne
- *Interaction* : la classe permettant de représenter une interaction avec une personne
- *Proposition* : la classe permettant de représenter une des multiples propositions disponibles dans une interaction
- *Action* : la classe permettant de représenter l'action entreprise lors du choix d'une proposition, comme :
  - donne un objet au joueur
  - consommer un objet du joueur
  - apprendre une connaissance au joueur
  - ...
- *Condition* : la classe permettant de vérifier si certaines conditions sont remplies par le joueur en fonction des objets et des connaissances qu'il possède
- *Difficulty* : la classe définissant la difficulté du jeu. Elle permet d'attribuer certains objets et certaines connaissances au joueur au début du jeu
- *Recipe* : la classe permettant de représenter une recette qui transforme un ensemble d'objets en d'autres objets dans l'inventaire du joueur

Pour construire cette syntaxe, et donc le méta-modèle, nous nous sommes appuyé sur les exigences données dans le sujet. Ce travail a été effectué en construisant petit à petit un diagramme de classes UML en fonction des exigences et des entités listées dans celles-ci.

La construction s'est faite assez naturellement. En effet, les exigences mettent en évidence les classes constituant le méta-modèle et les interactions entre celles-ci. En construisant de manière incrémentale le diagramme de classe et en indiquant à quelles exigences chaque incrément répondait, on arrivait au final à une structure qui répondait aux besoins décrits.

Il s'est ensuite agi d'écrire en XText la syntaxe associée afin de bien savoir dans quelle partie du fichier on définissait un objet du jeu, et quelles parties nécessitait des références vers tel ou tel objet.

Par exemple, la création de tous les objets de type *Item* se fait dans le corps du *Game*, puis sont ensuite référencés dans les classes l'utilisant, comme *Path* qui permet de donner des objets au joueur lorsqu'on le traverse. À l'inverse, un *People* sera défini dans un objet de type *Place* car il est fortement lié à ce lieu, il ne peut pas se déplacer.

Il s'agissait alors de savoir si les liens entre les classes était des dépendances ou des compositions.

Nous sommes au final arrivés à un fichier XText fonctionnel qui permet de décrire complètement un jeu selon les exigences. L'implantation effectuée permet de définir toutes les opérations nécessaires à la chaîne de validation

complète du modèle, que nous expliquerons dans le suite de ce rapport, sans apporter de modification structurelle importante à la syntaxe.

2.1.1 Les contraintes OCL du modèle

La construction de la syntaxe textuelle XText permet de répondre à la grande majorité des exigences exprimées dans le sujet. Cependant, certaines d'entre elles ne peuvent pas être exprimées seulement avec le XText et le méta-modèle Ecore. Pour cela, il faut associer au méta-modèle un fichier de contraintes OCL.

Les contraintes qui ne pouvaient pas être vérifiées étaient en particulier :
— la contrainte du poids maximal porté par le joueur
— les contraintes sur l'unicité des chemins et personnes obligatoires, visibles et ouverts/actives
Nous avons pu exprimer ces contraintes dans le fichier *games.ocl*, ce qui permet de vérifier que le modèle de jeu créé vérifier bien les exigences décrites et que le jeu est jouable au démarrage.

De plus, les contraintes imposant que les attributs soient bien initialisés et de valeur correcte ont été ajoutées pour toutes les classes.

Chapitre 3

Exemples

3.1 Exemple du Sphinx

Après avoir générer le méta-modéle grâce au la syntaxe textuelle nous avons pu créer un modèle de jeu, ici le jeu du Sphinx proposé dans le sujet.
Ce jeu est composé de trois places : *Enigme, Sucess, Echec* ; d'un personnage : le *Sphinx* ; d'objets *Tentative* modélisant les vies restantes ; et de chemin permettant d'aller d'une place à une autre.
Grâce à notre modélisation, nous avons pu ajouter au Sphinx la possibilité de *Consommer* une *Tentative* ou donner une *Connaissance : Reussite* au *Joueur*. Mais aussi de se déplacer à la place *Echec* si le nombre de Tentative est nul.
Dans le cas de cet exemple basique, notre modélisation est tout à fait satisfaisante. C'est pourquoi nous avons réalisé un exemple un peu plus évolué.

Pour cette exemple, nous avons écrit un *.net* qui permet de modéliser ce modèle en PetriNet et ainsi pu écrire quelques vérifications en LTL. En revanche, la démarche utiliser pour le réaliser ne fut pas du tout générique, une telle version est détaillée dans la partie quatre.

3.2 Exemple plus évolué

Pour ce modèle, nous avons choisi de rajouter un deuxième personnage et un deuxième *Objet* : Tentative2. Dans la cas d'une réussite au premier *Sphinx*, il lui donne trois Tentative2.
En revanche pour ce modèle plus complet, il manque juste une *Recette*, seul élément non testé du méta-modèle.

Enfin, nous avons pu valider ces modèles grâce aux contraintes OCL.

Chapitre 4

Transformation modèle à modèle d'un jeu vers un réseau de Pétri

4.1 Principe de la transformation

La transformation qui nous intéresse ici est celle d'un modèle de jeu conforme au méta-modèle défini vers un modèle de réseau de Pétri conforme au méta-modèle PetriNet. Cela permet ainsi de vérifier avec des propriétés LTL que l'on peut finir le jeu et qu'il ne finira pas dans un état incohérent.

Cette transformation est très complexe car chaque classe du méta-modèle *Games* interagit avec beaucoup d'autres classes.

Cette transformation n'a pas pu être réalisée dans sa globalité, mais nous avons pu avancer dans la conception de la transformation et l'identification des classes d'un jeu vers des classes d'un réseau de Pétri. Voici les identification que nous avons pu effectuer jusqu'à présent :

Place "place" devient
— une place "place"
— pour chaque Path "path" un arc de la place vers la transition "path"

Path "path" devient
— une transition "path"
— un arc de la transition "path" vers la Place d'arrivée
— pour chaque itemGiven, un arc vers la place de l'inventaire associée
— pour chaque knowledge, un arc vers la place des connaissances associée

Knowledge devient une place "p_knowledge"
Item "item" devient :
— une place "p_item_inv" pour représenter l'inventaire du joueur
— pour chaque Place "place" : une place "p_item_place" pour pouvoir le déposer dans la place

Player devient un jeton dans la place de départ du jeu

Interaction "inter" devient :
— une place "inter" pour représenter la présence dans l'interaction
Si l'interaction est obligatoire dans le lieu, alors elle est directement reliée à la transition représentant le chemin amenant à ce lieu, se substituant au lieu.

Nous n'avons pas réussi à identifier les autres éléments du jeu en élément de réseau de Pétri, en particulier les conditions, ou les objets donnés, qui dépendent eux-mêmes de conditions, et dont on ne peut pas certifier qu'ils seront donnés par le joueur (problème d'activation de transition).

4.2 La transformation en *ATL*

Nous avons voulu écrire cette transformation de manière incrémentale. Cependant, il est vite apparu que nous étions dans l'impossibilité d'effectuer les opérations que nous souhaitions, en particulier de créer les places liées à tous les objets du jeu dans tous les lieux du jeu.

Ainsi, nous n'avons donc pas pu réaliser cette étape, et par conséquent exprimer les contraintes LTL qui s'exprimeraient sur les réseaux de Pétri créés.

Chapitre 5

Implémentation en Java

5.1 Processus d'implémentation

L'implémentation en Java s'est faite assez naturellement avec le *XText* et au diagramme UML que nous avions fait avant de débuter l'écriture du *XText* : il fallait juste créer les classes manuellement avec leurs différents attributs, et rajouter les méthodes autres que les getters et les setters. Par exemple, nous avons ajouté des méthodes `addKnowledge`, `addItem` et d'autres, afin d'ajouter facilement un objet du jeu à l'inventaire du joueur, ou un chemin à un lieu.

5.2 Modifications du *XText* suite à l'implémentation

Durant l'implémentation, nous nous sommes rendus compte que les classes `Action` et `Condition` ont toutes les deux besoin d'une référence à l'instance de `Game`. En effet, quand on vérifie si une condition est vraie, on a besoin d'avoir accès au joueur pour vérifier si il a la/les connaissance(s) requise(s), ou le bon nombre d'objets. De même, lors d'une action qui consiste à donner un objet ou une connaissance au joueur, on a besoin d'un accès à cet objet. De plus lorsque l'action est de prendre un chemin, il faut avoir accès au jeu, pour pouvoir modifier son lieu courant. Nous aurions aussi pu passer l'instance de `Game` lors de l'appel aux fonctions `doAction()` de la classe `Action` ou de la méthode `isTrue()` de la classe `Condition`.

Chapitre 6

Transformation modèle à texte d'un jeu vers un fichier .java

Dans cette partie, le but est de générer un fichier *JAVA* qui sera le main du jeu. En s'appuyant sur l'implémentation du modèle créée précédemment nous devions arriver à générer la création de tous les objets nécessaire afin de pouvoir jouer directement après la génération.
Afin de répondre à ce besoin nous avons utiliser *Acceleo* qui permet de faire de la transformation modèle à texte.

6.1 L'utilisation de template

Afin de clarifier et de rendre plus lisible le code nous avons choisi d'utiliser des template qui ne sont autre que des fonctions appelées dans le partie principale du code. Nous avons donc défini trois template afin de transformer, respectivement, les *Path*, les *People* et les *Interaction*. Ces trois templates prennent en paramètre le jeu ainsi que l'objet à transformer en JAVA.

Ci desous un exemple de template utilisé pour la transformation de personnes :

```
[template public displayPeopleCreation(aGame: Game, peop: People)]

    [comment] CREATION DES INTERACTIONS [/comment]
    [displayInteractCreation(aGame, peop.interaction, peop)/]

    [comment] CREATION DES CONDITIONS DE VISIBILITE DES PEOPLE[/comment]
    List<Condition> condIsVisible = new ArrayList<Condition>();
    [for (c : Condition | aGame.people.conditionsVisible)]
            Condition ['cond_'+c.name/] = new Condition("[c.name/]",
            ['player_'+aGame.player.name/]);
            condIsVisible.add(cond_[c.name/]);
    [/for]

    [comment] CREATION DES CONDITIONS D'ACTIVITE DES PEOPLE[/comment]
    List<Condition> condActive = new ArrayList<Condition>();
    [for (c : Condition | aGame.people.conditionsActive)]
            Condition ['cond_'+c.name/] = new Condition("[c.name/]",
            ['player_'+aGame.player.name/]);
            condActive.add(cond_[c.name/]);
    [/for]

    People [peop.name/] = new People("[peop.name/]",
    [peop.visible/]==1, [peop.isMandatory/]==1, ['inter_'+peop.name+'_'+peop.interaction.name/],
    condIsVisible, condActive);
[/template]
```

6.2 Les difficultés

La difficulté majeure de cette partie du projet a été de comprendre le fonctionnement d'Acceleo. En effet nous avons eu certains problèmes notamment lors de la transformation de variable où l'adresse mémoire s'affichait au

lieu de la valeur String. De ce fait nous avons du faire cette transformation petit à petit et tester, à chaque fois, le bon fonctionnement. De plus, étant donné qu'il est nécessaire de redémarrer Eclipse à chaque modification du fichier *.mtl* pour regénérer un fichier *.java*, le développement de cette transforamation fut très long.

Au final notre transformation fonctionne sur l'exemple 1 et permet de générer des jeux relativement basique. En effet, vu que nous n'avions pas d'exemple plus volumineux au niveau des objets (plusieurs personnes, plusieurs intéractions, etc...) nous ne pouvons pas garantir que la transformation fonctionne dans tous les cas et pour tous types de jeu.

Chapitre 7

Éditeur graphique avec *Sirius*

7.1 Points de vue

7.1.1 Territoire

Dans le point de vue territoire, on trouvera les différents lieux (`Place`), les chemins (`Path`) et et les différentes conditions d'ouverture et de visibilité de ces chemins.

7.1.2 Interactions

Dans le point de vue interactions, chaque place est représentée par un conteneur, et contient chaque objet, connaissance et personnage (`Item`, `Knowledge` et `People`)qui est dedans.

7.2 Choix graphiques

7.2.1 Point de vue *Territoire*

Une `Place` est représentée par un carré dont la couleur varie en fonction de certains paramètres :
— Vert si c'est le lieu de départ du jeu
— Orange si c'est un lieu de fin du jeu
— Gris sinon
Un `Path` est représenté par une flèche entre deux `Place` :
— Rouge si le chemin est obligatoire
— Noir sinon
— Trait plein si le chemin est visible
— Trait pointillé sinon

7.2.2 Point de vue *Interactions*

Une `Place` est représentée par un conteneur avec son nom en haut. La couleur ne change pas en fonction des paramètres cités plus haut, car ceci est réservé au point de vue *Territoire*. Dans chaque conteneur se trouve une liste d'`Item`, de `People` et de `Knowlede`. Ici leur couleur ne peut pas changer, puisque *Sirius* ne permet pas de changer la forme et la couleur d'un noeud dans un conteneur.

7.3 Avancement

Notre éditeur ne peut créer autre chose qu'une `Place` dans chaque point de vue, mais la transformation d'un `xmi` en graphique fonctionne parfaitement, au détail près des conditions d'ouverture et de visibilité des `Path`, dont l'explication est donnée ci-dessous.

7.4 Difficultés

La principale difficulté est dans l'éditeur (création de xmi avec l'éditeur graphique). On peut créer des `Place` sans aucun problème dans les deux points de vue, mais les autres cela ne marche pas.

Le problème pour le `Path` est qu'ils sont déclarés dans le `Game`, et qu'il y a une référence à chaque `Path` qui part d'une `Place` sous forme de liste dans cette dernière. La création du path fonctionne dans le `Game`, mais nous n'avons

pas réussi a trouver en ligne comment ajouter l'instance créée dans la liste de la `Place` de départ correspondante. Ceci ne permet donc pas d'afficher le `Path` qui est créé. De même pour les `Item`, `People` et `Knowledge` qui sont dans chaque `Place` doivent être créés dans le `Game` et ajoutés à la liste correspondante dans la `Place`. L'ajout dans cette liste est le même problème irrésolu que pour le `Path`.

Une autre limitation est les labels des différents éléments. En effet, on ne peut pas faire de l'Acceleo bien construit, on ne peut faire qu'en une ligne, et moins de choses sont possibles. Ainsi nous voulions mettre comme label des `Path` les différentes conditions d'ouverture et de visibilité, mais ces conditions reposent sur soit des `Knowledge`, soit des `ItemInCondition` (qui est une classe qui contient un `Item` et la quantité souhaitée, ainsi qu'un booléen `mustBeExact` si le compte demandé doit être exact), soit les deux. Ces deux éléments sont chacun stockés dans une liste. Le problème ici est la que la syntaxe réduite ne permettait pas de faire quelque chose de complexe comme vérifier si une des deux listes est vide, et afficher tous les éléments nécéssaires (à la fois les `Knowledge` et les `ItemInCondition` sous une forme lisible.

Chapitre 8

Conclusion

Pour conclure ce projet fut très intéressant par le fait qu'automatiser tout ce genre de traitements est très intéressant dans le monde du travail afin de réduire le temps de développement des choses "faciles". Cependant développer ces transformations automatiques était très long et fastidieux car devoir redémarrer Eclipse à chaque modification du fichier est très problématique car cela ralenti énormément l'avancée du travail. De plus les nombreuses erreurs incompréhensibles fournies par Eclipse nous ont grandement ralenti, et le peu de documentation présent sur internet ne nous a pas permis de régler l'ensemble de nos problèmes.

Au final au niveau des exigences nous avons réussi à **créer le fichier Xtext** ainsi que les **contraintes OCL** qui lui sont associées ce qui nous a permis de créer des fichiers décrivant un jeu avec une syntaxe textuelles concrète. D'autre part nous avons réussi à **implémenter le modèle en JAVA** ce qui nous a permis de créer le jeu de l'exemple 1. Ensuite nous avons développé le fichier **Acceleo** qui permet de générer automatiquement le *main* d'un jeu et ainsi y jouer. Cependant nous avons eu des problèmes au niveau du Sirius qui n'est pas totalement fonctionnel. En effet il est possible, à partir d'un fichier .xmi, de visualiser le Sirius mais il n'est pas possible de rajouter des éléments à partir de l'éditeur graphique. Pour finir la transformation ATL ainsi que les contraintes LTL ne sont pas développés car la transformation ATL s'est avérée beaucoup plus compliquée que prévue et nous nous sommes heurté à de nombreux sous-problèmes et sous-sous-problèmes qui nous ont empêché de réaliser cette partie.