



Rapport final
Ingénierie Dirigée par les Modèles
Modélisation, Vérification et Génération de Jeux

FAN Yanghai
MAIGNAN Nicolas
PUECH Pierre
ROUX Thibault
SADURNI Thomas

Table des matières

1	Introduction	3
2	Modèle du jeu	3
2.1	Syntaxe concrète textuelle Xtext	3
2.2	Contraintes OCL	3
3	Exemple du jeu Enigme	4
4	Transformation modèle à modèle vers les réseaux de Petri	4
4.1	Les conditions	4
4.2	Remarques	4
5	Transformation modèle à texte	5
5.1	Vers les propriétés LTL	5
5.2	Vers le langage Java	6
6	Exemples : Mini RPG	6
7	Conclusion	8

1 Introduction

Ce document est un rapport du projet d'Ingénierie Dirigée par les Modèles. Celui-ci portait sur la conception d'une chaîne de modélisation, de vérification et de génération de code pour des jeux de parcours. Dans notre cas, nous avons choisi de générer le code en Java, pour avoir une interface texte simple permettant de tester le jeu décrit par le modèle et de valider la jouabilité.

La modélisation consistera à concevoir un langage dédié pour créer le jeu sous forme d'un modèle et la vérification permettra d'assurer qu'il existe bien une solution pour le jeu.

Nous utiliserons donc les outils vus en TD comme Xtext, OCL, Acceleo, LTL, Tina.

2 Modèle du jeu

2.1 Syntaxe concrète textuelle Xtext

Contrairement aux TPs/mini-projet, nous générons le métamodèle à partir de la syntaxe concrète du langage définie en Xtext, nous pourrons ensuite générer automatiquement le méta-modèle *Ecore*.

Voici les attributs de notre Xtext :

- Game : Le jeu en lui-même, composé d'un *explorateur*, de *lieux* (avec un lieu de départ spécifique) et de *GameElements*.
- GameElement : Peut être un *Lieu*, un *Chemin*, un *Objet*, ou une *Connaissance*. Ce sont les éléments de base d'un jeu de découverte.
- EntiteLieu : Peut être une *Personne*, un *PackObjet*, ou une *ConnaissanceLieu* et contient une condition de visibilité dans le jeu.
- ConnaissanceLieu : Les connaissances acquises dans un lieu.
- Explorateur : Le joueur principal du jeu.
- Personne : Les *PNJ* du jeu avec qui l'explorateur peut interagir.
- Lieu : Un lieu du jeu.
- Objet : Un objet dans le jeu avec une taille.
- PackObjet : Un pack d'objet avec une quantité d'objets à l'intérieur.
- Chemin : Un chemin liant un lieu à un autre. Il peut : être ouvert, être visible, donner des récompenses, ou forcer l'explorateur à donner des objets pour pouvoir être accessible.
- Connaissance : Une connaissance pour l'explorateur.
- Condition : Une condition pour acquérir une connaissance, un objet, un lieu, un chemin. Défini à partir d'opérateurs, de conjonction, de littéraux que nous ne détaillerons pas ici.
- Recompense : Une récompense pour l'explorateur. Peut être une connaissance ou un objet avec une condition d'obtention.
- Action : Une action de l'explorateur. Peut consommer des objets ou des connaissances et peut être réalisable sous condition(s).
- Interaction : Une interaction avec l'explorateur. Sous forme de liste de choix, peut donner des récompenses.
- Choix : Un choix de l'explorateur, sous forme de liste d'actions.

2.2 Contraintes OCL

La syntaxe textuelle permet de répondre à la grande partie des exigences données dans le sujet. Nous n'avons donc pas rajouté énormément de contraintes dans le fichier OCL (*game.ocl*). Hormis les contraintes de nom non vide sur certains attributs, nous avons rajouter la contrainte correspondant a l'exigence *E6* : *Le nombre d'objets que peut posséder un explorateur est limité par la taille cumulée des objets possédés* avec la taille de l'inventaire de l'explorateur ne pouvant pas être dépassée par la taille de la somme des objets de l'inventaire.

3 Exemple du jeu Enigme

Une fois que nous avons fait la génération du méta-modèle avec Xtext, nous sommes passé à l'exemple du sujet, nommé *enigme.game*. Cet exemple est caractérisé par 3 lieux : *Succès Echec* et *Enigme* qualifiés clairement par leur nom et une personne, le *Sphinx*. Le nombre de réponses possibles est caractérisé par l'objet *Tentative* limité à 3, la connaissance est *Réussite* et le chemin menant de *Enigme* à *Echec* est visible seulement si le nombre de *Tentative* est nul.

Ainsi pour cet exemple, nous avons codé à la main ce que donnera plus tard la génération de code en Java dans le fichier *enigme.java*, un exemple de PetriNet (*enigme.net*) et ses propriétés LTL (*enigme.ltl*).

4 Transformation modèle à modèle vers les réseaux de Petri

La transformation qui nous intéresse ici est la transformation d'un modèle de jeu vers un modèle de réseau de Petri, que nous avons fait au cours du miniprojet.

Avec ce modèle, nous pourrions vérifier que le jeu peut finir dans un état cohérent.

Nous citons ci-dessous quelques transformations :

- Le jeu devient nécessairement un PetriNet.
- Un lieu du jeu devient une place de PetriNet.
- Un chemin du jeu devient une transition liant deux lieux, menant à une structure de condition si besoin.
- Une interaction est une place avec une transition par choix de départ (avec structure de condition).
- Un choix est une place qui a des transitions vers les places des choix suivant (avec structure de condition)
- Une récompense est une transition qui ajoute des connaissances ou des objets selon des conditions (connaissance pas encore obtenue, place disponible dans l'inventaire).
- Un objet devient deux places, une avec la quantité actuellement possédée, et l'autre avec la quantité maximale au vue de la taille de l'inventaire totale moins la quantité actuelle (la somme des jetons des deux places fera donc toujours la quantité maximum de l'objet que l'explorateur peut garder dans son inventaire). Il y a également une place par lieu pour ce type d'objet, pour savoir combien d'occurrences de l'objet sont au sol dans le lieu, avec une transition pour prendre un objet et une autre pour en poser (avec conditions).

Pour chaque règle, nous avons ajouté sa description en commentaire pour plus de clarté.

4.1 Les conditions

Le principe de la modélisation est le suivant : on part d'une place *condition* sur laquelle se branche le chemin ou l'interaction dont on doit vérifier la condition, une place *condition vérifiée* qui se branche sur l'exécution du chemin ou de l'interaction et enfin une place *condition fausse* qui permet de revenir sur le lieux ou le personnage car la condition n'est pas vérifiée. Pour vérifier la condition (sur l'exemple ci-dessus, la condition est $(a = \text{true} \text{ et } b = \text{true}) \text{ ou } (c < 3)$ avec a, b des connaissances et c un objet dont la taille d'inventaire limite la possession à 10 exemplaires maximum), il faut que au moins une des disjonction soit vérifié. Ainsi, on crée pour chaque disjonction une transition de *condition* à *condition vérifié* avec des readArc sur les places des connaissances correspondantes (connaissance/objet x si on doit vérifier $x = \text{true}$ ou $x \geq n$, inverse x si on doit vérifier $x = \text{false}$ ou $x < n$). Ainsi, si une disjonction est vérifié, une des transitions sera empruntable et le traitement post-condition sera atteint. Sinon, pour montrer qu'une condition est fausse, il faut montrer que son inverse est vrai. On a alors des transitions en séries pour chaque ancienne disjonction, avec des transition en parallèle pour chaque ancienne conjonction. Des transtions de retour sont présentes entre chaque disjonction pour ne pas atteindre un état bloquant du réseau de petri si la condition est vraie mais que l'on commence par vouloir montrer le contraire.

4.2 Remarques

Nous avons eu beaucoup de mal à transformer nos *Litteral* et nos *Conditions*, qui utilisent des conditions booléennes, en élément de PetriNet. Nous n'avons pas entièrement fini la transformation, et il y a forcément des améliorations possibles sur ce qu'on a implanté, mais nous n'avons pas eu assez de temps.

Pour générer le *.net*, on se sert du miniprojet en utilisant le fichier *petrinet2tina*.

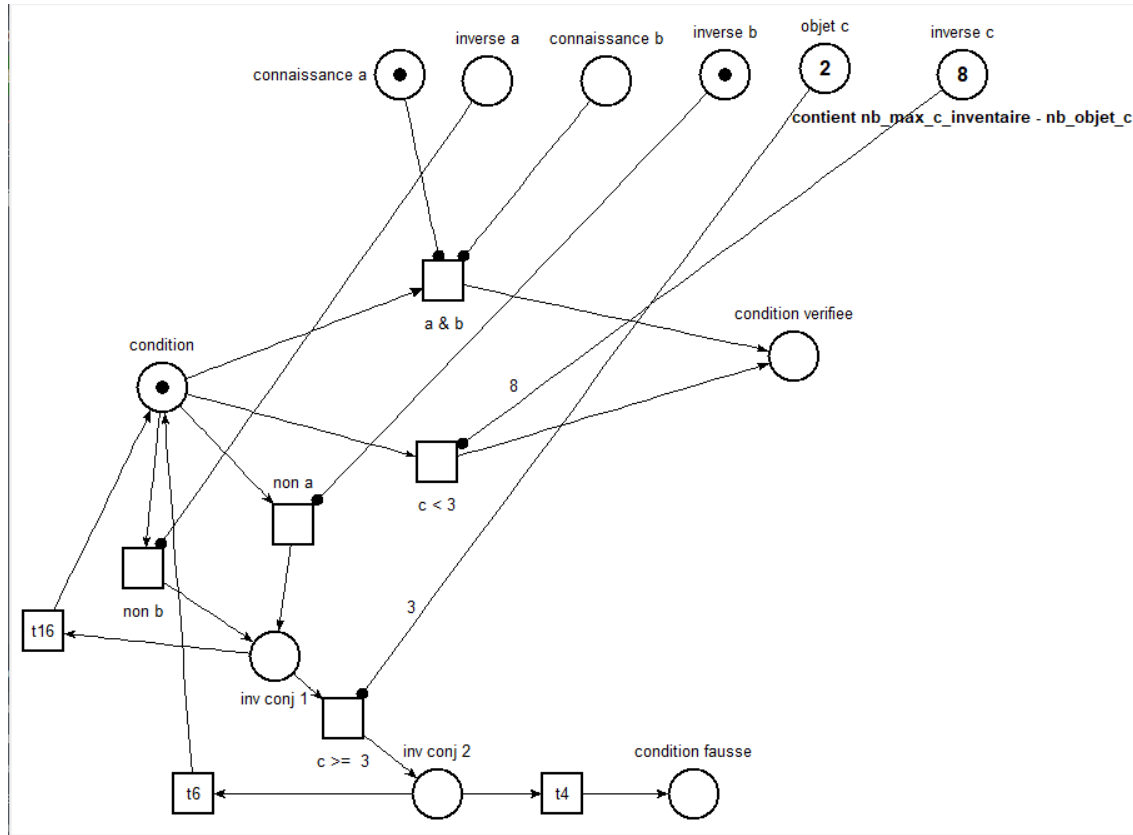


FIGURE 1 – Modélisation d'une condition en réseau de Petri

5 Transformation modèle à texte

5.1 Vers les propriétés LTL

Nous avons fait les propriétés LTL mais comme l'ATL ne fonctionne pas correctement, nous n'avons pas pu les tester.

```
[comment encoding = UTF-8 /]
[module toLtl('http://www.game.idmprojet/Game')]

[template public gameToLtl(aGame : Game)]
[comment @main/]
[file (aGame.name + '.ltl', false, 'UTF-8')]

op finished = [for (la:Lieu | aGame.lieuxArrivee) separator ('\n')]
[la.name/][/for];

['[]' /] (finished => dead);
['[]' /] (dead => finished);

[for (la : Lieu | aGame.lieuxArrivee) separator ('\n')]
- <> [la.name/];
[/for]

[/file]
[/template]
```

FIGURE 2 – Propriétés LTL

L'idée est de mettre une condition *finished* à vraie si on est sur un lieu d'arrivée, à ce moment là on vérifie que si on a *finished* alors on a blocage et s'il y a un blocage, il est *finished*. Ceci permet de vérifier que le jeu finit bien dans un lieu d'arrivée.

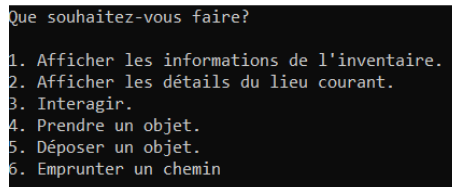
5.2 Vers le langage Java

Pour la transformation vers le langage Java nous utilisons Acceleo et nous nous appuyons sur l'exemple de l'énigme que nous avons codé en Java et à la main.

Nous commençons par créer le fichier sous le nom *nomDuJeu.java* puis nous créons la fonction *main*, qui permettra à l'utilisateur de lancer le jeu.

Organisation du fichier :

- Définition de l'énumération des lieux avec la fonction *definitionLieuxToJava()*
- Définition des objets, des connaissances, de la localisation et de l'inventaire de l'explorateur avec les fonctions respectives *definitionObjetsToJava()*, *definitionConnaissancesToJava()*, *initialisationExplorateurToJava()*
- Ensuite, on entre dans une boucle et on affiche les caractéristiques d'un lieu avec les choix possibles pour l'utilisateur dans celui-ci avec *lieuToJava()* qui utilise *descriptionToJava()* affichant le contenu d'un *Text*.
- Définition de toutes les fonctions nécessaires pour générer le jeu en Java, les détailler toutes ici serait trop long mais leurs noms sont le plus souvent très explicites. Par exemple :
 - *menu* : affiche le menu pour l'utilisateur



```
Que souhaitez-vous faire?
1. Afficher les informations de l'inventaire.
2. Afficher les détails du lieu courant.
3. Interagir.
4. Prendre un objet.
5. Déposer un objet.
6. Emprunter un chemin
```

FIGURE 3 – Exemple du menu

- *infosLieu* : renseigne les infos d'un lieu comme les personnages, les chemins et les objets visibles.
- *deposerObjet* : affiche d'abord les objets possédés puis demande au joueur quel(s) objet(s) il veut poser dans le lieu où il se trouve.
- *ramasserObjet* : permet au joueur de ramasser une quantité d'objet ramassable dans le lieu
- *prendreChemin* : laisse choisir le chemin que le joueur peut emprunter.
- *"attribut"ToJava* : convertissent les attributs du *Xtext* en Java.

Le principe de transformation en Java est simple : on associe à chaque objet un int pour sa quantité dans l'inventaire, un int pour chaque lieu où il est présent et un int pour chaque lieu où il peut être déposé, et à chaque connaissance un bool pour l'inventaire et un bool pour chaque lieu où elle est présente. A partir de là, on vérifie facilement les conditions comme des combinaisons logique de ces variables.

Le reste du jeu est alors une série de switch case et de if très simples sur les choix du joueur, où les choix qui sont proposés au joueur sont ceux respectant leur condition.

Sur nos exemples (*voir prochaine section*), la transformation en Java fonctionne correctement.

6 Exemples : Mini RPG

Mini RPG est, comme son nom l'indique un jeu RPG réduit. Un explorateur (le joueur) navigue entre différents lieux (*Armurie*, la *Plaine*, le *Cimetiere*, le *Banquet...*). Il peut ramasser des objets, interagir avec des Personnes présentes dans le lieu où il se trouve (il peut provoquer le *Golem* en duel, attention à ne pas mourir!). Globalement le jeu se présente sous la forme suivante :

1. Une *armurerie* où le joueur commence avec 1 *hache* et 1 *pioche* au sol, et un *Artisan* qui peut crafter 1 *lance – pierre* si le joueur lui donne 1 *corde* et 2 *brindilles*.
2. Une *plaine* où il y a un *tréant*, un *golem* et un *phœnix*. Le *tréant* est tuable avec la *hache* et peut donner 2 *brindilles*, le *golem* est tuable avec la *pioche* et donne 1 *pierre*, le *phœnix* est visible si le joueur a sa *localisation_phœnix* ou le *lance – pierre*, et il peut lui récupérer 1 *plume* s'il lui tire dessus avec le *lance – pierre* en utilisant une *pierre*.

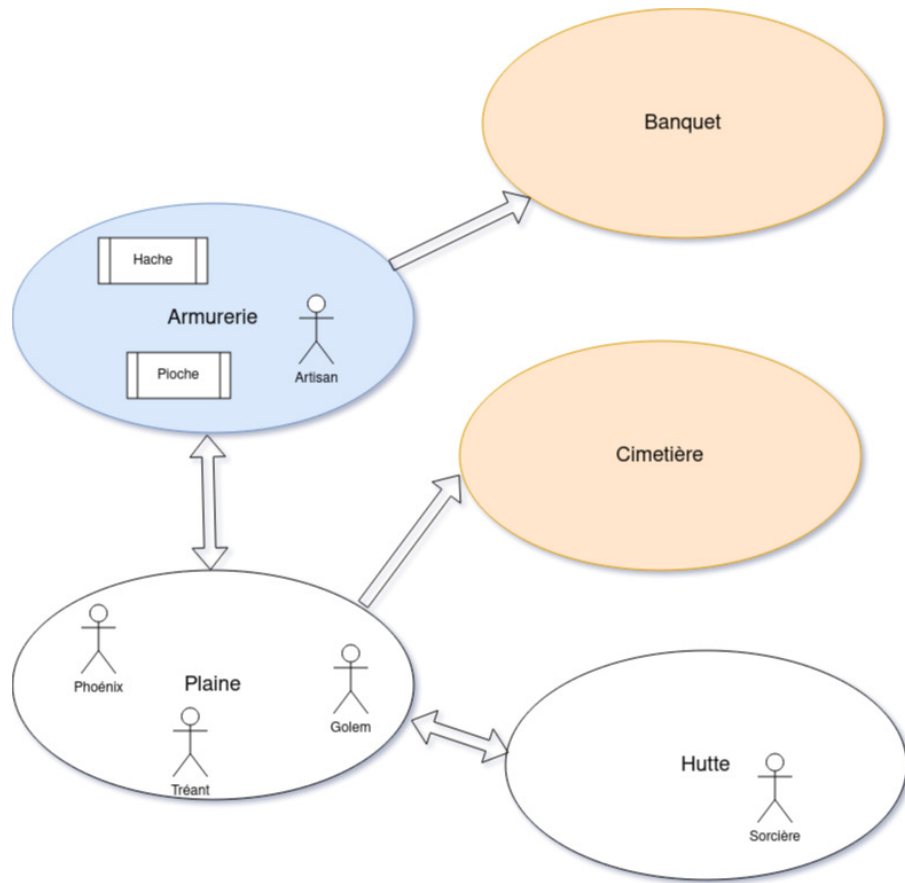


FIGURE 4 – Disposition des lieux, des personnes et des objets au sol de Mini RPG

3. une *hutte* où on peut parler avec une *sorcière*, qui donne la *localisation_phœnix* et un *antidote* si on lui donne une *plume*.
4. un *cimetière* (fin de jeu) dont le chemin est visible si le joueur est mort (combat à main nue contre tréant ou golem).
5. un *banquet* (fin de jeu) dont le chemin apparaît si le joueur a tué un *tréant* ou un *golem*, et il y meurt s'il n'a pas l'*antidote*.

Les transformations en Java des deux exemples (*MiniRPG* et *Enigme*) sont disponibles dans les livrables (respectivement *Mini_Rpg.java* et *GameEnigme.java*). Évidemment, les codes ne sont pas faits pour être lus par l'utilisateur. Le fichier généré pour notre *MiniRPG* fait plus de 6000 lignes !

7 Conclusion

Pour conclure, ce projet fut intéressant de part sa ludicité. En effet, modéliser un jeu est plus compréhensible et plus ludique que ce qu'on a eu l'occasion de modéliser au cours des TPs et du miniprojet. Cependant, développer les transformations sur Eclipse entraîne un redémarrage de l'Eclipse de déploiement à chaque modification du fichier, ce qui ralentit fortement le travail. De plus, pour l'*ATL*, Eclipse ne semblait pas connaître le modèle ce qui nous a posé de nombreux problèmes.

Les livrables rendus sont quasiment tous fonctionnels, nous y avons ajouté notre exemple ainsi que celui du sujet transformés en *Java*, *LTL* ou *PetriNet*.

Ainsi, le modèle du jeu est fait avec *Xtext* et nous y associons les contraintes OCL pour générer le meta-modèle.

La génération de code en *Java* fonctionne correctement : à partir d'un jeu, vous pouvez le transformer en un fichier *Java* et le tester.

Il est aussi possible de transformer un jeu en *PetriNet*. En revanche, cette partie a été très compliquée à réaliser, nous avons passé énormément de temps sur l'*ATL* pour essayer d'obtenir un *PetriNet* correct...