Projet d'Ingénierie Dirigée par les Modèles : Modélisation, Vérification et Génération de Jeux

ARNOLD Cyprien, BARROSO Pierre, DUONG Tom, HABRE Joe

Département Sciences du Numérique - 2ème année
2021-2022

Table des matières

Table des figures

1 Introduction

Le but du projet était de produire une chaîne de modélisation, de vérification et de génération de code pour des jeux de parcours/découverte. La modélisation consiste à concevoir un langage dédié pour décrire le jeu sous la forme d'un modèle et à implanter les outils d'édition, vérification et génération associés. La vérification, elle, permet d'assurer qu'il existe une solution pour le jeu décrit par un modèle (c'est-à-dire traduire un modèle de jeu en un réseau de Petri). Et enfin, la génération de code permet de construire un prototype avec une interface texte simple permettant de tester le jeu décrit par un modèle et de valider la jouabilité et l'intérêt du jeu avant de développer le contenu multimédia.

Nous décrirons dans la suite de ce rapport comment nous avons abordé ces différents thèmes et comment nous les avons implémentés.

2 Syntaxe concrète du langage de modélisation du jeu

Nous avions pour mot d'ordre de créer une syntaxe simple d'utilisation et compréhensible. Ainsi, pour la rendre la plus lisible possible, nous avons utilisé dès que la structure du métamodèle le permettait, des références. Un élément du jeu, une fois créé, est accessible via son ID, et peut donc être réutilisé autant de fois que nécessaire par les autres éléments. Cela évite de devoir redéfinir un élément à chaque utilisation, et donc évite les nombreuses accolades imbriquées, très souvent causes des syntaxes complexes.

La création d'un jeu se fait donc sous forme "d'arbre" : D'abord le *Jeu* lui-même, puis le *Territoire* (la carte du Monde), l'*Explorateur* (le joueur), et enfin, tous les autres éléments du jeu, qui une fois définis, peuvent s'inter-référencer afin de créer le jeu complet de manière lisible. Ci-après la définition des éléments du jeu énigme.

```
bonne Choix R1 {
    reponse "l'homme"
    connaisDon reussite
    interaction oedipe
    objetCons QteObjet uneTentative {
        qte 1
        objet tentative
    }
},
Choix R2 {
    reponse "Cyprien éméché"
    interaction oedipe
    objetCons QteObjet uneTentative {
        qte 1
        objet tentative
    }
},
Connaissance reussite {
    choix R1
},
QteObjet uneTentative {
    qte 1
    objet tentative
},
obligatoire Personne sphinx {
    place enigme
    interactions oedipe
    visible ConditionPersonne visibleSphinx {
        connaissanceInterdite reussite
        objetRequis uneTentative
    }
},
Interaction oedipe {
    question "Qu'est ce qui a 4 jambes, puis 2 jambes, puis 3 jambes?"
    personne sphinx
    choix R1, R2
}
```

FIGURE 1 – Illustration de notre grammaire

3 Génération du métamodèle Ecore

Nous avons dans un premier temps traduit toutes les exigences fournies sur les jeux sous forme d'un métamodèle, que nous avons ensuite fait évoluer à des fins de faciliter les différentes transformations à réaliser. Nous avons notamment souvent utilisé les références *eOpposite* entre les différentes classes du métamodèle car cela permet de coder des transformations sur un objet d'arité finie et d'éviter des boucles périlleuses. Nous avons finalement obtenu le métamodèle dont le diagramme de classe est présenté sur la figure ci-dessous.

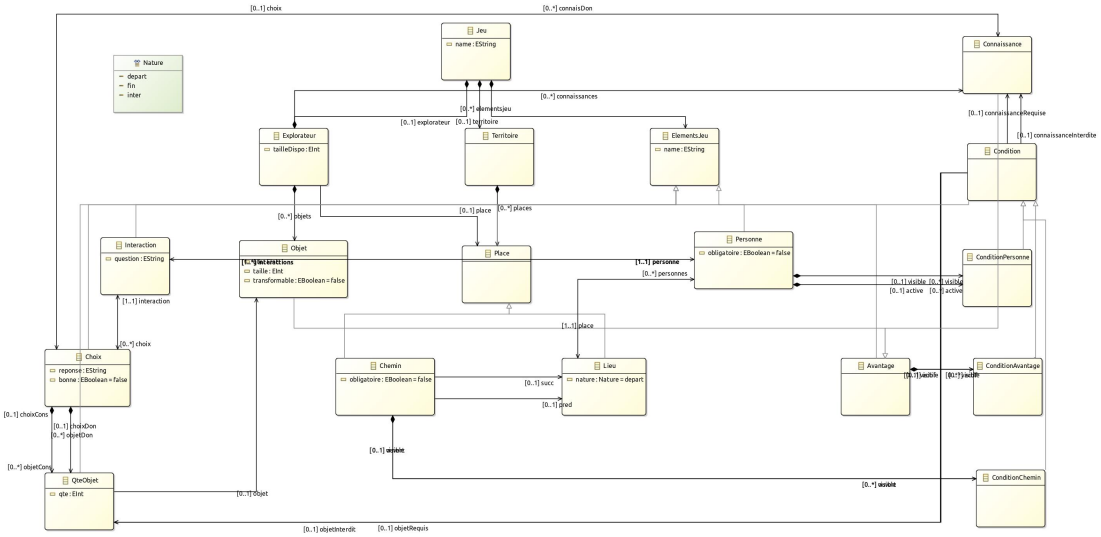

FIGURE 2 – Diagramme de classe de *Game*

4 Contraintes OCL

OCL est avant tout un langage de requête pour calculer une expression sur un modèle en s'appuyant sur sa syntaxe (son métamodèle). Le principe d'*OCL* est donc d'établir formellement les responsabilités d'une classe et de ses méthodes. Dans ce projet, nous définissons des contraintes au niveau des noms des *Personnes*, *Lieux*, *Chemins*, *Objets*, *Connaissances* ainsi que les *Conditions*. Ces noms ont nécessairement au moins deux caractères, commençant soit par une majuscule soit par une minuscule, selon la classe. De plus, deux *Personnes* ne peuvent pas avoir le même nom, deux *Lieux* et deux *Chemins* non plus.

Outre les noms, la taille disponible dans le 'sac' de l'*Explorateur* est toujours supérieure ou égale à 0. Concernant les *Lieux*, il n'y en a qu'un seul de type **Start**. Les *Chemins*, eux, ne peuvent pas être réflexifs. C'est un choix de conception qui nous semblait en adéquation avec les exigences du sujet : si on veut pouvoir parcourir un chemin dans les deux sens, il faut donc en créer deux différents. Et enfin, la taille d'un objet, tout comme la quantité d'objets possédés, sont toujours positives.

5 Transformation modèle à texte

5.1 Transformation vers Java

Nous avons choisi Java comme langage de programmation cible pour le prototype interactif en mode texte. Notre transformation permet de générer un code Java à partir du modèle respectant le métamodèle *Game*. Une fois exécuté, notre code Java lance un mini jeu interactif.
Voici un exemple d'exécution du prototype associé à l'exemple Enigme :

```
Veuillez entrer votre nom :
Cyprien

Cyprien, vous êtes à Enigme
Vous parlez avec Sphinx
[Question...]
0 - oui
1 - non
0

Cyprien, vous prenez le chemin obligatoire : Victoire
Vous pouvez :
0 - Afficher inventaire
1 - Avancer
Entrez le chiffre de votre choix :
0

Cyprien, vous possédez :
 - les objets suivants :
    Tentative x2
 - les connaissances suivantes :
    Reussite

Cyprien, vous êtes à Victoire
Vous pouvez :
0 - Afficher inventaire
1 - Avancer
Entrez le chiffre de votre choix :
```

FIGURE 3 – Exemple du prototype interactif avec le modèle Enigme du sujet

5.2 Transformation vers LTL

Cette partie consistait à écrire un code Acceleo qui, à partir d'un modèle xmi respectant le méta-modèle *Game*, génère un fichier LTL qui assure des propriétés que doit respecter le modèle une fois transformé en réseau de Petri. Nous avons défini 4 règles que le modèle doit respecter :

- Il faut que le réseau de pétri puisse se terminer (c'est à dire qu'aucune transition est activable) à un moment donné.
- Si le réseau de petri s'arrête, alors il doit y avoir un jeton dans une *Place* représentant un *Lieu* d'arrivée.
- Il n'est pas possible qu'il y ait plus de 1 jeton à la fois dans toutes les *Places* représentant un *Lieu* d'arrivée.
- Toutes les *Places* correspondantes à un chemin et se terminant par *init* ne peuvent avoir plus d'un jeton.

```
1 [] <> dead ; # Il y a toujours un arret du jeu
2
3 op finis  =  Succes \/ Echec \/  0 ;
4 [] (dead => finis); # Si on ne peut plus rien faire, on est dans un lieu final
5
6 [] ( 0 + Succes + Echec <= 1); # On ne peut pas etre à plus d'une place fin en meme temps
7
8 # Toutes les places correspondant au chemins et  par init sont <=1
9 [] (Victoire_first <=1);
10 [] (Defaite_first <=1);
```

FIGURE 4 – Exemple d'un fichier LTL pour l'exemple Enigme du sujet

5.3 Transformation vers Tina

Pour cette transformation, nous avons utilisé le code Acceleo du mini-projet d'octobre qui, à partir d'un modèle de Petri, génère un fichier .net.

6 Transformation texte à modèle

Sirius est un outil qui permet une transformation texte à modèle. Cet outil a pour objectif de présenter graphiquement une vue d'un modèle et ainsi pouvoir éditer graphiquement ce modèle, sous la forme d'un graphe composé de noeuds et d'arcs.

Comme on peut le voir sur la figure 5, nous avons plusieurs *Lieux* (Toulouse, Sucess ..) représentés par un rectangle orange. Ces lieux ont été directement ajoutés grâce à Sirius, en déployant la palette *Outils Principaux* en haut à droite et en appuyant sur *Créer Lieu*, qui nous créera un rectangle orange auquel on ajoutera le nom de ce *Lieu*. De plus, les *Personnes* sont représentées par des ellipses bleues (Joe, Pierre ...) déployées de la même façon que les *Lieux*. Ce même processus est répété pour tous les éléments du jeu.

En effet, tout le jeu peut être représenté graphiquement en déployant la palette correspondante, et en ajoutant à la main l'élément désiré. La seule différence entre ces éléments du jeu, sur une vue graphique, est la manière dont ils sont représentés (cercles, rectangles, flèches,..) de couleur différentes (rouge, orange, jaune,..). Néanmoins, cette différence est très importante et nécessaire puisque sur une vue graphique, la seule manière de différencier les éléments entre eux est d'associer à chaque élément une forme/couleur différente. Nous avons également réparti ces éléments en calques permettant à l'utilisateur de filtrer ce qu'il veut voir.

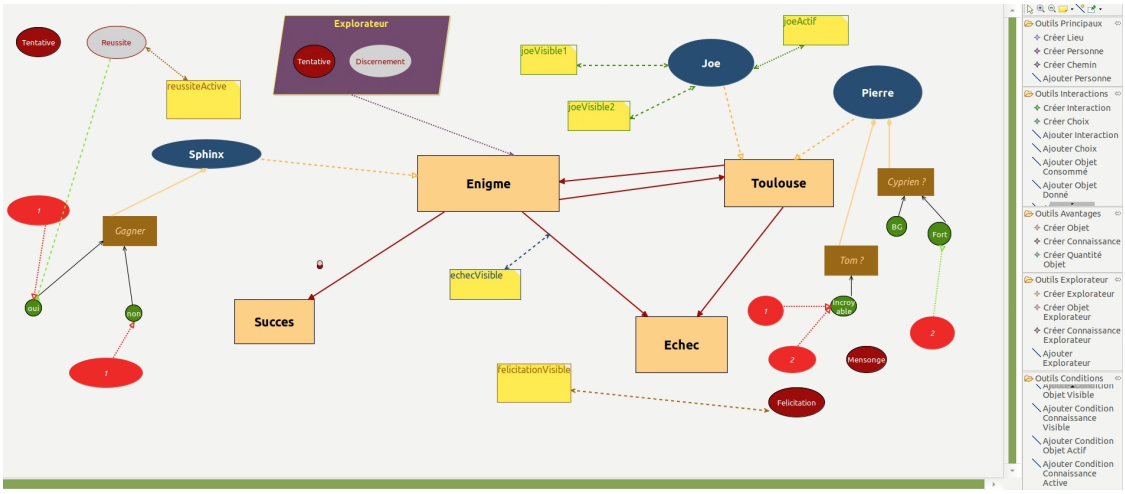

FIGURE 5 – Exemple d'un modèle représenté sur Sirius.

7 Transformation modèle à modèle

Afin de vérifier la qualité et la faisabilité des jeux produits selon notre métamodèle *Game*, nous avons du mettre en place une transformation de ce modèle vers celui des réseaux de *Petri*. Nous avons repris le métamodèle que nous avions défini lors du mini-projet et nous n'avons eu que de très légères modifications à y apporter. Pour cette transformation modèle à modèle nous avons utilisé *ATL*.

Nous avons tout d'abord créé à la main quelques petits réseaux de *Petri* correspondant à des mini-jeux afin d'essayer de remarquer des similitudes et donc en déduire des procédures de construction applicables sur n'importe quel modèle de jeu conforme à *Game*. Nous avions des réseaux simples mais à chaque nouveau jeu utilisant de nouvelles fonctionnalités autorisées par les exigences, nous devions adapter la création pour prendre en compte les nouveaux éléments. Nous nous sommes finalement retrouvés avec des réseaux de *Petri* bien plus complexes, même pour des petits jeux tels que Enigme proposé dans le sujet, mais dont la construction était automatique. Nous sommes alors passés à la phase de code réelle et nous avons du faire évoluer le métamodèle *Game* en ajoutant des relations *eOpposite* pour ne pas avoir à faire de boucles *for* car c'est très compliqué en ATL et fortement déprécié.

On peut observer sur la figure 6 qu'on a une place *Init* en début de jeu, reliée à la première transition activable *Start_JeuEnigme*. On arrive alors dans le premier **Lieu** de notre jeu, ici *Enigme_init*, d'où on aura accès à la première **Personne** obligatoire. Nous avons plusieurs places pour les **Lieux** (*Enigme_init*, *Enigme_end*), sauf si ce sont des lieux **finaux** (*Echec*, *Succes*).

Une **Personne** est représentée par plusieurs places et transitions : elle nous permettent de vérifier les conditions de visibilité et d'activité. Les **Conditions** peuvent s'appliquer à plusieurs types (Personne, Chemin, Avantage) et contenir jusqu'à 4 informations (Objet interdit, Objet requis, Connaissance interdite, Connaissance requise). Pour traduire certaines conditions sur le réseau de *Petri*, nous avons du introduire de nouveaux types d'arcs, permettant notamment d'activer une transition si la place pointée ne possède pas de jetons. C'est ce type d'arc qu'on utilise par exemple sur *init_defaite_visible* pour vérifier que l'explorateur n'a en effet plus d'objets *tentative* pour accéder au **Chemin** *Defaite*.

Les chemins sont remplacés par une transition reliée à la dernière place de son **Lieu** prédécesseur, et à la première de son successeur. On y rattache également les conditions de visibilité et d'ouverture, et une place *_first* pour s'assurer qu'on ne peut franchir qu'une fois la transition. Pour toutes les **Conditions**, comme il n'est pas nécessaire d'en poser, on y branche une place vide *_def* par défaut et un arc permettant d'activer la transition sans avoir de jetons. On ne l'initialise pas à 1 avec un arc classique car cette place peut être utilisée pour bloquer la transition si on y ajoute un jeton.

Les **Personnes** sont représentées par leurs **Interactions** et leurs **Conditions**. Les **Interactions** proposent plusieurs **Choix**, qui peuvent être des **bonnes** réponses ou non. En fonction de cet attribut on branche uniquement sur la place *_oblig* correspondant à une personne obligatoire, ou également au point de départ *_init* du **Lieu** s'il s'agit d'une mauvaise réponse. Les **Choix** peuvent consommer des **Objets**, en fournir d'autres, ainsi que des **Connaissances**. Pour les **Objets**, on utilise le type **QteObjet** et son attribut **qte** pour initialiser le poids de l'arc. Quant aux **Connaissances**, on prends le jetons de la place *_absente* et on l'ajoute à celle *_presente*.

Pour terminer le passage par un **Lieu**, il faut donc avoir obtenu les jetons dans toutes les places *_oblig* pour pouvoir déclencher la transition *finish_*. Les différentes **Personnes** et **Interactions** se branchent en parallèle entre les places de début et de fin des **Lieux** où ils se trouvent. Une **Personne** non obligatoire n'aura pas de place *_oblig* et on pourra donc terminer la traversée sans interagir avec elle.

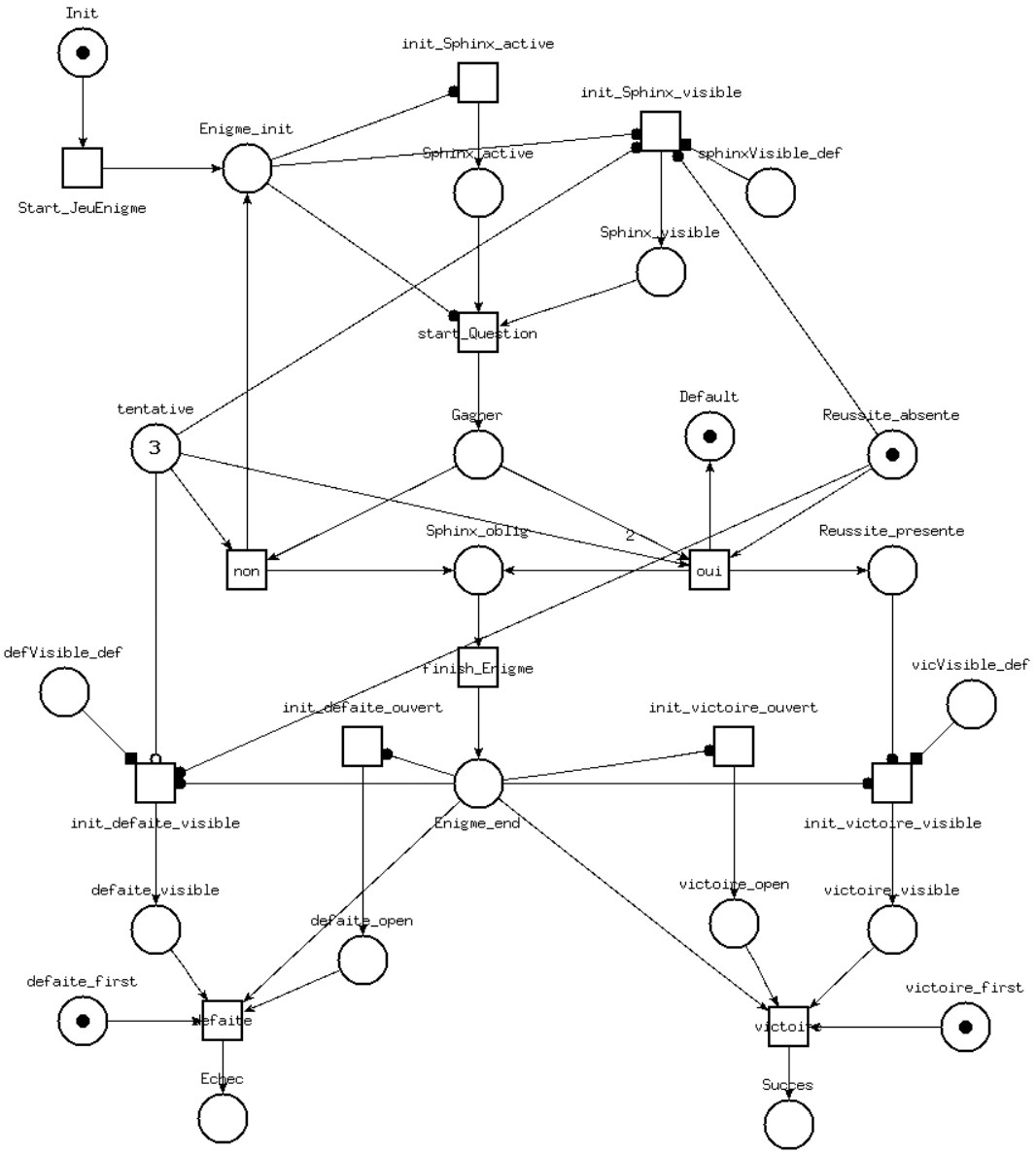

FIGURE 6 – Réseau de Petri généré pour le jeu Enigme

8 Conclusion

Ce projet nous a permis d'utiliser les concepts vus en cours et TP sur un exemple complexe. Ce sujet, même s'il demandait un certains nombre d'exigences, laissait une certaine liberté quant à la création de notre métamodèle de *Game*. C'est cet aspect qui nous a le plus posé problème car il fallait régulièrement modifier le métamodèle pour ne pas avoir des transformations trop compliquées à concevoir. Celle qui nous a posé le plus de problème était la transformation modèle à modèle, de *Game* à *Petri*. En effet, il était difficile de généraliser la création d'un réseau de Petri à tout modèle *Game* donné.

Par ces différentes réflexions, nous nous sommes aperçu de l'importance de la conception d'un bon méta-modèle puisque cela influençait toutes les transformations. De plus, nous aurions dû davantage nous focaliser sur la création du bon métamodèle dès le départ pour éviter d'avoir à revenir modifier les différentes transformations dès qu'une modification du modèle était effectuée. Cela nous a fait perdre un peu de temps dans le projet car il était difficile de bien se répartir les tâches et nous avons d'autant plus pu comprendre l'importance d'un gestionnaire de versions afin de se transmettre régulièrement les modifications effectuées et pouvoir revenir à des versions précédentes si besoin. Nous avons donc utilisé *Git* pour ce projet.