Université Paul Sabatier

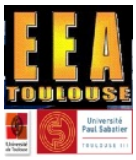
- M2 ISTR -

Modélisation, Analyse et Simulation de systèmes à événements discrets

Auteurs :
Johanne Bakalara
Claire Paya
Alexandre Armengaud
Lionel Mery
Lucien Rakotomalala
Ibrahim Atto
David Tocaven

Superviseurs :
Yann Labit
Julien Vanderstraeten

Client :
Sylvain Durola



18 avril 2018

Table des matières

Introduction

Ce projet se concentre sur la modélisation et l'analyse d'un système à événement discret composé d'un labyrinthe cartésien dynamique et de deux objets se déplaçant à l'intérieur. Certaines hypothèses de travail ont été établies, notamment le choix d'utiliser le formalisme de modélisation des automates finis, ainsi qu'une approche ensembliste et non stochastique.
Nous avons mis en œuvre notre projet sous Matlab car notre étude théorique nous a amené à une modélisation largement basée sur des matrices et sur des tableaux et que Matlab étant un langage de haut niveau que nous connaissons était le plus adapté à cette application. Nous avons choisi une conception orientée objet et avons implémenté une interface interactive et des outils d'analyse basés sur la modélisation du labyrinthe avec des automates déterministes et non déterministes.
Pour commencer, nous avons réalisé une interface permettant de jouer en mode automatique, semi-automatique ou manuelle dans laquelle on trouve des commandes pour le fonctionnement en mode auto des murs et objets.
Par la suite, nous avons fait des vérifications : une logicielle pour tester le coeur du code et une validation formelle pour analyser si les commandes produites précédemment pouvaient toujours amener un objet à la sortie. Pour se faire nous avons étudié le labyrinthe composé d'un unique objet.
Nous avons ensuite mis en place 3 scénarios dont le but n'était plus de vérifier les commandes mais d'en créer. Pour le scénario 1, l'objectif était d'atteindre la sortie ou d'aller dans un état bloquant. Pour le scénario 2, le but était de trouver la sortie sans connaître la position initiale de l'objet dans le labyrinthe. Pour le scénario 3 le but était de faire les mêmes vérifications et recherche de commande que pour un objet mais en ajoutant les contraintes amenées par la présence d'un deuxième objet.

Chapitre 1

Interface 2 joueurs

1.1 Les règles du jeu

Commençons par définir les règles du jeu. Il y a deux objets (pions) dans le labyrinthe. L'un est représenté par pacman et l'autre par un fantôme. Le but de pacman est d'atteindre la sortie sans se faire attrapé plus de quatre fois par le fantôme. Le fantôme doit attraper pacman avant qu'il n'atteigne la sortie. Attention, si l'un des deux se retrouve bloqué entre quatre murs, il a perdu !
La partie se déroule comme suit : Les murs bougent (les murs verticaux vers le haut ou les murs horizontaux vers la droite), pacman bouge, le fantôme bouge. Il est interdit de rester sur place.
Nous avons dû définir ce que chacun connaît du système. Le labyrinthe est omniscient, il connaît les positions de chacun et l'ordonnancement. Pacman connait la position de la sortie et les murs autour de lui. Le fantôme ne connaît pas la sortie, connaît les murs autour de lui et peut voir pacman dans son couloir lorsqu'il n'est caché par aucun mur (oui il est très fort..).

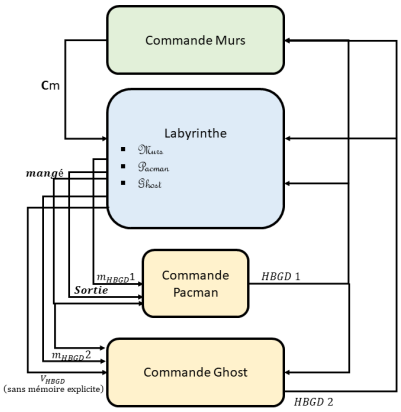


FIGURE 1.1 – Définition des ES connus pour chaque élément

1.2 L'interface

Nous nous trouvons actuellement dans le dossier src/laby2players
L'interface de base a été développée selon un modèle séquentiel et implémentée en langage objet avec Matlab. Voici son allure :

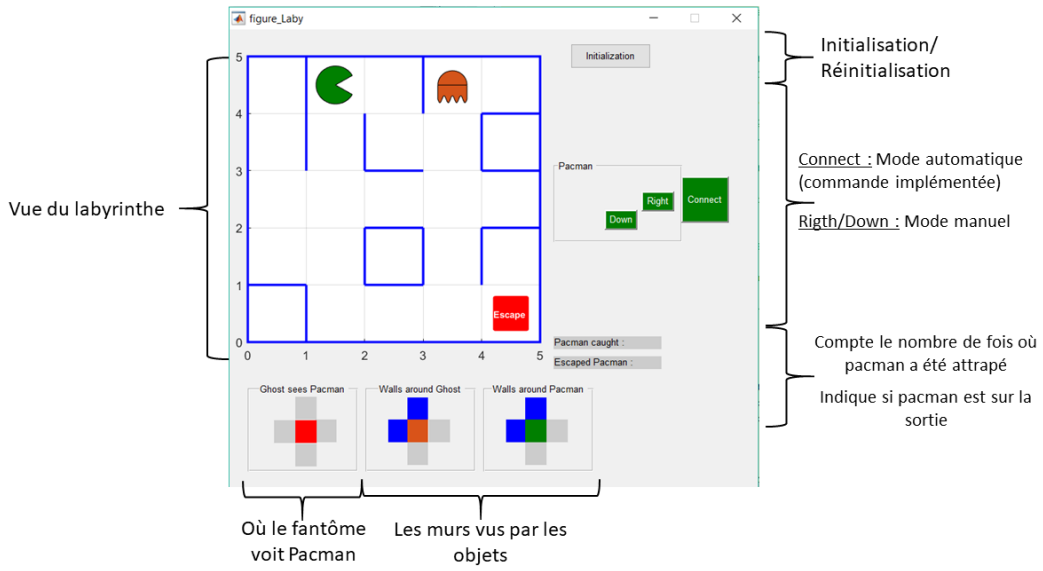


FIGURE 1.2 – L'interface

Sa création fût l'objet du premier bloc de projet.
L'interface a été réalisée grâce à la création de figure avec l'outil *GUI* proposé par Matlab. Nous nous sommes beaucoup référé à l'aide Matlab pour comprendre et découvrir toutes les fonctionnalités offerte par la *GUI*

Le code est basé sur une implémentation par bloc FMG où chaque vecteurs d'entrées déterminera ce qui est connu par l'objet. Ces vecteurs d'entrées seront utilisés par le bloc F pour actualiser l'état du modèle, c'est à dire soit l'état des murs ou la position des objets dans le labyrinthe pour le *Model Laby*, soit l'état de la commande pour les *Model Walls*, *Model Pacman* et *Model Ghost*.

Ensuite, chaque vecteur de sortie sera généré par le bloc G, en utilisant l'information sur l'état du système donnée par le bloc M.

FIGURE 1.3 – Bloc FMG

C'est *figure_laby* qui s'occupera de fera le lien entre l'utilisateur et les commandes automatiques. L'implémentation se déroule autour de plusieurs classes énumérée dans le diagramme de classe en figure 1.4

1.3 Description des modèles

Vous trouverez dans cette section la description des modèles utilisés dans l'interface.

1.3.1 ModelSED

Classe mère abstraite qui fait hérité les méthodes $f()$ $m()$ et $g()$ ainsi que les attributs *presentState* et *initialState* à toutes les classe filles qui sont :

- *Model Ghost* qui permet de commander le fantôme. Vous trouverez la documentation des commandes implémenté dans 2.2
- *Model Pacman* qui permet de commander le Pacman. Vous trouverez la documentation des commandes implémenté dans 2.1

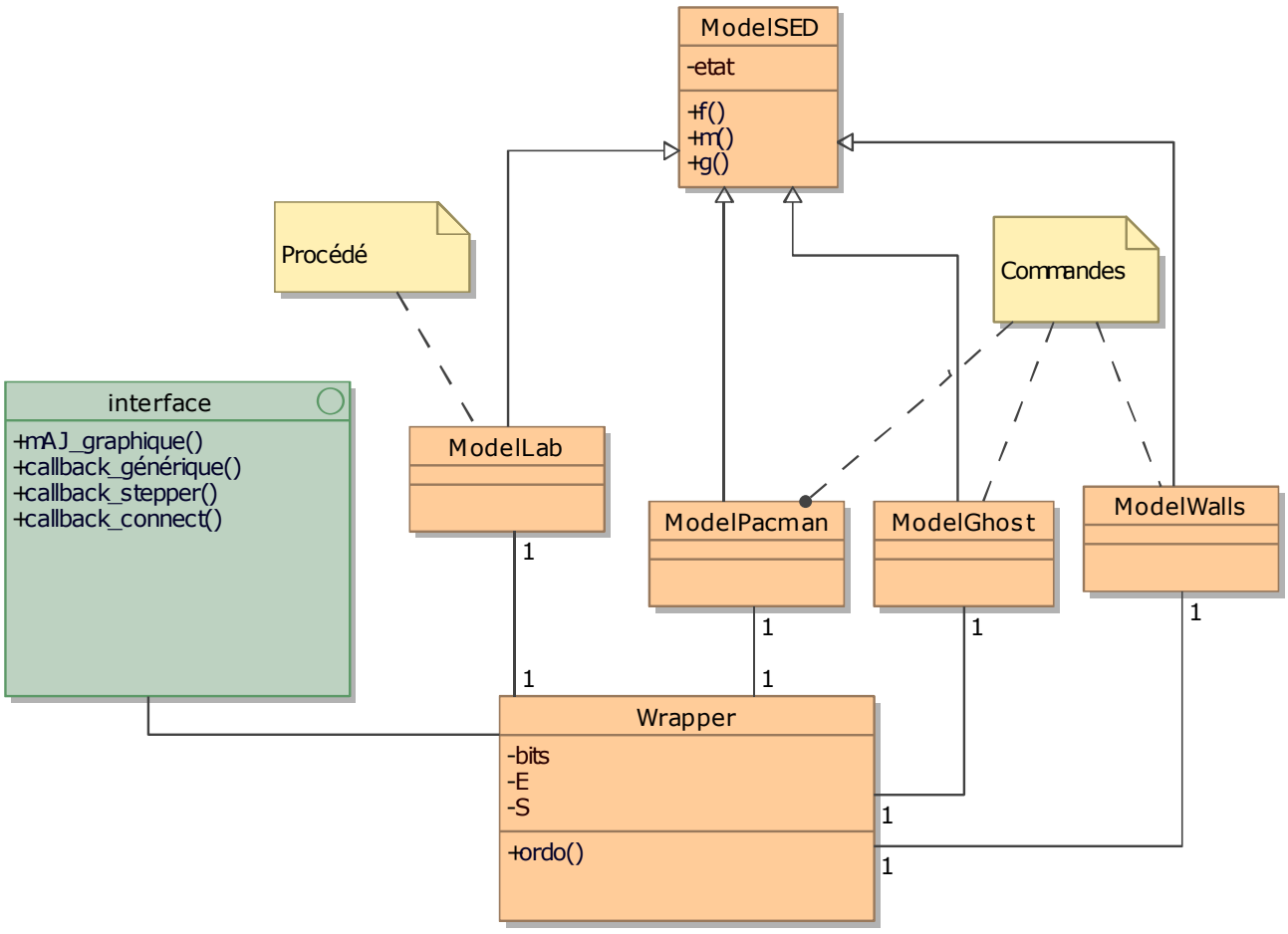


FIGURE 1.4 – Modèle UML du GUI

- *Model Laby* décrit dans cette section.
- *Model Walls* qui permet de commander le Walls. Vous trouverez la documentation des commandes implémenté dans 2.3
- *Stop Condition* décrit aussi dans cette section.

1.3.2 ModelLaby

Classe héritant de *ModelSED* conntenue dans le fichier : /src/laby1player/ModelLaby.m et /src/laby2player/ModelLaby.m.

Classe héritant de ModelSED, on calcule les positions des murs et des objets entre chaque tour ici. Elle connaît également la sortie et le nombre de fois où pacman est attrapé. Elle est volontairement extrêmement condensée.

Bloc F L'évolution de l'état du labyrinthe est effectué dans cette méthode : nous y calculons les matrices de murs et la position des objets en fonction des entrées du labyrinthe. Par exemple, si l'entrée *Murs descendent* est activé, alors la fonction va renvoyer l'actualisation des matrices de murs.

Le format des entrées admises par cette fonction est dévellopé dans un vecteur de booléen. Il est défini en commentaire du code source de l'interface, dans le fichier *figure_laby.m*, dans le dossier *laby1player* ou *laby2player*, dans la fonction *function ui_Callback(hObject, eventdata, handles)*.

Bloc M Nous actualisons l'état présent du modèle en fonction de celui calculé dans la fonction f. Pour cela, vous devez le mettre dans les inputs de la fonction. La méthode attend aussi un booléen pour lui indiquer si le modèle doit être remis à l'état initial.

Il est à noter que le calcul du compteur *caught*, c'est à dire le compteur du nombre de fois qu'a été attrapé le Pacman 1.1, est aussi effectué dans cette méthode (dans la version 2 joueurs). Pour cela, nous faisons appel aux positions des deux objets et à la connaissance des murs, pour inspecter qu'il n'y a pas de murs entre les deux objets.

Bloc G La fonction de génération des sorties est implémentée dans la méthode portant ce nom. Nous allons dans la liste ci dessous vous détailler l'ensemble des sorties de ce modèle et expliquer la méthode de calcul utilisée.

- Murs autour d'un objet : Permet d'identifier quel(s) mur(s) (Haut, Bas, Gauche et/ou Droite) sont autour d'un objet. Ces sorties sont représentées sur un vecteur identique pour tous les objets, décrit dans le constructeur de la classe "Wrapper", qui est mis à VRAI si un mur se trouve dans la direction et 0 sinon.
- Pacman est sur la sortie : En fonction de la position de l'objet PACMAN et de la position de la sortie ESCAPE, détermine si l'objet est sorti ou non du labyrinthe.
- Le ghost voit le Pacman : (Uniquement dans la version 2 joueurs) Si les positions des objets Pacman et Ghost ont le même x ou le même y, et qu'aucun mur ne se trouve entre les deux, alors le ghost peut voir le pacman. Comme pour le vecteur "Murs autour d'un objet", cette sortie est représentée sur un vecteur où chaque élément indique une direction et est mis à 1 quand la vue du pacman par le ghost est VRAI, 0 sinon.

1.3.3 StopCondition

Classe héritant de *ModelSED* conntenue dans le fichier : /src/laby1player/StopCondition.m et /src/laby2player/StopCondition.m.

Cette classe permet d'établir l'arrêt du labyrinthe. Pour cela, elle admet en entrée les 2 (ou 4 pour labyrinthe à 2 joueurs) paramètres qui permettent d'arrêter le modèle du labyrinthe :

- *noEscape* variable booléenne qui indique si le Pacman se trouve sur la sortie.
- *pacmanWallsBreak* vecteur contenant les murs autour du Pacman.
- *ghostWallsBreak* (pour labyrinthe à 2 joueurs) vecteur contenant les murs autour du Ghost.
- *caught* (pour labyrinthe à 2 joueurs) entier représentatn le nombre de fois que le Pacman a été capturé par le Ghost.

Elle contient aussi une évolution sous forme de bloc "FMG" décrite précédemment.

1.4 Description du "Wrapper"

Nous allons dans cette partie détaillé les fichiers *wrapper.m*. Vous les trouverez les sources dans : *src/laby1player* et *src/laby2player*.

Cette section est entièrement dédié aux explications sur la classe *Wrapper*. Vous y trouverez dans un premier temps une description des attributs contenu dans la classe puis nous détaillerons dans 3 sous-sections les méthodes importantes.

1.4.1 Attributs du Wrapper

La description des attributs de cette classe se situe en début du code source de la classe. Nous avons placé dans cette liste d'attributs l'ensemble des états des modèles connectés par le "Wrapper" : *ModelLaby*, *commandWalls*, *commandPacman*, *stopCondition* et *commandGhost* (pour le labyrinthe 2 joueurs). Nous y avons aussi placé une variable servant de compteur : *whoPlay*. Elle permettra de connaître, selon l'ordonnancement, quel modèle doit être appelé.

Cette classe contient également 2 (ou 3 pour 2 joueurs) attributs booléen pour la connection des commandes automatique. Ces variables ont pour fonctions d'activer le mode de fonctionnement automatique du labyrinthe. Elles sont mise à jour dans la fonction *updateConnexion* que nous décrivons en détail dans 1.4.2 et sont utilisées dans la fonction *orderer* décrite en 1.4.4.

Enfin, la classe possède deux attributs permettant le bon fonctionnement des connexions : il s'agit de *in* et *out*. Ce vecteur et cette cellule (respectivement) ont le rôle de médium entre les modèles : la cellule *out* va contenir les sorties du labyrinthe (i.e les sorties de la classe *ModelLaby*) pour les envoyer en entrées aux modèles de commandes. Le vecteur *in* va quand à lui contenir les sorties des commandes pour qu'elles puissent être envoyé au labyrinthe. Ces 2 derniers attributs sont sous utilisé dans la méthode décrite en 1.4.4.

Les initialisations de ces attributs sont effectuées dans le constructeur de la classe. Nous attirons votre attention sur l'instanciation des attributs contenant les instances des modèles : ceux-ci ont besoin de recevoir leur état initial. Celui ci sera stocké dans chaque modèle et ne pourra être changer au cours de l'éxécution du programme.

Remarque Les initialisations des paramètres du labyrinthe peuvent être modifiés lors de la construction de l'objet *wrapper*, dans la fonction *figure_ Laby_ OpeningFcn* appartenant à l'interface Graphique. Elle est donc contenu dans le fichier *figure_ laby.m*, situé dans le même dossier que le fichier qui vous présenté en ce moment.

1.4.2 Mise à jour des connexions

Cette méthode s'appelle *updateConnexion* et a pour fonction d'actualiser les 2 (ou 3) booléens de connexion. Pour donner un exemple de son fonctionnement, quand un utilisateur clique sur le bouton connexion depuis l'interface présenté en **??**, l'interface appelle cette fonction pour mettre à jour le bit de connexion.

1.4.3 Initialisation du labyrinthe

Nous avons rapidement été confronté au problèmes de ré-initialisation de notre labyrinthe : chaque modèle devait rappeler son état initial en même temps. Nous avons donc choisi de créer cette méthode qui permet d'appeler tout les méthodes m des modèles SED et de lancer l'initialisation de chaque modèle. Pour rappel, les méthodes m permettent de mémoriser l'état du labyrinthe comme le montre le schéma en **??**.

1.4.4 Ordonnancement des modèles

Pour permettre le bon ordonnancement d'appel des modèles, nous avons développé une méthode nommé *orderer*. C'est dans cette fonction que nous avons implémenté les tours de jeux : d'abord les murs puis le Pacman (et le Ghost pour le labyrinthe à 2 joueurs) avec entre chaque une mise a jour du labyrinthe. Cet ordonnancement est géré grâce à l'attribut *whoPlay* que nous vous présentions en début de section : il prend la valeur 0 ou 1 (ou 2 dans le cas 2 joueurs) pour connaître quel modèle doit être appelé : murs ou Pacman (ou Ghost) respectivement.

La méthode gère également le mode manuel/automatique/semi-manuel/ à l'aide de ces attributs *objetbit* : si ce bit de connexion n'est pas à 1, alors la méthode ne fait pas appel au modèle correspondant. Dans ce cas, le vecteur *in* qui est envoyé au labyrinthe est celui construit par l'interface. Sinon, le modèle de l'objet connecté est instancié : appel des méthodes f, m puis g de celui-ci.

Remarque Les entrées des modèles des murs et des objets est contenu dans la cellule *out*. il est donc possible de modifier les entrées possibles de chaque modèle au niveau des appels des méthodes f de chaque modèle.

Chapitre 2

Commandes

Dans un premier temps nous avons créé des commandes intuitives que nous avons implémentées dans les classes dédiées ($ModelPacman$, $ModelGhost$ et $ModelWalls$). Les commandes intuitives sont détaillées ci-après. Dans la suite du projet nous en avons créé de nouvelles plus performantes, qui remplaceront celles-ci.

2.1 Commandes de pacman

2.1.1 Avec priorité

Cette commande vérifie si elle peut aller à droite (pas de mur) sinon en bas, sinon à gauche, sinon en haut :

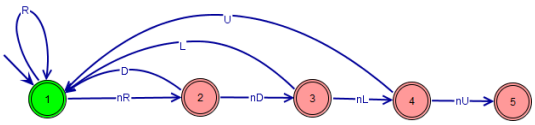


FIGURE 2.1 – Commande simple de Pacman

2.1.2 Avec mémoire

Cette commande garde en mémoire un vecteur à 4 entrées. L'entrée 1 passe à 1 si on a fait droite, l'entrée 2 à 1 si on a fait bas, l'entrée 3 à 1 si on a fait gauche, l'entrée 4 à 1 si on a fait haut. Une fois qu'on a testé les quatre possibilités, le vecteur est remis à 0.
Sa représentation complète sous forme automate est très conséquente (environ 120 états), vous la trouverez dans le code dans le dossier $laby\_1\_player\_automaton$, en voici une partie pour donner une idée :

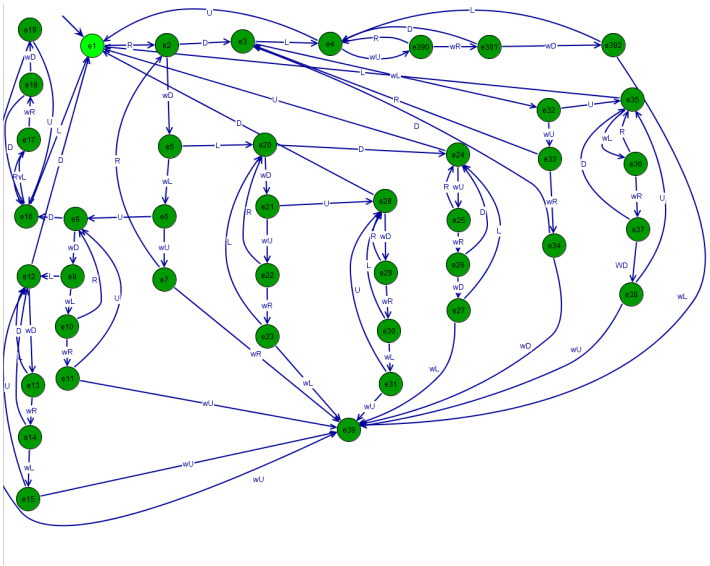


FIGURE 2.2 – Commande Memory de Pacman

2.2 Commandes de ghost

2.2.1 Ghost voit Pacman

Avec cette commande si ghost voit pacman dans son couloir (aucune séparation par des murs) il va dans sa direction. S'il ne le voit pas il fonctionne avec une priorité (haut, bas, gauche, droite).

2.2.2 Ghost ne reste pas bloqué

Cette commande respecte les mêmes conditions que précédemment mais prend aussi en compte qu'il ne peut pas aller sur une case dans laquelle il pourra se retrouver bloqué au prochain tour. S'il ne voit pas ghost ou que si là ou il le voit il peut se retrouver bloqué, il suit la même règle de priorité que précédemment, toujours en évitant la case bloquante.

Ces commandes n'ont pas été modélisées, car il faut les adapter au procédé complet avec deux objets.

2.3 Commandes des murs

Les murs verticaux peuvent seulement se déplacer vers le bas et les murs horizontaux vers la droite.

2.3.1 Alternée

Les murs horizontaux se déplacent à leur tour, puis les murs verticaux au tour d'après. Ils alternent donc à chaque tour de jeu.

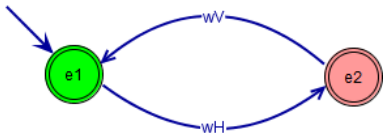


FIGURE 2.3 – Commande des murs alternée

2.3.2 Une fois sur deux

Les murs ne se déplacent qu'une fois sur deux et de façon alterné comme précédemment. Attention cette commande n'est pas compatible avec la commande de "ghost ne reste pas bloqué". En effet, ghost n'a pas accès aux informations du labyrinthe, il connait juste la situation initiale et leur commande, il calcule donc la prochaine position des murs (il ne la lit pas !). Cette commande de ghost a été conçue pour fonctionner avec la première commande de mur.

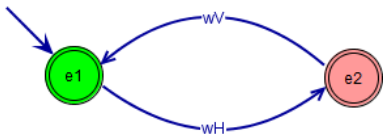


FIGURE 2.4 – Commande des murs une fois sur deux

Chapitre 3

Vérifications et validations

3.1 Programme de simulation et représentation graphique

Afin de pouvoir tester l'exécution du procédé et des différentes commandes (du Pacman, du fantôme et des murs), nous avons réalisé un script (*Simulation.m*) qui exécute le procédé commandé sans interface graphique. Il lance l'évolution jusqu'à ce que l'on remplisse une condition d'arrêt (le Pacman est sorti, un objet est bloqué ou le fantôme a attrapé le Pacman). Ensuite, il affiche un compte rendu de l'exécution dans le terminal (nombre d'évolutions et raison de l'arrêt). Puis, il sauvegarde le flux de sorties de toutes les itérations dans un fichier *state.mat* et appelle une fonction qui génère une vidéo et des images afin d'avoir une représentation graphique de l'évolution du labyrinthe. Le tout est stocké dans un dossier nommé en fonction de la date. La fonction de génération de rendu graphique existe en 2 versions. La première est une version basique *CreatePituresAndVideo.m* (figure 3.1). La seconde nécessite un lot de textures (qui peuvent être modifiable) *CreatePituresAndVideo_textured.m* (figure 3.2).

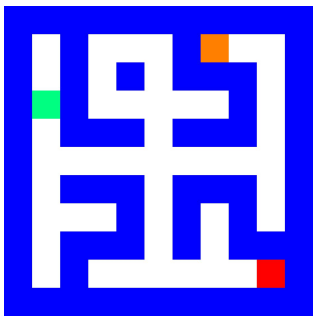


FIGURE 3.1 – Image de la première version.

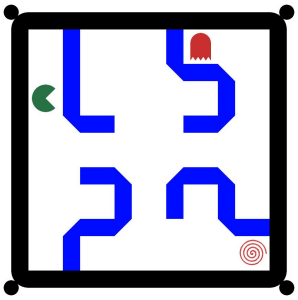


FIGURE 3.2 – Image de la seconde version.

Il est notable que la seconde version donne un meilleur résultat visuel (c'est celle que nous détaillerons ci-après) mais c'est la première version que nous avons utilisée pour les premiers tests logicielles desquels nous avons issus quelques bugs (voir partie Vérification Logicielle).

3.1.1 Script de Visualisation

Au début de ce script, dans la partie *PARAMETERS*, on a fixé le nombre maximal d'itérations à 100. Il peut être modifié en fonction du besoin. Celui-ci permet de brider la longueur de la simulation pour éviter d'éventuel cas de boucle infinie ou de grande simulation qui impliquerait un gros stockage de données. Ensuite, toujours dans la même partie, sont fixées les conditions initiale du labyrinthe (position initiale des objets, de la sortie et des murs) et des commandes. La seconde partie, *MAIN SCRIPT* commence par la création d'une instance de la classe *Wrapper* et la connexion de toutes les commandes. Elle contient également une boucle dans laquelle est lancée la simulation jusqu'à ce qu'une des conditions d'arrêt soient satisfaites ou que nous aillons atteint le nombre maximum d'itérations. La fin de cette partie permet de générer le rapport dans le terminal. La dernière partie est l'appel de la fonction de génération de rendu image et vidéo.

3.1.2 Fonction de génération d'images et de vidéo

Comme vu précédemment, la fonction *CreatePituresAndVideo_textured* utilise un ensemble d'images comme texture et les positionnements de façon à créer une image fidèle de l'état du labyrinthe. Pour cela, nous avons créé 12 images présentées sur la figure 3.3. Les images correspondent à :

0 Une case vide.

1 Une case occupée par Pacman.

2 Une case occupée par Ghost.

3 Une case où est la sortie.

4 Une case où Pacman est sur la sortie.

5 à 8 et 23 à 26 Les bordures de l'image.

9 à 12 Les coins des bordures.

13 et 15 L'absence de murs.

14 et 16 Les murs.

17 Les croisements de murs vides.

18 Les croisements de murs pleins.

19 à 22 Les croisements de murs obliques

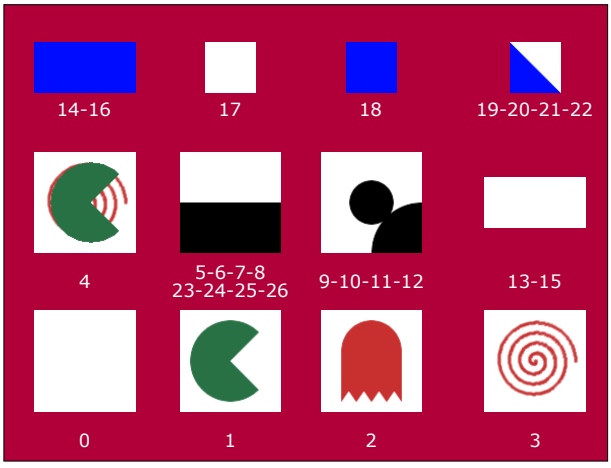


FIGURE 3.3 – Pack de texture utilisée

9	6	2 3	6	2 3	6	10
5	a	b				7
26	c	d				24
5						7
26						24
5						7
11	8	2 5	8	2 5	8	12

FIGURE 3.4 – Exemple de placement d'indice pour un labyrinthe 3x3.

Celles-ci sont organisées grâce à un ensemble d'indice, comme par exemple un labyrinthe 3x3, figure 3.4, où les indices en bleu sont statiques à tous les états du labyrinthe et où les indices en rouge, qui sont présents sur toutes cases similaires, sont susceptibles de changer.

a Les cases (indices 1 à 4).

b Les murs verticaux (indices 16 et 17).

c Les murs horizontaux (indices 15 et 16).

d Les croisements (indices 17 à 22).

Dans une premier partie, nous plaçons les indices correspondant aux éléments a afficher dans une matrice et dans une seconde partie nous plaçons les images correspondantes dans un lot de trois matrices (pour les trois canaux de couleur RGB). Une troisième partie enregistre les matrices sous forme d'images et génère la vidéo.

3.2 Vérification Logicielle

Les vérifications que nous avons effectuer son fait a partir du master daté du 30/01/2018. Si vous teste certaine validation maintenant elle ne fonctionnera plus vraiment en vu des modifications effectuer au cours du projet.

3.2.0.1 La case de sortie ne bouge pas

Nous allons vérifier que la case de notre sortie du labyrinthe ne change pas de position pendant l'utilisation. Pour cela nous devons faire une analyse structurelle de notre code et une analyse de tous les cas possibles de notre case sortie. Pour faire l'analyse structurelle de notre code nous avons cherché tous les endroits du code où l'emplacement de la case sortie change. Notre analyse nous a donnée le document suivant :

La case sortie ne bouge pas		
	Escape modifiée	**Escape non modifiée**
figure_Laby.m	X	Ajoutée dans le handle
Wrapper	Lue	X
ModelWalls		X
ModelSED		X
ModelPacman		X
ModelGhost		X
ModelCommand		X
ModelLaby	Définie une fois dans l'initialisation	

Conclusion	On a validé que la case sortie de bougeait pas avec une analyse du code classe par classe. On ne modifie escape quand dans notre initialisation dans la classe ModelLaby

FIGURE 3.5 – Table des recherches de modification de l'emplacement de la case

Nous avons aussi testé tous les cas possibles d'emplacement de notre sortie sur un labyrinthe 5x5 sans changé les murs. De ce test nous n'avons pas observé de problème.

3.2.0.2 Fantôme et Pacman jamais sur la même case

Pour tester que le fantôme et le Pacman ne sont jamais sur la même case nous avons fait un test "de pire cas possible" (on met des situations où on pense pourvoir faire apparaître ce bug s'il existe) sur différents labyrinthes. Pour cela nous avons placé plusieurs conditions initiales différentes, les tests sont faits avec un nombre de murs différents et plusieurs positions du fantôme et de Pacman différente. Pour nous retrouver dans les différentes situations nous avons fait le fichier excel suivant :

Ghost/Pacman jamais sur la même case

	Matrice de murs	Matrice de verticaux	Situation initiale Pacman	Situation initiale Ghost	Resultat du test
Test 1	V= [0010; 0000;0001;1001;0001]	H= [00100;00100;01000;10000]	[3,1]	[4,1]	ghost jamais sur la meme case que pacman
Test 2	V= [0010; 0000;0001;1001;0001]	H= [00100;00100;01000;10000]	[1,5]	[5,1]	ghost jamais sur la meme case que pacman
Test 3	V= [0010; 0000;0001;1001;0001]	H= [00100;00100;01000;10000]	[4,2]	[2,2]	ghost jamais sur la meme case que pacman
Test 4	V= [0010; 0000;0001;1001;0001]	H= [00100;00100;01000;10000]	[3,3]	[3,4]	ghost jamais sur la meme case que pacman
Test 5	V= [0000; 0000;0000;0000;0000]	H= [00000;00000;00000;00000]	[3,1]	[4,1]	ghost jamais sur la meme case que pacman
Test 6	V= [0000; 0000;0000;0000;0000]	H= [00000;00000;00000;00000]	[1,5]	[5,1]	ghost jamais sur la meme case que pacman
Test 7	V= [0000; 0000;0000;0000;0000]	H= [00000;00000;00000;00000]	[4,2]	[2,2]	ghost jamais sur la meme case que pacman
Test 8	V= [0000; 0000;0000;0000;0000]	H= [00000;00000;00000;00000]	[3,3]	[3,4]	ghost jamais sur la meme case que pacman
Test 9	V= [1010; 1101;0000;0111;1000]	H= [00010;01010;01010;01001]	[3,1]	[4,1]	ghost jamais sur la meme case que pacman
Test 10	V= [1010; 1101;0000;0111;1000]	H= [00010;01010;01010;01001]	[1,5]	[5,1]	ghost jamais sur la meme case que pacman
Test 11	V= [1010; 1101;0000;0111;1000]	H= [00010;01010;01010;01001]	[4,2]	[2,2]	ghost jamais sur la meme case que pacman
Test 12	V= [1010; 1101;0000;0111;1000]	H= [00010;01010;01010;01001]	[3,3]	[3,4]	ghost jamais sur la meme case que pacman
Test 13	V=[1011; 1011;1000;1101;1010]	H=[10111;10100;10100;10011]	[3,1]	[4,1]	ghost jamais sur la meme case que pacman
Test 14	V=[1011; 1011;1000;1101;1010]	H=[10111;10100;10100;10011]	[1,5]	[5,1]	ghost jamais sur la meme case que pacman
Test 15	V=[1011; 1011;1000;1101;1010]	H=[10111;10100;10100;10011]	[4,2]	[2,2]	ghost jamais sur la meme case que pacman
Test 16	V=[1011; 1011;1000;1101;1010]	H=[10111;10100;10100;10011]	[3,3]	[3,4]	ghost jamais sur la meme case que pacman
Test 17	V= [1010; 1101;0000;0111;1000]	H= [00010;01010;01010;01001]	[3 3]	[3,3]	ghost et pacman sur la meme case mais aucun mouvement possible

FIGURE 3.6 – Excel de la vérification de Ghost et Pacman jamais sur la même case

Des situations initiales suivantes nous avons lancé la première fonction de simulation qui permet de récupérer l'état du labyrinthe du début jusqu'à la fin de la simulation. Nous avons fait un oracle (c'est un programme qui va tourner en parallèle et qui nous permet de vérifier que les résultats obtenus par le code principale et le même que celui que l'on obtient par ce programme). Notre oracle va vérifier à chaque mouvement que la position de notre fantôme et du Pacman sont bien différentes.

Nous avons trouvé un bug lors de cette situation initiale : si les deux objets sont sur la même case à l'état initial, il n'y a aucun mouvement possible. Il faudrait vérifier les situations initiales dès le départ pour éviter ce cas qui devrait être impossible à arriver.

Les différents oracles que nous avons fait se trouvent dans le dossier validation2 qui se trouve dans le répertoire laby2player/validation.

3.2.0.3 Les murs sont infranchissables

Pour ce test nous nous somme placés dans deux situation différentes, l'une avec les murs qui ne bougent pas et l'autre avec les murs qui bougent. On a des situations initiale de murs différentes. Les différentes situations initiales sont récapitulées dans le fichier excel suivant :

Les murs sont infranchissables

Les murs ne bougent pas

	Matrice de murs	Matrice de verticaux	Situation initiale Pacman	Situation initiale Ghost	escape
Test 1	V= [1000; 1001;1111;1001;0000]	H= [01110;00100;00100;01110]	[2,3]	[5,1]	[4 4]
Test 2	V= [1000; 1001;1001;1001;0001]	H= [11111;00000;00000;01110]	[2,1]	[1,1]	
Test 3	V= [0010; 0000;0001;1001;0001]	H= [1 11 00;11000 ;10011;00111]	[2,1]	[4,1]	
Test 4	V= [1000; 1001;1111;1001;0000]	H= [01110;00100;00100;01110]	[4,4]	[1,4]	[3 3]
Les mus bougents					
Test 5	V= [0010; 0110; 0000;0000;0010]	H= [11000;01000;00000;00000]	[1,1]	[5,5]	
Test 6	V= [0010; 0110; 0000;0000;0010]	H= [11000;01000;00000;00000]	[5,5]	[1,1]	

FIGURE 3.7 – Résultats pour les murs sont infranchissables

Pour vérifier que nos murs sont bien infranchissables dans notre oracle nous avons vérifié qu'à chaque déplacement du pacman ou du fantôme il n'y a pas de murs. On peut retrouver nos oracles dans le dossier validation3 dans le répertoire laby2player/validation.

3.2.0.4 Les murs bougent dans la direction choisie

Pour tester les murs qui bougent dans la direction choisie on s'est placé dans plusieurs situation différentes. Pour une situation donnée il faut obtenir les murs après un cycle. Quand on a les murs que l'on a obtenus et

les murs que l'on a initialement on les compare.
On peut retrouver nos oracles dans le dossier validation4 dans répertoire laby2player/validation.

3.2.0.5 Le programme s'arrête quand le pacman est sur la sortie

Pour faire ce test nous avons utilisé plusieurs situations initiales différentes. Nous avons aussi dû faire ce test sur l'interface 1 joueur. On fait les situations suivantes :

Le programme s'arrête lorsque Pacman est sur la sortie

	Matrice de murs verticaux	Matrice de murs horizontaux	Situation du Pacman	Situation de la sortie
Test 1	V= [0 0 1 0 ; 0 0 0 0 ; 0 0 0 1 ; 1 0 0 1 ; 0 0 0 1]	H= [0 0 1 0 0 ; 0 0 1 0 0 ; 0 1 0 0 0 ; 1 0 0 0 0]	[3,1]	[4,1]
Test 2	V= [0 0 1 0 ; 0 0 0 0 ; 0 0 0 1 ; 1 0 0 1 ; 0 0 0 1]	H= [0 0 1 0 0 ; 0 0 1 0 0 ; 0 1 0 0 0 ; 1 0 0 0 0]	[1,5]	[5,1]
Test 3	V= [0 0 1 0 ; 0 0 0 0 ; 0 0 0 1 ; 1 0 0 1 ; 0 0 0 1]	H= [0 0 1 0 0 ; 0 0 1 0 0 ; 0 1 0 0 0 ; 1 0 0 0 0]	[4,2]	[2,2]
Test 4	V= [0 0 1 0 ; 0 0 0 0 ; 0 0 0 1 ; 1 0 0 1 ; 0 0 0 1]	H= [0 0 1 0 0 ; 0 0 1 0 0 ; 0 1 0 0 0 ; 1 0 0 0 0]	[3,3]	[3,4]
Test 5	V= [0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0]	H= [0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0]	[3,1]	[4,1]
Test 6	V= [0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0]	H= [0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0]	[1,5]	[5,1]
Test 7	V= [0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0]	H= [0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0]	[4,2]	[2,2]
Test 8	V= [0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0]	H= [0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0]	[3,3]	[3,4]
Test 9	V= [1 0 1 0 ; 1 1 0 1 ; 0 0 0 0 ; 0 1 1 1 ; 1 0 0 0]	H= [0 0 0 1 0 ; 0 1 0 1 0 ; 0 1 0 1 0 ; 0 1 0 0 1]	[3,1]	[4,1]
Test 10	V= [1 0 1 0 ; 1 1 0 1 ; 0 0 0 0 ; 0 1 1 1 ; 1 0 0 0]	H= [0 0 0 1 0 ; 0 1 0 1 0 ; 0 1 0 1 0 ; 0 1 0 0 1]	[1,5]	[5,1]
Test 11	V= [1 0 1 0 ; 1 1 0 1 ; 0 0 0 0 ; 0 1 1 1 ; 1 0 0 0]	H= [0 0 0 1 0 ; 0 1 0 1 0 ; 0 1 0 1 0 ; 0 1 0 0 1]	[4,2]	[2,2]
Test 12	V= [1 0 1 0 ; 1 1 0 1 ; 0 0 0 0 ; 0 1 1 1 ; 1 0 0 0]	H= [0 0 0 1 0 ; 0 1 0 1 0 ; 0 1 0 1 0 ; 0 1 0 0 1]	[3,3]	[3,4]
Test 13	V=[1 0 1 1 ; 1 0 1 1 ; 1 0 0 0 ; 1 1 0 1 ; 1 0 1 0]	H=[1 0 1 1 1 ; 1 0 1 0 0 ; 1 0 1 0 0 ; 1 0 0 1 1]	[3,1]	[4,1]
Test 14	V=[1 0 1 1 ; 1 0 1 1 ; 1 0 0 0 ; 1 1 0 1 ; 1 0 1 0]	H=[1 0 1 1 1 ; 1 0 1 0 0 ; 1 0 1 0 0 ; 1 0 0 1 1]	[1,5]	[5,1]
Test 15	V=[1 0 1 1 ; 1 0 1 1 ; 1 0 0 0 ; 1 1 0 1 ; 1 0 1 0]	H=[1 0 1 1 1 ; 1 0 1 0 0 ; 1 0 1 0 0 ; 1 0 0 1 1]	[4,2]	[2,2]
Test 16	V=[1 0 1 1 ; 1 0 1 1 ; 1 0 0 0 ; 1 1 0 1 ; 1 0 1 0]	H=[1 0 1 1 1 ; 1 0 1 0 0 ; 1 0 1 0 0 ; 1 0 0 1 1]	[3,3]	[3,4]
Test 17	V= [1 0 1 0 ; 1 1 0 1 ; 0 0 0 0 ; 0 1 1 1 ; 1 0 0 0]	H= [0 0 0 1 0 ; 0 1 0 1 0 ; 0 1 0 1 0 ; 0 1 0 0 1]	[3 3]	[3,3]

FIGURE 3.8 – Résultats pour "Le programme s'arrête quand le pacman est sur la sortie"

On peut retrouver nos oracles dans le dossier validation7 dans le répertoire laby2player/validation.

3.2.0.6 Le fantôme peut manger pacman

Pour faire ces tests nous avons fait plusieurs situations initiales différentes :

Ghost peut manger Pacman

	Matrice de murs verticaux	Matrice de murs horizontaux	Situation du Pacman	Situation de Ghost	Situation de la sortie
Test 1	V= [1 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0]	H= [1 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0]	[1,1]	[1,2]	[3,4]
Test 2	V= [0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0]	H= [0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0]	[3,4]	[2,4]	[3,4]
Test 3	V= [0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0]	H= [0 1 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0]	[1,2]	[2,2]	[3,4]
Test 4	V= [0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0]	H= [0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0]	[1,2]	[2,2]	[3,4]
Test 5	V= [0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0 ; 0 0 0 0]	H= [0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0 ; 0 0 0 0 0]	[1,2]	[1,3]	[3,4]

FIGURE 3.9 – Exel des situations initiales

Notre oracle va regarder si quand notre fantôme est à coté de notre pacman on a bien une incrémentation de notre compteur *caught*. Mais rapidement on s'est rendu compte qu'il y a un problème dans notre implémentation : quand les murs vont bouger alors que le fantôme et à coté du pacman le compteur s'incrémente. Il s'incrémente donc 2 fois : Une fois quand il se retrouvent à côté et une autre fois quand les murs se déplacent. Nous n'avons pas réussi encore a réglé ce problème. On peut retrouver nos oracles dans le dossier validation8 dans répertoire laby2player/validation.

3.3 Validation formelle

Pour la validation formelle nous avons créé plusieurs fonctions sur *Matlab* qui nous permettent de générer le modèle de procédé du labyrinthe et les contraintes, le modèle de commande des mouvements des murs et en

dernier lieu l'ordonnancement imposé (mouvement des murs puis mouvement des objets). Nous ne faisons que la modélisation 1 objet, la modélisation des 2 objets étant une des étapes restante à ce projet. Nous allons vous exposer dans cette partie l'ensemble des moyens que nous avons mis en place pour formaliser le labyrinthe.

3.3.1 Génération des modèles

Nous allons dans cette subsection détaillé les fichiers qui se trouvent dans le dossier *src/laby1player/automaton/modelGenerator*.

3.3.1.1 Description des scripts

Vous trouverez dans le dossier spécifié un ensemble de fonctions qui permettent de générer plusieurs fichiers texte. Cette génération doit être lancée depuis le script principal qui porte le nom de : modelGenerator.m. Il est possible de modifier les paramètres du labyrinthe dans ce script principal, en modifiant les variables *wallsV*, *wallsH*, *pacman*, *escape* et *sched* qui sont respectivement la matrice des murs verticaux, des murs horizontaux, la position initiale du Pacman, la position (fixe) de la sortie et le séquencement qui sera appliqué.

Ce script principal fait ensuite appel à 3 fonctions : *AutomatonStrutureLabyCreation*, *AutomatonWallsConstraintsCreation* et enfin *AutomatonSchedulingCreation* qui vont se charger de créer les automates (vous trouverez une explication de chacun de ces automates dans le paragraphe décrivant les modèles). Ces fonctions renvoient des structures contenant l'ensemble des états et fonctions de transitions des automates qu'elles décrivent. Les structures sont en beaucoup de points ressemblantes à la classe AutomateGraph (3.3.3).

Enfin le script principal utilise les structures développées dans le paragraphe précédent pour en extraire les états/transitions de chaque automates. Il peut ensuite les écrire dans des fichiers textes (détaillés dans le paragraphe suivant) ou alors laisser les structures tel quel pour un e utilisation dans un autre script.

3.3.1.2 Description des modèles

Tous les modèles créés par ces fonctions sont enregistrés sous format texte compatible avec *Desuma* (*SaveDesumaFile*). Par la suite on a importé ces fichiers dans *Desuma* qui a créé les automates (*lab*, *walls*, *scheduling*) et dont nous nous sommes servi pour effectuer le produit parallèle entre les automates. Pour faire tourner ce code il faut allez dans répertoire modelGenerator et lancée le programme modelGenerator.m ce répertoire ce trouve dans src/laby1player/automaton.

Quand les modèles sont devenus trop lourds, nous avons chercher à implémenter le produit parallèle en *Matlab* (voir plus de détail ci-dessous 3.3.3).

On obtient bien une représentation automate de notre procédé, on va ensuite utiliser les outils de *Desuma* pour vérifier les propriétés de notre labyrinthe et pouvoir effectuer des vérifications pour le valider.
On a remarqué que *Desuma* ne nous affiche pas les états non accessibles, nous avons seulement les états accessibles et co-accessibles. Par exemple si on prend le labyrinthe suivant :

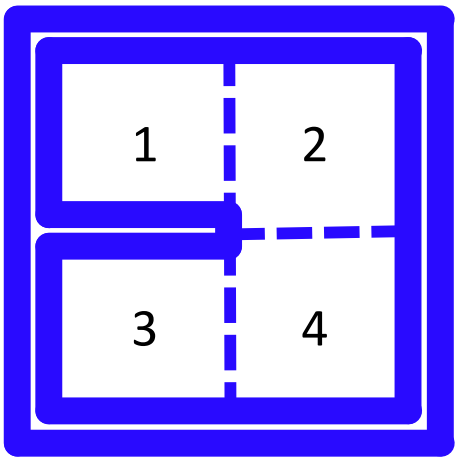


FIGURE 3.10 – Labyrinthe 2x2

De ce labyrinthe nous allons obtenir la modélisation du labyrinthe suivante (le fichier *lab*) :

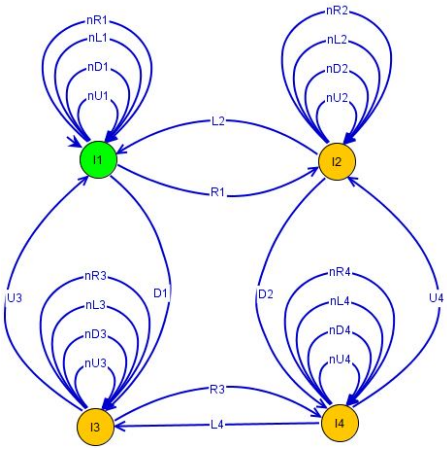


FIGURE 3.11 – modélisation du Labyrinthe 2x2

Sur chaque état on indique quels sont les murs qu'il y a autour de cette case.

On obtient l'automate suivant pour l'ordonnancement (fichier *scheduling*) :

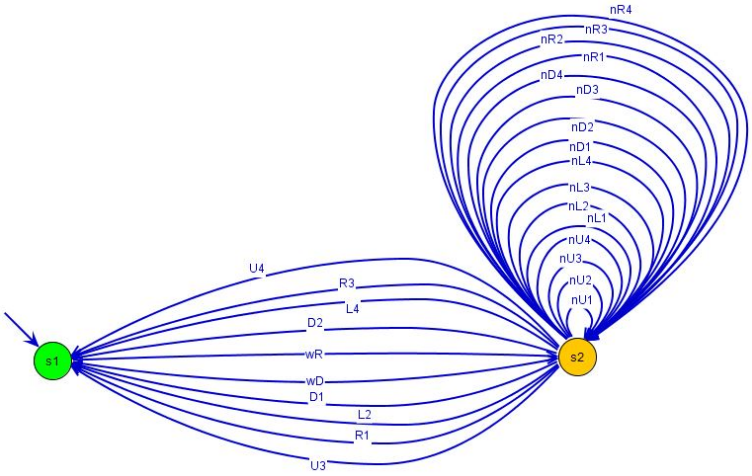


FIGURE 3.12 – modélisation de l'ordonnancement du Labyrinthe 2x2

On commence par les murs puis par les objets. On modélise tous les mouvements autorisés par rapport à la configuration des murs. On obtient un cycle des murs pour un labyrinthe on revient à la situation initiale des murs. Le cycle se trouve par 2 x dimension_laby. On obtient la modélisation suivante (fichier *walls*) :

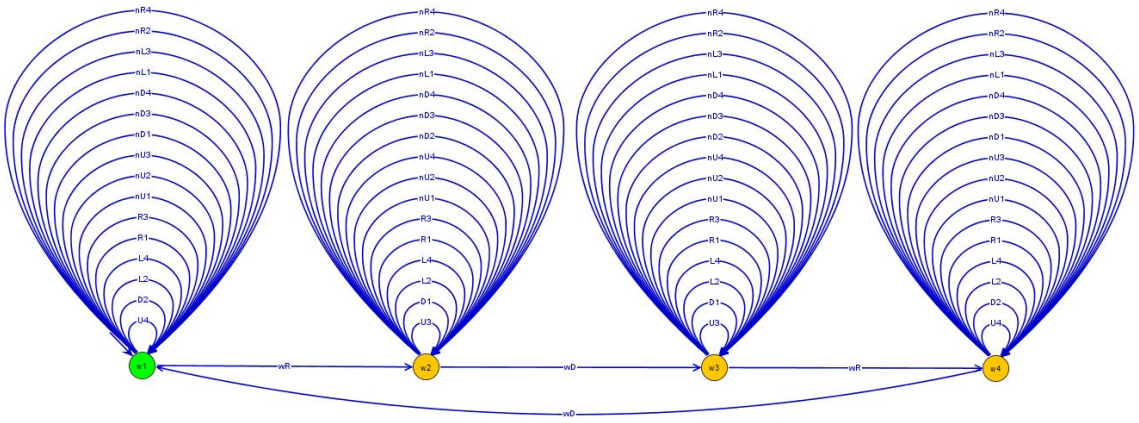


FIGURE 3.13 – modélisation de l'ordonnancement du Labyrinthe 2x2

Sur chaque état on a les informations sur les murs autour et les déplacements autorisé.

État	chemin non autorisé	remarques
1	$L_1, D_1, U_1, R_2, D_2, L_3, D_3, U_3, D_4, R_4$	On ne peut pas allez a gauche, bas et haut de la case 1. On ne peut pas allez a droit et haut de la case 2. On ne peut pas allez a gauche, bas et haut de la case 3. On ne peut pas allez a droit et bas de la case 4.
2	$R_1, D_1, L_2, D_2, U_2, D_3, R_3, L_4, D_4, U_4$	On ne peut pas allez a droit et haut de la case 1. On ne peut pas allez a gauche, bas et haut de la case 2. On ne peut pas allez a gauche, bas et haut de la case 3. On ne peut pas allez a droit et bas de la case 4.
3	$R_1, D_1, L_2, D_2, U_2, D_3, R_3, L_4, D_4, U_4$	On ne peut pas allez a droit et haut de la case 1. On ne peut pas allez a gauche, bas et haut de la case 2. On ne peut pas allez a gauche, bas et haut de la case 3. On ne peut pas allez a droit et bas de la case 4.
4	$L_1, D_1, U_1, R_2, D_2, L_3, D_3, U_3, D_4, R_4$	On ne peut pas allez a gauche, bas et haut de la case 1. On ne peut pas allez a droit et haut de la case 2. On ne peut pas allez a gauche, bas et haut de la case 3. On ne peut pas allez a droit et bas de la case 4.

Après avoir fait les 3 modélisations nous faisons le produit parallèle pour obtenir le procédé de notre labyrinthe avec toutes les dynamiques possibles. Nous obtenons un procédé avec des numéros pour chaque déplacement : l'utilisation de ces numéros est nécessaire pour que les transitions ne soit pas confondu lors du calcul du produit parallèle. Cependant, ces numéros deviennent maintenant un problème, il faut donc faire un raffinement qui servira à enlever chaque numéro. Pour cela il faut enregistrer le fichier *Desuma* de notre procédé sous forme fsm (Fichier ← Export ← DMES File (J'AI PAS DESUMA SOUS LA MAIN, A VERIFIER)). Puis utiliser le programme *rafineAutomaton.m* qui se trouve dans le répertoire *src/laby1player/automaton/optimalCommand*.

Quand le procédé est construit, nous pouvons faire un produit parallèle avec notre commande pour Pacman. Nous obtenons ainsi l'ensemble des états qui sont atteignables par cette commande dans le labyrinthe étudié.

Nous sommes arrivés rapidement à la conclusion que la commande mémoire n'était pas très efficace, le pacman ne trouve pas souvent la sortie. La commande avec priorité n'est pas très efficace nous plus mais la sortie est atteinte plus facilement par le Pacman.

Mais rapidement nous avons eu des problèmes avec le logiciel *Desuma* dû au nombre important de transitions et d'états ainsi qu'à cause de la redondance de l'exercice (Ouvrir des fichiers, Copier - Coller, exporter...). Nous avons donc décidé de passer par *Matlab* pour faire notre produit parallèle et automatiser entièrement ce qui vient d'être décrit dans cette partie (3.3.3). Cependant, nous avons créer un sript dans un premier temps qui permet d'utiliser l'automate généré dans cette subsection pour en déterminer le(s) chemins (s'il existent) qui mène le Pacman vers la sortie. Ce sujet sera l'objet de notre prochain sous-section.

3.3.2 Déterminer l'ensemble des chemins

L'ensemble des scripts décrit ici sont disponibles dans *src/laby1player/automaton/optimalCommand*.

Dans le cas où l'automate obtenu après la composition parralèle possède toujours 1 (ou plusieurs) état(s) marqué(s), nous avons souhaité été capable de déterminé, de manière générique, quels était les séquence amenant à ces états marqués depuis l'état initial. Pour cela, nous avons fait appel à une librairie *Matlab* : *graph*. Cet ensemble de fonction utilise des algorithmes issus de la résolution de Bellman-Ford pour calculer, à partir d'un graph, l'ensemble des chemins possibles, en les clasant du plus court au plus long.

Pour adapter cette fonction à notre automate, nous avons été obligé de transposer l'automate en un graph. C'est l'objet de la fonction *optimalCommand* disponible dans le fichier *.m* qui porte son nom. Cette fonction prend en entrée l'ensemble des matrices de transition qui permettent de décrire les transitions de l'automate, l'état de départ et l'état d'arrivé de l'algorithme. Elle associe un poid équivalent à chaque transition et supprime les transitions stables (elles ne sont pas acceptées par l'algorithme). La fonction affiche ensuite tout les chemins qui ont été trouvé par l'algorithme, du plus court au plus long.

Pour permettre son utilisation directe avec l'automate calculé via *DESUMA* dans RFERENCE ICI LULU ! ! !, nous avons crée un script principal *main*. ce programme va récupérer les données depuis un fichier *DESUMA*, en extraire les matrices de transitions pour les envoyer ensuite vers *optimalCommand*.

Remarque Pour l'utiliser avec un procédé, vous devez modifier le nom appelé dans la ligne 17. Veuillez faire attention aux noms utilisé aux transitions dans l'automate que vous appelé : ils doievent avoir le même nom que dans la liste instancié à l ligne 16.

3.3.3 Produit parallèle avec Matlab

Par soucis de simplification, nous avons choisi d'effectuer le calcul du produit parallèle avec *Matlab*. Il nous a donc été nécessaire de créer un ensemble de classe et fonction pour pouvoir jouir de cette possibilité. Pour cela, nous avons développé l'algorithme de la manière suivante : Soit un premier automate $A1$ avec un langage L_{A1} et un ensemble de matrice de transition : M_{A1}. Soit un second automate $A2$ avec un langage L_{A2} et un ensemble de matrice de transition : M_{A2}.
Nous avons l'automate A qui est le produit parallèle de $A1$ et de $A2$ qui s'écrit :

$$A = A1 /\!\!/ A2$$
$$L_A = L_{A1} \bigcup L_{A2} \text{ avec } L_A \text{ Langage de A}$$

Pour l'ensemble des matrices de A, nous devons procéder de la sorte : pour chaque événements communs aux deux automates, alors :

$$M_A = M_{A1} \otimes M_{A2} \tag{3.1}$$

Sinon, pour chaque événements de A1 (qui n'est pas inclus dans L_{A2}) :

$$M_A = M_{A1} \otimes \mathbb{1} \tag{3.2}$$

Sinon pour chaque événements de A2 (qui n'est pas inclus dans L_{A1}) :

$$M_A = \mathbb{1} \otimes M_{A2} \tag{3.3}$$

La fonction *Matlab* développée est disponible dans src/laby1player/automaton/ParrallelComposition.m.

Utilisation avec un script complet Pour pouvoir exécuter la vérification d'un objectif avec *Matlab*, vous pouvez utiliser le script *MainLaby.m*, disponible dans src/laby1player/automaton/. Ce script va automatiquement généré un procédé grâce aux scripts utilisés dans le début de cette section et lancé depuis ModelGenerator.m. (Si vous souhaitez modifier le labyrinthe, veuillez-vous référer à la documentation du code source) Une fois les modèles obtenus, il les enregistre sous forme d'un objet appartenant à la classe *AutomateGraph*. Le script effectue ensuite automatiquement un produit parallèle pour obtenir le modèle de procédé complet.

Il vous demande ensuite de choisir avec quel objectif vous voulez composez le modèle de commande. Le script s'occupe ensuite d'établir ce modèle avec un dernier produit parallèle.

Classe Automate déterministe : AutomateGraph L'objectif de cette classe *Matlab* est de permettre une utilisation des outils développés pour la validation formelle du labyrinthe. Elle est composé de 4 attributs sous forme de structure : "state", "transition", "matrixTrans", "langage" et "vector", les détails de chaque attributs est disponible dans le code source de la classe (fichier AutomateGraph.m dans le dossier : src/-laby_1_player/automaton). Chaque objet *AutomateGraph* peut accéder à une liste de méthodes qui vont permettre en autre de modifier son langage, d'effectuer des passages entre des description différente, tel que la transformation d'une description vectorielle à matricielle, ou encore d'une description Etats/Transition vers une description en matrice.

Il existe aussi des méthodes plus complexes : la méthode PathResearche est une fonction qui va permettre de trouver l'ensemble des chemins menant d'un état de départ à un état d'arrivé. -Donc, pour obtenir le langage marqué, il suffit de placé l'état initial et l'état marqué respectivement) Une dernière méthode sortStateAutomate permet d'éliminer les états inaccessible de l'automate, en parcourant chaque transition. De plus ample explication sont disponibles dans le fichier présenté au début de ce paragraphe.

Chapitre 4

Les scénarios

Cette partie a été traitée lors du dernier bloc de projet. Elle consiste à créer de nouvelle commandes plus efficace que celles créées de façon intuitive (mémoire et priority).
Pour se faire nous avons commencé par générer automatiquement le modèle complet du labyrinthe pour 1 objet duquel nous avons cherché à trouver une commande atteindre un objectif. Dans un second temps nous avons intégré une notion de non déterminisme porté par l'état initial puis nous avons essayé de trouver des commandes pour atteindre un objectif dans un labyrinthe contenant les 2 objets.

4.1 Scénario 1 - Trouver la commande à partir d'un objectif

4.1.1 objectif 1 : Atteindre la sortie

Nous composons le modèle avec l'objectif et nous obtenons un automate contenant toutes les séquences respectant l'objectif.
La composition parallèle se fait automatiquement en lançant le programme *main_laby* dans laby1player/automaton. Si vous souhaitez tester des objectifs différents il faut ajouter le modèle en .fsm et le rajouter à la suite de choix dans *main_laby*. Cette partie du code renvoie toutes les séquences les plus courtes amenant à chaque état marqué. S'il n'existe pas de chemin, un message s'affiche.

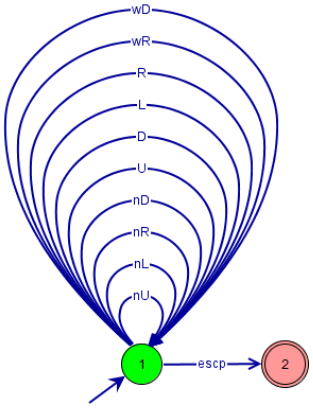


FIGURE 4.1 – Exemple pour un labyrinthe 5x5

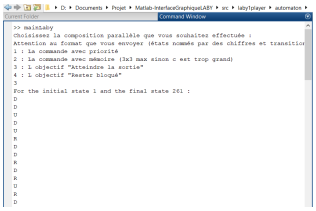


FIGURE 4.2 – Exemple pour un labyrinthe 5x5

4.1.2 objectif 2 : Arriver dans un état bloquant

On peut appliquer la même démarche que précédemment pour n'importe quel objectif, on s'est donc amusé à chercher à bloquer pacman. Comme précédemment on peut voir s'il peut se retrouver bloqué ou non.

4.2 Scénario 2 - Trouver une commande dans un contexte non déterministe

Pour ce scénario, nous avons réutilisé les connaissances acquises lors du projet de M1. Dans ce projet nous avions étudié les automates non déterministes et avions notamment codé la génération de Treillis (voir le rapport

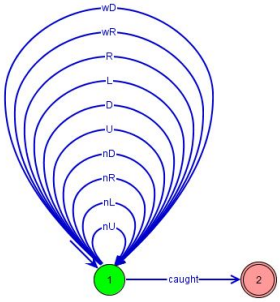


FIGURE 4.3 – Exemple pour un labyrinthe 5x5

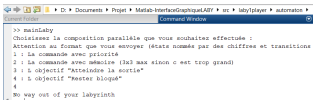


FIGURE 4.4 – Exemple pour un labyrinthe 5x5

de M1 pour la construction et le concept de Treillis). Pour ce projet nous avons décidé de réutilisé la classe Automate que nous avions créées ainsi que la génération de Treillis. On le trouve dans le dossier *automate_nd*. Le principe de ce scénario consiste à trouver la sortie sans connaître la position initiale exacte de l'objet. Le code renvoie une séquence avec laquelle nous sommes sûres d'arriver à la sortie. Nous avons appliqué cela avec les conditions suivantes :

— On peut prendre une transition qui amène dans le même état

— Les murs sont statiques

En lançant le main, on obtient la séquence accessible pour un automate non déterministe. Attention dès qu'on dépasse la taille d'un labyrinthe 4x4, le temps de calcul devient très long. Pour une question de rapidité, on a déjà généré le treillis pour le labyrinthe 5x5 et on l'a conservé dans le dossier en format *.mat*. Le treillis est toujours créé au complet donc on peut utiliser ce treillis pour n'importe quel ensemble d'état initial si on veut réutiliser ce labyrinthe 5x5.

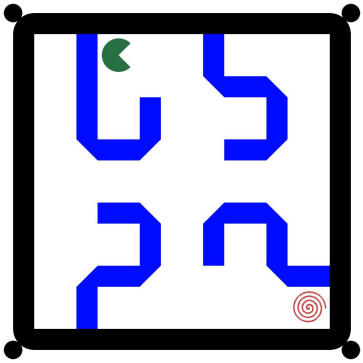


FIGURE 4.5 – Labyrinthe 5x5 testé

4.3 Scénario 3- Trouver des commandes optimales pour le modèle avec les 2 objets

Pour ce scénario on est parti sur le même principe que celui des précédent scénarios avec un unique objet. Du fait que l'on se retrouve avec 2 objets dans ce cas de figure on arrive à un niveau de complexité plus élevé pour la génération automatique du modèle, car on se retrouve avec 2 labyrinthes distincts pour chacun des objets avec la contrainte qu'ils ne puissent pas être sur la même case. Le but c'est d'essayer de trouver comme dans les autres cas, des commandes optimales pour différents objectifs telles que :

4.3.1 Objectif 1 : Le pacman arrive sur Escape en moins de coup possible

Cette démarche est identique à celle avec un seul objet présenté dans le premier scénario.

4.3.2 Objectif 2 : Le pacman se fait tout le temps attraper par ghost

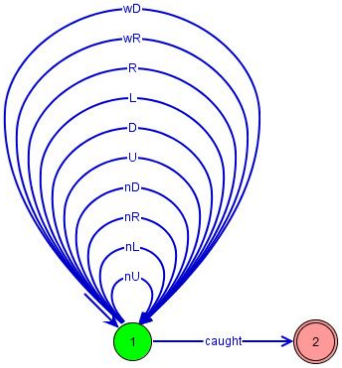


FIGURE 4.6 – Pacman se fait attraper par ghost

4.3.3 Objectif 3 : Le pacman ou ghost se retrouve bloquer entre 4 murs

Pour ce cas de figure il faut élaborer exactement la même commande que pour celui avec un objet.
On a pas encore développer ces objectifs car on est toujours sur la phase de génération automatique du modèle avec 2 objets.

Chapitre 5

Le petite guide d'utilisation

5.1 Organisation du code

A l'ouverture du dossier principal on trouve quatre sous dossiers :

— Doc : contenant la documentation rattachée au projet (rapport, état de l'art et la doc générale)
— Src : contenant le code
— UML : contenant l'UML du code (voir le READ ME)

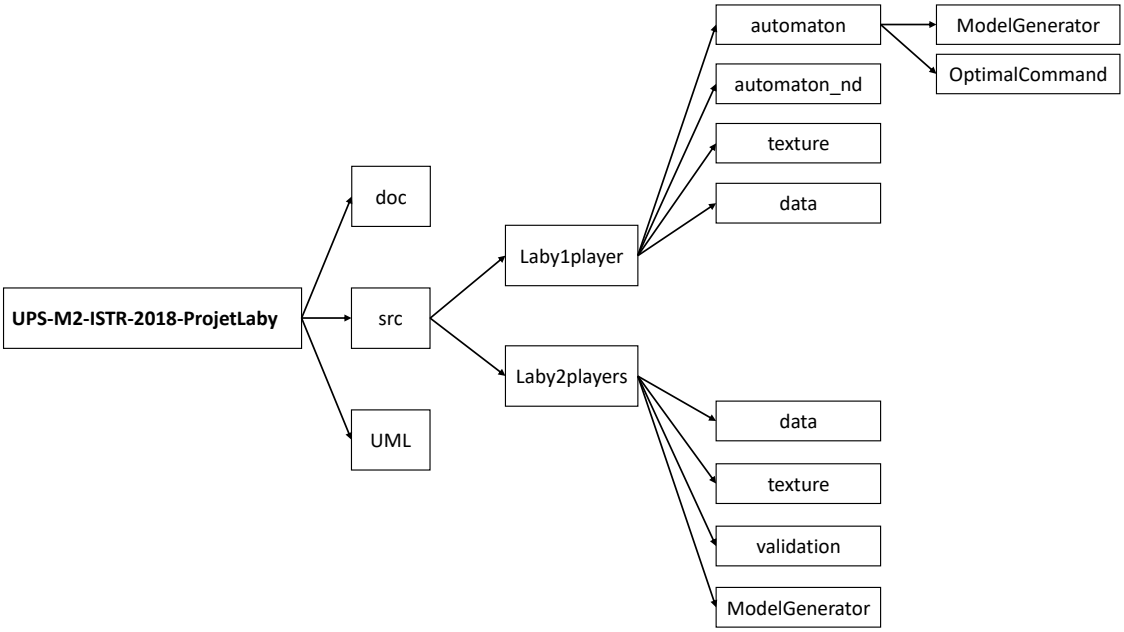


FIGURE 5.1 – Arborescence du dépôt

5.2 Comment lancer l'interface ?

Il y a deux interfaces : celle à 1 objet et celle à 2 objets.
Pour lancer celle à 2 objets, on ouvre le dossier *Laby2players* et on lance le fichier *main*. Pour tout ce qui se rapporte à son utilisation, se référer au chapitre 1 - Interface **??**
Pour lancer l'interface à 1 joueur, on ouvre le dossier *laby1player* et on lance le fichier *main*. Pour tout ce qui se rapporte à son utilisation, se référer au chapitre 1 - Interface **??**.

5.2.1 Comment changer la figure ?

La figure est modifiable avec *figure_ laby.fig* mais implique beaucoup de changements au niveau de *figure-Laby ;m*, du *Wrapper* et du code en règle général. Se référer au chapitre 1 **??** pour le fonctionnement global du code.

5.2.2 Comment changer l'ordonnancement ?

Il est possible de changer l'ordonnancement depuis le *Wrapper* (valable pour les deux interfaces). Comme précédemment, se référer au chapitre 1 **??**

5.2.3 Comment changer la commande des objets ?

Il est possible d'implémenter ses propres commandes pour pacman, ghost ou les murs dans leur fonction respective : ModelPacman, ModelGhost, ModelWalls qu'on trouve dans les deux dossiers *Laby1player* et *Laby2players*. Se référer au chapitre 2 sur les commandes pour l'explication de l'implémentation **??**.

5.3 Comment lancer les validations logicielles ?

Les validations ont été faites pour l'interface à 2 joueurs. On ouvre le dossier *Laby2players*, puis le dossier de validation. Chaque dossier de validation contient l'oracle à faire tourner sur la première version du code.
Le code affiche en ligne de commande s'il y a un problème ou non. Pour cette partie une seconde interface a été créée, une vidéo de tous les coups ou des 100 premiers coups est générée automatiquement dans le but de pouvoir observer visuellement certains cas particuliers. Cette vidéo est enregistrée dans le dossier *data*. La fonction vidéo existe également pour la version à 1 objet. Dans les deux cas on lance le fichier *simulation*. Se référer au chapitre 3 - Vérification et Validation pour plus de détails **??**.

5.4 Comment lancer la validation formelle ?

La validation formelle a été réalisée pour un joueur. On va donc dans le dossier *Laby1player* puis on ouvre le dossier *automaton* duquel on lance le *main*. De là un menu s'affiche dans lequel on choisit la commande à valider.
Le code renvoie s'il existe des séquences et si oui lesquelles pour atteindre l'état marqué. La saisie du labyrinthe étudié se fait dans le dossier *modelGenerator*, dans le fichier *modelGenerator.m*. Attention la position initiale est toujours fixée à la case 1. Se référer au chapitre 3 pour plus de détails **??**.

5.4.1 Comment générer le procédé indépendamment ?

Le modèle de procédé est calculé directement en fonction du labyrinthe donné dans le dossier *modelGenerator* dans le fichier *modelGenerator.m*. Si vous souhaitez le générer indépendamment, il faut aller dans le dossier *modelGenerator* et lancer le fichier *modelGenerator.m* puis le raffiner avec la fonction *rafineAutomaton.m*. Se référer au chapitre 3 pour plus de détails **??**.

5.4.2 Comment valider ma propre commande ?

Si vous créez votre propre commande, il faut la créer sous Desuma et l'enregistrer sous le format *.fsm* et l'enregistrer dans le dossier *automaton*. Attention le langage doit être en accord avec le langage connu du procédé et lors de la saisie sous Desuma les états doivent être nommés avec des numéros (par exemple : 1 et non e1 ou état1). Pour plus de détails se référer au chapitre2 - Les commandes **??**.
Une fois la commande enregistrée lancer le *main* normalement et suivez les indications du menu.

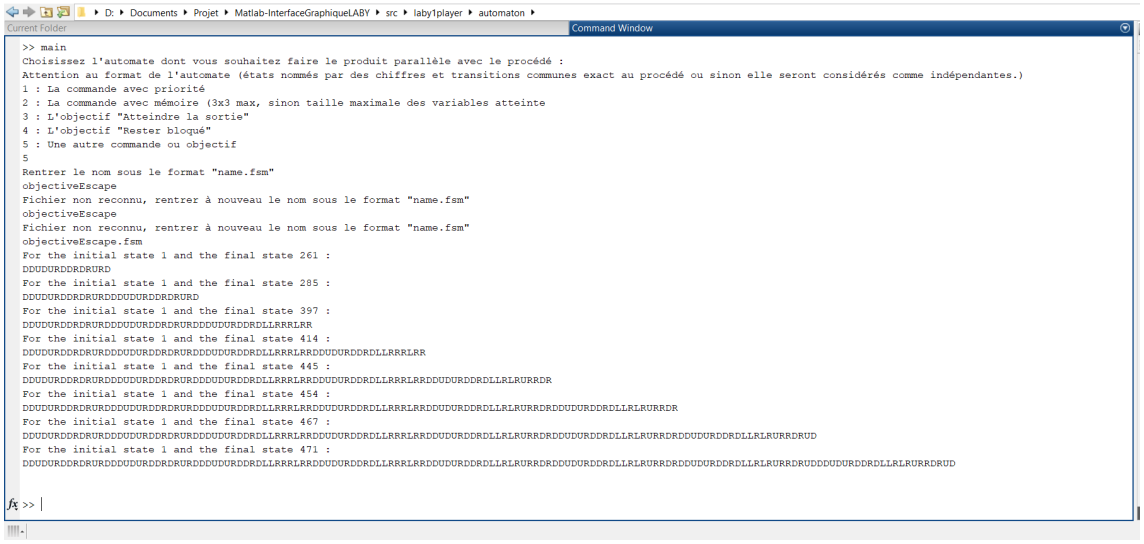


FIGURE 5.2 – Exécution du produit parallèle pour la vérification et le scénario 1

5.5 Comment lancer le scénario 1 ?

Le scénario 1 est prévu pour 1 objet, on se place dans le dossier *laby1player* puis on ouvre le dossier *automaton* duquel on lance le fichier *main*. C'est le même code que précédemment sauf qu'on utilise le produit parallèle pour trouver une commande pour un objectif et non pour valider une commande. Du menu qui s'affiche on choisit l'objectif à atteindre.
Le code renvoie s'il existe une séquence et si oui, la plus optimale pour atteindre l'objectif.

5.5.1 Comment tester mon propre objectif ?

Même démarche que pour *Comment valider ma propre commande ?*.

5.6 Comment lancer le scénario 2 ?

Le scénario 2 est prévu pour 1 objet, on se place dans le dossier *laby1player* puis on ouvre le dossier *automaton_nd* duquel on lance le fichier *main*. Un menu s'affiche pour choisir l'ensemble d'états initiaux et l'état marqué.
Le code renvoie la séquence pour atteindre l'état marqué. Voir le chapitre 4 - Les scénarios pour plus de détails ??.

5.7 Comment lancer le scénario 3 ?

Il ne fonctionne pas encore mais la génération automatique de modèle a commencé à être codé dans le sous dossier *ModelGenerator* de *Laby2players*.

Chapitre 6

Conclusions

Annexes