Implémentation d'un Joueur de Réseaux de Pétri sur Robot Lego

Domaine Systèmes Embarqués 2009-2010

Modèles de systèmes embarqués : modèles discrets, modèles hybrides

Yann Ameho
Marcelo Aravjo
Yannick Bisiaux
Nadège Dussouy
Vincent Garbi
Thiago Medeiros
Jean-Baptiste Vallart

Professeur Responsable : Janette Cardoso

ISAE-SUPERO Novembre 2009

Table des matières

0) Introduction

Le but de ce projet est d'intégrer un joueur de Réseau de Pétri sur un des robots Lego du LIA de Supaero. Le travail consiste à réaliser cette intégration de manière quasi automatique quelque soit le réseau de Pétri choisi initialement, le but final est de permettre à un utilisateur extérieur de créer son réseau, de le traduire en termes compréhensibles par un joueur intégré sur le Robot pour finalement assister à son jeu.

Le travail a été réparti dans trois équipes. Une équipe chargée de réaliser un réseau correcte à l'aide de Tina, une équipe chargée de réaliser une interface permettant de traduire ce réseau à destination de l'équipe réalisant le joueur propre à implémenter sur le robot Lego.

Les caractéristiques du robot sur lequel nous travaillons sont les suivantes :

Capteurs

Le robot dispose de quatre capteurs :

- Le capteur tactile renvoie un booléen (0 ou 1) suivant qu'il est pressé ou non.
- Le capteur sonore renvoie un niveau d'intensité sonore exprimé en dB. Ce capteur peut-être configuré pour mesurer ce niveau dans le spectre sonore audible.
- Le capteur photosensible renvoie un niveau de gris (entre 0 et 100%) perçu
- Le capteur ultrason mesure la distance qui le sépare du premier objet faisant obstacle. Il renvoie un nombre de centimètres à cet obstacle. La mesure renvoyée est en fait une moyenne des 10 derniers échantillons relevés.

NB : Malgré ce qui est indiqué sur la notice d'utilisation Lego, le capteur ultrason ne fonctionne correctement que s'il est branché sur un port différent du port 4.

Actionneurs

Le robot dispose de trois moteurs identiques. Ces trois moteurs peuvent être commandés en vitesse ou en position. On peut les faire s'arrêter en roue libre (grâce aux frottements) ou en forçant l'arrêt. De plus, il est possible de récupérer une mesure de vitesse et de position sur chacun des moteurs. Au démarrage du robot, la position des moteurs est réinitialisée à zéro. Dans notre cas d'utilisation, cela signifie que le robot doit démarrer pinces fermées.

Le présent rapport se divise en parties pour chacune des trois équipes citées précédemment.

1) Travail sous TINA

Dans cette partie, nous avons créé les réseaux de Pétri avec le logiciel Tina qui seront en suite joués sur le robot. Selon les différents capteurs et les capacités du robot nous avons élaboré différents réseaux de Pétri plus ou moins complexes.

A) Réseaux de Pétri pour les tests

Nous avons tout d'abord produit des réseaux simples et avec peu de places et transitions afin de les tester sur le robot.

a) Avancer

Présentation

Le premier réseau permet de tester les actions avancer et arrêter ainsi que les capteurs de son et d'ultrason. Avec le logiciel Tina, nous avons construit le réseau de la Figure1.

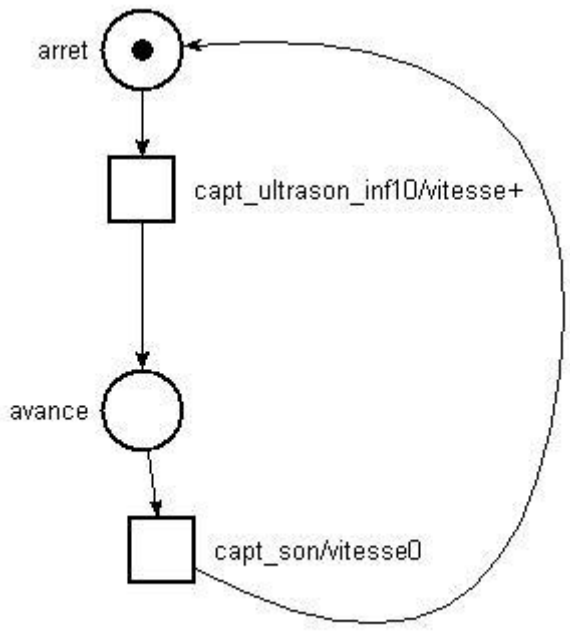


Figure 1

Avec ce réseau, le robot peut:

- Avancer.
- S'arrêter.

Les conditions à satisfaire pour réaliser les actions sont :

- Le capteur à ultrason détecte une distance inférieure à 10 cm.
- Le capteur de son détecte un son supérieur à 50% de sa valeur maximale.
-

Analyse par énumération de marquage

Pour analyser le réseau de Pétri, nous avons utilisé le graphe de marquage (Figure 2) proposé par la logiciel Tina. Comme le réseau est une simple séquence, nous n'avons pas la réalisation d'actions non voulue.

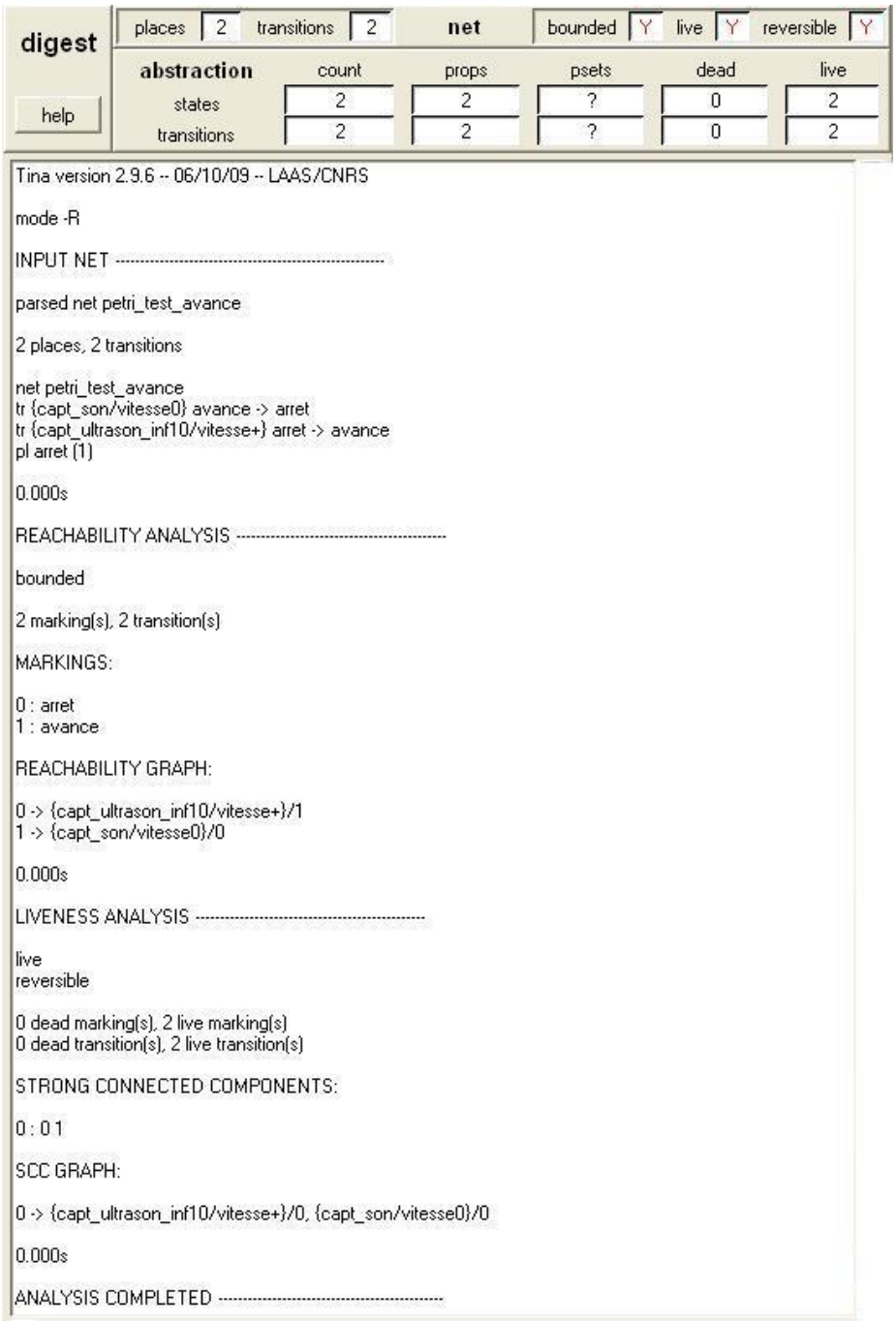


Figure 2

De plus avec l'analyse de Tina, nous nous assurons que le réseau est borné, vivant et réinitialisable.

Analyse structurelle

L'analyse structurelle (Figure 3) à l'aide du logiciel Tina confirme que le réseau est borné car chaque place du réseau appartient à au moins une composante.

P-SEMI-FLOWS GENERATING SET ----------

invariant

arret avance

0.000s

T-SEMI-FLOWS GENERATING SET ----------

consistent

{capt_son/vitesse0} {capt_ultrason_inf10/vitesse+}

0.000s

ANALYSIS COMPLETED ----------

Figure 3

Interprétation du réseau

D'après les analyses et la configuration du réseau, le robot ne peut se trouver que dans deux états possibles : soit il avance, soit il est arrêté.

De plus, ce changement d'état s'effectue avec des évènements externes au réseau. Le réseau est donc non autonome.

b) Reculer

Présentation

Le second réseau (Figure 4) représente les actions reculer et s'arrêter et les conditions sur les capteurs d'ultrason et de son.

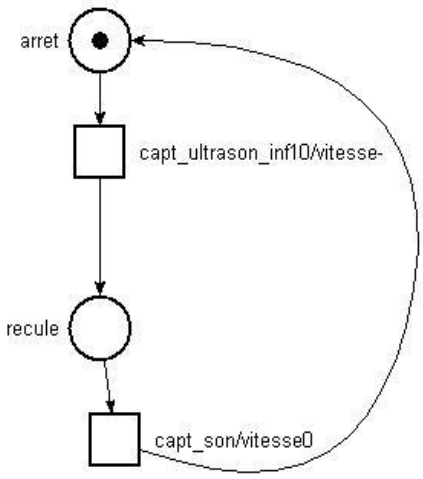


Figure 4

Avec ce réseau, Les actions du robot sont:

- Reculer.
- S'arrêter.

Les conditions à satisfaire sont les même que dans le test avancer.

Analyse par énumération des marquages

Le graphe de marquage (Figure 5) nous montre que le réseau répond bien à nos attentes. De plus avec l'analyse de Tina, nous nous assurons que le réseau est borné, vivant et réinitialisable.

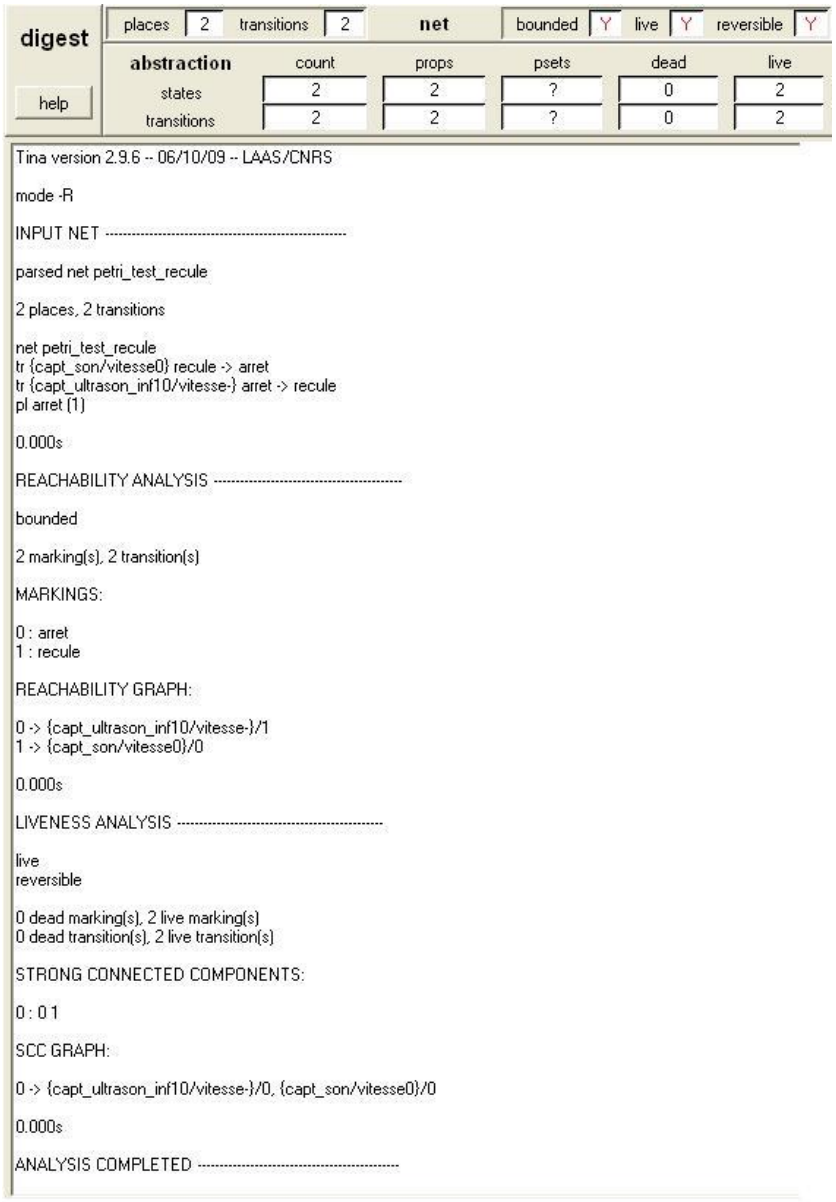


Figure 5

Analyse structurelle

D'après l'analyse structurelle (Figure 6) du logiciel Tina, nous retrouvons bien la propriété du réseau borné.

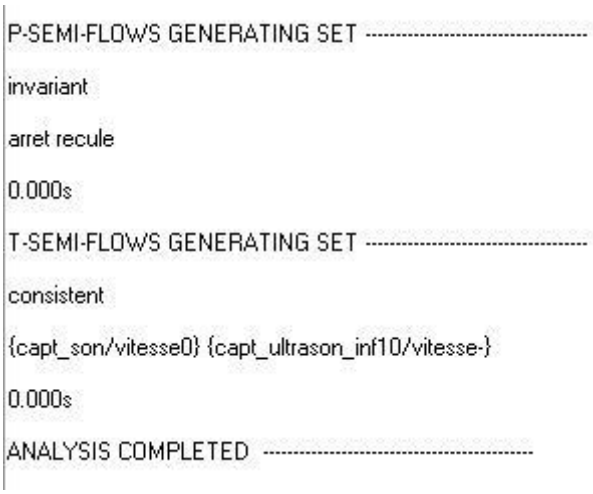


Figure 6

Interprétation du réseau

Comme le réseau précédent, le robot ne peut faire que deux actions indépendantes : soit il recule, soit il est arrêté. De même il est non autonome à cause de la présence des capteurs au niveau des transitions du réseau.

c) Attraper une balle

Présentation

Dans ce dernier test, nous examinons les actions ouvrir et fermer les pinces du robot ainsi que les actions avancer et reculer déjà connues. Les nouvelles conditions utilisent les capteurs de contact et celui de détection de niveau de gris.

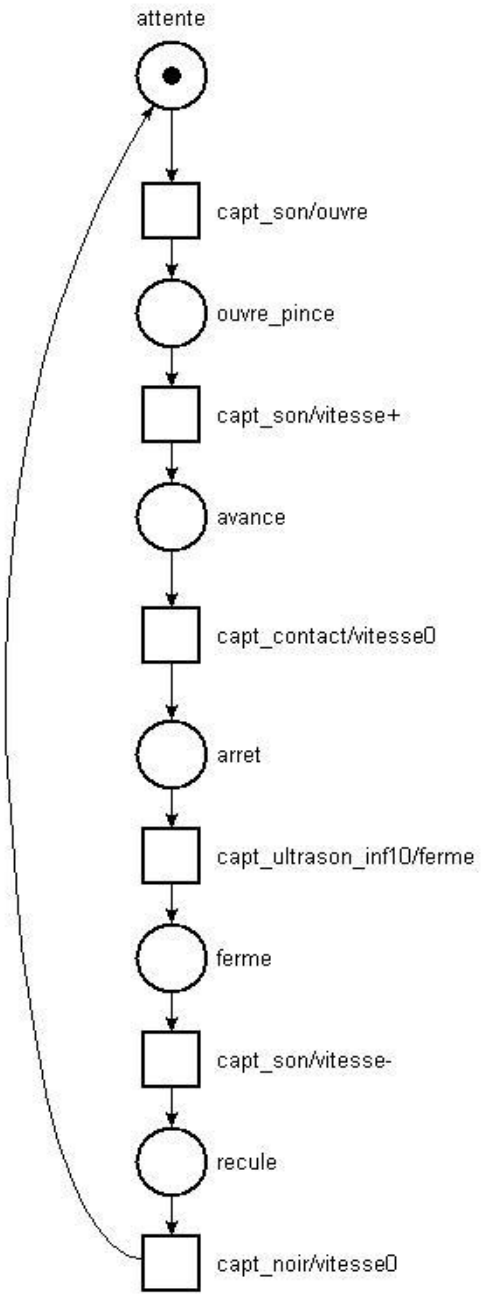


Figure 7

Avec ce réseau, le robot peut:

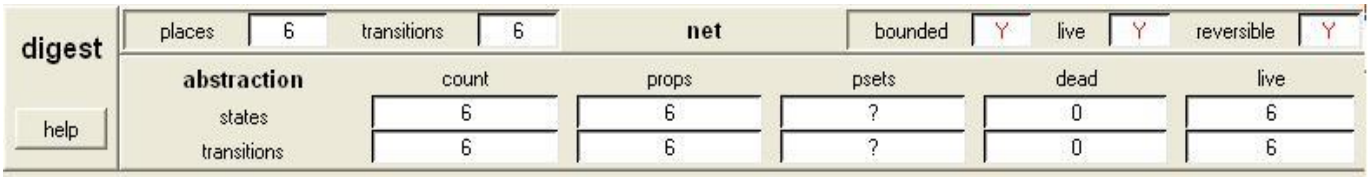
- Reculer.
- Ouvrir les pinces
- Fermer les pinces
- Avancer
- S'arrêter.

Les conditions à satisfaire pour que le robot réalise ce réseau sont :

- Le capteur à ultrason détecte une distance inférieure à 10 cm.
- Le capteur de son détecte un son supérieur à 50% de sa valeur maximale.
- Le capteur de contact signale au robot lorsque celui-ci est en contact avec la balle.
- Le capteur de détection de nuance de gris renvoie une valeur de gris supérieure à 80%.

Analyse par énumération du marquage

Le graphe de marquage (Figure 8) du logiciel Tina nous indique que le réseau est borné, vivant et réinitialisable.




```
mode -R

INPUT NET ---------------------------------------------------

parsed net petri_test_attrape_balle

6 places, 6 transitions

net petri_test_attrape_balle
tr {capt_contact/vitesse0} avance -> arret
tr {capt_noir/vitesse0} recule -> attente
tr {capt_son/ouvre} attente -> ouvre_pince
tr {capt_son/vitesse+} ouvre_pince -> avance
tr {capt_son/vitesse-} ferme -> recule
tr {capt_ultrason_inf10/ferme} arret -> ferme
pl attente (1)

0.000s

REACHABILITY ANALYSIS ---------------------------------------

bounded

6 marking(s), 6 transition(s)

MARKINGS:

0 : attente
1 : ouvre_pince
2 : avance
3 : arret
4 : ferme
5 : recule

REACHABILITY GRAPH:

0 -> {capt_son/ouvre}/1
1 -> {capt_son/vitesse+}/2
2 -> {capt_contact/vitesse0}/3
3 -> {capt_ultrason_inf10/ferme}/4
4 -> {capt_son/vitesse-}/5
5 -> {capt_noir/vitesse0}/0

0.000s

LIVENESS ANALYSIS -------------------------------------------

live
reversible

0 dead marking(s), 6 live marking(s)
0 dead transition(s), 6 live transition(s)

STRONG CONNECTED COMPONENTS:

0 : 0 1 2 3 4 5

SCC GRAPH:

0 -> {capt_son/ouvre}/0, {capt_son/vitesse+}/0, {capt_contact/vitesse0}/0, {capt_ultrason_inf10/ferme}/0, {capt_son/vitesse-}/0, {capt_noir/vitesse0}/0

0.000s

ANALYSIS COMPLETED ------------------------------------------
```

Figure 8

Analyse structurelle

Comme dans les réseaux précédents, l'analyse structurelle (Figure 9) de celui-ci confirme que le réseau est borné car chaque place appartient à au moins une composante

P-SEMI-FLOWS GENERATING SET ----------

invariant

arret attente avance ferme ouvre_pince recule

0.000s

T-SEMI-FLOWS GENERATING SET ----------

consistent

{capt_contact/vitesse0} {capt_noir/vitesse0} {capt_son/ouvre} {capt_son/vitesse+} {capt_son/vitesse-} {capt_ultrason_inf10/ferme}

0.000s

ANALYSIS COMPLETED ----------

Figure 9

Interprétation du réseau

Dans ce réseau, le robot effectue toutes les actions simples qui peut effectuer et il utilise tous ses capteurs dans les transitions. Ce réseau est toujours non autonome car nous faisons intervenir des capteurs.

B) Réseau de Pétri pour les missions finales

Dans les missions finales, nous avons cherché à utiliser toutes les actions réalisables par le robot et tous les capteurs qu'il possède.

a) Première mission finale

La première mission finale est une simple séquence afin d'utiliser toutes les capacités du robot sans la présence de conflit ou parallélisme afin d'assurer un bon fonctionnement du robot.

Présentation

Le réseau de Pétri (Figure 10) de cette première mission finale permet mettre en œuvre toutes les actions et capteurs du robot.

Avec ce réseau, le robot peut:

- Avancer.
- S'arrêter.
- Reculer.
- Faire ½ tour.
- Fermer pince.
- Ouvrir pince.

Les conditions à satisfaire pour réaliser les actions sont :

- Le capteur à ultrason détecte une distance inférieure à 10 cm et supérieure à 30cm.
- Le capteur de détection de nuance de gris renvoie une valeur de gris supérieure à 80%.
- Le capteur de contact signale au robot lorsque celui-ci est en contact avec la balle.
- Le capteur de son détecte un son supérieur à 50% de sa valeur maximale.

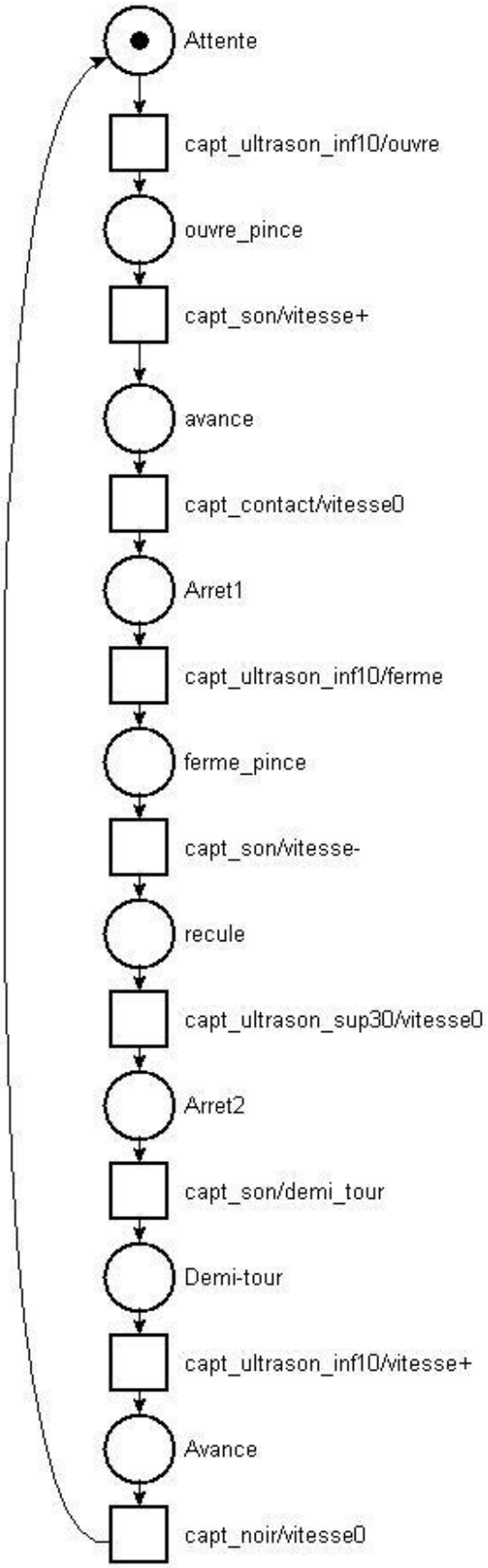
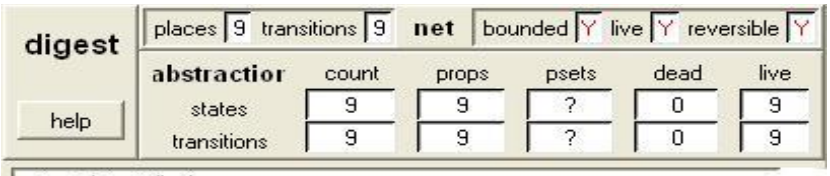

Attente
capt_ultrason_inf10/ouvre
ouvre_pince
capt_son/vitesse+
avance
capt_contact/vitesse0
Arret1
capt_ultrason_inf10/ferme
ferme_pince
capt_son/vitesse-
recule
capt_ultrason_sup30/vitesse0
Arret2
capt_son/demi_tour
Demi-tour
capt_ultrason_inf10/vitesse+
Avance
capt_noir/vitesse0

Figure 10

Analyse par énumération de marquage

D'après le graphe de marquage (Figure 11) du logiciel Tina, nous nous assurons que le réseau de pétri est borné, vivant et réinitialisable.




```
net petri_test_final
tr {capt_contact/vitesse0} avance -> Arret1
tr {capt_noir/vitesse0} Avance -> Attente
tr {capt_son/demi_tour} Arret2 -> {Demi-tour}
tr {capt_son/vitesse+} ouvre_pince -> avance
tr {capt_son/vitesse-} ferme_pince -> recule
tr {capt_ultrason_inf10/ferme} Arret1 -> ferme_pince
tr {capt_ultrason_inf10/ouvre} Attente -> ouvre_pince
tr {capt_ultrason_inf10/vitesse+} {Demi-tour} -> Avance
tr {capt_ultrason_sup30/vitesse0} recule -> Arret2
pl Attente (1)

0.000s

REACHABILITY ANALYSIS ----------------------------------------

bounded

9 marking(s), 9 transition(s)

MARKINGS:

0 : Attente
1 : ouvre_pince
2 : avance
3 : Arret1
4 : ferme_pince
5 : recule
6 : Arret2
7 : {Demi-tour}
8 : Avance

REACHABILITY GRAPH:

0 -> {capt_ultrason_inf10/ouvre}/1
1 -> {capt_son/vitesse+}/2
2 -> {capt_contact/vitesse0}/3
3 -> {capt_ultrason_inf10/ferme}/4
4 -> {capt_son/vitesse-}/5
5 -> {capt_ultrason_sup30/vitesse0}/6
6 -> {capt_son/demi_tour}/7
7 -> {capt_ultrason_inf10/vitesse+}/8
8 -> {capt_noir/vitesse0}/0

0.016s

LIVENESS ANALYSIS ----------------------------------------

live
reversible

0 dead marking(s), 9 live marking(s)
0 dead transition(s), 9 live transition(s)

STRONG CONNECTED COMPONENTS:

0 : 0 1 2 3 4 5 6 7 8

SCC GRAPH:
```

Figure 11

Analyse structurelle

Cette analyse (Figure 12) confirme bien que le réseau est borné.

```
P-SEMI-FLOWS GENERATING SET ----------

invariant

Arret1 Arret2 Attente Avance {Demi-tour} avance ferme_pince ouvre_pince recule

0.000s

T-SEMI-FLOWS GENERATING SET ----------

consistent

{capt_contact/vitesse0} {capt_noir/vitesse0} {capt_son/demi_tour} {capt_son/vitesse+} {capt_son/vitesse-} {capt_ultrason_inf10/ferme} {capt_ultrason_inf10/ouvre} {capt_ultrason_inf10/vitesse+} {capt_ultrason_sup30/vitesse0}

0.000s

ANALYSIS COMPLETED ----------
```

Figure 12

Interprétation du réseau

Dans ce réseau le robot commence par ouvrir les pinces lorsque l'on passe à une distance inférieure à 10 cm devant le capteur d'ultrason. Ensuite il avance quand il détecte un son. Il finit par s'arrêter quand le capteur de contact est sollicité : le robot est en contact avec la balle. On repasse la main devant le capteur ultrason pour que le robot ferme ses pinces. Une fois les pinces fermée on tape dans les mains afin que le robot recule. Celui-ci recule jusqu'à que son capteur d'ultrason détecte une distance supérieure à 30 cm d'un obstacle de référence placé derrière la balle. Le robot effectue un demi-tour lorsqu'il perçoit un son. Pour finir, il avance jusqu'à ce qu'il détecte une ligne noire au sol grâce à son capteur de nuance de gris.

Dans cette séquence, nous utilisons toutes les capacités du robot d'une façon simple.

Le réseau reste non autonome car il dépend d'évènements externes au réseau.

b) Deuxième mission finale

Après avoir fait tous les réseaux de Pétri pour les modèles simples et les avoir testé, nous avons raffiné le modèle complet du système.

Présentation

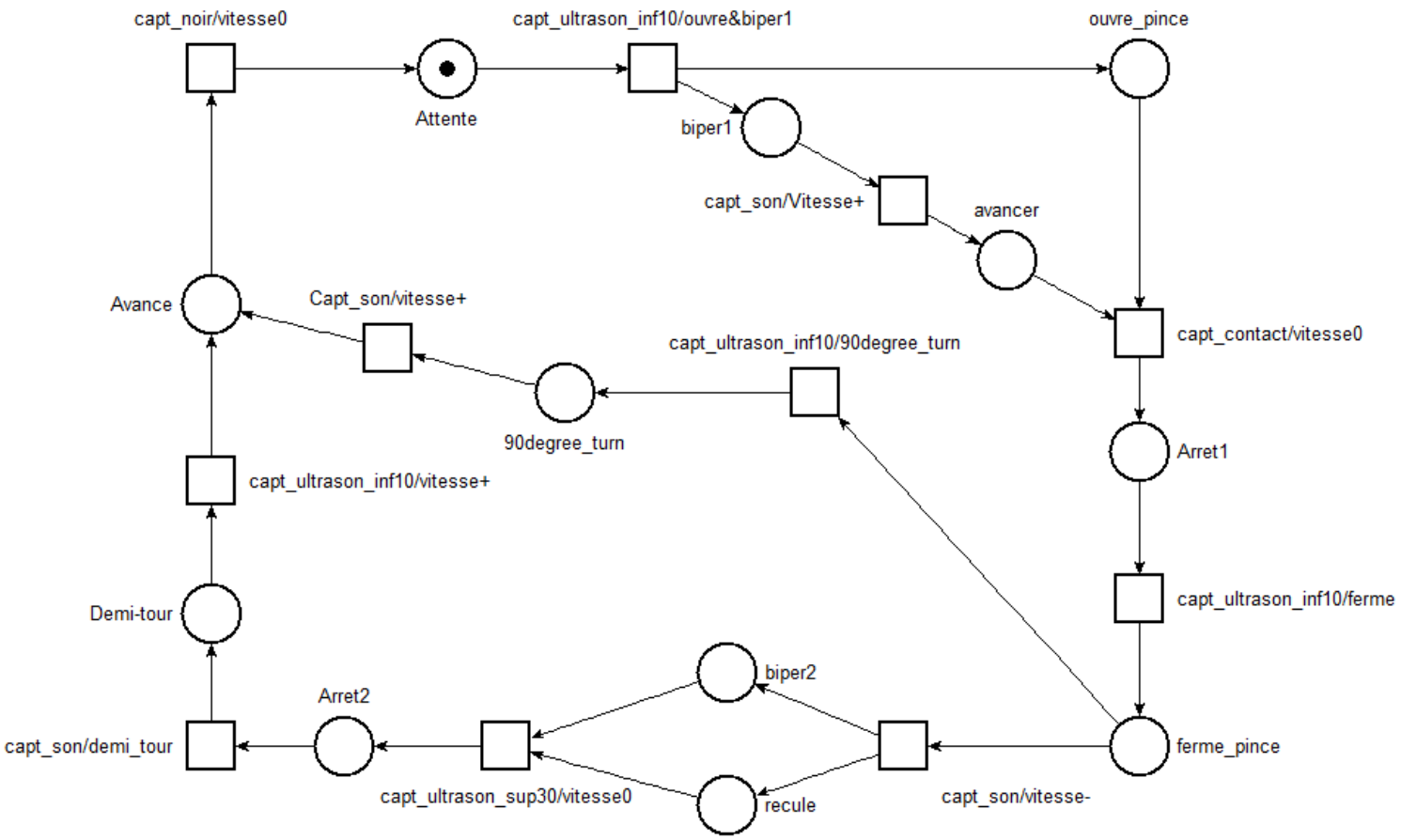


Figure 13

Le réseau de Pétri qui décrit le modèle final du robot se trouve dans la Figure 10.

Avec ce réseau, le robot peut:

- Avancer.
- S'arrêter.
- Reculer.
- Biper.
- Faire ½ tour.
- Faire ¼ tour.
- Fermer pince.
- Ouvrir pince.

Les conditions à satisfaire pour réaliser les actions sont :

- Le capteur à ultrason détecte une distance inférieure à 10 cm et supérieure à 30cm.
- Le capteur de détection de nuance de gris renvoie une valeur de gris supérieure à 80%.
- Le capteur de contact signale au robot lorsque celui-ci est en contact avec la balle.
- Le capteur de son détecte un son supérieur à 50% de sa valeur maximale.

Au début, on active le capteur à ultrason en passant la main devant à une distance inférieure à 10 cm. Cette action va mettre en œuvre un parallélisme qui correspond aux actions de *biper* et *ouvrir pince*. Dans ce parallélisme, il y a un troisième état – *avancer* – qui est exécuté après la détection du son par le capteur sonore. En ce moment-là, le robot avance avec ses pinces ouvertes et attends que la prochaine transition soit activée, comme illustré par Figure 11 au-dessous.

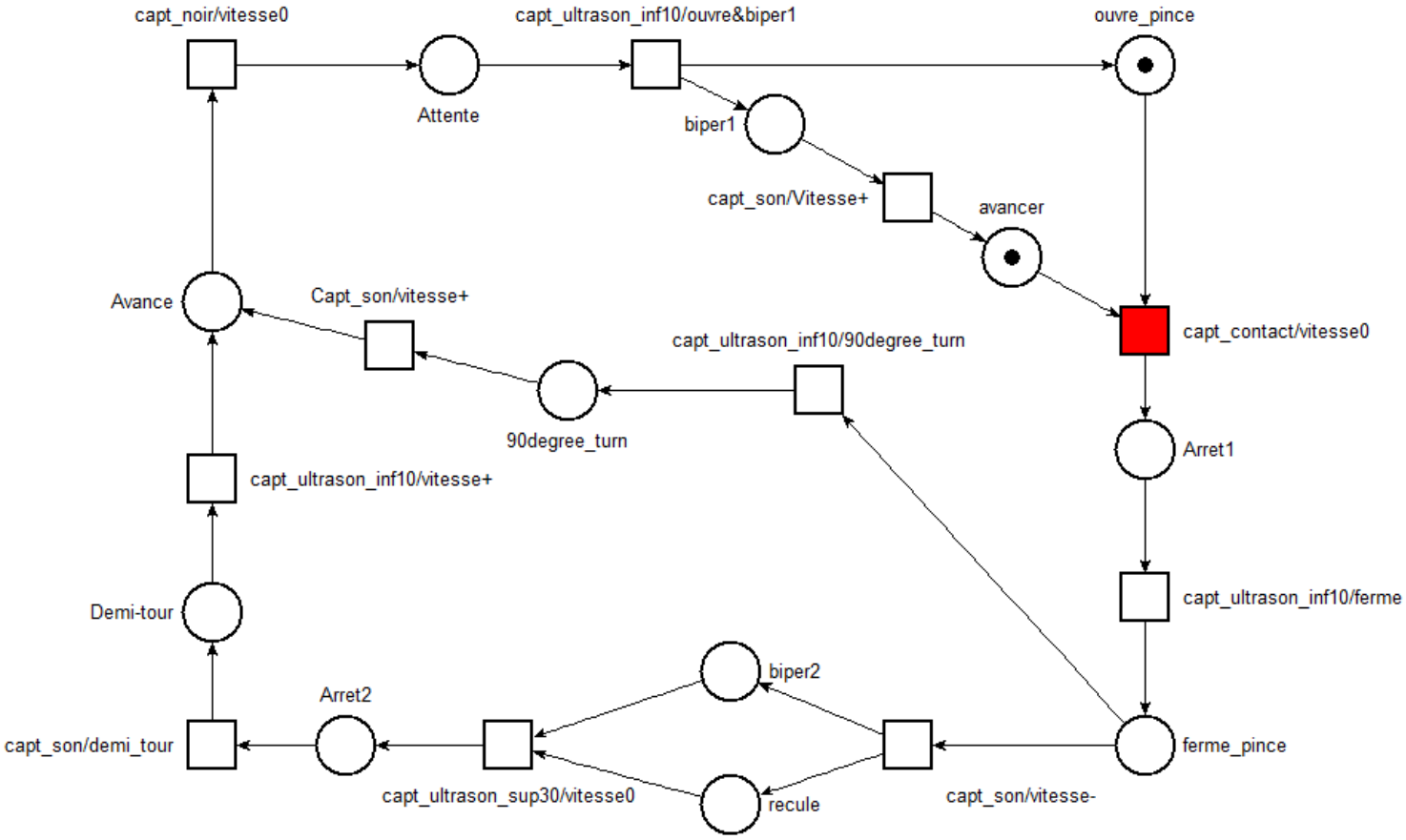


Figure 14

Quand le robot touche la balle, son capteur de contact lui signale qu'il est en contact avec la balle. Puisque le capteur a été actionné, le robot s'arrêt. Ensuite, il faut qu'on sensibilise une autre fois le capteur à ultrason à une distance inférieur à 10 cm pour qu'il ferme sa pince et prenne la balle. En cet état, on a le choix de tourner 90º ou de reculer en bipant. La Figure 12 montre ce choix.

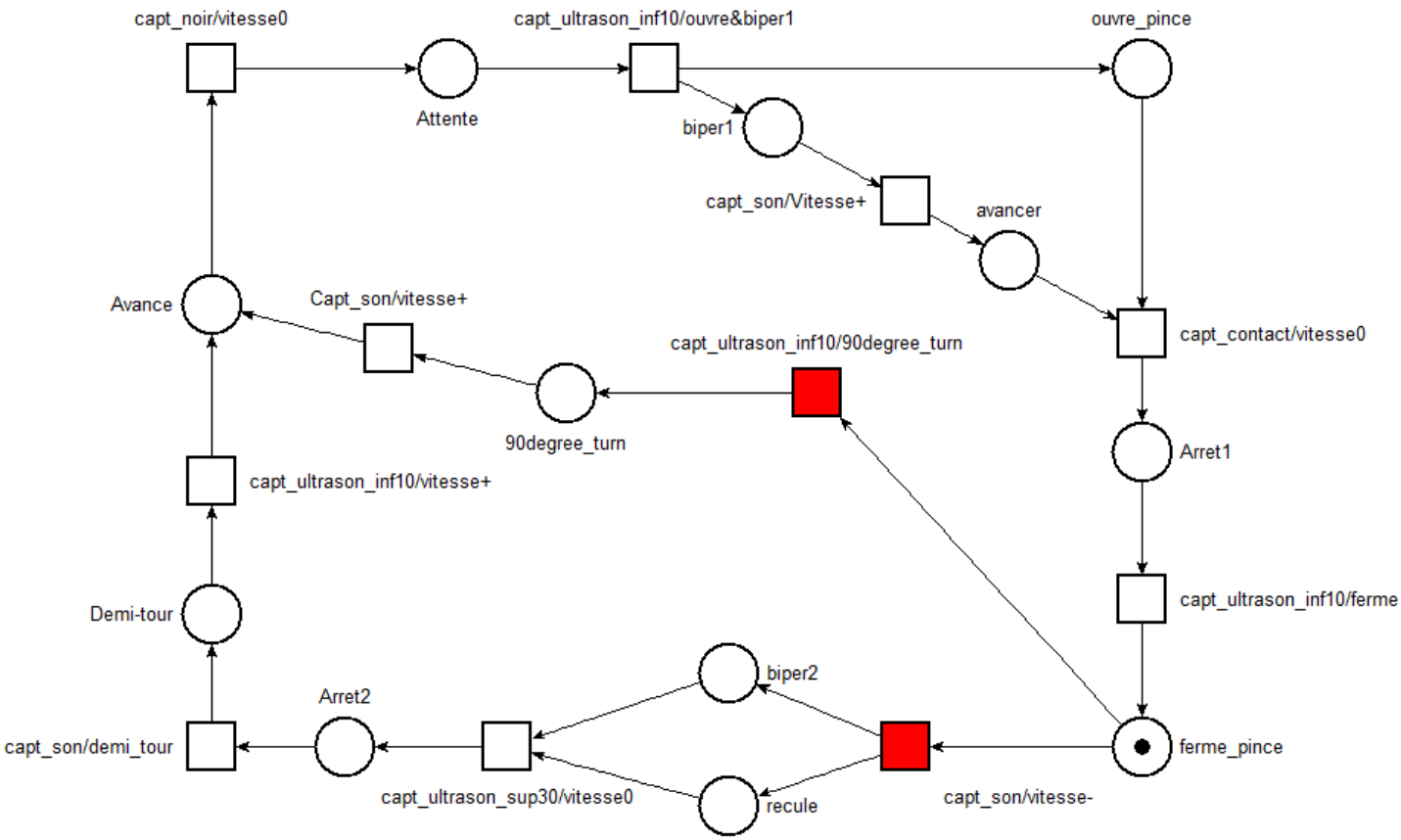


Figure 15

Si l'option de tourner de 90º est choisie, le robot va donc tourner et après son capteur de son va attendre un signal sonore pour permettre le robot d'avancer. Si l'autre option est choisie, on aura un autre parallélisme qui comprendre les actions de biper et reculer – Figure 13.

Pour arriver à l'état *Arret2* il faut que le capteur à ultrason détecte une distance de 30 cm par rapport à la position où la balle a été prise. Afin de réussir à mesurer la distance exacte, il est nécessaire d'utiliser un obstacle de référence pour que le signal ultrason puisse retourner une valeur au robot. Après l'état arrêter, le robot doit faire un demi tour pour avance jusqu'à la fin du chemin – la ligne noir. La première transition qui devra être franchie a besoin d'un signal sonore. Une fois que le son a été capté, le robot tourne et attend pour le capteur à ultrason capter une distance inférieure à 10 cm pour avancer.

Finalement, le robot va s'arrêter quand il va détecter la ligne noire sur le chemin.

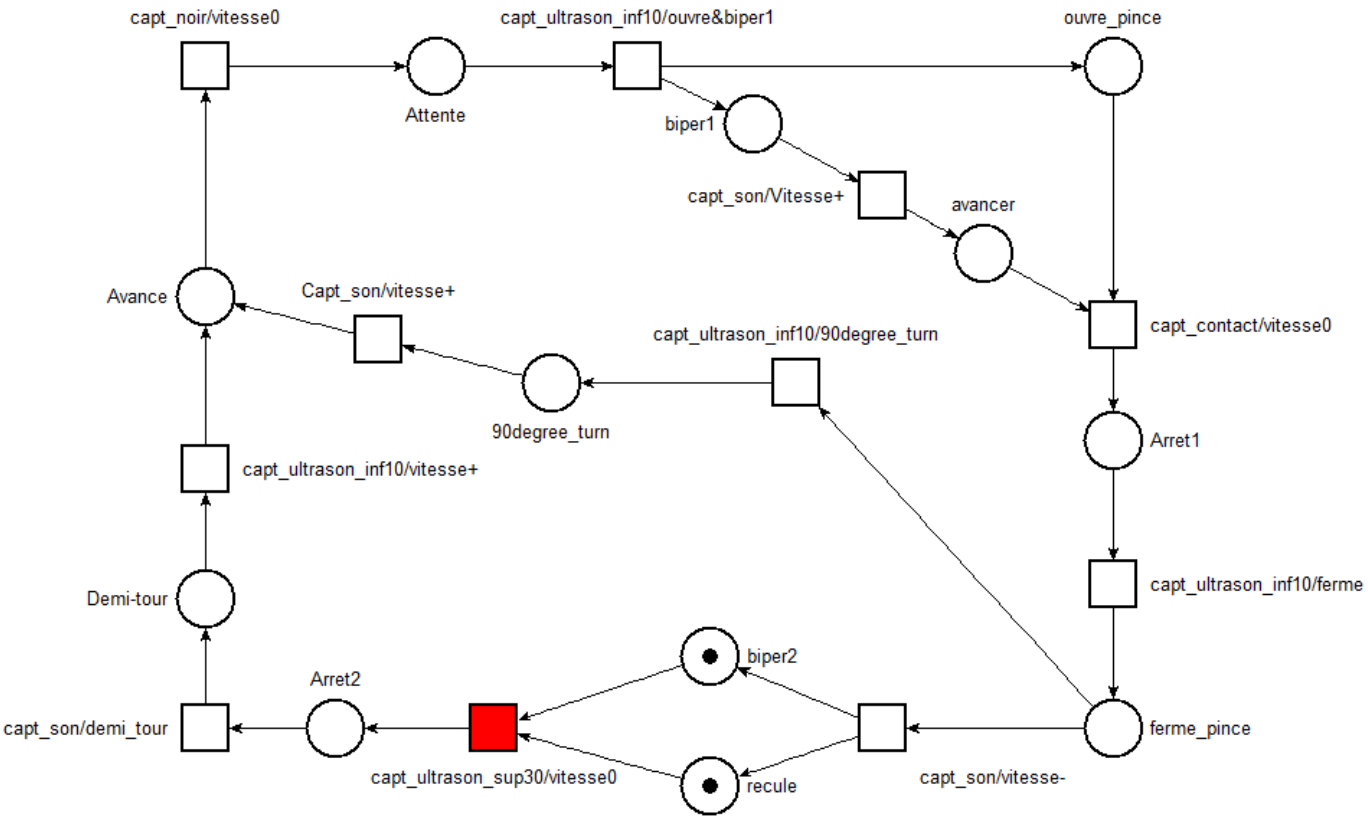


Figure 16

Analyse par énumération de marquage

Pour analyser le réseau de Pétri, nous avons utilisé le graphe de marquage (Figure 14) proposé par le logiciel Tina. De plus avec l'analyse de Tina, nous nous assurons que le réseau est borné, vivant et réinitialisable.

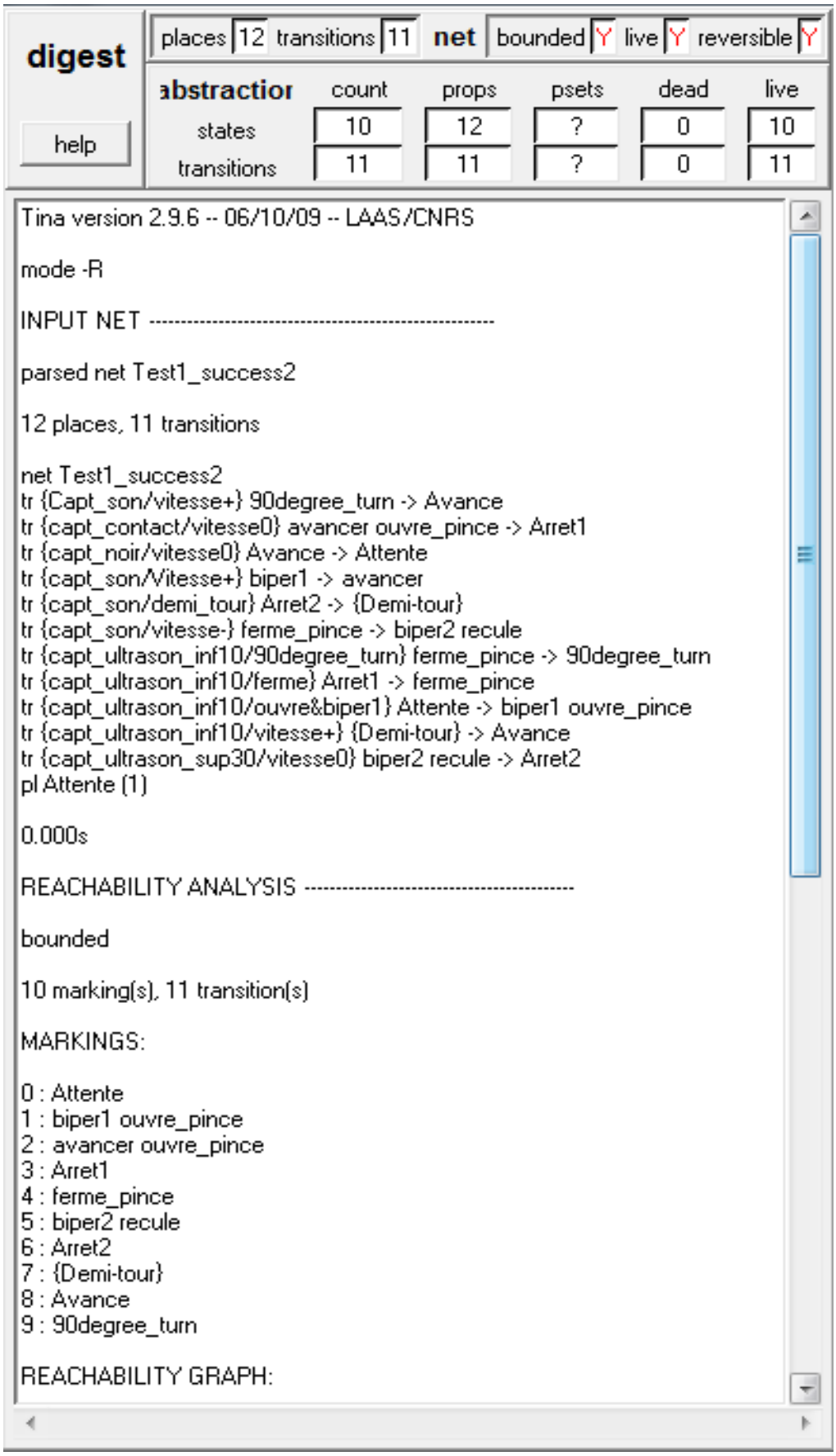


Figure 17. Première partie d'analyse du graphe de marquage

digest

help

places	12	transitions	11	net	bounded	Y	live	Y	reversible	Y

abstraction	count	props	psets	dead	live
states	10	12	?	0	10
transitions	11	11	?	0	11

```
1 : biper1 ouvre_pince
2 : avancer ouvre_pince
3 : Arret1
4 : ferme_pince
5 : biper2 recule
6 : Arret2
7 : {Demi-tour}
8 : Avance
9 : 90degree_turn

REACHABILITY GRAPH:

0 -> {capt_ultrason_inf10/ouvre&biper1}/1
1 -> {capt_son/Vitesse+}/2
2 -> {capt_contact/vitesse0}/3
3 -> {capt_ultrason_inf10/ferme}/4
4 -> {capt_son/vitesse-}/5, {capt_ultrason_inf10/90degree_turn}/9
5 -> {capt_ultrason_sup30/vitesse0}/6
6 -> {capt_son/demi_tour}/7
7 -> {capt_ultrason_inf10/vitesse+}/8
8 -> {capt_noir/vitesse0}/0
9 -> {Capt_son/vitesse+}/8

0.000s

LIVENESS ANALYSIS ----------------------------------------

live
reversible

0 dead marking(s), 10 live marking(s)
0 dead transition(s), 11 live transition(s)

STRONG CONNECTED COMPONENTS:

0 : 0 1 2 3 4 9 5 6 7 8

SCC GRAPH:

0 -> {capt_ultrason_inf10/ouvre&biper1}/0, {capt_son/Vitesse+}/0, {capt_contact/vitesse0}/0, {capt_ultrason_inf10/ferme}/0, {capt_son/vitesse-}/0, {capt_ultrason_inf10/90degree_turn}/0, {Capt_son/vitesse+}/0,

 {capt_ultrason_sup30/vitesse0}/0, {capt_son/demi_tour}/0, {capt_ultrason_inf10/vitesse+}/0, {capt_noir/vitesse0}/0,

0.000s

ANALYSIS COMPLETED ----------------------------------------
```

Figure 18. Deuxième partie d'analyse du graphe de marquage

Analyse structurelle

L'analyse structurelle (Figure 15) à l'aide du logiciel Tina confirme que le réseau est borné car chaque place du réseau appartient à au moins une composante.

```
Struct version 2.9.6 -- 06/10/09 -- LAAS/CNRS

parsed net Test1_success2

12 places, 11 transitions

net Test1_success2
tr {Capt_son/vitesse+} 90degree_turn -> Avance
tr {capt_contact/vitesse0} avancer ouvre_pince -> Arret1
tr {capt_noir/vitesse0} Avance -> Attente
tr {capt_son/Vitesse+} biper1 -> avancer
tr {capt_son/demi_tour} Arret2 -> {Demi-tour}
tr {capt_son/vitesse-} ferme_pince -> biper2 recule
tr {capt_ultrason_inf10/90degree_turn} ferme_pince -> 90degree_turn
tr {capt_ultrason_inf10/ferme} Arret1 -> ferme_pince
tr {capt_ultrason_inf10/ouvre&biper1} Attente -> biper1 ouvre_pince
tr {capt_ultrason_inf10/vitesse+} {Demi-tour} -> Avance
tr {capt_ultrason_sup30/vitesse0} biper2 recule -> Arret2
pl Attente (1)

0.000s

P-SEMI-FLOWS GENERATING SET ----------------------------------

invariant

90degree_turn Arret1 Arret2 Attente Avance {Demi-tour} avancer biper1 ferme_pince recule
90degree_turn Arret1 Arret2 Attente Avance {Demi-tour} ferme_pince ouvre_pince recule
90degree_turn Arret1 Arret2 Attente Avance {Demi-tour} avancer biper1 biper2 ferme_pince
90degree_turn Arret1 Arret2 Attente Avance {Demi-tour} biper2 ferme_pince ouvre_pince

0.000s

T-SEMI-FLOWS GENERATING SET ----------------------------------

consistent

{Capt_son/vitesse+} {capt_contact/vitesse0} {capt_noir/vitesse0} {capt_son/Vitesse+} {capt_ultrason_inf10/90degree_turn} {capt_ultrason_inf10/ferme} {capt_ultrason_inf10/ouvre&biper1}
{capt_contact/vitesse0} {capt_noir/vitesse0} {capt_son/Vitesse+} {capt_son/demi_tour} {capt_son/vitesse-} {capt_ultrason_inf10/ferme} {capt_ultrason_inf10/ouvre&biper1} {capt_ultrason_inf10/vitesse+} {capt_ultrason_sup30/vitesse0}

0.000s

ANALYSIS COMPLETED  ----------------------------------------
```

Nous concluons que le système fonctionne bien et respecte les contraintes.

2) Traducteur

L'équipe traducteur est chargée de fournir à l'équipe Joueur un fichier texte formaté de manière à ce que l'intégration du Réseau de Pétri construit à l'aide de Tina se fasse le plus facilement possible sur le robot Lego.

Pour intégrer ces données afin qu'elles soient jouées, il manque un certain nombre d'informations, comme l'association formelle des capteurs et actionneurs en jeu dans le RDP à ceux effectivement disponibles. Il faut en effet fournir un fichier texte au robot qui soit normalisé selon des règles prédéfinies, car réaliser un fichier sous Tina n'implique pas de respecter la dénomination exacte des capteurs et des actionneurs, rendant alors la réalisation du traducteur nécessaire.

Notre choix s'est porté sur la réalisation d'une IHM permettant à l'utilisateur de lire le fichier Tina, d'y détecter les transitions et les places, d'y associer actionneurs et capteurs avec les valeurs souhaitées, et enfin de l'exporter dans un format normalisé et lisible par le Joueur implémenté sur le robot.

A) Présentation générale de l'IHM Traducteur.

L'IHM se présente sous la forme d'une fenêtre à plusieurs onglets :

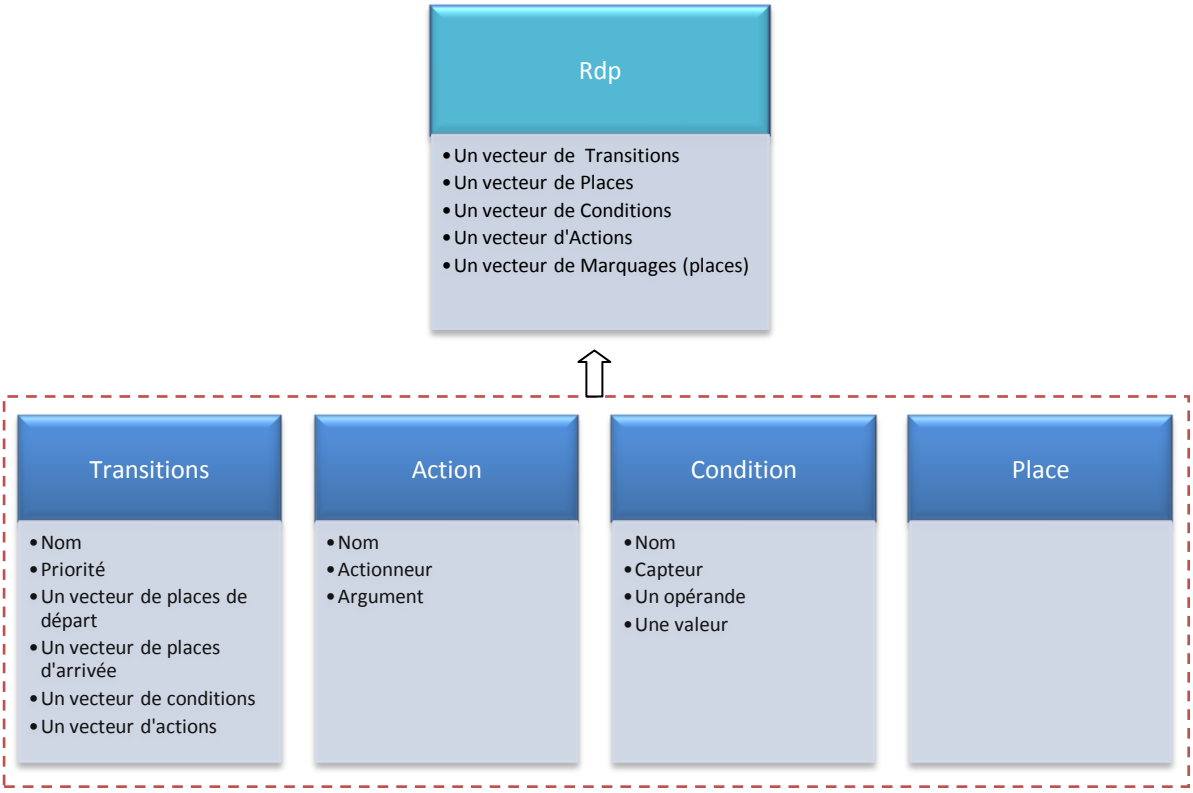
- L'onglet Transitions permet d'associer les capteurs et actionneurs aux transitions détectées lors de l'import.
- L'onglet Actions permet de définir les Actions possibles et les actionneurs associés.
- L'onglet Capteur permet de définir les conditions sur les transitions et les valeurs de capteurs associées.

L'objectif est de fournir un outil intuitif et facile de prise en main à l'utilisateur.

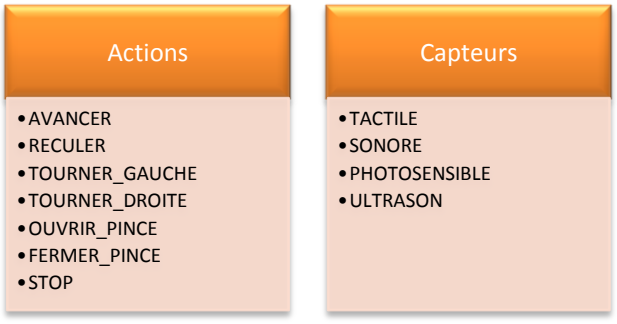
B) Principes de l'architecture

L'IHM a été codée en Java sous l'interface de développement NetBeans. Bien que le travail de codage constitue une part non négligeable de cette partie, nous ne ferons ici pas figurer le code et certains choix techniques propres au langage choisi.

Le principe repose sur la construction, à l'aide du fichier importé, d'une instance Réseau de Pétri aux attributs suivants :



La liste des Actionneurs et des Capteurs utilisée étant :



Au fur et à mesure de son travail sur l'IHM, l'utilisateur construit une instance de la classe Rdp, en donnant une valeur à chacun de ses paramètres. Ce processus n'est pas automatique par choix, car il dépend du réseau de Pétri à jouer, et car l'intervention humaine agit en contrôle tout au long de celui là via certaines fonctions implémentées lors de la conception.
Notons également que des avertissements sous forme de popup sont inclus dans l'IHM afin de prévenir certaines mauvaises manipulations.

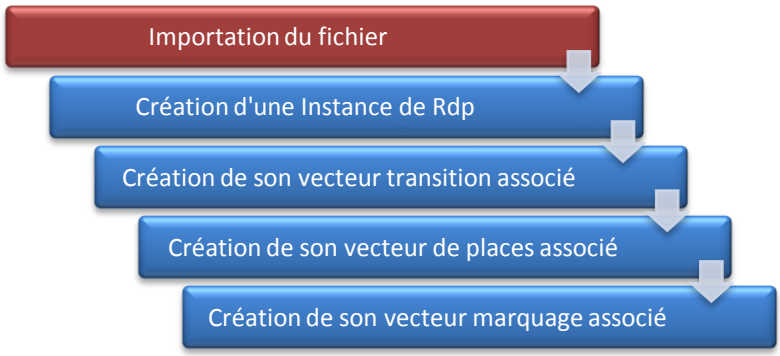
C) Importation du fichier TINA

A ce stade, l'utilisateur a à sa disposition un fichier texte décrivant le réseau de Pétri qu'il a réalisé à l'aide du logiciel Tina. Ce fichier texte se présente sous la forme suivante :

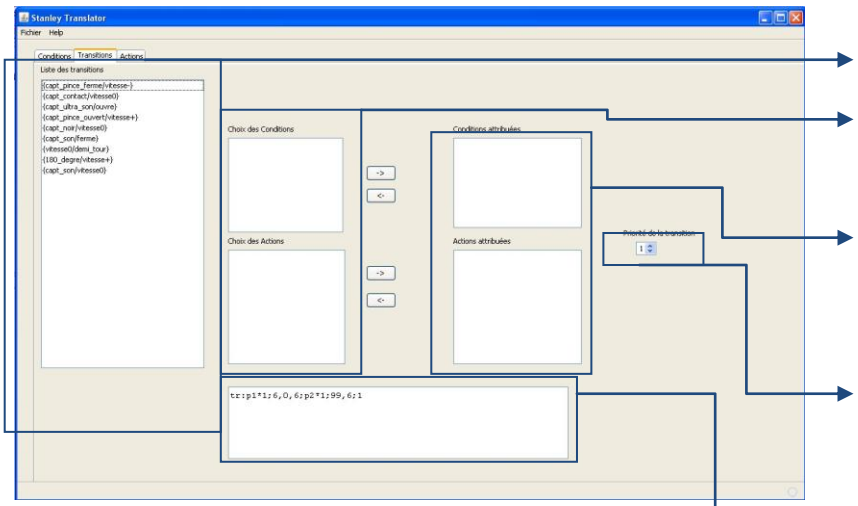
```
tr {capt_contact/vitesse0} [0,w[ avance*2 -> Arret1*3
tr {capt_ultra_son/ouvre} [0,w[ Attente -> ouvre_pince
tr {capt_pince_ouvert/vitesse+} [0,w[ ouvre_pince -> avance
tr {capt_pince_ferme/vitesse-} [0,w[ ferme_pince -> recule
```

Chaque ligne représente une transition (`tr`). Entre accolades se trouvent le capteur sensibilisant la transition et l'action associée à la transition, informations séparées par un / (`capt_contact/vitesse0`). Viennent ensuite un intervalle de temps utilisé dans le cadre de réseaux de Pétri temporisés (`[0,w[`), et la liste des places de départ et d'arrivée ainsi que leur multiplicité (`avance*2 -> Arret1*3`). *(Note : le fichier texte présenté ici ne décrit pas de mission concrète)*

L'importation de ce fichier se fait via le menu Fichier de la barre de menu. Le logiciel se charge alors de détecter les transitions listées dans ce fichier, le nom qu'il leur a été donné ainsi que les places associées et leur multiplicité.



Une fois ces données créées, la liste des transitions et le nom qui leur a été donné dans Tina est intégrée dans l'onglet Transitions.



Affichage du nom des transitions.

Affichage des Actions et Conditions définies par l'utilisateur.

Affichage des Actions et Conditions associées à la transition sélectée.

Affichage de la priorité de la transition (défaut :1).

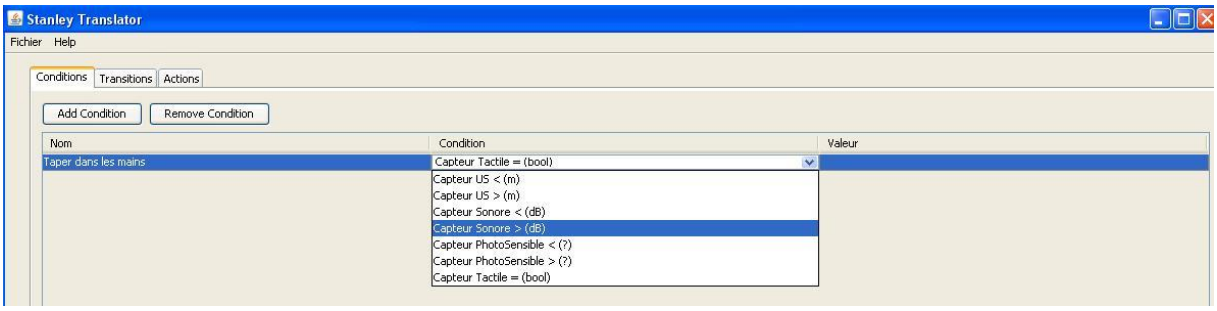
Affichage d'un résumé des transitions et de leur données.

Le résumé des transitions et des données qui leur ont été attribuées ne font pas figurer directement les noms des actions et des transitions, par soucis de lisibilité (leur nombre n'étant pas limité). L'ordre est le suivant :

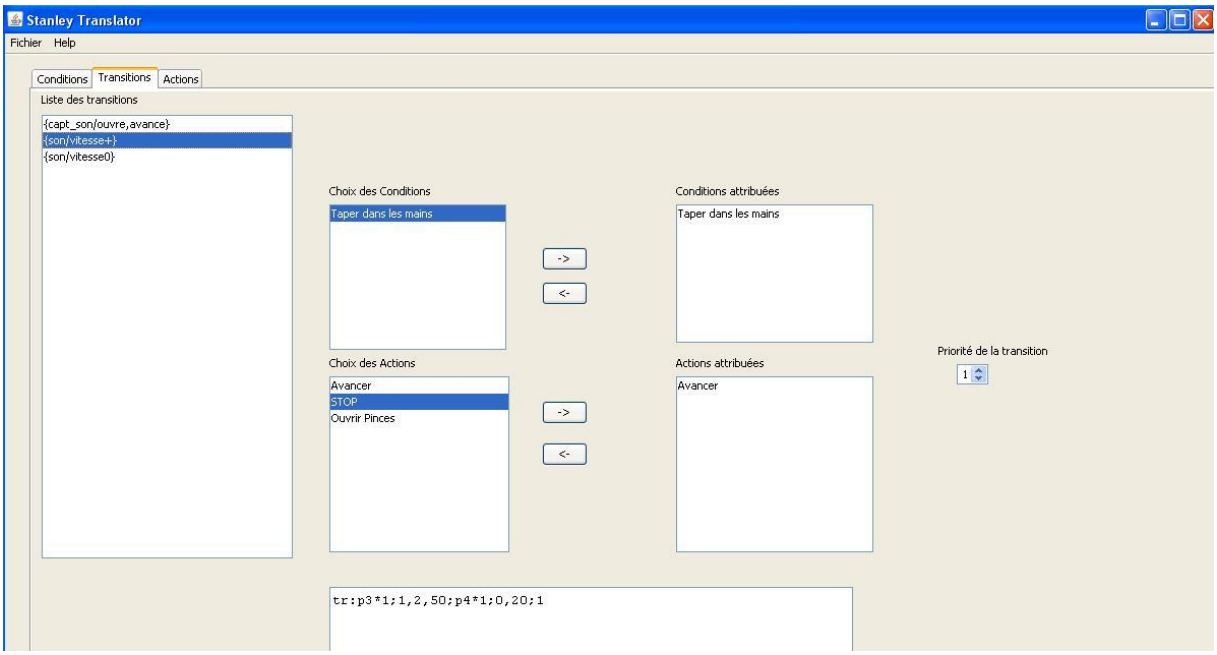
Actions	Capteurs	Opérandes
• 1 AVANCER • 2 RECULER • 3 TOURNER_GAUCHE • 4 TOURNER_DROITE • 5 OUVRIR_PINCE • 6 FERMER_PINCE • 7 STOP	• 1 TACTILE • 2 SONORE • 3 PHOTOSENSIBLE • 4 ULTRASON	• 1 = • 2 ≤ • 3 ≥ • 4 < • 5 >

D) Association des Actionneurs et Capteurs et de leur argument

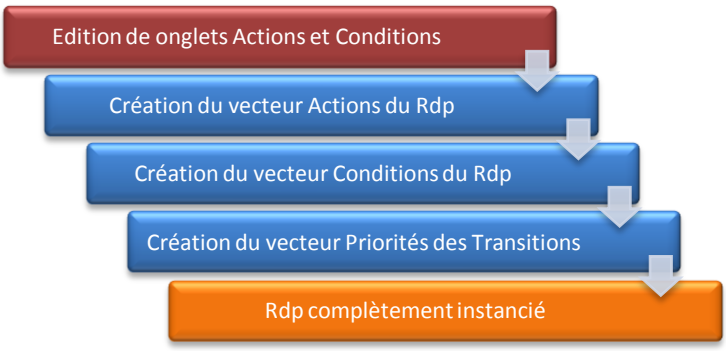
Une fois le fichier importé, l'utilisateur peut éditer la liste des conditions et des actions à sa disposition via les onglets Action et Condition. Il lui suffit pour cela d'ajouter Conditions et Actions à l'aide du bouton associé, et de modifier les champs de données. Les intitulés des conditions et actions qu'il entre sont alors intégrées dans les listes de l'onglet transition. Notons que ceux-ci peuvent être différents de ceux initialement entrés dans Tina, et qu'un même capteur ou actionneur peut apparaitre plusieurs fois dans la liste, avec des arguments et un nom différents. Par exemple, l'utilisateur peut choisir de définir une condition de proximité d'un mur, nommée Prox_Mur, utilisant le capteur ultra son du robot, ainsi qu'une condition distance d'un mur, nommée Dist_Mur, utilisant le même capteur, mais avec un argument différent.



Une fois les onglets des Actions et des Conditions édités, l'utilisateur peut continuer de construire le réseau de pétri, en complétant les transitions détectées dans le fichier importé. Il associe ainsi à chaque transition des actions et des conditions sensibilisantes. En cas de parallélisme, il a également la possibilité d'ajouter des ordres de priorités aux transitions.



Une fois ces trois étapes terminées, les attributs de la classe Rdp sont tous instanciés, et l'on peut procéder à l'exportation.



E) Exportation du Fichier

L'exportation du fichier se fait sur le principe de la « textualisation » de la classe Rdp. L'utilisateur choisit de placer son fichier où il le souhaite. Celui-ci est formaté de manière à être lu de manière correcte par le robot, et se présente sous la forme :

```
4,1,0,0,0
tr:p1*1;1,2,50;p2*1;4,1;1
tr:p3*1;1,2,50;p4*1;0,20;2
tr:p4*1;1,2,50;p1*1;6,1;3
```

La première ligne fait figurer : le nombre de places (4 ici), puis le marquage initial de chacune des places, dans l'ordre (de p1 à p4 donc). Les lignes suivantes correspondent aux transitions, avec dans l'ordre, la place de départ et sa multiplicité, le numéro du capteur associé[1], l'opérateur et la valeur (qui forment alors une condition), la place d'arrivée et sa multiplicité, et enfin le numéro d'actionneur associé à l'action, son argument suivis de la priorité de la transition.

[1] Rappel : on ne désigne dans ce fichier pas les actionneurs et les capteurs par leur nom, mais par un numéro. Il en est de même pour les opérateurs associés à une condition. Voir précédemment pour les références.

3) Joueur

A) Conception

L'équipe a divisé le travail en trois parties de volumes à peu près égaux qui correspondent au découpage conceptuel. On présente donc chacune de ces parties, avant de rappeler le schéma de conception global dans la dernière section.

a) Joueur de Réseau de Pétri

Le joueur de Réseau de Pétri est bâti autour de trois classes. L'une d'elle modélise le joueur lui-même, quand les deux autres modélisent respectivement les transitions et les conditions affectées à ces transitions. **Il n'y a pas de classe représentant les places** ; nous considérions que le marquage pouvait être un attribut du joueur, car aucune méthode ne lui est associée.

On commence par donner la conception de la classe joueur, en Tableau 1 :

Tableau 1 - Classe Joueur

Joueur
marquage :: int[] transitions :: Vector<Transition> sensibilisees :: Vector<Transition>
Constructeurs : void :: Joueur() void :: Joueur(int nbPlaces) void :: Joueur(int nbPlaces,Vector<Transition> transitions,Vector<Transition> sensibilisees) *Méthodes de Classe :* Vector<Transition> Jouer() *Getters* *Setters*

La méthode Jouer est le cœur de tout le code embarqué, c'est elle qui a la charge d'évaluer les transitions, et le ces échéant de les franchir. On s'y attarde donc maintenant en donnant son algorithme. On ne reviendra toutefois sur les particularités que ce choix entraîne que dans Particularités de Fonctionnement. Cet algorithme fait lui-même appel à certaines fonctions, qui ne seront décrites que plus loin, puisqu'appartenant aux classes **Transition** et **Condition**. On considère toutefois leurs noms comme étant suffisamment transparents pour que cela ne perturbe pas la compréhension de l'algorithme.

Algorithme de Jouer :

```
priorité(auxiliaire)=0
Pour i allant de 0 à Taille de transitions faire
        Si transitions(i) est sensibilisée
                Ajouter(transitions(i)) à sensibilisees
                Si transitions(i) est franchissable
                        Si priorité(transitions(i)) > priorité(auxiliaire)
                                auxiliaire=transitions(i)
                        Fin
                Fin
        Fin
Fin
Si priorité(auxiliaire) > 0
        Franchir(auxiliaire)
        Vider sensibilisees
        Retourner sensibilisees
Sinon
        Retourner sensibilisees
Fin
```

Nous pouvons maintenant aborder la conception de Transition, donnée dans le Tableau 2:

Tableau 2 - Classe Transition

Transition
entree :: int[] sortie :: int[] conditions :: Vector<Condition> actions :: Vector<Action> priorite :: int
Constructeurs : void :: Trannsition() void :: Transition(int[] entree, int[] sortie, Vector<Condition> conditions, Vector<Action> actions) *Méthodes de Classe :* void Franchir(int[] marquage) int isFranchissable(int[] marquage) int isSensibilisee(int[] marquage) *Getters* *Setters*

On passe maintenant en revue l'ensemble des fonctions de cette classe. La fonction « isSensibilisee » permet de vérifier que le marquage des places d'entrée d'une transition est supérieur ou égal aux attentes pour franchir la transition. Elle renvoie 1 donc ce cas, et 0 sinon. On a toutefois besoin d'une deuxième fonction pour savoir si la transition est franchissable. En effet, cette

transition sera le plus souvent conditionnée, à la valeur d'un capteur par exemple**. Vérifier le marquage d'entrée n'est donc pas suffisant**. La fonction « isFranchissable » remédie à ce problème, en vérifiant d'abord que la transition est sensibilisée (appel de « isSensibilisee ») puis en regardant l'ensemble des conditions associées. Une fois de plus, cette méthode renvoie 1 si la transition est franchissable et 0 sinon. La méthode « Franchir » vide les places d'entrée et remplit les places de sorties, selon la transition considérée, passée en paramètre.

Le dernier maillon de ce joueur minimal est donc la classe condition, présentée dans le Tableau 3 :

Tableau 3 - Classe Condition

Condition
capteur :: int opérateur :: int seuil :: double robot :: Tribot
Constructeurs : void :: Condition(int capteur, double seuil, int operateur, Tribot robot) *Méthodes de Classe :* int isValidee() *Getters* *Setters*

A partir des données sur le capteur, le seuil et l'opérateur, la méthode « isValidée » reconstruit l'équation d'évaluation de la condition, interroge la valeur capteur par l'intermédiaire du robot et évalue la validité de cette condition. Elle renvoie 1 si la condition est valide, 0 sinon.

b) *Loader*

Le but du Loader est de prélever dans un fichier texte l'ensemble des informations nécessaires à la constitution d'un réseau de Pétri virtuel.

Il est donc composé de deux parties distinctes : une en charge de la lecture (parsage) et l'autre dont le but est la constitution du réseau.

On ne revient pas en détails ici sur la lecture du fichier texte. La seule information pertinente tenant dans le choix du format du fichier texte, elle est détaillée dans la section de ce rapport traitant du traducteur. Le code de parsage est pleinement fonctionnel et se révèle capable d'identifier clairement. On notera que la méthode principale de la classe Loader (à savoir load) prend en charge les exceptions d'entrée-sortie.

c) *Code de Pilotage*

Le code de pilotage fait interface entre le joueur de réseau de Petri et le robot à proprement parler. C'est l'appel des méthodes de la classe Tribot qui permet de récupérer les valeurs des capteurs d'une part et de donner des ordres aux robots d'autres part. Cette classe d'interface est spécifique au modèle de robot que nous avons choisi. Les actions réalisables sont liés à la

configuration du robot et aux associations mécaniques faites sur les axes moteurs. C'est la raison au nom de la classe Tribot et non simplement Robot.

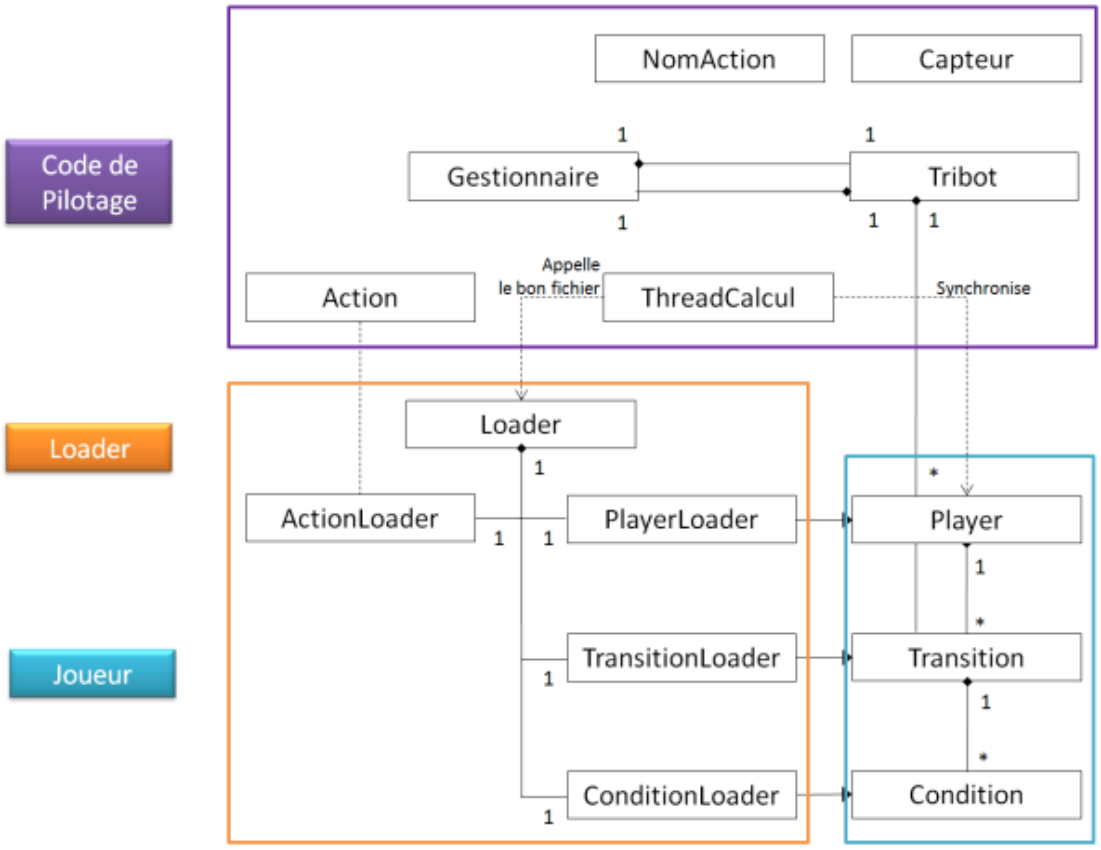
Est associé à la classe Tribot une classe Gestionnaire d'abonnements qui récupère l'ensemble des transitions sensibilisées au moment où le joueur ne peut plus tirer de transitions. Le tribot est alors en charge de rafraîchir ses capteurs et de vérifier grâce à ce gestionnaire si une des transitions sensibilisées est devenue franchissable du fait de nouvelles valeurs capteurs. Si tel est le cas, le gestionnaire informe le joueur de la transition devenue franchissable.

Les retours des méthodes du Tribot permettant d'exécuter des actions ne sont pas bloquants afin que l'exécution d'une action ne bloque pas le déroulement du code. Un tel blocage aurait été contraire au modèle de joueur de réseau de Petri en n'autorisant pas des actions simultanées par exemple. Cependant, l'inconvénient de cette façon de faire est qu'on ne connaît pas le moment de fin d'une action.

En ce qui concerne les valeurs capteurs, une évolution de la classe Tribot pourrait être de récupérer plus d'informations capteur sur le robot. Il serait par exemple possible de connaître la position de chacun des moteurs. On pourrait également estimé la position et l'angle du robot à partir des ordres donnés aux moteurs et des positions angulaires des moteurs.

Enfin, grâce à un timer, on pourrait inclure une information de temps afin de pouvoir implémenter un joueur de réseau de Petri temporisé.

d) Récapitulatif



B) Particularités de Fonctionnement

a) Sans Priorité : Transitions Tirées Jusqu'à Epuisement

Par défaut les transitions sont initialisées à une priorité égale à 1. Cela signifie, vu l'algorithme exposé ci-avant pour le jeu du réseau, qu'une transition validée est tirée jusqu'à épuisement. En d'autres termes, si aucune transition ne dispose d'une priorité supérieure à chaque nouveau jeu on tirera la même transition jusqu'à ce que son marquage ne le permette plus. A titre d'exemple, le réseau utilisé pour les tests génériques – visible en Figure 19 – verra sa transition T_0 tirée c fois si aucune priorité ou condition n'est spécifiée.

b) Résolution des Conflits : Transitions Priorisées

En cas de conflit structurel, il fallait faire en sorte que ce ne soit pas l'ordre de déclaration qui décide seul de quelle transition doit être tirée. C'est la solution pour laquelle une transition peut-être équipée d'une priorité, typée en nombre entier, qui permet de résoudre les conflits. Si le conflit persiste il relève cette fois de l'erreur de conception : le concepteur n'a pas spécifié les bonnes priorités sur les transitions parallèles.

Le lecteur notera qu'une transition vide – ie initialisée sans arguments concernant ses conditions ou actions associées – se voit attribuée la priorité de 0 qui ne correspond à aucun cas réel. Nous avions besoin de cette disposition pour la gestion de l'algorithme de jouer. En effet, la priorité nominale d'une transition étant 1, il fallait que la transition auxiliaire que doit écraser toute transition sensibilisée ait une priorité inférieure. C'est un tour de programmation, et il ne faut rien y voir de plus qu'une commodité.

c) Séquence de Tir de Temps Nul

Nous considérons, comme dans les réseaux de Pétri, qu'une séquence de tir s'effectue en temps nul (nous n'avons pas implémenté les automates/réseaux de Pétri temporisés). Ceci signifie qu'entre deux appels de la fonction jouer on n'actualise pas les valeurs des capteurs. De la sorte des transitions qui n'étaient pas franchissables au début de la séquence de tir ne le deviennent pas en cours de séquence et le principe de franchissement immédiat est respecté. Le lecteur avisé notera qu'il existe effectivement une méthode de rafraichissement des valeurs capteurs dans la classe joueur, appelée au début de la méthode jouer. Cette méthode vise simplement à s'assurer que les valeurs capteurs utilisées dans le joueur sont bien les mêmes que celles utilisées par la classe Tribot. Néanmoins, cette classe et elle seule s'occupe de vérifier et actualiser la valeur physique des quatre capteurs du robot.

L'équipe de développement a ici fait un choix qui lui semblait correspondre le plus à la théorie des réseaux de Pétri. Il est bien entendu possible d'obtenir une méthode d'exécution différente en faisant un choix différent.

d) Réseaux de Pétri non Temporisés

Le programme de Jeu de Réseaux de Pétri présenté dans le cadre de ce travail est non temporisé. L'ajout de conditions temporelles, à l'aide de « counters/timers » par exemple, doit être considéré comme une amélioration à posteriori du code de base proposé dans le cadre de ce projet. On conseille au lecteur de se référer à c) Code de Pilotage, pour consulter les améliorations possibles dans ce domaine.

e) Abonnement aux Transitions Sensibilisées

L'algorithme de jeu proposé dans ce projet est donc relativement simpliste. Toutefois, le souci d'optimisation des opérations embarquées n'a pas été totalement inexistant. La méthode Joueur ::Jouer parcours en effet plusieurs fois la liste des transitions. Dans le cas de mission simples, ce n'est peut-être pas très contraignant. En revanche quand les réseaux grandissent en nombres de places et surtout de transitions, cela peut vite devenir une contrainte. Bien loin de considérer les impératifs de commande en temps réel dans ce travail d'étude, bous avons pourtant eu conscience du fait qu'une optimisation était possible. Et la réponse donnée à ce problème spécifique est l'abonnement aux transitions sensibilisées.

Le concept de l'abonnement est de récolter l'ensemble des transitions sensibilisées (renvoyé par la méthode Joueur ::Jouer) et de tester en priorité ces transitions là quand la valeur des capteurs change. En effet, si le joueur ne franchit plus de transitions de lui-même (état inactif), alors les prochaines transitions à être franchies seront forcément des transitions dont le marquage d'entrée vérifie les conditions Pré, c'est-à-dire ce que nous avons appelé des transitions sensibilisées.

On peut se rendre compte facilement de cette dualité de fonctionnement en observant les modes du joueur sur l'écran de contrôle de la brique *Lejos*. Le mode « actif » renvoie à la méthode Joueur ::Jouer où les transitions franchissables sont franchies selon un *modus operandi* détaillé dans les points précédents. Dans le mode « inactif » en revanche, le programme embarqué met en sommeil le fonctionnement normal du joueur, et vérifie chacune des transitions sensibilisées (et donc abonnées) en fonction des nouvelles valeurs capteurs testées régulièrement. Si les conditions associées à l'une des ces transitions sont réalisées, alors il teste i la transition est toujours sensibilisée puis franchissable. Il la franchit le cas échant et relance la méthode générique Joueur ::Jouer.

C) Procédures de Test

Les tests ont commencé par une vérification générique du fonctionnement du joueur de Réseau de Pétri. Nous avons testé successivement le bon fonctionnement du joueur dans un cas sans condition. Puis nous avons vérifié l'établissement des priorités entre les transitions. Enfin, grâce à un générateur de nombres aléatoires, nous avons pu proposer des valeurs aléatoires pour les capteurs et donc tester le fonctionnement des conditions. Même si ce test ne correspond pas à un cas réel (valeurs des capteurs non continues), il est néanmoins parfaitement suffisant pour vérifier la réaction du programme aux changements de mesure.

On précise que tous les tests ont été conduits sur le Réseau de Pétri suivant, avec un marquage initial de cette facture (en ayant toujours p supérieur à c).

On tient à préciser que ces tests ont été organisées autour du code réel – ie embarqué in fine – mais que seul une partie de celui-ci a été inclus dans cette procédure. En effet, une partie du code est nécessairement dépendant de l'API *Lejos* qui spécifie le fonctionnement du robot pour l'OS Java embarqué. On a donc utilisé seulement les Classes Joueur, Transition, Condition et Emulateur (générateur de nombres aléatoires). Il y a un léger changement entre le code de test et le programme final toutefois. En effet, dans le code embarqué les valeurs des capteurs sont générées par la classe Tribot, tandis qu'elles le sont ici par le classe Emulateur. Il faut donc permuter les deux objets, sachant que leurs méthodes associées portent les mêmes noms (et notamment getValeurs()) pour des raisons évidentes de commodité.

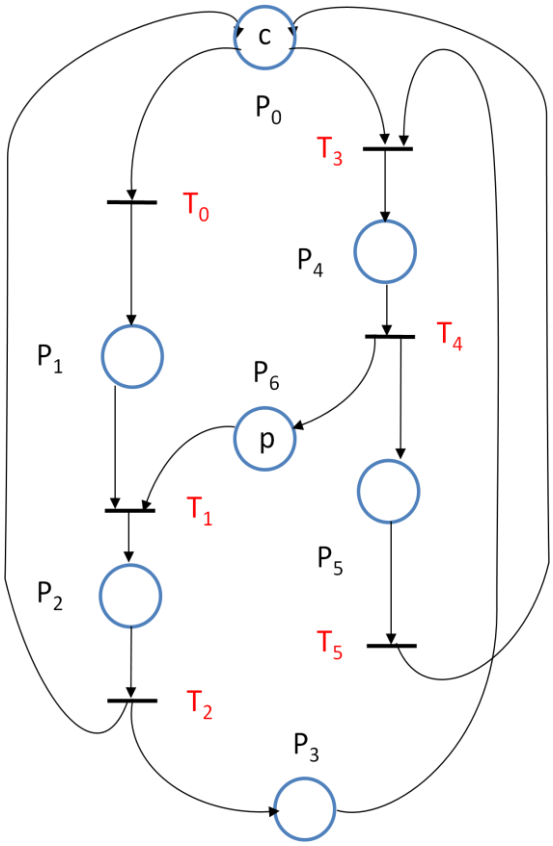


Figure 19 - Réseau de Pétri de Test

Pour information, les places sont numérotées à partir de 0, simplement pour éviter la confusion avec les résultats données par le code, où la numérotation des tableaux commence à 0. Il en va de même pour les transitions.

On vérifie bien sur ce test les particularités de fonctionnement mises en évidence par al conception. En premier lieu on a bien tirage des transitions jusqu'à épuisement quand celles-ci sont non conditionnées et non priorisées. Nous avons pu valider sur ce réseau le système des priorités en affectant une priorité de deux à la transition T_1. Dans ce cas on pouvait constater les franchissements alternatifs de T_0 et T_1. On notera que T_2 ayant le même niveau de priorité que T_0, on tirait après T_0 T_1 de nouveau ceci provoquant le remplissage de la place P_1. Il est bien évidemment possible de reproduire le test avec une priorité de trois affectée à T_2, ce qui fonctionne également. On ne peut toutefois pas faire ici l'inventaire complet des tests unitaires effectués. On a également vérifié le principe de réaction aux conditions en affectant plusieurs de ces transitions de conditions portant sur des nombres générés aléatoirement. Ici encore le test a été concluant, nous décidant à passer à une épreuve de plus grande envergure en embarquant un réseau de Pétri minimal, comme décrit en Figure 20.

Pour simplifier les choses nous avons simplement choisi un marquage initial c=1. Nous avons souhaité tester plusieurs choses. En premier lieu la bonne réaction de l'ensemble des capteurs. Nous souhaitions également valider les différents types d'équations possibles (égalité, compraisons). Il nous fallait ensuite vérifier que plusieurs conditions superposées sur une même transition étaient compatibles avec le code tel que nous l'avons écrit. Enfin, il fallait vérifier le bon déclanchement des

actions. C'est la raison pour laquelle nous avons choisi le jeu d'actions et de conditions présentées dans le.

Tableau 4 - Liste des Transitions de Test

Transition	Conditions	Actions
1	Egalité Tactile + Seuil Ultrason	Avancer
2	Seuil Sonore	Reculer
3	Egalité Tactile	Ouvrir Pince
4	Seuil Photosensible	Fermer Pince

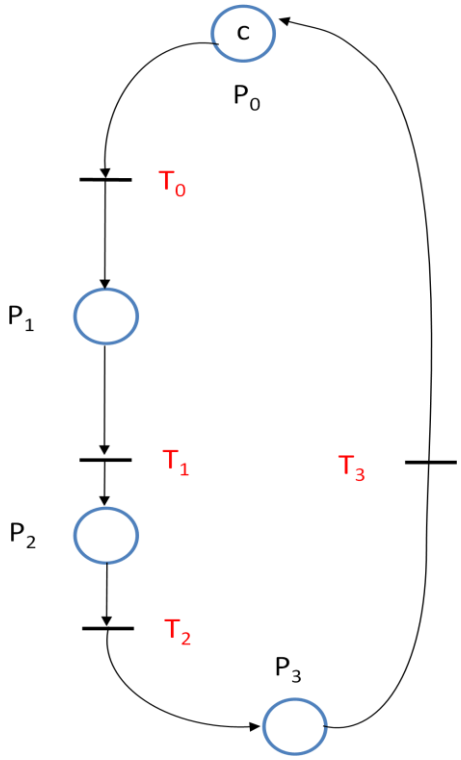


Figure 20 - Réseau de Pétri de Test Embarqué

On n'a pas testé au cours de cette phase les codes destinés à faire tourner le robot, dans la mesure où ils ne sont que des combinaisons linéaires des ordres AVANCER et RECULER. Les pinces sont certes elles aussi pilotées par des moteurs du même type. On aurait dans l'absolu se passer de tests pour cette action aussi. Nous avons toutefois souhaité vérifier la bonne tenue du mécanisme ainsi que le choix des valeurs de butée.

On notera que cette phase de test simpliste nous a également permis de mieux nous rendre compte du fonctionnement des capteurs. C'est vrai pour le capteur sonore qui connaît quelques troubles à l'initialisation. Cela l'est également pour le photosensible dont les niveaux pratiques nous étaient inconnus avant de procéder à l'ensemble de ces tests.

D) Résultats

Le résultat principal de cette partie est naturellement un joueur de Réseau de Pétri « embarquable » sur le robot Lego mis à notre disposition pour cette étude.

Il permet, en dépit de son manque de maturité et des nombreuses améliorations possibles, le jeu de missions simples et fait la démonstration de la faisabilité d'une commande à base de Réseaux de Pétri.

Toutefois, le code de jeu pur dépasse largement le cadre de ce projet et peut être réutilisé dans d'autres applications fondées sur le langage JAVA.

4) Conclusion

Le travail réalisé ici nous a permis de découvrir les tenants et les aboutissants de quelques aspects liés à l'utilisation de réseaux de Pétri pour la réalisation d'un objectif bien fixé, comme les problèmes de parallélisme ou le caractère instantané des transitions d'un réseau non temporisé. Les perspectives ouvertes par ce travail sont nombreuses. Citons par exemple la possibilité d'intégrer dans notre joueur une partie destinée à prendre en compte les réseaux temporisés, et de l'améliorer afin de prendre en compte des missions plus complexes avec des Réseaux de Pétri plus grands. Nous avons également dans notre travail pris pour principe le contrôle par l'utilisateur de l'interface de traduction, qui pourrait sur un projet plus étendu dans le temps être automatisée.