RAPPORT DE PROJET

HMEE303 : ARCHITECTURE DE CONTRÔLE

Pilotage d'une cellule d'assemblage robotisée

Auteurs :
MM. ARTUSO et SANCHEZ

Tuteur : M. NAVARRO
Doctorant équipe IDH (LIRMM)

Année 2015 - 2016

Table des matières

Table des figures

1 Introduction

Dans le cadre d'un projet concernant les architectures de contrôle, M. NAVARRO, Doctorant au LIRMM, équipe IDH, nous a proposé d'optimiser la commande d'une cellule d'assemblage robotisée : un robot de manipulation (KUKA LWR 4) est situé dans une chaîne d'assemblage.

Le contexte est le suivant : le KUKA LWR 4 a accès à un tapis d'approvisionnement de pièces de trois types, un tapis d'évacuation et un poste d'assemblage. Le robot doit assembler des ensembles de trois pièces différentes qui arrivent aléatoirement sur le tapis d'approvisionnement. Notre rôle est d'améliorer le système existant en optimisant la gestion des actions à réaliser en fonction de la prise des pièces et des déplacements du robot.

Le cahier des charges mis en place par M. NAVARRO, est défini autour de trois objectifs. Le premier est de réaliser la modélisation du système en réseaux de Petri afin d'avoir une vision plus concrète de notre système robotique. Pour cela nous avons utilisé le logiciel «TINA» (TIme petri Net Analyzer) développé par le L.A.A.S.

Ensuite, le second objectif est de porter intérêt aux caractéristiques des réseaux de Petri créés afin de vérifier le respect par exemple des règles de vivacité et de réinitiabilité, indispensables à un tel système.

Enfin, le dernier objectif est de transcrire les différents réseaux de Petri en langage C++, sous la forme de fonctions de type «machine à états finis». La programmation C++ du système est ensuite portée sous le logiciel V-REP (Virtual Robot Experimentation Plateform) de COPPELIA ROBOTICS afin de visualiser le respect du fonctionnement du pilotage de la cellule d'assemblage robotisée.

2 Modélisation sous réseaux de Petri

Pour la modélisation du système complet, plusieurs réseaux de Petri ont été réalisés pour découper celui-ci en plusieurs sous-systèmes, généralement en fonction de ses différents composants. Nous avons crées les réseaux de Petri suivants :

Le tapis 1, **le tapis 2**, **la caméra**, **l'assembleur** et **le robot** qui représentent les réseaux de Petri «Esclaves» (informations à envoyer, tâches à réaliser).

Le contrôleur, **la gestion du stock** et **la position du robot** qui représentent les réseaux de Petri «Maîtres» , ils sont considérés comme l'intelligence du système (ordres à envoyer, informations à recevoir).

Nous détaillons ci-dessous le rôle particulier de chacun de nos réseaux de Petri : dans un premier temps, sont présentés les réseaux de Petri «Esclaves», puis les réseaux de Petri «Maîtres» dans un second temps.

2.1 Les réseaux de Petri «Esclaves»

Pour chacun de nos réseaux de Petri, nous listons de façon exhaustive les informations nécessaires à recevoir et les ordres ou informations à envoyer aux autres réseaux. La vivacité et les autres caractéristiques des réseaux de Petri sont abordées en fin de section.
Commençons par étudier le réseau de Petri du tapis d'approvisionnement des pièces (Voir Figure 1) :

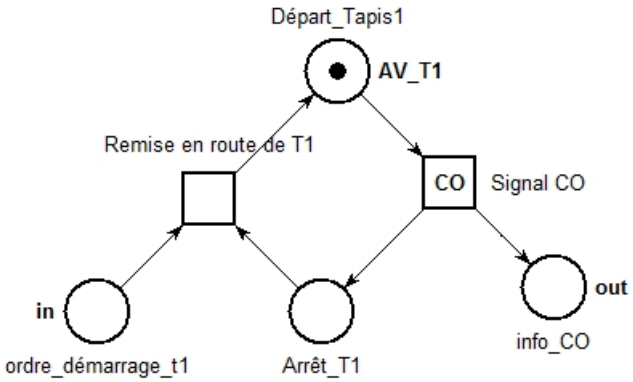

FIGURE 1 – Réseau de Petri complet du tapis d'approvisionnement.

Description : au démarrage du programme, le tapis d'approvisionnement est en marche, si celui-ci reçoit le «signal CO», il informe qu'il a reçu le signal et s'arrête. Étant arrêté, il ne peut redémarrer qu'à condition d'avoir reçu le signal «ordre_démarrage_t1».

Son rôle est donc très simple : il s'agit d'amener les pièces à assembler à la portée du robot et de s'arrêter lorsqu'une pièce arrive en butée du tapis.

— **Les entrées/sorties du tapis d'approvisionnement :**
[Réception] : ordre_démarrage_t1.
[Envoi] : info_CO.

Le tapis d'évacuation présente quelques similarités (Voir Figure 2) :
Description : au démarrage du programme, le tapis d'évacuation est en

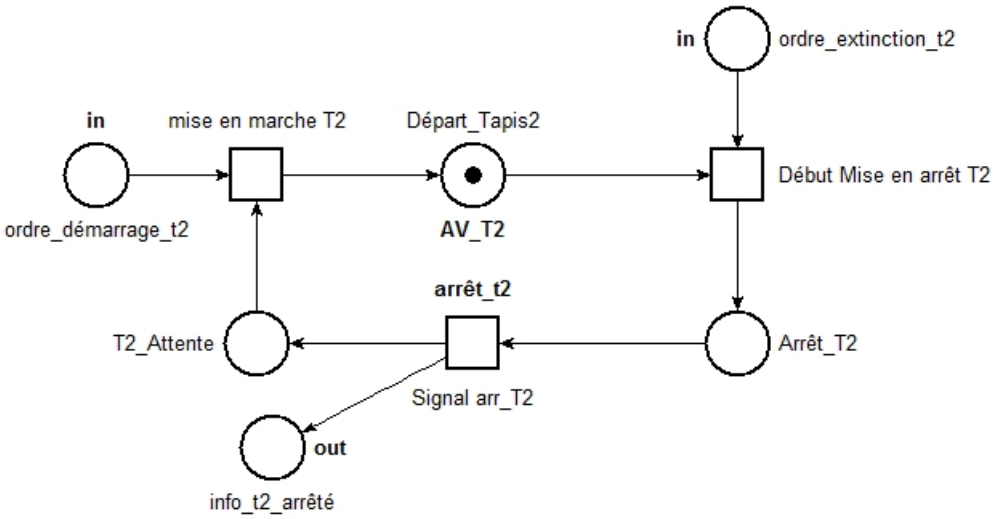

FIGURE 2 – Réseau de Petri complet du tapis d'évacuation.

marche, si celui-ci reçoit le signal «ordre_extinction_t2», il débute sa mise en arrêt et s'arrête après un certain temps. Une fois arrêté, il transmet un signal informant qu'il est arrêté. Il ne peut redémarrer qu'à condition d'avoir reçu le signal «ordre_démarrage_t2».

Son rôle est de s'arrêter avant la pose d'une pièce à évacuer par le robot et de redémarrer quand le robot a fini la pose.

— **Les entrées/sorties du tapis d'évacuation :**
[Réception] : ordre_démarrage_t2, ordre_extinction_t2.
[Envoi] : info_t2_arrêté.

Passons maintenant à des réseaux un peu plus évolués tels que la caméra (Voir Figure 3).

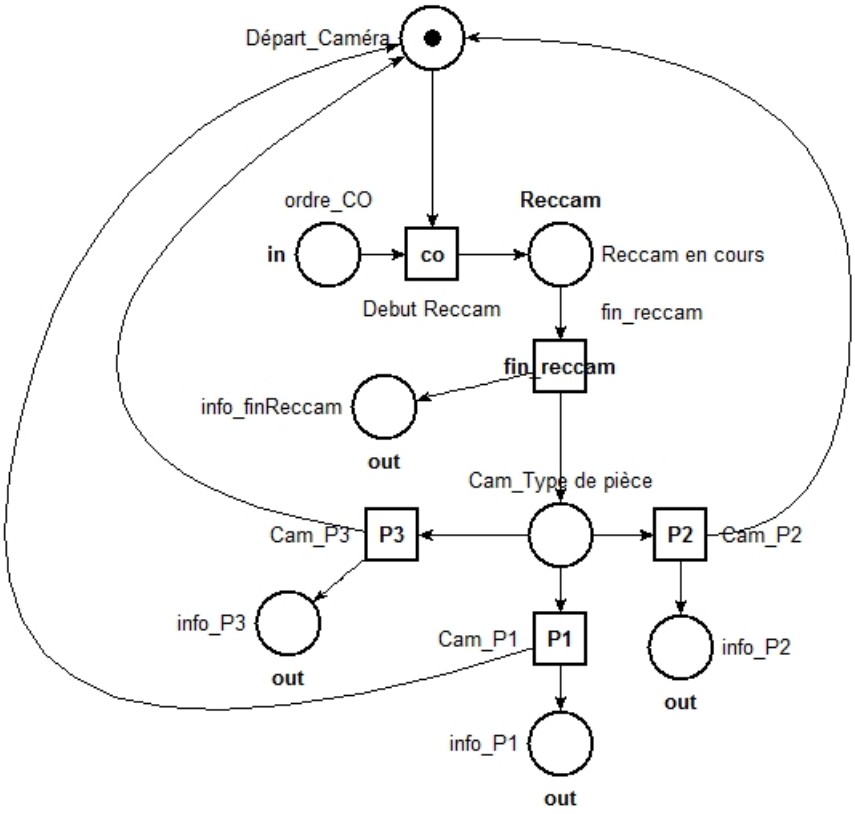

FIGURE 3 – Réseau de Petri complet de la caméra.

Description : au démarrage du programme, la caméra attend de recevoir le signal «ordre_CO» . À sa réception, elle réalise l'action de reconnaissance de la pièce, une fois faite, elle transmet un signal informant que la reconnaissance est finie puis elle indique par un autre signal le type de pièce qui a été détecté.

Son rôle est de déterminer le type de la pièce présente en butée du tapis d'approvisionnement et de transmettre cette information.

— **Les entrées/sorties de la caméra :**
[Réception] : ordre_CO.
[Envoi] : info_finReccam, info_P1, info_P2, info_P3.

Détaillons l'architecture de l'assembleur (Voir Figure 4) :

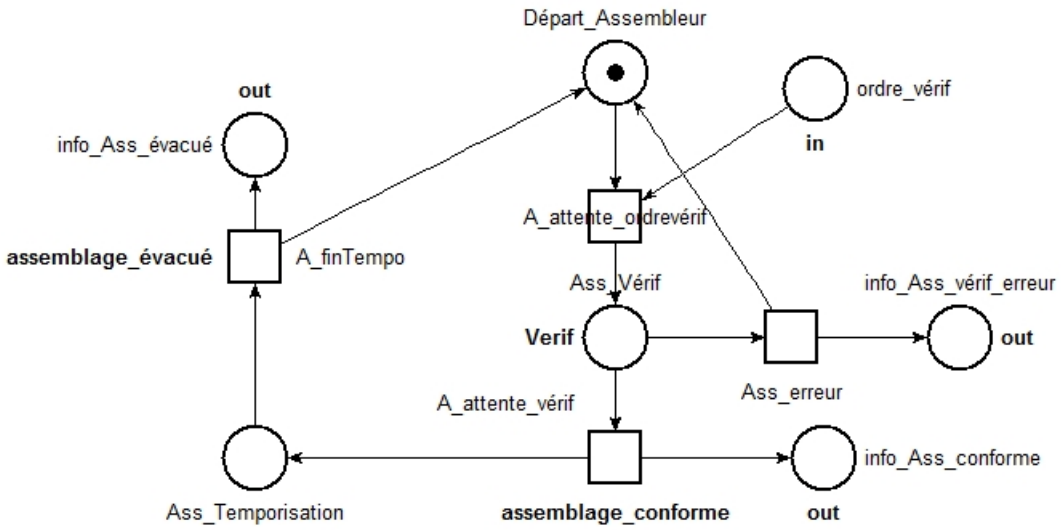

FIGURE 4 – Réseau de Petri complet de l'assembleur.

Description : au démarrage du programme, l'assembleur attend de recevoir le signal «ordre_vérif» . À sa réception, il vérifie son contenu (les pièces qu'il possède), une fois la vérification faite, deux actions sont possibles. Soit le contenu n'est pas conforme et l'assembleur envoie un signal d'erreur. Soit le contenu est conforme et l'assembleur transmet un signal informant la conformité de l'assemblage. Après, une petite temporisation, il évacue l'assemblage, transmet un signal informant que l'évacuation a été réalisée puis retourne dans son état initial «Départ_Assembleur».

Son rôle est de vérifier si le contenu de l'assemblage est conforme, transmettre cette information et doit aussi assurer l'évacuation en cas de conformité de l'assemblage.

— **Les entrées/sorties de l'assembleur :**
[Réception] : ordre_vérif.
[Envoi] : info_Ass_évacué, info_Ass_conforme, info_Ass_vérif_erreur [1].

1. Information supprimée dans le réseau de Petri final car bloquant la simulation

Le robot est le plus élaboré des réseaux de Petri «Esclaves» (Voir Figure 5) :

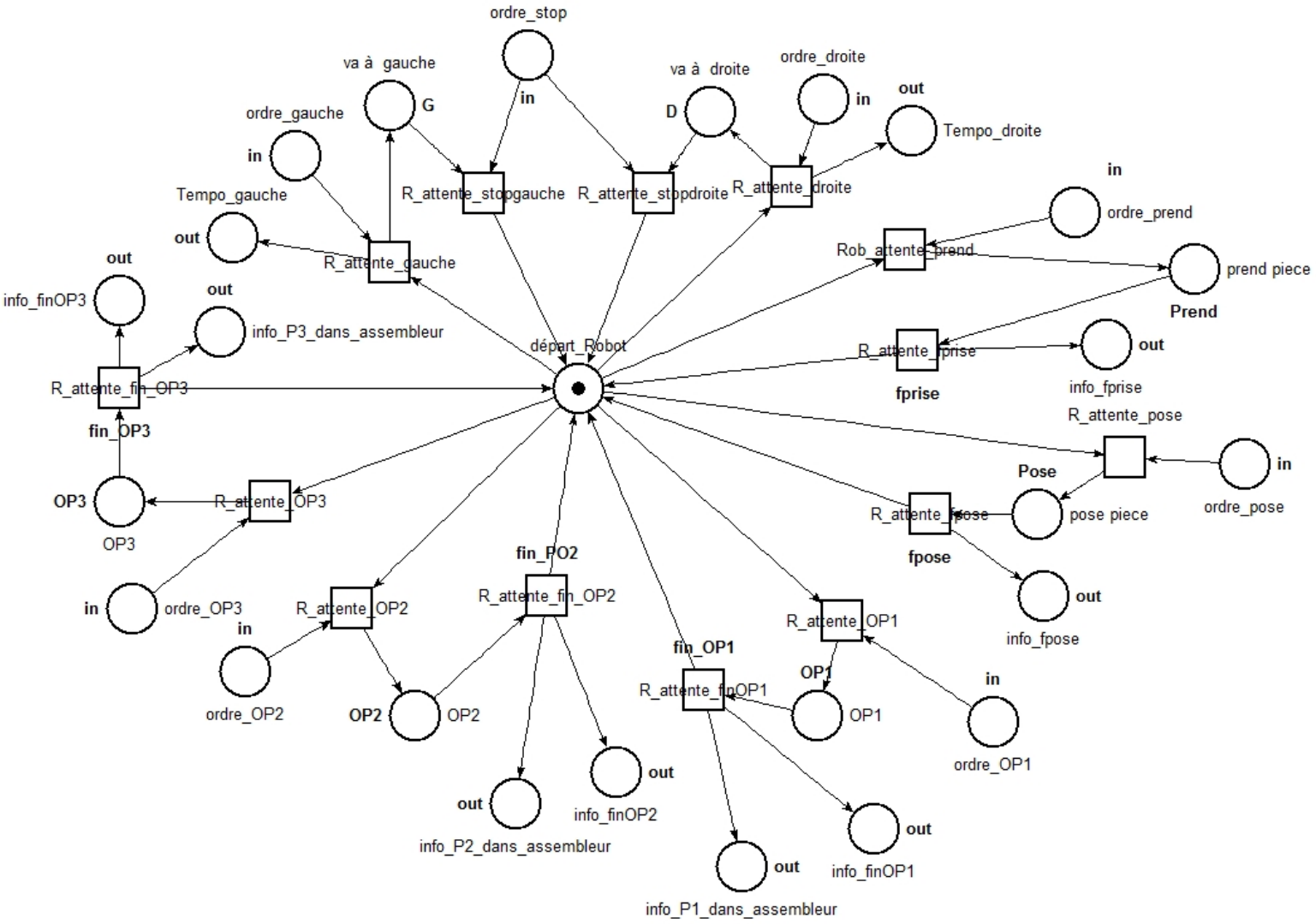

FIGURE 5 – Réseau de Petri complet du robot.

Au vu de la multitude des possibilités d'actions, nous ne détaillons pas ici le réseau mais seulement le principe. Au démarrage le robot est dans son état initial «Départ_Robot» et attend de recevoir un ordre quelconque pour réaliser l'action correspondante. Une fois l'action réalisée, il transmet un signal stipulant que l'action est terminée puis il retourne dans son état initial, attendant un nouvel ordre.

Son rôle est de prendre la pièce en butée du tapis d'approvisionnement, de l'amener sur l'assembleur ou sur le tapis d'évacuation selon les cas et d'informer certains réseaux à chaque fin d'action.

— **Les entrées/sorties du robot :**
[Réception] : ordre_gauche, ordre_droite, ordre_OP1, ordre_OP2, ordre_OP3, ordre_stop, ordre_prend, ordre_pose.
[Envoi] : info_finOP1, info_finOP2, info_finOP3, info_fpose, info_fprise, Tempo_gauche, Tempo_droite.

La présentation des réseaux de Petri «Escalves» étant faite, nous allons maintenant parler de leurs caractéristiques. Pour les déterminer, nous avons utilisé l'outil de simulation disponible sur le logiciel TINA. Pour cela, il est nécessaire d'enlever toutes les entrées et sorties des réseaux de Petri : il s'agit des versions "simplifiées" disponibles dans l'archive.

Les caractéristiques de tous nos réseaux de Petri «Esclaves» sont les suivantes :

— **«vivant»** : Sans blocage, depuis le marquage initial, toutes les transitions peuvent être tirées par une séquence de franchissement.

— **«sauf»** : Aucune des places n'accumule plus de 1 jeton à tout moment, aussi nommé **«1-borné»**

De part ces deux propriétés, le réseau de Petri est dit **«conforme»**.

— **«propre»** : le réseau est réinitialisable (ou réversible). C'est-à-dire que depuis le marquage initial on peut, par un séquence de franchissement, atteindre une nouvelle fois le marquage initial.

De part ces trois propriétés, le réseau de Petri est dit **«bien formé»**.

— **«sans bloquage»** : aucune étape n'est un marquage puits : à tout moment le réseau peut évoluer d'un état de marquage vers un nouvel état.

— **«pur»** : il n'existe pas de transition ayant une place d'entrée qui soit à la fois une place de sortie de cette transition.

— **«exempt d'arcs à pondération»** : tous les arcs possèdent la même pondération unitaire.

2.2 Les réseaux de Petri «Maîtres»

Commençons par le plus simple, la position du robot (Voir Figure 6) :

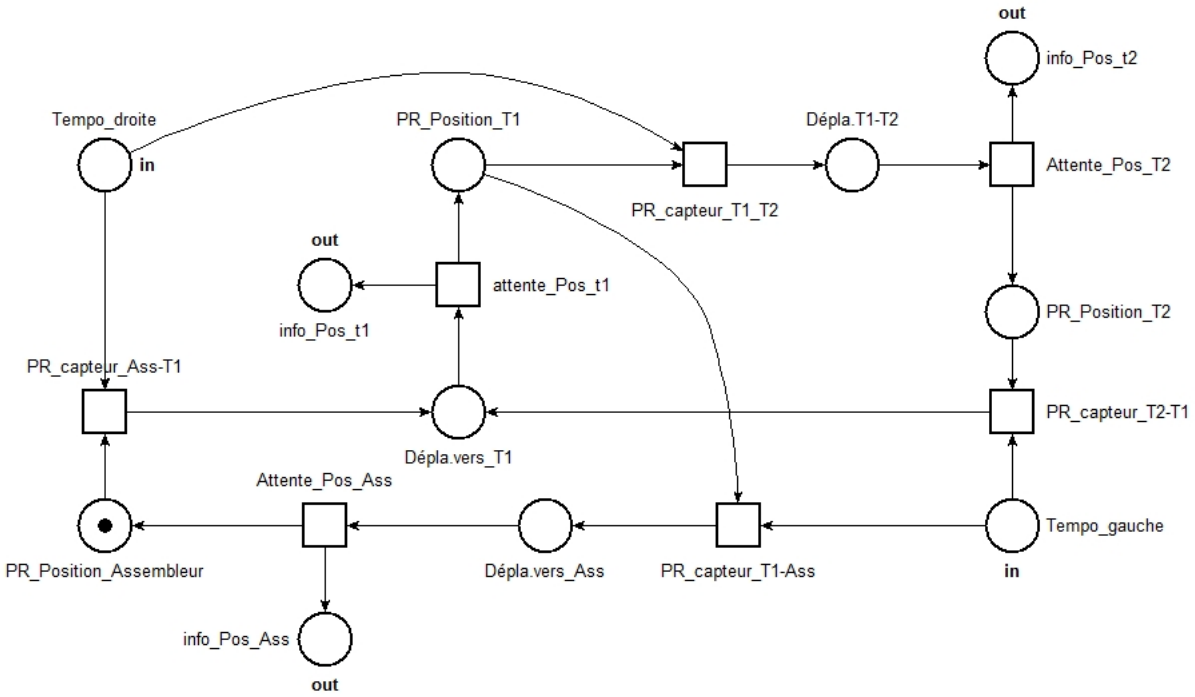

FIGURE 6 – Réseau de Petri complet de la position du robot.

Le robot possède des capteurs lui indiquant sa position actuelle, il peut être :
- devant l'assembleur (info_Pos_Ass)
- devant le tapis d'approvisionnement (info_Pos_t1)
- devant le tapis d'évacuation (info_Pos_t2)
- en cours de déplacement (Avant de franchir la transition)

Ces repères étant nécessaires à la réalisation de certaines actions, il est important de fournir ces informations au contrôleur. L'unique but de ce réseau de Petri est donc d'informer le contrôleur sur la position actuelle du robot.

— **Les entrées/sorties de la position du robot :**
[Réception] : ordre_gauche, ordre_droite, tempo_gauche, tempo_droite.
[Envoi] : info_Pos_t1, info_Pos_t2, info_Pos_Ass.

Passons maintenant à la gestion du stock (Voir Figure 7) :

Avant toute chose, il est important de stipuler que nous avons choisi, en vue « d'optimisation », de remplir l'assembleur en fonction de ses places libres et non pas dans un ordre prédéfini. C'est-à-dire : initialement le robot amène les pièces vers l'assembleur dans l'ordre : P1, P2 puis P3 mais ces pièces arrivent aléatoirement sur le tapis d'approvisionnement (Tapis 1), à chaque fois que la pièce détectée n'est pas la bonne il faut l'évacuer et ce même si l'emplacement de cette pièce dans l'assembleur est disponible, c'est donc une grosse perte de temps.

Nous avons donc décidé de mettre en place un système de **gestion du stock** pour pallier ce défaut. Nous avons choisi d'amener la pièce vers l'assembleur si la place, qui lui est réservée, est disponible et ce quel que soit l'ordre d'arrivée des pièces sur le tapis d'approvisionnement.

Désormais, il faut faire attention aux pièces que l'on a déjà prises et s'en souvenir. De manière à ordonner au robot d'évacuer la pièce disponible sur le tapis d'approvisionnement si on la possède déjà sur l'assembleur ou de la lui apporter s'il la lui manque.

Le rôle de la gestion du stock est de fournir les informations au **contrôleur** concernant les actions futures du robot en fonction de la pièce disponible sur le tapis d'approvisionnement et des places disponibles sur l'assembleur.

— **La gestion du stock :**
[Réception] : info_finOP1, info_finOP2, info_finOP3, info_Ass_évacué, info_Ass_conforme, info_P1, info_P2, info_P3
[Envoi] : ordre_CO, Evacuation, Assemblage, ordre_vérif, P1, P2, P3.

Le dernier des réseaux de Petri «Maîtres» est le contrôleur, le grand superviseur du système (Voir Figure 8). Il est le lien entre les différents corps du système.

Le rôle du **contrôleur** est de récolter les différentes informations nécessaires aux actions et d'émettre les ordres et les informations attendus par les différents corps du système.

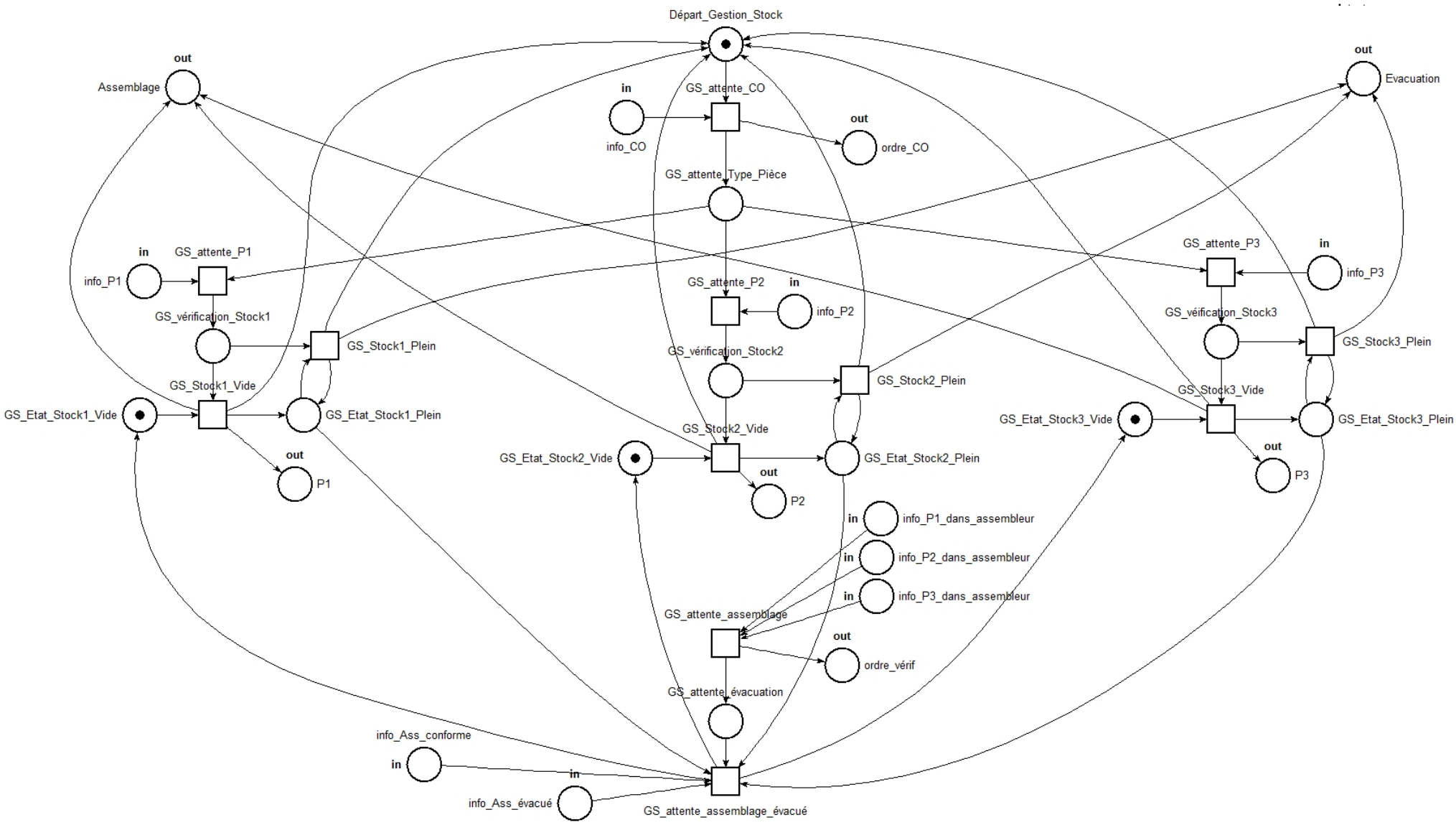

FIGURE 7 – Réseau de Petri complet de la gestion du «stock» : présence de pièce en assembleur.

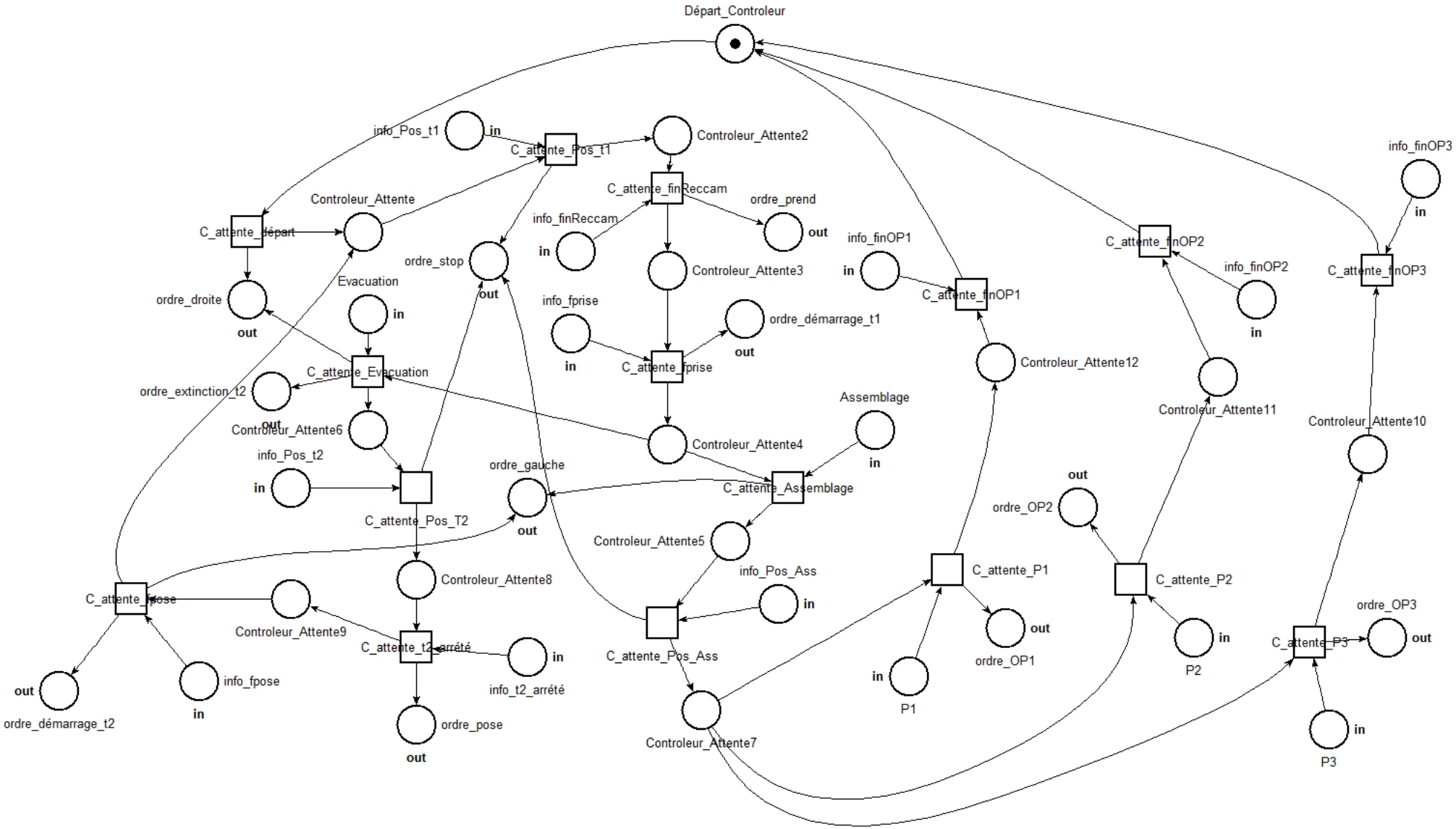

FIGURE 8 – Réseau de Petri complet du contrôleur.

— **Les entrées/sorties du contrôleur :**
[Réception] : info_finOP1, info_finOP2, info_finOP3, info_Pos_t1, info_Pos_t2, info_Pos_Ass, info_finReccam, P1, P2, P3, info_fprise, info_fpose, Assemblage, Evacuation, info_t2_arrêté.
[Envoi] : ordre_gauche, ordre_droit, ordre_OP1, ordre_OP2, ordre_OP3, ordre_stop, ordre_prend, ordre_pose, ordre_démarrage_t1 ordre_démarrage_t2, ordre_extinction_t2 .

Avec toutes ces entrées et sorties, le contrôleur est connecté à presque tous les réseaux de Petri (sauf l'assembleur) ce qui en fait le réseau le plus complexe, notamment lors de sa programmation en C++.
La présentation des réseaux de Petri «Maîtres» étant terminée, nous allons maintenant parler de leurs caractéristiques. Comme pour les réseaux «Esclaves», le logiciel TINA a été très utile.

Les caractéristiques de tous nos réseaux de Petri «Maîtres» sont les suivantes :

— **«vivant»** : Sans blocage, depuis le marquage initial, toutes les transitions peuvent être tirées par une séquence de franchissement.

— **«sauf»** : Aucune des places n'accumule plus de 1 jeton à tout moment, aussi nommé **«1-borné»**

De part ces deux propriétés, le réseau de Petri est dit **«conforme»**.

— **«propre»** : le réseau est réinitialisable (ou réversible). C'est-à-dire que depuis le marquage initial on peut, par un séquence de franchissement, atteindre une nouvelle fois le marquage initial.

De part ces trois propriétés, le réseau de Petri est dit **«bien formé»**.

— **«sans bloquage»** : aucune étape n'est un marquage puits : à tout moment le réseau peut évoluer d'un état de marquage vers un nouvel état.

— **«pur» :**[2] Il n'existe pas de transition ayant une place d'entrée qui soit à la fois une place de sortie de cette transition.

— **«exempt d'arcs à pondération»** : tous les arcs possèdent la même pondération unitaire.

2. Cela n'est vrai que pour la position du robot et le contrôleur. La gestion du stock n'est pas un réseau de Petri «pur» vu que par trois fois (GS_Stock1_Plein, GS_Stock2_Plein, GS_Stock3_Plein) on a des transitions qui ont respectivement la même place en entrée et en sortie.

3 Traduction du réseau de Petri au C++

Dans cette partie, nous nous intéressons à la façon choisie de transcrire notre réseau de Petri : comment passer de sa représentation graphique sous Tina à son implémentation en langage C++ ?

3.1 Du Petri à la programmation multitâche

Notre représentation en réseaux de Petri du système permet une visualisation plus facile des différentes actions possibles et des instructions nécessaires pour les réaliser.

En programmation, les actions sont définies comme des tâches (une tâche est un flot d'exécution qui est traduite par une séquence de code) et les instructions comme des signaux (un signal est un message envoyé à un processus).

Notre cellule d'assemblage robotisée est composée de plusieurs systèmes pouvant chacun d'eux exécuter une tâche ou plus. Nous avons choisi la programmation multitâche pour les raisons suivantes :
- L'implémentation d'un tel programme à l'aide d'un traitement synchrone serait trop long.
- Certains de nos systèmes ont une attente à durée indéterminée (ex : attente du signal A pour réaliser l'action E).
- Nous avons deux systèmes de contrôle qui transmettent les informations en fonction de l'avancement des autres systèmes.

La programmation multitâche est composée de deux familles : les processus et les threads. Leur principale différence se situe au niveau de la mémoire et au moyen d'y accéder :
- Les processus ont leur propre espace d'adressage mémoire et doivent, pour communiquer, utiliser des moyens spécifiques tels que les "pipes" ([fr.] tuyaux).
- Les threads issus d'un même processus partagent la même zone de mémoire ce qui rend la communication entre threads plus facile mais plus périlleuse en cas de modification simultanée d'une variable locale.

Nous avons choisi de programmer notre cellule d'assemblage robotisée avec des threads pour leur facilité de communication. L'utilisation de plusieurs threads est plus représentative de notre système comparée à l'utilisation de processus (Père et X fils) en "forkant" [3] notre programme.

3. néologisme : utilisation de la commande fork()

3.2 Méthodologie de programmation

Concernant le codage, la lecture du code fourni est sûrement plus parlante, néanmoins nous détaillons ici les points clés de notre programme.

Tout d'abord, chaque système dispose de son propre fichier cpp et de son header. Cela afin de déboguer, en cas d'erreur, plus facilement le programme en réduisant l'espace de recherche de la faute. Mais également pour ne remplacer qu'une partie du programme total si l'on doit changer de robot par exemple.

De plus, chaque système a été décrit de la façon suivante :
- Appels aux headers et autres libraires
- Énumération des différents états possibles
- Fonction des tâches à réaliser en fonction de l'état courant.

Et ce de façon systématique pour suivre une même logique de codage.

Enfin, nous disposons d'un fichier principal contenant l'appel aux librairies, la déclaration des threads avec leur fonction associée et le lancement de la simulation. Ainsi qu'un fichier header "struct" contenant la structure dans laquelle est définie tous les signaux que nous utilisons dans les différents programmes.

3.3 Description complète de la programmation d'une tâche

Nous détaillons ici le fichier «assembleur.cpp» à titre d'exemple :

```
#include <iostream>
#include <thread>
#include "simulator.h"
#include "struct.h"
#include "assembleur.h"
```

On inclut les librairies : **iostream** pour apporter des fonctions C++ telles que les «cout», **thread** pour la programmation multitâche. **simulator.h**, **assembleur.h**, **struct.h** sont les headers contenant la déclaration des fonctions, des classes et de la structure utilisée pour les signaux.

```
using namespace std;
```

Cela permet de s'affranchir de l'obligation de préfixer par *std* :: tous les éléments issus de la bibliothèque standard de C++.

```
enum EtatsAssembleur {
    DepartAssembleur,
    Verification,
    Temporisation
};
```

Il s'agit là d'une énumération «EtatsAssembleur», on crée en réalité une machine à états qui possède 3 états différents : DepartAssembleur, Verification et Temporisation. Le programme ne peut être que dans une seul état à la fois. C'est le lien direct avec nos réseaux de Petri, détaillés en sections 2.1 et 2.2.

Voici la construction de la fonction préposée à l'assembleur :

```
void tache_assembleur(struct signaux *sig) {
    EtatsAssembleur etat = DepartAssembleur;
    while(1)
    {
            //switch entre les différents états de la fonction
    }
}
```

La fonction, ne retournant aucune valeur, peut être déclarée de type «void». Elle nécessite en paramètres les signaux provenant de la structure «struct» disposée dans le header «struct.h» vu plus haut. Au démarrage, on initialise la machine à état à l'état «DepartAssembleur». On rentre ensuite dans la boucle principale de la fonction («while(1)») que nous détaillons ci-dessous.

```
switch(etat)
{
    case DepartAssembleur:
        sig-> ordre_verif.wait();
        etat = Verification;
    break;
```

Le switch...case est une alternative plus évoluée à la fonction conditionnelle **if**, le programme va directement au cas correspondant. Le break permet de sortir du «cas courant».

Si la fonction est dans l'état «DepartAssembleur», elle attend («wait()») d'avoir reçu l'ordre_verif, si ce signal est reçu elle passe en état «Verification».

```
case Verification:
    sim.set_Verif(true);
    if(sim.read_assemblage_conforme())
    {
        etat = Temporisation;
        sig-> info_Ass_conforme.notify();
    }
break;
```

Si la fonction est dans l'état «Verification», en mettant le booléen à «true», on démarre la tâche de vérification de l'assemblage. Si on reçoit le signal affirmant l'assemblage comme conforme, la fonction passe en état «Temporisation».

La fonction transmet(«notify()») également un signal stipulant que grâce à la vérification on a déterminé que l'assemblage est conforme. Ce signal est transmis au thread qui en a besoin (à savoir le contrôleur).

```
case Temporisation:
    sim.set_Verif(false);
    while(not sim.read_assemblage_evacue());
    etat = DepartAssembleur;
    sig-> info_Ass_evacue.notify();
break;
}
```

Si la fonction est dans l'état «Temporisation», en mettant le booléen à «false», on éteint la tâche de vérification de l'assemblage.

Tant qu'on lit un résultat faux (booléen) du signal «assemblage_evacue», on reboucle immédiatement. Cette action n'est pas optimale puisse qu'elle oblige le thread à travailler constamment mais elle est pour nous nécessaire en cas d'une attente indéfinie du signal désiré.

Lorsque le signal est reçu, la fonction passe en état «DepartAssembleur» puis transmet un signal affirmant que l'assemblage a bel et bien été évacué. Ce signal est transmis à la gestion du stock pour remettre à zéro les compteurs des stocks pleins afin de commencer un nouveau cycle.

4 Bilan et perspectives

Les travaux réalisés lors de ce projet nous ont permis de faire nos premiers pas en matière d'optimisation de système et d'éprouver une réelle satisfaction lorsque le robot a suivi nos instructions.

Plus humainement, ce projet a également été l'occasion de réaliser un travail collaboratif ce qui permet de nous détacher un peu de notre statut d'étudiant et de mieux nous préparer à la vie active.

Un des buts actuels de l'optimisation des architectures de contrôle ou de la robotique en général, est d'être toujours plus rapide, d'avoir un meilleur rendement.. Mais il s'agit aussi de donner plus d'autonomie aux robots afin de limiter l'intervention de l'Homme dans des tâches répétitives comme l'assemblage par exemple.

À ce sujet, nous pouvons émettre quelques pistes quant aux améliorations qui pourraient être mises en place sur notre projet :

— Pour une facilité de lecture de la console en cas de débogage par exemple, il serait envisageable d'y protéger l'écriture pour ne pas que les threads puissent y écrire simultanément.

— Pour des notions d'économies énergétiques, il n'est pas obligatoire que le tapis d'évacuation soit toujours en marche, il suffit de faire avancer la pièce sur le tapis suffisant loin pour qu'elle ne gène pas la pose de la future pièce évacuée.

— D'un point de vue de configuration on pourrait penser que la fin du tapis d'évacuation débouche sur le tapis d'approvisionnement, vu que les pièces évacuées n'étaient juste pas celles attendues par le robot à un certain instant mais elles pourront très bien l'être à un autre instant (au prochain cycle d'assemblage par exemple).