Utilisation d'Esterel dans un contexte asynchrone : une application robotique

Eve Coste-Maniere

INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1139

Programme 6
Robotique, Image et Vision

UTILISATION D'ESTEREL DANS UN CONTEXTE ASYNCHRONE : UNE APPLICATION ROBOTIQUE

Eve COSTE-MANIERE

Décembre 1989

RR-1139

Utilisation d'ESTEREL dans un contexte asynchrone : Une application robotique.

Eve Coste-Manière

Ecole des Mines de Paris
Centre de Mathématiques Appliquées
Sophia Antipolis 06565 Valbonne Cedex
eve@cma.cma.fr

Novembre 1989

Résumé : La complexité croissante des applications robotiques nécessite l'utilisation d'outils logiciels et matériels puissants autorisant l'exécution d'actions variées selon un ordonnancement précis.

Nous utilisons les caractéristiques offertes par le langage ESTEREL pour gérer le séquencement d'un scénario d'assemblage. ESTEREL est un langage de programmation de haut niveau, synchrone et paralléle, de style impératif. Nous développons plusieurs programmes illustrant les différents avantages qu'offre un tel langage pour la robotique : modularité, parallélisme, déterminisme associé à l'hypothése de synchronisme, gestion de la communication par signaux, intégration facile des traitements d'exception ...

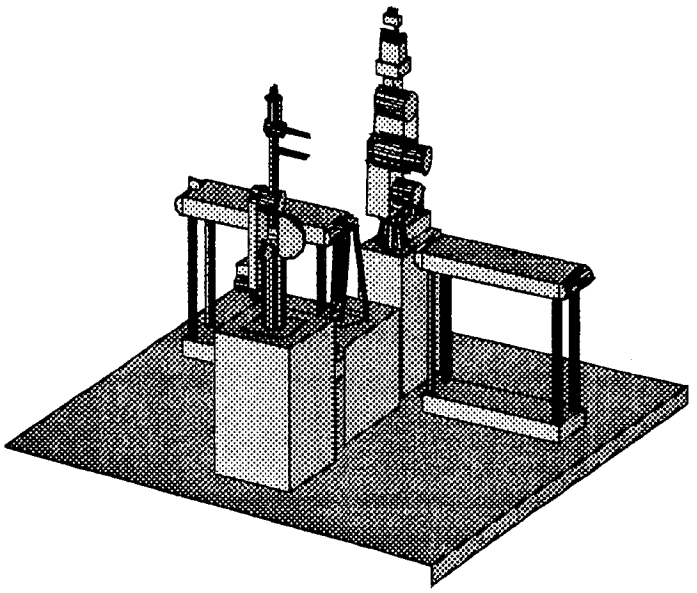
Nous dégagerons de cette étude une méthodologie de programmation adaptée aux applications robotiques afin de permettre une utilisation efficace des concepts d'ESTEREL pour la gestion du parallélisme et du temps réel.

Abstract : The increasing complexity of robotics applications demands powerful software and hardware tools allowing the execution of various tasks as well as efficient time management.

Our goal here is to exploit the language ESTEREL to control sequencing in robotics. ESTEREL is a high level, imperative language, offering parallel and synchronous facilities. Using a variety of programming examples, we illustrate the advantages of the language:modularity, parallelism, determinism, communication handling with signals, and so on.

This report will promote an effective programming style allowing the user to benefit from ESTEREL in the realization of parallel processes and the integration of real-time constraints.

Table des matières



1 Introduction

Une cellule robotisée est constituée d'un ensemble d'éléments tels que manipulateurs, convoyeurs, capteurs, qui coopèrent étroitement afin de réaliser un ensemble de tâches préalablement définies. Cette coopération nécessite un échange d'information précis entre la cellule elle-même et le monde extérieur mais aussi entre chacun des équipements robotisés constituant cette cellule. La problématique principale de la programmation d'une cellule de ce type se situe au niveau de la synchronisation et de la communication entre les différentes parties du système qui évoluent en parallèle.

Sur réception de certains événements ou en émettant des informations vers l'environnement, l'atelier de fabrication robotisé, doit exécuter un ensemble de tâches, généralement d'assemblage, en respectant certaines contraintes temporelles. Une telle cellule est un exemple de systèmes réactifs tel que les a définis A.Pnuelli : il s'agit de systèmes qui réagissent à des entrées provenant de façon répétitive de leur environnement en produisant eux-mêmes des sorties vers cet environnement. Ces systémes sont à l'origine du développement du langage parallèle synchrone ESTEREL.

Notre but est d'utiliser les caractéristiques de ce langage pour programmer un *Automate d'Application* robotique, c'est à dire une tâche informatique (au sens classique de l'informatique temps réel) qui *gére* le séquencement dans le temps de différentes tâches robot (ex : génération de trajectoire avec évitement d'obstacles, saisie ...) [1].

Le parallélisme d'ESTEREL, sa modularité, le style de programmation et le déterminisme induits par l'*hypothèse de synchronisme* sur laquelle est basée le langage apparaissent comme autant de facilités permettant la description du parallélisme et de la synchronisation entre plusieurs robots. Il offre ainsi des caractéristiques qui font généralement défaut aux langages de programmation classiquement utilisés pour la programmation des robots. Nous ne donnons ici qu'une brève description d'ESTEREL et incitons le lecteur à découvrir plus en détail ce langage au travers de [9] pour une simple introduction au langage, [10] pour une présentation complète de la sémantique du langage, [11] pour le manuel de référence, [12], [13]et [14] pour des exemples de programmes, [15] pour une étude détaillée de la sémantique et de l'implémentation du langage.

L'intérét de cette étude est double : elle montre les possibilités offertes par ESTEREL pour programmer avec succés des applications robotiques, elle permet d'étudier l'interfaçage d'un programme ESTEREL avec un environnement asynchrone. Nous avons développé des programmes parallèles et modulaires permettant la réalisation de plusieurs tâches robotisées. La modularité correspond à une décomposition significative de l'application. Chaque module posséde une fonctionnalité propre et peut ainsi être facilement adapté et réutilisé dans une nouvelle application : notre décomposition logicielle est flexible. Elle permet la réalisation de plusieurs tâches par différentes cellules de configurations matérielles variées.

Nous décrivons ici les différentes méthodes de programmation abordées. Il en existe à l'heure actuelle cinq : les trois premières correspondent au fonctionnement nominal de l'application, les deux dernières traitent le cas d'exception particulier où l'un des deux robots tombe en panne.

Méthode 1 En suivant la règle *une fonction- un module*, nous associons un module à chacun des manipulateurs physiques constituant l'application. Nous aboutissons à un programme complexe et peu utilisable effectivement.

Méthode 2 Nous supprimons une grande partie du parallélisme intrinsèque de l'application, pour aboutir à un programme *séquentiel*, simple mais peu efficace au niveau optimisation d'exécution. Cette décomposition s'éloigne légèrement des spécifications du cahier des charges. Elle constitue en fait une étape intermédiaire permettant d'aboutir à la méthode suivante. On décompose l'application en tâches robot, laissant de côté la constitution physique du système.

Méthode 3 Le programme précédent s'étoffe et rend compte de la structure d'exécution en *pipe-line* naturelle à l'application. La conception est proche de la précédente.

Nous comparons ici les trois méthodes : taille de code, efficacité ...

Méthode 4 et 5 Des modifications locales du code des versions 2 et 3, conduisent aux programmes de traitement d'erreurs, permettant à l'un des deux robots de continuer à assurer le fonctionnement de l'atelier lorsque son homologue tombe en panne. Cette récupération d'erreur est effectuée lorsque les deux robots sont actifs en séquence (cas 2) et dans le cas plus intéressant où les deux robots fonctionnent en parallèle (cas 3).

Les méthodes décrites nous ont permis de mettre en évidence certaines caractéristiques d'ESTEREL propres à satisfaire les exigences robotiques. Une nouvelle façon d'aborder la décomposition de l'application robotique et le développement d'un outil de simulation graphique permettront par la suite d'exploiter plus encore les propriétés de ce langage.

2 La programmation des systèmes robotisés

L'utilisation de robots dans des ateliers flexibles offre aux industries actuelles la possibilité d'accroître leurs performances. Dans certains secteurs industriels comme le secteur nucléaire ou aéronautique, l'environnement hostile rend l'intervention humaine impossible . Pour d'autres domaine d'activités (secteurs manufacturiers et automobile), la productivité est l'un des facteurs déterminants permettant d'atteindre un niveau de compétitivité satisfaisant. La qualité des communications entre les outils sophistiqués utilisés conditionne l'efficacité du système robotisé employé. Il s'agit par conséquent de gérer correctement les informations du système et d'établir une méthodologie de programmation des tâches robot en exploitant toutes les possibilités offertes par les outils matériel et logiciel.

Le développement historique des robots, depuis les simples robots "pick and place" jusqu'aux outils sophistiqués actuels, fut accompagné par le développement

de méthodes de programmation [3,2] de complexité et de puissance croissante. La première méthode connue revenait à faire exécuter des mouvements par un robot à l'aide de matrice de diodes (plug board), les premières machines-outils, dites programmables, fonctionnaient ainsi.

De nouveaux paramètres furent ensuite introduits (vitesse, temps). L'étape suivante consista à programmer les robots par apprentissage. Cette méthode permet au robot de restituer, à l'exécution, une trajectoire mémorisée lors de la phase d'apprentissage. Ces méthodes permettent de réaliser des actions simples. Elles s'avèrent plus délicates à mettre en oeuvre lorsque l'on souhaite effectuer un ensemble de tâches évoluées, telles que le suivi d'une trajectoire précise ou encore des opérations d'assemblages complexes. Elles sont en effet dépourvues de primitives logiques. A l'heure actuelle 90% des robots industriels sont encore programmés en ayant recours à de telles technologies.

Les dernières améliorations dans le domaine de la programmation robotique consistèrent à introduire la programmation textuelle et l'on assiste depuis plusieurs décades au développement de langages de programmation classiques, et de haut niveau complétés par des outils de simulation graphique (CAO, TAO). Ces langages constituent une interface homme-machine qui permet à l'utilisateur de programmer un ou plusieurs robots en décrivant diverses relations cinématiques à l'aide d'instructions textuelles. Un nombre important de ces langages (plus d'une centaine) existe actuellement. Aucun ne satisfait réellement l'utilisateur : certains offrent uniquement la possibilité de manipuler des repères (LM), d'autres dépendent fortement du robot à programmer. Les informations capteurs sont difficiles à prendre en compte, la gestion des différents axes des manipulateurs n'est pas des plus aisée. Ils sont souvent dénués de parallélisme limitant ainsi la réalisation de tâches concurrentes. La gestion du temps est assez rustique (VALII et ML). Ils sont asynchrones (LMAC). Leur utilisation requiert en général des connaissances techniques importantes.

L'Intelligence Artificielle (I.A.) est généralement utilisée à un niveau de décision et planification, l'objectif final étant la génération automatique des programmes de commande de robots industriels. l'I.A. cherche à donner aux robots les moyens de décider eux mêmes de leurs actions à partir d'objectifs définissant les tâches en utilisant différents outils informatiques tels que générateur de plans. La C.A.O (outil de Conception Assistée par Ordinateur tel que CATIA) permet de simuler et de visualiser les tâches complexes, et la T.A.O (Téléopération Assistée par Ordinateur) est indispensable chaque fois que la sécurité l'exige et lorsque l'on atteint les limites du savoir faire technologique ou logiciel.

Toutes ces méthodes doivent permettre de programmer un ensemble de tâches robotisées, c'est à dire de décrire l'ensemble des sous tâches à réaliser et de spécifier les liens de dépendances temporelles existant entre ces différentes sous tâches [2]. Nous utilisons ici ESTEREL dans le but de décrire rigoureusement le séquencement des activités des systèmes robotisés. Les avantages offerts par le langage seront mis en exergue au cours de notre présentation.

3 Description d'ESTEREL

ESTEREL est un langage de programmation de haut niveau, de style impératif, destiné à la programmation des Systèmes Réactifs. De tels systèmes répondent à un flux d'événements d'entrée provenant du monde extérieur en produisant un flux d'événements de sortie vers l'environnement. En l'absence d'événement d'entrée, ces systèmes sont inactifs. Une propriété importante d'un système réactif est son *déterminisme* : il répond toujours de la même manière aux mêmes séquences d'entrées.

Les systèmes réactifs synchrones (i.e. qui réagissent de façon instantanée aux simulations provenant de l'extérieur) constituent un modèle idéal de nombreuses applications temps réel (protocoles de communications, systèmes de transmission ou encore systèmes robotisés dans notre exemple). Les outils de programmation classiques sont mal adaptés à la programmation de ces systèmes réactifs synchrones. Certains (comme les réseaux de Pétri) manquent de primitives de haut niveau. Les langages de programmation (ADA, OCCAM) présentent des inconvénients liés à l'asynchronisme et au non déterminisme qui leur sont propre. ESTEREL abandonne l'hypothèse d'asynchronisme, à la base des langages classiques, pour la remplacer par une hypothèse de *synchronisme fort.* On suppose ainsi que les sorties du programme sont fournies de manière absolument synchrone aux entrées, donc que leur calcul "ne prend pas de temps".

Nous donnons ici quelques caractéristiques du langage ESTEREL :

- Un programme ESTEREL est constitué d'une collection de *modules* qui peuvent communiquer et échanger de l'information. La structure de module permet une programmation hiérarchique. Il existe deux opérateurs de composition de modules : l'opérateur "||" rend le *parallélisme* explicite au niveau de l'utilisateur. Dans un parallèle toutes les composantes sont lancées simultanément, ce qui est traduit par :"faire une action *en même temps qu*'une autre"; le parallèle se termine instantanément dès que toutes ses composantes sont terminées (i.e. les deux branches sont terminées!). L'opérateur ";" de composition *séquentielle* permet d'exécuter une suite d'instructions, c'est à dire que l'on "fait une action *puis* une autre".

- La gestion de la communication, de la synchronisation et du partage des données est assurée par l'emploi de *signaux* portant éventuellement des valeurs. Ce sont les signaux émis par l'extérieur qui définissent les événements d'entrée d'un programme ESTEREL. Ils sont la seule cause de réaction d'un programme. Un programme ESTEREL réagit instantanément à la réception de signaux d'entrée en émettant lui-même des signaux de sortie vers son environnement. Un programme peut aussi émettre et recevoir des signaux internes ou locaux servant à la communication entre sous processus internes. Ces signaux sont non visibles de l'extérieur. Tous les signaux sont émis "à la cantonade" instantanément (Diffusion) : toutes les parties du programme reçoivent les mêmes signaux au même instant.

- Nous avons dit qu'un programme ESTEREL est inactif lorsqu'il ne reçoit pas de signaux d'entrée. Ce qui signifie qu'un tel programme ne possède pas d'horloge

interne (ou privilégiée). Les *instants* d'exécution du système correspondent à la réception des signaux d'entrée. Chaque type de signal définit ainsi une unité de temps et peut être géré à l'aide de constructions habituellement réservées au temps (ex : le chien de garde).

- L'hypothèse de synchronisme à la base d'ESTEREL permet de simplifier les raisonnements sur le temps : certaines instructions ont une durée nulle, le contrôle ne prend pas de temps.
- Cette hypothèse rend de plus le langage *déterministe*. Ce qui facilite la réalisation, la mise au point et l'étude des comportements des programmes réalisés.
- Le langage a reçu une sémantique mathématique rigoureuse et complète sur laquelle repose son implémentation.
- Le compilateur ESTEREL traduit un programme en un *automate fini déterministe*. Cet automate est une forme compilée des réactions possibles à toutes les histoires d'événements d'entrée, dans laquelle le parallélisme a été séquentialisé. Le langage ESTEREL permet de combiner la souplesse d'écriture d'un langage parallèle de haut niveau avec l'efficacité des automates (lors de l'exécution). La traduction en automate fini offre de plus l'avantage de pouvoir interfacer un programme ESTEREL avec des systèmes de preuve pour automates finis (outils formels de raisonnement).

4 Description de l'application

L'application que nous considérons est fictive : il n'existe actuellement aucune implantation physique de ce système multirobots-multicapteurs. Le scénario énoncé ci-dessous constitue donc un simple exemple de travail. Cependant le développement d'un système de ce type [8] permettant de faire coopérer manipulateurs et capteurs est en cours de développement au sein du projet PRISME[1] de l'I.N.R.I.A [2]. La conception de notre application est telle qu'elle devrait pouvoir s'intégrer dans ce système distribué dédié à la robotique. Celui-ci se compose de plusieurs manipulateurs et capteurs que l'on sépare en modules (cartes de calcul et d'interfaces communicant par bus) pour permettre une configuration du système aisément adaptable aux besoins de l'application. Un réseau local Temps-Réel, dont les protocoles sont également programmés en ESTEREL[6,7], permet le transfert d'informations entre les différentes entités physiques. L'architecture d'un tel système est organisée en 3 niveaux : le niveau application où les différentes étapes de l'application sont décrites, le niveau commande où l'on réalise l'élaboration de la commande et le niveau système qui concerne le matériel. Notre but est d'obtenir un *Automate d'Application* (niveau application) manipulant des tâches robot (commande, générateur de trajectoire) et des signaux (événememts).

[1] Programmation des Robots Industriels et des Systèmes Manipulateurs Evolués
[2] Institut National de Recherche en Informatique et Automatique

4.1 Acteurs

L'assemblage décrit nécessite la *coopération* de cinq manipulateurs ou objets (configuration physique représentée figure 1) :

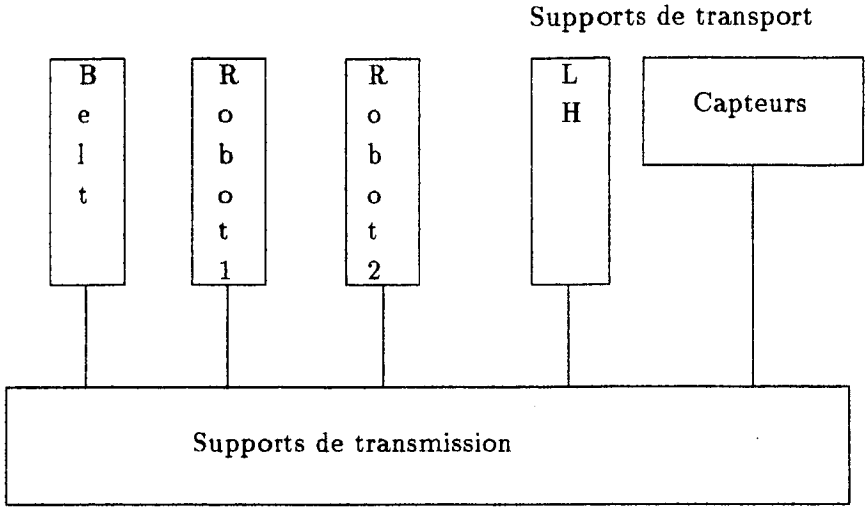


Figure 1: Système multirobots-multicapteurs

- Un robot manipulateur ROBOT1 ou R1 à 6 ddl[3] pour saisir les objets sur les convoyeurs et les insérer,
- Un robot manipulateur ROBOT2 ou R2 pour la préhension des pièces préassemblées,
- Une main gauche LH (Left Hand dans la terminologie anglo-saxone) :robot parallèle [4] à 6 ddl comportant un système de verrouillage de pièces, et devant coopérer avec R1 dans la phase d'assemblage.
- Deux tapis roulants (belt1 ou B1 et belt2 ou B2) pour le convoyage des objets.

Le système physique comporte en outre de nombreux capteurs physiques (capteurs infra rouge anticollision, capteurs de vision, de force, ultra-sons pour la saisie des objets, détecteurs de présence ...). Ces capteurs physiques génèrent des signaux témoins des changements d'états du système, qu'il sera nécessaire d'interfacer avec le programme ESTEREL. Ils constituent en fait l'ensemble des signaux d'entrée des programmes détaillés.

[3] degrés de liberté

4.2 Scénario

Le scénario suivant correspond au fonctionnement *nominal* de l'application :

Le ROBOT1 saisit une pièce sur l'un des convoyeurs(B1), l'installe sur la main gauche (LH). La pièce est ensuite maintenue en place. Le ROBOT1 saisit une autre pièce sur le second convoyeur (B2) et l'assemble avec la première en coopération avec la main gauche (LH). L'ensemble est déverrouillé. Le ROBOT2 peut alors le saisir et aller l'insérer dans la structure complexe.

Une première approche consiste à spécifier une coopération sans incident entre les différents acteurs de l'assemblage. Nous étudierons, ultérieurement, le traitement des multiples cas de disfonctionnement nécessitant la prise en compte de signaux d'alarmes. La modularité d'ESTEREL et la flexibilité qu'elle induit, facilitent l'intégration de ces alarmes (réutilisation du code de l'application, ajout de nouvelles instructions).

Le fonctionnement nominal se traduit par l'enchaînement de tâches robot. L'exécution de ces tâches est séquencée par l'arrivée de un ou plusieurs signaux, fournis par le système matériel. Nous ne nous intéressons pas ici aux algorithmes des tâches robot mais uniquement à leur synchronisation avec le reste du monde. Nous réalisons un contrôleur d'exécution de la cellule flexible en concentrant notre programmation sur les aspects logiques de l'application.

5 Signaux d'interface Entrées/Sorties - Tâches Robots

Les signaux décrits sont communs aux différentes versions envisagées par la suite. Nous ne citerons aucun des signaux qui interviennent lorsque l'un des deux robots tombe en panne (fonctionnement dégradé). Ils se différencient des signaux correspondant au fonctionnement nominal par le préfixe $R\_$ mais conduisent à l'exécution des mêmes tâches par l'un ou l'autre des manipulateurs.
Remarque : Les signaux préfixés par $X\_$ sont amenés à disparaître par la suite. Ils permettent en fait de simuler la primitive ESTEREL **exec** en cours d'implémentation. Cette instruction permet d'exécuter des procédures dont le temps de calcul n'est pas négligeable (cas de notre application). Elle sous-traite l'exécution d'une tâche à la partie *Traitement de Données* du système d'exploitation introduisant ainsi une notion de traitement *asynchrone*.

5.1 Dépose OBJET1 sur main gauche

Cette action nécessite la réalisation des sous tâches suivantes (on note $\triangleright$*input* les signaux d'entrée et $\triangleleft$ *output* les signaux de sortie de notre application). Notons que certains modules ESTEREL associés à des tâches précises seront utilisés plusieurs fois. Chaque module correspond à une tâche et une seule. Le "renommage" des signaux que l'on peut utiliser en ESTEREL permettra de différencier physiquement les tâches et de les associer à leur destinataire propre.

- Mise en marche du convoyeur1 : ($\triangleleft$ *X_BELT1_MOTION*). Le convoyeur est stoppé lorsque la pièce est détectée ($\triangleright$ *X_VICINITY_OF_B1*).

- Activation du robot1 ($\triangleleft$ *X_R1_TRAJ_TOWARD_B1*). La tâche activée doit permettre un déplacement du robot vers un point proche de la pièce ($\triangleright$ *X_OBJECT_VICINITY_R1B1*).

- Saisie de la pièce par le ROBOT1 ($\triangleleft$ *X_GRIPPING_ON_BELT*). Cette tâche effectuée physiquement par le robot englobe ici différentes sous-tâches qui restent tout au long de l'application transparentes à l'utilisateur : avance sous contrôle capteurs, détection de la position correcte de l'objet, saisie effective de l'objet, fermeture de la pince, test de présence de l'objet. Cette séquence effectuée par la partie traitement de données du système d'exploitation est du ressort de la commande de robots. Le contrôle est rendu au programme ESTEREL lorsque $\triangleright$ *X_OBJECT_IN_GRIP* est reçu.

- Déplacement lent du robot vers la main gauche ($\triangleleft$ *X_TRAJECTORY_TRACKING_TOWARD_LEFT_HAND* et $\triangleright$ *X_LEFT_HAND_VICINITY*).

- Dépose de l'objet1 dans la main gauche ($\triangleleft$ *X_PUTTING_THE_OBJECT_IN_LEFT_HAND* et $\triangleright$ *X_OBJECT_IS_IN_ LEFT_HAND* correspondant à une phase de verrouillage).

5.2 Insertion OBJET2 dans la main gauche

- Verrouillage de l'objet par la main gauche ($\triangleleft$ *X_LOCKING_THE_OBJECT* et $\triangleright$ *X_OBJECT_LOCKED*).

- Avance convoyeur numéro 2 : mêmes signaux que pour le tapis 1 indicés par 2.

- Saisie de l'OBJET2 sur le deuxième convoyeur, déplacement vers la main gauche (identique aux cas précédent avec indiçage 2).

- préinsertion des deux objets : cette phase nécessite la coopération de la main gauche ($\triangleleft$ *X_INSERTION_TASK*) et du robot ($\triangleleft$ *X_INSERTING_THE_OBJECT_IN_LEFT_HAND*). Les deux tâches effectuées conjointement sont arrêtées sur réception de $\triangleright$ *X_OBJECT_INSERTED*.

5.3 Saisie par ROBOT2 dans la main gauche

- Saisie de l'assemblage par le deuxième robot : un suivi de trajectoire de R2 depuis sa position actuelle jusqu'à la main gauche doit être effectué en premier lieu ($\triangleleft$ *X_R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND* et $\triangleright$ *X_ASSEMBLYVICINITY_R2*). L'objet doit être déverrouillé dans la main gauche ($\triangleleft$ *X_UNLOCKING_THE_ASSEMBLY* et $\triangleright$ *X_ASSEMBLY_UNLOCKED*). La pince du ROBOT2 doit ensuite saisir le préassemblage ($\triangleleft$ *X_GRIPPING_THE_ASSEMBLY* et $\triangleright$ *X_ASSEMBLY_GRIPPED*).

5.4 Insertion en bout de chaîne

- Transport des objets préassemblés en bout de chaîne : le robot doit se déplacer sous contrôle capteurs vers le lieu d'insertion ($\triangleleft$*X_MOVING_TOWARD_INSERTING_PLACE* et $\triangleright$*X_INCERTION_VICINITY*), puis enfin insérer les objets ($\triangleleft$*X_INSERTING_THE_ASSEMBLY* et $\triangleright$*X_ASSEMBLY_INSERTED*).

L'ordre d'exécution de ces tâches dépend de la méthode de programmation envisagée et des contraintes temporelles.

Les signaux listés ci-dessus sont des signaux purs. Comme nous l'avons dit précédemment nous nous intéressons uniquement à l'aspect logique de la programmation. Il n'est donc pas utile de connaître les valeurs des calculs effectués par les différentes tâches.

6 Version1 : un module-un manipulateur

Lors de l'analyse de l'application, une première approche consiste à traduire le fonctionnement nominal en associant un module ESTEREL à chacun des manipulateurs. Nous sommes toujours guidés par le principe de modularité que les notions de synchronisme et de parallélisme rendent aisément applicable en ESTEREL.

Ces modules sont constitués d'un ensemble de tâches effectuées *séquentiellement*. En effet, en considérant isolément l'un des robots on constate qu'il ne peut réaliser qu'une chose à la fois et qu'une tâche ne peut commencer que lorsque celle qui la précède est terminée. Ces modules de gestion sont ensuite mis en paralléle et leur coopération est assurée par échange de signaux. Ces signaux sont tous déclarés *locaux* par le programme principal. Ils sont utilisés pour assurer une synchronisation correcte entre les différentes activités (information de contrôle) et illustrent les notions de *diffusion*, de *communication synchrone* et la facilité d'utilisation qui leur sont propres.

6.1 Architecture parallèle

Les tâches décrites précédemment sont réparties au sein des modules suivants :

- Le module `ROBOT1_CONTROLLER` séquence l'exécution de tâches propres au ROBOT1 selon l'ordre suivant : suivi de trajectoire vers B1, saisie de l'OBJET1 (O1), suivi de trajectoire vers la main gauche (LH), dépose de l'objet dans la main gauche (LH) lorsque cette dernière est disponible, déplacement vers B2, préhension de l'OBJET2 (O2), nouvelle trajectoire vers la main gauche (LH), insertion en coopération avec la main gauche (LH) lorsqu'elle est prête.

- le module `ROBOT2_CONTROLLER` enchaîne les activités de suivi de trajectoire vers la main gauche (LH), de saisie d'objet dans la main gauche (LH) quand cela est possible, de déplacement vers le lieu d'insertion et enfin d'insertion en bout de chaîne.

- le module **LEFT_HAND_CONTROLLER** : son rôle consiste à signaler à son environnement (en l'occurence aux robots) l'état dans lequel se trouve la main gauche (LH). Il commande l'exécution des tâches de verrouillage, insertion conjointe avec le ROBOT1, déverrouillage.

- Le module **BELT_CONTROLLER** se caractérise par une tâche unique : avance tapis.

L'architecture générale est illustrée sur la figure 2 représentant les relations existant entre les différents modules. Ce schéma en block diagrammes, réalisé à l'aide de l'outil graphique AUTOGRAPH [16] permet de comprendre la structure du programme principal.

Les trois premiers modules sont étroitement liés : la main gauche (LH) apparaît comme une *ressource commune* aux deux robots et la dépendance entre les modules réside dans le partage de cette entité. En régime permanent, le ROBOT1 peut demander l'accès à la main gauche (LH) soit pour y déposer le premier objet, soit pour insérer (avec coopération de la main gauche) l'OBJET2 avec l'OBJET1. Le ROBOT2 s'approprie la ressource lorsqu'il vient saisir les deux objets.

La main gauche apparaît, ici, comme l'ordonnanceur gérant le scénario en pipeline. Il est nécessaire de définir précisement les intéractions locales, ainsi que les différentes sections critiques d'accès à la main gauche.

6.2 Partage de la main gauche

Le programme conçu gère le partage de la ressource-main gauche par l'intermédiaire d'un *contrôle réparti* : une mémoire est associée à chacune des sections critiques d'accès à la ressource partagée.

L'emploi de telles mémoires illustre deux points importants du style de programmation en ESTEREL :

⟶ Nous pouvons utiliser un nombre important de signaux de contrôle purs pour permettre une décomposition bien modulaire de l'application.

⟶ Nous communiquons des informations d'états entre les différents modules par l'intermédiaire de dialogues instantanés.

Nous décrivons en détail le dialogue instantané instauré pour accéder à une section critique particulière : l'insertion des deux objets dans la main gauche. Celle ci peut être prête ou non à coopérer. Lorsque le robot souhaite insérer un objet il pose la question "main gauche es-tu prête?" et se met en attente du signal "je suis prête". On réalise ici un mécanisme de synchronisation de type rendez-vous. L'émission du signal peut suivre deux chemins différents :

- La question "main gauche es tu prête" intervient *avant* que LH soit prête. Dès qu'elle termine les tâches en cours, le signal "je suis prête est émis" (débloquant l'attente du ROBOT1) et il n'est pas utile de mémoriser le signal.

- Elle intervient *après*. Le module mémorisateur évite que le signal "je suis prête", de nature fugace, ne soit perdu, et permet, lorsque la question est posée, de renseigner le ROBOT1 sur l'état de la main gauche.

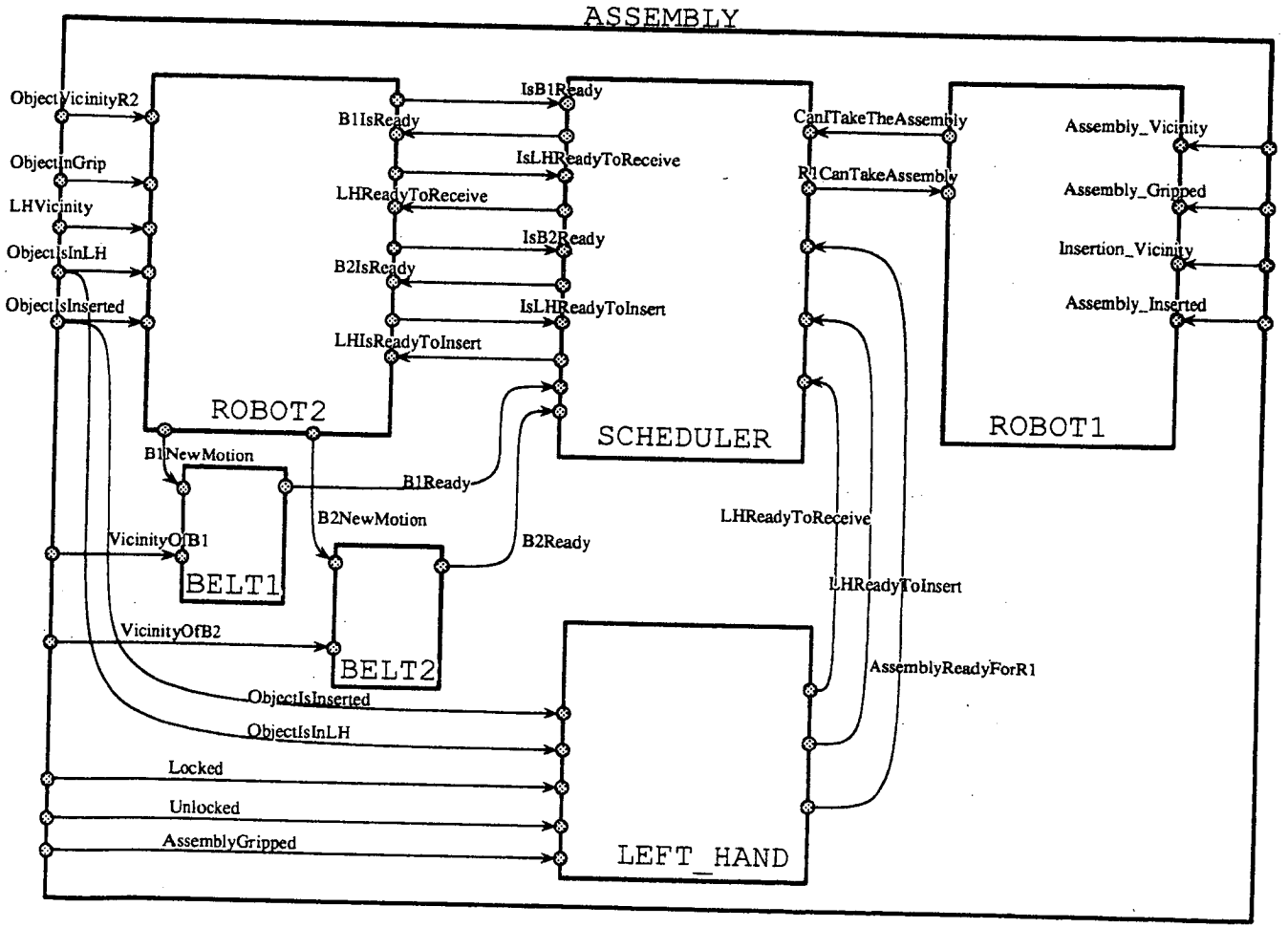


Figure 2: Architecture parallèle

Le contrôle de l'accès aux ressources partagées est intégré à chacun des modules gérant le séquencement des différents manipulateurs. Il est représenté de façon schématique sur la figure 3. Les mémoires ne sont plus actives en permanence mais le deviennent au moment critique de façon à éviter la perte d'information, les cycles de causalité et les étreintes fatales (deadlock).

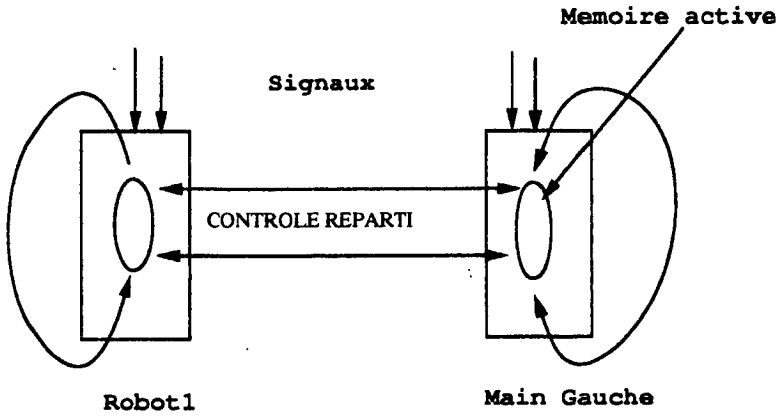


Figure 3: Contrôle réparti

6.3 La mémoire active

Nous allons décrire plus en détails le rôle de la mémoire active.

Mémorisation

Le code générique d'une bascule simple est le suivant :

```
await END_TASK;
          do
            loop
               emit OK
           each QUESTION
          upto BUSY;
```

Cet exemple introduit plusieurs primitives d'ESTEREL : **emit**, **await**, **do - upto**, **loop - each**. Rappelons que les signaux sont diffusés instantanément à toutes les composantes du programme. Pour émettre un signal S, on écrit simplement **emit** S. L'attente d'une occurence de ce même signal s'écrit **await** S. Le chien de garde **do - watching** permet de définir l'instruction de garde temporelle **upto**. Cette instruction ne se termine que sur réception de l'événement et non si son corps se termine : **do inst upto occ**. Enfin la primitive **loop - each** permet de traiter le déclenchement répétitif d'actions sur réception du signal S : **loop inst each** S. La primitive ";" traduit la séquence.

Dans le cas de notre exemple, on attend la fin de la tâche avant de déclarer la main gauche disponible(`await END_TASK`). La construction `loop-each` permet de rendre compte correctement de l'état de la ressource : si la ressource est prête avant la question, le signal sera émis sur réception de la question, sinon il sera tout de même émis une fois pour débloquer l'attente de l'interrogateur. Avec ce type de boucle le "corps" de la boucle est toujours exécuté au moins une fois. A chaque réception du signal `QUESTION`, le corps de la boucle est à nouveau exécuté. L'occurence du signal `BUSY` tue le corps de la mémoire.

Les cycles de causalité

Avec la construction précédente, le ROBOT1, par exemple, sait lorsque l'état de la main gauche est libre. Dans le même instant, il s'approprie la ressource et doit indiquer que l'insertion peut commencer. La main gauche devient simultanément indisponible. Nous considérons qu'un *instant* contient un événement formé par l'occurence d'un signal ou de plusieurs signaux simultanés.

L'hypothèse de synchronisme peut engendrer des paradoxes temporels dès que les signaux d'entrées d'une instruction dépendent des signaux qu'elle peut émettre dans le même instant. Il s'agit de problèmes de causalité, analogues aux deadlocks des langages asynchrones. Le problème de dialogue instantané apparaît ici clairement : la main gauche se trouve dans le *même instant* à la fois libre et occupée. Supposons que le signal `BUSY` soit absent, le signal `OK` est alors présent (au premier instant). Or si `OK` est présent, celà implique que `BUSY` devient présent. Dans le même instant on devrait donc avoir `BUSY` absent et présent. Ce qui est impossible. D'autre part, si l'on suppose `BUSY` présent, on ne peut pas émettre `OK` (soit `OK` absent). Dans ce cas, `BUSY` ne peut pas être présent. Soit la description suivante du cycle de causalité :

$$\begin{cases} \text{si } \overline{BUSY} & \Rightarrow & OK \\ \text{si OK} & \Rightarrow & BUSY \end{cases}$$

$$et \begin{cases} \text{si BUSY} & \Rightarrow & \overline{OK} \\ \text{si } \overline{OK} & \Rightarrow & \overline{BUSY} \end{cases}$$

Des moyens d'analyse statique permettent de localiser les cycles de causalité créés et de conserver uniquement les programmes causalement corrects.

Nous avons transformé le code précédent pour supprimer les dialogues entre le programme et *son futur instantané* (cycle détecté par le compilateur).

```
% les commentaires commencent par %
loop
   trap CYCLE in
      await immediate READY do
         loop
             emit OK
```

```
        each QUESTION
    ||
      await BUSY do
         exit CYCLE
      end
   end  % trap
end % loop
```

Nous utilisons pour cela de nouvelles primitives ESTEREL. `loop - end` est l'instruction classique de boucle infinie. La construction **`trap inst in inst end`** constitue un puissant mécanisme de sortie de bloc. Cette instruction se comporte comme son corps **`inst`** et se termine normalement si le corps ne déclare pas une exception. Sinon la construction **`trap`** permet de sortir du bloc et d'exécuter en séquence le code qui suit la construction **`trap`**. On utilise ici la primitive impérative de parallélisme `||`. Enfin, l'emploi de **`immediate`** permet de spécifier que l'on veut que le système réagisse dans l'instant présent, et non dans l'instant d'après. Avec **`await immediate`** on attend une occurence de signaux dans l'instant présent ou futur alors qu'avec **`await`** on attend forcément une occurence à partir de l'instant futur.

Le cycle de causalité est supprimé : dans le même instant, l'émission du signal **`OK`** et la réception du signal **`BUSY`** peuvent avoir lieu. L'occurence de **`BUSY`** permet de sortir du bloc (**`trap`**) et de tuer le corps de la boucle. L'état de la ressource partagée est alors cohérent avec l'évolution du système et on passe ensuite à la séquence de mémorisation ou de travail suivante. Notons cependant que pour supprimer les cycles de causalité nous employons une construction parallèle qui, en ESTEREL, ne prend pas de temps à l'exécution (car résolu à la compilation).

Les dead locks

Lors de la conception initiale du programme, nous avions pensé mettre en parallèle modules de mémorisation et modules de gestion rendant ainsi l'état des ressources préoccupantes accessible en permanence. Cette architecture s'est avérée impraticable. Elle introduisait en effet des deadlocks que les outils de preuve AUTOGRAPH [16] et AUTO [17] permettent de localiser (numéro de l'état puits, plus courte "histoire" d'entrée permettant d'atteindre cet état). A partir de la liste de signaux fournie par AUTO, nous avons étudié en le simulant, le comportement du programme. Le deadlock était directement lié à l'architecture du programme envisagée. La mémoire mise en parallèle n'était jamais inactive et nous assistions dans le même instant à la suite d'événements suivante : la main gauche devient libre, le ROBOT1 s'approprie la ressource et doit donc la déclarer occupée. La mémoire effectuant une boucle parallèle reste en permanence dans l'état "occupé".

Nous avons donc activé la mémoire uniquement lorsque nécessaire. Nous supprimons ainsi deadlocks, cycle de causalité et restons plus proche de la réalité physique.

6.4 Mise au point du Programme

Une manière partielle de tester un programme consiste à simuler son exécution et vérifier que son comportement correspond à celui spécifié. La procédure de simulation est la suivante : un programme détecte les événements d'entrée (rentrés au clavier par l'utilisateur), appelle l'automate et les procédures affectées aux signaux d'entrée. Les événements de sortie sont affichés à l'écran.

La mise au point du programme par simulation n'est pas des plus aisée surtout lorsque, comme dans notre exemple, le nombre de signaux à gérer est important. Elle permet cependant de mettre en évidence un problème important : associer un module à chacun des manipulateurs physiques et exécuter les différents modules en parallèle produit un automate comportant un grand nombre d'états. Cette abondance d'états n'est pas justifiée : certains modules attendent des signaux qui n'ont en fait aucune chance d'être émis tant que certaines tâches parallèles ne sont pas terminées.

Le comportement d'un programme est entièrement décrit par un automate à contrôle fini. A une propriété du programme que l'on souhaite vérifier, correspond une propriété de l'automate qui peut être établie sur celui-ci. Comme le compilateur traduit les programmes en un automate qu'il est possible d'interfacer avec un système travaillant sur les automates finis. Ce qui permet de vérifier un programme réactif. Le système AUTO permet de calculer des comportements partiels d'automates finis, en considérant, par exemple, uniquement certaines actions du programme, ou en identifiant certaines séquences d'actions. On peut ainsi étudier le comportement d'un programme ESTEREL d'une façon observationnelle, de façon à confirmer, ou à infirmer, des propriétés. Nous avons notamment pu détecter la présence de deadlock grâce à ce système.

7 Version2 : programme séquentiel

La vision totalement parallèle de l'application n'est pas tout à fait adéquate. La décomposition précédente ne tient pas compte de la séquentialité des tâches à réaliser. Pour réduire ce "parallélisme outrancier", nous programmons l'application de façon *séquentielle*, en tenant compte du fait que certaines tâches doivent être effectuées les unes après les autres. L'idée initiale consiste à lancer l'exécution des différentes tâches uniquement lorsque nécessaire, en évitant, que comme précédemment elles n'attendent des événements dont la probabilité d'apparition est nulle. Un séquenceur gère l'ordonnancement des modules. La décomposition modulaire (architecture distribuée) de la version 1 est remplacée par un enchaînement de tâches robot (*un module-une tâche*). Le programme, de conception plus simple, établit un dialogue entre un *séquenceur* et tous les modules (ou toutes les tâches) comme schématisé sur la figure suivante (figure 4).

Le code produit provient d'une décomposition en tâches fonctionnelles. Nous avons répertorié quatre tâches fonctionnelles (une tâche-un but-des ressources) : transport de l'OBJET1 dans la main gauche ($T1$), préinsertion des deux objets ($T2$), préhension du préassemblage ($T3$), insertion finale ($T4$). Remarquons que l'on considère ici les

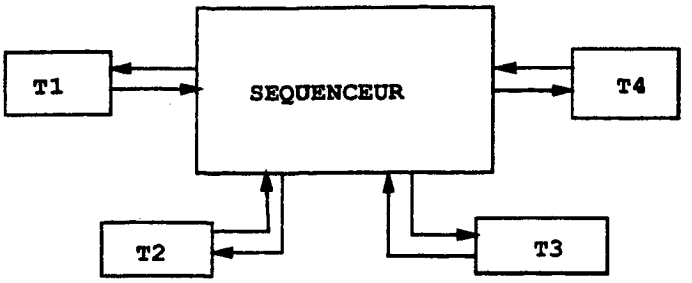


Figure 4: Séquencement des activités

fonctions réalisées par l'atelier de production et non plus, comme dans la première approche, les tâches associées à chacun des manipulateurs physiques. Le partage des ressources mises en jeu par les tâches est alors différent. Une fonction particulière peut nécessiter la coopération de plusieurs manipulateurs. Etudions par exemple la tâche $T1$: elle exige la disponibilité du ROBOT1, du convoyeur numéro 1 (B1) et de la main gauche (LH). Soit trois ressources. Ici, nous n'avons pas à gérer le partage de ces ressources car la construction séquentielle envisagée impose l'exclusion mutuelle des ressources. Nous verrons, dans la section suivante, comment gérer le partage des ressources.

Les modules qui apparaissent sont alors les suivants :

- **SCHEDULER** : il appelle successivement les quatre tâches, soit le code :

```
loop
    copymodule TASK1;
    copymodule TASK2;
    copymodule TASK3;
    copymodule TASK4;
end
```

- TASK1, TASK2, TASK3, TASK4 : la programmation de ces modules conserve une part du parallélisme des actions à accomplir. On décompose chaque tâche en un ensemble de sous tâches réalisées *en même temps* ou *séquentiellement*. Chaque module correspond en fait à un mini séquenceur envoyant aux robots et manipulateurs une succession d'ordres où les mots clés sont *faire*, *puis* et *en même temps que*, que l'on peut écrire en ESTEREL :

```
[  copymodule EXECUTE_ACTION1
; % puis
   copymodule EXECUTE_ACTION2   ]
||  % en meme temps que
  copymodule EXECUTE_ACTION3
```

Le programme est simple, concis, partiellement représentatif de l'application. Il ne comporte aucun cycle de causalité puisque on attend toujours la fin d'une tâche avant d'en exécuter une autre. En outre aucun dialogue instantané n'est instauré entre tâches ou encore à l'intérieur d'une même tâche. L'outil graphique AUTOGRAPH permet de visualiser l'automate produit par compilation du programme. Il est représenté figure 5 avec la convention suivante : $S?$ représente la réception du signal S et $S!$ son émission. Nous constatons notamment que le nombre d'états est faible. Il est en rapport avec la complexité de l'application et correspond à l'automate que l'on aurait pu réaliser "à la main".

8 Version3 : le pipeline

Le rendement (mesuré par la durée d'un assemblage) ne correspond pas à l'idée naturelle de l'application : la construction séquentielle impose que le ROBOT1 attende la fin des mouvements du ROBOT2 avant d'exécuter une nouvelle tâche. Partant de cette constatation, nous avons étoffé le programme en le parallélisant : nous nous intéressons ici au régime permanent du système, la phase d'initialisation (régime de transition) sera traitée ultérieurement.

8.1 Etude des différents modules

Nous gardons la décomposition fonctionnelle élaborée lors de l'approche séquentielle. Les tâches $T1$ et $T2$ s'approprient les ressources ROBOT1, convoyeur 1 (B1) (ou convoyeur2 B2) et main gauche. $T3$ et $T4$ nécessitent l'intervention du ROBOT2 et de la main gauche (LH). La tâche $T3$ ne peut commencer que lorsque la main gauche contient les deux objets préinsérés. La tâche $T1$ ne peut recommencer que lorsque le ROBOT2 a libéré la main gauche. Les deux robots partagent donc la ressource main gauche (LH) qui conditionne ainsi leur évolution. Il est donc nécessaire d'établir un dialogue entre chacun des manipulateurs et la main gauche. L'architecture envisagée consiste à mettre en parallèle deux modules effectuant en séquence les tâches $T1$ - $T2$ (ressources : ROBOT1, main gauche (LH), tapis) et $T3$ - $T4$ (resources : ROBOT2, main gauche (LH)). Nous utilisons pour gérer la structure en pipeline de l'application, une mémoire similaire à celle décrite dans la première partie.

- Le code du module ROBOT1_MAINLY est le suivant :

```
loop
    copymodule T1;

    copymodule T2_1;
    [ copymodule INSERTION_TASK
  ||
    copymodule MEMORY_R2_CAN_MOVE ];
    copymodule MEMORY_LH_CAN_UNLOCK;
each R1_GO_TO_GRIP_OBJECT1
```

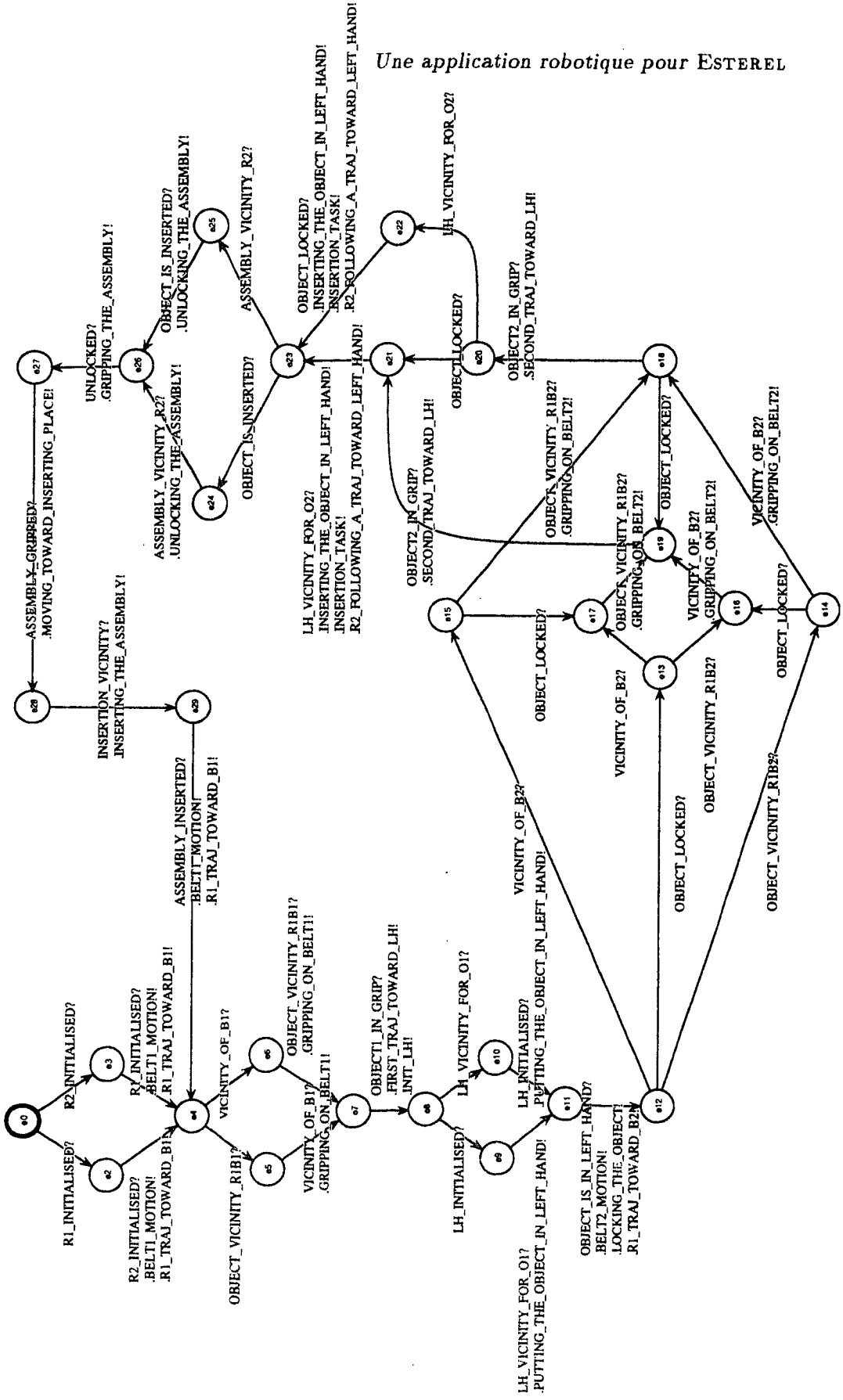


Figure 5: Automate de l'application séquentielle

où T2_1 correspond à la tâche $T2$ sans la tâche d'insertion.

- Le module ROBOT2_MAINLY attend une réponse à la question "puis-je commencer" avant d'effectuer un suivi de trajectoire. Le code correspondant est donné ci-après.

```
loop
   emit CAN_R2_START;
   await immediate R2_CAN_START;
   copymodule R2_FOLLOWING_A_TRAJ_TOWARD_LH;
   emit R2_MOVED;
   ...
   emit R1_GO_TO_GRIP_OBJECT1; ...
end
```

On remarque que le code produit est très similaire à celui de la version précédente. Les tâches sont sensiblement les mêmes. Elles sont seulement enrichies de dialogues instantanés (réglés par l'emploi de mémoires) établis entre les deux robots.

Le régime stationnaire est instauré en utilisant des séquences d'attente en début (cas du ROBOT2) ou en fin de boucle (cas du ROBOT1). Lors de la première exécution le ROBOT1 commence à travailler, il autorise ensuite le ROBOT2 à démarrer débloquant ainsi l'attente de ce dernier. Le ROBOT1 attend un signe du ROBOT2 pour pouvoir effectuer à nouveau sa boucle.

8.2 Mise au point du programme

Nous vérifions par simulation du programme (possible car le nombre d'états obtenus par compilation est faible) que le fonctionnement obtenu correspond effectivement à celui spécifié. Nous avons, en particulier, vérifié que le comportement du programme était identique à celui précédemment décrit, lorsque nous supposions que les deux robots fonctionnaient en séquence.

9 Comparaison des programmes

Avant de poursuivre notre description, comparons les trois versions précédentes. L'efficacité d'un programme ESTEREL se mesure principalement par la taille de son automate. c'est à dire par le nombre d'appels de transitions et par le nombre d'états de l'automate.

Nous donnons ci dessous le résultat de la compilation des différents programmes. Ils ont été compilés en utilisant le système ESTEREL V3. Les tailles des automates générés sont les suivantes :

Version	*états*	*actions*	*octets*	*transitions*
1	723	45	130159	6558
2	30	40	6751	144
3	70	40	12776	440

où les actions correspondent aux actions élémentaires à effectuer (calcul, test logique, assignation ...), les octets représentent l'occupation mémoire de l'automate, les transitions sont étiquetées par les actions et permettent de passer d'un état à un autre.

Ce tableau appelle les remarques suivantes :

- La différence d'états existant entre la version 1 et la version 3 est choquante par rapport à la complexité de l'application. Ces deux programmes décrivent les mêmes fonctionnalités. La taille du code de la version 1 n'est absolument pas en relation avec la fonctionnalité spécifiée. Cette abondance d'états (Version1) est liée à l'architecture employée. Les modules associés aux manipulateurs physiques, attendent en permanence des événements d'entrées provenant de leur environnement (emploi du parallélisme). Or, l'histoire logique de notre application est telle qu'il est physiquement impossible de recevoir certains signaux avant que d'autres n'aient été émis. Elle n'est absolument pas respectée, ce qui entraine la création d'un grand nombre d'états *inutiles* (car jamais atteints).

  Pour éliminer ces états non explorés, une solution consiste à réaliser un *filtre* permettant de recréer l'histoire réelle du système. Ce qui reviendrait à établir des relations d'antériorité, ou de postériorité entre signaux éliminant ainsi l'attente illicite de signaux. Ce filtre, mis en parallèle avec le reste de l'application, consommerait les signaux physiques et produirait en échange des signaux locaux, évitant ainsi une création de code inutile.

- Notons enfin que les versions 2 et 3 ne spécifiant pas les mêmes fonctionnalités, il est normal que le nombre d'états soit différent. Rappelons que si nous simulons les deux programmes en respectant l'ordre des tâches effectuées séquentiellement nous aboutissons aux mêmes résultats. Les deux versions ne diffèrent donc pas suffisamment pour que le nombre d'actions soit modifié.

  Les actions caractérisent la complexité des opérations de base réalisées. Elles sont peu nombreuses dans ces deux exemples car on s'intéresse à l'aspect logique de l'application. Elles sont sensiblement identiques.

10 Version4 et 5 : traitement d'exception

Nous envisageons ici un cas d'exception particulier permettant de ne pas arrêter le programme lorsque l'un des deux robots tombe en panne. Le robot, dont le fonctionnement n'est pas altéré, termine les tâches qu'il etait en train d'effectuer et se charge ensuite d'accomplir les tâches habituellement réalisées par le robot défectueux. Pour

ce faire il est nécessaire de connaître l'état de chacun des deux robots lors de la panne et d'intimer ensuite les ordres corrects à l'unique robot valide. Nous supposons enfin que lorsque les deux robots sont en panne, une alarme est émise et que l'application est automatiquement arrêtée.

10.1 Récupération d'erreurs : cas séquentiel

Le problème de récupération est assez trivial lorsque les deux robots fonctionnent en séquence. En effet, il n'est pas nécessaire, dans ce cas, de savoir quel robot est tombé en panne. Il suffit de repérer la tâche non achevée, de passer à l'exécution de la suivante, si la tâche n'est pas blocante, ou dans le cas contraire, de recommencer la tâche et de passer ensuite en séquence. On entend par tâche blocante : une tâche entraînant par exemple la perte d'un objet à insérer.

Pour diffuser son état au monde extérieur le programme émet un signal chaque fois qu'une tâche est correctement terminée. L'architecture du programme consiste alors à mettre en parallèle un module chargé de mémoriser ce signal. Soit les modules suivants :

- module SCHEDULER : il correspond au fonctionnement nominal, identique à celui de la version 2. On réutilise par conséquent le code des quatre tâches principales et on insère des signaux de bonne complétion de tâche.

```
loop
    copymodule T1;
    emit OK_T1;
    copymodule T2;
    emit OK_T2;
    copymodule T3;
    emit OK_T3;
    copymodule T4;
    emit OK_T4;
end
```

- Module BREAK_DOWN_HANDLING : son rôle est d'attendre, soit un signal ▷OK_Ti, soit le signal ▷STRAT_SEQUENCE correspondant à l'occurence de l'un des deux signaux ▷Ri_KAPUT.

```
trap SUPERVISING_APPLICATION in
   loop
      await
          case OK_T1
          case STRAT_SEQUENCE do
              copymodule ONE_ROBOT_PERFORMING_ALL_THE_TASKS
              exit SUPERVISING_APPLICATION
```

```
        end;
        await
            case OK_T2
            case STRAT_SEQUENCE do
                copymodule ROB_GRIP_OBJECT2
                exit SUPERVISING_APPLICATION
        end;
        await
            case OK_T3
            case STRAT_SEQUENCE do
                copymodule ROB_NEAR_ASSEMBLY
                exit SUPERVISING_APPLICATION
        end;
        await
            case OK_T4
            case STRAT_SEQUENCE do
                exit SUPERVISING_APPLICATION
        end;
    end %loop end %trap
```

Cette structure permet d'illustrer l'emploi de la primitive **await-case**. Les branches de l'attente multiple sont exclusives. Les traitements réalisés sont différents : si le signal ▷OK_Ti (défini en local au niveau du module principal) est reçu, on passe en séquence et on attend un autre signal ▷OK_T(i+1). S'il ne l'est pas on sait exactement où le robot est tombé en panne et on effectue le traitement adéquat. Ensuite, on atteint un nouveau régime permanent correspondant au fonctionnement d'un seul robot.

Les modules ONE_ROBOT_PERFORMING_ALL_THE_TASKS, ROB_GRIP_OBJECT2, ROB_NEAR_ASSEMBLY effectuent respectivement les tâches $T1$-$T2$-$T3$-$T4$, $T2$-$T3$-$T4$, $T3$-$T4$. Le nombre d'états associé à chacun de ces modules est 27, 19, 7. Le module BREAK_DOWN_HANDLING compte 53 états, et le programme complet de récupération d'erreurs 149 états. Or ces deux derniers automates ne sont pas minimaux. L'utilisation d'un critère d'équivalence observationnelle et du système AUTO (on identifie deux états à partir desquels on peut réaliser les mêmes actions et arriver sur des états à nouveau équivalents) nous permet de réduire le nombre d'états de ces automates à, respectivement, 31 et 105 états. La non minimalité du code produit est inhérente à l'application programmée. En effet, le module BREAK_DOWN_HANDLING effectue selon les cas les séquences $T1$-$T2$-$T3$-$T4$, $T2$-$T3$-$T4$, $T3$-$T4$. Pour réaliser cet enchaînement, nous sommes donc réduits à appeler plusieurs fois les différentes tâches, ce qui revient à dupliquer le code. Ce problème est liée à la structure actuelle de la primitive ESTEREL **copymodule** qui effectue une recopie syntaxique du code défini dans un module.

Nous pouvons aboutir à un code produit minimal, au risque de diminuer la modularité et la lisibilité des programmes écrits. Celà rend la programmation beaucoup

moins intuitive, imposant un choix judicieux de combinaison de signaux et de codage pour que chaque sous-module soit effectivement minimal. Ce qui risque de masquer l'aspect logique des synchronisations entre modules. Il est important de remarquer que cette non minimalité du code n'est pas un aspect négatif de la programmation en ESTEREL. En effet, il existe un outil qui permet de transformer l'automate produit par compilation en un automate minimal. La modularité nous semble bien plus importante que la minimalité, et nous gardons notre première décomposition.

Remarque : Cette possibilité de reconfigurer le programme illustre cependant la flexibilité apportée par le langage ESTEREL. Nous obtenons deux programmes de structures totalement différentes conduisant à la même exécution. Nous utilisons cette flexibilité pour essayer de trouver les structures les plus représentatives des applications robotiques.

10.2 Recouvrement d'erreurs en parallèle

Nous contrôlons le déroulement de l'application en utilisant la même idée : un signal local est émis pour signaler la bonne complétion de chaque tâche, un module de contrôle utilise ensuite ces signaux pour effectuer un traitement d'erreur correct selon l'état des robots.

Se pose ici un problème supplémentaire. Nous devons en effet connaître l'état de chacun des deux robots au moment de la panne. On a alors recours à un module de mémorisation associé à chacun des deux modules de commande. Soit la décomposition correspondante :

- le module ROBOT1_MAINLY :

```
loop
    copymodule T1;
    emit OK_T1;
    copymodule T_1_2;
   [
    copymodule INSERTION_TASK;
    emit OK_T2;
  ||
    < dialogue avec ROBOT2 >
each R1_GO_TO_GRIP
```

- le module ROBOT2_MAINLY :

```
loop
    <dialogue avec ROBOT1
     saisie de l'assemblage>
    emit OK_T3;
    copymodule T4;
    emit OK_T4;
end
```

- Les modules `R1_STATE` et `R2_STATE` : leur corps réutilise trois fois le module de mémorisation décrit de façon à renseigner l'extérieur quant à l'état des manipulateurs (en attente ou effectuant une des tâches associées). Le code de la mémoire est le suivant :

```
do
  loop
      emit Ri_STATE;
  each WHAT_ARE_THEY_DOING
upto NO_MORE_ANSWER
```

- Le module `BREAK_DOWN_HANDLING`

  Nous utilisons, là encore, la primitive `await - case` pour réaliser une attente multiple sur les signaux `R1_KAPUT` et `R2_KAPUT`. Sur réception de l'un de ces deux signaux, on stoppe l'exécution du module commandant le robot défectueux et l'on s'enquièrt (dans le même instant) de l'état des deux robots. L'information d'état transite par l'intermédiaire des *signaux locaux* `OK_Ti` associés aux modules `Rj_STATE`. Nous n'utilisons pas, comme celà pourrait se faire avec des langages classiques, de variables ou de signaux valués sur lesquels on peut réaliser des tests. Nous remplaçons ici la structure de test "classique" *if - then - else*, par un test sur la *présence et l'absence* de signaux.

  On effectue en séquence le traitement d'erreur adéquat. Les robots peuvent être chacun dans trois états différents. Il y a donc 9 cas à examiner pour chaque robot tombant en panne, soit au total *18 configurations* possibles du système au moment d'une panne. Nous devons nous assurer que chacun de ces cas est correctement traité, et nous utilisons ici la primitive *present - then - else* permettant de réaliser des tests emboités. Pour plus de détails voir le code en annexe.

La mise au point du programme par simulation nous à permis de vérifier que le traitement effectué dans chacun des cas d'erreurs était satisfaisant. Cependant cette simulation n'a pas été des plus aisée car le nombre de signaux à gérer est important. Nous souhaitons vérifier, par la suite, que l'on peut retrouver, dans l'automate global, la spécification correcte du fonctionnement, analyser l'automate et détecter d'éventuelles anomalies. Nous utiliserons pour celà le système AUTO et les critères d'observation.

11 Conclusion

11.1 Discussion

L'exemple traité dans cet article illustre les particularités du langage ESTEREL. Conçu pour permettre une programmation aisée des sytèmes réactifs, il offre de nombreux avantages qui font généralement défaut aux différents langages de programmation pour robots [3].

L'une des raisons pour lesquelles ces langages ne font pas l'unanimité est qu'ils s'adressent souvent à un type particulier de robot, ce qui nuit à la modularité des programmes : changer de robot signifie changer de langage, devenir un expert du nouveau langage et enfin reprogrammer l'application. En outre la mise au point des programmes n'est pas des plus facile : les langages utilisés sont asynchrones, non déterministes et la conséquence de l'exécution des programmes n'est donc pas prévisible à l'avance. L'opérateur est obligé de "tester" le programme dans des conditions d'exécution diverses et variées pour connaître son comportement et vérifier son adéquation avec les spécifications du cahier des charges. Enfin la plupart de ces langages sont dépourvus de primitives de haut niveau permettant de gérer efficacement parallélisme et temps.

L'utilisation d'ESTEREL pour obtenir un automate d'application permet de se libérer des inconvénients précités : la première caractéristique du langage facilite l'écriture d'un programme. Elle correspond à la notion de *temps multiforme*. Il n'existe pas en ESTEREL de temps privilégié et tout événement répétitif definit un temps propre. Ce qui permet de considérer notre système robotique comme un système multi horloge à temps discret. Cette unification de la gestion du temps fait que tout signal reçoit un traitement identique et que les structures de contrôle ne diffèrent pas d'un signal à un autre. Pour manipuler le temps absolu, il suffit d'écouter un signal d'entrée particulier, produit par exemple par l'horloge système et baptisé Seconde.

L'*hypothèse de synchronisme* affirme que le système réagit instantanément aux signaux. Elle introduit un nouveau style de programmation lié à la diffusion des signaux et s'avère particulièrement efficace pour la prise en compte d'alarmes. Les effets des programmes de contrôle des robots dépendent de facteurs extérieurs habituellement mal contrôlés. L'utilisation d'ESTEREL permet une réaction instantanée des systèmes robotisés lors d'un changement des conditions de travail. L'emploi de la diffusion des signaux facilite l'écriture de programmes *modulaires* et flexibles à composants réutilisables autorisant une reconfiguration de la cellule. Ceci permet d'améliorer l'efficacité et les performances d'une installation robotique. L'émission de signaux "à la cantonade" permet une programmation plus souple. Lorsque nous souhaitons connaitre l'état de l'un des robots, nous générons de nouveaux signaux, dits signaux locaux, transparents pour le système robotisé au moment de l'exécution du programme. Les signaux `OK_Ti` correspondent *exactement* à la fin des tâches considérées. Ces signaux n'ont pas d'adresse particulière, les modules les attendant consomment l'information qu'ils transportent. Il n'est donc pas nécessaire de relier explicitemment les modules entre eux. De plus le compilateur ne génère pas de code supplémentaire lorsque l'on utilise de tels signaux : il détermine si un signal est présent ou absent au

cours de chaque transition de l'automate.

Une autre vertu de cette hypothèse est qu'elle rend l'automate *déterministe* ce qui simplifie la mise au point, et l'étude du comportement des programmes (tests reproductibles). On peut en effet compiler les programmes parallèles sous la forme d'un programme séquentiel unique. Ceci permet, d'une part de supprimer les temps de basculement de contexte qui sont habituellement perdus avec des langages asynchrones classiques et d'autre part de prévoir et même calculer les temps d'exécution, et prouver ainsi que l'hypothèse de temps nul est justifiée.

L'automate fini, obtenu après compilation, contient toutes les propriétés du programme. Des méthodes de vérification formelle (AUTO et AUTOGRAPH) de comportement utilisent cet automate et permettent de mettre au point un programme (preuves) *avant* son exécution. Ce qui est un avantage sur la plupart des langages classiques pour qui mise au point et exécution sont étroitement liées. La vérification se fait donc sans exécuter le programme, et l'interface avec le monde réel est réalisée lorsque toutes les propriétés du programme ont été vérifiées. A l'exécution, seuls les problèmes d'interface proprement dits seront à prendre en compte. La production d'un code intermédiaire, traduisible en langage cible de haut niveau, rend les programmes de séquencement totalement indépendants des robots à programmer.

11.2 Système de simulation

Nous étudions, dans le cadre des recherches sur les machines d'exécution du langage ESTEREL, les rapports entre l'automate d'application produit et les implémentations possibles de celui ci sur l'architecture logicielle particulière RHOMEO [4] [5]. Nous proposons ainsi un outil graphique de simulation qui aide au développement des programmes de gestion de cellules flexibles. L'outil présenté facilite la modélisation du procédé manufacturier (pièces et objets produits). Il utilise les possibilités offertes par le langage ESTEREL et par le logiciel C.A.O. RHOMEO développé dans le projet PRISME à l'I.N.R.I.A. Il permet de définir et choisir le (ou les) robot(s) à utiliser (géométrie du porteur, du volume atteignable, contraintes articulaires), la cellule robotisée (implantation relative des robots et de la périrobotique). On peut de plus simuler la tâche (contrôle des collisions, génération de trajectoire en environnement très encombré avec évitement d'obstacles) et visualiser la trajectoire au fur et à mesure des calculs. On teste ainsi la faisabilité des tâches.

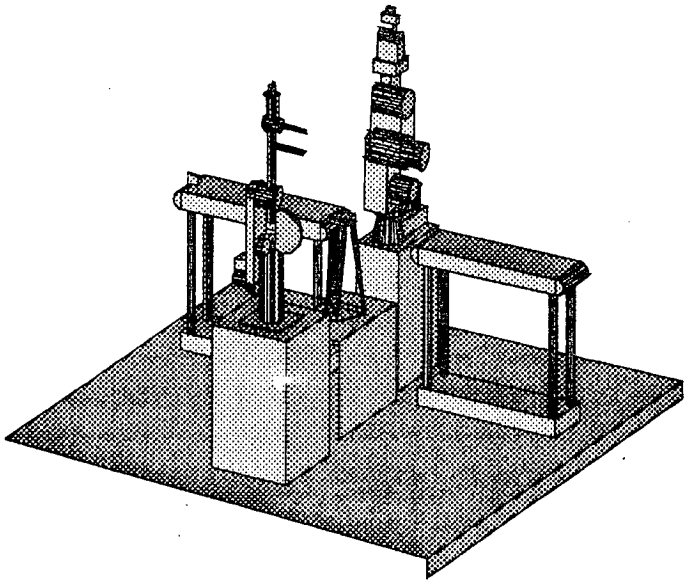
11.3 Perspectives

Nous avons présenté, dans cette étude,trois manières différentes d'aborder notre application. Nous envisagerons dans l'avenir un nouveau style de programmation : les objets (pièces à assembler) diffusent des requètes vers le monde extérieur qu'il s'agira de prendre en compte. Cette approche, de type serveur, devrait grandement faciliter la programmation de l'application. Ce n'est qu'après l'exploitation de cette idée que nous pourrons dégager une méthodologie effective de programmation (comparaison des

[4] Représentation Hiérarchiques des Objets et des Manipulateurs pour l'Evitement d'Obstacles

avantages et des inconvénients liés à chacune des méthodes envisagées). Le résultat de l'étude actuelle permet cependant d'éliminer, la première décomposition : ***un module-un objet physique.***

La notion de *modules* disponible en ESTEREL devrait permettre de réaliser un interpreteur de haut niveau. On peut en effet envisager de créer une bibliothèque de modules, chacun étant dédié à une tâche robot particulière (génération de trajectoire ...) et avoir ensuite recours à une programmation implicite. En partant d'une représentation abstraite des tâches ("prend un stylo plume et ecris ton nom"), on aboutirait à une représentation concrète d'un programme robotique.



Références

[1] "Technique de la robotique. Tome 1. architectures et commandes" sous la direction de J.D. Boissonnat, B. Faverjon, J.P. Merlet. Edition Hermés, Paris 1988.

[2] "Apport du Génie Logiciel et de la conception par objets à la programmation des tâches robotisées", Marie Claude Thomas. Thèse de Doctorat d'Etat mention Informatique. Université de Nice-Sophia Antipolis-LISAN-CNRS, Fevrier 1989.

[3] "Languages for Sensor-Based Control in Robotics" Computer and Systems Sciences, Vol 29. NATO ASI series.

[4] "Manipulateurs parallèles" J.P. Merlet. Rapport INRIA numéro 1003. Mars 1989.

[5] "Hierarchical Object Models For Efficient Anti-Collision Algorithms" B. Faverjon. 1989 IEEE International Conference on Robotics and Automation. May 14-19 1989.

[6] "Un Réseau Local Temps-Réel pour la Robotique", M. Meija, D. Simon, JJ. Borreliy, IRISA.

[7] "Contribution à la Conception d'un Réseau Local Temps-Réel pour la Robotique", M. Meija. Thèse d'Informatique, Université de Rennes I (1989).

[8] " A Distributed System for Robotic Applicatoins", P. Belmans, JJ. Borrelly, M. Meija. IRISA .

[9] "Synchronous Programming of Reactive Systems : an Introduction to ESTEREL". G. Berry, Ph. Couronné, G. Gonthier. Rapport INRIA 647

[10] "The Synchronous Programming Language : Design, Semantics, Implementation". G. Berry, G. Gonthier. Rapport INRIA 842.

[11] "ESTEREL System Manuals". G. Berry, F. Boussinot, Ph. Couronné, G. Gonthier. Rapport Ecole des Mines/INRIA (1986).

[12] "ESTEREL Programming Examples", G. Berry, F. Boussinot, Ph. Couronné, G. Gonthier. Rapport Ecole des Mines/INRIA (1986).

[13] "Esterel v3 Programming Examples : Programming a Digital Watch". G. Berry. Rapport Ecole des Mines (1989).

[14] "Incremental Development of an HDLC protocol in ESTEREL", G. Berry, G. Gonthier. A paraitre.

[15] "Sémantiques et modèles d'exécution des langages réactifs synchrones; application à ESTEREL. G. Gonthier. Thèse d'informatique, Université d'Orsay (1988).

[16] "An AUTOGRAPH primer", V. Roy, R. de Simone. à paraître.

[17] "Aboard AUTO", R. de Simone, D. Vergamini. Rapport INRIA à paraître.

A Les modules communs aux différentes applications

A.1 Les tâches propres au ROBOT1

```
module R1_FOLLOWING_A_TRAJ_TOWARD_B:
input
     OBJECT_VICINITY_R1B;
output
     R1_TRAJ_TOWARD_B;

trap MOVE_B in          % toward th BELT number 1
          emit R1_TRAJ_TOWARD_B
        ||
          await OBJECT_VICINITY_R1B; exit MOVE_B
end;
.

module GRIPPING_ON_BELT:
input
     OBJECT_IN_GRIP;
output
GRIPPING_ON_BELT;

do
      emit GRIPPING_ON_BELT              % macro task: shifting slowly
                                          % under sensor control
                                          % stopping the robot
                                          % gripping
                                          % testing

upto OBJECT_IN_GRIP
.

module TRAJ_TRACKING_TOWARD_LEFT_HAND:
input
     LEFT_HAND_VICINITY;
output
TRAJECTORY_TRACKING_TOWARD_LEFT_HAND;
do
    emit TRAJECTORY_TRACKING_TOWARD_LEFT_HAND

upto LEFT_HAND_VICINITY
.

module PUTTING_THE_OBJECT_IN_LEFT_HAND:
input
     OBJECT_IS_IN_LEFT_HAND;
output
PUTTING_THE_OBJECT_IN_LEFT_HAND;

do
```

```
    emit PUTTING_THE_OBJECT_IN_LEFT_HAND
upto OBJECT_IS_IN_LEFT_HAND
.

module INSERTING_THE_OBJECT_IN_LEFT_HAND:
input
     OBJECT_IS_INSERTED;
output
      INSERTING_THE_OBJECT_IN_LEFT_HAND;

do
    emit INSERTING_THE_OBJECT_IN_LEFT_HAND

upto OBJECT_IS_INSERTED
.
```

A.2 Les tâches propres au ROBOT2

```
module R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND:
input
     ASSEMBLY_VICINITY_R2;
output
        R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND;

trap OBJECT_IN_LEFT_HAND in

       emit R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND
     ||
       await ASSEMBLY_VICINITY_R2; exit OBJECT_IN_LEFT_HAND;
end;
.

module GRIPPING_THE_ASSEMBLY:
input
    ASSEMBLY_GRIPPED;
output
      GRIPPING_THE_ASSEMBLY;

do
    emit  GRIPPING_THE_ASSEMBLY
upto ASSEMBLY_GRIPPED;
.

module MOVING_TOWARD_INSERTING_PLACE:
input
    INSERTION_VICINITY;
output
     MOVING_TOWARD_INSERTING_PLACE;
do
    emit MOVING_TOWARD_INSERTING_PLACE
```

```
upto INSERTION_VICINITY;
.

module INSERTING_THE_ASSEMBLY:

input
     ASSEMBLY_INSERTED;

output
    INSERTING_THE_ASSEMBLY;
do
   emit INSERTING_THE_ASSEMBLY
upto ASSEMBLY_INSERTED;
.
```

A.3 La tâche du convoyeur

```
module BELT_CONTROL:
input
     VICINITY_OF_B;
output
     BELT_MOTION;
% Normal Sequential Behaviour of Belt number 1
   do
      emit BELT_MOTION % toward the environement
   upto VICINITY_OF_B
.
```

A.4 Les tâches propres a la main gauche

```
module LOCKING_THE_OBJECT:
input
      OBJECT_LOCKED;
output
      LOCKING_THE_OBJECT;

do
   emit LOCKING_THE_OBJECT
upto OBJECT_LOCKED
.

module LH_INSERTION:
input
     OBJECT_IS_INSERTED;
output
      INSERTION_TASK;
do
   emit INSERTION_TASK
upto OBJECT_IS_INSERTED
.
```

```
module UNLOCKING_THE_ASSEMBLY:
input
     UNLOCKED;
output
      UNLOCKING_THE_ASSEMBLY;

do
   emit UNLOCKING_THE_ASSEMBLY
upto UNLOCKED
.
```

A.5 Les tâches propres aux deux robots (récupération d'erreurs)

```
module R_FOLLOWING_A_TRAJ_TOWARD_B:
input
     OBJECT_VICINITY_RB;
output
     R_TRAJ_TOWARD_B;

trap MOVE_B in          % toward  BELT
          emit R_TRAJ_TOWARD_B
        ||
          await OBJECT_VICINITY_RB;
          exit MOVE_B
end
.

module R_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND:
input
     ASSEMBLY_VICINITY_R;
output
        R_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND;

trap OBJECT_IN_LEFT_HAND in
       emit R_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND
     ||
       await ASSEMBLY_VICINITY_R;
       exit OBJECT_IN_LEFT_HAND
end
.
```

B Un module-un manipulateur

```
module SUPERUSER:
input
     X_R1_INITIALISED,
     X_R2_INITIALISED,
     X_B1_INITIALISED,
     X_B2_INITIALISED,
     X_LH_INITIALISED,
     X_VICINITY_OF_B1,
     X_VICINITY_OF_B2,
     X_OBJECT_VICINITY_R1B1,
     X_OBJECT_VICINITY_R1B2,
     X_OBJECT1_IN_GRIP,
     X_OBJECT2_IN_GRIP,
     X_LH_VICINITY_FOR_O1,
     X_LH_VICINITY_FOR_O2,
     X_OBJECT_IS_IN_LEFT_HAND,
     X_OBJECT_LOCKED,
     X_OBJECT_IS_INSERTED,
     X_UNLOCKED,
     X_OBJECT_INSERTED,
     X_ASSEMBLY_VICINITY_R2,
     X_ASSEMBLY_GRIPPED,
     X_INSERTION_VICINITY,
     X_ASSEMBLY_INSERTED;
output
     X_INIT_BELT1,
     X_INIT_BELT2,
     X_INIT_LH,
     X_INIT_R1,
     X_INIT_R2,
     X_BELT1_MOTION,
     X_BELT2_MOTION,
     X_R1_TRAJ_TOWARD_B1,
     X_R1_TRAJ_TOWARD_B2,
     X_GRIPPING_ON_BELT1,
     X_GRIPPING_ON_BELT2,
     X_FIRST_TRAJ_TOWARD_LH,
     X_SECOND_TRAJ_TOWARD_LH,
     X_PUTTING_THE_OBJECT_IN_LEFT_HAND,
     X_LOCKING_THE_OBJECT,
     X_INSERTING_THE_OBJECT_IN_LEFT_HAND,
     X_INSERTION_TASK,
     X_UNLOCKING_THE_ASSEMBLY,
     X_R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND,
     X_GRIPPING_THE_ASSEMBLY,
     X_MOVING_TOWARD_INSERTING_PLACE,
     X_INSERTING_THE_ASSEMBLY;
% Declaration of Local Signals
```

```
signal
     NO_PIECE_TO_GRIP_ON_B1,
     B1_CAN_MOVE_AGAIN,
     B1_NEW_MOTION,
     IS_B1_READY,
     YES_B1_IS_READY,
     NO_PIECE_TO_GRIP_ON_B2,
     B2_CAN_MOVE_AGAIN,
     B2_NEW_MOTION,
     STARTING_INSERTION,
     IS_B2_READY,
     YES_B2_IS_READY,
     IS_LH_READY_TO_RECEIVE,
     LH_IS_READY_TO_RECEIVE,
     R1_PUT_OBJECT_IN_LEFT_HAND,
     LH_CAN_LOCK_O1,
     LH_CAN_RECEIVE,
     IS_LH_READY_TO_INSERT,
     LH_IS_READY_TO_INSERT,
     LH_CAN_INSERT,
     CAN_I_TAKE_ASSEMBLY,
     ASSEMBLY_CAN_BE_TAKEN,
     R2_TAKING_THE_ASSEMBLY
in

   copymodule BELT_CONTROLLER[
         signal X_VICINITY_OF_B1/VICINITY_OF_B,
                X_B1_INITIALISED/B_INITIALISED,
                X_IS_B1_READY/IS_B_READY,
                X_B1_NEW_MOTION/B_NEW_MOTION,
                X_INIT_BELT1/INIT_BELT,
                X_BELT1_MOTION/BELT_MOTION,
                NO_PIECE_TO_GRIP_ON_B1/
                NO_PIECE_TO_GRIP_ON_B,
                B1_CAN_MOVE_AGAIN/B_CAN_MOVE_AGAIN]
||
   copymodule ROBOT1_CONTROLLER
||
   copymodule LEFT_HAND_CONTROLLER
||
   copymodule BELT_CONTROLLER[
         signal X_VICINITY_OF_B2/VICINITY_OF_B,
                X_B2_INITIALISED/B_INITIALISED,
                IS_B2_READY/IS_B_READY,
                B2_NEW_MOTION/B_NEW_MOTION,
                X_INIT_BELT2/INIT_BELT,
                X_BELT2_MOTION/BELT_MOTION,
                NO_PIECE_TO_GRIP_ON_B2/
                NO_PIECE_TO_GRIP_ON_B,
                B2_CAN_MOVE_AGAIN/B_CAN_MOVE_AGAIN]
```

```
|| copymodule  ROBOT2_CONTROLLER
end   % signal
.

module ROBOT1_CONTROLLER:

  emit INIT_R1; %Initialisation (exec)
  await R1_INITIALISED;
  loop
       copymodule  R1_FOLLOWING_A_TRAJ_TOWARD_B [
            signal X_OBJECT_VICINITY_R1B1/OBJECT_VICINITY_R1B,
                   YES_B1_IS_READY/YES_B_IS_READY,
                   B1_CAN_MOVE_AGAIN/B_CAN_MOVE_AGAIN,
                   X_R1_TRAJ_TOWARD_B1/R1_TRAJ_TOWARD_B,
                   IS_B1_READY/IS_B_READY];
       copymodule GRIPPING_ON_BELT [
            signal X_GRIPPING_ON_BELT1/GRIPPING_ON_BELT,
                   X_OBJECT1_IN_GRIP/OBJECT_IN_GRIP];
       emit B1_NEW_MOTION;  % toward  Belt1
       copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND[
            signal X_LH_VICINITY_FOR_O1/LEFT_HAND_VICINITY,
                   X_FIRST_TRAJ_TOWARD_LH/
                   TRAJECTORY_TRACKING_TOWARD_LEFT_HAND];
       emit IS_LH_READY_TO_RECEIVE;
       await immediate LH_CAN_RECEIVE ;
       emit R1_PUT_OBJECT_IN_LEFT_HAND;
       copymodule PUTTING_THE_OBJECT_IN_LEFT_HAND;
       emit LH_CAN_LOCK_O1;
       copymodule  R1_FOLLOWING_A_TRAJ_TOWARD_B [
            signal X_OBJECT_VICINITY_R1B2/OBJECT_VICINITY_R1B,
                   YES_B2_IS_READY/YES_B_IS_READY,
                   B2_CAN_MOVE_AGAIN/B_CAN_MOVE_AGAIN,
                   X_R1_TRAJ_TOWARD_B2/R1_TRAJ_TOWARD_B,
                   IS_B2_READY/IS_B_READY];
       copymodule GRIPPING_ON_BELT [
            signal X_GRIPPING_ON_BELT2/GRIPPING_ON_BELT,
                   X_OBJECT2_IN_GRIP/OBJECT_IN_GRIP];
       emit B2_NEW_MOTION;
       copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND[
            signal X_LH_VICINITY_FOR_O2/LEFT_HAND_VICINITY,
                   X_SECOND_TRAJ_TOWARD_LH/
                   TRAJECTORY_TRACKING_TOWARD_LEFT_HAND];
       emit IS_LH_READY_TO_INSERT;
await immediate LH_CAN_INSERT;
       emit STARTING_INSERTION;
       copymodule INSERTING_THE_OBJECT_IN_LEFT_HAND
  end % loop
.

module ROBOT2_CONTROLLER:
```

```
emit INIT_R2;
await R2_INITIALISED;
loop
   copymodule R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND;
   emit CAN_I_TAKE_ASSEMBLY;
   await immediate ASSEMBLY_CAN_BE_TAKEN;
   emit R2_TAKING_THE_ASSEMBLY;
   copymodule GRIPPING_THE_ASSEMBLY;  % the LH is ready to use
   copymodule MOVING_TOWARD_INSERTING_PLACE;
   copymodule INSERTING_THE_ASSEMBLY;
end
.

module LEFT_HAND_CONTROLLER:
loop
    emit X_INIT_LH;   % putting the Left Hand in right
% position ready to receive objects
    await X_LH_INITIALISED;
    trap NOW_RECEIVING in  % to provide causality errors
    [
      loop
        emit  LH_CAN_RECEIVE
      each  IS_LH_READY_TO_RECEIVE
    ||
      await immediate R1_PUT_OBJECT_IN_LEFT_HAND do   % from the Rob1
         exit NOW_RECEIVING
      end
    ]
    end;
    await  LH_CAN_LOCK_O1;
    copymodule LOCKING_THE_OBJECT;
    trap NOW_INSERTING in
    [
      loop
        emit  LH_CAN_INSERT
      each  IS_LH_READY_TO_INSERT
    ||
      await immediate STARTING_INSERTION do     % from the Rob2
         exit NOW_INSERTING
      end
    ]
    end; %trap
    copymodule LH_INSERTION;
    copymodule UNLOCKING_THE_ASSEMBLY;
    trap NOW_EMPTY in
    [
      loop
        emit  ASSEMBLY_CAN_BE_TAKEN
      each  CAN_I_TAKE_ASSEMBLY
```

```
    ||
      await immediate R2_TAKING_THE_ASSEMBLY do      % from R2
         exit NOW_EMPTY
      end
    ]
    end %trap
end %loop
```

C Programme séquentiel

```
module SCHEDULER:
input
     X_R1_INITIALISED,
     X_R2_INITIALISED,
     X_OBJECT_VICINITY_R1B1,
     X_VICINITY_OF_B1,
     X_OBJECT1_IN_GRIP,
     X_LH_VICINITY_FOR_O1,
     X_LH_INITIALISED,
     X_OBJECT_IS_IN_LEFT_HAND,
     X_OBJECT_VICINITY_R1B2,
     X_OBJECT_LOCKED,
     X_VICINITY_OF_B2,
     X_OBJECT2_IN_GRIP,
     X_LH_VICINITY_FOR_O2,
     X_OBJECT_IS_INSERTED,
     X_ASSEMBLY_VICINITY_R2,
     X_UNLOCKED,
     X_ASSEMBLY_GRIPPED,
     X_INSERTION_VICINITY,
     X_ASSEMBLY_INSERTED;
output
       X_INIT_R1,
       X_INIT_R2,
       X_R1_TRAJ_TOWARD_B1,
       X_BELT1_MOTION,
       X_GRIPPING_ON_BELT1,
       X_FIRST_TRAJ_TOWARD_LH,
       X_INIT_LH,
       X_PUTTING_THE_OBJECT_IN_LEFT_HAND,
       X_R1_TRAJ_TOWARD_B2,
       X_LOCKING_THE_OBJECT,
       X_BELT2_MOTION,
       X_GRIPPING_ON_BELT2,
       X_SECOND_TRAJ_TOWARD_LH,
       X_INSERTING_THE_OBJECT_IN_LEFT_HAND,
       X_INSERTION_TASK,
       X_R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND,
       X_UNLOCKING_THE_ASSEMBLY,
       X_GRIPPING_THE_ASSEMBLY,
       X_MOVING_TOWARD_INSERTING_PLACE,
       X_INSERTING_THE_ASSEMBLY;

% Initialisation phase for ROBOT1 & ROBOT2.
[
    emit X_INIT_R1;
    await X_R1_INITIALISED;
  ||
```

```
    emit X_INIT_R2;
    await X_R2_INITIALISED;
];
% End of initialisation => starting robot1 and belt1 motion.
  loop
       [
         copymodule  R1_FOLLOWING_A_TRAJ_TOWARD_B [
                  signal X_OBJECT_VICINITY_R1B1/OBJECT_VICINITY_R1B,
                         X_R1_TRAJ_TOWARD_B1/R1_TRAJ_TOWARD_B];
       ||
         copymodule BELT_CONTROL[
                  signal X_VICINITY_OF_B1/VICINITY_OF_B,
                         X_BELT1_MOTION/BELT_MOTION];
       ];
% Robot1 can grip Object1
        copymodule GRIPPING_ON_BELT [
                  signal X_GRIPPING_ON_BELT1/GRIPPING_ON_BELT,
                         X_OBJECT1_IN_GRIP/OBJECT_IN_GRIP];
% initialising left Hand. Traj tracking for Robo1 1
        [
         copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND[
                   signal X_LH_VICINITY_FOR_O1/LEFT_HAND_VICINITY,
                          X_FIRST_TRAJ_TOWARD_LH/
                           TRAJECTORY_TRACKING_TOWARD_LEFT_HAND];
        ||
         emit X_INIT_LH; % putting the Left Hand in right position
% ready to receive objects
         await X_LH_INITIALISED;
        ];
%Robot1 can put Object1 in LeftHand.
        copymodule PUTTING_THE_OBJECT_IN_LEFT_HAND;
         [
         copymodule LOCKING_THE_OBJECT;
        ||
             [copymodule  R1_FOLLOWING_A_TRAJ_TOWARD_B [
                  signal X_OBJECT_VICINITY_R1B2/OBJECT_VICINITY_R1B,
                         X_R1_TRAJ_TOWARD_B2/R1_TRAJ_TOWARD_B];
            ||
              copymodule BELT_CONTROL[
                  signal X_VICINITY_OF_B2/VICINITY_OF_B,
                         X_BELT2_MOTION/BELT_MOTION];
             ];
               copymodule GRIPPING_ON_BELT [
                   signal X_GRIPPING_ON_BELT2/GRIPPING_ON_BELT,
                          X_OBJECT2_IN_GRIP/OBJECT_IN_GRIP];
               copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND[
                    signal X_LH_VICINITY_FOR_O2/LEFT_HAND_VICINITY,
                           X_SECOND_TRAJ_TOWARD_LH/
                           TRAJECTORY_TRACKING_TOWARD_LEFT_HAND];
        ];
```

```
% insertion won't start before the locking phase ends (that is to say
% LH is ready for insertion
         [
          copymodule INSERTING_THE_OBJECT_IN_LEFT_HAND
        ||                              %cooperation between
          copymodule LH_INSERTION;      %LH & R1 for insertion
        ||
          copymodule R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND;
         ];                                    %R2's motion.
         copymodule UNLOCKING_THE_ASSEMBLY; %LH's job
         copymodule GRIPPING_THE_ASSEMBLY; %Rob2 can take the assembly
         copymodule MOVING_TOWARD_INSERTING_PLACE; %and bring it where
         copymodule INSERTING_THE_ASSEMBLY; %it will be inserted.
end %loop
```

D Le pipeline

```
module ASSEMBLY:

% Here see the declaration of the last programm
% In addition, we've got a declaration of local signals.

signal
      R1_GO_TO_GRIP_OBJECT1,
      CAN_R2_START,
      R2_CAN_START_ITS_MOTION,
      R2_MOVED,
      CAN_LH_UNLOCK,
      LH_CAN_UNLOCK,
      LH_NOW_UNLOCKED
in
% Initialisation phase : The application will not start if one
% (or both) robot(s) is (are) not ready.
[
    emit X_INIT_R1;
    await X_R1_INITIALISED;
  ||
    emit X_INIT_R2;
    await X_R2_INITIALISED;
];
% End of initialisation
% starting ROBOT1's jobs
% cooperation between different objects : Belt1 & Belt2 & Left Hand.
  [
   copymodule ROBOT1MAINLY;
||
   copymodule ROBOT2MAINLY;
  ]
end %signal
.


module ROBOT1MAINLY:

% DECLARATION of input and output signals

loop        %loop1

% starting ROBOT1 & Belt1
       [
         copymodule  R1_FOLLOWING_A_TRAJ_TOWARD_B [
               signal X_OBJECT_VICINITY_R1B1/OBJECT_VICINITY_R1B,
                      X_R1_TRAJ_TOWARD_B1/R1_TRAJ_TOWARD_B];
       ||
         copymodule BELT_CONTROL[
```

```
              signal X_VICINITY_OF_B1/VICINITY_OF_B,
                     X_BELT1_MOTION/BELT_MOTION];
      ];
% Robot1 can grip Object1
       copymodule GRIPPING_ON_BELT [
            signal X_GRIPPING_ON_BELT1/GRIPPING_ON_BELT,
                   X_OBJECT1_IN_GRIP/OBJECT_IN_GRIP];
% initialising left Hand. Tra tracking for Robo1 1
       [
         copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND[
               signal X_LH_VICINITY_FOR_O1/LEFT_HAND_VICINITY,
                      X_FIRST_TRAJ_TOWARD_LH/
                      TRAJECTORY_TRACKING_TOWARD_LEFT_HAND];
       ||
         emit X_INIT_LH; % putting the Left Hand in right position
                         % ready to receive objects
         await X_LH_INITIALISED;
       ];
%Robot1 can put Object1 in LeftHand.
       copymodule PUTTING_THE_OBJECT_IN_LEFT_HAND;
        [
         copymodule LOCKING_THE_OBJECT;
       ||
            [copymodule  R1_FOLLOWING_A_TRAJ_TOWARD_B [
                  signal X_OBJECT_VICINITY_R1B2/OBJECT_VICINITY_R1B,
                         X_R1_TRAJ_TOWARD_B2/R1_TRAJ_TOWARD_B];
           ||
             copymodule BELT_CONTROL[
                  signal X_VICINITY_OF_B2/VICINITY_OF_B,
                         X_BELT2_MOTION/BELT_MOTION];
            ];
              copymodule GRIPPING_ON_BELT [
                   signal X_GRIPPING_ON_BELT2/GRIPPING_ON_BELT,
                          X_OBJECT2_IN_GRIP/OBJECT_IN_GRIP];
              copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND[
                   signal X_LH_VICINITY_FOR_O2/LEFT_HAND_VICINITY,
                          X_SECOND_TRAJ_TOWARD_LH/
                          TRAJECTORY_TRACKING_TOWARD_LEFT_HAND];
        ];
% insertion won't start before the locking phase ends
% (that is to say LH is ready for insertion)
        [
         copymodule INSERTING_THE_OBJECT_IN_LEFT_HAND;
       ||                          % cooperation between
         copymodule LH_INSERTION;
       ||                          % LH & R1 for insertion
         trap R2_MOTION in
           [
            loop
              emit R2_CAN_START_ITS_MOTION;
```

```
            each CAN_R2_START
           ||
            await  R2_MOVED; exit R2_MOTION;
           ]
          end;
        ];
           trap LH_NOW_WORKING in
             [
              loop
                  emit LH_CAN_UNLOCK;
               each CAN_LH_UNLOCK
             ||
              await LH_NOW_UNLOCKED; exit LH_NOW_WORKING;
              ]
            end;
each R1_GO_TO_GRIP_OBJECT1   %loop1 : parallelism to optimise.
.

module ROBOT2MAINLY:

% DECALARATION OF SIGNALS

loop
         emit CAN_R2_START;
         await  immediate R2_CAN_START_ITS_MOTION;
         copymodule R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND;
         emit R2_MOVED;                % R2's motion.
         emit CAN_LH_UNLOCK;
         await immediate LH_CAN_UNLOCK;
         copymodule UNLOCKING_THE_ASSEMBLY; % LH's job
         emit LH_NOW_UNLOCKED;
         copymodule GRIPPING_THE_ASSEMBLY; % R2 can take the assembly
         emit R1_GO_TO_GRIP_OBJECT1;       % R1 can perform its
                                           % tasks again and bring
         copymodule MOVING_TOWARD_INSERTING_PLACE; %  it where
         copymodule INSERTING_THE_ASSEMBLY;  % it will be inserted.
end %loop
.
```

E Traitement d'exception

E.1 Séquentiel

```
module ASSEMBLY_BREAKING_DOWN:
input
     START_APPLICATION,
     X_R1_INITIALISED,
     X_R2_INITIALISED,
     X_OBJECT_VICINITY_R1B1,
     X_VICINITY_OF_B1,
```

```
      X_OBJECT1_IN_GRIP,
      X_LH_VICINITY_FOR_O1,
      X_LH_INITIALISED,
      X_OBJECT_IS_IN_LEFT_HAND,
      X_OBJECT_VICINITY_R1B2,
      X_OBJECT_LOCKED,
      X_VICINITY_OF_B2,
      X_OBJECT2_IN_GRIP,
      X_LH_VICINITY_FOR_O2,
      X_OBJECT_IS_INSERTED,
      X_ASSEMBLY_VICINITY_R2,
      X_UNLOCKED,
      X_ASSEMBLY_GRIPPED,
      X_INSERTION_VICINITY,
      X_ASSEMBLY_INSERTED,
      R1_KAPUT,
      R2_KAPUT,
      OBJECT_VICINITY_RB1,
      OBJECT_VICINITY_RB2,
      ASSEMBLY_VICINITY_R,
      R_VICINITY_OF_B1,
      R_OBJECT1_IN_GRIP,
      R_LH_VICINITY_FOR_O1,
      R_LH_INITIALISED,
      R_OBJECT_IS_IN_LEFT_HAND,
      R_OBJECT_LOCKED,
      R_VICINITY_OF_B2,
      R_OBJECT2_IN_GRIP,
      R_LH_VICINITY_FOR_O2,
      R_OBJECT_IS_INSERTED,
      R_UNLOCKED,
      R_ASSEMBLY_GRIPPED,
      R_INSERTION_VICINITY,
      R_ASSEMBLY_INSERTED;
output
        PROBLEME_AT_INITIALISATION,
        LETS_START_AGAIN,
        X_INIT_R1,
        X_INIT_R2,
        R_TRAJ_TOWARD_B1,
        X_BELT1_MOTION,
        X_GRIPPING_ON_BELT1,
        X_FIRST_TRAJ_TOWARD_LH,
        X_INIT_LH,
        X_PUTTING_THE_OBJECT_IN_LEFT_HAND,
        R_TRAJ_TOWARD_B2,
        X_LOCKING_THE_OBJECT,
        BELT2_MOTION,
        X_GRIPPING_ON_BELT2,
        X_SECOND_TRAJ_TOWARD_LH,
```

```
        X_INSERTING_THE_OBJECT_IN_LEFT_HAND,
        X_INSERTION_TASK,
        X_UNLOCKING_THE_ASSEMBLY,
        X_GRIPPING_THE_ASSEMBLY,
        X_MOVING_TOWARD_INSERTING_PLACE,
        X_INSERTING_THE_ASSEMBLY,
        TOTAL_BREAK_DOWN,
        X_R1_TRAJ_TOWARD_B1,
        X_R1_TRAJ_TOWARD_B2,
        X_R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND,
        BOTH_ROBOTS_ARE_OUT_OF_ORDER,
        R_BELT1_MOTION,
        R_GRIPPING_ON_BELT1,
        R_FIRST_TRAJ_TOWARD_LH,
        R_INIT_LH,
        R_PUTTING_THE_OBJECT_IN_LEFT_HAND,
        R_LOCKING_THE_OBJECT,
        R_BELT2_MOTION,
        R_GRIPPING_ON_BELT2,
        R_SECOND_TRAJ_TOWARD_LH,
        R_INSERTING_THE_OBJECT_IN_LEFT_HAND,
        R_INSERTION_TASK,
        R_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND,
        R_UNLOCKING_THE_ASSEMBLY,
        R_GRIPPING_THE_ASSEMBLY,
        R_MOVING_TOWARD_INSERTING_PLACE,
        R_INSERTING_THE_ASSEMBLY;
signal
     OK_OBJECT1,
     OK_PRE_INSERTION,
     OK_ASSEMBLY_GRIPPED,
     OK_ASSEMBLY_INSERTED,
     START_SEQUENCE
in

 every START_APPLICATION do
      trap BIG_PROBLEM in
       [
         trap ONE_ROBOT_IS_WORKING in
            await
                 case R1_KAPUT
                 case R2_KAPUT
            end;
            emit START_SEQUENCE;
            exit ONE_ROBOT_IS_WORKING
     ||
            copymodule SCHEDULER; exit BIG_PROBLEM
        end       %trap ONE_ROBOT_IS_WORKING
     ||
       trap OTHER_ROB_BREAKING_DOWN in
```

```
          copymodule BREAK_DOWN_HANDLING;
          exit OTHER_ROB_BREAKING_DOWN
    ||
          [
            await R1_KAPUT
          ||
            await R2_KAPUT
          ];
          emit BOTH_ROBOTS_ARE_OUT_OF_ORDER;
          exit BIG_PROBLEM
       end     %trap OTHER_ROB_BREAKING_DOWN
      ];
      trap LAST_ROBOT_OUT_OF_ORDER in
        [
          loop
              copymodule ONE_ROBOT_PERFORMING_ALL_THE_TASKS;
          end
       ||
          await
             case R1_KAPUT
             case R2_KAPUT
          end;
          exit LAST_ROBOT_OUT_OF_ORDER
        ]
       handle LAST_ROBOT_OUT_OF_ORDER do
    emit BOTH_ROBOTS_ARE_OUT_OF_ORDER;
       end %trap LAST_ROBOT_OUT_OF_ORDER
     handle BIG_PROBLEM do
          emit LETS_START_AGAIN;
     end  % trap INITIAL_PROBLEM

  end  %every
end  % signal


module SCHEDULER:
  trap BREAK_DOWN in
% Initialisation phase.For ROBOT1 & ROBOT2.
    [
      emit INIT_R1;
      await
           case X_R1_INITIALISED
           case R1_KAPUT do exit BREAK_DOWN
           case R2_KAPUT  do exit BREAK_DOWN
      end
  ||
      emit INIT_R2;
      await
           case X_R2_INITIALISED
           case R1_KAPUT do exit BREAK_DOWN
```

```
            case R2_KAPUT do exit BREAK_DOWN
      end
     ];
% End of initialisation => starting robot1 and belt1 motion.
      loop
            [
              copymodule  R1_FOLLOWING_A_TRAJ_TOWARD_B [
                    signal X_OBJECT_VICINITY_R1B1/
                           OBJECT_VICINITY_R1B,
                           X_R1_TRAJ_TOWARD_B1/R1_TRAJ_TOWARD_B];
          ||
              copymodule BELT_CONTROL[
                    signal X_VICINITY_OF_B1/VICINITY_OF_B,
                           X_BELT1_MOTION/BELT_MOTION];
            ];
% Robot1 can grip Object1
             copymodule GRIPPING_ON_BELT [
                  signal X_GRIPPING_ON_BELT1/GRIPPING_ON_BELT,
                         X_OBJECT1_IN_GRIP/OBJECT_IN_GRIP];
% initialising left Hand. Traj tracking for Robo1 1
           [
              copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND[
                    signal X_LH_VICINITY_FOR_O1/LEFT_HAND_VICINITY,
                           X_FIRST_TRAJ_TOWARD_LH/
                           TRAJECTORY_TRACKING_TOWARD_LEFT_HAND];
         ||
            emit X_INIT_LH;
  % putting the Left Hand in right position
% ready to receive objects
            await X_LH_INITIALISED;
           ];
%Robot1 can put Object1 in LeftHand.
            copymodule PUTTING_THE_OBJECT_IN_LEFT_HAND;
            emit OK_OBJECT1;
           [
             copymodule LOCKING_THE_OBJECT;
         ||
              [
                 copymodule  R1_FOLLOWING_A_TRAJ_TOWARD_B [
                       signal X_OBJECT_VICINITY_R1B2/
                              OBJECT_VICINITY_R1B,
                              X_R1_TRAJ_TOWARD_B2/R1_TRAJ_TOWARD_B];
            ||
                 copymodule BELT_CONTROL[
                       signal X_VICINITY_OF_B2/VICINITY_OF_B,
                              X_BELT2_MOTION/BELT_MOTION];
              ];
                 copymodule GRIPPING_ON_BELT [
                       signal X_GRIPPING_ON_BELT2/GRIPPING_ON_BELT,
                              X_OBJECT2_IN_GRIP/OBJECT_IN_GRIP];
```

```
                copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND[
                     signal X_LH_VICINITY_FOR_02/LEFT_HAND_VICINITY,
                            X_SECOND_TRAJ_TOWARD_LH/
                            TRAJECTORY_TRACKING_TOWARD_LEFT_HAND];
          ];
      % insertion won't start before the locking phase ends
      % (that is to say LH is  ready for insertion
          [
            copymodule INSERTING_THE_OBJECT_IN_LEFT_HAND
%cooperation between
         ||
            copymodule LH_INSERTION;
                       %LH & R1 for insertion
          ];
           emit OK_PRE_INSERTION;
          copymodule R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND; % motion
          copymodule UNLOCKING_THE_ASSEMBLY;        %LH's job
          copymodule GRIPPING_THE_ASSEMBLY;
          emit OK_ASSEMBLY_GRIPPED;
          copymodule MOVING_TOWARD_INSERTING_PLACE;
          copymodule INSERTING_THE_ASSEMBLY;
   emit OK_ASSEMBLY_INSERTED;
      end  %loop
  handle  BREAK_DOWN do
emit PROBLEME_AT_INITIALISATION;
  end
.

module BREAK_DOWN_HANDLING:
trap SUPERVISING_APPLICATION in
    loop
        await
           case OK_OBJECT1
           case START_SEQUENCE do
%                    copymodule ONE_ROBOT_PERFORMING_ALL_THE_TASKS;
                 exit SUPERVISING_APPLICATION
        end;
        await
           case OK_PRE_INSERTION
           case START_SEQUENCE do
                    copymodule ROB_GRIP_OBJECT2;
                 exit SUPERVISING_APPLICATION
        end;
        await
           case OK_ASSEMBLY_GRIPPED
           case START_SEQUENCE do
                    copymodule ROB_NEAR_ASSEMBLY;
                 exit SUPERVISING_APPLICATION
        end;
        await
```

```
          case OK_ASSEMBLY_INSERTED
          case START_SEQUENCE do
               exit SUPERVISING_APPLICATION;
       end
    end    %loop
end     %trap SUPERVISING_APPLICATION
.

module ONE_ROBOT_PERFORMING_ALL_THE_TASKS:
     [
         copymodule  R_FOLLOWING_A_TRAJ_TOWARD_B [
               signal OBJECT_VICINITY_RB1/OBJECT_VICINITY_RB,
                      R_TRAJ_TOWARD_B1/R_TRAJ_TOWARD_B];
   ||
         copymodule BELT_CONTROL[
               signal R_VICINITY_OF_B1/VICINITY_OF_B,
                      R_BELT1_MOTION/BELT_MOTION];
     ];
        copymodule GRIPPING_ON_BELT [
             signal R_GRIPPING_ON_BELT1/GRIPPING_ON_BELT,
                    R_OBJECT1_IN_GRIP/OBJECT_IN_GRIP];
      [
          copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND[
               signal R_LH_VICINITY_FOR_O1/LEFT_HAND_VICINITY,
                      R_FIRST_TRAJ_TOWARD_LH/
                      TRAJECTORY_TRACKING_TOWARD_LEFT_HAND];
    ||
          emit R_INIT_LH;
  await R_LH_INITIALISED;
      ];
          copymodule PUTTING_THE_OBJECT_IN_LEFT_HAND[
               signal R_OBJECT_IS_IN_LEFT_HAND/
                      OBJECT_IS_IN_LEFT_HAND,
                      R_PUTTING_THE_OBJECT_IN_LEFT_HAND/
                      PUTTING_THE_OBJECT_IN_LEFT_HAND];
          copymodule ROB_GRIP_OBJECT2;
.

module ROB_GRIP_OBJECT2:
      [
          copymodule LOCKING_THE_OBJECT[
               signal R_OBJECT_LOCKED/OBJECT_LOCKED,
                      R_LOCKING_THE_OBJECT/LOCKING_THE_OBJECT];
    ||
          [
            copymodule  R_FOLLOWING_A_TRAJ_TOWARD_B [
                 signal OBJECT_VICINITY_RB2/OBJECT_VICINITY_RB,
                        R_TRAJ_TOWARD_B2/R_TRAJ_TOWARD_B];
         ||
            copymodule BELT_CONTROL[
```

```
                signal R_VICINITY_OF_B2/VICINITY_OF_B,
                       R_BELT2_MOTION/BELT_MOTION];
        ];
          copymodule GRIPPING_ON_BELT [
                signal R_GRIPPING_ON_BELT2/GRIPPING_ON_BELT,
                       R_OBJECT2_IN_GRIP/OBJECT_IN_GRIP];
           copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND[
                 signal R_LH_VICINITY_FOR_02/LEFT_HAND_VICINITY,
                        R_SECOND_TRAJ_TOWARD_LH/
                        TRAJECTORY_TRACKING_TOWARD_LEFT_HAND];
     ];
      [
        copymodule INSERTING_THE_OBJECT_IN_LEFT_HAND [
              signal R_OBJECT_IS_INSERTED/OBJECT_IS_INSERTED,
                     R_INSERTING_THE_OBJECT_IN_LEFT_HAND/
                     INSERTING_THE_OBJECT_IN_LEFT_HAND];
    ||
        copymodule LH_INSERTION[
              signal R_OBJECT_IS_INSERTED/OBJECT_IS_INSERTED,
                     R_INSERTION_TASK/INSERTION_TASK];
      ];
        copymodule ROB_NEAR_ASSEMBLY;

module ROB_NEAR_ASSEMBLY:
        copymodule R_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND;
        copymodule UNLOCKING_THE_ASSEMBLY[
              signal R_UNLOCKED/UNLOCKED,
                     R_UNLOCKING_THE_ASSEMBLY/
                     UNLOCKING_THE_ASSEMBLY];
        copymodule GRIPPING_THE_ASSEMBLY[
              signal R_ASSEMBLY_GRIPPED/ASSEMBLY_GRIPPED,
                     R_GRIPPING_THE_ASSEMBLY/GRIPPING_THE_ASSEMBLY];
        copymodule MOVING_TOWARD_INSERTING_PLACE[
              signal R_INSERTION_VICINITY/INSERTION_VICINITY,
                     R_MOVING_TOWARD_INSERTING_PLACE/
                     MOVING_TOWARD_INSERTING_PLACE];
        copymodule INSERTING_THE_ASSEMBLY[
              signal R_ASSEMBLY_INSERTED/ASSEMBLY_INSERTED,
                     R_INSERTING_THE_ASSEMBLY/
                     INSERTING_THE_ASSEMBLY];
```

E.2 Parallèle

Remarque : On donne ici uniquement les corps des différents modules.

```
module ASSEMBLY_BREAKING_DOWN:
   every START_APPLICATION do
      do
         trap ROBOT_BREAKING_DOWN in
            [
                copymodule ASSEMBLY;
            ||
                copymodule R1_STATE;
            ||
                copymodule R2_STATE;
            ||
                copymodule BREAK_DOWN_HANDLING;
                exit ROBOT_BREAKING_DOWN
            ]
         end; %trap ROBOT_BREAKING_DOWN
  % same sequence than when both robots are working sequentialy
         loop
            copymodule ONE_ROBOT_PERFORMING_ALL_THE_TASKS;
            copymodule ROB_GRIP_OBJECT2;
            copymodule ROB_NEAR_ASSEMBLY;
         end   % loop
      upto BOTH_ROBOTS_ARE_OUT_OF_ORDER
    ||
      copymodule AND [signal R1_KAPUT/FIRST,
                             R2_KAPUT/SECOND,
                             BOTH_ROBOTS_ARE_OUT_OF_ORDER/ORDER]
   end  % every
.

module ASSEMBLY:
    trap ROB1_STOPPED in
       await START1_SEQUENCE;
       exit ROB1_STOPPED
     ||
       copymodule ROBOT1MAINLY
    end % trap ROB1_STOPPED
  ||
    trap ROB2_STOPPED in
       await START2_SEQUENCE;
       exit ROB2_STOPPED
     ||
       copymodule ROBOT2MAINLY
    end % trap ROB2_STOPPED
.

module R1_STATE:
    loop
```

```
        do
          await immediate BEGIN_OBJECT1_TASK;
             loop
                 emit R1_PERFORMING_T1;
             each WHAT_ARE_THEY_DOING
         upto OK_OBJECT1;
         do
           loop
               emit R1_PERFORMING_T2;
           each WHAT_ARE_THEY_DOING
         upto OK_PRE_INSERTION;
         do
           loop
               emit R1_WAITING_FOR_R2
           each WHAT_ARE_THEY_DOING
         upto BEGIN_OBJECT1_TASK;
     end    %loop
.

module R2_STATE:
    loop
        trap R2_WAIT in
         [
          loop
               emit R2_WAITING_TO_START;
          each WHAT_ARE_THEY_DOING
        ||
        await immediate  R2_CAN_START_ITS_MOTION;
        exit R2_WAIT
         ]
        end;
        do
          loop
               emit R2_PERFORMING_T3;
           each WHAT_ARE_THEY_DOING
         upto OK_ASSEMBLY_GRIPPED;
        do
          loop
               emit R2_PERFORMING_T4;
          each WHAT_ARE_THEY_DOING
        upto  OK_ASSEMBLY_INSERTED;
    end    %loop
.

module BREAK_DOWN_HANDLING:
await
  case R1_KAPUT do
     emit START1_SEQUENCE;
     emit WHAT_ARE_THEY_DOING;
    [
```

```
    present R1_PERFORMING_T1
    then
        present R2_WAITING_TO_START
        else
              await OK_ASSEMBLY_INSERTED;
        end
    end
  ||
    present R1_PERFORMING_T2
    then
         present R2_PERFORMING_T4
         then
             await OK_ASSEMBLY_INSERTED;
         end;
         emit START2_SEQUENCE;
         copymodule ROB_GRIP_OBJECT2;
         copymodule ROB_NEAR_ASSEMBLY;
    end
  ||
    present R1_WAITING_FOR_R2
    then
         present R2_PERFORMING_T4
         then
             await OK_ASSEMBLY_INSERTED;
             emit START2_SEQUENCE;
             copymodule ROB_NEAR_ASSEMBLY;
         else
             do
               loop
                   emit LH_CAN_UNLOCK;
               each CAN_LH_UNLOCK
             upto OK_ASSEMBLY_INSERTED;
             emit START2_SEQUENCE;
          end
     end
   ];
 case R2_KAPUT do
    emit START2_SEQUENCE;
    emit WHAT_ARE_THEY_DOING;
   [
    present R2_PERFORMING_T4
    then
        present R1_WAITING_FOR_R2
        else
            await OK_PRE_INSERTION;
        end
    end
  ||
    present R2_PERFORMING_T3
    then
```

```
          present R1_PERFORMING_T2
          then
              await OK_PRE_INSERTION;
          end
     end
    ||
     present R2_WAITING_TO_START
     then
          await OK_PRE_INSERTION;
     end
    ];
      emit START1_SEQUENCE;
      copymodule ROB_NEAR_ASSEMBLY;
   end    % await case

module AND:

 [
  await FIRST;
||
  await SECOND;
 ];
 emit ORDER;

module ROBOT1MAINLY:
   loop        %loop1
        % starting ROBOT1 & Belt1
        [
         emit BEGIN_OBJECT1_TASK;
         copymodule  R1_FOLLOWING_A_TRAJ_TOWARD_B [
               signal OBJECT_VICINITY_R1B1/OBJECT_VICINITY_R1B,
                      R1_TRAJ_TOWARD_B1/R1_TRAJ_TOWARD_B];
      ||
         copymodule BELT_CONTROL[
               signal VICINITY_OF_B1/VICINITY_OF_B,
                      BELT1_MOTION/BELT_MOTION];
        ];
        % Robot1 can grip Object1
        copymodule GRIPPING_ON_BELT [
              signal GRIPPING_ON_BELT1/GRIPPING_ON_BELT,
                     OBJECT1_IN_GRIP/OBJECT_IN_GRIP];
        % initialising left Hand. Tra tracking for Robo1 1
        [
          copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND[
                signal LH_VICINITY_FOR_O1/LEFT_HAND_VICINITY,
                       FIRST_TRAJ_TOWARD_LH/
                       TRAJECTORY_TRACKING_TOWARD_LEFT_HAND];
      ||
```

```
  emit INIT_LH;          % putting the Left Hand in position
                         % ready to receive objects
  await LH_INITIALISED;
];
%Robot1 can put Object1 in LeftHand.
copymodule PUTTING_THE_OBJECT_IN_LEFT_HAND;
emit OK_OBJECT1;
[
  copymodule LOCKING_THE_OBJECT;
||
  [
       copymodule  R1_FOLLOWING_A_TRAJ_TOWARD_B [
             signal OBJECT_VICINITY_R1B2/
                    OBJECT_VICINITY_R1B,
                    R1_TRAJ_TOWARD_B2/R1_TRAJ_TOWARD_B];
    ||
       copymodule BELT_CONTROL[
            signal VICINITY_OF_B2/VICINITY_OF_B,
                   BELT2_MOTION/BELT_MOTION];
  ];
  copymodule GRIPPING_ON_BELT [
        signal GRIPPING_ON_BELT2/GRIPPING_ON_BELT,
               OBJECT2_IN_GRIP/OBJECT_IN_GRIP];

  copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND[
        signal LH_VICINITY_FOR_02/LEFT_HAND_VICINITY,
               SECOND_TRAJ_TOWARD_LH/
               TRAJECTORY_TRACKING_TOWARD_LEFT_HAND];
];
% insertion won't start before the locking phase ends
%(that is to say LH is ready for insertion
[
  copymodule INSERTING_THE_OBJECT_IN_LEFT_HAND;
||                          % cooperation between
  copymodule LH_INSERTION; % LH & R1 for insertion
  emit OK_PRE_INSERTION
||
  trap R2_MOTION in
    loop
      emit R2_CAN_START_ITS_MOTION;
    each CAN_R2_START
   ||
    await  R2_MOVED;
    exit R2_MOTION;
  end
];
trap LH_NOW_WORKING in
      loop
          emit LH_CAN_UNLOCK;
      each CAN_LH_UNLOCK
```

```
            ||
             await LH_NOW_UNLOCKED  ;
             exit LH_NOW_WORKING;
        end
   each R1_GO_TO_GRIP_OBJECT1
.

module ROBOT2MAINLY:
loop
        emit CAN_R2_START;
        await immediate R2_CAN_START_ITS_MOTION;
        copymodule R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND;
        emit R2_MOVED;
        emit CAN_LH_UNLOCK;
        await immediate LH_CAN_UNLOCK;
        copymodule UNLOCKING_THE_ASSEMBLY;
        emit LH_NOW_UNLOCKED;
        copymodule GRIPPING_THE_ASSEMBLY;
        emit OK_ASSEMBLY_GRIPPED;
        emit R1_GO_TO_GRIP_OBJECT1;
        copymodule MOVING_TOWARD_INSERTING_PLACE;
        copymodule INSERTING_THE_ASSEMBLY;
        emit OK_ASSEMBLY_INSERTED;
end %loop
.
```

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249 - 6399