SYSTÈME DE GÉNÉRATION DE CODE POUR DES USINES INDUSTRIELLES MODULAIRES

par

Daniel Côté

mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maîtrise en génie logiciel (M. Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, décembre 2004

Le 21 février 2005
Date

le jury a accepté le mémoire de M. Daniel Côté dans sa version finale.

Membres du jury

M. Richard St-Denis
Directeur
Département d'informatique

M. Philippe Micheau
Membre
Département de génie mécanique

M. Martin Beaudry
Président-rapporteur
Département d'informatique

Your file Votre référence
ISBN: 0-494-05895-1
Our file Notre référence
ISBN: 0-494-05895-1

SOMMAIRE

L'automatisation industrielle date du début de l'ère industrielle. Elle bénéficie ainsi d'une riche pratique, comme en témoigne le domaine des procédés industriels modernes avec sa panoplie de dispositifs tout aussi ingénieux qu'utiles. Dans ce contexte, il n'est pas surprenant de constater, dès le début du XX^{e} siècle, l'apparition d'une discipline théorique, la cybernétique, qui est propre à ce domaine. L'avènement de l'ordinateur et la miniaturisation, en ajoutant à la versatilité et à la complexité des procédés industriels, n'ont fait qu'accentuer le besoin d'une telle discipline. On peut toutefois se surprendre que, malgré cette longue histoire, et malgré des avancés considérables dans les différents modèles théoriques de ce qu'il convient d'appeler « le contrôle des processus en temps réel », les méthodes pour la programmation de ces systèmes demeurent, aujourd'hui encore, toujours si empiriques. Quant à l'utilisation de méthodes formelles pour la spécification de tels systèmes, on en entend à peu près pas parler dans l'industrie. Il semble qu'il y ait un problème sérieux à passer de la théorie à la pratique. Ce mémoire s'inscrit dans ce contexte. Il explore le problème de la génération de code pour des usines industrielles modulaires contrôlées par des automates programmables.

REMERCIEMENTS

Je tiens à remercier mon directeur de recherche M. Richard St-Denis pour le soutien financier qu'il m'a accordé durant mon travail sur ce projet. Mais je tiens aussi à le remercier particulièrement pour ses multiples conseils, critiques et suggestions qui ont souvent été déterminants dans l'orientation et le déroulement du projet. Enfin, je le remercie pour tout le patient travail de lecture et de critique des épreuves de rédaction de ce mémoire.

Je remercie également les autres membres du jury chargés d'évaluer ce mémoire, M. Martin Beaudry professeur agrégé au département d'informatique et M. Philippe Micheau professeur agrégé au département de génie mécanique, pour avoir pris la peine de lire et critiquer le texte final et ainsi m'avoir aidé à y apporter les dernières corrections.

Je remercie la Faculté des sciences de l'Université de Sherbrooke pour le soutien financier apporté à mes études supérieures à travers leur programme de bourses institutionnelles dont j'ai eu l'honneur d'être récipiendaire à l'automne 2002.

Je remercie aussi, parmi les ressources externes ayant contribuées au projet, M. René Desrochers de l'école Le Triolet, maître d'œuvre dans l'assemblage de l'usine-école MPS sur laquelle nous avons réalisé ce projet, pour toutes les informations complémentaires qu'il m'a fournies ainsi que les conseils pratiques concernant les aspects techniques du MPS.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

INTRODUCTION

Dans la seconde moitié du XX^{e} siècle l'automatisation des procédés industriels s'est beaucoup développée. En réponse à la demande croissante de biens de consommation courante, et face aux pressions de la compétition, il a fallu accroître la production tout en diminuant les prix et en maintenant ou en améliorant la qualité des produits manufacturés. Il en a résulté une automatisation croissante des procédés manufacturiers.

L'une des conséquences indirectes de cette automatisation accrue, a été la diminution du cycle de vie commercial des produits fabriqués. Non seulement les consommateurs exigent-ils constamment de nouvelles versions améliorées des produits existants, mais l'innovation et les nouveaux développements technologiques forcent constamment les manufacturiers à reconfigurer leurs nouvelles chaînes de montage automatisées pour les adapter aux nouvelles exigences quand ce n'est pas tout simplement pour fabriquer un nouveau produit en remplacement d'un autre devenu obsolète. Ainsi, en réponse à des cycles manufacturiers de plus en plus courts, l'industrie elle-même a réagi en produisant le concept d'usine modulaire dont les composantes (vérins pneumatiques, relais et autres) sont réutilisables et peuvent être reconfigurés pour s'adapter rapidement aux nouveaux procédés.

Les usines modulaires fonctionnent bien sûr sous le contrôle d'ordinateurs, une de leurs composantes est d'ailleurs un petit ordinateur particulièrement robuste adapté au type de fonction qu'il doit exercer dans un environnement d'opération souvent hostile ; il s'agit de l'automate programmable. Or, en raison des besoins de reconfiguration de plus en plus fréquents et aux cycles manufacturiers de plus en plus courts, la pression se fait sentir pour accélérer le processus de développement des logiciels de contrôle. Le programmeur agissant comme intermédiaire entre l'expert du domaine et la machine ne suffit plus. Il est de plus en plus impératif de permettre à des spécialistes du domaine (ici le contrôle d'automates industriels) d'exprimer leur savoir-faire directement sur des outils logiciels de spécification, d'où la machine elle-même pourra générer le code de l'application et le télécharger au besoin

sur les automates programmables. En d'autres termes, il faut automatiser le processus de conception des logiciels de contrôle.

Le problème de la génération de code à partir d'une spécification symbolique (provenant d'une méthode formelle ou non), ayant pour cible des automates programmables ou des ordinateurs de contrôle plus conventionnels, prend donc une importance de plus en plus grande dans ce type d'application. C'est là le contexte dans lequel nous avons initié ce projet.

Problème

La programmation d'un automate programmable (aussi appelé PLC, de l'anglais *Programmable Logic Controller*) se fait souvent directement à l'aide d'un environnement de programmation similaire à ce qui est utilisé pour n'importe quel autre type de logiciel. Forcément, le processus de conception du logiciel est aussi le même ; il est nécessaire d'avoir recours à un spécialiste (le programmeur) qui doit comprendre la spécification du procédé et ses exigences de contrôle pour les traduire en un ou plusieurs programmes. Or, il existe déjà dans le domaine du contrôle de procédés des formalismes normalisés et même des méthodes formelles permettant de décrire les problèmes de contrôle adéquatement. Certaines de ces notations, tel le GRAFCET, sont déjà suffisamment précises pour être traduites directement en code exécutable par des automates programmables ou des ordinateurs conventionnels.

Du côté des méthodes formelles la situation n'est pas toujours rendue à un stade aussi évolué. Bien qu'il soit certain qu'une méthode formelle (qui est basée sur un modèle mathématique précis) puisse permettre la génération automatique de code à partir d'une spécification, il reste souvent à trouver une stratégie qui permette la génération d'un code compact et efficace. Cette situation est particulièrement vraie quand on considère la génération de code pour des automates programmables en raison de leurs limitations inhérentes.

Les automates programmables sont souvent de capacité mémoire restreinte et possèdent habituellement des jeux d'instructions réduits ; leur système d'exploitation est habituellement très rudimentaire, de type répartiteur cyclique. Ces limitations ont une contrepartie, les automates programmables sont beaucoup plus prévisibles que les ordinateurs conventionnels avec leurs systèmes d'exploitation sophistiqués, ce qui les rend beaucoup plus adéquats dans des tâches où les temps de réponse sont critiques pour la sécurité ou le bon fonctionnement d'un procédé. Ils sont aussi conçus de façon plus robuste, pour être utilisés dans le même milieu physique que l'équipement qu'ils contrôlent ; réduisant ainsi les coûts et la distance des branchements avec l'équipement à contrôler.

Les ordinateurs conventionnels introduisent aussi des difficultés indésirables dans les problèmes de contrôle puisqu'en général, par souci d'économie, on les connecte à des automates programmables (en mode asymétrique) à travers un ensemble de protocoles de communication (par exemple TCP/IP). Ces protocoles introduisent en général un comportement difficile à caractériser, par exemple lorsque l'on veut utiliser des délais, ce qui rend l'aspect temps réel moins précis.

Le problème de la génération de code est potentiellement vaste, surtout si l'on tente de couvrir tous les aspects impliqués par exemple dans la génération de code à partir d'une spécification par grafcets. Le GRAFCET étant un formalisme graphique, il faut considérer tous les aspects d'interface usager graphique. De plus, puisque le GRAFCET n'est pas une méthode formelle, il y a aussi les questions de simulation et de mise au point à considérer.

Pour limiter l'étendue du problème, nous considérons deux objectifs à atteindre : la génération de code pour automates programmables et la génération de code pour un ordinateur conventionnel (par exemple un PC sous un système d'exploitation temps réel comme QNX). Il nous semble que la génération de code pour automates programmables présente plus d'intérêt puisqu'elle correspond à une pratique très répandue avec les usines modulaires. Nous nous limitons donc à la génération de code pour des automates programmables.

Nous réalisons notre objectif sous les hypothèses suivantes :

1. le système à contrôler est une usine modulaire ;

2. une usine-école *Festo* (appelé le MPS, de l'anglais *Modular Production System*) est utilisée pour illustrer et valider les méthodes de génération de code ;

3. la génération de code est faite à partir de formalismes (méthode formelle ou non) de spécification de procédés ;

4. les méthodes de génération de code doivent couvrir la totalité des formalismes choisis, ou tout au moins une portion significative.

Nous avons retenu deux formalismes : le GRAFCET qui n'est pas basé sur une méthode formelle mais qui est très répandu dans l'industrie, et les automates à états finis déterministes (AFD) tels qu'utilisés dans le cadre de la théorie du contrôle des systèmes à événements discrets (SED).

Nous avons considéré la génération automatique de code, mais il nous a été impossible d'obtenir les spécifications du langage machine des automates programmables que nous avions à notre disposition (*Koyo* de *Automationdirect.com*™). Nous avons donc dû nous restreindre à spécifier les schémas de génération de code à utiliser dans le processus de génération et à vérifier ces derniers par génération manuelle de code.

Méthodologie

Nous avons fixé au début du projet un cadre méthodologique pour notre investigation, que nous passons ici en revue.

Dans l'usine modulaire MPS il y a deux éléments relativement indépendants à considérer :

1. les composantes de l'usine et la façon dont elles se combinent entre elles ;

2. les matériaux qui y sont traités (déplacements, changements d'état).

Notre traitement des composants du MPS couvre les deux formalismes choisis. Il s'agit d'un problème commun indépendant du formalisme. Il faut définir et pouvoir consigner certaines de leurs caractéristiques comme partie intégrante de la spécification du système, en particulier :

1. identifier les composantes atomiques et leurs caractéristiques fonctionnelles (par exemple, les vérins pneumatiques ont un mouvement rectiligne et servent à déplacer des matériaux ou à guider un dispositif selon un axe) ;

2. identifier les caractéristiques de la commande de chaque dispositif comme le type et le nombre d'actionneurs, le type et le nombre de capteurs, le régime de la commande et du comportement (par exemple, les vérins se présentent en deux variétés pour ce qui est de la commande et du comportement : monostable et bistable piloté ou non) ;

3. identifier la façon dont les composantes sont assemblées physiquement pour donner des dispositifs plus complexes.

Nous ne traitons pas le dernier point vu sa complexité. Nous nous contentons d'identifier les dispositifs complexes et de donner une méthode de description hiérarchique de composition. Par exemple, une grue est nécessairement formée d'une flèche (peut-être un balancier qui se comporte comme un vérin) et d'une pince (peut-être une ventouse pneumatique) ; optionnellement, elle peut aussi posséder un treuil (un vérin) et un pont. La hiérarchie peut

être exprimée par simple convention de nomenclature. Par exemple, pour la grue de la cellule de traitement `C1`, on a `C1.Grue.Pince` pour la pince constituée d'une ventouse pneumatique et `C1.Grue.Flèche` pour le balancier qui tient lieu de flèche à la grue.

D'autres aspects de l'assemblage des composantes du MPS sont considérés comme des parties intégrantes de la spécification mécanique. Par exemple les contraintes de coordination entre la flèche d'une grue et sa pince constituent des éléments exprimés dans la spécification formelle.

Nous ne prenons en considération que les matériaux qui circulent dans l'usine pour la spécification par grafcets, et seulement de façon partielle. L'exemple que nous utilisons pour la théorie du contrôle des SED ne fait pas intervenir cet aspect dû à sa simplicité. Ici l'emphase est mise sur la recherche d'une stratégie de génération de code viable pour usage sur un automate programmable, un problème déjà assez complexe en lui-même. Pour la spécification par grafcets, nous couvrons toute l'usine, mais l'utilisation d'automates programmables indépendants constitue une solution de contrôle distribuée (il y a trois cellules de traitement indépendantes sur le MPS, donc trois automates programmables). Les automates programmables qui contrôlent les trois cellules n'ont pas de lien de communication entre-eux ; nous ne pouvons donc pas transmettre les caractéristiques d'usinage accumulées pour une pièce d'une cellule à l'autre.

On peut aborder les problèmes d'usinage sur le MPS de façon progressive par niveau de difficulté croissante pour la génération de code. Nous considérons trois problèmes à résoudre en séquence, puisque chacun ajoute un raffinement au précédent :

1. le problème du transport des matériaux, d'un point de transit ou d'un point de traitement à un autre, d'un bout à l'autre du circuit que doivent suivre les matériaux, sans traitement ;

2. le problème du traitement des matériaux ;

3. le problème du comportement en situation de panne (défaillance d'un capteur ou mauvais fonctionnement d'un dispositif).

De façon générale, nous n'abordons pas le problème du comportement en situation de panne.

Pour la spécification par grafcets, nous traitons les deux premiers problèmes (transport et traitement des matériaux) simultanément puisqu'il n'y a pas d'avantage à procéder en deux étapes.

Pour ce qui est de la spécification selon la théorie du contrôle des SED, le problème traité ne comporte qu'un aspect transport.

Résultats

Dans le cas de la spécification par grafcets, nous dégageons un ensemble de schémas de génération de code qui permettent certainement la génération automatique de code. Cet ensemble de schémas de génération de code est testé manuellement, c'est-à-dire que nous dressons une spécification par grafcets de l'ensemble de l'usine MPS à partir de la description textuelle fournie par le manufacturier. Nous procédons ensuite à sa codification selon les schémas de génération de code précédemment élaborés. En pratique la démarche est bien sûr un peu plus sinueuse. Une première implémentation sur automates programmables doit nous permettre d'identifier les schémas de génération de code eux-mêmes. Elle nous permet aussi d'exposer les problèmes qui existent dans notre spécification par grafcets. Une fois les schémas identifiés et la spécification par grafcets mise au point, le code de l'application est revisé pour correspondre aux schémas de génération de code finaux.

En ce qui concerne nos expérimentations avec la théorie du contrôle, nous avons restreint le problème à une dimension qui nous permet de l'examiner visuellement étant donné que l'outillage dont nous disposons, bien qu'utile, ne nous permet pas d'examiner de très gros

AFD. Le calcul d'un contrôleur en théorie du contrôle souffre d'un problème d'explosion combinatoire du nombre d'états qui peut aisément le rendre impraticable. Le problème traité n'est pas trivial, mais ne comporte qu'un aspect transport. Nous tentons d'abord une implémentation directe d'un contrôleur (un AFD), comme elle est souvent suggérée dans la littérature scientifique. Nous développons ensuite une stratégie d'implémentation plus efficace basée sur certains résultats de la théorie. Nous procédons finalement à son implémentation en identifiant un schéma de génération de code correspondant à cette nouvelle stratégie.

Relativement à la théorie du contrôle, nous obtenons donc les résultats suivants :

1. Nous déterminons plusieurs raisons pour lesquelles une implémentation directe sur un automate programmable n'est pas viable (mis à part l'explosion combinatoire du nombre d'états), ou du moins peu efficace.

2. Nous concevons une stratégie d'implémentation dérivée de la notion de SBFC (*State Feedback Control*, une notion théorique développée dans [WON04]) qui est viable pour l'implémentation de la fonction de rétroaction d'un contrôleur sur un automate programmable malgré ses limitations.

3. Nous développons un schéma de génération de code adéquat pour ce dernier type d'implémentation qui est viable pour la génération automatique.

Organisation du mémoire

Ce mémoire comporte cinq chapitres et trois annexes. Le premier chapitre couvre l'aspect physique des usines modulaires et dégage des principes pour leur description à l'aide d'une spécification formelle. Une description complète du MPS se trouve en annexe 1. Le deuxième chapitre traite des particularités des automates programmables, que nous utilisons pour nos travaux, et du langage cible pour nos schémas de génération de code, le langage *Ladder* (aussi

appelé « logique à relais »). Le troisième chapitre traite de la génération de code pour une spécification écrite en GRAFCET. Une spécification complète du procédé implémenté sur le MPS se trouve en annexe 2. Le quatrième chapitre traite de la génération de code pour l'implémentation de contrôleurs obtenus à l'aide de la théorie du contrôle des SED et des problèmes que celle-ci soulève. Une partie de la solution se trouve en annexe 3. Finalement, le cinquième chapitre fait état de nos conclusions.

CHAPITRE 1
NOTIONS D'AUTOMATIQUE

Puisque ce mémoire traite d'un sujet fort spécialisé, nous nous attardons d'abord à illustrer les concepts et à définir le vocabulaire que nous utilisons dans ce mémoire. Pour réaliser cet objectif, nous précisons le problème qui nous intéresse, celui du contrôle de procédés industriels, pour en dégager les principaux aspects. Ensuite nous proposons une grille d'analyse qui nous permet d'examiner les différentes parties du problème, pour dégager, d'un enchevêtrement souvent considérable de détails techniques, les éléments essentiels.

Bien sûr, le contrôle de procédés industriels pris dans son ensemble déborde largement le cadre de ce mémoire, nous devons donc réduire la portée de la discussion sous peine de nous en tenir qu'à des généralités. Puisque nous avons réalisé l'ensemble de nos expérimentations sur un type particulier de procédé industriel que l'on désigne sous le vocable « d'usine modulaire », nous restreignons notre étude à ce type de système. Ceci n'affecte en rien la validité de notre analyse puisque, à proprement parler, ce type de procédé industriel n'est pas fondamentalement différent, au niveau mécanique, de ses prédécesseurs. La seule différence notable réside dans la présence de micro-ordinateurs locaux et spécialisés (les automates programmables), faisant partie intégrante de ces systèmes.

En restreignant ainsi la discussion, ceci nous permet plus aisément d'illustrer nos commentaires à l'aide d'une usine-école, le MPS (*Modular Production System*) de la compagnie *Festo*, sur laquelle nous avons travaillé et pour laquelle nous disposons d'une spécification détaillée (voir les références [FES98] [FES98a] [FES98b] [FES99] et [FES99a]).

Pour donner une idée de l'aspect physique du système, nous donnons à la figure 1 une photographie de l'usine-école MPS.

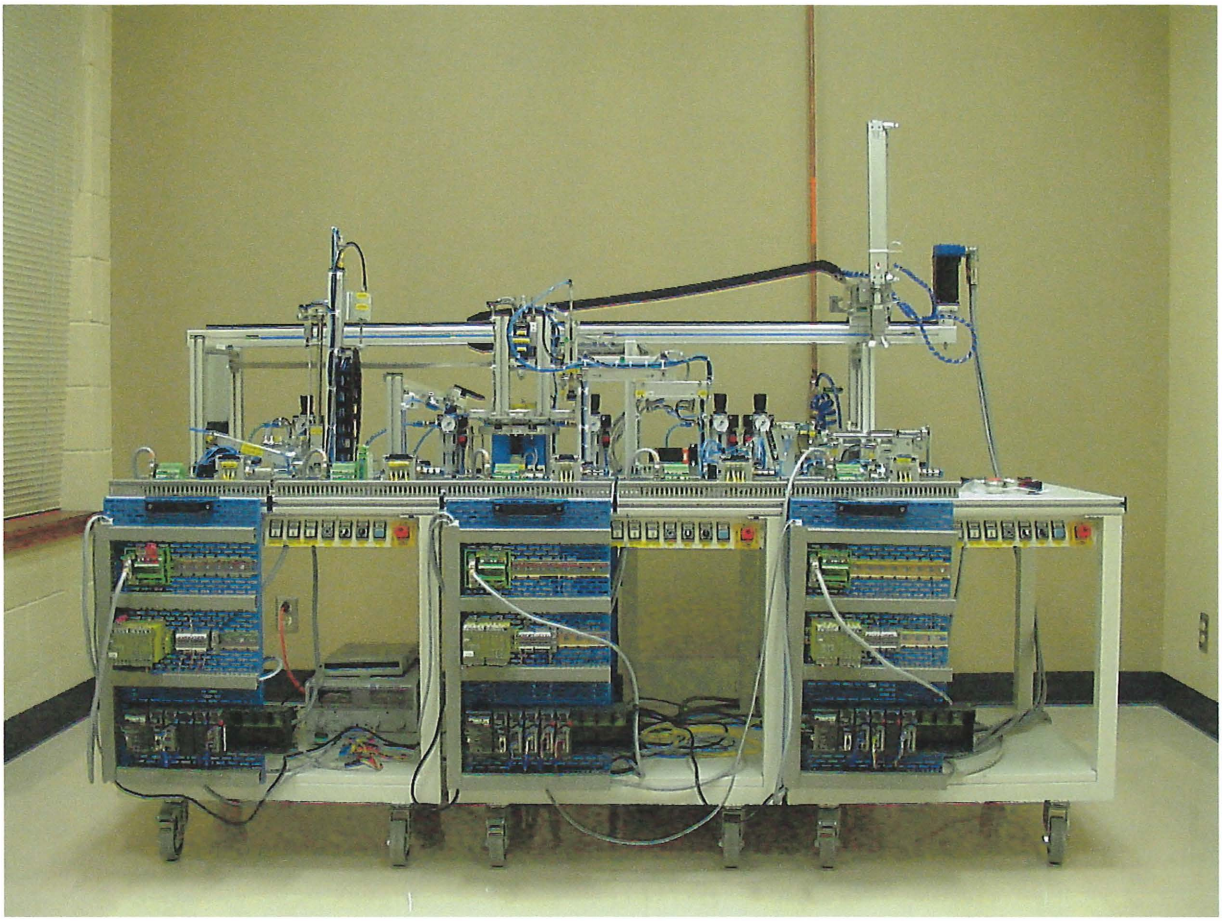

Figure 1. Usine-école MPS de la compagnie *Festo*

1.1 Usine modulaire, le contexte général

Pour fixer les idées, nous donnons d'abord une brève définition d'une usine modulaire, à partir de laquelle nous faisons ensuite notre analyse.

Une usine modulaire, comme son nom l'indique, se compose d'un ensemble de dispositifs mécaniques plus ou moins complexes, les modules, assemblés dans le but d'accomplir une tâche manufacturière spécifique. De plus, puisqu'un processus manufacturier est généralement subdivisé en étapes discrètes, on introduit un élément supplémentaire de modularisation plus

ou moins arbitraire, la cellule de traitement, qui correspond généralement à une étape du processus manufacturier.

Physiquement, on peut donc concevoir une usine modulaire comme une construction hiérarchique à trois niveaux :

1. l'usine elle-même ;

2. l'ensemble des cellules qui la constituent ;

3. l'ensemble des dispositifs élémentaires constituant les cellules.

On peut considérer qu'il s'agit ici de l'aspect d'ingénierie mécanique de l'usine. Bien que l'aspect qui nous intéresse plus particulièrement soit celui du contrôle de l'usine, nous devons tout de même obtenir une spécification du processus manufacturier implémenté par l'usine afin de pouvoir définir les éléments du problème de contrôle.

Cette définition souffrirait d'une importante lacune si l'on ne mentionnait pas l'instrumentation de surveillance et de commande, car, de toute évidence, pour pouvoir contrôler un processus manufacturier, il est nécessaire d'agir sur l'état de l'usine en déclenchant certaines opérations de transformation au moment opportun. L'usine comporte donc :

1. un ensemble de capteurs qui servent à évaluer l'état du système à tout moment ;

2. un ensemble d'actionneurs qui servent à modifier l'état du système en commandant le déclenchement d'activités précises.

À proprement parler, l'état global du système se compose de l'état de ces deux ensembles : capteurs et actionneurs. L'évaluation du « moment opportun », mentionné précédemment, s'effectue, en général, à partir d'un sous-ensemble d'éléments de l'état global du système.

Bien que la présence d'instrumentation de surveillance et de commande implique un aspect mécanique (par exemple le positionnement des capteurs), considérer celle-ci comme partie intégrante de l'aspect mécanique de l'usine ne présente que peu d'utilité. Il s'agit plutôt d'une couche ajoutée aux différents dispositifs mécaniques pour en connaître l'état et agir sur le processus.

Finalement les usines modulaires sont généralement dotées de micro-ordinateurs particuliers, appelés PLC (pour *Programmable Logic Controller* ou automates programmables en français), qui permettent de leur assurer une certaine autonomie de contrôle et de les connecter à un réseau d'ordinateurs de contrôle composés de systèmes plus puissants ayant une portée plus globale dans le processus manufacturier.

Le PLC fait bien sûr partie de l'instrumentation de surveillance et de commande de l'usine modulaire, cependant son rôle central lui confère un statut particulier. D'abord, par sa nature et sa fonction, le PLC introduit la possibilité d'utiliser de la logique programmée (par opposition à la logique câblée) pour le contrôle de l'usine. De plus, il s'agit d'un point de concentration auquel aboutissent tous les signaux en provenance des capteurs et d'où partent tous les signaux de commande des actionneurs. Le logiciel d'un ordinateur de contrôle externe ne peut interagir avec l'usine qu'à travers le PLC. Du point de vue du génie logiciel, il s'agit donc d'un élément clef de l'architecture physique dont les caractéristiques influencent significativement l'architecture logicielle de l'application de contrôle.

1.2 Usine modulaire, l'aspect physique

L'aspect mécanique des usines modulaires est de loin le plus complexe. Mais bien qu'il soit nécessaire à l'ingénieur mécanique de considérer tous les diagrammes de montage,

l'agencement des dispositifs dans l'espace ainsi que la myriade de leurs caractéristiques mécaniques, il n'est heureusement pas nécessaire d'inclure tous ces détails dans une spécification utilisable à des fins de contrôle. Bien sûr, à la limite, il peut être nécessaire de caractériser le détail du comportement d'un ou de plusieurs dispositifs mécaniques, pour en extraire des caractéristiques qui devront être prises en compte dans la logique de contrôle, mais de façon générale, un schéma simplifié suffit. Une question se pose tout de même. Que doit-on considérer pour répondre aux besoins de la modélisation d'un problème de contrôle pour une usine modulaire ?

D'après notre définition, une usine modulaire se présente comme un assemblage de dispositifs mécaniques. Ceci suggère que nous devons caractériser ces dispositifs en termes fonctionnels. Nous devons tenter de répondre à deux questions.

1. Quels sont les différents types de dispositif, et quelles sont leurs caractéristiques fonctionnelles ?

2. Quels sont les éléments importants du comportement de l'ensemble (c'est-à-dire de l'usine elle-même) ?

1.2.1 Schéma fonctionnel de l'usine

Une usine industrielle opère des transformations sur des éléments de matière première. Il peut s'agir d'usinage, d'assemblage ou de tout autre traitement. Ainsi, vu de l'extérieur, des éléments de matière première entrent dans une usine à des points précis. Ces éléments circulent dans l'usine et y subissent une série de transformations. Ces dernières s'effectuent en général à des positions physiques précises. Les transformations ont lieu par étapes discrètes, et il y a alternance entre déplacement de la matière première d'une part, et transformation de cette dernière d'autre part. Ultimement, un produit fini atteint l'une des positions de sortie d'où il est retiré de l'usine. Ce schéma général s'applique tout aussi bien aux parties d'une usine, c'est-à-dire aux cellules de traitement.

De cette simple description, on peut déjà obtenir un modèle fort utile, un genre de schéma fonctionnel de l'usine ou de ses cellules de traitement. D'après ce schéma, une usine industrielle se compose d'un certain nombre de circuits (potentiellement interconnectés en certains points), eux-mêmes composés d'étapes discrètes de traitement, le long desquelles la matière première passe dans un ordre prédéterminé, subissant une série de transformations nécessaires à la fabrication d'un produit fini qui apparaît alors à la sortie de l'usine. Ce modèle se prête immédiatement à une représentation par graphe orienté, où les nœuds dénotent des étapes de traitement, des points d'entrée ou de sortie, ou simplement des points de transit entre des cellules de traitement, et où les arcs indiquent les mouvements permis (et nécessaires) entre les différentes étapes de traitement. Ce graphe peut d'ailleurs être enrichi par l'usage de noeuds et d'arcs stylisés dont la forme serait conventionnellement suggestive soit de la fonction (par exemple, entreposage, usinage, assemblage), soit du type de dispositif utilisé (par exemple, vérin, courroie d'entraînement, glissoir). On peut, simultanément ou alternativement, adjoindre aux éléments du graphe des annotations indexées pour en augmenter le niveau de détail tout en évitant l'encombrement.

À titre d'exemple, la figure 2 montre le schéma fonctionnel physique de l'usine-école MPS, sur lequel nous avons superposé le partitionnement en stations proposé par le manufacturier et correspondant à des cellules de traitement. On peut y identifier : la station de livraison des pièces (A), la station de test de pré-usinage (B), la station d'usinage (C), la station de tri post-usinage (D) et la grue de réinsertion des pièces (E). On peut noter que seules les positions « intéressantes » se retrouvent sur le diagramme ; on peut modifier le diagramme au besoin pour y donner plus ou moins de détails.

On peut voir qu'on y retrouve, sous forme compacte, plusieurs éléments d'information utiles.

1. L'ensemble des points intéressants (du point de vue du processus manufacturier) s'y retrouve :

    a) les points d'entrée et de sortie (1 et 5) ;

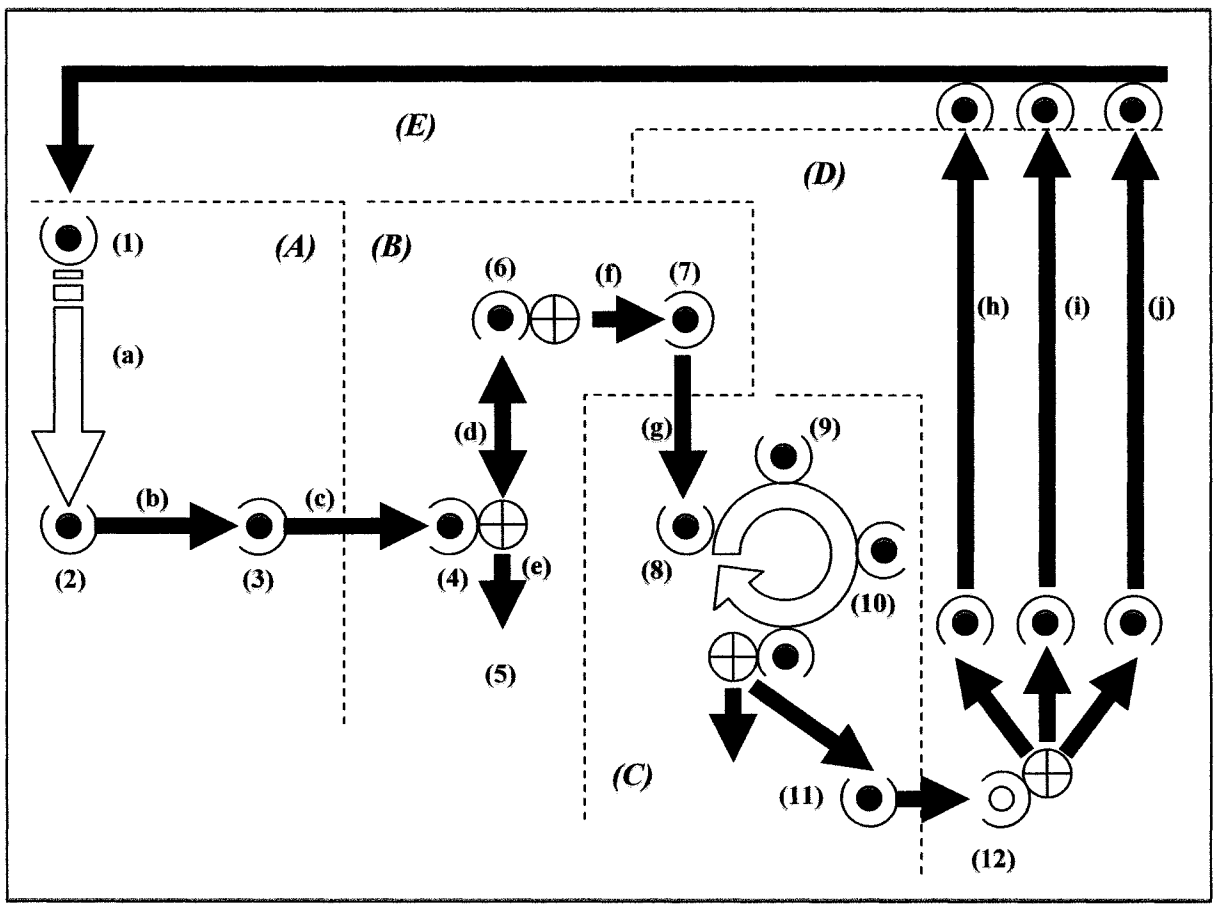

Figure 2. Schéma fonctionnel physique de l'usine-école MPS

b) les points où il y a une interaction entre des dispositifs (3, 4 et 7, 8) ;

c) les points où il y a transit (par exemple 2 et 3), usinage (9), test (4, 6 et 12).

2. L'ordonnancement des étapes du processus (représenté à l'aide de flèches), et donc la définition des circuits de traitement, s'y retrouve explicitement.

3. On peut identifier les points de prise de décisions (représenté par des cercles munis d'une croix).

Indirectement, on y voit aussi l'agencement topologique des différents dispositifs, et jusqu'à un certain point, leur agrégation en cellules de traitement. Moyennant qu'on y adjoigne une

liste indexée d'annotations, on pourra compléter le diagramme par des descriptions adéquatement détaillées dans la notation requise.

Le diagramme nous donne aussi une idée générale de la localisation des équipements constituant l'usine, mais il ne contient aucune référence aux détails mécaniques nécessaires à l'assemblage de l'appareil. Les détails de l'agencement spacial des dispositifs mécaniques entre-eux augmentent considérablement l'encombrement dans le diagramme, sans rien ajouter d'utile à la perspective fonctionnelle. Ces détails sont donc volontairement ignorés dans le schéma fonctionnel physique, et il faudra référer aux diagrammes de montage détaillés pour les obtenir.

Il est tout aussi utile de considérer les aspects que le diagramme passe sous silence.

1. Le diagramme ne montre pas l'instrumentation de surveillance et de contrôle. Nous nous attardons sur ce point dans la section 1.3. Notons seulement qu'il serait de peu d'utilité de pouvoir localiser cet équipement ici, en tout cas, dans une perspective de contrôle.

2. On ne retrouve rien non plus du côté de l'alimentation en électricité (outillage de puissance) ou en air comprimé des différents dispositifs.

De façon générale, nous laissons ce genre de considérations de côté dans notre argumentation. Nous devons supposer, pour simplifier le problème, que l'alimentation est adéquate, que l'appareil est correctement assemblé et que les dispositifs fonctionnent toujours sans panne.

1.2.2 Les dispositifs contrôlés

Il y a une grande variété de dispositifs contrôlés, et ici, il nous suffit de donner quelques exemples pertinents.

Même sur le MPS, la liste des dispositifs compte plusieurs items : des vérins de types divers, des grues de types divers, un ascenseur, une perceuse, des instruments de mesure, un carrousel et une courroie d'entraînement. Nous dégageons ce qui est important.

Premièrement, la plupart de ces dispositifs présentent beaucoup plus de similarités fonctionnelles qu'on peut le suspecter à priori, et c'est sur l'ensemble des similitudes fonctionnelles que l'on peut baser une classification assez simple de nos dispositifs. Il est intéressant d'observer que plusieurs des dispositifs sont composés de vérins. Il n'est donc pas surprenant que ces derniers héritent des caractéristiques d'un vérin (en ce qui concerne le contrôle). Il en est de même pour les dispositifs actionnés par un moteur électrique.

Deuxièmement, nous pouvons classer les dispositifs en deux grandes familles : les dispositifs passifs et les dispositifs actifs.

Les dispositifs passifs se caractérisent par le fait qu'on ne puisse pas en contrôler le fonctionnement, et que leur principe mécanique est en général la gravité. Plus souvent qu'autrement, ils servent au transport et à l'entreposage des éléments de matière première. Ils ont donc une capacité d'entreposage, un point d'entrée et un point de sortie, et une politique de distribution (la plupart du temps FIFO, c'est-à-dire premier arrivé, premier sorti). Ils peuvent être munis d'un dispositif de contrôle (comme une barrière actionnée par un vérin) à l'entrée, à la sortie ou aux deux. Sur le MPS, on peut citer comme exemples : le silo d'injection des pièces (point **a** sur la figure 2), la coulisse d'admission des pièces à la station d'usinage (point **f** sur la figure 2) dont la capacité est de une pièce et dont la sortie est contrôlée par un vérin, les trois coulisses d'entreposage de la station de tri post-usinage (points **h**, **i** et **j** sur la figure 2).

Les dispositifs actifs sont plus diversifiés, mais on peut en général les classifier par principe mécanique. Comme noté précédemment, la plupart des dispositifs du MPS sont composés soit de vérins, soit de moteurs électriques. Contrôler un dispositif composé d'un ensemble de

vérins comme par exemple une grue, revient donc à contrôler un ensemble de vérins de manière ordonnée ; il en va de même pour un ensemble vérins/moteurs. Pour cette raison il convient ici de s'arrêter un moment sur les caractéristiques des vérins (en particulier) et des moteurs électriques.

Tout d'abord, les moteurs électriques sont les plus simples à décrire. Leur fonctionnement se fait à l'aide de deux états, « en fonction » et « hors fonction », et il suffit de contrôler un relais qui en pilote l'alimentation. Sur le MPS, le carrousel qui positionne les pièces pour la séquence d'usinage (perçage et contrôle de qualité) est un exemple de ce type de dispositif. On peut citer deux autres exemples : la perceuse de la station d'usinage et le tapis roulant de la station de tri post-usinage. Cela peut paraître étonnant, mais du point de vue du contrôle (et c'est presque vrai du point de vue fonctionnel aussi), le carrousel et le tapis roulant ont presque les mêmes caractéristiques, car ils ont un point d'entrée et un point de sortie, une capacité, et leur principe mécanique est un moteur électrique.

Les vérins offrent une plus grande variété. Premièrement, ils ont deux types de comportement : le comportement monostable et le comportement bistable. Cette caractérisation est relative à l'état de la commande. Deuxièmement, les vérins à comportement bistable, peuvent avoir une action pilotée ou non. Finalement, une caractéristique importante des vérins est leur temps de réaction.

Un vérin à comportement monostable, comme leur nom le suggère, n'a qu'une seule position stable et sa commande ne comporte que deux états ; commande activée ou désactivée. La position considérée stable (allongée ou rétractée) est celle qui correspond à une commande désactivée. On aura donc un vérin monostable à position normalement allongée (ou inversement rétractée). De façon générale lorsque la commande est désactivée, le vérin se trouve en position stable, lorsque la commande est activée, il se trouve dans l'autre position. Un vérin à comportement monostable ne peut pas être dans une position intermédiaire de façon stable.

Un vérin à comportement bistable peut être dans deux positions (allongée ou rétractée) de façon stable, c'est-à-dire que même si on désactive sa commande une fois qu'il a atteint une de ces positions, sa position ne changera pas. En général leur comportement correspond à ce qu'on appelle une bascule RS en circuit logique. Sa commande comporte trois états valides et un état instable. De plus, son action peut être soit pilotée ou non.

Un vérin bistable à action pilotée, peut s'arrêter en position intermédiaire (c'est-à-dire entre ses deux états stables) si la commande du vérin n'est pas maintenue jusqu'à ce qu'il atteigne un état stable. La figure 3, donne le fonctionnement d'un vérin à commande bistable piloté. Notez dans la figure 3 la présence d'un bloc de valves fermées (au centre de la vanne pneumatique à ressort), cet arrangement permet d'obtenir un arrêt du dispositif dans une position intermédiaire indéterminée.

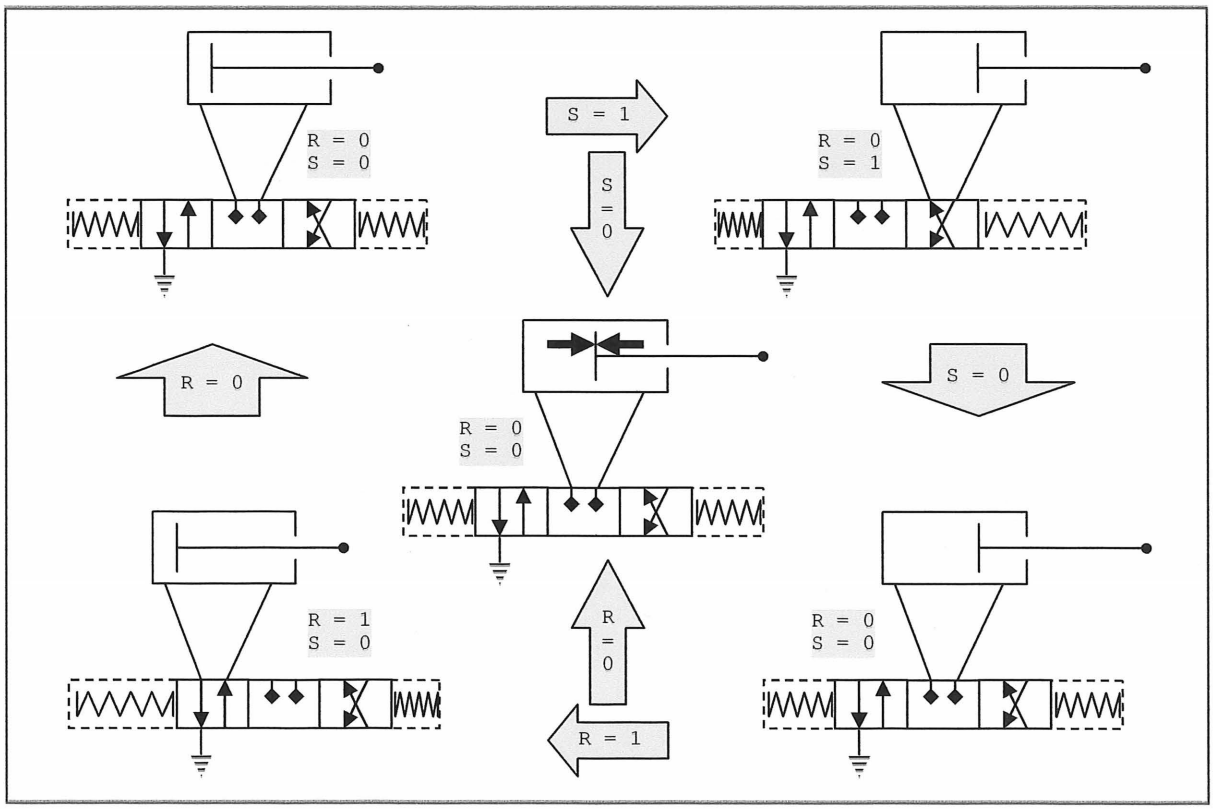

Figure 3. États d'un vérin à commande bistable piloté

La fermeture des valves permet de maintenir la pression dans les deux chambres de compression du vérin et ainsi assurer un arrêt en position intermédiaire. Ce comportement ne peut pas être obtenu par un dispositif non piloté parce qu'il n'y a pas de bloc de valves fermées ni de ressort. Pour le reste, les deux diagrammes sont essentiellement les mêmes, entre autres, l'état où les deux commandes (R et S) sont actives est illégal pour les deux types de vérin. Il en va de même pour tout dispositif utilisant ce type de commande.

Dans le cas d'un vérin bistable à action non pilotée, il suffit de donner une commande de changement de position pour s'assurer que le vérin obéisse. Il n'est pas nécessaire de maintenir la commande d'un vérin à action bistable non piloté ; un tel vérin ne peut pas assumer de position intermédiaire de façon stable. La figure 4 présente le fonctionnement d'un vérin à commande bistable non piloté.

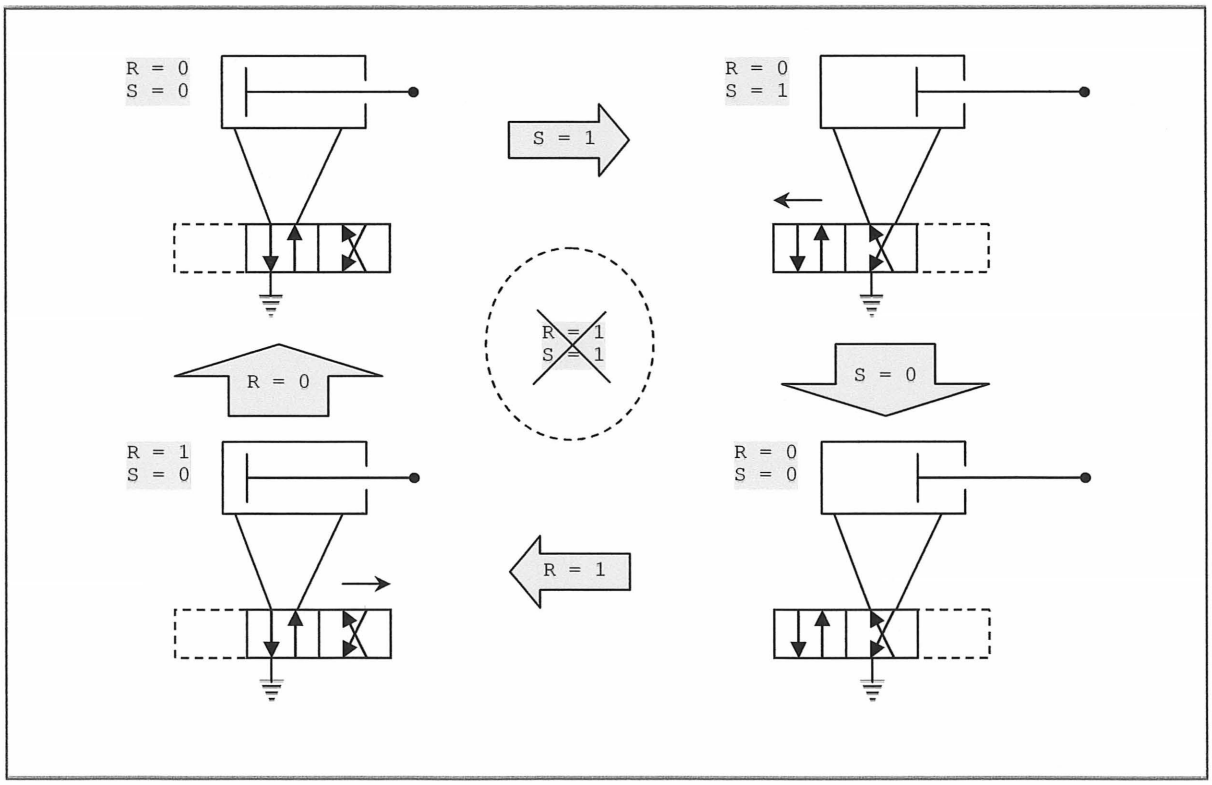

Figure 4. États d'un vérin à commande bistable non piloté

1.3 Usine modulaire, l'instrumentation de surveillance et de commande

Cette section est somme toute assez simple car bien qu'il existe une grande variété de capteurs et d'actionneurs, le seul point d'intérêt ici est le principe de leur commande. Il ne sert à rien d'inventorier ces différents dispositifs.

Rappelons que, l'instrumentation de surveillance et de contrôle forme un genre de couche superposée à la mécanique de l'usine modulaire. On pourrait s'intéresser à la disposition des capteurs et des actionneurs sur le matériel de l'usine, ainsi qu'à leur branchement aux PLC puisque ceux-ci constituent le cœur matériel de l'architecture de contrôle. Mais pour abréger, comme dans les sections précédentes, posons-nous la question de savoir ce qui est essentiel dans la perspective du contrôle.

1.3.1 Le schéma d'interconnexion

Le schéma d'interconnexion revêt un caractère crucial dans l'ingénierie de l'usine. Ce schéma permet de savoir ce que les capteurs mesurent ou détectent, et ce que les actionneurs commandent. Ici nous ne nous intéressons pas au schéma lui-même, mais plutôt à un produit dérivé : le tableau d'assignation des terminaux d'un PLC. Nous assumons que ce tableau nous est fourni à priori, mais il doit contenir un certain nombre d'informations pour être utile.

Nous verrons plus tard (section 1.4) qu'un PLC concentre les signaux en provenance des capteurs et distribue les signaux de commande ; les capteurs et les actionneurs de l'usine doivent donc être branchés au PLC. Au niveau d'un PLC nous appelons « terminal » un point de branchement, soit en provenance d'un capteur, soit en direction d'un actionneur. Ces terminaux se voient attribuer un nom logique (dans l'espace du PLC) et il importe de savoir ce que chaque terminal représente.

Il n'est pas nécessaire de disposer du schéma de branchement des capteurs et des actionneurs mais simplement d'un tableau qui nous donne :

1. le nom logique de chaque terminal d'un PLC ;

2. la désignation (sur le schéma fonctionnel vue précédemment) de l'équipement relatif au capteur ou à l'actionneur ;

3. le type du capteur (actionneur) ;

4. une description brève, mais précise de ce que le capteur mesure ou de ce que l'actionneur commande.

Cette information suffit en général et résume, pour fin de contrôle, toute l'information contenue dans le schéma physique et le schéma d'interconnexion. Il est cependant important que l'information (entre autres, l'information fonctionnelle, comme ce qu'un capteur mesure) soit exacte.

La figure 5 donne une description compacte de l'un des dispositifs du MPS qui répond à toutes les exigences que nous venons d'énumérer. Dans cette fiche descriptive, on emprunte aux conventions utilisées dans le MPS : les capteurs discrets sont notés par `Xn` où *n* est un indice, similairement, les actionneurs discrets sont notés `Yn`. La forme en équation regroupe de façon compacte tous les terminaux impliqués ; les actionneurs d'un côté et les capteurs de l'autre. Cette forme de description a l'avantage de regrouper les terminaux par dispositif (une unité fonctionnelle dans l'usine). Un ensemble complet de ces descriptions remplace avantageusement un schéma d'interconnexion.

1.3.2 Principes de détection, de mesure et de commande

L'ensemble des capteurs et des actionneurs d'une usine se divise en deux grandes catégories : ceux qui fonctionnent sur un signal discret (type booléen) et ceux qui fonctionnent sur un intervalle d'un signal continu.

```
MPS.C1.Livraison.Grue.Flèche = { (Y0,Y1) : (X2,X3) }
```

Dispositif : Flèche de la grue de la station de livraison (type fonctionnel : vérin)
Localisation : Station de livraison (section A, arc **c** sur la figure 2).
Commande : bistable piloté

$(\overline{Y0}, Y1)$: Ramène le balancier (la flèche) vers le dock de chargement

$(Y0, \overline{Y1})$: Ramène le balancier (la flèche) vers la plate-forme de test

Capteurs :

$(X2, \overline{X3})$: Flèche au dock de chargement

$(\overline{X2}, X3)$: Flèche à la plate-forme de test

N.B. : La configuration $(X2, X3)$ des capteurs n'est pas physiquement possible.

Figure 5. Description de la grue à balancier de la station de livraison du MPS

Les capteurs et les actionneurs qui fonctionnent sur un signal discret seront représentés par une valeur booléenne (un bit d'information dans l'espace mémoire d'un PLC), dont le « 0 » correspond à « hors fonction » et le « 1 » correspond à « en fonction ». Le terminal correspondant du PLC a comme fonction (entre autres) de convertir ce signal dans une valeur appropriée s'il s'agit d'un actionneur, et inversement s'il s'agit d'un capteur.

Les capteurs et actionneurs qui fonctionnent sur un intervalle d'un signal continu sont représentés par un registre qui contient la valeur digitalisée du signal correspondant. Le terminal correspondant est habituellement constitué d'un convertisseur analogique vers numérique (dans le cas d'un capteur) ou numérique vers analogique (dans le cas d'un actionneur). La partie analogique dépend de ce que le dispositif mesure ou de ce que la commande requiert (courant, voltage et puissance appropriée).

Ainsi, au plus bas niveau, celui du PLC, le logiciel de contrôle ne voit que des indicateurs binaires et des registres. Il ne faut cependant pas oublier que l'ingénierie de l'usine requiert beaucoup plus de détails.

1.4 Usine modulaire, les aspects liés au génie logiciel

Du point de vue logiciel, un PLC représente le dispositif le plus important d'un procédé. C'est à travers cet appareil que toutes les tâches de surveillance et de commande peuvent être accomplies. Or un PLC est un micro-ordinateur un peu particulier parce ses caractéristiques les plus importantes ne sont pas la versatilité et la flexibilité, mais plutôt la robustesse et la fiabilité. Cet ordinateur doit résister à des conditions que la plupart des ordinateurs (même de nos jours) ne pourraient pas tolérer : température, niveau de vibration élevé et environnements hostiles (très poussiéreux et électroniquement « bruyants »). Puisqu'il ne s'agit pas d'un ordinateur comme les autres, et qu'il joue un rôle central dans la commande d'un procédé, cet appareil aura un impact certain sur le logiciel de contrôle, il vaut donc la peine de l'examiner en détail, et d'en comprendre les fonctions et, surtout, les limitations.

1.4.1 Automate programmable

Un automate programmable, ou PLC, assume plusieurs fonctions essentielles au sein d'un procédé industriel.

1. Il concentre les signaux en provenance des capteurs et il distribue ceux en direction des actionneurs. À cette fin il doit aussi faire les conversions nécessaires pour respecter les caractéristiques électriques des différents équipements qui y sont branchés.

2. Il peut être muni d'une unité programmable lui permettant de supporter du logiciel qui constitue une partie ou la totalité de la fonction de contrôle du procédé industriel.

3. Il peut supporter une connexion avec un autre ordinateur soit pour des fins de programmation, soit pour des fins de contrôle distribué, ou simplement pour des fins de surveillance.

La figure 6 nous montre un automate programmable du MPS. Physiquement il se présente comme une petite boîte composée d'une alimentation en deux parties isolées l'une de l'autre : une partie sert à l'ordinateur, l'autre à l'instrumentation de surveillance et de commande qui lui est connectée. Comme tout ordinateur, il comporte un bus auquel on peut raccorder des interfaces d'entrée (pour les capteurs) et de sortie (pour les actionneurs). Sa mémoire volatile est habituellement restreinte (64 à 128 Ko), mais elle est doublée d'une mémoire rémanente (*flash-ROM*) qui lui permet de conserver sa programmation et sa configuration même lorsqu'il est mis hors tension. Un automate programmable contient aussi une mémoire rémanente en lecture seulement, que l'on peut considérer comme faisant partie de la machine (bien que l'on puisse la réécrire) : le « *firmware* ». Cette mémoire contient le système d'exploitation et le code machine formant la bibliothèque des routines du système (en anglais *system library*) ; on peut considérer qu'il s'agit de l'environnement logiciel de la machine. Cet environnement peut être remplacé, bien que peu fréquemment, par une nouvelle version du système en provenance du manufacturier, et à l'aide d'une procédure spéciale, mais on ne peut jamais y écrire du code d'application rédigé par l'usager.

Les modules d'interface du PLC sont munis de connecteurs externes qui se branchent directement sur les dispositifs de l'usine modulaire. Ils doivent donc en respecter les caractéristiques électriques : courant continu ou alternatif, puissance requise et impédance entre autres. L'une des fonctions des modules d'interface est de transformer les signaux numériques de l'ordinateur du PLC en signaux correspondants du côté commande (et vice versa), puisque les deux types de signaux ont des caractéristiques différentes, incompatibles. Nous nommons les points de connexion aux interfaces du PLC les « terminaux » du PLC.

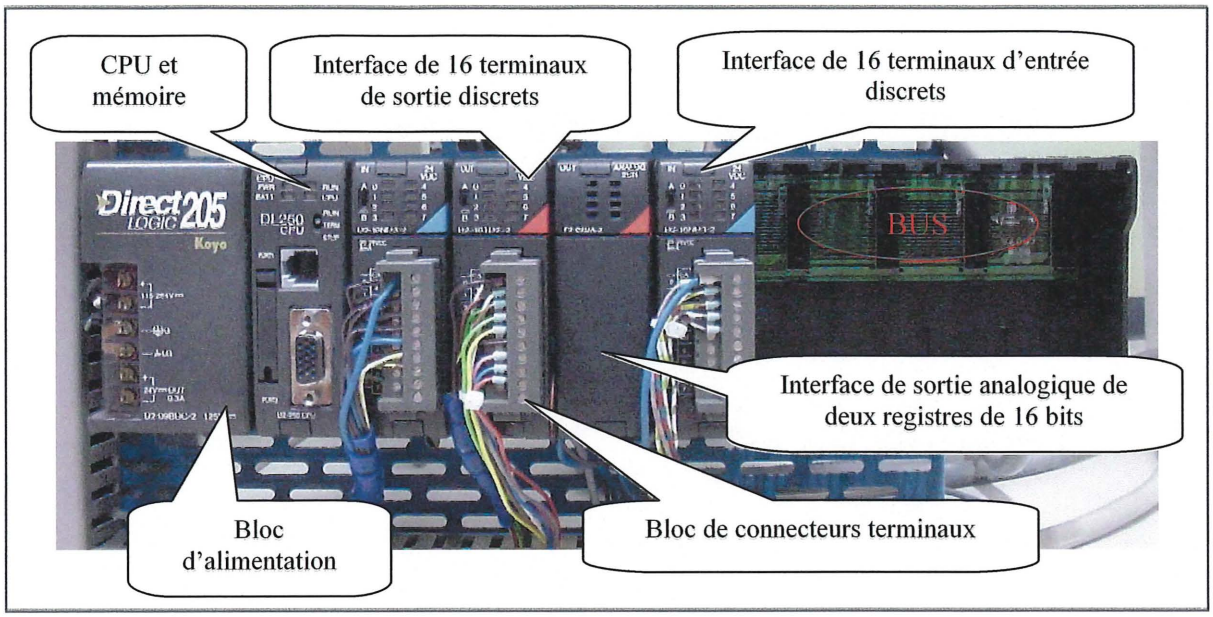

Figure 6. Automate programmable

Un PLC affecte un nom logique à chaque terminal lorsqu'il démarre, et effectue une projection (*memory mapping*) des signaux présents aux terminaux sur une partie de sa mémoire interne, et ce à chaque cycle. De cette façon, l'ensemble des dernières valeurs des capteurs est disponible au logiciel de contrôle (résidant sur le PLC) au début d'un cycle. Aussi, le logiciel de contrôle met à jour la valeur d'un actionneur par simple adressage mémoire et le PLC fait la mise à jour effective de cet actionneur à la fin de son cycle.

Les modules d'interface d'un PLC se divisent principalement en deux catégories.

1. Les modules d'interface à terminaux discrets. Ces terminaux sont l'équivalent d'un bit d'information, ils n'assument que deux états : « activé » ou « désactivé », par exemple un capteur optique peut détecter la présence ou l'absence d'une pièce dans un dispositif.

2. Les modules d'interface analogiques. Ces terminaux interfacent un PLC à des signaux continus sur un intervalle donné, soit en courant, soit en voltage. Ils

contiennent un ou plusieurs registres dont la taille peut varier d'un module à l'autre (par exemple 16 ou 32 bits), mais qui sont généralement uniformes pour un module donné ; la taille de ces registres est fonction de la puissance de résolution de la conversion requise. Les modules analogiques sont aussi munis d'un convertisseur analogique/digital qui fonctionne dans un sens ou dans l'autre (dépendant s'il s'agit de terminaux d'entrée ou de sortie) et qui effectue des conversions entre le voltage (ou l'intensité du courant électrique) échantillonné sur un canal analogique d'une part, et un registre du module d'autre part. Par exemple, on peut brancher un thermocouple sur un canal analogique et lire la valeur du voltage à sa sortie dans l'un des registres d'un module analogique. On peut ainsi déterminer la température (après calcul) mesurée par le thermocouple. Chaque canal analogique possède, en général, son propre registre sur le module d'interface.

Chacune de ces catégories de module d'interface se présente en deux variétés (pour un total de quatre types) :

1. les modules d'interface d'entrée ou capteurs (pour la surveillance) ;

2. les modules d'interface de sortie ou actionneurs (pour la commande).

Il y a bien sûr des modules d'interface mixtes qui combinent soit des terminaux d'entrée et de sortie ensemble, soit des terminaux analogiques et des terminaux discrets ensemble, ou encore toutes les catégories confondues. Cependant la plupart du temps, les modules se présentent dans l'un ou l'autre des quatre types énumérés précédemment (par exemple, le module d'interface à seize terminaux d'entrée discrets sur la figure 6).

Avant tout, un PLC doit être prévisible ; on dit aussi déterministe. C'est pourquoi le système d'exploitation d'un PLC est très élémentaire. En général il s'agit d'une version légère d'un système de type répartiteur cyclique (*cyclic executive*, [SHA01]) qui permet une

caractérisation très exacte et très fiable du temps d'exécution. De la même façon les périphériques d'un PLC (i.e. ses modules d'interface) doivent aussi faire l'objet de caractérisations temporelles très strictes.

Tel que son nom le suggère, en fonctionnement normal, un répartiteur cyclique possède un cycle qui est répété sans fin par le PLC. Une partie de ce cycle se déroule dans le système d'exploitation du PLC, l'autre dans le code de l'usager. Le manufacturier doit fournir, avec le système d'exploitation, une description du cycle et une ou plusieurs formules permettant de calculer le temps d'exécution passé dans le système d'exploitation pour chaque cycle, dépendant de paramètres variables tels que la configuration des modules d'interface du PLC. Le manufacturier doit aussi fournir une série de formules (à raison d'une formule pour chaque instruction en code machine de l'UCT) permettant au programmeur de calculer le temps d'exécution de son programme de contrôle sous les mêmes paramètres.

À titre d'exemple simplifié, voici à peu de chose près, le cycle d'un PLC :

1. mise à jour de la copie mémoire de l'état des terminaux d'entrée (capteurs) ;

2. exécution de la logique de contrôle résidante dans la mémoire du PLC (la valeur de la copie mémoire des actionneurs peut alors changer) ;

3. mise à jour des actionneurs sur les cartes d'interface avec les nouvelles valeurs calculées en mémoire par le logiciel de contrôle résidant à l'étape 2 ;

4. partie administrative (entre autres les diagnostiques d'évaluation de la viabilité du système et la mise à jour des indicateurs d'alarme et d'avertissement accessibles au programme de contrôle).

Ce cycle est répété indéfiniment jusqu'à une commande d'arrêt manuelle, ou jusqu'à ce que le système détecte une erreur qu'il ne peut ni corriger ni récupérer, et qui nécessite un arrêt en faute. Bien sûr un PLC surveille le temps mis par le logiciel de contrôle à s'exécuter, et le termine prématurément si, pour une raison ou pour une autre, il prend trop de temps. Dans ce dernier cas, le PLC enregistre une erreur.

Toutes ces caractéristiques font d'un PLC un engin dont la programmation ne peut pas être très sophistiquée. La plupart des PLC sont programmés dans un genre d'assembleur très élémentaire. Le jeu d'instructions est très pauvre, le niveau de récursivité limité (pour les PLC dont est muni le MPS, la limite est de huit niveaux), l'essentiel de l'arithmétique se fait en nombres entiers. La plupart des PLC utilisent un langage graphique pour leur programmation, appelé *Ladder Logic*, nom qui est probablement dû à l'aspect visuel des programmes rédigés dans ce langage et qui ressemblent littéralement à des échelles. L'aspect graphique importe peu cependant ; les caractéristiques du langage sont beaucoup plus importantes et font état de ses limitations. Un programme *Ladder* prend nécessairement l'une de deux formes :

1. la forme d'un ensemble de circuits similaires à des circuits logiques combinatoires qui sont systématiquement évalués à chaque cycle du PLC ;

2. la forme d'automates à états, proches des machines de Mealy et de Moore [BOO67].

On peut bien sûr utiliser les deux formes dans un même programme, en fait, on y est même obligé puisque la forme en automate ne fait que partitionner l'ensemble des circuits de logique combinatoire, introduisant une notion d'état, pour donner au programmeur un meilleur contrôle des conditions dans lesquelles ces circuits auront un effet.

Le principe en est simple. Chaque circuit se compose de deux parties : un prédicat et une action conséquente (pouvant être composée de plusieurs instructions machine). Dans le cas

d'un programme ayant la forme d'un ensemble de circuits sans notion d'état, chaque prédicat est évalué en séquence et, selon le résultat du prédicat (un résultat toujours booléen), l'action associée est exécutée ou non. On peut spécifier l'exécution de l'action sous n'importe quel des deux résultats possibles du prédicat (spécification en logique positive ou négative). Lorsqu'il s'agit d'un programme en forme d'automate, chaque état de l'automate contient un ensemble de circuits tels que décrit précédemment, mais seuls les circuits contenu dans des états actifs au cours du cycle courant seront évalués. Les circuits contenus dans des états inactifs ne seront tout simplement pas évalués pendant ce cycle du PLC.

CHAPITRE 2
PARTICULARITÉS DES AUTOMATES PROGRAMMABLES

Il nous faut connaître certaines particularités de la machine cible pour aborder la génération de code. Comme nous utilisons principalement un automate programmable comme machine cible, et que ce genre de machine est peu connu et limité dans ses capacités, il nous faut d'abord en décrire les particularités les moins intuitives.

Pour faciliter la discussion, il est préférable de se mettre dans un contexte particulier. Nos discussions sont basées sur les automates programmables de marque *DirectLOGIC*™ modèle DL205 de la compagnie *Automationdirect.com*™, utilisant un UCT modèle DL250, aussi connus sous le nom de *Koyo*. Ces automates programmables sont typiques de ce que l'on retrouve sur le marché, ainsi nous ne perdons pas en généralité à nous restreindre à un produit particulier et nous y gagnons en clarté. L'essentiel de l'information contenue dans cette section se retrouve dans la référence technique [AUT00], particulièrement aux chapitres 3, 5 et 7.

Rappelons brièvement que le programme d'un automate programmable travaille sur une image mémoire de ses capteurs et de ses actionneurs. L'état des capteurs est lu juste avant l'exécution du programme qui modifie éventuellement l'état de l'image mémoire des actionneurs. L'état des actionneurs est mis à jour juste après l'exécution du programme (nous négligeons ici la phase de vérification administrative qui suit normalement la mise à jour des actionneurs). Cette dernière séquence constitue le cycle de l'automate programmable qui est répété indéfiniment. L'automate dispose de deux types de capteur et d'actionneur : type discret (valeur booléenne) et type registre. En plus un automate programmable dispose d'autres ressources mémoire qui peuvent lui servir à maintenir une information d'état ou à calculer des valeurs dérivées nécessaires au fonctionnement de la logique programmée.

Les automates programmables *DirectLOGIC*™ ont une convention de nomenclature servant à dénoter de façon succincte la nature et le type de chaque élément mémoire. Cette convention permet au programme un adressage par nom des variables de son espace de calcul. Comme on utilise cette convention continuellement dans nos discussions, il est utile de la connaître. Le tableau 1 donne les principaux éléments de cette nomenclature.

Pour les éléments discrets, dans une expression booléenne, la notation `Xn` dénote que le capteur est au niveau « 1 » logique et la notation $\overline{\texttt{Xn}}$ que le capteur est au niveau « 0 » logique. Il en est de même pour les actionneurs et les relais de toutes sortes. Les expressions d'affectation suivent les conventions usuelles.

L'unité centrale DL250 des automates DL205 possède un jeu d'instructions réduit lui permettant de manipuler des variables booléennes, des nombres entiers de 16 et 32 bits, des nombres en virgule flottante sur 32 bits et des chaînes d'octets soit individuellement soit dans des tables. Le jeu d'instructions est décrit en détail dans [AUT00] au chapitre 5 pour les instructions appelées *RLL* Standard et au chapitre 7 pour les instructions permettant de programmer des machines à états (instructions *RLL Plus*). Il convient cependant de faire quelques remarques sur l'UCT.

L'UCT DL250 ne comporte qu'un seul registre et un ensemble d'indicateurs d'état (par exemple, débordement de capacité, signe du nombre dans l'accumulateur). Elle est munie d'une pile limitée à huit niveaux. Sa programmation rappelle la programmation d'une calculatrice à notation polonaise inversée (par exemple une calculatrice HP) avec plus de limitations.

La programmation d'un automate DL205 se fait en *Ladder Logic*, un langage graphique. Bien sûr chaque expression *Ladder* est ultimement traduite en langage machine, mais pour garder la discussion compréhensible, nous utilisons principalement des diagrammes avec des expressions *Ladder*. Nous donnons ici quelques exemples commentés.

Tableau 1. Nomenclature des éléments de mémoire d'un PLC *DirectLOGIC*™

Forme	Description	Nombre (max.)
Xn	Capteur discret	512
Yn	Actionneur discret	512
VXn	Capteur, registre de 16 bits	-
VYn	Actionneur, registre de 16 bits	-
WXn	Capteur, registre de 32 bits	-
WYn	Actionneur, registre de 32 bits	-
Cn	Relais interne (valeur booléenne), peut être modifié et lu	1024
Sn	Relais d'état (*Stage*) interne (valeur booléenne), sert dans la programmation d'un automate, dénote si un état est actif ou non	1024
Vn	Mot mémoire de 16 bits	-
Tn	Relais d'état d'une minuterie, indique quand une minuterie est échue (une minuterie a aussi une valeur associée en mémoire, c'est-à-dire un élément de type *Vn*)	256
CTn	Relais d'état d'un compteur à niveau, indique si le compteur a atteint un niveau prédéfini (même remarque que pour une minuterie)	128
SPn	Relais spécial utilisé par le système pour dénoter une condition ; d'intérêt particulier SP0 dénote le démarrage du PLC, SP1 a toujours la valeur « *vrai* ».	-

L'expression de base en *Ladder* (dont on a un exemple en figure 7) est un genre d'action conditionnelle ayant la forme $P \Rightarrow$ liste d'actions, où P est une formule logique contenant des expressions booléennes et des expressions relationnelles. Le conséquent de l'expression est une liste d'actions sous forme d'instructions DL250 brutes. La sémantique d'une expression de base est l'exécution de la liste d'actions si la formule logique est satisfaite et ce, chaque fois que la formule est évaluée (à chaque cycle).

Il y a deux styles de programme. Le premier style appelé *RLL Standard* n'est qu'une suite d'expressions de base, évaluées l'une à la suite de l'autre (dans l'ordre de leur écriture) à chaque cycle de l'automate. C'est le cas, si l'on place d'autres expressions similaires à la première à la suite dans l'exemple de la figure 7.

Le second style, dont on trouve un exemple en figure 8, consiste à partitionner un (ou plusieurs) processus en étapes (*Stage* en anglais), chaque étape contenant une liste d'expressions *Ladder* de base. La notion d'étape est proche de la notion d'état, mais ici, plusieurs étapes peuvent être actives à la fois (mais pas nécessairement). Des instructions spéciales sont fournies pour activer et désactiver des étapes.

On appelle ce dernier style, *Stage programming* en anglais, que nous traduisons par « programmation en étapes » pour ne pas trop nous éloigner de la terminologie anglaise.

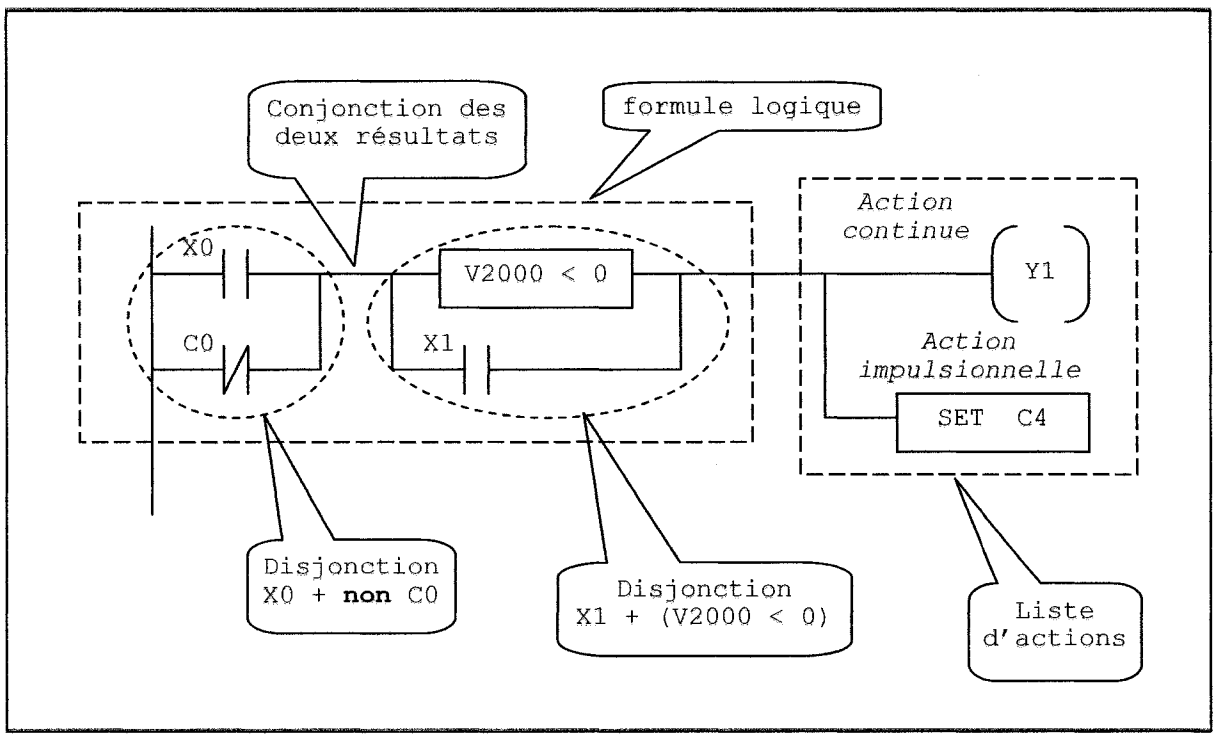

Figure 7. Expression de base en *Ladder* (RLL Standard)

On peut noter à la figure 7 que la liste d'actions contient une action dans un genre de demi-cercle (appelé *coil* dans la documentation technique). Une action de ce type est qualifiée d'action continue. Dans une action continue, la sortie (ici l'actionneur `Y1`) suit fidèlement l'état de la formule, et le type de la variable doit être booléen puisque la formule a un résultat booléen. De plus `Y1` (la variable) ne peut pas être sous le contrôle d'une autre expression au risque d'obtenir des conflits. La figure contient aussi une action dite impulsionnelle ou mémorisée (**SET** `C4`) dans une boîte. Les actions impulsionnelles ne sont pas sujettes à la règle d'exclusivité, c'est-à-dire que l'on peut utiliser la même variable cible (ici `C4`) comme cible d'une action impulsionnelle dans une autre instruction de base. Les instructions impulsionnelles ont une réelle utilité dans la programmation en étapes.

On peut voir à la figure 8 l'expression d'un événement ; le symbole aux deux lignes parallèles avec un front montant d'impulsion entre les deux.

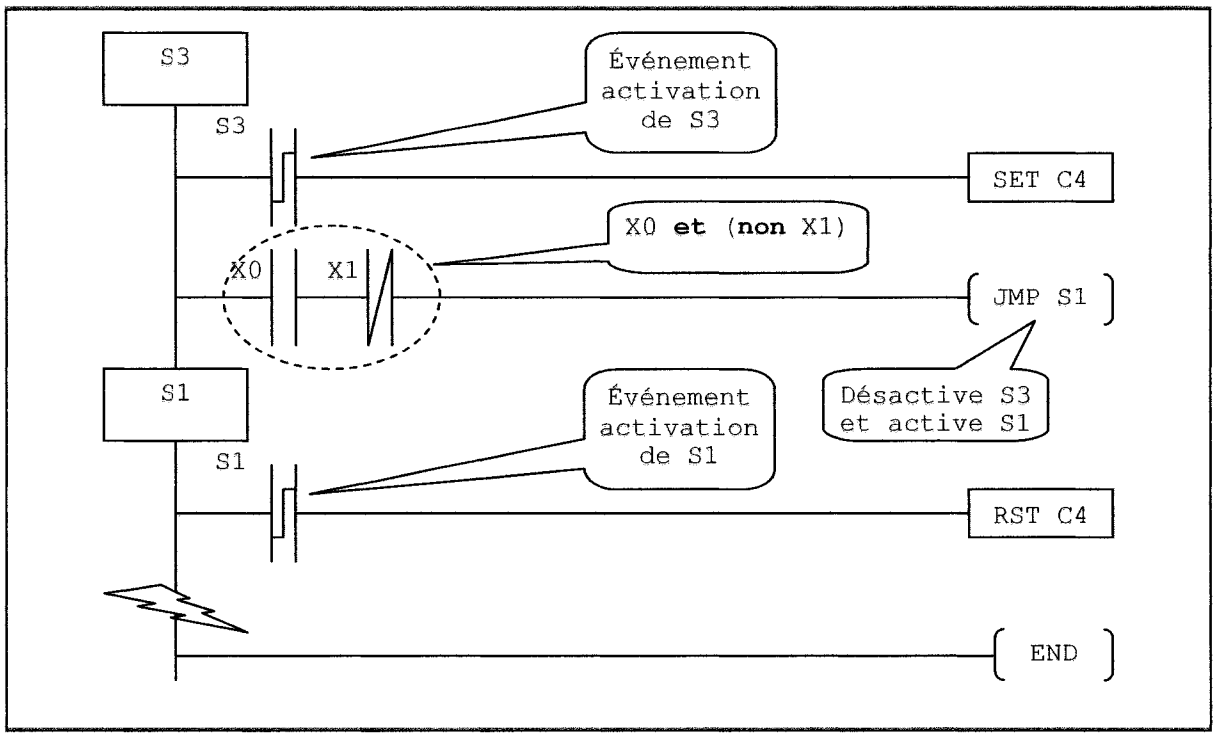

Figure 8. Exemple d'un programme *RLL Plus* (machine à états)

Un événement (par exemple le front montant ou descendant d'un capteur) n'est réalisé que pour un seul cycle de l'automate, lorsque la variable qui lui est associée fait une transition de « 0 » vers « 1 », pour le front montant et de « 1 » vers « 0 » pour un front descendant.

De façon générale, les deux lignes parallèles verticales symbolisent un relais. Lorsqu'il y a une barre oblique entre les deux, le relais travaille en logique négative (normalement fermé). Sans barre entre les deux, le relais travaille en logique positive. Dans la figure 8 on voit l'expression d'une variable en logique négative (la négation logique), le relais étiqueté `X1`.

CHAPITRE 3
LA SPÉCIFICATION PAR GRAFCET

Le formalisme GRAFCET est le produit d'un effort de recherche et de normalisation de l'AFCET (Association Française de Cybernétique Économique et Technique). Il a été adopté comme norme internationale [CEI88] portant le n° IEC 848 en 1988 et qui a été revisée en 1991. L'usage de grafcets s'est répandu dans l'industrie, surtout en Europe, où ils sont habituellement la norme dans une spécification contractuelle. Apparemment le nom de GRAFCET viendrait autant de « grAFCET », puisqu'il s'agit d'un formalisme graphique élaboré à l'AFCET, que de « GRAphe Fonctionnel de Commande Étape-Transition » (en anglais *Sequential Function Charts*).

Considérant la large diffusion de spécifications par GRAFCET dans l'industrie (du moins en Europe) et leur aspect normatif qui en font un genre d'état de l'art, il semble d'intérêt d'en présenter ici le formalisme et d'en examiner l'aspect de génération de code pour des automates programmables. La spécification par GRAFCET a déjà fait l'objet d'efforts de recherche considérables, en particulier pour tenter d'appliquer certains résultats attrayants des réseaux de Petri (les grafcets sains, voir [DAV92] au chapitre 5) desquels ils sont un dérivé.

Finalement, bien que depuis la normalisation, les nouveaux automates programmables aient été modifiés pour permettre la matérialisation de spécifications par GRAFCET, la plupart des automates programmables sont loin de satisfaire les contraintes énoncées dans la norme pour leur interprétation. En particulier, la norme requiert d'utiliser un algorithme avec recherche d'un état stable. Cette recherche requiert une itération sur le grafcet jusqu'à un arrêt de progression, ce qui génère potentiellement d'autres événements (activations et désactivations d'étapes) qui doivent être mémorisés pour l'évaluation des autres grafcets dans l'espace d'exécution. La plupart des PLC ne semblent pas équipés d'un langage permettant l'implémentation d'un tel algorithme (certainement pas ceux de *Automationdirect.com*™ que nous utilisons). Dans ce contexte, il est légitime de se demander : comment est-il possible de

traduire une spécification par GRAFCET vers du code d'automate programmable et quelles sont les difficultés à surmonter pour y arriver ?

3.1 GRAFCET, le formalisme

Le GRAFCET est un formalisme graphique. La figure 9 en illustre les éléments fondamentaux.

Dans la figure on peut distinguer les éléments suivants.

1. Les boîtes correspondant aux **étapes**, dont l'une est désignée comme **étape initiale**. Une étape a un état ; elle est soit active, soit inactive. L'étape initiale est la seule active au démarrage du dispositif commandé. On associe généralement à une étape une liste d'actions (possiblement vide).

2. Les **transitions** qui correspondent à des traits verticaux entre étapes, traversés d'une barre. Une transition est, soit franchissable, soit infranchissable. Elle est franchissable si, et seulement si, toutes les étapes qui la précède sont actives (on dit alors que la transition est validée) et la réceptivité qui lui est associée a la valeur *vrai*. On associe toujours à une transition une expression à valeur booléenne (elle peut être textuelle) que l'on nomme sa **réceptivité**. On place la réceptivité côte à côte avec la barre horizontale d'une transition. La réceptivité agit comme condition de garde de la transition.

3. Les **liaisons** qui sont les segments d'arcs (toujours orientés) liant une ou plusieurs étapes à une transition ou une transition à une ou plusieurs étapes. Les segments d'arcs dans un grafcet sont toujours implicitement orientés parce qu'ils suivent une direction conventionnelle (de haut en bas et de gauche à droite). Les arcs orientés explicitement correspondent à ceux qui ne suivent pas le sens conventionnel. La figure 10 donne les différents types de liaison.

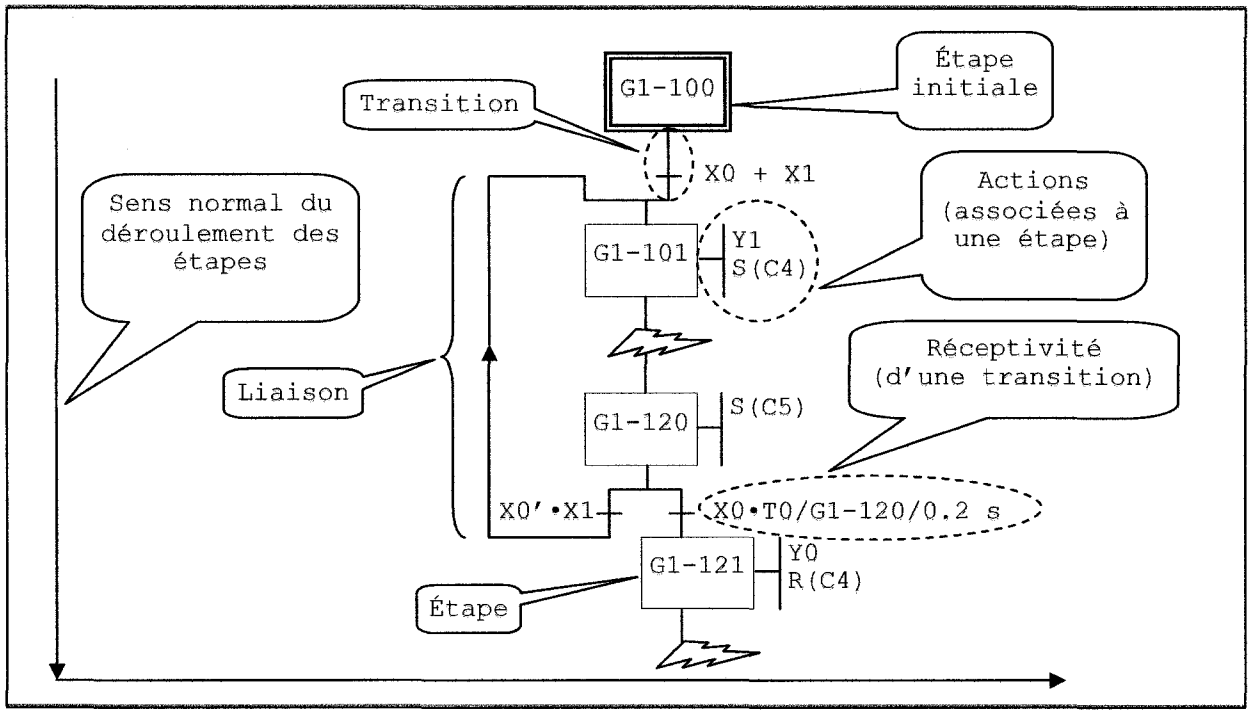

Figure 9. Éléments fondamentaux d'un grafcet

La spécification d'un système est habituellement constituée de plusieurs grafcets, dont l'exécution s'effectue en parallèle et de façon asynchrone.

En plus d'indicateurs booléens et de registres en provenance des capteurs et des actionneurs, les termes utilisés dans les diverses expressions admettent les éléments suivants :

1. l'usage d'éléments de mémoire ;

2. les expressions de temporisation comme l'exemple de la figure 9 noté « `T0/G1-120/0.2 s` » qui a la valeur *vrai* sitôt que l'étape qui la précède (G1-120) a été activée pour deux dixièmes de seconde et *faux* autrement (il y a beaucoup de variations pour les expressions de temporisation, mais cette dernière forme est la plus fréquente) ;

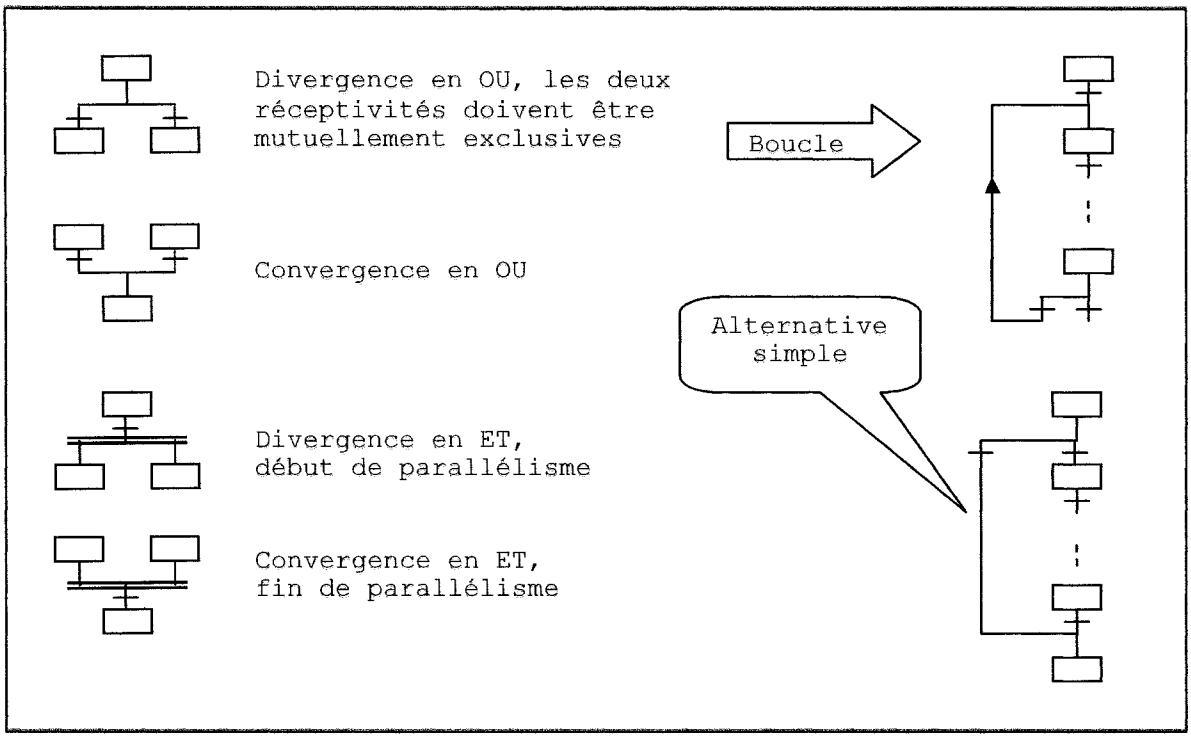

Figure 10. Les types de liaison possibles dans un grafcet

3. un indicateur booléen est implicitement associé à une étape (de n'importe lequel des grafcets du système), cet indicateur a la valeur *vrai* lorsque l'étape est active, et *faux* autrement ;

4. les événements, par exemple les fronts montant ↑X0 et descendant ↓X0.

La norme exige que les noms des indicateurs associés aux étapes aient la forme « *Xn* », où « *n* » est un numéro d'étape. Ceci entre en conflit avec les conventions de nomenclature des automates programmables *Koyo*, alors nous adoptons une autre convention toute aussi claire ; les noms d'étapes d'un grafcet ont la forme « *Gn-m* », où « *n* » est un numéro de grafcet et « *m* » un numéro d'étape dans le grafcet. L'indicateur associé aura le même nom que l'étape.

L'expression d'une réceptivité doit livrer un résultat booléen, qui peut comprendre des expressions relationnelles. La disjonction logique est exprimée par l'opérateur « + », la conjonction par un point et la négation par une apostrophe. On utilise les parenthèses pour l'associativité. Il faut distinguer ici entre opérations arithmétiques (qui peuvent se produire dans les listes d'actions) et expressions logiques puisque les expressions arithmétiques utilisent les opérateurs habituels. Bien sûr ces expressions admettent aussi des événements.

Parmi les actions, on distingue :

1. les actions continues (par exemple `Y1` dans la figure 9),

2. les actions mémorisées (on dit aussi impulsionnelles), par exemple *S*(`C4`) dans la figure 9 qui signifie l'affectation de la valeur *vrai* à l'indicateur booléen `C4`,

3. les actions conditionnelles de forme « *<action>* si *<condition>* »,

4. les actions temporisées qui ne sont rien d'autre que des actions conditionnelles dont la condition est une expression de temporisation.

Finalement, la norme est assortie de cinq règles, que l'on retrouve dans le tableau 2, qui définissent le comportement des grafcets.

Notez que R5 signifie, entre autres, qu'une temporisation ne serait pas réinitialisée, une action mémorisée (impulsionnelle) ne serait pas exécutée une seconde fois et une action continue ne changerait pas d'état.

Pour donner une signification plus précise au formalisme concernant la façon dont se déroule l'exécution d'une commande spécifiée par grafcets, il est nécessaire de traiter de son implémentation.

Tableau 2. Règles d'évolution des grafcets

R1 :	La situation initiale est constituée de toutes les étapes initiales de tous les grafcets du système, et uniquement de ces étapes. Autrement dit, au démarrage du système, seules les étapes initiales des grafcets sont actives.
R2 :	Une transition est soit validée, soit non validée. Elle est validée lorsque toutes les étapes immédiatement précédentes sont actives. Une transition validée est franchissable si sa réceptivité associée a la valeur *vrai*, elle n'est pas franchissable sinon. Une transition franchissable est obligatoirement franchie.
R3 :	Le franchissement d'une transition entraîne l'activation de toutes les étapes immédiatement suivantes et la désactivation de toutes les étapes immédiatement précédentes (s'il n'y a pas de divergence ou convergence en *ET*, une seule étape est concernée, en amont ou en aval).
R4 :	Plusieurs transitions simultanément franchissables sont simultanément franchies.
R5 :	Si une étape doit être à la fois activée et désactivée, elle reste active.

La norme complète cet ensemble de règles par deux algorithmes pouvant servir à l'interprétation d'une spécification par GRAFCET ; l'un avec recherche d'un état stable, l'autre sans recherche de stabilité (c'est-à-dire que chaque grafcet évolue à partir d'une seule transition franchissable par itération).

Certaines extensions ont été proposées pour amender la norme IEC 848 du formalisme GRAFCET :

1. les transitions sources qui n'ont pas d'étape précédente et les transitions puits qui n'ont pas d'étape suivante ;

2. les macro étapes qui déclenchent de façon synchrone l'exécution d'un autre grafcet (une sous-machine) et dont la transition de sortie est synchronisée sur l'atteinte

d'une étape de fin de la sous-machine ; il s'agit d'un genre d'appel de sous-routine ;

3. la capacité de contrôler des grafcets à partir d'un autre grafcet maître désigné sous les noms de « figeage » et de « forçage ».

Ces extensions relèvent plus de l'aspect structural que de l'aspect fonctionnel des grafcets. Ils aident à mieux structurer un système et facilitent la gestion de sa complexité, sans toutefois en augmenter la puissance. Du point de vue de génie logiciel, et dans la perspective d'un processus de développement de système de type classique, ces extensions sont importantes, mais pour ce qui est de notre problème, la génération de code pour des automates programmables, nous pouvons nous permettre de les ignorer.

3.2 GRAFCET, schémas de génération de code

Dans cette section, nous supposons que nous avons à implémenter une spécification par GRAFCET sur l'usine-école MPS. Une telle spécification a d'ailleurs été élaborée et programmée pour caractériser l'usine-école ; on peut la retrouver en annexe 2. Les schémas de génération de code de cette section sont donc spécifiques aux automates programmables DL205 de *Automationdirect.com*™. Les schémas sont donnés sous forme de diagrammes *Ladder Logic*.

Il nous faut d'abord donner forme aux notions fondamentales du formalisme, les notions d'étape (incluant l'étape initiale), de transition et de liaison. Ces dernières définissent la structure *Ladder* d'un grafcet dans un automate programmable.

La figure 11 donne la structure générale du code à générer pour matérialiser un grafcet sur un automate programmable DL205. Il faut noter ici que cette solution exclue le cas d'un grafcet à une seule étape. Ce dernier cas n'exige pas de transition (conséquence de **R5** dans le tableau 2) et autrement, se traite comme si on générait le code pour S_i dans la figure.

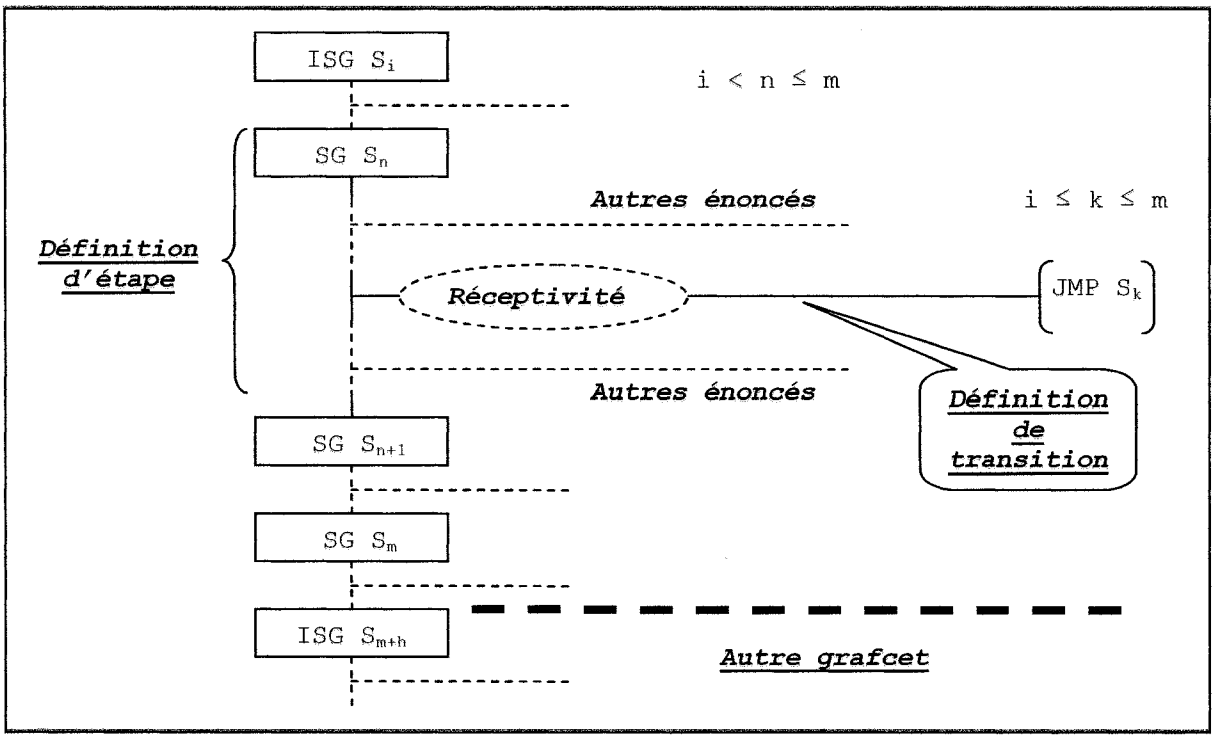

Figure 11. Schéma de code pour la structure d'un grafcet sur un PLC

Pour simplifier, nous avons supposé que les indices des S_i sont consécutifs, ce que l'automate programmable n'exige pas ; nous avons utilisé cette simplification pour mettre en évidence que l'étape cible d'une transition doit faire partie du grafcet pour lequel on génère le code. Il serait illégal de brancher d'un grafcet à l'autre. À proprement parler, l'automate n'exige pas non plus que les définitions d'étape soient contiguës, mais nous ne mentionnons ceci qu'à titre informatif.

On peut voir que le code correspondant à un grafcet se compose d'une séquence linéaire de définitions d'étape. Chaque étape est générée selon le schéma suivant.

1. Pour créer une étape initiale, on génère d'abord une instruction **ISG** (*Initial Stage* en *Ladder*). Pour une étape, on utilise une instruction **SG** (*Stage*) ; la seule différence entre les deux, c'est qu'au démarrage de l'automate toutes les étapes

initiales sont automatiquement activées. On doit associer un indicateur booléen `Sn` à une étape (relais d'étape). Les relais d'étape correspondent à des éléments mémoire (un bit) de l'automate, ils ont la valeur *vrai* si l'étape est active et *faux* autrement.

2. À l'intérieur de l'étape, on doit générer le code correspondant à la liste d'actions associées à l'étape à l'aide des schémas correspondant aux différents types d'action. Notez sur la figure, que le code correspondant aux actions (il peut ne pas y en avoir pour une étape) peut être étalé de part et d'autre du code généré pour une transition. Il y a cependant des règles à suivre.

3. Il faut aussi générer dans la définition d'étape le code correspondant à une transition. Pour l'énoncé *Ladder* correspondant à une transition, la position dans l'étape n'a pas d'importance.

La génération de code pour une transition utilise le schéma suivant.

1. On génère le code correspondant à l'expression de la réceptivité de la transition dans le grafcet, qui constitue la partie formule logique (*P*) de l'expression *Ladder*.

2. L'action associée à cette formule est une instruction **JMP** avec en argument le nom de l'indicateur booléen associé à l'étape cible. On peut inverser l'action du prédicat en utilisant un **NJMP** plutôt qu'un **JMP**, mais il est préférable d'utiliser une stratégie uniforme et d'inverser le résultat de la formule au besoin (puisqu'on dispose des instructions *Ladder* nécessaires).

Notons tout de suite qu'il peut y avoir plusieurs transitions définies pour une seule étape (divergence en *OU*). Elles sont toutes traitées de la même façon.

Aussi, la liste d'actions associée à un énoncé *Ladder* peut comporter plusieurs actions, mais si on associait d'autres actions (que l'instruction **JMP**) à un énoncé *Ladder* matérialisant une transition d'un grafcet, il faudrait que ces actions soient spécifiques à la transition (non pas à l'étape) ce qui ne se produit pas dans un grafcet. Encore une fois, il est préférable d'utiliser une stratégie de traduction uniforme et cohérente avec le formalisme, c'est pourquoi nous n'associons jamais d'autre action que l'énoncé **JMP** à la définition *Ladder* d'une transition d'un grafcet.

Il est pertinent de faire une remarque sur le déroulement de l'exécution des actions à l'intérieur d'une étape, particulièrement lorsque se produit une transition. Nous avons à plusieurs reprises déclaré que l'ordre dans lequel les actions sont spécifiées importait peu (en général). Ceci tient aux faits suivants.

1. Sur l'automate programmable DL205 le code de chaque étape (instruction **SG** ou **ISG**) génère une séquence linéaire de code machine qui sera exécutée si et seulement si l'indicateur associé à l'étape (relais `Sn`) a la valeur *vrai*.

2. Lorsqu'une séquence de code machine correspondant à une étape est exécutée, elle est exécutée du début à la fin.

3. L'effet de l'instruction **JMP** (ou **NJMP**) n'est pas celle d'un « *GOTO* » classique. Cette instruction modifie plutôt les indicateurs d'état associés aux étapes (relais `Sn`), elle met à *faux* la valeur de l'indicateur associé à l'étape courante et met à *vrai* la valeur de l'indicateur associé à l'étape cible (paramètre du **JMP**), mais elle n'interrompt pas l'exécution linéaire du code de l'étape courante.

L'effet général du partitionnement en étapes (instructions **SG** ou **ISG**) ressemble donc à une série d'instructions « *IF* » sans clause « *ELSE* » en programmation classique. Par exemple, en langage « *C* » nous obtenons :

```
if ( S0 ) then {
  <instructions associées à l'indicateur S0>
}

if ( S1 ) then {
  <instructions associées à l'indicateur S1>
}

...
if ( Sn ) then {
  <instructions associées à l'indicateur Sn>
}
```

Avant d'aborder la génération de code pour les différents types de liaison, et pour clore avec les schémas de code correspondant aux notions fondamentales du GRAFCET, nous donnons les schémas de génération de code pour différents types d'action.

Les étapes sont ultimement associées à une liste d'actions. Nous devons décrire les schémas de code pour implémenter les différents types d'action : mémorisée (ou impulsionnelle), continue, conditionnelle et temporisée. Nous avons sciemment différé la discussion des transitions temporisées jusqu'ici pour traiter les deux problèmes (actions temporisées et transitions temporisées) côte à côte puisqu'ils font appel aux mêmes concepts. Nous faisons donc une légère digression sur les transitions temporisées.

La figure 12 donne le schéma de génération de code *Ladder* pour les actions continues dans leurs formes conditionnelle et inconditionnelle. On voit aisément que la forme inconditionnelle utilise la forme conditionnelle avec une expression logique toujours vérifiée (l'indicateur spécial `SP1` fourni par le DL205).

La cible des actions continues est toujours constituée d'éléments booléens discrets (par exemple un actionneur) puisqu'une action continue dans un grafcet sert à activer ou désactiver un mécanisme pendant le séjour dans une étape. Il faut donc utiliser une instruction « *coil* » *Ladder* pour spécifier une action continue (conditionnelle ou non).

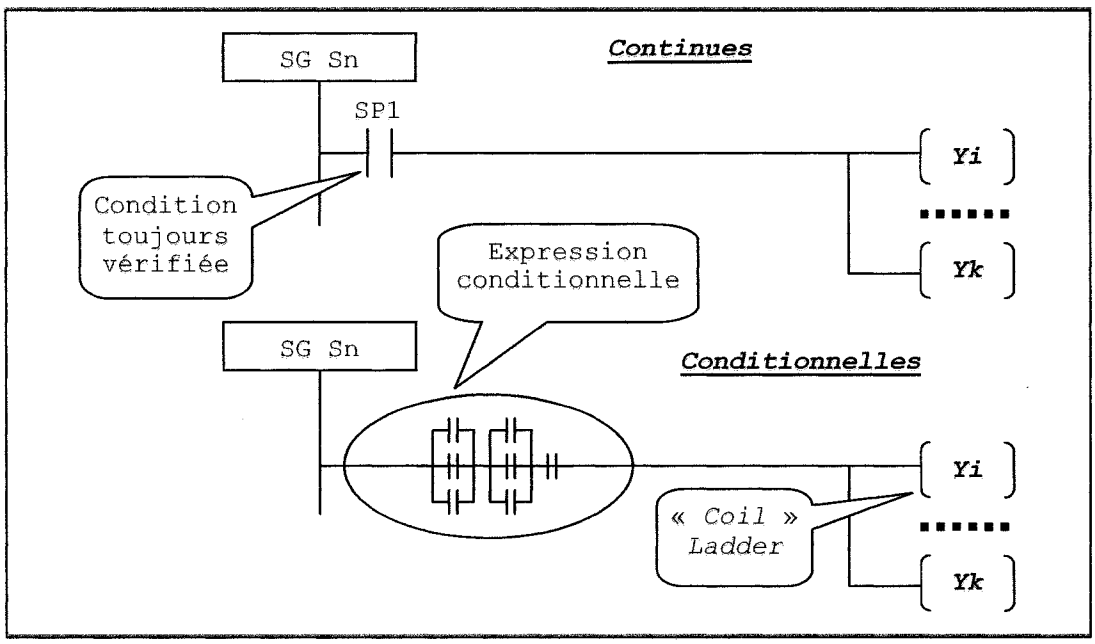

Figure 12. Actions continues et conditionnelles en *Ladder*

Plusieurs actions peuvent être regroupées dans une expression *Ladder*. On peut aussi utiliser une expression *Ladder* par action. C'est un choix pour la génération de code. Il peut y avoir plusieurs expressions *Ladder* d'actions continues ou conditionnelles dans une même étape, et leur position par rapport à l'expression de transition importe peu. Naturellement, dans ce contexte, l'expression conditionnelle d'une action ne devrait pas contenir d'événement, puisque l'occurrence d'un événement est un phénomène fugitif et donnerait à l'action une nature impulsionnelle (la norme ne l'interdit pas, mais cette pratique n'est pas conséquente avec l'intention des actions continues).

Une action conditionnelle suit fidèlement l'état d'une expression conditionnelle donnée dans le grafcet. Dans ce cas, la condition est générée par la formule logique associée à l'action dans le grafcet. Par exemple une action comme `Y2` si (`X3` • `Y0'`) générerait une expression *Ladder* dont la cible serait `Y2` et la formule logique (`X3` • `Y0'`) au lieu de `SP1`.

Il est important de noter que la cible d'une instruction « *coil* » en *Ladder* doit faire l'objet d'une surveillance étroite, car si un indicateur booléen est contrôlé à deux endroits différents il peut en résulter des conflits. Par exemple, deux grafcets indépendants, contrôlant tous les deux `Y2` avec des actions continues (ou un mélange d'action continues et mémorisées) pourraient aisément entrer en conflit. La politique la plus sage est souvent de n'utiliser qu'une et une seule instruction « *coil* » pour une cible donnée dans l'ensemble des grafcets constituants un système, ce qui est très restrictif. Lorsqu'il est nécessaire d'activer un mécanisme dans une étape et de le désactiver dans une autre, par exemple, ou encore de partager la gestion d'un mécanisme entre deux grafcets, il faut avoir recours à des action impulsionnelles.

La figure 13 donne le schéma de génération de code pour les actions impulsionnelles. Notez la présence de front montant ou descendant comme prédicats de l'expression *Ladder*.

Les actions impulsionnelles (souvent appelées actions mémorisées) sont des actions qu'il ne faut exécuter qu'une fois lors d'une visite à une étape, soit à l'entrée dans l'étape, soit à la sortie de l'étape. Par exemple, une action impulsionnelle représente un calcul, ou l'activation d'un actionneur (alors que sa désactivation se fait dans une autre étape), ou encore la lecture d'un instrument comme le micromètre du MPS.

Notons qu'une action conditionnelle dont l'un (ou plusieurs) des termes de l'expression conditionnelle est un événement, est techniquement une action impulsionnelle. Mais elle est traitée selon le schéma de génération de code d'une action conditionnelle que nous avons déjà vu. Les actions impulsionnelles traitées ici sont celles qui ne sont pas assorties d'une condition dont au moins un des termes est un événement.

On voit sur la figure 13 qu'on peut toujours soumettre l'exécution d'une action impulsionnelle à une condition (pourvue qu'aucun de ses termes ne soit un événement). Il s'agit de la forme conditionnelle d'une action mémorisée du type « *<action>* si *<condition>* » dans un grafcet.

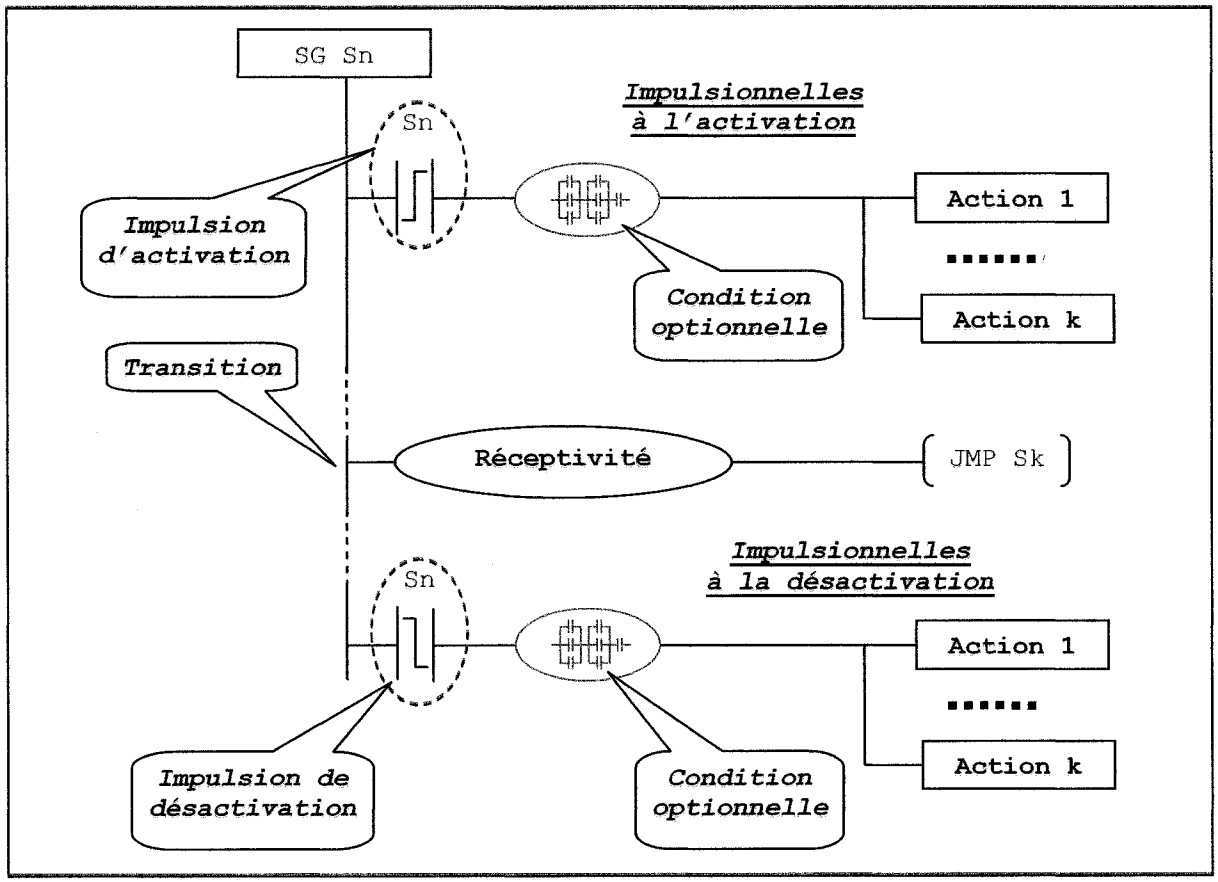

Figure 13. Actions impulsionnelles en *Ladder*

Notez que la forme impulsionnelle d'un « *coil* » *Ladder* est une instruction **SET** ou **RST**, avec la cible booléenne. Il n'y aura donc jamais de « *coil* » *Ladder* dans les listes d'instructions des expressions *Ladder* traduisant des actions impulsionnelles. Les instructions *Ladder* utilisées pour traduire des actions impulsionnelles sont appelées « *box* » en *Ladder*.

Dans la figure 13, on peut noter que l'exécution est soumise à l'occurrence d'un front montant ou descendant de l'indicateur booléen `Sn` associé à l'étape (un événement). C'est ce qui assure que la liste d'instructions du conséquent de l'expression *Ladder* n'est exécutée qu'une fois par passage dans cette étape. On a ainsi le choix d'exécuter les actions impulsionnelles à l'entrée ou à la sortie de l'étape. Cependant, si on décide d'exécuter à la sortie de l'étape, il faut placer

l'expression *Ladder* après l'expression de transition, sinon le front descendant sera déjà disparu au prochain cycle de l'automate et la liste d'actions ne sera jamais exécutée.

Il y a d'ailleurs un problème avec la version utilisant le front montant. Si une étape subséquente active une étape précédente (dans l'ordre du texte du programme *Ladder*), le front montant du relais d'étape ne sera jamais perçu à cette étape nouvellement activée.

La solution à ce problème est illustrée à la figure 14. Il s'agit d'insérer une étape *Ladder* additionnelle précédant (dans le texte du programme) l'étape utilisant le front montant. La seule action effectuée dans cette étape est une transition inconditionnelle à l'étape qui nous intéresse (celle qui utilise le front montant de *Sn*).

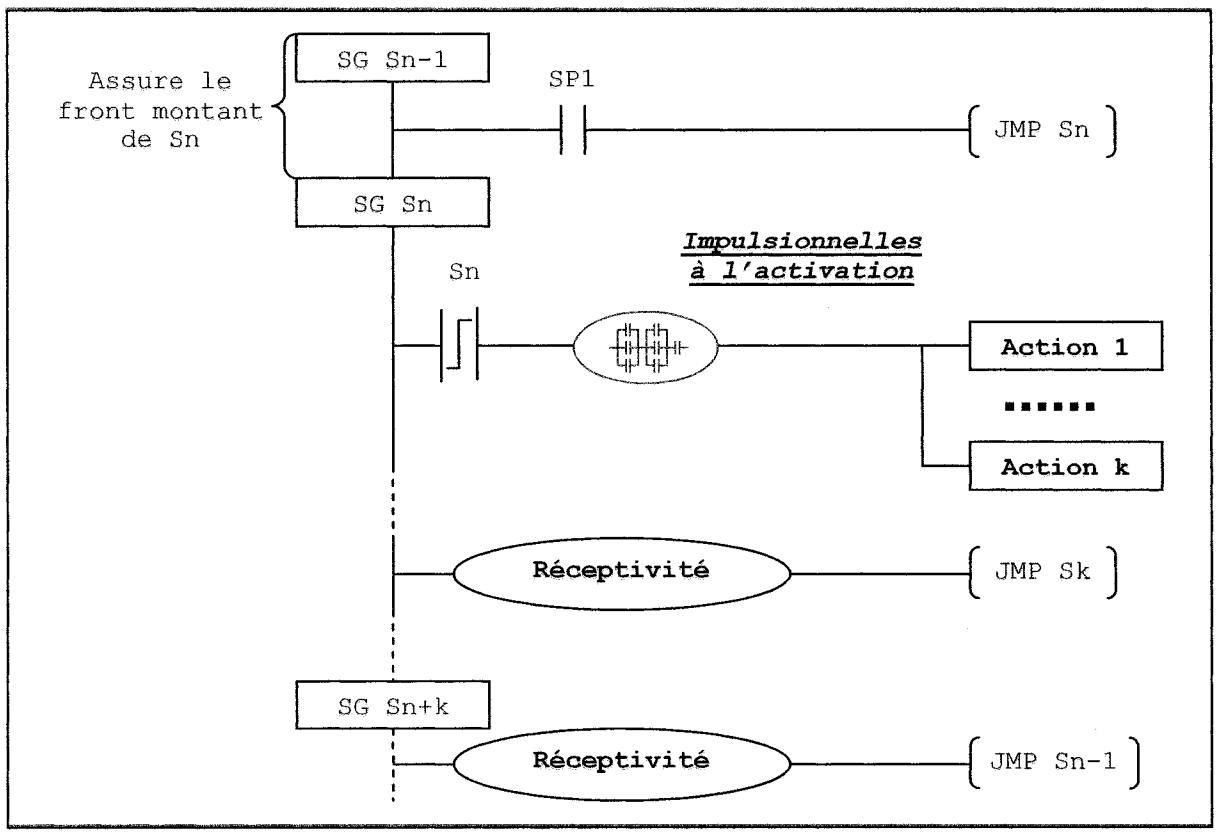

Figure 14. Actions impulsionnelles, solution au problème d'exécution à l'entrée d'étape

Naturellement, on peut avoir plusieurs expressions *Ladder* pour un ensemble d'actions impulsionnelles. On peut aussi les regrouper toutes dans une même expression. C'est un choix de génération de code. Pourvu que les exigences de positionnement par rapport à l'expression de transition sont respectées, il n'y a pas d'autres exigences de positionnement de ces instructions que celles qui sont naturelles aux actions elles-mêmes (par exemple une expression de calcul parenthésée).

La figure 15 présente quatre schémas de génération de code *Ladder* en relation avec les temporisations.

On note d'abord que pour utiliser une temporisation dans une étape, il faut précéder les énoncés *Ladder* qui utilisent cette temporisation d'un énoncé *Ladder* de minuterie **TMR**. En *Ladder*, chaque énoncé de minuterie **TMR** a deux paramètres : un indicateur booléen interne `Tn` et un délai en dixième de seconde. L'indicateur `Tn` a la valeur *vrai* lorsque la minuterie est échue. La minuterie (énoncé **TMR**) est réinitialisée à zéro sitôt que sa formule a la valeur *faux*. Dans notre cas, la minuterie est mise à zéro lorsque l'étape est inactive puisque nous utilisons l'indicateur d'étape `Sn`. La minuterie est active et cumule le temps en dixième de seconde tant que sa formule logique a la valeur *vrai* ; donc tant que l'étape est active.

Ainsi, dans l'ordre donné en figure 15, nous avons les schémas de traduction en énoncés *Ladder* pour les cas suivants.

1. Une transition temporisée qui peut faire partie d'une réceptivité dont l'expression pourrait être plus complexe. Ce qui distingue cet énoncé de transition par rapport à d'autres est l'usage de la forme positive de l'indicateur `Tn` qui signifie que la transition ne peut être effectuée avant le délai spécifié dans la réceptivité. Notez ici que dans les exemples de grafcets donnés, les transitions temporisées sont toujours relatives à l'étape qui précède immédiatement la transition. Nous n'utilisons pas de temporisation plus exotique.

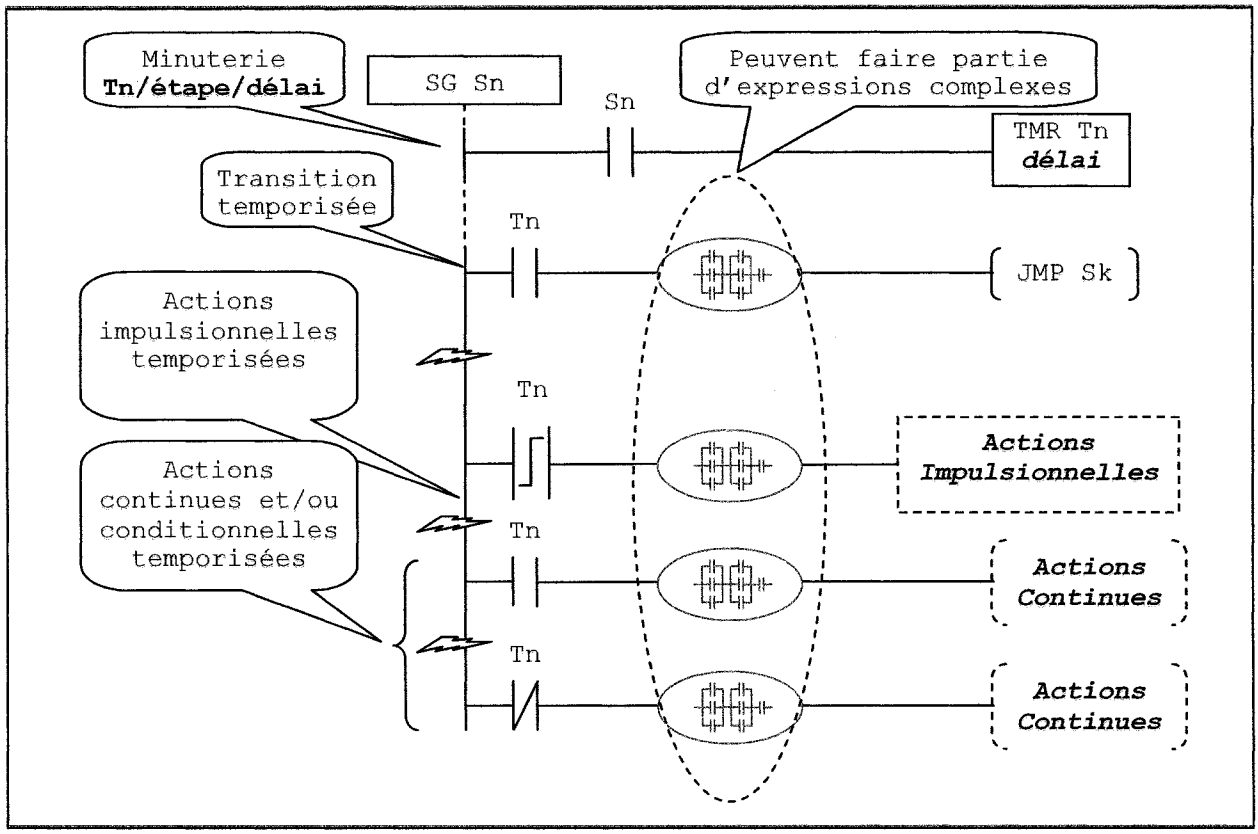

Figure 15. Actions et transitions temporisées en *Ladder*

2. Les actions impulsionnelles utiliseront le front montant de la minuterie auquel on peut associer une condition arbitraire.

3. Les actions continues et conditionnelles pourront avoir deux formes ; la forme utilisant la valeur *vrai* de l'indicateur `Tn` dans leur prédicat (forme en logique positive), qui signifie que la liste d'actions ne devient active qu'à l'échéance de la temporisation, et la forme utilisant la valeur inversée de l'indicateur `Tn` (forme en logique négative), qui signifie que la liste d'actions ne pourra être active que pour un délai égal ou inférieur à celui de la temporisation.

Pour terminer notre inventaire des schémas de code nécessaires pour établir une traduction d'une spécification GRAFCET vers un programme *Ladder*, il nous reste à déterminer

comment nous devons traduire les différentes liaisons : divergence et convergence en *OU*, divergence et convergence en *ET*.

La figure 16 illustre la traduction d'une divergence en *OU* en *Ladder*. On ne montre ici qu'une divergence en deux branches, mais la divergence en *OU* n'est pas limitée à deux branches ; il pourrait donc y avoir plus de deux énoncés de transition. Les réceptivités d'une divergence en *OU* devraient être des expressions conditionnelles mutuellement exclusives, autrement on pourrait se retrouver avec les deux branches de la divergence ayant des étapes actives simultanément (une forme de parallélisme).

La norme n'interdit pas cette situation, mais ce serait contraire à l'esprit de la divergence en *OU*. La divergence en *OU* est utilisée pour exprimer une alternative (comme un énoncé « *IF* » à une ou deux branches) ou pour contrôler une itération (énoncé « *WHILE* ») dans un grafcet, comme on le ferait en programmation structurée.

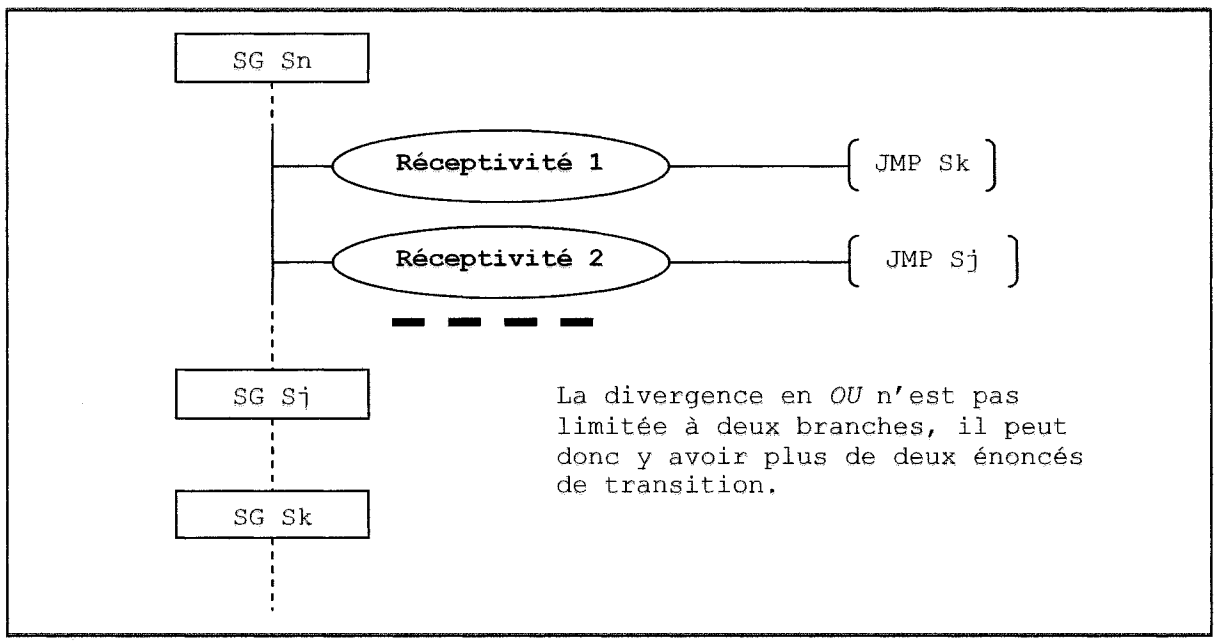

Figure 16. Divergence en *OU* en *Ladder*

La convergence en *OU*, le cas où deux (ou plusieurs) branches d'une alternative se rencontrent, ne demande aucun autre schéma de génération de code que ceux que nous avons déjà vus. En effet, il suffit que l'étape cible des deux énoncés de transition (appartenant à deux étapes distinctes) soit la même.

La divergence en *ET*, dont on peut voir le schéma de génération de code à la figure 17 ne présente pas de problème particulier. Il suffit d'associer à l'expression de la réceptivité une liste d'énoncés **JMP**, un pour chaque étape cible de la liaison. Toutes ces étapes deviendront actives immédiatement après l'exécution de la liste d'énoncés **JMP**.

La convergence en *ET*, présentée à la figure 18, requiert cependant l'usage de deux instructions spéciales : l'instruction **CV** qui définit une étape de convergence et l'instruction **CVJMP**. Cette dernière agit comme un rendez-vous, c'est-à-dire que son exécution désactive toutes les étapes de convergence qui la précèdent immédiatement et active l'étape qui est sa cible (son paramètre).

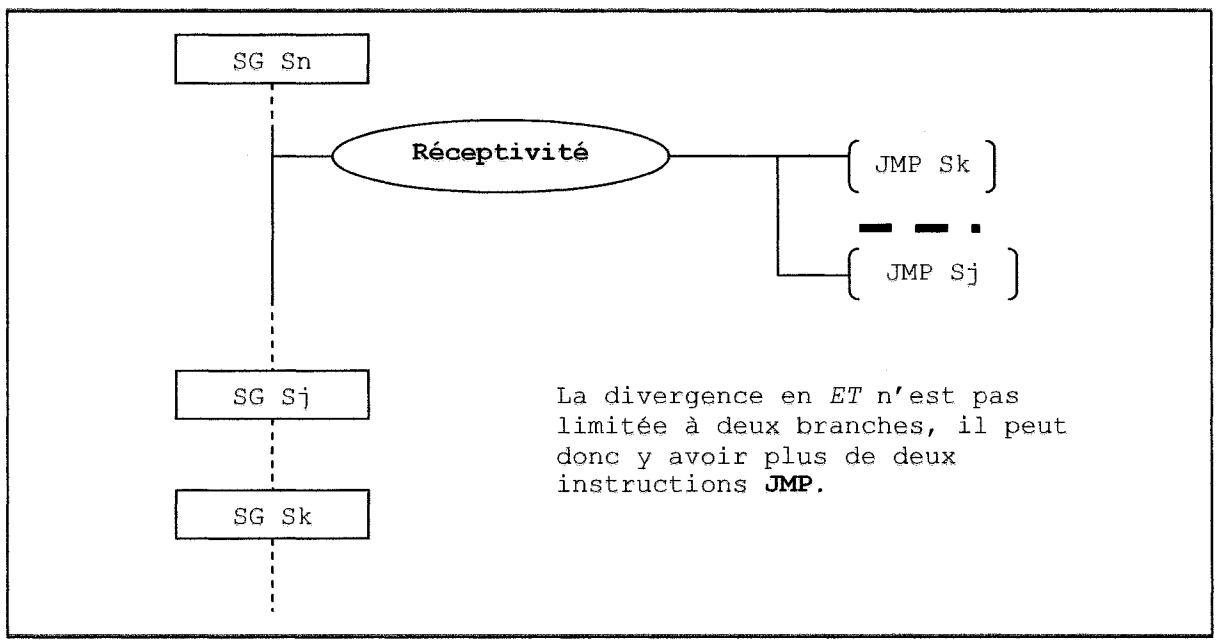

Figure 17. Divergence en *ET* en *Ladder*

Notez bien que toutes les étapes de convergence (instructions **CV**) d'une même convergence en *ET* doivent être groupées ensemble et suivies immédiatement (avant la définition d'une autre étape) de l'instruction de transition de convergence (**CVJMP**). Dans cet arrangement, l'instruction **CVJMP** ne s'exécutera que lorsque toutes les étapes de convergence qui la précèdent sont actives (la condition du rendez-vous).

Les instructions **CV** requièrent un indicateur booléen d'étape comme paramètre de la même façon que les instructions **SG**. Ces deux instructions *Ladder* définissent une étape. Cependant l'instruction **CV** ne peut être utilisée que dans le cadre d'une transition de convergence (matérialisation d'une convergence en *ET*) dont le contexte exact est illustré à la figure 18.

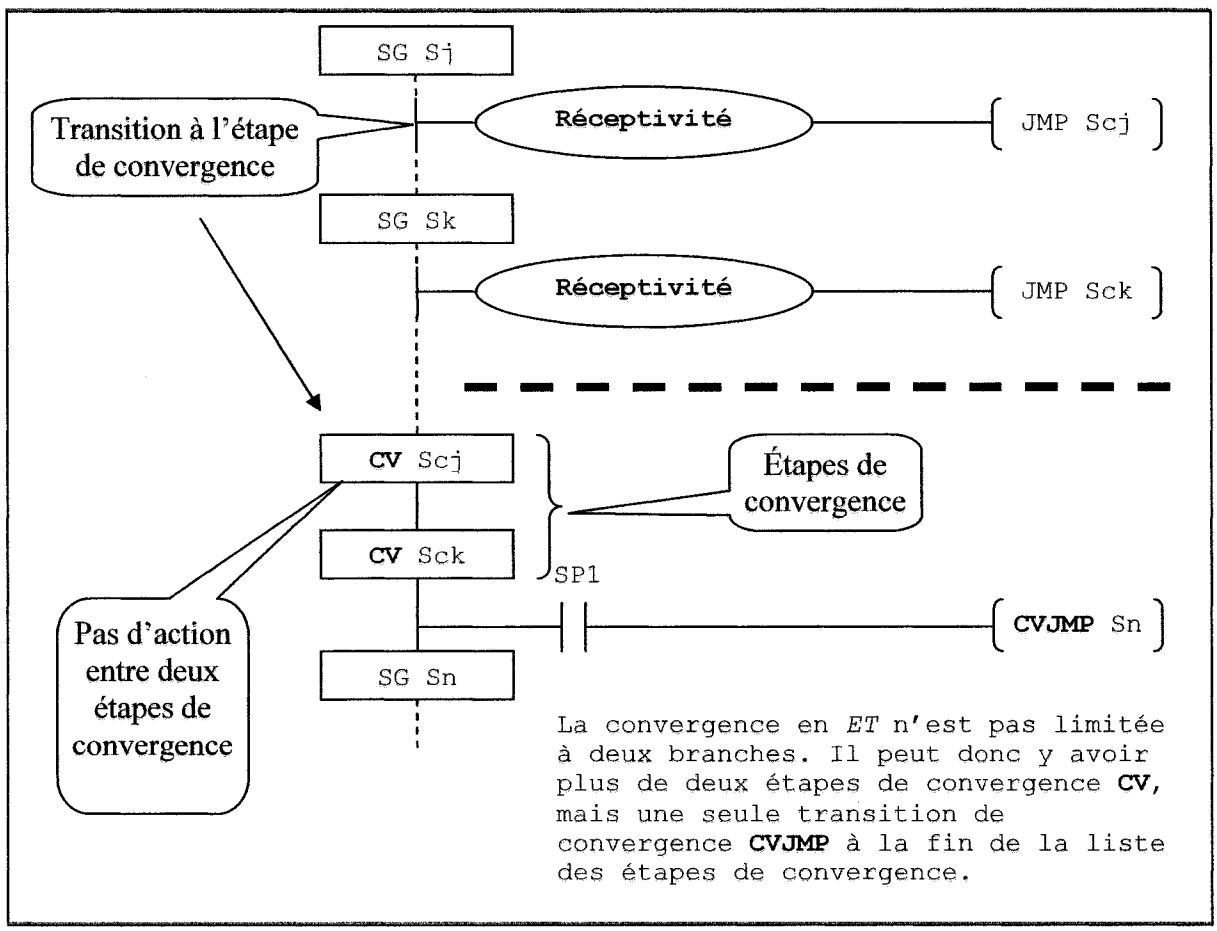

Figure 18. Convergence en *ET* en *Ladder*

Contrairement à l'instruction **SG**, aucune instruction *Ladder* ne peut être associée à une étape définie par une instruction **CV** sauf pour la dernière de la liste (*Sck* dans la figure). La figure n'illustre qu'une convergence en *ET* à deux branches, mais le schéma se généralise pour une convergence à plus de deux branches.

Divergence et convergence en *ET* introduisent une certaine forme très limitée de parallélisme dans un grafcet. Effectivement, à l'intérieur d'un seul grafcet, c'est la seule forme admissible de parallélisme. On peut voir l'effet comme le démarrage simultané de plusieurs machines composant un ensemble, mais la « boucle principale » doit attendre ensuite que toutes ces machines terminent leur tâche respective avant de reprendre son cours. C'est là une limitation très stricte (au niveau structural).

Les divergences et convergences en *ET* ne sont qu'un cas particulier de l'exécution parallèle des grafcets dans un système. Dans le premier cas, les mécanismes de synchronisation sont implicites au formalisme (synchronisation de type rendez-vous).

Un mécanisme de synchronisation entre grafcets est aussi nécessaire dans un système, puisque ceux-ci s'exécutent en parallèle. La synchronisation entre grafcets est en fait beaucoup plus utile que la divergence en *ET*. Bien qu'il ne s'agisse pas à proprement parler d'un schéma de code, nous illustrons tout de même, avec la figure 19, une méthode générale pour effectuer une synchronisation de type *rendez-vous* entre deux grafcets. Cette méthode se généralise aisément pour plusieurs grafcets. Il s'agit alors d'utiliser un grafcet pour coordonner le *rendez-vous*. Dans la figure 19, on peut voir que si deux grafcets ont besoin de se synchroniser, la forme suggérée procure un interlock symétrique, c'est-à-dire que les deux grafcets doivent attendre tour à tour que l'autre ait franchi une certaine étape avant de progresser. Cette forme de rendez-vous symétrique fonctionne sur les automates programmables *Koyo*. S'il y avait plus de deux grafcets impliqués, il faudrait avoir recours à des expressions de réceptivité plus complexes et, à la limite, utiliser un grafcet dont la seule fonction est la gestion du *rendez-vous*.

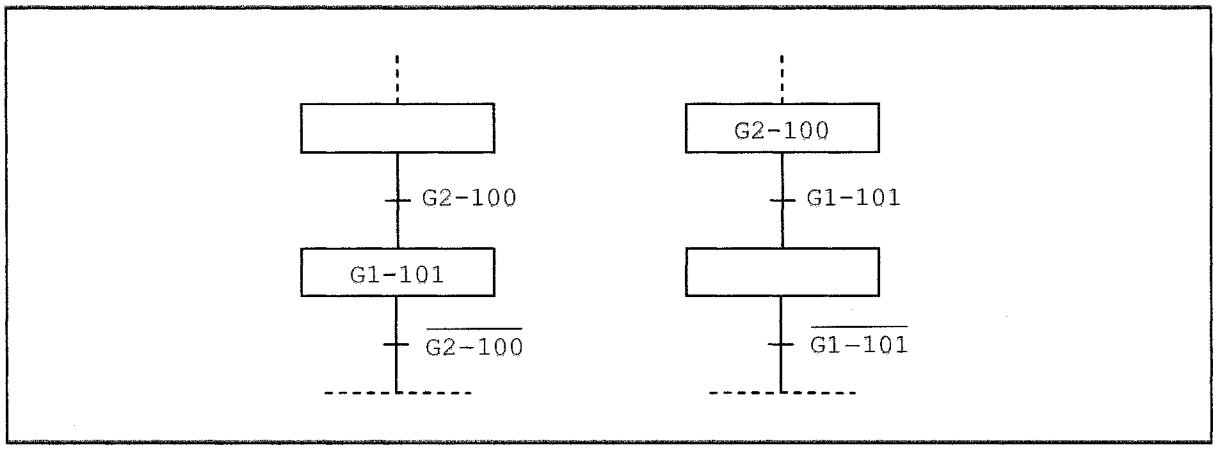

Figure 19. Synchronisation de type *rendez-vous* entre grafcets

Ceci termine l'examen de l'ensemble des schémas de code nécessaires pour établir une traduction des concepts fondamentaux d'une spécification par GRAFCET vers un programme *Ladder* sur les automates programmables *Koyo*.

3.3 Automates programmables et aspect algorithmique du GRAFCET

Pour terminer la traduction de grafcets en code pour des automates programmables, nous devons aborder les problèmes soulevés par l'exécution, dans l'automate programmable, du code généré. Ceci couvre les questions relatives à l'interprétation des grafcets traduits par les schémas de génération de code par rapport à l'aspect algorithmique soulevé dans la norme. Il est légitime de se demander si le code généré par les schémas de génération de code s'exécute conformément à ce que la norme spécifie, et sinon, quels sont les problèmes que cela soulève.

L'exécution du code généré pour un automate programmable ne correspond pas à ce que spécifie la norme. La norme donne deux algorithmes de simulation des grafcets ; l'un avec recherche d'un état stable et l'autre sans. L'algorithme avec recherche d'un état stable itère sur l'ensemble des grafcets d'un système, les faisant avancer jusqu'à ce qu'aucun d'entre eux ne puisse progresser davantage avant de mettre à jour le nouvel état des actionneurs calculés au

cours du cycle. L'algorithme sans recherche de stabilité, ne fait progresser chaque grafcet que d'une étape au plus au cours d'un cycle.

Nous avons reproduit à la figure 20 le pseudo code illustrant l'exécution du code généré pour la programmation par étape. Il n'y a bien sûr aucun parallélisme réel sur un automate programmable puisqu'il s'agit d'un ordinateur à un seul processeur. Le code des étapes est exécuté en séquence selon l'ordre du texte du programme si l'étape est active. On obtient un parallélisme apparent du fait que plusieurs étapes peuvent être actives à n'importe quel moment.

Une instruction de transition d'étape *Ladder* (**JMP** ou **CVJMP**) n'effectue pas un transfert de contrôle (*GOTO*), mais change simplement l'état des indicateurs booléens d'activation d'étape (les `Sn`). Une instruction « **JMP** `Sk` » exécutée à l'intérieur de l'étape `Sn` lorsqu'elle est active, résultera en une séquence « `Sn` ← *faux* » et « `Sk` ← *vrai* » en plus d'exécuter tous les autres énoncés associés à l'étape `Sn`. Si l'étape `Sk` se trouve à suivre l'étape `Sn` dans le texte du programme, elle sera exécutée à l'intérieur du cycle courant, si par contre `Sk` précède `Sn` dans le texte, il faudra attendre le cycle suivant pour son exécution. Puisque `Sk` pourrait à son tour effectuer une transition à l'intérieur du même cycle, on voit que ce mécanique d'exécution ne correspond pas à un algorithme avec recherche d'un état stable (cas où `Sk` précède `Sn`) ni à l'algorithme sans recherche d'un état stable (cas où `Sk` suit `Sn`).

Pour pouvoir remédier cette situation, il faudrait que les automates programmables cibles soient munis d'instructions d'itération, ce qui les rendrait possiblement moins déterministes.

Concernant les automates programmables à notre disposition, il ne semble pas possible de résoudre ce problème. Nous devons donc nous poser la question de la viabilité de la traduction que nous proposons et tenter d'identifier les problèmes qu'occasionne cette situation pour pouvoir, soit y remédier, soit les contourner.

```
<Acquisition de l'image mémoire des capteurs>

if ( S0 ) then {
  <instructions associées à l'indicateur S0>
}

if ( S1 ) then {
  <instructions associées à l'indicateur S1>
}

...
if ( Sn ) then {
  <instructions associées à l'indicateur Sn>
}

<Mise à jour de la valeur des actionneurs>
```

Figure 20. Exécution des étapes sur un PLC *Automationdirect.com*™

La question de la viabilité est difficile à vérifier. Dans le contexte d'un système à un seul grafcet, même si l'atteinte d'un état stable peut requérir plusieurs cycles, il semble intuitivement raisonnable de supposer (nonobstant un grafcet n'ayant pas d'état stable) que cet état stable sera ultimement atteint. La seule différence avec l'algorithme utilisant recherche de stabilité est que les actionneurs sont mis à jour plus d'une fois pendant la recherche de cet état stable. Pour des systèmes à plusieurs grafcets, le même raisonnement intuitif peut être fait. La solution exposée en annexe 2, obtenue à l'aide des schémas de génération de code suggérés dans cette section, montre que cela est possible et dans ce cas, viable, sur les automates programmables *Koyo*.

Quant aux autres problèmes, ils sont tous liés à l'ordre implicite d'exécution qu'impose la logique d'exécution de la figure 20.

Il y a principalement deux problèmes liés à la séquence implicite d'exécution imposée par les automates programmables :

1. la nécessité d'avoir des rendez-vous symétriques lorsque deux grafcets doivent se synchroniser ;

2. la perte d'événements liés aux actionneurs et aux indicateurs d'état des étapes.

À la figure 19, il y a une forme de grafcet qui implémente un *rendez-vous* symétrique entre deux grafcets, c'est-à-dire que les deux grafcets impliqués devront bloquer tour à tour (s'attendre mutuellement) pour passer le *rendez-vous*. La forme asymétrique (c'est-à-dire où un seul des deux grafcets attend l'autre) se comportera différemment selon que l'un ou l'autre des grafcets est exécuté d'abord. Il y a donc une différence de comportement induite par le séquencement. La forme symétrique n'est pas susceptible à cette différence de comportement. Il s'agit d'une heuristique de programmation.

Le problème de la perte d'événements liés aux actionneurs et aux indicateurs d'état des étapes est plus sérieux. La Figure 14 nous donne une heuristique de génération de code pour éviter la perte d'un événement d'activation d'étape dans le cas où on effectue une transition (**JMP**) vers l'arrière (dans le texte du programme *Ladder*). Cette heuristique est rendue nécessaire pour pouvoir exécuter les actions impulsionnelles liées à une étape au moment de l'activation de cette étape. Le lien avec le séquencement est clair. Plus grave encore, l'usage d'un événement d'activation ou de désactivation d'actionneur (forme ↑Yn ou ↓Yn), pourrait ne pas fonctionner selon sa position dans le texte du programme. Un actionneur peut être modifié n'importe où dans le texte du programme, un front montant ou descendant de cet actionneur (correspondant à un changement d'état) ne persistera que jusqu'à la fin du cycle courant. Si le texte d'un énoncé se servant d'un événement d'actionneur précède le texte de l'énoncé qui le modifie, le changement d'état de cet actionneur ne sera jamais perçu. Il n'y a pas d'autre solution à ce problème qu'un positionnement correct des énoncés.

CHAPITRE 4
LA THÉORIE DU CONTRÔLE DES SYSTÈMES À ÉVÉNEMENTS DISCRETS

La théorie du contrôle des systèmes à événements discrets (SED) est une méthode formelle. Elle repose sur l'algèbre des langages réguliers. Contrairement au GRAFCET, elle permet le calcul d'un « contrôleur » pour un SED dont on décrit formellement le comportement, ainsi que les contraintes sous lesquelles il doit fonctionner, à l'aide d'automates à états finis déterministes (AFD). L'intérêt de cette théorie est sa capacité de synthèse automatique de la logique du contrôleur, garantissant au système en boucle fermée certaines propriétés comme l'absence d'impasse et la contrôlabilité.

On peut trouver un traitement en profondeur de cette théorie dans l'article de synthèse de Ramadge et Wonham [RM89], ainsi que dans le livre de Kumar et Garg [KG95]. Nous nous bornons ici à en donner une brève mise en contexte.

4.1 Théorie du contrôle des SED, brève mise en contexte

Un système à événements discrets appelé SED (aussi appelé DES en anglais pour *Discrete Event System*) est un système dont l'état évolue au fur et à mesure que se produisent des changements discrets, appelés événements, dans l'ensemble du système. Il est donc caractérisé par un ensemble d'états distincts et par un ensemble de transitions entre ces états, qui se produisent lors de l'occurrence asynchrone d'événements[1].

Ces caractéristiques permettent de modéliser le comportement d'un SED à l'aide d'un AFD. Le modèle de comportement d'un SED définit donc dans ce cas particulier un langage régulier. Il est alors possible d'utiliser l'algèbre des treillis pour manipuler ce langage (puisque l'ensemble des langages sur un alphabet avec la relation d'ordre partiel d'inclusion forme un

[1] Traduction libre de l'introduction de [KG95].

treillis booléen) afin de faire en sorte, par calcul, que le langage du contrôleur d'un SED possède certaines propriétés.

La théorie procède en divisant les événements en deux catégories : les événements contrôlables (par exemple la commande d'un actionneur) et les événements incontrôlables (par exemple la réaction de l'environnement à l'effet d'une commande). Si un événement incontrôlable peut se produire dans un certain état du système, il est impossible de l'inhiber, et tenter de le faire serait absurde. De plus, le système ne doit jamais être paralysé par l'effet du contrôleur.

Les deux propriétés que doit posséder le contrôleur d'un SED sont donc les suivantes.

1. Le contrôleur d'un SED ne doit pas mener le système dans une impasse qui immobiliserait totalement le système ; c'est la propriété de non blocage.

2. Le contrôleur d'un SED ne doit jamais tenter d'inhiber un événement dit incontrôlable, on dit alors qu'il est Σ_u-permissif (Σ_u représente l'ensemble des événements incontrôlables d'un système).

La théorie suggère implicitement une méthode pour extraire un contrôleur à partir d'un ensemble de spécifications sous forme d'AFD, dont voici les grandes lignes.

1. On doit d'abord mettre au point un AFD qui décrit, le plus fidèlement possible, le comportement du système lorsqu'il n'est soumis à aucune contrainte. Ce modèle est appelé le modèle du comportement libre du système et forme la base pour le calcul du contrôleur. Le langage généré par cet AFD comporte toutes les séquences d'événements pouvant se produire dans le système.

2. On doit ensuite modéliser sous forme d'un ou plusieurs AFD l'ensemble des contraintes que le système doit respecter. Il y en a deux grandes catégories :

   - les contraintes de vivacité qui décrivent comment le système progresse vers des buts spécifiques constituant des tâches à l'intérieur de l'activité du système ;

   - les contraintes de sécurité qui décrivent les situations qui ne doivent jamais se produire dans le système pour prévenir les accidents et les bris.

3. En combinant toutes ces spécifications (par produit synchrone de leur AFD), on obtient une spécification du comportement désiré du système. Cependant la spécification obtenue n'est pas nécessairement libre d'impasse ou même contrôlable.

4. Si la spécification n'est pas contrôlable, on peut calculer le plus grand sous-langage contrôlable de l'AFD obtenu. On s'assure de cette propriété en éliminant, par itérations successives, les états du contrôleur qui interdisent une transition sur un événement incontrôlable faisant partie du comportement libre du système. Il s'agit ici du calcul d'un sous-langage de celui du SED qui constitue un « point fixe » par rapport à la propriété de contrôlabilité.

5. On prend soin d'assurer à chaque itération que l'AFD ne donne lieu à aucune impasse. On s'assure de cette propriété en éliminant les états non finaux de l'automate à partir desquels il est impossible d'atteindre un état final (l'équivalent d'un état poubelle dans un AFD).

Si le résultat n'est pas vide (langage ne comportant aucune chaîne d'événements), l'AFD obtenu constitue le contrôleur le plus permissif respectant les contraintes spécifiées, un résultat

optimal. Un tel contrôleur est appelé un contrôleur par composition synchrone, et la théorie s'applique longuement à prouver l'équivalence d'un tel contrôleur avec un contrôleur utilisant une fonction de rétroaction.

On doit assumer que l'ensemble des spécifications fournies à l'algorithme de calcul sont justes et décrivent correctement à la fois le comportement libre du système et l'ensemble des contraintes de vivacité et de sécurité qui s'y appliquent.

4.2 Contrôleur par composition synchrone, implémentation directe

Les schémas de génération de code pour un contrôleur par composition synchrone sur un automate programmable sont étonnamment simples. Pour illustrer ceci, nous donnons un exemple très simple. La figure 21 donne la spécification du comportement libre du vérin d'injection de la station de livraison du MPS (`C1.Livraison.Injecteur` en annexe 1).

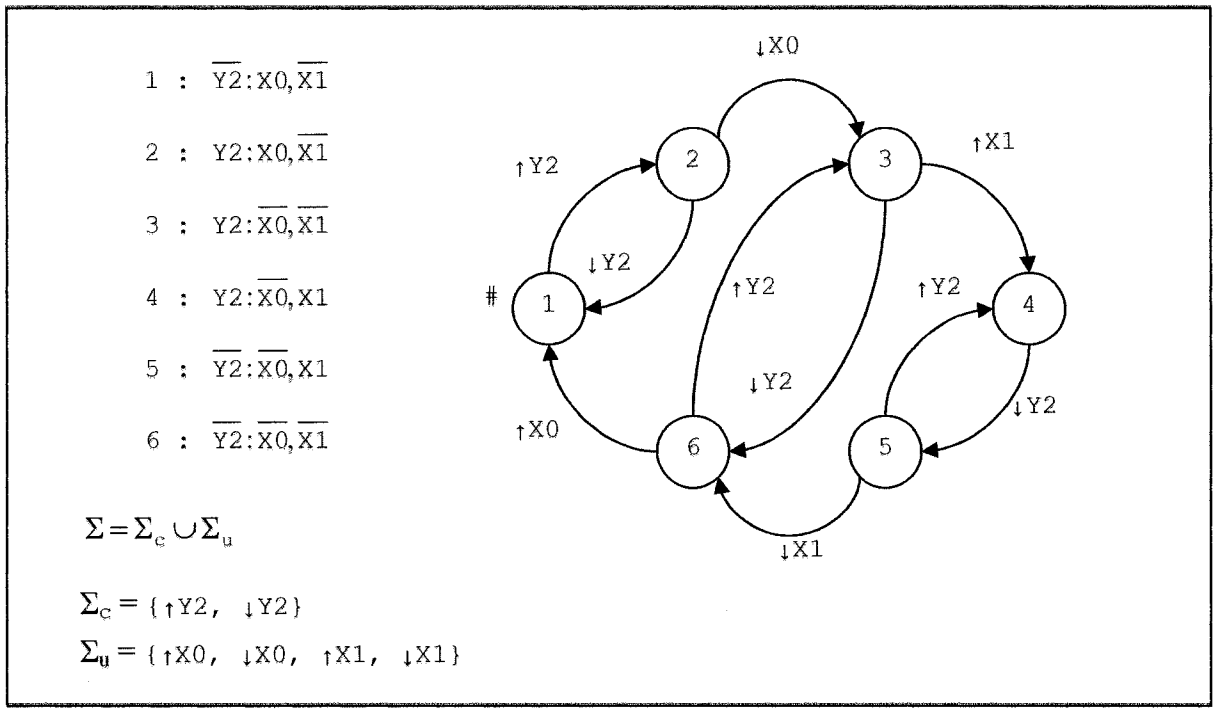

Figure 21. Comportement libre du vérin d'injection de la station de livraison du MPS

Pour nous aligner sur les conventions de la théorie, nous définissons nos états relativement à la configuration des actionneurs et des capteurs du dispositif (un genre de vecteur d'état). Les événements contrôlables (Σ_c) sont les fronts montants et descendants des actionneurs (ici il n'y en a qu'un, `Y2`). Les événements incontrôlables (Σ_u) sont les fronts montants et descendants des capteurs `X0` et `X1`. L'état initial du dispositif (l'état n° 1) est dénoté sur le diagramme état-transition par une étiquette spéciale « # » près du nœud. Puisqu'il s'agit d'un AFD de comportement libre et que le langage d'intérêt est le langage généré par l'AFD, tous les états sont finaux (donc ils n'ont pas été spécialement marqués).

On peut noter sur la figure 21 que la chaîne d'événements « `(↑Y2↓Y2)*↑Y2↓X0↑X1` » conduit au même résultat que la chaîne « `↑Y2↓X0↑X1` » qui est plus simple et plus efficace. Ce genre de considération nous mène à élaborer une contrainte de vivacité locale telle que spécifiée en figure 22. Cette dernière figure nous donne le style général d'un automate de contrainte. On note qu'un l'automate de contrainte a le même alphabet que l'automate à contraindre (ou les automates dans le cas d'une contrainte entre deux ou plusieurs automates).

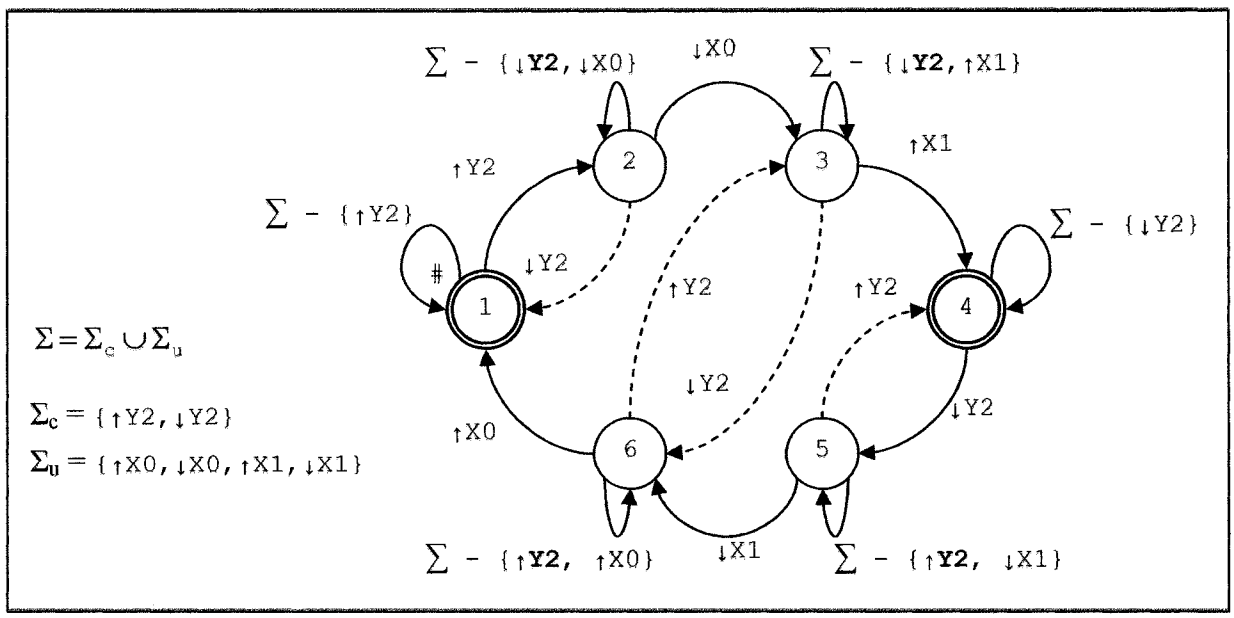

Figure 22. Contrainte de vivacité locale sur le vérin d'injection

Cette dernière propriété est importante car l'automate de contrainte ne doit pas ajouter de chaînes d'événements au comportement libre mais plutôt sélectionner les chaînes du comportement libre qui satisfont la contrainte.

Le produit synchrone d'automates (dont on trouve une définition formelle dans [KG95] à la section 1.3.1) correspond à une intersection de langages lorsque les alphabets sont identiques. Le résultat doit nécessairement être un sous-langage de celui des deux automates, ce qui correspond à ce que nous recherchons.

On peut noter aussi dans l'automate de contrainte l'ajout de transitions « identité » (transition à partir d'un état vers lui-même) dont les étiquettes ont la forme « $\sum$ - {↓**Yn**, ↓Xm, ...} ». Ce genre de transition a pour but de ne pas interdire les éléments du comportement libre qui ne sont pas concernés par la contrainte. Puisque le produit synchrone d'AFD aux alphabets identiques constitue une intersection de langages, ces transitions « identité » ne se retrouveront pas dans le résultat à moins qu'elles ne soient aussi spécifiées dans le comportement libre. Par contre si elles n'étaient pas spécifiées dans l'automate de contrainte, elles induiraient potentiellement l'élimination de transitions présentes dans le comportement libre.

Dans la forme « $\sum$ - {↓**Yn**, ↓Xm, ...} », les éléments en caractères gras sont les éléments que la contrainte vise à inhiber explicitement dans le comportement libre, les autres sont les transitions que l'automate de contrainte utilise. Pour mieux faire ressortir le but de l'automate de contrainte, nous avons inclus dans la figure 22, les transitions (en traits pointillés) que l'automate vise à inhiber dans le comportement libre.

On peut retrouver le résultat du produit synchrone des automates des figures 21 et 22 à la figure 23, dont l'effet est d'éliminer les chaînes d'événements ayant la forme « ...(↑Y2↓Y2)*↑Y2... » en les remplaçant par des formes « ...↑Y2... ». On peut considérer que l'automate de la figure 23 constitue un contrôleur par composition synchrone du vérin d'injection du MPS, si l'on ne considère que le vérin et sa contrainte de vivacité locale.

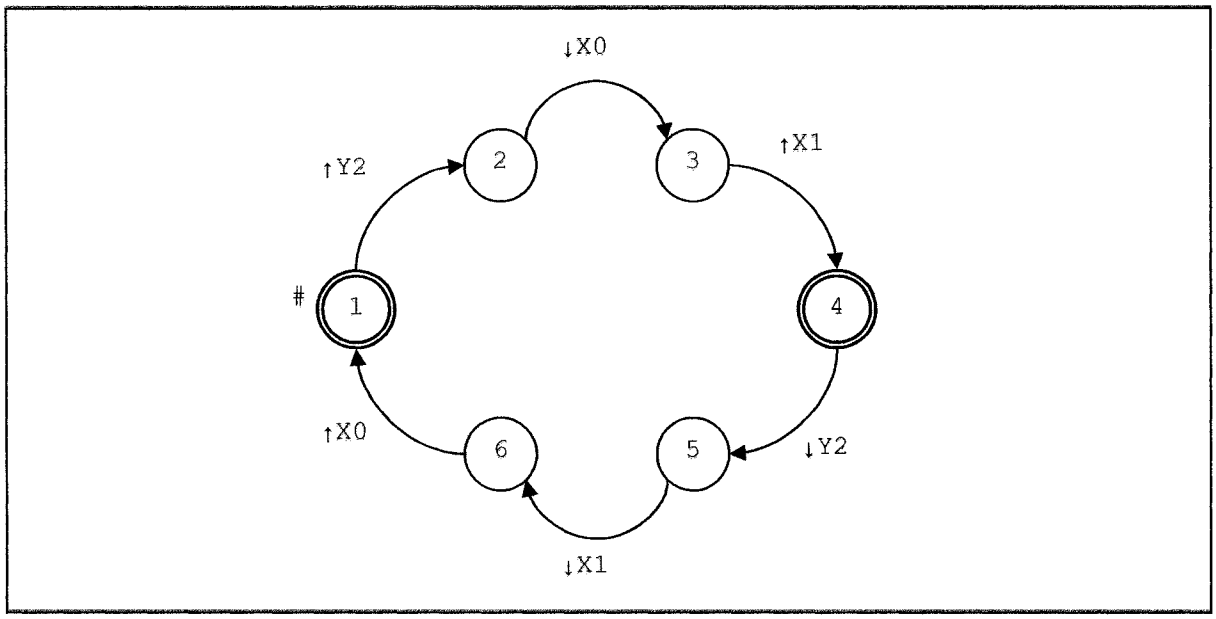

Figure 23. Résultat du produit du comportement libre et de la contrainte

Puisque que ce contrôleur a été obtenu à partir du comportement libre que l'on a restreint par contrainte, il suffit pour l'implémenter, de générer le code de l'automate programmable qui lui correspond, la fonction de rétroaction est implicite dans l'AFD du contrôleur.

On retrouve à la figure 24 les schémas de code *Ladder* qui correspondent aux transitions de l'automate. Puisqu'il n'y a pas d'action associée aux états dans l'automate, le code de l'automate programmable n'est constitué que d'étapes et de transitions (au sens *Ladder*). Une transition sur un événement incontrôlable (front montant ou descendant) est gardée par une expression booléenne d'occurrence de front. Une transition sur événement contrôlable cause l'événement en question (un front montant par l'instruction **SET**, un front descendant par l'instruction **RST**). Le contrôleur est monolithique, donc il n'y a qu'une seule étape active en tout temps.

Nous avons sélectionné notre exemple avec soin, autant pour éviter l'encombrement des figures que pour conserver l'intelligibilité des explications. Le dispositif modélisé est simple.

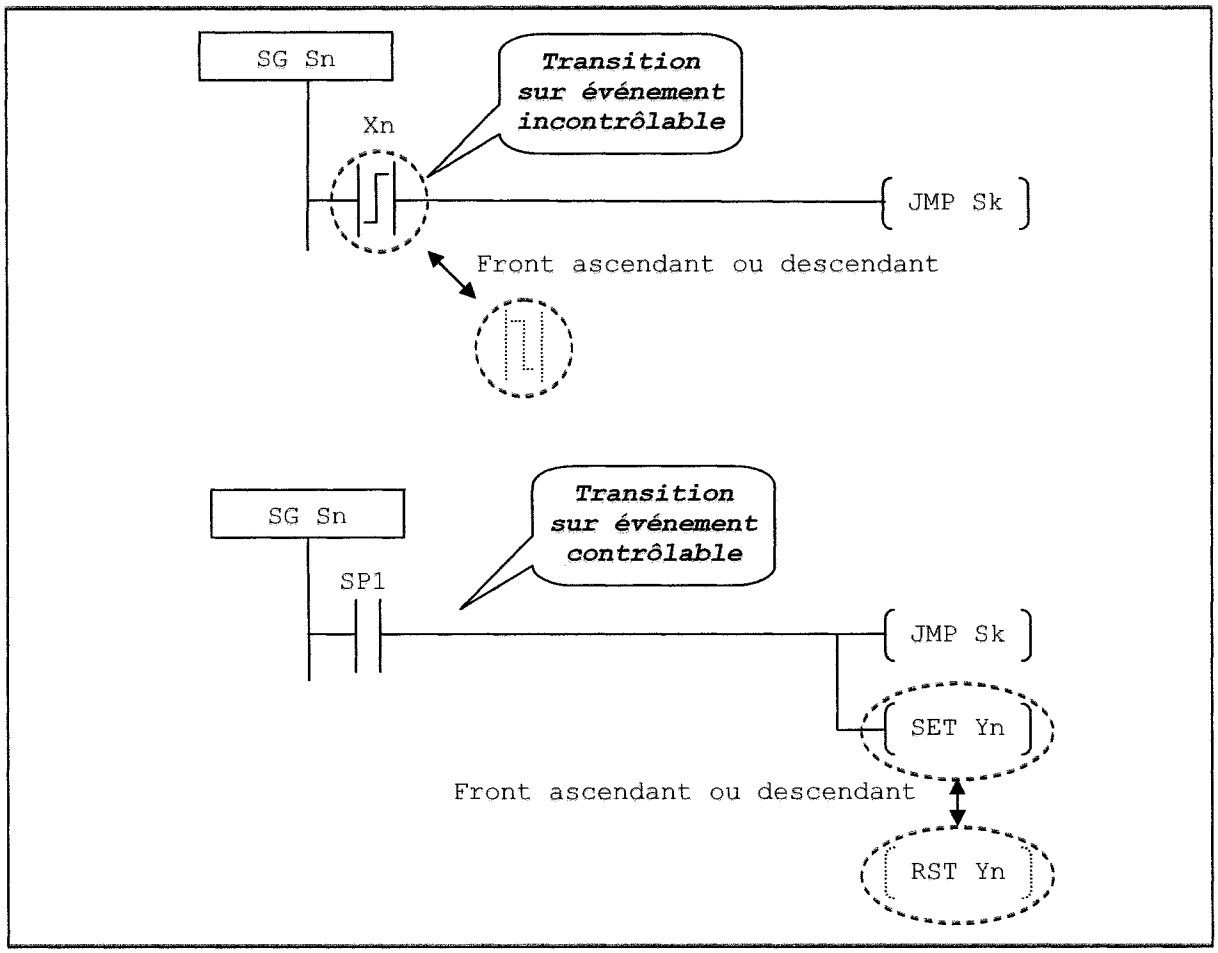

Figure 24. Schémas de code *Ladder* pour le contrôleur du vérin d'injection

La contrainte peut être spécifiée à l'aide d'une transformation de l'automate du comportement libre sur lequel on a pris soin de n'inhiber que des événements contrôlables. Dans ces circonstances, l'automate du contrôleur résultant a la même forme que l'automate du comportement libre, à quelques transitions sur des événements contrôlables près. De plus, l'automate obtenu pour le contrôleur ne comporte qu'une transition par état.

Malgré la simplicité de notre exemple, la procédure suivie pour l'obtention d'un contrôleur reste valide dans un cas plus général.

Nous devons cependant nous poser certaines questions pour nous assurer que nos schémas de génération de code sont adéquats.

1. Qu'arrive-t-il dans le cas plus réaliste où il y a plus d'une transition à partir d'un état ? Il y a au moins trois cas.

   Cas 1 : Il y a plusieurs transitions sur des événements incontrôlables.

   Cas 2 : Il y a plusieurs transitions sur des événements contrôlables.

   Cas 3 : Il y a plusieurs transitions sur des événements contrôlables et incontrôlables.

   On doit savoir si la situation peut se présenter, et le cas échéant, quels problèmes une telle situation présente pour la génération de code ainsi que les solutions qu'on peut y apporter.

2. On peut noter aussi dans la figure 24 que nous avons dû donner une sémantique précise à la notion de transition sur un événement contrôlable, c'est-à-dire que la transition est inconditionnelle et qu'elle cause en réalité l'événement. Il semble y avoir une incohérence entre cette sémantique et la sémantique d'un événement contrôlable dans un automate de contrainte (qui ne fait que réagir à l'événement qu'il soit contrôlable ou non). Y a-t-il vraiment une incohérence ?

Pour le premier problème, il est très facile d'imaginer des situations où ce problème se présente dans l'un ou l'autre des trois cas décrits. La figure 25 montre l'automate d'une contrainte réaliste tirée du MPS. Cette contrainte exprime que le vérin d'injection de la station de livraison ne peut entrer en fonction tant que la grue ne se trouve pas en position à la plate-forme de test.

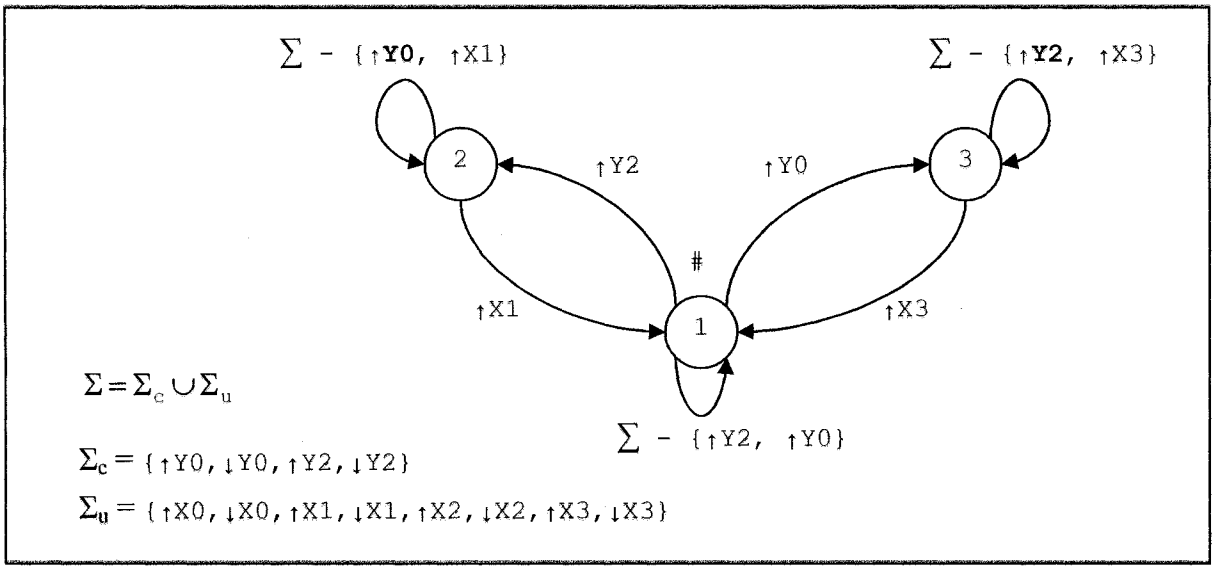

Figure 25. Expression d'une contrainte de sécurité du MPS

Symétriquement, la grue ne peut se déplacer en position au dock de chargement que si une injection n'est pas en cours ou qu'elle est terminée.

Il est évident que les deux transitions sur ↑Y2 et ↑Y0 se retrouveront dans le résultat du produit synchrone entre l'AFD de la figure 25 et ceux du comportement libre du vérin d'injection et de la grue (cas 2 du problème). Les situations impliquant les cas 1 et 3 du problème sont tout aussi fréquentes.

Le problème illustré à la figure 25 est d'ailleurs mentionné dans [CHK03] qui décrit la synthèse d'un contrôleur pour une ligne d'assemblage.

Dans cet article (en introduction), les auteurs suggèrent que ce genre de problème ne peut être résolu qu'en ajoutant une phase supplémentaire à la procédure de synthèse du contrôleur qu'ils nomment « *Controller Extraction* » (point 4 de la procédure de conception de contrôleur dans l'article). Les auteurs ajoutent que le contrôleur ne peut admettre au plus qu'une seule transition sur événement contrôlable dans n'importe lequel de ses états. La sémantique

proposée ici semble donc être similaire à celle que ces auteurs utilisent. Dans leur solution, la phase d'extraction du contrôleur a été réalisée par une méthode *ad-hoc* appliquée sur un contrôleur par composition synchrone obtenu par la procédure telle que décrite précédemment. Ce problème semble donc être un problème de synthèse de contrôleur, et notre schéma de génération de code de la figure 24 semble être adéquat.

Le cas 3 du premier problème n'est pas mentionné explicitement dans [CHK03]. Mais il semble nécessaire d'établir une politique d'arbitrage pour résoudre le problème. En effet, doit-on attendre l'occurrence de l'événement incontrôlable ou provoquer l'occurrence de l'événement contrôlable ? Une politique *ad-hoc* que l'on peut adopter serait de différer l'application d'un événement contrôlable le plus tard possible.

Finalement, le cas 1 peut se produire, mais ne devrait pas exiger l'application d'une politique de sélection à la phase d'extraction du contrôleur, puisque le calcul du comportement contrôlable maximal ne tentera jamais d'inhiber un événement incontrôlable. On peut donc, en modifiant le schéma de génération de code, accommoder plusieurs transitions sur des événements incontrôlables à partir d'un même état du contrôleur. Le principe consiste à rendre les conditions de garde des transitions mutuellement exclusives, et à procurer une transition privilégiée pour chaque combinaison d'événements simultanés (ce qui peut être coûteux). Pour trois événements (incontrôlables) e_0, e_1 et e_2, il y aura quatre combinaisons possibles d'occurrence simultanée, soit : (e_0, e_1), (e_0, e_2), (e_1, e_2) et (e_0, e_1, e_2). On voit aisément qu'il y a une explosion d'ordre exponentiel qui pourrait aisément devenir critique.

Nous ne pouvons pas éviter de rendre les conditions de garde mutuellement exclusives. Il faut se rappeler que l'instruction *Ladder* **JMP** n'est pas un transfert de contrôle, et que toutes les autres instructions d'une étape seront aussi exécutées même si un **JMP** est exécuté. La conséquence, si l'on ne rendait pas les conditions de garde mutuellement exclusives, serait potentiellement catastrophique. Nous nous retrouverions avec un automate qui fonctionnerait

de façon non déterministe (plusieurs de ses états effectueraient des transitions contrôlables ou non de façon simultanée).

Notons que ce problème est particulier à l'application de la théorie, car selon la théorie l'occurrence d'événements ne consomme pas de temps et qu'en conséquence deux événements ne peuvent pas être simultanés. Sur un automate programmable cependant, le cycle de l'automate requiert du temps, et deux ou plusieurs capteurs peuvent changer d'état durant ce laps de temps, ces événements seront perçus simultanément à la prochaine itération de l'automate programmable, peu importe si l'un s'est produit avant l'autre dans les faits. Notons que l'usage d'un ordinateur externe (plutôt que d'un automate programmable) ne changerait rien à ce problème.

Quant à notre deuxième problème, il est illustré en figure 25. Dans un automate de contrainte comme celui-ci, on interprète généralement les transitions sur ↑Y2 et ↑Y0 (des événements contrôlables) comme des réactions à l'occurrence d'événements. Il serait bizarre de penser que cet automate (exprimant une contrainte du système) provoque l'occurrence de ces événements ; son rôle est plutôt de les inhiber dans certaines circonstances. Tandis que dans l'automate du comportement libre de la figure 21, il est plus naturel de penser que l'automate provoque l'occurrence de ↑Y2 et ↓Y2, quoique ce ne soit pas clair. En revanche, dans l'automate du contrôleur de la figure 23, la signification intuitive est claire.

La théorie elle-même ne fait pas de différence entre événements contrôlables et incontrôlables quant à leur origine, les deux types d'événement se produisent spontanément dans un système, donnant au SED un caractère apparemment stochastique. Dans ce modèle des SED, il n'y a pas de notion de commande en tant que générateur d'événements contrôlables.

Ce problème est clairement formulé dans [CAR99], qui fait partie d'une série d'articles qui tente d'utiliser la théorie du contrôle des SED en conjonction avec le GRAFCET. Dans cette série d'articles, [ZAY99] s'attarde à établir les fondements de l'usage de la théorie du contrôle

des SED en conjonction avec le GRAFCET, [ZAY01] traite de l'usage de la théorie dans une démarche de vérification et validation d'une spécification par grafcets et [CAR99] ainsi que [ZAY99a] examinent l'aspect synthèse d'un contrôleur correct à partir d'une spécification par grafcets à laquelle est adjointe un modèle du comportement libre du système et de ses contraintes.

Tous ces articles proposent, pour donner une solution à un problème similaire au nôtre, de modéliser la commande d'un SED comme une machine à part entière spécifiée par un grafcet. En bref, les auteurs ajoutent au modèle une spécification par grafcets de la commande du système qui résout l'ambiguïté sémantique. Cette spécification résout du même coup les problèmes d'ordonnancement d'événements contrôlables de notre problème n° 1. À partir du modèle du comportement libre du système (incluant la commande, qu'on obtient en dressant le graphe des situations stables du grafcet), on peut calculer un contrôleur selon les principes de la théorie du contrôle des SED. Cette démarche est, jusqu'à présent similaire à la nôtre sauf pour l'addition du graphe des situations stables du grafcet qui ajoute à la complexité. Le contrôleur obtenu sert ensuite (soit dans une boucle en ligne ou hors ligne) à limiter le comportement (les actions) du grafcet de contrôle par rétroaction.

C'est bien sûr une solution au problème, mais l'addition du graphe des situations stables d'un grafcet ne fait qu'empirer la complexité du calcul du contrôleur déjà assez élevée.

Contrairement à cette approche, dans [CHK03] les auteurs ne tentent pas de résoudre l'ambiguïté sémantique de la théorie. On reconnaît que le calcul du contrôleur et son implémentation physique sur une machine réelle sont deux choses distinctes, et que le résultat théorique (le contrôleur par composition synchrone) demande des ajustements pour être implémenté. On doit résoudre la question sémantique dans ce contexte et on doit aussi résoudre en conséquence la question des transitions multiples à partir d'un état du contrôleur (la phase d'extraction du contrôleur). Ces préoccupations ne sont pas d'ordre théorique.

Notez que dans le cas d'un contrôleur obtenu par composition synchrone, le seul artéfact théorique dont on doit préciser la sémantique est le contrôleur lui-même ; la fonction de rétroaction est implicite dans la structure même de l'AFD du contrôleur. Ainsi, l'implémentation directe du contrôleur de composition synchrone permet d'éviter d'avoir à préciser la notion de « commande » pour laquelle il n'y a pas de support explicite dans la théorie.

En résumé, par rapport aux deux problèmes qui nous préoccupent, on peut considérer que le problème de l'ambiguïté sémantique du modèle de la théorie des SED relativement aux événements contrôlables n'est pas soluble dans le contexte théorique. Il doit être résolu au moment de l'implémentation d'un contrôleur obtenu par composition synchrone. De plus la solution de ce problème requiert la résolution des situations (dans le contrôleur) où plusieurs transitions existent à partir d'un même état. La solution de ce dernier problème ne peut pas se faire sans une connaissance approfondie du système modélisé puisque les choix qui doivent être faits auront un impact sur le fonctionnement du système. Il y a deux façons de résoudre ce problème.

1. On peut spécifier des contraintes d'ordonnancement supplémentaires qui auront comme effet de supprimer les transitions multiples.

2. On peut faire une inspection du contrôleur et résoudre les problèmes au cas par cas (solution *ad-hoc* adoptée dans [CHK03]).

Dans les deux cas, on doit extraire le contrôleur de la solution calculée sur la base de la théorie du contrôle, ce qui suppose une exploration de l'espace des états.

4.3 Difficultés avec l'implémentation directe

Le principal problème, dans le contexte de la génération de code pour des automates programmables à partir de contrôleurs obtenus par composition synchrone, est l'explosion

combinatoire de l'espace des états. C'est un problème connu mais ses ramifications le sont moins.

Tout d'abord, pour donner une idée de l'ampleur du problème, nous avons déjà donné à la figure 21 le comportement libre du vérin d'injection de la station de livraison du MPS. Par une démarche similaire, systématique et très homogène, on peut dresser le comportement libre de chaque dispositif des stations de livraison et de test de pré-usinage constituant la cellule de traitement n^0 1, qui est sous le contrôle d'un automate programmable *Koyo*. À titre d'exemple voici la division par dispositif et le nombre d'états requis pour modéliser chacun d'entre-eux (dans l'ordre de la figure 2).

1. Le silo de chargement à l'entrée requiert trois états, car le capteur `X6` a des transitoires.

2. Le vérin d'injection vu en figure 21, requiert six états.

3. La flèche de la grue de transbordement requiert neuf états.

4. La ventouse pneumatique qui sert de pince à la grue de transbordement requiert huit états.

5. La plate-forme de test requiert trois états, pour détecter l'absence de pièce ou la présence d'une pièce valide ou invalide.

6. Le vérin d'évacuation de la plate-forme de test requiert six états, son fonctionnement est similaire à (2).

7. L'ascenseur de la station de test a un fonctionnement similaire à (3) et requiert donc neuf états.

8. Le vérin du micromètre, de fonctionnement similaire à (2), requiert six états.

9. Le micromètre lui-même comporte trois états, résultat valide, invalide et indéterminé.

10. La passerelle de transfert à la station d'usinage (vide ou occupée) comporte deux états.

11. Le vérin, constituant la barrière de la passerelle, requiert quatre états.

12. Le signal en provenance de la station d'usinage (prêt ou non à admettre une nouvelle pièce à l'usinage, capteur `X4`) requiert deux états.

Le produit synchrone des automates de ces dispositifs donne :

$$3 \times 6 \times 9 \times 8 \times 3 \times 6 \times 9 \times 6 \times 3 \times 2 \times 4 \times 2$$

On obtient 60 466 176 états. Puisque l'addition d'automates de contrainte ne ferait qu'ajouter à ce total, il est bien évident qu'une implémentation sur un automate programmable *Koyo* muni d'une UCT DL250, qui ne peut accommoder que 1024 étapes *Ladder*, est impensable à moins de trouver une façon de fusionner des états. Même une implémentation sur un ordinateur externe plus puissant (avec un système d'exploitation plus conventionnel) n'est pas clairement faisable à priori. Sans compter qu'en ayant recours à un ordinateur externe on ajoute des composantes logicielles impossibles à caractériser temporellement avec exactitude. En conséquence, on sacrifie le comportement déterministe de l'automate programmable et on augmente les délais d'acquisition de l'état des capteurs et celui de mise à jour des actionneurs (problème de simultanéité dans la perception d'événements), entre autres problèmes.

À part l'impossibilité physique d'une génération de code naïve pour ce genre de contrôleur, on peut voir aussi que l'explosion combinatoire rend difficile l'exploration de l'espace d'états du

contrôleur. Nous avons mentionné, à la section précédente, qu'il est impossible d'implémenter directement (en général) un contrôleur obtenu par composition synchrone. Il est nécessaire de pouvoir apporter des solutions au problème de transitions multiples à partir d'un même état (cas où il y a des transitions contrôlables). Or, que l'on règle le problème au cas par cas, comme dans [CHK03] ou que l'on introduise des contraintes supplémentaires de séquençage (ce qui complexifie davantage), on doit explorer l'espace d'états du contrôleur produit par application de la théorie, pour en extraire un contrôleur que l'on peut implémenter.

Il est permis de croire qu'avec de bons outils informatiques (probablement graphiques), il serait pensable de faire cette exploration, mais sans ce genre d'outil, il ne semble pas très réaliste qu'on puisse y parvenir.

L'exemple de [CHK03] est instructif à ce sujet. Les auteurs s'attardent beaucoup à tenter de réduire la complexité du contrôleur résultant de la théorie. Ils utilisent entre autres les techniques suivantes.

D'une part les auteurs prennent soins d'effectuer les différents produits d'automates dans un ordre particulier, déterminé de façon heuristique. Nous avons déjà noté que l'ordre dans lequel les produits sont faits de façon incrémentale peut aider à faire des vérifications de contrôlabilité (ou de cohérence entre contraintes) plus faciles à analyser ; ceci se produit quand le plus grand sous-langage contrôlable d'un ensemble de dispositifs associés à leur contrainte se révèle être très petit, par exemple lorsqu'il n'y a pas (ou très peu) de parallélisme possible entre les dispositifs. La station de test pré-usinage en est un bon exemple sur le MPS.

D'autre part, les auteurs font un usage abondant des propriétés des différents langages spécifiés par les automates de comportement libre et de contrainte, pour éviter d'avoir à vérifier la contrôlabilité et la cohérence par produit synchrone explicite. Il n'est pas facile ici de savoir si ces considérations (l'usage de propriétés) ont influencé la façon dont ont été élaborés les AFD de la spécification des contraintes.

Tout ce qui précède illustre assez bien une autre difficulté de la théorie du contrôle des SED : il est difficile de définir le niveau de granularité requis pour la modélisation et il est difficile d'obtenir de bonnes spécifications sous forme d'AFD. Dans la plupart des cas, par exemple, la spécification du comportement libre n'est pas nécessairement fidèle à la réalité. Dresser une spécification fidèle ne pourrait être fait qu'avec un appareillage nous permettant d'examiner et d'analyser des traces d'événements du système en cours d'opération.

Similairement, la spécification modulaire de contraintes exige qu'on puisse vérifier la contrôlabilité, autant que possible de façon modulaire. Or non seulement existe-t-il en général plusieurs AFD pouvant exprimer la même contrainte, mais l'effet de l'une ou l'autre solution sur l'ensemble du système est en général impossible à prévoir.

Si toutes les contraintes s'exerçant sur un système peuvent être exprimées directement sur les AFD du comportement libre des dispositifs constituants, sans avoir recours à des AFD de contrainte distincts de ceux du comportement libre, alors il est possible d'élaborer un contrôleur par composition synchrone sans augmenter la complexité du calcul au-delà de la composition synchrone de tous les dispositifs ensemble. Cette complexité demeure trop élevée, mais l'automate de composition synchrone des comportements libres des dispositifs est nécessairement plus grand que l'automate du comportement légal du système, puisque l'automate du comportement libre génère des traces d'événements ne respectant pas les contraintes à satisfaire. On devrait, en conséquence pouvoir s'attendre à ce que le contrôleur contienne moins d'états et de transitions que l'automate du comportement libre du système.

Une telle méthode reposerait sur des spécifications détaillées des comportements libres, c'est-à-dire sur des spécifications où on peut (et on doit) associer chaque état d'un automate de comportement libre à une configuration de ses actionneurs et de ses capteurs (ceux qui sont pertinent à son fonctionnement). Nous avons déjà montré à la figure 23 le résultat de l'application d'une contrainte de vivacité locale que l'on a élaborée à partir de l'automate de comportement libre de la figure 21 (vérin d'injection de la station de livraison). Le résultat de

la procédure suivie est toujours un sous-automate de l'automate à contraindre, car il correspond à un sous-langage du dispositif.

Nous sommes en droit de penser que si la modélisation du comportement libre des deux dispositifs (par exemple le vérin d'injection et la grue de transbordement) est complète, dans le sens que chacun de leurs états correspond à une configuration précise de leurs capteurs et de leurs actionneurs respectifs, le résultat du produit synchrone des deux automates (possiblement pré-contraints avec les contraintes locales d'abord) pourrait servir de base à l'élaboration des contraintes supplémentaires applicables aux deux systèmes combinés (s'il y en a). De cette façon, par itérations successives, où on combine deux dispositifs déjà pré-contraints, on utilise le résultat (par inspection) pour le contraindre d'avantage et on répète cette manœuvre jusqu'à inclure tous les dispositifs, on devrait obtenir un contrôleur dont le nombre d'états ne peut dépasser celui de l'automate du comportement libre du système.

De plus puisque, à chaque itération, on contraint l'automate « combiné » à l'aide d'une version de lui-même, le résultat contraint contient le même nombre d'états que l'automate original sans contrainte (il s'agit d'une intersection). En fait, la procédure consiste presque (mais pas tout à fait) à combiner deux automates par produit synchrone et à éliminer du résultat (à la main) les transitions qui violent les contraintes du système. La procédure est un peu plus compliquée que cela parce qu'il faut appliquer les calculs de contrôlabilité et de non blocage à chaque itération. Mais on peut voir que la complexité de la solution augmenterait, soit aussi vite, soit moins vite que celle d'un produit synchrone pur et simple des comportements libres.

Bien sûr, il faut que tous les états du système (en termes de ses capteurs et de ses actionneurs incluant des dispositifs comme le micromètre du MPS) soient représentés pour que la solution ne requière que « d'éliminer » des transitions.

Aussi, une telle procédure devrait reposer sur une instrumentation logicielle adéquate pour permettre l'inspection de l'espace d'états des résultats intermédiaires. Mais on peut noter qu'un tel instrument logiciel n'a pas besoin de reposer sur des représentations graphiques (d'ailleurs impossibles avec tant d'états). Les spécifications complètes nous permettent d'associer une configuration (ou un petit nombre de configurations) des capteurs et des actionneurs à chaque état de l'AFD à inspecter. Il devient donc assez facile de trouver automatiquement quels sont les états de l'AFD considéré qui violent des contraintes de sécurité ou de vivacité (ou les deux), à condition bien sûr de pouvoir exprimer ces contraintes sous forme d'expressions booléennes.

4.4 Une approche alternative à l'implémentation directe

On retrouve dans [WON04] la notion de SBFC (*State Feedback Control*). Wonham définit un SFBC comme la fonction totale suivante :

$$f : Q \to \Gamma, \quad \Gamma = \{ \Sigma' \subseteq \Sigma \mid \Sigma' \supseteq \Sigma_u \}$$

Notez que les sous-ensembles d'événements constituant l'ensemble Γ contiennent tous les événements incontrôlables du système Σ_u.

Il définit aussi, à partir de f, une famille de fonctions f_σ (que nous appelons fonctions de rétroaction restreintes à σ) :

$$f_\sigma : Q \to \{0,1\}, \quad f_\sigma(q) = 1 \text{ ssi } \sigma \in f(q)$$

Ce qu'il est important d'observer ici, c'est que les fonctions f et f_σ sont définies sur Q, l'espace d'états de l'AFD du comportement libre d'un système. L'état de la machine représentant les contraintes n'apparaît pas dans ces définitions.

Ce qui est intéressant, c'est l'ensemble des fonctions de rétroaction f_σ pour σ contrôlable, puisque par définition f_σ est *vrai* pour σ incontrôlable quelque soit l'état de l'AFD du comportement libre (Σ_u-invariance). On ne considère donc que les fonctions f_σ qui indiquent quand un événement contrôlable est permis. Or puisque les f_σ sont définies sur Q, il suffit de connaître l'état courant de l'AFD du comportement libre pour savoir si un événement contrôlable (quel qu'il soit) est permis ou inhibé.

Enfin, puisque dans notre sémantique de génération de code, une transition sur un événement contrôlable est systématiquement déclenchée, il suffit d'implémenter, pour chaque σ contrôlable, une transition gardée par une condition dont l'expression est la disjonction des $f_\sigma(q)$ pour lesquelles la transition sur σ est permise. On ne s'intéresse donc qu'aux cas où $f_\sigma(q) = 1$.

La disjonction considérée risque d'être très volumineuse puisque f inclut un événement contrôlable σ (c'est-à-dire que cet événement est permis) même pour les états de l'AFD du comportement libre à partir desquels il n'y a pas de transition physiquement possible sur σ. Une simplification évidente est donc de restreindre l'ensemble des termes de la disjonction aux $f_\sigma(q)$ pour lesquels l'état considéré possède une transition sur σ dans l'AFD du comportement libre.

Aussi, il faut se rappeler que ce qui nous intéresse est le comportement du système sous contrainte, c'est-à-dire l'AFD du contrôleur. Puisque l'application de contraintes réduit généralement le nombre des états accessibles du comportement libre, il est possible de réduire encore plus le nombre de termes de la disjonction en ne considérant que les états du comportement libre accessibles sous contraintes, c'est-à-dire ceux qui font partie du contrôleur.

Il faut cependant que la propriété selon laquelle le calcul de la rétroaction ne nécessite que la connaissance de l'état du comportement libre reste vraie. Malheureusement, dans le cas général, cette propriété n'est pas toujours vérifiée.

Un contrôleur dérivé par composition synchrone avec l'algorithme de Ramadge et Wonham semble être équivalent à un DSFBC (section 7.6 de [WON04]) défini comme un fonction :

$$f : Q \times Y \rightarrow \Gamma, \quad \Gamma = \{\Sigma' \subseteq \Sigma \mid \Sigma' \supseteq \Sigma_u\}$$

où Y est l'ensemble d'états d'un AFD appelé la « mémoire » dans [WON04]. Comme précédemment, on peut définir, à partir de cette définition de f une famille de fonctions :

$$f_\sigma : Q \times Y \rightarrow \{0,1\}, \quad f_\sigma((q,y)) = 1 \text{ ssi } \sigma \in f((q,y))$$

On peut observer que dans des conditions où les contraintes s'exerçant sur un système dépendent exclusivement de l'état du comportement libre dans lequel le système se trouve, l'état de la mémoire n'est pas nécessaire au calcul de la rétroaction, et on se retrouve dans le cas d'un SFBC.

Le schéma de génération de code pour un SFBC se réduit donc à un ensemble d'énoncés *Ladder* de la forme générale illustrée à la figure 26. Il y a un énoncé par événement contrôlable dont la partie *action* cause l'événement. La formule logique est composée de deux conditions : une condition dite nécessaire et une condition dite suffisante pour effectuer la transition.

La condition nécessaire doit être une formule logique dont le résultat est *vrai* lorsque le système se trouve dans un état du comportement libre où une transition sur l'événement contrôlable considéré est possible.

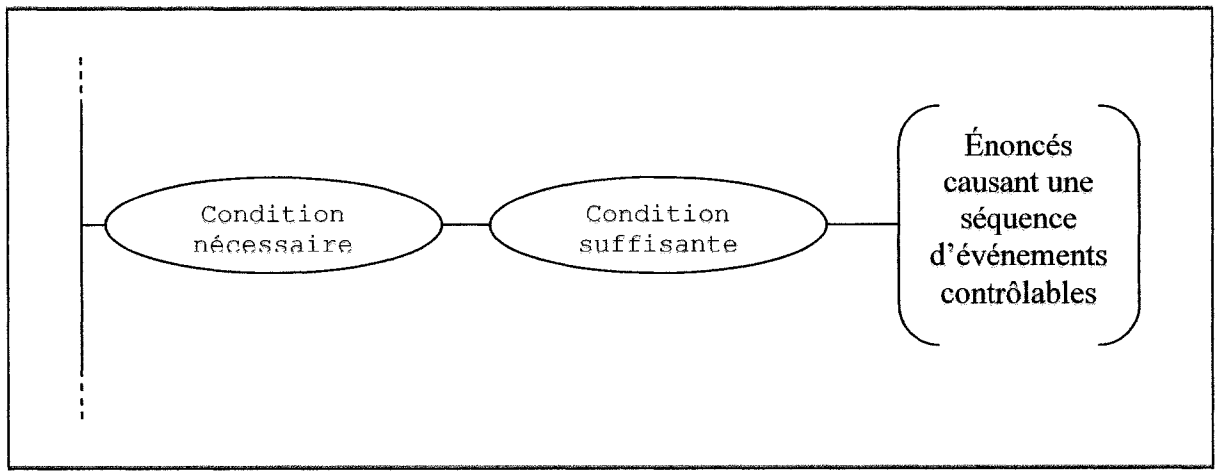

Figure 26. Schéma de génération de code général pour un SFBC

La condition suffisante doit être une formule logique dont le résultat est *vrai* lorsque le système se trouve dans un état du comportement libre où une transition sur l'événement contrôlable considéré n'est pas inhibée par une contrainte.

CHAPITRE 5
APPLICATIONS ET RÉSULTATS

Pour illustrer et, jusqu'à un certain point, tester la validité des analyses qui précèdent, ou simplement les invalider, nous avons procédé à la conception de deux applications.

Nous avons d'abord établi une spécification par grafcets et généré manuellement le code qui lui correspond à partir des schémas de génération de code obtenus de l'analyse faite à la section 3.2.

Nous avons procédé ensuite à l'élaboration d'une spécification conforme à la théorie du contrôle des systèmes à événements discrets. À partir de cette spécification, nous avons calculé le comportement contrôlable maximal du système spécifié, que nous avons tenté d'implémenter de deux façons différentes en générant le code manuellement à partir de schémas de génération de code. La première implémentation, une implémentation naïve du contrôleur, a été générée en utilisant les schémas de génération de code de la section 4.2. La deuxième implémentation a nécessité une analyse complémentaire et de nouveaux schémas de génération de code.

Toutes ces implémentations utilisaient comme machine cible des automates programmables *Koyo* de *Automation direct.com*™. Les calculs pour l'obtention d'un contrôleur conforme à la théorie du contrôle des systèmes à événements discrets ont été réalisés à l'aide de l'outil SUCSEDES conçu au Département d'informatique de l'Université de Sherbrooke [SDR99].

De façon générale pour ces applications, le code *Ladder* généré a dû être programmé en utilisant le logiciel *DirectSOFT*™. Puisqu'il s'agit d'une notation graphique difficile à mettre en page et atteignant parfois une taille considérable, nous avons décidé de l'archiver sur un disque compact accompagnant ce mémoire. Le disque compact comprend aussi les résultats intermédiaires détaillés et les spécifications d'AFD utilisés pour le travail avec l'outil

SUCSEDES. Pour les références au code qui suivent, nous ne mentionnons que le nom des fichiers qui se trouvent sur le disque compact.

5.1 Implémentation par GRAFCET

La spécification par grafcets a été élaborée conformément aux spécifications de la compagnie *Festo* fournies dans les manuels [FES98], [FES98a], [FES98b], [FES99] et [FES99a]. Elle couvre toute l'usine-école (les trois cellules de traitement) et les aspects transport et traitement de notre cadre méthodologique. Cette spécification figure en annexe 2.

Le code généré se trouve sur le disque compact accompagnant ce mémoire dans trois projets *DirectSOFT* [AUT99].

- Le fichier `Distribution1.prj` contient le code pour la cellule `C1`, soit la station de livraison et la station de test.

- Le fichier `Usinage1.prj` contient le code pour la cellule `C2`, soit la station d'usinage et la station de manutention.

- Le fichier `Triage1.prj` contient le code pour la cellule `C3`, soit la station de tri et la grue de réinsertion.

Ce code a été généré conformément aux schémas de génération de code de la section 3.2 et fonctionne de façon satisfaisante.

Nous n'avons pas traité le problème de la modélisation des pièces (la matière première) dans cette spécification, parce que, faute de communication entre les automates programmables, il était impossible de transmettre les caractéristiques d'une pièce d'un automate à l'autre. Ainsi ce problème de modélisation des pièces n'est traité que localement de cellule en cellule, sans transmission entre cellules, de sorte que la dernière cellule (qui correspond à la station de tri et la grue de réinsertion) ne dispose pour le tri, que d'un critère synthétique artificiel alors que le tri devrait être basé sur les caractéristiques des pièces une fois traitées.

Dans le code généré, il existe principalement deux déviations des schémas de génération de la section 3.2.

1. Des étapes de démarrage sont ajoutées (cellule `C1`, stations de livraison et de test) pour vérifier que les conditions initiales du système sont réalisées avant de démarrer. Celles-ci ne sont pas absolument nécessaires et auraient pu être données en spécification. Ces étapes ont été introduites pour faciliter la mise au point de la spécification.

2. Un moniteur est ajouté dans la cellule `C3` (station de tri et grue de réinsertion). Il s'agit d'un genre de grafcet hiérarchique (comme permis dans la norme) qui utilise le « forçage » et le « figeage » pour gérer l'arrêt d'urgence. Encore une fois, l'usage de ce moniteur n'était pas nécessaire au fonctionnement, mais très utile pendant la mise au point puisque l'utilisation de la grue de réinsertion présentait de réels dangers à l'opération.

5.2 Théorie du contrôle des systèmes à événements discrets

Pour l'implémentation d'un contrôleur issu de la théorie du contrôle des SED (systèmes à événements discrets), il est nécessaire de réduire le problème de l'explosion combinatoire sous peine de ne pas pouvoir illustrer adéquatement la démarche d'implémentation. D'ailleurs, l'élaboration d'une spécification du système et surtout des contraintes qui s'y appliquent est un art qui demeure difficile et requiert souvent l'inspection de l'espace d'états du résultat (le contrôleur de composition synchrone). Sans un outil informatisé pour effectuer cette inspection, il devient vite difficile de diagnostiquer (par inspection manuelle de l'espace d'états du contrôleur) si les contraintes spécifiées remplissent le but visé. Cependant, si l'on restreint trop sévèrement le problème, on augmente le risque d'obtenir une spécification triviale qui n'aurait de valeur que l'illustration.

En fonctionnant sous l'hypothèse qu'un contrôleur pour un système démontrant peu de parallélisme devrait présenter un espace d'états restreint, nous avons décidé de nous concentrer sur un sous-problème de l'usine-école MPS, la station de livraison dont on peut voir l'aspect physique à la figure 41 de l'annexe 1.

Dans la station de livraison, il n'y a qu'un problème à résoudre, celui du transport d'une pièce à partir du barillet du silo de chargement, en passant par le dock de chargement (par une extension de l'injecteur), et aboutissant à la plate-forme de test (par transbordement à l'aide de la grue). Le processus admet peu de parallélisme. D'une part une nouvelle pièce peut apparaître au silo à n'importe quel moment, d'autre part l'injecteur peut entrer en fonction pendant que la grue dépose une pièce sur la plate-forme de test. Toutes les autres activités se déroulent de façon séquentielle. Le système est simple, mais il n'est pas trivial. Il comporte quatre dispositifs et plusieurs contraintes à modéliser. Il procure une bonne illustration des problèmes de modélisation et de génération de code.

5.2.1 Spécifications des comportements et calcul du contrôleur

Le calcul du contrôleur requiert que l'on élabore deux spécifications sous forme d'automates à états finis (AFD) :

1. la spécification du comportement libre du système (comportement du système lorsqu'il n'est soumis à aucune contrainte) ;

2. la spécification des contraintes, sous lesquelles le système doit fonctionner, qui sont de deux types :

    - les contraintes de vivacité qui expriment ce que le système doit faire, c'est-à-dire l'ensemble des sous-tâches que le système doit accomplir ainsi que leur séquence relative ;

- les contraintes de sécurité qui expriment les situations qui doivent être évitées.

Pour élaborer la spécification du comportement libre du système, nous procédons de façon modulaire. Nous avons donc besoin d'un modèle de chacun des quatre dispositifs impliqués.

1. Le barillet à la base du silo de la station de livraison (`C1.Livraison.Silo` en annexe 1) qui est muni d'un capteur (`X6`) détectant la présence d'une pièce.

2. L'injecteur couplé au barillet (`C1.Livraison.Injecteur`) constitué d'un vérin à action monostable.

3. La flèche de la grue de transbordement (`C1.Livraison.Grue.Flèche`) dont l'action est similaire à un vérin à action bistable piloté.

4. La pince de la grue de transbordement, une ventouse pneumatique (`C1.Livraison.Grue.Pince`).

Puisque le contexte est clair, nous désignons chacun de ces dispositifs par les noms suivants, avec la majuscule pour les distinguer dans le texte : Barillet, Injecteur, Flèche et Pince.

Pour élaborer la spécification du comportement libre tout en limitant l'explosion combinatoire, nous avons dû recourir à des hypothèses sur le fonctionnement des dispositifs sous l'effet de leurs contraintes de vivacité locales.

La figure 27 donne le comportement libre de la Flèche avant qu'on ne lui impose des contraintes de vivacité locales. Ce comportement est instable et comporte notamment des transitoires sur les événements incontrôlables ↑`X2`, ↓`X2`, ↑`X3` et ↓`X3` qui augmentent le nombre de transitions et ultimement le nombre d'états du comportement libre.

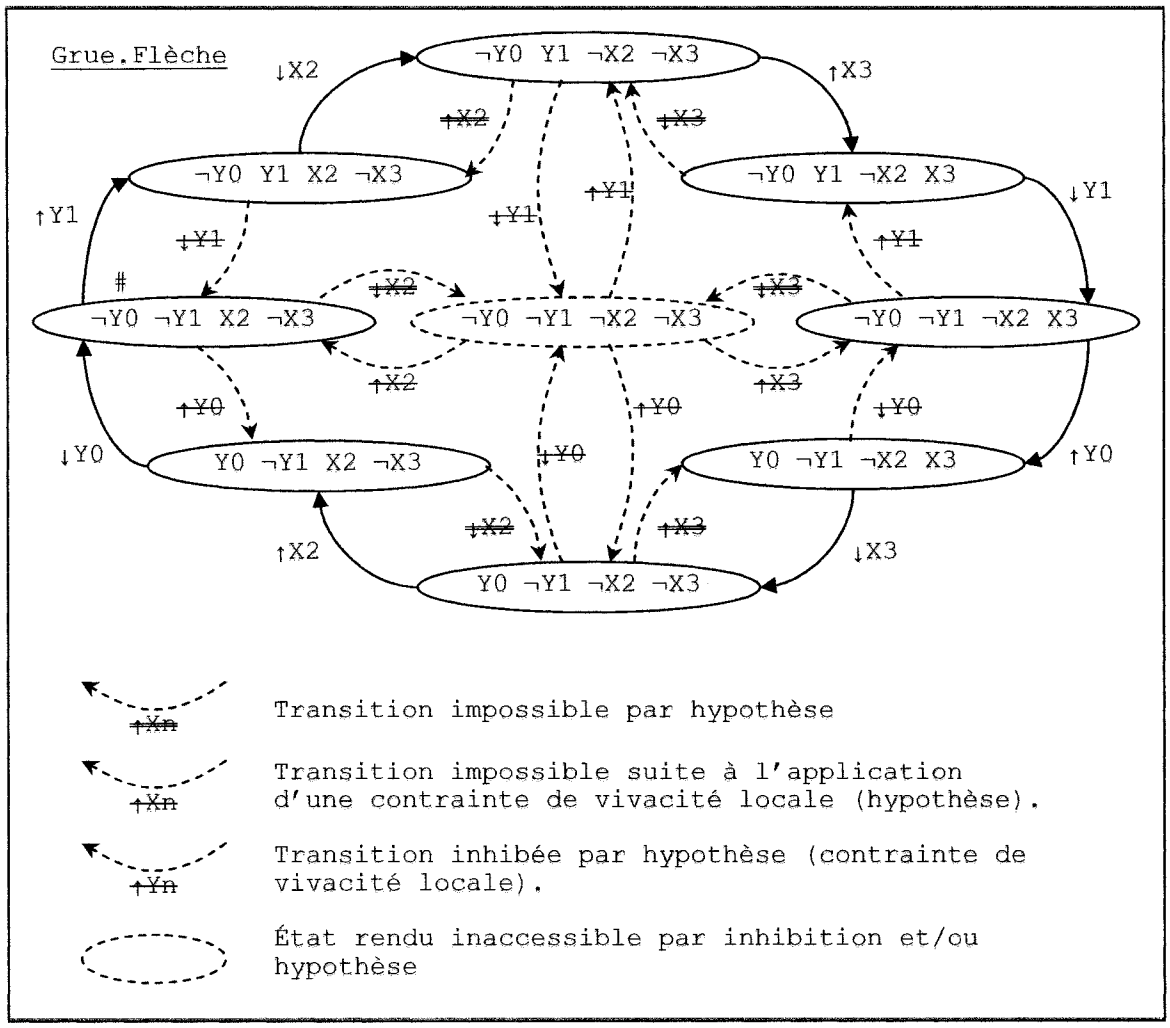

Figure 27. Comportement libre de la Flèche

Le comportement modélisé est cependant assez fidèle à la réalité si on admet la possibilité que des chaînes de transitions comme $(\uparrow Y1 \downarrow Y1)^* \uparrow Y1$ soient possibles, c'est-à-dire si on admet qu'on puisse activer et désactiver l'ordre de positionnement de la Flèche vers la droite un nombre arbitraire de fois avant de finalement le laisser stable. L'inertie du système mécanique ainsi contrôlé rend alors possible l'apparition de transitoires dans les événements incontrôlables. Cependant, ce comportement ne correspond pas à une commande cohérente de la Flèche.

On observe donc que dans des conditions normales de commande, un ordre donné à un dispositif a pour but un résultat (ici le déplacement de la Flèche de sa position initiale au dock de chargement vers sa position stable à la plate-forme de test) et que la commande devrait être maintenue jusqu'à l'obtention de ce résultat. En imposant une contrainte de vivacité locale qui interdit un comportement erratique de la commande, on stabilise le comportement libre du dispositif et, ultimement, on élimine les transitoires des événements incontrôlables, ce qui simplifie considérablement l'AFD du comportement libre. On doit cependant faire deux hypothèses.

1. Une commande ne sera donnée qu'à partir d'un état « stable » (c'est-à-dire un état à partir duquel toute autre évolution dépend d'un changement dans la commande) et sera maintenue jusqu'à l'atteinte d'un autre état stable (le résultat anticipé de la commande).

2. Dans le régime de fonctionnement qui résulte de la première hypothèse, les conditions physiques d'opération du système commandé seront telles qu'on ne connaîtra pas de transitoires dans les changements d'état des capteurs (en pointillé dans la figure 27).

La figure 28 montre le résultat de l'application de ces deux hypothèses sur le comportement libre de la Flèche. L'hypothèse n° 1 est raisonnable et l'hypothèse n° 2 s'est révélée correcte après expérimentation. Il faut noter que l'hypothèse n° 1 semble généralement applicable, mais il est risqué d'admettre l'hypothèse n° 2 puisqu'elle pourrait équivaloir à l'inhibition d'événements incontrôlables et rendrait ainsi le modèle du comportement libre incorrect. Pour nos besoins, nous admettons les deux hypothèses puisque l'expérimentation semble nous donner raison.

L'hypothèse n° 1 correspond à l'application d'une contrainte de vivacité locale interdisant toutes les transitions sur $\uparrow$Y1, $\downarrow$Y1, $\uparrow$Y0 et $\downarrow$Y0 indiquées en traits pointillés dans la figure 27.

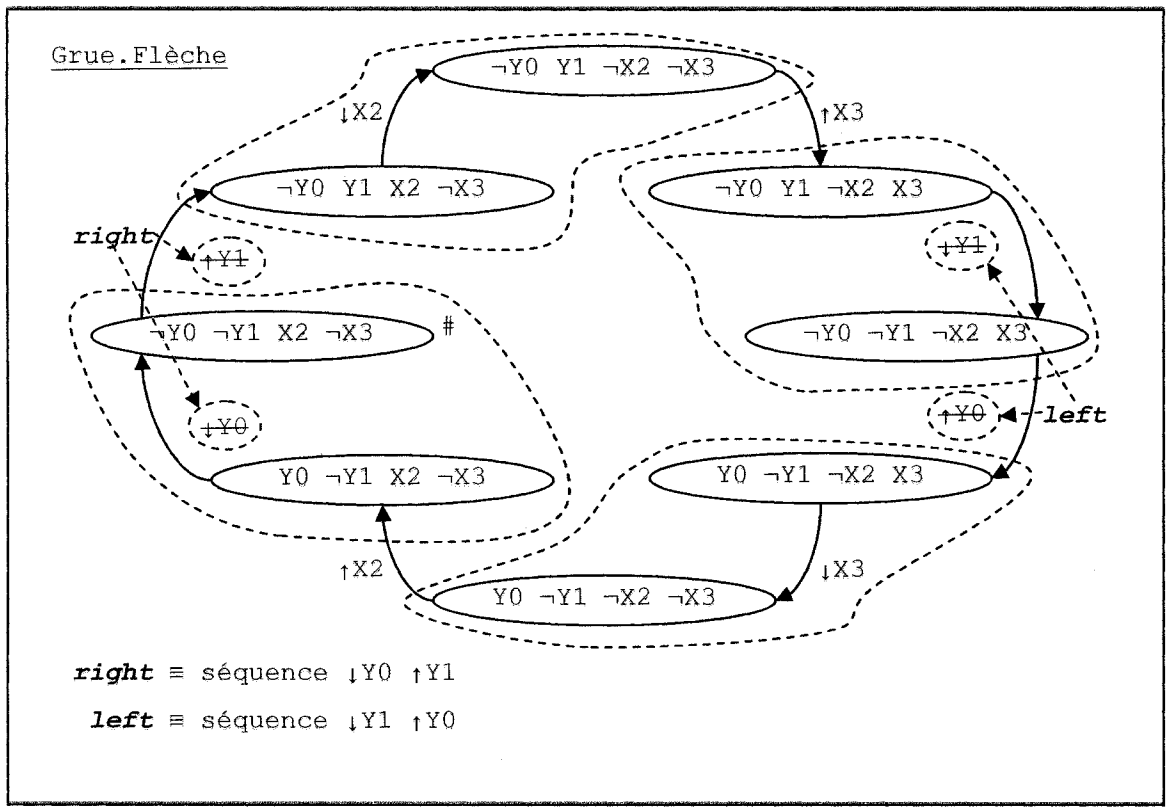

Figure 28. AFD simplifié de la Flèche sous contrainte de vivacité locale

L'hypothèse n° 2 nous permet d'éliminer les transitions en traits pointillés sur les événements incontrôlables dans la même figure. On élimine ainsi un état qui devient inaccessible et on simplifie considérablement l'AFD.

La figure 28 illustre une autre des hypothèses simplificatrices que nous avons faite pour réduire l'espace d'états dans le cas d'une commande bistable (Flèche et Pince de la grue). Dans ce diagramme on voit que l'ordre de déplacer la Flèche vers la droite est constitué de deux événements contrôlables, ↓Y0 et ↑Y1, que l'on peut assumer être donnés en séquence sans aucun autre événement intervenant entre les deux.

L'ordre inverse peut être traité de façon similaire. On obtient alors le modèle de comportement libre de la Flèche de la figure 29.

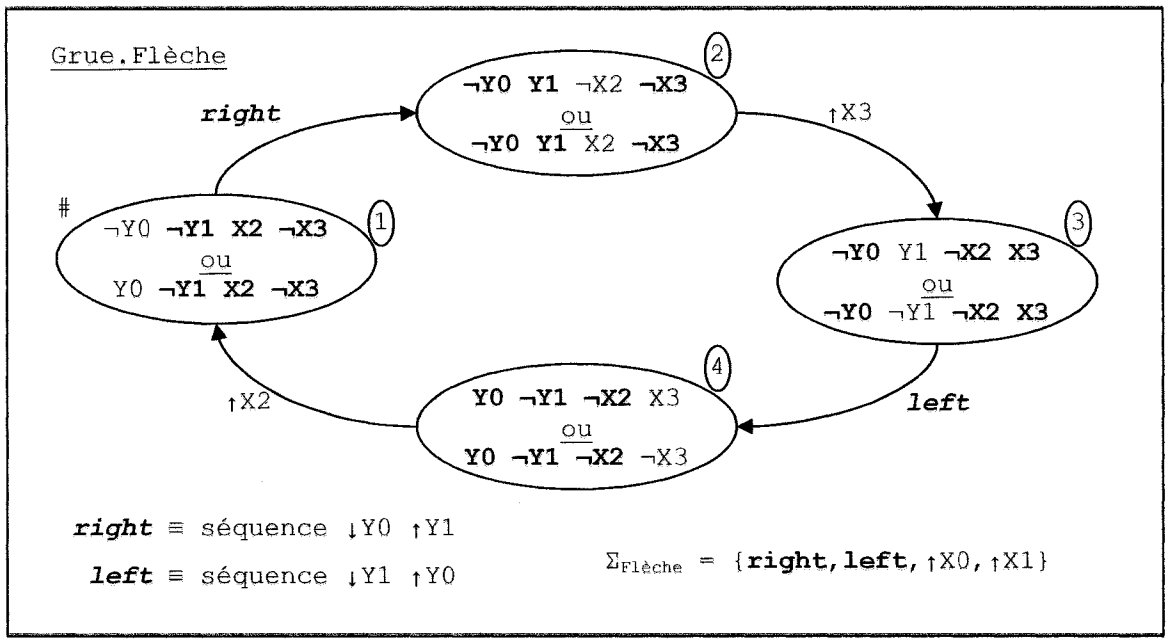

Figure 29. AFD du comportement libre de la Flèche après simplification

En utilisant des manipulations similaires pour les AFD de comportement libre (ou pré-contraint localement pour la vivacité) de la Pince et de l'Injecteur, on obtient les spécifications de comportement libre des autres dispositifs du système présentées à la figure 30. Les modèles de comportement libre de l'Injecteur et de la Flèche (qui ont été pré-contraints localement pour la vivacité) les font osciller indéfiniment entre deux positions dites stables. La Pince, prise seule, connaît une impasse si aucune pièce ne se trouve sous la ventouse pneumatique, mais ce comportement libre assume les conditions d'opération de l'ensemble des dispositifs de façon complémentaire, nous le considérons donc adéquat.

Il faut noter que dans une situation réelle, il serait probablement hasardeux de faire autant d'hypothèses de fonctionnement. Certaines de ces hypothèses sont raisonnables, d'autres se sont révélées adéquates expérimentalement. Toutefois, le comportement libre de la Pince laisse à désirer. Ceci pose assurément la question de la qualité de la spécification du comportement libre. Plus la spécification du comportement libre est fidèle à la réalité, moins il y a de possibilité d'erreur dans le contrôleur calculé.

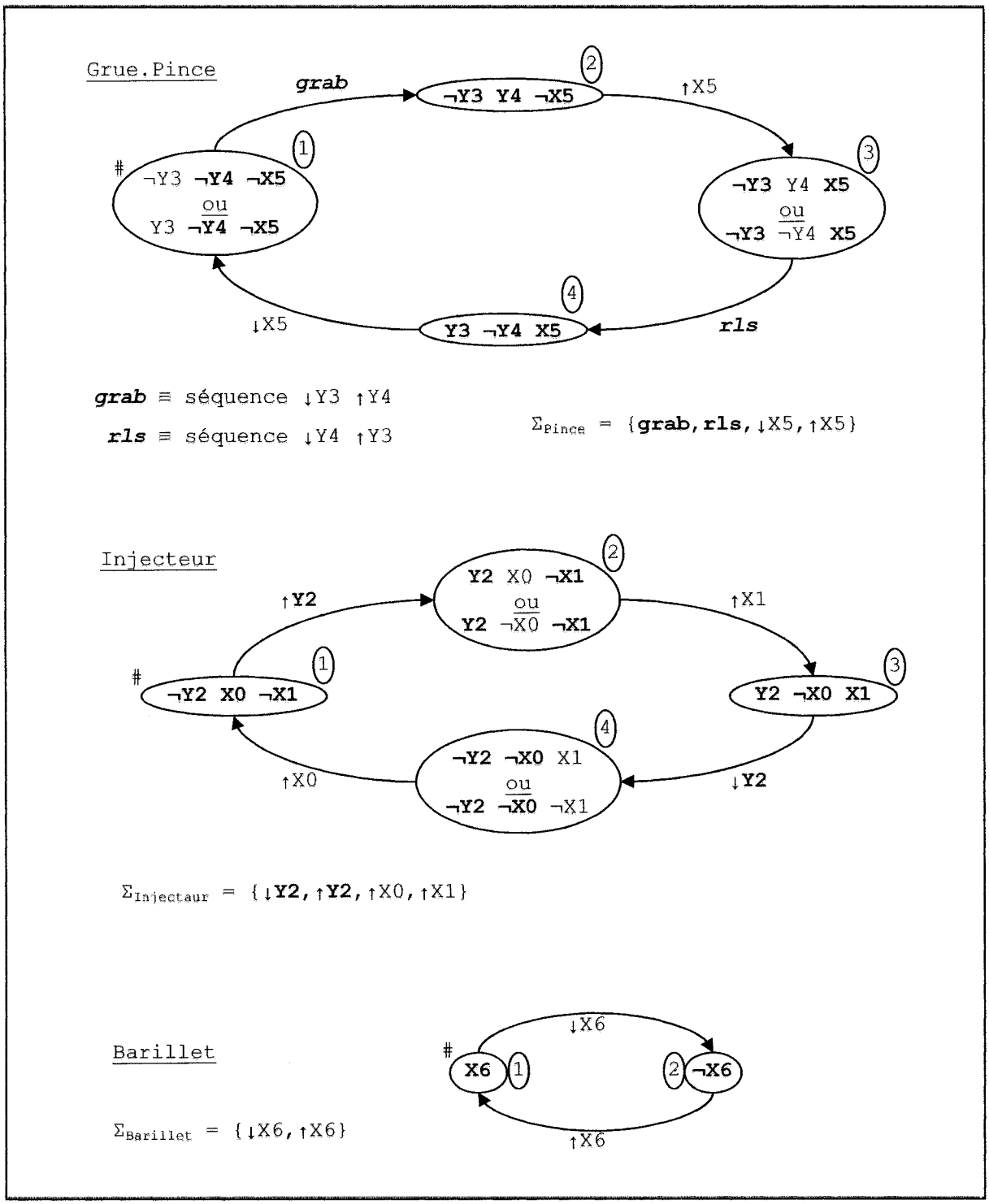

Figure 30. Comportement libre de la Pince de l'Injecteur et du Barillet

Ici nous avons dû recourir à des hypothèses très sévères pour pouvoir limiter l'explosion combinatoire afin de nous permettre de disposer d'une méthode systématique (bien qu'heuristique) pour spécifier les contraintes du système. Nous avons constaté qu'il était souvent plus facile d'exprimer les contraintes sous forme de contraintes exercées par un dispositif (ou un ensemble de dispositifs) sur un autre. Par exemple la figure 31 illustre, sous forme d'AFD les contraintes que le Barillet impose sur l'injecteur ($C_{Barillet-Injecteur}$), celles que la Flèche exerce sur la Pince ($C_{Flèche-Pince}$) et celles que la Pince exerce sur la Flèche ($C_{Pince-Flèche}$), qui sont les seules contraintes entre des paires de dispositifs simples.

La figure 32 illustre les contraintes que la Grue (c'est-à-dire l'ensemble formé de la Flèche et de la Pince) exerce sur l'ensemble Barillet-Injecteur ($C_{Grue-Bar+Inj}$). Finalement, la figure 33 contient les contraintes que l'ensemble Barillet-Injecteur exerce sur la Pince ($C_{Bar+Inj-Pince}$) et sur la Flèche ($C_{Bar+Inj-Flèche}$). Pour ces dernières, il aurait d'ailleurs été préférable de les exprimer par un seul AFD construit à partir du comportement libre de l'ensemble Barillet-Injecteur et qui aurait combiné toutes les contraintes sur la Grue. Nous aurions ainsi obtenu moins d'états au total. En tout, nous obtenons six AFD de contrainte pour le système dans son ensemble.

L'AFD de contrainte $C_{Barillet-Injecteur}$ contraint l'Injecteur à n'injecter que lorsqu'une pièce est disponible dans le barillet.

L'AFD de contrainte $C_{Pince-Flèche}$ exprime que la Flèche ne peut entrer en opération tant que la Pince n'a pas fini l'opération qu'elle est en train d'effectuer (le cas échéant), et qu'une fois une pièce saisie, la Flèche doit nécessairement aller la déposer à la plate-forme de test.

L'AFD $C_{Flèche-Pince}$ exprime trois contraintes.

1. Que la Pince ne peut changer d'état pendant que la Flèche est en opération (en mouvement vers le dock de chargement ou vers la plate-forme de test).

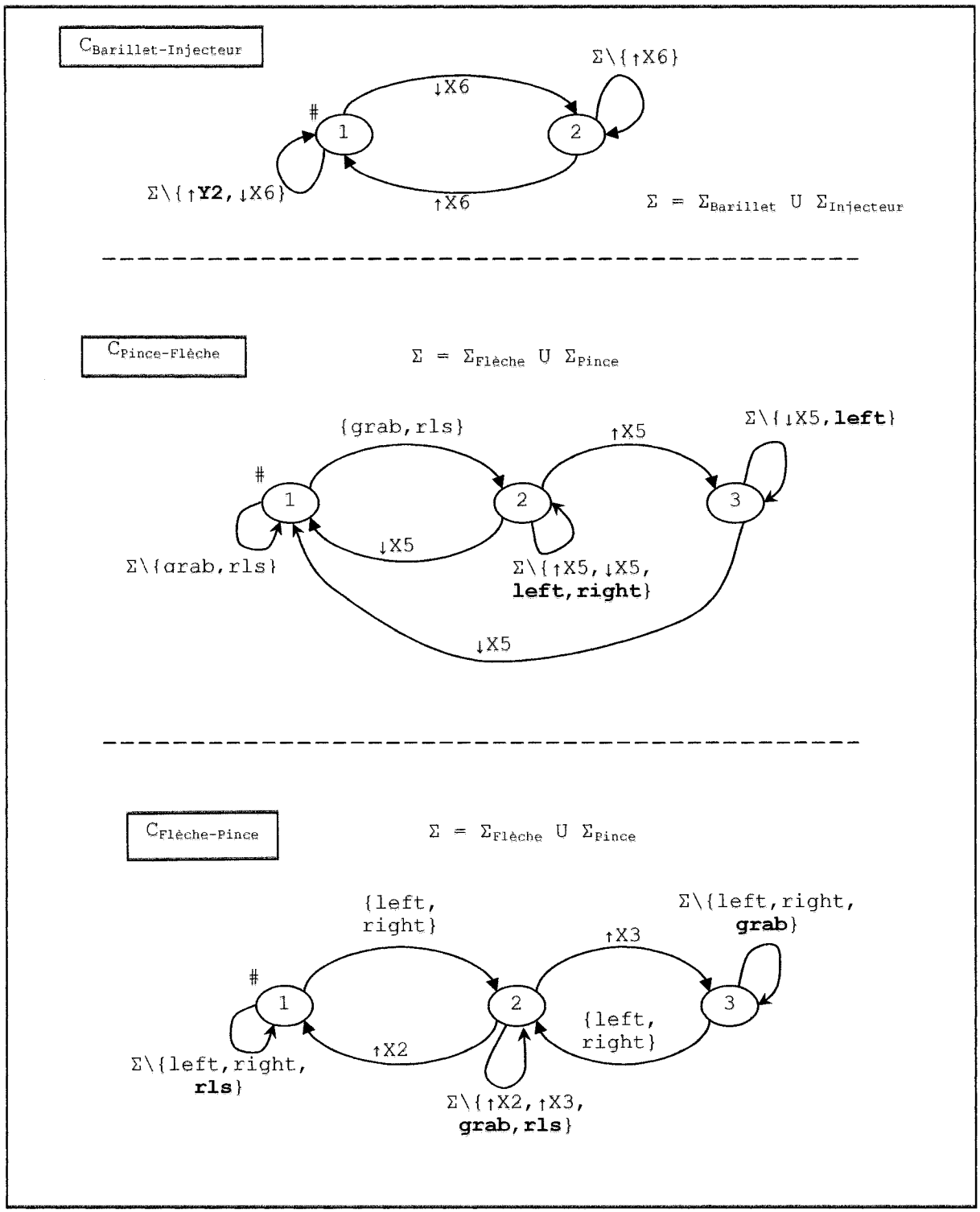

Figure 31. Contraintes entre le Barillet et l'Injecteur, et entre la Flèche et la Pince

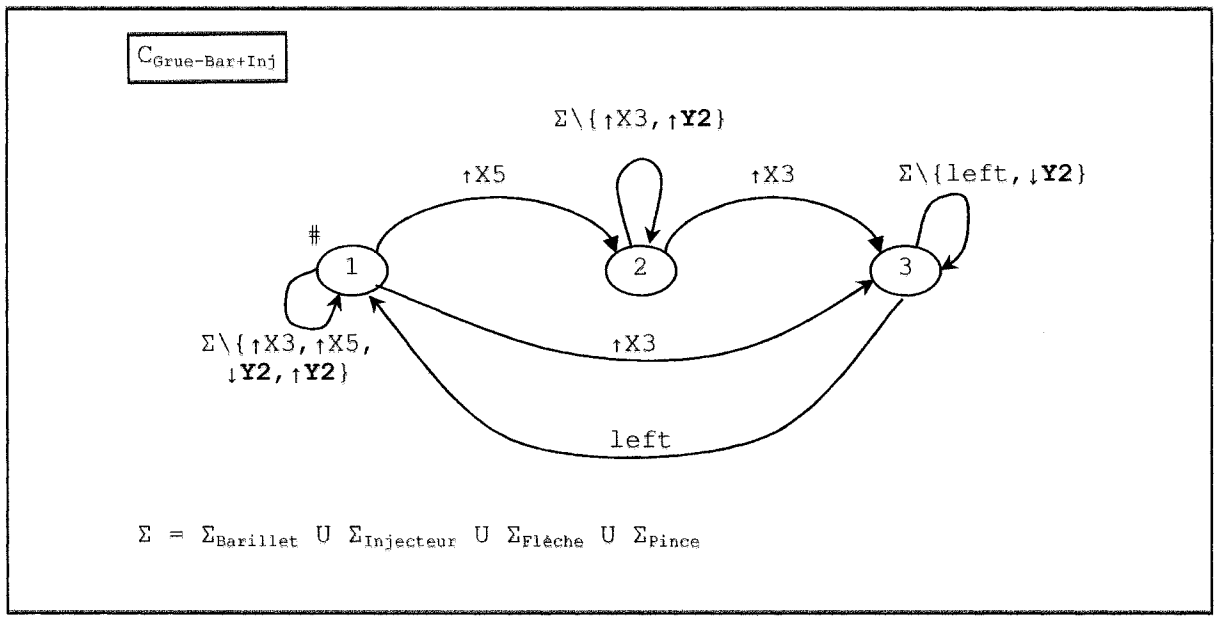

Figure 32. Contraintes de la Grue sur l'ensemble Barillet-Injecteur

2. Que lorsque le Flèche est au dock de chargement, la Pince ne peut que saisir une pièce.

3. Que lorsque la Flèche est en position à la plate-forme de test, la Pince ne peut que déposer une pièce sur la plate-forme.

L'AFD $C_{Grue-Bar+Inj}$ exprime deux contraintes.

1. Que l'Injecteur ne peut pas changer d'état pendant que la Grue est au dock de chargement à moins qu'une pièce ait été saisie.

2. Qu'une fois à la plate-forme de test, l'injection est permise, mais qu'une fois injecté, l'Injecteur doit attendre que la pièce injectée soit saisie.

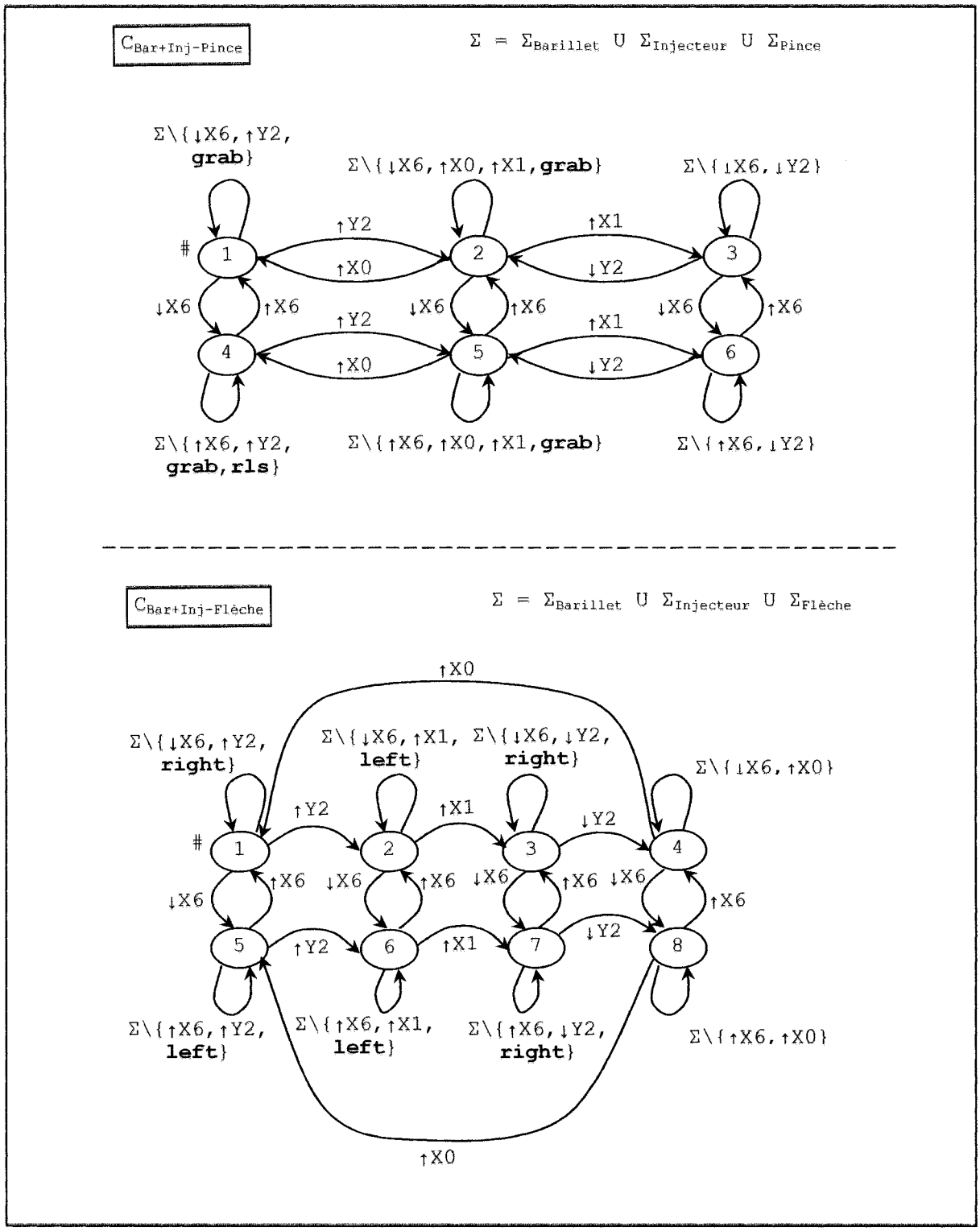

Figure 33. Contraintes de l'ensemble Barillet-Injecteur sur la Flèche et sur la Pince

L'AFD de contrainte $C_{Bar+Inj-Pince}$ prévient que la Pince entre en fonction lorsque le dock de chargement est vide (l'Injecteur n'est pas en position d'injection) et empêche la Pince de relâcher une pièce qu'elle vient juste de saisir au dock de chargement.

Finalement l'AFD de contrainte $C_{Bar+Inj-Flèche}$ empêche la Flèche de se diriger vers le dock de chargement (*left*) lorsqu'une injection est en cours et empêche la flèche de se diriger vers la plate-forme de test (*right*) tant qu'on est pas certain que l'Injecteur ne tient plus la pièce en position injectée au dock de chargement. Cet AFD sert aussi à retenir la Flèche au dock de chargement lorsque celui-ci est vide et qu'il n'y a pas de pièce dans le barillet empêchant ainsi la Flèche d'osciller (état n° 1 de l'AFD). Il sert aussi à donner préséance à l'injection (état n° 5 de l'AFD) lorsque simultanément, l'Injecteur est prêt à injecter une nouvelle pièce et la Flèche est prête à revenir au dock de chargement (après avoir déposée une pièce à la plate-forme de test). Cette dernière contrainte n'a été ajoutée qu'à posteriori pour résoudre un problème de séquencement.

Le calcul du contrôleur suit le schéma habituel dans le cas d'une spécification modulaire comme celle-ci.

1. On obtient le comportement libre du système par produit synchrone des AFD de comportement libre :

   $$\text{Livraison} \leftarrow \text{Injecteur} \parallel \text{Barillet} \parallel \text{Flèche} \parallel \text{Pince}$$

2. On obtient l'AFD des contraintes à appliquer :

   $$\text{Contraintes} \leftarrow C_{Barillet-Injecteur} \parallel C_{Pince-Flèche} \parallel C_{Flèche-Pince} \parallel C_{Grue-Bar+Inj} \parallel C_{Bar+Inj-Pince} \parallel C_{Bar+Inj-Flèche}$$

Finalement, en utilisant l'option trois (3) de l'outil SUCSEDES (qui calcule le contrôleur par l'application de l'algorithme de Ramadge et Wonham [RM89]) avec comme paramètres les deux spécifications préalablement élaborées, on obtient deux résultats :

1. l'AFD du contrôleur (contrôleur de composition synchrone),

2. une fonction d'inhibition, qui donne pour chaque état de l'AFD du contrôleur l'ensemble des événements contrôlables inhibés par rapport au comportement libre du système.

Ces deux résultats se trouvent en annexe 3. Nous avons pris soin d'obtenir un contrôleur non ambigu, c'est-à-dire qu'il n'y a au plus qu'une transition sur un événement contrôlable quelque soit l'état du contrôleur obtenu.

5.2.2 Implémentation naïve du contrôleur

À partir du contrôleur obtenu à la section 5.2.1, nous avons effectué une implémentation manuelle naïve (c'est-à-dire directe) de la machine d'état du contrôleur (contrôleur de composition synchrone) conformément aux schémas de génération de code de la section 4.2. Le code généré se trouve sur le disque compact accompagnant le mémoire dans un projet *DirectSOFT* (`LivraisonRW.prj`).

Comme le contrôleur calculé ne comporte au plus qu'une transition sur un événement contrôlable quelque soit l'état de l'AFD (non ambiguïté), le problème d'arbitrage entre transitions contrôlables ne se présente pas. Cependant, la plupart des états du contrôleur possèdent plus d'une transition, il a donc été nécessaire d'arbitrer entre ces transitions à l'aide des politiques suivantes.

1. On privilégie toujours les transitions sur un événement incontrôlable par rapport aux transitions sur un événement contrôlable.

2. Lorsqu'il y a plusieurs transitions sur un événement incontrôlable, l'ordre dans lequel elles sont effectuées est arbitraire. Pour notre solution, cet ordre correspond à l'ordre d'apparition dans la liste de l'AFD.

La figure 34 montre le schéma de génération de code modifié que nous avons utilisé pour cette implémentation. La liste des énoncés *Ladder* suit la priorité (arbitraire) des événements, la transition sur un événement contrôlable (s'il y en a une) se situe à la fin. Les énoncés *Ladder* correspondant à une transition sont (et doivent être) mutuellement exclusifs.

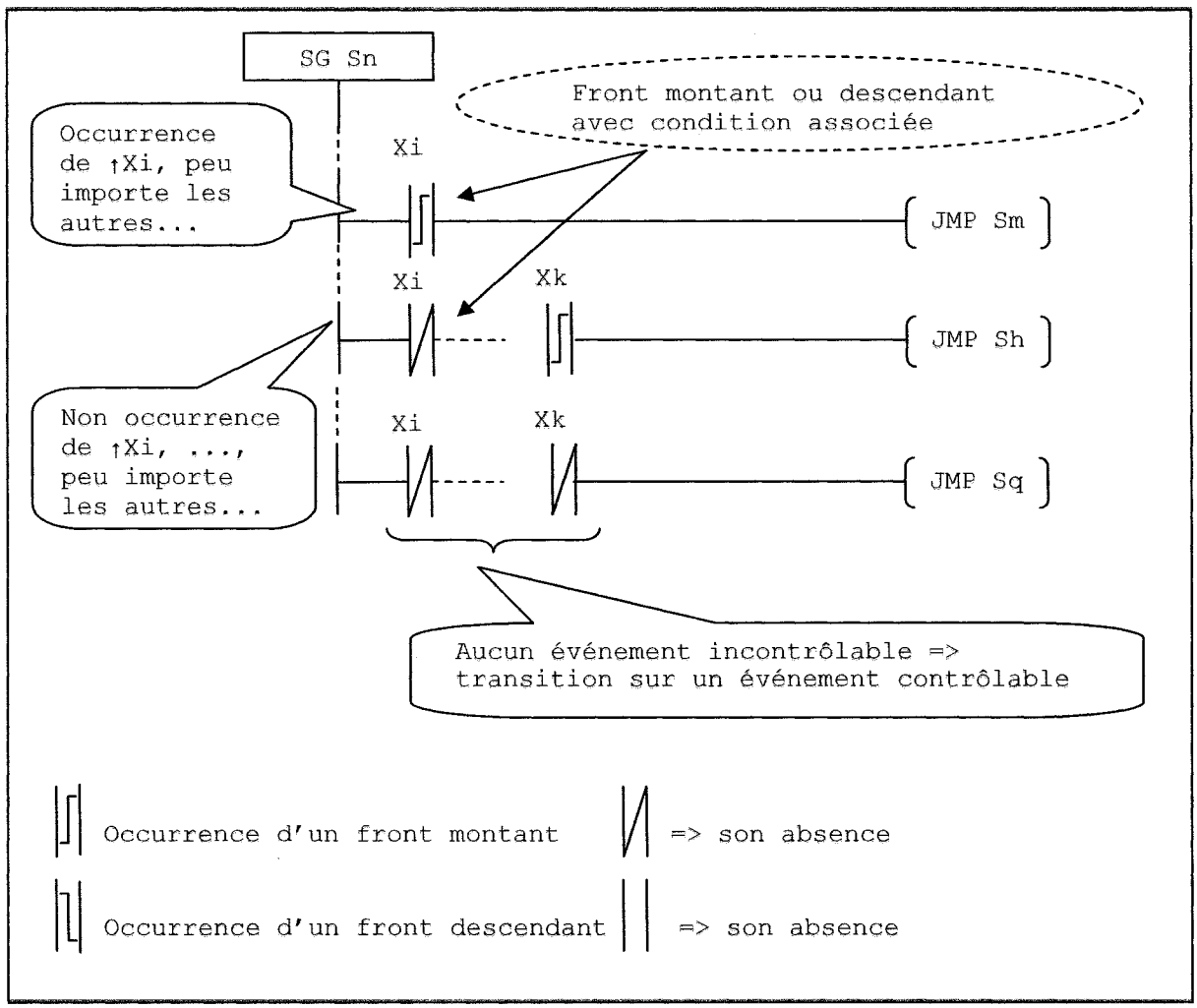

Figure 34. Schéma de génération de code pour arbitrage de transitions

Il est bien évident que le nombre d'énoncés *Ladder* pour les transitions croît de façon proportionnelle à la somme des cœfficients du binôme, ce qui est catastrophique. De plus si le nombre d'états croît lui aussi très rapidement et qu'il y a une possibilité réelle de perdre des événements incontrôlables (comme c'est le cas sur les automates *Koyo*), l'AFD se désynchronise et ne reflète plus correctement l'état du contrôleur. On doit donc en conclure qu'une implémentation directe est irréaliste sur un automate programmable.

Malgré tout ce que nous venons de voir, nous avons été surpris de constater que l'implémentation naïve réalisée fonctionne correctement. Ceci est probablement dû à la petite taille de l'application et à la nature très séquentielle du système. Cette stratégie d'implémentation n'est cependant pas fiable pour les raisons énumérées précédemment.

Nous pouvons nous demander si l'implémentation directe sur un ordinateur plus puissant et plus versatile serait viable.

Il est possible remédier à tous les problèmes découverts ici avec un ordinateur plus versatile qu'un PLC. Il s'agit d'avoir une file d'attente d'événements incontrôlables que l'on peut distribuer itérativement à un algorithme simulant un AFD, jusqu'à ce que l'AFD ait « consommé » tous les événements incontrôlables. L'algorithme entre ensuite dans un mode d'itération spécial sans événement à « consommer » et qui permet de n'effectuer que des transitions sur un événement contrôlable (donc de causer ces événements, ce qui constitue la commande) jusqu'à ce que l'AFD ne puisse plus progresser. Si le modèle du comportement libre est fidèle à la réalité, l'AFD du contrôleur ne devrait pas bloquer ni se désynchroniser. L'AFD n'est alors qu'une simple table de transitions. Le problème de l'explosion combinatoire subsiste, mais ce problème est inhérent à la théorie.

5.2.3 Raisonnements pour une implémentation plus efficace du contrôleur

Puisque la stratégie d'implémentation directe de l'AFD du contrôleur ne semble pas réaliste pour un ordinateur cible comme un automate programmable, il y a lieu de se demander s'il est possible d'adopter une autre stratégie.

Notons d'abord que, pour le contrôleur de la section 5.2.1, il n'est pas nécessaire de tenir compte de l'état de l'AFD de contrainte. En examinant de près l'AFD du contrôleur (en annexe 3), on peut constater que chacun de ses états est caractérisé par un état distinct de l'AFD du comportement libre. Ainsi, le calcul de l'état de l'AFD du comportement libre suffit pour évaluer f_σ selon la première définition. On peut donc appliquer la stratégie d'implémentation exposée à la section 4.4 pour le cas d'un SFBC. Il nous reste à trouver une méthode systématique pour formuler les deux conditions de chaque énoncé *Ladder* à générer.

Considérons quelques propriétés de notre spécification du comportement libre du système.

Le comportement libre du système est obtenu par produit synchrone des AFD du comportement libre des composantes :

$$\text{Livraison} \leftarrow \text{Injecteur} \parallel \text{Barillet} \parallel \text{Flèche} \parallel \text{Pince}$$

Ainsi, dans l'AFD du comportement libre du système, chaque état peut être caractérisé par un élément du produit cartésien :

$$\{ 1, 2, 3, 4 \} \times \{ 1, 2 \} \times \{ 1, 2, 3, 4 \} \times \{ 1, 2, 3, 4 \}$$

Il s'agit d'un 4-tuplet ordonné (s_1, s_2, s_3, s_4) où s_1 est un état de l'AFD de l'Injecteur, s_2 un état de l'AFD du Barillet, s_3 un état de l'AFD de la Flèche et s_4 un état de l'AFD de la Pince. Nous appelons le 4-tuplet (s_1, s_2, s_3, s_4) l'état global du système et les s_i les états locaux des composantes.

De plus, chacun des AFD de comportement libre utilisé pour calculer le comportement libre du système dispose d'un vecteur d'état constitué des actionneurs et des capteurs associés au dispositif. Par exemple, la Flèche est associée aux actionneurs `Y0` et `Y1` ainsi qu'aux capteurs `X2` et `X3` (voir en annexe 1). Or pour chacun des états du comportement libre d'un dispositif, il existe une configuration de ce vecteur d'état qui le caractérise uniquement (voir à la figure 29 et à la figure 30, les éléments en caractère gras). Par exemple, la configuration `¬Y1, X2, ¬X3` caractérise uniquement l'état n° 1 de l'AFD du comportement libre de la Flèche. On dispose donc d'une formule logique pour calculer l'état local de tous les composants au cours d'un cycle de l'automate programmable. Par extension, la conjonction de ces formules logiques nous permet de déterminer l'état global du système.

Finalement, le calcul du contrôleur nous fournit une fonction de rétroaction qui nous donne les états du contrôleur où un événement contrôlable donné est permis. Il s'agit d'une fonction totale, elle couvre donc tous les états de l'AFD du contrôleur abstrait, dont la cible est l'ensemble des événements contrôlables du système sans exception. Puisque le but recherché est l'implémentation de la fonction de rétroaction à partir des états accessibles du système opérant sous contrainte (l'action exercée par l'AFD du contrôleur), cette fonction suffit.

La procédure de génération de code pour le schéma de la figure 35 est donc constituée des étapes suivantes.

1. Formuler la condition nécessaire à l'aide du modèle de comportement libre du système et des configurations des vecteurs d'état pertinents.

2. Formuler la condition suffisante à l'aide de la fonction de rétroaction fournie par SUCSEDES et des vecteurs d'états pertinents.

3. Formuler la séquence d'énoncés nécessaire à l'implémentation de la transition.

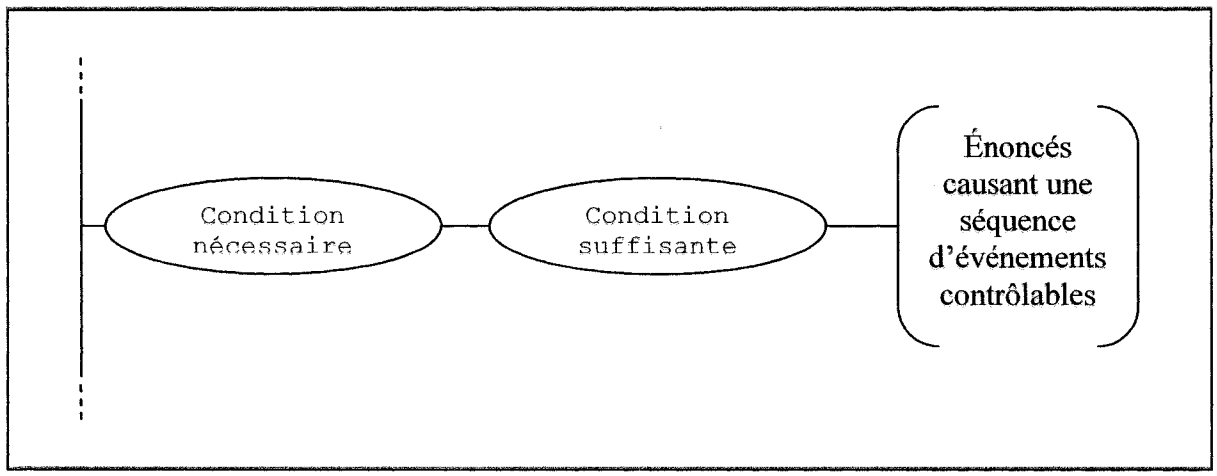

Figure 35. Schéma de génération de code pour la fonction de rétroaction

Ces trois étapes sont répétées pour chaque événement contrôlable du système et forment un programme *Ladder* de type conventionnel.

La condition nécessaire ne peut pas être extraite de la fonction de rétroaction puisque celle-ci permet un événement contrôlable même dans les états globaux où cet événement ne peut pas physiquement se produire. Il faut donc extraire cette condition de l'automate de comportement libre. Or dans le modèle de la section 5.2.1 un actionneur n'est sous le contrôle que d'un et un seul dispositif. De plus, dans ce modèle, un événement contrôlable n'est possible que dans un seul état du dispositif. Donc, la condition nécessaire de la figure 35 ne sera constituée que de la formule logique qui correspond à l'état local du dispositif où l'événement est possible. Par exemple pour l'événement *right* de la Flèche, la condition nécessaire est donnée par la conjonction $\neg Y1 \wedge X2 \wedge \neg X3$ qui correspond à l'état n° 1 de la Flèche. Si plusieurs états locaux étaient impliqués, on aurait une formule logique en forme normale disjonctive pour l'ensemble des états locaux impliqués.

La condition suffisante est extraite de la fonction de rétroaction. Il faut regrouper tous les états globaux pour lesquels la condition nécessaire est satisfaite, c'est-à-dire toutes les entrées du tableau où l'état local (ou la combinaison d'états locaux) correspond à la condition nécessaire.

Dans ce sous-ensemble d'états globaux, on ne retient que ceux pour lesquels l'événement considéré est permis. Chaque état global correspond à une conjonction de formules correspondant à leur tour aux états locaux qui le constituent. Toutes ces formules auront un facteur commun ; le facteur représentant la condition nécessaire qui est déjà assumée *vrai* et que l'on peut factoriser. La forme normale disjonctive des formules résultantes constitue la condition nécessaire.

Un exemple complet est donné à la section 5.2.4.

La séquence d'énoncés causant l'événement contrôlable considéré, se situe à l'extrême droite sur la deuxième ligne dans la figure 35. Dans notre exemple, il s'agit toujours d'un ou plusieurs énoncés *Ladder* **SET** ou **RST** puisque le contrôleur agit sur des actionneurs booléens. Les actionneurs basés sur des registres utiliseraient simplement des instructions différentes.

La sémantique d'un événement contrôlable est la même ici que dans la stratégie d'implémentation directe, à savoir que si un événement contrôlable est possible dans le comportement libre du système, et s'il n'est pas inhibé dans l'état courant, il est nécessairement causé et la transition correspondante franchie.

5.2.4 Implémentation efficace du contrôleur

Nous avons implémenté manuellement le contrôleur obtenu à la section 5.2.1 à l'aide de la stratégie et du schéma de génération de code développé à la section 5.2.3. Le code généré se trouve sur le disque compact accompagnant le mémoire dans un projet *DirectSOFT* (`LivraisonRW2.prj`), dont un extrait (pour illustration) est présenté à la figure 36. Nous illustrons ici la procédure d'implémentation en nous servant de la transition sur l'événement contrôlable *right* de la Flèche.

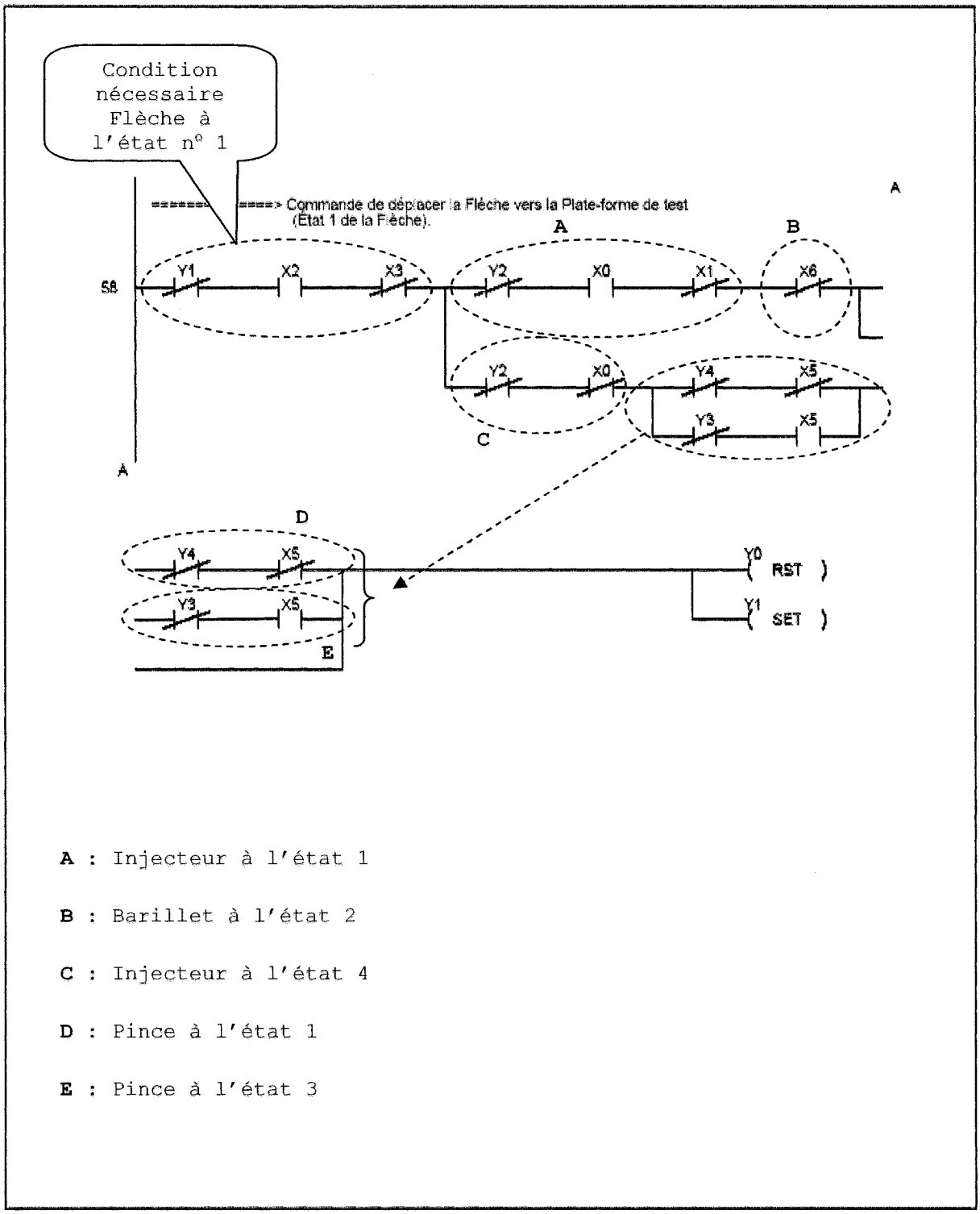

Figure 36. Implémentation efficace de la transition "*right*"

Le tableau 3 donne un extrait de la fonction d'inhibition obtenue par calcul du contrôleur (voir en annexe 3) pour les états de l'AFD du contrôleur, où l'AFD de la Flèche se trouve dans l'état n° 1 (au dock de chargement, troisième colonne du tableau). Ces états sont les seuls à considérer pour l'implémentation de la transition sur l'événement *right*, puisque ce sont les seuls à partir desquels le comportement libre admet une transition sur cet événement. Nous avons donc ici la condition nécessaire, la formule logique ¬Y1 ^ X2 ^ ¬X3 (figure 29), reflétée en figure 36.

La condition suffisante, correspondant aux entrées ombragées dans le tableau, est une disjonction de six termes d'au plus trois facteurs chaque (puisque nous avons factorisé la condition nécessaire) que nous avons simplifiés sans algorithme précis dans la figure 36. Pour vérifier l'exactitude des expressions il n'y a qu'à consulter la figure 29 et la figure 30. On peut noter que la transition étiquetée *right* est en fait constituée de la séquence ↓Y0↑Y1, matérialisée par des instructions *Ladder* **RST** pour les fronts descendants et **SET** pour les fronts ascendants.

Tableau 3. Extrait de la fonction d'inhibition du contrôleur

États du contrôleur												
Comportements				Contraintes						Inhibitions		
1	1	1	1	1	1	1	1	1	1	upY2	right	grab
1	2	1	1	2	5	1	4	1	1	upY2	grab	
1	2	1	3	2	5	3	4	1	2	upY2	rls	
1	1	1	3	1	1	3	1	1	2	upY2	right	rls
3	2	1	1	2	7	1	6	1	1	dnY2	right	
3	1	1	1	1	3	1	3	1	1	dnY2	right	
3	2	1	2	2	7	2	6	1	1	dnY2	right	
3	1	1	2	1	3	2	3	1	1	dnY2	right	
3	2	1	3	2	7	3	6	1	2	right	rls	
3	1	1	3	1	3	3	3	1	2	right	rls	
4	2	1	1	2	8	1	5	1	1	grab		
4	1	1	1	1	4	1	2	1	1	grab		
4	2	1	3	2	8	3	5	1	2	rls		
4	1	1	3	1	4	3	2	1	2	rls		

L'implémentation complète est constituée de six expressions *Ladder*, une pour chaque événement contrôlable du système.

Ces expressions sont obtenues de la même façon que nous venons de décrire et sont structurées en un programme *Ladder* de forme conventionnelle (sans utiliser la programmation par étape).

Notre implémentation utilise la programmation *Ladder* par étape mais, pendant le fonctionnement du contrôleur, il n'y a que l'étape « `S2` » qui soit active (celle contenant les expressions de transition). Les autres étapes n'ont été introduites que pour bien contrôler les configurations initiales des vecteurs d'état et gérer l'arrêt d'urgence.

La taille du programme est de 137 instructions sur un automate programmable *Koyo*, ce qui est comparable à ce que l'on peut obtenir avec un grafcet pour la même fonctionnalité.

L'implémentation obtenue fonctionne correctement. En plus tous les problèmes dont souffrait l'implémentation naïve du contrôleur ont été réglés. Il n'y a plus de risque de perte d'événements incontrôlables, aucun besoin d'arbitrer sur des transitions multiples à partir d'un même état.

L'implémentation résiste même à un contrôleur ambigu (dans sa forme actuelle). Dans ce cas elle implémente une politique de sélection arbitraire par défaut (basée sur la position des énoncés dans le texte du programme) pour sélectionner l'un des deux (ou plusieurs) événements contrôlables à causer. Cette politique pourrait ne pas être adéquate, mais il n'y a pas de contrainte qui soit violée. Pour que cette dernière caractéristique demeure vraie cependant, l'évaluation de chaque expression doit être basée sur l'état actuel de la machine abstraite. Cette condition exige que si une transition est déclenchée, puisqu'elle change l'état de la machine abstraite, on recalcule les éléments qui dépendent des actionneurs qui viennent de changer.

L'implémentation proposée ici recalcule tout l'état abstrait pertinent dans chaque expression *Ladder* de rétroaction à partir des éléments des vecteurs d'état eux-mêmes, donc elle suit implicitement toute évolution de la machine abstraite pendant le calcul de la rétroaction.

Les caractéristiques d'une telle stratégie d'implémentation sont attrayantes.

1. Le code généré est compact.

2. Éventuellement sa performance peut être mesurée et aussi améliorée par optimisation.

3. Les formules logiques sont naturellement exprimées en forme disjonctive normale, et toutes les techniques de simplification connues peuvent éventuellement être appliquées.

4. Même une implémentation sur ordinateur externe s'en trouverait très simplifiée.

L'expression des conditions suffisantes pourrait éventuellement devenir assez volumineuse. Mais il faut considérer que plus une disjonction a de termes, plus il y a de possibilités de factorisation (donc de simplification). Aussi, il est possible de contourner le problème en calculant des résultats intermédiaires en préambule à l'évaluation d'une rétroaction pour diminuer la taille des expressions *Ladder* à évaluer. Ceci est particulièrement utile dans le cas où la complexité d'une expression atteint une des limites de l'automate programmable (comme par exemple la profondeur de la pile sur les automates *Koyo* qui est limitée à huit niveaux ; dans notre programme nous en utilisons déjà cinq). Il serait intéressant d'investiguer si la complexité de ces expressions est susceptible d'augmenter dans des proportions aussi alarmantes, parce qu'un tel résultat serait contraire à l'intuition. Finalement, si la condition suffisante peut être avantageusement exprimée par des conditions où l'événement est inhibé, on peut utiliser la négation logique de la condition d'inhibition comme optimisation.

Un problème plus sérieux se pose concernant les limitations de cette stratégie d'implémentation identifiées à la section 5.2.3. Ce contrôleur s'est révélé être un cas particulier d'un DSFBC que l'on pouvait ramener à un SFBC parce que chaque état du contrôleur pouvait être caractérisé par un état distinct de l'AFD du comportement libre. Il y a certainement lieu d'investiguer cette question pour savoir si ce type de contrôleur est peu fréquent ou s'il constitue une sous-classe utile de DSFBC.

Il serait probablement intéressant d'investiguer plus avant la notion de DSFBC pour voir si la même technique utilisée pour le calcul de l'état courant de l'AFD du comportement libre (l'association d'un vecteur d'état) ne pourrait pas être utilisée aussi pour déterminer l'état de la « mémoire » mentionnée dans la définition d'un DSFBC. Il se pourrait que ces derniers vecteurs d'état soient constitués d'éléments mémoire à la place des états des capteurs et des actionneurs. Il se pourrait aussi que l'on doive les synthétiser pendant l'opération de la rétroaction, mais la même stratégie d'implémentation de la rétroaction serait alors possible.

CONCLUSION

Dans ce mémoire, nous avons développé des schémas de génération de code pour automates programmables à partir d'une spécification par grafcets. Nous avons aussi développé une stratégie d'implémentation sur automates programmables de la fonction de rétroaction d'un contrôleur calculé à partir d'un algorithme basé sur la théorie du contrôle des systèmes à événements discrets, ainsi que le schéma de génération de code correspondant. Ce dernier résultat permet d'éviter d'avoir à reproduire la structure de transition de l'AFD du contrôleur, qui présente des problèmes apparemment insurmontables pour l'implémentation sur automates programmables.

Les schémas de génération de code proposés pour l'implémentation d'une spécification par grafcets comportent des limitations par rapport à la norme internationale. Ils ne couvrent pas les extensions incluses dans cette norme, notamment le forçage, le figeage et les macro-étapes.

La stratégie d'implémentation de la fonction de rétroaction d'un contrôleur, issue de la théorie du contrôle des SED, est pour l'instant limitée aux cas où il est possible d'établir un SFBC à partir de la fonction d'inhibition obtenue. Il y aurait lieu d'investiguer plus en détail le cas plus général de SFBC dynamique (DSFBC) qui couvre une famille plus large et plus utile de systèmes, notamment tous les systèmes où le calcul de la fonction de rétroaction exige la mémorisation d'une partie de l'historique du comportement du système.

Il n'a pas été possible dans ce projet de réaliser les outils logiciels de génération automatique du code, mais une telle réalisation semble très faisable à partir des résultats de ce projet. La génération de code pour une spécification par grafcets nécessiterait un travail important du côté de l'interface graphique d'interaction, à moins que l'on conçoive un langage de spécification qui pourrait servir de base à la génération de code. De plus, puisque les grafcets nécessitent en général une phase de mise au point, une fonction de simulation serait sans doute un atout dans une telle réalisation.

Pour ce qui est de la génération de code pour la fonction de rétroaction d'un contrôleur de SED, il semble qu'une fonction d'inhibition telle que fournie par l'outil SUCSEDES (ou l'équivalent) suffit.

Dans un cas comme dans l'autre, il faudrait disposer de la spécification complète du langage machine des automates programmables ciblés pour pouvoir générer du code machine directement. Aussi, pour des outils aussi complexes couvrant plus que la simple génération de code, il serait avantageux d'utiliser une approche où le générateur de code ne serait qu'un module, de sorte que l'on puisse générer du code pour plusieurs types d'automate programmable à partir d'un même système.

Annexe 1

L'usine-école MPS, une description détaillée

Dans cette annexe, nous donnons une description détaillée des dispositifs constituant l'usine-école MPS, ainsi qu'un portrait global. L'essentiel de l'information contenue dans cette annexe est puisée dans l'ensemble des références techniques suivantes : [FES98] [FES98a] [FES98b] [FES99] [FES99a].

Division en blocs fonctionnels

L'usine-école MPS se divise grossièrement en trois cellules de traitement, une par PLC. Si l'on respecte l'usage terminologique du manufacturier la division est la suivante.

1. Les stations de livraison et de test de pré-usinage constituent la première cellule de traitement.
2. Les stations d'usinage et de manutention constituent la seconde cellule.
3. La troisième cellule est formée de la station de tri post-usinage et d'une grue de réinsertion.

La grue de réinsertion ne fait pas partie de l'usine originale, mais elle a été ajoutée pour permettre à l'usine de fonctionner en circuit fermé.

La figure 37 nous donne la localisation physique des cellules de traitement. On peut aussi voir les PLC, les consoles d'opération (A) et la grue de réinsertion (B). Les pièces à traiter sont introduites dans le silo de la station de livraison et se déplacent de gauche à droite vers la station de tri post-usinage au fur et à mesure que le traitement s'effectue.

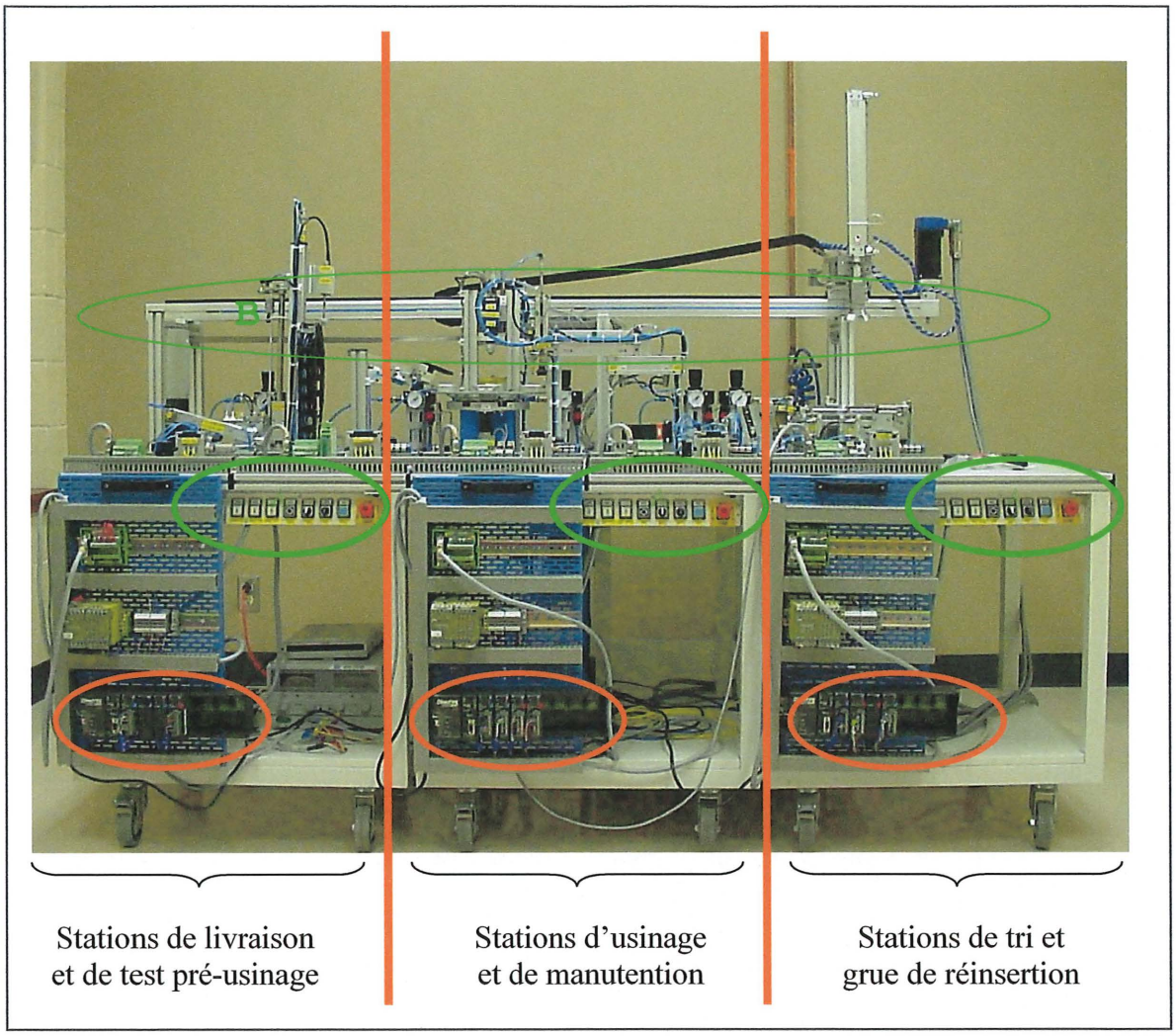

Figure 37. Usine-école MPS, blocs fonctionnels

Nomenclature

Pour faciliter les explications et ne pas avoir à répéter continuellement des locutions verbales telles que « le barillet du silo de la station de livraison », nous adoptons une convention de nomenclature basée sur la hiérarchie d'agrégation du matériel selon la structure suivante :

MPS.<Cellule>.<Station>.<Dispositif>

À titre d'exemple, la cellule n° 1 est constituée de deux stations :

1. la station de livraison ;
2. et la station de test pré-usinage.

À son tour la station de livraison est constituée de trois dispositifs :

1. le silo de chargement, à la base duquel on retrouve un barillet ;
2. un vérin d'injection qui déplace une pièce du barillet vers un dock de chargement ;
3. une grue de transbordement dont la fonction est de prendre une pièce au dock de chargement et de la placer sur la plate-forme de la station de test pré-usinage.

Dans l'ordre respectif nous avons donc les dispositifs suivants :

1. `MPS.C1.Livraison.Silo` pour le silo de la station de livraison ;
2. `MPS.C1.Livraison.Injecteur` pour le vérin d'injection ;
3. `MPS.C1.Livraison.Grue` pour la grue de transbordement.

La grue de transbordement est elle-même formée de deux dispositifs : un balancier et une ventouse pneumatique. On devine aisément qu'il ne s'agit que d'ajouter un qualificatif au nom de la grue pour obtenir le nom de chacun de ces dispositifs. De façon générale, on peut omettre le niveau hiérarchique supérieur (c'est-à-dire « `MPS.` ») lorsque le contexte est clair. On a donc, pour les PLC les désignations `MPS.C1.PLC`, `MPS.C2.PLC` et `MPS.C3.PLC`.

Particularités concernant les vérins

Pour la description des dispositifs, par souci de brièveté, nous utilisons une forme en équation dont voici le schéma général :

<Dispositif> = { (<*actionneurs*>) : (<*capteurs*>) }

Ce genre d'équation regroupe tous les actionneurs et les capteurs directement reliés au fonctionnement d'un dispositif d'une façon compacte. On ajoute une description des configurations significatives des capteurs et des actionneurs, ainsi que toute annotation jugée pertinente pour ce dispositif comme dans l'exemple de la figure 38. Dans certains cas on peut adjoindre le diagramme d'un automate d'états pour décrire son fonctionnement.

L'exemple de la figure 38 présente un dispositif doté de capteurs pour chacune de ses positions significatives. Dans ce cas il s'agit d'un dispositif ayant les mêmes caractéristiques qu'un vérin. Il arrive cependant que certains de ces capteurs soient absents et dans ce cas, la position demeure significative pour l'opération du dispositif. Il nous faut remédier à cette carence.

MPS.C1.Livraison.Grue.Flèche = {(Y0,Y1):(X2,X3)}

Description : Grue à balancier de la station de livraison.
Commande : Bistable piloté.
Actionneurs :
($\overline{Y0}$,Y1) : Flèche vers MPS.C1.Test.Plate-forme.
(Y0,$\overline{Y1}$) : Flèche vers MPS.C1.Livraison.Dock.
Capteurs :
($\overline{X2}$,$\overline{X3}$) : Flèche en position intermédiaire.
($\overline{X2}$,X3) : Flèche en position à MPS.C1.Test.Plate-forme.
(X2,$\overline{X3}$) : Flèche en position à MPS.C1.Livraison.Dock.
N.B. :
(Y0,Y1) : Configuration illégale.
(X2,X3) : Physiquement impossible.

Figure 38. Description du dispositif MPS.C1.Livraison.Grue.Flèche

Pour remédier à cette situation, on utilise des délais comme dans l'exemple de la figure 39 qui décrit la barrière du glissoir d'admission à la station d'usinage et de manutention. Puisqu'il s'agit d'un vérin à commande monostable, on a le délai caractéristique de réaction d'abord, puis le délai de retour à la position stable ensuite. Il faut donc sept dixièmes de seconde à partir de l'application de la commande pour être certain que la barrière soit en position ouverte et un dixième de seconde pour sa fermeture.

Nous n'avons ici discuté que des particularités concernant les vérins ou de dispositifs ayant des caractéristiques fonctionnelles équivalentes. Il faut noter que ce genre de spécification est aussi utilisé dans le cas d'autres dispositifs. Dans ce dernier cas, il faut donner une description plus abondante de la façon dont le dispositif se comporte pour bien décrire ses positions significatives, mais l'usage de délais demeure valide.

Particularités concernant les grues

L'usine-école MPS comprend trois grues :

1. `MPS.C1.Livraison.Grue ;`

`MPS.C1.Test.Barrière = {(Y14):(0.7 s,0.1 s)}`

Description : Barrière du glissoir d'admission à la station d'usinage.
Commande : Monostable.
Actionneurs :
(Y14) : Fermer la barrière.
Aucun capteurs :
$(Y14 + 0.7\,s)$: Barrière ouverte.
$(\overline{Y14} + 0.1\,s)$: Barrière fermée.

Figure 39. Description du dispositif `MPS.C1.Test.Barrière`

2. `MPS.C2.Manutention.Grue` ;
3. et `MPS.C3.Réinjection.Grue`.

Ces grues semblent très différentes à priori, puisque `MPS.C1.Livraison.Grue` n'est composée que d'un balancier et d'une ventouse pneumatique, alors que `MPS.C3.Réinjection.Grue` contient plusieurs vérins et est actionnée par un servomoteur. Pour ne pas se perdre dans des considérations de nomenclature, il faut considérer une grue selon ses composants fonctionnels. On constate alors que les trois grues de l'usine-école ont la même structure.

Une grue sert à transporter du matériel d'un point à un autre dans l'espace, et à cette fin elle peut devoir agir selon une, deux ou trois dimensions. Au minimum, une grue nécessite une flèche et une pince (c'est-à-dire un outil de préhension, pour nous ici, une ventouse pneumatique est une pince), c'est le cas de `MPS.C1.Livraison.Grue` ; celle-ci n'agit que dans le plan vertical.

De façon générale, une grue peut aussi disposer d'un pont, pour le mouvement parallèle au sol, et d'un treuil pour le mouvement vertical. Pour une grue, nous avons donc les composants suivants.

1. La flèche qui imprime un mouvement parallèle au sol dans un plan. C'est le cas du balancier dans `MPS.C1.Livraison.Grue`, du servomoteur dans `MPS.C3.Réinjection.Grue` et de la flèche semi-circulaire dans `MPS.C2.Manutention.Grue`. Cette partie de la grue n'est pas optionnelle.
2. Une pince ou outil de préhension ; ventouse pneumatique ou la pince comme telle dans `MPS.C3.Réinjection.Grue`. Ce composant est aussi nécessaire.
3. Un pont qui imprime un mouvement parallèle au sol aussi mais habituellement dans un axe différent du plan de la flèche (à moins qu'il n'en soit une extension comme c'est le cas de `MPS.C2.Manutention.Grue`. Ce composant est optionnel.
4. Finalement, un treuil qui imprime un mouvement vertical. Ce composant est aussi optionnel ; `MPS.C2.Manutention.Grue` et `MPS.C3.Réinjection.Grue` en sont dotées.

Nous traiterons toutes ces grues de la même façon.

La console d'opération

Chaque cellule de traitement est munie d'une console d'opération. Ces consoles sont désignées : `C1.Console`, `C2.Console` et `C3.Console`. On les retrouve à la figure 37 en (A), dont on peut voir le détail à la figure 40.

De tous ces dispositifs, seul le bouton d'arrêt d'urgence a une fonction prédéfinie ayant préséance sur le logiciel. Tous les autres dispositifs sont à la disposition du programmeur pour implémenter des fonctions de contrôle du logiciel dans le PLC de la cellule correspondante. Nous donnons ici de façon générique, le comportement de ces dispositifs (les mêmes sur chaque console), y compris le fonctionnement du bouton d'arrêt d'urgence. Nous donnons ensuite un tableau d'assignation des terminaux qu'on retrouve sur chacune des consoles.

Le bouton d'arrêt d'urgence est un poussoir particulier. Il a deux positions stables distinctes : retirée et enfoncée. Il est aussi muni d'un voyant lumineux qui est sous le contrôle de la logique câblée (donc inaccessible au programme du PLC). Il est cependant associé à un capteur de sorte que le programme du PLC peut en connaître l'état. Supposons un capteur X_i associé à un bouton d'arrêt d'urgence, alors :

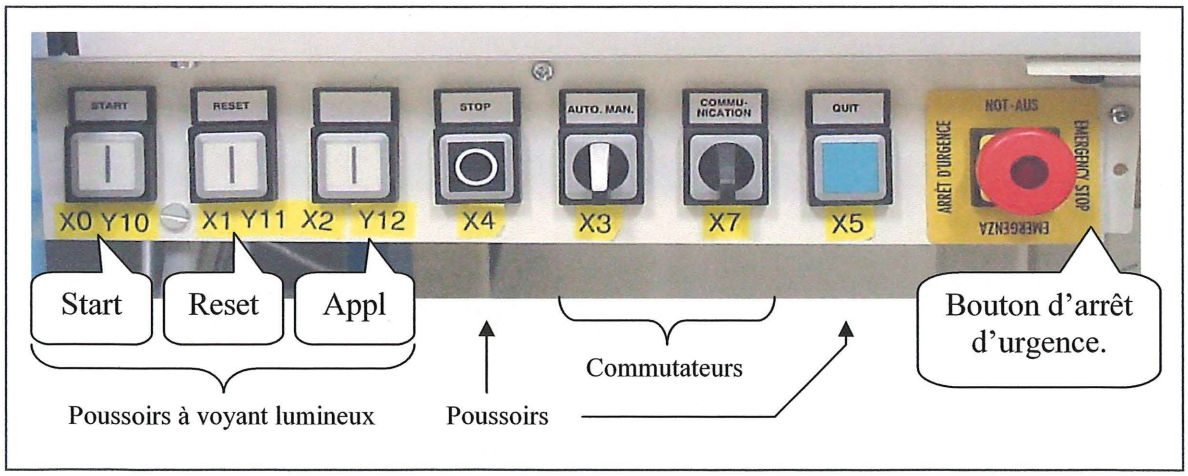

Figure 40. Console d'opération

(X_i) signifie que la cellule est en état d'arrêt d'urgence, et inversement en l'absence d'un signal au capteur.

En état d'arrêt d'urgence, le voyant lumineux du bouton d'arrêt d'urgence est allumé, tous les actionneurs sont forcés à zéro, ce qui veut dire que le programme du PLC ne peut pas les changer, et l'alimentation électrique de puissance est coupée. Notez que le programme du PLC n'est pas nécessairement interrompu et peut continuer à fonctionner.

Pour mettre une cellule en fonction à partir d'un arrêt d'urgence (c'est-à-dire quand le voyant du bouton est allumé), il faut tirer le bouton d'arrêt d'urgence vers soi jusqu'à ce qu'on entende le verrou du bouton s'enclencher, et ensuite appuyer sur le poussoir lumineux « RESET ». Alors, le voyant lumineux devrait s'éteindre et on devrait entendre les relais mordre discrètement ; signe que la cellule vient d'entrer en fonction.

Pour mettre une cellule en arrêt d'urgence lorsqu'elle est en fonction (voyant lumineux éteint), il suffit de pousser le bouton d'arrêt d'urgence jusqu'à l'entendre se verrouiller. Le voyant devrait s'allumer.

Un bouton poussoir à voyant lumineux est un dispositif de la forme $\{ (Y_j) : (X_i) \}$ où :

(X_i) signifie que le bouton est en position enfoncée (position instable).

(Y_j) commande le voyant lumineux.

Un bouton poussoir (à voyant lumineux ou non) n'a qu'une position stable qui correspond à $(\overline{X_i})$ dans l'équation ci-dessus.

Il y a trois boutons poussoir à voyant lumineux sur une console d'opération : « START », « RESET » et un troisième sans étiquette (« APPL » dans la figure 40). De ces trois poussoirs,

seul « RESET » a une fonction lors de la mise en fonction d'une cellule à partir d'un arrêt d'urgence. Les autres, malgré leur étiquette, n'ont aucune fonction câblée.

Il y a deux poussoirs simples : « STOP » et « QUIT », encore une fois sans fonction câblée. Un poussoir simple est un dispositif de la forme { (X_i) }, ayant le même comportement qu'un poussoir à voyant lumineux, mais sans la partie commande, pour le voyant. En particulier, il n'a qu'une position stable.

Finalement, la console a deux commutateurs à deux positions ; « AUTO/MAN » et « COMMUNICATION » sans fonction câblée. Un commutateur à deux positions est un dispositif de la forme { (X_i) } à deux positions stables, où :

(X_i) signifie commutateur en position stable 1 ;

$(\overline{X_i})$ signifie commutateur en position stable 2.

Malgré les étiquettes, aucune interprétation n'est liée à un commutateur par défaut. On peut assigner par programmation n'importe quelle signification aux capteurs associés à ces dispositifs. Le tableau 4 donne l'assignation des terminaux des PLC aux différentes consoles.

Tableau 4. Assignation des terminaux des PLC pour chaque console d'opération

Dispositif / Console (cellule)	START (*PL*)*	RESET (PL)*	APPL (*PL*)*	STOP (*PS*)**	AUTO/ MAN (C)***	COMM (C)***	QUIT (*PS*)**	ARRÊT URGENCE
C1	(Y5,X40)	(Y6,X41)	(Y7,X42)	X44	X43	X7	X45	X46
C2	(Y10,X0)	(Y11,X1)	(Y12,X2)	X4	X3	X7	X5	X6
C3	(Y10,X0)	(Y11,X1)	(Y12,X2)	X4	X3	X7	X5	X6

* PL : Poussoir à voyant lumineux
** *PS*: Poussoir simple
*** *C* : Commutateur à deux positions

Notez que dans le tableau 4, les indices sont en octal.

La station de livraison

Le premier PLC (celui d'extrême gauche quand on fait face au MPS) définit et contrôle la cellule `MPS.C1`. Nous désignons ce PLC par `MPS.C1.PLC`. Cette cellule est composée de deux stations (selon la nomenclature du manufacturier) : la station de livraison (désignée `C1.Livraison`) et la station de test pré-usinage (désignée `C1.Test`). Nous décrivons ici les dispositifs de la station `C1.Livraison`, que l'on peut voir à la figure 41. Cette station comporte trois dispositifs.

1. `C1.Livraison.Silo`, composé d'une colonne et d'un barillet muni d'un capteur optique.
2. `C1.Livraison.Injecteur`, composé d'un vérin à commande monostable à position normalement en extension, équipé de deux capteurs de fin de course.
3. `C1.Livraison.Grue`, elle-même composée de deux dispositifs :

   `C1.Livraison.Grue.Flèche`, un balancier ;
   `C1.Livraison.Grue.Ventouse`, l'outil de préhension qui est en fait une ventouse pneumatique à vide (la « pince » de la grue).

`C1.Livraison.Silo` = { `(X6)` }

Description : Silo de chargement des pièces, dispositif passif de capacité 12 et politique de distribution FIFO. Le silo est formé d'une colonne en plexiglas à la base de laquelle il y a un barillet muni d'un capteur optique qui détecte la présence d'une pièce.

Commande : Dispositif passif.

Actionneurs : Aucun.

Capteurs : Un capteur discret.

$(\overline{X6})$: Une pièce est présente dans le barillet à la base de la colonne du silo.

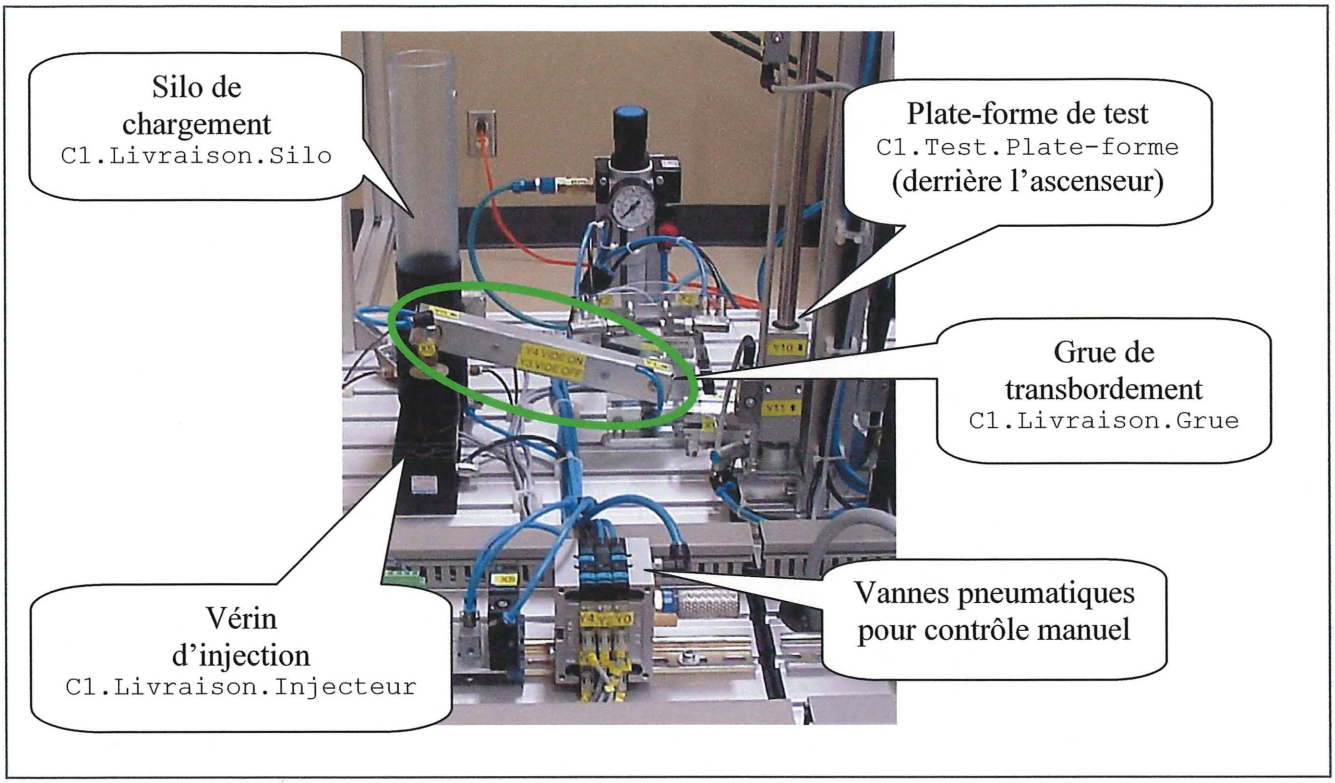

Figure 41. Station de livraison de la première cellule de traitement

Notes :

Le capteur `X6` a des états transitoires parasites puisque l'injecteur obstrue le barillet durant une opération d'injection. Le capteur n'est significatif que si le capteur `X0` du dispositif d'injection est au niveau logique *vrai*, signifiant que l'injecteur n'obstrue plus le barillet. De plus le front descendant de `X6` doit être temporisé pour en assurer la validité.

`C1.Livraison.Injecteur` = { `(Y2) : (X0,X1)` }

Description : Vérin d'injection ; transporte une pièce présente dans le barillet en position au dock de chargement et maintient cette pièce en position pour la grue de transbordement.

Commande : Discrète monostable.

Actionneurs :

`(Y2)` : Injecter une pièce.

Capteurs : Capteurs discrets de fin de course.

($X0, \overline{X1}$) : Le vérin est en position réarmée (normalement allongée) et n'obstrue pas le barillet du silo.

($\overline{X0}, X1$) : Le vérin est en position au dock de chargement et tient la pièce fermement en position.

($\overline{X0}, \overline{X1}$) : Le vérin est en déplacement (et obstrue le barillet).

Notes :

La configuration (X0,X1) des capteurs est physiquement impossible.

C1.Livraison.Grue.Flèche = { (Y0,Y1) : (X2,X3) }

Description : Balancier de la grue de transbordement ; déplace la grue entre le dock de chargement et la plate-forme de test.

Commande : Discrète bistable piloté.

Actionneurs :

($Y0, \overline{Y1}$) : Déplacer le balancier vers le dock de chargement.

($\overline{Y0}, Y1$) : Déplacer le balancier vers la plate-forme de test.

Capteurs : Capteurs discrets de fin de course.

($X2, \overline{X3}$) : Le balancier est en position au dock de chargement.

($\overline{X2}, X3$) : Le balancier est en position à la plate-forme de test.

($\overline{X2}, \overline{X3}$) : Le balancier est en position intermédiaire entre la plate-forme de test et le dock de chargement, il peut être en mouvement ou à l'arrêt.

Notes :

La configuration (X2,X3) des capteurs est physiquement impossible.

En position intermédiaire, il est impossible de garantir que le balancier n'obstrue pas le mouvement de l'ascenseur de la station de test. Même l'usage d'une temporisation sur la tombée de l'un ou l'autre des capteurs ne peut garantir la position du balancier en position intermédiaire puisque celle-ci dépend de la pression dans le circuit pneumatique et du gabarit des conduites d'air comprimée (perte de pression par friction).

En position au dock de chargement, la ventouse de la grue se trouve à obstruer le libre mouvement d'une pièce lors d'une injection. La tombée de X2 est suffisante pour assurer que la ventouse dégage l'injecteur, mais insuffisante pour assurer qu'une pièce saisie par la grue dégage l'injecteur.

En position à la plate-forme de test, le balancier obstrue le mouvement de l'ascenseur de la station de test (danger réel de bris matériel). La seule façon d'être certain que le balancier dégage l'ascenseur est d'obtenir la présence de X2.

C1.Livraison.Grue.Ventouse = { (Y3,Y4) : (X5) }

Description : Ventouse pneumatique à succion ; outil de préhension de la grue.

Commande : Discrète, bistable non piloté.

Actionneurs :

($Y3, \overline{Y4}$) : Désactive la succion si elle était en fonction.

($\overline{Y3}, Y4$) : Active la succion si elle n'était pas en fonction.

Capteurs : Un capteur discret pour détecter la saisie d'une pièce.

(X5) : La ventouse tient une pièce (la succion doit nécessairement être en fonction).

Notes :

S'il n'y a pas de pièce sous la ventouse lorsque la succion est activée, on n'obtient jamais X5.

La station de test pré-usinage

La station de test pré-usinage dont on peut voir une photo à la figure 42, suit physiquement la station de livraison dans l'usine-école MPS. Sur cette figure on peut distinguer trois dispositifs : l'ascenseur, le micromètre et la passerelle de transfert vers la station d'usinage. Les deux autres dispositifs, la plate-forme de test et son vérin d'évacuation, se trouvent à la base de l'ascenseur sur la photographie, et on peut en voir le détail à la figure 43. La station de test pré-usinage se compose donc de cinq dispositifs.

La plate-forme pour les tests de pré-usinage se compose de deux dispositifs : un lecteur de caractéristiques physiques et un évacuateur qui pousse une pièce présente sur la plate-forme vers un autre dispositif qui dépend du résultat des tests. Il y a deux tests : l'un s'assure que les caractéristiques physiques de la pièce sont adéquates (matériaux et coloration), l'autre vérifie que l'épaisseur de la pièce répond à une norme.

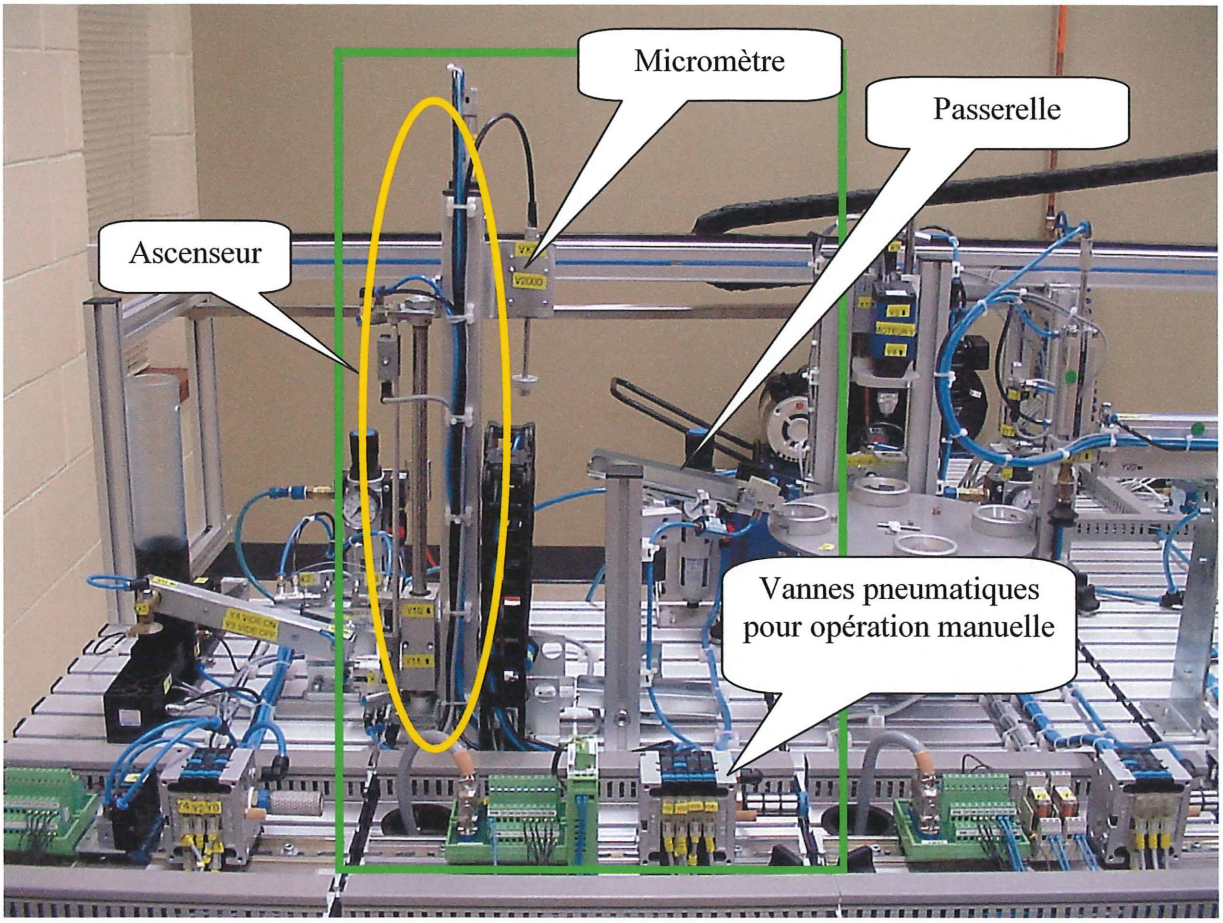

Figure 42. Station de test pré-usinage de la première cellule de traitement

`C1.Test.Plate-forme.Lecteur` = { `(X10,X11,X12)` }

Description : Le lecteur évalue les caractéristiques physiques d'une pièce déposée sur la plate-forme ; composition et couleur.

Commande : Dispositif passif (pas de commande).

Actionneurs : Aucun.

Capteurs : Trois capteurs discrets qui évaluent la constitution physique de la pièce sur la plate-forme.

`(X10)` : Capteur inductif, la pièce en examen est constituée de métal (aluminium dans notre cas).

`(X11)` : Capteur capacitif, une pièce est présente sur la plate-forme.

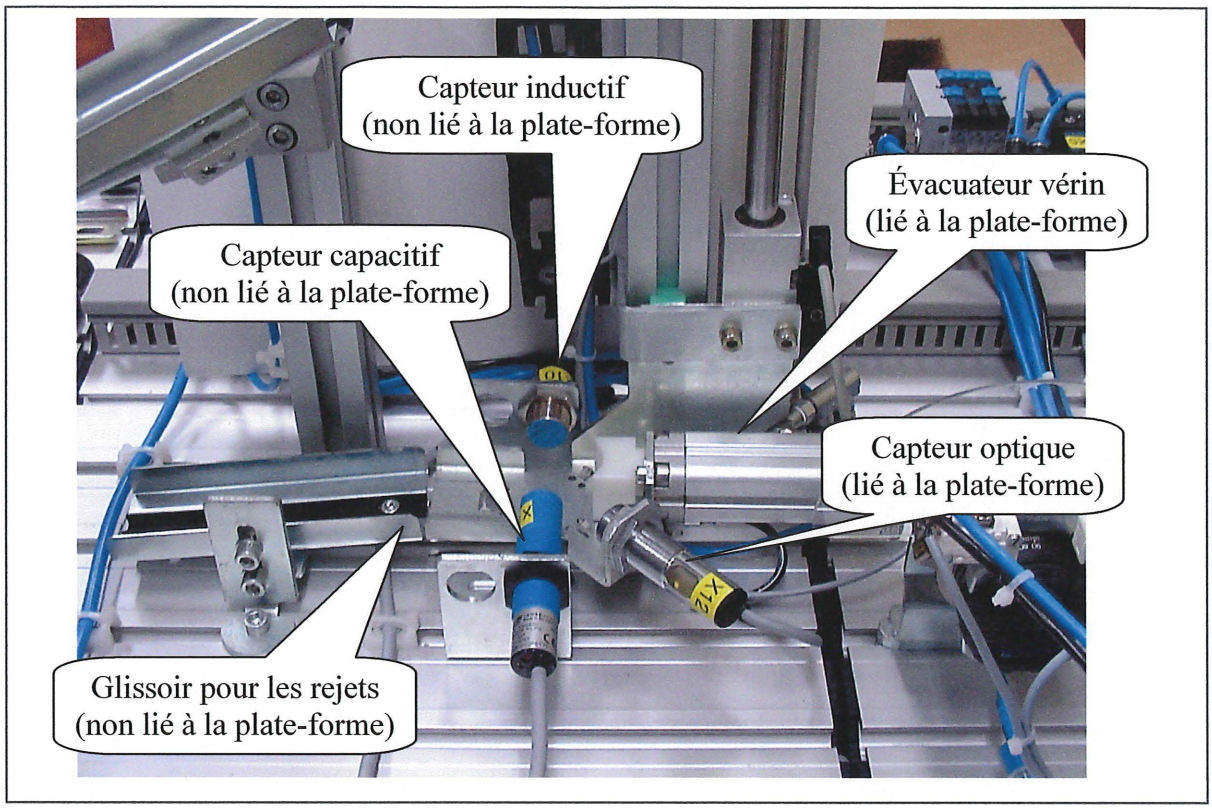

Figure 43. Plate-forme de test et évacuateur

(X12) : Capteur optique, la pièce en examen est de couleur claire (dans notre cas, elle n'est pas de couleur noire).

Notes :

Le tableau 5 donne l'interprétation des différentes configurations que peuvent prendre les capteurs de la plate-forme. Les entrées de ce tableau qui n'ont pas d'interprétation sont invalides et doivent provoquer le rejet de la pièce.

Il y a deux fonctions booléennes significatives : $(\overline{X11} \wedge \overline{X10} \wedge \overline{X12})$ qui dénote positivement l'absence de pièce sur la plate-forme et $(X11 \wedge (\overline{X10} \vee X12))$ qui dénote une pièce valide sur la plate-forme.

Il est préférable d'introduire une temporisation lorsque une pièce apparaît sur la plate-forme puisque les capteurs pourraient subir des transitoires avant de se stabiliser. Une temporisation de 0.1 s devrait suffire.

Tableau 5. Interprétation de l'état des capteurs de la plate-forme de test

X11	X10	X12	Interprétation
0	0	0	Pas de pièce sur la plate-forme
0	0	1	
0	1	0	
0	1	1	
1	0	0	Pièce de plastique de couleur foncée (noire)
1	0	1	Pièce de plastique de couleur claire (rouge)
1	1	0	
1	1	1	Pièce métallique de couleur claire

`C1.Test.Plate-forme.Évacuateur = { (Y12) : (X15, 0.5 s) }`

Description : L'évacuateur est constitué d'un vérin attenant à la plate-forme de test (il bouge avec la plate-forme). Il sert à pousser une pièce hors de la plate-forme soit dans la coulisse d'évacuation, lorsque l'ascenseur est en bas et que la pièce est rejetée au test de caractéristiques physiques, ou encore sur la passerelle d'admission à la station d'usinage, lorsque l'ascenseur est en haut et que la pièce passe avec succès tous les tests de pré-usinage.

Commande : Vérin à commande monostable à position normalement rétractée.

Actionneurs : Type discret.

`(Y12)` : Initier l'évacuation d'une pièce (déploie le vérin).

Capteurs : Type discret, détecteur de fin de course.

`(X15)` : Le vérin a regagné sa position normale.

Notes :

Il est nécessaire d'utiliser une temporisation au moment de l'évacuation d'une pièce puisque le vérin ne dispose pas d'un capteur de fin de course indiquant qu'il est complètement déployé.

`C1.Test.Ascenseur = { (Y10,Y11) : (X13,X14) }`

Description : L'ascenseur monte et descend la plate-forme de test. Pour admettre une pièce à la station de test pré-usinage, l'ascenseur doit être descendu et sa plate-forme de test

doit être vide. Après l'admission d'une pièce, si la pièce passe avec succès le test des caractéristiques physiques, l'ascenseur monte la pièce au micromètre pour effectuer le test de son épaisseur.

Commande : Vérin à commande bistable piloté.

Actionneurs : Type discret.

(Y10, $\overline{\text{Y11}}$) : Descendre l'ascenseur si elle est en haut.

($\overline{\text{Y10}}$, Y11) : Monter l'ascenseur si elle est en bas.

Capteurs : Type discret, capteur de fin de course du vérin.

(X13, $\overline{\text{X14}}$) : L'ascenseur est en bas.

($\overline{\text{X13}}$, X14) : L'ascenseur est en haut.

($\overline{\text{X13}}$, $\overline{\text{X14}}$) : L'ascenseur est en position intermédiaire, impossible de dire où exactement.

Notes :

Attention, l'ascenseur ne doit pas bouger à moins que `C1.Livraison.Grue` ne soit en position au dock de chargement, puisque c'est la seule position de cette grue qui assure qu'elle n'obstrue pas les mouvements de l'ascenseur et qu'il y a risque réel de bris de matériel.

Inversement, pour que `C1.Livraison.Grue` puisse livrer une pièce à la station de test de pré-usinage, l'ascenseur doit être descendu et sa plate-forme de test doit être vide.

La seule façon de s'assurer que `C1.Livraison.Grue` dégage effectivement le dock de chargement, en particulier lorsque la grue a saisie une pièce, est de l'amener en position à la plate-forme de test, ce qui requiert au minimum que l'ascenseur soit stable en bas.

`C1.Test.Micromètre` se compose de deux dispositifs : un piqueur qui amène le micromètre en place pour une lecture et un registre pour récupérer la valeur de la mesure.

`C1.Test.Micromètre.Piqueur` = { `(Y13) : (X16,0.3 s)` }

Description : Le piqueur est composé d'un vérin pour positionner le micromètre à sa position de mesure.

Commande : Vérin à commande monostable à position normalement rétractée.

Actionneurs : Un actionneur discret.

`(Y13)` : Engager le micromètre.

Capteurs : Un capteur discret de fin de course.

(X16) : Le micromètre est complètement engagé à sa position de mesure (vérin complètement déployé).

Notes :

Il est nécessaire d'utiliser une temporisation si l'on veut être certain que le micromètre a complètement dégagé la position de mesure et réintégré sa position normale, puisque le vérin n'a pas de capteur de fin de course pour sa position normale.

`C1.Test.Micromètre.Registre = { (VX20) }`

Description : Le micromètre contient un registre de 16 bits dont le contenu ne sera valide que lorsque le micromètre sera physiquement engagé et stabilisé à sa position de mesure.

Commande : Aucune (voir la commande du piqueur).

Actionneurs : Aucun.

Capteurs : Un registre de 16 bits.

(VX20) : Quand le micromètre est complètement engagé à sa position de mesure, ce registre contient la valeur en millivolts correspondant à l'épaisseur de la pièce.

Notes :

Il peut être préférable d'utiliser une temporisation (courte, disons 0.1 s) au moment ou l'on détecte que le micromètre est complètement engagé, pour permettre au dispositif de se stabiliser.

Il y a trois épaisseurs de pièce : 22.5 mm, 25 mm et 27.5 mm. Ces valeurs correspondent respectivement à 908 mV, 1295 mV et 1651 mV.

Les pièces de 27.5 mm causent une panne au dock de chargement ; le capteur X2 ne se vérifie jamais car la pièce est plus épaisse que la tolérance mécanique du capteur.

Le dispositif mesure le voltage à la sortie d'un potentiomètre qui devrait être linéaire à partir de zéro millivolt. Cependant le potentiomètre ne semble pas être linéaire et présente une dérive d'environ 10 mV par 2.5 mm qui elle, semble linéaire. Il est donc difficile de donner des valeurs de seuils précises ; il faudrait connaître la courbe du potentiomètre ou la déduire, il semble préférable de fonctionner empiriquement.

`C1.Test.Passerelle` est un dispositif passif de capacité de une pièce et de politique de distribution FIFO ; elle est donc vide ou pleine. Elle est munie de deux dispositifs ; une barrière à la sortie (lui permettant de retenir une pièce) et un indicateur discret lui permettant de savoir si la station d'usinage qui se trouve en aval, est dans un état qui lui permet d'accepter une nouvelle pièce ou non.

`C1.Test.Passerelle.Barrière = { (Y14) : (0.1 s, 0.3 s) }`

Description : Barrière de rétention à la sortie de la passerelle, ouverte ou fermée.

Commande : Vérin à commande monostable à position normalement déployée (c'est-à-dire que sans alimentation, la barrière est fermée).

Actionneurs : Type discret.

`(Y14)` : Provoque l'ouverture de la barrière.

Capteurs : Aucun sur le vérin de la barrière.

Notes :

Il est nécessaire d'utiliser des temporisations pour chacune des positions du vérin, puisqu'il n'y a aucun capteur sur ce vérin.

Les temporisations sur les réactions du vérin ne tiennent pas compte du délai requis pour qu'une pièce, présente sur la passerelle à l'ouverture, ait le temps de passer et de dégager la barrière. Refermer trop tôt pourrait coincer la pièce ou simplement l'empêcher de passer. Une temporisation de 0.7 s permet à une pièce de dégager la barrière. Donc on peut considérer, si on voulait simplifier, que ce dispositif a un délai de latence à l'ouverture de 0.7 s.

`C1.Test.Passerelle.Usinage-prête = { (X4) }`

Description : La passerelle doit utiliser un signal en provenance de la station d'usinage pour coordonner son action avec cette dernière.

Commande : Aucune ; dispositif passif.

Actionneurs : Aucun.

Capteurs : Un signal discret contrôlé par la station d'usinage.

`(X4)` : La station d'usinage est dans un état permettant l'admission d'une nouvelle pièce.

Notes :

Puisqu'il est impossible à la station d'usinage de connaître l'état de la station de test, c'est la station d'usinage qui devra tenir compte du délai de latence de la barrière, et coordonner son action avec la sienne.

La station d'usinage

La figure 44 nous montre la station d'usinage.

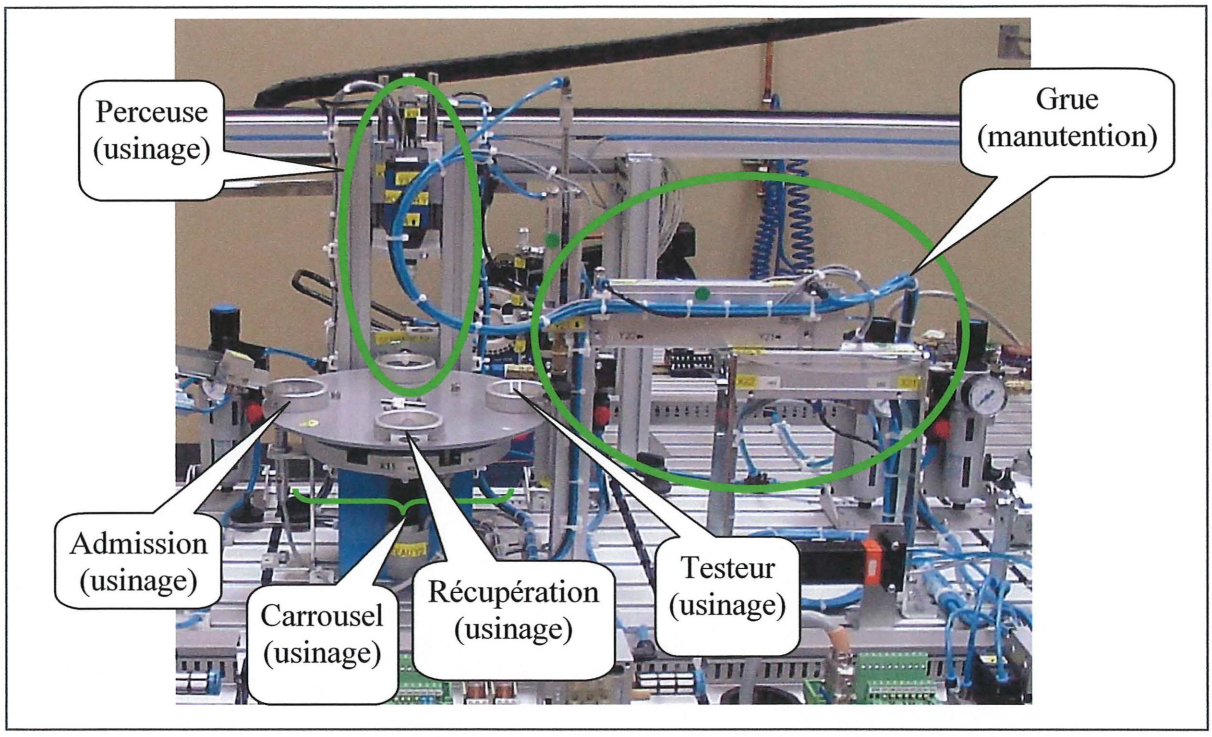

Figure 44. Station d'usinage de l'usine-école MPS

La station d'usinage comporte trois dispositifs principaux : un carrousel muni de quatre puits d'usinage, un dispositif de perçage et un dispositif pour tester la profondeur de la perforation (un test de contrôle de qualité de l'usinage).

Il y a quatre positions d'usinage : l'admission, la perceuse, le testeur et la position de récupération où la grue de la station de manutention récupère les pièces usinées.

`C2.Usinage.Plateau` est composé de deux dispositifs : un carrousel pour le positionner dans l'une de ses quatre positions valides et une sentinelle d'admission pour lui permettre de signaler à la station en aval (la station de test pré-usinage) quand il est prêt à recevoir une nouvelle pièce pour l'usinage. Le plateau est muni de quatre puits, disposés aux points cardinaux, qui servent à recevoir des pièces pour l'usinage. Le carrousel a comme fonction de tourner le plateau et de maintenir les puits en bon alignement avec les positions permettant

l'usinage. La position d'admission que l'on peut voir sur la figure est groupée avec le plateau parce que sa fonction est intimement liée à celle du mécanisme de positionnement des puits d'usinage (qu'on peut distinguer sur la figure).

`C2.Usinage.Plateau.Carrousel = { (Y2) : (X11) }`

Description : Moteur avec engrenage et un capteur permettant d'effectuer une rotation des puits d'usinage et de maintenir l'alignement des puits avec les positions d'usinage.

Commande : Un simple relais.

Actionneurs :

`(Y2)` : Moteur alimenté.

Capteurs : Inductif discret.

`(X11)` : La présence de ce capteur indique l'alignement des puits avec les positions d'usinage.

Notes :

Pour tourner le plateau d'une position, on doit activer la commande, attendre la disparition de `X11`, ensuite attendre que `X11` réapparaisse de nouveau et immédiatement couper l'alimentation du moteur. Cette manœuvre est très sensible aux délais entre l'apparition de `X11` et la coupure de l'alimentation du moteur (dû à l'inertie du mécanisme d'horlogerie à l'arrêt du moteur). Ce délai ne doit pas dépasser 0.1 s.

`C2.Usinage.Plateau.Sentinelle = { (Y0) : (X10) }`

Description : Il s'agit d'un dispositif passif de rétroaction ; la commande ici contrôle la rétroaction (l'apparition d'un indicateur dans la station en aval : `C1.X4`).

Commande : Un simple relais.

Actionneurs : Contrairement à l'usage, l'actionneur ici est une sortie du dispositif.

`(Y0)` : La station d'usinage est en position de recevoir une nouvelle pièce.

Capteurs : Discret capacitif (détecte la présence d'une pièce dans le puit d'admission).

`(X10)` : Il y a une pièce dans le puit à la position d'admission.

Notes :

Lorsque le plateau est stable à l'arrêt, c'est-à-dire que tous les puits sont vides ou que les dispositifs des autres puits (perceuse, testeur ou grue de manutention) n'ont pas terminé leur opération, il est possible d'admettre une nouvelle pièce à l'usinage en

position d'admission si le puit d'admission est vide. Donc, dans les conditions énumérées, $\overline{\texttt{X10}} \Rightarrow \texttt{Y0}$, et puisque $\texttt{C2.Y0} \Rightarrow \texttt{C1.X4}$, la station de test de pré-usinage pourra éventuellement ouvrir la barrière de sa passerelle.

Notez que la station de test pré-usinage ne peut pas signaler à la station d'usinage qu'une pièce est prête pour usinage. Ainsi, c'est la station d'usinage qui devra gérer la possibilité qu'une pièce soit déjà en transit vers le puit d'admission lorsqu'elle doit se déclarer inapte à la recevoir une pièce (quand elle met `Y0` à zéro). Il suffit que le plateau attende un temps supérieur au délai de latence de la barrière de la passerelle (voir les notes de `C1.Test.Passerelle.Barrière`), après avoir mis `Y0` à zéro, avant de commencer à tourner les puits.

`C2.Usinage.Perceuse` est composé de trois dispositifs : un piqueur pour guider la perceuse verticalement vers le bas lors du perçage, un étau pour permettre le serrage de la pièce que l'on veut percer et le moteur de la perceuse.

`C2.Usinage.Perceuse.Piqueur` = { `(Y3,Y4)` : `(X14,X15)` }

Description : Le corps de la perceuse est monté sur un patin admettant un mouvement vertical rectiligne contrôlé par un vérin : le piqueur. Le piqueur sert à contraindre physiquement le mouvement lors du perçage pour assurer une perforation bien droite et uniforme ainsi que l'application d'une force adéquate pour atteindre la profondeur voulue.

Commande : Vérin à commande bistable non piloté.

Actionneurs :

$(\texttt{Y3}, \overline{\texttt{Y4}})$: Abaisser le piqueur.

$(\overline{\texttt{Y3}}, \texttt{Y4})$: Remonter le piqueur.

Capteurs : Deux capteurs discrets de fin de course du vérin.

$(\texttt{X14}, \overline{\texttt{X15}})$: Piqueur en haut (dégage complètement la position).

$(\overline{\texttt{X14}}, \texttt{X15})$: Piqueur en bas (en fin de course).

Notes :

Il est essentiel que le plateau ne soit pas en mouvement quand on abaisse le piqueur, et qu'il demeure à l'arrêt durant toute l'opération jusqu'à ce que le piqueur soit complètement remonté sinon il peut y avoir bris de matériel.

Il faut aussi que l'étau enserre la pièce à percer avant que l'opération de perçage ne soit initiée et la maintienne fermement durant toute l'opération.

C2.Usinage.Perceuse.Étau = { (Y5) : (X12,X13) }

Description : À la base de la position de perçage se trouve un dispositif de serrage appelé l'étau, qui s'insère à la base du puit (par une fente) pour enserrer la pièce et la maintenir en position pendant l'opération de perçage.

Commande : Vérin à commande monostable.

Actionneurs : Discret.

(Y5) : Serrer la pièce.

Capteurs : Deux capteurs discrets de fin de course du vérin.

($X12,\overline{X13}$) : Étau ouvert.

($\overline{X12},X13$) : Étau en position de serrage (maintient la pièce fermement).

Notes :

Il faut que l'étau enserre la pièce à percer avant que l'opération de perçage ne soit initiée et la maintienne fermement durant toute l'opération.

C2.Usinage.Perceuse.Moteur = { (Y1) }

Description : Le moteur est actionné par un simple relais.

Commande : Relais.

Actionneurs : Discret.

(Y1) : Moteur de la perceuse en fonction.

Capteurs : Aucun.

Notes :

Il n'y a aucun capteur pour s'assurer que le moteur a atteint sa vitesse nominale de perçage ; l'usage d'une courte temporisation (0.1 s) pourrait être indiqué.

C2.Usinage.Testeur = { (Y6) : (X16,X17) }

Description : Une tige métallique (la sonde) montée sur un patin à action verticale et actionnée par un vérin servant à vérifier que la profondeur de la perforation est suffisante.

Commande : Vérin à action monostable.

Actionneurs : Discret.

(Y6) : Enfoncer la sonde.

Capteurs : Deux capteurs discrets de fin de course du vérin.

(X16,$\overline{X17}$) : Sonde désengagée complètement (sonde en haut).

($\overline{X16}$,X17) : La sonde a atteint la profondeur nominale pour la perforation (fin de course du vérin en bas).

Notes :

Il est essentiel que le plateau ne soit pas en mouvement quand on abaisse la sonde, et qu'il demeure à l'arrêt durant toute l'opération jusqu'à ce que la sonde soit complètement remontée sinon il peut y avoir bris de matériel.

Un test positif est concluant quand la sonde atteint la profondeur nominale pour la perforation. Mais un test négatif requiert que la condition ($\overline{X16}$,X17) ne se produise pas. Le délai de latence pour abaisser complètement la sonde du testeur est de 0.4 s. Il faut donc, lorsqu'on abaisse la sonde, attendre que la condition ($\overline{X16}$,X17) se produise (résultat positif), pendant 0.4 s avant de pouvoir conclure à un résultat négatif.

La station de manutention

La station de manutention ne se compose que d'une grue à flèche semi-circulaire et d'un glissoir servant à recueillir les pièces rejetées au test de contrôle de qualité d'usinage. La fonction de cette grue est de récupérer les pièces usinées dans le puit de la position de récupération du plateau d'usinage, pour les transférer soit au glissoir des rejets (si la pièce échoue le contrôle de qualité d'usinage), soit à l'admission de la station de tri. Le seul dispositif actif et contrôlé est la grue. Dans cette application, la forme semi-circulaire de la grue ne lui donne aucune propriété supplémentaire ; elle n'est utilisée que comme une grue à balancier.

`C2.Manutention.Grue` se compose de quatre dispositifs : la flèche, la pince (cette dernière constituée d'une ventouse pneumatique à vide), un pont et un treuil.

`C2.Manutention.Grue.Flèche = { (Y23,Y24) : (X21,X22) }`

Description : Dispositif semi-circulaire à deux positions extrêmes se comportant comme un vérin.

Commande : Bistable piloté.

Actionneurs : Discrets.

($\overline{Y23}$,Y24) : Tourner la flèche vers la station d'usinage.

(Y23,$\overline{Y24}$) : Tourner la flèche vers la station de tri.

Capteurs : Discret ; fin de course de la flèche.

($\overline{X21}$,X22) : Flèche vers la station d'usinage (position stable).

(X21,$\overline{X22}$) : Flèche vers la station de tri (position stable).

($\overline{X21}$,$\overline{X22}$) : Flèche en position intermédiaire indéterminée.

Notes :

L'opération de la flèche prend deux secondes complètes ce qui est assez long.
Pour des considérations de sécurité, il est préférable de ne bouger la flèche que lorsque le pont est rétracté et le treuil relevé.

C2.Manutention.Grue.Pince = { (Y25,Y26) : (X20) }

Description : Ventouse pneumatique à succion.

Commande : Bistable non piloté.

Actionneurs : Discret.

(Y25,$\overline{Y26}$) : Activer la succion.

($\overline{Y25}$,Y26) : Désactiver la succion.

Capteurs : Un capteur discret qui détermine si une pièce est en prise par la ventouse.

(X20) : La ventouse tient une pièce.

Notes :

À la position de récupération du plateau d'usinage, s'il n'y a pas de pièce dans le puit, la ventouse ne touche pas le fond du puit même lorsque le treuil est abaissé complètement.

C2.Manutention.Grue.Pont = { (Y21,Y22) : (X24,X23) }

Description : Le pont de cette grue lui donne deux positions pour allonger la portée de la flèche. Du côté station d'usinage, le pont n'a qu'une position valide : la position allongée qui l'amène au-dessus du puit d'évacuation du carrousel d'usinage. La position rétractée ne sert qu'à dégager le plateau d'usinage. Du côté station de tri, la position rétractée se situe au-dessus de la coulisse d'évacuation des rejets. C'est là que l'on doit déposer les pièces qui ne passent pas le test de profondeur nominale de la perforation

(contrôle de qualité de l'usinage). La position allongée se situe au-dessus du convoyeur du trieur pour les pièces qui ont passé le contrôle de qualité de l'usinage avec succès.

Commande : Vérin à commande bistable non piloté.

Actionneurs : Discrets.

(Y21,$\overline{\text{Y22}}$) : Rétracter le pont.

($\overline{\text{Y21}}$,Y22) : Allonger le pont.

Capteurs : Deux capteurs discrets de fin de course du vérin.

($\overline{\text{X23}}$,X24) : Pont rétracté.

(X23,$\overline{\text{X24}}$) : Pont allongé.

Notes :

Pendant un déplacement de la flèche, le pont doit être en position rétracté et le treuil en position relevé.

Lorsque l'on manipule le pont, le treuil doit être en position relevé.

C2.Manutention.Grue.Treuil = { (Y27) : (X25,X26) }

Description : Assure le mouvement vertical de la pince.

Commande : Vérin à commande monostable.

Actionneurs : Discret.

(Y27) : Abaisser la pince.

Capteurs : Deux capteurs discrets de fin de course du vérin.

(X25,$\overline{\text{X26}}$) : Treuil relevé (pince en haut).

($\overline{\text{X25}}$,X26) : Treuil abaissé (pince en bas).

Notes :

Attention, on ne peut pas se fier à la condition ($\overline{\text{X25}}$,X26) pour savoir quand il faut remonter le treuil lorsque l'on récupère une pièce au fond du puit en position récupération sur le plateau d'usinage. En effet, si une pièce se trouve dans le puit, le treuil de cette grue n'atteindra jamais sa fin de course (l'épaisseur de la pièce l'en empêche). Pire encore, l'atteinte de la condition ($\overline{\text{X25}}$,X26) signale qu'aucune pièce ne se trouve dans le puit ; une erreur. Pour savoir quand remonter le treuil, il faut détecter le moment où la pince (ventouse pneumatique) de la grue a saisi une pièce (voir la description de la pince) ; ce qui veut dire qu'il faut démarrer la succion de la pince avant d'abaisser le treuil.

Le treuil ne devrait être abaissé que lorsque l'on ne manipule ni le pont ni la flèche.

La station de tri

La station de tri, que l'on peut voir à la figure 45, se compose de deux dispositifs actifs : le convoyeur qui déplace les pièces admises à l'intérieur de l'aiguilleur et permet le tri, et l'aiguilleur qui permet la sélection d'un point de sortie parmi trois à l'aide d'un système de butées d'obstruction.

À la sortie du convoyeur se trouvent trois chutes (des glissoirs), une pour chaque sortie du convoyeur, permettant l'accumulation des pièces triées par catégorie. Ces chutes sont des dispositifs passifs non contrôlés de capacité 10 (approximativement) avec une politique de distribution FIFO numérotés de gauche à droite : `C3.tri.Chute1`, `C3.tri.Chute2` et `C3.tri.Chute3`. Il n'y a aucun dispositif contrôlé ni à l'entrée ni à la sortie de ces chutes, et leur principe d'opération est la gravité.

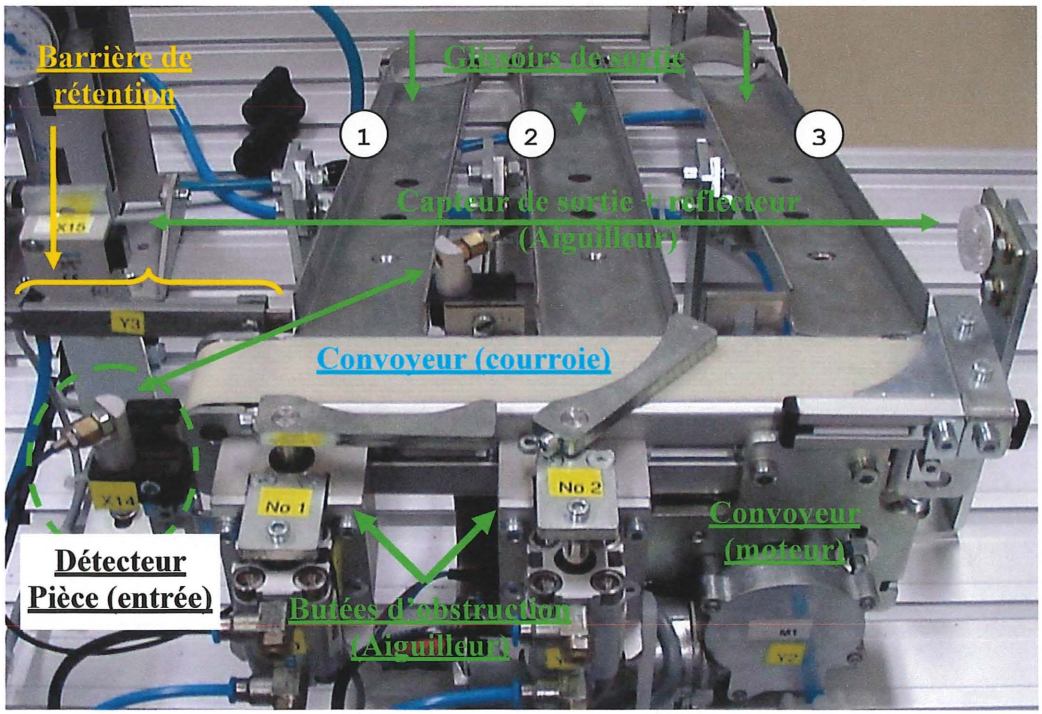

Figure 45. Station de tri de l'usine–école MPS

C3.Tri.Convoyeur = { (Y2) : (X14,X15) }

Description : Le convoyeur peut détecter lorsqu'une pièce se présente à son entrée et lorsqu'une pièce est présente en sortie.

Commande : Relais.

Actionneurs : Actionneur discret (un relais).

(Y2) : Démarrer la courroie (moteur) du convoyeur.

Capteurs : Deux capteurs optiques discrets à l'entrée et à la sortie du convoyeur.

($\overline{X14}$) : Une pièce est présente à l'entrée du convoyeur.

(X15) : Une pièce traverse la sortie du convoyeur.

Notes :

Puisque la chute que la pièce emprunte est déterminée par l'aiguilleur, il ne doit y avoir qu'une pièce à la fois en transit dans le convoyeur ; le capteur de sortie ne peut pas discriminer quelle chute est empruntée pour la sortie du convoyeur.

C3.Tri.Aiguilleur se compose de trois dispositifs actifs : deux butées d'obstruction pour la sélection d'une chute et une barrière de rétention à la chute n° 1.

C3.Tri.Aiguilleur.Butée1 = { (Y0) : (X10,X11) }

Description : Une butée d'obstruction peut être mise en travers de la courroie du convoyeur pour contraindre une pièce défilant sur la courroie à tomber dans la chute qui correspond à cette butée.

Commande : Vérin à commande monostable.

Actionneurs : Un actionneur discret.

(Y0) : Mettre la butée en état d'obstruction.

Capteurs : Deux capteurs discrets de fin de course sur le vérin.

(X10,$\overline{X11}$) : La butée n'est pas en état d'obstruction.

($\overline{X10}$,X11) : La butée est en état d'obstruction.

Notes :

Lorsque la butée n° 1 est en obstruction, le convoyeur sélectionne la chute n° 1 de la station de tri.

Lorsque la butée n° 1 est en obstruction, l'état de la butée n° 2 n'a pas d'effet, il y a donc préséance de la butée n° 1 sur la butée n° 2.

Lorsque la butée n° 1 n'est pas en obstruction, si la butée n° 2 est en obstruction, le convoyeur sélectionne la chute n° 2 de la station de tri.
Si aucune des deux butées n'est en obstruction, le convoyeur sélectionne la chute n° 3 de la station de tri.

`C3.Tri.Aiguilleur.Butée2 = { (Y1) : (X12,X13) }`

Description : Une butée d'obstruction peut être mise en travers de la courroie du convoyeur pour contraindre une pièce défilant sur la courroie à tomber dans la chute qui correspond à cette butée.

Commande : Vérin à commande monostable.

Actionneurs : Un actionneur discret.

(Y1) : Mettre la butée en état d'obstruction.

Capteurs : Deux capteurs discrets de fin de course sur le vérin.

(X12,$\overline{X13}$) : La butée n'est pas en état d'obstruction.

($\overline{X12}$,X13) : La butée est en état d'obstruction.

Notes :

Lorsque la butée n° 1 est en obstruction, le convoyeur sélectionne la chute n° 1 de la station de tri.
Lorsque la butée n° 1 est en obstruction, l'état de la butée n° 2 n'a pas d'effet, il y a donc précédence de la butée n° 1 sur la butée n° 2.
Lorsque la butée n° 1 n'est pas en obstruction, si la butée n° 2 est en obstruction, le convoyeur sélectionne la chute n° 2 de la station de tri.
Si aucune des deux butées n'est en obstruction, le convoyeur sélectionne la chute n° 3 de la station de tri.

`C3.Tri.Aiguilleur.Barrière = { (Y3) : (0.1 s, 0.1 s) }`

Description : Barrière d'obstruction à la chute n° 1. Cette barrière empêchera une pièce de s'engager dans la chute n° 1 peu importe l'état de la butée d'obstruction n°1 et du convoyeur. Sa fonction n'est pas très claire. Peut-être est-il préférable de la fermer quand la butée n° 1 n'est pas en état d'obstruction, pour être certain que la pièce mise sur le convoyeur s'engagera correctement sur la courroie car il y a une certaine instabilité à ce moment.

Commande : Vérin à commande monostable.

Actionneurs : Un actionneur discret.

(Y3) : Barrière en état d'obstruction.

Capteurs : Aucun capteur.

Notes :

Puisqu'il n'y a aucun capteur de fin de course sur le vérin, il pourrait être nécessaire d'utiliser des temporisations. Les délais d'ouverture et de fermeture sont cependant très courts (de l'ordre de moins d'un dixième de seconde).

La grue de réinsertion

La grue de réinsertion, que l'on peut voir à la figure 46, se compose de cinq dispositifs : une pince, un treuil, un pont, une flèche et un dispositif d'arrêt d'urgence manuel. La fonction de cette grue est de reprendre les pièces livrées dans les puits à la sortie de chaque chute de la station de tri, pour les réintroduire dans le silo de la station de livraison, fermant ainsi la boucle. Cette manœuvre a pour but de permettre un fonctionnement continu de l'usine-école MPS.

C3.Réinsertion.Grue.Arrêt = { (*<Bouton d'arrêt d'urgence manuel>*) : (X26,X27) }

Description : Ce dispositif manuel est rendu nécessaire parce que la grue (à cause de sa portée) peut entrer en collision avec plusieurs des dispositifs des autres stations du MPS. De plus, le servomoteur est très puissant et offre des caractéristiques d'accélération et de freinage qui le rendent dangereux pendant l'ajustement et la programmation du contrôleur.

Commande : Bouton à deux positions d'opération manuelle.

Actionneurs : Un actionneur à opération manuelle (pas sous un contrôle programmé).

Bouton en position retirée : Grue en arrêt d'urgence.
Bouton en position enfoncée : Grue sous un contrôle programmé.

Capteurs : Deux capteurs discrets détectant l'état du bouton.

$(X26, \overline{X27})$: Grue en état d'arrêt d'urgence.

$(\overline{X26}, X27)$: Grue en fonction (sous un contrôle programmé).

Notes :

On peut voir sur la figure 47 l'opération du bouton d'arrêt d'urgence de la grue de réinsertion. Sur la figure, la grue est en fonction et l'opérateur s'apprête à enfoncer le bouton pour la mettre en arrêt d'urgence.

Ce dispositif n'a pas d'effet sur l'état de la station de tri qui demeure opérationnelle.

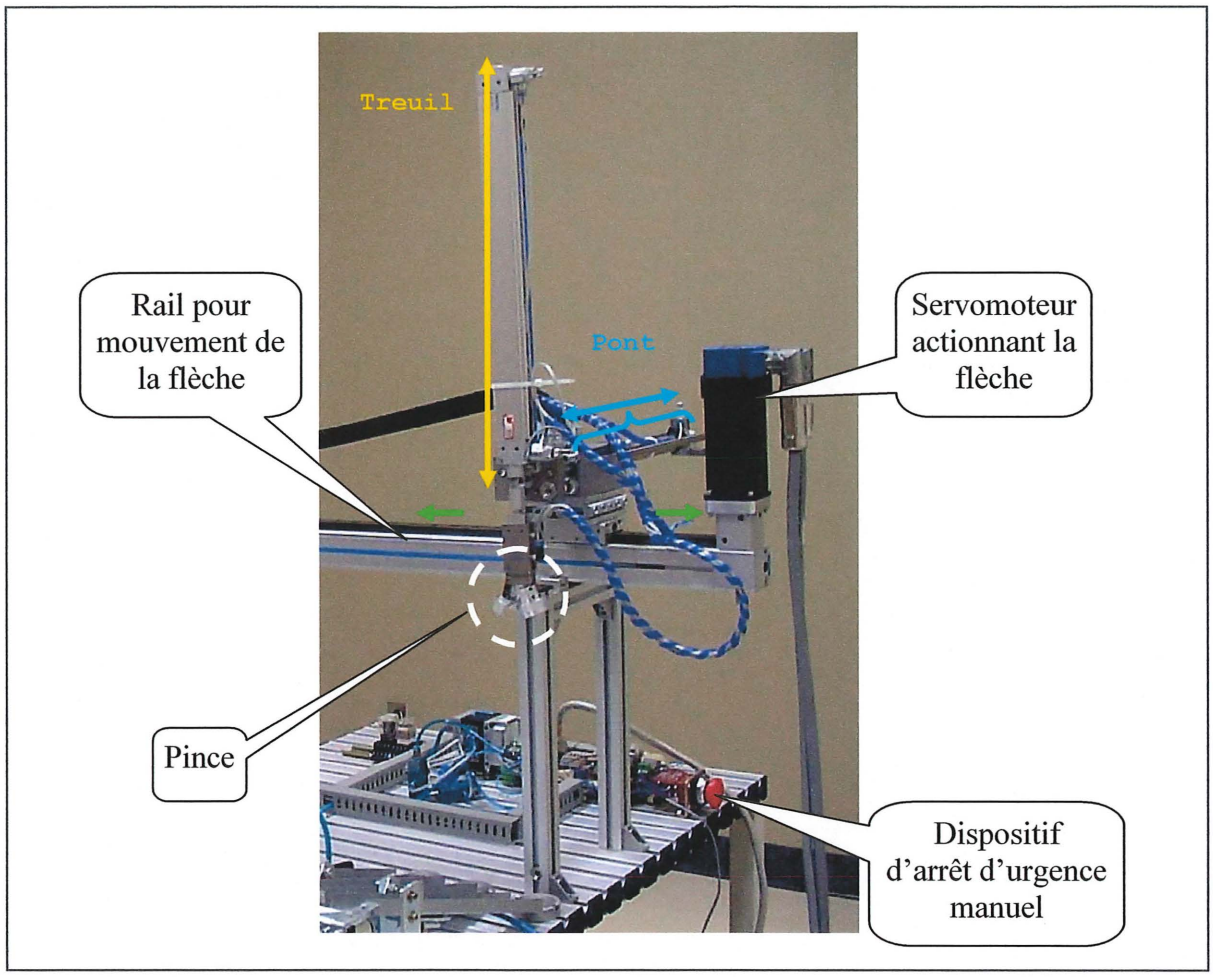

Figure 46. Grue de réinsertion de l'usine-école MPS

`C3.Réinsertion.Grue.Pince` = { `(Y4,Y5) : (X20,X21)` }

Description : Outil de préhension de la grue.

Commande : Équivalent à un vérin à commande bistable non piloté.

Actionneurs : Deux actionneurs discrets.

($Y4, \overline{Y5}$) : Fermer la pince.

($\overline{Y4}, Y5$) : Ouvrir la pince.

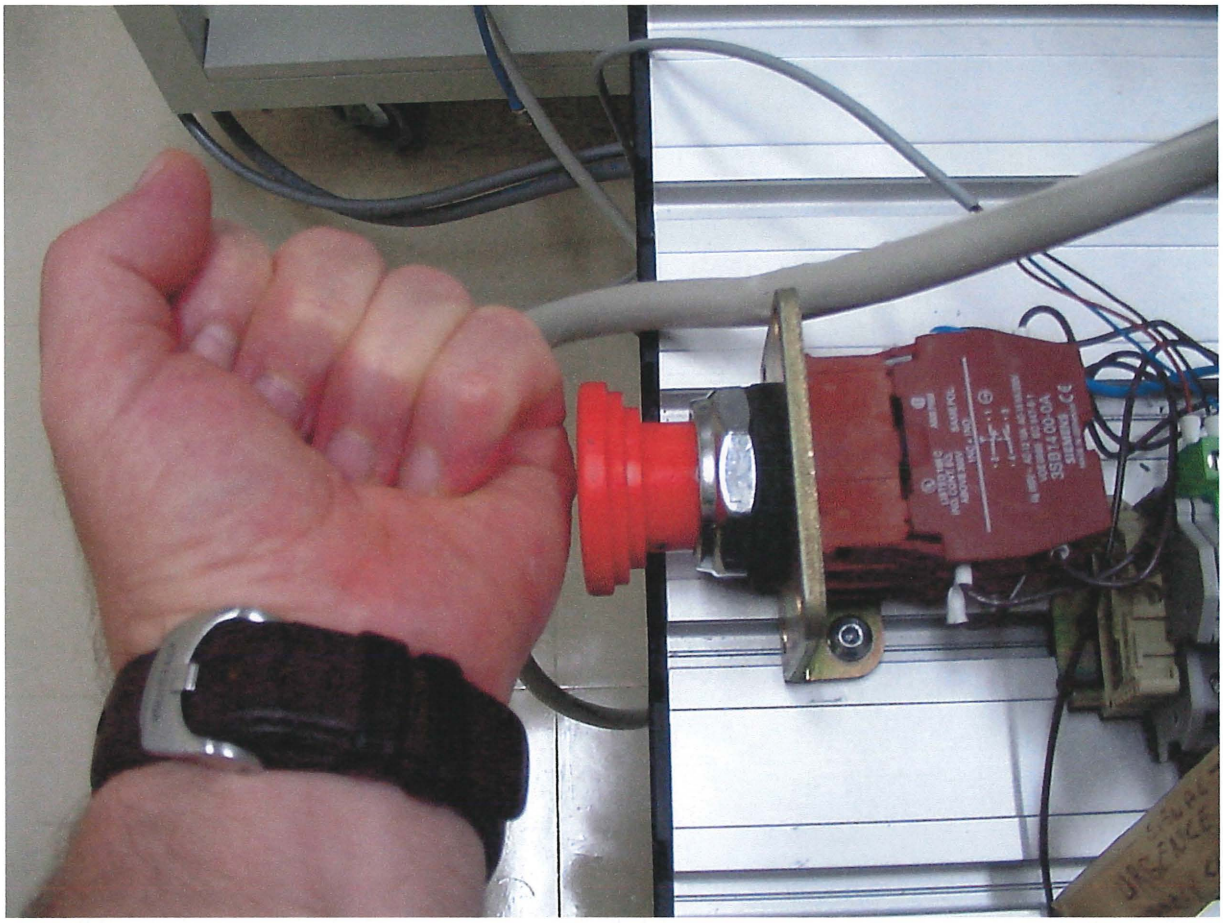

Figure 47. Opération du bouton d'arrêt d'urgence de la grue de réinsertion

Capteurs : Deux capteurs discrets de fin de course.

($X20, \overline{X21}$) : Pince fermée.

($\overline{X20}, X21$) : Pince ouverte.

Notes :

Aucune note particulière.

C3.Réinsertion.Grue.Treuil = { (Y6) : (X22,X23) }

Description : Le mouvement vertical de la pince est assuré par un treuil.

Commande : Vérin à commande monostable en position normale rétractée.

Actionneurs : Un actionneur discret.

(Y6) : Abaisser le treuil.

Capteurs : Deux capteurs de fin de course du vérin.

($X22,\overline{X23}$) : Treuil relevé.

($\overline{X22}$,X23) : Treuil abaissé.

Notes :

Le treuil ne devrait sous aucun prétexte être mis en opération lorsque la flèche est en mouvement. Lorsque la flèche est en mouvement, le treuil doit être relevé.

C3.Réinsertion.Grue.Pont = { (Y7) : (X24,X25) }

Description : Le pont étend la portée de la grue vers l'avant. Cette extension de portée est nécessaire pour rejoindre le silo de chargement de la station de livraison, mais il n'est pas nécessaire lorsque la grue se trouve aux puits de la station de tri.

Commande : Vérin à commande monostable en position normale rétractée.

Actionneurs : Un actionneur discret.

(Y7) : Avancer le pont.

Capteurs :

($X24,\overline{X25}$) : Pont reculé.

($\overline{X24}$,X25) : Pont avancé.

Notes :

Le pont ne devrait sous aucun prétexte être en mouvement pendant que la flèche est en déplacement. Lorsque la flèche est en déplacement, le pont devrait être en position rétractée et y rester tant que dure le mouvement de la flèche.

C3.Réinsertion.Grue.Flèche = { (Y13,VY20) }

Description : La flèche imprime à la grue entière un mouvement latéral qui s'étend de la zone des puits de la station de tri au silo de la station de livraison.

Commande : Servomoteur positionné par niveau de voltage.

Actionneurs : Un actionneur discret pour déclencher la calibration du servomoteur et un registre de 16 bits pour recevoir la commande de positionnement du servomoteur.

(Y13) : Déclenche la séquence de calibration du servomoteur.

On doit mettre une commande valide de positionnement de la flèche au point de voltage « 0 » avant de mettre la grue en fonction. Il faut ensuite placer la grue manuellement en position excentrée vers la droite (à droite du centre physique du rail du servomoteur). Finalement, on sort la grue d'arrêt d'urgence (mise en

fonction). Il faut ensuite mettre `Y13` à « 1 » et maintenir cet actionneur à 1 pour toute la durée de la séquence de calibration qui peut prendre une quinzaine de secondes.

En opération normale, pour pouvoir placer des commandes de mouvement à la flèche, cet actionneur doit être réglé à $\overline{\texttt{Y13}}$. On ne peut se servir de façon sûre de cette fonction (calibration) qu'en sortant d'un arrêt d'urgence, pour re-calibrer il faut refaire un arrêt d'urgence de la grue.

`(VY20)` : Registre qui reçoit dans les bits 0 à 11 (convention *little endian*) la valeur absolue du niveau de voltage (à l'échelle) correspondant à la position de la flèche que l'on veut obtenir. L'échelle du registre couvre de -2 047 à +2 047 unité pour une plage de voltage de -10 V à +10 V respectivement, la valeur absolue ne doit pas dépasser 2 047 (sinon il y a roulement du registre 2 048 = 0 et 2 049 = 1 et ainsi de suite). Le bit 12 doit être mis à zéro. Les bits 13 et 14 contiennent l'adresse du canal analogique sur lequel doit se faire la sortie de voltage, pour nous il s'agit du canal n° 1, ce qui veut dire que le bit 13 doit être mis à 1 et le bit 14 à 0. Finalement, le bit 15 est le bit de signe ; s'il est à 1, la valeur absolue désignée par les bits 0 à 11 du registre est interprétée comme une valeur négative, s'il est à 0 c'est une valeur positive.

Capteurs : Aucun, c'est ce qui rend l'opération de la flèche hasardeuse.

Notes :

Le nombre $27F6_{16}$ correspond à la position du silo de la station de livraison.

Le nombre $A55B_{16}$ correspond au puit de la chute n° 1 (le puits le plus intérieur) de la station de tri.

Le nombre $A605_{16}$ correspond au puit de la chute n° 2 (au milieu) de la station de tri.

Le nombre $A6D4_{16}$ correspond au puit de la chute n° 3 (le puits le plus extérieur) de la station de tri.

Annexe 2

Spécification fonctionnelle de l'usine-école MPS par grafcets

Cellule C1, station de livraison et station de test.

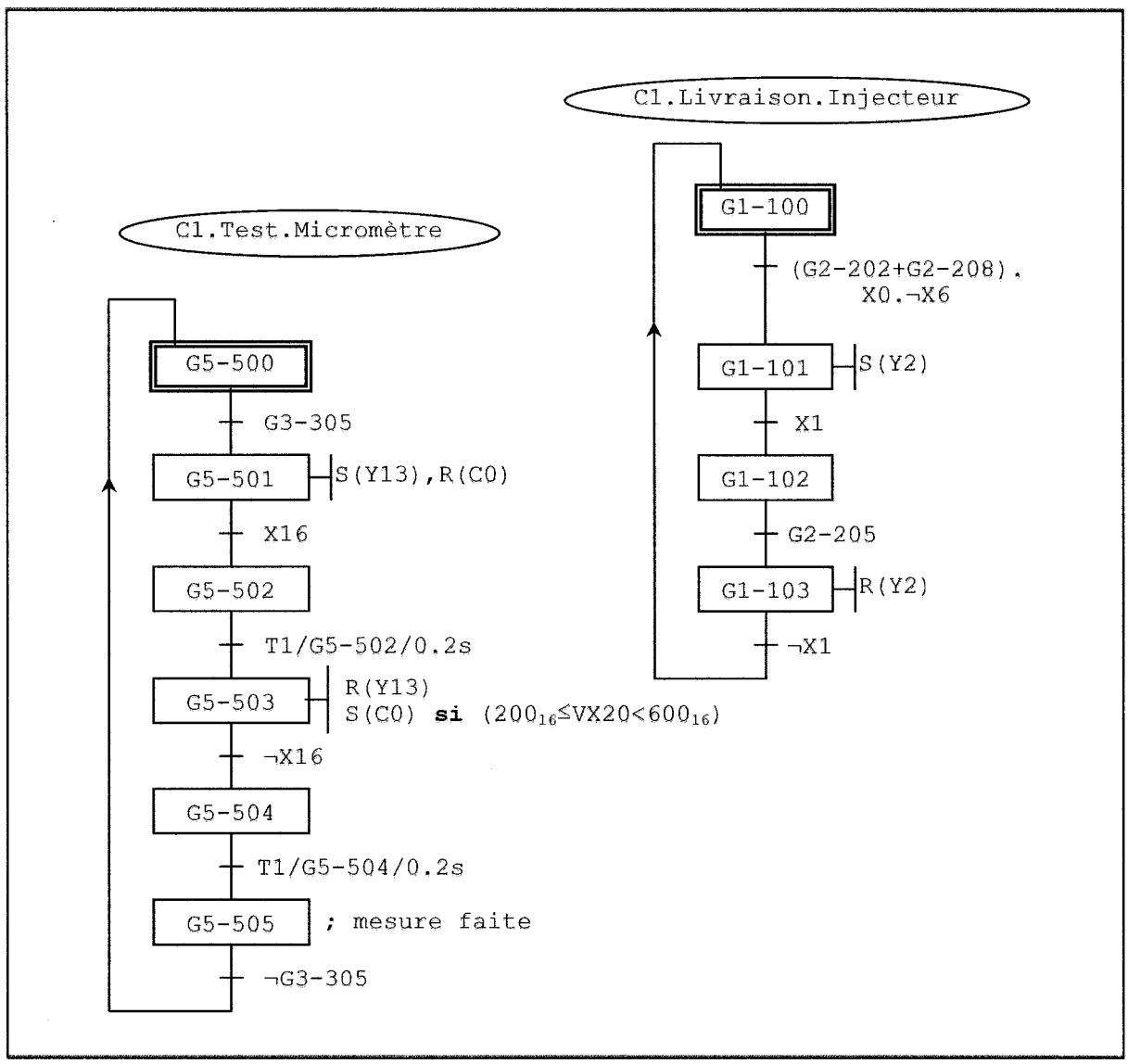

Figure 48. Grafcets du micromètre et de l'injecteur

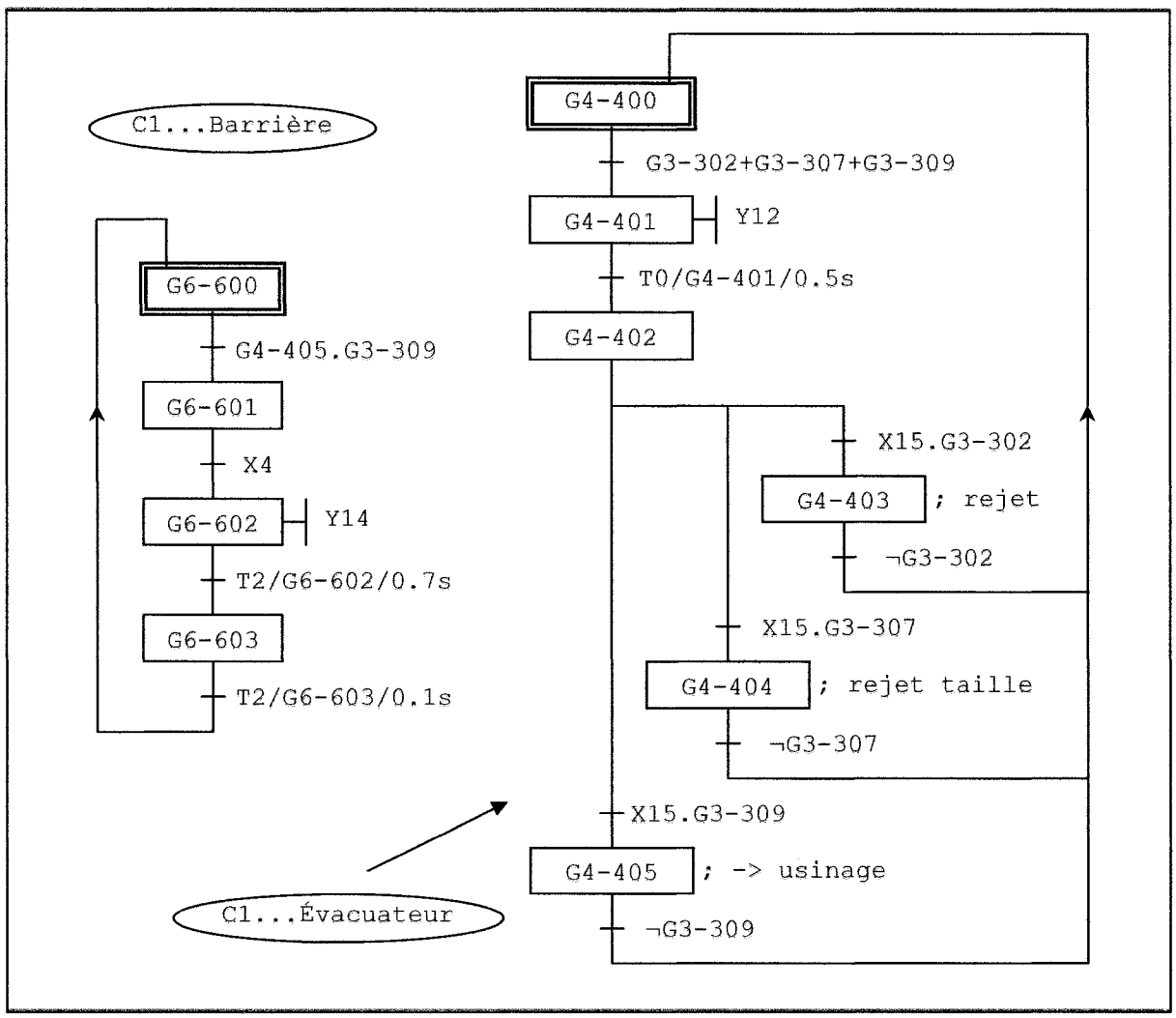

Figure 49. Grafcets de la barrière et de l'évacuateur

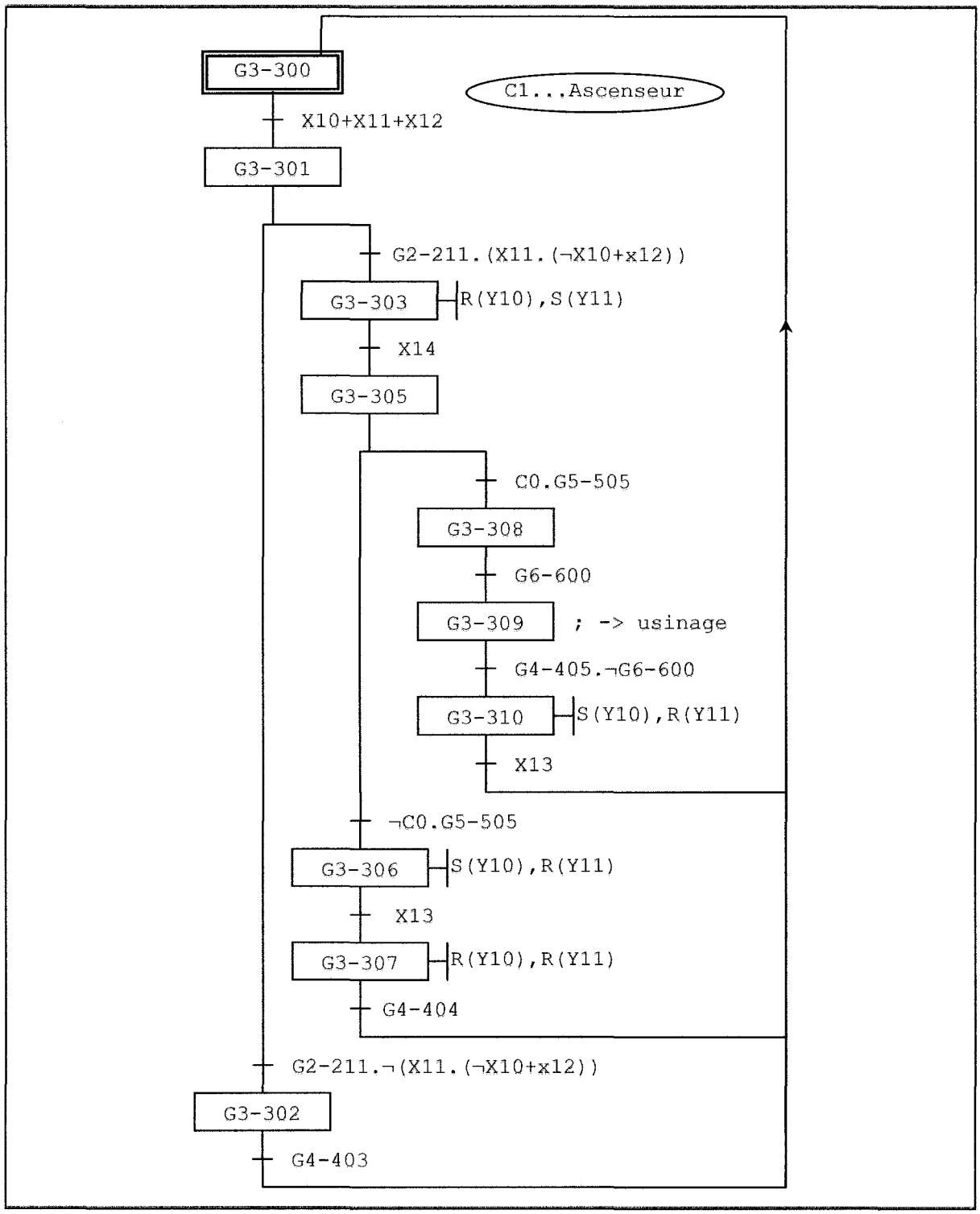

Figure 50. Grafcet de l'ascenseur

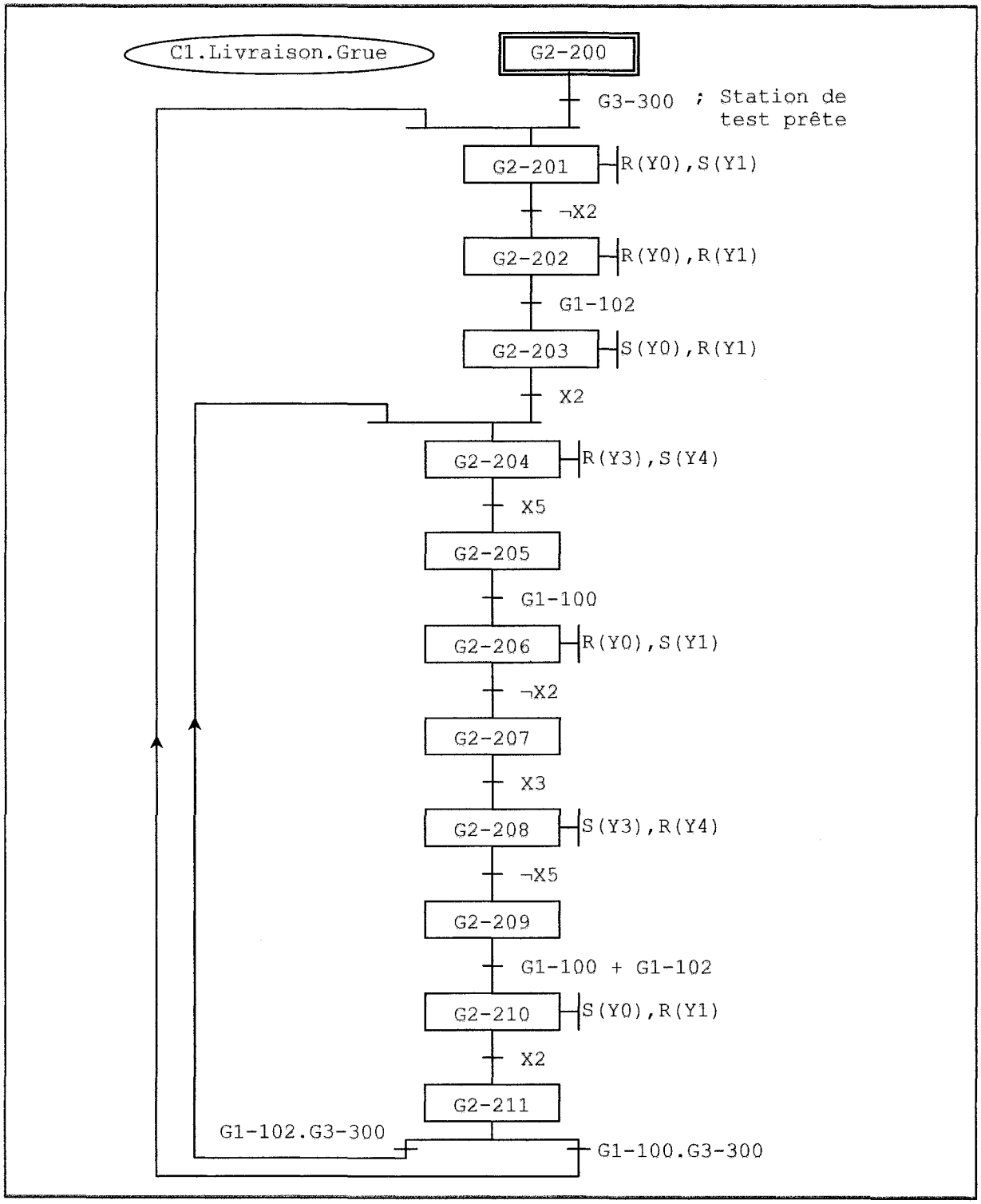

Figure 51. Grafcet de la grue

Cellule C2, station d'usinage et station de manutention.

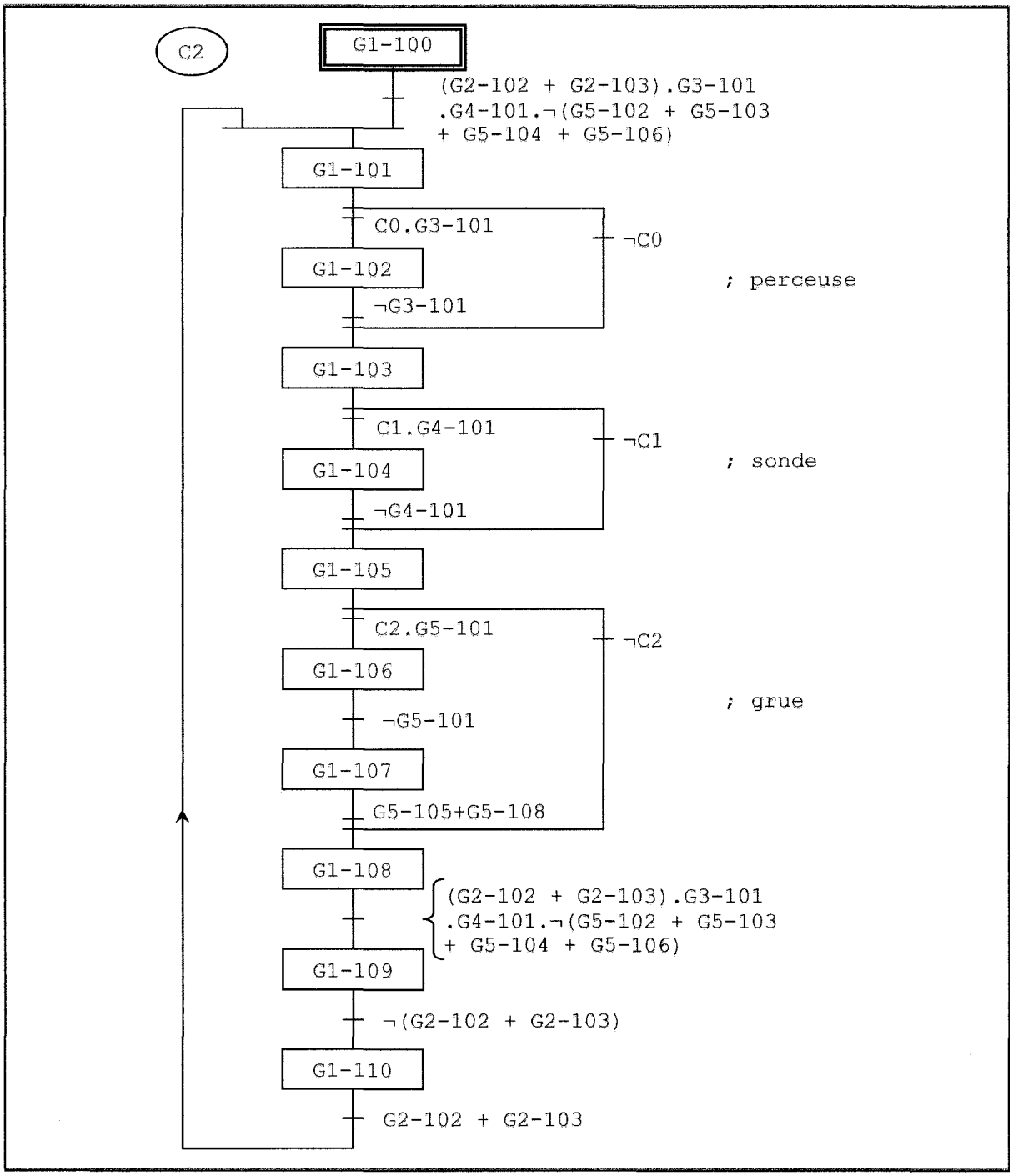

Figure 52. Grafcet coordonnateur de la cellule C2

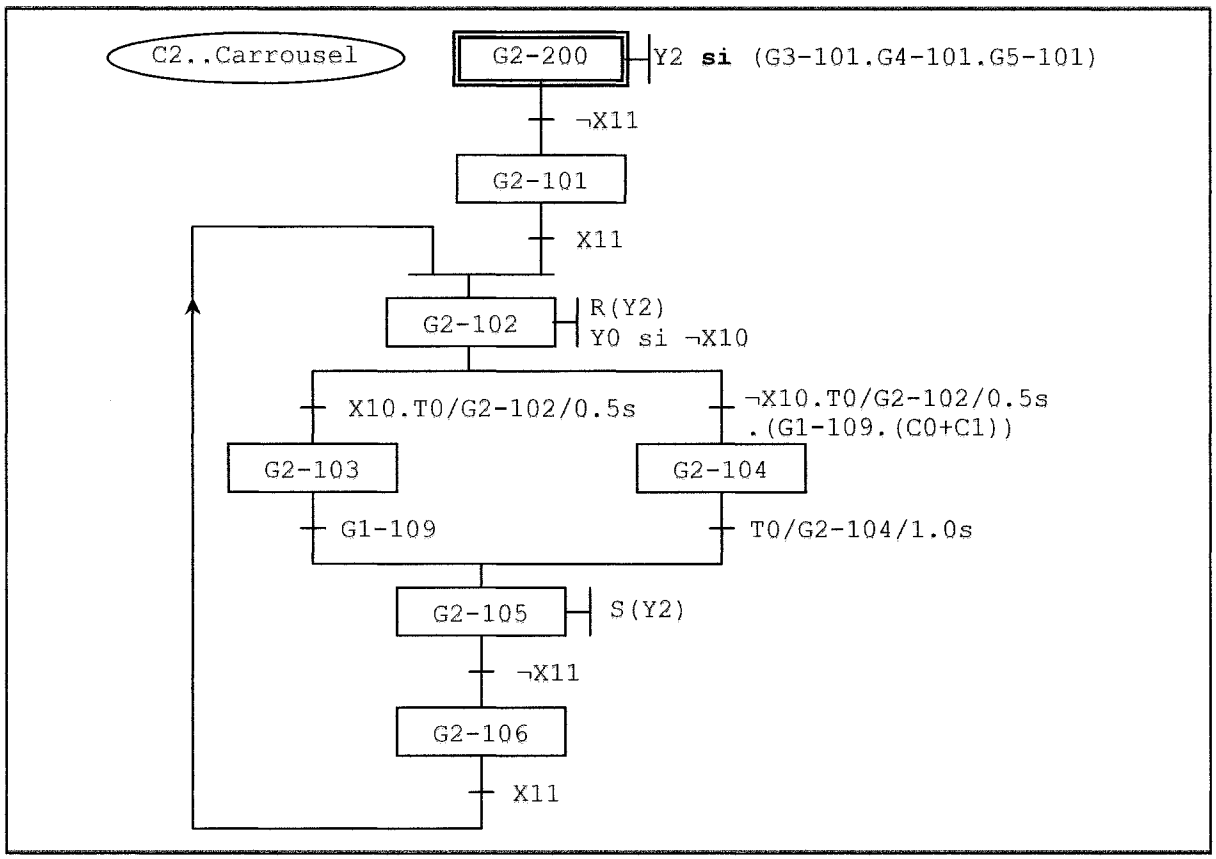

Figure 53. Grafcet du carrousel

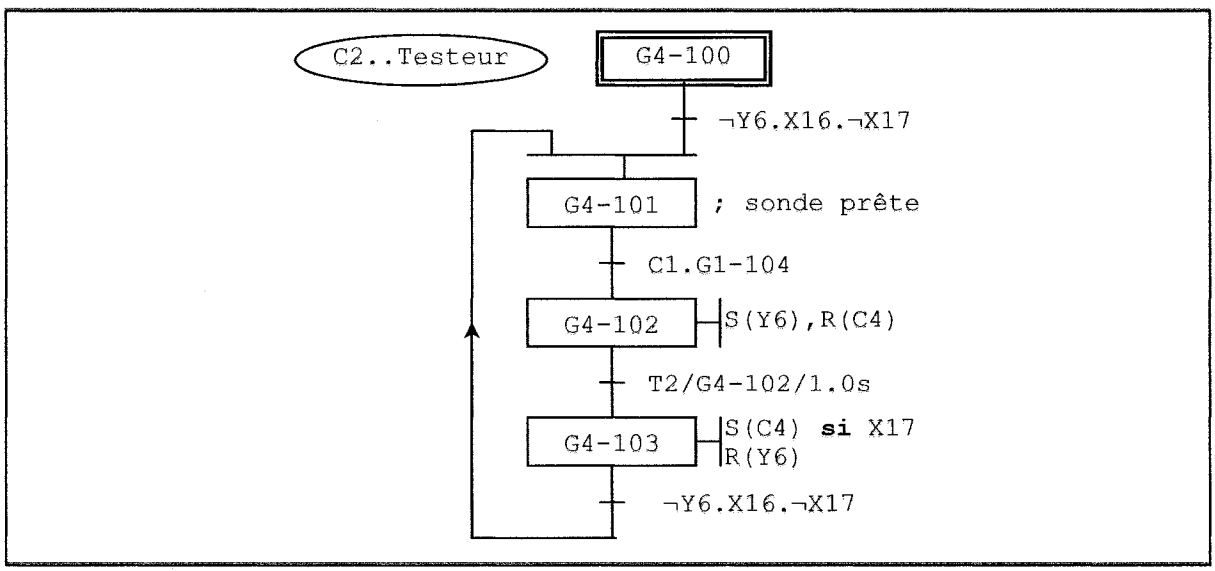

Figure 54. Grafcet du testeur

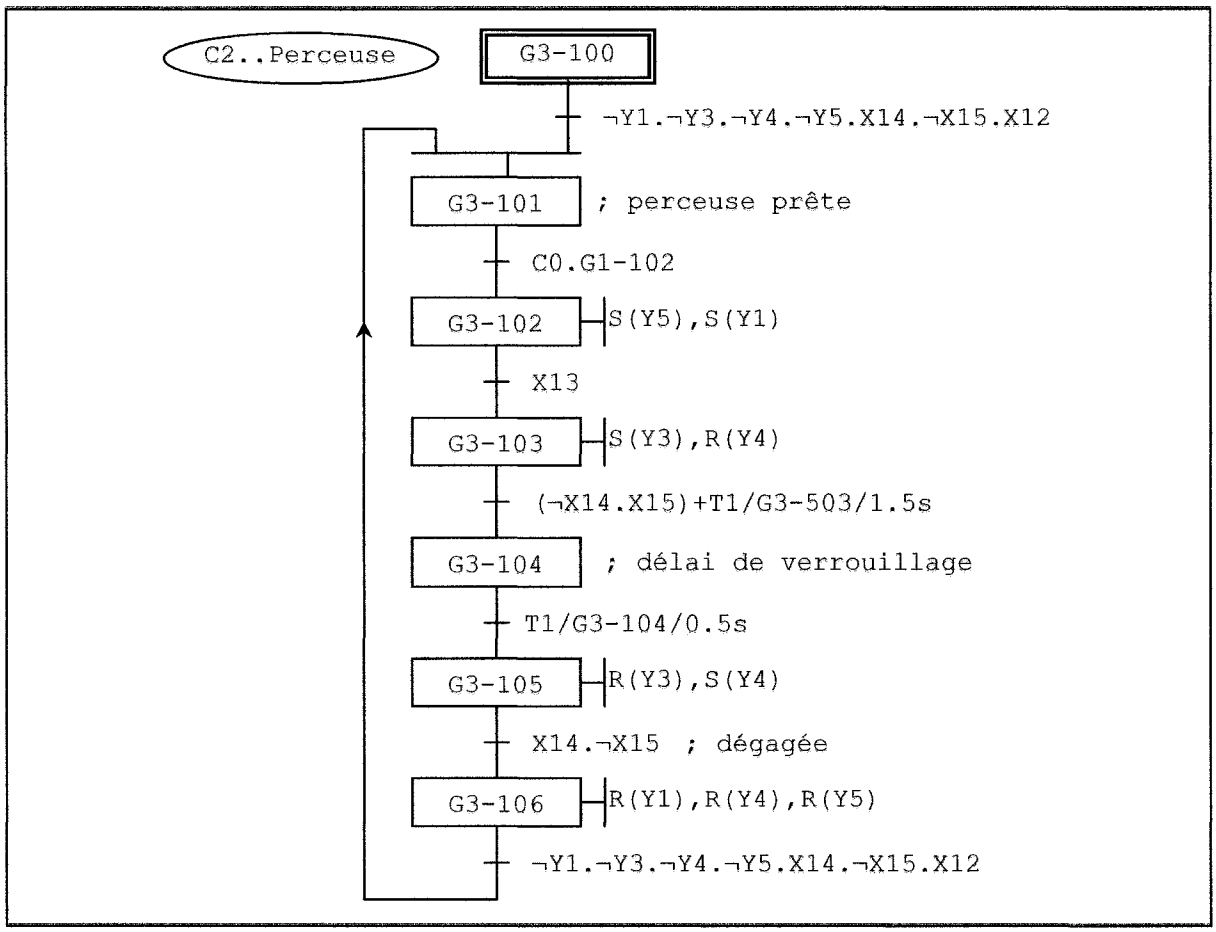

Figure 55. Grafcet de la perceuse

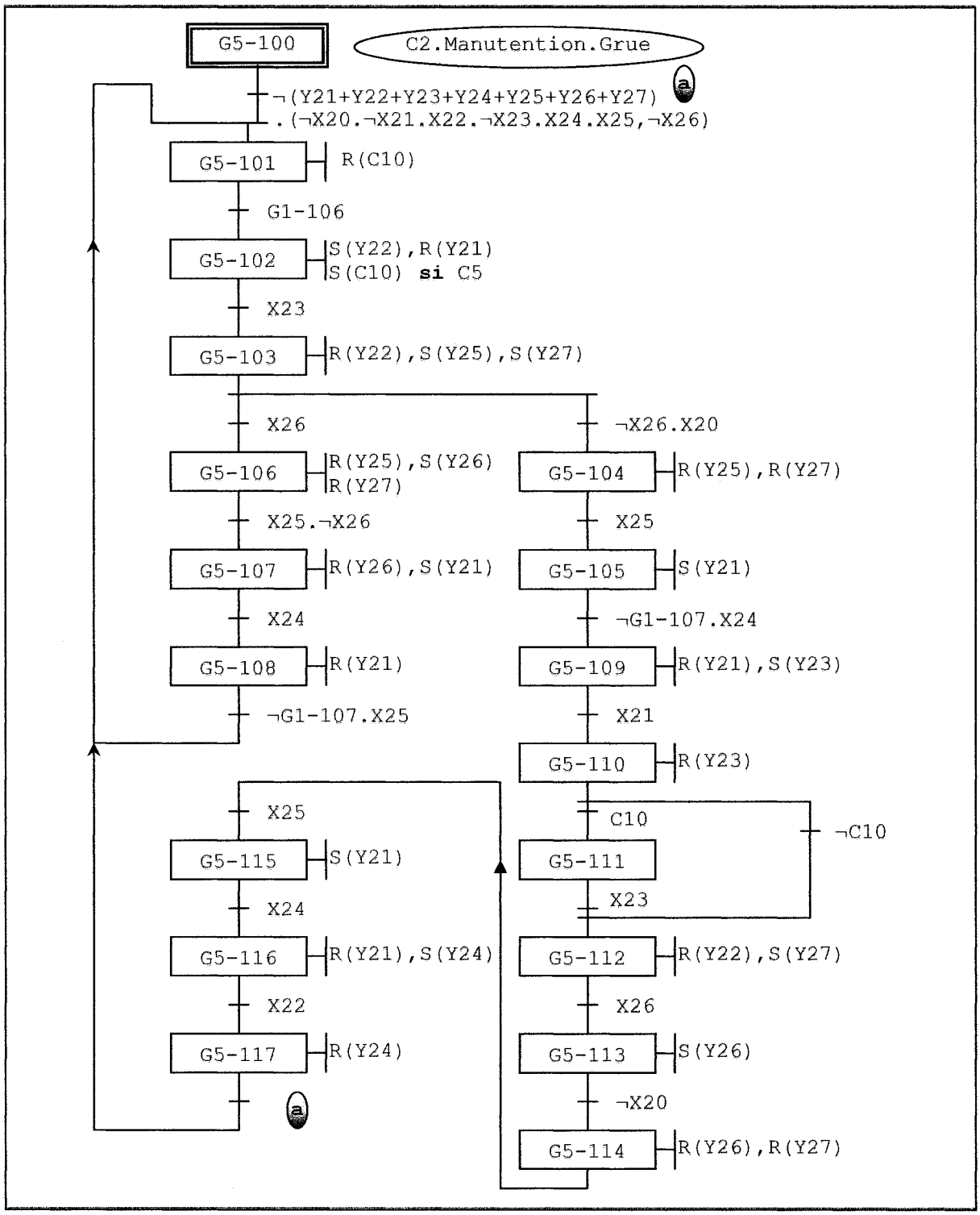

Figure 56. Grafcet de la grue de manutention

Cellule C3, station de tri et grue de réinsertion.

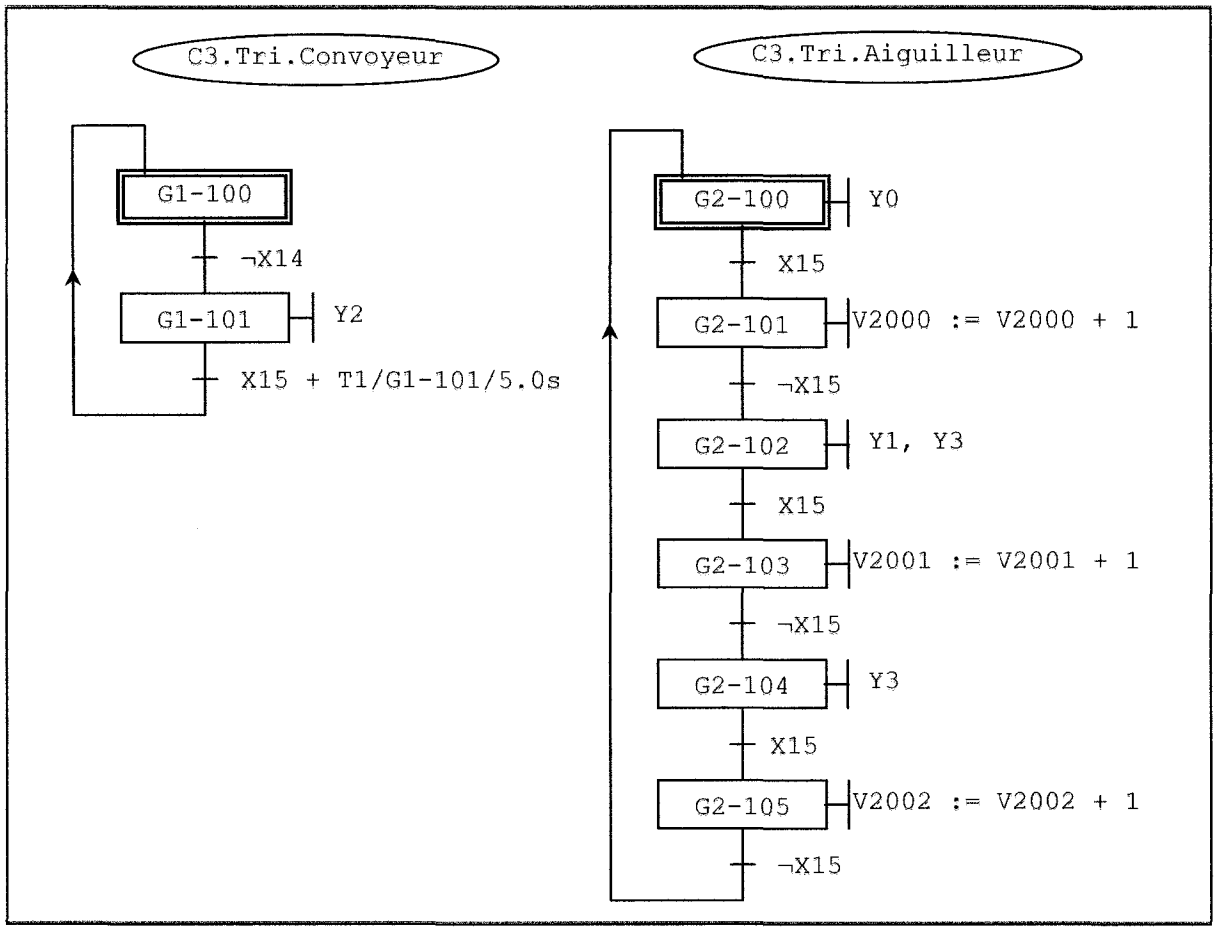

Figure 57. Grafcet du convoyeur et de l'aiguilleur

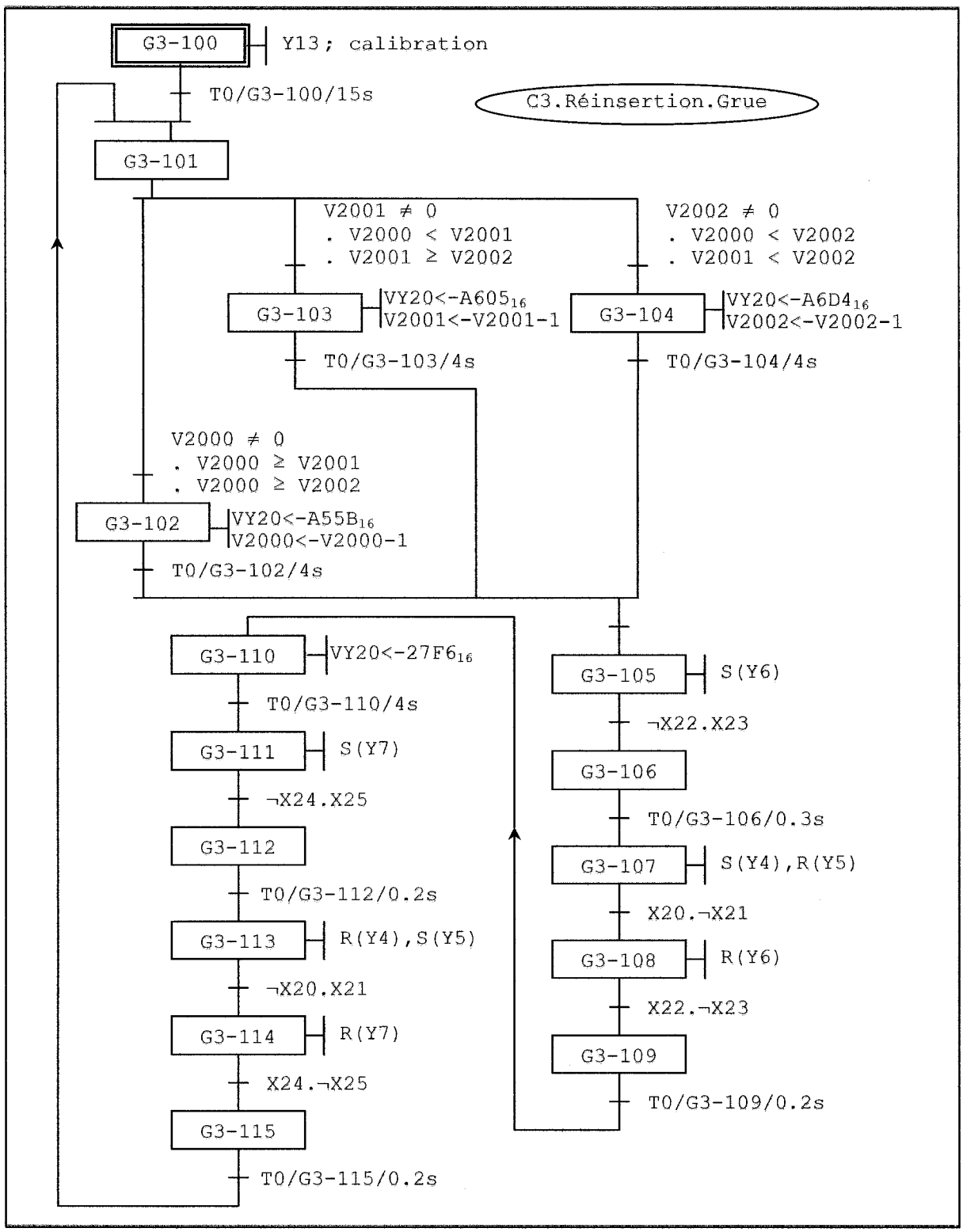

Figure 58. Grafcet de la grue de réinsertion

Annexe 3

Contrôleur de la station de livraison (théorie du contrôle des SED)

Liste de l'AFD du contrôleur obtenu avec l'ensemble des spécifications contenues dans la section 5.2.1, d'après la méthode décrite (`Livraison.sup` de SUCSEDES).

```
AUTOMATON
LivraisonLibre
EVENTS: 14
upY2 dnY2 upX1 upX0 upX6
dnX6 right left upX3 upX2
grab rls upX5 dnX5
END
CONTROLLABLE_EVENTS:
upY2 dnY2 right left grab
rls
END
STATES: 52
( 1 1 1 1 1 1 1 1 1 1 ) ( 1 2 1 1 2 5 1 4 1 1 ) ( 1 2 2 1 2 5 1 4 2 1 )
( 1 1 2 1 1 1 1 1 2 1 ) ( 1 2 3 1 2 5 1 4 3 3 ) ( 1 1 3 1 1 1 1 1 3 3 )
( 2 2 3 1 2 6 1 5 3 3 ) ( 1 1 4 1 1 1 1 1 2 1 ) ( 3 2 3 1 2 7 1 6 3 3 )
( 2 1 3 1 1 2 1 2 3 3 ) ( 1 2 4 1 2 5 1 4 2 1 ) ( 3 1 3 1 1 3 1 3 3 3 )
( 3 2 4 1 2 7 1 6 2 1 ) ( 3 1 4 1 1 3 1 3 2 1 ) ( 3 2 1 1 2 7 1 6 1 1 )
( 3 1 1 1 1 3 1 3 1 1 ) ( 3 2 1 2 2 7 2 6 1 1 ) ( 3 1 1 2 1 3 2 3 1 1 )
( 3 2 1 3 2 7 3 6 1 2 ) ( 3 1 1 3 1 3 3 3 1 2 ) ( 4 2 1 3 2 8 3 5 1 2 )
( 4 1 1 3 1 4 3 2 1 2 ) ( 1 2 1 3 2 5 3 4 1 2 ) ( 4 2 2 3 2 8 3 5 2 2 )
( 1 1 1 3 1 1 3 1 1 2 ) ( 4 1 2 3 1 4 3 2 2 2 ) ( 1 2 2 3 2 5 3 4 2 2 )
( 4 2 3 3 2 8 3 5 3 3 ) ( 1 1 2 3 1 1 3 1 2 2 ) ( 4 1 3 3 1 4 3 2 3 3 )
( 1 2 3 3 2 5 3 4 3 3 ) ( 4 2 3 4 2 8 3 5 3 3 ) ( 1 1 3 3 1 1 3 1 3 3 )
( 4 1 3 4 1 4 3 2 3 3 ) ( 2 2 3 3 2 6 3 5 3 3 ) ( 1 2 3 4 2 5 3 4 3 3 )
( 4 2 3 1 2 8 1 5 3 3 ) ( 1 1 3 4 1 1 3 1 3 3 ) ( 4 1 3 1 1 4 1 2 3 3 )
( 3 2 3 3 2 7 3 6 3 3 ) ( 2 1 3 3 1 2 3 2 3 3 ) ( 2 2 3 4 2 6 3 5 3 3 )
( 4 2 4 1 2 8 1 5 2 1 ) ( 4 1 4 1 1 4 1 2 2 1 ) ( 3 1 3 3 1 3 3 3 3 3 )
( 3 2 3 4 2 7 3 6 3 3 ) ( 2 1 3 4 1 2 3 2 3 3 ) ( 4 2 1 1 2 8 1 5 1 1 )
( 4 1 1 1 1 4 1 2 1 1 ) ( 3 1 3 4 1 3 3 3 3 3 ) ( 4 2 2 1 2 8 1 5 2 1 )
( 4 1 2 1 1 4 1 2 2 1 )
END
MARKED_STATES:

END
INITIAL_STATE:
( 1 1 1 1 1 1 1 1 1 1 )
TRANSITIONS: 123
( 1 1 1 1 1 1 1 1 1 1 ) : dnX6 : ( 1 2 1 1 2 5 1 4 1 1 )
( 1 2 1 1 2 5 1 4 1 1 ) : upX6 : ( 1 1 1 1 1 1 1 1 1 1 )
( 1 2 1 1 2 5 1 4 1 1 ) : right : ( 1 2 2 1 2 5 1 4 2 1 )
( 1 2 2 1 2 5 1 4 2 1 ) : upX6 : ( 1 1 2 1 1 1 1 1 2 1 )
( 1 2 2 1 2 5 1 4 2 1 ) : upX3 : ( 1 2 3 1 2 5 1 4 3 3 )
```

```
( 1 1 2 1 1 1 1 1 2 1 ) : dnX6 : ( 1 2 2 1 2 5 1 4 2 1 )
( 1 1 2 1 1 1 1 1 2 1 ) : upX3 : ( 1 1 3 1 1 1 1 1 3 3 )
( 1 2 3 1 2 5 1 4 3 3 ) : upY2 : ( 2 2 3 1 2 6 1 5 3 3 )
( 1 2 3 1 2 5 1 4 3 3 ) : upX6 : ( 1 1 3 1 1 1 1 1 3 3 )
( 1 1 3 1 1 1 1 1 3 3 ) : dnX6 : ( 1 2 3 1 2 5 1 4 3 3 )
( 1 1 3 1 1 1 1 1 3 3 ) : left : ( 1 1 4 1 1 1 1 1 2 1 )
( 2 2 3 1 2 6 1 5 3 3 ) : upX1 : ( 3 2 3 1 2 7 1 6 3 3 )
( 2 2 3 1 2 6 1 5 3 3 ) : upX6 : ( 2 1 3 1 1 2 1 2 3 3 )
( 1 1 4 1 1 1 1 1 2 1 ) : dnX6 : ( 1 2 4 1 2 5 1 4 2 1 )
( 1 1 4 1 1 1 1 1 2 1 ) : upX2 : ( 1 1 1 1 1 1 1 1 1 1 )
( 3 2 3 1 2 7 1 6 3 3 ) : upX6 : ( 3 1 3 1 1 3 1 3 3 3 )
( 3 2 3 1 2 7 1 6 3 3 ) : left : ( 3 2 4 1 2 7 1 6 2 1 )
( 2 1 3 1 1 2 1 2 3 3 ) : upX1 : ( 3 1 3 1 1 3 1 3 3 3 )
( 2 1 3 1 1 2 1 2 3 3 ) : dnX6 : ( 2 2 3 1 2 6 1 5 3 3 )
( 1 2 4 1 2 5 1 4 2 1 ) : upX6 : ( 1 1 4 1 1 1 1 1 2 1 )
( 1 2 4 1 2 5 1 4 2 1 ) : upX2 : ( 1 2 1 1 2 5 1 4 1 1 )
( 3 1 3 1 1 3 1 3 3 3 ) : dnX6 : ( 3 2 3 1 2 7 1 6 3 3 )
( 3 1 3 1 1 3 1 3 3 3 ) : left : ( 3 1 4 1 1 3 1 3 2 1 )
( 3 2 4 1 2 7 1 6 2 1 ) : upX6 : ( 3 1 4 1 1 3 1 3 2 1 )
( 3 2 4 1 2 7 1 6 2 1 ) : upX2 : ( 3 2 1 1 2 7 1 6 1 1 )
( 3 1 4 1 1 3 1 3 2 1 ) : dnX6 : ( 3 2 4 1 2 7 1 6 2 1 )
( 3 1 4 1 1 3 1 3 2 1 ) : upX2 : ( 3 1 1 1 1 3 1 3 1 1 )
( 3 2 1 1 2 7 1 6 1 1 ) : upX6 : ( 3 1 1 1 1 3 1 3 1 1 )
( 3 2 1 1 2 7 1 6 1 1 ) : grab : ( 3 2 1 2 2 7 2 6 1 1 )
( 3 1 1 1 1 3 1 3 1 1 ) : dnX6 : ( 3 2 1 1 2 7 1 6 1 1 )
( 3 1 1 1 1 3 1 3 1 1 ) : grab : ( 3 1 1 2 1 3 2 3 1 1 )
( 3 2 1 2 2 7 2 6 1 1 ) : upX6 : ( 3 1 1 2 1 3 2 3 1 1 )
( 3 2 1 2 2 7 2 6 1 1 ) : upX5 : ( 3 2 1 3 2 7 3 6 1 2 )
( 3 1 1 2 1 3 2 3 1 1 ) : dnX6 : ( 3 2 1 2 2 7 2 6 1 1 )
( 3 1 1 2 1 3 2 3 1 1 ) : upX5 : ( 3 1 1 3 1 3 3 3 1 2 )
( 3 2 1 3 2 7 3 6 1 2 ) : dnY2 : ( 4 2 1 3 2 8 3 5 1 2 )
( 3 2 1 3 2 7 3 6 1 2 ) : upX6 : ( 3 1 1 3 1 3 3 3 1 2 )
( 3 1 1 3 1 3 3 3 1 2 ) : dnY2 : ( 4 1 1 3 1 4 3 2 1 2 )
( 3 1 1 3 1 3 3 3 1 2 ) : dnX6 : ( 3 2 1 3 2 7 3 6 1 2 )
( 4 2 1 3 2 8 3 5 1 2 ) : upX0 : ( 1 2 1 3 2 5 3 4 1 2 )
( 4 2 1 3 2 8 3 5 1 2 ) : upX6 : ( 4 1 1 3 1 4 3 2 1 2 )
( 4 2 1 3 2 8 3 5 1 2 ) : right : ( 4 2 2 3 2 8 3 5 2 2 )
( 4 1 1 3 1 4 3 2 1 2 ) : upX0 : ( 1 1 1 3 1 1 3 1 1 2 )
( 4 1 1 3 1 4 3 2 1 2 ) : dnX6 : ( 4 2 1 3 2 8 3 5 1 2 )
( 4 1 1 3 1 4 3 2 1 2 ) : right : ( 4 1 2 3 1 4 3 2 2 2 )
( 1 2 1 3 2 5 3 4 1 2 ) : upX6 : ( 1 1 1 3 1 1 3 1 1 2 )
( 1 2 1 3 2 5 3 4 1 2 ) : right : ( 1 2 2 3 2 5 3 4 2 2 )
( 4 2 2 3 2 8 3 5 2 2 ) : upX0 : ( 1 2 2 3 2 5 3 4 2 2 )
( 4 2 2 3 2 8 3 5 2 2 ) : upX6 : ( 4 1 2 3 1 4 3 2 2 2 )
( 4 2 2 3 2 8 3 5 2 2 ) : upX3 : ( 4 2 3 3 2 8 3 5 3 3 )
( 1 1 1 3 1 1 3 1 1 2 ) : dnX6 : ( 1 2 1 3 2 5 3 4 1 2 )
( 4 1 2 3 1 4 3 2 2 2 ) : upX0 : ( 1 1 2 3 1 1 3 1 2 2 )
( 4 1 2 3 1 4 3 2 2 2 ) : dnX6 : ( 4 2 2 3 2 8 3 5 2 2 )
( 4 1 2 3 1 4 3 2 2 2 ) : upX3 : ( 4 1 3 3 1 4 3 2 3 3 )
( 1 2 2 3 2 5 3 4 2 2 ) : upX6 : ( 1 1 2 3 1 1 3 1 2 2 )
( 1 2 2 3 2 5 3 4 2 2 ) : upX3 : ( 1 2 3 3 2 5 3 4 3 3 )
( 4 2 3 3 2 8 3 5 3 3 ) : upX0 : ( 1 2 3 3 2 5 3 4 3 3 )
( 4 2 3 3 2 8 3 5 3 3 ) : upX6 : ( 4 1 3 3 1 4 3 2 3 3 )
( 4 2 3 3 2 8 3 5 3 3 ) : rls : ( 4 2 3 4 2 8 3 5 3 3 )
```

```
( 1 1 2 3 1 1 3 1 2 2 ) : dnX6 : ( 1 2 2 3 2 5 3 4 2 2 )
( 1 1 2 3 1 1 3 1 2 2 ) : upX3 : ( 1 1 3 3 1 1 3 1 3 3 )
( 4 1 3 3 1 4 3 2 3 3 ) : upX0 : ( 1 1 3 3 1 1 3 1 3 3 )
( 4 1 3 3 1 4 3 2 3 3 ) : dnX6 : ( 4 2 3 3 2 8 3 5 3 3 )
( 4 1 3 3 1 4 3 2 3 3 ) : rls : ( 4 1 3 4 1 4 3 2 3 3 )
( 1 2 3 3 2 5 3 4 3 3 ) : upY2 : ( 2 2 3 3 2 6 3 5 3 3 )
( 1 2 3 3 2 5 3 4 3 3 ) : upX6 : ( 1 1 3 3 1 1 3 1 3 3 )
( 4 2 3 4 2 8 3 5 3 3 ) : upX0 : ( 1 2 3 4 2 5 3 4 3 3 )
( 4 2 3 4 2 8 3 5 3 3 ) : upX6 : ( 4 1 3 4 1 4 3 2 3 3 )
( 4 2 3 4 2 8 3 5 3 3 ) : dnX5 : ( 4 2 3 1 2 8 1 5 3 3 )
( 1 1 3 3 1 1 3 1 3 3 ) : dnX6 : ( 1 2 3 3 2 5 3 4 3 3 )
( 1 1 3 3 1 1 3 1 3 3 ) : rls : ( 1 1 3 4 1 1 3 1 3 3 )
( 4 1 3 4 1 4 3 2 3 3 ) : upX0 : ( 1 1 3 4 1 1 3 1 3 3 )
( 4 1 3 4 1 4 3 2 3 3 ) : dnX6 : ( 4 2 3 4 2 8 3 5 3 3 )
( 4 1 3 4 1 4 3 2 3 3 ) : dnX5 : ( 4 1 3 1 1 4 1 2 3 3 )
( 2 2 3 3 2 6 3 5 3 3 ) : upX1 : ( 3 2 3 3 2 7 3 6 3 3 )
( 2 2 3 3 2 6 3 5 3 3 ) : upX6 : ( 2 1 3 3 1 2 3 2 3 3 )
( 2 2 3 3 2 6 3 5 3 3 ) : rls : ( 2 2 3 4 2 6 3 5 3 3 )
( 1 2 3 4 2 5 3 4 3 3 ) : upY2 : ( 2 2 3 4 2 6 3 5 3 3 )
( 1 2 3 4 2 5 3 4 3 3 ) : upX6 : ( 1 1 3 4 1 1 3 1 3 3 )
( 1 2 3 4 2 5 3 4 3 3 ) : dnX5 : ( 1 2 3 1 2 5 1 4 3 3 )
( 4 2 3 1 2 8 1 5 3 3 ) : upX0 : ( 1 2 3 1 2 5 1 4 3 3 )
( 4 2 3 1 2 8 1 5 3 3 ) : upX6 : ( 4 1 3 1 1 4 1 2 3 3 )
( 4 2 3 1 2 8 1 5 3 3 ) : left : ( 4 2 4 1 2 8 1 5 2 1 )
( 1 1 3 4 1 1 3 1 3 3 ) : dnX6 : ( 1 2 3 4 2 5 3 4 3 3 )
( 1 1 3 4 1 1 3 1 3 3 ) : dnX5 : ( 1 1 3 1 1 1 1 1 3 3 )
( 4 1 3 1 1 4 1 2 3 3 ) : upX0 : ( 1 1 3 1 1 1 1 1 3 3 )
( 4 1 3 1 1 4 1 2 3 3 ) : dnX6 : ( 4 2 3 1 2 8 1 5 3 3 )
( 4 1 3 1 1 4 1 2 3 3 ) : left : ( 4 1 4 1 1 4 1 2 2 1 )
( 3 2 3 3 2 7 3 6 3 3 ) : upX6 : ( 3 1 3 3 1 3 3 3 3 3 )
( 3 2 3 3 2 7 3 6 3 3 ) : rls : ( 3 2 3 4 2 7 3 6 3 3 )
( 2 1 3 3 1 2 3 2 3 3 ) : upX1 : ( 3 1 3 3 1 3 3 3 3 3 )
( 2 1 3 3 1 2 3 2 3 3 ) : dnX6 : ( 2 2 3 3 2 6 3 5 3 3 )
( 2 1 3 3 1 2 3 2 3 3 ) : rls : ( 2 1 3 4 1 2 3 2 3 3 )
( 2 2 3 4 2 6 3 5 3 3 ) : upX1 : ( 3 2 3 4 2 7 3 6 3 3 )
( 2 2 3 4 2 6 3 5 3 3 ) : upX6 : ( 2 1 3 4 1 2 3 2 3 3 )
( 2 2 3 4 2 6 3 5 3 3 ) : dnX5 : ( 2 2 3 1 2 6 1 5 3 3 )
( 4 2 4 1 2 8 1 5 2 1 ) : upX0 : ( 1 2 4 1 2 5 1 4 2 1 )
( 4 2 4 1 2 8 1 5 2 1 ) : upX6 : ( 4 1 4 1 1 4 1 2 2 1 )
( 4 2 4 1 2 8 1 5 2 1 ) : upX2 : ( 4 2 1 1 2 8 1 5 1 1 )
( 4 1 4 1 1 4 1 2 2 1 ) : upX0 : ( 1 1 4 1 1 1 1 1 2 1 )
( 4 1 4 1 1 4 1 2 2 1 ) : dnX6 : ( 4 2 4 1 2 8 1 5 2 1 )
( 4 1 4 1 1 4 1 2 2 1 ) : upX2 : ( 4 1 1 1 1 4 1 2 1 1 )
( 3 1 3 3 1 3 3 3 3 3 ) : dnX6 : ( 3 2 3 3 2 7 3 6 3 3 )
( 3 1 3 3 1 3 3 3 3 3 ) : rls : ( 3 1 3 4 1 3 3 3 3 3 )
( 3 2 3 4 2 7 3 6 3 3 ) : upX6 : ( 3 1 3 4 1 3 3 3 3 3 )
( 3 2 3 4 2 7 3 6 3 3 ) : dnX5 : ( 3 2 3 1 2 7 1 6 3 3 )
( 2 1 3 4 1 2 3 2 3 3 ) : upX1 : ( 3 1 3 4 1 3 3 3 3 3 )
( 2 1 3 4 1 2 3 2 3 3 ) : dnX6 : ( 2 2 3 4 2 6 3 5 3 3 )
( 2 1 3 4 1 2 3 2 3 3 ) : dnX5 : ( 2 1 3 1 1 2 1 2 3 3 )
( 4 2 1 1 2 8 1 5 1 1 ) : upX0 : ( 1 2 1 1 2 5 1 4 1 1 )
( 4 2 1 1 2 8 1 5 1 1 ) : upX6 : ( 4 1 1 1 1 4 1 2 1 1 )
( 4 2 1 1 2 8 1 5 1 1 ) : right : ( 4 2 2 1 2 8 1 5 2 1 )
( 4 1 1 1 1 4 1 2 1 1 ) : upX0 : ( 1 1 1 1 1 1 1 1 1 1 )
```

```
( 4 1 1 1 1 4 1 2 1 1 ) : dnX6 : ( 4 2 1 1 2 8 1 5 1 1 )
( 4 1 1 1 1 4 1 2 1 1 ) : right : ( 4 1 2 1 1 4 1 2 2 1 )
( 3 1 3 4 1 3 3 3 3 3 ) : dnX6 : ( 3 2 3 4 2 7 3 6 3 3 )
( 3 1 3 4 1 3 3 3 3 3 ) : dnX5 : ( 3 1 3 1 1 3 1 3 3 3 )
( 4 2 2 1 2 8 1 5 2 1 ) : upX0 : ( 1 2 2 1 2 5 1 4 2 1 )
( 4 2 2 1 2 8 1 5 2 1 ) : upX6 : ( 4 1 2 1 1 4 1 2 2 1 )
( 4 2 2 1 2 8 1 5 2 1 ) : upX3 : ( 4 2 3 1 2 8 1 5 3 3 )
( 4 1 2 1 1 4 1 2 2 1 ) : upX0 : ( 1 1 2 1 1 1 1 1 2 1 )
( 4 1 2 1 1 4 1 2 2 1 ) : dnX6 : ( 4 2 2 1 2 8 1 5 2 1 )
( 4 1 2 1 1 4 1 2 2 1 ) : upX3 : ( 4 1 3 1 1 4 1 2 3 3 )
END
```

Le tableau 6 nous donne une représentation de la fonction d'inhibition (`Livraison.feedback` de SUCSEDES), dont les entrées sont triées et groupées par configuration de sortie (état de l'inhibition). Notez qu'il y a 12 configurations différentes de l'inhibition, et que chaque configuration partitionne l'ensemble des états du comportement libre de la machine (portion de l'état du contrôleur). L'état de l'inhibition ne dépend donc que de l'état courant de la machine abstraite du comportement libre dans lequel elle se trouve. En complémentant les sorties de cette fonction par rapport à l'alphabet du système et en éliminant les colonnes qui correspondent aux contraintes, on obtient une fonction assez près d'un SBFC qui nous donne quels sont les événements permis dans un état donné de la machine du comportement libre du système opérant sous contrainte.

Tableau 6. Fonction d'inhibition triée par configuration de sortie

États du contrôleur											
Comportements				Contraintes						Inhibitions	
3	2	3	1	2	7	1	6	3	3	dnY2	grab
3	1	3	1	1	3	1	3	3	3	dnY2	grab
3	2	4	1	2	7	1	6	2	1	dnY2	grab
3	1	4	1	1	3	1	3	2	1	dnY2	grab
3	2	3	3	2	7	3	6	3	3	dnY2	left
3	1	3	3	1	3	3	3	3	3	dnY2	left
3	2	3	4	2	7	3	6	3	3	dnY2	left
3	1	3	4	1	3	3	3	3	3	dnY2	left
3	2	1	1	2	7	1	6	1	1	dnY2	right
3	1	1	1	1	3	1	3	1	1	dnY2	right
3	2	1	2	2	7	2	6	1	1	dnY2	right
3	1	1	2	1	3	2	3	1	1	dnY2	right
4	2	3	1	2	8	1	5	3	3	grab	
4	1	3	1	1	4	1	2	3	3	grab	
4	2	4	1	2	8	1	5	2	1	grab	
4	1	4	1	1	4	1	2	2	1	grab	
4	2	1	1	2	8	1	5	1	1	grab	
4	1	1	1	1	4	1	2	1	1	grab	
4	2	2	1	2	8	1	5	2	1	grab	
4	1	2	1	1	4	1	2	2	1	grab	
4	2	3	3	2	8	3	5	3	3	left	
4	1	3	3	1	4	3	2	3	3	left	
4	2	3	4	2	8	3	5	3	3	left	
4	1	3	4	1	4	3	2	3	3	left	
2	2	3	3	2	6	3	5	3	3	left	

États du contrôleur												
Comportements				Contraintes						Inhibitions		
1	2	3	4	2	5	3	4	3	3	left		
2	1	3	3	1	2	3	2	3	3	left		
2	2	3	4	2	6	3	5	3	3	left		
2	1	3	4	1	2	3	2	3	3	left		
1	2	3	1	2	5	1	4	3	3	left	grab	
2	2	3	1	2	6	1	5	3	3	left	grab	
2	1	3	1	1	2	1	2	3	3	left	grab	
1	2	3	3	2	5	3	4	3	3	left	rls	
3	2	1	3	2	7	3	6	1	2	right	rls	
3	1	1	3	1	3	3	3	1	2	right	rls	
4	2	1	3	2	8	3	5	1	2	rls		
4	1	1	3	1	4	3	2	1	2	rls		
4	2	2	3	2	8	3	5	2	2	rls		
4	1	2	3	1	4	3	2	2	2	rls		
1	2	1	1	2	5	1	4	1	1	upY2	grab	
1	2	2	1	2	5	1	4	2	1	upY2	grab	
1	1	2	1	1	1	1	1	2	1	upY2	grab	
1	1	3	1	1	1	1	1	3	3	upY2	grab	
1	1	4	1	1	1	1	1	2	1	upY2	grab	
1	2	4	1	2	5	1	4	2	1	upY2	grab	
1	1	3	3	1	1	3	1	3	3	upY2	left	
1	1	3	4	1	1	3	1	3	3	upY2	left	
1	1	1	1	1	1	1	1	1	1	upY2	right	grab
1	1	1	3	1	1	3	1	1	2	upY2	right	rls
1	2	1	3	2	5	3	4	1	2	upY2	rls	
1	2	2	3	2	5	3	4	2	2	upY2	rls	
1	1	2	3	1	1	3	1	2	2	upY2	rls	

BIBLIOGRAPHIE

[AUT00] Automationdirect.com, *DL205 PLC User Manual*, Manual no. D2-USER-M, Automationdirect.com Inc., 2000.

[AUT99] Automationdirect.com, *DirectSOFT32 Programming Software User Manual*, Manual no. DS-DSOFT32-M, Automationdirect.com Inc., 1999.

[BOO67] T. L. Booth, *Sequential Machines and Automata Theory*, John Wiley and Sons Inc., New York, 1967.

[CAR99] V. Carré-Ménétrier, C. Ndjab Hagbebell, and J. Zaytoon, Methods and tools for the synthesis of an optimal control implementation for Grafcet, *APII-JESA* Vol. 33, No. 8, 1999, pp. 1073-1092.

[CEI88] Commission Électronique Internationale, *CEI 848 : Établissement de diagrammes fonctionnels pour systèmes de commande*, Genève, IEC Editions, première édition, 1988.

[CHK03] V. Chandra, Z. Huang, and R. Kumar, Automated Control Synthesis for an Assembly Line using Discrete Event System Control Theory, *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, Vol. 33, No. 2, 2003, pp. 284-289.

[DAV92] R. David et H. Alla, *Du Grafcet aux réseaux de Petri*, Hermès, Paris, deuxième édition, 1992.

[FES98] Festo, *Modular Production System – Testing Station Manual*, Technical Document no. 360-174, Festo Didactic GmbH & Co., 1998.

[FES98a] Festo, *Modular Production System – Processing Station Manual*, Technical Document no. 360-175, Festo Didactic GmbH & Co., 1998.

[FES98b] Festo, *Modular Production System – Sorting Station Manual*, Technical Document no. 360-182, Festo Didactic GmbH & Co., 1998.

[FES99] Festo, *Modular Production System – Distribution Station Manual*, Technical Document no. 360-173, Festo Didactic GmbH & Co., 1999.

[FES99a] Festo, *Modular Production System – Handling Station with Insertion Device Manual*, Technical Document no. 360-177, Festo Didactic GmbH & Co., 1999.

[KG95] R. Kumar and V. K. Garg, *Modeling and Control of Logical Discrete Event Systems*, Kluwer Academic Publishers Group, Norwell, MA, USA, 1995.

[RM89] P. J. G. Ramadge and W. M. Wonham, The Control of Discrete Event Systems, *Proceedings of the IEEE*, Vol. 77, No. 1, 1989, pp. 81-98.

[SDR99] R. St-Denis, *SUCSEDES : A Toolbox for Controller Development*, Département d'informatique, Université de Sherbrooke, mai 1999.

[SHA01] A. C. Shaw, *Real-Time Systems and Software*, John Wiley & sons Inc., New-York, 2001.

[WON04] W. M. Wonham and E. S. Rogers Sr., *Supervisory Control of Discrete-Event Systems*, ECE 1636F/1637S 2004-05, Systems Control Group, University of Toronto, July 2004.

[ZAY01] J. Zaytoon et V. Carré-Ménétrier, Grafcet et graphe d'état: comportement, raffinement, vérification et validation, *International Journal of Production Research*, Vol. 39, No. 2, 2001, pp. 329-345.

[ZAY99] J. Zaytoon et V. Carré-Ménétrier, Synthesis of control implementation for discrete manufacturing systems, *APII-JESA* Vol. 33, No. 7, 1999, pp. 751-782.

[ZAY99a] J. Zaytoon, C. Ndjab Hagbebell et V. Carré-Ménétrier, Grafcet et graphes d'états : synthèse hors ligne de la commande, *APII-JESA*, Vol. 33, No. 7, 1999, pp. 783-814.