Conception, développement et évaluation d'un langage de programmation adapté aux applications industrielles : llC.

Hugo Delchini

Université Paris VII

THESE

présentée par

Hugo DELCHINI

pour obtenir le titre de Docteur de l'Université Paris VII
(spécialité informatique fondamentale)

Sujet : conception, développement et évaluation d'un langage de programmation adapté aux applications industrielles : IIC.

Soutenue le 15 Fevrier 1995 devant la commission d'examen :

M. NIVAT	**Président du jury**
M. RIVOAL	**Rapporteur**
A. ARNOLD	**Rapporteur**
F. BOUSSINOT	**Examinateur**
JM. RIFFLET	**Examinateur**

Résumé

Le développement d'applications informatiques passe souvent par l'utilisation de langages évolués pour la programmation et de systèmes d'exploitation pour la gestion de l'exécution. Une famille de langages de programmation (les LPC pour "Langages a Parallélisme Compilé") peuvent procurer les mêmes fonctionnalités et avantages qu'un langage de programmation couplé à un système d'exploitation multi-tâche. Notre intention est de montrer que les LPC ont certains attraits supplémentaires surtout dans le domaine d'application qu'est l'informatique industrielle. Pour mettre ceci en évidence, nous avons développé plusieurs versions d'une même application en utilisant d'une part un LPC et, d'autre part, un langage évolué classique avec un système d'exploitation. Ensuite, nous avons fait une comparaison chiffrée des différentes versions.

1. Introduction.

Le développement d'applications d'informatique industrielle passe très souvent, de nos jours, par l'utilisation de langages évolués pour la programmation et par l'emploi de systèmes d'exploitation pour le contrôle de l'exécution. Un système d'exploitation offre des services que le programmeur utilise au travers du langage de programmation pour implanter une application sur un ordinateur.

L'utilisation d'un langage de programmation évolué associé à un système d'exploitation, permettant par exemple l'exécution simultanée de plusieurs tâches, procure un certain confort de conception et de programmation des applications, dont il est rapidement difficile de se passer. Cette association permet de plus de rendre des applications à peu près indépendantes du matériel tout en procurant des facilités de maintenance et d'évolution.

Une famille de langages de programmation de haut niveau (que nous appellerons "LPC" pour Langages à Parallélisme Compilé) intégrant des fonctionnalités de systèmes d'exploitation (gestion de tâches, synchronisation et communications entre tâches, ...) procure, comme nous le verrons plus loin, les mêmes avantages que ceux cités précédemment. On admet, d'un point de vue théorique, comme dans [Halbwachs91], que ces langages apportent des avantages supplémentaires en terme de portabilité, de

modélisation et d'efficacité d'exécution. C'est ce dernier avantage qui retiendra plus particulièrement notre attention.

Notre intention est de montrer que les avantages des LPC par rapport aux couples "langage évolué et système d'exploitation" sont bien réels et qu'il peut par conséquent être plus intéressant de les utiliser pour résoudre certains problèmes.

Pour mener à bien ce travail, nous allons d'abord nous situer à l'intérieur d'un cadre dans lequel le développement d'applications d'informatique industrielle est courant. Nous dégagerons ensuite les principales caractéristiques de ces applications et nous présenterons des outils et des principes de programmation classiques permettant de les concevoir et de les implanter sur des ordinateurs en respectant les contraintes qu'elles imposent.

Nous introduirons ensuite les LPC et montrerons alors, en présentant certains d'entre eux, qu'ils peuvent également constituer des solutions pour la programmation des applications qui nous intéressent, avec des avantages supplémentaires. Par la suite, nous présenterons le LPC que nous avons défini et particulièrement étudié pour favoriser l'efficacité d'exécution des *applications*.

Dans la dernière partie de ce travail, nous présenterons plusieurs versions d'une application typique du cadre que nous nous sommes donné. Nous programmerons chaque version en utilisant soit un langage évolué classique avec un système d'exploitation, soit en utilisant le langage que nous avons défini. Nous comparerons ensuite des temps significatifs mesurés pour chaque version de l'application, ce qui nous permettra, entre autres, de savoir si notre langage apporte bien un plus au niveau de l'efficacité d'exécution.

1.1. Description du cadre de travail et exemples d'applications.

Nous allons brièvement décrire le contexte industriel dans lequel nous allons travailler. Il s'agit d'un organisme de recherche rattaché à l'administration française où les activités et développements, aussi bien en informatique que dans d'autres domaines, sont à peu de choses près équivalents à ceux de certains secteurs de l'industrie privée.

1.1.1. Présentation du LPNHE.

Le Laboratoire de Physique Nucléaire et des Hautes Energies se situe au cœur du campus de l'université de Jussieu (Paris VI et VII). il est associé aux deux universités et à l'IN2P3 (Institut National de Physique Nucléaire et de Physique des Particules), institut du CNRS. C'est au sein de ce laboratoire que je travaille en tant qu'ingénieur d'étude en informatique. L'activité technique principale de ce laboratoire est le développement d'appareils de mesures (appelés en général "grands instruments") pour des expériences de physique des particules. Les domaines dans lesquels nous disposons de compétences techniques sont la mécanique, l'électronique et l'informatique.

Le LPNHE emploie une centaine de personnes. Il se compose de techniciens, d'ingénieurs et d'un groupe d'enseignants/chercheurs en physique. L'équipe technique compte une quarantaine de personnes dont une douzaine en informatique. Des équipes d'électronique et de mécanique travaillent à la réalisation des instruments de mesures pour les expériences. Les informaticiens du laboratoire sont répartis en deux groupes, l'un travaillant directement sur les expériences concernant le laboratoire et l'autre assurant des tâches d'intérêt général (gestion des ordinateurs locaux, du réseau et de tous les périphériques associés, gestion des outils de conception assistée par ordinateur pour l'électronique ou la mécanique, ...).

A l'heure actuelle, le LPNHE est impliqué dans plusieurs expériences. J'ai participé à la construction et la mise au point d'une partie du système d'acquisition de données de l'expérience H1 ainsi que de l'expérience CAT. Nous allons les présenter rapidement pour donner une idée des spécificités de leurs systèmes informatiques. Les environnements de développement que nous verrons plus loin, devront nous permettre de respecter les contraintes qu'introduisent ces spécificités.

1.1.2. L'expérience H1.

H1 est un détecteur de particules installé sur l'accélérateur HERA à Hambourg en Allemagne. HERA est un anneau de collisions électron-proton dont l'objectif principal est l'étude de la structure du proton et la vérification de certaines prédictions des modèles théoriques actuels de physique des particules (on pourra se référer à [H1] pour des compléments d'information).

La période des croisements de particules au niveau du détecteur est très courte (toutes les 96 nanosecondes contre 22 microsecondes au LEP, "Large Electron/Positron collider", au CERN, Centre Européen de Recherche Nucléaire).

L'informatique intervient dans l'expérience pour réaliser la lecture, ou acquisition de données, et le contrôle des mesures prises par les capteurs (environs 70000 voies) du détecteur lors des collisions entre particules. Le déclenchement de la lecture est fait par une interruption à laquelle réagit l'application informatique d'acquisition des données. La contrainte essentielle donnée au système d'acquisition est de travailler le plus rapidement possible afin de ne pas introduire un temps mort excessif par rapport à la fréquence des croisements de particules dans le détecteur et donc de minimiser les pertes de données relatives à des interactions intéressantes.

Un dispositif de filtrage des données permet à chaque croisement de décider si l'interaction qui vient d'avoir lieu est intéressante ou non pour la physique étudiée. Ce dispositif réduit la fréquence d'acquisition au niveau d'une centaine de Hertz. Les voies de mesures non touchées par des particules sont éliminées par des traitements en amont du système informatique ce qui permet aussi de réduire le volume d'information à traiter.

Il est important que ce dispositif d'acquisition fonctionne le plus rapidement possible car les physiciens doivent disposer d'un grand nombre de données afin de valider des prédictions avec une bonne statistique. La durée de fonctionnement d'une telle expérience est d'environ 10 ans et elle doit fournir un maximum de données exploitables.

Le système d'acquisition se compose d'une carte électronique avec un processeur sous le contrôle d'un système d'exploitation adapté pour les applications industrielles (VRTX-32®). On dispose également d'une interface matérielle entre le processeur et les appareils de mesures. Le logiciel doit assurer l'acquisition et le transfert des données vers un système centralisateur qui les stocke pour analyse ultérieure.

1.1.3. L'expérience CAT (Cherenkov Array at Thémis).

Cette expérience consiste en partie en la construction d'une caméra haute définition pour la détection "Cherenkov" du rayonnement gamma provenant

de sources galactiques ou extragalactiques (voir [CAT] pour plus de détails). Le site de l'expérience est l'ancienne centrale solaire d'EDF dans les Pyrénées (Thémis

L'informatique intervient au même niveau que dans l'expérience H1 mais les données à lire proviennent de phénomènes physiques dont l'instant d'arrivée est indéterminé. En effet, dans H1, une interaction physique aura lieu à l'instant connu de croisement des particules dans le détecteur, ce qui dépend de la fréquence de fonctionnement de l'accélérateur. Alors que dans CAT, la nature ne nous informe pas de l'arrivée d'une particule dans les capteurs de la caméra de manière précise. Il est seulement possible de déterminer une fréquence moyenne d'arrivée des particules (estimée pour l'instant à une centaine de Hertz).

Le nombre de voies de mesure (600 environs) est beaucoup moins important que dans H1. La difficulté est donc moindre mais l'aspect temps réel est tout aussi important pour les mêmes raisons (rapidité de réaction et de fonctionnement pour perdre le moins d'événements possible et obtenir de bonnes statistiques sur une période limitée).

Le dispositif d'acquisition de données est à peu de choses près constitué d'un matériel équivalent à celui de l'expérience H1 mais le système d'exploitation utilisé est ici LynxOS® que nous décrirons en détail plus loin.

1.2. Caractéristiques des applications décrites.

Les applications d'acquisition de données développées au laboratoire pour les expériences suivent pratiquement toujours le schéma des dispositifs que nous venons de décrire en ce qui concerne la partie informatique. Leurs principales caractéristiques sont :

La réaction à un ou plusieurs signaux de déclenchement : c'est le cas de la plupart des applications industrielles, elles réagissent à des stimuli qui déclenchent des actions comme la lecture de données, l'émission d'autres stimuli, etc. Lorsque l'application est prête en permanence pendant son fonctionnement à répondre à chaque stimulus sans introduire de délai dans le traitement des stimuli suivants, elle est alors dite "temps réel". Dans ce cas, on peut considérer que les réactions sont synchrones avec l'arrivée des stimuli.

L'existence de plusieurs traitements plus ou moins indépendants : les applications sont souvent constituées de tâches qui contiennent des traitements indépendants et qui communiquent via des mécanismes adaptés (synchronisation, exclusion mutuelle pour les données, ...). Par exemple, pour H1, une tâche s'occupe de la lecture des données et une autre des transferts. Cette organisation permet, de surcroît, un découplage des deux traitements et donc de faire éventuellement fonctionner des dispositifs matériels en parallèle (interfaces asynchrones entre deux bus). Il peut aussi être utile de pouvoir spécifier différentes tâches indépendamment pour permettre de structurer plus aisément l'application.

L'exécution sur du matériel spécifique et un contact direct avec celui-ci : ce matériel est d'ordinaire développé au laboratoire car il doit la plupart du temps répondre à certaines spécificités (bus particuliers, interfaces vers des appareils de mesures non standards, ...). Il est souvent en évolution et le logiciel qui a été fabriqué pour l'exploiter doit pouvoir s'adapter à ces évolutions. C'est souvent le cas pour les systèmes industriels même si la normalisation est de plus en plus à la mode. On peut aussi considérer que le matériel fourni par des constructeurs est en constante évolution. Souvent, entre le démarrage de l'étude du dispositif d'acquisition et les premières prises de données exploitables, il peut s'écouler plusieurs années. Or, en informatique et en électronique, l'évolution technique est très rapide. Le logiciel doit aussi pouvoir suivre une évolution du matériel entre le début et la fin du montage d'une expérience.

Après avoir dégagé les principales caractéristiques des applications qui nous intéressent, nous allons présenter une partie des outils et moyens dont on dispose pour les développer.

2. Différentes solutions de développement.

Il existe plusieurs manières de développer des applications, que ce soit dans le domaine de l'informatique industrielle ou non. Notre propos dans cette partie est d'en présenter quelques-unes qui sont représentatives des méthodes utilisées dans l'industrie pour des machines monoprocesseur tout en nous permettant de satisfaire certaines contraintes présentées plus haut.

Une première organisation possible pour le développement d'applications "temps réel" ou embarquées, est la compilation croisée. Sur une machine centrale on utilise un compilateur, pour un langage évolué ou non, qui produit du code exécutable pour la machine sur laquelle l'application doit finalement s'exécuter (nous parlerons de machine cible). Quand les programmes sont développés, ils sont téléchargés dans la mémoire de la machine cible via une connexion physique telle qu'une ligne série et on peut alors les exécuter.

La machine cible doit, par exemple, disposer d'un programme intégré stocké en mémoire morte que l'on appelle couramment "moniteur". Un moniteur permet de réaliser des commandes de base sur une machine (visualisation des registres du processeur, de la mémoire, assemblage ou désassemblage de programmes en mémoire, téléchargement de données ou de programmes vers la mémoire, ...) : c'est un outil de très bas niveau qui est très proche de la machine. On peut également disposer d'un simulateur

logiciel de processeur sur la machine centrale de développement. Si c'est le cas, une simulation de l'exécution des programmes peut être effectuée avant de les télécharger sur la machine cible, ceci permettant une mise au point plus commode suivant les qualités du simulateur qui doit imiter au mieux le processeur cible ainsi que tous les périphériques qui l'entourent.

Une deuxième manière de procéder est d'utiliser un compilateur et des outils de mise au point directement sur la machine cible. Cette dernière doit alors être sous le contrôle d'un système d'exploitation qui permet d'utiliser des compilateurs et d'autres outils de développement. Ce système d'exploitation permet également de démarrer l'exécution des programmes que l'on développe. Les simulateurs ne sont plus utiles et on peut disposer d'outils de mise au point symbolique (débogueurs symboliques et autres).

Passons maintenant à la description de certaines solutions pour programmer les applications que nous avons caractérisées. Ces solutions sont utilisables avec ou sans compilation croisée.

2.1. Solution sans système d'exploitation, avec un langage évolué.

La première approche que nous allons présenter est certainement la plus ancienne. On peut considérer que beaucoup d'applications industrielles ont été et sont développées en l'employant. Elle consiste en l'utilisation d'un langage de programmation, de haut niveau ou non, qui va permettre de développer une application : celle-ci ne s'exécutera pas sous le contrôle d'un système d'exploitation. Tout est à la charge du programmeur qui n'a pas la possibilité d'utiliser des objets évolués pour concevoir l'application et la structurer (tâches, outils de communication et de synchronisation entre elles, ...). On emploiera certainement un moniteur pour le téléchargement des programmes et leur mise au point. L'environnement d'exécution se réduit quasiment au matériel.

En fait, ici, la seule chose qui peut avoir de l'importance est le choix du langage de programmation. Il peut être avantageux d'utiliser un langage évolué, avec beaucoup de bibliothèques de fonctions qui pourront, par exemple, remplacer quelques-uns des services d'un système d'exploitation. Le choix du langage peut aussi être induit par des raisons plus contingentes telles que la disponibilité d'un compilateur pour le matériel employé. Il peut

être intéressant de choisir un langage normalisé dont l'utilisation est répandue.

Il existe plusieurs langages qui respectent ces critères. Celui que nous retiendrons est le langage C décrit dans [K&R] ou éventuellement C++ qui en est la version objet. Le choix entre les deux n'est presque qu'une question de goût ou d'habitudes. Il faut noter que la gestion des objets engendre, la plupart du temps, un surcoût en terme de taille mémoire et de complexité du code exécutable. Le contrôle de ces coûts supplémentaires est incertain alors qu'il est nécessaire dans notre domaine d'application de maîtriser au mieux tous les aspects du développement d'un programme pour être sûr que celui-ci respecte bien les contraintes qui lui sont imposées.

A l'origine, le langage C a été conçu, entre autres, pour le développement de systèmes d'exploitation. Il convient donc théoriquement au développement de logiciels de base et d'applications de bas niveau (c'est à dire en contact direct avec le matériel). Ceci est un argument en faveur du choix de ce langage car, le plus fréquemment, les applications industrielles d'acquisition et de traitement de données qui nous intéressent sont de bas niveau.

Un motif supplémentaire pour le choix de ce langage est l'existence de compilateur pour beaucoup de processeurs existants. De plus, un organisme, la "Free Software Foundation" (FSF), propose gratuitement les fichiers sources d'un compilateur C (GNUCC ou GCC) relativement bien fait et portable qui respecte la norme ANSI ("American National Standard Institute"). Pour l'adapter à une nouvelle machine, son processeur doit être décrit suivant une méthode normalisée par la FSF (instructions, nombre de registres, ...). A partir de cette description, le compilateur est alors capable de produire du code pour cette nouvelle machine.

Cette adaptation à différentes machines est un travail conséquent mais il a été fait pour les processeurs les plus répandus (familles MC680X0®, Intel™, AMD29000®, ...). Cet organisme propose en plus un compilateur C++ (GNUC++) ainsi qu'un outil de mise au point symbolique (GDB), un outil permettant de faire de la compilation séparée (GNUMAKE) et bien d'autres choses, toujours gratuitement.

Le compilateur "GCC" utilise une méthode générale d'optimisation du code produit en se basant sur la description du processeur. On peut donc disposer non seulement d'un compilateur, mais aussi, et automatiquement, d'un optimiseur qui est en moyenne assez efficace. Nous n'entrerons pas plus dans le détail sur ce sujet sachant que pour plus d'information on peut se référer à la documentation gratuite [GNU].

Les applications industrielles ont été souvent développées sans utiliser des systèmes d'exploitation pour en assurer le contrôle lors de l'exécution. Les raisons sont soit historiques (utilisation d'une méthode connue sans en envisager une autre), soit dues à un objectif de compatibilité avec des logiciels ou des matériels déjà utilisés. Les spécifications du logiciel et du matériel peuvent aussi inciter à utiliser cette solution élémentaire. En effet, si l'application est très simple, il n'est alors pas forcément utile de compliquer les choses. Par exemple, vouloir utiliser un système d'exploitation sur un ordinateur à base de micro-contrôleur (version simplifiée d'un microprocesseur) n'est certainement pas réaliste.

Les programmes que l'on écrit en suivant cette solution, sont généralement composés essentiellement de routines d'interruptions. Le cœur en est une boucle sans fin, éventuellement vide et des interruptions matérielles déclenchent des actions. Une ou plusieurs interruptions spéciales peuvent rythmer le fonctionnement du programme, par exemple, des interruptions à intervalles réguliers provoquées par une horloge matérielle ("timer"). D'autres interruptions vont être associées à des périphériques du processeur (convertisseurs de signal analogique en signal numérique ou l'inverse, registres de commandes d'appareils, circuits de communication avec d'autres machines, ...).

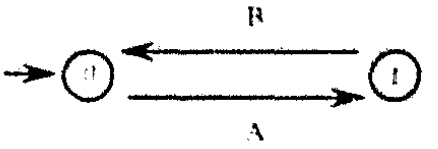
Même si on utilise un langage de programmation comme le C on est souvent obligé dans ce contexte de développer certaines parties en assembleur. Les applications sont de très bas niveau et le processeur communique avec des périphériques au travers d'instructions particulières, surtout les micro-contrôleurs dont c'est la vocation. De toutes manières, il est pratiquement hors de question d'imaginer des applications portables avec ce principe et les problèmes de maintenance ne sont pas simples. Le programmeur, ayant toute liberté, peut utiliser un style de programmation

personnel et il sera alors difficile à d'autres informaticiens de faire évoluer l'application.

En utilisant le même contexte d'exécution sans système d'exploitation, on peut donner aux applications une structure qui permettra de normaliser un peu la programmation. La plus utilisée est certainement la structuration en automates finis (systèmes de transitions ou encore machines à états).

Les automates finis que nous allons considérer sont constitués d'un ensemble d'états (les états possibles du système sont en nombre fini, d'ou le qualificatif pour ces automates), d'un ensemble de transitions qui relient un état à un autre, et d'un alphabet dont les lettres étiquettent les transitions. On suppose également l'existence d'une entrée de données dans l'automate, ces données prenant leurs valeurs dans l'alphabet. Des valeurs sont ainsi successivement introduites dans l'automate pendant son fonctionnement.

Voici un exemple d'automate fini décrit par une représentation graphique classique :



Le fonctionnement d'un tel automate est le suivant : à un moment donné, le système se trouve dans un des états possibles et en fonction de la valeur courante en entrée (prise dans l'alphabet), une transition lui est appliquée : le système change d'état courant alors que l'entrée est consommée. Ce fonctionnement est souvent réalisé en pratique par un programme que l'on appelle "moteur" qui utilise une description d'un automate (table de transitions) pour effectuer des transitions (changements d'état).

Cet automate possède deux états : "0" et "1", deux transitions (les flèches étiquetées), et un alphabet à deux lettres : {A,B}. L'état "0" est dit initial car c'est dans celui-ci que le système se trouve au commencement de l'exécution, ceci est indiqué par la flèche non étiquetée. Cet automate fini décrit un comportement, ce qui est le cas de tous les automates finis et c'est pourquoi on peut les utiliser pour structurer une application. On peut interpréter un automate fini comme un "résumé" du fonctionnement d'un programme. Un

organigramme peut par exemple être considéré comme un automate fini. Le comportement est ici : quand on est dans l'état "0", si l'entrée courante est "A", on passe dans l'état "1", sinon on reste dans l'état "0". On procède de la même manière symétriquement quand on est dans l'état 1 mais on réagit à l'entrée "B". Les entrées peuvent être associées à des stimuli externes, des interruptions par exemple et l'automate décrit alors les enchaînements possibles des états du système en réaction à des interruptions externes.

L'utilisation de ce moyen de structuration des applications peut rendre des services mais la complexité ne doit pas être trop importante car des automates finis avec beaucoup d'états et de transitions deviennent vite incompréhensibles. De plus, toute modification du comportement escompté pour l'application engendre une modification de l'automate qui peut être importante. Par conséquent, il n'est pas recommandé d'utiliser cette méthode si cela complique les choses. Par contre, pour des applications simples, l'utilisation de ce principe peut apporter une vision claire de l'organisation du programme et permettre alors à plusieurs personnes de la comprendre sans trop de difficultés. Il est bien entendu possible d'utiliser les automates finis pour modéliser le fonctionnement d'applications même si celles-ci s'exécutent sous le contrôle de systèmes d'exploitation.

Nous verrons plus loin que les LPC (Langages à Parallélisme Compilé cités dans l'introduction) utilisent les automates finis pour modéliser le fonctionnement des programmes.

Citons un autre outil mathématique pouvant permettre de structurer et de décrire des applications : les réseaux de Pétri. Nous n'en parlerons pas plus, sinon pour dire que c'est une extension des automates finis qui permet de spécifier des applications réparties (sur plusieurs processeurs). Ceci n'est pas notre propos étant donné que nous ne nous intéressons qu'aux machines à un processeur.

Même si elle nous permet de résoudre tous les problèmes et de satisfaire toutes les contraintes, nous n'adopterons pas cette solution de programmation pour notre évaluation en dernière partie de ce travail. Nous l'avons présentée car elle est encore valable dans certains cas très précis. La seule chose que nous retiendrons est le choix du langage C pour la

programmation. Nous allons maintenant voir un autre type de contexte d'exécution lié à un environnement de programmation.

2.2. L'approche avec exécutif et langage évolué.

L'utilisation d'un tel environnement de développement et d'exécution commence à se répandre pour le développement d'applications industrielles. Une première raison est le progrès du matériel qui rend possible l'utilisation de systèmes d'exploitation permettant alors de satisfaire plus de contraintes au niveau des temps de réponse. Une autre est le développement de systèmes d'exploitation spécialement conçus pour l'exécution et le contrôle d'applications industrielles.

Le langage C est associé à chaque système que nous allons présenter. Le couple langage/système offre une solution pour le développement des applications qui nous intéressent. Nous allons présenter deux systèmes d'exploitation représentatifs de ceux utilisés dans l'industrie à l'heure actuelle. Le premier de conception relativement ancienne est très bien connu au laboratoire et il est assez répandu. Le second commence à l'être : il est récent et respecte des normes ce qui est un bon point de nos jours où la standardisation est de rigueur. Citons également "OS9" qui est un système d'exploitation très utilisé dans notre domaine d'application mais que nous ne présenterons pas pour mieux nous concentrer sur les deux autres.

Nous allons également nous pencher sur le langage de programmation ADA. Nous expliquerons plus loin pourquoi nous classons ce langage avec les systèmes d'exploitation. Dans la partie d'évaluation des environnements de développement et d'exécution, nous utiliserons les deux systèmes d'exploitation avec le langage C, c'est pourquoi nous en faisons une description assez détaillée.

2.2.1. Un système répandu : VRTX-32.

2.2.1.1. Description.

Toutes les informations sur le système et le matériel dont nous allons parler proviennent de [VRTX-32] et de [AMD29000].

VRTX-32 ("Versatile Real Time eXecutive") est un système conçu pour permettre à des applications industrielles de répondre le mieux possible aux contraintes qu'on leur impose. Il est constitué d'un noyau que l'on peut adapter à un problème donné. C'est une organisation classique pour ce genre de système. Nous disposons au laboratoire d'une version pour le processeur AMD29000 ("Advanced Micro Devices") que nous avons implanté nous même, mais il existe des versions pour d'autres processeurs (68000, ...). Le matériel sur lequel s'exécute VRTX-32 a été développé au laboratoire et il est utilisé dans l'expérience H1 dont nous avons parlé dans l'introduction

Les développements en langage C se font sur d'autres machines que la carte AMD29000. On dispose d'un compilateur C 29000 sur PC, et un autre sur station de travail SUN8. Les applications sont téléchargées sur la carte AMD29000 via un système de communication mis au point au laboratoire dont le support physique est le bus VME et, elles s'exécutent sous le contrôle de VRTX-32. Nous employons donc un environnement de développement avec compilation croisée. Il est possible pour VRTX-32 de gérer des disques et donc de disposer d'un environnement de développement intégré à la machine cible. Mais au laboratoire nous avons choisi l'autre option pour des raisons de coût et de temps de développement du matériel (inutile de mettre au point une interface vers des disques).

Nous allons dégager maintenant les principales caractéristiques de ce système d'exploitation. VRTX-32 est un exécutif multitâches et mono-utilisateur. Cela signifie que plusieurs tâches se partagent le temps d'exécution du processeur et qu'il n'y a pas de notion d'utilisateurs distincts. Il n'y a que trois niveaux d'exécution : le mode système, le mode utilisateur unique et le niveau d'exécution des routines d'interruption. Dans l'implantation que nous avons faite au laboratoire, le système d'exploitation et les tâches de l'utilisateur s'exécutent dans le même espace d'adressage physique pour les instructions et les données. Il n'y a pas de protection pour les accès à la mémoire : une tâche quelconque peut accéder à toute la mémoire (système ou non).

Il est possible de réaliser une implantation de VRTX-32 avec une gestion de mémoire paginée et virtuelle, mais cela impose de tout programmer soi-même, la version de base du système n'assurant aucune gestion de ce type. Les applications que nous avons à développer au laboratoire ne nécessitant

pas de gestion particulière de la mémoire, nous avons choisi la solution la plus simple. Il faut cependant remarquer que la mémoire virtuelle a un intérêt dans l'informatique industrielle : elle permet une division de la mémoire physique en plusieurs espaces d'adressage et, par conséquent, offre une possibilité de protection de chaque espace par des mécanismes matériels. Une tâche possède son espace propre et ne peut pas accéder à celui des autres. C'est un point sur lequel nous reviendrons dans la présentation de LynxOS.

Le système peut être installé et s'exécuter en "ROM" ("Read Only Memory") ou mémoire morte. Ceci est un détail pratique, mais cette fonctionnalité est souvent utilisée en informatique industrielle pour réaliser des applications embarquées et autonomes : un système et une application installés en mémoire morte constituent un ensemble qui peut ne pas être en contact avec l'extérieur et qui ne nécessite pas de téléchargement.

Les appels système ne suivent aucune norme particulière, ce qui n'est pas une qualité à notre époque. Par contre, le compilateur C dont nous disposons respecte la norme ANSI.

Le noyau est livré sous la forme d'un fichier source en assembleur. Ce fichier contient le code exécutable du système codé par une série de déclarations d'octets. On ne dispose pas des vrais programmes sources du système mais avec ce fichier, on peut assembler tout le système et son logiciel d'initialisation. L'écriture du logiciel d'initialisation est à la charge de l'utilisateur, ce que nous avons fait au laboratoire. Il comporte des fonctions interfaces entre le système et les périphériques (ports séries, "timer", ...). C'est dans ce logiciel que l'on configure le système pour un problème donné : on doit en fait construire une table d'initialisation décrivant toutes les ressources matérielles, le nombre de tâches, la taille des piles, etc. Le développement et la mise au point de ce logiciel d'initialisation sont assez fastidieux et quand cela est terminé, il reste à développer l'application utilisateur.

On peut compléter la gestion des tâches que fait le système en fournissant dans le logiciel de configuration, trois fonctions appelées par le système lors de la création de tâche, de la commutation d'une tâche à une autre (commutation de contexte) et de la terminaison de tâche. Ces fonctions sont également assez difficiles à mettre au point, mais elles apportent une certaine souplesse. Nous avons, par exemple, utilisé ce principe dans notre

implantation, pour gérer un coprocesseur arithmétique (gestion des registres du coprocesseur pour chaque tâche). Pour une gestion de mémoire virtuelle, on utiliserait aussi ces fonctions (gestion des tables de pages pour une nouvelle tâche ainsi que pour une tâche se terminant, initialisation du matériel gérant l'adressage virtuel et sa mise à jour lors d'une commutation de contexte, ...).

Le système gère l'allocation dynamique de mémoire et les entrées/sorties sur des terminaux mais, pour ceci, il faut lui donner des fonctions d'interface ("driver" ou pilote) dans le logiciel de configuration. Par contre, les fonctions standard d'entrées/sorties ("printf()", "scanf()", ...) et de gestion dynamique de la mémoire ("malloc()", "free()", ...) du C ne sont pas fournies et il faut les réaliser.

Dans la version 29000, certains registres sont utilisés uniquement par le système, ceci n'étant possible que parce que le processeur en possède beaucoup (192). C'est une organisation interne qui permet d'augmenter l'efficacité d'exécution : inutile de sauvegarder tous les registres quand on passe d'une tâche au système (appel système) et d'en rétablir à l'inverse. Mais le grand nombre de registres par tâche est une pénalité car les commutations de contextes sont très coûteuses (sauvegarde et rétablissement d'un nombre important de registres). Dans la version 68000 ce n'est pas le cas car ce processeur possède beaucoup moins de registres. Par contre, le système n'a pas de registres privés.

Après cette présentation de l'organisation générale de VRTX-32 nous allons décrire plus en détail les objets que le système gère.

2.2.1.2. Les tâches.

Comme nous l'avons dit, VRTX-32 est un système multitâches. Une tâche VRTX-32 correspond à l'exécution d'un programme par le processeur. Le système gère la répartition du temps d'exécution entre toutes les tâches actives à un instant donné. Une tâche a la possibilité de créer d'autres tâches mais il n'y a pas de lien de parenté entre elles. Une tâche ne peut pas a priori savoir quelle tâche l'a créée. Les tâches naissent et meurent sans que cela ait des répercutions autres que des commutations de contextes en fonction des

priorités. Par exemple, il n'y a pas de rendez-vous implicite entre une tâche et sa mère.

Chaque tâche possède une priorité qui lui est donnée à sa création. Les priorités permettent au système de déterminer à quelle tâche, parmi toutes celles qui sont actives, le processeur va être attribué à un moment donné (ordonnancement). Ces priorités sont fixes, c'est-à-dire qu'elles ne varient pas en fonction du temps. Dans les systèmes multitâches orientés vers le partage du temps, comme UNIX® par exemple, la priorité des tâches évolue avec le temps. Une tâche qui consomme beaucoup de temps d'exécution verra sa priorité diminuer pendant un certain laps de temps et revenir à sa valeur initiale ensuite. Ce mécanisme permet au système d'exploitation d'être équitable, et donc à toutes les tâches actives d'avoir la garantie de s'exécuter. Dans un système non équitable, une tâche peut monopoliser le processeur, sans jamais permettre aux autres tâches de s'exécuter.

VRTX-32 n'est pas un système équitable pour la répartition du temps d'exécution entre des tâches de priorités différentes. Pour l'ordonnancement des tâches, le système gère des files de tâches construites suivant les priorités. Deux modes de fonctionnement sont possibles. Dans le premier, les tâches non bloquées de la file ayant la plus forte priorité s'exécutent tour à tour durant un temps maximum appelé quantum (environ 10 millisecondes). Le système reprend le contrôle du processeur à l'expiration de chaque quantum par le truchement d'une interruption produite par une horloge matérielle.

Cette gestion par quantum fait que VRTX-32 répartit équitablement le temps d'exécution du processeur pour les tâches d'une même priorité. Dans le second mode de fonctionnement le quantum est infini et, c'est seulement quand la tâche en cours d'exécution relâche d'elle même le processeur que le système en élit une autre suivant les priorités.

Nous avons vu précédemment que l'espace d'adressage est unique pour toutes les tâches ainsi que pour le système. On peut quand même signaler que chaque tâche possède une pile qui lui est propre. Cette pile permet d'implanter (via un langage évolué ou non) des mécanismes d'appels de fonctions avec passage de paramètres et allocations dynamiques de variables

locales à une fonction. Le fait que chaque tâche ait une pile réservée n'empêche pas une tâche d'accéder à la pile d'une autre tâche.

Dans VRTX-32, on trouve toutes les opérations courantes pour manipuler les tâches. Nous allons les décrire sans entrer dans les détails, le but étant simplement de lister les fonctionnalités :

Création : la création d'une tâche se fait avec spécification de priorité et de mode d'exécution : superviseur ou utilisateur. La nouvelle tâche hérite des valeurs des registres de données et d'adressages de la tâche créatrice. Si la routine d'extension de cette opération existe, elle est exécutée.

Terminaison : possibilité de terminaison artificielle d'une tâche, une tâche pouvant se terminer elle même. Si la routine d'extension de cet appel existe, elle est exécutée.

Suspension : suspension de l'exécution d'une tâche, une tâche pouvant se suspendre elle-même en cas de besoin.

Reprise : après qu'une tâche ait été suspendue, il est possible de lui faire reprendre son exécution.

Priorité : changement de priorité d'une tâche alors qu'elle est en cours d'exécution.

Informations : *récupération de l'adresse du bloc de contrôle d'une tâche.* On peut ainsi obtenir des renseignements : activité ou non-activité, priorité, *ou des informations plus critiques comme l'adresse de pile, etc.*

C'est parce que VRTX-32 est un système conçu pour les applications industrielles qu'il n'est pas équitable. De plus, une tâche très prioritaire peut ainsi monopoliser le processeur et ne le relâcher que lorsqu'elle le décide. Par ailleurs, des appels système supplémentaires permettent à une tâche de verrouiller la commutation de contexte et la gestion avec quantum d'exécution. Une fois ce verrouillage effectué, tout ce passe comme s'il n'y avait plus qu'une seule tâche dans le système. VRTX-32 devient alors un système mono-tâche. Ces fonctions sont :

Verrouillage : verrouillage du système par la tâche appelante. Elle est ensuite l'unique tâche à s'exécuter, même si elle crée d'autres tâches, celles-ci

ne débuteront pas leur exécution. Une fois le système dans cet état, l'unique tâche ne doit pas réaliser d'appels bloquants.

Déverrouillage : déverrouillage du système qui redevient multitâches dans le mode d'ordonnancement en vigueur au moment du verrouillage.

Signalons enfin qu'une tâche particulière, appelée "Idle task" (tâche nulle) est active quand aucune autre tâche ne l'est et que sa priorité est généralement très faible. L'utilisateur a la possibilité de remplacer la tâche nulle par une autre qu'il précise au moment du démarrage de VRTX-32. Cette fonctionnalité peut s'avérer utile pour certaines applications dans lesquelles il doit y avoir une activité de base ayant une basse priorité.

2.2.1.3. Les communications entre tâches.

VRTX-32 fournit, pour la communication entre les tâches, quatre types d'objets dont il assure la gestion. Ces derniers sont assez courants et leur utilisation combinée ou non permet d'implanter les schémas de communications les plus classiques (producteur/consommateur, serveur/client, ...) ou les plus originaux (tout ce que l'on peut imaginer).

Les moyens de communication mis à la disposition de l'utilisateur sont (dans tous les cas d'appels système bloquants, on a la possibilité de préciser un temps d'attente maximum) :

Boîtes aux lettres : une boite aux lettres est une zone mémoire de 32 bits. On peut poster une valeur, cette action n'est jamais bloquante (erreur si boite aux lettres pleine). Le retrait d'une valeur est bloquant si la boite est vide mais on peut aussi faire un retrait non bloquant. Si plusieurs tâches sont en attente d'une valeur, c'est la plus prioritaire qui est activée quand la boite n'est plus vide.

Files de valeurs : une file est une suite de valeurs de 32 bits chacune. On précise la taille d'une file lors de sa création, la gestion est "FIFO" ("First In First Out"). Comme pour les boites aux lettres, l'insertion d'une valeur n'est jamais bloquante. Le retrait l'est si la file est vide, mais on peut aussi faire un retrait non bloquant. On a la possibilité de savoir combien il y a d'éléments dans une file. On peut aussi déposer un élément qui va doubler tous les autres déjà présents dans la file. On peut également disposer d'une copie du

message en tête de file sans le retirer et sans que cela soit bloquant. On précise à la création de la file, comment les tâches sont débloquées si elles sont plusieurs en attente sur la file vide. Deux solutions : la tâche la plus prioritaire est réactivée (comme pour les boîtes aux lettres) ou, c'est la première tâche qui s'est mise en attente qui est réactivée (ensuite la deuxième, donc dans l'ordre des blocages).

Les ensembles de drapeaux : un ensemble de drapeaux est constitué de 32 drapeaux (un mot de 32 bits pour chaque ensemble). Pour l'action de consultation, on donne un masque qui sélectionne des drapeaux dans l'ensemble et, un paramètre indique si tous (attente "and") ou au moins un (attente "or") des drapeaux sélectionnés doivent être à la valeur 1. Si la condition est satisfaite, la consultation n'est pas bloquante. La consultation ne change jamais de valeurs dans l'ensemble. Un appel système permet de positionner des drapeaux dans un ensemble, un autre d'en dépositionner. Le réveil des tâches sur un ensemble de drapeaux se fait selon les priorités. Il se peut que plusieurs tâches soient réveillées en même temps.

Les sémaphores n-aires : un semaphore n-aire peut avoir une valeur entière comprise entre 0 et 65535. C'est un objet bien connu qui a été introduit par Dijkstra pour gérer l'exclusion mutuelle vis-à-vis d'une ressource partagée. On peut ici augmenter la valeur d'un sémaphore de un, ce qui n'est jamais bloquant. Une tâche peut décrémenter un sémaphore, mais si la valeur est nulle, la tâche est bloquée. Il y a, de la même manière que pour les files de valeurs, un choix possible entre deux possibilités pour le réveil des tâches bloquées sur un sémaphore.

On dispose par ailleurs d'un autre moyen d'échanger de l'information entre des tâches : la mémoire partagée (il n'y a qu'un seul espace d'adressage, comme nous l'avons dit plus tôt). L'utilisation combinée d'un des outils présentés et de la mémoire partagée permet des communications aussi complexes que l'on veut.

2.2.1.4. Les interruptions.

Nous allons tout d'abord préciser quelques points clés concernant les interruptions de manière générale.

Le concept d'interruption du processeur par les périphériques est un mécanisme de base sur lequel s'appuie l'écriture d'un système d'exploitation. Les appels système sont aussi généralement réalisés par des interruptions logicielles qui ont le même comportement que les interruptions matérielles mais sont engendrées par une instruction du processeur. Lorsque le processeur prend en compte une interruption, il passe généralement dans un mode de fonctionnement spécial (mode maître ou privilégié) dans lequel il lui est possible de masquer certaines interruptions (ce masquage peut éventuellement être automatique). De plus, certains registres sont la plupart du temps sauvegardés automatiquement ou, au moins, ne peuvent plus être modifiés (on dit qu'ils sont "gelés").

Ce mode de fonctionnement spécial permet au système d'exploitation d'assurer la gestion de toutes les ressources avec cohérence : sauvegarder des informations sur les tâches, gérer les outils de communication système, etc. Ceci tout en étant sûr de ne pas être interrompu à son tour dans certaines phases critiques. Le système gère aussi les périphériques en utilisant les interruptions, un périphérique informant le système que des données sont disponibles (ou l'inverse) en l'interrompant.

Dans l'informatique industrielle, les interruptions sont un mécanisme très souvent utilisé, notamment pour ce que l'on appelle l'acquisition de données (voir les exemples en introduction). La plupart du temps, le programme de lecture est prévenu que des données sont disponibles par une interruption. Quand une interruption de ce type survient, il est très important que le programme réagisse le plus vite possible (cette contrainte est à définir lors des spécifications). Les interruptions gérées par le système d'exploitation peuvent interférer avec les interruptions des périphériques et différer la réaction d'une application d'acquisition de données.

On peut distinguer deux types de systèmes d'exploitation en ce qui concerne la gestion des interruptions. Certains ne sont pas interruptibles tant que le processeur exécute des instructions en mode système (ou privilégié). Les appels système sont alors indivisibles : ils s'exécutent sans être interrompu une seule fois et si une interruption survient, son traitement est différé. On dit de ces systèmes d'exploitation qu'ils sont non préemptifs en mode privilégié. D'après ce que nous avons dit précédemment, on peut dire que le fait de ne pas permettre la préemption dans le mode privilégié est très

mauvais pour un système qui doit gérer des applications industrielles. VRTX-32 est un système préemptif dans le mode privilégié, c'est naturel puisqu'il est étudié pour le pilotage d'applications industrielles. Dans ce système, les interruptions sont inhibées le moins souvent possible, et la plupart des appels système sont interruptibles pendant la majeure partie de leur temps d'exécution.

Dans VRTX-32, l'utilisateur dispose d'une pile pour sauvegarder des registres au début d'une routine d'interruption. Une telle routine peut être directement activée par le matériel, sans passer par un pilote d'interruption comme c'est le cas dans beaucoup de systèmes d'exploitation. Ceci est fait pour favoriser, encore une fois, les applications industrielles en réduisant le temps de latence logicielle des interruptions.

Une routine d'interruption peut communiquer avec une tâche VRTX-32 via les outils fournis par le système. Par exemple, une routine d'interruption peut poster un message dans une boite aux lettres sur laquelle une tâche est en attente. Cette action déclenchera l'activation de la tâche lors du retour au niveau de l'exécution des tâches en respectant les priorités.

Néanmoins, pour pouvoir utiliser ces mécanismes, il faut respecter un certain protocole avec le système : il est nécessaire de lui signaler l'entrée et la sortie d'une routine d'interruption si on veut utiliser des appels système dans cette dernière. A la fin du traitement de l'interruption, on retourne explicitement dans le système en mode superviseur et c'est à ce moment que tous les appels système réalisés dans la routine d'interruption peuvent engendrer une ou plusieurs commutations de contextes. Ce protocole est assez lourd mais il l'est beaucoup moins que dans d'autres systèmes d'exploitation tout en permettant de ne pas trop dégrader les temps de réponse d'une tâche à une interruption.

Nous allons présenter un deuxième système adapté au contrôle d'applications industrielles : LynxOS.

2.2.2. Les extensions "temps-réel" de POSIX et LynxOS.

2.2.2.1. Description de POSIX.

Avant de commencer la description du système d'exploitation Lynx comme nous l'avons fait pour VRTX-32, nous allons préciser quelques points sur la norme POSIX ("Portable Operating System Interface", X pour la ressemblance avec UNIX). Ce dont nous allons parler ici est tiré essentiellement de [LYNX], de [JMR93] et [JMR94].

POSIX a le numéro 1003 dans le répertoire de l'IEEE ("Institute of Electrical and Electronics Engineers") dont une des vocations est la mise au point de normes industrielles. Pour l'instant POSIX n'est pas une norme internationale mais c'est le but que IEEE se donne et un autre organisme a déjà accepté une partie de POSIX sous la référence ISO 9945 ("International Standards Organization").

Cette norme est constituée de descriptions d'interfaces entre des programmes et un système d'exploitation. Ces interfaces spécifient les types des données, les noms de types, les noms de fonctions, le comportement des fonctions, etc. Nous ne parlerons ici que de POSIX associée avec le langage de programmation ANSI-C qui est normalisé. Les programmes développés en respectant les interfaces POSIX sont, dans une grande mesure, indépendants d'un système d'exploitation particulier. Par exemple, la plupart des systèmes UNIX proposent maintenant l'interface POSIX, mais on la trouve aussi sur le système OpenVMS® de la société DEC™ ainsi que sur le système LynxOS dont nous allons parler plus en détail.

Il y a plusieurs subdivisions de la norme, celle de base est 1003.1, celles concernant l'informatique "temps réel" sont 1003.4 ("Real time extentions") et 1003.4a ("Real time threads extentions"). La subdivision 1003.1 correspond à ISO 9945 et les extentions "temps réel" doivent y être intégrées dans un futur proche.

2.2.2.2. Description de LynxOS.

Une des premières caractéristiques de Lynx qui le différencie de VRTX-32 est le souci de respect de normes. Pour le langage de programmation, c'est

ANSI-C avec l'environnement de développement GNU (et toutes les bibliothèques de fonctions ANSI) dont nous avons parlé précédemment. Pour l'interface avec le système, Lynx a adopté POSIX dans son intégralité (1003.1, 1003.2, 1003.4, 1003.4a). De plus, Lynx est compatible avec les deux grands courants du système UNIX, à savoir : "System V" ("American Telegraph & Telephone") et BSD ("Berkeley Software Distribution").

Lynx est un système multitâches et multi-utilisateurs. On retrouve les mêmes niveaux d'exécution que dans VRTX-32 mais, une notion d'utilisateur que l'on peut apparenter à celle d'UNIX y est incluse. Il existe plusieurs plates-formes supportant Lynx, les plus importantes sont : les PC compatibles IBM™, les cartes Motorola MVME®, les stations SUN, certaines machines HP™, l'IBM RS6000 PowerPC®. Le système est livré sur bande ou sur disquettes suivant les plates-formes. Il se présente sous forme d'un noyau binaire, comme UNIX et, on dispose de tous les fichiers objets qui servent à le construire. Ceci est utile quand on veut intégrer une extension comme un pilote pour un type particulier de périphérique que Lynx ne supporte pas dans sa version de base (on fait une édition de liens de tous les fichiers systèmes avec les ajouts). Il est donc possible d'étendre le système d'exploitation de manière beaucoup plus générale et plus simple que VRTX-32. Les disques durs et autres mémoires de masse sont gérés par le système par l'intermédiaire de pilotes suivant le même schéma que dans UNIX.

La taille du noyau peut varier suivant les différentes configurations. La plus petite est d'environ 200 Ko et au maximum elle atteint 1 Mo (avec toutes les extensions disponibles). Il est possible de stocker le système en mémoire morte avec un système de fichiers minimal. De la même manière il est possible de démarrer une machine avec Lynx en le téléchargeant via un réseau local. Ces deux dernières possibilités permettent de développer des applications autonomes et minimales sur des machines constituées d'un environnement matériel réduit. De plus, la possibilité de démarrage via un réseau permet à plusieurs machines d'utiliser automatiquement la même configuration du système.

Même si Lynx intègre beaucoup de normes et de principes proches d'UNIX, le noyau du système est assez différent. En effet, Lynx est un système adapté au contrôle de l'exécution d'applications industrielles contrairement à UNIX. Le cœur du système est original et répond à peu de choses près aux mêmes

caractéristiques que VRTX-32. Par contre, l'environnement de développement est intégré à la machine cible. Dans notre laboratoire, nous possédons une carte Motorola mettant en œuvre le processeur MC68040 (MVME162) sous le contrôle de Lynx avec, entre autres, un disque dur et une interface vers le réseau local. La mise au point des programmes peut se faire directement sur la machine à l'aide de tous les outils habituels. Bien que cette machine équipée de Lynx constitue une plate-forme avec toutes les facilités modernes, elle est aussi une machine industrielle qui permet de contrôler des applications "temps réel" nécessitant le respect de contraintes strictes.

Pour ce qui est de la gestion de la mémoire, Lynx utilise une gestion paginée, la mémoire n'étant pas forcément virtuelle (suivant la configuration). Mais de toute façon, chaque programme en cours d'exécution (processus) possède un espace d'adressage propre que personne ne peut violer. Nous reviendrons sur une des particularités de la gestion de la mémoire virtuelle dans la partie suivante. Le système a son espace d'adressage et les pages lui appartenant sont toujours résidentes en mémoire vive. Si la mémoire virtuelle n'est pas engagée, toutes les pages de tous les processus (système ou non) sont en mémoire vive. Si elle l'est, une zone d'un disque dur doit être prévue pour stocker momentanément les pages non utilisées chassées de la mémoire vive en cas de besoin (fonction que l'on désigne couramment sous le nom de "swapping").

Une telle protection de l'espace d'adressage de chaque processus apporte une certaine tolérance aux fautes par le système. Un processus accédant à une adresse ne lui appartenant pas sera vraisemblablement terminé par le système d'exploitation sans avoir pu causer de dommages en perturbant d'autres processus. On comprend qu'un tel accès pourrait être dramatique sous notre version de VRTX-32 puisqu'il n'y a pas de protections.

Nous allons maintenant nous intéresser aux différentes possibilités que le système Lynx offre en matière d'exécution de programmes.

2.2.2.3. Processus et activités.

L'entité d'exécution de base de Lynx porte le même nom que dans le système UNIX, c'est le processus. Mais ici, un processus possède une priorité d'exécution qui est fixe en fonction du temps. La gestion de l'ordonnancement

est réalisée, comme dans VRTX-32, avec des files de tâches construites suivant les priorités et un quantum maximum (éventuellement infini) de temps d'exécution qui permet un minimum d'équité. Ici, le quantum est associé à une file tout entière donc à un certain niveau de priorité et il peut être différent suivant les niveaux. Lynx est donc équitable pour la répartition du temps d'exécution uniquement entre des tâches de même priorité (on peut rapprocher cette gestion de celle des classes de processus "temps réel" de la version "System V.4" d'UNIX).

Si la mémoire virtuelle est engagée, les pages mémoire d'un processus ne sont pas forcément résidentes en mémoire centrale. Lynx assure cette gestion par l'intermédiaire d'un processus système qui détient une priorité d'exécution comme tous les processus. Cela signifie que les priorités des processus influent sur la gestion de la mémoire virtuelle. Un processus ayant une priorité plus forte que celle du processus de gestion de la mémoire gardera ses pages en mémoire vive en permanence : seuls les processus de plus faible priorité que le gestionnaire de mémoire virtuelle peuvent avoir des pages non résidentes.

En plus de ce mécanisme automatique, il est possible pour un processus de verrouiller toutes ses pages en mémoire centrale indépendamment des priorités. Ces possibilités sont à utiliser avec prudence car elles peuvent conduire à la monopolisation de la mémoire centrale par un processus et, dans ce cas, les autres processus n'auront plus de mémoire pour s'exécuter (situation de famine vis-à-vis de la mémoire).

Le processus Lynx doit en fait être considéré comme l'unité d'encapsulation des ressources telles que les descripteurs de fichiers, l'espace d'adressage, le répertoire de travail, etc. L'unité représentant l'exécution d'un programme est différente et on la désigne sous le nom d'activité ("thread"). Un processus Lynx est donc composé d'une ou de plusieurs activités qui partagent les ressources communes du processus. En particulier, les activités d'un processus s'exécutent toutes dans le même espace d'adressage défini par le processus.

Comme pour les tâches de VRTX-32, chaque activité possède une pile privée pour les appels de fonctions et les variables automatiques ainsi qu'une priorité qui lui est attribuée lors de sa création. Un processus contient

toujours une activité spéciale : l'activité initiale (ce détail est particulier à Lynx, il n'est pas spécifié dans POSIX). Un processus Lynx avec son activité initiale constitue l'équivalent d'un processus au sens classique d'UNIX (sans distinction entre les ressources et l'unité d'exécution).

Comme une tâche de VRTX-32, une activité peut monopoliser tout le temps d'exécution du système et l'ordonnancement des activités n'est pas forcément équitable. Nous retrouvons cette caractéristique toujours pour les mêmes raisons car les activités sont destinées à être utilisées pour construire des applications industrielles respectant des contraintes strictes.

Lynx met à la disposition du programmeur ces objets que sont les activités en respectant la définition qu'en fait la norme POSIX. Pour plus de détails en ce qui concerne les différences entre processus et activités ainsi que sur les activités elles mêmes, on peut se référer à [JMR94]. Nous allons maintenant voir les principales opérations que l'on peut effectuer sur les activités via l'interface POSIX.

Création : création d'une activité. Il y a attribution par le système d'un identificateur d'activité unique qui pourra servir dans d'autres appels. On a la possibilité de donner des attributs à l'activité : priorité, taille de la pile, type d'ordonnancement (FIFO, ROUND ROBIN, DEFAULT), etc. Il existe des valeurs par défaut de ces attributs.

Terminaison d'une activité par elle-même : terminaison de l'exécution d'une activité avec émission d'un code de retour qui peut être une adresse de données dans la pile. Ces données sont alors accessibles par les autres activités et c'est pourquoi les ressources, telles que la pile, ne sont pas relâchées même si l'exécution est terminée.

Terminaison d'une activité par une autre : cette fonctionnalité permet à une activité de tuer une autre activité du même processus. L'activité tuée se retrouve dans le même état que si elle s'était terminée elle-même mais avec un code de retour spécial.

Détachement: libération de toutes les ressources associées à une activité (pile, ...). Les données disponibles dans la pile d'une activité terminée ne le sont plus après son détachement.

En ce qui concerne l'ordonnancement des activités (et des processus), il est possible de choisir entre plusieurs algorithmes. Celui par défaut est l'algorithme avec quantum de temps d'exécution qui tient compte fortement des priorités fixes. Dans ce cas, le quantum peut être changé pour chaque niveau de priorité. Les algorithmes "FIFO" et "ROUND ROBIN" sont les mêmes mais, avec un quantum général infini et invariable pour le premier et un quantum invariable par niveau de priorité dans le second.

Nous allons maintenant présenter quels moyens de communication Lynx offre pour les processus et pour les activités.

2.2.2.4. Les communications.

On trouve deux types de communications : un premier adapté aux processus et un second aux activités. Les outils de communication inter processus sont ceux proposés dans les versions "System V" et BSD du système UNIX. On trouve les IPC ("Inter Process Communications") et les "STREAMS" de "System V" ainsi que les "sockets" dans le domaine d'adressage UNIX des versions BSD. La norme POSIX a remodelé dans 1003.4 "Draft 9" les IPC de "System V". Les outils de communications sont essentiellement les mêmes (sémaphores, mémoire partagée, files de messages), avec quelques différences visant à en faciliter l'utilisation.

En ce qui concerne les activités d'un même processus, on trouve des outils simples pour plus d'efficacité dont l'utilisation combinée permet des schémas de communication complexes. Il ne faut pas oublier que les activités d'un processus disposent d'un moyen de communication qui est la mémoire partagée correspondant à l'espace d'adressage de ce processus. Voici une grande partie de ce que le système Lynx propose conformément à POSIX :

Les rendez-vous : une activité peut attendre la terminaison d'une autre activité du même processus en utilisant cette possibilité. Elle récupère alors le code de retour de l'activité qui s'est terminée. Ce code de retour peut être interprété comme une adresse avec laquelle il est possible d'accéder à des données dans la pile de l'activité terminée avant son détachement.

Des objets permettant l'exclusion mutuelle appelés "mutex" : c'est un outil comparable à un sémaphore binaire mais c'est l'activité qui prend possession du mutex qui doit le libérer. Une activité qui essaye de prendre un

mutex alors qu'il n'est pas libre, est bloquée jusqu'à ce que l'activité l'ayant pris le libère. Ce mécanisme est destiné à assurer l'exclusion mutuelle vis-à-vis d'une ressource partagée qui doit être manipulée en section critique.

Les variables de conditions : ce moyen s'utilise conjointement avec les mutex et permet de réaliser des synchronisations plus complexes. Après avoir pris un mutex, une activité se met en attente sur une variable de condition. A ce moment le mutex est libéré dans l'attente de la condition. Quand une activité signale la condition, le mutex est pris automatiquement par l'activité en attente et elle se réveille. Il est possible de réveiller une activité parmi plusieurs en attente sur la même condition en utilisant le même mutex. Il est aussi possible de réveiller toutes les activités en attente sur une même condition.

On peut ajouter qu'une activité peut évidemment utiliser les outils adaptés à la communication entre les processus. Lynx donne aussi la possibilité d'utiliser des outils de communication non décrits par la norme POSIX. Il existe par exemple un appel système permettant d'associer une adresse physique à l'espace d'adressage paginé d'un processus et, par conséquent, de ses activités. Ceci permet à une application d'accéder à des ressources physiques comme un bus externe, comme c'est le cas sur la carte MVME162 du laboratoire. En utilisant une mémoire vive sur le bus VME, deux processus ou activités peuvent échanger des données. Lynx fournit également une version rapide de sémaphores binaires et n-aires ainsi que des segments de mémoire partagée non conformes à POSIX.

Les signaux logiciels équivalents à ceux d'UNIX sont aussi un moyen de communication inter processus et inter activités. Un signal envoyé à un processus est pris en compte au hasard par une de ses activités qui accepte ce signal. Si aucune ne l'accepte, le signal est mémorisé et sera envoyé à une activité au hasard parmi celles qui l'accepteront plus tard. Un signal envoyé à une activité particulière d'un processus ne peut pas être pris en compte par ses autres activités.

2.2.2.5. Les interruptions.

Contrairement à VRTX-32, Lynx ne permet pas à une application utilisateur de prendre en compte des interruptions matérielles directement.

D'après les concepteurs du système, ce mode de fonctionnement fait que c'est l'utilisateur qui doit la plupart du temps réaliser des services que le système d'exploitation devrait assurer. La pratique de VRTX-32 nous a montré que si ce principe avait des défauts (ceux dor nous avons parlé dans la partie équivalente sur VRTX-32) , il avait également des avantages. Par exemple, s'il est pas nécessaire de faire appel au système dans la routine d'interruption cela peut être avantageux, surtout si les interruptions ne passent pas pa un pilote car on dispose alors d'un temps de réponse qui est celui du matériel.

' - seul moyen de gérer des interruptions dans Lynx est de passer par des p périphériques. Un pilote est essentiellement un ensemble de rou s ("open","read","write","close",...). Une routine permet à une application de faire des lectures sur un périphérique et une autre sert à faire des écriture ns le mode de fonctionnement habituel, la lecture est bloquante s pas de données disponibles. L'écriture peut aussi être bloquante, si la zone tampon associée au périphérique est pleine par exemple.

Si le périphérique est capable de provoquer des interruptions matérielles, il va utiliser ce principe pour prévenir le processeur que des données sont disponibles ou que la zone tampon n'est plus pleine. Le pilote du périphérique gère ces interruptions et les blocages/déblocages des applications faisant des entrées/sorties.

Or, les travaux effectués dans les routines d'interruption d'un pilote sont la plupart du temps des transferts de données ou des dialogues. Ceux-ci peuvent prendre du temps et dans ce cas la routine d'interruption peut monopoliser le temps d'exécution. Dans le domaine d'utilisation de Lynx, ce principe est plutôt gênant. Pour remédier à cela tout en mettant des outils de haut niveau à la disposition des programmeurs, Lynx permet de créer des activités particulières : les activités système ("system threads"). De telles activités privilégiées, contrairement aux autres, s'exécutent dans l'espace d'adressage du système Lynx.

Un pilote a la possibilité d'être écrit avec une routine d'interruption minimale débloquant une activité système qui va effectuer le travail qui aurait traditionnellement dû se faire dans la routine d'interruption. Une activité système a une priorité et est gérée comme les autres activités ; une action plus prioritaire qu'elle peut donc l'interrompre. Par exemple, un accès disque

en lecture peut être interrompu par une acquisition de données prioritaire si le pilote du disque est constitué d'une activité système de priorité plus faible que celle gérant l'acquisition. Le fait d'être dans l'espace d'adressage du système permet en plus à l'activité une communication simple et rapide avec ce dernier.

Le moyen de communication entre les routines d'interruption et les activités système est le sémaphore n-aire (ce moyen n'est pas décrit dans la norme POSIX, il est particulier à Lynx). Il existe un pilote d'interruption sur lequel toutes les interruptions sont dirigées. Quand une interruption survient, le pilote est exécuté, son travail consiste à appeler une routine qu'un pilote de périphérique a installée lors de son initialisation. Ce principe permet une utilisation des interruptions à un niveau de programmation assez élevé. Mais, introduit un délai supplémentaire entre le moment où une interruption survient et le moment où l'exécution de la vraie routine d'interruption débute.

Après cette description rapide des fonctionnalités et des principes de Lynx ainsi que des extensions "temps réel" de POSIX, nous allons présenter sommairement le langage ADA.

2.2.3. Le langage ADA.

Les informations présentées ici viennent de [Booch86] et de [ADA83].

2.2.3.1. Description du langage et des tâches ADA.

Le langage ADA est un langage de programmation complet comme le langage C ou le PASCAL. Il est de haut niveau et donne au programmeur beaucoup de facilités. L'un de ses domaines d'application privilégié est l'informatique industrielle. Nous ne parlerons ici que des fonctionnalités de ADA en tant qu'exécutif permettant la spécification de programmes composés de tâches asynchrones pouvant communiquer. Il existe deux normes décrivant ce langage : ANSI et ISO.

Il est possible de spécifier des taches dans un programme ADA. L'exécution indépendante est réalisée soit par un noyau de système ajouté à chaque programme exécutable, soit par un système d'exploitation (système hôte) non fourni par le compilateur ADA. Dans le premier cas, le compilateur ADA construit un exécutable à partir des fichiers sources du programme et

ensuite, il y ajoute un système d'exploitation réduit qui contrôle les tâches pendant l'exécution. Dans la seconde méthode, le compilateur est lié à un système d'exploitation et il associe automatiquement une tâche système à chaque tâche ADA pendant la compilation. Au moment de l'exécution, c'est le système d'exploitation hôte qui contrôle l'exécution des tâches (cela peut être UNIX, par exemple, ou un système équivalent).

Il est prévu dans le langage de donner des priorités aux tâches sous forme d'entiers ("pragma PRIORITY"). Ces priorités ont une signification différente suivant l'implantation, avec système hôte ou non. Les priorités sont utiles pour indiquer l'importance d'une tâche mais pas pour la synchronisation. L'algorithme d'ordonnancement n'est pas précisé dans la norme. Celle-ci ne précise pas non plus comment départager deux tâches qui ont la même priorité.

Un programme ADA est constitué de tâches dont la gestion est comparable à celle des processus UNIX ou des activités Lynx ou encore des tâches VRTX-32. C'est pourquoi nous avons classé ADA avec les systèmes d'exploitation.

2.2.3.2. Les communications entre tâches ADA.

Le langage fournit essentiellement un moyen de communication : le rendez-vous binaire avec passage d'information. Une tâche ADA peut accepter des rendez-vous avec une autre tâche via l'opérateur "accept". Le rendez-vous se fait quand au moins une tâche le sollicite et qu'une autre tâche est en attente sur ce rendez-vous avec l'opérateur "accept" (les priorités interviennent si plusieurs tâches sollicitent le rendez-vous). Un rendez-vous est toujours bloquant si une seule tâche participante est présente. Il peut y avoir passage d'information de la tâche sollicitеuse vers la tâche en attente un peu comme un passage de paramètre lors d'un appel de fonction.

On peut, avec l'opérateur "select", se mettre en attente sur plusieurs rendez-vous en même temps suivant des conditions sur des valeurs de variables locales à une tâche. Un seul des rendez-vous possibles se fera à la fois et si aucun n'est réalisable, il est possible de spécifier une action par défaut. Si plusieurs rendez-vous sont possibles en même temps, les priorités des tâches en attente interviennent : c'est la plus prioritaire qui sera

débloquée. Si toutes les tâches sont à la même priorité, c'est l'implantation de l'ordonnancement qui détermine celle qui est réactivée.

Il est aussi possible d'utiliser la mémoire partagée en ADA. On peut, à l'aide d'une directive de compilation, indiquer qu'une donnée est accessible par plusieurs tâches. Cette alternative peut être utile pour réaliser des schémas de communication complexes mais normalement les rendez-vous avec passage d'information suffisent.

2.2.3.3. Les interruptions.

Il est permis d'associer une interruption matérielle à un rendez-vous ADA. Une seule tâche détient alors le droit de se mettre en attente sur ce rendez-vous. Quand l'interruption survient, le rendez-vous est réalisé et la tâche en attente se réveille. La routine d'interruption (donc la tâche) a la possibilité de récupérer des paramètres en lecture lors du rendez-vous. En fait, l'interruption est vue comme une tâche avec une priorité maximum supérieure à celles de toutes les tâches. Par conséquent, le rendez-vous associé à une interruption est très prioritaire. L'algorithme d'ordonnancement des tâches n'intervient pas lors de la prise en compte d'une interruption.

Le principe de prise en compte des interruptions n'est pas présenté dans la norme du langage. Cependant, étant donné le mode de gestion de l'exécution des tâches, on peut supposer que la plupart des implantations suivent un principe analogue à celui des systèmes d'exploitation classiques. Ce principe peut être, par exemple, celui de VRTX-32 (prise en compte des interruptions sans passer par un pilote) ou celui de Lynx.

2.2.4. Résumé des fonctionnalités.

Nous allons ici résumer les caractéristiques qui nous semblent intéressantes dans les environements d'exécution que nous avons présentés puisqu'elles permettent de réaliser l'implantation d'applications suivant les contraintes que nous nous sommes imposées.

Mis à part le principe sans exécutif, ils offrent tous la possibilité d'exprimer des applications sous forme de tâches plus ou moins indépendantes pouvant communiquer et se synchroniser. Cette fonctionnalité est fondamentale pour le développement de programmes structurés dans le domaine qui est le nôtre.

Les systèmes d'exploitation présentés sont préemptifs en mode privilégié ou intègrent un principe équivalent. Puisque les applications que nous voulons construire réagissent souvent à des interruptions, cette caractéristique est bien adaptée.

Avec ou sans la compilation croisée, il est possible de programmer dans un langage de haut niveau avec des outils de mise au point. L'assembleur peut n'être utilisé que dans des cas précis, en tous cas pas de manière importante. Il faut quand même signaler que la portabilité des applications d'un système d'exploitation à un autre n'est pas encore simple, même si elles sont écrites dans un langage évolué et normalisé qui permet bien de respecter une norme pour l'interface avec le système.

Une réponse possible à ce problème de portabilité, qui entraîne aussi d'autres avantages, réside dans une famille de langages de programmation que nous allons décrire

2.3. Les "Langages à Parallélisme Compilé" (LPC).

Les LPC ont fait leur apparition assez récemment, au début des années 80 pour Esterel. Nous ne présenterons que certains de ces langages en commençant par Esterel qui est un des plus anciens. Ensuite, après les descriptions, nous ferons une comparaison entre l'approche "langage évolué avec système d'exploitation" et l'approche LPC.

Suite à cette comparaison, nous décrirons le langage que nous avons construit qui adopte à peu de choses près les principes de ceux que nous allons voir. Notre langage représentera les LPC dans la partie d'évaluation. Les informations que nous allons présenter ici sont tirées de [Halbwachs91] en ce qui concerne Esterel, Lustre et Argos. Voyons d'abord les concepts de base avec Esterel.

2.3.1. Le langage Esterel.

2.3.1.1. Description sommaire du langage.

Esterel a été développé par une équipe commune au Centre de Mathématiques Appliquées de l'École des Mines et à l'INRIA (Institut National de Recherche en Informatique et Automatique). C'est un langage de programmation qui permet essentiellement la description et la validation de comportements et, c'est pourquoi il n'offre pas de structures de données complexes ou d'autres outils que l'on trouve dans des langages classiques. Ce langage permet somme toute de conserver les habitudes de programmation classiques en donnant la possibilité de structurer les applications avec des tâches qui communiquent entre elles comme nous allons le voir plus loin.

Un programme Esterel décrit un système réactif, c'est-à-dire, un système qui réagit à des stimuli en provenance de l'extérieur et qui, en réponse, produit à son tour des stimuli. Le résultat d'une compilation Esterel est un automate fini codé par un programme dans un langage de haut niveau appelé langage hôte (C, ADA, ...). Les transitions de l'automate sont réalisées par l'intermédiaire d'un moteur exprimé dans le langage hôte. L'exécution de ce programme, après compilation du fichier source dans le langage hôte, réalise *le comportement décrit par le code Esterel de départ.* Cette exécution se fait par des appels successifs au moteur de l'automate. L'utilisation d'un langage hote permet à un programme Esterel d'être facilement portable d'une machine vers une autre.

Le systeme reactif ne constitue généralement pas une application dans sa totalité. Normalement, une application est composée d'un ou plusieurs noyaux réactifs et d'un programme principal qui fait fonctionner le ou les noyaux réactifs en appelant regulièrement le ou les moteurs des automates respectifs tout en assurant d'autres fonctions (initialisations, gestion des interruptions, *gestion des entrées sorties, gestion des erreurs, ...).*

La syntaxe du langage est comparable à celle du langage PASCAL ou bien de ADA même s'il n'a pratiquement rien en commun avec eux en ce qui concerne les principes. Il existe une variante d'Esterel dont la syntaxe est *proche de celle du langage C ("Reactive C")* ce qui nous semble mieux

correspondre aux habitudes des programmeurs dans le milieu de l'informatique industrielle.

Il est possible dans un programme Esterel de définir des variables locales, d'appeler des fonctions du langage hôte, de mettre des instructions en séquence, de décrire des boucles, des tests, des expressions simples avec des opérateurs courants. Des modules peuvent être définis et inclus dans un programme ce qui donne la possibilité de structurer l'application que l'on veut développer (une inclusion est une recopie littérale d'un module). Les variables globales n'existent pas dans Esterel.

2.3.1.2. Les tâches et la communication.

A l'aide d'un opérateur du langage (noté "||"), il est possible de spécifier des tâches s'exécutant en parallèle dans un programme Esterel. L'outil de communication interne entre les tâches est le même que celui entre le *système réactif et l'extérieur : le signal. Il s'agit de l'unique moyen de* communication entre les tâches dans Esterel : en particulier le concept de variable partagée n'y existe pas. Un signal peut être en provenance ou dirigé vers l'extérieur du système réactif (les stimuli en entrée ou en sortie dont nous avons parlé précédemment). Il peut être utilisé à seule fin de communication interne entre tâches. Une tâche peut envoyer un signal (opérateur "emit") et, à ce moment, toutes les tâches en attente de ce signal (avec "do halt watching <sig>" par exemple) le prennent en compte.

L'attente d'un signal peut être bloquante mais une tâche a la possibilité de tester la présence d'un signal à un moment donné (avec "present"). Il est permis d'associer une valeur à un signal : l'émission peut alors engendrer une modification de la valeur courante. Une tâche peut récupérer la valeur associée à un signal quand elle le prend en compte. Un signal externe peut être emis explicitement par une tâche comme un signal interne et tout se passe comme si l'émission provenait de l'extérieur.

L'interface entre un système réactif et le monde extérieur est simple. L'émission par le système d'un signal déclaré comme étant extérieur est transformée en appel de fonction dans le langage hôte : le programmeur doit fournir une fonction pour chaque signal en sortie du système réactif. Inversement, le compilateur Esterel construit une fonction dans le langage

hôte pour chaque signal extérieur en entrée. L'appel de cette fonction pendant l'exécution provoquera l'émission du signal pour le système. Un signal externe peut être déclaré comme une entrée et une sortie, il peut alors être pris en compte ou émis par le système réactif (il y aura une fonction en entrée et une en sortie).

Le fonctionnement d'un système réactif avec un seul noyau peut donc se résumer ainsi : un programme principal émet des signaux vers le noyau reactif en appelant les fonctions associées, il appelle ensuite le moteur de l'automate qui à son tour va produire les réactions en appelant les fonctions des signaux émis suivant les transitions effectuées. Ce mode de fonctionnement avec automate fini est assez efficace en ce qui concerne le temps d'exécution et donc intéressant pour des applications d'informatique industrielle.

Nous allons maintenant présenter succinctement comment le compilateur Esterel gère le parallélisme.

2.3.1.3. La gestion du parallélisme.

La gestion des tâches est ici complètement différente de celle utilisée dans les systèmes d'exploitation comme ceux que nous avons présentés plus haut. Nous parlons ici de parallélisme compilé, c'est à dire que si plusieurs tâches sont décrites dans un programme Esterel, après compilation en automate il n'existe plus qu'une tâche dont l'exécution simule l'exécution en parallèle des tâches d'origine. Le fonctionnement de cette tâche unique est complètement décrit dans l'automate fini résultat. La condition indispensable pour que ce principe puisse s'appliquer est que le nombre de tâches soit fixé avant la compilation. En d'autres termes, il n'est pas possible de créer des tâches pendant l'exécution d'un programme Esterel (pas de création dynamique comme il est possible de le faire avec un système d'exploitation).

Un système d'exploitation simule l'exécution concurrente de tâches indépendantes en gérant la répartition du temps au moment de l'exécution alors que, dans le cas d'un langage comme Esterel, la répartition du temps d'exécution entre les tâches se fait au moment de la compilation. Chaque tâche Esterel est en fait considérée comme un automate fini. Le compilateur Esterel combine les automates d'un programme pour construire un autre

automate qui décrit le fonctionnement en parallèle des automates initiaux. Cette combinaison est appelée "*produit synchronisé ou synchrone*" et nous reviendrons plus en détail sur celui-ci lors de la présentation du langage que nous proposons.

Les automates finis sont des objets mathématiques bien définis dont l'utilisation permet de formaliser de manière commode des comportements de programmes. De nombreuses propriétés intéressantes sont connues et vérifiables automatiquement pour ces objets. La production d'un automate fini comme résultat de compilation permet de pouvoir vérifier des propriétés sur ce dernier (blocages, déterminisme, ...). Par conséquent, Esterel permet non seulement de spécifier des programmes mais aussi de valider certaines propriétés de leur comportement par l'intermédiaire des automates associés. Des outils de vérifications automatiques de propriétés ont été développés à l'INRIA.

Mais revenons à un point crucial pour les applications industrielles en présentant comment gérer les interruptions avec Esterel.

2.3.1.4. La gestion des interruptions.

Le schéma de fonctionnement d'un système réactif typique décrit avec Esterel vise à permettre de répondre à des interruptions matérielles. En effet, le principe de communication de base étant le signal, il sera tout naturel d'associer un de ceux-ci à une interruption. Pour signaler l'occurrence d'une interruption à un noyau réactif, le programmeur utilise une fonction associée à un signal en entrée du système.

Le mécanisme est simple et la prise en compte de l'interruption par l'automate doit être assez rapide. On peut noter que les fonctions utilisées pour indiquer la présence d'un signal à l'automate ont un rôle comparable à celui des pilotes d'interruption du système Lynx. Mais, une différence essentielle avec les systèmes d'exploitation est que, ici, le système réactif a la possibilité d'accepter en permanence les interruptions matérielles car à aucun moment il n'est nécessaire qu'elles soient masquées. Nous reviendrons sur ce point plus bas dans la comparaison entre les systèmes et les LPC.

2.3.2. Autres langages de la même famille.

D'autres langages ont été développés, notamment par le Laboratoire de Génie Informatique de l'IMAG de Grenoble. Nous allons les présenter très sommairement.

2.3.2.1. Lustre.

Comme pour Esterel, le résultat de la compilation d'un programme Lustre est un automate fini codé dans un langage hôte de haut niveau. Par contre, le langage Lustre ne se présente pas comme les langages de programmation classiques. Un programme Lustre est une suite de déclarations d'opérateurs (nœuds) avec des entrées et des sorties. Un opérateur peut être utilisé dans la déclaration d'un autre opérateur.

L'utilisation des opérateurs est induite par le fait que Lustre repose sur le principe de programmation "flot de données". Celui-ci consiste à voir un programme sous forme d'équations implantées comme des réseaux d'opérateurs. Par exemple, un opérateur "+" a deux entrées et une sortie qui peut être connectée à un autre opérateur et ainsi de suite. Le fonctionnement d'un réseau dépend de la présence des données en entrée de chaque opérateur. Un opérateur ne peut s'exécuter que si toutes ses entrées sont présentes. Ce modèle d'exécution intègre implicitement du parallélisme si on considère qu'à un moment donné, tous les opérateurs pouvant s'exécuter le font en même temps.

L'exécution d'un programme Lustre est rythmée par au moins une horloge de temps. En fait, à chaque top de l'horloge, s'il n'y en a qu'une, tous les opérateurs applicables sont exécutés. Un moteur est utilisé pour faire fonctionner l'automate résultat d'une compilation suivant le même principe que dans Esterel.

Dans la partie d'évaluation de quelques solutions de développement, nous présenterons rapidement le langage LabVIEW® qui suit le principe "flot de données" pour la gestion de l'exécution (voir [LabVIEW] pour plus de détails). C'est avec ce langage que nous réaliserons le système de mesure des temps d'exécution.

2.3.2.2. Argos.

Argos est pratiquement équivalent à Esterel. La différence principale se situe au niveau de sa syntaxe qui est particulièrement originale puisqu'elle est graphique. Un programme Argos est composé d'automates finis exprimés sous forme graphique et il est par exemple possible de combiner des automates avec des opérateurs de mise en parallèle (suivant le même principe que Esterel). Une combinaison d'automates donne un automate résultat qui peut à son tour être combiné avec d'autres.

Une compilation Argos produit un automate fini exprimé dans un langage hôte comme pour Esterel et Lustre.

2.4. Comparaison des solutions "langage/système" et LPC.

Toutes les solutions de développement que nous avons envisagées permettent de programmer les applications par lesquelles nous sommes intéressés et ce en utilisant des fonctionnalités classiques (plusieurs tâches qui communiquent et se synchronisent). Après ces descriptions de plusieurs environnements de développement et d'exécution d'applications, nous allons comparer les principes des approches langage/système et langage à parallélisme compilé. Cette comparaison est "théorique" et nous tenterons de vérifier nos dires dans la partie d'évaluation.

2.4.1. Inconvénients de l'approche LPC.

Une limitation des LPC par rapport aux systèmes est liée à l'opération qui sert à la compilation du parallélisme : le produit synchrone d'automates finis. En effet, la taille du résultat d'un produit synchrone en terme de nombre d'états et de transitions, croit très vite en fonction de la taille des automates opérandes, de leur nombre et des connexions entre ces automates et le monde extérieur (stimuli en entrée). Mais, cette limitation est rarement atteinte dans la plupart des applications courantes. Cette limitation est inexistante quand on traite un problème en utilisant un système d'exploitation.

De la même manière, la complexité du produit synchrone entraîne une taille mémoire plus grande pour l'exécutable d'un programme exprimé dans un LPC que pour un programme équivalent sous le contrôle d'un système

d'exploitation. Cependant, un système d'exploitation consomme lui-même un espace mémoire de taille non négligeable par rapport à celle utilisée par les applications qu'il contrôle.

Une différence fondamentale entre l'approche système et l'approche LPC est l'aspect dynamique que l'on trouve dans un cas et pas dans l'autre. On peut généralement créer ou tuer des tâches de manière dynamique dans un système d'exploitation. Avec les LPC cela est impossible puisque tout ce qui concerne la simulation du parallélisme est traité de manière statique pendant la compilation. Cela restreint l'utilisation de ces langages à des problèmes ne nécessitant pas de création dynamique de tâches.

2.4.2. Avantages des LPC.

Les LPC permettent de développer des programmes sans que ceux-ci fassent appel aux services d'un système d'exploitation pendant leur exécution. Un programme directement écrit dans un de ces langages est théoriquement complètement portable d'une machine vers une autre : ce n'est qu'au niveau des routines écrites en assembleur (interruptions, traitements rapides, ...) que peuvent se poser des problèmes de portabilité. Ces problèmes existent dans tous les cas (système ou LPC) si l'on veut passer d'une machine à une autre. L'utilisation d'un langage hôte permet de compiler la même application pour plusieurs machines différentes et d'utiliser soit le principe de développement croisé, soit intégré à la machine cible.

Même sans changer de machine, il peut y avoir avantage à utiliser ces langages. Par exemple, un changement de version de matériel, si celui-ci est sous le contrôle d'un système d'exploitation, peut entrainer des adaptations du système difficiles et fastidieuses (ajout d'un coprocesseur arithmétique par exemple). Avec un langage à parallélisme compilé, le problème est reporté au niveau du compilateur C (ou autre langage hôte) : il suffit donc de compiler avec une option permettant de prendre en compte le coprocesseur.

Les LPC permettent la description d'applications sous forme de tâches communiquant entre elles, comme dans le cas d'un système d'exploitation conjugué avec un langage de programmation. Ce sont des langages de haut niveau qui se suffisent à eux-mêmes. Ceci est absolument nécessaire de nos jours où il est de moins en moins intéressant de programmer avec des outils

de bas niveau. On ne continue à utiliser l'assembleur que dans quelques cas : pour repondre aux *exigences de rapidité, de coûts ou de simplicité*

Une différence essentielle entre les LPC et les systèmes d'exploitation se situe au niveau du traitement des interruptions. Cette différence est très importante dans le domaine d'application qu'est l'informatique temps réel. Dans un systeme d'exploitation, même conçu pour le temps réel, il y a des séquences d'instructions durant lesquelles on interdit au processeur de prendre en compte des interruptions. L'objectif est de preserver la cohérence du système pendant certaines sections critiques comme la gestion d'une pile ou d'une liste chaînée, la gestion de certains appels système qui doivent être indivisibles vis-à-vis des tâches, etc. Ceci s'avère très gênant dans un environnement temps réel ou l'on veut très souvent minimiser le temps de prise en compte des interruptions (pour atteindre idéalement celui du matériel). Avec un LPC, on a la possibilité de permettre la prise en compte des interruptions à tout instant car le parallelisme étant compilé on a deux niveaux d'exécution : la tâche représentant les tâches exprimées dans le langage et les interruptions. Il n'y a donc pas de problème de cohérence à assurer comme dans un système d'exploitation.

De plus, la communication entre une routine traitant une interruption et un système est souvent très coûteuse en temps (passage d'une information à une tâche) car elle engendre nécessairement un nouveau choix de la tâche à activer dans le système. Avec un LPC, la communication entre ces deux niveaux se fait par des variables ou des fonctions globales donc plus simplement (une valeur de chaque variable ou fonction véhicule l'information pour les tâches). On peut aussi déduire de cette remarque que les routines d'interruptions d'une application exprimée dans un LPC peuvent être très courtes et donc favoriser le niveau d'execution des tâches qui se partagent le processeur.

Dans un système d'exploitation multitâches sur une machine monoprocesseur la gestion des tâches consiste en un partage des ressources de la machine et principalement du processeur. Lorsque le système alloue la ressource "processeur" à une tâche T1, il doit sauvegarder toute l'information concernant la tâche active T2 du processeur vers la mémoire et ensuite, recharger l'information concernant T1 de la mémoire vers le processeur. C'est là le minimum de la gestion des tâches qu'on appelle communément

commutation de contexte. Or cette commutation est très coûteuse car elle comporte deux fois plus d'accès à la mémoire qu'il y a de registres dans le processeur. Avec un LPC cette commutation est inexistante car le parallélisme est compilé et donc le partage du processeur entre les tâches est déterminé une fois pour toutes durant la compilation sans créer de surcharge au moment de l'exécution.

La remarque précédente indique que le partage du processeur entre les tâches d'un programme est déterminé à la compilation. On règle ce partage dans le programme en utilisant les outils de communication fournis par le langage. Ceci peut être intéressant et correspond à l'attribution de priorités aux tâches d'une application dans un système d'exploitation. Les LPC permettent de régler finement la répartition du temps entre tâches à l'écriture du programme alors qu'avec un système d'exploitation les priorités sont gérées avec moins de precision mais elles peuvent changer pendant l'exécution.

On a vu qu'un des principes de ces langages est de produire du code C (ou autre) comme résultat d'une compilation. Ceci fait que toutes les tâches décrites dans un programme sont contenues dans un programme qui les simule. Quand on compile le code C produit par le compilateur d'un LPC avec un compilateur C, ce dernier a une vision globale de toutes les tâches et il peut donc optimiser de manière générale la gestion des données, des registres, de l'assembleur engendré. Or pour une application équivalente sous le contrôle d'un système d'exploitation, l'optimisation du compilateur C ne peut être que locale à chaque tâche puisqu'elles sont décrites séparément. De nos jours, les compilateurs C utilisent des techniques d'optimisation très sophistiquées dont les applications programmées avec un LPC peuvent bénéficier de manière plus importante que lorsqu'elles sont destinées à s'exécuter sous le contrôle d'un système d'exploitation.

2.4.3. Conclusion de la comparaison.

On voit, après cette comparaison, que bien évidemment les langages à parallélisme compilé ne peuvent pas résoudre tous les problèmes. Le domaine d'application est restreint, mais l'informatique temps réel semble être un de ceux auquel ils sont bien adaptés. Ils permettent de structurer les applications, de les rendre portables et d'aboutir à une maintenance simple.

La rapidité d'exécution d'une application programmée directement dans un LPC doit pouvoir être au moins égale à celle d'une application équivalente, s'exécutant sous le contrôle d'un système d'exploitation temps réel. Nous étudierons en détail ce point dans la dernière partie de notre travail en évaluant certains temps de réponse pour une application représentative de celles que l'on développe couramment au laboratoire dans le cadre des expériences de physique des hautes énergies.

Après avoir présenté différentes méthodes de développement, nous allons proposer la définition d'un LPC suivant les règles générales de ces langages et nous l'utiliserons sur des applications pratiques. Ce langage va nous permettre de faire la comparaison entre les différentes approches dans la dernière partie de ce travail. Il doit également nous permettre de prouver qu'il est possible de conserver les habitudes de programmations issues de l'utilisation courante de systèmes d'exploitation couplés à des langages évolués tout en apportant les avantages des LPC surtout en ce qui concerne l'efficacité d'exécution.

3. Le langage llC.

3.1. Les principes de llC.

3.1.1. Description générale et syntaxe.

3.1.1.1. Description

La syntaxe de llC est celle du C : ce choix a été fait pour des raisons pratiques puisque le domaine d'application de llC est l'informatique industrielle et scientifique dans laquelle le langage C est de plus en plus utilisé. Dans la suite, on considère que le lecteur est familiarisé avec le langage C décrit dans [K&R] ainsi qu'avec quelques fonctions standards de la norme ANSI.

Une première fonctionnalité supplémentaire de llC par rapport au C est la possibilité d'exprimer que des instructions sont à exécuter en parallèle. On appellera communément "tâche" un bloc d'instructions C mis en parallèle avec au moins un autre bloc. Nous avons retenu ici le même principe de parallélisme compilé que dans les autres LPC et nous considérerons que les programmes llC sont destinés à s'exécuter sur des machines ayant un seul processeur. Le résultat d'une compilation d'un programme llC est un programme C codant un automate fini. L'exécution du code engendré par sa compilation réalise le déroulement en parallèle des tâches du programme llC.

Des moyens de communication entre les tâches sont fournis car il est très rare qu'une application ne soit constituée que de tâches indépendantes. En IIC on dispose de trois moyens de communication : la mémoire partagée, le rendez-vous et la possibilité pour une tâche d'interrompre une ou plusieurs autres tâches. Les deux premiers moyens de communication sont proches de certains de ceux que l'on trouve dans les systèmes d'exploitation ou dans ADA. Le dernier est comparable à l'instruction "trap <sig> in <inst>" d'Esterel telle qu'elle est décrite dans [Halbwachs91].

3.1.1.2. Les déclarations.

Nous allons maintenant présenter de manière plus détaillée la syntaxe du langage. La grammaire qui va être décrit est en fait très proche de celle donnée dans le manuel de référence du langage C [K&R]. On a dans IIC la possibilité de déclarer des identificateurs de rendez vous au début d'une fonction C. Ils constituent des noms de rendez-vous pouvant être utilisés dans des instructions spécifiques que nous présenterons plus loin. Il n'est pas permis de déclarer des variables au début d'un bloc C se trouvant à l'intérieur d'une fonction : on ne peut déclarer que des variables globalement ou en déclarer au début d'une fonction.

Les règles de grammaire qui vont suivre sont décrites sous une forme courante que l'on peut rapprocher du formalisme de l'utilitaire "yacc" d'UNIX. Le terminal "**meeting**" joue le rôle d'un type de donnée qui annonce une liste de déclarations de rendez-vous. "Id" représente un identificateur de rendez-vous et "**constant**" le nombre de tâches devant être présentes au rendez-vous pour que celui-ci se réalise.

Les déclarations sont décrites par ces règles sachant que la variable "declaration" représente toute déclaration possible dans le langage C tel que décrit dans [K&R] :

declaration_list : meeting_declaration

declaration_list : declaration_list meeting_declaration

declaration_list : declaration

declaration_list : declaration_list declaration

meeting_declaration : **meeting** meeting_list ;

meeting_list : **id : constant**

meeting_list : meeting_list , **id : constant**

Pour illustrer ceci, voici des exemples de déclarations valides :

```
static struct {
    int
        i,
        j;
} buff[256]; /* Variable globale. */

func(p0,p1)
int p0; /* Paramètres. */
char *p1;
{
int i; /* Variable locale. */
meeting rdv:2; /* Rendez-vous binaire. */
meeting
    rdv1:4, /* Rendez-vous nécessitant 4 tâches pour se réaliser. */
    rdv2:3; /* Rendez-vous n'en nécessitant que 3. */

    /* corps de la fonction. */
}
```

Les variables "buff", "p0", "p1", "i" seront visibles dans chaque tâche du corps de la fonction "func". Le rendez-vous "rdv", par exemple, sera utilisable dans chaque tâche de "func".

3.1.1.3. Les instructions.

Comme pour les déclarations de variables, llC respecte la grammaire du C pour les instructions. Cependant, pour le moment, l'instruction "switch()" n'est pas permise (en effet, elle n'est pas absolument nécessaire pour réaliser ce qui nous intéresse dans ce travail).

En llC, les expressions sont exactement celles du langage C et elles permettent de définir les instructions simples du langage. Une instruction simple a la forme : "expression ;". Les expressions sont une combinaison des opérateurs usuels (arithmétiques, logiques, bit à bit, ...), de constantes et de variables qu'il faut déclarer. Pour plus de détail en ce qui concerne les expressions en C on consultera le manuel de référence.

Il est par ailleurs possible de grouper des instructions afin qu'elles soient considérées comme des instructions simples. Nous verrons l'utilité de ceci dans la partie concernant la sémantique de llC. Voici la description de l'instruction de groupement avec le mot clé "**group**". Notez que la variable "statement" se trouve dans la grammaire originale.

<statement : expr_statement

expr_statement : **group** { <anything> }

Ici '<anything>' représente n'importe quelle suite de caractères. On considère tout ce qui se trouve entre les deux accolades comme étant une instruction simple. Voici un exemple de séquence d'instructions simples construite avec des expressions et un groupement :

```
func()
{
int
    i,
    j;

    i += 1 ;

    /* Ceci est une seule instruction simple. */
    j = ( function(&i,&j) , (j ? (i/j) : (i)) ) ;

    /* Ceci est considéré comme une seule instruction simple. */
    group { for ( i=0 ; i<j ; i++) printf("%d\n",j); }
}
```

En ce qui concerne les instructions plus complexes, les règles suivantes sont ajoutées à la grammaire de [K&R] avec les mots clés et les terminaux en gras ("**id**" représente un identificateur de bloc ou de rendez-vous) :

statement : conc_statement

statement : sync_statement

statement : block_statement

statement : **break (id) ;**

conc_statement : **execute** statement **and** statement

sync_statement : **when (id)** statement

sync_statement : **when (id)** statement **else** statement

sync_statement : **join (id)**

block_statement : **block (id)** statement

C'est l'instruction "**execute** statement **and** statement" qui donne la possibilité de spécifier que deux instructions sont à exécuter en parallèle. Précisons que la variable "statement" peut se dériver en un bloc d'instructions composées ("compound_statement").

Les instructions "join()" et "when()" permettent les communications entre tâches via des rendez-vous qui doivent avoir été déclarés au préalable. Si on considère cette déclaration de rendez-vous : "meeting rdv:3;", il faut au moins trois tâches bloquées sur une instruction "join(rdv)" pour que le rendez-vous se réalise. Les tâches reprennent l'exécution en séquence quand le rendez-vous est réalisé. Il est possible que cela ne se produise jamais et on se trouve alors dans une situation de blocage.

L'instruction "when()" permet de tester la faisabilité d'un rendez-vous à la manière du "if() ... else ..." du C. Cette instruction fonctionne comme "join()" mais si le rendez-vous se réalise, la partie "then" est exécutée, autrement c'est la partie "else". Il n'est donc pas possible qu'une tâche soit bloquée avec cette instruction.

L'instruction "break()" combinée avec "block()" permet d'exprimer qu'une tâche en interrompt une autre. "block()" permet de donner un nom à un bloc d'instructions et "break()" d'interrompre l'exécution d'un ou de plusieurs blocs. Les blocs peuvent se situer dans des tâches différentes et il est possible d'utiliser plusieurs fois le même identificateur. Une instruction "break(ndb)" fait que toutes les tâches se trouvant dans un bloc de nom "ndb" reprennent l'exécution en séquence après la fin du bloc. L'instruction "break" simple a le sens courant du langage C sauf dans un cas que nous verrons plus loin en 3.2.5.

3.1.1.4. Exemples illustrant la syntaxe.

Nous allons maintenant examiner deux exemples de programmes très simples qui illustrent la grammaire de llC. Considérons dans un premier temps le fichier "prog0.llc" :

```
main(argc,argv)
int
    argc;

char
    *argv[];
{
int
    i,
    j;

    if(argc!=3) exit(1);

    if(sscanf(argv[1],"%d",&i)!=1) exit(1);
    if(sscanf(argv[2],"%d",&j)!=1) exit(1);

    execute {
        while(i>0) printf("i=%d\n",i--);
    }
    and {
        while(j>0) printf("j=%d\n",j--);
    }

    exit(0);
}
```

Ici, deux tâches complètement indépendantes exécutent chacune une boucle bornée.

Le fichier suivant, "prog1.llc", est un exemple de deux tâches qui communiquent via des rendez-vous et de la mémoire partagée :

```
#include "prog1.h"

Produire(d)
int
    *d;
{
    *d=(int)random()&0xff; /* Tirage au sort d'un nombre entre 0 et 255. */
    printf("prod(%d)\n",*d);
}

Consommer(v)
int
    v;
{
    printf("cons(%d)\n",v);
}

main()
{
int
    donnée; /* Variable partagée. */

meeting
    production:2,
    consommation:2;

    srandom(getpid()); /* Initialisation du générateur de nombres aléatoires. */

    block(programme)

        execute {
            for(;;) {
                join(production);
                Produire(&donnée);
                join(consommation);
            }
        }
        and {
            for(;;) {
                join(production);
                join(consommation);
                if(donnée==FIN) break(programme);
                else Consommer(donnée);
            }
        }
}
```

Ceci avec le fichier "prog1.h" suivant :

```
extern long
    random();

extern int
    srandom();

#define FIN 0
```

Ce second exemple comporte deux tâches qui communiquent par des rendez-vous : "production" et "consommation". La première tâche est un producteur, la seconde un consommateur. Le comportement du programme est le suivant : la tâche 2 attend qu'une donnée soit produite par la tâche 1 ; si cette donnée est "FIN", la tâche 2 arrête l'exécution et sinon elle la consomme et se met en attente d'une nouvelle donnée. Le rendez-vous "production" se réalise quand la tâche 2 est prête à recevoir une donnée (autorisation de production). Le rendez-vous "consommation" est effectué quand la tâche 1 a produit une nouvelle donnée (autorisation de consommation). Les deux tâches communiquent en réalisant un rendez-vous avec passage d'information à l'aide de deux rendez-vous simples de llC et d'une donnée commune.

Pour obtenir un code exécutable de l'un de ces programmes llC il faut d'abord compiler "prog0.llc" (ou "prog1.llc") pour obtenir un fichier "prog0.c" (respectivement "prog1.c") par exemple et ensuite produire un exécutable "prog0" (respectivement "prog1") avec un compilateur C pour la machine cible. Si on exécute un de ces programmes sous le contrôle d'un système d'exploitation, une seule tâche (ou processus) système réalisera toutes les tâches décrites par l'exemple de programme llC.

Nous avons maintenant résumé les principes et la syntaxe de llC et nous allons en décrire la sémantique.

3.1.2. Sémantique de llC.

3.1.2.1. Sémantique des instructions simples.

Les instructions simples à base d'expressions d'une tâche sont indivisibles (ou atomiques), c'est à dire qu'une expression s'exécute toujours entièrement sans être interrompue par une autre tâche llC (mais elle peut être

interrompue par une interruption matérielle). Par contre, une séquence d'instructions simples dans une tâche peut tout à fait être découpée : il est donc possible que des instructions simples d'autres tâches se trouvent intercalées entre les instructions de la séquence lors de l'exécution.

Cette remarque entraîne que l'opérateur "," du C a en IIC un rôle particulier. En effet, il permet de grouper des expressions et donc de construire une instruction simple unique et atomique à partir de plusieurs d'entre elles.

Un appel de fonction peut être utilisé pour former une expression en accord avec la grammaire du langage C. Il en est de même dans IIC et par conséquent, un appel de fonction est atomique car il constitue une instruction simple. Cela signifie que lorsqu'une tâche fait appel à une fonction, les tâches du programme IIC ne progressent plus tant que la fonction s'exécute. Cette propriété peut être utilisée pour effectuer une action de manière atomique sans que les tâches puissent évoluer pendant ce temps.

Un autre moyen de grouper des instructions simples à base d'expressions est l'opérateur "group". En fait, l'instruction simple atomique "expr1,expr2;" est équivalente en terme d'atomicité à "group { expr1; expr2; }". De surcroît, cet opérateur permet de rendre atomique des instructions composées contenant des structures de contrôle. On peut par exemple rendre une boucle indivisible comme dans :

```
group { for ( i=0 ; i<j ; i++) printf(" %d\n",i);  }
```

Ceci peut être utile quand il est nécessaire de faire un traitement de manière atomique vis-à-vis des autres tâches sans passer par une fonction.

Les expressions que l'on trouve dans les tests ("if()", "while()", "do ...while()", "for()") sont également atomiques comme les instructions simples construites avec des expressions mais il n'est pas permis d'utiliser l'opérateur "group" dans les tests. L'atomicité des tests donne aussi un rôle particulier à l'opérateur "?:" du C. En effet, avec cet opérateur on a la possibilité de réaliser l'opérateur bien connu : "test and set". Pour ceci, on utilise "?:" avec une structure de contrôle de IIC ("if()", "while()", "do ...while()", "for()"). Par exemple, si on écrit :

```
if ( i ? i-=1, TRUE :  FALSE ) { /* corps du if. */ }
```

on peut dire que le corps du "If()" est exécuté en fonction de la valeur de "I" et de manière atomique, si le corps du "If()" est exécuté, la valeur de "I" a été décrémentée. Si une autre tâche utilise la même variable "I" avec le même protocole, la variable sert de "test and set" et peut donc servir de protection pour une ressource partagée. Le corps du "If" est alors une section critique qui s'exécute avec exclusion mutuelle. Pour relâcher la ressource partagée après l'avoir accaparée, une tâche augmente la variable "I" de un.

3.1.2.2. Sémantique d'un programme IIC.

Rappelons ici quelques définitions de [Arnold90] sur lesquelles repose la définition de notre produit synchrone ainsi que celle de la notion de tâche en IIC :

Le système de transitions étiquetées : un tel système est comparable aux automates finis dont nous avons parlé au 2.1. Son rôle est ici de modéliser le fonctionnement d'un processus (exécution d'un programme). Un processus est caractérisé par ses états, par ses changements d'états ou transitions et par l'alphabet A qui étiquette ses transitions.

Un système de transition étiqueté est un quintuplet $Q = < S, T, a, b, l >$ où S est un ensemble d'états fini, T est un ensemble de transitions également fini, a et b sont deux applications de T dans S qui à toute transition t de T associent les deux états $a(t)$ et $b(t)$ qui sont respectivement l'origine et le but de la transition t, l est une application de T dans A qui à toute transition t associe son étiquette $l(t)$.

Les étiquettes dont nous parlons en ce qui concerne IIC sont des instructions simples ou des résultats de tests simples ou encore des résultats de tests de faisabilité de rendez-vous.

Le produit libre de systèmes de transitions : plusieurs systèmes de transitions étiquetés peuvent être combinés pour obtenir un système de processus progressant indépendamment les uns des autres. Un tel système de processus est obtenu en effectuant le produit libre synchrone de plusieurs systèmes de transitions que nous appellerons des composants. Considérons n systèmes de transitions $Q_i = < S_i, T_i, a_i, b_i, l_i >$ avec les alphabets A_i pour

$i = 1, ..., n$. Le produit libre $Q_1 \times ... \times Q_n$ de ces n systèmes de transitions est le système de transitions $Q = < S, T, a, b, l >$ d'alphabet A et défini par :

$$S = S_1 \times ... \times S_n$$
$$T = T_1 \times ... \times T_n$$
$$a(t_1, ..., t_n) = < a_1(t_1), ..., a_n(t_n) >$$
$$b(t_1, ..., t_n) = < b_1(t_1), ..., b_n(t_n) >$$
$$l(t_1, ..., t_n) = < l_1(t_1), ..., l_n(t_n) >$$

L'alphabet A est un sous ensemble de $A_1 \times ... \times A_n$ contenant les n uplets définis par l . Un état global du système de processus est composé de la réunion de certains états des composants à un moment donné. Le passage d'un état global à un autre est réalisé par l'application d'une transition globale qui est la réunion de plusieurs transitions locales des composants. Une *transition globale est étiquetée par la réunion des étiquettes locales* des transitions qui la composent. Le produit est qualifié de synchrone car on considère que chaque composant effectue une et une seule transition lors d'une transition globale.

Une étiquette globale est dans IIC une liste de résultats de tests et d'instructions simples. Si un programme IIC ne contient pas de communications via des rendez-vous, il est un système de processus tel que nous venons de le définir d'après [Arnold90].

Le produit synchronisé de systèmes de transitions : si les processus communiquent entre eux, il se peut que des transitions de leur produit libre synchrone ne soient pas possibles (en raison de leurs interactions). On définit alors le système de processus communiquants comme un produit libre de systèmes de transitions étiquetés associé à un ensemble d'étiquettes globales *permises dit "contrainte de synchronisation". Une transition* à partir d'un état du système de processus communiquant ne peut se faire que si son étiquette appartient à la contrainte de synchronisation. Une étiquette globale contenue dans la contrainte de synchronisation est appelée : "vecteur de synchronisation". Un programme IIC dont les processus communiquent en utilisant des rendez-vous est un système de processus communiquants.

Une telle modélisation des programmes IIC permet de définir leur sémantique et donc la sémantique du langage. L'opération de combinaison

des systèmes de transitions étiquetés (tâches) d'un programme IIC permet de surcroît de réaliser la compilation du parallélisme comme dans les autres langages que nous avons présentés.

A chaque tâche IIC est associé un automate fini (système de transitions) en considérant chaque structure de contrôle du C comme un automate fini et en interprétant les instructions simples et les résultats des tests du programme comme des actions qui étiquettent les transitions de cet automate. En ce qui concerne les automates nous parlerons souvent d'actions plutôt que d'instructions simples ou de résultat de tests. A une tâche est associé un automate assez proche d'un organigramme représentant son fonctionnement.

L'état initial d'un automate est celui que l'on rejoint en séquence au début de la tâche. Chaque tache a un état terminal qui est un état n'ayant pas de transition à l'intérieur de l'automate qui lui est associé (l'état 9 avec "+" dans l'exemple suivant). L'état terminal d une tâche n'est pas forcément rejoint. On associe ainsi effectivement un automate fini à une tâche avec un ensemble d'états bien défini, un ensemble de transitions entre les états, un état initial ainsi qu'un état terminal.

L'instruction de synchronisation "when(rdv)" crée des transitions étiquetées par "rdv possible" (notée "rdv P" dans la suite) et "rdv impossible" (notée "rdv I" dans la suite). Lors de la compilation, si le rendez-vous est possible dans un état, la tâche effectuera la transition étiquetée par "rdv P" et l'autre sinon. Il en est de même pour l'instruction "join(rdv)" mais la transition "rdv I" fait rester la tâche dans le même état. Il existe un rendez-vous implicite entre deux tâches d'une même instruction "**execute** statement **and** statement". En réalité, une instruction "**execute** statement **and** statement" ne se termine que quand les deux tâches en parallèle se terminent, on a donc un rendez-vous à la fin de l'instruction.

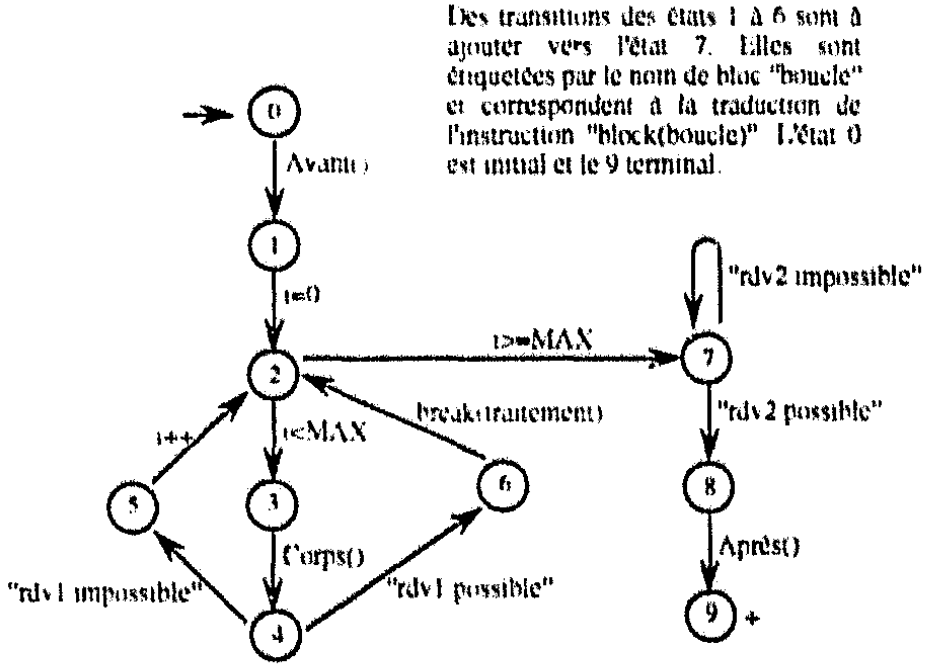
Lorsqu'un bloc est nommé avec l'instruction "block(ndb)", une transition vers l'état terminal du bloc est ajoutée à partir de tous les états de l'automate correspondant au bloc. Ces transitions supplémentaires sont étiquetées par le nom du bloc ("boucle" dans la figure suivante) ce qui signifie qu'elles sont automatiquement appliquées si une instruction "break(ndb)" est rencontrée pendant la compilation. Seules les tâches ayant de telles transitions à partir

de l'état où "break(ndb)" apparaît sont concernées lors de l'application d'une instruction "break(ndb)".

Par exemple, à la tâche IIC exprimée dans la première partie de l'instruction "**execute** statement **and** statement" suivante :

```
execute (
    Avant () ;
    block(boucle) {
        for ( i=0 ; i < MAX ;) (
            Corps () ;
            when(rdv1)
                break(traitement)
            else
                i++;
        )
    }
    join(rdv2);
    Après () ;
}
and (
    /*
        Autre tâche dont les traitements sont sans importance mais qui
        comporte certainement une ou plusieurs instructions "break(boucle)"
        et probablement au moins un "block(traitement)".
    */
}
```

on associe cet automate fini :



On procède de cette manière pour toutes les structures de contrôle et instructions du langage.

L'opération qui permet de compiler un ensemble de tâches IIC en une seule est le produit synchronisé de systèmes de transitions défini plus haut. En partant de l'état global initial qui est la réunion des états initiaux des tâches, on applique des transitions globales. Une transition globale correspond à la réunion de plusieurs transitions locales et son application donne un nouvel état global. Le principe est réitéré jusqu'à ce que tous les états globaux aient été atteints.

Les rendez-vous permettent en fait de définir la contrainte de synchronisation pour un programme particulier. Dans un état global il est toujours possible de savoir si un rendez-vous est réalisable en comptant le nombre de participants. Si le rendez-vous est faisable, les participants empruntent tous la transition locale étiquetée par "rendez-vous P" ou, inversement, par "rendez-vous I" si le rendez-vous n'a pas lieu. Il faut que le nombre de participants à un rendez-vous soit supérieur ou égal à la cardinalité de ce dernier pour qu'il se réalise. La cardinalité d'un rendez-vous est la constante qui lui est associée lors de sa déclaration.

La contrainte de synchronisation est un ensemble d'actions globales permises pour un produit d'automates. Une action globale est bien un vecteur de synchronisation dans notre produit car elle respecte la faisabilité des rendez-vous. L'ensemble des actions globales rencontrées pendant la compilation d'un système de processus IIC communicant constitue exactement la contrainte de synchronisation pour ce dernier.

3.1.3. Le code engendré par une compilation IIC.

La contrainte de synchronisation et l'ensemble des états atteignables calculés pendant la compilation d'un groupe de tâches IIC permet de produire un programme qui, lors de son exécution, réalise le comportement exprimé dans le programme source. L'enchaînement des états globaux par exécution des vecteurs de synchronisation peut se faire de deux manières différentes. Il est possible de choisir la méthode de production du code avec des options du compilateur IIC. Nous allons décrire les deux possibilités et ce qui les différencie.

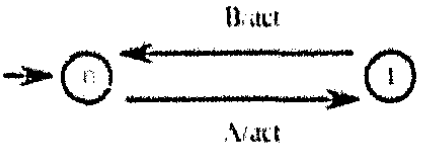
3.1.3.1. Compilation totale du parallélisme.

Cette première option du compilateur llC engendre une production de code où le parallélisme est totalement compilé. Avec cette option, la contrainte de synchronisation est complètement codée dans le programme généré. Un changement d'état global est réalisé par un simple branchement impératif. Le code produit est en général de taille importante car il contient toute l'information concernant la contrainte de synchronisation associée au produit. Le programme a une organisation que nous allons décrire dans ce qui suit.

Un automate fini est généralement codé sous forme de table de transitions dont l'exécution est réalisée par un moteur. Mais il peut aussi être codé sous forme d'un programme C qui réalise directement le fonctionnement de l'automate.

L'exemple suivant illustre ce point :

Considérons l'automate fini :



Il comporte deux états et deux transitions, l'état 0 est initial et il n'y a pas d'état terminal. L'alphabet de l'automate est {A,B}. On passe de l'état 0 à l'état 1 par une transition en exécutant l'action "act" et inversement pour l'autre transition en exécutant toujours la même action. Chaque transition est étiquetée par un couple terminal/action qui signifie que l'on emprunte cette transition si le terminal est présent dans l'entrée de l'automate et qu'on exécute alors l'action associée. Si aucune transition n'est possible à un moment donné, l'état courant reste le même et aucune action n'est exécutée.

On peut coder cet automate sous forme de table comme ceci :

état / terminal	A	B
0	action "act", goto 1	action vide, goto 0
1	action vide, goto 1	action "act", goto 0

un moteur permettra de le faire fonctionner :

```
EtatCourant = 0;
PourToujours {
    ExécuterAction ( Table [ EtatCourant , Entrée ] ) ;
    EtatCourant = EtatRejoint ( Table [ EtatCourant , Entrée ] ) ;
}
```

Le même automate serait codé de la manière suivante si on suit la logique de llC pour la première option de compilation :

```
Etat0 :
    Si Entrée != 'A' Alors Rejoindre Etat0 ;
    Sinon {
        Exécuter("act");
        Rejoindre Etat1;
    }

Etat1 :
    Si Entrée != 'B' Alors Rejoindre Etat1 ;
    Sinon {
        Exécuter("act");
        Rejoindre Etat0;
    }
```

On peut se rendre compte ici que le code de l'action "act" est recopié autant de fois qu'il y a de transitions qui la comportent. Si l'action est une suite d'instructions C, cette suite va être recopiée à chaque "Exécuter("act")" (recopie littérale de l'action). Mais si l'action est un appel de fonction C, c'est l'appel que l'on va recopier. La vision globale de l'automate (il n'est pas découpé en table d'actions) doit permettre au compilateur C des optimisations meilleures que dans le cas d'un automate tabulé.

3.1.3.2. Interprétation du parallélisme à l'exécution.

Dans cette seconde option de compilation, les vecteurs de synchronisation sont évalués pendant l'exécution du programme. Les transitions globales se font après avoir déterminé l'état global suivant à partir de l'état global courant. C'est pourquoi nous parlons de parallélisme interprété dans ce cas car la contrainte de synchronisation est calculée au coup par coup pendant l'exécution du programme. Malgré ces différences de gestion du parallélisme, le comportement d'un programme à l'exécution est toujours le même quel que soit l'option avec laquelle il a été compilé car le produit synchronisé est défini de la même manière dans tous les cas.

La taille du code engendré est beaucoup moins importante que dans le cas de l'option précédente car l'information concernant la contrainte de synchronisation n'est pas contenue dans le programme généré. Pour cette raison, l'exécution d'un programme compilé avec cette option est plus lente car une partie du temps d'exécution est consacré à l'évaluation des vecteurs de synchronisation et le reste à l'exécution des actions et tests des automates.

Dans cette option on peut considérer que l'automate résultat de la compilation d'un programme IIC est codé sous forme tabulé (tables d'actions et de tests). Le moteur est toujours la même fonction C capable de trouver le vecteur de synchronisation qui permet de passer de l'état global courant au suivant.

Pour les exemples qui suivent nous ne présentons que le résultat de la compilation pour la première option où la contrainte de synchronisation est évaluée une fois pour toutes. On trouvera en annexe A un exemple de résultat de compilation avec la seconde option accompagné de commentaires.

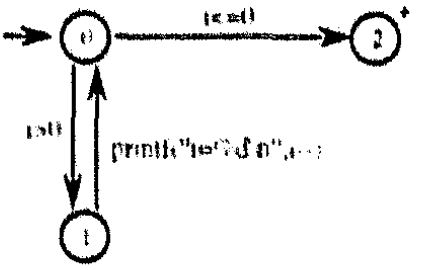
3.2. La compilation.

3.2.1. La traduction d'un programme en automate.

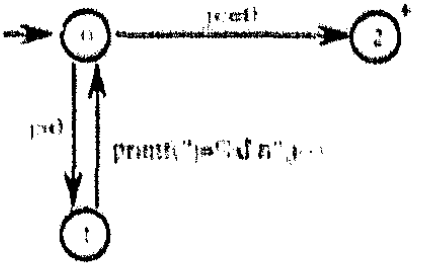
Pour illustrer les explications que nous avons données en décrivant la sémantique de IIC, nous allons maintenant voir les traductions en automates des deux exemples que nous avons présentés au 3.1.1.4. Dans ces exemples, seules les parties mises en parallèle, que nous avons appelées tâches, ont une

traduction en automates. Elles sont numérotées par ordre d'apparition dans le programme source. Tout d'abord, voyons le resultat pour "prog0.lle" :

Tâche 1 :



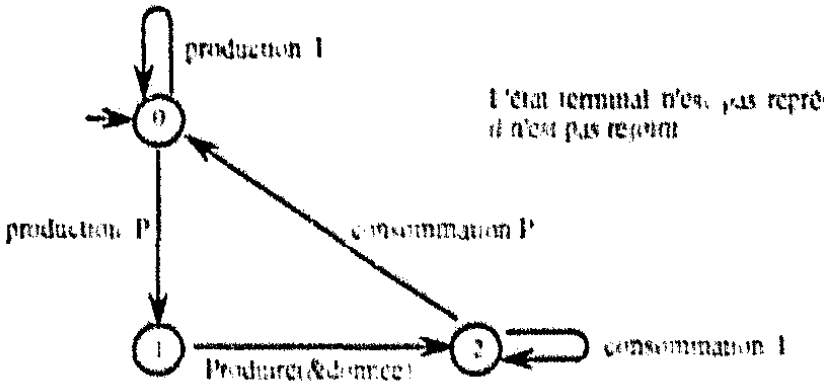
Tache 2



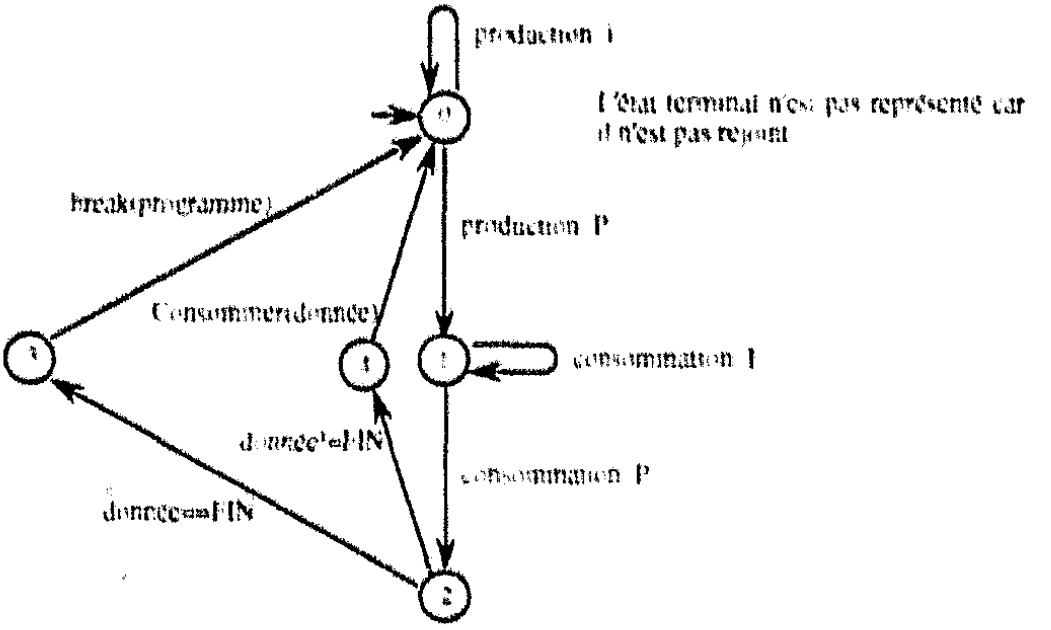
Dans ces deux automates on a numéroté les états : les états initiaux ont le numéro 0 et, les automates rejoignent tout les deux leur état terminal (noté avec un "+") Les transitions etiquetées correspondent à des résultats de tests ou à des actions effectuées pendant la transition.

Les deux automates obtenus pour les tâches de l'exemple 'prog1.lle' sont alors :

Tâche 1 :



Tâche 2



La représentation de ces deux automates est du même type que pour les deux précédents. Cependant, ici, aucun automate ne rejoint son état terminal. Nous verrons plus loin que l'état terminal du produit de ces deux automates est rejoint grâce à l'instruction "break()" dans le deuxième automate. A noter que les deux automates sont dans un bloc de nom "programme". Il existe donc une transition étiquetée par "programme" qui part de chaque état de chaque automate vers un état terminal commun. Cette

transition est appliquée au système tout entier si le produit passe par une action globale etiquetée par "breakiprogramme"

3.2.2. Des exemples de produit synchrone.

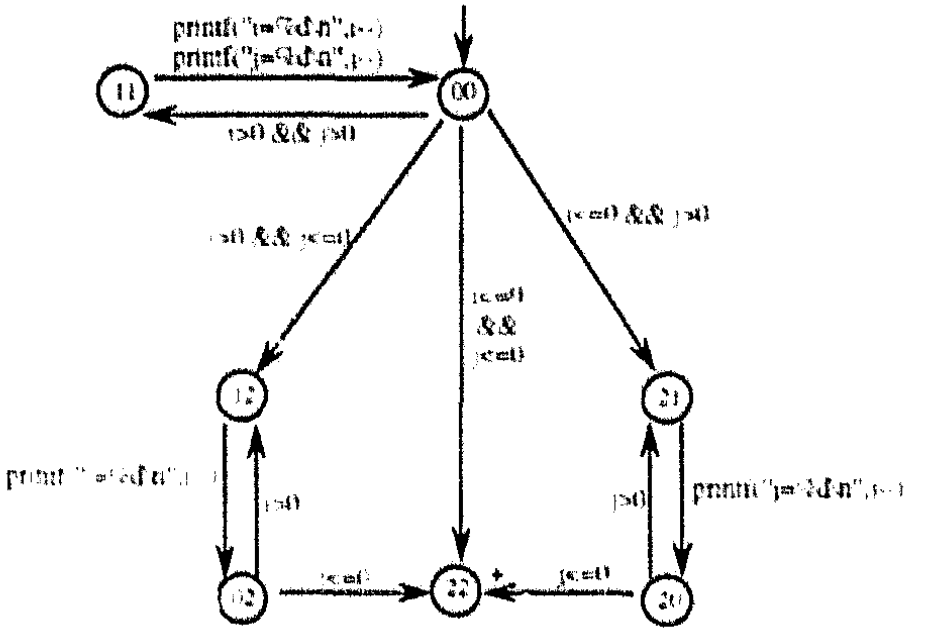
L'automate resultat du produit synchrone de deux automates au moins est réalisé comme suit

L'état initial est la fusion des états initiaux de chaque automate opérande, les autres états sont les fusions des états atteignables à partir de l'état initial. On procede de la même manière pour chaque état atteignable du produit. Un état du produit est atteignable si la fusion des transitions pour chaque état opérande l'atteint. La fusion de plusieurs états locaux aux automates dont on réalise le produit donne un etat global du produit.

Les transitions globales sont etiquetées simplement par la mise en sequence des actions locales de chaque constituant du produit. En ce qui concerne les tests locaux, ils sont imbriqués pour donner un test global dont le resultat etiquette une transition globale. L'imbrication de plusieurs tests locaux donne un arbre de tests dont les feuilles sont des transitions globales.

Le produit synchrone s'arrête quand tous les etats atteignables ont été visités. Il y a un nombre fini d'etats atteignables puisque chaque automate opérande a un nombre fini d'etats et de transitions.

Le schéma qui suit montre le résultat du produit des deux automates de "prog0.lic". Dans cet automate, chaque état est la réunion de deux états des automates opérandes. Les deux états sont rappelés dans chaque état global du produit.



Détaillons l'automate obtenu par ce produit synchrone. Comme nous l'avons dit, l'état initial s'obtient par la fusion des états initiaux de chaque tâche et il est ici numéroté "00". En partant de cet état nous allons trouver tous les états du produit. L'arbre de branchement à partir de l'état "00" est la combinaison des arbres de branchement à partir des états "0" et "0" des tâches 1 et 2. Dans la tâche 1, on a comme arbre de branchements un test simple sur la valeur de la variable "i" : "i>0" ; ce test nous conduit à l'état 1 si "i>0", sinon dans l'état 2. C'est la même chose pour la tâche 2 mais sur la variable "j". En fusionnant ces tests on obtient un arbre de tests avec quatre possibilités :

- soit "i>0 && j>0" (1)
- soit "i<=0 && j>0" (2)
- soit "i>0 && j<=0" (3)
- soit "i<=0 && j<=0" (4)

Avec la possibilité (1) on passe pour la tâche 1 de l'état "0" à l'état "1" et il en est de même pour la tâche 2, on passe donc dans le produit de l'état "00" à

l'état "11". Avec le même raisonnement, on passe par (2) de l'état "00" à l'état "21", par (3) de "00" à "12" et par (4) de "00" à "22" qui se trouve être l'état terminal du produit (fusion de tous les états terminaux). Chaque transition, étiquetée par le résultat d'un test dans les deux tâches, est un branchement synchrone, c'est à dire qu'il correspond à une transition dans chaque tâche d'origine.

En partant de l'état initial "00" nous avons donc rejoint les états : "11", "21", "12" et "22". Pour l'état "11" la seule transition possible est vers l'état global "00". *Cette transition est étiquetée par une action globale qui est la mise en séquence des actions locales :*

```
printf("i=%d",i--)
printf("j=%d",j--)
```

Nous n'avons pas atteint de nouvel état à partir de "11". Il nous reste encore les états "21" et "12" à traiter. Pour "21" la seule transition globale applicable est vers l'état "20" et elle est étiquetée par :

```
printf("j=%d",j--)
```

Cet etat est un peu particulier car la tâche 1 a ici rejoint son état terminal. On considère alors que seule la tâche 2 est active et la tâche 1 attend simplement que la tâche 2 atteigne son état terminal à son tour. La tâche 1 est donc bloquée, on a un rendez-vous entre les deux tâches qui se réalise quand elles ont toutes deux rejoint leur état terminal (l'état "22" en fait). Donc, l'arbre de tests de "20" ne tient compte que de l'état "0" de la tâche 2. Ce qui nous donne ici deux solutions :

- soit "j>0" (5)
- soit "j<=0" (6)

Par le cas (5) on passe de "20" à "21" puisque la tâche 1 n'évolue plus et que le test (5) nous fait passer de "0" en "1" dans la tache 2. Et, par (6), on passe de "20" à "22" (état terminal, rendez-vous entre les deux tâches et terminaison de l'exécution en parallèle). Nous n'avons toujours pas de nouvel état atteint à partir de "20". Pour "12" c'est exactement la même chose mais symétriquement (le test se faisant sur la variable "i"). Il ne reste ensuite que l'état "22" qui n'est qu'une étiquette à laquelle on se branche quand on a atteint l'état terminal dans le produit.

Le resultat du compilateur llC pour le programme source de "prog0.llc" est alors le suivant (avec l'option de compilation totale du parallélisme) :

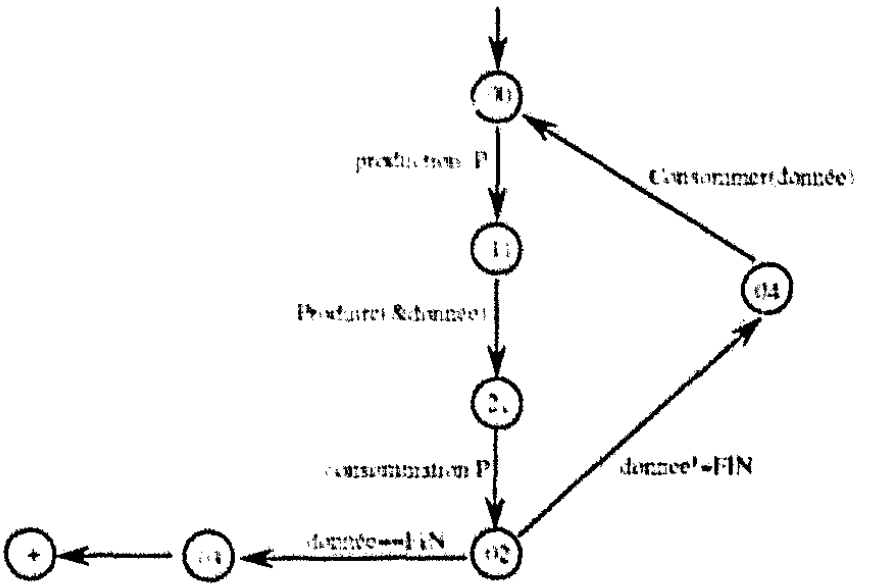
```
main(argc,argv)
int
    argc;

char
    *argv[];
{
int
    i,
    j;

 if (!(argc!=3)) goto _18_ ;
 exit(1) ;
_18_ :
 if (!(sscanf(argv[1],"%d",&i)!=1)) goto _2e_ ;
 exit(1) ;
_2e_ :
 if (!(sscanf(argv[2],"%d",&j)!=1)) goto _44_ ;
 exit(1) ;
_44_ :
_00 :
 if(!(i))
 _02 :
  if(!(j))
   goto _03 ;
  else
   /* goto _04 */ ;
 else
  if(!(j))
   goto _06 ;
  else
   goto _07 ;
/* _04 : */
printf("j=%d\n",j--) ;
 goto _02 ;
_06 :
 printf("i=%d\n",i--) ;
 if(!(i))
  goto _03 ;
 else
  goto _06 ;
_07 :
 printf("i=%d\n",i--) ;
 printf("j=%d\n",j--) ;
 goto _00 ;
_03 : ;
}
```

Le code de l'automate résultat commence à partir de l'étiquette "_00". Celle-ci correspond à l'état "00" du schéma, "_07" à "11", "_04" à "21", "_06" à "12" et enfin "_03" à "22". On peut également remarquer que l'état "20" du schéma, juste après "_04" dans le code, a été optimisé : en effet il réutilise le test dont le code a déjà été produit dans l'état "_00" ("00"). Sur une machine UNIX, nous avons compilé ce programme pour obtenir un exécutable nommé "prog0".

Voici un exemple d'exécution de ce programme :

```
% prog0 3 4
i=3
j=4
i=2
j=3
i=1
j=2
j=1
%
```

Nous allons maintenant détailler le produit synchrone pour "prog1.lic" comme nous l'avons fait pour "prog0.lic". Cet exemple comporte des communications entre des tâches. Le résultat du produit est l'automate suivant :



Pour obtenir cet automate, nous avons suivi la méthode décrite précédemment mais il a fallu traiter les synchronisations par rendez-vous et l'instruction "break()". C'est ce que nous allons détailler dans ce qui suit.

L'état initial est bien sur "00". Par ailleurs, dans cet exemple, on trouve dans l'etat "0" de la tâche 1 et de la tâche 2 une demande de rendez-vous sur "production". Ce dernier doit se faire entre au moins deux tâches d'après la déclaration : "meeting production:2;" dans le fichier source. Donc dans l'état "00" ce rendez-vous se fait immédiatement et les deux tâches empruntent chacune la transition étiquetée par "production P". Les deux tâches passent alors dans l'état "1" de manière synchrone ce qui correspond à l'état "11" dans le produit.

La tâche 2 se trouve alors sur l'instruction "join(consommation)", elle n'évolue pas et reste dans l'état "1" car ici le rendez-vous n'est pas possible (un peu comme pour le rendez-vous dans l'état terminal pour l'exemple "prog0.lic" mais explicitement). Donc de l'état "11" on passe dans l'état "21" par une transition globale étiquetée par l'action suivante :

Produire(&donnée)

Cet état nous conduit instantanément en "02" puisque le rendez-vous sur "consommation" peut alors se faire. La tâche 1 est maintenant bloquée à son tour mais sur l'instruction "join(production)". On a par contre cet arbre de branchements par la tâche 2 :

- soit "donnée==FIN" (1)

- soit "donnée!=FIN" (2)

Dans le cas (1) on passe de l'état "02" à "03" et avec (2) de "02" a "04". L'état "03" est particulier car en "3" dans la tâche 2 on a une transition locale étiquetée par l'instruction "break(programme)" qui ici peut s'appliquer puisqu'on se trouve dans le bloc "programme". En atteignant "03" on rejoint par conséquent automatiquement l'etat terminal de cet exemple (l'instruction "break()" ne fait pas forcément rejoindre un état terminal, c'est simplement ici un cas particulier car le bloc "programme" contient les deux automates). L'état "04" a ensuite une seule transition globale sortante étiquetée par :

Consommer(donnée)

ce qui conduit dans l'état initial "00" que l'on a déjà traité et nous avons alors parcouru tous les états de ce produit.

Le résultat de la compilation du fichier 'prog1.lc' avec l'option de compilation totale du parallélisme est :

```
extern long
    random();

extern int
    srandom();

Produire(d)
int
    *d;
{
 *d=(int)random()&0xff ;
 printf("prod(%d)\n",*d) ;
}


Consommer(v)
int
    v;
{
 printf("%d\n",v) ;
}

main()
{
int
    donnée;

 srandom(getpid()) ;
_00 :
 Produire(&donnée) ;
 if(donnée==0)
  goto _03 ;
 else
  /* goto _04 */ ;
 Consommer(donnée) ;
 goto _00 ;
_03 :
 ;

}
```

On peut se rendre compte que l'état "00" (étiquette "_00") est confondu avec l'état "11". L'état "03" a pour étiquette "_03" et les autres états n'ont pas besoin d'étiquette ici. Un exemple d'exécution suit, toujours sur une machine UNIX :

```
% prog1
prod(8)
cons(8)
prod(15)
cons(15)
prod(11)
cons(11)
prod(134)
cons(134)
prod(2)
cons(2)
prod(0)
%
```

Signalons qu'en IIC, l'ordre d'apparition et l'imbrication des tâches dans le fichier source déterminent l'ordre d'évaluation des tests et l'entrelacement des actions dans le résultat du produit. Si on considère deux tâches décrites comme ceci dans un fichier source :

```
execute {
    /* Tâche 1. */
}
and {
    /* Tâche 2. */
}
```

il est toujours vrai qu'un test "test1" de la tâche 1 et "test2" de la deuxième se combinent de cette manière dans le produit (si on doit les combiner) :

```
if( test1 )
    if( test2 ) goto ...
    else goto ...
else
    if( test2 ) goto ...
    else goto ...
```

De plus, les séquences d'actions suivantes

act1_tâche1 , act2_tâche1 ; act3_tâche1 ;

act1_tâche2 ; act2_tâche2 ;

sont toujours entrelacées comme ceci :

act1_tâche1 ; act1_tâche2 ; act2_tâche1 ; act2_tâche2 ; act3_tâche1 ;

On peut par exemple utiliser ce principe pour donner la priorité à une tâche sur une autre si les deux utilisent le principe de "test and set" ("?:" ou "si-alors-sinon" utilisé dans un test). Il est certain que c'est la première tâche dans l'ordre d'apparition qui prendra le "test and set" avant la seconde s'il est prenable.

L'entrelacement des actions peut, quand à lui, être modulé à l'aide de l'opérateur "," et de l'instruction "group". Ces derniers permettent de rendre atomique l'exécution d'une séquence d'actions et ainsi, de modifier l'entrelacement effectué par le produit synchrone.

3.2.3. Les optimisations du compilateur.

Après avoir décrit le produit synchrone en général, nous allons maintenant passer à des aspects plus particuliers. Nous avons vu dans l'exemple "prog0.llc" que le compilateur llC fait certaines optimisations. Ces optimisations sont toujours du même type : on tente de mettre en facteur des parties de code que l'on retrouve dans plusieurs transitions du produit. Ceci est vrai pour les arbres de branchements (comme dans l'exemple "prog0.llc") et aussi pour les actions globales. Les optimisations n'ont de sens qu'en ce qui concerne l'option de compilation totale du parallélisme. Pour l'option où il est interprété, les mises en facteurs sont automatiques pendant l'exécution.

Pour les arbres de branchements, on regarde simplement, quand on doit en évaluer un, si on ne l'a pas déjà fait dans un autre état. Si c'est le cas, un simple branchement est réalisé vers l'endroit où l'arbre existe. On applique cette méthode pour tous les sous-arbres d'un arbre de branchements. Les arbres de branchements comportent 2^n branchements (feuilles) si **n** est le nombre de tests à combiner. Mais avec cette optimisation, il est possible de trouver des arbres non symétriques dans le code produit par une compilation.

c'est à dire avec un nombre de branches inférieur à 2^n. On peut dire que cette optimisation revient à une mise en facteur de certains sous arbres (ou d'arbres entiers). En principe, un arbre de branchements n'est produit qu'une seule fois dans le résultat.

La même optimisation est faite pour les séquences d'actions. Dans une transition du produit, on vérifie pour toutes les sous-listes qu'il n'en existe pas d'équivalentes dans une autre transition déjà traitée. Si on trouve une liste équivalente, un branchement est engendré sans produire de code. Comme pour les arbres de branchements, cela revient à une mise en facteur (un partage) des actions entre les transitions.

3.2.4. La synchronisation entre automates.

Nous allons ici revoir le mécanisme de synchronisation par rendez-vous de IIC. C'est à l'aide des rendez-vous conjugués avec la mémoire partagée que les tâches peuvent communiquer de l'information. Nous avons vu dans l'exemple "prog1.iic" comment on traite les rendez-vous pendant le produit synchrone. Nous allons en parler de manière plus générale.

Il existe deux types de rendez-vous. Nous avons pu en voir un premier dans l'exemple "prog0.iic" : c'est le rendez-vous implicite à la fin d'une instruction "**execute** statement **and** statement". Dans IIC, on considère que deux tâches en parallèle se donnent rendez-vous à leur fin (à l'état terminal du produit). Ceci est valable pour deux tâches ou plus en cas d'imbrication de plusieurs instructions de mise en parallèle. Le deuxième type, les rendez-vous nommés, sont une généralisation du rendez-vous implicite.

Quand, pendant le produit de deux automates, une tâche se trouve bloquée sur un rendez-vous, elle ne progresse plus jusqu'à ce que le nombre de tâches requises pour ce rendez-vous soit atteint. Il faut en fait qu'il y ait au moins le nombre de tâches requises, s'il y en a plus on considère le rendez-vous comme faisable. Si cette condition est satisfaite, toutes les tâches progressent en passant chacune dans l'état suivant de leur automate.

Si plusieurs rendez-vous sont possibles à la suite dans un état, ils se font tous et il n'y a pas d'états intermédiaires. Il est possible, à l'inverse, que des rendez-vous ne se réalisent jamais, on a alors des blocages.

3.2.5. Interruption d'un automate par un autre.

L'autre moyen de communication entre des tâches IIC est l'instruction "break()". On peut l'utiliser avec ou sans identificateur de bloc. Comme pour les rendez-vous, on dispose d'une version simple de cet opérateur. Si, dans une tâche, on trouve un "break" sans identificateur de bloc au niveau d'une instruction "**execute** statement **and** statement", cela aura pour effet de terminer l'exécution en parallèle pour les deux tâches. L'exécution reprend alors après l'instruction parallèle (passage en séquence après "**execute** statement **and** statement").

La version avec identificateur de bloc est une évolution de la version simple. En effectuant l'instruction "break(**id**)", une tâche fait passer toutes les tâches se trouvant dans un bloc nommé "**id**", en séquence après la fin du bloc. Ceci est valable pour la tâche qui execute le "break(**id**)" si elle se trouve dans un bloc de nom "**id**". Avec ce moyen de communication, une tâche peut interrompre un traitement qu'une ou plusieurs autres tâches étaient en train d'effectuer.

On peut en utilisant ces opérateurs, implanter des rendez-vous :

```
execute {
    /* Tâche 1. */
    ...
    block (rendez_vous) for(;;);
    ...
}
and {
    /* Tâche 2. */
    ...
    break (rendez_vous);
    ...
}
```

La tâche 1, en entrant dans le bloc "rendez_vous", attend que la tâche 2 l'en fasse sortir. Quand cela ce produit, les deux tâches continuent en séquence et on peut donc considérer qu'elles ont réalisé un rendez-vous.

3.3. Exemple "producteur/consommateur".

Nous allons maintenant étudier un exemple un peu plus évolué que les précédents. Cet exemple est composé de deux tâches, l'une est un producteur et l'autre un consommateur. La structure qui leur permet de communiquer est une file. On désire gérer la communication de telle manière que le producteur se bloque s'il tente d'ajouter un élément dans la file alors que celle-ci est pleine. A l'inverse, le consommateur doit se bloquer s'il tente de prendre un élément de la file et que celle-ci est vide. La file est initialisée à vide au début du programme. L'implantation de ce principe de communication nous permet de commencer à montrer que IIC donne bien la possibilité de conserver des habitudes de programmation classique puisqu'on trouve ce moyen de communication dans beaucoup de systèmes d'exploitation.

Pour cet exemple, nous allons donner deux solutions. La première solution dite "statique" utilise, pour la gestion de la file, les outils de communication de IIC suivants : les rendez-vous et la mémoire partagée. La seconde solution, que l'on qualifiera de "dynamique" utilise le mécanisme "test and set" et la mémoire partagée. Pour les deux solutions le fichier source est le même car seul un fichier d'inclusion les différencie. On trouve dans un de ces fichiers, une série de macro-définitions qui déterminent des actions sur une file (déclaration, initialisation, enfiler, défiler, ...).

Il y a donc un fichier d'inclusion par version et un fichier source commun nommé "pc.lic" (on notera la compilation conditionnelle avec l'inclusion de "spc.h" dans un cas et de "dpc.h" dans l'autre) :

```
#define NULL        0
#define FOREVER     for(;;)
#define QSIZE       5

#ifdef STATIC
#include "spc.h" /* Communications statiques. */
#else
#include "dpc.h" /* Communications dynamiques. */
#endif

extern long random(); /* Fonction de production de nombres aléatoires. */

/*-------------------------------------------------------------------*/
prod(i) int i; { printf("prod(%d)\n",i); }
/*-------------------------------------------------------------------*/
cons(i) int i; { printf("cons(%d)\n",i); }
/*-------------------------------------------------------------------*/
main(argc,argv)
int
    argc;

char
    *argv[];
{
int max;
register int ival, oval, i;

_QDeclaration(iq,int,QSIZE); /* Declaration de la file. */

    /* Récupération des paramètres. */
    if(argc!=2) exit(1);
    if(sscanf(argv[1],"%d",&max)!=1) exit(1);

    srandom(getpid()); /* Initialisation du producteur de nombres aléatoires. */
    _QInit(iq); /* Initialisation de la file. */

    block(pc) {
        execute {
            /* Producteur. */
            FOREVER {
                oval=random()&0xff; /* Tirage au sort d'une valeur. */
                _QPost(iq,oval); /* On dépose la valeur dans la file. */
                prod(oval);
            }
        }
        and
        execute {
            /* Consommateur. */
            for(i=0;i<max;i++) {
                _QPend(iq,ival);      /* Attente d'une valeur dans la file. */
```

```
            cons(ival);
        }
        break(pc); /* Terminaison de tout le programme. */
    }
    and
        /* Tache de gestion de la file (seulement en mode "STATIC"). */
        _QUse(iq);
  }
}
/*-----------------------------------------------------------------------*/
```

3.3.1. Solution statique avec l'instruction "when()".

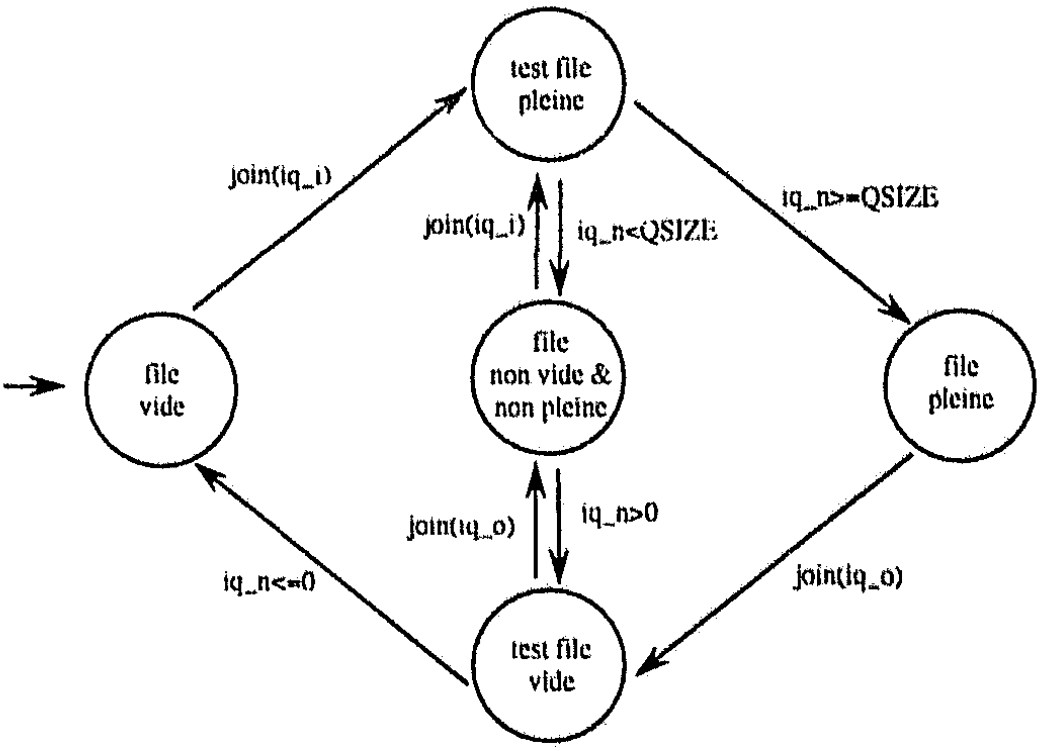
La première solution est celle dans laquelle les communications sont gérées avec les rendez-vous de llC. Voici le fichier d'inclusion "spc.h" dont nous commenterons les macro-définitions ensuite :

```
#define _QN(q)  q/**/_
#define _QF(q,f) _QN(q)/**/f

#define _QDeclaration(q,t,s)\
register t *_QF(q,i),*_QF(q,o); register int _QF(q,n);\
static t _QF(q,b)[s]; meeting _QF(q,i):2,_QF(q,o):2

#define _QSize(q) (sizeof(_QF(q,b))/sizeof(*_QF(q,b)))
#define _QInit(q) _QF(q,n)=0,_QF(q,i)=_QF(q,o)=_QF(q,b)

#define _QPost(q,v)\
join( _QF(q,i) );\
*_QF(q,i)=(v) ,\
_QF(q,i)=(_QF(q,i)<&_QF(q,b)[(_QSize(q)-1)])?(_QF(q,i)+1):(_QF(q,b))\

#define _QPend(q,r)\
join( _QF(q,o) );\
(r)=*_QF(q,o) ,\
_QF(q,o)=(_QF(q,o)<&_QF(q,b)[(_QSize(q)-1)])?(_QF(q,o)+1):(_QF(q,b))

#define _QUse(q)\
{\
uflow :    join(_QF(q,i)); _QF(q,n)++;\
t_oflow :  if(_QF(q,n)>=_QSize(q)) { join(_QF(q,o)); _QF(q,n)--; } else goto io;\
t_uflow :  if(_QF(q,n)<=0) goto uflow;\
io : \
    for(;;)\
        when(_QF(q,i)) { _QF(q,n)++; goto t_oflow; }\
        else\
            when(_QF(q,o)) { _QF(q,n)--; goto t_uflow; }\
}
```

"_QF()" permet de construire des identificateurs d'après le nom de la file pour les différentes variables qui la constituent. Dans notre exemple, la file se nomme "iq" (pour "integer queue") et pour construire le nom "iq_n" on utilise : "_QF(iq,n)". La macro-définition la plus importante pour cette solution est "_QUse()". Utilisée comme une tâche, elle permet d'assurer la gestion de la file. La file contient des entiers stockés dans une zone mémoire "iq_b[]". Les autres constituants sont : un pointeur en entrée "iq_i", un pointeur en sortie "iq_o", le nombre d'éléments dans la file "iq_n". Chaque pointeur est initialisé au début de la zone de stockage et la parcourt circulairement.

Dans le cas présent, le rendez-vous "iq_i" (même identificateur que le pointeur en entrée mais de type "meeting") permet à la tâche de gestion de donner l'autorisation d'enfiler pour le producteur. Un rendez-vous symétrique est associé à la file pour le consommateur : "iq_o" (même identificateur que le pointeur en sortie mais de type "meeting"). Il est utilisé par la tâche de gestion pour autoriser un retrait d'élément de la file.

La tâche de gestion contient le protocole d'accès à la ressource partagée qu'est la file. En fait, la seule ressource vraiment partagée est "iq_n" : le nombre d'éléments dans la file à un moment donné. La gestion peut être représentée graphiquement à l'aide de cet automate fini (ne pas confondre cet automate avec les automates des tâches IIC, on utilise une représentation du protocole sous forme d'automate pour des raisons pratiques) :



Le fonctionnement est le suivant : la tâche de gestion réalise les rendez-vous avec le producteur ou le consommateur suivant l'état de la variable "iq_n". Si la variable est nulle, elle n'accepte que les rendez-vous avec le producteur. Si "iq_n" est égale au nombre maximum d'éléments possible dans la file, seuls les rendez-vous avec le consommateur sont acceptés.

Quand le nombre d'éléments de la file est compris entre 0 et le maximum, la tâche de gestion permet les rendez-vous avec le producteur et le consommateur. Ceci est réalisé en imbriquant des instructions "when()" ce qui implante un mécanisme proche de l'instruction "select" de ADA en permettant une sélection entre plusieurs rendez-vous en même temps. Le premier rendez-vous testé est le plus prioritaire (ici la production) et le dernier est le moins prioritaire (ici la consommation).

3.3.2. Solution dynamique avec "test and set".

Ce fichier d'inclusion permet de réaliser la seconde solution :

```
#define _QN(q)   q/**/_
#define _QF(q,f) _QN(q)/**/f

#define _QDeclaration(q,t,s)\
register t *_QF(q,j),*_QF(q,o); register int _QF(q,n); static t _QF(q,b)[s]

#define _QSize(q) (sizeof(_QF(q,b))/sizeof(*_QF(q,b)))
#define _QInit(q) _QF(q,n)=0,_QF(q,j)=_QF(q,o)=_QF(q,b)

#define _QPost(q,v)\
while(\
    _QF(q,n)>=_QSize(q) ? \
    1\
    :\
    (_QF(q,n)++, *_QF(q,j)=(v), _QF(q,j)=(_QF(q,j)<&_QF(q,b)[(_QSize(q)-1)]) ?\
                                    _QF(q,j)+1 : _QF(q,b), 0)\
)

#define _QPend(q,r)\
while(\
    _QF(q,n)==0 ? \
    1\
    :\
    (_QF(q,n)--, (r)=*_QF(q,o), _QF(q,o)=(_QF(q,o)<&_QF(q,b)[(_QSize(q)-1)]) ?\
                                    _QF(q,o)+1 : _QF(q,b), 0)\
)

#define _QUse(q) { } /* Tâche vide. */
```

Les données constituant la file sont les mêmes que dans la solution précédente. Les changements se situent au niveau des macro-définitions pour enfiler et défiler. De plus, la macro-définition devant être utilisée comme tâche de gestion de la file décrit une tâche vide. L'accès et la modification de la ressource partagée "lq_n" sont faits de manière atomique en utilisant le mécanisme "test and set".

L'action de production teste si de la place est disponible dans la file avant d'enfiler une valeur. Si c'est le cas, le nombre de valeurs est augmenté de 1 de manière atomique par rapport au test et la boucle "while()" est terminée. Rien n'est changé s'il n'y a pas de place et la tâche appelante reboucle. Cela se passe de la même manière pour le consommateur mais symétriquement (test sur la file vide).

3.3.3. Comparaison des solutions.

Les deux solutions ne sont pas exactement équivalentes en ce qui concerne l'exécution. La première favorise le producteur très fortement. L'exécution commence par le remplissage de la file au maximum par le producteur et ensuite après chaque consommation, il y a immédiatement une production. Ce comportement est dû à la gestion de la file qui privilégie la production dans la sélection réalisée avec des imbrications de l'instruction "when()".

Pour la deuxième solution, les deux tâches ont la même "priorité" vis à vis de la file : rien dans le "test and set" ne favorise une tâche par rapport à l'autre ; c'est la valeur de la variable "lq_n" qui compte. L'exécution est donc une suite de productions et de consommations du début à la fin. Mais la différence principale est toute autre. Dans cette solution, la gestion de la file est résolue entièrement pendant l'exécution avec des tests dynamiques. Alors que dans l'autre solution, la gestion de la file est codée en partie pendant la compilation et en partie pendant l'exécution.

Ces deux types de gestion permettent un fonctionnement asynchrone du producteur par rapport au consommateur. C'est à dire que le producteur pourrait déposer plusieurs données dans la file avant que le consommateur n'en retire. De cette manière, le producteur pourrait par exemple réagir à une interruption correspondant à un signal de production ; quand elle surviendrait, une donnée serait déposée dans la file. Cette donnée pourrait être le résultat de la lecture d'un périphérique (convertisseur analogique/numérique, sonde de température, chronomètre, ...). Avec un tel producteur associé à une tâche de consommation faisant des traitements sur les données (filtrage numérique, histogrammes, statistiques, stockage sur disque, ...), on obtiendrait une structure typique d'application d'acquisition et de traitement de données.

3.4. Un cadre de réflexion classique.

L'utilisation des systèmes d'exploitation pour contrôler des applications est courante et la plupart des programmeurs sont habitués à concevoir des applications dans ce cadre de réflexion. Il est possible avec IIC de conserver ces habitudes de programmation et de conception des applications en utilisant quelques principes que nous allons décrire.

3.4.1. Ordonnancement et répartition du temps d'exécution.

Un premier point concerne la gestion des tâches qui constituent une application. On a habituellement la possibilité de donner des priorités aux tâches sous le contrôle d'un système d'exploitation et le programmeur utilise ceci pour moduler l'algorithme d'ordonnancement du système. Une tâche prioritaire doit en principe obtenir le processeur le plus rapidement possible si elle devient prête à s'exécuter et elle doit pouvoir le garder jusqu'à ce qu'elle le rende que de manière explicite. Nous avons vu dans la présentation de LynxOS et de VRTX que c'est souvent une caractéristique des systèmes d'exploitation multitâches non équitables.

Il est possible de reprendre ce principe avec IIC et c'est ce qui a été fait pour l'application d'évaluation (voir la partie 4). Nous allons ici nous placer dans un cadre plus général.

Considérons une application constituée de deux tâches indépendantes T1 et T2, T1 devant être plus prioritaire que T2. Si on utilise un système d'exploitation, on exprimerait ceci en attribuant des priorités différentes aux deux tâches lors de leur création et ensuite le système s'efforcerait de respecter ces priorités pendant l'exécution de l'application.

Pour une application équivalente programmée avec IIC, il n'est pas possible de spécifier des priorités de la même manière. Avant de décrire comment obtenir le même effet, nous allons exprimer ce que l'on comprend intuitivement lorsque l'on parle de "priorités" dans un système d'exploitation non équitable pour la répartition du temps d'exécution entre différentes tâches. Reprenons notre exemple avec deux tâches T1 et T2, T1 est plus prioritaire que T2 signifie que T2 ne s'exécute que lorsque T1 est bloquée en attente d'un événement quelconque. Dès que T1 peut s'exécuter, car

l'événement attendu s'est produit, le système doit interrompre T2 et faire en sorte que T1 reprenne son exécution le plus rapidement possible.

C'est en se basant sur ces significations intuitives qu'il est possible d'entrevoir une solution permettant de faire la même chose avec IIC. On associe pour cela un rendez-vous binaire de contrôle à la tâche T2. Le programme source codant cette dernière va être découpé en tranches de temps en utilisant le rendez-vous de contrôle. Chaque demande de rendez-vous de contrôle correspond à une demande d'autorisation de progression pour T2 et c'est T1 qui accorde cette autorisation quand elle est en attente d'un événement quelconque (interruption, fin de transfert de mémoire à mémoire, fin d'une action exécutée par un périphérique, ...). Par exemple, on peut écrire :

```
main()
{
meeting
    T2ctrl:2;

    execute { /* T1 */
        for(;;) {
            while( /* Condition bloquante. */ ) {
                when(T2ctrl); /* Autorisation de progression. */
            }
            /* Traitement prioritaire. */
            ...
        }
    }
    and { /* T2 */
        for(;;) {
            action_1;
            join(T2ctrl);
            action_2;
            join(T2ctrl);
            ...
            action_n;
            join(T2ctrl);
        }
    }
}
```

Le découpage de T2 en tranches de temps est réglable simplement en augmentant ou en diminuant le nombre des demandes de rendez-vous de contrôle. Si T1 doit être très prioritaire par rapport à T2 on doit découper T2 de manière importante (beaucoup de demandes de rendez-vous de contrôle) et T1 n'accordera l'autorisation d'exécution que rarement (ce qui correspond à la

diminution du quantum dans le cas d'un système d'exploitation). On procédera inversement si T2 doit être seulement un peut moins prioritaire que T1

L'ajustement peut être très précis dans le cas de IIC car les temps de commutation entre tâches sont très courts et le découpage en tranches de temps d'une tâche peu prioritaire peut être très fin (il est possible d'intercaler des demandes de rendez-vous entre chaque instruction simple).

3.4.2. Les outils classiques de communication.

De même que pour la répartition du temps d'exécution entre les tâches d'une application, le programmeur est généralement habitué à utiliser certains outils et objets de communication que l'on trouve dans les systèmes d'exploitation. Comme tous les systèmes multitâches, VRTX et LynxOS proposent de tels outils et ils en ont même en commun. Dans le cas de IIC, on dispose de la mémoire partagée, des rendez-vous et de la possibilité offerte à une tâche d'interrompre un traitement en cours dans une autre tâche.

Nous voulons ici mettre l'accent sur le fait qu'il est possible de réaliser les outils classiques de communication entre des tâches à l'aide de ce dont on dispose dans IIC.

L'utilisation du préprocesseur C permet de définir des macro-instructions qui implantent un schéma de communication particulier suivant les besoins d'une application. Nous avons déjà montré ceci dans l'exemple "producteur/consommateur" en réalisant une communication par file entre deux tâches (avec une gestion "FIFO"). L'application d'évaluation constituera un autre exemple puisqu'on y a implanté et utilisé les objets de synchronisation classiques que sont les sémaphores n-aires. L'utilisation du préprocesseur C n'est pas obligatoire mais elle permet de réaliser une implantation claire et structurée des outils dont on désire disposer.

Pour réaliser des sémaphores binaires on procéderait de la même manière que dans les exemples cités. Considérons une application constituée de deux tâches T1 et T2 devant communiquer par l'intermédiaire d'un sémaphore binaire. Nous allons réaliser essentiellement six macro-instructions pour implanter cet objet en IIC :

```
#define BSPostMName(bs,t) bs_##t##post ; /* Nom d'un rendez-vous. */
#define BSPendMName(bs,t) bs_##t##pendm /* Nom d'un rendez-vous. */
#define BSDecl(bs,t) meeting BSPostMName(bs,t):2,BSPendMName(bs,t):2
#define BSPost(bs,t) when(BSPostMName(bs,t)) {} else {}
#define BSPend(bs,t) join(BSPendMName(bs,t))
#define BSUse(bs,t) when(BSPostMName(bs,t)) join(BSPendMName(bs,t)) else {}
```

Pour déclarer le sémaphore pour chaque tâche on utilisera BSDecl(), pour prendre le sémaphore on utilisera BSPend() (équivalent de l'opération P de Dijkstra) et BSPost() (opération V de Dijkstra) pour le libérer. La macro instruction BSUse() réalise pour une seule tâche le fonctionnement du sémaphore binaire initialement libre et elle doit être utilisée dans une tâche IIC comme dans l'exemple suivant :

```
main()
{
    BSDecl(sem,T1);
    BSDecl(sem,T2);

    execute { /* T1 */
        ...
        BSPend(sem,T1); /* Prise du sémaphore. */
        /* Section critique propre à T1. */
        BSPost(sem,T1); /* Libération du sémaphore. */
        ...
        ...
    }
    and
    execute { /* T2 */
        ...
        ...
        BSPend(sem,T2); /* Prise du sémaphore. */
        /* Section critique propre à T2. */
        BSPost(sem,T2); /* Libération du sémaphore. */
        ...
    }
    and {
        for(;;) {
            BSUse(sem,T1); /* Gestion du sémaphore binaire "sem" pour T1. */
            BSUse(sem,T2); /* Gestion du sémaphore binaire "sem" pour T2. */
        }
    }
}
```

3.5. Quelques remarques pratiques pour la programmation en IIC.

Les remarques qui suivent s'adressent au programmeur en IIC. Elles sont tirées de l'expérience de programmation acquise dans le cadre de ce travail.

3.5.1. Utilisation de l'instruction "group".

Nous avons vu que l'instruction "group" permet de rendre atomique des structures de contrôles. Par exemple, un test peut être groupé de cette manière :

group { if(/* test */) { /* action en rapport avec le résultat du test */ } }

Il peut être avantageux d'utiliser cette possibilité dans une application pour réduire la taille du résultat du produit synchrone. Ainsi, dans l'application d'evaluation, on groupe les tests qui gèrent des erreurs. En effet, la détection d'une erreur terminant l'exécution, il n'est pas nécessaire que le test associé soit compilé en parallèle. Ceci permet de gérer correctement les erreurs sans rien retirer à l'efficacité d'exécution tout en minimisant la taille du code produit.

3.5.2. Utilisation des options de compilation.

Le compilateur IIC possède deux options de compilation en ce qui concerne la gestion du parallélisme : parallélisme compilé (PC) ou parallélisme interprété (PI). Nous avons exposé les différences entre les deux options dans la partie 3.1.3. et nous allons ici indiquer comment les exploiter.

Considérons une application qui comporte plusieurs fonctions programmées en IIC. Chaque fonction contient la description de plusieurs tâches à exécuter en parallèle. Il est préférable de réserver l'option PC pour les fonctions contenant des tâches devant s'exécuter avec une efficacité maximum. Le programmeur doit décider quels sont les traitements dont l'efficacité est importante et utiliser la bonne option de compilation (PC ou PI) selon le cas. Ce principe vise à minimiser la taille des codes exécutables ainsi que les temps de compilation qui sont plus important avec l'option PC.

Dans le même ordre d'idées, l'option PI peut être utilisée systématiquement pendant la phase de mise au point d'une application, toujours pour minimiser

les temps de compilation. On dispose ainsi durant la phase de mise au point, d'une compilation rapide mais d'une exécution lente. Une fois que l'application fonctionne, on peut employer l'option PC et obtenir un programme exécutable beaucoup plus efficace mais dont la compilation a duré plus longtemps qu'avec l'option PI.

Le parallélisme interprété peut également suffire pour certaines applications qui ne nécessitent pas des temps de réponse très courts. Autrement dit, une application compilée avec cette option peut avoir des performances acceptables pour le problème posé. Dans ce cas, la limitation due à l'explosion combinatoire lié à l'option PC disparaît et il est par conséquent possible de programmer des applications aussi complexes et volumineuses que l'on veut en utilisant llC.

3.5.3. Simulation.

Le compilateur llC existe sur plusieurs machines UNIX (notamment SUN, HP, DEC Station ...) qui possèdent une puissance de calcul importante. La mise au point d'une application peut se faire en développant deux versions, une sur une machine puissante et une seconde sur la machine cible sur laquelle l'application doit finalement s'exécuter. La première version est une simulation et elle est utile pour tester des principes et des mécanismes dans un environnement où les temps de compilation sont réduits. La seconde version est la version finale qui réalise complètement l'application escomptée mais sa compilation peut prendre beaucoup de temps.

Nous avons appliqué ce principe pour développer des applications pour la carte à base de processeur AMD29000 que nous avons utilisée pour l'évaluation dans la partie 4. La machine centrale était une station SUN sur laquelle nous avons développé des simulations d'applications avant de les compiler pour une exécution sur la carte elle même.

4. Évaluation par comparaison de certaines solutions de développement.

Cette partie consiste en la comparaison de trois des approches de développement présentées plus haut. Nous utiliserons toujours le langage C suivant la norme ANSI. Nous allons comparer les deux systèmes d'exploitation VRTX-32 et LynxOS avec le langage HC. Nous utilisons HC comme le representant des langages à parallelisme compilé que nous avons vus. HC met en œuvre certains de leurs principes avec une syntaxe très proche du langage C dans tous les programmes que nous allons décrire. Notre but est de comparer de manière pratique deux choses : les systèmes d'exploitation et les LPC. Cette évaluation a pour objet de valider les hypothèses que nous avons faites dans la partie sur la comparaison "théorique" entre les systèmes d'exploitation et les LPC.

Nous parlerons de l'application d'évaluation ou de comparaison car il n'y en a qu'une mais nous allons en construire plusieurs versions en utilisant les trois methodes que nous avons retenues. La comparaison va se faire à deux niveaux, sur certains temps d'exécution ou de réponse à des stimuli et sur la taille mémoire nécessaire à chaque version de l'application.

4.1. Présentation de l'application d'évaluation.

Dans un premier temps, nous allons décrire l'application et justifier son caractère représentatif de celles couramment développées au laboratoire. Ce point est important car les cas d'école sont à écarter pour une évaluation significative.

4.1.1. L'application et son contexte industriel.

Pour nous rapprocher des applications habituelles du laboratoire, nous allons construire un système d'acquisition de données très proche de ceux utilisés sur les bancs de tests et sur les expériences. De surcroît, de tels systèmes de mesures sont courants dans l'industrie même s'ils ne constituent qu'une petite partie des développements.

4.1.1.1. Le système et les configurations matérielles.

Le matériel de mesure est un convertisseur analogique/numérique (ADC pour "Analog to Digital Converter"). La nature des signaux analogiques convertis importe peu. Nous voulons simplement effectuer des lectures de données et construire avec elles un histogramme tout en calculant une moyenne et un écart type sur la série de valeurs. Ce genre de programme est tout à fait classique, il permet l'analyse statistique d'un phénomène physique ce qui correspond le plus souvent à la demande des physiciens du laboratoire aussi bien pour les expériences que pour les tests préalables.

La communication entre l'ADC et les machines effectuant sa lecture se fait via un bus normalisé "VME" (32 bits d'adresse et de données, 64 dans les versions récentes). Ce bus industriel très utilisé permet à un processeur d'accéder à des périphériques de la même manière qu'à une cellule mémoire. Une adresse sur le bus VME va, par exemple, correspondre à un des registres de contrôle de l'ADC, une autre à une cellule mémoire de l'ADC qui contiendra un résultat de conversion à un moment donné. La lecture des données converties par l'ADC se fait donc comme la lecture de données en mémoire normale en respectant un protocole de communication que nous décrirons plus bas.

Nous disposons de deux cartes équipées de processeur pour lire l'ADC et effectuer les traitements sur les données. La première est la carte construite autour du processeur AMD29000 dont nous avons parlé dans la partie d'introduction. Nous ferons deux versions de l'application sur cette carte, une sous le contrôle du système VRTX-32 et une autre codée en IIC directement. Nous procéderons de la même manière avec la seconde carte à base de processeur MC68040 qui est, elle, sous le contrôle de LynxOS.

Il sera aussi intéressant de confronter les deux versions en IIC car l'une sera implantée sur un processeur RISC ("Restricted Instruction Set Computer", AMD29000) et l'autre sur un CISC ("Complex Instruction Set Computer", MC68040). Nous essayerons de dégager les avantages et inconvénients de chaque technologie pour IIC et les LPC en général.

4.1.1.2. Organisation générale du logiciel.

L'application va être construite suivant un schéma classique. L'ADC convertit les signaux de ses entrées analogiques chaque fois qu'il reçoit un signal spécial que l'on appelle une porte (une demande de conversion). Pendant la conversion, il émet un signal que nous allons utiliser en l'envoyant au système d'acquisition pour le prévenir qu'une donnée va bientôt être disponible. L'ADC possède une mémoire interne lui permettant de stocker 16 données converties pour chacun de ses 8 canaux en entrée.

Le système devra compter 16 interruptions puis lire les résultats dans la mémoire de l'ADC le plus rapidement possible pour permettre à ce dernier de reprendre son travail de conversion (les conversions ne sont plus possibles quand la mémoire interne de l'ADC est pleine). Le protocole de synchronisation entre l'ADC et le processeur de traitement est simple puisqu'il est entièrement contrôlé par l'interruption et par un registre de l'ADC qui indique que la mémoire est pleine, ce qui permet de vérifier que c'est bien le cas après 16 interruptions.

Pour trouver à quelle fréquence maximum une version de l'application peut fonctionner en temps réel, on compte pendant un temps fixe le nombre de portes envoyées à l'ADC. Dans le même temps, on compte le nombre de signaux indiquant que l'ADC est en train de réaliser une conversion (ces signaux sont ceux envoyés à l'application d'acquisition). Si dans cet intervalle

de temps on en obtient le même nombre, nous considérerons que l'application a fonctionné en temps réel (aucune demande de conversion n'a été perdue). Le seul moment pendant lequel des demandes de conversion peuvent être perdues est celui durant lequel la mémoire de l'ADC est pleine. Le temps que prend le système d'acquisition pour vider la mémoire de l'ADC conditionne donc directement la fréquence à laquelle l'ensemble peut fonctionner en temps réel.

A noter que cette définition de l'expression "temps réel" est particulière à cette application. Ce n'est pas tout a fait la même que celle que nous avons donnée dans l'introduction. Elle signifie ici qu'aucun évènement n'a été perdu pendant l'acquisition et non que toutes les interruptions ont été traitées en respectant une contrainte de temps précise. Il s'agit en quelque sorte d'une *notion de temps réel en moyenne* et non au sens strict du terme. Cette définition est suffisante pour notre application car la perte d'un évènement précis n'a pas de conséquences pour l'analyse statistique. Il est préférable de favoriser le taux d'acquisition général même si cela engendre la perte de quelques évènements.

Chaque version de l'application est divisée en deux tâches, aussi bien pour les systèmes d'exploitation que pour IIC. Une première assure la prise en compte des interruptions ainsi que la lecture des données, nous l'appellerons le producteur. Une deuxième traite les données en mettant à jour un histogramme et une somme des valeurs successives pour calculer une moyenne et un écart type à la fin de l'exécution. Les résultats peuvent ensuite être recueillis et utilisés par des logiciels interactifs qui permettront de visualiser l'histogramme et faire d'autres traitements. Nous appellerons la deuxième tâche le consommateur.

La communication entre les deux tâches se fait via une zone mémoire qui contient les résultats de conversions lues par le producteur. Cette zone est gérée de la même manière dans toutes les versions : la production (dépôts d'une valeur dans la mémoire) est bloquante s'il n'y a plus de place disponible et la consommation (retrait d'une valeur) est bloquante si la mémoire est vide.

Deux choses importantes sont à mentionner ici. Le choix de cette organisation pour l'application a été fait pour tenter d'exploiter au mieux le matériel dont on dispose. Nous allons, pour montrer que ceci est important,

décrire une organisation "naïve" de la même application. Si on utilise un fonctionnement sans interruption avec une seule tâche qui attend la fin des conversions, effectue la lecture et les traitements, on introduit ce que l'on appelle des temps morts, à savoir des temps inexploités qui font que pour un même nombre de données à lire, l'application naïve va prendre beaucoup plus de temps que celle ayant l'organisation plus complexe. Dans la version naïve, le temps mort est maximum puisque les actions se déroulent séquentiellement.

On peut distinguer deux types de temps mort. Un premier se situe au niveau de la conversion puisque pendant que la tâche en attend une, elle ne fait rien d'autre. Dans ce mode de fonctionnement, la tâche va tester un registre de l'ADC pour savoir si celui-ci a terminé 16 conversions. Le temps passé à faire cette action de test est purement et simplement perdu. C'est pour récupérer une grande partie de ce temps que l'on fonctionne avec des interruptions qui indiquent la fin de chaque conversion. Quand 16 interruptions ont été comptées, l'application peut aller lire directement les données en étant assurée que celles-ci sont disponibles donc en perdant un minimum de temps. Pendant le temps de conversion, l'application peut effectuer autre chose et, par conséquent, récupérer du temps mort.

Le deuxième temps perdu est appelé "temps mort de lecture". En effet, au bout de 16 conversions, le producteur lance un cycle de lecture sur le bus VME pour récupérer les données. Or, ce cycle dure un temps bien plus important qu'un cycle de lecture dans la mémoire centrale de l'ordinateur (environ un facteur 5 à 10 pour un processeur fonctionnant à une fréquence de 25 MHz). Pendant la lecture d'une donnée, le processeur est bloqué (on parle d'accès synchrone) sans pouvoir exécuter d'autres instructions. Pour remédier à cela, il faut avoir un dispositif matériel permettant de faire des accès asynchrones sur le bus VME : le processeur lance le cycle de lecture et reprend aussitôt l'exécution des instructions suivantes pendant que le cycle se déroule, à condition qu'elles soient indépendantes du bus VME (pour ne pas gêner le cycle en cours).

C'est pour exploiter les accès asynchrones que l'on utilise deux tâches. Le producteur va démarrer le cycle de lecture et, pendant ce temps, il va permettre au consommateur de traiter des données déjà lues. On utilise une

mémoire tampon entre les deux tâches car celle-ci permet de découpler la lecture du traitement et ainsi de récupérer le temps mort.

Le découplage lecture/traitement revient à introduire un décalage entre la donnée en cours de lecture et celle en cours de traitement. On obtient ainsi un effet de "pipeline" qui fait qu'il est possible, une fois le "pipeline" amorcé avec une donnée de décalage, d'effectuer un traitement simultanément avec une lecture ou une conversion. C'est un peu de cette manière que l'on utilise les puces DMA ("Direct Memory Access") pour exploiter avec un processeur des temps de transferts de données entre la mémoire centrale et des périphériques. Le décalage maximum est fixé à une valeur qui doit assurer que le producteur ne sera quasiment jamais bloqué par le consommateur.

L'application de ce principe n'est pas toujours possible. Il n'est intéressant de le faire que si le temps de commutation du producteur au consommateur est nettement inférieur au temps que dure le cycle de lecture. Le temps de prise en compte de la fin de la lecture par le producteur peut aussi avoir son importance, comme nous le verrons plus loin. S'il n'est pas possible de récupérer du temps mort de lecture en utilisant ce principe, il permet de toute façon de récupérer du temps mort de conversion tout en autorisant une programmation plus simple et plus claire de l'application.

D'une manière générale on essaye de minimiser les temps morts dans une application d'acquisition de données. Le matériel que nous allons utiliser permet dans tous les cas de faire des accès asynchrones au bus VME et de prendre en compte des interruptions. Donc nous allons pouvoir essayer d'utiliser le même schéma et tenter d'engendrer le moins possible de temps morts dans chaque version de notre application.

Nous allons maintenant voir une description du système d'évaluation ainsi que des détails sur les mesures que nous allons faire.

4.1.2. Le système d'évaluation.

4.1.2.1. Description des mesures.

Nous allons pour chaque version mesurer les mêmes temps d'exécution. Pour avoir une bonne statistique nous allons pour chacune construire un histogramme avec une moyenne et un écart type (excepté pour les mesures de

fréquences). Nous étudions en fait un système d'acquisition de données à l'aide d'un système équivalent. Nous procédons comme les physiciens qui conçoivent des expériences pour valider des théories. Ici, nous essayons de comparer les LPC et les systèmes d'exploitation donc il nous faut monter une expérience adéquate pour valider les conclusions de la comparaison théorique.

La fréquence de fonctionnement en temps réel sera évaluée pour chaque version. Cette mesure est la plus intéressante car elle dépend du fonctionnement particulier de chaque version de l'application. Le temps de lecture de l'ADC sera également évalué. Il doit théoriquement être en accord avec la fréquence de fonctionnement en temps réel. En effet, ces deux mesures sont étroitement liées car c'est uniquement pendant la lecture de sa mémoire que l'ADC ne peut pas effectuer de conversions. Donc c'est pendant la lecture des données que des portes peuvent être perdues et c'est précisément quand cela se produit que le système ne fonctionne pas en temps reel.

Nous quantifierons le temps de latence des interruptions : c'est celui qui sépare *le moment où l'interruption survient* et l'instant de l'exécution de la première instruction de la routine associée. Ce temps doit être le plus proche possible de celui du matériel ce qui ne sera pas forcément le cas pour les versions *sous le contrôle des systèmes d'exploitation, même si ceux-ci sont* préemptifs en mode privilégié.

Nous allons mesurer le temps d'activation de la tâche de lecture. Il doit normalement être supérieur au temps de latence et il peut varier suivant la charge du système d'acquisition. En effet, si la mémoire tampon entre les deux tâches n'est pas vide, le consommateur peut être actif et quand une interruption survient il faut commuter vers le producteur. Donc, les temps de commutation de contextes vont influencer le temps de réaction du producteur à l'interruption dans le cas des systèmes d'exploitation.

Tout système d'acquisition du type de celui que nous allons étudier réagit en général à un signal de lecture produit par des appareils dont le travail est de déterminer si les données sont intéressantes à lire (on peut se référer à la partie d'introduction pour plus de détail). Par exemple, la porte indiquant à l'ADC de numériser ses entrées peut être produite par un dispositif décidant à

tout moment si suffisamment de capteurs contiennent de l'information intéressante (les signaux produit par les capteurs constituent les entrées analogiques de l'ADC). Il se trouve qu'une telle porte est généralement produite à une fréquence moyenne fixe et que sa répartition dans le temps est aléatoire en raison de la nature du phénomène étudié (les capteurs ne contiennent pas toujours de l'information intéressante).

Cette situation correspond à un mode de fonctionnement réaliste des systèmes d'acquisition qui nous intéressent. Il nous est possible de produire une porte avec une fréquence moyenne fixe mais avec une répartition aléatoire dans le temps à l'aide d'un dispositif matériel qu'il est inutile de décrire en détail. Nous mesurerons le pourcentage de pertes de données en fonction de la fréquence moyenne d'une porte ayant ces caractéristiques pour étudier comment se comporte chaque version de l'application si on augmente la fréquence dans une situation réaliste.

Nous étudierons l'effet que peut avoir la variation de la taille du bloc transféré à chaque accès asynchrone sur le bus VME (variation du découpage de la mémoire de l'ADC en blocs). Il peut être intéressant de tracer et de comparer des courbes de fréquence de fonctionnement en temps réel en fonction du découpage du bloc transféré. Ceci n'est réalisable que pour les deux versions s'exécutant sur la carte à base de MC68040 car elle seule dispose d'un périphérique de transfert permettant de faire varier la taille du bloc transféré durant un cycle VME. Sur l'autre matériel que nous utilisons, il n'est possible de transférer qu'un mot mémoire par cycle du bus VME.

Des signaux indiquant le début et la fin de la mesure d'un temps seront émis dans chaque version de l'application. Cette émission est réalisée dans chaque application par un accès à un matériel spécifique via le bus VME. Le fait que ce moyen soit utilisé introduit une perturbation au niveau des mesures de temps puisqu'un accès sur le bus prend lui même un temps non négligeable.

Mais, étant donné que l'on se contente de comparer des temps mesurés pour chaque version sur un même matériel, ces surplus de temps ajoutés aux mesures ne biaisent pas les comparaisons puisqu'ils existent dans tous les cas. Ceci serait gênant si nous voulions mesurer des temps en absolu de manière précise et non en comparer.

En ce qui concerne les mesures de fréquences, ce problème n'existe pas car ce n'est pas le logiciel mais, soit le matériel de conversion soit un générateur d'impulsions qui émettent de manière autonome des signaux dont on évalue la fréquence.

4.1.2.2. Le matériel et le logiciel d'évaluation (LabVIEW®).

Pour effectuer les mesures que nous avons énumérées plus haut, nous allons utiliser un appareil piloté par un programme. L'appareil est un chronomètre programmable qui permet de mesurer le temps qui s'écoule entre deux impulsions. Le logiciel s'exécute sur un ordinateur Macintosh® sur lequel on dispose d'un langage de programmation : LabVIEW. Ce langage original a une syntaxe graphique et suit l'approche "flot de données" comme Lustre. Il permet de développer facilement des programmes pour réaliser des bancs de tests avec divers appareils de mesures. La communication entre le Macintosh et le chronomètre se fait via le bus d'instrumentation normalisé CAMAC.

Le logiciel permet de faire des mesures de temps pendant une certaine durée et quand celles-ci sont faites, de visualiser un histogramme ainsi qu'une valeur moyenne et un écart type. Les mesures sont faites au rythme du Macintosh et donc il se peut que certaines soient perdues. En fait le logiciel de mesure prend des temps au hasard par rapport au fonctionnement de l'application qu'il évalue. La valeur moyenne donne une estimation générale du temps mesuré et l'écart type associé permet de connaître la dispersion des mesures. Dans le cas de mesure de fréquences, des impulsions sont comptées pendant un temps précis ce qui permet de calculer une fréquence en Hertz.

4.2. Les particularités des différentes solutions.

Nous allons maintenant présenter chaque version de l'application individuellement. Même si nous conservons dans chaque cas l'organisation générale décrite plus haut, il est bien évident que nous tentons d'utiliser au mieux les spécificités de chaque système d'exploitation ainsi que de IIC. Il n'est en aucune façon question de favoriser une version particulière ce qui fausserait les résultats. Par contre, nous utiliserons les mêmes fonctionnalités matérielles pour chaque version logicielle.

Nous ferons ensuite une présentation et une analyse des résultats en tentant d'expliquer les différences.

4.2.1. Avec VRTX-32 sur carte AMD29000.

Dans les explications qui suivent, nous désignerons respectivement par P et V les opérations consistant à retrancher ou ajouter une unité à un sémaphore n-aire.

L'application a été développée en langage C, à l'exception de la routine traitant les interruptions. Ceci est principalement dû au compilateur C qui ne nous permet pas de programmer une routine d'interruption, sous forme de fonction par exemple. De ce fait, le code de la routine est de taille assez importante et délicat à programmer car il est nécessaire d'y faire des appels au système VRTX-32 pour la communication avec la tâche de lecture et, par conséquent, de respecter un certain protocole.

La gestion des accès à la zone tampon de cette version est réalisée à l'aide de deux sémaphores n-aires fournis par le système d'exploitation VRTX-32. Le premier sémaphore ("In") correspond au nombre de paquets de 16 événements pouvant être écrits dans la zone tampon et le second ("Out"), au nombre de paquets de 16 événements présents dans la zone tampon à un moment donné. Le producteur réalise une opération P sur le sémaphore "In" avant de lire l'ADC. Il obtient ainsi une place libre pour stocker le paquet de 16 événements. Quand l'événement est lu, il réalise une opération V sur le sémaphore "Out" pour indiquer qu'un paquet de 16 événements supplémentaires est présent dans la zone tampon.

Le consommateur réalise une opération P sur "Out" pour obtenir un paquet de 16 événements à traiter et ensuite, il effectue l'opération V sur le sémaphore "In" pour indiquer qu un emplacement est libre pour un nouveau paquet. La somme des valeurs de "In" et "Out" est donc en permanence égale à la taille totale de la zone tampon. Cette taille est la plus importante possible en fonction de la mémoire dont on dispose sur la machine à base de processeur AMD29000

Le producteur est créé avec une priorité plus importante que le consommateur car c'est lui qui assure la fonction primordiale de lecture de l'ADC conditionnant complètement la fréquence à laquelle le système

d'acquisition fonctionne en temps réel. Le traitement d'un paquet de 16 événements par le consommateur peut donc etre interrompu et suspendu par le producteur quand la mémoire de l'ADC est pleine. La tâche de lecture ne permet pas à la tâche de traitement de s'exécuter pendant la lecture de la mémoire de l'ADC car le matériel ne rend les accès asynchrones possibles que pour un seul mot mémoire à la fois. Il est cependant possible de gérer quelques variables locales au producteur pendant le déroulement d'un accès unique. Cette version ne peut donc pas se trouver dans la situation de traiter des données pendant un transfert.

A propos des interruptions, signalons que le déblocage de la tâche de lecture est effectué un peu avant d'avoir compté les 16 interruptions indiquant que la mémoire de l'ADC est pleine, ceci pour compenser le temps assez long d'activation de la tâche de lecture par la routine d'interruption. Cette dernière active la tâche de lecture par l'intermédiaire d'un sémaphore, et le temps nécessaire au système VRTX-32 pour réveiller le producteur correspond au temps qu'il faut à l'ADC pour effectuer 4 conversions. Ce réveil avec anticipation de la tâche de lecture permet de faire en sorte que l'ADC soit libéré le plus vite possible après avoir rempli sa mémoire de données converties.

La taille de l'application complète est d'environ 25Ko (sans compter la zone tampon) et il faut aussi tenir compte de la taille du système d'exploitation qui est assez importante (200Ko).

4.2.2. Avec IIC sur carte AMD29000.

Nous avons ici choisi d'utiliser des sémaphores n-aires pour la gestion de la zone tampon et nous les avons implantés en utilisant les outils de communication de IIC. Ceci nous a permis de réaliser cette version de manière pratiquement identique à celle s'exécutant sous le contrôle de VRTX-32.

Comme précédemment, la lecture de la mémoire de l'ADC s'effectue de manière atomique par rapport aux traitements car l'interface dont on dispose sur la carte AMD29000 ne nous permet pas d'avoir suffisamment de temps pour faire autre chose que la préparation de la lecture du mot suivant pendant un cycle sur le bus VME.

Les langages IIC et C ont été employés en permanence sauf dans la routine d'interruption que le compilateur C ne nous permet pas de programmer. Cette routine de traitement de l'interruption est très courte et peu compliquée car elle ne contient pas d'interaction avec un système d'exploitation. La communication entre la routine d'interruption et la tâche de lecture se fait par l'intermédiaire d'un registre du processeur dédié uniquement à cette fonction. Le déblocage du producteur est réalisé avec moins d'anticipation que dans la version VRTX-32 car le temps de réaction de la tâche de lecture à la routine d'interruption est plus court (il est inférieur au temps nécessaire à l'ADC pour effectuer une conversion).

Les différentes priorités du producteur et du consommateur sont exprimées en utilisant également les outils de synchronisation de IIC que sont les rendez-vous (voir partie 3, ordonnancement et répartition du temps d'exécution). La taille de cette version est d'environ 38Ko sans compter la mémoire utilisée pour la zone tampon que l'on fixe toujours au maximum possible.

4.2.3. Avec LynxOS sur carte MC68040.

Avec LynxOS, la gestion des interruptions matérielles ne peut s'effectuer autrement qu'en utilisant un pilote de périphérique. Cette contrainte implique de surcroit l'utilisation du pilote d'interruption intégré au système d'exploitation. Nous verrons plus loin que ces obligations ont des répercutions sur l'efficacité de l'application.

Une activité système a été définie pour réaliser le producteur et elle constitue l'essentiel du pilote. Nous voulons ici insister sur la facilité de développement du pilote car il est possible d'utiliser le langage C pour en coder l'intégralité, y compris les routines de traitement d'interruptions. Le programme est très différent de celui de la version VRTX-32 du fait de l'utilisation d'un pilote (utilisation obligatoire de l'interface définie par LynxOS).

Nous employons ici la même organisation que précédemment pour la gestion de la zone tampon et les priorités des deux tâches sont fixées lors de leur création suivant le même ordre que dans VRTX-32. Il existe, à l'opposé, une différence importante au niveau des accès VME car nous disposons d'un

périphérique permettant d'effectuer une lecture asynchrone de l'intégralité de la mémoire de l'ADC en une seule fois.

Il nous a fallu réaliser deux versions de l'application sous le contrôle de LynxOS suivant le mode de fonctionnement du périphérique réalisant les accès sur le bus VME (nous appellerons ce dernier un DMA pour "Direct Memory Access"). Dans le premier mode, la fin du transfert assuré par le DMA est indiquée au système par l'intermédiaire d'une interruption tandis que dans le second, elle est détectée de manière active par la tâche de lecture par scrutation d'un registre d'état du DMA.

La réalisation de ces deux versions nous a permis de mettre en évidence que le mode de fonctionnement le plus adapté à notre application était celui avec attente active. Ceci parce que le temps de réaction de la tâche aux interruptions est trop important par rapport au temps de lecture et que, de plus, il n'est pas possible de commuter deux fois pendant le déroulement du transfert pour laisser du temps d'exécution à la tâche de traitement. Le temps de transfert des données de la mémoire de l'ADC vers la mémoire locale de la station Motorola est trop court pour permettre aux actions de traitement et de transfert de se dérouler simultanément.

La routine de gestion de l'interruption indiquant que la mémoire de l'ADC doit être lue, débloque l'activité système de lecture avec 3 coups d'anticipation en accord avec le temps de déblocage induit par le système d'exploitation. La communication entre les deux est assurée par un sémaphore du système LynxOS.

La taille de l'application est ici la somme des tailles du pilote plus celle du processus l'utilisant. Cette somme est d'environ 29Ko et il faut en plus tenir compte de la taille du système d'exploitation qui est au minimum de 400Ko. La zone tampon est la plus grande possible comme dans les versions présentées précédemment.

4.2.4. Avec IIC sur carte MC68040.

Cette version est constituée du même fichier source que la version AMD29000. Seules les parties étant directement en contact avec le matériel sont différentes et elles sont traitées en utilisant les possibilités qu'offre le préprocesseur C en matière de compilation conditionnelle. L'implantation des

sémaphores n-aires ainsi que la gestion des priorités sont identiques à la version AMD29000.

Le code de la tâche de traitement est plus découpé que dans l'autre version IIC (voir partie 3 pour des précisions sur le découpage d'une tâche peu prioritaire) car le temps durant lequel le transfert des données se déroule peut être utilisé par la tâche de traitement. C'est la seule des quatre versions dans laquelle l'application de ce principe est possible et la fin du transfert des données est ici détectée par scrutation comme dans la version précédente.

La routine d'interruption anticipe de 1 coup le déblocage de la tâche de lecture et les interruptions sont ici traitées sans l'intermédiaire d'un pilote bien que le démarrage de l'application se fasse sous le contrôle de LynxOS. Une fois que l'application s'exécute, le système d'exploitation ne la contrôle plus et elle ne réalise aucun appel à ce dernier. La communication entre la routine d'interruption et la tâche de lecture est réalisée en utilisant une zone mémoire dédiée et elle peut s'apparenter à celle de l'autre version IIC (nous n'avons pas dédié un registre du processeur car celui-ci n'en a que très peu). La taille totale de la version IIC 68000 de l'application est de 42Ko avec une zone tampon maximum.

4.3. Interprétation des résultats.

Nous allons ici comparer les résultats obtenus pour montrer les avantages et les inconvénients qu'apportent les outils utilisés pour programmer les quatre versions de l'application présentée plus haut. Nous désignerons les deux versions s'exécutant sur la carte à base de processeur MC68040 par IIC 68000 et LynxOS. Les deux autres seront désignés par IIC 29000 et VRTX-32.

On peut déjà remarquer que la taille des applications est beaucoup moins importante en ce qui concerne les versions IIC que les versions sous le contrôle de systèmes d'exploitation, si on tient compte de la taille occupée par ces derniers.

4.3.1. Fréquence de fonctionnement en temps réel pour une porte répartie uniformément dans le temps.

Résultats des différentes versions :

IIC 68000 : 13000 Hertz

LynxOS : 8000 Hertz

IIC 29000 : 5500 Hertz

VRTX-32 : 4450 Hertz

Sur chaque matériel, c'est la version programmée avec IIC qui permet la fréquence de fonctionnement en temps réel la plus élevée. Les mesures nous permettent de constater un gain de 60 % entre la version LynxOS et la version IIC 68000. Ce gain est de 25 % entre la version s'exécutant sous le contrôle de VRTX-32 et la version IIC 29000.

Le gain moins important pour les applications s'exécutant sur la carte AMD29000 provient de l'interface avec le bus VME. Cette interface ne nous permet pas de programmer l'application IIC 29000, ainsi que celle sous le contrôle de VRTX-32, de telle manière que des transferts se déroulent simultanément avec des traitements de données. Le résultat meilleur de IIC 29000 s'explique alors simplement par la surcharge introduite par la gestion des tâches et de la communication entre tâches dans le cas de VRTX-32.

A l'inverse, la version IIC 68000 est programmée de telle manière que les transferts de données se déroulent pendant le traitement de données lues précédemment. La version LynxOS la plus performante n'utilisant pas ce principe, la différence de performance entre ces deux versions est très importante, bien plus qu'entre les deux précédentes.

4.3.2. Temps de réaction de la routine d'interruption aux interruptions externes.

Résultats des différentes versions pour cette mesure :

IIC 68000 : 3700 nanosecondes

LynxOS : 5400 nanosecondes

IIC 29000 : 680 nanosecondes

VRTX-32 : 680 nanosecondes

Les versions VRTX-32 et IIC 29000 sont équivalentes pour cette mesure alors que l'on pouvait s'attendre à un léger mieux en faveur de IIC. En effet, VRTX-32 contient certaines parties qui s'exécutent interruptions fermées ce qui peut avoir pour conséquence de différer le traitement des interruptions survenant lors de l'exécution d'une de ces parties. IIC permet à l'inverse la prise en compte des interruptions en permanence durant l'exécution de l'application. L'équivalence des deux versions montre que IIC n'apporte pas d'efficacité supplémentaire pour la prise en compte des interruptions par rapport à VRTX-32, tout du moins pour cette application. La probabilité de se trouver dans le cas défavorable pour VRTX-32 est si faible que cela n'a pas d'influence.

Cependant, il n'en est pas de même pour les versions IIC 68000 et LynxOS. Nous pouvons considérer dans ce cas que la différence s'explique par le fait que LynxOS implique l'utilisation d'un pilote d'interruptions alors dans le cas de IIC, les interruptions sont directement dirigées vers la routine qui les traite. Il est fort probable que sans le pilote d'interruptions, on aurait obtenu le même résultat pour IIC 68000 et LynxOS.

4.3.3. Temps de réaction de la tâche de lecture au signal indiquant que la mémoire de l'ADC est pleine.

Résultat de chaque version pour cette mesure :

IIC 68000 : 3370 nanosecondes

LynxOS : 54500 nanosecondes

IIC 29000 : 1300 nanosecondes

VRTX-32 : 76900 nanosecondes

Cette mesure montre dans tous les cas que IIC permet d'obtenir des temps très courts de réaction d'une tâche à une interruption. Le résultat de la version LynxOS montre en particulier que le fait de détecter la fin de transfert des données avec scrutation d'un registre d'état de l'ADC est bien meilleur que par interruption. En effet, le temps minimum de lecture des données est de 45 microsecondes (8 canaux * 16 événements * (200 nanosecondes interface VME ADC + 150 nanosecondes du DMA)) ce qui est inférieur au temps de réaction d'une tâche aux interruptions (54 microsecondes). Donc l'action de prise en compte de la fin de lecture dure plus de temps que la lecture elle même. Ce qui signifie que si on tente de récupérer du temps mort de lecture sous LynxOS, on double le temps de lecture lui-même.

Si le temps de lecture avait été plus important que le temps de prise en compte de l'interruption, il aurait été envisageable d'adopter le mode avec interruption pour permettre l'exécution d'action simultanément (relâchement du processeur par le producteur dès le transfert démarré, attente par opération P sur un sémaphore sur lequel la routine d'interruption réaliserait l'opération V). Mais il faudrait alors que le temps de commutation du producteur vers le consommateur et inversement ne prenne pas un temps trop important par rapport au temps de lecture pour permettre l'exécution d'un nombre significatif d'instruction par la tâche de traitement. Nous n'avons pas vérifié si cela était le cas car le temps de réaction aux interruptions nous suffit pour choisir la version avec scrutation.

La différence entre les versions IIC 29000 et VRTX-32 est encore plus importante. Cela peut provenir du fait qu'un système multitâches implanté sur un processeur de type RISC est très pénalisé par le grand nombre de registres à gérer lors des commutations de contextes. Alors que, à l'inverse, dans le cas de IIC, le grand nombre de registres permet d'en dédier certains, une fois pour toutes, à des actions prioritaires comme la communication entre la routine d'interruption et la tâche de lecture.

4.3.4. Temps de lecture avec gestion de la zone tampon entre le producteur et le consommateur.

Résultat de chaque version :

IIC 68000 : 49 microsecondes

LynxOS : 80 microsecondes

IIC 29000 : 162 microsecondes

VRTX-32 : 203 microsecondes

Il n'y a pas beaucoup de différence pour cette mesure entre la version IIC 29000 et la version VRTX-32 même si la première est meilleure. Cela peut s'expliquer par le fait que l'utilisation du matériel est identique dans tous les cas (lecture atomique). La différence provient certainement de la gestion de la zone tampon avec les sémaphores qui dans le cas de IIC est compilée une fois pour toutes et optimisée par le compilateur C alors que pour VRTX-32 on utilise des appels systèmes qui prennent beaucoup plus de temps d'exécution sans pouvoir être optimisés lors de la compilation.

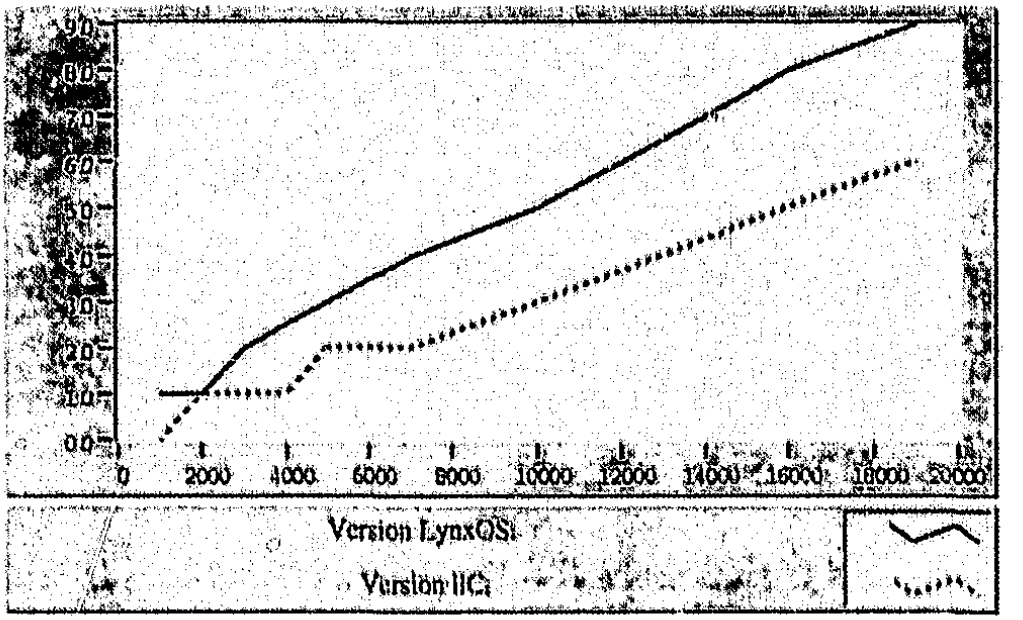
Pour les versions IIC 68000 et LynxOS, la différence s'explique de la même manière. Par contre, la fréquence de fonctionnement en temps réel est beaucoup plus importante car nous avons, dans la version IIC 68000, la possibilité de récupérer le temps mort de lecture alors que les performances sont mauvaises si on tente de faire cela dans la version LynxOS (temps de réaction trop important des tâches aux interruptions et temps de lecture certainement trop court par rapport au temps de commutation d'une tâche à l'autre).

4.3.5. Pourcentage de portes perdues en fonction de la fréquence.

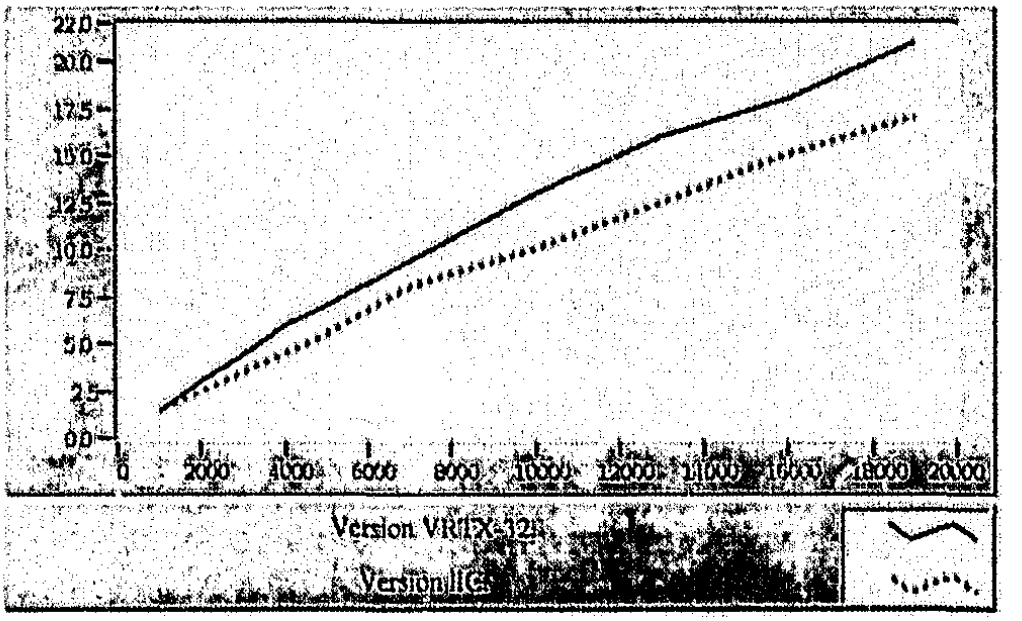
Les tables qui suivent donne les résultats de chaque version :

IIC 68000 :	0 %	à 1000 Hertz
	1 %	à 2000 Hertz
	1 %	à 3000 Hertz
	1 %	à 4000 Hertz
	2 %	à 5000 Hertz
	2 %	à 7000 Hertz
	3 %	à 10000 Hertz
	4 %	à 13000 Hertz
	5 %	à 16000 Hertz
	6 %	à 19000 Hertz
LynxOS :	1 %	à 1000 Hertz
	1 %	à 2000 Hertz
	2 %	à 3000 Hertz
	2,5 %	à 4000 Hertz
	3 %	à 5000 Hertz
	4 %	à 7000 Hertz
	5 %	à 10000 Hertz
	6,5 %	à 13000 Hertz
	8 %	à 16000 Hertz
	9 %	à 19000 Hertz
IIC 29000 :	1,5 %	à 1000 Hertz
	2,5 %	à 2000 Hertz
	3,5 %	à 3000 Hertz
	4,5 %	à 4000 Hertz
	5,5 %	à 5000 Hertz
	8 %	à 7000 Hertz
	10 %	à 10000 Hertz
	12,5 %	à 13000 Hertz
	15 %	à 16000 Hertz
	17 %	à 19000 Hertz
VRTX-32 :	1,5 %	à 1000 Hertz
	3 %	à 2000 Hertz
	4,5 %	à 3000 Hertz
	6 %	à 4000 Hertz
	7 %	à 5000 Hertz
	9,5 %	à 7000 Hertz
	13 %	à 10000 Hertz
	16 %	à 13000 Hertz
	18 %	à 16000 Hertz
	21 %	à 19000 Hertz

Le graphique suivant permet de comparer la version IIC 68000 et la version LynxOS pour une série de mesure (pourcentage de portes perdues en Y et fréquence en X) :



Ce graphique ci-après permet de réaliser la même comparaison pour les versions IIC 29000 et VRTX-32 (pourcentage de portes perdues en Y et fréquence en X) :



On peut supposer qu'à partir d'une certaine fréquence, les deux courbes sont confondues et que l'on obtient 100 % de pertes (si les interruptions

surviennent à une fréquence supérieure à la fréquence de fonctionnement du processeur). Mais nous avons effectué les mesures dans l'intervalle [1000 Hertz, 19000 Hertz] car on peut observer des différences entre les versions dans cette plage.

Pour commencer, notons que les courbes des versions IIC sont en dessous des courbes des versions sous le contrôle des systèmes d'exploitation. De plus, toutes les courbes sont linéaires pour cette série de fréquences. Le résultat important est la différence entre les pentes des droites car celles des versions VRTX-32 et LynxOS croissent plus rapidement que celles des versions IIC.

Ces constatations permettent de conclure que pour un même taux de pertes, IIC permet de faire une acquisition à une fréquence plus élevée, ce phénomène s'amplifiant quand la fréquence augmente. En d'autres termes, sur une même durée, avec une fréquence moyenne d'arrivée des portes fixée, une version IIC de l'application donne la possibilité d'acquérir plus de données qu'une des versions système et donc de faire une analyse statistique de ces données avec moins d'erreur (d'une manière générale, l'erreur des analyses statistiques diminue quand la taille de l'échantillon analysé augmente).

4.3.6. Fréquence de fonctionnement en temps réel en fonction du découpage des paquets de 16 événements lors du transfert.

Liste des résultats :

Version IIC 68000 :

13000 Hertz pour 1 bloc
12400 Hertz pour 2 blocs
11250 Hertz pour 4 blocs
9400 Hertz pour 8 blocs
7650 Hertz pour 16 blocs
5300 Hertz pour 32 blocs
3450 Hertz pour 64 blocs
1950 Hertz pour 128 blocs

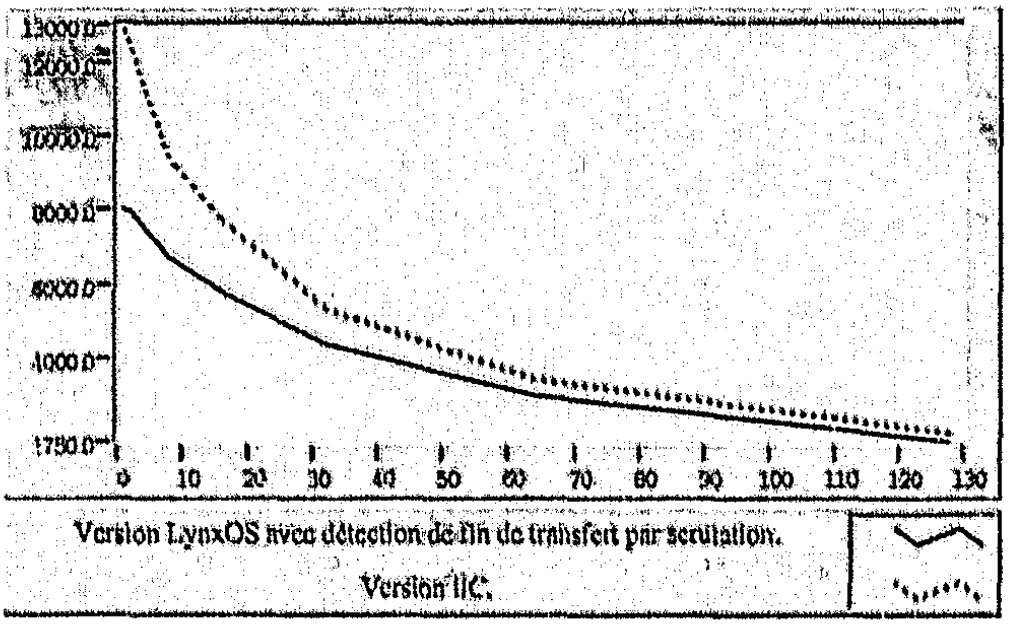
Version LynxOS avec détection de la fin de transfert d'un bloc par scrutation :

8020 Hertz pour 1 bloc
7950 Hertz pour 2 blocs
7500 Hertz pour 4 blocs
6750 Hertz pour 8 blocs
5800 Hertz pour 16 blocs
4400 Hertz pour 32 blocs
3000 Hertz pour 64 blocs
1750 Hertz pour 128 blocs

Version LynxOS avec détection de la fin de transfert d'un bloc par interruption :

5750 Hertz pour 1 bloc
3800 Hertz pour 2 blocs
2350 Hertz pour 4 blocs
1300 Hertz pour 8 blocs
670 Hertz pour 16 blocs
350 Hertz pour 32 blocs
180 Hertz pour 64 blocs
150 Hertz pour 128 blocs

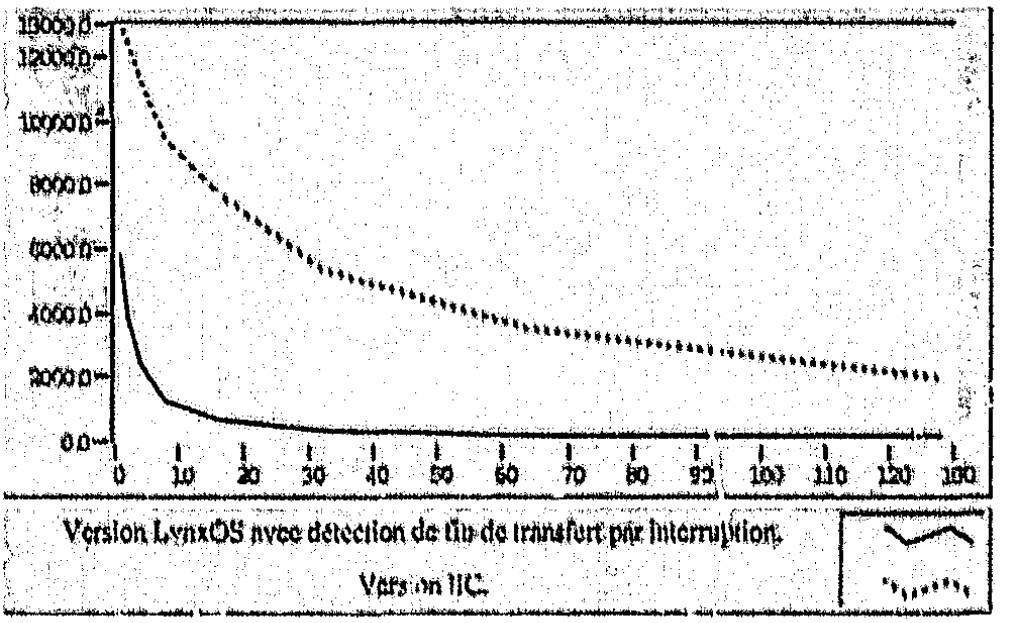
Le graphique suivant permet de comparer la version IIC 68000 et la version LynxOS avec détection de fin de transfert par scrutation (fréquence en Y et nombre de blocs en X) :



On peut remarquer que si les deux courbes ont la même forme, celle de la version IIC 68000 est constamment au-dessus de celle de la version LynxOS. La version LynxOS a un mode de lecture tel que la totalité de la lecture se déroule de manière atomique par rapport aux traitements des données. La version IIC 68000 permet l'exécution d'action en parallèle comme nous l'avons décrit précédemment.

On peut ici se rendre compte que la fréquence de fonctionnement en temps réel chute rapidement pour la version IIC quand le nombre de blocs augmente. En effet, la taille des blocs se faisant de plus en plus petite, le temps de transfert devient plus court et le temps exploitable en parallèle également. Donc plus le nombre de bloc augmente, moins il est possible d'effectuer des actions simultanées.

Pour se rendre compte de l'impact du mode de lecture avec détection de fin de transfert par interruption dans le cas de la version LynxOS, nous avons réalisé le graphique suivant avec la même version IIC 68000 que dans le graphique précédent (fréquence en Y et nombre de blocs en X) :



Les deux courbes n'ont plus ici exactement la même forme. Ce graphique montre que dans le cas de LynxOS, les performances décroissent très rapidement en fonction du nombre de bloc en raison du doublement du temps de lecture pour chaque bloc. De plus, dès la première mesure avec un seul bloc, la fréquence de fonctionnement en temps réel est inférieure à celle de la version avec scrutation dans la même situation.

4.3.7. Processeur "RISC" et processeur "CISC".

Une différence courante entre les processeurs RISC et les CISC est le nombre de registres qui est souvent beaucoup plus important dans le cas du premier type de machine. Nous avons vu que dans le cas d'un système d'exploitation, le grand nombre de registres induisait une surcharge lors des commutations de contexte. Cette surcharge est beaucoup moins importante pour les processeurs CISC.

En ce qui concerne IIC, le grand nombre de registres est utile et permet d'en dédier certains pour une fonction particulière. Mais l'essentiel réside au niveau des optimisations du compilateur C que l'on utilise avec le compilateur

IIC. Dans le cas d'un processeur RISC avec beaucoup de registres, le compilateur C va optimiser au mieux l'entrelacement des instructions simples du résultat d'une compilation IIC. L'entrelacement des instructions en provenance de tâches relativement indépendantes optimisées par un compilateur C, permet d'exploiter au mieux l'architecture d'un processeur RISC qui est capable d'exécuter plusieurs instructions dans un seul cycle si elles sont indépendantes.

Par contre, dans le cas d'un processeur CISC, si on ne dispose pas de beaucoup de registres, il faut éviter les séquences divisibles trop importantes car cela produit beaucoup d'entrelacement et donc des chargements et écrasements de registres en permanence, même si on tient aussi compte des optimisations des compilateurs C.

Les processeurs RISC ne possèdent généralement que des instructions rudimentaires que l'on doit combiner pour obtenir l'équivalent d'une seule instruction pour un processeur CISC. Ceci doit normalement entraîner une taille des programmes exécutables plus importante pour un processeur RISC que pour un processeur CISC. Mais, nous n'avons pas constaté de différences de tailles significatives pour l'application que nous avons implantée.

5. Conclusion.

5.1. Conclusions issues des comparaisons précédentes.

Les points de conclusion que nous allons lister ici sont tirés de la comparaison précédente. Ils ne s'appuient donc que sur des résultats obtenus en considérant une seule application dont nous avons développé plusieurs versions. Nous pensions que même dans ce cadre restreint, il est possible de généraliser certains des points de conclusion et de considérer que certains des résultats sont valables quel que soit le contexte. Quelques points de cette conclusion concerne les LPC en général même si nous ne citons que notre langage.

Au niveau des temps de prise en compte des interruptions par les routines d'interruption nous n'avons pas constaté de différences. L'utilisation de IIC n'apporte donc apparemment rien dans le cas de l'application que nous avons considérée. On peut tout de même rappeler que IIC permet de conserver en permanence la possibilité de prendre en compte des interruptions alors que c'est impossible dans le cas d'un système d'exploitation comme ceux que nous avons présentés. Il est probable que dans certains cas cela puisse avoir de l'importance. Rappelons que IIC permet de plus de se passer de pilotes d'interruptions comme celui que l'on trouve dans LynxOS.

Comme nous l'avions pressenti au début de ce travail, IIC permet de récupérer du temps normalement consommé par un système d'exploitation.

en particulier au niveau des commutations de contexte mais aussi de la gestion d'objets de communication entre processus et de l'activation de tâches par des routines d'interruption. Le fait de rendre possible des optimisations globales de manière importante au niveau du code C généré apporte également une efficacité accrue par rapport aux systèmes d'exploitation et ce, quel que soit le processeur utilisé. Toutes ces remarques sont à considérer en sachant qu'il est possible de conserver avec IIC un cadre de réflexion proche des habitudes courantes en matière de conception d'applications.

Ces améliorations de temps de réponse et la disparition d'une grande partie des temps de gestion du parallélisme qu'apporte IIC permettent de mieux exploiter les spécificités d'un matériel qu'avec des systèmes d'exploitation ; nous avons pu le constater en comparant les versions LynxOS et IIC 68000 à propos de la récupération du temps mort de transfert. Avec les systèmes d'exploitation il est possible d'adopter ce genre de principes mais la limite à partir de laquelle ce n'est plus intéressant se situe bien avant la limite qui s'applique à IIC.

Il est très important de noter que les versions IIC ont des codes source quasiment identiques quel que soit le matériel. Les versions "système" ont une organisation identique mais des codes sources sensiblement différents. IIC apporte donc une portabilité très importante dans le cas de cette application et certainement de manière plus générale.

Nous tenons également à signaler que le compilateur IIC disponible nous a parfaitement permis de réaliser les développements que nous voulions faire dans le cadre de la comparaison. Nous voulons mettre l'accent sur le fait que l'outil dont on dispose est parfaitement opérationnel même en tenant compte des petites restrictions que nous avons signalées lors de sa présentation.

5.2. Domaine d'utilisation de IIC.

Nous avons montré qu'il est possible de conserver avec IIC les habitudes de programmation issues de l'utilisation de systèmes d'exploitation et de langages de programmation évolués. Nous avons également mis en évidence que IIC apportait des avantages réels en matière d'efficacité d'exécution et de portabilité. Il a également été montré que IIC permet une exploitation de

certains dispositifs matériels quand cela n'est plus raisonnable avec un système d'exploitation.

Cependant, il est évident qu'un tel langage ne peut servir à résoudre tous les problèmes que l'on peut se poser. Nous allons essayer de dégager les quelques caractéristiques générales de ceux pour lesquels IIC peut donner une solution intéressante.

Notre langage doit convenir plus particulièrement pour développer des applications embarquées sur des machines avec peu de mémoire et de ressources en général. Surtout quand il est pratique d'avoir un aspect multi-tâche alors que l'implantation d'un système d'exploitation est impossible. Si les contraintes de temps de réponse et de rapidité d'exécution sont peu importantes, il est de plus possible d'utiliser l'option "parallélisme interprété". Les seules restrictions quand à la taille de l'application que l'on veut programmer sont alors celles liées au matériel.

5.3. Quelques extentions possibles du langage IIC.

Un premier travail qui reste à effectuer est de compléter le compilateur IIC afin d'accepter exactement la grammaire du langage C. La première version n'avait que pour but de prouver que le langage était viable et qu'il apportait bien les avantages que nous avions envisagés. Le traitement de l'instruction "switch" peut se faire de la même manière que celui de l'instruction "if". Les déclarations locales à un bloc peuvent être reportées en début de fonction avec un renommage en fonction du bloc d'origine.

La modélisation des programmes IIC sous forme de systèmes de processus communiquants offre la possibilité de vérifier automatiquement certaines propriétés sur les applications que l'on développe. La première d'entre elles est la preuve de l'absence de blocages pour chaque rendez-vous utilisé. Il serait donc intéressant d'exploiter ces possibilités comme cela a été fait pour les autres langages que nous avons présentés (Esterel, ...).

Comme nous l'avons dit lors de la présentation des autres LPC, il a été développé à l'INRIA des outils permettant la vérification automatique de propriétés sur les automates résultats de produits libres synchronisés. Ces outils utilisent un codage des automates appelé OC pour "Object Code" (présenté dans [Halbwachs91]). Esterel et Lustre peuvent par exemple

produire le résultat d'une compilation en OC. Tous les outils de vérification existants peuvent donc être considérés comme étant communs à Lustre et à Esterel.

Il serait certainement intéressant de pouvoir produire les automates résultats de compilation IIC en OC, afin de pouvoir utiliser les outils disponibles. Sans entrer dans les détails, nous avons le sentiment que ceci est possible, même s'il existe certainement un problème à cause d'une légère différence de sémantique entre IIC et les autres LPC mentionnés.

La gestion des priorités des tâches est à l'heure actuelle possible avec IIC. Cependant, il faut spécifier soi-même le découpage des tâches peu prioritaires et le relâchement du processeur dans les tâches prioritaires. Il serait en partie possible d'avoir une gestion automatique des priorités suivant le schéma décrit. Une tache de faible priorité pourrait automatiquement être découpée en tranches par le compilateur IIC (il faudrait alors définir la notion de "tranche" de manière précise). Par contre, le programmeur devrait toujours spécifier lui-même les relâchements du processeur dans les tâches quand cela est nécessaire.

6. Bibliographie.

[ADA83] : Reference Manual for ADA, norme ANSI, 1983.

[AMD29000] : The Am29050 Microprocessor, Advanced Micro Devices, 1991.

[Arnold90] : Systèmes de transitions finis et sémantique des processus communicants, André Arnold, T.S.I - Bordas, vol 3 n°9, 1990.

[Booch86] : Grady Booch, software engineering with ADA, The Benjamin/Cummings Publishing Company, 1986.

[CAT] : Cherenkov Array at Themis, Rapport de projet de Juillet 1993.

[GNU] : Documentation Générale des Outils "GNU", Free Software Foundation 675 Mass Ave, Cambridge MA 02139.

[H1] : Réalisation d'un système d'acquisition de données dans le cadre de l'expérience H1. These de Luigi Del Buono, Université Paris VII, 1989.

[Halbwachs91] : Conception de systèmes réactifs, Les langages synchrones, N.Halbwachs IMAG/LGI - Grenoble, 1991.

[JMR93] : La Programmation sous UNIX, Jean-Marie Rifflet, Ediscience International, 1993.

[JMR94] : La Communication sous UNIX 2e édition, Chap. 3 : POSIX et les "threads", Jean-Marie Rifflet, Ediscience International, 1994.

[K&R] : The ANSI C programming language, Brian W.Kernighan and Dennis M.Ritchie, Prentice Hall Software Series, 1988.

[LabVIEW] : LabVIEW Graphical Programming for Instrumentation, Technical Publications of National Instruments Corporation, 1994.

[LYNX] : LynxOS reference manual, Lynx Real Time Systems, 1993.

[VRTX-32] : The VRTX-32/29000 User's Guide, Ready Systems, 1990.

Table des matières

Annexe.

Annexe A : deux manières de traiter le parallélisme de IIC.

Nous allons ici présenter le résultat du compilateur IIC pour le programme "prog0.iic" dans le cas de l'option de compilation avec parallélisme interprété. Nous ne présenterons pas celui de "prog1.iic" afin de ne pas surcharger cette annexe.

Le programme généré dans le cas de cette option de compilation est structuré sous forme d'une boucle principale dont le corps peut être divisé en deux parties. La première est simplement l'appel d'une fonction trouvant le vecteur de synchronisation à appliquer à partir de l'état courant. La seconde partie consiste en l'exécution du vecteur de synchronisation déterminé précédemment. Cette exécution fait passer le système de processus communiquants dans un nouvel état global courant.

On peut alors se rendre compte que l'exécution du corps de la boucle correspond à l'application d'un vecteur de synchronisation du système de processus communiquants. La boucle se termine quand une transition globale mène dans l'état terminal du système de processus communiquants.

La fonction trouvant les vecteurs de synchronisation successifs est toujours la même quel que soit le programme IIC et nous ne la ferons pas figurer dans les programmes qui suivent. La compilation consiste dans le cas

présent en l'évaluation de tables d'actions et de tests qui permettent de réaliser l'exécution des actions globales.

Pour les exemples qui suivent, la taille du code produit n'est pas plus petite que dans l'option avec compilation totale du parallélisme. Cela s'explique par le fait que ces programmes sont très petits et qu'ils ne correspondent pas au traitement d'un problème réel. Pour des programmes plus réalistes, la différence peut être très importante. Nous avons pu constater des tailles divisées par 4 ou plus en ce qui concerne les applications développées dans le cadre de tests et plus un programme est important plus cette différence de taille va être importante.

Voici le résultat de la compilation de "prog0.llc" :

```
/*
    Déclaration de types de données pour la gestion
    des tâches et des communications.
*/
struct __TCB_struct;
typedef struct __INST_struct {

    unsigned char h;
    union {
        unsigned int  i;
        unsigned long o;
        struct { unsigned int i; unsigned long o; } r;
        struct { unsigned int i; unsigned long o; } j;
        struct { unsigned int i; unsigned long o0,o1; } w;
        struct { unsigned int i; unsigned long o0,o1; } b;
        struct { unsigned long pc1,pc2; struct __TCB_struct *tcb1,*tcb2; } f;
    } b;

} __INST__ ;

typedef struct __BSC_struct {

    unsigned short id;
    unsigned long  pc;

} __BSC__ ;

typedef struct __MEET_struct {

    unsigned int card;
    unsigned int n;

} __MEET__ ;

typedef struct __TCB_struct {
```

```
    unsigned char type;
    unsigned int nb;
    __BSC__ *bstack;
    union {
        struct { struct __TCB_struct *l,*r; } s;
        struct { unsigned long pc; __INST__ i; } l;
    } b;

} __TCB__ ;

/*
    Code de la fonction "main()".
*/
main(argc,argv)
int
    argc;

char
    *argv[];
{
int
    i,
    j;

if (!(argc!=3)) goto _18_ ;
exit(1) ;
_18_ :
if (!(sscanf(argv[1],"%d",&i)!=1)) goto _2e_ ;
exit(1) ;
_2e_ :
if (!(sscanf(argv[2],"%d",&j)!=1)) goto _44_ ;
exit(1) ;
_44_ :
{ /* Début de l'exécution parallèle. */

extern __TCB__
  _main_0_state_0,
  _main_0_state_1,
  _main_0_state_2;

extern void
  _main_0_init();

extern void
  _main_0_collect();

extern int
  __er__();

register unsigned int
  __nt__,
  __pf__;

/*
```

```
    Initialisation des structures de données. On fixe en fait l'état courant
comme
    étant l'état initial du système de processus communiquants.
 */
 _main_0_init();

 /* Boucle principale. */
 for(;;) {

  /*
    Appel de la fonction traitant les vecteurs de synchronisation
    à partir de l'état courant.
  */
  if((__nt__=__er__(0,&_main_0_state_0,_main_0_collect))==1) break;

  /*
    A partir d'ici jusqu'à la fin de la boucle principale le code est structuré
    sous forme de tables d'actions et de tests. Ces tables servent à l'exécution
    des actions globales.
  */

  __pf__ = 0 ;

  if(_main_0_state_1.type==0) switch (_main_0_state_1.b.l.pc) {
  case 0x62 :
   printf("i=%d\n",i--) ;
   _main_0_state_1.b.l.pc = 0x6b ;
   __pf__ ++ ;
   break ;
  }
  if(_main_0_state_2.type==0) switch (_main_0_state_2.b.l.pc) {
  case 0x7e :
   printf("j=%d\n",j--) ;
   _main_0_state_2.b.l.pc = 0x87 ;
   __pf__ ++ ;
   break ;
  }

  if ( __pf__ != 0 ) continue ;

  if(_main_0_state_1.type==0) switch (_main_0_state_1.b.l.pc) {
  case 0x6b :
   _main_0_state_1.b.l.pc = 0x55 ;
   __pf__ ++ ;
   break ;
  }
  if(_main_0_state_2.type==0) switch (_main_0_state_2.b.l.pc) {
  case 0x87 :
   _main_0_state_2.b.l.pc = 0x71 ;
   __pf__ ++ ;
   break ;
  }

  if ( __pf__ == __nt__ ) continue ;
```

```
  if(_main_0_state_1.type==0) switch (_main_0_state_1.b.l.pc) {
  case 0x55 :
   if (!(i)) _main_0_state_1.b.l.pc = 0x70 ;
   else _main_0_state_1.b.l.pc = 0x62 ;
   break ;
  }
  if(_main_0_state_2.type==0) switch (_main_0_state_2.b.l.pc) {
  case 0x71 :
   if (!(j)) _main_0_state_2.b.l.pc = 0x8c ;
   else _main_0_state_2.b.l.pc = 0x7e ;
   break ;
  }

 } /* Fin de la boucle principale. */

 } /* Fin de l'exécution parallèle. */

} /* Fin de main(). */

/* Déclaration de données pour la gestion des tâches. */
__TCB__
  _main_0_state_0,
  _main_0_state_1,
  _main_0_state_2;

/* Fonction d'initialisation de l'état initial. */
void _main_0_init()
{
 _main_0_state_0.type=1;
 _main_0_state_0.b.s.l=&_main_0_state_1;
 _main_0_state_0.b.s.r=&_main_0_state_2;
 _main_0_state_0.nb=0;
 _main_0_state_0.bstack=(__BSC__ *)0;
 _main_0_state_1.type=0;
 _main_0_state_1.b.l.pc = 0x55 ;
 _main_0_state_1.nb=0;
 _main_0_state_1.bstack=(__BSC__ *)0;
 _main_0_state_2.type=0;
 _main_0_state_2.b.l.pc = 0x71 ;
 _main_0_state_2.nb=0;
 _main_0_state_2.bstack=(__BSC__ *)0;
}

/* Fonction utilisée pour la gestion des vecteurs de synchronisation. */
void _main_0_collect()
{
 if(_main_0_state_1.type==0) switch (_main_0_state_1.b.l.pc) {
 case 0x70 :
  _main_0_state_1.b.l.i.h = 1 ;
  _main_0_state_1.b.l.i.b.o = 0x71 ;
  break ;
 default :
  _main_0_state_1.b.l.i.h = 0 ;
  break ;
 }
```

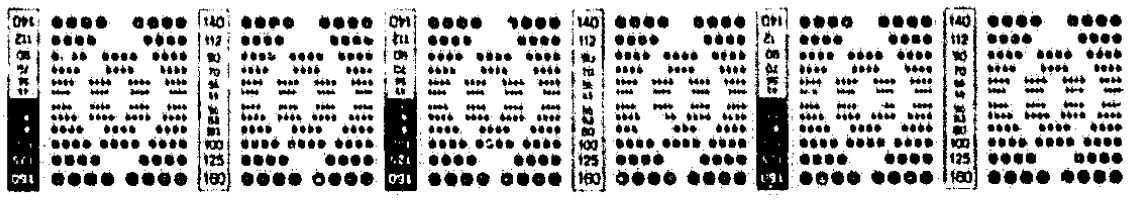
```
if(_main_0_state_2.type==0) switch (_main_0_state_2.b.l.pc) {
case 0x8c :
  _main_0_state_2.b.l.i.h = 1 ;
  _main_0_state_2.b.l.i.b.o = 0x8d ;
  break ;
default :
  _main_0_state_2.b.l.i.h = 0 ;
  break ;
}
}
```

Pour "prog1.llc" on obtient quelque chose de similaire mais durant l'exécution les rendez-vous et les interruptions seront traités puisque ce programme en comporte.

FIN

graphicom

MIRE ISO N° 1
NF Z 43-007
AFNOR
Cedex 7 - 92080 PARIS-LA-DÉFENSE



MINISTERE DE L'EDUCATION NATIONALE ET DE LA CULTURE
DIRECTION DE LA PROGRAMMATION ET DU DEVELOPPEMENT UNIVERSITAIRE
DIRECTION DE LA RECHERCHE ET DES ETUDES DOCTORALES

FORMULAIRE D'ENREGISTREMENT DE THESE SOUTENUE

A FOURNIR OBLIGATOIREMENT POUR TOUTE SOUTENANCE DE THESE EN 3 EXEMPLAIRES, QUE CETTE THESE SOIT OU NON REPRODUITE

N° D'ENREGISTREMENT (3 à 10)

N° CARTE D'ETUDIANT : 7.7.6.9.6

A REMPLIR PAR LE CANDIDAT, A L'ENCRE NOIRE, EN MAJUSCULES D'IMPRIMERIE, UN CARACTERE PAR CASE
NE RIEN INSCRIRE DANS LES CASES GRISEES

NOM DE L'ETABLISSEMENT OU DE L'UNIVERSITE : UNIVERSITE PARIS 7 DENIS DIDEROT

TB — Code de l'établissement ou de l'université (cf notice) → 0751722R (11 à 18)

TYPE DE DOCTORAT : Nouveau régime — 3e cycle — Ingénieur — Autre

CODE TITRE : M — Mme — Mlle

NATIONALITE : Française — CEE — Autre

NOM (sous lequel a été soutenue la thèse) → DELCHINI (21 à 40)

NOM de jeune fille (pour les femmes mariées, si différent de ci-dessus) → (41 à 54)

PRENOM → HUGO (55 à 66)

DIRECTEUR THESE (nom, prénom) : M / Mme / Mlle RIFFLET JEAN-MARIE (67 à 72)

DATE DE SOUTENANCE → Année 95 (73 à 76) Mois 04 (77 à 78)

TC — DISCIPLINE : 001D02

NUMERO DE LA SECTION CNU DU DIRECTEUR DE THESE (cf notice) :

11 D — 13 I — 15 S.S. — 17 S.S. — 19 S.S. — 21 S.S. — 23 C — 26 O — 31 P — 33 CSU

SPECIALITE : INFORMATIQUE

ADRESSE PERMANENTE DU CANDIDAT

TD

N° ET NOM DE RUE

COMMUNE

CODE POSTAL

DATE DE NAISSANCE

A REMPLIR PAR LE CANDIDAT - DACTYLOGRAPHIER OBLIGATOIREMENT EN UTILISANT MINUSCULES ET MAJUSCULES, ET EXCLUSIVEMENT DES CARACTERES LATINS.
Aucun symbole spécial (alphabet grec, chiffres romains, symboles mathématiques...) n'est admis.

TITRE DE LA THESE EN FRANCAIS

TE — Conception, développement et évaluation d'un langage de

TF — programmation adapté aux applications industrielles : IIC.

TG

TH (11)

INDICATIONS BIBLIOGRAPHIQUES

Nombre de pages : 134

Nombre de volumes : 1

Nombre de références bibliographiques : 14

Présence de matériel d'accompagnement : OUI — NON

S'IL Y A LIEU : IDENTIFIANT DE LA THESE ATTRIBUE PAR L'ETABLISSEMENT

TX (1 2) 00 (11)

NUMERO D'IDENTIFICATION : 95PA07701 6

A REMPLIR PAR LE CANDIDAT - DACTYLOGRAPHIER OBLIGATOIREMENT EN UTILISANT MINUSCULES ET MAJUSCULES, ET EXCLUSIVEMENT DES CARACTERES LATINS.
Aucun symbole spécial (alphabet grec, chiffres romains, symboles mathématiques...) n'est admis.

TITRE DE LA THESE EN ANGLAIS

Design, implementation and evaluation of a programming language suitable for industrial applications : llC.

RESUME DE LA THESE EN FRANCAIS

Le développement d'applications informatiques passe souvent par l'utilisation de langages évolués pour la programmation et de systèmes d'exploitation pour la gestion de l'exécution. Une famille de langages de programmation (les LPC pour "Langages à Parallélisme Compilé") peuvent procurer les mêmes fonctionnalités et avantages qu'un langage de programmation couplé à un système d'exploitation multi-tâche. Notre intention est de montrer que les LPC ont certains attraits supplémentaires surtout dans le domaine d'application qu'est l'informatique industrielle. Pour mettre ceci en évidence, nous avons développé plusieurs versions d'une même application en utilisant d'une part un LPC et, d'autre part, un langage évolué classique avec un système d'exploitation. Ensuite, nous avons fait une comparaison chiffrée des différentes versions.

RESUME DE LA THESE EN ANGLAIS
(facultatif dans les disciplines de santé)

Computing applications are often written using a high level computer language for programming and an operating system for execution handling. A family of programming language (CPL : "Compiled Parallelism Languages") offers features and advantages similar to a programming language coupled with a multi-task operating system. We intend here to point out that CPLs bring several more advantages especially for industrial computing applications. To carry this out, we developed several versions of an application using a CPL on one hand, and a common language with an operating system on the other hand. We then compared the versions.

35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52

11 13 132

Un exemplaire du guide "le signalement et la valorisation de la thèse" sera remis avec les 3 formulaires à chaque candidat.

Les informations demandées sur ce formulaire sont destinées :

1 - à l'exception de l'adresse et de la date de naissance, à la constitution de la banque de données TELETHESES.

La déclaration est obligatoire en application de l'arrêté du 25 septembre 1985 relatif au dépôt, signalement et reproduction des thèses ou travaux présentés en soutenance en vue du doctorat.

Le droit d'accès et de rectification prévu par la loi n°78-17 du 6 janvier 1978 relative à l'informatique, aux fichiers et aux libertés, peut s'exercer auprès du :

Centre national du Catalogue collectif national - TELETHESES, 5 rue Auguste Vacquerie, 75116 PARIS.

Le fichier TELETHESES a fait l'objet d'une déclaration à la CNIL (arrêté du 10 septembre 1991 ; article IV publié au Journal officiel du 1 octobre 1991).

2 - à l'exception de l'adresse de l'étudiant, des titre, résumé et mots clés de la thèse, à la constitution de la banque de données DOCT à finalité statistique.

Le droit d'accès et de rectification prévu par la loi n°78-17 du 6 janvier 1978 relative à l'informatique, aux fichiers et aux libertés, peut s'exercer auprès du service de la scolarité de l'établissement d'enseignement supérieur habilité à délivrer le doctorat dans lequel l'étudiant est inscrit.

Le fichier DOCT a fait l'objet d'une déclaration à la CNIL (arrêté du 13 septembre 1991 publié au B.O.E.N. n°39 du 7 novembre 1991).

MOTS-CLES OU VEDETTES-MATIERE EN FRANCAIS (1 mot-clé par case)

LANGAGE DE PROGRAMMATION	SYSTEME D'EXPLOITATION
INFORMATIQUE INDUSTRIELLE	
INFORMATIQUE TEMPS-REEL	
MARTELISME COMPILE	

63
64
65
66
68
69
70

INFORMATIONS SUR LA SCOLARITE : DEA

Code 939909 Intitulé

RATTACHEMENT DU LABORATOIRE D'ACCUEIL DU DOCTORANT

Etablissement d'enseignement supérieur	Non	0	Université	1	Ecole d'ingénieur	2	Autre	3	
Organisme public de recherche	Non	0	CNRS	1	INSERM	2	INRIA	3	Autre 4
Organisme privé	Non	0	Oui	1					

Autorisation, par le jury, de reproduction de la thèse → X 1 OUI | 2 NON | 3 NON (corrections non effectuées)

DATE ET SIGNATURE DU CANDIDAT	CACHET DU SERVICE DE DOCTORAT	DATE ET CACHET DE LA BIBLIOTHEQUE DE L'ETABLISSEMENT
05/01/95		BIUS JUSSIEU THÈSES PARIS

N.B. : La non-autorisation de reproduction ne dispense pas du signalement dans la base de données Téléthèses