Etude d'une réalisation mixte (Asynchrone/Synchrone) d'un système temps-réel

ANDRE Charles - FANCELLI Luc

Laboratoire de Signaux et Systèmes
Equipe de l'I3S - URA 1376 du CNRS
Université de NICE - Sophia Antipolis
41 Bd Napoléon III F-06041 NICE Cédex

Résumé: Notre étude traite de la programmation de la partie commande de systèmes automatisés. Il s'agit de systèmes temps-réel que l'on peut concevoir comme un ensemble de tâches coopérantes s'exécutant en parallèle et soumises à des contraintes temporelles souvent strictes. Pour leur mise en œuvre effective, nous présentons une approche mixte, asynchrone/synchrone, alliant les possibilités offertes par l'utilisation d'un exécutif temps-réel du type SCEPTRE à l'efficacité et au déterminisme du code produit par le compilateur d'un langage synchrone (ESTEREL). Cette approche est basée sur une décomposition de l'application en tâches (entités actives) coopérantes traitant les fonctionnalités de l'application (partie transformationnelle). Celles-ci sont implémentées sous forme de tâches dont l'exécution peut être différées. Leur gestion est confiée, comme d'ordinaire, à un exécutif temps-réel, mais également à l'automate engendré par le compilateur du langage synchrone ESTEREL. Cet automate est une traduction du noyau de synchronisation de l'application et a pour objet de prendre en compte les synchronisations fortes telles que les suspensions de processus et les arrêts d'urgence, de même que certaines activités réflexes de l'automatisme à concevoir. Ces traitements constituent la partie réactive de l'automatisme.

Dans cet article, nous comparons, les mérites respectifs des approches purement asynchrone, purement synchrone et mixte. Nous tirons de cette étude des enseignements pour la conception d'applications présentant à la fois des caractéristiques de systèmes transformationnels et réactifs.

mots clés: Systèmes temps-réel, parallélisme, réalisation, langages asynchrones, langages synchrones.

Abstract: Our study deals with the implementation of the control part of an automatic system. Those "real-time" systems can be designed as a collection of concurrent tasks running under timing constraints. In order to program such systems we propose a mixed approach (asynchronous and synchronous). This approach inherits the tasking management from a SCEPTRE-like real-time executive and the determinism of the code from a synchronous language compiler (ESTEREL). The application is decomposed into cooperating tasks. The functionalities associated with the tasks are programmed by an algorithmic language invoking the primitives of a real-time executive (transformational part). The synchronization skeleton takes into account the strong synchronizations such as process suspensions and reflex activities (reactive part). It is expressed by an ESTEREL program. Tasks are scheduled by the kernel of the executive **and** by the automaton generated by the ESTEREL compiler.

In this paper we compare purely asynchronous, purely synchronous and mixed approaches. Then we outline a method suitable for the design of systems which are both reactive and transformational.

keywords: Real-time systems, concurrency, implementation, asynchronous languages, synchronous languages.

I. INTRODUCTION

Nous nous intéressons à des **réalisations programmées d'automatismes** présentant à la fois des caractéristiques des systèmes transformationnels et des systèmes réactifs. Ces qualificatifs ont été introduits par D. HAREL et A. PNUELI [HaPnu85] pour distinguer deux grandes classes de systèmes programmés. Pour les premiers, les évolutions se font à l'initiative du programme et, ils sont essentiellement chargés de fournir des résultats à partir de données. Les seconds, sont quant à eux, soumis à un environnement qui leur est imposé et les évolutions sont pilotées par des signaux (occurrences d'événements) provenant de cet environnement.

Industriellement, de nombreux systèmes réactifs sont réalisés à l'aide d'automates programmables. Ce choix est justifié par le fait que les réactions à produire par le système de commande sont du type "réflexe", que les informations provenant de l'environnement sont certes nombreuses mais simples (souvent binaires) et que les algorithmes de traitement sont élémentaires. Cependant, lorsque l'automatisme doit également réagir à des informations plus riches nécessitant par exemple des opérations d'archivage, d'analyse syntaxique et d'interprétation, les automates programmables deviennent insuffisants et on a généralement recours à des ordinateurs.

Les **systèmes** dits **"temps-réel"** appartiennent à ce type d'automatisme nécessitant des échanges pouvant être riches en informations mais également des activités réflexes. Leur mise en œuvre laisse apparaître des caractéristiques différentes de celles rencontrées en informatique classique ou séquentielle du fait du **parallélisme** et de la **coopération** (existant entre les entités actives) qu'ils induisent. Outre ces aspects de "communication", de "synchronisation" et de partage de ressources, ces systèmes sont soumis à des **contraintes temporelles**, liées aux notions de promptitude et de débit d'informations, qu'il est impératif de contrôler pour satisfaire au bon fonctionnement du processus étudié.

Ces systèmes sont généralement constitués d'une partie opérative (PO) et d'un **système de contrôle/commande** (SCC) assujetti à l'évolution dynamique du procédé. L'exigence temps-réel nécessite quant à elle que le SCC réponde à tout stimulus externe en un temps qui garantisse que le système automatisé reste parfaitement maîtrisé. Dans le but de contrôler et de piloter le procédé, le SCC est relié à des capteurs, grâce auxquels il a une image, souvent fugace, de l'état de l'environnement (PO). La commande est quant à elle effectuée par l'intermédiaire d'actionneurs. L'automatisme au sein de tels systèmes est généralement développé suivant deux types d'approches: l'**approche asynchrone**, qui s'appuie sur un langage temps-réel ou sur un exécutif multitâche temps-réel, et l'**approche synchrone** qui peut être abordée à l'aide d'un langage synchrone.

L'approche asynchrone favorise la décomposition de l'application en niveaux hiérarchiques et met en valeur ses fonctionnalités. Ceci permet de concevoir des systèmes complexes. A partir de là, une structuration en tâches (ensemble d'actions à exécuter séquentiellement) est effectuée. Grâce aux possibilités de l'outil logiciel utilisé (exécutif, langage temps-réel), les tâches peuvent s'exécuter en parallèle (parallélisme apparent pour un système monoprocesseur).

- Dans le cas des exécutifs, les mécanismes permettant le fonctionnement multitâche sont invoquées par des appels procéduraux, et la gestion des tâches est à la charge du noyau.

- Dans le cas des langages asynchrones modernes, la notion de tâche est un concept de base. Des instructions particulières facilitent la communication et la synchronisation entre entités actives. Généralement, le compilateur construit un gestionnaire de tâches directement inclus dans l'application.

Cette approche asynchrone du problème facilite la représentation du parallélisme et la coopération entre tâches. De plus, la phase de mise au point est facilitée par la décomposition de l'application. <u>Cependant, son efficacité est réduite par les nombreuses commutations de contexte, et son comportement n'est pas suffisamment prédictible pour des applications fortement contraintes temporellement. Ce dernier inconvénient est une conséquence directe des possibilités de</u>

préemption et de l'asynchronisme des communications. La vérification des contraintes est par là même difficile et souvent impossible.

L'approche synchrone trouve toute sa mesure dans le cas de systèmes complètement réactifs. Les modèles adaptés à ce mode de conception sont basés sur l'hypothèse de forte synchronicité**:** la réponse à tout événement externe est supposée synchrone à leur occurrence. Le GRAFCET est un modèle synchrone très apprécié pour la mise en œuvre des automatismes logiques. Plus récemment, les langages synchrones ont été développés. Ils permettent d'appréhender des traitements plus complexes et bénéficient des avantages offerts par les langages de programmation (structuration de haut niveau des programmes, des données et du contrôle). Un compilateur de ces langages engendre un automate pouvant être considéré comme un "ordonnanceur compilé". Le parallélisme du programme est exploité et mis à plat lors de la phase de compilation ce qui permet d'éliminer les surcoûts (overheads) de gestion à l'exécution et donc d'accroître les performances du système. Il s'en suit une grande efficacité à l'exécution. Cependant, les aspects de coopération sont réduits à leur plus simple expression ce qui tend à limiter les domaines d'application de ces langages aux systèmes essentiellement réactifs. En effet, ils sont, par leurs caractéristiques, mal adaptés pour les traitements algorithmiques ou pour des échanges de messages nécessitant des temps d'exécution importants.

Afin de permettre une programmation lisible, correcte et non ambiguë d'un système temps-réel, nous avons retenu une mise en œuvre alliant le déterminisme et l'efficacité des langages synchrones à la décomposition hiérarchique et fonctionnelle favorisée par les approches asynchrones.

Pour mieux cerner les spécificités des systèmes réactifs et transformationnels, nous présentons dans une première partie un exemple simplifié qui servira de support à notre argumentation. La structuration hiérarchique de cette application met en évidence le noyau de synchronisation constituant sa partie réactive et la partie plus algorithmique reflétant sa partie transformationnelle. Dans un deuxième temps, nous exposons les problèmes spécifiques aux réalisations purement asynchrones ou purement synchrones et nous présentons les motivations qui nous ont amenés à choisir une approche mixte asynchrone/synchrone pour la mise en œuvre des systèmes temps-réel. Nous évoquons dans une troisième partie, les problèmes et les contraintes d'implémentation induits par une telle démarche. Nous concluons par une présentation des apports d'une telle approche pour la conception de ces types d'applications.

II. PRESENTATION DE L'APPLICATION

L'application retenue, adaptée d'un exemple proposé dans [Yourdon85] met en évidence certaines caractéristiques inhérentes aux systèmes temps-réel. La prise en compte de contraintes temporelles à caractère multiforme [BeCoGo87] n'est pas le sujet principal de cette étude. L'application présentée n'est qu'un support permettant d'appuyer notre technique de mise en œuvre sur un exemple pratique.

II.1 PRESENTATION GENERALE

Le système de contrôle/commande à concevoir doit permettre de piloter un environnement constituant une chaîne de remplissage de bouteilles. Ce système est constitué d'une cuve contenant un liquide dont le pH doit être maintenu constant et dont le niveau doit être contrôlé. Un opérateur peut, à tout moment, lancer le système de remplissage. Cette opération consiste à déclencher le positionnement d'une bouteille sur un tapis d'acheminement. En bout de course, la bouteille s'arrête sur le plateau d'une balance et percute un capteur indiquant sa présence. Cette occurrence déclenche l'ouverture de la vanne de remplissage. Lorsque le poids de la bouteille correspond à la valeur souhaitée, le remplissage est achevé, la bouteille est étiquetée et est retirée du plateau. Cette action a pour principal effet de relâcher le capteur de présence permettant ainsi à une autre bouteille d'être placée sur le tapis d'acheminement. L'opérateur de ligne peut à tout instant suspendre l'exécution de la ligne pour réaliser des opérations externes ou pour intervenir directement. Les systèmes de régulation de pH et de niveau doivent s'exécuter de façon parallèle car il est certain que le remplissage de la bouteille nécessite le réajustement du niveau du liquide et une réévaluation de la valeur du pH. Si la valeur du pH dépasse une valeur limite (pH>pHref+0,3), le système entier doit être suspendu et le pH doit être corrigé manuellement. Un superviseur peut inhiber ou permettre le lancement du système complet à tout instant.

Les figures 1 schématise la partie opérative. La figure 2 structure l'application en fonctionnalités et montre clairement l'influence sélective des événements.

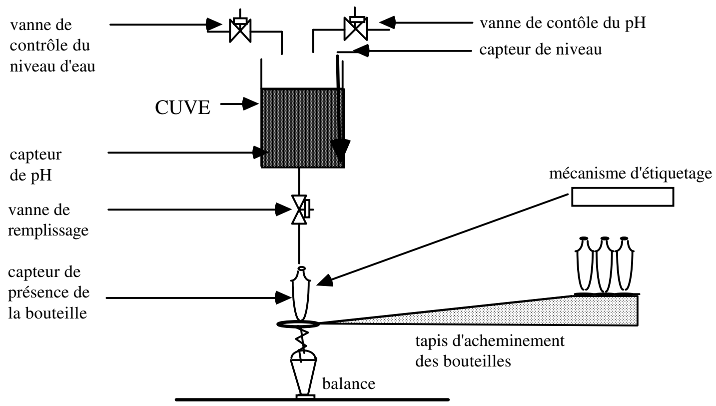

fig.1: chaîne de remplissage de bouteilles

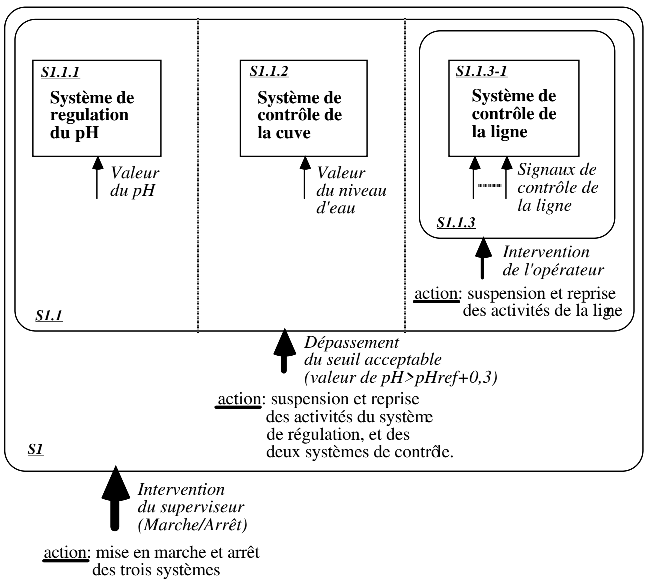

fig.2: schéma de la hiérarchisation du système de remplissage de bouteilles

II.1. DECOMPOSITION DE L'APPLICATION

Il apparaît au sein de notre application deux parties différentes au sens du comportement:

- L'une, essentiellement pilotée par des événements externes ("event driven"), présente des caractéristiques propres aux systèmes réactifs. Les informations échangées sont si simples que des mécanismes de signalisation sont suffisants pour déclencher l'exécution de traitements effectués dans la seconde partie. On se ramène ici à de simples synchronisations ou à des activités d'entrées/sorties de bas niveau.

- La seconde partie nécessite des traitements de type algorithmique (interprétations de commandes, algorithmes de régulation, échanges de messages riches en informations) et elle est de ce fait davantage orientée vers un comportement relevant des systèmes transformationnels.

remarque: une telle décomposition peut être étendue à une large classe de systèmes temps-réel. En effet, de tels systèmes se distinguent des systèmes complètement réactifs ou purement transformationnels par le fait qu'ils allient des caractéristiques propres à chacune de ces deux grandes classes d'automatismes programmés. Il est donc judicieux de distinguer au sein de ces systèmes, une partie traitant essentiellement des coopérations entre entités actives du procédé (transfert de messages riches en informations, transformations et interprétation de commandes) et une partie traitant des activités de synchronisation pures ou d'activités réflexes telles que les entrées/sorties de bas niveau. Notre stratégie est donc de distinguer la partie transformationnelle chargée des traitements algorithmiques, de la partie réactive caractérisant le squelette de synchronisation et chargée de la gestion des évolutions du système complet. Ainsi, sur l'exemple (fig.2), l'intervention du superviseur, le dépassement du seuil acceptable, l'intervention de l'opérateur et à un niveau moindre, les acquisitions des signaux de contrôle de la ligne, des valeurs du pH ou du niveau d'eau sont des contraintes de synchronisation fortes du système qui nécessitent des interventions immédiates et synchrones à leur occurrence. Cette partie revêt l'aspect d'un ordonnanceur d'événements hiérarchisés et correspond à la partie réactive de notre application. D'un autre coté, les systèmes S1.1.1, S1.1.2, S1.1.3-1 et les traitements résultant de l'occurrence des contraintes de synchronisation citées requièrent des traitements plus élaborés mettant en jeu des modifications de structures de données ou des traitements plus importants. Ces dernières parties correspondent à la partie transformationnelle de notre application.

III. METHODES DE DEVELOPPEMENT

Avant de décrire l'approche mixte asynchrone/synchrone préconisée, nous rappelons les caractéristiques propres à chacune des techniques de développement (purement asynchrone ou purement synchrone).

III.1. APPROCHES ASYNCHRONES

Ces approches sont liées à un mode de programmation dans lequel les communications sont asynchrones et dédiées à un partenaire bien identifié. Cet asynchronisme est la source même du non déterminisme de ces approches.

III.1.1. Présentation générale

L'approche asynchrone que l'on a d'un système est intrinsèquement liée à l'idée que l'on se fait du problème. Ainsi, dans un tel type de démarche, on sera plutôt tenté de concevoir l'application à traiter en définissant une, voire des fonctions de transformation entre un ensemble de données et un ensemble de résultats. Ce parallélisme de conception [Rayn81] engendre une décomposition de l'automatisme sous forme structurée, hiérarchique et fonctionnelle. La mise en œuvre de tels systèmes dit "systèmes transformationnels" est généralement effectuée par le biais d'un "outil" asynchrone qu'il soit de type langage ou exécutif. Pour ces "outils", le concept de base repose sur la création de processus (tâches) pouvant s'exécuter en parallèle et communiquant de manière asynchrone. Ces entités actives sont constituées d'actions élémentaires s'exécutant séquentiellement. Le comportement du système de contrôle est alors conçu sous la forme d'un ensemble d'actions dont les instants d'activation sont conditionnés par l'occurrence d'événements émis par d'autres actions ou par le procédé contrôlé. Les événements sont liés à un certain degré d'urgence (relatif à la priorité de la tâche à lancer) pouvant nécessiter la préemption d'un processus en cours d'exécution (si le nombre de processus est supérieur au nombre de processeurs de l'architecture).

Avec l'utilisation d'un exécutif, le multiplexage dynamique des actions est transparent à l'utilisateur. Il est effectué par un noyau chargé de l'ordonnancement. Celui-ci fournit au concepteur un ensemble de primitives (actions atomiques) qui lui permettent la manipulation d'objets favorisant la coopération (communication et synchronisation entre tâches) et la gestion du temps. Une proposition de norme, SCEPTRE [BNI84], a été avancée par le BNI[1]. Elle propose un ensemble de primitives couvrant les principales fonctions pour la réalisation d'applications temps-réel. Cet ensemble n'est portable que par sa spécification et il faut prévoir une réalisation adaptée à chaque machine disposant de ce noyau. Cependant, si cet effort est fait, les programmes se limitant à l'utilisation des primitives proposées seront portables sur toutes machines disposant de ce noyau [BNI84].

L'utilisation d'un langage asynchrone peut décharger l'utilisateur de la programmation de mécanismes de coopération à condition qu'il se limite aux mécanismes implémentés par le langage (Rendez vous ADA, par exemple). Ceci n'exclut pas que le concepteur puisse définir son propre ordonnanceur (solution couramment rencontrée en Modula2 [Schip86]).

Très souvent, les applications développées utilisent les aspects multitâche mais ne prennent pas réellement en compte les aspects temporels. Dans les exécutifs dits temps-réel, on trouve des extensions par rapport à SCEPTRE. Le but de ces extensions est d'utiliser une "horloge temps-réel" afin de permettre des sollicitations périodiques et des mécanismes de temporisation limitant les attentes (timeouts). Il est à noter que les langages asynchrones essaient aussi d'inclure de tels concepts (le "delay" en ADA par exemple).

III.1.2. Exemples informels d'implémentation

Nous présentons deux exemples (issus de notre application) qui illustrent l'intérêt de cette approche asynchrone et qui nous sensibilisent aux problèmes résultants de leur implémentation.

[1] BNI : Bureau d'orientation de la Normalisation en Informatique.

Exemple 1 : *Communication par boîtes à lettres et niveaux de priorité.*
L'application étudiée permet la saisie de commandes provenant d'un clavier et émises par l'opérateur de ligne. Pour gérer cette acquisition, une tâche ad hoc doit être développée. Cette tâche construit progressivement la commande. Lorsque celle-ci est complètement définie, la tâche communique l'information à un interprète de commandes par l'intermédiaire d'une boîte à lettres. A propos de cette tâche, il est important de noter que la programmation multitâche doit être abordée différemment d'une programmation séquentielle ordinaire. En effet, l'instruction de lecture d'un caractère est traduite par le compilateur comme une attente active et, afin de ne pas bloquer le système, il est nécessaire d'affecter une priorité faible à cette tâche. Le choix des priorités présente souvent un caractère critique dans un système multitâche.

Exemple 2 :*Tâches périodiques, interruption et préemption.*
La programmation de système de commande requiert généralement l'activation périodique de tâches. C'est le cas dans le système de commande étudié où les systèmes de régulation du pH et du niveau d'eau doivent être activés périodiquement. Cette périodicité peut être obtenue par l'installation de gérants (handlers) d'interruption sollicités par une horloge physique. Les exécutifs offrent des facilités quant à la conception et l'installation de ces gérants (utilisation des objets de type "delay" en Realtime Craft[2] par exemple). A chaque sollicitation (mécanisme d'interruption), le gérant exécute son code en dehors du contrôle de l'ordonnanceur. Il est bien évident que cette exécution doit être brève et ne doit pas modifier directement les objets manipulés par les tâches différées dont l'ordonnanceur à la responsabilité. Généralement, le gérant se contente de signaler des événements ou de mettre des informations dans une file. Si cet usage des interruptions s'avère commode pour l'utilisateur, il n'en est pas moins dangereux. On oublie trop souvent, que l'entrée dans une région inhibe l'ordonnanceur mais en aucun cas les interruptions.

III.1.3. Problèmes inhérents à cette approche

Ces outils favorisent une méthode de programmation structurelle et simple. Les mécanismes de coopération et d'expression du parallélisme diffèrent sémantiquement et syntaxiquement d'un langage à un autre ou d'un exécutif à un autre mais restent basés sur des concepts communs. Il résulte d'une telle approche une décomposition claire et hiérarchisée de l'application. Cependant, si les aspects communications et synchronisations sur événements avec "timeout" sont favorisés par une telle démarche, il est à noter qu'elle est très pénalisante sur de nombreux points. Ainsi, à cause de l'asynchronisme, les appels au noyau ne sont pas exécutés dans un ordre figé. Les exécutions ne sont pas strictement reproductibles et, de ce fait, il est pratiquement impossible d'évaluer les retards introduits par ces appels enchaînés. Pour garantir les temps de réponse, il est nécessaire d'être plus contraignant en choisissant par exemple des stratégies d'ordonnancement particulières [ZRS87]. Au niveau de la mise au point, les tests sont rendus très difficiles du fait du non déterminisme. Enfin, on relèvera que leur sémantique informelle n'est pas réellement un atout.

III.2. APPROCHE SYNCHRONE

L'approche par langage synchrone se distingue des approches asynchrones traditionnelles par le fait qu'elle est basée sur une hypothèse de forte synchronicité apportant déterminisme et efficacité à l'automate produit après compilation. Le mode de communication choisi est le mode par diffusion instantanée des signaux. Ce mode de communication assimilable à une transmission

---

[2] Realtime Craft (RTC) est un exécutif temps-réel multitâche basé sur SCEPTRE [GSI87] et développé par la société GSI-TECSI. Il est fortement orienté sur SCEPTRE et est enrichi de notions temporelles (attente avec timeout), et d'objets de niveau plus élevé (sémaphores, boîtes aux lettres, retards).

par radio permet à l'ensemble des modules réceptifs de recevoir la même information au même instant. La prise en compte de l'information n'est pas différée comme elle peut l'être dans le cadre des approches asynchrones.

III.2.1. Présentation générale

Les langages synchrones ont été étudiés initialement pour procurer une méthode de programmation naturelle des activités de contrôle/commande de processus temps-réel, d'automatismes divers (pouvant faire intervenir des contraintes temporelles) ou plus généralement de systèmes réactifs. Ces langages abandonnent l'hypothèse d'asynchronisme à la base des langages classiques et la remplacent par une hypothèse de synchronicité plus ou moins forte [BCCG85]. Dans une telle hypothèse, le système fonctionne "comme si" les sorties étaient synchrones aux entrées et les actions étaient exécutées en ne "prenant pas de temps". Ainsi, le programme réagit "instantanément" à une entrée en mettant à jour son contexte et en produisant les sorties correspondantes. Le modèle d'exécution est alors rendu déterministe ce qui à des conséquences importantes pour la conception d'une application:

• Par compilation, on peut obtenir un automate à états finis déterministe, sémantiquement équivalent à un programme parallèle. Cette possibilité de mettre "à plat" le parallélisme dans la phase de compilation permet de supprimer le temps perdu à l'exécution pour les commutations de contexte.

• Pour ces langages, tout événement ou signal externe est considéré comme un élément de synchronisation. Chaque signal peut servir à définir comme une échelle de temps pour l'ensemble du programme. La prise en compte de temps "multiforme" ne pose pas de problèmes particuliers.

• Sur les programmes conçus à l'aide de ces langages, le temps d'exécution peut être calculé et l'approximation de temps nul peut être éprouvée vis à vis du comportement de l'environnement. De plus, le franchissement d'une transition de l'automate s'effectue en temps constant et ce, quelle que soit la taille du programme. On peut ainsi maîtriser les temps d'exécution et vérifier si les contraintes temporelles sont respectées. Ceci permet également de savoir si la modélisation synchrone (idéalisation) est physiquement acceptable. Il est certain que pour un procédé physique ayant une constante de temps de l'ordre de l'heure, des réactions s'exécutant en 30 mn, ne sont pas en violation avec l'hypothèse de synchronicité.

Au niveau des communications, ces langages sont basés sur un mécanisme unique de "diffusion instantanée" des signaux (cas d'ESTEREL [BeGon]) ou de flots de données synchrones (cas de LUSTRE [HCPP87] et de SIGNAL [BBGG85]). Les communications sont prises en compte par le compilateur, mais, contrairement aux solutions asynchrones, il n'y aura d'appels à des mécanismes spécifiques à l'exécution.

Outre les avantages précédemment cités, on peut signaler que des tests de cohérence sont possibles sur ces programmes grâce à des règles d'écriture (syntaxiques et contextuelles) strictes [HPOG88]. La sémantique de ces langages est définie de façon mathématique. Cette caractéristique permet d'obtenir après compilation un automate traduisant fidèlement le programme écrit. De plus, les instructions sont peu nombreuses, mais puissantes et bien ciblées, si bien que le concepteur se familiarise rapidement avec leur syntaxe et leur sémantique.

III.2.2. ESTEREL

Si les exécutifs temps-réel sont largement répandus, il n'en est pas de même pour les langages synchrones et notamment pour ESTEREL que nous utilisons. Notre but n'est pas de le présenter complètement. Nous voulons seulement illustrer son apport à la programmation de notre exemple. Pour plus de détails, nous invitons le lecteur à se reporter au manuel du système ESTEREL [BeCoGo87].

Les caractéristiques essentielles de ce langage de haut niveau peuvent être énumérées de la façon suivante:

• Le temps est multiforme et discret. Il correspond à un temps logique permettant d'exprimer de façon tout à fait naturelle les contraintes temporelles auxquelles sont soumis les systèmes. Ce temps logique est compté en nombre d'occurrences d'un signal répétitif et a l'avantage indéniable de donner une vue synchrone d'événements survenant de façon asynchrone dans le temps. Ainsi, la seconde, le mètre, "le nombre de crocodiles traversant une voie ferrée" [BCCG85] peuvent être des temps logiques considérés comme des événements de synchronisation du système. Les constructions du langage portent indifféremment sur tous les signaux.

• La communication est faite par diffusion instantanée de signaux. Du fait de l'hypothèse de synchronicité tout se passe comme si l'ensemble du programme était situé dans un médium de communication accessible par chaque point du programme. Ce mécanisme de communication (voire de synchronisation) est généralement assimilé à un mécanisme de diffusion d'un poste radio. Toutes les personnes à l'écoute peuvent recevoir le message en même temps (du fait de l'hypothèse de temps nul).

• Une suite d'événements est une histoire. Un événement est un mot (pouvant être vide) traduisant l'occurrence de signaux simultanés. De ce fait, une histoire d'entrée du système est une suite d'événements. Le programme réagit en transformateur d'histoires déterministe: il produit la même histoire de sortie pour une même histoire d'entrée dans le même contexte. On remarquera que cette caractéristique est généralement fausse si l'on utilise des approches asynchrones.

• Le style est impératif. Le langage dispose des structures de contrôle itérative (loop…end) et conditionnelle (if…then…else…end) de même que des opérateurs de composition séquentielle (;) et parallèle (| |). Il est à noter que l'hypothèse de synchronicité induit des couplages très forts entre branches parallèles. Ces couplages fréquents dans les systèmes réactifs sont difficiles à exprimer dans les approches asynchrones. Les instructions ESTEREL manipulent soit des variables locales qui peuvent être modifiées par affectation soit des signaux purs ou valués. Le langage dispose de primitives impératives classiques, de primitives temporelles, de la structure de module permettant une programmation hiérarchique et d'une instruction permettant d'arrêter toute évolution (halt). L'instruction la plus originale et la plus puissante est incontestablement la préemption instantanée (do…watching…). Elle permet notamment d'exprimer des gardes temporelles multiformes. **%** débute un commentaire qui se poursuit jusqu'en fin de ligne. **?** X permet de récupérer la valeur portée par le signal ou le capteur "X".

La syntaxe et la sémantique des instructions ESTEREL sont largement présentées dans la bibliographie concernant ce langage.

III.2.3. L'automate ESTEREL

La compilation d'un programme ESTEREL produit un automate à états fini déterministe du type MEALY généralisé (du fait de l'utilisation de variables internes). L'automate engendré est un programme séquentiel efficace, sémantiquement équivalent au programme initial et caractérisant un système de contrôle/commande dont les seuls instants d'activation sont ceux engendrés par l'occurrence de signaux externes purs ou valués. L'automate peut prendre connaissance de l'état de capteurs grâce à des éléments particuliers appelés: "Sensors" [BeCoGo87].

On constate que par ses caractéristiques structurantes, impératives et déterministes, le langage ESTEREL semble particulièrement bien adapté pour transcrire le squelette de synchronisation de notre application. La hiérarchisation appartenant au langage, il est donc simple de contrôler l'agencement de tâches prenant en compte les fonctionnalités de l'automatisme et de commander soit des périphériques d'entrée/sortie soit l'exécution de tâches spécifiques. La simplicité de conception de ce noyau synchrone réside dans la possibilité d'exprimer un parallélisme logique pénible à traduire en code séquentiel. L'utilisation d'ESTEREL permet ici de bénéficier du travail produit par le compilateur qui a à charge de traduire le parallélisme. Il est à noter que cette démarche n'est pas simple si l'on considère une approche purement asynchrone. Cette difficulté amène souvent à des choix d'implémentation n'optimisant pas les liens entre les différentes contraintes de synchronisation.

III.2.3 Problèmes inhérents à cette approche

Les adeptes de la programmation asynchrone sont généralement choqués par l'hypothèse de forte synchronicité. Dans les applications courantes, de nombreuses actions nécessitent une durée d'exécution non négligeable. Pour rester dans le cadre d'une approche purement réactive et donc favorable aux langages synchrones, on est amené à distinguer des signaux marquant le début et la fin de ces "actions qui durent". Mais, ceci ne fait que reporter les exécutions correspondantes dans l'environnement de l'automate.

III.3. APPROCHE MIXTE ASYNCHRONE/SYNCHRONE

Pour pallier les difficultés relatives à un développement suivant une approche complètement asynchrone ou purement synchrone, nous avons choisi d'étudier une approche mixte qui tire avantages des possibilités offertes par les deux approches. L'originalité de notre démarche réside dans le fait de concevoir ces deux approches comme complémentaires pour la mise en œuvre des systèmes temps-réel.

Comme nous l'avons laissé entendre dans la partie traitant de la décomposition de l'application, notre démarche est basée sur la structuration de l'application en deux parties. L'une, essentiellement réactive, traduit le squelette de synchronisation de l'application de même que certaines activités réflexes ne requérant pas de "temps d'exécution" (au sens des langages synchrones). L'autre, transformationnelle par essence, prend en compte les fonctionnalités requises pour le système de commande.

Ces types d'applications présentent donc des caractéristiques nécessitant des traitements algorithmiques pour la partie transformationnelle et des traitements synchrones pour la partie réactive. De plus, afin d'accroître les performances de l'automatisme programmé des évolutions parallèles sont requises. Aucun langage algorithmique d'usage général ne facilite ce type de programmation. De plus, même si un langage algorithmique d'usage général permettrait de surmonter ces problèmes, la programmation de la gestion multitâches et de la partie réactive serait un exercice difficile, peu rentable, source d'erreurs et peu réutilisable. Face à cette inadéquation des langages algorithmiques traditionnels à prendre en compte les problèmes posés par la programmation concurrente, les concepteurs se penchent généralement vers une solution consistant à utiliser un langage algorithmique classique (le C par exemple) et des outils dédiés pour traiter ce qui n'est pas directement dans la philosophie du langage. C'est également cette approche que nous avons retenu.

Ainsi, pour la gestion du parallélisme, nous avons recours à un exécutif multitâche. Ne prenant pas en compte les contraintes temporelles indispensables pour nos applications, un système d'exploitation très général comme UNIX a été rejeté. L'emploi d'un système comme OS9 que l'on qualifie souvent d'UNIX temps-réel n'a pas, pour sa lourdeur (héritage d'UNIX) et surtout pour les difficultés de maîtriser les durées d'exécution des primitives du système, retenu notre attention. Nous avons préféré limiter le rôle du système à l'essentiel de la gestion des tâches et de la communication (asynchrone) entre elles. Les exécutifs conformes à la norme SCEPTRE nous ont semblé, de ce point de vue, suffisamment simples et efficaces. Parmi eux, nous avons retenu RealTime Craft (RTC). RTC est une implémentation de SCEPTRE, enrichie d'objets élaborés (boîtes aux lettres, sémaphores, temporisations) et offrant des gestions intégrées pour les temps de garde. Ce type d'exécutif spécialisé est très apprécié pour les applications embarquées soumises à des contraintes temporelles sévères. La taille réduite du noyau (3 K.octets), son caractère "ROMable", sa relative efficacité, et la possibilité de connaître le coût en temps d'exécution des diverses primitives sont autant d'avantages plaidant en sa faveur.

C'est certainement au niveau des aspects réactifs que la programmation classique est la plus impuissante. Un programmeur habile et patient pourrait coder lui-même un automate représentant le squelette de synchronisation de l'application. C'est un exercice périlleux lorsqu'il s'agit d'applications en vraie grandeur où le parallélisme est important et les stimuli nombreux. De plus, la moindre variation dans les spécifications du système peut entraîner une modification radicale de la structure de l'automate. Pour ces raisons, nous avons préféré confier la construction de l'automate au compilateur ESTEREL qui a été conçu spécialement dans ce but et dont une option permet d'engendrer l'automate sous la forme d'un module en langage C.

Compte tenu de toutes ces constatations, nous avons choisi d'utiliser un langage algorithmique classique complété par des outils dédiés d'une part pour les aspects multitâches et communications asynchrones (exécutif RTC) et d'autre part pour les aspects réactifs (langage ESTEREL).

Dans les approches purement asynchrones, la gestion de la coopération entre tâches est confiée au noyau de l'exécutif. Dans notre approche, l'automate, chargé de traduire le squelette de synchronisation de l'application, pourrait apparaître comme un ordonnanceur faisant double emploi avec celui de l'exécutif. Il n'en est rien. En effet, ils jouent ici des rôles complémentaires: alors que l'automate est chargé d'apporter plus de déterminisme et d'efficacité dans l'ordonnancement, l'exécutif est quant à lui chargé de fournir l'environnement d'exécution asynchrone qui convienne à l'automate synchrone. La figure 3 résume notre stratégie d'implémentation.

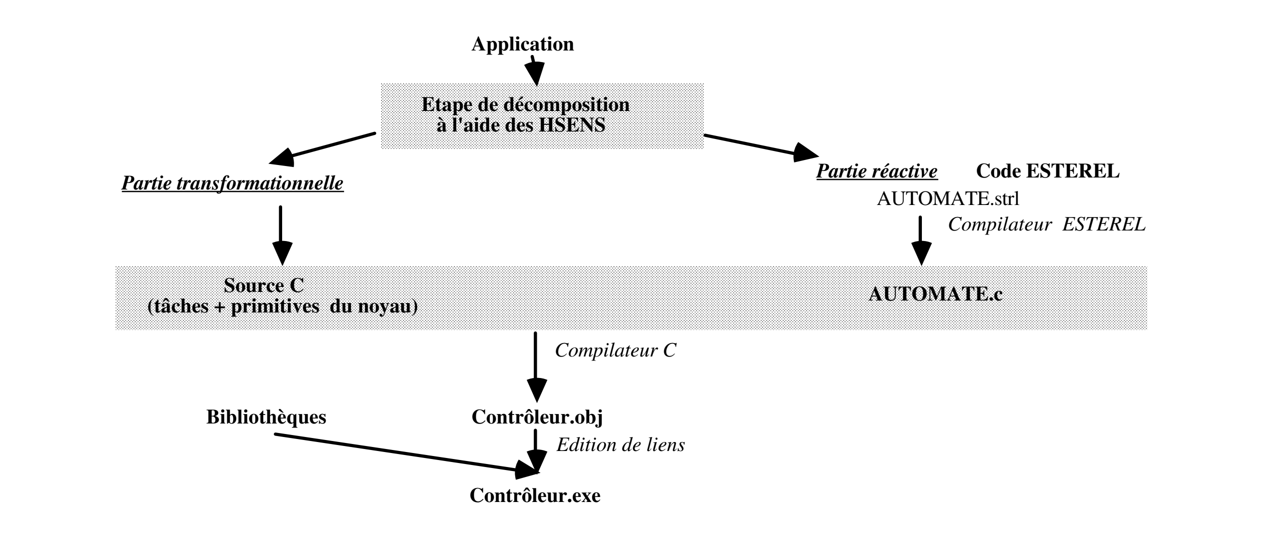

fig.3: stratégie d'implémentation

L'idée de coopération entre les parties transformationnelles et réactives découlent directement de la décomposition fonctionnelle de l'application. Leur mise en œuvre a toutefois soulevé de sérieux problèmes. Cette remarque n'a rien d'étonnant et reflète les différences conceptuelles entourant les approches synchrones et asynchrones et les caractéristiques distinguant les applications transformationnelles des systèmes réactifs.

IV. PROBLEMES D'IMPLEMENTATION

Dans cette partie traitant des problèmes d'implémentation dus à la mise en place d'une approche mixte synchrone/asynchrone nous supposons connus les mécanismes de préemption et les opérations sous-jacentes à une interruption.

IV.1. IMPLEMENTATION DE L'AUTOMATE

Afin de comprendre les problèmes d'implémentation rencontrés, il est important à ce niveau de saisir comment sont exécutés les changements d'état de l'automate produit par le compilateur ESTEREL. Comme nous l'avons mentionné, l'automate produit peut être obtenu sous la forme d'un module C. Celui-ci comprend les structures de données et de contrôle nécessaires pour simuler l'automate qu'il traduit. Cette fonction C, que nous appellerons automate(), exécute à chacun de ses appels, le code correspondant à une transition de l'automate. Puisqu'il s'agit de la réalisation d'un système réactif, les évolutions de cet automate sont déclenchées par des stimuli externes produits par l'environnement. Afin de transmettre ce type d'information (les stimuli) vers l'automate, il est nécessaire de mettre au point une interface logicielle permettant d'une part de signaler à l'automate l'occurrence de signaux externes et d'autre part d'émettre vers l'environnement des actions résultant de l'exécution d'une transition. Cet interfaçage est réalisé par le biais de fonctions d'entrée/sortie (entre l'automate et l'environnement logiciel) (cf. fig.4).

- Les fonctions d'entrée, de nom générique I_<signal>, sont entièrement construites par le compilateur ESTEREL. Leur rôle est de transmettre, lors de leur appel, les occurrences (information fugitive) et, éventuellement, les valeurs des signaux (information rémanente) provenant de l'environnement et à destination de l'automate.

- Les fonctions de sortie, nommées O_<signal>, sont appelées de façon automatique par la fonction automate(). Leur sémantique doit être précisée par l'utilisateur qui est tenu d'écrire le corps de ces fonctions. Chacune reflète l'action devant être effectuée lorsque le signal qui lui est associé doit être émis vers l'environnement.

• Il existe une troisième catégorie de fonction pouvant être appelée par l'automate et dont le corps doit être écrit par le concepteur. Il s'agit des fonctions associées aux capteurs. Celles-ci, nommées S_<signal> permettent d'effectuer des lectures dans l'environnement.

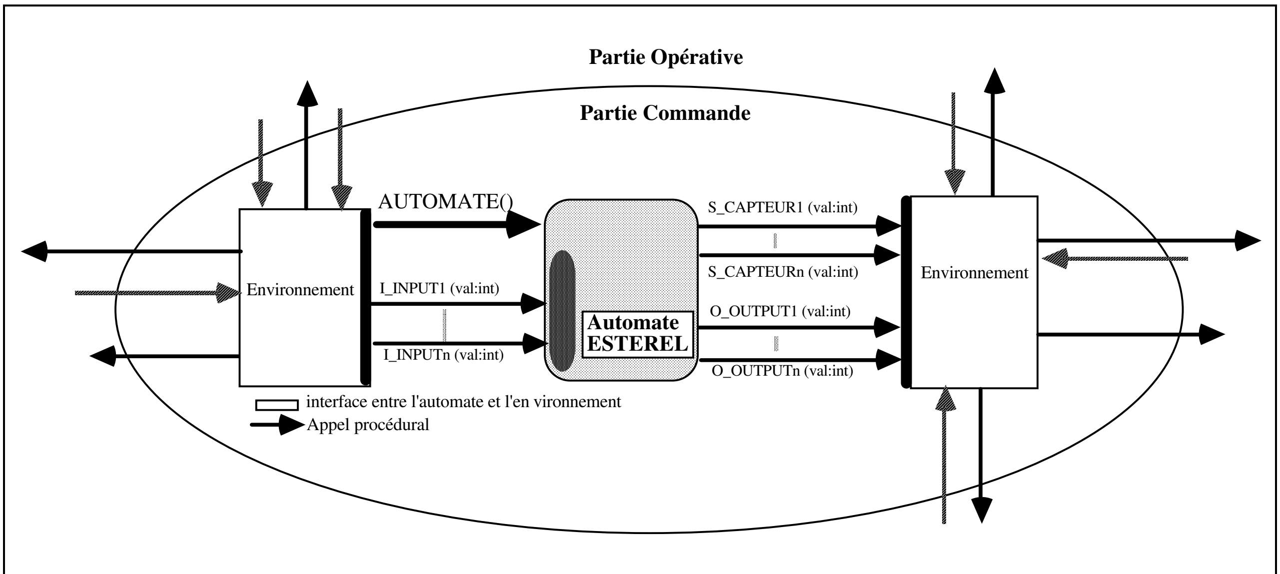

fig.4: situation de l'automate dans le système de contrôle/commande

Par invocation des fonctions d'entrée, on transmet à l'automate une image de l'état de l'environnement. L'invocation de la fonction automate() déclenche quant à elle un changement d'état. Lors de l'exécution de cette fonction les actions suivantes sont effectuées:

• 1) observation de l'image de l'environnement et vérification des réceptivités.
• 2) si une réceptivité au moins est satisfaite
 alors
 a) déterminer LA transition à effectuer (aspect déterministe)
 b) réaliser les actions associées à la transition
 c) aller dans le nouvel état
• 3) retourner à l'appelant

Notons que l'implémentation d'ESTEREL repose fortement sur l'hypothèse de forte synchronicité et de ce fait, afin de respecter la sémantique du langage ESTEREL, une transition doit se faire en "temps nul" (c'est-à-dire dans un temps négligeable vis à vis des constantes de temps du système commandé). Etant donné qu'une transition "ne dure pas", il était inutile de prévoir un automate réentrant. Il est alors de la responsabilité du programmeur de garantir des appels non réentrants à l'automate.

Malgré ces comportements bien ordonnés qui ne posent pas de problème dans un contexte séquentiel, il apparaît que certaines précautions sont à prendre en compte pour insérer l'automate au sein d'une application destinée à s'exécuter dans un environnement multitâches temps-réel.

IV.2. PROBLEME DE POSITIONNEMENT DE L'IMAGE

Dans un système à événements discrets et à évolutions synchrones, l'arrivée d'un signal, la transition induite et les actions associées doivent se faire instantanément. En pratique, il faut réaliser de façon indivisible ces divers traitements. Dans le cas d'une programmation séquentielle aucun problème ne se pose, et toutes les entrées faites avant l'appel à l'automate sont considérées par celui-ci comme étant simultanées. Il n'en va pas de même dans un environnement multitâches si l'on ne prend pas de précautions pour garantir le comportement souhaité. En effet, le problème majeur est qu'il peut y avoir préemption de la tâche positionnant les entrées. Ainsi, sur la figure 5-B une préemption durant le positionnement de l'image par la tâche1 modifiera le comportement de l'automate. De ce fait, l'automate ne perçoit qu'une **image partiellement mise à jour** de l'environnement et évoluera en fonction de cette image erronée. Si la préemption intervient durant l'exécution de l'automate alors il y a possibilité de **violation de la non réentrance** et les résultats ne peuvent plus être garantis. Ces situations peuvent créer des fonctionnements aléatoires inadmissibles.

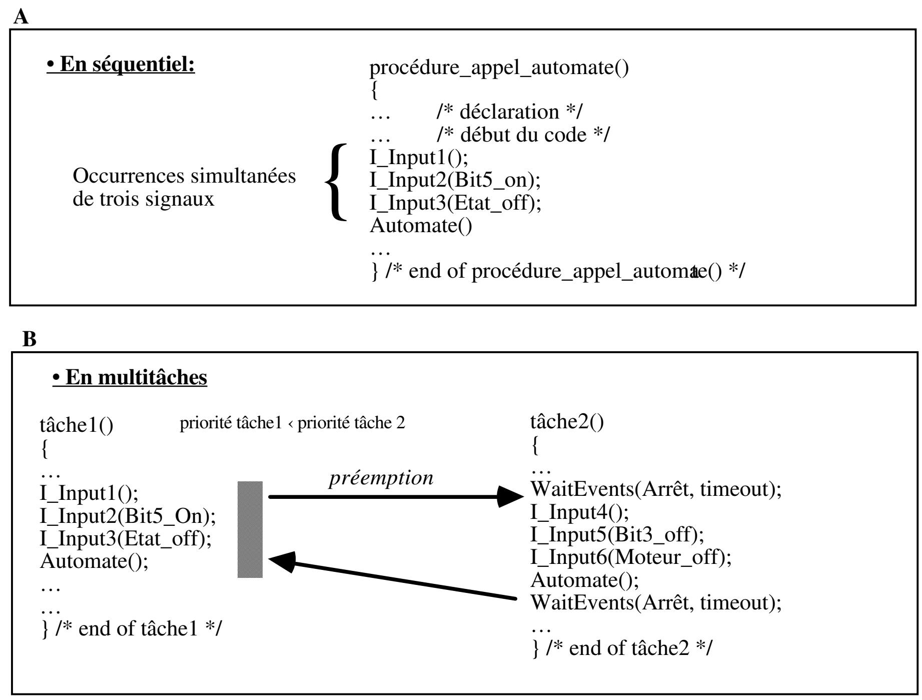

fig.5: problèmes de positionnement de l'image

Cet exemple indique clairement qu'une évolution contrôlée de l'automate nécessitera dans un environnement multitâches des mécanismes d'exclusion permettant de garantir l'exécution indivisible du positionnement des entrées et des transitions de l'automate. L'exécutif utilisé (RTC) procure un type de mécanisme satisfaisant cette contrainte. En effet, les primitives EnterRegion() et LeaveRegion() peuvent être utilisées pour inhiber l'ordonnanceur durant le temps où une tâche positionne ses entrées et fait l'appel à l'automate.

IV.3. DIFFERENTES FORMES DE SYNCHRONISATION

En s'inspirant de ce qui existe en réalisation matérielle on peut envisager deux modes de fonctionnement: un mode **auto-synchronisé** et un mode **synchronisé** (sur horloge). Ces modes de fonctionnement dépendent à la fois du type d'application traitée et du mode fonctionnement que l'on souhaite pour l'exécution. Ainsi, pour une application essentiellement pilotée par l'occurrence d'événements externes, il est préférable de choisir un mode auto-synchronisé. Si l'application nécessite des exécutions ayant une cohérence temporelle forte, il sera préférable de choisir un mode déclenché sur les occurrences d'une horloge de base servant alors de référence temporelle pour l'application. Ces modes de fonctionnement comme nous allons le voir ne dépendent que du positionnement au sein de l'automatisme de l'appel à la fonction représentant l'automate.

A

1) Synchronisation avec appel explicite de l'automate
<u>Mode auto-Synchronisé</u>

<u>Exécution</u>

```
/* code */
…
EnterRegion();          -> 1) l'ordonnanceur est inhibé.
I_Input1();           }
I_Input2(Bit5_on);    }  -> 2) positionnement de l'image.
I_Input3(Etat_off);   }
Automate();             -> 3) évolution de l'automate.
LeaveRegion             -> 4) l'ordonnanceur est validé
…
/* code */
```

B

2) Synchronisation et appel implicite de l'automate
<u>Mode synchronisé sur horloge</u>

<u>Exécution</u>

```
/* code */
…
EnterRegion();          -> 1) l'ordonnanceur est inhibé.
I_Input1();           }
I_Input2(Bit5_on);    }  -> 2) positionnement de l'image.
I_Input3(Etat_off);   }
LeaveRegion             -> 3) l'ordonnanceur est validé
…
/* code */
L'automate évoluera sur le prochain tic d'horloge
```

fig.6: différentes formes de synchronisation

Dans le premier mode (auto-synchronisé), l'arrivée d'un signal provoque l'acquisition de données suivie de la transition de l'automate. C'est l'analogue du mode pulsé pour les circuits séquentiels. Dans le deuxième mode (synchronisé), les phases d'acquisition des entrées et d'évolution de l'automate sont disjointes. Les transitions de l'automate sont déclenchées par un signal répétitif (généralement périodique).

Pour garantir ces modes de fonctionnement, il faut rendre indivisibles et exclusives certaines sections de programme. La phase d'acquisition des entrées ainsi que la phase d'évolution de l'automate doivent être indivisibles. Pour le mode auto-synchronisé, il faut de plus, que ces deux phases soient exécutées séquentiellement et de façon indivisible. Avec un exécutif temps-réel, ceci s'obtient par l'utilisation des objets de type "région". La figure 6 indique clairement la façon d'implémenter les deux modes de fonctionnement.

Dans le cas 1), les appels à l'automate sont explicites et ne sont effectués que lorsqu'il y a occurrence d'événements. Dans le cas 2), les appels à l'automate se font par le biais d'un signal d'horloge émis par une tâche dédiée de forte priorité et ce qu'il y ait eu ou non occurrence d'événements. Pour ce dernier mode, tous les événements se réalisant entre deux Tics d'horloge sont considérés comme simultanés (l'horloge peut évidemment être périodique). Ce type de fonctionnement est courant dans les automates programmables industriels.

IV.4. L'INTERET DES ACTIONS BREVES

Lors d'une transition, les actions sont effectuées par des appels en séquence de fonctions de sortie. On ne sort de l'automate que lorsque toutes les actions sont terminées. Dans la mesure où ces actions sont brèves, elles pourront paraître comme simultanées à leur cause conformément à l'hypothèse de forte synchronicité. C'est le cas par exemple de la fonction O_Acheminer_La_Bouteille() qui se contente d'émettre une commande de mise en route du moteur du tapis d'acheminement (fig.7).

- Code ESTEREL

```
.....
emit   Acheminer_La_Bouteille ();
.....
```

- Procédure appelée dans l'environnement

```
O_ Acheminer_La_Bouteille()
{
  outp (Adresse_Moteur, Moteur_ON);
}
```

fig.7: exemple de procédure de sortie brève

Par contre, les actions qui "durent", posent de sérieux problèmes car l'environnement peut changer de façon significative alors que l'évolution (l'exécution des fonctions de sortie relatives aux signaux reçus) n'est pas terminée. Ainsi, plusieurs TOPS d'horloge peuvent se produire pendant l'exécution d'une procédure de sortie. Ceci a pour conséquence immédiate une distorsion temporelle entre l'état de l'environnement et l'image perçue par l'automate. Une deuxième conséquence peut être le déclenchement d'une nouvelle transition, sollicitée par une tâche plus prioritaire, qui entraînerait un nouvel appel de l'automate. Le code de celui-ci n'étant pas réentrant, l'évolution ultérieure ne serait alors plus cohérente.

remarque: Il est évident, qu'en aucun cas une fonction de sortie ne doit faire appel à des primitives du noyau susceptibles de suspendre la tâche appelante. Cette situation mènerait aux problèmes explicités ci-dessus.

Les interruptions de l'horloge physique sont traitées par un gérant d'interruption qui échappe au contrôle du noyau. A chaque interruption, le gérant exécute sa procédure qui réalise entre autre un Signaler(TOP, HORLOGE) (primitive SignalEvent (Num_tache_HORLOGE, evenement_TOP) en RTC) (fig.8). HORLOGE est la tâche de plus forte priorité, gérée par l'exécutif et responsable de la gestion du temps pour l'application. De ce fait elle diffuse à son tour le temps sous la forme de "tics" d'horloge à destination des autres tâches. Tant que la durée d'inhibition de l'ordonnanceur est inférieure à la période des Tops (fig.8), il n'y a pas de risque de perte des tics d'horloge. Dans le cas contraire, il convient de prévoir un mécanisme de compensation permettant l'émission en rafale de plusieurs tics. Ainsi, deux types d'horloge sont implémentables: une classique et une améliorée permettant le rattrapage des signaux manqués. Il est certain que si l'application est correctement conçue, un tel mécanisme n'est pas à mettre en œuvre car concevoir une telle compensation indique implicitement que le théorème de Shannon n'est pas respecté.

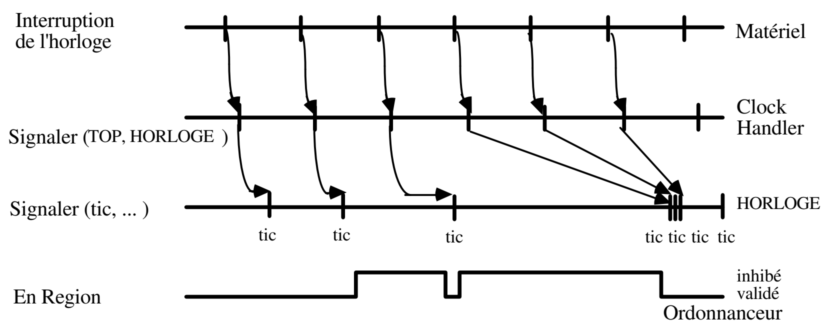

fig.8: conséquences des inhibitions de l'ordonnanceur

IV.5. EVENEMENTS PRIORITAIRES ET INDIVISIBILITE

Dans les systèmes de commande, il doit exister des possibilités d'inhiber certaines actions ou de figer les évolutions. L'arrêt d'urgence en est un exemple. Cet arrêt n'admet pas d'être différé quant à son effet. Ce genre de traitement peut être mis en œuvre au niveau des tâches gérées par l'exécutif en utilisant les mécanismes d'exception. Rappelons qu'une exception est généralement levée lorsqu'un événement prévu mais non désiré se réalise. Du fait de l'asynchronisme et des priorités, il n'est pas toujours possible de garantir un temps de réponse à ces sollicitations.

Ces traitements exceptionnels, ont souvent des effets sur les actions les plus élémentaires appartenant à la partie synchrone de l'application. Le concepteur à donc tout intérêt à prévoir des signaux pour traiter ce genre de situation au niveau de la partie la plus réactive c'est-à-dire, au niveau du programme ESTEREL. Ceci s'exprime facilement en utilisant des instructions gardées (do…watching <occurrences> timeout …end;) du langage ESTEREL. L'automate étant rattaché à la tâche de plus forte priorité, son intervention sur l'occurrence d'un signal d'exception sera immédiate.

IV.6. ANALYSE DES PERFORMANCES

Un code particulièrement efficace est requis pour l'implémentation des systèmes temps-réel. Au travers de notre expérience nous cherchons à montrer que notre démarche est raisonnable du point de vue temps d'exécution. Il est en effet certain que si l'implémentation mixte ne permet pas d'obtenir un gain au niveau des temps d'exécution, elle perd de sa crédibilité.

Les analyses ont été effectuées sur le code produit pour un microprocesseur 8086 à 8Mhz. Nous avons évalué les temps sur les valeurs fournies dans [GSI87] pour l'analyse du programme asynchrone et sur les temps calculés à partir du code assembleur correspondant à l'automate [AnFa90]. Le nombre de cycles machine obtenu nous a semblé être un bon critère d'évaluation de performances. Nous avons comparé l'implémentation d'un programme entièrement asynchrone permettant de lancer deux tâches d'acquisition (de la valeur du pH et de la valeur du niveau d'eau dans la cuve) en parallèle. Le même exemple a été implémenté par le biais de l'approche mixte où les tâches précédentes ont été codées comme deux modules ESTEREL évoluant en parallèle.

Alors que l'approche purement asynchrone accuse un temps d'exécution de **3560+k cycles** machine pour effectuer effectivement un appel aux tâches et réaliser les actions attendues, l'approche mixte utilisant l'automate généré par le compilateur ESTEREL nécessite un temps de **1781+k cycles** machine (k correspondant à du code commun aux deux solutions). Des détails sont donnés dans [AnFa90].

Il convient de remarquer que l'approche mixte permet d'obtenir un gain de temps non négligeable. Cette évaluation justifie d'un point de vue efficacité le choix fait pour notre type d'implémentation. Ces résultats étaient, somme toute, prédictibles. En effet, les différences de performance sont dûes à l'écart existant entre un appel de fonction (partie synchrone) et un changement de contexte (partie asynchrone) pour lequel le surcoût est loin d'être négligeable. La possibilité qui est offerte par le biais de l'automate de réduire le nombre de changement de contexte est un avantage indéniable quant aux temps d'exécution.

V.METHODE DE CONCEPTION

Notre technique de modélisation permet de mieux cerner les caractéristiques d'un système. Elle rend explicite le squelette de synchronisation et met en évidence les fonctionnalités de l'automatisme à développer. Les contraintes de synchronisation, le parallélisme et les coopérations mises en place par le modèle rendent compte du comportement dynamique imposé au système. Notre stratégie de conception de l'application repose sur le codage du noyau de synchronisation au moyen du langage ESTEREL et sur le codage des parties transformationnelles en tâches à part entière.

L'implémentation du squelette de synchronisation va nous permettre de concevoir un ordonnanceur efficace dont le rôle est de cadencer l'évolution des tâches asynchrones de façon déterministe. La technique d'implémentation se décompose en trois étapes. La première traite du codage en "pseudo-ESTEREL" du noyau de synchronisation. La deuxième est la traduction du pseudo-ESTEREL en ESTEREL classique qui sera ensuite compilé et simulé. La troisième correspond au développement des tâches asynchrones et de l'environnement nécessaire à l'interfaçage entre l'automate ESTEREL et le reste du code constituant l'automatisme.

Pour notre exemple, le pseudo-ESTEREL du système de commande est décrit sur la figure 12. Il est à noter que nous utilisons une instruction particulière *on <S> do <stat1> suspending <stat2> end;* dont la sémantique intuitive peut être décrite de la façon suivante: lors de l'exécution de cette instruction, stat2 est exécuté. Si l'occurrence du signal S survient avant l'échéance de stat2 alors stat1 est exécuté et les occurrences de S sont masquées. La fin de stat1 relance l'exécution de stat2 au point de contrôle quitté sur l'occurrence de S et valide les occurrences prochaines de S. L'instruction (on <S> do <stat1> suspending <stat2> end;) termine avec la fin de stat2. Cette instruction n'appartient pas aux versions commercialisées du langage ESTEREL mais a pu être implémentée à l'aide des primitives du langage.

Le programme fig.12 est une traduction du schéma de contrôle de l'application (fig.2) en pseudo-ESTEREL. Les annotations sur la gauche ne font pas partie du programme, ce ne sont que des surcharges textuelles rappelant la structuration de l'application. Les instructions en italiques font références à des traitements dont le temps d'exécution n'est pas négligeable et qui doivent être pris en compte dans l'environnement asynchrone sous la forme de tâches gérées par l'exécutif RTC. La description de ce noyau de synchronisation est conforme à la structuration établie dans le phase de modélisation. Certaines actions de bas niveau sont directement prises en charge au niveau du squelette de synchronisation afin que l'activité de type réflexe qu'elles représentent ne soit pas différée.

```
                    …
                    loop
                      …
                      await LANCEMENT_DU_SYSTEME;
                      do
                         on pH_Hors_Limite do
                           effectuer corrections du pH
                         suspending
                              effectuer initialisations du système
                                 [
                                 on Intervention_Opérateur do
                                   effectuer les actions de l'operateur
                                 suspending
S1
                                      loop
       S1.1
                                           await Bouteille_A_Remplir;
                                           emit Acheminer_La_Bouteille;
                                           await Capteur_On
             S1.1.3-1                      emit Declencher_Le_Remplissage;
                                           await Bouteille_Pleine;
                                           effectuer les actions de fin de remplissage
                                           emit Etiqueter;
                                           await Capteur_Off
                                      end
                                 end ]
                              ||
                                 [
                                  loop
                                      await Valeur_du_pH;
                                      if (?Valeur_du_pH>Valeur_du_pH_de_Reference)
             S1.1.1                   then
                                             effectuer la régulation du pH

                                       end
                                  end ]
S1     S1.1                   ||
                                [
                                 loop

                                    await Valeur_du_Niveau;
             S1.1.2                 if (?Valeur_du_Niveau>Valeur_du_Niveau_de_Reference)
                                    then
                                           effectuer la régulation du Niveau
                                     end
                                end ]

                        end % fin de suspending pH_Hors_Limite
                    watching Arret_du_Superviseur
                    timeout

                           effectuer les traitements de fin d'activité
                    end;
                    …
                 end;
                 …
```

S1, S1.1, S1.1.1, S1.1.2, S1.1.3-1 font référence au schéma de contrôle fig.2

fig.12: pseudo-ESTEREL de l'application

Les instructions en italiques, représentant l'exécution d'actions asynchrones, doivent être décomposées dans la deuxième étape (traduction du pseudo-ESTEREL en ESTEREL pur). Afin de permettre leur exécution effective dans l'environnement géré par le noyau, il est nécessaire d'invoquer les services de signalisation du noyau. La figure suivante indique un exemple type des mécanismes à mettre en jeu pour traduire cette exécution asynchrone à partir de l'automate ESTEREL.

…
effectuer la régulation du pH →
…

```
…
do
    emit REGULER_pH;                % appel à une tâche
    await Fin_Regulation_pH         % attente de la fin de la tâche
watching temps_de_régulation        % traitement d'exception si le
timeout     …                       % temps imparti à la tâche est
            …                       % dépassé
end;
…
```

emit REGULER_pH; →

```
O_REGULER_pH()
{
  SignalEvent(Num_tache_de_regulation, REGULER);
} /* fin de O_REGULER_pH() */
```

SignalEvent(Num_Tache, S) est la primitive RTC permettant d'émettre le signal S à la tâche didentificateur Num_Tache

/* tâche gérée par RTC */

WaitEvents(s,temps_d_attente,buffer_de_reception); est la primitive RTC permettant d' attendre (attente passive) l'occurrence de l'événement S.
ClearEvents(Liste_d_Evenements); est la primitive RTC permettant de réinitialiser les événements contenus dans Liste_d_Evenements.

```
void tache_de_regulation(void)
{
  … /* déclarations */
  … /* initialisations */

  faire_sans_fin {
    WaitEvents(reguler, Attente_Infinie, &Evenement_Recu);
    ClearEvents(reguler);
    faire_tant_que (pH!= pH_de_Reference) {
        /* effectuer l'algorithme de régulation */
    } /* fin faire_tant_que */
    SignalEvent(Num_tache_Automate, Fin_Regulation_pH);
  } /* fin faire_sans_fin */

} /* fin tache_de_regulation() */
```

fig.13: exécution d'actions asynchrones

L'automate ESTEREL est incorporé au sein de l'environnement asynchrone de l'exécutif par l'intermédiaire d'une tâche appelée "Automate". La structure de cette tâche "Automate" (gérée par RTC et effectuant les accès à la fonction "automate()" générée par le compilateur ESTEREL) est caractéristique de l'environnement à développer pour transférer les informations sous la forme de d'événements externes à la procédure synchrone. Comme nous l'avons indiqué dans la partie traitant des problèmes d'intégration, cette tâche doit être de forte priorité. Celle-ci a le comportement de l'automate: elle ne fait rien en dehors d'attendre l'occurrence de signaux externes. Ceux-ci sont codés et lui parviennent sous la forme de simples signalisations. Lorsque l'ordonnanceur détecte l'émission d'un signal à destination de la tâche Automate, il préempte la tâche courante et donne le contrôle à la première en raison de sa priorité. Le décodage de l' (voire des) événement (s) reçu (s) donne lieu à l'appel des fonctions qui leurs sont associées dans l'interface générée par le compilateur. Cette action effectuée, il reste à la tâche Automate à invoquer la fonction automate pour réaliser une transition d'état. La figure suivante donne le schéma fonctionnel de la tâche Automate.

```
void Automate(void)
{
   … /* déclarations */
   … /* initialisations */

   fair_sans_fin {

         WaitEvents(Liste_d_Evenements, Attente_Infinie, &Evenements_recu);
         /* Liste_d_Evenements est la liste d'événements attendus par l'automate */
         EnterRegion()
            if (EventsOccurred(LANCEMENT_DU_SYSTEME)) I_LANCEMENT_DU_SYSTEME();
            …
            if (EventsOccurred(Capteur_On)) I_Capteur_On();
            if (EventsOccurred(Capteur_Off)) I_Capteur_Off();
            …
            if (EventsOccurred(Valeur_du_pH)) I_Valeur_du_pH();
            if (EventsOccurred(Fin_Regulation_ pH)) I_Fin_Regulation_pH();
            …
            ClearEvents(Liste_d_Evenements);
            automate(); /* exécution d'un cycle de la procédure générée par le compilateur ESTEREL */
         LeaveRegion();
    } /* fin faire_sans_fin */
 } /* fin de la tâche Automate() */
```

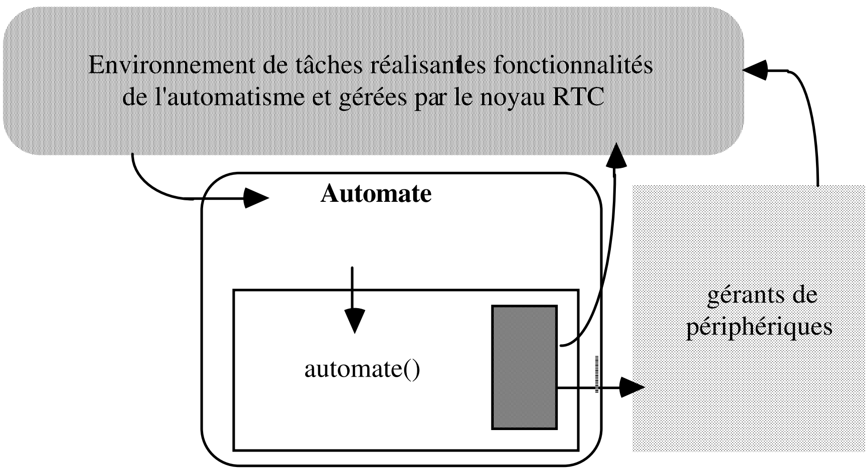

fig.14: schéma de la tâche Automate

Il est à noter que la protection effectuée autour de l'appel à l'automate n'est pas forcément nécessaire dans le cadre de cette application où une seule tâche est responsable de l'appel à l'automate et que sa priorité est la plus élevée. Le fonctionnement choisi pour cette application est un mode auto-synchronisé. Dans le cadre où une certaine cohérence temporelle était à prendre en compte pour l'exécution des actions asynchrones, un mode de fonctionnement synchronisé sur horloge aurait été mieux adapté.

Le code ESTEREL réalisé et éprouvé (par simulation), les tâches asynchrones (conçues de façon traditionnelle) achevées, il ne reste qu'à compiler et faire l'édition des liens pour obtenir le contrôleur de l'application.

VI. CONCLUSION

Les systèmes étudiés présentent à la fois des caractéristiques de systèmes transformationnels et de systèmes réactifs. La partie transformationnelle, généralement conçue et gérée de façon asynchrone, requiert de nombreux mécanismes de coopération. La partie réactive, caractérisée par des échanges fréquents avec l'environnement relève plutôt du domaine d'applications des langages synchrones. Les approches généralement utilisées pour implémenter ces types d'automatisme sont soit purement asynchrones (par langage temps-réel ou par exécutif multitâche temps-réel) soit purement synchrones. Elles se trouvent de ce fait, mal adaptées pour prendre en compte certaines caractéristiques comportementales de ces systèmes. Notre objectif a été de mettre au point une technique d'implémentation utilisant les **outils** conçus et **dédiés pour chaque type de comportement**: l'approche asynchrone pour la partie transformationnelle, l'approche synchrone pour la partie réactive.

A partir d'un exemple utilisé comme support d'argumentation, nous avons présenté des problèmes propres à la conception de systèmes de commande temps-réel. Pour pallier l'inadéquation des approches traditionnelles nous avons proposé une démarche originale. Celle-ci est à la fois synchrone et asynchrone. La stratégie de mise en œuvre adoptée permet des améliorations à différents niveaux de conception.

• La structuration et la lisibilité des programmes sont les atouts apportés par l'utilisation d'outils dédiés d'une part pour le développement du squelette de synchronisation et la prise en compte d'activités réflexes, d'autre part pour le traitement des parties algorithmiques.

• L'efficacité du code généré résulte du code produit par le compilateur du langage synchrone ESTEREL qui permet de s'affranchir d'un bon nombre de changement de contexte (pénalisant dans le cadre d'une approche purement asynchrone) lors de l'exécution.

• L'accroissement du déterminisme d'exécution est l'amélioration la plus significative apportée par notre technique de mise en œuvre. En effet, elle permet de donner au système un caractère prédictible (du fait du déterminisme de la partie réactive traitant le squelette de synchronisation de l'application) permettant de conforter le concepteur sur les comportements attendus de l'automatisme et de donner à l'application un caractère déterministe rendant reproductibles les exécutions.

La décomposition en partie essentiellement transformationnelle et en partie essentiellement réactive peut être étendue à une large classe de systèmes "temps-réel". Les nombreuses méthodes de conception existantes (Yourdon, SADT, DARTS ... cf [Lude86]) n'exploitent pas cette complémentarité intrinsèquement liée à ces types de systèmes. Suite à notre étude, il nous paraît important de lister des points qui devraient être pris en compte dans les méthodes de conception.

1) Distinguer au sein de l'application étudiée, la partie transformationnelle de la partie réactive. Cette analyse permet de séparer deux modes de fonctionnement. La partie transformationnelle nécessite des mécanismes de coopération importants (transferts de messages riches en informations) et n'est généralement pas soumise à des contraintes temporelles critiques. Les communications y sont le plus souvent asynchrones. La partie réactive, pour sa part, est soumise à des contraintes temporelles pouvant être strictes

demandant des réactions "immédiates" à l'occurrence d'événements externes. La communication est alors élémentaire (un simple mot, par exemple) et souvent synchrone.

2) Décomposer la partie essentiellement transformationnelle en éléments fonctionnels. Ceci permet sa structuration en tâches et donne à l'application un caractère modulaire. Sa programmation effective est réalisée par le biais des primitives de synchronisation, de communication et d'exclusion mutuelle des exécutifs temps-réel.

3) Choisir un langage synchrone pour la programmation de la partie réactive (synchronisations fortes de l'automatisme et activités réflexes) de l'application. Cette phase nécessite une analyse préliminaire des interactions de l'automatisme avec son environnement afin de distinguer les éléments "actifs" (associés aux réceptivités de l'automate) des éléments "passifs" (paramètres de l'automate ne déclenchant pas d'évolution). Cette étude permet de d'identifier les "signaux" et les "capteurs" (au sens ESTEREL du terme) de l'application. Les actions à effectuer lors du franchissement d'une transition doivent être brèves (action sur des ports d'Entrée/Sortie, par exemple) et appelables par des signaux de sortie appropriés. Les "actions qui durent", doivent, pour leur part, impérativement faire appel à des tâches différées afin de libérer l'automate. Celui-ci, peut éventuellement se mettre en attente de l'occurrence d'un signal d'entrée indiquant l'échéance de ce type d'action.
Il est intéressant de profiter à ce niveau des possibilités de composition offertes par les langages synchrones. Leurs compilateurs sont capables d'engendrer des automates complexes qu'il serait difficile d'obtenir manuellement par composition d'automates.

4) Concevoir un gérant d'interruptions pour le contrôle du temps. Il doit être implémenté avec précaution afin de ne pas perdre de Tops. Il ne doit pas modifier directement les tâches de l'application.
Mettre en place une tâche HORLOGE (réalisant l'interface avec le gérant de l'horloge temps-réel du système) gérée par le noyau et de très forte priorité afin de réaliser la diffusion du temps vers l'automate.

5) Choisir un mode de fonctionnement auto-synchronisé ou synchronisé sur une horloge de base lors de la conception de l'interface entre la partie modulaire développée avec l'exécutif et l'automate. Ces modes de fonctionnement dépendent du type de l'application. Si une certaine cohérence temporelle est nécessaire alors le deuxième mode de fonctionnement sera plus adapté.

Il est certain que la conception de tels systèmes passe par des méthodes de spécification, de modélisation et d'analyse adaptées. Nous avons par ailleurs, développé un modèle répondant à ces besoins et capable d'aborder des problèmes à la fois synchrones et asynchrones. Ce modèle, les HSENS (Hierarchical Synchronized Elementary Nets Systems) [Fan90], s'inscrit dans le cadre de la théorie générale des réseaux (Petri). Il représente une abréviation des SENS (Synchronized Elementary Nets Systems) [André90] qui sont une extension des réseaux élémentaires [Thiag86] [Roz86]. L'automatisation de la mise en œuvre partant de la description du système de commande au moyen des HSENS correspond aux travaux en cours.

VII. BIBLIOGRAPHIE

[André90] **C. André.**
Synchronized Elementary Net Systems
G. Rozenberg ed., Lecture Notes in Computer Science n°424 , Springer Verlag -
(1990) pp. 51-76.

[AnFa90] **C. André, L. Fancelli**
A mixed implementation of a RealTime system
EuroMicro, Amsterdam, 1990

[BBGG85] **A. Benveniste, P. Bournai, T. Gautier, P. Le Guernic.**
Signal: A data-flow oriented language for signal processing.
Rapport de recherche INRIA n° 378 - (Mars 85)

[BCCG85] **G. Berry, L. Cosserat, P. Couronné, G. Gonthier.**
Systèmes réactifs et programmation synchrone.
Actes du premier colloque C_cube - Angoulême - (Sept 85)

[BeCoGo87] **G. Berry, P. Couronné, G. Gonthier.**
Synchronous programming of reactive systems: An introduction to ESTEREL.
Rapport de recherche INRIA n° 647 - (Mars 87)

[BeGon] **G. Berry, G. Gonthier.**
The ESTEREL synchronous programming language: Design, Semantics, Implementation.
Technical report. Ecole des Mines, Sophia-Antipolis

[BNI84] **Bureau d'orientation de la Normalisation en Informatique.**
SCEPTRE: proposition de noyau normalisé pour les exécutifs temps réel.
Techniques et sciences informatique - vol 3 - (1984)

[Fan90] **L. Fancelli.**
Méthode de conception mixte asynchrone/synchrone pour la réalisation de systèmes de commande de processus discontinus.
Thèse de doctorat de 3ème cycle - Université de Nice - à paraître (Juill. 90)

[GRTR88] **Groupe de Réflexion Temps Réel.**
Le temps-réel.
Département SPI du CNRS. - (Juin 88)

[GSI87] **Gsi-Tecsi.**
RTC: XEC86 Executive user's guide - (Mai 87)

[HaPnu85] **D. Harel, A. Pnueli.**
On the development of reactive systems.
CS85 - 02, The Weizmann Institute of Science, Rehovot, Israel - (1985)

[HCPP87] **N. Halbwachs, P. Caspi, D. Pilaud, J. Plaice.**
Le langage LUSTRE et sa sémantique opérationnelle.
Actes du deuxième colloque C-cube - (Mai 87)

[HPOG88] **N. Halbwachs, D. Pilaud, F. Ouabdesselam, A-C. Glory.**
Specifying, programming and verifying real-time systems using a synchronous declarative language.
Projet SPECTRE. Groupe spécification et Analyse des systèmes - (Juil 88)

[Lude86] **J. Ludewig.**
Practical methods and tools for specification.
Embedded systems. Springer-Verlag - (Mars 86)

[Perrin87] **G.R. Perrin.**
Programmation parallèle: Point de vue sur les langages et les méthodes.
T.S.I vol6, n°2 - (1987)

[Rayn81] **M. Raynal.**
Contribution à l'étude de la coopération entre processus dans les langages et les systèmes informatiques.
Thèse d'état, Université de Rennes - (1981)

[Roz86] **G. Rozenberg.**
Behaviour of Elementary Net Systems
Advanced Course on Petri Nets, Bad Honnef - (Sept. 1986)

[Schip86] **A. Schiper.**
Programmation concurrente.
Presse polytechnique romande - (1986)

[Thiag86] **P.S. Thiagarajan.**
Elementary Net Systems
Advanced Course on Petri Nets, Part I, pp 60-94 - (1986)

[Young87] **S.J. Young.**
Realtime languages: Design and development.
Ellis Horwood - (1987)

[Youdon85] **S.J. Mellor, P.T. Ward.**
Structured Development for realtime systems (tome I,II,III)
Yourdon press computing series - (1985)

[ZRS87] **W. Zhao, K. Ramamritham, J.A. Stankovic**
Scheduling tasks with resource requirements in hard real time systems.
IEEE transactions on software engineering - (Mai 1987)