# Validation Temporelle d'Applications Temps Réel Distribuées à Contraintes Strictes

Gaëlle Largeteau-Skapin, Dominique Geniet

# Validation Temporelle d'Applications Temps Réel Distribuées à Contraintes Strictes.

Gaëlle LARGETEAU
largeteau@ensma.fr

Dominique GENIET
dgeniet@ensma.fr

LISI / ENSMA
Téléport 2 - 1 avenue Clément Ader
BP 40109
86961 Futuroscope Chasseneuil cedex
Tel : (+33/0) 5 49 49 80 63
Fax : (+33/0) 5 49 49 80 64

**Résumé** : *Ce papier décrit une technique de validation temporelle d'applications temps réel distribuées à contraintes strictes, par le biais de langages rationnels. Dans cette étude, nous tenons compte des impératifs matériels qu'impose la machine cible. En effet, certaines caractéristiques temporelles des tâches sont dépendantes du matériel, ce qui influe sur le langage modèle. Nous faisons l'hypothèse qu'il n'y a pas de migration des tâches sur les différents sites du système et nous n'étudions pas le problème du placement. Nous appliquons enfin cette méthode à un protocole temps réel : CAN.*

**Abstract** *: This paper deals with temporal validation of distributed hard real time systems. We consider here the target machine physical properties. Indeed, several temporal characteristics of some tasks depend on hardware properties, and then influence the model. We suppose that there is no task migration, and we did not study the tasks allocation problem. We valid this method to the protocol CAN.*

**Mots clés** : validation , temps réel, modèle synchrone, langages rationnels, systèmes distribué, réseau.

**Key words** : validation, real time, synchronous model, rationnal language, distributed systems, network

## Introduction

Un système temps réel est d'une part **réactif** et d'autre part **concurrent** (toutes les opérations relatives au pilotage du procédé doivent progresser simultanément). Il est donc naturellement composé d'un ensemble de **tâches** élémentaires, chacune d'elles implémentant une réaction ( ou une partie de réaction) que le système devra fournir. Cet ensemble de tâches est divisé en deux grandes classes de tâches : les tâches **périodiques** ( ce sont les tâches de lecture et de traitement des informations fournies par les capteurs) ; les tâches **apériodiques** et **sporadiques** ( liées aux signaux d'alarmes et aux actions d'un éventuel opérateur).

De façon générale, un système temps réel se distingue d'un système traditionnel par sa capacité à garantir que les tâches respectent certaines échéances. La validité d'un tel système tient à la justesse de ses résultats et à son respect des **contraintes temporelles**. Il existe deux grandes classes de systèmes temps réel : Les systèmes temps réel dur et ceux temps réel mou. Lorsque le non respect des échéances a des conséquences irrémédiables, le système est dit dur ou à contraintes strictes. Nous allons ici n'étudier que les systèmes temps réel dur composés de tâches périodiques uniquement.

Valider temporellement un système temps réel consiste à prouver a priori que quel que soit le flot d'événement entrant, le système sera toujours capable de réagir conformément à ses contraintes temporelles. La validation temporelle est donc un processus de décision qui concerne des séquences d'ordonnancement des tâches. L'ordonnancement s'aborde usuellement à travers deux approches : D'une part, l'approche « **en ligne** » consiste à piloter l'application en choisissant, à chaque commutation de contexte, la tâche à élire. Or, l'ordonnancement de systèmes de tâches en présence de ressources critiques est NP-complet [BRH90]. Cet état de fait rend tout algorithme d'ordonnancement *en ligne* non optimal (dans le sens où une validation s'appuyant sur cet algorithme peut diagnostiquer une configuration ordonnançable comme non ordinnançable). Le théorème de cyclicité des ordonnancements dans le cas mono-processeur [Gro99] montre qu'en l'absence de condition analytique nécessaires et suffisantes d'ordonnançabilité (dès que synchronisation ou partage de ressources apparaissent, en pratique), la validation d'algorithmes d'ordonnancement en-ligne est du même niveau de complexité. La validation d'ordonnancement *en ligne* de systèmes de tâches interdépendantes est donc de complexité exponentielle, et l'on sait *a priori* que la décision obtenue est peut être erronée!

Pour répondre à ce problème, l'approche « **hors ligne** » a été abordée : elle consiste à rechercher ( par le biais de simulations) l'existence d'au moins une

séquence d'ordonnancement qui satisfait les contraintes temporelles. Elle s'appuie sur des modèles temporels de l'application à valider.

La rapide évolution technologique de ces dernières années, principalement dans le domaine des réseaux de communication, a induit une utilisation de plus en plus fréquente de **systèmes distribués** comme support pour les applications temps réel à contraintes strictes. Ces systèmes distribués sont basés sur des protocoles temps réel (FIP, CAN, CSMA/DCR) qui offrent des possibilités de prise en compte des contraintes temporelles dans l'envoi des messages. Très peu d'études existent à ce jour sur ce thème. Elles s'intègrent principalement dans l'approche en ligne [S98] : les messages sont vus comme des tâches temps réel particulières ordonnancées de manière non-préemptive sur un processeur spécifique : le réseau.

L'ordonnancement de systèmes distribués étant NP-difficile, l'approche en ligne reste non optimale.

Dans ce papier, nous nous basons sur l'approche hors ligne définie dans [G00][GL01] pour proposer une technique de validation temporelle de tels systèmes. Le principe de notre travail est l'intégration des protocoles de communication au modèle et, par voie de conséquences, l'adaptation du modèle aux cibles comportant des processeurs fonctionnant à des cadences différentes.

Nous présentons, dans un premier temps, le modèle dans le cadre des systèmes temps réel centralisé avec tâches à durée fixe. Nous décrivons ensuite une méthode de modélisation des systèmes distribués tenant compte des différences de vitesse entre le réseau et les stations. Nous faisons l'hypothèse qu'il n'y a pas de migration des tâches sur les différents sites du système et nous n'étudions pas le problème du placement.

Nous validons enfin cette méthode par l'exemple d'un système distribué fonctionnant sur un réseau CAN (Controller Area Network).

# 1 Validation de systèmes centralisés

## 1.1 Modélisation temporelle de tâches

### 1.1.1 Spécification temporelle de tâches

Une tâche temps réel périodique est définie par un ensemble de comportements, i.e. de séquences d'exécution, qui caractérise la contrainte temporelle de durée d'exécution $C_i$ ( durée d'exécution dans le pire cas ). $C_i$ est un paramètre dépendant des caractéristiques physiques de la cible, ici, $C_i$ est calculé en nombre d'instructions élémentaires. Nous supposons que toutes les instructions élémentaires durent une unité de temps. Les autres contraintes temporelles sont :

$r_i$ : date de première occurrence de l'événement qui déclenche la tâche i.

$D_i$ : Délai critique relatif.

$T_i$ : intervalle de temps entre deux occurrences successives de la tâche i.

Les valeurs des paramètres $r_i$, $D_i$, $T_i$ sont données en *secondes*. Durant l'intervalle $[r_i, r_i+T_i[$, une tâche doit être active $C_i$ unités de temps sur $[r_i, r_i+D_i[$ et 0 unité de temps sur $[r_i+D_i, r_i+T_i[$.

### 1.1.2 Modèle temporel

Soit $\tau$ une tâche temps réel soumise aux contraintes temporelles r, D et T. Le code $\xi$ de $\tau$ (ou **séquence d'exécution**) est un mot sur l'ensemble P des instructions élémentaires. La durée d'exécution C de $\tau$ est calculée à partir de ce code. Lors de son exécution, $\tau$ peut être active ou suspendue suivant qu'elle possède ou non le processeur. Pour chaque unité de temps u, l'observation de l'état de $\tau$ permet de construire une séquence des périodes d'activation / désactivation de $\tau$. L'ensemble des séquences respectant la spécification temporelle constitue le modèle temporel de $\tau$. Dans la suite, notre objectif est de construire l'ensemble de ces séquences, i.e. de construire le langage rationnel $L_u(\tau)$ collectant l'ensemble des comportements temporels valides de $\tau$.

#### 1.1.2.1 Principes

Dans notre approche, le temps est implicite : chaque tâche effectue une action par unité de temps. Notons **a** une unité de temps pendant laquelle $\tau$ s'exécute, $\bullet$ une unité de temps durant laquelle $\tau$ est suspendue et $\Sigma$ l'alphabet $\{a,\bullet\}$.

Le modèle temporel associé à $\xi$ est le mot $\phi(\xi)$, où $\phi$ est le morphisme de concaténation $P \rightarrow \{a\}$. La longueur de $\phi(\xi)$ donne d'une part le nombre d'instructions exécutées par $\tau$, et d'autre part sa durée (toutes les instructions élémentaires sont supposées durer une unité de temps). Le modèle temporel $L_u(\tau)$ de $\tau$ est obtenu en explicitant les périodes d'inactivité du système (à l'aide du symbole $\bullet$). Notons **III** l'opération de *Shuffle*[1], la forme générique de ce modèle est donnée dans [G00] par[2] $L_u(\tau)=\text{Centre}(\bullet^r((\bullet^{D-C}$ **III** $\phi(\xi))\bullet^{T-D})^*)$. La composante $\bullet^r$ correspond à l'inactivité de la tâche avant sa première activation : la date de première activation r est donc supposée être exprimée en *unités de temps*, le mot $\bullet^r$ ayant pour objet de modéliser l'état de $\tau$ pendant tout l'intervalle de temps $[0,r[$. La composante $(\bullet^{D-C}$ **III** $\phi(\xi))\bullet^{T-D}$ modélise l'ensemble des séquences d'allocation processeur compatibles avec les contraintes temporelles de $\tau$. Tous les mots de cet

[1] Le *Shuffle* noté **III**, est défini par $\forall a \in \Sigma$, a **III** $\varepsilon$ = a et $\forall(a,b,w,w') \in \Sigma^2 \times (\Sigma^*)^2$, aw **III** bw'= a (w **III** bw') $\cup$ b (aw **III** w').

[2] Le centre de L est l'ensemble des préfixes de L indéfiniment prolongeables dans L, algébriquement, Centre ( L*) = L*.FacteursGauches(L).

ensemble partagent la même longueur : T. Tout mot de $L_{\mathbf{u}}(\tau)$ est appelé comportement temporel valide de $\tau$.

Dans les études précédentes, r, D et T sont toujours exprimés en nombre d'unités de temps. Lorsque l'on considère plusieurs sites, cette simplification n'est plus utilisable. Suivant le site sur lequel $\tau$ s'exécute, la durée en seconde de C varie car les différents sites possèdent potentiellement des vitesses de calcul différentes.

#### 1.1.2.2 Intégration des caractéristiques physiques de la machine cible

$\tau$ s'exécute sur un processeur qui possède des caractéristiques temporelles particulières : nous appelons **unité de temps** l'intervalle de temps qui sépare deux tics d'horloge, et **cadence** l'inverse de sa durée. Dans le cas multiprocesseur, nous supposons que tous les processeurs d'un même site sont asservis à la même horloge : ils fonctionnent de manière synchrone. La durée d'un comportement donné $\omega$ de $\tau$ vaut donc $|\omega|\,\mathbf{u_S}$ sur le site S, et $|\omega|\,\mathbf{u_T}$ sur le site T. La modélisation des caractéristiques temporelles héritées de $\tau$ (échéance, période, etc.) ne se traduit donc plus simplement dans le langage en terme direct de nombre d'occurrences de $\bullet$, mais en terme de nombre d'occurrences de $\bullet$ nécessaire et suffisant pour modéliser le temps d'inactivité correspondant sur le processeur cible.

Notation : Appelons $L_{\mathbf{u}}(\tau)$ le langage qui contient l'ensemble des comportements de $\tau$ temporellement valides sur un processeur dont l'unité de temps est **u.**

Considérons, par exemple, une tâche $\tau$, de caractéristiques (r,D,T,C)=(3ms,8ms,10ms,3ut).

Pour **u** = 1ms nous avons $L_{\mathbf{1ms}}(\tau)=\text{Centre}(\bullet^3((\bullet^5\mathbf{III}a^3)\bullet^2)^*)$, et pour **u**=250µs : $L_{\mathbf{250\mu s}}(\tau)=\text{Centre}(\bullet^{12}((\bullet^{\mathbf{29}}\mathbf{III}a^{\mathbf{3}})\bullet^8)^*)$. La proportion entre les **a** et les $\bullet$ varie avec la cadence du processeur cible. Nous montrons dans la suite qu'une tâche $\tau$, définie par des caractéristiques temporelles (r,D,T,C), et destinée à s'exécuter sur un processeur de cadence $\mathbf{c}\in\mathbb{Q}^{+*}$ (on note **u**=1/**c**), peut toujours être associée à un langage rationnel $L_{\mathbf{u}}(\tau)$.

Considérons $\omega \in (\bullet^{D-C}\ \mathbf{III}\ \phi(\xi))\ \bullet^{T-D}$. Le nombre $|\omega|_\bullet$ donne, comptabilisée en unité de temps, la durée $Y_i$ (sur une période de $\tau$) pendant laquelle $\tau$ n'a pas de processeur alloué, et $|\omega|-|\omega|_\bullet$ la durée $Y_a$ pendant laquelle elle dispose d'un processeur. Les spécifications temporelles r, D, T de $\tau$ ne dépendent pas des performances du processeur sur lequel $\tau$ s'exécute, mais de caractéristiques physiques du procédé piloté par l'application (éventuellement via des dépendances en chaînes, exprimées sous la forme de contraintes de bout en bout). Du point de

vue du modèle, ce sont donc des constantes. Par contre, C, constante en *nombre d'instructions élémentaires*, varie en temps, en fonction des performances du processeur. La valeur de C est donc fixe, mesurée en *unités de temps*, alors que les valeurs de r, D et T sont fixes, mesurées en *secondes*. Pour intégrer les caractéristiques du processeur au modèle, les valeurs mesurées en secondes doivent donc être exprimées en *unité de temps*. Soit u l'unité de temps[3] du processeur. x secondes correspondent à x/u unités de temps. Les valeurs r, C et T mesurées en *secondes* valent respectivement, en *unité de temps,* r/u, D/u et T/u.

En général les contraintes temporelles r, D et T sont de l'ordre de $10^{-1}$ s et l'unité de temps u de l'ordre de $10^{-5}$s. Sous ces hypothèses, nous supposons, à une approximation négligeable près, que r/u, D/u et T/u sont toujours entiers. Le langage $L_{\mathbf{u}}(\tau)$=Centre($\bullet^{r/u}((\bullet^{(D/u\ -C)}\ \mathbf{III}\ \phi(\xi))\ \bullet^{(T-D)/u})^{*}$) modélise les comportements de la tâche en tenant compte des performances du processeur sur lequel elle s'exécute.

Nous notons $^{\circ}\tau$ l'ensemble $\{L_u(\tau), u\in \mathbb{Q}^{*+}\}$.

### 1.2 Modèle temporel d'une application temps réel

Nous présentons dans [GL01] une technique, basée sur le modèle Arnold-Nivat [A94], pour exprimer l'ensemble des s''quences d'ordonnancement valides d'un système de tâches. L'idée consiste d'une part à associer à chaque ressource critique $R_j$ (processeur, ressource, message, etc.) une tâche virtuelle $V_{R_j}$ (modélisée par un langage rationnel $L(V_{R_j})$), puis à associer au système $T=(\tau_i)_{i\in[1,n]}$ le produit de Hadamard des $L(\tau_i)$ et des $L(V_{R_j})$: c'est le langage accepté par le produit des automates d'acceptation des $L(\tau_i)$, donc écrit sur l'alphabet $\Pi_{i\in[1,n]}(\Sigma_i)\times \Pi_j(\Sigma_j)$. Soit $L_T$ ce langage. Appelons S l'ensemble des vecteurs de $\Pi_{i\in[1,n]}(\Sigma_i)$ décrivant des configurations valides (respect de l'exclusion mutuelle sur un processeur, sur une ressource, etc.). Nous montrons dans [GL01] que le langage[4] $Proj_{1..n}(Centre(L_T \cap S^{*}))$ collecte l'ensemble des séquences d'ordonnancement valides, du point de vue de la gestion des ressources et du point de vue du respect des contraintes temporelles. C'est ce modèle que nous utilisons ici.

[3] l'unité de temps est la durée d'une instruction non interruptible (niveau assembleur). Notre hypothèse est donc ici que les différentes instructions utilisées partagent toute la même durée.

[4] On projette sur les n premières composantes, de façon à ne conserver a posteriori que l'état des $\tau_i$: les composantes supprimées correspondent aux tâches virtuelles qui tracent l'état des ressources, et qui n'ont plus d'utilité par la suite.

### 1.3 La validation temporelle [G00]

La validation temporelle d'un système de tâches $(\tau_i)_{i\in I}$ consiste à décider si la configuration $(\tau_i)_{i\in I}$ est ordonnançable dans le respect de ses échéances. Cette décision est obtenue en évaluant le prédicat $(L((\tau_i)_{i\in I})= \varnothing)$ grâce à la construction de l'automate associé au langage $L((\tau_i)_{i\in I})$. Si il existe un comportement valide $(L((\tau_i)_{i\in I})= \neq \varnothing)$ alors la configuration est ordonnançable, sinon, elle ne l'est pas. Cette stratégie permet de décider de l'ordonnançabilité d'un système de tâches.

La construction de l'automate du langage $L((\tau_i)_{i\in I})$ s'effectue via des produits filtrés[5] d'automates. La complexité au pire de l'évaluation du prédicat est en $O(ppcm_{i\in[1,n]}(T_i))$. Des études, exposées dans [GL01], ont été menées sur la complexité spatiale de ce calcul. Ces études montrent que les cas les plus défavorables interviennent lorsque le système est peu contraint, en effet, il y a généralement de nombreuses séquences d'ordonnancements possible donc un grand nombre d'états dans l'automate résultat.

## 2 Validation de systèmes distribués

La validation de systèmes répartis impose d'intégrer au modèle l'ensemble des aspects *communication*, et donc, notamment, les transferts de messages soumis à des échéances strictes via un réseau.

Dans notre approche, chaque site dispose d'une horloge locale, à laquelle est assujetti l'ensemble des processeurs du site. L'intervalle de temps qui, sur chaque site, sépare deux tics d'horloge, est donc le même pour l'ensemble des processeurs. Le modèle exprime ce parallélisme synchrone à travers l'opération de produit de Hadamard $\Omega$.

Un système réparti est constitué d'un ensemble de sites, fonctionnant a priori à des vitesses différentes (i.e. avec des unités de temps différentes), et communiquant via un réseau. L'utilisation de notre modèle pour la validation d'applications réparties impose donc de l'adapter au cas de tels ensembles de composants.

Dans cette étude, on supposera, en raison de différences d'échelle entre la vitesse des processeurs et les paramètres temporels que nous manipulons, le démarrage simultané des horloges de chacun des sites. Dans un premier temps il est nécessaire de tenir compte de la différence de vitesse entre les différents sites. En effet, le modèle actuel, notamment le produit de Hadamard, ne permet pas la modélisation de systèmes multi-cadencés. Dans un second temps, nous étudions

[5] pour le partage de processeurs, de ressources et la communication.

l'intégration de la modélisation des protocoles réseaux ainsi que la validation d'applications temps réel distribuées.

## 2.1 Prise en compte de la multi-cadence des sites

### 2.1.1 Les homothétiques de $L_u$

Dans le processus de construction de °τ, on prend en compte, dans le modèle, les différences de vitesses des CPU. Nous obtenons donc une classe de langages dans les mots desquels la proportion entre le nombre de a et le nombre de • est fonction de la cadence. Considérons $L_u \in$ °τ. Nous appelons **granularité** l'unité temporelle utilisée pour mesurer la durée d'une lettre dans les mots de $L_u$.

Exemple : Soit L={a} un langage que nous associons à l'unité de temps *1ms* : la durée associée à la lettre **a** est 1ms. Considérons par ailleurs le langage $M=\{a^n\}$ associé à la durée $^1/_n$ ms. L et M sont sémantiquement équivalents car ils contiennent tous les deux le même comportement, dont la sémantique est « τ est dans l'état *a* pendant 1ms » : $|a|\times 1=|a^n|\times {}^1/_n$ . Remarquons que $N=\{a^n\}$ associé à une durée de $^1/_{(n-1)}$ n'est pas équivalent à L : $|a|\times 1\neq|a^n|\times {}^1/_{(n-1)}$.

Terminologie : M est l'observation de L avec la **granularité $^1/_n$**.

Notre objectif est de construire l'ensemble $\overline{L_u(\tau)}$ des langages décrivant les comportements définis par $L_u$ dans des granularités différentes, c'est à dire l'ensemble des langages ${}_gL_u(\tau)$ associés à la tâche τ s'exécutant sur un site d'unité de temps **u** observé avec une granularité **g**.

Remarque : ${}_uL_u(\tau) = L_u(\tau)$.

Pour construire $\overline{L_u(\tau)}$, nous utilisons l'isomorphisme ψ :

$\psi_{u,g} : Pc \cup \{ a, \bullet\} \rightarrow (Pc \cup \{ a, \bullet\})^p$ tel que p = u/g

$\forall x \notin Pc, \psi(x)= x^p$

$\forall x \in \{P,S\}, \psi(x)= a^{p-1}.x$

$\forall x \in \{V,R\}, \psi(x)= x.a^{p-1}$ (voir Figure 1)

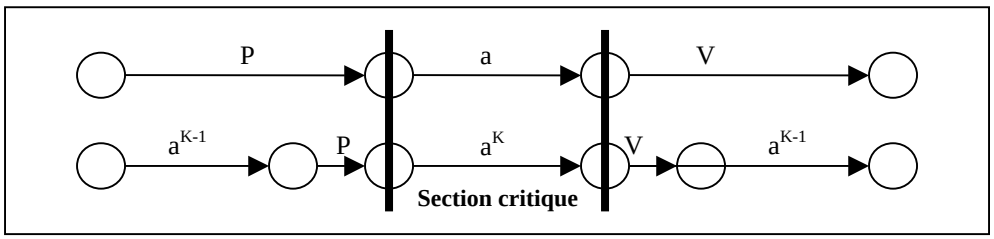


**Figure 1 : cas particulier des instructions critiques**

Notons $L_{u,\psi} = \{{}_gL_u(\tau), {}_gL_u(\tau) \subset \Sigma^* / \exists\, g \in \mathbb{Q}^{*+},\ u \in g\mathbb{N}^*, {}_gL_u(\tau) = \psi_{u,g}(L_u(\tau))\}$.

**Proposition 1** $\overline{L_u(\tau)}$ est exactement $L_{u,\psi}$.

Preuve

$L \in L_{u,\psi} \Rightarrow \exists\, g \in \mathbb{Q}^{*+}\ L_u(\tau) = \psi_{g,u}(L)$ on peut donc associer L à $\tau$.

Soit x une granularité et ${}_xL$ le langage associé à $\tau$. Alors, ${}_xL = \psi_{u,x}(L_u(\tau))$ et donc ${}_xL \in L_{u,\psi}$.

**Remarque** $\psi$ est compatible avec le produit de Hadamard : si L et L' sont deux langages possédant la même unité de temps u alors $L \Omega L'$ est sémantiquement équivalent à $\psi_{u,x}(L \Omega L')$, pour tout x.

### 2.1.2 Langages homothétiques compatibles

Soient ${}_{g_1}L_{u_1}$ et ${}_{g_2}L_{u_2}$. Remarquons que ${}_{g_1}L_{u_1} \Omega\, {}_{g_2}L_{u_2}$ possède une sémantique temporelle (on peut lui associer une unité de temps) si et seulement si $g_1 = g_2$. Or, lors de la modélisation d'une application distribuée, le fonctionnement simultané des sites est représenté par l'application du produit de Hadamard $\Omega$ sur les différents langages associés. Notre objectif est donc maintenant de déterminer une granularité g qui, appliquée à tous les sites, donne une sémantique temporelle à ce produit.

Soit $D(x) = \{z \in \mathbb{Q}^{*+} / x \in z\mathbb{N}\}$ l'ensemble des diviseurs de x

Soient x et $y \in \mathbb{Q}^{*+}$, on pose : $x \wedge y = \sup(D(x \wedge y)) = \sup(D(x) \cap D(y))$

Soient x et $y \in \mathbb{Q}^{*+}$, $\exists\, !\, z \in \mathbb{Q}^{*+} / x \wedge y = z$

On a $D(x) \subset ]-\infty, x]$. Ainsi, on obtient : $D(x) \cap D(y) \subset ]-\infty, \inf(x,y)]$. Or $D(x) \cap D(y)$ est un fermé de $\mathbb{R}$, il atteint donc sa borne supérieure.

En conséquence, $\exists\, !\, z \in \mathbb{Q}^{*+} / z = \sup(D(x) \cap D(y))$

Remarquons que $\wedge_{/\mathbb{Z}}$ = PGCD. $\wedge$ généralise donc aux rationnels la notion de PGCD dans $\mathbb{Z}$.

Soient $L_{u_1}(\tau_1)$ et $L_{u_2}(\tau_2)$, et $g = u_1 \wedge u_2$.

On calcule ${}_gL_{u_1}(\tau_1) = \psi_{u_1,g}(L_{u_1}(\tau_1))$ et ${}_gL_{u_2}(\tau_2) = \psi_{u_2,g}(L_{u_2}(\tau_2))$.

${}_gL_{u_1}(\tau_1) \in L_{\overline{u_1(\tau_1)}}$ ; ${}_gL_{u_2}(\tau_2) \in L_{u_2}(\overline{\tau_2})$, ces deux langages représentent l'observation des deux tâches $\tau_1$ et $\tau_2$ à la même granularité g.

Nous avons donc :

**Théorème 1** Soient $L_{u_1}(\tau_1)$ et $L_{u_2}(\tau_2)$.

Alors, $\exists\, g \in \mathbb{Q}^{*+}$, ${}_g L_{u_1}(\tau_1) \subset \Sigma^*$, ${}_g L_{u_2}(\tau_2) \subset \Sigma^*$ tels que

${}_g L_{u_1}(\tau_1) \in \overline{L_{u_1}(\tau_1)}$ ; ${}_g L_{u_2}(\tau_2) \in \overline{L_{u_2}(\tau_2)}$

Ce résultat permet d'obtenir un ensemble de langages qui partagent la même granularité. Les langages obtenus sont maximaux : il n'existe pas de granularité g'>g telle que l'on puisse trouver un ensemble de langages compatibles pour le produit de Hadamard. De plus, le morphisme $\psi$ fournit naturellement un algorithme pour la construction de cet ensemble de langages. La composition des systèmes placés sur les différents sites peut donc s'exprimer par le produit de Hadamard.

## 2.2 Intégration de protocoles de communication

Dans le paragraphe précédent, nous avons établi la validité de notre modèle pour le cas réparti. On applique donc le produit de Hadamard aux langages correspondant à chacun des sites. Nous faisons ici l'hypothèse de non migration des tâches. Le système de tâches $(\tau_i)_{i\in I}$ est réparti suivant les sites $(S_j)_{j\in J}$. Soit $(\tau_i)_{i\in I_n}$ l'ensemble des tâches qui s'exécutent sur le site $S_n$, on note $L_{u_n}(S_n)$ le langage $L_{u_n}((\tau_i)_{i\in I_n})$.

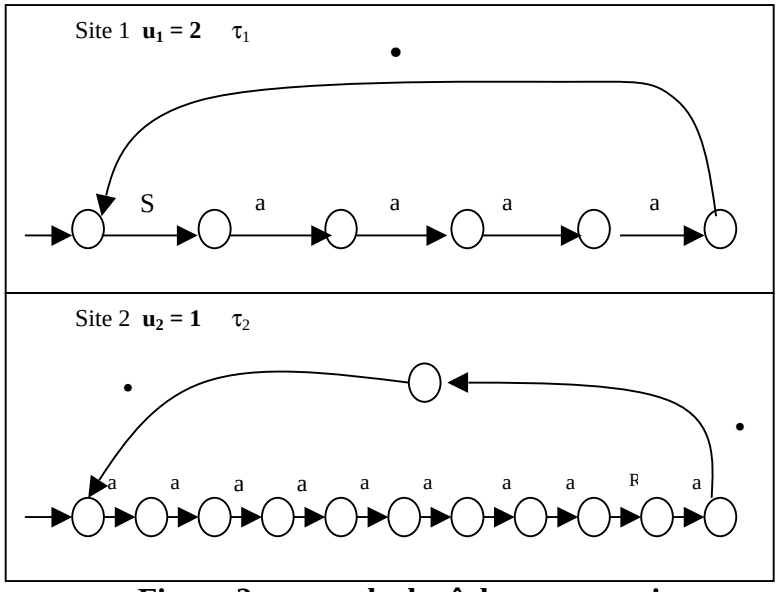


**Figure 2 : exemple de tâches communicantes**

Soit $(L_{u_i}(S_i))_{i\in J}$ l'ensemble des langages associés aux sites (voir exemple en figure 2). On applique le théorème 1 (section 2.1.2):

Soit $G=\wedge_{i\in J}(u_i)$, et $({}_GL(S_i))_{i\in J}$ l'ensemble des langages tels que : $\forall\ i\in J$ $L(S_i)= \psi_{u_i,G}(L_{u_i}(S_i))$.

Les langages $({}_GL(S_i))_{i\in J}$ ont tous la même granularité G, on peut donc calculer ${}_GL = (\Omega({}_GL(S_i))_{i\in J})$. ${}_GL$ exprime le fonctionnement du système $(\tau_i)_{i\in I}$ sur les sites$(S_j)_{j\in J}$.(voir exemple en figure 3 ).

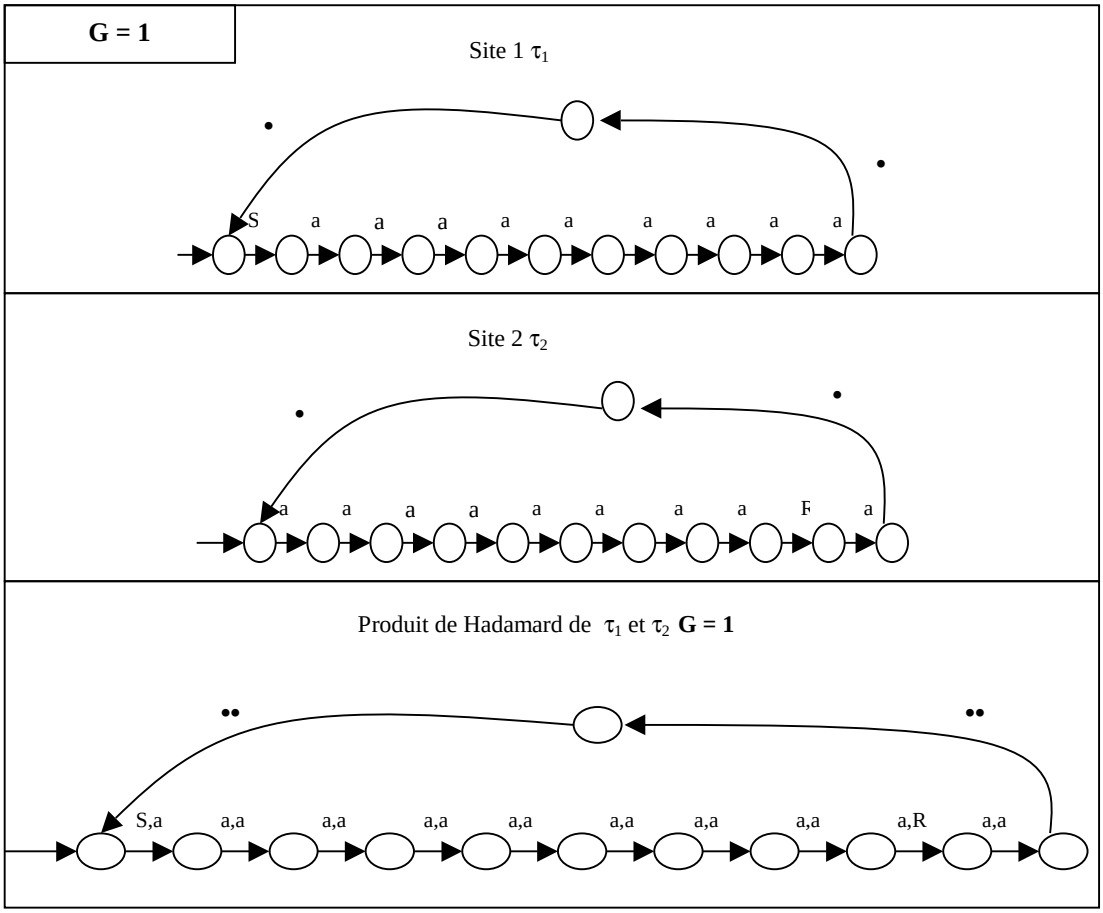


**Figure 3 : exemple de modélisation d'ATRD**

Pour valider temporellement ce système, nous devons maintenant intégrer les protocoles de communication au modèle.

### 2.2.1 Modélisation des comportements temporels du réseau

Certaines tâches de l'application communiquent via le réseau. Notre problème est maintenant de garantir que le transfert de message se fera toujours dans les temps. Pour cela, il est d'abord nécessaire d'avoir un modèle pour les comportements du réseau (voir structure d'un réseau présenté en Figure 4).

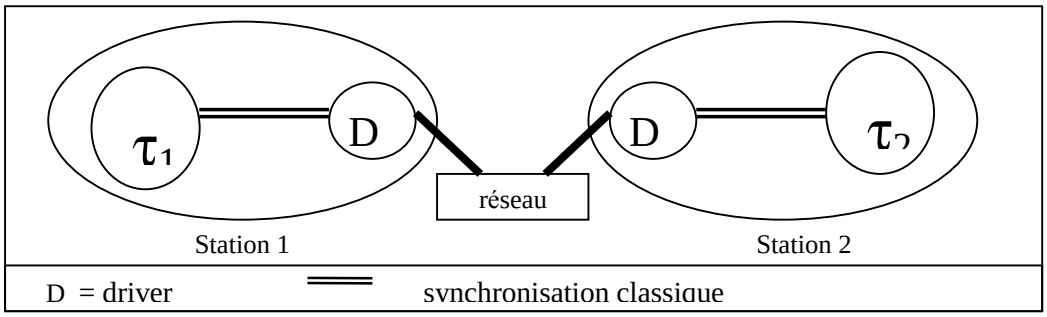


**Figure 4: structure de réseau**

Nous commençons par modéliser les comportements d'un driver, puis l'exécution concurrente de l'ensemble des drivers et enfin nous vérifions la compatibilité de la communication avec les contraintes temporelles du système.

La tâche Driver effectue les transmissions en suivant le protocole réseau. Chaque site du réseau possède un exemplaire de cette tâche Driver qui s'exécute sur un processeur qui lui est dédié. Le langage D des comportements de la tâche Driver s'obtient à partir du protocole qu'elle implémente. L'unité de temps correspondant au langage D est la durée $g_D$ d'émission d'un bit. Au départ, ce langage est observé avec une granularité égale à son unité de temps. On a $\forall i \in I$, ${}_{g_D}D_{g_D}$ est le langage associé au driver i. Tous les drivers fonctionnant suivant le même protocole, ils partagent le même code et donc le même langage.

Soit ${}_{g_D}R = \underset{i \in I}{\Omega}{}_{\text{Protocole}}\ {}_{g_D}D$, le langage contenant les comportements simultanés des drivers. (voir exemple en Figure 5). L'ensemble de synchronisation est l'ensemble des comportements verifiant le prédicat qui exprime les contraintes de communication imposées aux drivers par le protocole. Cet ensemble est donc totalement défini par ce prédicat.

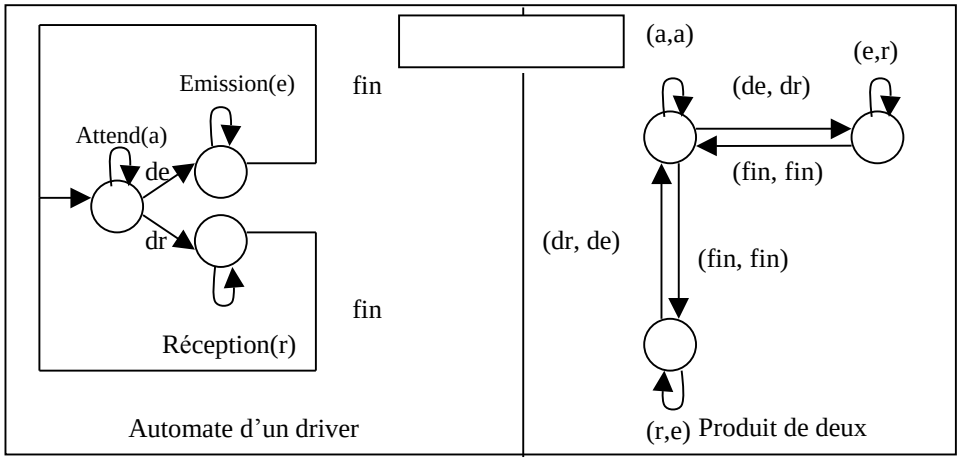


**Figure 5 : exemple de modélisation de drivers**

### 2.2.2 La tâche chronomètre

Nous disposons maintenant du langage collectant les comportements temporels de l'application ainsi que celui des comportements temporels des drivers. Nous cherchons dans la suite un moyen de vérifier la compatibilité entre l'émission d'un message et les contraintes temporelles de l'application.

Pour garantir les échéances de l'application, le message doit être émis dans un temps limité. Un message est donc contraint par un délai critique que nous calculons. Nous utilisons ici une tâche virtuelle Chr qui évalue le délai maximum effectif dédié au transfert du message pour une configuration donnée. La tâche chronomètre commence à compter au moment de l'envoie du message, effectue une incrémentation a chaque nouvelle unité de temps puis arrête le compte dès réception du message.

Le délai ainsi calculé, associé à une modélisation des différents comportements du réseau, permet de décider la compatibilité temporelle de la transmission avec les contraintes temporelles de la configuration. La granularité de Chr est G .(notation : $_{G}$Chr, voir Figure 6).

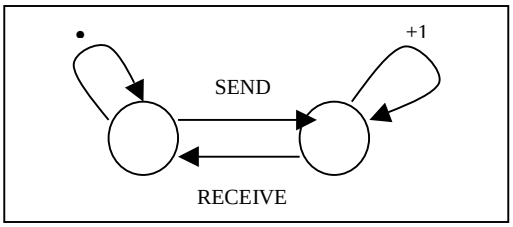


**Figure 6 : chronomètre**

Soit $_{G}$L' = $_{G}$L $\Omega_{R}$ $_{G}$Chr, calculé avec une synchronisation classique de type « Ressource » sur le chronomètre [GL01]. Ce langage contient les comportements du système (stations, chronomètre ) valides du point de vue du fonctionnement du chronomètre. $_{G}$Pchr = $\Pi_{Chr}$ ($_{G}$L') est la projection sur la composante chronomètre de ce langage. $_{G}$Pchr fournit le temps entre l'envoi et la réception du message.(voir exemple en Figure 7).

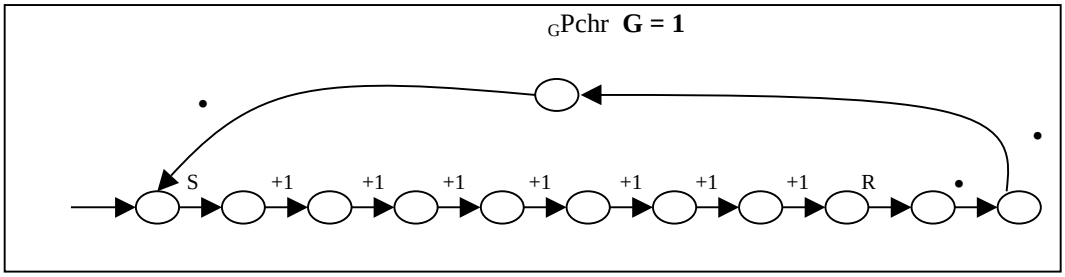


**Figure 7: exemple d'utilisation du chronomètre**

### 2.2.3 Utilisation de Chr pour la validation

Pour tester la validité de la transmission, nous devons pouvoir effectuer le produit de Hadamard des langages : $_{G}$Pchr et $_{g_D}$R. Pour cela, nous appliquons le théorème 1 (section 2.1.2): soient G'= G$\wedge g_{D,}$ $_{G'}$Pchr'= $\psi_{G,\ G'}$($_{G}$Pchr), $_{G'}$R'=$\psi_{g_D,G'}$($_{g_D}$R) (voir l'exemple présenté en Figure 8).

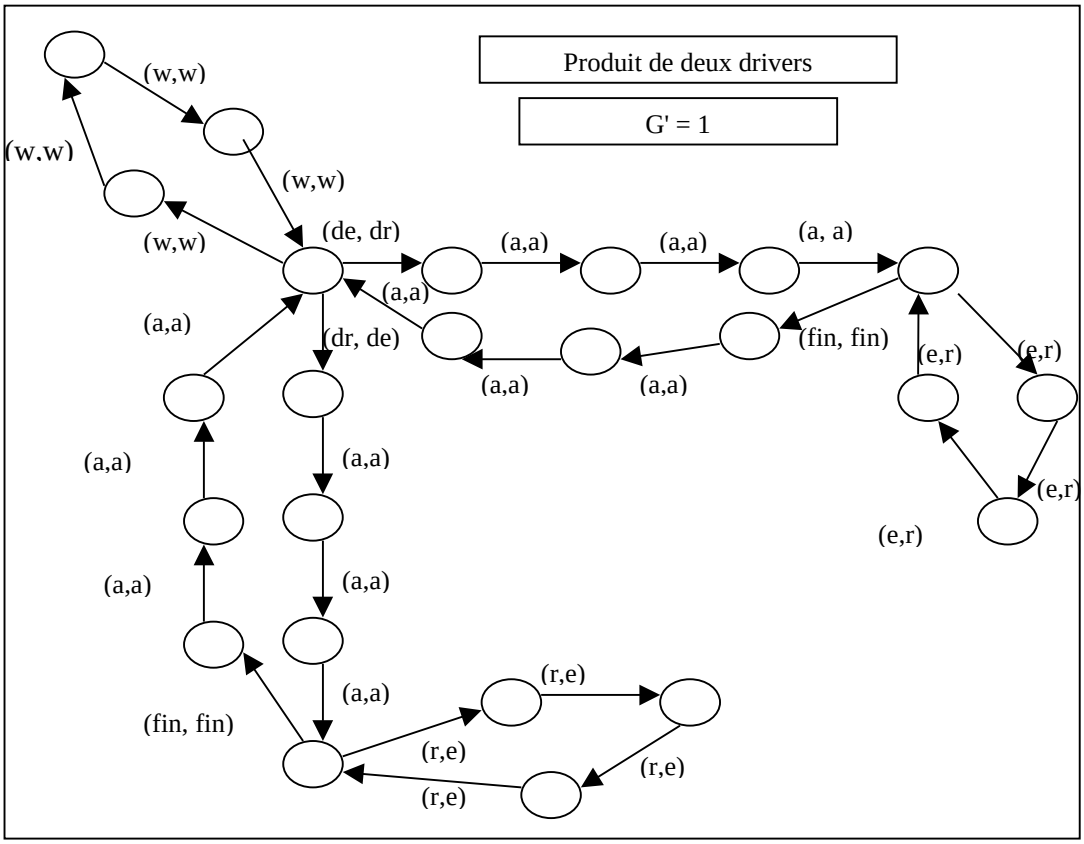


**Figure 8 : modification de granularité de l'automate des drivers**

Le langage $_{G'}$L=Centre($_{G'}$Pchr' $\Omega_{S_r}$ $_{G'}$R' ) collecte l'ensemble des ordonnancements valides ( au sens du respect des contraintes temporelles ) de

messages sur le réseau. Le test de validité est le même que dans le cadre de la validation de système centralisé :

si L = ∅ alors il n'existe aucun comportement temporel valide, sinon, il en existe au moins un.

En utilisant le langage $_{G'}$Pchr et le langage $_{G'}$R ainsi qu'une synchronisation classique de type ressource, on peut donc valider une application distribuée.

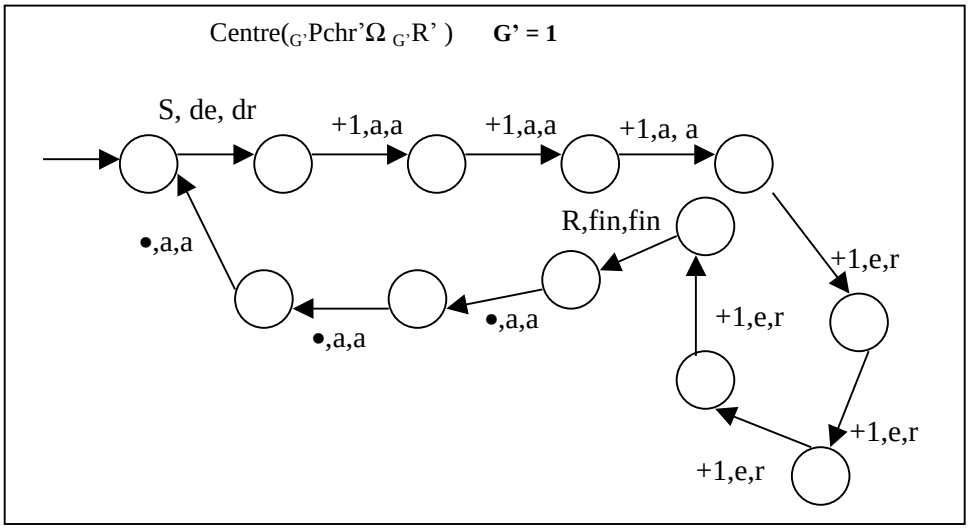


**Figure 9: automate de décision**

Sur l'exemple (voir résultat présenté en Figure 9), les deux tâches $\tau_1$ et $\tau_2$ peuvent communiquer en respectant leurs contraintes temporelles. En effet, le langage reconnu par l'automate n'est pas vide.

Dans la suite, nous montrons la faisabilité de cette technique sur l'exemple du protocole CAN.

# 3 Etude de cas : Validation d'une application basée sur CAN

Le protocole CAN ( Controller Area Network )[ISO94b,ISO94a] supporte la distribution de commandes en temps réel avec un haut niveau de sécurité. Il s'agit d'un protocole de la sous couche MAC (Medium Access Control ) basé sur la technique d'écoute CSMA /CA [P99]

La couche physique du protocole CAN décrit la représentation détaillée du bit, mais pas le moyen de transport et les niveaux des signaux. La couche liaison doit notamment corriger les erreurs qui ont pu se produire au premier niveau. Elle

est entièrement décrite dans le protocole CAN. La sous-couche LLC (Logical Link Control) effectue le filtrage des messages, la notification des surcharges (overload), le recouvrement des erreurs. La sous-couche MAC (Medium Access Control), cœur du protocole CAN, effectue la mise en trame du message, l'arbitrage, l'acquittement, la détection des erreurs, la signalisation des erreurs. Les couches réseau, transport, session et présentation sont vides. La couche application, donne aux applications le moyen d'accéder aux couches inférieures. Cette couche n'est pas vide pour le protocole CAN, mais sa spécification est laissée à l'utilisateur.

Le bus peut avoir l'une des deux valeurs logiques 0 (bit dominant) ou 1 (bit récessif). Dans le cas d'une transmission simultanée , le conflit est résolu par un arbitrage bit à bit (non destructif ) tout au long du contenu de l'identificateur du message. Cet arbitrage repose sur un « ou câblé » au niveau du bus. Ce mécanisme d'arbitrage garantit qu'il y aura ni de perte de temps ni d'information. L'identificateur définit une priorité fixe au message.

L'information est envoyée dans un format défini et de longueur maximale limitée, la transmission est donc aussi limité dans le temps. Les trames sont constituées de sept champs présentés en Figure 10.

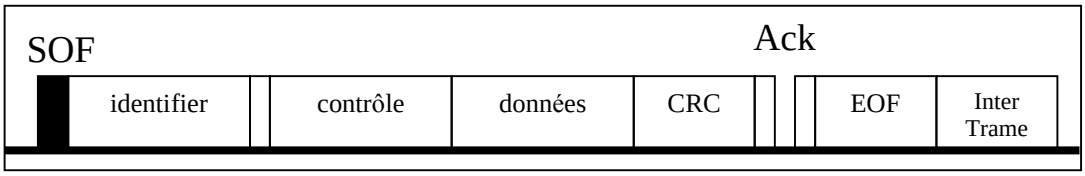


**Figure 10: format des trames CAN**

Le protocole CAN intègre de nombreux dispositifs de signalisation et détection d'erreurs. L'écoute du bus permet à l'émetteur de vérifier que le niveau du bus correspond au niveau du bit émis. Toute détection d'erreur est signalée à tous les nœuds par un *error-flag*. Ce signal est constitué de 6 bits consécutifs de même niveau. De plus, chaque station possède deux compteurs : un *transmit error counter* et un *receive error counter* qui évoluent suivant les échecs ou succès des émissions. Une station peut être dans trois états : erreur active, erreur passive, bus_off. Le passage de l'un à l'autre de ces états s'effectue en fonction des valeurs des deux compteurs.

### 3.1 Le protocole CAN sans gestion d'erreur

Le comportement du driver CAN est défini par un algorithme dans la norme ISO [ISO94b,ISO94a]. L'automate correspondant est donné en Figure 11.

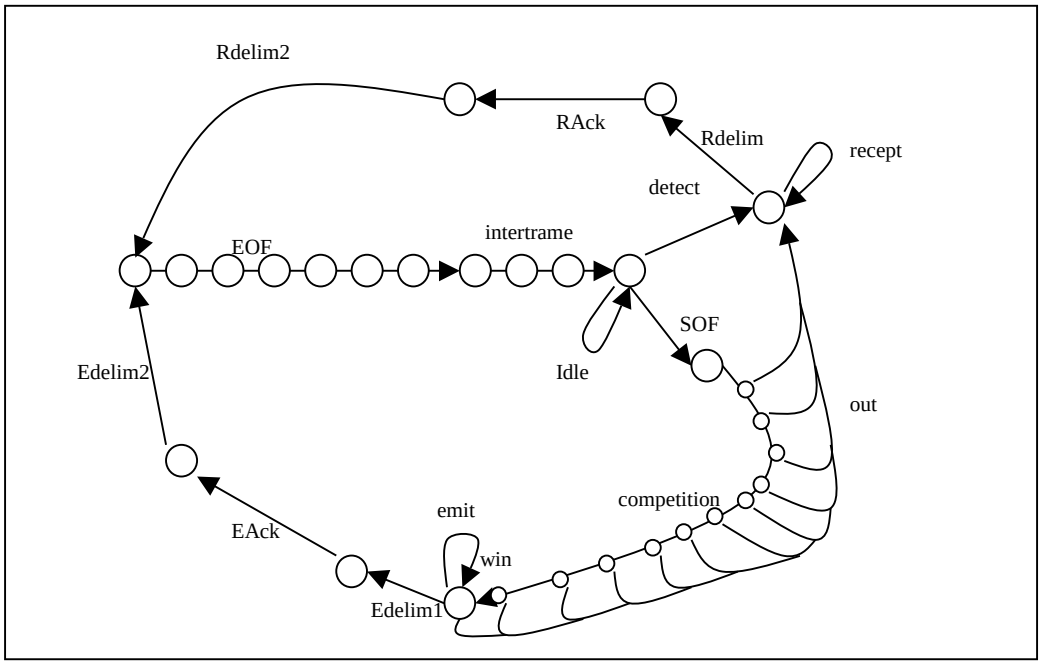


**Figure 11 : automate du driver**

### 3.2 Intégration de la synchronisation des drivers dans la modélisation

Pour obtenir le fonctionnement de l'ensemble des drivers, nous utilisons la technique du produit synchronisé qui s'appuie sur un ensemble de synchronisation. Ici, nous caractérisons l'appartenance à cet ensemble via la satisfaction d'un prédicat, dont nous donnons la négation ci-dessous.

La synchronisation s'exprime par la disjonction d'un ensemble de prédicats :

- **( " i w[i]=Idle).**

  Ce prédicat garantit que tous les drivers sont en état d'attente si aucun d'eux ne veut émettre et que le bus est vide. (Figure 12.0 ):

- **( |{i/ w[i] = SOF }| + |{ i/ w[i] = detect }| = N).**

  Lorsqu'un échange est lancé, les drivers se séparent en deux catégories : ceux qui ont quelque chose à émettre (et qui donc émettent un SOF) et ceux qui n'ont rien à émettre (et qui doivent détecter le début de l'échange pour pouvoir se placer en réception (Figure 12.1)).

- **($i/ w[i] = competition Ù " j  w[j] Î {competition, out, recept} ).**

Lorsque le début de l'échange est synchronisé, on garantit que toutes les stations sont soit en train d'écouter, soit en train de concourir pour l'arbitrage. Ainsi, lors de cette phase, chaque driver peut effectuer l'une des trois actions suivantes : « recept », « competition » et « out » (lorsqu'il est éliminé de la compétition (Figure 12.2)).

- **( $! i w[i] = win Ù "j, j ¹ i, w[j] Î { out, recept }).**

  Lorsque la compétition prend fin, il ne peut y avoir qu'un seul vainqueur. Les autres drivers doivent être en réception sauf éventuellement les derniers éliminés(Figure 12.3).

- **($! i w[i] = emit Ù "j, j ¹ i, w[j] = recept ).**

  On synchronise alors l'émission avec la réception, un seul émetteur, les autres sont récepteurs (Figure 12.4) :

- **( $! i w[i] = Edelim1Ù "j, j ¹ i, w[j] = Edelim1 ).**

  Enfin, on synchronise la fin de l'émission (Figure 12.5) par :

- **( $! i w[i] = Edelim2 Ù "j, j ¹ i, w[j] = Rdelim2 ).**

  Pour garantir que tous les drivers sont au niveau de l'acquittement en même temps (Figure 12.6 ).

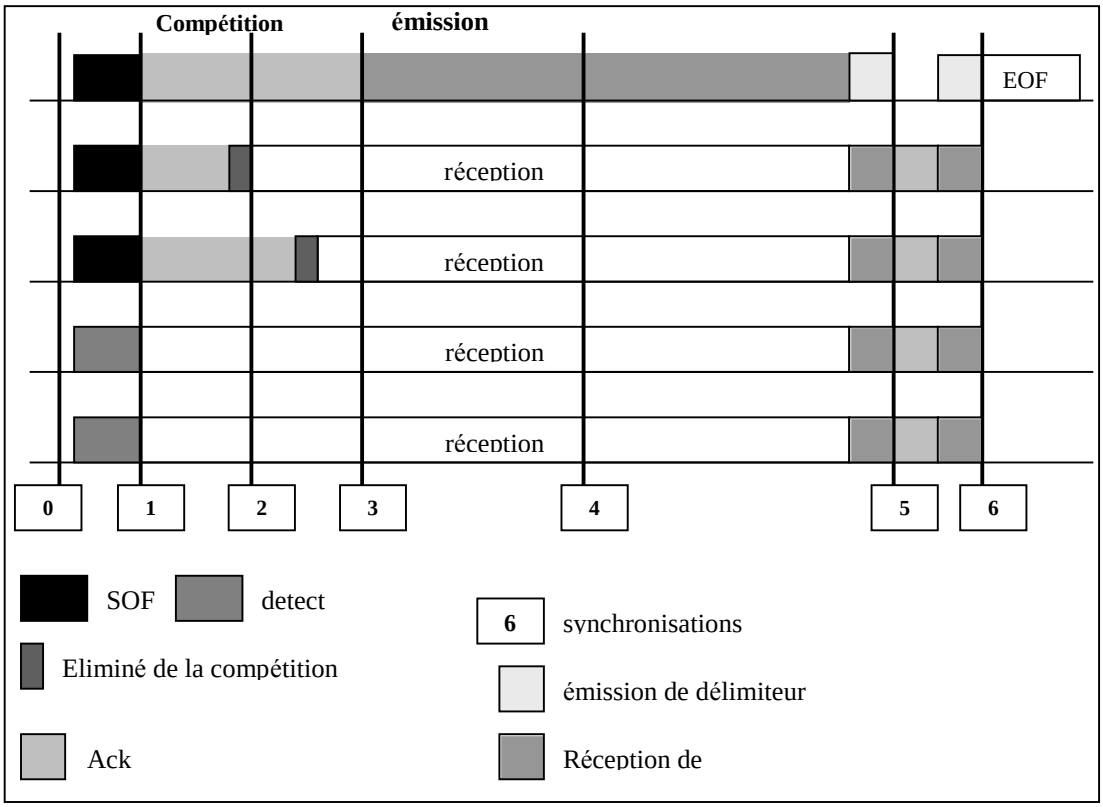


**Figure 12 : exemple d'exécution simultanée de drivers**

Pour N drivers et un N-uplet w, la fonction de synchronisation est récapitulée en Figure 13.

$( \forall\ i\ w_i = Idle ) \vee$

$( |\{i/\ w_i = SOF \}| + |\{\ i/\ w_i = detect \}| = N ) \vee$

$(\exists i/\ w_i = competition \wedge \forall j\ \ (w_j \in \{competition, out, recept\})) \vee$

$( \exists!\ i\ w_i = win \wedge \forall j,\ \ j \neq i, (w_j \in \{\ out, recept\ \})) \vee$

$( \exists!\ i\ w_i = emit \wedge \forall j,\ \ j \neq i, (w_j = recept\ )) \vee$

$( \exists!\ i\ w_i = Edelim1 \wedge \forall j,\ \ j \neq i, (w_j = Edelim1\ )) \vee$

$( \exists!\ i\ w_i = Edelim2 \wedge \forall j,\ \ j \neq i, (w_j = Rdelim2\ ))$

**Figure 13 : fonction de synchronisation**

Cette analyse nous fournit l'ensemble des instructions critiques Pc ( celles qui interviennent dans la fonction de synchronisation). Le modèle temporel ( voir Figure 14) est obtenu par projection du langage $L_D$ du driver (Figure 11) sur $Pc \cup \{a\}$.

Notons Pn l'ensemble des actions du driver qui ne sont pas synchronisées (non-critiques). La projection utilisée est définie par :

$\phi$ : $Pc \cup Pn \rightarrow Pc \cup \{a\}$

$\phi / Pc = Id$

$\phi / Pn = (x \rightarrow a\ )$

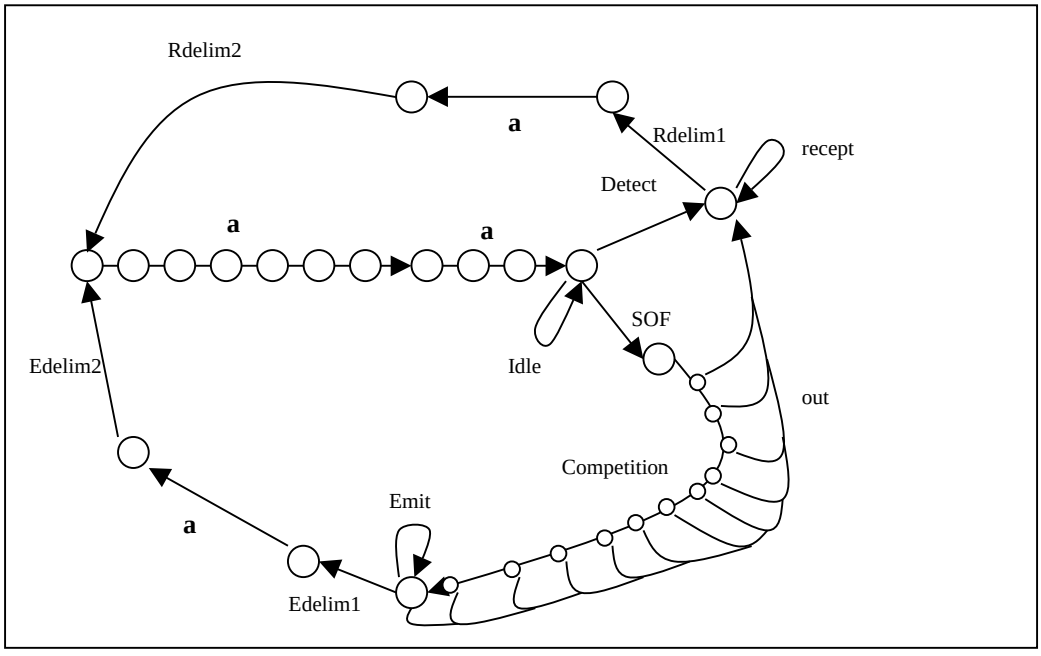


**Figure 14 : modèle temporel $A_D$ du driver**

### 3.3 Gestion d'erreur dans CAN

La prise en compte de la gestion d'erreur modifie légèrement le langage. $L_D$. La démarche reste la même ( voir l'automate modifié en Figure 15).

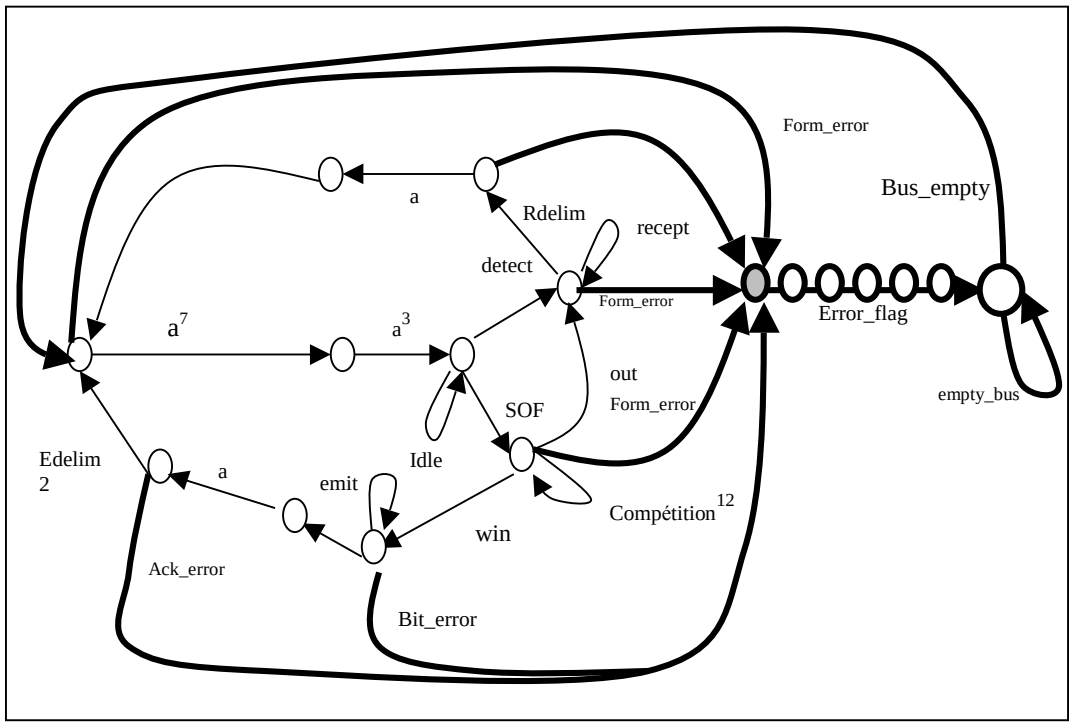


**Figure 15 : automate $A_D$ avec prise en compte de la détection d'erreur**

Les récepteurs détectent les erreurs de forme de la trame et contrôlent la correction du message ( vérification du CRC). L'émetteur, grâce à un système d'écoute, détecte les erreurs de bits ( bit écouté différent du bit émis ) ainsi que les erreurs d'acquittement.

Quel que soit le type de l'erreur détectée, le driver émet un « error_flag » ( sur l'automate de la Figure 15, le début de cette émission est représenté par l'état grisé ).

$( \forall\ i\ w_i = Idle ) \vee$

$( |\{i/\ w_i = SOF\ \}| + |\{\ i/\ w_i = detect\ \}| = N|).\vee$

$(\exists i/\ w_i = competition \wedge \forall j\ \ ( w_j \in \{competition,out,recept,$ **error, error_flag**$\})) \vee$

$( \exists!\ i\ w_i = win \wedge \forall j,\ \ j \neq i, ( w_j \in \{out, recept,$ **error, error_flag** $\})) \vee$

$( \exists!\ i\ w_i = emit \wedge \forall j, j \neq i,( w_j \in \{recept,$ **error, error_flag** $\})) \vee$

$( \exists!\ i\ w_i = Edelim1 \wedge \forall j,\ \ j \neq i, ( w_j \in \{$ Edelim1, **error , error_flag** $\})) \vee$

$( \exists!\ i\ w_i = Edelim2 \wedge \forall j,\ \ j \neq i, ( w_j \in \{$ Rdelim2, **error, error_flag** $\}))\vee$

**$( |\{ i / w_i =$ empty_bus$\}| + |\{ i / w_i =$ error $\}|+|\{ i / w_i =$ error_flag$\}| = N ) \vee$**

**$(\ "\ i\ w_i =$ bus_empty$)$**

**Figure 16 : fonction de synchronisation**

Ces contraintes sont modélisées via la fonction de synchronisation (voir les éléments rajoutés par rapport à la fonction précédente, présentés en gras en Figure 16). Ces éléments correspondent à la synchronisation des détections et signalisations des erreurs dont un exemple est présenté Figure 17. Des erreurs de forme peuvent être détectées soit lors de la compétition, soit pendant l'émission des éléments fixes du message.

De plus, certains drivers peuvent commencer à émettre un « error_flag » sans que les autres aient détecté une erreur. L'erreur sera détectée par tous les drivers au bout d'un temps maximum qui dépend de la géométrie du réseau. Une fois le bus libre (niveau récessif ), tous les drivers réinitialisent leur communication.

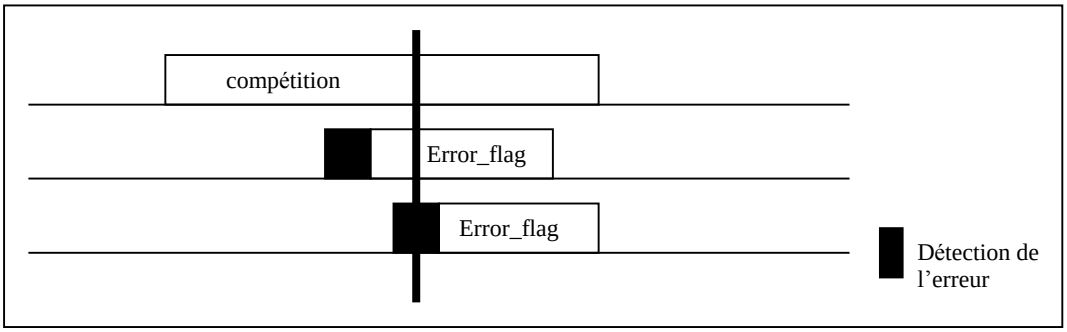


**Figure 17 : synchronisation de la phase compétition**

### 3.4 Le protocole CAN avec gestion des états dégradés

Les comportements temporels du driver dans les états d'erreur active et d'erreur passive sont identiques, seule la valeur des bits émis pour l' « *error_flag* » diffère. Pour notre modèle temporel il n'est donc pas nécessaire de différencier ces deux états. En revanche l'état « bus_off » implique une déconnexion du bus qui peut entraîner des problèmes de communication et donc de synchronisation entre tâches. Il est donc nécessaire de représenter cet état. Nous obtenons l'automate présenté en Figure 18.

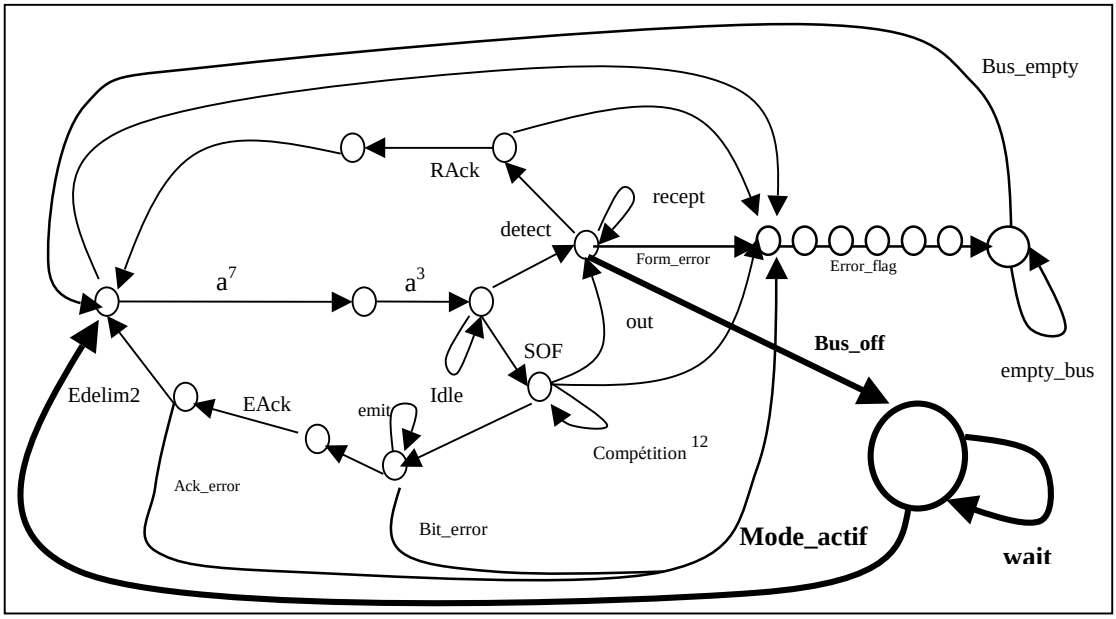


**Figure 18 : automate du driver, gestion des états dégradés**.

Par une démarche semblable au cas précédent, nous obtenons la fonction de synchronisation permettant de synchroniser le produit de Hadamard de N drivers (voir Figure 19). Les ajouts sont dus au fait qu'à chaque instant, une station peut être dans l'état « *bus_off* ».

$(\forall i\ w_i = Idle \vee \mathbf{w_i \hat{I}\ \{ Bus\_off,\ wait, Mode\_actif \}}) \vee$
$(|\{i/\ w_i = SOF\}| + |\{\ i/\ w_i = detect\}| + \mathbf{|\{\ i/\ w_i = Bus\_off\ \acute{U}\ w_i = wait\ \acute{U}\ w_i = Mode\_actif\ \}|} = N|) \vee$
$(\exists i/\ w_i = competition \wedge \forall j\ \ (w_j \in \{competition, out, recept, error, error\_flag,\ \mathbf{Bus\_off, wait, Mode\_actif}\ \})) \vee$
$(\exists!\ i\ w_i = win \wedge \forall j,\ j \neq i, (w_j \in \{\ out, recept, error, \mathbf{Bus\_off, wait, Mode\_actif}\ \})) \vee$
$(\exists!\ i\ w_i = emit \wedge \forall j, j \neq i, (w_j \in \{recept, error, error\_flag, \mathbf{Bus\_off, wait, Mode\_actif}\ \})) \vee$
$(\exists!\ i\ w_i = Edelim1 \wedge \forall j,\ j \neq i, (w_j \in \{\ Edelim1, error, error\_flag, \mathbf{Bus\_off, wait, Mode\_actif}\ \}))\vee$
$(\exists!\ i\ w_i = Edelim2 \wedge \forall j,\ j \neq i, (w_j \in \{\ Rdelim2, error, error\_flag, \mathbf{Bus\_off, wait, Mode\_actif}\ \})) \vee$
$(|\{\ i/\ w_i = empty\_bus\}| + |\{\ i/\ w_i = error\}| + |\{\ i/\ w_i = error\_flag\}| + \mathbf{|\{\ i/\ w_i = Bus\_off\ \acute{U}\ w_i = wait\ \acute{U}\ w_i = Mode\_actif\ \}|} = N) \vee$
$(\forall i\ w_i = bus\_empty \vee \mathbf{w_i = Bus\_off \acute{U}\ w_i = wait\ \acute{U}\ w_i = Mode\_actif})$

**Figure 19 : fonction de synchronisation**

Dans cette partie, nous avons raffiné le modèle, pour prendre en compte les différences de cadence des différents composants d'un réseau. La solution proposée peut être utilisée dans le cadre des systèmes centralisés multiprocesseurs sans horloge globale. Cette méthode est applicable à tout type de réseau et protocoles dès lors que le protocole supporte une modélisation par automate fini.

# 4 Conclusion

Nous sommes donc capables, avec la technique des langages $L_u(\tau)$, de valider temporellement des systèmes temps réel à contraintes strictes répartis, qui s'appuient sur des protocoles de communications modélisables à l'aide de langages rationnels. L'étude du cas de CAN valide cette approche. Pour ce faire, le modèle a été étendu : une tâche est maintenant associée à une classe de langages. Chacun de ces langages est paramétré par une unité de temps. Le modèle appliqué à la tâche est le langage de sa classe associée correspondant au processeur sur lequel elle s'exécute.

Notre méthode de modélisation de systèmes distribués peut être appliquée à n'importe quel protocole, la seule contrainte étant de définir le langage correspondant à ce protocole pour obtenir le langage ***D***. Le résultat obtenu avec cette méthode est une décision d'ordonnançabilité de l'application sur une architecture distribuée.

Pour tenir compte de la différence de cadence des différents éléments d'un système distribué, le modèle a du être raffiné par rapport aux études précédentes, en lui intégrant la quantification de l'unité de temps. Notons bien qu'il ne s'agit toute fois pas d'une temporisation du modèle : les méthodes de décision utilisées restent basées sur les résultats de la théorie des langages sans temps. Cette méthode peut aussi être appliquée sur un système centralisé multiprocesseurs si les différents processeurs n'ont pas la même vitesse.

L'un des apports centraux de cette approche est l'établissement de la cyclicité des ordonnancements en environnement multi-processeurs réparti : ce résultat est un corrolaire immédiat (lemme de l'étoile) du fait que l'ensemble des ordonnancements valides est un langage rationnel.

La poursuite de cette étude est en cours. Les deux principales directions sont :

- L'étude de la migration partielle ou totale des tâches sur les différents sites du système.
- La validation de la méthode sur une configuration réelle de système distribué.

## Bibliographie


[A94] A. Arnold, *Finite transition systems*, Prentice Hall, 1994.

[BRH90] S.K. Baruah, L.E. Rosier, R.R. Howell, *Algorithms and Complexity Concernig the Preemptive Scheduling of Periodic, Real-Time Tasks on one Processor*, Real-Time Systems, 1990

[G00] D. Geniet, *Validation d'applications temps réel à contraintes strictes à l'aide de langages rationnels*, RTS'2000, 2000.

[GL01] D.Geniet, G.Largeteau, *Validation d'applications temps réel strictes à durées variables à l'aide de langages rationnels*, MSR'2001, Toulouse, octobre 2001.

[Gro99] E.Grolleau, *Ordonnancement temps téel hors-ligne optimal à l'aide de réseaux de Petri en environnement monoprocesseur et multi-processeurs*, Thèse.ENSMA,1999.p. 301-324.

[ISO94a] International Standard Organization. *Interchange of Digital Information – Controller Area Network for high-speed Communication*. Technical Report 11898, ISO, 1994.

[ISO94b] International Standard Organization. Road Vehicles – *Low Sped serial data communication –Part 2 : Low Speed Controller Area Network*. Technical Report 11519-2, ISO, 1994.

[P99] D.Paret. Le bus CAN : description -de la théorie à la pratique. DUNOD, Paris, 1999.

[S98] S.Saad-bouzefrane, *Etude temporelle des applications temps réel distribuées à contraintes strictes basée sur une analyse d'ordonnançabilité*. Thèse. ENSMA.1998.