Modélisation et vérification de systèmes intégrés asynchrones dans le langage synchrone Lustre : application aux systèmes avioniques[1]

Frédéric Boniol[a] & Gérard Bel[b] & Jack Foisseau[a]

ONERA-CERT
[a] *Département Traitement de l'Information et Modélisation*
[b] *Département Conduite des Systèmes et Dynamique du vol*
2 av. E. Belin 31055 Toulouse
{boniol,bel,foisseau}@cert.fr

Résumé

P. Caspi *et al.* ont proposé récemment dans [7] une modélisation de la notion de "dérive d'horloges de même fréquence" dans le langage synchrone à flots de données Lustre. L'objet de cet article est de généraliser cette démarche et proposer une abstraction de la notion d'asynchronisme dans les systèmes distribués embarqués. L'intérêt escompté est de permettre la vérification formelle et globale de systèmes distribués asynchrones au moyen des outils de la programmation synchrone. Cet article repose en grande partie sur un exemple issu du domaine avionique.

1. Introduction

1.1. Contexte et problématique générale

Les programmes informatiques occupent une place grandissante dans les systèmes complexes tels qu'une centrale nucléaire ou un aéronef. La croissance rapide de la puissance des calculateurs permet aujourd'hui de leur confier des fonctions importantes, souvent vitales, au cœur de ces systèmes. Par exemple, le développement de l'informatique embarquée a permis la conception d'avions aérodynamiquement instables, artificiellement stabilisés par un système de commandes numériques de vol.

Qu'ils soient intégrés dans un train, un avion ou une centrale nucléaire, ces systèmes informatiques partagent des caractéristiques communes. Ils sont généralement :

- critiques, leurs défaillances pouvant avoir des conséquences catastrophiques,
- réactifs et temps-réel, c'est-à-dire assujettis au comportement de l'application et de son environnement,
- distribués sur un réseau de calculateurs souvent asynchrones, pour des raisons liées à la fois à la géographie de l'application (distribution des capteurs et des actionneurs par exemple) et à la sûreté de fonctionnement.

1. Ce travail a été réalisé dans le cadre du projet de recherche PRISME-AVN financé par l'ONERA (Office National d'Etudes et Recherches Aérospatiales) et d'une étude financée par Dassault-Aviation.

Par exemple, les systèmes vitaux d'un aéronef moderne tel que l'Airbus A340 (pilote automatique, commandes de vol, système de freinage...) sont implémentés par plus de 50 calculateurs répartis dans l'aéronef et qui communiquent par échanges de messages via des bus numériques. De part cette configuration distribuée, et pour des raisons de tolérance aux fautes, ces calculateurs sont asynchrones entre eux : il n'existe pas d'horloge commune, ni de mécanisme de synchronisation. Les horloges de ces calculateurs peuvent donc dériver les unes par rapport aux autres. De même, les traitements informatiques exécutés sur chaque calculateur ont des durées parfois variables et peuvent être soumis à des phénomènes de gigue non déterministe.

Se pose alors le problème de la validation de tels systèmes, étant entendu que cette validation doit être globale et doit prendre en considération l'asynchronisme inhérent au système.

Une approche explorée récemment par A. Benveniste *et al.* [2, 3] et par DASSAULT-AVIATION [8] consiste à spécifier un système global, tel qu'un système avionique, dans un formalisme synchrone (en l'occurence SIGNAL et ESTEREL), puis dans un deuxième temps morceler ce système (par des techniques de compilation séparée) pour le distribuer sur un réseau de calculateurs, et enfin synthétiser un protocole d'exécution et de synchronisation réparti sur l'ensemble du réseau de telle sorte que la sémantique synchrone de l'application soit préservée par l'implémentation distribuée. Ce protocole repose sur des échanges de messages qui imlémentent les relations de causalité entre les processus de l'application.

Malheureusement, cette démarche n'est pas applicable dans le domaine de l'avionique civile. Du fait du caractère largement multi-partenaires des grands projets aéronautiques civils, l'approche traditionnellement suivie par des avioneurs tels que AEROSPATIALE consiste à concevoir et valider *séparément* chaque équipement de l'aéronef, puis à simuler leur intégration au sein de l'aéronef. Chaque équipement est spécifié dans un formalisme synchrone tel que LUSTRE, puis plongé dans un monde globalement asynchrone. Cette approche se heurte cependant à deux difficultés :

1. l'évaluation de la robustesse d'un processus échantillonné face à l'asynchronisme de ses entrées, question abordée par C. André *et al.* dans [1] pour des processus spécifiés en ESTEREL,
2. la modélisation de l'asynchronisme de l'architecture globale, par nature difficile à reproduire et à explorer par des techniques de simulation. La tâche de validation du système global en est rendu de ce fait plus difficile est plus longue.

Cet article se focalise sur le point 2 ci-dessus. En réponse à cette difficulté, P. Caspi *et al.* ont récemment proposé dans [7] une modélisation de la notion de "dérive d'horloges de même fréquence" dans le langage synchrone à flots de données LUSTRE. L'intérêt de cette approche est de permettre la modélisation d'un système distribué asynchrone, par essence non déterministe, sous la forme (abstraite) d'un programme synchrone, dans l'objectif d'en permettre la vérification automatique par des techniques d'analyse de modèles ("model checking").

Cet article se propose de généraliser cette approche (*cf.* section 3), pour ensuite l'appliquer à un exemple du domaine avionique (*cf.* section 4).

1.2. Asynchronisme dans les systèmes intégrés temps réel

Les applications intégrées temps réel, telles que les systèmes avioniques, sont généralement *distribuées*, c'est-à-dire composées d'un ensemble de processus E_i (souvent périodiques) qui communiquent et s'exécutent en parallèle, et alloués sur des processeurs interconnectés par un ensemble de bus à diffusion. Compte tenu du caractère critique de ces systèmes, ces processeurs et ces bus sont généralement dimensionnés de telle sorte que les processus E_i respectent leurs échéances, et que la transmission d'information le long d'un bus à diffusion puisse être considérée comme instantanée.

On peut cependant identifier trois sources d'asynchronisme au sein de tels systèmes, à savoir les dérives entre les horloges de chaque processeur, d'éventuelles gigues dans le séquencement des processus, et les latences d'exécution (parfois variables) de ces processus.

Dérive d'horloges. Pour des raisons de tolérance aux pannes, il n'existe généralement pas dans les aéronefs civils de mécanismes de synchronisation d'horloges. Les horloges des calculateurs peuvent donc dériver les unes par rapport aux autres. Cette dérive, éventuellement variable, est cependant souvent bornée.

Formellement, une dérive d'horloge peut être modélisée comme une "imprécision" bornée sur la période de l'horloges exprimée par rapport temps réel absolu. Soit H une horloge de période nominale (*i.e.* sans dérive) T, on dira que H est soumise à une dérive maximale d pour exprimer que la période effective T_e de H varie dans l'intervalle

$$T_e \in [T - d, T + d]$$

Par souci de simplicité, nous supposerons dans le reste de l'article que les périodes nominales T et les dérives maximales d sont à valeurs dans l'ensemble des entiers.

Gigue. Le deuxième élément d'asynchronisme fréquemment rencontré dans les calculateurs temps réel est le phénomène de gigue, c'est-à-dire de retard entre l'instant prévu d'apparition d'un événement (tel qu'une activation de tâche, une production de donnée...) et son instant d'apparition effective. Un tel retard, éventuellement variable, peut fréquemment être borné par une valeur maximale.

Considérons par exemple un calculateur C de période nominale T_c soumis éventuellement à une dérive de valeur maximale d_c. Considérons une fonction périodique F de période $T_f = kT_c$ éventuellement soumise à une gigue de valeur maximale g_f. Alors, les instants effectifs d'exécution de F sont définis par la suite

$$t_e^n = t^n + g_e^n \quad \text{où } g_e^n \text{ est la gigue effective de } F \text{ à cet instant} : g_e^n \in [0, g_f]$$

avec

$$\begin{cases} t^n = t^{n-1} + kT_e^n, & n > 0 \\ t^0 = 0 \end{cases}$$

où T_e^n est la période effective du calculateur au nième instant :

$$T_e^n \in [T_c - d_c, T_c + d_c]$$

Latence d'exécution. La troisième source d'asynchronisme dans un système distribué provient des latences d'exécution, souvent variables, des processus du système.

Considérons par exemple un calculateur C de période nominale T_c soumis éventuellement à une dérive de valeur maximale d_c. Considérons une fonction périodique F de latence maximale l_f, de période $T_f = kT_c$, et éventuellement soumise à une gigue de valeur maximale g_f. Alors, les instants de terminaison effective de F sont définis par la suite

$$t\_fin_e^n = t_e^n + l_e^n \quad \text{où } l_e^n \text{ est la latence effective de } F \text{ à cet instant} : l_e^n \in [0, l_f]$$

où t_e^n est la date d'activation effective de la fonction (*cf.* paragraphe précédent).

Communication inter-processus. Aux trois phénomènes physiques mentionnés ci-dessus s'ajoute une caractéristique applicative des systèmes considérés, à savoir le caractère non synchronisant des communications.

Pour des raisons de tolérance aux défaillances, les communications inter-processus au sein d'un système critique, tel qu'un système avionique, reposent généralement sur le principe de la *diffusion non bloquante de un vers plusieurs.* Ce type de communication présente en effet l'avantage, à l'inverse par exemple du *rendez-vous*, d'une part d'être par nature plus tolérant aux pannes ou aux blocages d'un processus, et d'autre part de permettre le test ou l'évaluation d'une application au moyen "d'observateurs" [10], c'est-à-dire au moyen de programmes placés en parallèle de l'application et qui espionnent les diffusions effectuées par chacun des processus de celle-ci, sans nécessiter pour cela de modifications au sein de ces processus ou de l'application. C'est ce type d'observateur qui est à la base des outils de vérification de programmes LUSTRE [11].

En revanche, le principe de communication par diffusion présente l'inconvénient de ne pas offrir de moyens implicites de synchronisation des applications, mais au contraire d'en préserver l'asynchronisme interne.

En résumé, si l'on considère que la transmission d'informations via des bus numériques est instantanée et que seul l'accès au médium de communication nécessite une durée non nulle éventuellement non déterministe (durée qui peut être intégrée dans la latence d'exécution des processus), alors les notions de dérive d'horloges, de gigue, et de latence d'exécution permettent de rendre compte de l'ensemble des comportements asynchrones inhérents aux systèmes distribués intégrés. La formalisation de ces notions sous une forme opérationnelle (*i.e.* analysable par algorithme) permettra alors la vérification du comportement d'un système asynchrone par des outils automatiques reposant sur des techniques d'analyse de modèles.

1.3. Démarche explorée

Le problème consiste donc à exhiber une formalisation opérationnelle des notions de dérive d'horloges, de gigue, et de latence d'exécution. L'approche suivie dans cet article est une généralisation la piste explorée récemment par P. Caspi *et al.* dans le cadre du projet CRISYS [7]. L'idée consiste à abstraire le comportement global d'un système distribué asynchrone sous la forme d'une fonction LUSTRE construite par la composition

- d'un ensemble d'équations (et éventuellement d'assertions pour les processus non-déterministes) qui modélisent le comportement (abstrait) de chaque processus de l'application ainsi que leurs communications,
- d'un ensemble d'assertions (vues comme des prédicats) qui caractérisent les dérives éventuelles entre les horloges des calculateurs du système considéré,
- d'un ensemble d'assertions qui caractérisent les gigues éventuelles et les latences d'exécution des processus du système.

Une assertion LUSTRE est un prédicat qui permet de définir un ensemble de comportements. L'idée proposée dans [7] et reprise dans cet article consiste à utiliser le mécanisme des assertions pour caractériser l'ensemble des entrelacements des horloges (des calculateurs et des fonctions) d'un système distribué compatibles avec les hypothèses de dérives, de gigues, et de latences d'exécution qui le caractérisent. L'intérêt de cette approche est de permettre d'une part la modélisation complète d'un système distribué et de son asynchronisme sous une forme opérationnelle LUSTRE, et d'autre part son analyse formelle au moyen d'outils de vérification de modèles tels que LESAR [11].

Cet article est organisé comme suit. La section 2 rappelle brièvement les principes fondamentaux du langage LUSTRE. Les principes de modélisation de l'asynchronisme en LUSTRE sont ensuite exposés en section 3, avant d'être illustrés sur une application du domaine avionique (section 4).

2. Brefs rappels sur le langage LUSTRE

LUSTRE [5, 6] est un langage *déclaratif* synchrone de type *flots de données* initialement conçu en 1984 par P. Caspi et N. Halbwachs pour la programmation d'applications du domaine de l'automatique et du temps-réel.

Les variables ou expressions du langage désignent des flots, c'est-à-dire des couples formés d'une suite éventuellement infinie de valeurs typées (entier, réel, ou booléen), et d'une horloge représentant la suite des instants où le flot est présent. La valeur d'un flot au *nième* instant de son horloge est égale au *nième* élément de sa suite de valeurs. Un programme LUSTRE, appelé `node`, est une fonction qui, à chaque arrivée d'une *vague* de valeurs sur ses flots d'entrée, retourne instantanément une vague de valeurs sur ses flots de sortie.

Les variables du langage sont donc des flots typés. Elles sont classées en trois catégories : les variables d'entrée, internes, et de sortie. Chaque variable interne ou de sortie est définie par une unique équation (au sens mathématique du terme) du type "$x = f(e1, \ldots, e_n)$", où $f(e1, \ldots, e_n)$ est une expression composée d'opérateurs agissant sur les flots $e_1 \ldots e_n$ point par point.

Outre les opérateurs arithmétiques ou booléens classiques, LUSTRE possède quatre opérateurs temporels.

- l'opérateur $\rightarrow$ ("suivi de") qui permet d'associer une valeur initiale à un flot,
- l'opérateur `pre` qui permet de mémoriser la valeur d'une expression ou d'un flot à l'instant précédent,
- l'opérateur `when` qui permet d'échantillonner une expression ou un flot sur une horloge plus lente,
- et enfin l'opérateur `current` qui permet de "sur-échantillonner" une expression ou un flot sur l'horloge immédiatement plus rapide.

Le langage LUSTRE possède également un mécanisme de déclaration d'hypothèses qui permet d'exprimer des contraintes ou des relations entre les flots d'entrée et de sortie du système, et par suite de modéliser des hypothèses sur le comportement de son environnement. Par exemple l'assertion

```
assert(current((X=0) when (true ->pre(H))))
```

traduit l'hypothèse que le flot numérique X est nul à l'instant initial, ainsi qu'à chaque instant consécutif à une valeur vraie sur le flot H.

La modélisation des concepts de dérive, gigue et latence proposée dans la section suivante repose sur cette notion d'assertion.

Le lecteur intéressé par une description plus détaillée du langage LUSTRE est invité à se reporter à [5, 6].

3. Modélisation de l'asynchronisme en LUSTRE

3.1. Dérive d'horloge

Le concept de dérive d'horloges, tel que présenté précédemment, est une notion quantitative exprimée par rapport au temps absolu. L'idée proposée dans [7], et que nous exploitons dans cet article, consiste à transformer cette notion de dérive absolue en *dérive relative* de deux ou n horloges. Modéliser les dérives de deux horloges consiste alors à caractériser, à l'aide d'assertions, l'ensemble (ou plus précisément une sur-approximation de cet ensemble) de leurs entrelacements compatibles avec la définition métrique de leur période et leur dérive maximale.

Considérons le prédicat

$$au\_plus\_N\_dans\_M(n, H_1, m, H_2)$$

où n, m sont des constantes entières, H_1, H_2 des horloges (*i.e.* des flots booléens), et tel que $au\_plus\_N\_dans\_M(n, H_1, m, H_2)$ est vrai si et seulement si tout intervalle ouvert à droite défini par m tops consécutifs de H_2 (*i.e.* $m-1$ cycles) contient au maximum n occurrences de l'horloge H_1 (*i.e.*, n valeurs vraies de H_1) :

$$au\_plus\_N\_dans\_M(n, H_1, m, H_2) \quad \text{ssi} \quad \forall i, j, k \;\; (([t_1^j, t_1^{j+k}] \;\subset\; [t_2^i, t_2^{i+m}[) \;\;\Rightarrow\;\; k \leq n)$$

```
node au_plus_N_dans_2 (const N : int; H1, H2 : bool) returns (ok : bool);
var
  occH1_dans_H2 : bool^(N+1);
let
  occH1_dans_H2[0] = if (H2 and not H1) then false
                     else if (H2 and H1) then true
                     else if H1 then true
                     else (false -> pre(occH1_dans_H2[0]));
  occH1_dans_H2[1..N] = false^(N) ->
                     if H2^(N) then false^(N)
                     else if (H1^(N) and pre(occH1_dans_H2[0..N-1])) then true^(N)
                     else pre(occH1_dans_H2[1..N]);
  ok = not occH1_dans_H2[N];
tel.
```

FIG. 1 – *Définition* LUSTRE *du prédicat "au_plus_N_dans_2"*

où $(t_1^i)_{i\in\mathbb{N}}$ et $(t_2^i)_{i\in\mathbb{N}}$ sont les suites des tops de H_1 et H_2 exprimées par rapport au temps réel (*cf.* section 1.2). Une instance du prédicat *au_plus_N_dans_M* pour $m = 2$ est définie en LUSTRE figure 1.

Considérons le prédicat symétrique

$$au\_moins\_N\_dans\_M(n, H_1, m, H_2)$$

où n, m sont des constantes entières, H_1, H_2 des horloges (*i.e.* des flots booléens), et tel que $au\_moins\_N\_dans\_M(n, H_1, m, H_2)$ est vrai si et seulement si tout intervalle ouvert à droite défini par m top consécutifs de H_2 (*i.e.* $m-1$ cycles) contient au minimum n occurrences de l'horloge H_1 :

$$au\_moins\_N\_dans\_M(n, H_1, m, H_2) \quad \text{ssi} \quad \forall i, \exists j \ \ ([t_1^j, t_1^{j+n}] \subset [t_2^i, t_2^{i+m}[)$$

Soit deux horloges H_1 et H_2 dont les périodes effectives T_1 et T_2 varient respectivement dans les intervalles $[a_1, b_1]$ et $[a_2, b_2]$. Le problème consiste alors à exprimer, par une formule logique au moyen d'instances des prédicats ci-dessus, une caractérisation de l'ensemble (ou d'une sur-approximation de celui-ci) des entrelacements possibles de H_1 et H_2. Deux cas se présentent, selon que les intervalles $[a_1, b_1]$ et $[a_2, b_2]$ sont disjoints ou non.

1er cas : $[a_2, b_2]$ et $[a_1, b_1]$ sont disjoints, $a_2 \leq b_2 < a_1 \leq b_1$.

Soit $M_1 = \frac{ppcm(a_2,b_1)}{b_1}$. Soit m un entier quelconque compris entre 1 et M_1, posons n le plus grand entier tel que $na_2 < mb_1$. Or, b_1 est la plus longue période effective possible de H_1 et a_2 la plus courte période effective possible de H_2. Alors n est le nombre maximum de cycles consécutifs de H_2 pouvant être inclus dans m cycles consécutifs de H_1, ou en d'autre termes

$$au\_plus\_N\_dans\_M(n+1, H_2, m+1, H_1) = vrai$$

De même, soit $M_1' = \frac{ppcm(a_1,b_2)}{a_1}$. Soit m' quelconque compris entre 1 et M_1', posons n' le plus grand entier tel que $n'b_2 < m'a_1$. Or, a_1 est la plus courte période effective possible de H_1, et b_2 est la plus longue période effective possible de H_2. Alors n' est le nombre minimum de cycles consécutifs de H_2 pouvant être inclus dans m' cycles consécutifs de H_1, ou en d'autre termes

$$au\_moins\_N\_dans\_M(n'+1, H_2, m'+1, H_1) = vrai$$

En conséquence

$$\left.\begin{array}{l} T_1 \in [a_1, b_1] \\ T_2 \in [a_2, b_2] \\ a_2 \le b_2 < a_1 \le b_1 \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \displaystyle\bigwedge_{m=1\ldots\frac{ppcm(a_2,b_1)}{b_1}} au\_plus\_N\_dans\_M(n+1, H_2, m+1, H_1) \\ \wedge \displaystyle\bigwedge_{m'=1\ldots\frac{ppcm(a_1,b_2)}{a_1}} au\_moins\_N\_dans\_M(n'+1, H_2, m'+1, H_1) \end{array}\right.$$

où pour chaque m, n est le plus grand entier tel que $na_2 < mb_1$, et pour chaque m', n' est le plus grand entier tel que $n'b_2 < m'a_1$

2ème cas : $[a_2, b_2]$ et $[a_1, b_1]$ ne sont pas disjoints, $a_1 \le b_2$ et $a_2 \le b_1$.

On procède par le même raisonnement que ci-dessus. Soit $M_1 = \frac{ppcm(a_2,b_1)}{b_1}$. Soit m quelconque compris entre 1 et M_1, posons n le plus grand entier tel que $na_2 < mb_1$. Or, b_1 est la période possible la plus longue de H_1 et a_2 la période possible la plus courte de H_2. Alors n est le nombre maximum de cycles consécutifs de H_2 pouvant être inclus dans m cycles consécutifs de H_1, ou en d'autre termes

$$au\_plus\_N\_dans\_M(n+1, H_2, m+1, H_1) = vrai$$

D'autre part, soit $M'_1 = \frac{ppcm(a_1,b_2)}{b_2}$. Soit m' quelconque compris entre 1 et M'_1, posons n' le plus grand entier tel que $n'a_1 < m'b_2$. Or, b_2 est la plus longue période effective possible de H_2 et a_1 la plus courte période effective possible de H_1. Alors n' est le nombre maximum de cycles consécutifs de H_1 pouvant être inclus dans m' cycles consécutifs de H_2, ou en d'autre termes

$$au\_plus\_N\_dans\_M(n'+1, H_1, m'+1, H_2) = vrai$$

En conséquence

$$\left.\begin{array}{l} T_1 \in [a_1, b_1] \\ T_2 \in [a_2, b_2] \\ a_1 \le b_2 \;\text{ et }\; a_2 \le b_1 \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \displaystyle\bigwedge_{m=1\ldots\frac{ppcm(a_2,b_1)}{b_1}} au\_plus\_N\_dans\_M(n+1, H_2, m+1, H_1) \\ \wedge \displaystyle\bigwedge_{m'=1\ldots\frac{ppcm(a_1,b_2)}{b_2}} au\_plus\_N\_dans\_M(n'+1, H_1, m'+1, H_2) \end{array}\right.$$

où pour chaque m, n est le plus grand entier tel que $na_2 < mb_1$, et pour chaque m', n' est le plus grand entier tel que $n'a_1 < m'b_2$

Ce qui permet de caractériser sous forme prédicative, et donc d'assertions LUSTRE, une sur-approximation des entrelacements d'horloges dérivant les unes par rapport aux autres.

Considérons par exemple deux horloges H_1 et H_2 de même période théorique $T = 30ms$ et de même dérive maximale $d = 10ms$. Les périodes effectives de H_1 et H_2 varient donc entre 20 et $40ms$. Alors en appliquant la formule ci-dessus (2ème cas), une sur-approximation des entrelacements de H_1 et H_2 peut être décrit en LUSTRE sous la forme de l'assertion

```
assert(au_plus_N_dans_2(2, H1, H2)
      and au_plus_N_dans_2(2, H2, H1));
```

où `au_plus_N_dans_2` est le prédicat LUSTRE défini figure 1. Ce prédicat retourne la valeur vraie tant que le nombre d'occurrences de l'horloge H_1 comptées entre deux occurrences de H_2 est inférieur ou égal à N. En d'autre termes, un cycle de H_1 ne peut contenir qu'au plus 2 cycles de H_2, et inversement.

Considérons un nouvel exemple composé de deux horloges H_3 et H_4 de périodes théoriques respectives $T_3 = 20ms$ et $T_4 = 4ms$ et de dérives maximales respectives $d_3 = 5ms$ et $d_4 = 1ms$.

Les périodes effectives de H_3 et H_4 varient donc respectivement entre 15 et $25ms$ et entre 3 et $5ms$. En conséquence, et conformément au raisonnement suivi ci-dessus (1er cas) :

- l'intervalle compris entre 2 tops de H_3 (soit une durée maximale égale à $25ms$) ne peut contenir qu'au plus 9 tops de H_4 (plage temporelle de durée minimale égale à $24ms$),
- l'intervalle compris entre 3 tops de H_3 (soit une durée maximale égale à $50ms$) ne peut contenir qu'au plus 17 tops de H_4 (plage temporelle de durée minimale égale à $48ms$),
- l'intervalle compris entre 4 tops de H_3 (soit une durée maximale égale à $75ms$) ne peut contenir qu'au plus 25 cycles de H_4, soit 25 tops de H_4 entre quatre tops de H_1,
- et enfin, compte tenu de la période maximale de H_4 et de la période minimale de H_3, un cycle de H_3 doit contenir au moins trois tops de H_4 (soit une durée maximale de $15ms$).

Ce qui peut s'exprimer en LUSTRE sous la forme de l'assertion :

```
assert(au_plus_N_dans_2(9, H4, H3)
       and au_plus_N_dans_3(17, H4, H3)
       and au_plus_N_dans_4(25, H4, H3)
       and au_moins_N_dans_2(3, H4, H3));
```

On retrouve la formulation générale établie ci-dessus (1er cas).

3.2. Gigue

Une gigue est un retard, éventuellement non-déterministe, entre l'instant prévu d'activation d'une tâche et l'instant de son activation effective (*cf.* figure 3). A l'instar de la démarche suivie pour la représentation des dérives d'horloges, une modélisation possible de ce phénomène de gigue consiste à faire abstraction de la durée de cette gigue pour ne considérer que l'entrelacement des tops de l'horloge théorique du processus et des instants d'activation effective.

Soit un processus P d'horloge "théorique" H_t (*i.e.*, en l'absence de gigue), soumis à un phénomène de gigue aléatoire. Notons alors H_g l'horloge de gigue de P, entendue comme la suite des instants d'activation effective de P. Sous l'hypothèse que P respecte ses échéances, c'est-à-dire termine toujours avant la prochaine occurrence de H_t, alors H_t et H_g sont exactement alternées ; chaque top de H_t est suivi par un top de H_g avant le prochain top de H_t (*cf.* figure 3). Ce qui peut s'exprimer en LUSTRE sous la forme de l'assertion :

```
assert(alternance(Ht, Hg));
```

où `alternance` est un prédicat LUSTRE défini figure 2. Ce prédicat est vrai si toute occurrence de H_t est suivie (éventuellement au même instant) d'une occurrence de H_g et réciproquement. Une telle assertion traduit ainsi le phénomène de gigue, c'est-à-dire la possibilité de retards (éventuellement nuls) entre les instants d'activation théorique d'un processus (H_t) et ses instants d'activation effective (H_g).

3.3. Latence d'exécution

Une latence d'exécution est un retard, éventuellement non-déterministe, entre l'instant d'activation effective d'un processus et l'instant de sa terminaison (*cf.* figure 3). A l'instar de la démarche suivie pour la représentation des dérives et des gigues, une modélisation possible des latences consiste à faire abstraction de la durée de ces latences pour ne considérer que l'entrelacement des tops de début et de fin d'exécution.

Soit un processus P d'horloge "théorique" H_t (*i.e.*, en l'absence de gigue). Notons H_g l'horloge de gigue de P. Notons H_{ter} l'horloge de terminaison de ce processus (entendue comme la suite des instants réels de terminaison de P). Sous l'hypothèse que P respecte ses échéances, c'est-à-dire termine

```
node alternance(H1,H2 : bool) returns (ok : bool);
var
  attente_H1 : bool;
let
  attente_H1 = if (H2 and not H1) then true
         else if (H2 and H1 and (true -> pre(atente_H1))) then true
         else if (H1 and not H2) then false
         else if (H1 and H2 and (false -> not pre(attente_H1))) then false
         else (true -> pre(attente_H1));
  ok = ((false -> not pre(attente_H1)) or H1 or not H2)
       and ((true -> pre(attente_H1)) or H2 or not H1);
tel.
```

FIG. 2 – *Définition* LUSTRE *du prédicat "alternance"*

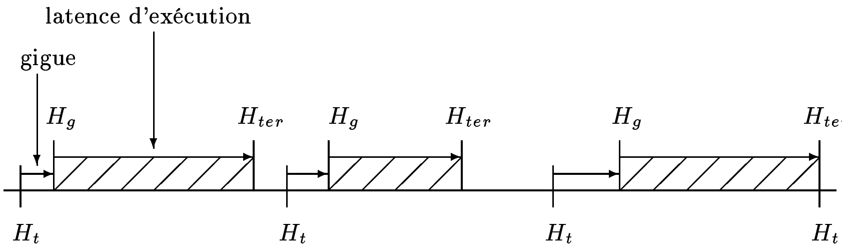

FIG. 3 – *Horloges théorique (H_t), de gigue (H_g), de terminaison (H_{ter}) d'un processus périodique*

toujours avant la prochaine occurrence de H_t, alors H_t, H_g et H_{ter} sont exactement alternées ; chaque top de H_t est suivi par un top de H_g, lui-même suivi par un top de H_{ter}, avant le prochain top de H_t (*cf.* figure 3). Ce qui peut s'exprimer en LUSTRE sous la forme de l'assertion :

```
assert(alternance(Ht, Hg) and alternance(Ht, Hter) and alternance(Hg, Hter));
```

Une telle assertion impose l'enchaînement des tops de H_t, H_g et H_{ter} tel que représenté sur la figure 3.

4. Exemple

Cette section se propose d'illustrer l'approche proposée sur un exemple du domaine avionique (simplifié pour la clarté de l'exposé).

4.1. Présentation informelle

4.1.1. Fonctionnalité, architecture et propriétés attendues

Considérons un système de pilotage d'une gouverne à bord d'un aéronef (*cf.* figure 4). La fonction de ce système est de commander une servo-commande en réponse aux consignes de pilotage (non représentées sur la figure) émises par l'équipage. Pour ce faire, ce système produit en permanence une commande (une tension électrique) sur une ligne analogique à destination de la gouverne. Pour des

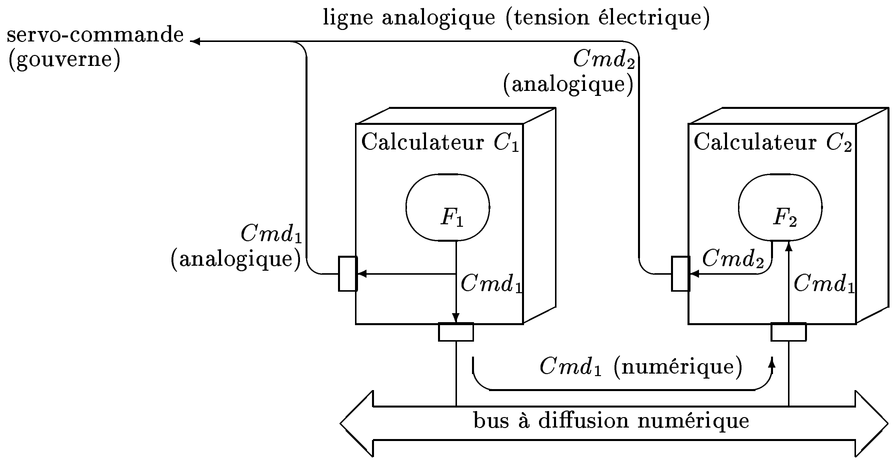

FIG. 4 – *Exemple : un système de pilotage de gouverne simplifié*

raisons de tolérance aux pannes, ce système est structuré en deux parties redondantes :

- une fonction périodique F_1, dite "maître", de période $T_1 = 20ms$, allouée sur le calculateur C_1 et qui produit une commande, dénommée Cmd_1, diffusée périodiquement sur un bus numérique et maintenue en permanence sur une ligne analogique (à destination de la gouverne),
- et une fonction périodique F_2, dite "esclave", de même période $T_2 = T_1 = 20ms$, allouée sur le calculateur C_2 et qui, en cas de défaillance de F_1, produit la commande Cmd_2 sur la ligne analogique. Du point de vue de F_2, la fonction F_1 est considérée comme défaillante si aucune occurrence de Cmd_1 n'est observée sur le bus numérique pendant 5 cycles d'horloge (*i.e.*, une durée de $5T_2 = 100ms$) ; dans ce cas, F_2 prend l'initiative d'émettre Cmd_2.

Les propriétés attendues d'un tel système sont :

P1 : la gouverne ne doit jamais être commandée simultanément par F_1 et par F_2, y compris en présence d'une défaillance de l'une des deux fonctions ;

P2 : la gouverne ne doit pas être "orpheline" (*i.e.*, non commandée) plus de $175ms$ (temps réel absolu), y compris en présence d'une défaillance de l'une des deux fonctions.

La première propriété exprime une exigence d'invariance et d'exclusion mutuelle sur la ligne analogique. La deuxième propriété est en revanche une exigence de vivacité bornée exprimant que toute plage temporelle de longueur égale à $175ms$ doit contenir au moins une occurrence de commande (Cmd_1 ou Cmd_2).

4.1.2. Hypothèses d'asynchronisme

De part son caractère distribué, l'application décrite figure 4 est globalement asynchrone. Supposons que, après une analyse de performances temps réel, cet asynchronisme soit quantifié par :

- dérives d'horloges : les horloges des calculateurs C_1 et C_2 (et donc des fonctions F_1 et F_2) peuvent être victimes de dérives de valeur maximale égale à $5ms$. En conséquences, les périodes effectives de F_1 et F_2 peuvent varier, indépendamment l'une de l'autre, entre 15 et $25ms$ (temps réel absolu) ;
- gigues : les fonctions F_1 et F_2 peuvent être victime d'un phénomène de gigue non déterministe à l'intérieur de leur période ; on supposera en revanche que ces fonctions respectent leurs échéances,

i.e., terminent avant leur prochain top d'horloge quelle que soit cette gigue ;
- latences d'exécution : F_1 et F_2 ont des durées d'exécution non nulles et variables ; en revanche, nous supposerons qu'elles respectent leurs échéances, c'est-à-dire terminent avant leur prochaine période.

Nous supposerons en revanche que le temps de diffusion du signal Cmd_1 de C_1 vers C_2 via le bus numérique est négligeable.

Enfin, par souci de simplicité, nous considérerons que les défaillances des fonctions et des calculateurs sont définitives, c'est-à-dire sans possibilité de réparation.

4.2. Formalisation LUSTRE de l'application et de son asynchronisme

Le problème consiste donc à étudier la robustesse de l'application à l'asynchronisme de son architecture, et plus précisément à démontrer la validité (ou l'invalidité) des exigences en présence de dérives, de gigues, et de latences d'exécution telles que quantifiées ci-dessus.

Pour ce faire, nous suivons la démarche proposée par P. Caspi *et al.* [7] et généralisée dans les sections précédentes. La première étape de cette démarche consiste à modéliser le comportement global de l'application (en tenant compte de son asynchronisme) dans le langage synchrone à flots de données LUSTRE.

4.2.1. Flots et horloges de l'application

La première phase de cette modélisation consiste à identifier les variables d'état du système. Notons :

- `defail_Fi` le flot booléen qui caractérise l'apparition d'une défaillance sur F_i $(i = 1, 2)$,
- `Ht_Fi` l'horloge (éventuellement "dérivante") de F_i $(i = 1, 2)$,
- `Hg_Fi` l'horloge effective de F_i prenant en compte les possibilités de gigue $(i = 1, 2)$,
- `Hter_Fi` l'horloge de terminaison de F_i, c'est-à-dire la suite des instants où sont produites les données calculées par F_i $(i = 1, 2)$.

Ces flots booléens peuvent être considérées comme les variables "libres" du système, *i.e.*, les variables dont l'évolution est seulement contrainte par des assertions modélisant l'asynchronisme de l'architecture. Elles modélisent l'impact sur l'application de l'architecture physique et de son asynchronisme.

Considérons ensuite les flots :

- `H_Cmdi` l'horloge d'émission (effective) de la commande Cmd_i $(i = 1, 2)$,
- `numCmd1` la commande numérique produite par F_1 (abstraite par un booléen)
- `analCmdi` la commande analogique produite par F_i $(i = 1, 2)$,
- `fresh_Cmd1_in_F2` le "bit de refresh" du message Cmd_1 en entrée de C_2, c'est-à-dire un flot booléen vrai si et seulement si la valeur de Cmd_1 disponible en entrée de C_2 a été rafraîchie depuis sa dernière consommation par F_2.

4.2.2. Abstraction du comportement des fonctions

L'application est composée de deux fonctions dont il faut abstraire et définir le comportement sous la forme d'équations LUSTRE.

La première fonction produit, en l'absence de toute défaillance, le signal `numCmd1` lors de chacune de ses activations (donc de façon synchrone avec son horloge de terminaison `Hter_F1`). Le signal `numCmd1` est donc synchrone avec l'horloge `H_Cmd1` définie par :

```
H_Cmd1 = Hter_F1 and never(defail_F1);
```

La seconde fonction produit, en l'absence de toute défaillance, le signal numCmd2 lors de chacune de ses activations si aucune occurrence de l'ordre Cmd_1 émis par F_1 n'a été reçue sur le calculateur C_2 depuis 5 tops de l'horloge effective de F_2 (*i.e.*, Hg_F2). L'horloge d'émission de numCmd2 est donc par définition :

```
H_Cmd2 = Hter_F2 and never(defail_F2) and
         current(not une_fois_au_moins_depuis(5, fresh_Cmd1_in_F2 when Hg_F2));
```

où fresh_Cmd1_in_F2 est un flot booléen caractérisant la "fraîcheur" de l'information Cmd_1 en entrée de C_2, c'est-à-dire vrai depuis chaque émission de Cmd_1 (par F_1) jusqu'à sa consommation par F_2 (*i.e.* lors de son activation) et faux hors de ces plages temporelles :

```
fresh_Cmd1_in_F2 = H_Cmd1 -> if H_Cmd1 then true
                             else if pre(Hg_F2) then false
                             else pre(fresh_Cmd1_in_F2);
```

D'autre part, la vérification des exigences P1 et P2 ne nécessitant pas la connaissance des valeurs concrètes de la commande Cmd_i $(i = 1, 2)$, celle-ci sera abstraite par la valeur true lorsque le signal est produit, et false sinon. Le flot numCmdi sera donc constamment vrai (puisque synchrone avec son horloge d'émission).

```
numCmd1 = true when H_Cmd1;
numCmd2 = true when H_Cmd2;
```

En revanche, par construction chaque calculateur C_i maintient en permanence, en l'absence de défaillance, la dernière valeur de la commande calculée par F_i sur la ligne analogique (à partir du premier instant où cette commande est disponible). Le flot analCmdi, qui dénote cette valeur analogique, est donc considéré vrai depuis la première production d'une commande par F_i jusqu'à la défaillance de cette dernière, et faux hors de cette plage :

```
analcmd1 = if never(defail_F1) then current_Cmd1 else false;
current_Cmd1 = if never(H_Cmd1) then false else current(numCmd1);
analCmd2 = if never(defail_F2) then current_Cmd2 else false;
current_Cmd2 = if never(H_Cmd2) then false else current(numCmd2);
```

Ce qui modélise le comportement (abstrait) des fonctions F_1 et F_2.

4.2.3. Caractérisation de l'asynchronisme

Il reste alors à formaliser l'asynchronisme de l'architecture du système sous la forme d'un ensemble de contraintes sur les horloges des fonctions F_1 et F_2.

Chaque fonction F_i est rythmée par l'horloge de son calculateur Ht_Fi. Or, par hypothèse d'asynchronisme, les périodes de ces horloges varient entre 15 et $25ms$ (temps réel absolu). En conséquence :

- deux tops consécutifs de Ht_Fi ne peuvent être séparés que par au plus deux tops de Ht_Fj $(i \neq j)$,
- trois tops consécutifs de Ht_Fi ne peuvent être séparés que par au plus quatre tops de Ht_Fj $(i \neq j)$,
- et enfin quatre tops consécutifs de Ht_Fi ne peuvent être séparés que par au plus cinq tops de Ht_Fj $(i \neq j)$.

Ce qui s'exprime par les assertions :

```
assert(au_plus_N_dans_2(2, Ht_F1, Ht_F2)
       and au_plus_N_dans_3(4, Ht_F1, Ht_F2)
       and au_plus_N_dans_4(5, Ht_F1, Ht_F2));
```

```
assert(au_plus_N_dans_2(2, Ht_F2, Ht_F1)
       and au_plus_N_dans_3(4, Ht_F2, Ht_F1)
       and au_plus_N_dans_4(5, Ht_F2, Ht_F1));
```

De plus, chaque fonction F_i peut être victime d'un phénomène de gigue. L'horloge effective d'activation de F_i, notée `Hg_Fi`, peut donc être différente de l'horloge du calculateur `Ht_Fi`, mais est reliée à cette dernière par le prédicat `alternance` :

```
assert(alternance(Ht_F1, Hg_F1));
assert(alternance(Ht_F2, Hg_F2));
```

Enfin, les fonctions F_i ayant des latences d'exécution non nulles, les instants de terminaison de ces fonctions, dénoté par les horloges `Hter_Fi` ($i = 1, 2$) sont également contraints par des relations d'alternance :

```
assert(alternance(Ht_F1, Hter_F1) and alternance(Hg_F1, Hter_F1));
assert(alternance(Ht_F2, Hter_F2) and alternance(Hg_F2, Hter_F2));
```

Ce qui modélise l'asynchronisme interne de l'application.

4.3. Vérification

Le système et son asynchronisme ayant été formalisé en LUSTRE, il est maintenant possible d'étudier sa correction vis-à-vis des exigences P1 et P2. L'approche utilisée consiste à exprimer ces propriétés sous la forme d'observateurs LUSTRE [10] insérés dans le système à vérifier, puis explorer le graphe des états de ces observateurs à l'aide de l'outil de vérification de modèles LESAR [11].

La première exigence exprime une propriété d'exclusion entre F_1 et F_2 : F_1 et F_2 ne doivent jamais maintenir simultanément un ordre (électrique) sur la ligne analogique à destination de la gouverne ; propriété d'exclusion qui peut s'écrire en LUSTRE par :

```
P1 = not (analCmd1 and analCmd2);
```

La seconde exigence exprime une propriété de vivacité bornée ; la gouverne doit être commandée au moins une fois pendant tout intervalle temporel de $175ms$ (temps réel). En d'autres termes, deux occurrences consécutives d'ordre de gouverne (quelqu'en soit l'émetteur) ne peuvent être séparées par un intervalle de temps supérieur à $175ms$. Or, en remarquant que compte tenu des hypothèses de dérives d'horloges 7 périodes de `Ht_F1` ou `Ht_F2` durent au plus $175ms$, cette exigence peut alors s'exprimer sous une forme qui ne fait plus référence au temps absolu :

```
P2 = au_plus_N_dans_2(6, Ht_F1, H_Cmd1 or H_Cmd2)
     or au_plus_N_dans_2(6, Ht_F2, H_Cmd1 or H_Cmd2);
```

c'est-à-dire, deux occurrences consécutives d'ordre de gouverne doivent être séparées par au plus 6 tops de l'horloge `Ht_F1` ou 6 tops de l'horloge `Ht_F2` (6 tops = 7 périodes).

Nous supposerons en outre que la défaillance des deux fonctions F_1 et F_2 est un événement extrêmement improbable. Nous nous placerons donc dans l'hypothèse où une seule défaillance est possible durant la vie du système, ce qui s'exprime par l'assertion :

```
assert(never(defail_F1) or never(defail_F2));
```

Sous cette hypothèse, on peut démontrer à l'aide de l'outil LESAR que les deux propriétés P1 et P2 sont satisfaites. L'automate construit et exploré contient 76758 états et 929983 transitions, et nécessite un durée de vérification voisine de 4 minutes sur une station de travail Unix du type Sun Ultra-5_10 disposant d'une mémoire de 256 Mo.

De même, on peut démontrer à l'aide de LESAR qu'une diminution de la durée d'attente de Cmd_1 en entrée de F_2 (par exemple 4 cycles de F_2 au lieu de 5) falsifie P1, alors qu'une augmentation de cette attente (par exemple 6 cycles au lieu de 5) invalide P2.

5. Conclusion et perspectives

L'objet de cet article était de proposer une abstraction de la notion d'asynchronisme dans les systèmes distribués intégrés, tels que les systèmes embarqués, puis d'en proposer une formalisation dans le langage synchrone Lustre. L'intérêt d'une telle démarche est de permettre la vérification formelle et globale de systèmes distribués asynchrones par des techniques d'analyse de modèles [11], ou par des techniques reposant sur la génération et le jeu de scénarios de test [4, 9].

Cependant, une extension de l'exemple traité ci-dessus (composée de trois gouvernes, chacune étant commandée en mode nominal par un calculateur différent) se heurte au problème de l'explosion combinatoire, inhérent aux techniques de vérification de modèles, et amplifié ici par la modélisation de comportements asynchrones non déterministes. Or, la technique de modélisation de l'asynchronisme en Lustre, proposée par P. Caspi *et al.* et reprise ici, procède fondamentalement par expression de contraintes. Une seconde approche possible consisterait alors à explorer la voie de la programmation par contrainte et de la vérification par des techniques de satisfaction de contraintes. Cette voie est en cours d'exploration et semble, sur l'exemple précédent, éviter le mur de l'explosion combinatoire. Un second intérêt escompté serait de permettre, par une approche générative et non plus évaluative, de calculer l'asynchronisme maximal tolérable vis à vis d'un ensemble de propriétés à satisfaire. Cette approche est en cours d'investigation.

Remerciements

Les auteurs tiennent à remercier F. Pilarski et R. Mercadié de Aerospatiale, ainsi que F. Faurre et Y. Le Biannic de Dassault-Aviation pour les nombreuses discussions qui ont alimenté ce travail.

Références

[1] C. André, et H. Boufaïed. Vérification de comportement de machine d'exécution. *2ème Congrès sur la Modélisation des systèmes réactifs, MSR'99*, pages 121-130, Cachan, France, 1999.

[2] A. Benveniste, B. Caillaud, et P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code geenration. *Information and Computation*, 1999. à paraître.

[3] A. Benveniste, B. Caillaud, et P. Le Guernic. From synchrony to asynchronoy. *10th International Conference on Concurrency Theory, CONCUR'99*, Lecture Notes in Computer Science, vol. 1664, pages 162–177, 1999.

[4] L. du Bousquet, F. Ouabdesselam, J.L. Richier, et N. Zuanon. Environnement pour le test d'applications synchrones. *2ème Congrès sur la Modélisation des systèmes réactifs, MSR'99*, pages 51-60, Cachan, France, 1999.

[5] P. Caspi, N. Halbwachs, P. Pilaud, et P. Raymond. Programmation et vérification des systèmes réactifs : le langage Lustre. *Techniques et Sciences de l'Informatique*, vol. 10(2), pages 139-158, 1991.

[6] P. Caspi, N. Halbwachs, P. Pilaud, et P. Raymond. The synchronous dataflow programming language Lustre. *Proceedings of IEEE, Another Look at Real-time programming*, vol. 79(9), pages 1305-1319, 1991.

[7] CRISYS. First theoritical results. Rapport de recherche, projet CRISYS - EP 25 514, novembre 1998.

[8] O. Hainque, L. Pautet, Y. Le Biannic, et E. Nassor Cronos: a separate compilation toolset for modular Esterel Applications. *World Congress on Formal Methods in the Development of Computing Systems, FM'99*, Lecture Notes in Computer Science vol. 1709, pages 1836-1853, Toulouse, France, septembre 1999.

[9] P. Raymond, D. Weber, X. Nicollin, et N. Halbwachs Automatic Testing of Reactive Systems. *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998.

[10] N. Halbwachs, F. Lagnier, et P. Raymond Synchronous observer and the verification of reactive systems. *Third International Conference on Algebraic Methodology and Software Technology, AMAST'93*, Twente, 1993.

[11] C. Ratel. *Définition et réalisation d'un outil de vérification formelle de programmes Lustre : le système LESAR.* Thèse de doctorat, Institut National Polytechnique de Grenoble, juillet 1992.