Répartition de programmes synchrones temps réel

Rym Salem Habermehl

▶ **To cite this version:**

Rym Salem Habermehl. Répartition de programmes synchrones temps réel. Autre [cs.OH]. Université Joseph-Fourier - Grenoble I, 2001. Français. NNT : . tel-00004703

HAL Id: tel-00004703
https://theses.hal.science/tel-00004703

Submitted on 17 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ JOSEPH FOURIER – GRENOBLE 1
SCIENCES ET GÉOGRAPHIE

THÈSE

pour obtenir le grade de
DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER

Discipline : Informatique

présentée et soutenue publiquement par

Rym SALEM HABERMEHL

le 30 octobre 2001

Répartition de Programmes Synchrones Temps Réel

Composition du jury :

Président	**F.**OUABDESSELAM
Rapporteurs	**A.**BENVENISTE
	C.ANDRÉ
Directeur de thèse	**P.**CASPI
Examinateur	**H.**LE BERRE

Table des matières

1 Introduction **7**
1.1 L'approche synchrone 7
1.2 La répartition 8
1.3 Problèmes posés 8
1.4 Plan de lecture 8

I Programmation synchrone : le temps logique **11**

2 Généralité sur les langages synchrones **13**
2.1 Les besoins du temps réel 13
2.2 Langages et systèmes d'exploitation temps réel 14
2.3 Langages synchrones 14
2.4 Intérêt des langages synchrones 15

3 Le Langage Synchrone LUSTRE **17**
3.1 Introduction 17
3.2 Eléments de base du langage 18
3.2.1 Aspect flot de données 18
3.2.2 Abstraction synchrone 18
3.2.3 Opérateurs Usuels 18
3.2.4 Opérateurs Temporels 18
3.3 Principes de structuration 19
3.4 Principes sémantiques 20
3.5 Echantillonnages multiples 20
3.5.1 Bloqueur d'ordre 0 21
3.5.2 Notion d'horloge 21
3.6 Assertions 22

4 Exemple de spécification et de vérfication formelles **23**
4.1 Introduction 23
4.2 Spécification 23
4.2.1 L'environnement physique 23

4.2.2 La modélisation de l'environnement 24
4.2.3 Propriétés désirées 26
4.3 Le programme de commande 29
4.4 Le programme de verification 31
4.5 L'outil de vérification Lesar 31

5 Répartition en temps logique 35
5.1 Introduction 35
5.2 Réseaux de Kahn 35
5.2.1 Sémantique de Kahn 36
5.2.2 Lustre et sémantique de Kahn 37
5.2.3 Lustre réparti 38
5.3 Signal et la répartition 38
5.3.1 La sémantique de Signal 38
5.3.2 Les horloges de Signal 39
5.3.3 Exécutions asynchrones 39
5.3.4 Comparaison avec Lustre 40

II Programmation synchrone : le temps physique 41

6 Architectures temps réel 43
6.1 Evolution des pratiques 43
6.2 Intérêt de cette architecture 43

7 Limites de l'abstraction synchrone 45
7.1 Introduction 45
7.2 Vérification synchrone et temps réel 45
7.3 Conclusion 47

8 Modélisation du temps réel 49
8.1 Modélisation mathématique 49
8.1.1 Définitions de base 49
8.1.2 Redatation 49
8.1.3 Redatation et échantillonnage 50
8.1.4 Systèmes et signaux uniformément continus 50
8.1.5 Signaux à variabilité uniformément bornée 51
8.2 Modélisation en Lustre 52
8.2.1 Echantillonnage 52
8.2.2 Variabilité uniformément bornée 52
8.2.3 Délais non déterministes bornés 52
8.3 Application à l'exemple 54
8.3.1 Commande échantillonnée 54

8.3.2 Commande échantillonnée avec incertitude en entrée et sortie 56

III Répartition en temps physique 59

9 Modélisation quasi-synchrone 61
9.1 L'architecture quasi-synchrone 61
9.2 Propriétés de l'architecture 63
9.2.1 Equité bornée 63
9.2.2 Délai de communication borné 64
9.3 Simulation de l'architecture 64
9.3.1 Modélisation de la communication 64
9.3.2 Modélisation des horloges quasi-périodiques 65
9.3.3 Preuve du délai borné 65
9.4 Programmes quasi-synchrones 66

10 Systèmes continus et combinatoires 69
10.1 Systèmes continus 69
10.1.1 Systèmes uniformément continus 69
10.1.2 Systèmes en boucle fermée 70
10.2 Retards incohérents 70
10.3 Variabilité uniforme 72
10.4 Quelques propriétés importantes 72
10.5 Fonctions de confirmation 73
10.5.1 Preuve du théorème 10.5.1 75
10.6 Exemple 76

11 Systèmes séquentiels robustes 77
11.1 Introduction 77
11.2 Vérification de la robustesse des systèmes séquentiels 78
11.2.1 Robustesse syntaxique 78
11.2.2 Robustesse semi-sémantique 78
11.2.3 Robustesse sémantique 79
11.3 Un outil de vérification de la robustesse 79
11.4 Application à l'exemple 80

12 Systèmes séquentiels non robustes 83
12.1 Introduction 83
12.2 Protocole de répartition 83
12.2.1 Algorithme de synchronisation 83
12.3 Preuve de l'algorithme 85

13 Construction de systèmes robustes **87**
13.1 Introduction 87
13.2 Descriptions des changements 87
13.3 Absence de courses critiques 88
13.3.1 Condition nécessaire et suffisante 88
13.3.2 Systèmes vérifiant cette condition 88
13.4 Construction de systèmes robustes avec des flip-flops 88
13.4.1 Chaînes de causalité 89
13.5 Application à l'exemple 90

IV Tolérance aux fautes **97**

14 Tolérance aux fautes **99**
14.1 Introduction 99
14.2 Votes à seuils 99
14.3 Voteur à délais bornés 100
14.3.1 Voteur simple 100
14.3.2 Cas des tuples 101
14.4 Voteurs pour fonctions séquentielles 102
14.4.1 Approche générale 102
14.4.2 Voteur 2/2 pour fonctions séquentielles 103
14.4.3 De la détection de fautes à la tolérance aux fautes 105
14.5 Conclusion 107

15 Conclusion **109**

Chapitre 1

Introduction

Nos travaux se sont déroulés dans le cadre du projet CRISYS. Ce projet Esprit a réuni :

- des industriels utilisateurs :
  - Schneider Electric (coordinateur du projet),
  - Aérospatiale,
  - Siemens Electrocom,
  - Elf Aquitaine,
- un organisme lié à la certification, le CEA,
- des industriels pourvoyeurs d'outils :
  - Verilog, BSSE,
- des laboratoires universitaires :
  - GMD (équipe d'Axel Poigné),
  - Laboratoire d'automatique de Grenoble (H. Alla et R. David),
  - Verimag.

pour étudier les problèmes relatifs à l'utilisation de l'approche synchrone dans la programmation de systèmes de contrôle-commande répartis.

1.1 L'approche synchrone

La programmation synchrone est depuis longtemps utilisée pour faciliter la description des systèmes réactifs, devant réagir de façon continue à leur environnement physique. La classe des systèmes réactifs comprend certains systèmes industriels dits temps réels tels que les systèmes de contrôle-commande, automatismes, systèmes de surveillance de processus... Outre leur aspect réactif, ces applications présentent deux aspects importants, à savoir le parallélisme et la sûreté de fonctionnement.

La programmation synchrone offre des primitives idéalisées pour la concurrence et la communication. Un programme synchrone consulte périodiquement l'environnement, lit le jeu d'entrées et est supposé calculer les sorties correspondantes avant

la prochaine consultation de l'environnement. La séquence d'instants de consultation de l'environnement constitue ainsi une échelle de temps discrète. Entre deux instants consécutifs de cette échelle l'état du programme ne change pas. Cette échelle est donc en plus globale et est partagée par toutes les activités parallèles. Les raisonnements temporels sont alors plus faciles. En particulier, on peut exprimer une propriété de sûreté de fonctionnement caractérisant chaque instant de cette échelle.

1.2 La répartition

Certains systèmes réactifs tels que les automatismes et les systèmes de contrôle-commande utilisent des détecteurs et des activateurs qui sont physiquement répartis. L'exécution synchrone centralisée de programmes spécifiant de tels systèmes est donc difficile puisque l'hypothèse de synchronisme ne peut être facilement satisfaite. La répartition du calcul s'avère donc nécessaire. Un programme synchrone centralisé est décomposé en plusieurs sous-programmes s'exécutant chacun de façon synchrone avec une horloge indépendante des autres.

1.3 Problèmes posés

Le système synchrone centralisé constitue un modèle du système réparti qu'on souhaite obtenir. Dans ce modèle sont définis le comportement du système, les aspects du calcul et de communication. Tous ces aspects sont basés sur l'existence d'une échelle unique de temps. D'autre part, toute exécution répartie se caractérise essentiellement par l'absence d'une échelle unique de temps à priori. Pour obtenir cette échelle, il faut synchroniser les différentes parties du système réparti.

Dans le modèle logique, une bonne répartition est validée par rapport à la satisfaction de la propriété de sûreté en temps logique. Cette abstraction peut induire en erreur puisque la propriété peut être violée en temps réel. Il est donc indispensable de voir la validité de cette abstraction en temps réel. Ceci permet de définir le comportement souhaité du système réparti. Il faut ensuite décrire les conditions de l'exécution répartie en tenant compte du temps réel ainsi que les synchronisations nécessaires pour avoir le comportement souhaité.

1.4 Plan de lecture

Dans la première partie, nous présentons l'intérêt des langages synchrones en général et nous donnons un aperçu sur la syntaxe et la sémantique du langage Lustre. Ensuite, nous montrons sur un exemple l'utilisation de Lustre pour la spécification et la vérification. Les résultats ultérieurs seront illustrés sur cet exemple. Enfin, nous présentons les résultats antérieurs sur la répartition en temps logique.

Dans la deuxième partie, nous nous intéressons au temps physique. En particulier :

- le chapitre 6 rappellera les architectures supportant la programmation synchrone ainsi que le principe de l'abstraction synchrone,
- dans le chapitre 7, la validité de l'abstraction synchrone est remise en cause ; nous montrons sur deux exemples ses limites et nous introduisons les qualités requises sur l'environnement pour que la validation synchrone soit aussi valable en temps réel,
- dans le chapitre 8, nous présentons formellement ces qualités sur l'environnement, nous en présentons une modélisation mathématique puis en Lustre et nous illustrons ceci sur l'exemple du chapitre 4,

La répartition en temps réel est ensuite abordée dans la troisième partie. Une bonne répartition en temps réel suppose que la propriété de sûreté est valide en temps réel.

Dans le chapitre 9, nous présentons l'architecture quasi-synchrone, décrivant la synchronisation des différents calculateurs synchrones ainsi que leurs communications.

Dans les chapitres 10 et 11, nous présentons des systèmes particuliers n'ayant pas besoin d'une grande synchronisation pour être répartis. Dans le chapitre 12, nous donnons une solution générale pour la répartition et nous décrivons le protocole nécessaire pour la synchronisation.

Dans le chapitre 13 nous donnons une méthode pour construire en Lustre des programmes robustes pour la validation synchrone ainsi que la répartition.

Dans le chapitre 14 nous présentons les résultats sur la tolérance aux fautes.

Première partie

Programmation synchrone : le temps logique

Chapitre 2

Généralité sur les langages synchrones

2.1 Les besoins du temps réel

Il a été reconnu, depuis longtemps que le domaine du temps réel doit remplir les besoins suivants entre autres :

Rendre compte du parallélisme. Il y a deux raisons principales pour cela : d'une part, les programmes tournent en parallèle avec les environnements qu'ils commandent ou surveillent ; nous en verrons des exemples au chapitre 4. L'analyse et la synthèse de ces programmes demande donc, sous une forme ou une autre, un modèle exécutable de cet environnement. Donc, le langage de programmation doit fournir un moyen de rendre compte de ce parallélisme. D'un autre côté,, souvent cet environnement comporte plusieurs degrés de liberté qu'il faut commander en même temps. Par exemple, dans un programme de pilotage d'avion, il faut commander en même temps le roulis et le tangage. C'est donc une autre raison pour laquelle le langage doit pouvoir rendre compte du parallélisme.

Apporter des garanties de bornes sur la mémoire et le temps d'exécution. Beaucoup de systèmes présentent des contraintes temps réel durs et sont des systèmes critiques. Il est donc important que le langage présente des caractéristiques permettant de faciliter ces contrôles.

Permettre la répartition. Enfin, beaucoup de ces systèmes sont des systèmes répartis, pour des raisons évidentes de charge, de localisation des capteurs et actionneurs et de sûreté de fonctionnement (redondance).

```
initialize state;

loop each input event

    read other inputs;
    compute outputs and state;
    emit outputs

end loop
```

TAB. 2.1 – Programme synchrone

2.2 Langages et systèmes d'exploitation temps réel

A la fin des années soixante-dix, de nombreuses propositions visant à essayer de remplir ces exigences ont été établies par des informaticiens. Ces propositions ont été, souvent, fondées sur des principes issus des systèmes d'exploitation en temps partagé, c'est à dire sur des concepts structurants tel que :

- Synchronisation : sémaphores, moniteurs, processus séquentiels…
- Communication : mémoire partagée, messages, boites aux lettres…
- Synchronisation + communication : files d'attente, rendez-vous

Ce sont ces mêmes concepts que l'on trouve dans les propositions temps réel telles que CSP [21], OCCAM, la partie « tasking » de ADA et les nombreuses propositions de systèmes d'exploitation temps réel.

2.3 Langages synchrones

Pendant ce temps, les ingénieurs du domaine élaboraient leurs propres concepts et outils que nous verrons aux chapitres 6 et 9. Ce type de pratiques a donné naissance à *l'école de programmation synchrone*. Les activités de celle-ci ont consisté essentiellement en :

- Définir une structure de code objet du type de celle illustrée à la table 2.1.
- Définir plusieurs styles de code source : flot de donnée [17, 4], impératif [7], graphique [20, 27, 2]. Tous ces codes source donnent des codes objet semblables et ont en commun de proposer sous une forme ou une autre une construction parallèle remplissant donc une des exigences établies en 2.1. Il faut aussi remarquer que, pour pouvoir fournir un code objet comme celui montré par la table 2.1, cette construction parallèle doit être compilée et non interprétée comme dans les propositions asynchrones vues en 2.2.
- Equiper cette approche de compilateurs efficaces et d'outils formels [18, 8].

2.4 Intérêt des langages synchrones

Outre le fait de proposer des constructions parallèles et de faciliter, grâce à la structure du code objet la vérification des contraintes temps réel, la programmation synchrone a d'autres avantages qui ont été en particulier soulignés par Gérard Berry [7] :

- Le parallélisme synchrone est déterministe et produit moins d'états que le parallélisme asynchrone. Cela a deux conséquences : les programmes sont plus déterministes et plus faciles à mettre au point et à tester, et le problème d'explosion d'états rencontré lors de la vérification formelle est moindre.
- Le parallélisme synchrone fournit une notion « naturelle » de temps logique qui facilite les raisonnements temporels.

Tous ces intérêts font que ce type d'approche est très répandu en pratique, même si ces pratiques ne font pas toutes référence à l'approche synchrone.

Chapitre 3

Le Langage Synchrone LUSTRE

3.1 Introduction

Au début des années 80, plusieurs projets de contrôles-commandes de systèmes critiques ont vu le jour (commandes de vol électriques, contrôles de centrales nucléaires, métros automatiques, etc...) et ont posé le problème du choix de leur mise en œuvre cablée ou programmée. Si souvent les concepteurs ont choisi de conserver la technologie traditionnelle, certains ont osé franchir le pas et anticiper le prodigieux développement de l'usage des ordinateurs en temps-réel. Ce faisant, plusieurs problèmes se posaient à eux :

- Comment récupérer le savoir-faire issu des réalisations passées ?
- Comment se convaincre et convaincre les autorités de certification que leurs conceptions étaient correctes ou, au moins, suffisamment fiables ?
- Comment réduire les coûts de développement, de validation, et d'évolution ?
- Plus généralement, comment profiter des progrès des technologies logicielles, tout en respectant les contraintes opérationnelles nombreuses dues aux performances limitées des ordinateurs de l'époque ?

Ces questions ont reçu des réponses diverses. Le langage Lustre [17] a été conçu à cette époque comme une réponse possible à ces questions. Sa conception s'est fondée sur les principes suivants :

- s'inspirer des méthodes et outils de conception du domaine de façon à récupérer les savoir-faire et faciliter la communication entre les intervenants des projets,
- en y incorporant de solides bases d'informatique, garanties de consistance et autorisant les optimisations et même les validations formelles,
- et surtout en adoptant une approche langage plus que modèle, c'est-à-dire en cherchant à éviter la production manuelle de code, propice à erreur, au profit d'une véritable compilation, gage à la fois de sécurité et d'économie.

3.2 Eléments de base du langage

L'idée a été de s'inspirer des formalismes usuels en automatique et traitement de signal.

3.2.1 Aspect flot de données

Tous les objets de base (constantes, variables, expressions) représentent des signaux échantillonnés, que l'on appelle « flots », c'est-à-dire des suites, ou de façon équivalente des fonctions d'entiers naturels. Ainsi la constant `true` est la fonction qui vaut *true* pour tout entier naturel n

$$\forall n \in N : \texttt{true}(n) = true$$

3.2.2 Abstraction synchrone

En réalité, l'indice n est une abstraction du nombre nT, où T est la longueur de la période synchrone. Ainsi, chaque occurence d'un signal X à l'instant physique $n * T$ est représentée de façon abstraite par l'occurence d'ordre logique n.

$$\forall n \in N : \texttt{true}(n) = true \;\; signifie \;\; \texttt{true}(n * T) = true$$

Dans la suite, on utilisera toujours cette abstraction des indices.

3.2.3 Opérateurs Usuels

Les opérateurs arithmétiques et logiques usuels s'étendent à ces suites.

Exemple 3.2.1 Opération de choix

$$(\textbf{if } c \textbf{ then } x \textbf{ else } y)(n) = \textbf{if } c(n) \textbf{ then } x(n) \textbf{ else } y(n)$$

3.2.4 Opérateurs Temporels

Il y a deux opérateurs temporels de base, l'initialisation $->$ et le retard `pre`, dont la composition produit le retard initialisé des schémas échantillonnés.

$$\begin{aligned} (\texttt{X -> Y})(0) &= \texttt{X}(0) \\ \forall n \in N : (\texttt{X -> Y})(n+1) &= \texttt{Y}(n+1) \end{aligned}$$

$$\begin{aligned} (\texttt{pre X})(0) &= nil \\ \forall n \in N : (\texttt{pre X})(n+1) &= \texttt{X}(n) \end{aligned}$$

où nil a le sens d'un indéfini.

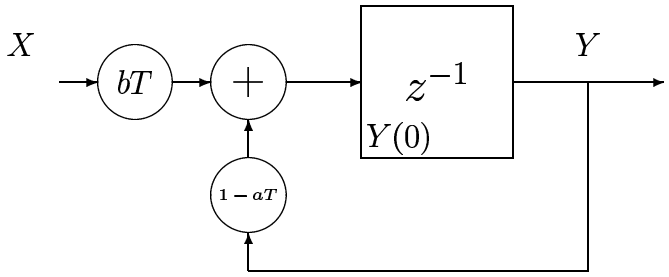


FIG. 3.1 – Un schéma échantillonné

On peut alors former des équations décrivant des schémas tels que celui de la figure 3.1 :

$$Y = Y0 -> \texttt{pre}(b * T * X + (1 - a * T) * Y);$$

qui traduit fidèlement le schéma. De telles équations sont les constructions de base du langage, l'équivalent des instructions dans un langage informatique classique.

3.3 Principes de structuration

Comme dans un langage classique, ces équations sont regroupées en paquets, appelés nœuds qui délimitent des interfaces et des portées de visibilité de noms de flots et qui permettent de déclarer les types des flots. Par exemple, un filtre linéaire de deuxième ordre pourra s'écrire :

```
const a, b, c, d, e : real;
node filter ( x : real)
     returns( y : real);
     var u,v: real;
     let
         y =  a*x + v;
         v =  0.0 -> pre(u + b*x - d*y);
         u =  0.0 -> pre(c*x - e*y);
     tel
```

Ce texte décrit d'abord des constantes et leurs types. Ensuite il décrit un nœud `filter` par ses flots d'entrée (`x`), ses flots de sortie (`y`) et ses flots locaux (`u,v`) avec leurs types puis par un ensemble d'équations.

3.4 Principes sémantiques

Cependant, il faut remarquer que ces équations sont en fait des définitions : le flot `y` est une suite infinie de valeurs complètement définie par cette équation. Cela veut dire

que `y` ne pourra pas être redéfini ailleurs dans la portée de cette définition (*principe de définition unique*). De ce fait, les sorties et les locaux d'un nœud doivent avoir une et une seule équation de définition.

De même, l'ordre dans lequel sont écrites plusieurs définitions dans une même portée est indifférent. Le langage est donc *déclaratif*.

Comme corollaire du principe de définition, on trouve le *principe de substitution* : toute variable peut être substituée par sa définition dans toute expression où elle apparaît dans sa portée. Ce puissant principe, inhabituel dans les langages impératifs, autorise de nombreuses manipulations formelles de code, à des fins d'optimisation ou de vérification.

Enfin, un nœud définit une fonction de ses flots d'entrée vers ses flots de sortie. Cette fonction pourra donc être utilisée pour former d'autres équations dans d'autres nœuds (la récursivité est interdite). Par exemple on pourra écrire :

```
z = filter(filter(u));
```

On a ainsi un langage *fonctionnel*.[1]

3.5 Echantillonnages multiples

Ce qui précède ne permet, pour le moment, que de décrire des systèmes monocadencés. Or, les automatismes sont souvent constitués de sous-systèmes ayant des dynamiques différentes. Comment prendre en compte cela ? Une solution aurait pu consister à introduire des diviseurs de fréquence, dans une optique « temps-réel ». Par souci de généralité et d'orthogonalité, un opérateur d'échantillonnage général sous condition, `when`, a été introduit qui va permettre toutes sortes d'échantillonnages, aussi bien réguliers qu'irréguliers. Son comportement est résumé par le tableau suivant :

`c`	t	f	t	t	f	$\ldots$
`x`	x_0	x_1	x_2	x_3	x_4	$\ldots$
`x when c`	x_0		x_2	x_3		$\ldots$

L'idée est que `when` efface du flot `x` les éléments correspondant aux indices où `c` vaut `false`. De ce fait, le flot `x when c` se trouve « échantillonné » par rapport au flot `x`.

Or, les expressions Lustre sont, on l'a vu,[2] des fonctions de flots. Cela signifie qu'une fonction alimentée par un flot plus lent travaillera effectivement plus rarement et produira des flots eux-mêmes plus rares.

Par exemple, on peut faire travailler notre filtre à fréquence moitié en écrivant le système d'équations :

[1] quoique réduit au premier ordre.

[2] Il est facile de se convaincre que `when` est aussi une fonction.

```
y    = filter(x when half);
half = true -> pre (not half);
```

En effet, `half` est un booléen qui n'est vrai qu'une fois sur deux. Le flot `x when half` est donc bien deux fois plus rare que `x`.

3.5.1 Bloqueur d'ordre 0

Il arrive aussi qu'un processus rapide ait besoin de données issues d'un processus plus lent. Une opération inverse, permet cette communication et correspond au diagramme suivant :

`c`	t	f	t	t	f	...
`x`	x_0	x_1	x_2	x_3	x_4	...
`x when c`	x_0		x_2	x_3		...
`current (x when c)`	x_0	x_0	x_2	x_3	x_3	...

On voit en effet comment elle remplit les trous du flot échantillonné par les valeurs qu'il avait précédemment.

3.5.2 Notion d'horloge

Contrairement aux autres primitives, `current` n'est pas une vraie fonction : en effet, elle ne reçoit que le flot échantillonné et donc ne connaît pas a priori les trous qu'elle doit combler. Ceci s'explique par la notion d'horloge. Le besoin de cette notion tient au fait que l'on ne peut pas librement mélanger des flots échantillonnés différemment. Considérons, par exemple, l'expresssion :

```
x + (x when half)
```

Il est clair que cette expression ne peut être calculée en temps réel pendant longtemps car `+` consomme un élément de chacune de ses entrées à chaque pas, alors que `x when half` est deux fois plus rare que `x`. Les éléments de `x` vont donc s'accumuler inexorablement et finir par déborder.

Pour éviter ce phénomène, il faut donc bannir des expressions telles que celle-là et convenir que les deux entrées de `+` ne doivent pas être filtrés différemment. On dira qu'elles doivent avoir la *même horloge*.

Mais, pour être sûre, cette restriction doit pouvoir être décidée statiquement, c'est-à-dire à la compilation. Cela veut dire que l'on doit pouvoir associer, statiquement, une horloge à chaque flot. On peut alors considérer que chaque flot porte en lui son horloge ou, si l'on veut, que la fonction `clock(x)` est définie à la compilation. De ce fait, `current` est bien, maintenant, une fonction car elle sait qu'elle doit combler les trous par rapport à l'horloge de son entrée.

3.6 Assertions

Il arrive souvent, en automatique, que l'on ait à considérer des hypothèses sur les entrées d'une fonction. C'est le cas, par exemple, lorsqu'on considère un système en boucle fermée : les entrées ne sont pas libres mais dépendent des sorties précédentes. Cela peut s'exprimer en Lustre grâce au mécanisme d'assertions, introduites par le mot-clé `assert`, et jointes au système d'équations d'un nœud.

Par exemple, au lieu de calculer `half`, on aurait pu considérer ce flot comme une entrée libre et ajouter l'assertion :

```
assert half ->(half = not(pre half));
```

Comme on le voit ci-dessus, `assert` prend en argument une expression de type booléen et a pour effet de contraindre tous les éléments de ce flot à prendre la valeur « vrai ».[3]

[3] Il faut remarquer que le signe `=` en Lustre est surchargé. Au lieu de représenter l'opérateur de définition comme dans une équation, il représente ici l'opérateur d'égalité booléenne :

$$\forall n \in N : (\mathtt{x} = \mathtt{y})(n) = (\mathtt{x}(n) = \mathtt{y}(n))$$

Chapitre 4

Exemple de spécification et de vérfication formelles

4.1 Introduction

Nous présentons ici un exemple de programme synchrone qui nous servira de cas d'étude pour la répartition de programmes. En même temps, il nous sert à illustrer les outils Lustre de vérification de programmes. Nous devrons donc exprimer les propriétés que l'on veut vérifier. Mais ces propriétés ne dépendent pas seulement du programme mais aussi de l'environnement sur lequel il agit. La vérification formelle d'un programme se présente donc [18] selon le schéma de la figure 4.1.

4.2 Spécification

L'exemple proposé est celui d'une voie unique de chemin de fer. Le programme de commande doit assurer que des accidents ne surviennent pas. Le système global est composé de trois grandes parties :

- l'environnement physique : la voie, les trains, les aiguillages, les capteurs de présence et les actionneurs qui activent les aiguillages et les feux.
- le programme de commande : un programme Lustre qui lit les capteurs et commande les actionneurs (feux et aiguillages).
- Les conducteurs de trains. Ils obéissent aux feux et ne reculent pas.

4.2.1 L'environnement physique

Il se compose des éléments suivants :

- la voie unique,
- les voies d'entrée et de sortie de la voie unique,
- sur chaque voie d'entrée, un capteur de présence,
- un aiguillage aux deux bouts de la voie unique,

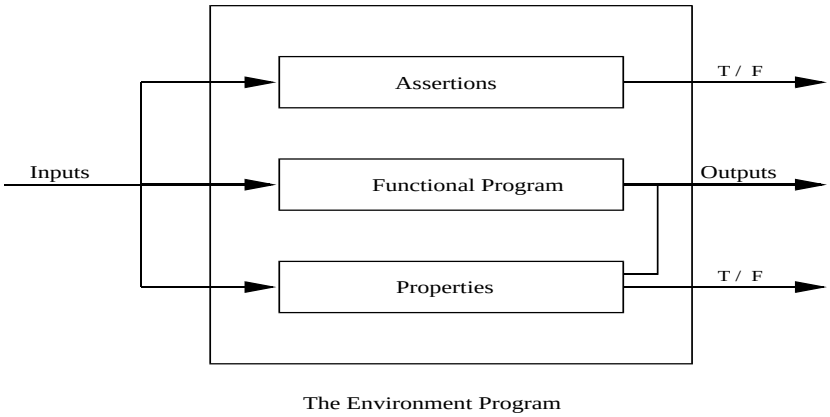


FIG. 4.1 – Schéma général de la vérification

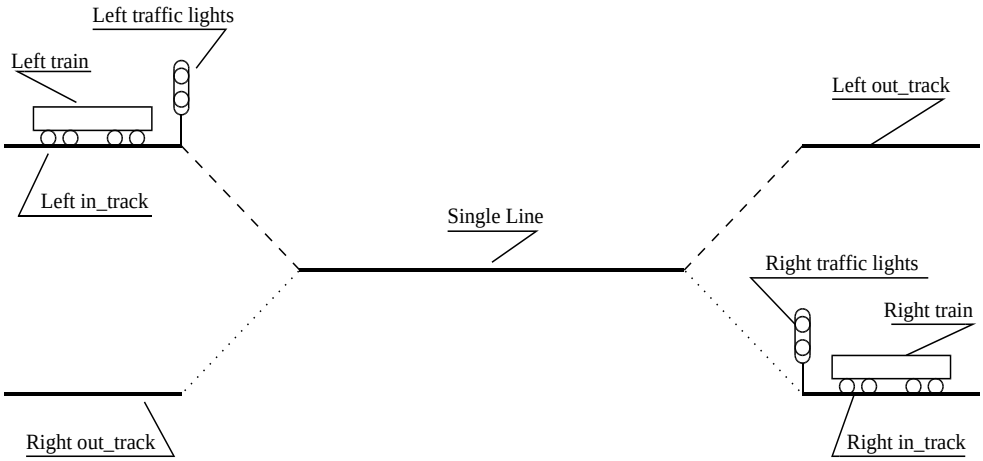


FIG. 4.2 – Schéma de la voie

- un capteur de présence sur la voie unique,
- des capteurs de position d'aiguilles,
- des feux à chaque bout de la voie unique,
- des trains venant soit de la gauche, soit de la droite.

La figure 4.2 illustre le schéma de voie.

4.2.2 La modélisation de l'environnement

Nous montrons ici comment on peut utiliser les assertions de Lustre pour modéliser les propriétés de l'environnement.

La voie unique est initialement vide. Soit `line_busy` le signal de présence d'un train sur la voie unique.

```
assert (not line_busy)-> true;
```

dit que la voie est initialement vide.

Les trains obéissent aux feux. Soit `left_train_here` le signal de présence d'un train sur la voie d'entrée de gauche, et `left_traffic_light` la commande du feux de gauche. L'assertion

```
assert (redge(not left_train_here)
        => (true -> pre(left_traffic_light)));
```

dit que pour que le capteur de présence passe de vrai à faux (`redge` est le noeud « front montant »), c'est à dire que le train passe de la voie d'entrée à la voie unique, il faut que le feux ait été vert à l'instant précédent. Par ailleurs, le noeud `redge` est défini comme suit :

```
node redge(a: bool) returns (e: bool)
let
e = true  -> a and not pre a;
tel
```

De meme,

```
assert (redge(not right_train_here)
        => (true -> pre(right_traffic_light)));
```

Les trains ne reculent pas. Lorsqu'un train quitte une voie d'entrée, c'est pour remplir la voie unique.

```
assert ((redge(not left_train_here)
         or redge(not right_train_here))
        => line_busy) ;
```

Ici, il faut faire très attention à ne pas sur-contraindre l'environnement. Si on avait écrit `redge(line_busy)` on aurait exclu le fait qu'un train peut déjà être sur la voie unique et donc on aurait *exclu les collisions* par construction sans intervention de la commande !

Les trains ne volent pas. Pour que la voie unique devienne occuppée, il faut qu'un train quitte une des voies d'entrée.

```
assert (redge(line_busy)
       =>((pre(left_traffic_light )
           and redge(not left_train_here))
          or
```

```
        (pre(right_traffic_light)
          and redge(not right_train_here))
        )) ;
```

Les aiguilles ne peuvent être dans deux positions. L'aiguille de gauche est soit dans la position d'entrée d'un train de gauche à droite, (`in_left_switch_cap` vraie) soit dans la position de sortie d'un train de droite à gauche.

```
assert (not(in_left_switch_cap and out_right_switch_cap)) ;
```

et de même

```
assert (not(in_right_switch_cap and out_left_switch_cap))  ;
```

Les aiguilles ne bougent que si elles sont commandées. On ne veut pas connaître le temps exact nécessaire à une aiguille pour se déplacer, mais on veut être sûr que, lorsqu'elle se déplace, c'est parce qu'on le lui a demandé.

```
assert (false->redge(in_left_switch_cap))
        => in_left_switch  ;
assert (false->redge(not in_left_switch_cap))
        => not in_left_switch) ;
```

où `in_left_switch` est l'ordre de bouger l'aiguille dans la position entrée gauche. On doit répéter cela pour toutes les positions et commandes d'aiguilles.

Pour tenir compte de toutes ces assertions, on peut les regrouper dans le nœud *environment* utilisé ensuite dans le programme de vérification comme assertion (assert environment(...)). La figure 4.3 définit ce nœud.

4.2.3 Propriétés désirées

Une fois qu'on a décrit l'environnement, on va spécifier les propriétés que l'on désire de la commande.

Non collision Une collision peut se produire :
- si des trains sont autorisés dans les deux sens,
- ou si un train est autorisé alors que la voie unique est occupée,

ce qui donne l'équation :

```
collision =

-- les trains peuvent aller dans les deux sens

        (left_traffic_light and right_traffic_light) or
```

```
node environment(line_busy, left_train_here, right_train_here,
                 in_left_switch_cap, out_left_switch_cap,
                 out_right_switch_cap, in_right_switch_cap,
                 left_traffic_light, right_traffic_light,
                 in_left_switch, out_left_switch,
                 out_right_switch, in_right_switch: bool)
     returns(prop: bool);
let
prop =

   ((not line_busy)-> true) and

   (redge(not left_train_here)
          => (true -> pre(left_traffic_light))) and

   (redge(not right_train_here)
          => (true -> pre(right_traffic_light))) and

   ((redge(not left_train_here)
           or redge(not right_train_here))
          => line_busy) and

   (redge(line_busy)
         =>((pre(left_traffic_light )
             and redge(not left_train_here))
            or
           (pre(right_traffic_light)
             and redge(not right_train_here))
           )) and

   (not(in_left_switch_cap and out_right_switch_cap)) and

   (not(in_right_switch_cap and out_left_switch_cap)) and

   (false->redge(in_left_switch_cap)) => in_left_switch and

   (false->redge(not in_left_switch_cap)) => not in_left_switch);

tel.
```

FIG. 4.3 – Hypothèses sur l’environnement

```
-- ou quand la voie unique n'est pas libre

        ((      (left_traffic_light and left_train_here)
             or (right_traffic_light and right_train_here))
         and line_busy);
```

Déraillement On autorise un mouvement alors que les aiguilles sont mal positionnées :

```
derailment =  (left_traffic_light and
                 not(in_left_switch_cap and out_left_switch_cap))
            or
              (right_traffic_light and
               not(out_right_switch_cap and in_right_switch_cap));
```

Équité Cette propriété, moins importante, permet d'assurer que les trains prennent la permission de passer à tour de rôle. En particulier, deux trains ne passent pas dans la même direction pendant qu'un autre train attend dans la direction contraire

```
starvation =

              not (once_from_to_(right_train_here,
                                 left_traffic_light,
                                 left_traffic_light))
             or  not (once_from_to_(left_train_here,
                                    right_traffic_light,
                                    right_traffic_light));
```

où `once_from_to_` est le nœud d'usage général suivant :

```
node once_from_to_( A, B, C : bool)
returns (onceAfromBtoC : bool);
let
   onceAfromBtoC = if (C and not B) then (
                        never(B) or
                        jafter(once_since_(A, B)) or
                        jafter(once_since_(C, B))
                    ) else true;
tel
```

défini à partir des nœuds suivants :

```
node never(X:bool) returns (never_X:bool);
let
 never_X = not once(X);
           -- not X -> not X and pre(never_X);
tel

node jafter(X : bool) returns (jafter_X : bool);
let
   jafter_X = false -> pre X;
tel

node once_since_( A, B : bool)
returns (onceAsinceB : bool);
let

   onceAsinceB = if never(B) then true
                 else if B then false
                 else (A or jafter(onceAsinceB));
tel
```

Finalement, on écrit que le système est sûr si la variable :

```
safety =  properties(
                    line_busy, left_train_here, right_train_here,
                    in_left_switch_cap, out_left_switch_cap,
                    out_right_switch_cap, in_right_switch_cap,
                    left_traffic_light, right_traffic_light,
                    in_left_switch, out_left_switch,
                    out_right_switch, in_right_switch);
```

est toujours vraie. Le nœud *properties* retourne la valeur booléenne prop calculée à partir des conditions ci-dessus :

```
prop = not(collision or derailment or starvation)
```

4.3 Le programme de commande

Ayant décrit l'environnement et les propriétés attendues, il reste à ecrire le programme de commande.

A cause de la propriété d'équité, il faut faire attention, lorsqu'il y a deux trains, à celui qu'on va faire passer. Pour cela, on définit la variable `priority` prise à vrai, par convention lorsqu'on veut faire passer le train de gauche.

```
  priority = if possible_change
             then if two_trains_here
```

```
                then (true -> not pre priority)
                else if left_train_here
                     then true
                     else false
           else(true -> pre priority);
```

où

```
  two_trains_here = left_train_here and right_train_here;
```

c'est à dire que lorsqu'il y a deux trains, on fait basculer la priorité.

La condition `possible_change` est très importante car elle nous dit quand faire bouger les aiguilles. On peut le faire quand un train rentre, alors que la voie unique est vide ou lorsqu'elle se vide et qu'un train est en attente.

```
  possible_change =  redge(one_train_here and not line_busy);
```

où

```
one_train_here = left_train_here or right_train_here;
```

On peut ensuite créer un chemin physique en commandant les aiguilles d'après le choix de passage qu'on a fait.

```
in_left_switch = priority;
out_left_switch = priority;
out_right_switch = not priority;
in_right_switch = not priority;
```

Lorsqu'on constate que le chemin est crée, c'est à dire lorsque la commande et les capteurs d'aiguilles sont cohérents, on sait qu'ils ne bougeront plus tant qu'on ne modifie pas la commande. On peut alors allumer les feux par :

```
  left_traffic_light =    false -> left_train_here
                        and in_left_switch
                        and out_left_switch
                        and in_left_switch_cap
                        and out_left_switch_cap
                        and not line_busy ;
```

et symmétriquement pour `right_traffic_light`.

Le programme de commande est alors décrit par le nœud `control_single_line` défini à partir de ces équations et avec l'interface suivante :

```
node control_single_line(
                line_busy, left_train_here, right_train_here,
                in_left_switch_cap, out_left_switch_cap,
                out_right_switch_cap, in_right_switch_cap: bool)
     returns(left_traffic_light, right_traffic_light, in_left_switch,
      out_left_switch, out_right_switch, in_right_switch: bool);

var

one_train_here: bool;
two_trains_here: bool;
priority: bool;
possible_change: bool;

let
...
tel
```

4.4 Le programme de verification

Les équations précédentes sont regroupées en noeuds, décrivant l'environnement (on remplace les assertions décrites précédemment par le calcul de la valeur de vérité d'une assertion globale), la propriété, et la commande.

Pour vérifier formellement la commande, on crée le noeud de verification qui calcule la valeur de vérité du système en boucle fermée, présenté dans la table 4.2.

On voit bien ici les entrées libres du programme : ce sont les réponses de l'environnement, (décisions des conducteurs de trains, des moteurs d'aiguilles), qui ne sont que partiellement déterminées par l'assertion.

4.5 L'outil de vérification Lesar

L'outil Lesar de vérification formelle comporte plusieurs principes. Celui illustré ici est l'outil énumératif. Il traduit ce programme booléen en automate et énumère les états. Si aucun état ne calcule la variable `safety` à faux alors que l'assertion n'a pas été violée, il dit que la propriété est vraie.

La table 4.1 indique le résultat. On voit que l'automate ne comporte que peu d'états. La vérification est instantanée. Nous pouvons donc considérer que le programme de commande est sûr.

```
[olan]$ lesar system.lus verif_safety -v -diag -merge
Pollux Version 1.33a
start normalisation ... done
start minimal network generation ..... done (237 -> 184 nodes)
building bdds ... 31 (on 31)
computing relevant statevars ... done (16 on 22)
DONE =>    48 states   404 transitions

TRUE PROPERTY
[olan]$
```

TAB. 4.1 – Résultat de Lesar

```
node verif_safety (
     line_busy, left_train_here, right_train_here,
     in_left_switch_cap, out_left_switch_cap,
     out_right_switch_cap, in_right_switch_cap: bool)
returns (safety:bool)

var left_traffic_light, right_traffic_light, in_left_switch,
     out_left_switch, out_right_switch, in_right_switch
     : bool;

let
     (left_traffic_light, right_traffic_light, in_left_switch,
      out_left_switch, out_right_switch, in_right_switch)
     =
     control_single_line(
                 line_busy, left_train_here, right_train_here,
                 in_left_switch_cap, out_left_switch_cap,
                 out_right_switch_cap, in_right_switch_cap);

assert environment(
                   line_busy, left_train_here, right_train_here,
                   in_left_switch_cap, out_left_switch_cap,
                   out_right_switch_cap, in_right_switch_cap,
                   left_traffic_light, right_traffic_light,
                   in_left_switch, out_left_switch,
                   out_right_switch, in_right_switch);

safety = properties(
                    line_busy, left_train_here, right_train_here,
                    in_left_switch_cap, out_left_switch_cap,
                    out_right_switch_cap, in_right_switch_cap,
                    left_traffic_light, right_traffic_light,
                    in_left_switch, out_left_switch,
                    out_right_switch, in_right_switch);

tel
```

TAB. 4.2 – Système en boucle fermée

Chapitre 5

Répartition en temps logique

5.1 Introduction

Nous avons vu en introduction que dans bien des cas, les domaines où s'applique la programmation synchrone exigent des solutions réparties. Comment peut-on appliquer celle-ci à de telles applications ? Des recherches ont été entreprises pour résoudre ce problème. Elles se sont fondées sur la recherche de sémantiques relachées permettant d'accepter des exécutions réparties "comme si elles étaient exactes". Nous allons étudier deux d'entre elles, la sémantique de Kahn applicable à Lustre [23, 11] et la sémantique relationnelle de Signal [3].

5.2 Réseaux de Kahn

Dans les années 70, Gilles Kahn s'interrogeait sur la façon de représenter le comportement de réseaux de processus *asynchrones, déterministes* communiquant entre eux par files FIFO, par exemple par des « sockets » Unix. La figure 5.1 représente un tel réseau.

L'idée est la suivante : supposons qu'un observateur enregistre l'histoire de chaque file, c'est à dire la suite finie ou infinie de valeurs initialement présentes ou entrées dans la file. Par exemple :

$$X = x_0, x_1, \ldots$$

Comme les processus sont supposés déterministes, l'histoire de la file de sortie d'un processus sera une fonction des histoires de ses files d'entrée : Les processus sont déterministes :

$$X_1 = P_1(Y, U_1)$$

Donc, le réseau « calcule » une solution d'un système d'équations :

$$X_1 \;=\; P_1(Y, U_1)$$

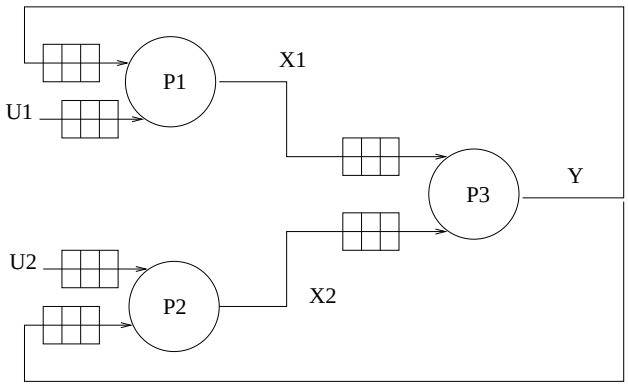


FIG. 5.1 – Un réseau de Kahn

$$\begin{aligned} X_2 &= P_2(Y, U_2) \\ Y &= P_3(X_1, X_2) \end{aligned}$$

Quel sens cela a-t-il ?

5.2.1 Sémantique de Kahn

Une histoire de type D prend ses valeurs dans l'ensemble :

$$D^\infty = D^* + D^\omega$$

somme des ensembles des suites finies et infinies. Cet ensemble est muni d'un ordre partiel $\leq$ (ordre préfixe)

$$x \leq y \Leftrightarrow \exists z : y = x@z$$

o$\check{\text{r}}$ @ est l'opérateur de concaténation, tel que :

- il y a un élément minimum ϵ ;

$$\forall x \in D^\infty : \epsilon \leq x$$

- toute suite d'histoires croissante (chaîne)

$$C = \{x_0 \leq x_1 \leq \ldots x_n \leq \ldots\}$$

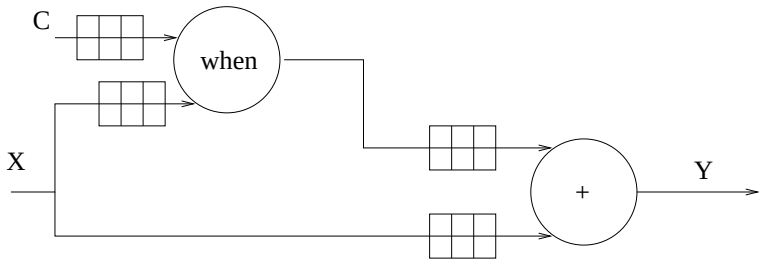
a une borne supérieure, c'est à dire un plus petit majorant :

$$\forall x \in C : x \leq sup\, C$$

$$\forall y \in D^\infty : (\forall x \in C : x \leq y) \Rightarrow sup\, C \leq y$$

On dit alors que $\Rightarrow (D^\infty, \leq, \epsilon)$ est un ordre partiel complet (CPO).

On peut alors définir des fonctions continues :



$$Y = X + (X\ when\ C)$$

FIG. 5.2 – Réseau de Kahn à files non bornables

– une fonction $f : D^\infty \Rightarrow D^\infty$ est continue si pour toute chaîne $C = x_0, \ldots x_n \ldots$

$$f(sup\ C) = sup\{f(x_i) | x_i \in C\}$$

et le théorème de Kleene dit que toute fonction continue sur un CPO a un plus petit point fixe, c'est à dire que l'équation $x = f(x)$ a une plus petite solution notée μf. Cela s'étend aussi aux systèmes d'équations avec variables libres. Le plus petit point fixe est alors une fonction continue des variables libres.

De plus cette solution se calcule par itérations successives en partant des suites vides :

$$\mu f = sup\{\epsilon \leq f(\epsilon) \leq \ldots f^n(\epsilon) \leq \ldots\}$$

Cela traduit bien le fonctionnement opérationnel des réseaux de Kahn.

5.2.2 Lustre et sémantique de Kahn

La sémantique de Lustre se déduit de celle des réseaux de Kahn en deux étapes :

La première étape est celle du calcul d'horloge : il vise à éliminer les réseaux qui ne peuvent pas s'exécuter avec une mémoire bornée comme celui de la figure 5.2.

Ensuite, le fonctionnement synchrone s'obtient en remplaçant les itérations chaotiques des réseaux de Kahn par des itérations ordonnées : on itère globalement sur le système d'equations puis on recommence. On définit ainsi une échelle de temps logique. Chaque itération correspond alors à un instant de cette échelle, et consiste à utiliser obligatoirement la tête de chacune des files d'attentes. En plus, au sein de chaque itération, un ordre est imposé par les dépendances instantanées. Ainsi, si la variable x dépend instantanément de y, alors la tête de la file de y est utilisée (lue ou écrite) avant celle de x. Globalement, on a l'ordre partiel suivant sur l'utilisation des éléments des files d'attente :

– Lecture des entrées (retirer la tête de chaque file d'entrée).
– Ecriture des variables à calculer en respectant la dépendance instantanée.
– Recommencer une autre itération, ceci constitue l'instant logique suivant.

Remarquons que pour les variables plus lentes, elles ne sont impliquées que dans les itérations où leurs horloges sont vraies.

5.2.3 Lustre réparti

Lustre réparti revient à garder le calcul d'horloge (qui garantit qu'il y a des exécutions à mémoire bornée) tout en relachant le fonctionnement synchrone c'est à dire que les divers processus itèrent indépendamment les uns des autres. La seule synchronisation se produit lorsqu'un processus se bloque en lecture d'une file vide. Il doit attendre. Dans la thèse d'Alain Girault [16], on voit comment on peut plus ou moins forcer des synchronisations en ajoutant des communications non fonctionneles forçant les processus à s'attendre les uns les autres.

5.3 Signal et la répartition

5.3.1 La sémantique de Signal

Contrairement à Lustre, Signal propose une sémantique relationnelle. Un processus P est un ensemble de comportements :

$$P \subseteq D_X^\omega \times D_Y^\omega \times \dots$$

où

- D^ω est l'ensemble des suites infinies de valeurs de D,
- $X, Y, \dots$ sont les noms des variables de P :

$$X \in N(P)$$

de sorte qu'on puisse écrire :

$$P \subseteq \prod_{X \in N(P)} V_X^\omega$$

Comment composer des processus et exprimer qu'ils communiquent ? L'idée est que, si un processus P produit la variable X et un autre Q lit cette même variable, à chaque comportement de P comportant uns suite s_X, il doit y avoir un comportement de Q comportant cette même suite. Un comportement du processus composé, noté $P||Q$ s'obtient donc en recherchant ces comportements partageant les mêmes suites sur les variables de communication et en les fusionnant, c'est à dire en ne gardant qu une fois les suites communes et en conservant toutes les autres. Cela se formalise de la façon suivante :

On définit l'opérateur de projection d'un processus sur un sous-ensemble de noms :

$$\Phi_{N'} P = \{ s \mid s \in \prod_{X \in N'} V_X^\omega \ \wedge \ \exists s', (s, s') \in P \}$$

Alors, la composition parallèle $P \mid\mid Q$ est le processus :

$$P \mid\mid Q = (\Phi^{-1}_{N(P)} P) \cap (\Phi^{-1}_{N(Q)} Q)$$

c'est à dire qu'on élargit chaque processus (par l'opérateur de projection inverse) à tous les autres noms possibles et on en fait l'intersection.

5.3.2 Les horloges de Signal

On a vu que, contrairement à Lustre, Signal ne considère que des suites infinies. Comment fait-on pour que des processus avancent moins vite que d'autres ? Cela se fait en considérant un élément « absent » appartenant au domaines de toutes les variables.

$$\forall X,\ V_X = V'_X + \{\perp\}$$

Il faut remarquer que, dans la composition parallèle, on considère les suites de variables de communication communes « avec absents ». Cela veut dire que les processus qui communiquent doivent se mettre d'accord non seulement sur les vraies valeurs qu'elles s'échangent, mais aussi avec les valeurs absentes. Le calcul d'horloge de Signal a pour effet de permettre cette mise en accord.

5.3.3 Exécutions asynchrones

Cependant, des exécutions à la Réseaux de Kahn de processus Signal ne sont pas toujours possibles : en effet, les seules valeurs transmises sont les vraies valeurs et non pas les absentes.

Pour formaliser ce fait, on introduit le opérateur ϕ_a d'oubli des absents, défini récursivement par :

$$\begin{aligned} \phi_a(\epsilon) &= \epsilon \\ \phi_a(\perp.s) &= \phi_a(s) \\ \phi_a(v.s) &= v.\phi_a(s) \end{aligned}$$

Cet opérateur s'étend naturellement aux ensembles de comportements.

Cependant, lorsqu'on oublie les absents, des suites infinies peuvent devenir finies. On doit donc étendre la définition sémantique des processus aux suites fines ou infinies en écrivant :

$$P \subseteq \prod_{X \in N(P)} V_X^{\infty}$$

Dans [3], deux conditions nécessaires et suffisantes sont établies pour une exécution de programmes Signal, en réseaux de Kahn :

L'isochronie. Deux processus sont isochrones si l'opérateur d'oubli des absents Φ_a distribue sur la composition parallèle :

$$\phi_a(P \parallel Q) = \phi_a(P) \parallel \phi_a(Q)$$

Cela garantit en effet qu'on aura les mêmes ensembles de comportements, que l'on exécute le programme de façon synchrone puis qu'on efface les absents ou bien qu'on exécute de façon asynchrone en ne communiquant que de vraies valeurs.

Endochronie Mais ce n'est pas suffisant car il s'agit d'un raisonnement global sur des ensembles de comportements. Or, ce qui nous intéresse, ce sont les comportements individuels et non leurs ensembles. C'est pourquoi on introduit la propriété d'endochronie qui dit que l'opérateur d'oubli des absents est injectif sur les comportements d'un processus.

$$\forall s, s' \in P,\ \phi_a(s) = \phi_a(s') \ \Rightarrow\ s = s'$$

Donc, si P et Q sont conjointement isochrones et si $P||Q$ est endochrone, on peut répartir P et Q et les exécuter en réseaux de Kahn.

5.3.4 Comparaison avec Lustre

Remarquons cependant que cette théorie n'ajoute rien à Lustre, qui est fonctionnel et dont la sémantique s'exprime sans « absent ». Les programmes Lustre valides, c'est à dire exécutables en synchrone sont naturellement endochrones et mutuellement isochrones.

Deuxième partie

Programmation synchrone : le temps physique

Chapitre 6

Architectures temps réel

6.1 Evolution des pratiques

En réalité, dans la pratique, la programmation synchrone est surtout utilisée dans sa version périodique, correspondant au programme de la table 6.1. La raison principale est historique. Elle vient de la façon dont on a remplacé les anciens circuits analogiques par des calculateurs numériques. Ceci est illustré à la figure 6.1 et a consisté à utiliser une horloge périodique temps-réel pour piloter

- des convertisseurs analogiques-digitaux en entrée,
- les activités de calcul du calculateur,
- des convertisseurs digital-analogiques en sortie.

6.2 Intérêt de cette architecture

Cette architecture a de nombreux intérêts pratiques en plus de ceux déjà mentionnés pour la programmation synchrone en général à la section 2.3 :

- Elle s'adapte très bien à la nécessité d'intégrer des équations diférentielles

```
initialize state;

loop each clock tick

   read other inputs;
   compute outputs and state;
   emit outputs

 end loop
```

TAB. 6.1 – Un programme synchrone périodique

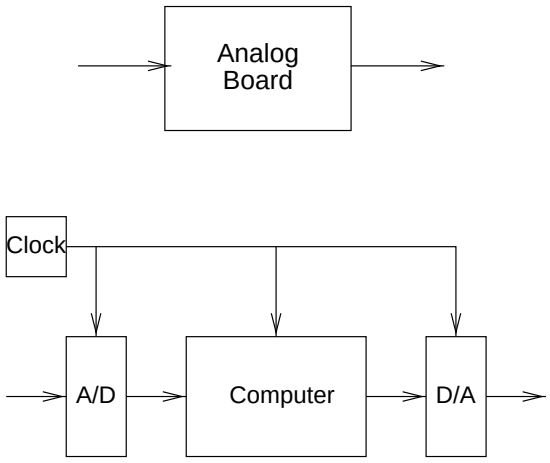


FIG. 6.1 – Evolution des calculateurs analogiques aux calculateurs numériques

en temps réel et aussi à la théorie mathématique des systèmes de commande échantillonnés [24].

– C'est une méthode simple, sûre et efficace :
  Il y a une seule interruption (l'horloge temps-réel), et cette interruption doit se produire uniquement lorsque le calculateur est oisif. Il n'y a donc pas besoin de sauver un contexte. Les besoins en système d'exploitation sont donc très limités et souvent les applications peuvent être implantées sur machines nues. C'est donc une méthode efficace car il n'y a pas de temps perdu en sauvegarde de contexte et c'est une méthode sûre car il est toujours difficile de valider un système d'exploitation temps réel.
– La vérification du fonctionnement temporel est assez simple : elle consiste à s'assurer que le pire temps de réaction est toujours plus petit que la période de l'horloge temps réel de sorte que l'interruption se produit en effet uniquement lorsque le calculateur a fini sa réaction et est donc oisif. Cela peut se faire de façon expérimentale ou par des calculs théoriques.

C'est donc une méthode intéressante surtout lorsqu'on a affaire à des systèmes critiques. C'est pourquoi elle est très utilisée dans ces domaines [9, 26, 6].

Chapitre 7

Limites de l'abstraction synchrone

7.1 Introduction

La programmation synchrone permet de décrire des comportements des systèmes, évoluant en temps réel, de façon périodique en assimilant la durée réelle d'une période pendant laquelle le système n'évolue pas à un instant ponctuel, appelé alors temps logique. Ceci suppose que les machines utilisées sont idéales, c'est à dire capables de capter tous les changements de l'environnement, mais aussi que la stabilité de l'environnement est facile à mesurer.

Dans ce chapitre nous présentons les limites de l'abstraction synchrone en soulignant les problèmes qui sont à l'origine de ces limites, ainsi que les conséquences notamment sur la vérification des propriétés de sîreté en temps réel.

7.2 Vérification synchrone et temps réel

La vérification synchrone assure la validité d'une propriété de sûreté en temps logique. Dans cette section, nous montrons que ce qui est validé en temps logique ne l'est pas forcément en temps réel. Nous analysons pour cela deux exemples.

Exemple 7.2.1 Calcul
On se propose de savoir si le signal b vaut bien 1 à chaque fois que a vaut 1. Cette propriété est spécifiée par le noeud suivant :

```
node imply(a,b: bool) returns (ok: bool);
let
ok = not a or b;
tel
```

Une conception possible de signaux ayant cette propriété est la suivante :

```
node concimply(x,y : bool) returns (a,b: bool);
```

```
let
a = x and y;
b = x;
tel
```

Remarquons que l'environnement tourne en parallèle avec le calcul des valeurs de a et b ainsi que l'observateur de la propriété. Notons que si la valeur de x change pendant que le calul se fait, la propriété $imply(a, b)$ n'est plus vérifiée en temps réel. Ceci nous montre que pour un calcul vrai en temps réel il ne faut pas que l'environnement change pendant le calcul. Ceci oblige d'avoir des entrées qui peuvent se stabiliser pendant un temps minimal.

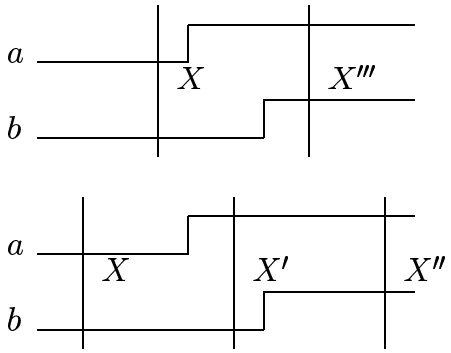


FIG. 7.1 – L'échantillonnage n'est pas déterministe

Echantillonnage et déterminisme La figure 7.1 montre deux échantillonnages périodiques d'un même couple de variables d'entrées (a, b). Dans le premier cas, a et b passent en même temps logique de 0 à 1, alors que dans le second cas il y a un passage intermédiaire. Un programme de contrôle ne devrait pas être sensible à ce déphasage dû à l'échantillonnage. La programmation synchrone ne fournit pas de moyens pour assurer le bon échantillonnage.

Même s'il n'est pas impossible d'avoir un environnement dont les signaux ont une borne inférieure de stabilité, l'exemple de l'échantillonnage non déterministe nous montre qu'il n'est pas évident de déduire l'existence d'un échantillonnage pendant une période de stabilité de l'environnement.

Exemple 7.2.2 Exclusion mutuelle
Soit une spécification de deux sorties booléennes u et v ne devant jamais être vraies en même temps.

solution triviale Cette spécification peut s'écrire en Lustre de la façon triviale suivante :

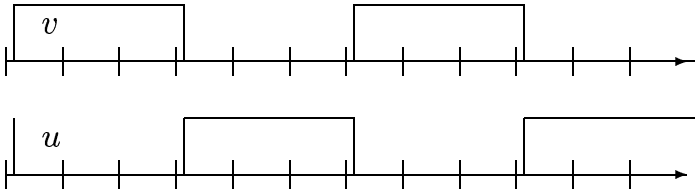


FIG. 7.2 – Un exemple d'exclusion mutuelle

```
node specif1(u,v bool) returns(prop: bool);

let prop = not (u and v);
tel
```

Une conception triviale réalisant cette spécification est la suivante :

```
node concept1(x : bool) returns(u, v : bool);

let u = x;
    v = not x;
tel
```

Il est évident que la propriété d'exclusion mutuelle est satisfaite une fois les deux valeurs de u et v sont calculées. Cependant, entre les calculs successifs de u et v cette propriété n'est jamais satisfaite. En effet, en disant que u et v ne doivent jamais prendre la valeur *vraie* en même temps nous sous-entendons le temps logique, c'est à dire à la fin de chaque période synchrone.

Dans cet exemple, aussi rapide que soit la réaction du contrôleur par rapport à l'environnement, la propriété de sûreté est violée pendant un certain temps réel.

7.3 Conclusion

Les deux exemples présentés ont mis en évidence les limites de l'abstraction synchrone. Le paramètre temps réel doit être considéré d'une part dans la détermination de la réaction du contrôleur, et d'autre part durant le calcul puisque les dépendances instantanées imposent un certain ordre du calcul pour un résultat cohérent.

Dans le chapitre suivant nous présentons en fonction de la description de l'environnement, des critères de stabilité ainsi que d'insensibilité aux changements permettant de garantir l'équivalence de la validité en temps logique et réel.

Chapitre 8

Modélisation du temps réel

8.1 Modélisation mathématique

8.1.1 Définitions de base

On considère d'abord des signaux qui sont simplement des fonctions de R dans R, et des systèmes qui sont des fonctions causales transformant des signaux. Ici, causale est pris dans le même sens que dans les réseaux de Kahn, c'est à dire que le futur d'une entrée ne peut pas influencer le passé d'une sortie.

On définit aussi l'opérateur de retard Δ :

$$(\Delta^\tau\ x)t = x(t - \tau)$$

et les systèmes stationnaires :

$$\forall \tau, S(\Delta^\tau\ x) = \Delta^\tau(S\ x)$$

8.1.2 Redatation

La redatation (« retiming ») va permettre de rendre compte des incertitudes temporelles rencontrées au chapitre précédent et aussi les retards de communication et les asynchronismes des architectures que nous allons voir par la suite.

Définition 8.1.1 (Fonction de redatation) *Une fonction $r : R \to R$ est une fonction de redatation si elle est non décroissante et toujours inférieure à la fonction identité id.*

A toute fonction de redatation, on peut associer le système R_r défini par

$$R_r\ x = x \circ r$$

L'identité est évidemment une fonction de redatation et l'opérateur retard correspond aussi à une redatation.

Définition 8.1.2 (Redatations maximum et minimum) *Etant donné une redatation r on définit ses bornes maximum et minimum :*

$$r^M = \sup_t \ t - r(t)$$

$$r^m = \inf_t \ t - r(t)$$

Définition 8.1.3 (Redatation bornée) *Une fonction de redatation $r : R \to R$ est bornée si r^M existe.*

8.1.3 Redatation et échantillonnage

La redatation est très utile aussi parce qu'elle peut rendre compte de l'échantillonnage aussi bien périodique que non périodique :

Définition 8.1.4 (Echantillonneur) *Un échantillonneur est simplement une redatation bornée constante par morceaux.*

De plus un échantillonnage périodique de période T vérifie :

$$T = r^M - r^m$$

8.1.4 Systèmes et signaux uniformément continus

Nous étudions ses signaux et ses systèmes parce qu'ils sont à la base de beaucoup de systèmes de contrôle-commande et que les architectures de ces systèmes ont été initialement conçues pour eux. Il est donc important de comprendre comment ces systèmes sont pris en compte dans ces architectures.

Signaux uniformément continus La définition classique dit qu'un signal x est UC si

$$\forall \epsilon > 0, \exists \eta_x > 0, \forall t, t', |t - t'| \leq \eta_x \Rightarrow |x(t) - x(t')| \leq \epsilon$$

mais on peut aussi l'écrire :

Définition 8.1.5 (Signaux uniformément continus) *Un signal x est UC si il existe une fonction positive des erreurs vers les délais η telle que :*

$$\forall \epsilon > 0, \forall t, t', |t - t'| \leq \eta_x(\epsilon) \Rightarrow |x(t) - x(t')| \leq \epsilon$$

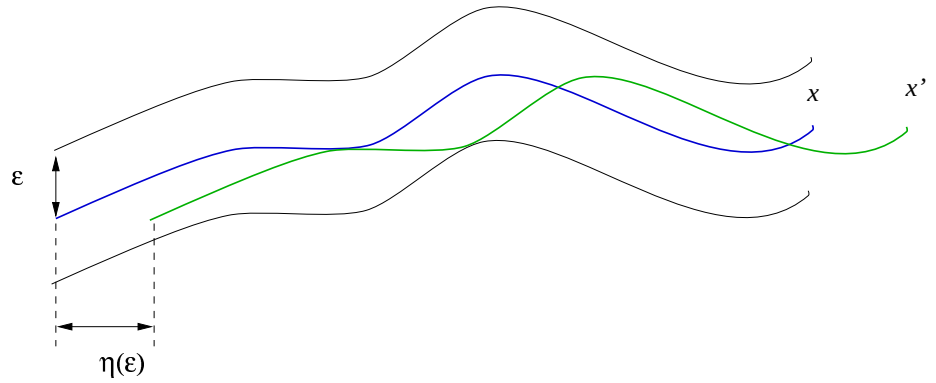


FIG. 8.1 – Continuité uniforme

Distance entre signaux Pour rester en cohérence avec les définitions précédentes on utilise la norme de signaux $||\ ||_\infty$, définie par :

$$||x||_\infty = \sup_{t \in R} |x(t)|$$

On peut alors faire le lien entre cette continuité et la redatation :

Théorème 8.1.1 (Continuité et redatation) *x est UC avec la fonction d'erreur η_x si et seulement si*

$$\forall \epsilon > 0, \forall r, r^M \leq \eta_x(\epsilon) \Rightarrow ||x - x \circ r||_\infty \leq \epsilon$$

On comprend donc la relation entre continuité et incertitude temporelle. Les signaux UC sont robustes par rapport à l'incertitude temporelle (figure 8.1). La raison principale pour cela est qu'ils ne varient pas trop vite (leur vitesse de variation est limitée).

8.1.5 Signaux à variabilité uniformément bornée

On essaie de trouver un équivalent de cette propriété de variation pas trop rapide pour les signaux discontinus, booléens par exemple. Evidemment, il n'est pas possible de parler de continuité, puisque, en certains points, la vitesse de variation est infiniment rapide. Cependant, on peut proposer la solution suivante :

Définition 8.1.6 (Compteur de discontinuité) *SOIT $dc_{t_1,t_2}(x)$ (pour discontinuity count) la fonction qui compte le nombre de points de discontinuité d'un signal booléen x dans un intervalle $[t_1, t_2[$.*

$$dc_{t_1,t_2}(x) = card\{t \mid x(t^-) \neq x(t^+) \wedge t_1 \leq t < t_2\}$$

où, comme d'habitude, $x(t^-), (x(t^+))$ est la limite à gauche (droite) de x à t.

Définition 8.1.7 (Signaux à variabilité uniformément bornée) *Un signal x est VUB s'il existe une fonction du nombre de discontinuités vers les délais, η_x, telle que :*

$$\forall n \in N^+, \forall t, t', |t - t'| \leq \eta_x(n) \Rightarrow dc_{t,t'}(x) \leq n$$

où N^+ est l'ensemble des entiers positifs. Cette définition copie celle de la continuité mais, en général, la seule valeur intéressante pour n est 1. Donc, $T_x = \eta_x(1)$ est le *temps de stabilité* minimum du signal x.

8.2 Modélisation en Lustre

Cette modélisation mathématique peut être facilement traduite en Lustre :

8.2.1 Echantillonnage

Nous avons déjà vu les primitives `when` et `current` qui permettent de décrire l´échantillonnage et le blocage.

8.2.2 Variabilité uniformément bornée

Il n'y a pas en Lustre de notion de temps réel. Mais on peut travailler en relatif. Pour décrire qu'un signal est à variabilité uniformément bornée, on dira simplement qu'il y a au moins `n` coups d'une horloge supposée périodique entre deux discontinuités du signal. Le noeud suivant (figure 8.2) calcule la valeur de vérité de cette propriété. Il suffira de l'introduire sous un `assert` pour garantir cette hypothèse.

8.2.3 Délais non déterministes bornés

Nous avons besoin d'exprimer les retards dûs à la prise en compte des signaux par l'échantillonneur et par le bloqueur. Plus précisément, nous considérons les retards d'un instant logique. En effet, grâce à la propriété de la variabilité uniformément bornée, nous savons que tout changement d'un signal est pris en compte au plus tard au coup d'horloge suivant. Ceci correspond en temps réel à une période comprise entre zéro et deux périodes d'horloge (bornes exclues).

La définition du nœud Bounded_Delay nous permet de dire si le le signal u2 représente ou non la prise en compte du signal u1. Si oui, nous disons alors que u2 est l'image retardée de `u1` de entre zéro et deux périodes d'horloge. Ce nœud sera utilisé pour définir des pseudo-entrées et sorties qui sont les prises en compte des entrées et sorties réelles. La validation de la propriété de sûreté sur les pseudo-entrées et sorties implique que cette propriété est valide aussi en temps réel. On parle alors de propriété de sûreté robuste vis-à-vis du temps réel.

```
node Bounded_Delay(u1, u2, cl : bool) returns (prop: bool);
var del : bool ;
let  prop = sample(1, u1, cl)
        and sample(1, u2, cl)
```

```
node s2(const n:int; a, cl:bool) returns (prop:bool)

-- cl is fast enough to take n samples of each level of a
-- n is at least 2

var ea:bool; s:bool^n;

let ea = edge(a);
    s[0]      = if cl then true
                else if ea then false
                     else (false -> pre s[0]);
    s[1..n-1] = if ea then false^(n-1)
                else if cl then (false^(n-1) -> pre s[0..n-2])
                     else (false^(n-1) -> pre s[1..n-1]) ;
    prop      = ea => (true -> pre s[n-1]);
tel

node s1(a, cl:bool) returns (prop:bool)

-- cl is fast enough to take 1 sample of each level of a

var ea:bool; s:bool;

let ea = edge(a);
    s      = if cl then true
             else if ea then false
                  else (false -> pre s);
    prop   = ea => (true -> pre s);
tel

node sample(const n:int; a, cl:bool) returns (prop: bool)

-- cl is fast enough to take n samples of each level of a

let
    prop = if n = 0 then true
           else if n = 1 then s1(a, cl)
                else s2(n, a, cl);
tel
```

FIG. 8.2 – Variabilité bornée en Lustre

```
        and del ;

  del = if cl
        then current (u2 when cl = (u1 when cl -> pre (u1 when cl)))
        else (true -> pre del);
tel
```

La figure 8.3 montre les deux signaux u1 et u2 vérifiant cette relation.

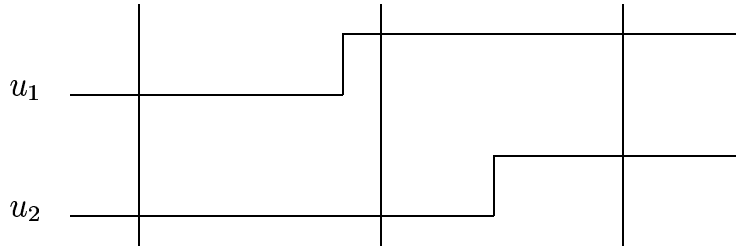


FIG. 8.3 – Délai non déterministe borné

8.3 Application à l'exemple

8.3.1 Commande échantillonnée

La preuve faite du contrôleur d'aiguillages dans le chapitre 4 était une preuve idéale qui supposait une machine infiniment rapide. On a vu au chapitre 6 que en général, les calculateurs utilisés étaient pilotés par une horloge temps réel périodique. La question est donc de savoir si la preuve que nous avons faite est robuste au passage d'une machine idéale à une machine périodique. Pour cela, nous utilisons les primitives d´échantillonnage et blocage et la variabilité bornée qui permet de mesurer les vitesses respectives de l'environnement et du programme. La figure 8.4 montre le programme de vérification de la propriété dans ce cas. On voit que le contrôleur est échantillonné mais que l'environnement et le calcul de la propriété ne le sont pas pas. Sa vitesse est définie par les assertions de variabilité bornée.

La figure 8.5 montre le résultat de Lesar. On voit que la propriété est vérifiée, un peu plus difficilement que précédemment. Cela montre une certaine robustesse.

8.3.2 Commande échantillonnée avec incertitude en entrée et sortie

La vérification précédente suppose cependant que l'échantillonnage est parfaitement exacte : les entrées sont toutes saisies en même temps et les sorties émises aussi

```
const n =1;
node verif_safety (cl,
     line_busy, left_train_here, right_train_here, in_left_switch_cap,
     out_left_switch_cap, out_right_switch_cap, in_right_switch_cap: bool)
returns (safety:bool)

var left_traffic_light, right_traffic_light, in_left_switch,
     out_left_switch, out_right_switch, in_right_switch
     : bool;

let
     (left_traffic_light, right_traffic_light, in_left_switch,
      out_left_switch, out_right_switch, in_right_switch)
     =
     if cl then current control_single_line(
                 (line_busy, left_train_here, right_train_here,
                  in_left_switch_cap, out_left_switch_cap,
                  out_right_switch_cap, in_right_switch_cap)when cl)
     else ((false,false,false,false,false,false)->pre(
           left_traffic_light, right_traffic_light, in_left_switch,
            out_left_switch, out_right_switch, in_right_switch));

-- assertions on sampling:

   assert sample(n,line_busy,cl) and
          sample(n,left_train_here,cl) and
          sample(n,right_train_here,cl) and
          sample(n,in_left_switch_cap,cl) and
          sample(n,out_left_switch_cap,cl) and
          sample(n,out_right_switch_cap,cl) and
          sample(n,in_right_switch_cap,cl);

assert environment(
     line_busy, left_train_here, right_train_here, in_left_switch_cap,
     out_left_switch_cap, out_right_switch_cap, in_right_switch_cap,
     left_traffic_light, right_traffic_light, in_left_switch,
     out_left_switch, out_right_switch, in_right_switch);

safety = properties(
     line_busy, left_train_here, right_train_here, in_left_switch_cap,
     out_left_switch_cap, out_right_switch_cap, in_right_switch_cap,
     left_traffic_light, right_traffic_light, in_left_switch,
     out_left_switch, out_right_switch, in_right_switch);
tel
```

FIG. 8.4 – Commande échantillonnée

```
[olan]$ lesar sampled.lus verif_safety -v -diag -merge
Pollux Version 1.33a
start normalisation ... done
start minimal network generation ..... done (425 -> 313 nodes)
building bdds ... 55 (on 55)
computing relevant statevars ... done (35 on 44)
DONE =>  1783 states 11774 transitions

TRUE PROPERTY
7.730u 7.770s 0:15.87 97.6%     0+0k 0+0io 684pf+0w
[olan]$
```

FIG. 8.5 – Vérification échantillonnée

en même temps. Nous avons discuté au chapitre 7 la validité de cette hypothèse et conclu quelle pouvait être dangereuse dans le temps réel. L'idée est alors de mettre des délais non déterministes bornés sur les entrées et les sorties du programme de commande. Le programme de vérification résultant est trop gros pour être montré mais la figure 8.6 montre le diagramme de bloc équivalent.[1]

La figure 8.7 montre l'application de Lesar à ce programme. La vérification est plus longue (20 mn) mais faisable. Elle montre que le programme est vraiment robuste aux incertitudes d'acquisition et d'action. Il est clair maintenant qu'on peut le mettre en oeuvre sans trop de soucis.

[1]En Scade, environnemnent de programmation graphique fondé sur Lustre et commercialisé par la société Telelogic.

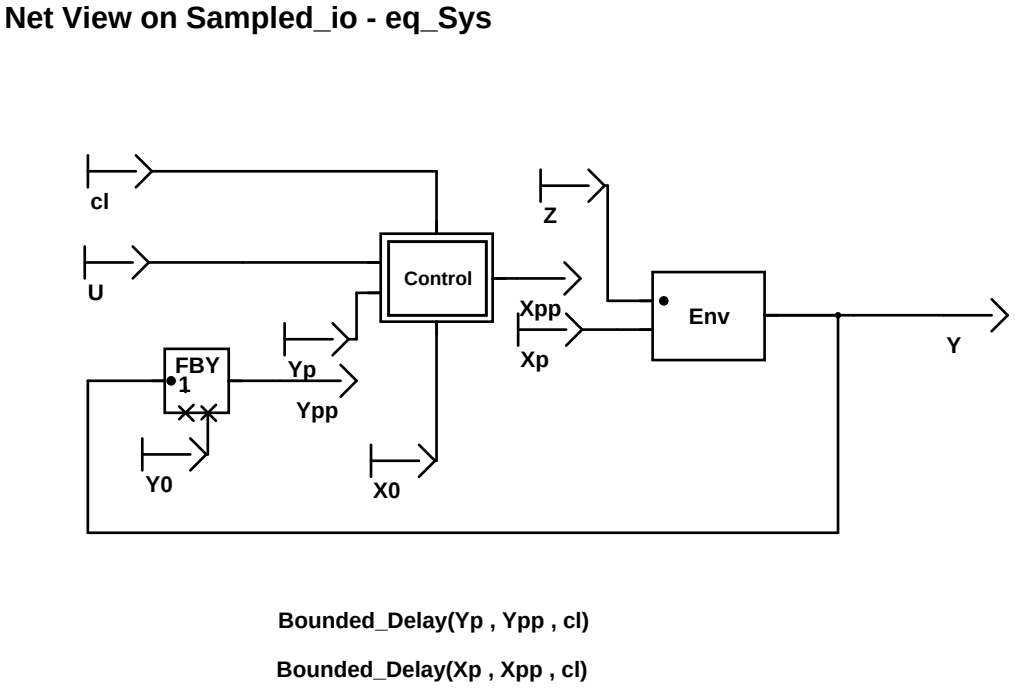


FIG. 8.6 – Un système de commande échantillonné avec incertitude d'entrée et de sortie : Yp et Xp représentent des pseudo-entrées et sorties (oracles).

```
[olan]$ lesar fuzzyio_sampled.lus verif_safety -v -diag -merge -states 10
Pollux Version 1.33a
start normalisation ... done
start minimal network generation ...... done (576 -> 407 nodes)
building bdds ... 105 (on 105)
computing relevant statevars ... done (52 on 64)
DONE => 16744 states 163916 transitions

TRUE PROPERTY
699.360u 818.860s 25:20.30 99.8%        0+0k 0+0io 685pf+0w
[olan]$
```

FIG. 8.7 – Vérification avec incertitude d'entrée et sortie

Troisième partie

Répartition en temps physique

Chapitre 9

Modélisation quasi-synchrone

9.1 L'architecture quasi-synchrone

Nous avons vu au chapitre 6 comment les ingénieurs automaticiens sont passés des calculateurs analogiques aux calculateurs numériques. Mais, en général, les grands systèmes de contrôle-commande comme, par exemple, un système de commande de vol pour un avion commercial ou un système de contrôle-commande d'installations chimiques sont composés de plus d'un calculateur. La figure 9.1 montre comment le schéma de la figure 6.1 a été utilisé dans ce cas.

L'idée a été de réutiliser le même schéma, avec une simple modification dans le cas où les calculateurs avaient à communiquer. Il aurait été possible de réutiliser exactement le même schéma que précédemment, c'est à dire de convertir les signaux en analogique, puis de les re-discrétiser, mais cela aurait été très peu économique. Il a suffi alors de remplacer cela par une transmission directe par exemple par une liaison série directe, de calculateur à calculateur. Ensuite, pour des raisons d'économie et d'efficacité, ces liaisons série ont été « empaquetées » dans des bus de terrain comme par exemple les bus FIP, CAN ou PROFIBUS [14].[1]

Cette méthode a beaucoup d'avantages :

- Elle est modulaire : elle permet de procéder à des remplacements successifs, selon les besoins et elle permet donc à plusieurs gammes de produits de coexister.
- Elle semble robuste : chaque calculateur est un organe intelligent et doué d'autonomie. Il comprend sa propre horloge temps-réel et même, éventuellement, sa propre alimentation.

  Il faut remarquer que la communication entre calculateurs est est très semblable à la communication entre les calculateurs et leurs environnements. Ce sont toujours de simples lectures et écritures périodiques, c'est à dire de l'échantillonnage. Il s'agit donc d'un mécanisme qui n'est pas bloquant et les calculateurs peuvent donc fonctionner périodiquement.
- En même temps, cette méthode est économique et efficace : on n'utilise que des

[1] Les bus Nervia et Arinc 429, utilisés par nos partenaires du projet Crisys sont aussi de cette nature.

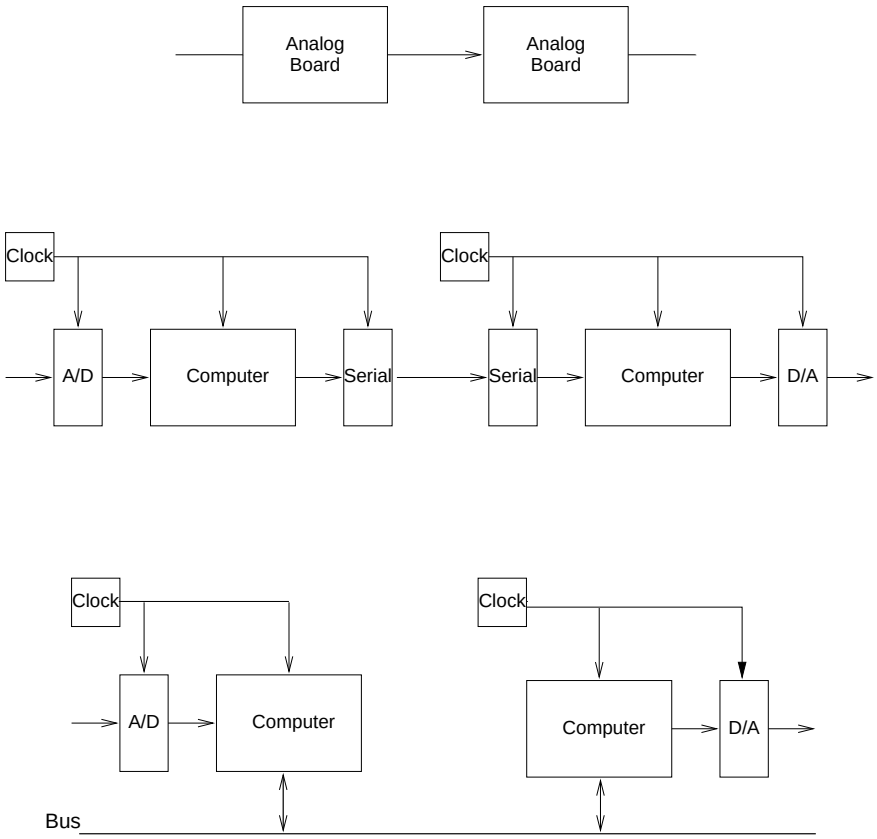


FIG. 9.1 – Des réseaux de cartes analogiques aux réseaux locaux

matériels standards très répandus et on peut profiter des évolutions en prix et en performances.

Cependant, il faut aussi remarquer que cette communication entre calculateurs est très peu conventionnelle pour un informaticien. Cela se voit à la figure 9.2 qui montre deux horloges temps réel. même si elles ont été exactement réglés à la même période et à la même phase, au bout d'un moment, cette situation va se dégrader parce qu'il n'y a pas de mécanisme de synchronisation entre horloges, contrairement à l'architecture. Même si les périodes d'horloge sont surveillées étroitement[2] et varient très peu, des problèmes surviennent :

- La communication entre les calculateurs est entachée d'erreurs : supposons, dans la figure 9.2 que l'horloge du haut soit une horloge d'écriture et celle du bas une horloge de lecture. On voit que même avec de très petites variations, il peut y avoir deux écritures entre deux lectures. Comme il n'y a pas de mécanisme de blocage, la première écriture est perdue. De la même façon, en inversant les rôles des horloges, on verrait que des données peuvent être lues deux fois et donc dupliquées.
- On voit aussi qu'au bout d'un certain temps, les deux horloges ne marqueront plus du tout la même heure. On ne peut pas se baser sur un temps absolu pour programmer ces systèmes.

9.2 Propriétés de l'architecture

9.2.1 Equité bornée

Cependant la situation est moins grave que dans les systèmes complètement asynchrones. Dans ceux-ci, la seule hypothèse que l'on fait sur la vitesse relative des différents sous-systèmes qui communiquent est l'hypothèse d'équité qui peut s'exprimer ainsi :

Propriété 9.2.1 (Equité) *Un processus ne peut pas s'exécuter une infinité de fois alors qu'un autre processus peut s'exécuter.*

Dans notre cas, cette propriété est remplacée par une propriété beaucoup plus forte qui est une propriété d'équité bornée et même bornée par le nombre 2 :

Propriété 9.2.2 (Equité bornée par 2) *Un processus ne peut pas s'exécuter plus de deux fois entre deux exécutions d'un autre processus.*

9.2.2 Délai de communication borné

On considère un composant écrivain qui écrit un signal x_w avec une période maximum T_w et un composant lecteur qui lit ce signal x avec une période maximum T_r et

[2] Dans tous ces systèmes, la période d'horloge doit être surveillées car sinon, le temps réel est perdu et la sûreté des systèmes peut être compromise [26].

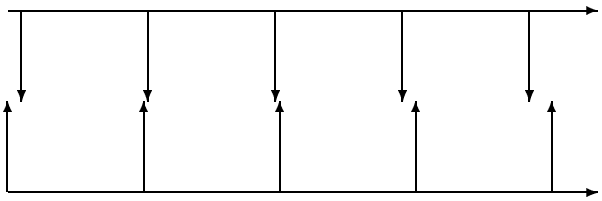


FIG. 9.2 – Deux horloges périodiques avec presque la même période

obtient x_r. Il est clair que, dans le pire des cas,

$$\begin{aligned} x_r(t) &= x(t - T_r) \\ x(t) &= x_w(t - T_w) \end{aligned}$$

Donc, dans le pire des cas,

$$x_r(t) = x_w(t - T_r - T_w)$$

et le délai de communication est borné par $T_r + T_w$. Cette abstraction peut être enrichie en prenant en compte le délai de communication physique ou même les autres erreurs de communication et des codes détecteurs d'erreurs produisant des échecs de communication. Si on suppose que le délai physique δ est borné bornés et que la probabilité pour avoir plus de n erreurs de communication successives est négligeable devant par exemple la probabilité d'autres sources de défaillance du système, on obtiendra toujours un délai global de communication borné :

$$T_w + \delta + nT_r$$

9.3 Simulation de l'architecture

9.3.1 Modélisation de la communication

La modélisation en Lustre de cette communication s'inspire de celle qui a déjà été faite pour l'échantillonnage simple au chapitre 8 et est fondée sur l'idée de mémoire partagée :

Mémoire partagée. Etant donné une expression `u` écrite dans la mémoire partagée à l'horloge `cw` et son contenu initial `v`, le contenu courant `uc` de la mémoire peut s´écrire

```
node mem(cw :bool; v : type; (u : type on cw))
returns (uc : type);
```

```
let uc = if cw then pre current u
         else ( v -> pre uc);
tel
```

où le retard permet de tenir compte des délais de communication physiques.[3]

Alors, la valeur lue dans la mémoire à l'horloge `cr` est

```
u'= mem(cw,v,u) when cr ;
```

9.3.2 Modélisation des horloges quasi-périodiques

Cela pourrait être fait dans un formalisme temps réel adéquat comme par exemple celui des automates temporisés [1] mais nous avons préféré, pour obtenir une démarche homogène rester dans le cadre des assertions Lustre et caractériser des horloges qui ont à peu près la même période grâce] la propriété d´équité bornée.

Propriété 9.3.1 (Horloges presque égales) *Aucune des deux horloges ne peut prendre plus de deux fois la valeur vrai, entre deux vrais successifs de l'autre horloge.*

Cela est ensuite formalisé en disant que le couple des deux horloges ne peut être représenté par l'expression régulière suivante :

$$\left[\begin{array}{c} t \\ - \end{array}\right] . \left[\begin{array}{c} f \\ f \end{array}\right]^* . \left[\begin{array}{c} t \\ f \end{array}\right] . \left[\begin{array}{c} f \\ f \end{array}\right]^* . \left[\begin{array}{c} t \\ - \end{array}\right]$$

ni par celle obtenue en échangeant les composantes. (Ici, $-$ est une valeur indifférente prenant la place d'une quelconque des deux valeurs $\{t, f\}$.)

Ces expressions régulières peuvent être reconnues par des automates d´état fini exprimables en Lustre, ce qui nous fournit le noeud `SamePeriod_2` donnant la valeur de vérité d'une assertion.[4]

On peut, si besoin être encore plus précis en remplaçant partout 2 par n. Mais l'expérience montre que 2 suffit souvent. Nous le montrons ici.

9.3.3 Preuve du délai borné

Intuitivement, si les écritures et les lectures ont à peu près la même période,les valeurs lues ne devraient pas être trop éloignées de celles écrites. C'est ce qui est formalisé dans le théorème suivant :

Théorème 9.3.1 *Sous l'assertion* `Same_Period_2(cw,cr)`*, la propriété :*

[3] s'ils sont petits par rapport aux périodes de lecture ou d'écriture. Si ce n'est pas le cas, une modélisation plus fine est toujours possible, bien que plus complexe.

[4] Cela pourrait être engendré automatiquement grâce à l'outil Reglo de génération de reconnaisseurs d'expressions régulières [30].

```
up = mem(cw,v,u) when cr
prop =   mem(cr,v,up) = mem(cw,v,u)
       or mem(cr,v,up) = mem(cw,v, v -> pre u)
       or mem(cr,v,up) = mem(cw,v, v -> pre (v -> pre u))
```

est toujours vraie.

Preuve Ce « théorème » est prouvé automatiquement par Lesar [19].

Il dit que toute valeur utilisée a été produite dans un intervalle d'au plus deux périodes. C'est exactement ce qui est annoncé en 9.2.2 lorsqu'on prend $\mathbf{1}_r^{\downarrow} = \mathbf{1}_w^{\downarrow}$ et $n = 1$.

9.4 Programmes quasi-synchrones

Nous pouvons maintenant modéliser de façon fiable le comportement de programmes synchrones implantés sur une architecture quasi-synchrone. Cette modélisation peut servir pour la simulation et le test et même pour la vérification formelle [12]. La figure 9.3 montre le contrôleur de voie unique réparti sur trois calculateurs quasi synchrones, un pour l'entrée à gauche, un pour l'entrée à droite, et un pour la voie unique.

```
node dis_control_single_line(
     cll,clc,clr,
     line_busy, left_train_here, right_train_here, in_left_switch_cap,
     out_left_switch_cap, out_right_switch_cap, in_right_switch_cap :bool
     )
returns(
     left_traffic_light, right_traffic_light,  in_left_switch,
     out_left_switch, out_right_switch, in_right_switch: bool
     );
var priority, lth, lb, rth:bool;
let
   (priority, lb) = if clc
                    then current control_single_line_c(
                                 (line_busy, lth, rth)when clc
                                 )
                    else ((false, false)->pre(priority,lb ));

    ( left_traffic_light, in_left_switch, out_left_switch, lth)
    =
    if cll
    then current control_single_line_l(
                (lb, left_train_here,
                 in_left_switch_cap,
                 out_left_switch_cap,  priority) when cll)
    else ((false,false,false,false)
          -> pre(left_traffic_light, in_left_switch, out_left_switch, lth

    (right_traffic_light, in_right_switch, out_right_switch, rth)
    =
    if clr
    then current control_single_line_l(
                (lb, right_train_here,
                 in_right_switch_cap,
                 out_right_switch_cap,  not priority) when clr)
    else ((false,false,false,false)
          -> pre(right_traffic_light, in_right_switch, out_right_switch,
tel
```

FIG. 9.3 – Contrôleur de voie unique réparti

Chapitre 10

Systèmes continus et combinatoires

10.1 Systèmes continus

De la même façon qu'au chapitre 8, on se pose la question de comprendre comment l'architecture quasi-synchrone s'adapte aux systèmes continus.

L 'idée est que les fonctions calculées sont elles-mêmes uniformément continues (figure 10.1).

10.1.1 Systèmes uniformément continus

Définition 10.1.1 (Systèmes uniformément continus) *Un système S est UC s'il existe une fonction positive η_S des erreurs vers les erreurs telle que :*

$$\forall \epsilon > 0, \forall x, x', ||x - x'||_\infty \leq \eta_S(\epsilon) \Rightarrow ||(S\ x) - (S\ x')||_\infty \leq \epsilon$$

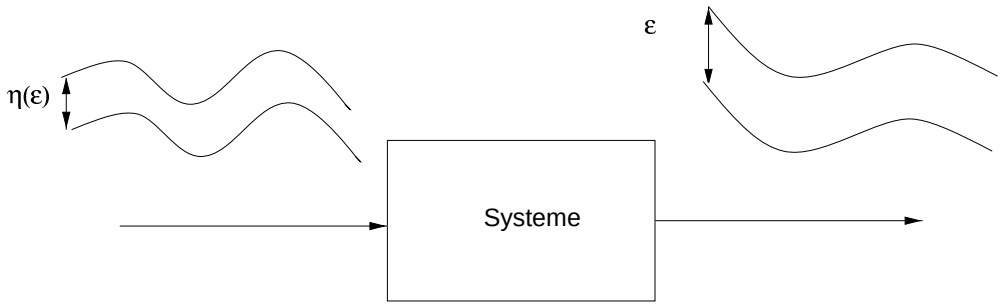


FIG. 10.1 – Système uniformément continu

Cette définition permet de poser la propriété importante : la continuité se propage dans les réseaux.

Théorème 10.1.1 (Propagation dans les réseaux) *Un système uniformément continu, stationnaire, alimenté par un signal uniformément continu émet un signal uniformément continu.*

Preuve :
Soit x UC, S UCS, et $\epsilon > 0$,

$$\forall x', ||x - x'||_\infty \leq \eta_S(\epsilon) \Rightarrow ||(S\ x) - (S\ x')||_\infty \leq \epsilon$$

et

$$\forall \tau, |\tau| \leq \eta_x(\eta_S(\epsilon)) \Rightarrow ||x - (\Delta^\tau\ x)||_\infty \leq \eta_S(\epsilon)$$

Donc,

$$\forall \tau, |\tau| \leq \eta_x(\eta_S(\epsilon)) \Rightarrow ||(S\ x) - (S\ (\Delta^\tau\ x)||_\infty \leq \epsilon$$

Mais $S(\Delta^\tau\ x) = \Delta^\tau(S\ x)$. On a donc

$$\eta_{Sx} = \eta_x \circ \eta_S$$

Fin.
Ce théorème permet donc de dire que, étant donné un réseau acyclique de fonctions, on peut trouver des bornes sur les délais entre fonctions et des bornes sur les erreurs en entrée telles que les erreurs en sortie soient dans des bornes données.

10.1.2 Systèmes en boucle fermée

Malheureusement, la situation est plus compliquée en réalité, car beaucoup de systèmes de commande ne sont pas uniformément continus. Par exemple, les PID très répandus n'ont pas cette propriété. Le terme intégral n'est pas stable et accumule les erreurs. L'erreur en sortie augmente indéfiniment lorsque le temps s'écoule (figure 10.2).

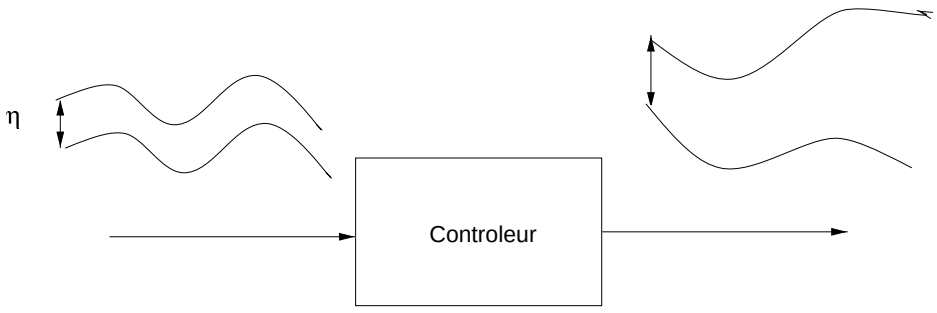


FIG. 10.2 – Contrôleur continu mais pas uniforme

En réalité, on peut récupérer la continuité uniforme grâce à la théorie de la stabilité. Le contrôleur non stable sert à stabiliser l'environnement de façon que le système en boucle fermée soit stable et donc uniformément continu (figure 10.3).

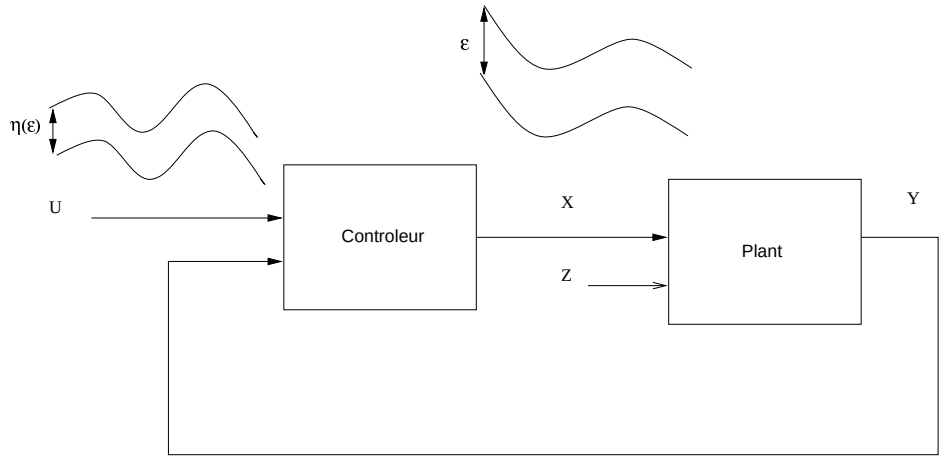


FIG. 10.3 – Système en boucle fermée

10.2 Retards incohérents

Nous avons donc vu comment, pour les systèmes continus, les délais produisent des erreurs qui se combinent au long des calculs. On peut, dans le cas de réseaux de calculs acycliques communiquant par des délais bornés, trouver des bornes supérieures sur ces délais et sur les erreurs des entrées pour que les erreurs en sortie soient bornées. Dans le cas de réseaux cycliques, le problème est plus compliqué et lié aux questions de stabilité. La question que l'on se pose ici est de savoir si on peut faire pareil avec les systèmes discontinus, au moins dans le cas des systèmes combinatoires et des réseaux acycliques.

Le problème que l'on rencontre est que les systèmes discrets n'ont pas d'erreurs. Il n'y a donc que des délais et les délais ne se combinent pas aussi bien que les erreurs. Cela est illustré à la figure 10.4 où l'on voit deux signaux booléens venant de deux calculateurs différents et arrivant sur un troisième calculateur. A cause de retards différents, celui-ci voit temporairement le couple de valeurs $x' = 0, y' = 1$ qui ne correspond à aucun couple de valeurs du couple d'origine. Cela peut se résumer par la propriété suivante :

Propriété 10.2.1 *Des retards sur les éléments d'un tuple peuvent ne pas correspondre à un couple retardé.*

Dans la suite, on va voir comment prendre en compte ce fait. Pour cela, il faut d'abord revenir sur la définition de la variabilité bornée que nous avons donné au chapitre 8.

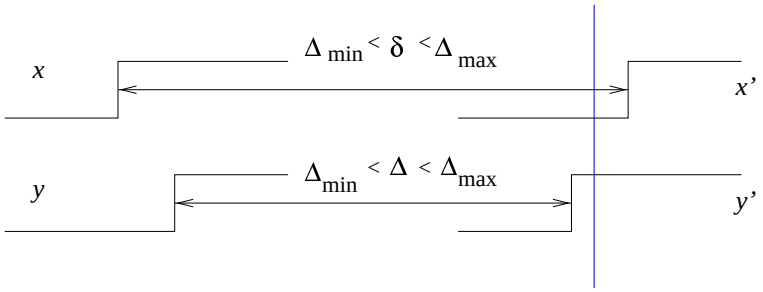


FIG. 10.4 – Des retards sur des tuples ne sont pas des tuples retardés

10.3 Variabilité uniforme

Le problème avec cette définition, c'est qu'elle ne se généralise pas facilement à des tuples de variables indépendantes : même si deux variables ne varient chacune qu'au plus tous les T, il n'y a pas de borne inférieure au temps de variation du tuple. On va s'en contenter et prendre la définition suivante :

Définition 10.3.1 (Variabilité uniformément bornée (VUB)) *Un signal, ou un tuple de signaux x est à variation uniformément bornée s'il existe un triplet T_x, θ_x, n_x tel que :*

$$\forall t, t', \quad |t - t'| \leq T_x \Rightarrow dc_{t,t'}(x) \leq n_x$$
$$\left.\begin{array}{c} |t - t'| \leq \theta_x \\ x(t) = x(t') \end{array}\right\} \Rightarrow dc_{t,t'}(x) = 0$$

Dans tout intervalle de durée pas plus grande que T_x, il n'y a pas plus de n_x points de discontinuité, et si deux points de même valeur sont séparés d'une durée pas plus grande que θ_x, alors il n'y a pas de points de discontinuité entre eux. Par exemple, un n-tuple de booléens x, ayant chacun le temps stable T donne : $T_x = T, \theta_x = T, n_x = n$. Notons que la dernière propriété a pour effet d'éliminer les parasites : si on fait passer un couple de variables indépendantes dans une fonction « ou exclusif » même si chaque signal varie très peu, l'écart entre deux fronts montants peut être très petit et la sortie du « ou exclusif » la montrer une variation temporaire très courte, qui peut être indétectable.

10.4 Quelques propriétés importantes

Théorème 10.4.1 (Tuples) *Un couple de signaux VUB $x, (T_x, \theta_x, n_x), y, (T_y, \theta_y, n_y)$ est un signal VUB (x, y) avec :*

$$T_{x,y} \quad = \quad \inf\{T_x, T_y\}$$

$$\begin{aligned}\theta_{x,y} &= \inf\{\theta_x, \theta_y\} \\ n_{x,y} &= n_x + n_y\end{aligned}$$

Théorème 10.4.2 (Temps de stabilité) *Dans tout intervalle de durée T_x, il existe un intervalle de durée au moins :*

$$\frac{T_x}{n_x + 1}$$

pendant lequel le signal reste constant.

Preuve :
En effet, n_x points de discontinuité partagent l'intervalle T_x en au plus n_x+1 intervalles et la somme de leurs durées est égale à T_x. Il est donc impossible qu'ils soient tous plus petits que :

$$\frac{T_x}{n_x + 1}$$

Fin.
Cette propriété est très importante parce que c'est elle qui va assurer le retard borné des fonctions de confirmation.

Théorème 10.4.3 (Temps de stabilité et retards)

$$T_{r\,x} = T_x - (r^M - r^m)$$

Théorème 10.4.4 (Retards et fonctions combinatoires) *Si f est combinatoire, alors,*

$$f(x, y) \circ r = f(x \circ r, y \circ r)$$

10.5 Fonctions de confirmation

Ces fonctions visent à transformer des retards incohérents en retards cohérents et vont permettre de répondre à la question suivante : étant donné x, r_x, y, r_y,peut-on trouver un r tel que :

$$f(x \circ r_x, y \circ r_y) = f(x, y) \circ r$$

Nous avons donc :
- des signaux VUB, $x, (T_x, \theta_x, n_x)$,
- des retards non déterministes mais bornés $x' = x \circ r_{x,x'}$
- des fonctions combinatoires,

et on cherche à calibrer ces retards comme on l'a fait pour les signaux et fonctions continues.

Définition 10.5.1 (Fonction de confirmation)

```
node Confirm(nmax:int; x0, x: tuple)
returns (z : tuple);
var n : int ;
let
  n, z = if x = x0 -> pre x
         then if n >= nmax - 1
              then 0, x
              else 0 -> pre n + 1, x0 -> pre z
         else 0, x0 -> pre z ;
tel
```

où x est un tuple de booléens et $x0$ est sa valeur initiale supposée connue. L'idée est de geler la sortie tant que l'entrée n'est pas restée constante pendant une durée

$$\tau = nmax * T_m$$

où T_m est la période minimum du calculateur.

Théorème 10.5.1 *Soit*

$$z = \texttt{Confirm}(\tau, (x_0, y_0), (x \circ r_x, y \circ r_y)) \tag{10.1}$$

$$\tau < \frac{\inf\{T_x - (r_x^M - r_x^m), T_y - (r_y^M - r_y^m)\}}{n_x + n_y + 1} \tag{10.2}$$

$$\tau > \sup\{r_x^M, r_y^M\} - \inf\{r_x^m, r_y^m\} \tag{10.3}$$

$$\tau > \inf\{\theta_x, \theta_y\} + \sup\{r_x^M, r_y^M\} - \inf\{r_x^m, r_y^m\} \tag{10.4}$$

Alors, il existe r_z tel que :

$$z = (x, y) \circ r_z \tag{10.5}$$

$$r_z^m \geq \tau + \inf\{r_x^m, r_y^m\} \tag{10.6}$$

$$r_z^M \leq (n_x + n_y + 1)\tau + \sup\{r_x^M, r_y^M\} \tag{10.7}$$

$$\theta_z = \tau \tag{10.8}$$

Ce théorème a été donné pour un couple de signaux mais il s'étend naturellement à des tuples quelconques. Il nous fournit donc un calcul permettant de relier les retards de communication avec les propriétés de variabilité uniforme des entrées et les paramètres des confirmateurs pour calculer les propriétés de variabilité bornée des sorties. Il s'utilise en général en sens inverse et permet de calculer des paramètres de confirmateurs, des bornes sur les délais de communication et les propriétés de variabilité bornée des entrées pour que les sorties de réseaux acycliques de fonctions combinatoires gardées par des confirmateurs aient des propriétés de variabilité bornée désirées. En ce sens, il apparaît comme un équivalent discontinu du théorème 10.1.1.

10.5.1 Preuve du théorème 10.5.1

Elle se fonde sur ces deux lemmes :

Lemme 10.5.1 *Si $\tau > r^M - r^m$ l'image par r de l'intervalle $[t-\tau, t]$ contient au moins deux valeurs différentes $t_1 = r(t-\tau)$ et $t_2 = r(t)$ et*

$$\begin{array}{rrl} t-\tau-r^M & \leq t_1 \leq & t-\tau-r^m \\ t-r^M & \leq t_2 \leq & t-r^m \\ & t_2 - t_1 \leq & \tau + r^M - r^m \end{array}$$

Lemme 10.5.2 *Si $\tau > \sup\{r_1^M - r_2^m, r_2^M - r_1^m\}$ les deux intervalles*

$$[r_1(t-\tau), r_1(t)] \ , \ [r_2(t-\tau), r_2(t)]$$

se recouvrent.

Preuve du théorème

(5) Pour prouver ce point, on construit r_z de la façon suivante :
- à $t_0 = 0$, $r_z(t_0) = 0$ garantit les conditions initiales.
- Aux instants t_{2i} tels que z prend une nouvelle valeur, parce que $x \circ r_x$ et $y \circ r_y$ sont restés constants pendant τ,à cause des lemmes précédents et grâce aux conditions (3) et (4), il existe certainement un instant t'_{2i} dans l'intervalle

  $$[t_{2i} - \tau - \inf\{r_x^m, r_y^m\}, t_{2i} - \sup\{r_x^M, r_y^M\}]$$

  tel que :

  $$(x,y)(t'_{2i}) = (x \circ r_x, y \circ r_y)(t_{2i})$$

  On choisit donc $r_z(t_{2i}) = t'_{2i}$.
- Soit t_{2i+1} le premier point de discontinuité à l'entrée du confirmateur succédant à une affectation de valeur à z. A cet instant, nous gelons r_z jusqu'à la fin de la prochaine période stable de durée τ. Entre temps, le temps progresse linéairement. Donc, on finit de construire la fonction r_z par :

$$r_z(t) = \begin{cases} t'_{2i} + (t - t_{2i}) & \texttt{si} \quad t \in [t_{2i}, t_{2i+1}] \\ r_z(t_{2i+1}) & \texttt{si} \quad t \in]t_{2i+1}, t_{2i+2}[\end{cases}$$

(6) est assuré aux points de discontinuité de z.

(7) est assuré parce que, grâce au théorème 10.4.2 et la condition (2), le plus grand retard de z survient lorsque une nouvelle discontinuité arrive à l'entrée du confirmateur juste avant que le délai τ n'expire, provoquant ainsi une nouvelle attente de durée τ. Mais cette situation ne peut pas se reproduire plus de $n_x + n_y$ fois de suite à cause de la variabilité bornée des entrées.

10.6 Exemple

Considérons le réseau acyclique :

$$f(g(x,y),z)$$

fait de deux fonctions combinatoires f et g où nous supposons que chaque entrée est d'arité 1 ($n_x = n_y = n_z = 1$). On veut trouver des bornes sur $T_x, T_y, T_z, \tau_f, \tau_g, r_x^M, r_y^M, r_z^M, r_g^M$[1] tel que, pour un $r^M = 30$ donné, il existe r tel que :

$$f(g(x,y),z) \circ r = f(C_{\tau_f}(g(C_{\tau_g}(x \circ r_x, y \circ r_y)) \circ r_g, z \circ r_z))$$

avec $r_x^m = r_y^m = r_z^m = r_g^m = 0$.

En supposant $r_G^M = 5 > r_z^M, n_G = 2$, on peut choisir $\tau_f = 5$ et trouver un r avec $r^M \leq 30$ tel que :

$$f(G,z) \circ r = f(C_5(G \circ r_G, z \circ r_z))$$

On peut ensuite décomposer r_G en $r_g \circ r'$ et choisir $r_g^M < 1$. Nous avons maintenant le sous-problème de trouver un r' avec $r'^M \leq 4$ tel que :

$$g(x,y) \circ r' = g(C_{\tau_g}(x \circ r_x, y \circ r_y))$$

En prenant $r_x^M = r_y^M < 1$ on peut choisir $\tau_g = 1$ qui résoud le problème. Il reste à trouver les temps de stabilité des signaux individuels. Heureusement, ils n'apparaissent qu'à droite des inégalités, ce qui permet de les choisir à volonté. On prendra par exemple $T_x = T_y = T_z > 40$.

Cet exemple montre cependant que ce système de mise en cohérence des retards est très coûteux en temps et performance. Il ne convient réellement qu'à des systèmes de dynamique très lente. Il semblerait que ce soit le cas des systèmes d'aiguillages, des centrales nucléaires et même des avions.

[1] Les θs ne sont pas importants ici parce qu'on suppose qu'il s'agit de booléens simples pour lesquels $\theta = T$.

Chapitre 11

Systèmes séquentiels robustes

11.1 Introduction

Dans le chapitre précédent, nous avons essaye de définir des caractéristiques de l'environnement physique qui permettent de répartir facilement les programmes de contrôle de tels environnements. Dans ce chapitre, nous nous intéressons plutot à des caractéristiques des programmes de contrôle, qui indépendamment de l'environnement, ont une certaine robustesse vis-à-vis de la répartition. En particulier, nous analysons les systèmes séquentiels pour lesquels il est difficile de compter sur les propriétés de l'environnement en vue de prévoir le comportement réparti.

Courses critiques Le problème avec ces systèmes est le phénomène de courses critiques [10, 32], illustré à la figure 11.1 :

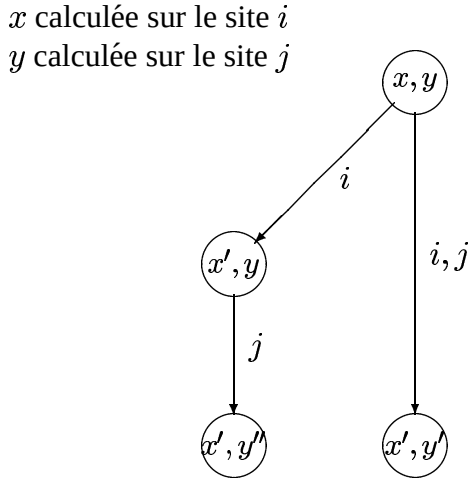


FIG. 11.1 – Une course critique

on y voit deux variables d'état calculées par des sites différents. Dans une approche synchrone, les deux variables sont calculées simultanément. Au contraire, lorsque chaque site a sa propre horloge, et si il y a une dépendance fonctionnelle entre elles, le résultat peut dépendre de l'ordre selon lequel elles sont calculées.
On peut donc voir les systèmes séquentiels comme ressemblant aux systèmes continus instables : il est difficile de les implanter de façon robuste. Il est donc important de vérifier l'absence de courses critiques.

Dans cette section, nous étudions les différentes façons de détecter l'absence de courses critiques. Nous présentons quelques critères d'absence de ces courses à se plaçant à des nivaux d'abstractions différents à savoir, des critères syntaxiques, semi-sémantiques ou sémantiques. Les critères les plus fins correspondent aux niveaux les plus détaillés.

11.2 Vérification de la robustesse des systèmes séquentiels

Dans cette section, nous étudions les différentes façons de détecter l'absence de courses critiques. Nous présentons quelques critères d'absence de ces courses se plaçant à des nivaux d'abstraction différents à savoir, des critères syntaxiques, semi-sémantiques ou sémantiques. Les critères les plus fins correspondent aux niveaux les plus détaillés.

Cette question a été étudiée dans le cadre du projet Crisys dans la thèse de Moez Yeddes [32].

11.2.1 Robustesse syntaxique

Premièrement, si les variables d'état calculées sur des sites distincts sont indépendantes, le phénomène de course critique ne peut pas se produire et on est ramené au cas combinatoire. Les dépendances entre ces variables peuvent être de deux types : les dépendances directes entre elles ou les dépendances par partage d'entrées communes.

11.2.2 Robustesse semi-sémantique

Une autre situation intéressante apparaît lorsque les variables d'état ne peuvent pas varier en même temps, que ce soit en synchrone ou en asynchrone. Alors, il n'y a pas de course critique non plus. Mais il y a deux raisons possibles pour que ces variables ne puissent pas changer en même temps :

Des raisons de temps. L'exemple du contrôleur d'aiguillage illustre bien ce phénomène : Lorsque un train à gauche passe le vert et rentre dans la voie unique, une course se fait entre les deux événements suivants :

– la mise au rouge du feu,
– et l'arrivée d'un nouveau train à gauche.

Si le train gagne la course, il trouve le feu vert et la collision ne peut être interdite. La seule façon que nous ayons trouvé d'éviter cette course est de garantir que le contrôleur va plus vite que le train et ferme le feu avant.

Raisons de causalité. On peut prendre deux exemples :

Dans le cas des aiguillages, le feux ne peut pas passer au vert en même temps que le train entre à gauche, parce qu'il faut que le train soit là avant que le feu ne passe au vert. *La présence du train est une cause du passage du feu au vert.*

Dans le cas de l'exclusion mutuelle de la figure 7.2, on peut transformer le programme non robuste en un programme robuste en décidant que :

y ne peut monter que lorsque z est descendu et inversement.

Insérer des chaînes de causalité est donc une façon de rendre les programmes robustes. Cela rappelle des styles de programmation asynchrones comme les *Message Sequence Charts* [28].

11.2.3 Robustesse sémantique

Enfin, la vérification sémantique consiste à vérifier que le cas de la figure 11.1 ne peut pas se produire. Dans sa thèse, Moez Yeddes a proposé des critères plus fins de vérification, fondés sur la localisation précise du calcul de ces informations.

11.3 Un outil de vérification de la robustesse

Un outil de vérification de la robustesse de programmes Lustre répartis est en cours de construction. Il adresse les niveaux syntaxiques et partiellement semi-sémantiques qui nous ont paru les plus intéressants, le niveau sémantique étant sans doute le plus complexe (il exige une énumération des états du programme global). Au niveau semi-sémantique, il n'aborde que les raisons de causalité. Nous n'avons pas trouvé de bonnes méthodes pour aborder les question de temps autre que des méthodes lourdes comme par exemple de demander à Lesar si ces deux variables peuvent varier en même temps.

Chaînes de causalité En revanche, les chaînes de causalité peuvent s'étudier de façon semi-sémantique, sans énumérer les états.

Dans sa thèse, Moez Yeddes se pose la question suivante : étant donné deux variables x et y, peuvent-elles être en course ? et il propose la solution suivante fondée sur la définition des conditions de Set :

Définition 11.3.1 (Set) *On appelle Set(x) une pré-condition nécessaire pour le passage de l'expression x de faux à vrai.*

et il en déduit le théorème :

Théorème 11.3.1 *Si*

$$\begin{aligned} Set(x)\ and\ Set(y) &= false \\ Set(not\ x)\ and\ Set(y) &= false \\ Set(x)\ and\ Set(not\ y) &= false \\ Set(not\ x)\ and\ Set(not\ y) &= false \end{aligned}$$

alors x et y ne peuvent pas changer en même temps.

L'avantage de cette idée est que, si les conditions Set sont approchées de façon assez larges, la vérification de ces conditions est peu coûteuse car elle ne demande pas de parcourir les états de l'automate. Cependant, cette méthode est parfois peu efficace parce qu'elle ne peut pas prendre en compte des raisonnements de concepteurs impliquant plusieurs événements (chaînes de causalité).

11.4 Application à l'exemple

Nous appliquons l'outil de vérification au contrôleur d'aiguillage réparti. Nous commençons par vérifier la robustesse syntaxique. Le programme s'avère robuste à ce niveau déja (programme séquentiel découplé et confluent, puisqu'il contient une seule variable d'état). Il n'est donc pas nécessaire de passer à la vérification semi-sémantique et sémantique.

Le programme se comporte donc comme un programme combinatoire. Il y a maintenant deux façons de procéder :

- équiper le programme de confirmateurs,
- ou le coupler avec une propriété robuste vis-à-vis du temps réel.

Comme nous avons vérifié le programme au chapitre 8 avec incertitude en entrée et sortie, nous savons déjà que la propriété est robuste, et nous pouvons l'implanter tel quel. Nous pouvons alors le simuler et même essayer de le re-vérifier en quasi-synchrone.

Simulation quasi-synchrone. La simulation intensive donne de bons résultats : il n'a pas été possible de mettre en évidence une violation des propriétés.

```
const n= 3;
node verif_safety (cll,clc,clr,line_busy, left_train_here,
     right_train_here,in_left_switch_cap, out_left_switch_cap,
     out_right_switch_cap,in_right_switch_cap: bool)
    returns (safety:bool);

var left_traffic_light, right_traffic_light, in_left_switch,
    out_left_switch, out_right_switch, in_right_switch: bool;

let  (left_traffic_light, right_traffic_light,  in_left_switch,
      out_left_switch, out_right_switch, in_right_switch)
     =
     dis_control_single_line(cll, clc, clr,
     line_busy, left_train_here, right_train_here, in_left_switch_cap,
     out_left_switch_cap, out_right_switch_cap, in_right_switch_cap);

-- assertions on clocks:

   assert    same_period(2,cll,clc) and
             same_period(2,clc,clr) and
             same_period(2,clr,cll);

-- assertions on sampling:

   assert     sample(n,line_busy,cll)
          and sample(n,left_train_here,cll)
          and sample(n,right_train_here,cll)
          and sample(n,in_left_switch_cap,cll)
          and sample(n,out_left_switch_cap,cll)
          and sample(n,out_right_switch_cap,cll)
          and sample(n,in_right_switch_cap,cll);
   assert     sample(n,line_busy,clc)
          and sample(n,left_train_here,clc)
          and sample(n,right_train_here,clc)
          and sample(n,in_left_switch_cap,clc)
          and sample(n,out_left_switch_cap,clc)
          and sample(n,out_right_switch_cap,clc)
          and sample(n,in_right_switch_cap,clc);
   assert     sample(n,line_busy,clr)
          and sample(n,left_train_here,clr)
          and sample(n,right_train_here,clr)
          and sample(n,in_left_switch_cap,clr)
          and sample(n,out_left_switch_cap,clr)
          and sample(n,out_right_switch_cap,clr)
          and sample(n,in_right_switch_cap,clr);

assert environment(
     line_busy, left_train_here, right_train_here, in_left_switch_cap,
     out_left_switch_cap, out_right_switch_cap, in_right_switch_cap,
     left_traffic_light, right_traffic_light, in_left_switch,
     out_left_switch, out_right_switch, in_right_switch);

safety = properties(
     line_busy, left_train_here, right_train_here, in_left_switch_cap,
     out_left_switch_cap, out_right_switch_cap, in_right_switch_cap,
     left_traffic_light, right_traffic_light, in_left_switch,
     out_left_switch, out_right_switch, in_right_switch);
tel
```

FIG. 11.2 – Programme de vérification quasi-synchrone

```
[olan]$
lesar distrain.lus verif_safety -v -diag -merge -states 10000000
Pollux Version 1.33a
start normalisation ... done
start minimal network generation .......... done (2132 -> 1211 nodes)
building bdds ... 192 (on 192)
computing relevant statevars ... done (157 on 164)
     state no : 1849225, remain : 4515433
107683.530u 126853.980s 65:12:34.03 8.4%        0+0k 0+0io 684pf+0w
[olan]$
```

FIG. 11.3 – Résultat de la vérification quasi-synchrone

Vérification quasi-synchrone. La figure 11.2 montre le programme de vérification quasi-synchrone.

Malheureusement, cette vérification échoue. La figure 11.3 montre que, au bout de trois jours, Lesar a exploré presque deux millions d'états sans pouvoir nier la propriété, mais il en reste encore au moins quatre millions et demi à explorer.

Cela donc nous conforte dans notre méthode de spécification de vérification synchrone et exécution quasi-synchrone.

Chapitre 12

Systèmes séquentiels non robustes

12.1 Introduction

Dans les chapitres précédents, les systèmes présentaient des propriétés permettant de les répartir de façon que chacune des parties tourne de manière presque autonome par rapport aux autres. Dans ce chapitre nous traitons des systèmes ne possédant pas cette particularité. Pour avoir le même comportement que le système centralisé, nous reproduisons l'exécution des différentes parties du système au sein d'une période synchrone globale . On retrouve donc les différentes étapes depuis l'acquisitions des entrées jusqu'à la sortie des résultats. Comme tout se passe en réparti et non plus en centralisé, des synchronisations sont nécessaires pour obtenir ces différentes étapes dans l'ordre. Le choix d'un facteur d'accélération convenable permet enfin de garantir que la durée de cette période synchrone globale est identique à celle du centralisé.

12.2 Protocole de répartition

Dans cette section nous décrivons la période synchrone en réparti. Nous déduisons les synchronisations nécessaires pour la cohérence de cette période et enfin nous présentons un algorithme de synchronisation des différentes parties du code réparti pour réaliser ces synchronisations.

12.2.1 Algorithme de synchronisation

Principe

1. chaque unité de contrôle maintient un compteur n servant à compter le nombre de périodes. Ces compteurs sont initialisés à 0. Ceci correspond à un état stable où l'unité n'a pas de nouvelles entrées ni de nouvel état et ne fait aucun calcul.
2. quand $n = 0$, si l'unité i remarque un changement de sa propre entrée ou état et

aucun changement des entrées et états des autres unités, elle diffuse ses changements et met son compteur $n = 3$.

3. quand $n = 0$, si l'unité i remarque un changement d'entrée ou d'état d'autres unités, elle met éventuellement son compteur $n = 2$ et diffuse ses éventuels changements propres.
4. à chaque exécution, n est décrémenté jusqu'a 0.
5. si $n = 1$, l'unité i exécute une transition.

Variables

Chaque unité i mémorise les variables suivantes :
- n pour compter le nombre de périodes,
- u, x entrées et variables d'état de l'unité,
- Ug, Xg : vecteur d'entrées global et vecteur de variables d'état global vus par l'unité,
- ug, xg : valeurs des entrées et variables d'état de l'unité diffusées sur le réseau.

Définition 12.2.1 (Synchronisation globale)

```
const u0 : input; x0 : state ; U0 : Input ; X0 : State;
node F(X: State , U: Input) returns (x: state);

node synchronize(Ug: Input; Xg : State; u: input)
     returns    (ug: input, xg: state);

var n : int; dif , Dif : bool;
        U : Input ; X : State ;
        x: state;
let
dif = not(u = u0 -> pre(ug)) or not (x = x0 -> pre(xg));
Dif = not(Ug = U0 -> pre(U)) or not (Xg = X0 -> pre(X));

n   = if ((0 -> pre(n)) = 0) and dif and not Dif
      then 3
      else if ((0 -> pre(n)) = 0) and Dif
           then 2
           else if ((0 -> pre(n)) > 0)
                then 0 -> pre(n)-1
                else 0 -> pre(n);

x   =  if (n = 1) then F(Xg, Ug) else (x0 -> pre(x));

ug, xg  =  if n > 1 then (x, u) else ((x0, u0) -> pre(xg, ug))
tel.
```

Propriétés de l'algorithme

Cet algorithme permet de considérer une nouvelle période synchrone répartie uniquement. Il est intrinsèquement robuste puisqu'il ne peut y avoir de blocage. En effet, les valeurs des partenaires sont uniquement lues (chantillonnes comme l'environnement). L'accélération demandée par cet algorithme est de l'ordre de 4 comparé au centralisé utilisant la même horloge.

12.3 Preuve de l'algorithme

La preuve se base sur le fait que les phases de sortie de résultats et de calcul sont strictement successives.

- Soient T_M, T_m les bornes maximales et minimales de l'horloge quasi-synchrone.
- τ, τ' des délais de calcul ou de transmission (non nuls),
- on suppose

$$T_M + \tau \quad < \quad 2T_m$$

- t l'instant de détection du premier changement d'entrée ou d'état par une unit quelconque.

Alors,

- $t + \tau + T_M + \tau$ est l'instant de la dernière écriture possible dans la mémoire correspondant à une unité qui s'exécute juste avant la première écriture $(t + \tau)$ (donc ne l'ayant pas vue) ; alors cette unité s'exécute encore après une période T et a besoin de τ pour écrire dans la mémoire,
- $t + \tau + 3T_M$ est l'instant au plus tard auquel cette unité exécute une transition ; ceci correspond à la dernière transition exécutée.
- $t + 3T$ est l'instant où l'unité ayant écrit en premier exécute une transition,
- $t + \tau + 2T_m$ est l'instant au plus tôt d'exécution d'une transition : il correspond à une unité qui aurait tourné juste après la première écriture ; cette unité attend juste deux périodes pour exécuter une transition.
- finalement, $t + \tau + 3T_m + \tau'$ est l'instant auquel cette unité pourra écrire de nouveau en mémoire ; c'est la nouvelle écriture au plus tôt en mémoire.

Ceci montre clairement que

- la première exécution d'une transition précède strictement la dernière écriture en mémoire et la dernière exécution précède strictement la première nouvelle écriture.
- Les phases d'acquisitions et d'exécutions de transitions sont alors disjointes.
- Ainsi, chaque unité s'exécute avec des entrées et des variables d'états cohérentes.

La figure 12.1 illustre bien la preuve.

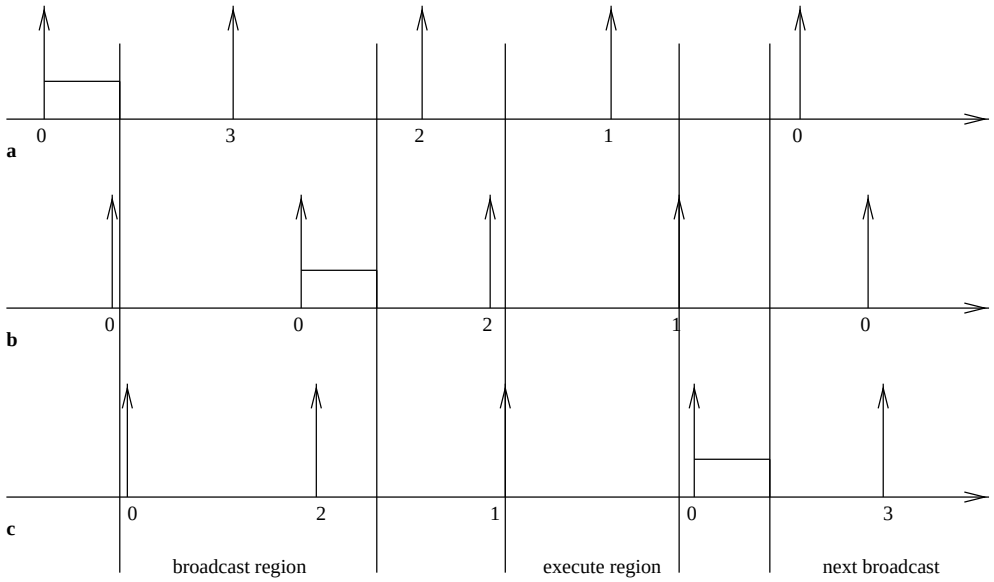


FIG. 12.1 – Illustration de la preuve : a) est la première unité à écrire, b) la dernière à exécuter , et c) la première à exécuter. Les nombres désignent les compteurs de chaque unité.

Chapitre 13

Construction de systèmes robustes

13.1 Introduction

Nous nous intéressons dans ce chapitre à la construction en Lustre de systèmes robustes pour la répartition, en particulier des systèmes ne présentant pas des courses critiques. Nous présentons la forme de ces systèmes . Nous déduisons ensuite une façon simple de les écrire en Lustre. Enfin, nous présentons une application de ce genre de constructions.

13.2 Descriptions des changements

Notons $x\uparrow$ le changement de la variable x de 0 à 1 et $x\downarrow$ le changement inverse.

$$\begin{array}{rcl} x\uparrow & = & \overline{pre(x)}.x \\ x\downarrow & = & pre(x).\overline{x} \end{array}$$

Remarquons que le calcul des fonctions $x\uparrow$ et $x\downarrow$ s'articule autour des termes $pre(x)$ et $\overline{pre(x)}$. Par ailleurs, toute variable x peut s'écrire sous la forme canonique

$$x = A.pre(x) + B.\overline{pre(x)}$$

Les changements s'écrivent alors

$$\begin{array}{rcl} x\uparrow & = & B \\ x\downarrow & = & \overline{A}. \end{array}$$

13.3 Absence de courses critiques

13.3.1 Condition nécessaire et suffisante

Rappelons que l'absence de courses critiques entre deux variables x et y s'exprime par les égalités suivantes :

$$x\uparrow.y\uparrow = x\downarrow.y\uparrow = x\uparrow.y\downarrow = x\downarrow.y\downarrow = faux$$

13.3.2 Systèmes vérifiant cette condition

Notre but est de construire en Lustre des systèmes robustes pour la répartition. Une telle construction se base sur la forme syntaxique des définitions des variables. Nous présentons donc des définitions de variables qui garantissent l'absence de courses critiques.

Partant des conditions d'absence de courses critiques, et sachant que l'obtention de la valeur $faux$ syntaxiquement est due uniquement à la multiplication de deux facteurs opposés écrits explicitement sous la forme F et $\overline{F}$, nous déduisons alors les formes suivantes des facteurs de changements :

$$\begin{array}{rcl} x\uparrow & = & \overline{\gamma}.\overline{\delta}.B \\ x\downarrow & = & \overline{A}.\overline{\alpha}.\overline{\beta} \\ y\uparrow & = & \alpha.\gamma.C \\ y\downarrow & = & \overline{D}.\delta.\beta \end{array}$$

Par identification, on déduit que les définitions des variables s'écrivent

$$\begin{array}{rcl} x & = & \overline{(\overline{A}.\overline{\alpha}.\overline{\beta})}.pre(x) + \overline{\gamma}.\overline{\delta}.B.\overline{pre(x)} \\ y & = & \overline{(\overline{D}.\delta.\beta)}.pre(y) + \alpha.\gamma.C.\overline{pre(y)} \end{array}$$

13.4 Construction de systèmes robustes avec des flip-flops

Remarquons que le facteur $x\downarrow$ correspond à la négation du facteur de $pre(x)$. Dans le cas où la négation n'est pas écrite explicitement, il n'est pas évident de pouvoir simplifier les produits de la condition d'absence de courses critiques.

La construction des systèmes robustes avec des flip-flops permet une description des variables sous la forme canonique et convenable pour la vérification. Les fonctions de base permettant la construction de tels systèmes sont décrites par les noeuds $flipflop$ et $lazyflipflop$.

Définition 13.4.1 (Flip-flops)

```
node flipflop   (set, reset : bool)
     returns    (x : bool);
let
   x = false -> (not(reset) and pre(x)) or (set and not(pre(x)));
tel
```

Remarquons que

$$\begin{array}{rcl} x\uparrow & = & \texttt{set and not pre x} \\ x\downarrow & = & \texttt{reset and pre x} \end{array}$$

Dans le second noeud, x ne change que lorsque $oracle$ vaut $vrai$.

Définition 13.4.2 (Lazy flips-flops)

```
node lazyflipflop (set, reset, oracle : bool)
     returns    (val : bool);
let
   x = flipflop(set and oracle, reset and oracle);

tel
```

Ces constructions permettent donc au programmeur de maîtriser les pré-conditions de changement de valeurs des variables du programme (transitions). On peut penser que cette maîtrise peut aider à construire des programmes robustes.

13.4.1 Chaînes de causalité

Lorsque l'on ne peut pas prouver directement l'absence de courses, l'idée est de se fonder sur des chaînes de causalité, éventuellement en intercalant des événements intermédiaires entre des événements que l'on veut séparer. Par exemple, pour séparer x et y, on peut penser intercaler z en créant la chaîne de causalité :

$$\ldots x \uparrow < y \uparrow < z \uparrow < y \downarrow < x \downarrow < z \downarrow < x \uparrow \ldots \tag{13.1}$$

où $<$ a le sens de « précède » ;

On pourrait penser programmer cette chaîne de la façon suivante :

```
px = false->pre x;
py = false->pre y;
pz = false->pre z;

x = flipflop(not pz, not py);
y = flipflop(px, pz);
z = flipflop(py, not px);
```

avec l'idée que l'on a ainsi traduit fidèlement la chaîne de causalité 13.1. En réalité, ce que l'on a écrit, c'est que y bas est une pré-condition de la descente de x. Mais la chaîne de causalité 13.1 exprime beaucoup plus que cela. En effet, la montée de z précède la descente de x qui, elle-même, précède la descente de z. Pour traduire fidèlement la chaîne, il faut donc aussi exprimer le fait que z haut est aussi une pré-condition de la montée de x. Pour pouvoir traduire pleinement la transitivité des relations de causalité, il faut donc prendre soin d'exprimer le plus possible de pré-conditions. Dans ce cas, on écrira donc :

```
px = false->pre x;
py = false->pre y;
pz = false->pre z;
```

```
x = flipflop((not py) and not pz, pz and not py);
y = flipflop(px and not pz, px and pz);
z = flipflop(py and px, (not px) and  not py);
```

Il est maintenant facile, par lecture directe des pré-conditions de montée et de descente de chaque variable de voir que tous les événements du système sont exclusifs. Il n'y a donc pas de course critique parce qu'il n'y a pas de courses du tout.

Par exemple, avant d'introduire z, on avait un problème parce que, quand x était haut et y bas, x pouvait descendre et y pouvait monter simultanément. maintenant, on a

$$\begin{aligned} Set(not\ x) &= \texttt{px and pz and not py} \\ Set(y) &= \texttt{px and (not pz) and not py} \end{aligned}$$

et ces deux propositions sont exclusives à cause de pz.

13.5 Application à l'exemple

Nous illustrons cette approche en montrant une programmation de l'exemple du chapitre 4 dans ce style de programmation.

Les principes suivis dans cette programmation sont les suivants :

- utiliser le plus possible de $flipflops$,
- connecter chaque nœud destiné à être réparti ou à modéliser l'environnement par des retards unité comme dans la figure 13.3. De cette façon, le code de ces nœuds sera exactement celui qui sera implanté. Cela nous permet d'éviter d'introduire des retards à l'intérieur de ces nœuds destinés à éviter des boucles de causalité. En effet ces retards ne seront plus utiles dans le code réparti quasi-synchrone, puisque les délais de communication éliminent les boucles de causalité.

Montrons maintenant comment cette programmation permet d'éliminer *par construction* certaines courses. L'exemple est celui d'une course entre `in_left_switch` produit par `control_single_line_l` et `in_left_switch_cap` produit par l'environnement.

On a naturellement les équations de l'environnement :

```
in_left_switch_cap = lazyflipflop (in_left_switch,
                                   not (in_left_switch),
                                   in_left_switch_o );
```

Pour assurer l'absence de course, on « ajoute » artificiellement[1] les pré-conditions :

[1]Cet ajout est artificiel parce que ces pré-conditions sont assurées de toutes façons par les autres pré-conditions. Les ajouter ne modifie donc pas le fonctionnement du programme mais facilite sa vérification.

```
in_left_switch   =  flipflop(...                          and
                            (not in_left_switch_cap )  and
     ... ,
                            ...        and
                            (in_left_switch_cap)  and
                            ...);
```

c'est à dire, on ne commande le switch en haut que s'il est bas et on ne l'abaisse que s'il est haut. Ainsi, l'absence de course est assurée.

Cette technique ne marche malheureusement pas toujours surtout entre l'environnement et les contrôleurs : on ne peut pas ajouter des pré-conditions sur l'environnement parce qu'on risque de le biaiser. Mais, dans ce cas, on peut toujours appliquer la technique de vérification centralisée avec incertitudes d'entrée et sortie présentée en 8.3.2 pour assurer la non-criticité des courses entre les contrôleurs et l'environnement.

```
node control_single_line_l(
     line_busy, left_train_here, in_left_switch_cap,
     out_left_switch_cap, priority,
     out_right_switch, in_right_switch,
     out_right_switch_cap, in_right_switch_cap: bool)
returns(
     left_traffic_light, in_left_switch,
     out_left_switch, lth: bool
     );

let

  in_left_switch   =  flipflop(priority and
                               (not line_busy)            and
                               (not out_right_switch)     and
                               (not out_right_switch_cap) and
                               (not in_right_switch)      and
                               (not in_right_switch_cap)  and
                               (not in_left_switch_cap )  and
       (not out_left_switch_cap) ,
                               (not priority)        and
                               (in_left_switch_cap)  and
                               (out_left_switch_cap) and
                               (not line_busy)       and
         not left_traffic_light);

  out_left_switch  =  in_left_switch  ;

  left_traffic_light = flipflop(    left_train_here
                                and (false -> pre left_train_here)
                                and priority
                                and (false->pre in_left_switch)
                                and (false->pre out_left_switch)
                                and in_left_switch_cap
                                and out_left_switch_cap
                                and not line_busy ,
                                line_busy and
edge(not left_train_here) );

  lth = left_train_here ;

tel
```

FIG. 13.1 – Contrôleur latéral

```
node control_single_line_c
  (line_busy, left_train_here, right_train_here : bool)
returns
  (left, right,  lb: bool);

var
  possible_change,
  one_train_here: bool;

let
  one_train_here  = left_train_here or right_train_here;

  possible_change =  edge(one_train_here and not line_busy);

  left = flipflop(possible_change and left_train_here,
                  possible_change and right_train_here );

  right= flipflop(possible_change and right_train_here and not left,
                  possible_change and left_train_here );


  lb = line_busy;
tel
```

FIG. 13.2 – Contrôleur central

```
node control_single_line
  (line_busy, left_train_here, right_train_here, in_left_switch_cap,
   out_left_switch_cap, out_right_switch_cap, in_right_switch_cap :bool
   )
returns
  (left_traffic_light, right_traffic_light,  in_left_switch,
   out_left_switch, out_right_switch, in_right_switch
   : bool
   );

var
  left, right,
  lth, rth, lb: bool;

let

  (left_traffic_light, in_left_switch, out_left_switch, lth) =

  control_single_line_l(lb, left_train_here, in_left_switch_cap,
                        out_left_switch_cap, false -> pre left,
                        false -> pre out_right_switch,
                        false -> pre in_right_switch,
                        false -> pre out_right_switch_cap,
                        false -> pre in_right_switch_cap);

  (right_traffic_light, in_right_switch, out_right_switch, rth) =

  control_single_line_l(lb, right_train_here, in_right_switch_cap,
                        out_right_switch_cap, false -> pre right,
                        false -> pre out_left_switch,
                        false -> pre in_left_switch,
                        false -> pre out_left_switch_cap,
                        false -> pre in_left_switch_cap);

  (left, right, lb) = control_single_line_c(line_busy, lth, rth);

tel
```

FIG. 13.3 – Architecture du contrôleur

```
node environment
  (left_traffic_light, right_traffic_light,
   in_left_switch, out_left_switch,
   out_right_switch, in_right_switch,
   left_train_o, right_train_o, central_train_o,
   in_left_switch_o, out_left_switch_o,
   out_right_switch_o, in_right_switch_o : bool)
returns
  (line_busy, left_train_here, right_train_here,
   in_left_switch_cap, out_left_switch_cap,
   out_right_switch_cap, in_right_switch_cap : bool)
let
  -- trains obey lights

  left_train_here = lazyflipflop (true,
                                  left_traffic_light,
                                  left_train_o );

  right_train_here = lazyflipflop (true,
                                   right_traffic_light,
                                   right_train_o );
  -- train do not go backward

  line_busy = flipflop (edge (not left_train_here)
                        or edge (not right_train_here),
                        central_train_o);

  -- switches move only if controlled

  in_left_switch_cap = lazyflipflop (in_left_switch,
                                     not (in_left_switch),
                                     in_left_switch_o );
  out_left_switch_cap = lazyflipflop (out_left_switch,
                                      not (out_left_switch),
                                      out_left_switch_o );
  in_right_switch_cap = lazyflipflop (in_right_switch,
                                      not (in_right_switch),
                                      in_right_switch_o );
  out_right_switch_cap = lazyflipflop (out_right_switch,
                                       not out_right_switch,
                                       out_right_switch_o );
tel
```

FIG. 13.4 – Environnement

Quatrième partie

Tolérance aux fautes

Chapitre 14

Tolérance aux fautes

14.1 Introduction

La question très importante de la tolérance aux fautes a été posée au cours du projet Crisys par un des examinateurs du projet, le professeur Kopetz de l'université technologique de Vienne. La tradition informatique, depuis les projets SIFT et FTMP [31, 22], pense que la seule voie de la haute sûreté de fonctionnement passe par des votes exacts et par conséquent par la synchronisation d'horloges et la résolution de problèmes de consensus [29, 15]. Cette approche a été systématisée par le professeur Kopetz dans la fameuse architecture dite « time-triggered » [25].

Cela nous a posé la question des bases théoriques des pratiques de nos partenaires dans ce domaine [9, 5]. Nous allons examiner successivement les cas des systèmes continus, combinatoires et séquentiels [13].

14.2 Votes à seuils

Nous avons vu que, pour les systèmes continus, l'approche quasi-synchrone permet de contrôler les erreurs dues aux incertitudes d'entrée et à la répartition. Cela permet de mettre au point des voteurs à seuils : deux unités ne sont déclarées discordantes que si leurs calculs diffèrent de plus de l'erreur maximum normale. Par exemple, un voteur majoritaire à seuil pourra être défini par :

Définition 14.2.1 (Voteur 2/3 à seuil)

```
const er : real ;
node voteur2/3(x1, x2, x3 :real)
     returns  (x : real, al : bool)
let
   x  = inf( sup(x1,x2), sup(x2,x3), sup(x3, x1));
   al = not (abs(x1 -x2) <= er or
             abs(x2 -x3) <= er or
```

```
            abs(x3 -x1) <= er) ;
tel
```

On calcule toujours la médiane, mais on fait une alarme dès qu'il n'y a pas au moins deux entrées dans la marge d'erreur.

14.3 Voteur à délais bornés

Le cas des fonctions combinatoires ressemble beaucoup à celui des fonctions continues, en remplaçant erreurs par délais. La théorie des confirmateurs nous permet de maîtriser les délais et de définir des voteurs à délais bornés : deux unités sont déclarés en désaccord si elles diffèrent pendant plus que le délai maximum normal.

14.3.1 Voteur simple

Considérons plusieurs copies d'un signal booléen x reçu par une unité de période maximum T_M, avec un délai maximum τ_x tel que $\tau_x + T_M < T_x$. Alors,

- l'intervalle maximum pendant lequel deux copies correctes peuvent différer est τ_x,
- le nombre maximum d'échantillons sur lesquels deux copies correctes peuvent différer est

$$nmax = E(\frac{\tau_x}{T_m}) + 1$$

où E est la fonction partie entière (inférieure).

Cela permet de concevoir des voteurs à délais bornés, pour signaux à délais bornés. Par exemple, un voteur 2/2 à délai borné pourrait être :

Définition 14.3.1 (voteur 2/2 à délai borné)

```
const nmax :int ; x0 : bool;

node voteur2/2(x1, x2 : bool)
     returns (x, al : bool);
var n :int ;
let
  x, al, n = if x1 = x2
             then x1, false -> pre al, 0
             else if 0 -> pre n < nmax -1
                  then x0 -> pre x, false -> pre al, 0 -> pre n + 1
                  else x0 -> pre x, true, nmax ;
tel
```

- ce voteur mémorise un compteur n initialisé à 0, sa précédente sortie avec une valeur initiale connue $x0$, et un signal d'alarme initialisé à faux.

- chaque fois que les deux entrés sont d'accord, il émet la valeur commune et remet à zéro le compteur et laisse l'alarme inchangée,
- sinon, si le compteur n'a pas atteint $nmax - 1$, il l'incrémente et émet la sortie précédente. L'alarme reste inchangée,
- sinon, il met l'alarme à vrai ; les autres variables sont alors indifférentes.

Théorème 14.3.1 *$voteur2/2$ émet une alarme dès que les deux entrées diffèrent pendant plus de $nmaxT_M$. Sinon, il délivre la valeur correcte avec un délai maximum égal à $(nmax + 1)T_M$.*

14.3.2 Cas des tuples

Pour voter sur des tuples, il faut combiner le précédent voteur avec les fonctions de confirmation. En effet, si on vote composante par composante, le tuple résultat peut ne jamais correspondre à une valeur de tuple cohérente.

Définition 14.3.2 (Voteur 2/2 à délai borné pour tuple)

```
const nmax : int; X0 : tuple;

node voteur2/2(X1, X2 : tuple)
     returns (X, al : bool);
var XP : tuple ;
let
  X = Confirm(nmax, X0, XP) ;
  for i = 1, n
     xpi, ali = voteur2/2(x1i, x2i);
  al = or (ali, i=1,n);
tel
```

En supposant que :
- $X_i, i = 1, 2$ sont des images à délais τ_x bornés du même tuple X, X,
- chaque composante de X a un temps de stabilité minimum T_x
- $nmax = E(\frac{\tau_x}{T_m}) + 1$
- $T_x > (n.nmax + 1)T_M$

on a le théorème :

Théorème 14.3.2 *$Voter2/2$ émet une alarme dès que deux copies d'une même composante diffèrent de plus de $nmaxT_M$. Sinon, il délivre une valeur correcte du tuple avec un délai maximum de $(n.nmax + 1)T_M$.*

Ce théorème est la combinaison des théorèmes 10.5.1 et 14.3.1.

14.4 Voteurs pour fonctions séquentielles

14.4.1 Approche générale

Une idée pour appliquer ce qui précède aux fonctions séquentielles serait de les transformer en fonctions combinatoires en extrayant la mémorisation d'état et en votant sur l´état aussi bien que sur les sorties.

Considérons un noeud dynamique :

```
node F(U : input) returns (S : output);
var X : local;
let
tel
```

On peut le transformer en un noeud combinatoire qui calcule la fonction de transition du précédent :

```
node TF(XSP : state; U : input) returns (XS : state);
let
tel
```

tel que l'équation Lustre :

```
XS = TF(XS0 -> pre XS, U);
```

fournisse une projection sur `S` égale à celle fournie par l'équation :

```
S = F(U);
```

L'idée serait alors de voter sur l'état :

```
XS1 = mem(cl1, XS0,
          TF(XS0->pre Vote((XS1,XS2,XS3)when cl1),
             Vote((U1,U2,U3)when cl1));
XS2 = mem(cl2, XS0,
          TF(XS0->pre Vote((XS1,XS2,XS3)when cl2),
             Vote((U1,U2,U3)when cl2));
XS3 = mem(cl3, XS0,
          TF(XS0->pre Vote((XS1,XS2,XS3)when cl3),
             Vote((U1,U2,U3)when cl2));
XS = Vote((XS1, XS2, XS3) when clv);
```

où `cl1, cl2, cl3` sont les horloges quasi-synchrones des unités répliquées et `clv` celle d'une unité devant utiliser le résultat `XS`.

Cette idée ne fonctionne cependant pas bien pour des raisons liées aux comportements byzantins [29] : on ne peut supposer qu'une unité défaillante va bien se tenir et par exemple rester silencieuse. Au contraire elle peut se comporter de façon maligne et induire en erreur ses partenaires. Ce n'est pas grave dans le cas combinatoire, parce

que les unités saines finiront par constater leur accord, mais, dans le cas séquentiel, c'est beaucoup plus grave : si l'unité 3 est défaillante, elle peut par exemple changer d'avis trop souvent et faire semblant d'être d'accord alternativement avec 1 puis 2 pendant que 1 et 2 sont en désaccord normal à cause de l'asynchronisme. Le voteur d'état interne de 2, voyant que 3 est d'accord avec lui, croira être majoritaire et fixera un nouvel état interne. Mais 1 pourra faire pareil avec sa propre valeur. Les deux unités bonnes stockeront des états internes différents et, à partir de là, pourront diverger. Une seule défaillance peut conduire à une perte de majorité, ce qui est insupportable.

14.4.2 Voteur 2/2 pour fonctions séquentielles

Heureusement, ce phénomène semble ne pas se produire pour les votes 2/2. Ceux-ci sont en général insensibles aux erreurs byzantines : une unité défaillante ne se compare qu'avec une unité bonne. Si elles sont d'accord, l'unité défaillante se comporte comme une bonne et on ne peut pas les distinguer. Si elles sont en désaccord, une alarme est levée et la défaillance est détectée au niveau du couple car on ne cherche pas à distinguer l'unité bonne de l'unité mauvaise. Cette faculté de ne pas être sensible aux comportements byzantins est peut-être la raison pour laquelle ce genre de stratégie est très populaire et utilisé par les partenaires du projet Crisys.

Pour construire ce voteur, on produit, à partir du noeud `TF` le noeud verb-TFSafe-de la façon suivante :

Définition 14.4.1 (Voteur2/2 pour fonctions séquentielles)

```
const nmaxU, nmaxX, nmaxXU : int; XS0 : state ; U0 : input;

node TFSafe(XSP : state ; alp : bool ; U1, U2 : input)
     returns (XS : state ; al : bool);
var XSI : state ;
    U , UP: input ;
    alu, alx : bool;
let
    UP, alu   = voteur2/2U (U1, U2) ;
    U         = Confirm(nmaxXU, U0, UP);
    XSI, alx  = voteur2/2X(XS0 -> pre XS, XSP)
    XS        = if XSI <> XS0 ->pre XSI or U <> U0 -> pre U
                then TF(XSI, U)
                else (XS0 -> pre XS);
    al       = alp or alx or alu;
tel
```

où `voteur2/2U, voteur2/2X` sont les voteurs combinatoires simples spécialisés avec les constantes respectives `nmaxU, U0, nmaxX, XS0`.

Ce noeud transformé s'utilise par paires quasi-synchrones :

```
  XS1, al1 = mem(cl1, (XS0, false),
```

```
                TFSafe((XS2, al2, U1, U2)when cl1));
XS2, al2 = mem(cl2, (XS0, false),
                TFSafe((XS1, al1, U1, U2)when cl2));
```

On voit ici le fonctionnement du noeud : il vote 2/2 sur les entrées, et il les retarde par un confirmateur ; il vote 2/2 sur son état interne et celui qu'il reçoit de son partenaire et il ne sollicite la fonction de transition que si l'un de ces deux termes change. On comprend bien ce fonctionnement si on se souvient que les voteurs 2/2 n'évoluent que si ils constatent un accord entre leurs arguments. Sinon, ils émettent le dernier accord constaté. Cela permet de s'assurer que, en absence de défaillance, les deux partenaires resteront bien synchronisés. On utilise ici les voteurs simples et non les voteurs pour tuples. En effet, pour les états, une copie vient du noeud lui même, et l'autre copie vient en une seule fois du partenaire. Il n'y a donc pas besoin de confirmation. Pour les entrées, la raison est différente : on a besoin de geler les entrées pendant un temps plus long que celui nécessaire pour mise en cohérence du tuple d'entrée. En effet, il ne faut pas que les entrées varient tant que le couple ne se soit pas mis d'accord sur un état interne.. On dissocie donc détection et confirmation.

On peut maintenant préciser les conditions de fonctionnement.

- $U_i, i = 1, 2$ sont des n_tuples dérivant d'un même U par des délais bornés par τ_u and $nmaxU = E(\frac{\tau_u}{T_m}) + 1$.
- le délai de transmission des états est borné par τ_X, and $nmaxX = E(\frac{\tau_X}{T_m}) + 1$.
- on suppose $\tau_X \leq \tau_u$
- $nmaxXU = n.(nmaxU + nmaxX) + 1$
- le temps de stabilité de chaque composante de U, T_u est supérieur $nmaxXU.T_M$.

Théorème 14.4.1 *En l'absence de fautes, les voteurs d'état délivrent un état correct avec un délai maximum* $(nmaxXU.T_M$ *Ils émettent une alarme si quelques composantes d'entrée sont en désaccord pendant plus de* $nmax_u T_M$ *ou si une des deux unités fonctionne incorrectement pendant plus de* $n.(nmax_u + nmax_X)T_M$.

Preuve :
La preuve est pas induction sur un chemin d'exécution :

Initialement les états et les entrées concordent.

On suppose un instant où les entrées et les états concordent. Par construction, rien ne peut changer tant que les entrées ne varient pas. Supposons un changement d'entrée. Il faut au plus $(nmaxU+1)T_M$ pour que chaque unité répercute correctement ce changement et au plus $(nmaxX + 1)T_M$ pour que les voteurs d'états répercutent correctement le nouvel état. Si, c'est le cas, on a retrouvé un état stable. Mais, il,se peut qu'entre temps une autre composante d'entrée change. Si l'accord sur le nouvel état ne s'est pas réalisé, aucun mal n'est fait parce que les voteurs d'état continuent d'émettre les anciennes valeurs. Si l'accord sur le nouvel état a été réalisé dans une unité mais pas dans l'autre, il sera certainement obtenu dan l'autre avant que le nouveau changement d'entrée ne soit pris en compte parce que les confirmateurs d'entrée gèlent les entrées suffisamment longtemps pour garantir un accord dans les deux unités. (cette partie de

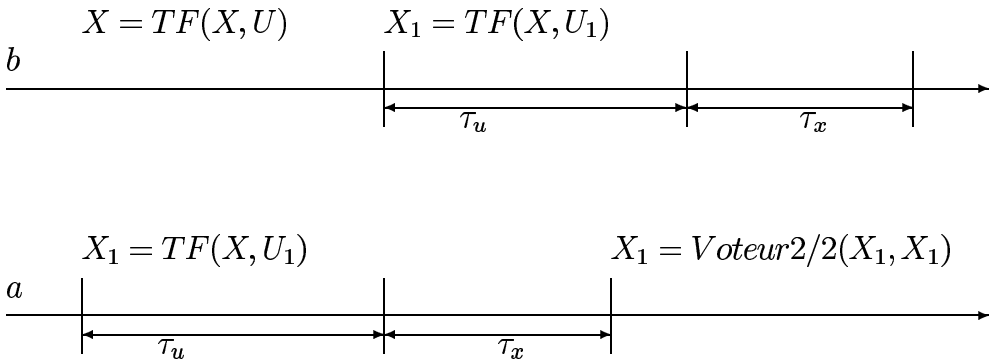


FIG. 14.1 – Illustration de la preuve : Le processus a voit l'entrée changer de U à U_1 le premier et calcule X_1. Ensuite, b voit le même changement et calcule aussi X_1. Ayant reçu X_1 de a, son voteur d'état se stabilise en X_1. Ensuite, le voteur d'état de a se stabilise en X_1 avant que l'entrée ne change à nouveau.

la preuve est illustrée à la figure 14.1). Par induction sur le nombre de composantes d'entrées, si un accord n'a pas été atteint avant, il sera obtenu lorsque toutes les composantes d'entrée auront changé car les entrées varient suffisamment lentement.

Enfin la partie alarme est claire par construction.
Fin.

Cette construction a été testée sur des exemples séquentiels simples. Des simulations quasi-synchrones étendues ont été menées avec satisfaction. Cependant, il faut remarquer que cette technique est coûteuse en temps et ne convient qu'à des processus très lents.

14.4.3 De la détection de fautes à la tolérance aux fautes

Les voteurs 2/2 permettent donc de détecter des défaillances et de construire des modules auto-vérifiés. Maintenant ces modules peuvent être combinés grâce à la redondance sélective pour construire des modules tolérant les fautes. Dans l'architecture de l'Airbus, cet objectif est atteint par une approche système globale : deux chaînes auto-vérifiés sont crées et une seule suffit à commander l'avion. Si une erreur est détectée, une des chaînes se suicide mais l'autre est encore capable de piloter l'avion.

Une autre possibilité est de créer des voteurs hybrides, mixtes, massifs et sélectifs : un système auto-vérifié primaire est actif jusqu'à une défaillance. Entre temps un système secondaire lit et vote sur les états du système primaire de façon à rester cohérent avec lui. Lorsque le système primaire défaille, le secondaire prend le contrôle de la situation s'il il est, lui-même encore correct.

Pour implanter cette solution, il faut d'abord résoudre le problème suivant : on bascule du primaire au secondaire lorsque un compteur de désaccord sature. Mais il est

possible que‘à ce moment, à cause de l’asynchronisme, le système secondaire ne soit pas lui-même déjà cohérent. Il ne faut pas alors lever une alarme dans le secondaire, mais au contraire lui laisser le temps de retrouver la cohérence. Le voteur2/2 à recalage suivant vise à résoudre le problème. $2/2$ voter :

Définition 14.4.2 (Voteur séquentiel $2/2$ à recalage)

```
const nmax :int ; x0 : bool;

node rvoteur2/2(x1, x2, reset : bool)
     returns (x, al : bool);
var n :int ;
let
  x, al = if x1 = x2
             then x1, false -> pre al
             else if 0 -> pre n < nmax -1
                  then x0 -> pre x, false -> pre al
                  else x0 -> pre x, true;
  n = if x1 = x2 or reset
      then 0
      else if 0 -> pre n < nmax -1
           then 0 -> pre n + 1
           else nmax ;
tel
```

On peur alors construire un voteur hybride :

Définition 14.4.3 (Voteur hybride $1/2x2/2$)

```
node voteur1/2x2/2(x1, x2, x3, x4 : bool)
     returns (x, al : bool) ;
var al1, reset, primary : bool ;
    xp1, xp2 : bool ;
let
  xp1, xp2 = if primary then (x1, x2) else (x3, x4) ;
  primary  = true -> pre (not al1) and primary ;
  reset    = (not primary) and (true -> pre primary) ;
  x, al1   = rvoteur2/2(xp1, xp2, reset);
  al       = al1 and (not primary);
tel
```

- $primary$ est initiallement vrai et le vote se faite sur les unités 1 et 2.
- Quand une défaillance survient dans une de ces deux unités, $primary$ devient faux pour toujours et le vote porte sur les unités 3 et 4.

14.5 Conclusion

On voit qu'on peut répondre affirmativement à la question du professeur Kopetz, ce qui conforte les pratiques des partenaires de Crisys. Cependant, on voit aussi, grâce au paramètrage des délais que ces techniques sont coûteuses en temps et conviennent à des processus très lents.

Chapitre 15

Conclusion

Nous avons donc présenté des méthodes et outils pour appliquer l'approche synchrone à la programmation de systèmes de contrôle-commande répartis :

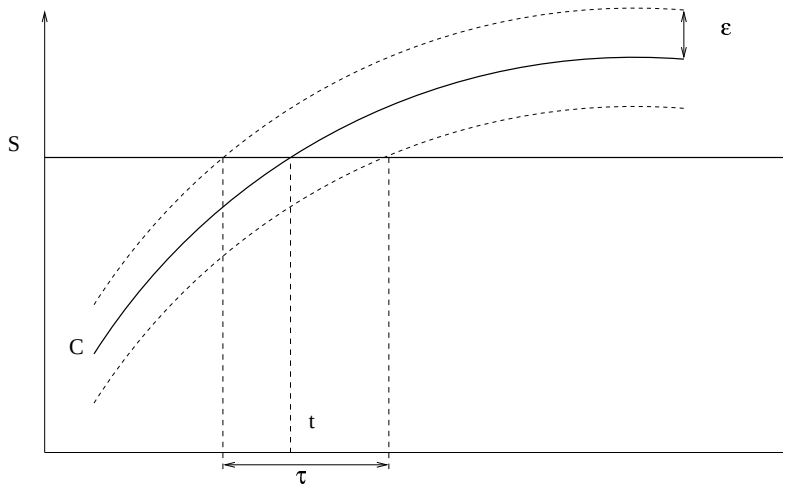
- nous avons proposé une façon de formaliser et simuler ces systèmes avec les outils de la programmation synchrone,
- nous avons proposé une théorie des confirmateurs, permettant de traiter les réseaux de fonctions combinatoires comme si elles étaient continues,
- nous avons esquissé une démarche de programmation robuste par construction pour les systèmes séquentiels répartis.
- nous avons esquissé une démarche de synchronisation logicielle dans le cas des systèmes séquentiels non robustes,
- nous avons proposé une approche de la tolérance aux fautes dans les systèmes répartis quasi-synchrones,

On peut remarquer que ce travail est loin d'être fini. Beaucoup d'aspects de ce travail n'ont pas pu être vraiment expérimentés et beaucoup de problèmes restent à étudier :

- Un problème important est celui des systèmes mixtes, continus/discrets. La figure 15.1 montre le cas d'un booléen obtenu par le franchissement d'un seuil par un signal continu. On voit que l'incertitude sur le signal, supposée bornée a pour effet de créer une incertitude sur la date du franchissement du seuil. Mais cette incertitude dépend de la dérivée du signal au point de franchissement. Si cette dérivée peut devenir aussi petite que l'on veut, ce délai peut ne pas être borné supérieurement et la démarche actuelle ne peut pas s'appliquer.
- Un autre problème est celui de prendre en compte les raisonnements temporels dans la preuve d'absence de course critique.

Plus généralement, il manque une théorie de la robustesse comprenant celle des systèmes continus et prenant en compte les aspects cités ci-dessus.

Remerciements. Nous remercions nos collègues du projet Crisys, P. Caspi de Verimag, M. Yeddes et R. David du LAG, C. Bodennec, C. Mazuet et N. Raynaud de Schneider Electric, R. Budde et A. Poigné de GMD et R. Mercadié de EADS-Airbus,



$$\tau = \frac{2\epsilon}{|\frac{dC}{dt}(t)|}$$

FIG. 15.1 – Franchissement de seuil

pour leur aide et contributions à ce travail.

Bibliographie

[1] R. Alur and D.L.Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[2] C. André. Representation and analysis of reactive behaviors : a synchronous approach. In *Proc. CESA'96*, Lille, July 1996.

[3] A. Benveniste, B. Caillaud, and P. Leguernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 162–177. Springer Verlag, 1999.

[4] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations : the SIGNAL language and its semantics. *Science of Computer Programming*, 16 :103–149, 1991.

[5] J.-L. Bergerand and E. Pilaud. Saga : A software development environment for dependability in automatic control. In *IFAC-SAFECOMP'88*. Pergamon Press, 1988.

[6] J.L. Bergerand and E. Pilaud. SAGA ; a software development environment for dependability in automatic control. In *SAFECOMP'88*. Pergamon Press, 1988.

[7] G. Berry and G. Gonthier. The ESTEREL synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[8] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process algebras and systems of communicating processes. In *Automatic Verification For Finite States Systems*, volume 407 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

[9] D. Brière, D. Ribot, D. Pilaud, and J.L. Camus. Methods and specification tools for Airbus on-board systems. In *Avionics Conference and Exhibition*, London, December 1994. ERA Technology.

[10] J. A. Brzozowski and C-J. H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.

[11] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3) :416–427, 1999. Research report INRIA 3491.

[12] P. Caspi, C. Mazuet, R. Salem, and D. Weber. Formal design of distributed control systems with Lustre. In *Proc. Safecomp'99*, September 1999.

[13] P. Caspi and R. Salem. Threshold and bounded-delay voting in critical control systems. In Mathai Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science*, pages 68–81, September 2000.

[14] A. Chatha. Fieldbus : The foundation for field control systems. *Control Engineering*, pages 47–50, May 1994.

[15] M.J. Fisher, N.A. Lynch, and M.S. Patterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2) :374–382, 1985.

[16] A. Girault. *Sur la répartition de programmes synchrones*. Thèse de doctorat, Institut National Polytechnique de Grenoble, janvier 1994.

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[18] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9) :785–793, september 1992.

[19] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[20] D. Harel. Statecharts : a visual approach to complex systems. *Science of Computer Programming*, 8(3), 1987.

[21] C.A.R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8) :666–676, 1978.

[22] A.H. Hopkins, T. Basil Smith, and J.H. Lala. FTMP :a highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, 66(10) :1221–1239, 1978.

[23] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74*. North Holland, 1974.

[24] K.J.Åström and B.Wittenmark. *Computer Controlled Systems*. Prentice-Hall, 1984.

[25] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems : the MARS approach. *IEEE Micro*, 9(1) :25–40, 1989.

[26] G. LeGoff. Using synchronous languages for interlocking. In *First International Conference on Computer Application in Transportation Systems*, 1996.

[27] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *Proc. of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*. Springer Verlag, August 1992.

[28] S. Mauw. The formalization of message sequence charts. *Computer Networks and ISDN Systems*, 28 :1643–1657, 1996.

[29] M. Pease, R.E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2) :228–237, 1980.

[30] P. Raymond. Recognizing regular expressions by means of dataflows networks. In *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)* Paderborn, Germany. LNCS 1099, Springer Verlag, July 1996.

[31] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Lewitt, P.M. Melliar-Smith, R.E Shostak, and Ch.B. Weinstock. SIFT : Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10) :1240–1255, 1978.

[32] M. Yeddes. *Contribution à une approche robuste pour la distribution de systèmes synchrones*. Thèse de doctorat, Institut National Polytechnique de Grenoble, novembre 2000.