

Programmation synchrone des systèmes réactifs : Application du langage synchrone LUSTRE

Alexandre Cortier

ONERA - Office National d'étude et de Recherche Aérospatiales

`alexandre.cortier@oncert.fr`

Cours - 2005/2006

Objectif du cours

- ❶ Exposer le principe des Techniques Formelles (TF).
- ❷ Exposer les principes de la programmation réactive à travers le langage synchrone Lustre, TF pour le temps réel..
- ❸ Présenter la théorie et la pratique de la preuve sur programme : Utilisation de Lesar, un model-checker pour Lustre.
- ❹ 3 séances de TPs sur Lustre : programmation d'un système de contrôle-commandes avec embarquement de code sur le noyau temps réel Vx-Works : étude d'un bras manipulateur 2 axes.

Objectif du cours

- ❶ Exposer **le principe des Techniques Formelles (TF)**.
- ❷ Exposer les **principes de la programmation réactive** à travers le **langage synchrone Lustre**, TF pour le temps réel..
- ❸ Présenter la théorie et la pratique de la **preuve sur programme** : Utilisation de **Lesar**, un model-checker pour Lustre.
- ❹ 3 séances de TPs sur Lustre : programmation d'un système de contrôle-commandes avec embarquement de code sur le noyau temps réel Vx-Works : étude d'un bras manipulateur 2 axes.

Objectif du cours

- ❶ Exposer **le principe des Techniques Formelles (TF)**.
- ❷ Exposer les **principes de la programmation réactive** à travers le **langage synchrone Lustre**, TF pour le temps réel..
- ❸ Présenter la théorie et la pratique de la **preuve sur programme** : Utilisation de **Lesar**, un model-checker pour Lustre.
- ❹ 3 séances de TPs sur Lustre : programmation d'un système de contrôle-commandes avec embarquement de code sur le noyau temps réel Vx-Works : étude d'un bras manipulateur 2 axes.

Objectif du cours

- ❶ Exposer le **principe des Techniques Formelles (TF)**.
- ❷ Exposer les **principes de la programmation réactive** à travers le **langage synchrone Lustre**, TF pour le temps réel..
- ❸ Présenter la théorie et la pratique de la **preuve sur programme** : Utilisation de **Lesar**, un model-checker pour Lustre.
- ❹ 3 séances de TPs sur Lustre : programmation d'un système de contrôle-commandes avec embarquement de code sur le noyau temps réel Vx-Works : étude d'un bras manipulateur 2 axes.

Objectif du cours

- ❶ Exposer le **principe des Techniques Formelles (TF)**.
- ❷ Exposer les **principes de la programmation réactive** à travers le **langage synchrone Lustre**, TF pour le temps réel..
- ❸ Présenter la théorie et la pratique de la **preuve sur programme** : Utilisation de **Lesar**, un model-checker pour Lustre.
- ❹ 3 séances de TPs sur Lustre : programmation d'un système de contrôle-commandes avec embarquement de code sur le noyau temps réel Vx-Works : étude d'un bras manipulateur 2 axes.

Table des Matières

- 1 Introduction
- 2 Introduction : Techniques Formelles
- 3 Rappel : Programmation Temps Réel classique
- 4 Programmation Réactive
- 5 LUSTRE : les principes de base
- 6 LUSTRE : détail du langage
- 7 Vérification formelle en Lustre
- 8 Flots et Horloges en pratique
- 9 Exemple complet : le chronomètre STOP_WATCH

Historique

Lustre est un langage de spécification particulièrement adapté pour la conception des systèmes de contrôle-commande temps réel.

- Début de sa conception en 1984 dans le Laboratoire de recherche **VERIMAG** à Grenoble... Cocorico !
- But : simplifier et automatiser l'implémentation de lois de commandes
- Origine : Notation graphique permettant la représentation de circuits logiques en automatique.
- Coopération : informaticiens et automaticiens ont mis la main à la pâte pour formaliser le tout...

Historique

Lustre est un langage de spécification particulièrement adapté pour la conception des systèmes de contrôle-commande temps réel.

- Début de sa conception en 1984 dans le Laboratoire de recherche **VERIMAG** à Grenoble... Cocorico !
- But : simplifier et automatiser l'implémentation de lois de commandes
- Origine : Notation graphique permettant la représentation de circuits logiques en automatique.
- Coopération : informaticiens et automaticiens ont mis la main à la pâte pour formaliser le tout...

Historique

Lustre est un langage de spécification particulièrement adapté pour la conception des systèmes de contrôle-commande temps réel.

- Début de sa conception en 1984 dans le Laboratoire de recherche **VERIMAG** à Grenoble... Cocorico !
- But : simplifier et automatiser l'implémentation de lois de commandes
- Origine : Notation graphique permettant la représentation de circuits logiques en automatique.
- Coopération : informaticiens et automaticiens ont mis la main à la pâte pour formaliser le tout...

Historique

Lustre est un langage de spécification particulièrement adapté pour la conception des systèmes de contrôle-commande temps réel.

- Début de sa conception en 1984 dans le Laboratoire de recherche **VERIMAG** à Grenoble... Cocorico !
- But : simplifier et automatiser l'implémentation de lois de commandes
- Origine : Notation graphique permettant la représentation de circuits logiques en automatique.
- Coopération : informaticiens et automaticiens ont mis la main à la pâte pour formaliser le tout...

LUSTRE et l'industrie

En 1993, Lustre a été transféré dans le milieu industriel avec grand succès.

Il est aujourd'hui intégré dans l'outil industriel **SCADE**, fourni par ESTEREL-Technologies, pour le développement de *systèmes de controle-commandes critiques*.

SCADE est notamment utilisé par **Airbus, Schneider Electric, Eurocopter...**

LUSTRE et l'industrie : SCADE 1

L'outil SCADE 1 est composé de :

- ① **Un langage textuel** : Lustre
 - Langage formel pour système réactif synchrone ;
- ② **Un langage graphique** : SCADE
 - Equivalence sémantique : $\text{Lustre} \equiv \text{SCADE}$;
 - Adapté aux flots de données ;
- ③ **Un atelier logiciel** :
 - Editeur graphique ;
 - Simulateur outil de preuve ;
 - Générateur de documentation, de code ;

LUSTRE et l'industrie : SCADE 2

Depuis la version 4 :

① Ajout des StateCharts d'ESTEREL :

- Toujours système réactif synchrone ;
- Equivalence : SCADE "pur" \equiv StateCharts ;

② En pratique :

- SCADE " pur" \equiv Flot de données ;
- StateCharts : système évènementiel ;
 - emission / réception d'évènements ;
 - possibilité de modéliser des automates temporisés ;

Table des Matières

- 1 Introduction
- 2 Introduction : Techniques Formelles
 - Qu'est-ce qu'une Technique Formelle ?
 - Classification des Techniques Formelles
 - Systèmes de preuves
 - Model Checker : Fonctionnement
 - Theorem Prover : Fonctionnement
 - Lustre + Lesar = Technique Formelle
- 3 Rappel : Programmation Temps Réel classique
- 4 Programmation Réactive
- 5 LUSTRE : les principes de base
- 6 LUSTRE : détail du langage
- 7 Vérification formelle en Lustre
- 8 Flots et Horloges en pratique
- 9 Exemple complet : le chronomètre STOP_WATCH

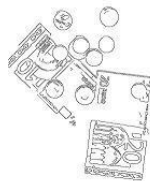
Notion intuitive de la notion de modèle



Abstraction



Modélisation



Code C

Système
à modéliser
(ex : distributeur)

Abstraction



Modélisation

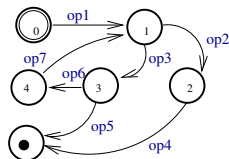


FIG.: Notion intuitive. Modèle = abstraction

Tests Vs Techniques Formelles

L'ambition initiale des techniques formelles : supprimer les phases de tests lors de la conception des programmes.

- Tests **coûteux** en temps et donc en... argent
 - Tests **fastidieux** pour le développeur...
 - Le test n'est **jamais exhaustif** : il ne peut pas vérifier toutes les exécutions possibles du système...
-
- On travaille sur un **modèle du système** : c'est une abstraction et l'on ne travaille donc pas sur le monde réel...
 - **Utilisation difficile** des Theorem Prover pour les néophytes.
 - **Problème d'explosion combinatoire** pour le Model Checking.

Tests Vs Techniques Formelles

L'ambition initiale des techniques formelles : supprimer les phases de tests lors de la conception des programmes.

- Tests **coûteux** en temps et donc en... argent
 - Tests **fastidieux** pour le développeur...
 - Le test n'est **jamais exhaustif** : il ne peut pas vérifier toutes les exécutions possibles du système...
-
- On travaille sur un **modèle du système** : c'est une abstraction et l'on ne travaille donc pas sur le monde réel...
 - **Utilisation difficile** des Theorem Prover pour les néophytes.
 - **Problème d'explosion combinatoire** pour le Model Checking.

Tests Vs Techniques Formelles

L'ambition initiale des techniques formelles : supprimer les phases de tests lors de la conception des programmes.

- Tests **coûteux** en temps et donc en... argent
- Tests **fastidieux** pour le développeur...
- Le test n'est **jamais exhaustif** : il ne peut pas vérifier toutes les exécutions possibles du système...

Mais c'était illusoire : l'utilisation des techniques formelles ne peut remplacer totalement les activités de tests.

- On travaille sur un **modèle du système** : c'est une abstraction et l'on ne travaille donc pas sur le monde réel...
- **Utilisation difficile** des Theorem Prover pour les néophytes.
- **Problème d'explosion combinatoire** pour le Model Checking.

Tests Vs Techniques Formelles

L'ambition initiale des techniques formelles : supprimer les phases de tests lors de la conception des programmes.

- Tests **coûteux** en temps et donc en... argent
- Tests **fastidieux** pour le développeur...
- Le test n'est **jamais exhaustif** : il ne peut pas vérifier toutes les exécutions possibles du système...

Mais c'était illusoire : l'utilisation des techniques formelles ne peut remplacer totalement les activités de tests.

- On travaille sur un **modèle du système** : c'est une abstraction et l'on ne travaille donc pas sur le monde réel...
- **Utilisation difficile** des Theorem Prover pour les néophytes.
- **Problème d'explosion combinatoire** pour le Model Checking.

Technique Formelle : définition rigoureuse

Définition : Technique Formelle

On appelle Technique Formelle (TF), l'association :

- 1 d'un *paradigme mathématique*, ou un langage de spécification munis d'une syntaxe et d'une sémantique formelle. Il permet la réalisation d'un **modèle** formelle du système.
- 2 d'un *système de preuve*.

Technique Formelle : définition rigoureuse

Définition : Technique Formelle

On appelle Technique Formelle (TF), l'association :

- ① d'un *paradigme mathématique*, ou un langage de spécification munis d'une syntaxe et d'une sémantique formelle. Il permet la réalisation d'un **modèle** formelle du système.
- ② d'un *système de preuve*.

Technique Formelle : définition rigoureuse

Définition : Technique Formelle

On appelle Technique Formelle (TF), l'association :

- ① d'un *paradigme mathématique*, ou un langage de spécification munis d'une syntaxe et d'une sémantique formelle. Il permet la réalisation d'un **modèle** formelle du système.
- ② d'un *système de preuve*.

Technique Formelle : définition rigoureuse

Définition : Technique Formelle

On appelle Technique Formelle (TF), l'association :

- ① d'un *paradigme mathématique*, ou un langage de spécification munis d'une syntaxe et d'une sémantique formelle. Il permet la réalisation d'un **modèle** formelle du système.
- ② d'un *système de preuve*.

Le système de preuve **permet la démonstration des propriétés relevantes du système**. Il travaille sur un modèle du système. La preuve sur programme permet de s'assurer que le modèle que nous avons réalisé se comporte de la bonne façon, c'est à dire qu'il vérifie des propriétés exigées, qu'il **n'a pas de bugs**.

Classification des Techniques Formelles suivant la sémantique

Suivant la sémantique adoptée nous pouvons distinguer deux types de spécifications formelles :

- ① *Approche basée sur modèles* ou *approche constructive, opérationnelle, basée sur état explicite* :
 - se fonde sur une sémantique à états (Automates...)
 - l'état est modifié par les opérations
 - les opérations modélisent l'aspect comportemental du système
 - Les propriétés sont exprimées par des expressions logiques sur les variables du système
 - \Rightarrow STE, Réseaux de Petri, B...
- ② *Approche algébrique* ou *fonctionnelle* :
 - sémantique définie par une algèbre
 - variables et opérations sont définies suivant cette algèbre
 - Le système est décrit par un ensemble d'équations
 - Ensemble d'équations = description du comportement
 - \Rightarrow LOTOS, LUSTRE...

Classification des Techniques Formelles suivant la sémantique

Suivant la sémantique adoptée nous pouvons distinguer deux types de spécifications formelles :

① *Approche basée sur modèles* ou *approche constructive, opérationnelle, basée sur état explicite* :

- se fonde sur une sémantique à états (Automates...)
- l'état est modifié par les opérations
- les opérations modélisent l'aspect comportemental du système
- Les propriétés sont exprimées par des expressions logiques sur les variables du système
- \Rightarrow STE, Réseaux de Petri, B...

② *Approche algébrique* ou *fonctionnelle* :

- sémantique définie par une algèbre
- variables et opérations sont définies suivant cette algèbre
- Le système est décrit par un ensemble d'équations
- Ensemble d'équations = description du comportement
- \Rightarrow LOTOS, LUSTRE...

Classification des Techniques Formelles suivant la sémantique

Suivant la sémantique adoptée nous pouvons distinguer deux types de spécifications formelles :

- ① *Approche basée sur modèles* ou *approche constructive, opérationnelle, basée sur état explicite* :
 - se fonde sur une sémantique à états (Automates...)
 - l'état est modifié par les opérations
 - les opérations modélisent l'aspect comportemental du système
 - Les propriétés sont exprimées par des expressions logiques sur les variables du système
 - \Rightarrow STE, Réseaux de Petri, B...
- ② *Approche algébrique* ou *fonctionnelle* :
 - sémantique définie par une algèbre
 - variables et opérations sont définies suivant cette algèbre
 - Le système est décrit par un ensemble d'équations
 - Ensemble d'équations = description du comportement
 - \Rightarrow LOTOS, LUSTRE..

Classification des Techniques Formelles suivant la sémantique

Suivant la sémantique adoptée nous pouvons distinguer deux types de spécifications formelles :

- ① *Approche basée sur modèles* ou *approche constructive, opérationnelle, basée sur état explicite* :
 - se fonde sur une sémantique à états (Automates...)
 - l'état est modifié par les opérations
 - les opérations modélisent l'aspect comportemental du système
 - Les propriétés sont exprimées par des expressions logiques sur les variables du système
 - \Rightarrow STE, Réseaux de Petri, B...
- ② *Approche algébrique* ou *fonctionnelle* :
 - sémantique définie par une algèbre
 - variables et opérations sont définies suivant cette algèbre
 - Le système est décrit par un ensemble d'équations
 - Ensemble d'équations = description du comportement
 - \Rightarrow LOTOS, LUSTRE..

Classification des Techniques Formelles suivant la sémantique

Suivant la sémantique adoptée nous pouvons distinguer deux types de spécifications formelles :

- ① *Approche basée sur modèles* ou *approche constructive, opérationnelle, basée sur état explicite* :
 - se fonde sur une sémantique à états (Automates...)
 - l'état est modifié par les opérations
 - les opérations modélisent l'aspect comportemental du système
 - Les propriétés sont exprimées par des expressions logiques sur les variables du système
 - \Rightarrow STE, Réseaux de Petri, B...
- ② *Approche algébrique* ou *fonctionnelle* :
 - sémantique définie par une algèbre
 - variables et opérations sont définies suivant cette algèbre
 - Le système est décrit par un ensemble d'équations
 - Ensemble d'équations = description du comportement
 - \Rightarrow LOTOS, LUSTRE..

Systèmes de preuves - Historique

- ① **Les débuts** : travail des logiciens cherchant à formaliser, le raisonnement logique et mathématique.
 - 300 après JC : le logicien **Aristote** réalise le premier système formel
 - Année 30 : programme d'**Hilbert** (tentative d'unification de l'ensemble des concepts mathématiques par le biais de la logique), travaux de **Gödel** (Théorème d'incomplétude)
 - \Rightarrow logique propositionnelle, logique du premier ordre, logique temporelle, théorie de la démonstration...
- ② **Suite à l'invention des ordinateurs** : utilisation de ces concepts pour tenir des raisonnements mathématiques \Rightarrow les **systèmes d'inférence** : Prolog, Systèmes experts...
- ③ Puis création de programmes permettant le raisonnement sur les programmes \Rightarrow **Vérification formelle**

Systèmes de preuves - Historique

- ① **Les débuts** : travail des logiciens cherchant à formaliser, le raisonnement logique et mathématique.
 - **300 après JC** : le logicien **Aristote** réalise le premier système formel
 - **Année 30** : programme d'**Hilbert** (tentative d'unification de l'ensemble des concepts mathématiques par le biais de la logique), travaux de **Gödel** (Théorème d'incomplétude)
 - \Rightarrow logique propositionnelle, logique du premier ordre, logique temporelle, théorie de la démonstration...
- ② **Suite à l'invention des ordinateurs** : utilisation de ces concepts pour tenir des raisonnements mathématiques \Rightarrow les **systèmes d'inférence** : Prolog, Systèmes experts...
- ③ Puis création de programmes permettant le raisonnement sur les programmes \Rightarrow **Vérification formelle**

Systèmes de preuves - Historique

- ① **Les débuts** : travail des logiciens cherchant à formaliser, le raisonnement logique et mathématique.
 - **300 après JC** : le logicien **Aristote** réalise le premier système formel
 - **Année 30** : programme d'**Hilbert** (tentative d'unification de l'ensemble des concepts mathématiques par le biais de la logique), travaux de **Gödel** (Théorème d'incomplétude)
 - \Rightarrow logique propositionnelle, logique du premier ordre, logique temporelle, théorie de la démonstration...
- ② **Suite à l'invention des ordinateurs** : utilisation de ces concepts pour tenir des raisonnements mathématiques \Rightarrow les **systèmes d'inférence** : Prolog, Systèmes experts...
- ③ Puis création de programmes permettant le raisonnement sur les programmes \Rightarrow **Vérification formelle**

Systèmes de preuves - Historique

- ① **Les débuts** : travail des logiciens cherchant à formaliser, le raisonnement logique et mathématique.
 - **300 après JC** : le logicien **Aristote** réalise le premier système formel
 - **Année 30** : programme d'**Hilbert** (tentative d'unification de l'ensemble des concepts mathématiques par le biais de la logique), travaux de **Gödel** (Théorème d'incomplétude)
 - \Rightarrow logique propositionnelle, logique du premier ordre, logique temporelle, théorie de la démonstration...
- ② **Suite à l'invention des ordinateurs** : utilisation de ces concepts pour tenir des raisonnements mathématiques \Rightarrow les **systèmes d'inférence** : Prolog, Systèmes experts...
- ③ Puis création de programmes permettant le raisonnement sur les programmes \Rightarrow **Vérification formelle**

Systèmes de preuves en informatique

Nous pouvons distinguer deux techniques de preuves sur modèle :

- ① *Theorem Proving (TP)* ou vérification par preuves :
 - utilisation d'une axiomatique et d'un système de déduction logique
 - Preuve = Théorème du système axiomatique considéré
 - Theorem prover : B, PVS, Coq ...
- ② *Model-Checking (MC)* ou vérification exhaustive :
 - se base une description du système sous forme d'automate
 - vérification d'une propriété par parcours exhaustif des états du système \rightarrow propriétés vrai quelques soit l'exécution du système.
 - Model-Checker : SPIN, VDM ...

Systèmes de preuves en informatique

Nous pouvons distinguer deux techniques de preuves sur modèle :

- ① *Theorem Proving (TP)* ou vérification par preuves :
 - utilisation d'une axiomatique et d'un système de déduction logique
 - Preuve = Théorème du système axiomatique considéré
 - Theorem prover : B, PVS, Coq ...
- ② *Model-Checking (MC)* ou vérification exhaustive :
 - se base une description du système sous forme d'automate
 - vérification d'une propriété par parcours exhaustif des états du système \Rightarrow propriétés vrai quelques soit l'exécution du système.
 - Model-Checker : SPIN, VDM ...

Systèmes de preuves en informatique

Nous pouvons distinguer deux techniques de preuves sur modèle :

- ① *Theorem Proving (TP)* ou vérification par preuves :
 - utilisation d'une axiomatique et d'un système de déduction logique
 - Preuve = Théorème du système axiomatique considéré
 - Theorem prover : B, PVS, Coq ...
- ② *Model-Checking (MC)* ou vérification exhaustive :
 - se base une description du système sous forme d'automate
 - vérification d'une propriété par parcours exhaustif des états du système \Rightarrow propriétés vrai quelques soit l'exécution du système.
 - Model-Checker : SPIN, VDM ...

Systèmes de preuves en informatique

Nous pouvons distinguer deux techniques de preuves sur modèle :

- ① *Theorem Proving (TP)* ou vérification par preuves :
 - utilisation d'une axiomatique et d'un système de déduction logique
 - Preuve = Théorème du système axiomatique considéré
 - Theorem prover : B, PVS, Coq ...
- ② *Model-Checking (MC)* ou vérification exhaustive :
 - se base une description du système sous forme d'automate
 - vérification d'une propriété par parcours exhaustif des états du système \Rightarrow propriétés vrai quelques soit l'exécution du système.
 - Model-Checker : SPIN, VDM ...

Systèmes de preuves en informatique

Nous pouvons distinguer deux techniques de preuves sur modèle :

- ① *Theorem Proving (TP)* ou vérification par preuves :
 - utilisation d'une axiomatique et d'un système de déduction logique
 - Preuve = Théorème du système axiomatique considéré
 - Theorem prover : B, PVS, Coq ...
- ② *Model-Checking (MC)* ou vérification exhaustive :
 - se base une description du système sous forme d'automate
 - vérification d'une propriété par parcours exhaustif des états du système \Rightarrow propriétés vrai quelques soit l'exécution du système.
 - Model-Checker : SPIN, VDM ...

Model-Checker : Fonctionnement

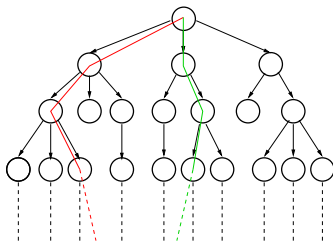
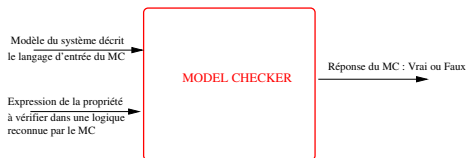


FIG.: Fonctionnement d'un model checker

Theorem Prover : Fonctionnement

Exemple de système formel : **Arithmétique de Peano**

Règles de déduction :

Langage :

$$L = \{0, S, +, \times\}$$

Grammaire des termes :

$$T = V \mid 0 \mid S(T) \mid T + T \mid T \times T$$

Grammaire des formules :

$$F = Atom \mid F \vee F \mid F \wedge F \mid F \Rightarrow F \mid \neg F \mid \exists x F \mid \forall x F$$

Axiomes du système formel :

$$A1 : \forall x \{S(x) \neq 0\}$$

$$A2 : \forall x \{x = 0 \vee \exists y (x = S(y))\}$$

$$A3 : \forall x, y \{S(x) = S(y) \Rightarrow x = y\}$$

$$A4 : \forall x \{x + 0 = x\}$$

$$A5 : \forall x, y \{x + S(y) = S(x + y)\}$$

$$A6 : \forall x \{x \times 0 = 0\}$$

$$A7 : \forall x, y \{x \times S(y) = x \times y + x\}$$

Axiome :

$$\frac{}{\Gamma, A \vdash A} ax$$

Affaiblissement :

$$\frac{\Gamma \vdash A}{\Gamma, B \vdash A} aff$$

Introduction de l'implication :

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow_i$$

Elimination de l'implication :

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \Rightarrow_e$$

Introduction de la conjonction :

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge_i$$

Elimination de la conjonction :

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge_e \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge_e \quad \dots$$

Exemple

Exemple de preuve du théorème : $\vdash A \leftrightarrow (A \rightarrow \perp)$

$$\frac{
 \frac{
 \frac{\Gamma \vdash \neg A^{ax}}{\Gamma \vdash \perp} \neg e
 }{
 \neg A \vdash (A \rightarrow \perp)
 } \rightarrow_i
 }{
 \vdash \neg A \rightarrow (A \rightarrow \perp)
 } \rightarrow_i
 \quad
 \frac{
 \frac{
 \frac{\Gamma_2 \vdash A \rightarrow \perp^{ax}}{\Gamma_2 \vdash \perp} \rightarrow_e
 }{
 A \rightarrow \perp \vdash \neg A
 } \neg i
 }{
 \vdash (A \rightarrow \perp) \rightarrow \neg A
 } \rightarrow_i
 }{
 \vdash \neg A \leftrightarrow (A \rightarrow \perp)
 } \wedge_i$$

Lustre + Lesar = Technique formelle

Lustre est un langage à syntaxe et sémantique formelle. C'est à dire :

- Syntaxe rigoureuse \Rightarrow les constructions du langage ne possèdent qu'une interprétation possible
- Sémantique à base mathématique : sémantique fonctionnelle à base de flots de données

Lustre + Lesar = Technique formelle

Lustre est un langage à syntaxe et sémantique formelle. C'est à dire :

- Syntaxe rigoureuse \Rightarrow les constructions du langage ne possèdent qu'une interprétation possible
- Sémantique à base mathématique : sémantique fonctionnelle à base de flots de données

Lustre + Lesar = Technique formelle

Lustre est un langage à syntaxe et sémantique formelle. C'est à dire :

- Syntaxe rigoureuse \Rightarrow les constructions du langage ne possèdent qu'une interprétation possible
- Sémantique à base mathématique : sémantique fonctionnelle à base de flots de données

Lustre + Lesar = Technique formelle

Lustre est un langage à syntaxe et sémantique formelle. C'est à dire :

- Syntaxe rigoureuse \Rightarrow les constructions du langage ne possèdent qu'une interprétation possible
- Sémantique à base mathématique : sémantique fonctionnelle à base de flots de données

LESAR est l'outil de preuve associé à Lustre : preuve par Model Checking.

Définition

LUSTRE + LESAR = Technique Formelle pour les systèmes réactifs

Lustre + Lesar = Technique formelle

Lustre est un langage à syntaxe et sémantique formelle. C'est à dire :

- Syntaxe rigoureuse \Rightarrow les constructions du langage ne possèdent qu'une interprétation possible
- Sémantique à base mathématique : sémantique fonctionnelle à base de flots de données

LESAR est l'outil de preuve associé à Lustre : preuve par Model Checking.

Définition

LUSTRE + LESAR = Technique Formelle pour les systèmes réactifs

Table des Matières

- 1 Introduction
- 2 Introduction : Techniques Formelles
- 3 **Rappel : Programmation Temps Réel classique**
 - Définition d'un système temps réel classique
 - Architecture d'un système informatique temps réel
 - Architecture : système multitâche
 - Noyau temps réel
- 4 Programmation Réactive
- 5 LUSTRE : les principes de base
- 6 LUSTRE : détail du langage
- 7 Vérification formelle en Lustre
- 8 Flots et Horloges en pratique
- 9 Exemple complet : le chronomètre STOP_WATCH

Définition d'un système informatique temps réel

Définition : Système temps réel

Un système informatique temps réel reçoit des informations sur l'état du procédé extérieur, traite ces données et, en fonction de son état interne et du résultat, évalue une décision qui agit sur cet environnement extérieur dans de sévères contraintes de temps afin d'assurer un **état stable**.

Un système temps réel ne possède pas les propriétés classiques des programmes séquentiels : **indépendance du résultat produit par rapport à la vitesse d'exécution et comportement reproductible**.

Caractéristiques d'un système temps réel :

- respect des contraintes temporelles (temps de réponse court)
- prise en compte des comportements concurrents (parallélisme de l'environnement)
- sûreté de fonctionnement
- prévisibilité
- grande diversité des dispositifs d'entrées/sorties

Architecture d'un système d'informatique temps réel

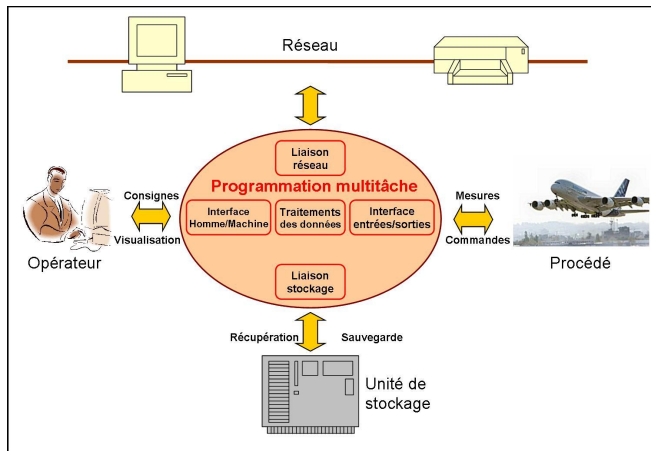


FIG.: Architecture d'un système temps réel

Programmation classique des systèmes temps réel

Pour des raisons de facilité de conception, mise en oeuvre et d'évolutivité, une application temps réel est un système multitâche :

- tâche associée à un ou des évènements
- tâche associée à une ou des réactions
- tâche associée à une entité externe à contrôler
- tâche associée à un traitement particulier
- ...

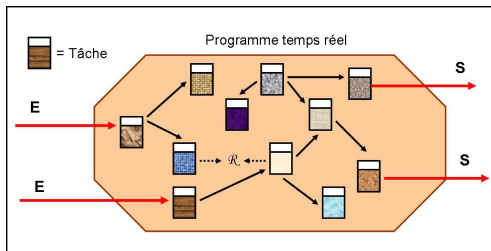


FIG.: Programmation multi-tâches

L'enchâînement des tâches

La partie logicielle d'une application temps réel peut être vue comme un **ensemble de tâches synchronisées**, **communicantes** et partageant des **ressources critiques**.

Le rôle essentiel du système informatique temps réel est donc de **gérer l'enchâînement et la concurrence des tâches** en optimisant l'occupation de l'unité centrale : c'est la **fonction d'ordonancement**.

Exécutif ou noyau temps réel

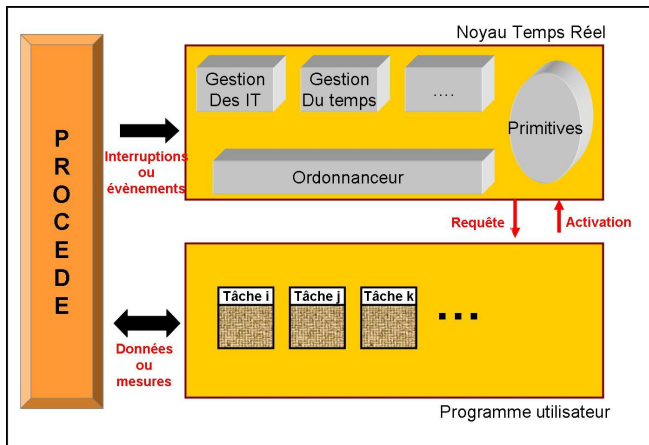


FIG.: Exécutif ou noyau temps réel

Les différents langages de programmation des applications temps réel

- **Langages classiques**

- langage **C**
- assembleurs
- langage **Ada** (no tasking)

- **Langages spécifiques temps réel**

- Langage **Ada** (Dod,1984), **Ada 95**
- Langage **Forth** (C.Moore, 1970) : fonctionnalités de gestion multi-tâche (mise en sommeil de tâches, attente d'évènements, envoi de signaux)
- langage **Modula-2** (N. Wirth, 1978) : gestion complète des tâches (type PROCESS) et fonctionnalités de synchronisation et de communication (SEND, RECEIVE, REPLY)
- langage **OCCAM** (Hoare) a été développé pour répondre aux problèmes de parallélisme et est surtout destiné aux machines à base de transputers

Exemple d'application temps réel industrielles

- **Programme SPOT4** (satellite image)
 - Matra Marconi Space / CNES
 - Spécification et conception : méthode HOOD
 - Langages : Ada, assembleur
- **Programme Ariane 5** (lanceur)
 - Aérospatiale / CNES
 - Spécification et conception : Hood
 - Langages : Ada (Alsys), noyau temps réel (ARTK, Alsys), Assembleur : 68020
- **Programme ISO** (Infrared Space Observatory)
 - Aérospatiale / ESA
 - Spécification et conception : SART et HOOD
 - Langages : Ada (15 000 lignes), Assembleur (11 000 lignes)
- **Programme SENIT8** (Porte-Avions Charles de Gaulle)
 - Dassault Electronique DCN-Ingénierie
 - Spécification et conception : SART et Ada-Buhr
 - Langages : Ada (11 000 lignes), C : 400 000 lignes)
- **Programme Rafale**
 - Dassault Electronique
 - Spécification et conception : SA-RT et OMT
 - Langages : Ada (800 000 lignes) (→ 1 500 000 lignes)

Table des Matières

- 1 Introduction
- 2 Introduction : Techniques Formelles
- 3 Rappel : Programmation Temps Réel classique
- 4 **Programmation Réactive**
 - Approche asynchrone / Approche synchrone
 - Approche synchrone : hypothèse du Synchronisme Fort
 - Classification des langages réactifs
- 5 LUSTRE : les principes de base
- 6 LUSTRE : détail du langage
- 7 Vérification formelle en Lustre
- 8 Flots et Horloges en pratique
- 9 Exemple complet : le chronomètre STOP_WATCH

Méthodes de programmation réactive

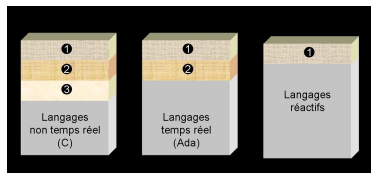


FIG.: Relation entre langages et noyau temps réel

- 1 : gestion des entrées/sorties (gestion des interruptions, gestion des interfaces d'entrées/sorties)
- 2 : ordonnancement des tâches (orchestration du fonctionnement normal, surveillance, changements de mode)
- 3 : relation entre tâches (synchronisation, communication, accès à une ressource en exclusion mutuelle, gestion du temps (compteur, chien de garde))

REMARQUE : Seuls les langages réactifs permettent une **description comportementale** ou opérationnelle du système et **ne nécessitent donc pas la mise en oeuvre d'une politique d'ordonnancement**

Approche asynchrone

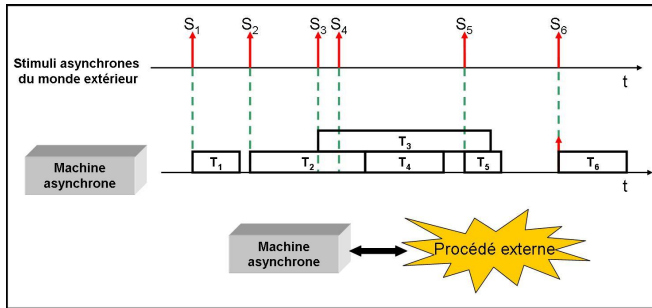


FIG.: Approche asynchrone

Ici, les stimuli externes sont pris en compte immédiatement. Le langage reproduit le comportement asynchrone du monde réel.

Pragmatique industrielle : Approche asynchrone

Chaque calculateur implante :

- des fonctions cycliques et des fonctions apériodiques
- des processus comme interface avec l'environnement (protocoles...)
- un OS ou un séquenceur pilotant l'activation des fonctions cycliques et apériodiques en fonction des signaux reçus, du mode courant...

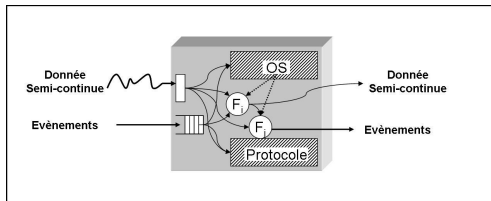


FIG.: Fonctionnement machine asynchrone

Ici, il n'y a **pas d'acquisition asynchrone en cours de traitement !**

Approche synchrone

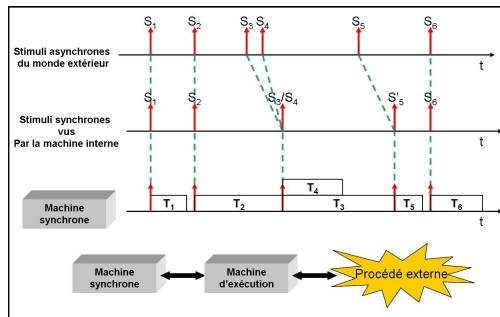
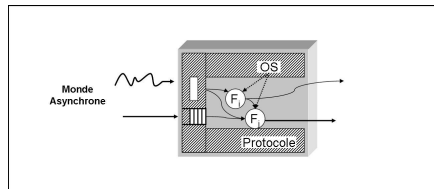


FIG.: Approche synchrone

Le temps de réaction aux stimuli est non nul ; mais les stimuli externes sont synchronisés par rapport aux tâches. Un nouvel évènement d'entrée ne peut arriver avant la fin de la réaction en cours.

Pragmatique industrielle : Approche synchrone



Dans ce cas, l'**acquisition est gelée au début de l'exécution** ! Cela revient à :

- figer l'environnement de F_k pendant son exécution
- bloquer le temps du point de vue de F_k

⇒ **hypothèse de localité synchrone pour la description des F_k**

Techniquement, l'asynchronisme est géré par le séquenceur :

- mémorisation des signaux reçus pendant l'exécution des F_k
- gestion de l'exécution des F_k

Notion : le Synchronisme Fort

Les langages synchrones ont été définis pour décrire des systèmes de processus qui interagissent entre eux = des systèmes réactifs.

Ces langages dits “réactifs” sont fondés sur **hypothèse du synchronisme fort** qui considère qu’un système réagit instantanément à tout stimuli issus de l’environnement. Cela revient à considérer :

- les temps d’exécution et de communication sont nuls
- les actions d’un système de processus sont instantanées
- les sorties sont synchrones avec les entrées et se déroulent aux instants où elles ont été sollicitées
- le système est inactif entre deux sollicitations

Synchronisme fort \Rightarrow Déterminisme

Les propriétés précédentes font que l'indéterminisme est levé et que les actions peuvent s'exécuter simultanément. Ces langages sont linéaires et permettent le **vrai parallélisme**. On parle de **déterminisme** du langage.

Notons que l'hypothèse de synchronisme nécessite que *le temps de réaction du système (à travers l'exécution de ses processus) soit plus petit que l'intervalle de temps qui sépare l'envoi de deux sollicitations, ou stimuli*.

Définition

Lustre est un langage réactif synchrone suivant le paradigme du synchronisme fort.

Les différents langages réactifs

- **Langages dits réactifs**
 - *langages synchrones* :
 - langages impératifs et textuels : **CSML**, **ESTEREL** (cf. Annexe 13)
 - langages déclaratifs à flots de données textuels/graphiques : **LUSTRE**, **SIGNAL**
 - outils de modélisation : **Grafcet**, **StateCharts**
 - *langages asynchrones* :
 - langage impératif et textuel : **ELECTRE**
 - outil de modélisation : **réseaux de petri**

Table des Matières

- 1 Introduction
- 2 Introduction : Techniques Formelles
- 3 Rappel : Programmation Temps Réel classique
- 4 Programmation Réactive
- 5 **LUSTRE : les principes de base**
 - Lustre : Langage à flots de données
 - Approche algébrique
 - Un système d'équations déterministes
- 6 LUSTRE : détail du langage
- 7 Vérification formelle en Lustre
- 8 Flots et Horloges en pratique
- 9 Exemple complet : le chronomètre STOP_WATCH

Approche flot de données

Lustre travaille sur des flots de données. Si X est une variable Lustre il désigne alors le flot $X = (x_1, x_2, \dots)$, x_n étant la valeur de X à l'instant n .

Exemple

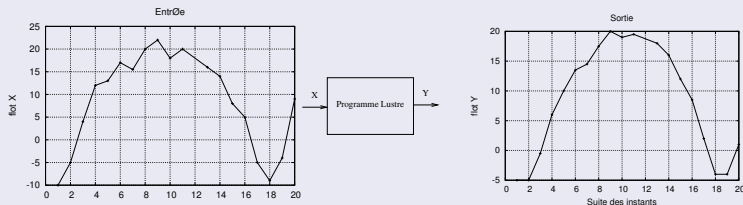


FIG.: Traitement de flots de données, par exemple un filtre

Approche flot de données

L'approche flot de données consiste à **représenter un programme par un graphe** où les nœuds sont des opérateurs qui **transforment des flots de données**.

Exemple

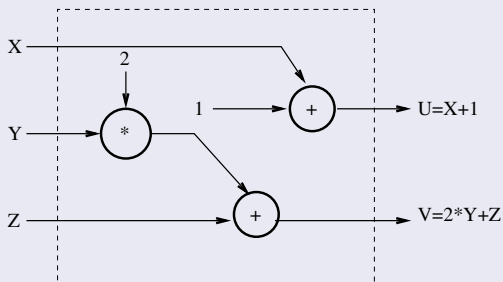


FIG.: Traitement de flots de données, par exemple un filtre

Avantages de l'approche flot de données

- ① Approche **fonctionnelle** qui possède toutes les propriétés mathématiques associés et en particulier :
 - l'absence d'effets de bord \Rightarrow
 - absence d'ordre dans l'écriture des instructions.
 - cela facilite la vérification, la réutilisation et la transformation de programmes.
- ② Approche **à parallélisme vrai** où les contraintes de séquençement et de synchronisation ne proviennent que des flots de données, i.e **ce sont les flots de données qui déterminent les séquences d'exécution et de synchronisation¹**. Cela facilite la synthèse de programmes parallèles.

Avantages de l'approche flot de données

- ① Approche **fonctionnelle** qui possède toutes les propriétés mathématiques associés et en particulier :
 - l'absence d'effets de bord \Rightarrow
 - absence d'ordre dans l'écriture des instructions.
 - cela facilite la vérification, la réutilisation et la transformation de programmes.
- ② Approche **à parallélisme vrai** où les contraintes de séquençement et de synchronisation ne proviennent que des flots de données, i.e **ce sont les flots de données qui déterminent les séquences d'exécution et de synchronisation**¹. Cela facilite la synthèse de programmes parallèles.

¹ Voir la section 6 (utilisation de l'OS) ou la section 7 (exécution)

Approche algébrique, ou déclarative

LUSTRE est un langage formel suivant une **approche algébrique**. On parle aussi d'**approche déclarative**.

Contrairement à un programme C, un programme LUSTRE consiste en un **ensemble d'équations algébriques** définissant les **relations entre flots d'entrées et de sorties**.

Définition : Equation algébrique

Soit $X = (x_1, x_2, \dots)$, $Y = (y_1, y_2, \dots)$ deux flots de réels d'entrées et $Z = (z_1, z_2, \dots)$ un flots de réel de sortie, alors la déclaration : $Z = X + Y$ signifie pour tout instant $n \in \mathbb{N}^*$ $z_n = x_n + y_n$.

Approche algébrique, ou déclarative

LUSTRE est un langage formel suivant une **approche algébrique**. On parle aussi d'**approche déclarative**.

Contrairement à un programme C, un programme LUSTRE consiste en un **ensemble d'équations algébriques** définissant les **relations entre flots d'entrées et de sorties**.

Définition : Equation algébrique

Soit $X = (x_1, x_2, \dots)$, $Y = (y_1, y_2, \dots)$ deux flots de réels d'entrées et $Z = (z_1, z_2, \dots)$ un flots de réel de sortie, alors la déclaration : $Z = X + Y$ signifie pour tout instant $n \in \mathbb{N}^*$ $z_n = x_n + y_n$.

Système d'équations déterministes

La résolution d'un système d'équations en mathématique aboutit à trois types de solutions possibles :

- ① Il existe **plusieurs solutions** (trop d'inconnues...)
- ② Il existe **aucune solution** (informations contradictoires au sein du système)

$$\left\{ \begin{array}{lcl} \dots & & \\ X & = & 3 \\ X & = & 4 \\ \dots & & \end{array} \right.$$

- ③ Il existe **une et une seule solution** pour chaque variable (système linéaire dont le déterminant est non nul)

Système d'équations déterministes

La résolution d'un système d'équations en mathématique aboutit à trois types de solutions possibles :

- ❶ Il existe **plusieurs solutions** (trop d'inconnues...)
- ❷ Il existe **aucune solution** (informations contradictoires au sein du système)

$$\left\{ \begin{array}{lcl} \dots & & \\ X & = & 3 \\ X & = & 4 \\ \dots & & \end{array} \right.$$

- ❸ Il existe **une et une seule solution** pour chaque variable (système linéaire dont le déterminant est non nul)

Système d'équations déterministes

La résolution d'un système d'équations en mathématique aboutit à trois types de solutions possibles :

- ❶ Il existe **plusieurs solutions** (trop d'inconnues...)
- ❷ Il existe **aucune solution** (informations contradictoires au sein du système)

$$\left\{ \begin{array}{lcl} \dots & & \\ X & = & 3 \\ X & = & 4 \\ \dots & & \end{array} \right.$$

- ❸ Il existe **une et une seule solution** pour chaque variable (système linéaire dont le déterminant est non nul)

Système d'équations déterministes

La résolution d'un système d'équations en mathématique aboutit à trois types de solutions possibles :

- ❶ Il existe **plusieurs solutions** (trop d'inconnues...)
- ❷ Il existe **aucune solution** (informations contradictoires au sein du système)

$$\left\{ \begin{array}{lcl} \dots & & \\ X & = & 3 \\ X & = & 4 \\ \dots & & \end{array} \right.$$

- ❸ Il existe **une et une seule solution** pour chaque variable (système linéaire dont le déterminant est non nul)

Système d'équations déterministe

Afin d' écrire uniquement des systèmes déterministes (une seule solution...), Lustre n'accepte que les systèmes d'équations de la forme :

$$\left\{ \begin{array}{l} Y = \dots \\ Z = \dots \end{array} \right. \iff \text{Interdit} \left\{ \begin{array}{l} Y = Z \\ Z = Y \end{array} \right.$$

Circularité non autorisée !

Système d'équations déterministe

Afin d' écrire uniquement des systèmes déterministes (une seule solution...), Lustre n'accepte que les systèmes d'équations de la forme :

$$\begin{cases} Y = \dots \\ Z = \dots \end{cases} \iff \text{Interdit} \begin{cases} Y = Z \\ Z = Y \end{cases}$$

Circularité non autorisée !

Une équation définit une égalité mathématique, pas une affectation :

$$\begin{cases} X = Y + Z \\ Z = U \end{cases} \iff \begin{cases} X = Y + U \\ Z = U \end{cases}$$

Système d'équations déterministe

Afin d' écrire uniquement des systèmes déterministes (une seule solution...), Lustre n'accepte que les systèmes d'équations de la forme :

$$\begin{cases} Y = \dots \\ Z = \dots \end{cases} \iff \text{Interdit } \begin{cases} Y = Z \\ Z = Y \end{cases}$$

Circularité non autorisée !

Une équation définit une égalité mathématique, pas une affectation :

$$\begin{cases} X = Y + Z \\ Z = U \end{cases} \iff \begin{cases} X = Y + U \\ Z = U \end{cases}$$

Les équations n'ont pas d'ordre :

$$\begin{cases} X = Y + Z \\ Z = U \end{cases} \iff \begin{cases} Z = U \\ X = Y + Z \end{cases}$$

Table des Matières

- 1 Introduction
- 2 Introduction : Techniques Formelles
- 3 Rappel : Programmation Temps Réel classique
- 4 Programmation Réactive
- 5 LUSTRE : les principes de base
- 6 **LUSTRE : détail du langage**
 - Structure d'un programme Lustre
 - Les types de données
 - Opérateurs classiques
 - Flots de données et horloges
 - Les opérateurs temporels
- 7 Vérification formelle en Lustre
- 8 Flots et Horloges en pratique
- 9 Exemple complet : le chronomètre STOP_WATCH

Avant de rentrer dans le détail

- Un programme Lustre définit un réseau de composants travaillant sur des flots. En Lustre il est possible de définir ses propres composants, on les appelle *nœuds*.
- Un programme Lustre consiste en un *nœud principal* et éventuellement d'autres *nœuds auxiliaires*².
- On appelle *prototype* du nœud la déclaration des flots d'entrées et de sorties du nœud. Ces flots doivent être typés.³
- Des *flots internes* peuvent être définis
- Le corps d'un nœud contient les *équations* définissant les flots de sorties et internes en fonction des flots d'entrées.

²Assure la modularité et le principe d'encapsulation

³Typage des données ET rythme de son horloge.

Avant de rentrer dans le détail

- Un programme Lustre définit un réseau de composants travaillant sur des flots. En Lustre il est possible de définir ses propres composants, on les appelle *nœuds*.
- Un programme Lustre consiste en un *nœud principal* et éventuellement d'autres *nœuds auxiliaires*².
- On appelle *prototype* du nœud la déclaration des flots d'entrées et de sorties du nœud. Ces flots doivent être typés.³
- Des *flots internes* peuvent être définis
- Le corps d'un nœud contient les *équations* définissant les flots de sorties et internes en fonction des flots d'entrées.

²Assure la modularité et le principe d'encapsulation

³Typage des données ET rythme de son horloge.

Avant de rentrer dans le détail

- Un programme Lustre définit un réseau de composants travaillant sur des flots. En Lustre il est possible de définir ses propres composants, on les appelle *nœuds*.
- Un programme Lustre consiste en un *nœud principal* et éventuellement d'autres *nœuds auxiliaires*².
- On appelle *prototype* du nœud la déclaration des flots d'entrées et de sorties du nœud. Ces flots doivent être typés.³
- Des *flots internes* peuvent être définis
- Le corps d'un nœud contient les *équations* définissant les flots de sorties et internes en fonction des flots d'entrées.

²Assure la modularité et le principe d'encapsulation

³Typage des données ET rythme de son horloge.

Avant de rentrer dans le détail

- Un programme Lustre définit un réseau de composants travaillant sur des flots. En Lustre il est possible de définir ses propres composants, on les appelle *nœuds*.
- Un programme Lustre consiste en un *nœud principal* et éventuellement d'autres *nœuds auxiliaires*².
- On appelle *prototype* du nœud la déclaration des flots d'entrées et de sorties du nœud. Ces flots doivent être typés.³
- Des *flots internes* peuvent être définis
- Le corps d'un nœud contient les *équations* définissant les flots de sorties et internes en fonction des flots d'entrées.

²Assure la modularité et le principe d'encapsulation

³Typage des données ET rythme de son horloge.

Avant de rentrer dans le détail

- Un programme Lustre définit un réseau de composants travaillant sur des flots. En Lustre il est possible de définir ses propres composants, on les appelle *nœuds*.
- Un programme Lustre consiste en un *nœud principal* et éventuellement d'autres *nœuds auxiliaires*².
- On appelle *prototype* du nœud la déclaration des flots d'entrées et de sorties du nœud. Ces flots doivent être typés.³
- Des *flots internes* peuvent être définis
- Le corps d'un nœud contient les *équations* définissant les flots de sorties et internes en fonction des flots d'entrées.

²Assure la modularité et le principe d'encapsulation

³Typage des données ET rythme de son horloge.

Avant de rentrer dans le détail

Exemple

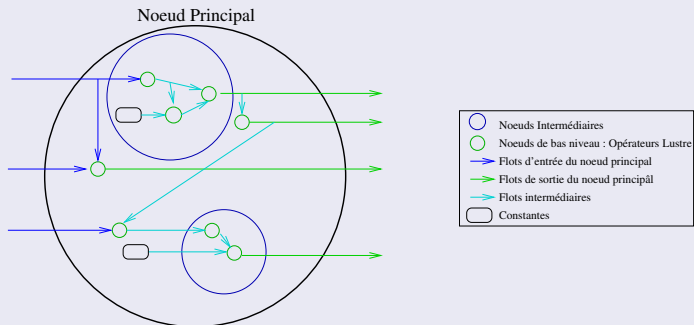


FIG.: Schéma représentant un programme Lustre

Structure d'un programme Lustre

```
[ déclaration des constantes ]  
[ déclaration de types et de fonctions externes ]  
node nom ( déclaration des flots d'entrées )  
    returns( déclaration des flots de sorties ) ;  
[ var  déclaration des flots internes ; ]  
let  
[ assertions ; ]  
    système d'équations définissant une et une seule fois les flots  
    internes et de sorties en fonction d'eux mêmes et des flots d'entrées  
        Eq1;  
        Eq2;  
        ... ;  
        Eqn;  
tel  
[ autres nœuds ]
```

Un exemple de programme Lustre

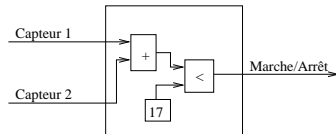


FIG.: Schéma d'un thermostat

A vous de jouer!!!!

Un exemple de programme Lustre

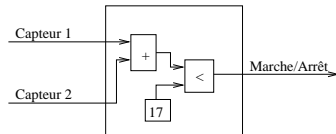


FIG.: Schéma d'un thermostat

A vous de jouer!!!!

Un exemple de programme Lustre

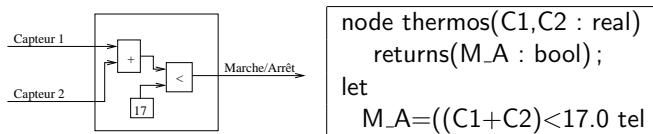


FIG.: Schéma et code Lustre d'un thermostat

Exemple d'une exécution :

| temps | 1 | 2 | 3 | 4 | ... |
|-------|------|------|------|------|-----|
| C1 | 10 | 13 | 15 | 12.6 | ... |
| C2 | 5 | 5.5 | 4 | 3 | ... |
| M_A | vrai | faux | faux | vrai | ... |

Attention : écrire $M_A=((C1+C2)<17)$ n'est pas accepté par le compilateur. Le compilateur détecte une erreur de typage...

Un exemple de programme Lustre

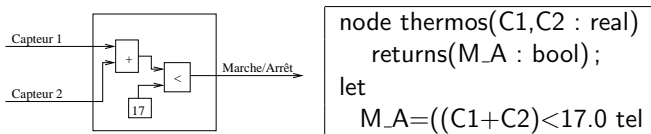


FIG.: Schéma et code Lustre d'un thermostat

Exemple d'une exécution :

| temps | 1 | 2 | 3 | 4 | ... |
|-------|------|------|------|------|-----|
| C1 | 10 | 13 | 15 | 12.6 | ... |
| C2 | 5 | 5.5 | 4 | 3 | ... |
| M_A | vrai | faux | faux | vrai | ... |

Attention : écrire $M_A=((C1+C2)<17)$ n'est pas accepté par le compilateur. Le compilateur détecte une erreur de typage...

Les types de données

Les types de données transportés par les flots sont :

- Les types de bases : `int`, `bool`, `real`
- Les tableaux : `int3`, `real52`, `[int, bool, [int, real]]` , ...
- Les n-uplets : `(bool, bool)`, `(int, bool, real)`, ...
(*type particulier qui n'a pas besoin d'être déclaré, il est manipulé directement dans le système d'équation*)

Les types de données

Les types de données transportés par les flots sont :

- Les types de bases : `int`, `bool`, `real`
- Les tableaux : `int3`, `real52`, `[int, bool, [int, real]]` , ...
- Les n-uplets : `(bool, bool)`, `(int, bool, real)`, ...
(type particulier qui n'a pas besoin d'être déclaré, il est manipulé directement dans le système d'équation)

Les types de données

Les types de données transportés par les flots sont :

- Les types de bases : `int`, `bool`, `real`
- Les tableaux : `int3`, `real52`, `[int, bool, [int, real]]` , ...
- Les n-uplets : `(bool, bool)`, `(int, bool, real)`, ...
(type particulier qui n'a pas besoin d'être déclaré, il est manipulé directement dans le système d'équation)

Les types de données

Les types de données transportés par les flots sont :

- Les types de bases : `int`, `bool`, `real`
- Les tableaux : `int3`, `real52`, `[int, bool, [int, real]]` , ...
- Les n-uplets : `(bool, bool)`, `(int, bool, real)`, ...
(type particulier qui n'a pas besoin d'être déclaré, il est manipulé directement dans le système d'équation)

Les types de données

Les types de données transportés par les flots sont :

- Les types de bases : `int`, `bool`, `real`
- Les tableaux : `int3`, `real52`, `[int, bool, [int, real]]` , ...
- Les n-uplets : `(bool, bool)`, `(int, bool, real)`, ...
(type particulier qui n'a pas besoin d'être déclaré, il est manipulé directement dans le système d'équation)

Et c'est tout !

Les types de données

Les types de données transportés par les flots sont :

- Les types de bases : `int`, `bool`, `real`
- Les tableaux : `int3`, `real52`, `[int, bool, [int, real]]` , ...
- Les n-uplets : `(bool, bool)`, `(int, bool, real)`, ...
(type particulier qui n'a pas besoin d'être déclaré, il est manipulé directement dans le système d'équation)

Et c'est tout !

Remarque : en vue de tenir l'hypothèse de synchronisme fort le langage ne contient aucune structure qui pourrait rendre non borné le temps de calcul (*fonctions récursives, types récursifs, les boucles, etc...*)

Les opérateurs classiques

Les opérateurs classiques

Les opérateurs arithmétiques :

- Binaire : +, -, *, div, mod, /
- Unaire : -

Les opérateurs classiques

Les opérateurs arithmétiques :

- Binaire : +, -, *, div, mod, /
- Unaire : -

Les opérateurs logiques :

- Binaire : or, xor, and, =>
- Unaire : not

Les opérateurs classiques

Les opérateurs arithmétiques :

- Binaire : +, -, *, div, mod, /
- Unaire : -

Les opérateurs logiques :

- Binaire : or, xor, and, =>
- Unaire : not

Les opérateurs de comparaison : =, <>, < , >, <=, >=

Les opérateurs classiques

Les opérateurs arithmétiques :

- Binaire : +, -, *, div, mod, /
- Unaire : -

Les opérateurs logiques :

- Binaire : or, xor, and, =>
- Unaire : not

Les opérateurs de comparaison : =, <>, < , >, <=, >=

L'opérateur de contrôle : `if . then . else` peut se traduire par
"quand . alors . sinon ."

Les opérateurs classiques

Les opérateurs arithmétiques :

- Binaire : +, -, *, div, mod, /
- Unaire : -

Les opérateurs logiques :

- Binaire : or, xor, and, =>
- Unaire : not

Les opérateurs de comparaison : =, <>, < , >, <=, >=

L'opérateur de contrôle : if . then . else peut se traduire par
"quand . alors . sinon ."

Les constantes :

- true = (true, true, ...)
- 1.45 = (1.45, 1.45, ...)

Exercice : Addition et Soustraction de flot d'entiers

Exemple

Soit AS un flot d'entrée booléen,

Soit X1, X2 deux flots d'entrées entiers,

Soit Y un flot de sortie entier,

$$Y = \begin{cases} X1 + X2 & \text{quand AS est vrai} \\ X2 - X1 & \text{autrement} \end{cases}$$

Exercice : Addition et Soustraction de flot d'entiers

Exemple

*Soit AS un flot d'entrée booléen,
Soit X1, X2 deux flots d'entrées entiers,
Soit Y un flot de sortie entier,*

$$Y = \begin{cases} X1 + X2 & \text{quand AS est vrai} \\ X2 - X1 & \text{autrement} \end{cases}$$

A vous de jouer!!!

Réaliser le programme lustre correspondant en utilisant une "notation graphique". Ecrivez ensuite le programme Lustre correspondant.

Correction : Addition et Soustraction de flot d'entiers

Exemple

Soit AS un flot d'entrée booléen,

Soit X1, X2 deux flots d'entrées entiers,

Soit Y un flot de sortie entier,

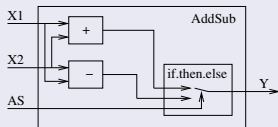
$$Y = \begin{cases} X1 + X2 & \text{quand AS est vrai} \\ X2 - X1 & \text{autrement} \end{cases}$$

Correction : Addition et Soustraction de flot d'entiers

Exemple

Soit AS un flot d'entrée booléen,
 Soit $X1, X2$ deux flots d'entrées entiers,
 Soit Y un flot de sortie entier,

$$Y = \begin{cases} X1 + X2 & \text{quand } AS \text{ est vrai} \\ X2 - X1 & \text{autrement} \end{cases}$$



```

node AddSub(AS :bool;X1,X2 :int)
  returns(Y :int);
let
  Y = if AS then X1+X2
      else X2-X1;
tel
    
```

Exercice

Ecrire un programme Lustre qui prend en entrée un flot réel E , en sortie deux flots : un flots booléen B qui devient vrai quand l'entrée dépasse 100 et un flot de sortie réel S qui renvoie 0 quand B est faux et qui renvoie la différence $E-100$ quand B est vrai.

Le prototype du nœud est le suivant :

```
node diff( $E$  : real)returns( $B$  :bool ; $S$  :real) ;
```

Correction

```
node diff(E : real)returns(B :bool ;S :real) ;
let
  B= E>100.0 ;
  S= if B then 0.0 else E-100.0 ;
tel
```

Les flots de données et les horloges

Définition : Flot de données

Un flot de données X est un couple constitué :

- ① d'une **suite infinie de valeurs** d'un certain type T , noté $(x_1, x_2, \dots, x_n, \dots)$
- ② d'une **horloge** définissant les instants où les valeurs du flot sont définis.

Définition : Horloge

Une **horloge** est soit :

- un flot de données de type booléen.
- ou bien une horloge de base.

Les flots de données et les horloges

Définition : Flot de données

Un flot de données X est un couple constitué :

- ❶ d'une **suite infinie de valeurs** d'un certain type T , noté $(x_1, x_2, \dots, x_n, \dots)$
- ❷ d'une **horloge** définissant les instants où les valeurs du flot sont définis.

Définition : Horloge

Une **horloge** est soit :

- un flot de données de type booléen.
- ou bien une horloge de base.

Les flots de données et les horloges

Définition : Flot de données

Un flot de données X est un couple constitué :

- ❶ d'une **suite infinie de valeurs** d'un certain type T , noté $(x_1, x_2, \dots, x_n, \dots)$
- ❷ d'une **horloge** définissant les instants où les valeurs du flot sont définis.

Définition : Horloge

Une **horloge** est soit :

- un flot de données de type booléen.
- ou bien une horloge de base.

Les flots de données et les horloges

Définition : Flots continus et semi-continus

Les flots ayant pour horloge l'horloge de base sont dit *continus*, les autres sont dit *semi-continus*.

Position dans la suite de valeurs du flot correspond à la notion intuitive d'*instant*.

Sémantique intuitive d'une horloge : un **flot** possédant une horloge T est **défini aux instants où son horloge T est vraie**. Pour l'horloge de base le flot est toujours défini.

Remarque :

- une horloge est un flot \Rightarrow elle possède une horloge et ainsi de suite... jusqu'à l'horloge de base.
- la notion de semi-continuité, est une notion de point de vue...

Les flots de données et les horloges

Définition : Flots continus et semi-continus

Les flots ayant pour horloge l'horloge de base sont dit *continus*, les autres sont dit *semi-continus*.

Position dans la suite de valeurs du flot correspond à la notion intuitive d'*instant*.

Sémantique intuitive d'une horloge : un **flot** possédant une horloge T est **défini aux instants où son horloge T est vraie**. Pour l'horloge de base le flot est toujours défini.

Remarque :

- une horloge est un flot \Rightarrow elle possède une horloge et ainsi de suite... jusqu'à l'horloge de base.
- la notion de semi-continuité, est une notion de point de vue...

Les flots de données et les horloges

Définition : Flots continus et semi-continus

Les flots ayant pour horloge l'horloge de base sont dit *continus*, les autres sont dit *semi-continus*.

Position dans la suite de valeurs du flot correspond à la notion intuitive d'*instant*.

Sémantique intuitive d'une horloge : un **flot** possédant une horloge T est **défini aux instants où son horloge T est vraie**. Pour l'horloge de base le flot est toujours défini.

Remarque :

- une horloge est un flot \Rightarrow elle possède une horloge et ainsi de suite... jusqu'à l'horloge de base.
- la notion de semi-continuité, est une notion de point de vue...

Les flots de données et les horloges

Définition : Flots continus et semi-continus

Les flots ayant pour horloge l'horloge de base sont dit *continus*, les autres sont dit *semi-continus*.

Position dans la suite de valeurs du flot correspond à la notion intuitive d'*instant*.

Sémantique intuitive d'une horloge : un **flot** possédant une horloge T est **défini aux instants où son horloge T est vraie**. Pour l'horloge de base le flot est toujours défini.

Remarque :

- une horloge est un flot \Rightarrow elle possède une horloge et ainsi de suite... jusqu'à l'horloge de base.
- la notion de semi-continuité, est une notion de point de vue...

Opérateurs temporels

- **pre** (*précédent*) : opérateur permettant de travailler sur le passé d'un flot.
- **—>** (*suivi de*) : opérateur permettant l'initialisation d'un flot.
- **when** (*sous-échantillonnage*) : opérateur permettant de transformer un flot continu en un flot semi-continu.
- **current** (*sur-échantillonnage*) : opérateur permettant de transformer un flot semi-continu en un flot continu.

Opérateurs temporels

- **pre** (*précédent*) : opérateur permettant de travailler sur le passé d'un flot.
- **—>** (*suivi de*) : opérateur permettant l'initialisation d'un flot.
- **when** (*sous-échantillonnage*) : opérateur permettant de transformer un flot continu en un flot semi-continu.
- **current** (*sur-échantillonnage*) : opérateur permettant de transformer un flot semi-continu en un flot continu.

Opérateurs temporels

- **pre** (*précédent*) : opérateur permettant de travailler sur le passé d'un flot.
- **—>** (*suivi de*) : opérateur permettant l'initialisation d'un flot.
- **when** (*sous-échantillonnage*) : opérateur permettant de transformer un flot continu en un flot semi-continu.
- **current** (*sur-échantillonnage*) : opérateur permettant de transformer un flot semi-continu en un flot continu.

Opérateurs temporels

- **pre** (*précédent*) : opérateur permettant de travailler sur le passé d'un flot.
- **—>** (*suivi de*) : opérateur permettant l'initialisation d'un flot.
- **when** (*sous-échantillonnage*) : opérateur permettant de transformer un flot continu en un flot semi-continu.
- **current** (*sur-échantillonnage*) : opérateur permettant de transformer un flot semi-continu en un flot continu.

Notation : *nil* l'indéterminé et - l'indéfini

$$X = (nil, x_1, x_2, -, -, x_3, \dots)$$

Notation : *nil* l'indéterminé et - l'indéfini

$$X = (nil, x_1, x_2, -, -, x_3, \dots)$$

nil représente potentiellement **n'importe quelle valeur**, il représente la seule indétermination qui peut apparaître en Lustre, mais celle-ci ne se manifeste (éventuellement) qu'à l'initialisation du système.

Notation : *nil* l'indéterminé et - l'indéfini

$$X = (nil, x_1, x_2, -, -, x_3, \dots)$$

nil représente potentiellement **n'importe quelle valeur**, il représente la seule indétermination qui peut apparaître en Lustre, mais celle-ci ne se manifeste (éventuellement) qu'à l'initialisation du système.

A différencier de : - la valeur indéfinie qui désigne une **valeur inexistante** d'un flot à un instant donné...

Notation : *nil* l'indéterminé et - l'indéfini

$$X = (nil, x_1, x_2, -, -, x_3, \dots)$$

nil représente potentiellement **n'importe quelle valeur**, il représente la seule indétermination qui peut apparaître en Lustre, mais celle-ci ne se manifeste (éventuellement) qu'à l'initialisation du système.

A différencier de : - la valeur indéfinie qui désigne une **valeur inexistante** d'un flot à un instant donné...

Attention : *nil* n'est pas un symbole du langage Lustre, c'est seulement une notation de ce cours...

Exemples

- Soit N un flot d'entiers sur l'horloge de base :
 $N = (4, 2, 7, 0, 8 \dots)$
- Soit M un flot d'entiers sur l'horloge de base :
 $N = (1, 2, 3, 10, 4 \dots)$
- Soit B un flot de booléens sur l'horloge de base :
 $B = (\text{vrai}, \text{faux}, \text{vrai}, \text{faux}, \dots)$

Alors :

$$A = \text{pre}(N \text{ when } B) = (nil, 4, -, 7, -, \dots)$$

L'opérateur **pre** : retour vers le passé...

Il permet de **mémoriser la valeur précédente d'un flot** ou un ensemble de flots.

Soit X le flot (X_1, \dots, X_n, \dots) alors :

- $pre(X)$ est le flot $(nil, X_1, \dots, X_n, \dots)$
- $pre(pre(X))$ est le flot $(nil, nil, X_1, \dots, X_n, \dots)$
- ...

Par extension, l'équation $(Y, Y') = pre(X, X')$ signifie :

- $Y_0 = nil, Y'_0 = nil$
- pour tout $n > 0, Y_n = X_{n-1}$ et $Y'_n = X'_{n-1}$

L'opérateur \rightarrow : suivi de

Il permet d'**initialiser un flot** ou un ensemble de flots.

Soit X le flot (X_1, \dots, X_n, \dots) et Y le flot $(Y_1, Y_2, \dots, Y_n, \dots)$
alors :

$X \rightarrow Y$ est le flot $(X_1, Y_2, \dots, Y_n, \dots)$

Par extension, l'équation $(Z, Z') = (X, X') \rightarrow (Y, Y')$ signifie :

- $Z_1 = X_1, Z'_1 = X'_1$
- pour tout $n > 0, Z_n = Y_n$ et $Z'_n = Y'_n$

L'opérateur **when** : opérateur de sous-échantillonnage

Soit B un flot booléen, soit X un flot de type T .

L'équation $Y = X \text{ when } B ;$ définit un flot semi-continu Y de type $T \text{ when } B$ égal à X quand B est vrai, indéfinie sinon.

Exemple

| | | | | | | | | | |
|--------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-----|
| <i>temps</i> | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
| X | 2 | 4 | 1 | 8 | 3 | 9 | 3 | 4 | ... |
| B | <i>vrai</i> | <i>faux</i> | <i>faux</i> | <i>vrai</i> | <i>faux</i> | <i>vrai</i> | <i>faux</i> | <i>vrai</i> | ... |
| Y | 2 | - | - | 8 | - | 9 | - | 4 | ... |

L'opérateur **when** : opérateur de sous-échantillonnage

Soit B un flot booléen, soit X un flot de type T .

L'équation $Y = X \text{ when } B ;$ définit un flot semi-continu Y de type $T \text{ when } B$ égal à X quand B est vrai, indéfinie sinon.

Exemple

| | | | | | | | | | |
|--------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-----|
| <i>temps</i> | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
| X | 2 | 4 | 1 | 8 | 3 | 9 | 3 | 4 | ... |
| B | <i>vrai</i> | <i>faux</i> | <i>faux</i> | <i>vrai</i> | <i>faux</i> | <i>vrai</i> | <i>faux</i> | <i>vrai</i> | ... |
| Y | 2 | - | - | 8 | - | 9 | - | 4 | ... |

L'opérateur **current** : prolongement, sur-échantillonnage

Soit X un flot semi-continu de type T when B ,

Soit Y un flot de type T

L'équation $Y = \text{current } X;$ définit un flot Y de type T égale à X quand X est définie égale à la dernière valeur définie de X quand X est indéfinie, ou **nil** s'il n'y a pas de dernière valeur.

Exemple

| <i>temps</i> | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|--------------|------------|---|---|---|---|---|---|---|-----|
| X | - | 2 | - | 8 | - | - | 4 | - | ... |
| Y | <i>nil</i> | 2 | 2 | 8 | 8 | 8 | 4 | 4 | ... |

L'opérateur **current** : prolongement, sur-échantillonnage

Soit X un flot semi-continu de type T when B ,

Soit Y un flot de type T

L'équation $Y = \text{current } X ;$ définit un flot Y de type T égale à X quand X est définie égale à la dernière valeur définie de X quand X est indéfinie, ou **nil** s'il n'y a pas de dernière valeur.

Exemple

| <i>temps</i> | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|--------------|------------|---|---|---|---|---|---|---|-----|
| X | - | 2 | - | 8 | - | - | 4 | - | ... |
| Y | <i>nil</i> | 2 | 2 | 8 | 8 | 8 | 4 | 4 | ... |

Exercice

Le but est de réaliser un nœud qui cumule les données d'entrées...

Soit X un flot d'entrées entier

Soit Y un flot de sorties entier

$$(Y_1, Y_2, Y_3, \dots) = (X_1, X_1 + X_2, X_1 + X_2 + X_3, \dots)$$

Prototype du nœud :

```
node CumSum( $X$  :int) returns ( $Y$  :int) ;
```

Correction

Le but est de réaliser un nœud qui cumule les données d'entrées...

Soit X un flot d'entrées entier

Soit Y un flot de sorties entier

$$(Y_1, Y_2, Y_3, \dots) = (X_1, X_1 + X_2, X_1 + X_2 + X_3, \dots)$$

```
node CumSum(X :int) returns(Y :int) ;  
let  
    Y = X -> (X + pre(Y)) ;  
tel
```

Exercice : Un compteur... Revival.

Soit RAS (remise à zero) un flot d'entrées booléen

Soit N un flot de sorties entier

A chaque instant le flot de sortie N s'incrémente de 1, quand RAS est vrai N revient à zero.

A vous de jouer !

Correction : Un compteur... Revival.

Soit RAS (remise à zero) un flot d'entrées booléen

Soit N un flot de sorties entier

A chaque instant le flot de sortie N s'incrmente de 1, quand RAS est vrai N revient à zero.

```
node compteur(RAZ :bool) returns(N :int);  
let  
  N = 0 -> if RAZ then 0 else (pre(N)+1);  
tel
```

Exemple : Utilisation du noeud Compteur

```
node dec(RAZ :bool) returns(N :int);
let
    N = -compteur(RAZ);
tel
```

```
node IncDec(RAZ,ID :bool) returns(N :int);
let
    N = if ID then compteur(RAZ) else dec(RAZ);
tel
```

Exemple

| <i>temps</i> | <i>1</i> | <i>2</i> | <i>3</i> | <i>4</i> | <i>5</i> | <i>6</i> | <i>7</i> | <i>8</i> |
|--------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| <i>RAZ</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> |
| <i>ID</i> | <i>vrai</i> | <i>vrai</i> | <i>vrai</i> | <i>vrai</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> | <i>vrai</i> |
| <i>N</i> | <i>0</i> | <i>1</i> | <i>2</i> | <i>3</i> | <i>-4</i> | <i>-5</i> | <i>-6</i> | <i>7</i> |

Exemple : Utilisation du noeud Compteur

```

node dec(RAZ :bool) returns(N :int);
let
    N = -compteur(RAZ);
tel
    
```

```

node IncDec(RAZ,ID :bool) returns(N :int);
let
    N = if ID then compteur(RAZ) else dec(RAZ);
tel
    
```

Exemple

| <i>temps</i> | <i>1</i> | <i>2</i> | <i>3</i> | <i>4</i> | <i>5</i> | <i>6</i> | <i>7</i> | <i>8</i> |
|--------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| <i>RAZ</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> |
| <i>ID</i> | <i>vrai</i> | <i>vrai</i> | <i>vrai</i> | <i>vrai</i> | <i>faux</i> | <i>faux</i> | <i>faux</i> | <i>vrai</i> |
| <i>N</i> | <i>0</i> | <i>1</i> | <i>2</i> | <i>3</i> | <i>-4</i> | <i>-5</i> | <i>-6</i> | <i>7</i> |

Exemple d'un compteur intermitant : **when** et **current**

Soit RAZ et B deux flots d'entrées booléens, N un flot de sorties entier. Quand B est vrai le compteur incrémente la sortie N .

Exemple d'un compteur intermitant : **when** et **current**

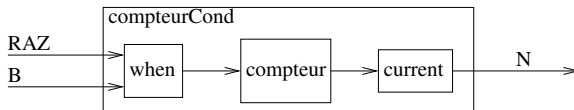
Soit RAZ et B deux flots d'entrées booléens, N un flot de sorties entier. Quand B est vrai le compteur incrémente la sortie N .

```
node compteurCond(RAZ,B :bool) returns(N :int);  
let  
  N=current(compteur(RAZ when B));  
tel
```

Exemple d'un compteur intermitant : **when** et **current**

Soit RAZ et B deux flots d'entrées booléens, N un flot de sorties entier. Quand B est vrai le compteur incrémente la sortie N .

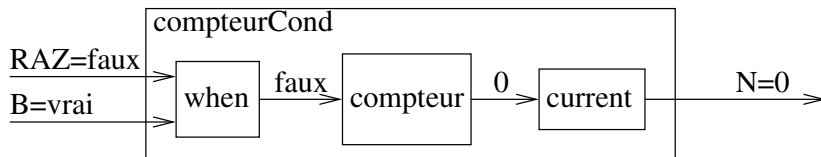
```
node compteurCond(RAZ,B :bool) returns(N :int);
let
    N=current(compteur(RAZ when B));
tel
```



Execution de compteurCond :

/

| |
|----------|
| temps=1 |
| RAZ=faux |
| B=vrai |
| N=0 |

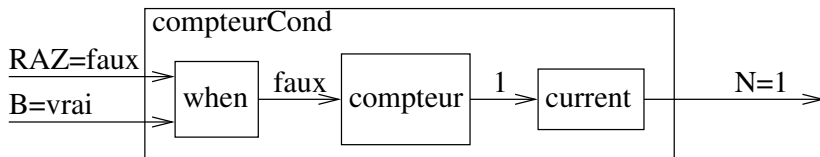


$$N = \text{current}(\text{compteur}(\text{RAZ when } B));$$

Execution de compteurCond :

/-

| |
|----------|
| temps=2 |
| RAZ=faux |
| B=vrai |
| N=1 |

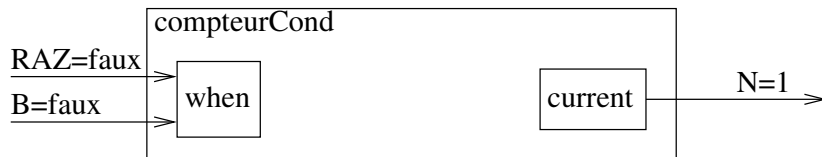


`N=current(compteur(RAZ when B));`

Execution de compteurCond :

Séquence d'exécution : $/-\backslash$

| |
|----------|
| temps=3 |
| RAZ=faux |
| B=faux |
| N=1 |

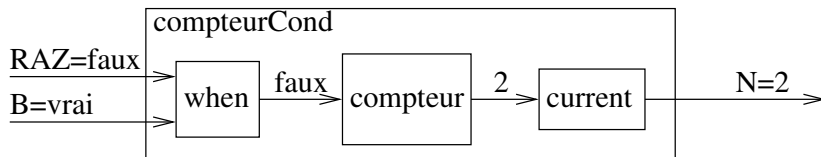


$N = \text{current}(\text{compteur}(\text{RAZ when B})) ;$

Execution de compteurCond :

/-_

| |
|----------|
| temps=4 |
| RAZ=faux |
| B=vrai |
| N=2 |

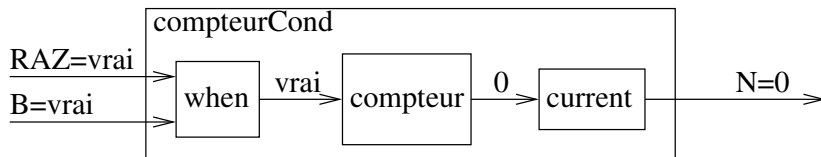


`N=current(compteur(RAZ when B));`

Execution de compteurCond :

/-_ - /

| |
|----------|
| temps=5 |
| RAZ=vrai |
| B=vrai |
| N=0 |

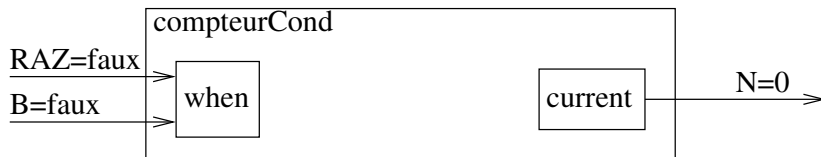


`N=current(compteur(RAZ when B));`

Execution de compteurCond :

/- \- /-

| |
|----------|
| temps=6 |
| RAZ=faux |
| B=faux |
| N=0 |

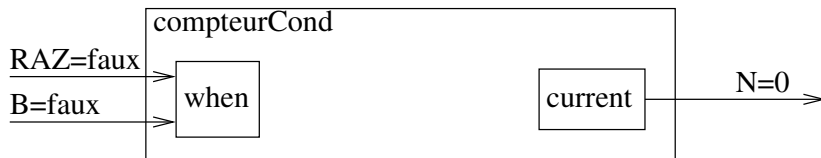


$N = \text{current}(\text{compteur}(\text{RAZ when B})) ;$

Execution de compteurCond :

\neg

| |
|----------|
| temps=7 |
| RAZ=faux |
| B=faux |
| N=0 |

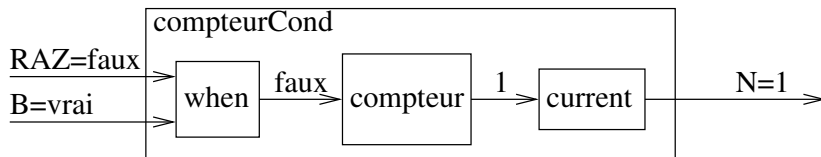


$N = \text{current}(\text{compteur}(\text{RAZ when } B));$

Execution de compteurCond :

$\neg B$

| |
|----------|
| temps=8 |
| RAZ=faux |
| B=vrai |
| N=1 |

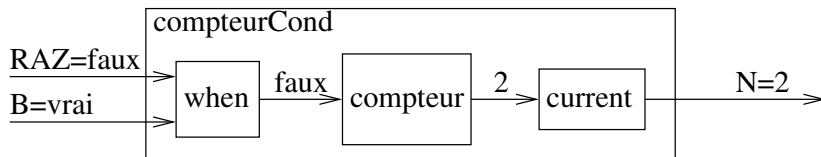


$N = \text{current}(\text{compteur}(\text{RAZ when } B))$;

Execution de compteurCond :

$/-\backslash_ - /-\backslash_ /$

| |
|----------|
| temps=9 |
| RAZ=faux |
| B=vrai |
| N=2 |

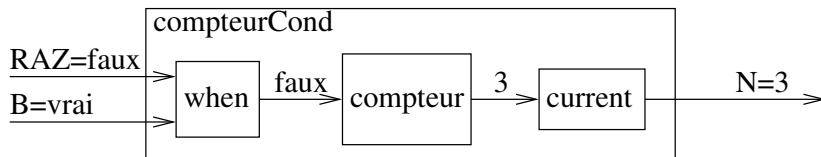


$N = \text{current}(\text{compteur}(\text{RAZ when } B));$

Execution de compteurCond :

$/-\backslash - /-\backslash - /-$

| |
|----------|
| temps=10 |
| RAZ=faux |
| B=vrai |
| N=3 |

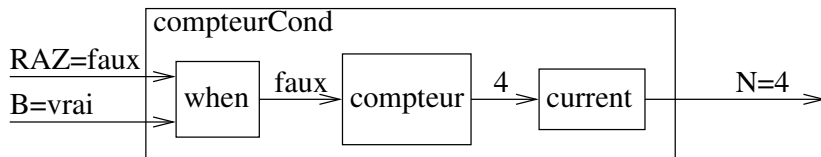


`N=current(compteur(RAZ when B));`

Execution de compteurCond :

\neg / \neg - / \neg - / \neg

| |
|----------|
| temps=11 |
| RAZ=faux |
| B=vrai |
| N=4 |



`N=current(compteur(RAZ when B));`

Remarque

Remarque suite à l'exemple précédent :

- Un nœud existe seulement lorsque ses flots d'entrées sont définis.
 - \Rightarrow Le prototype du noeud doit obligatoirement définir au moins un flot d'entrée ! Et ceci même si le flot d'entrée n'est pas utilisé..

Remarque

Remarque suite à l'exemple précédent :

- Un nœud existe seulement lorsque ses flots d'entrées sont définis.
 - \Rightarrow Le prototype du noeud doit obligatoirement définir au moins un flot d'entrée ! Et ceci même si le flot d'entrée n'est pas utilisé..

Mémo Récapitulatif

| B | false | true | false | true | false | false | true | true |
|--|--------------|-------------|--------------|-------------|--------------|--------------|-------------|-------------|
| X | X_1 | X_2 | X_3 | X_4 | X_5 | X_6 | X_7 | X_8 |
| Y | Y_1 | Y_2 | Y_3 | Y_4 | Y_5 | Y_6 | Y_7 | Y_8 |
| pre(X) | <i>nil</i> | X_1 | X_2 | X_3 | X_4 | X_5 | X_6 | X_7 |
| Y \rightarrow pre(X) | Y_1 | X_1 | X_2 | X_3 | X_4 | X_5 | X_6 | X_7 |
| Z=X when B | — | X_2 | — | X_4 | — | — | X_7 | X_8 |
| T= current Z | <i>nil</i> | X_2 | X_2 | X_4 | X_4 | X_4 | X_7 | X_8 |
| pre(Z) | — | <i>nil</i> | — | X_2 | — | — | X_4 | X_7 |
| 0 \rightarrow pre(Z) | — | 0 | — | X_2 | — | — | X_4 | X_7 |

Table des Matières

- 1 Introduction
- 2 Introduction : Techniques Formelles
- 3 Rappel : Programmation Temps Réel classique
- 4 Programmation Réactive
- 5 LUSTRE : les principes de base
- 6 LUSTRE : détail du langage
- 7 **Vérification formelle en Lustre**
 - De quoi a-t-on besoin ?
 - La notion d'assertion
 - Programme Lustre = automate
 - Safety Logic
 - Pratique : Schéma général de vérification en Lustre
 - Exprimer des propriétés temporelles : quelques opérateurs plus complexes
- 8 Flots et Horloges en pratique
- 9 Exemple complet : le chronomètre STOP_WATCH

Vérification formelle en Lustre

Pour faire de la vérification formelle, il faut :

- ❶ Un **modèle du programme** compréhensible par le prouveur utilisé
 - Ici modèle Lustre = modèle exprimable sous forme d'**automate**
- ❷ Un **prouveur** pour vérifier les propriétés logiques
 - on utilise ici le Model Checker **Lesar**
- ❸ Une **logique**⁴ pour décrire les propriétés à prouver
 - on utilise ici la logique temporelle **Safety Logic**
- ❹ Des **assertions** pour décrire les hypothèses sous lesquels on raisonne

⁴Remarque : La logique utilisée est exprimable en Lustre !

Vérification formelle en Lustre

Pour faire de la vérification formelle, il faut :

- ❶ Un **modèle du programme** compréhensible par le prouveur utilisé
 - Ici modèle Lustre = modèle exprimable sous forme d'**automate**
- ❷ Un **prouveur** pour vérifier les propriétés logiques
 - on utilise ici le Model Checker **Lesar**
- ❸ Une **logique**⁴ pour décrire les propriétés à prouver
 - on utilise ici la logique temporelle **Safety Logic**
- ❹ Des **assertions** pour décrire les hypothèses sous lesquels on raisonne

⁴Remarque : La logique utilisée est exprimable en Lustre !

Vérification formelle en Lustre

Pour faire de la vérification formelle, il faut :

- ❶ Un **modèle du programme** compréhensible par le prouveur utilisé
 - Ici modèle Lustre = modèle exprimable sous forme d'**automate**
- ❷ Un **prouveur** pour vérifier les propriétés logiques
 - on utilise ici le Model Checker **Lesar**
- ❸ Une **logique**⁴ pour décrire les propriétés à prouver
 - on utilise ici la logique temporelle **Safety Logic**
- ❹ Des **assertions** pour décrire les hypothèses sous lesquels on raisonne

⁴Remarque : La logique utilisée est exprimable en Lustre !

Vérification formelle en Lustre

Pour faire de la vérification formelle, il faut :

- ❶ Un **modèle du programme** compréhensible par le prouveur utilisé
 - Ici modèle Lustre = modèle exprimable sous forme d'**automate**
- ❷ Un **prouveur** pour vérifier les propriétés logiques
 - on utilise ici le Model Checker **Lesar**
- ❸ Une **logique**⁴ pour décrire les propriétés à prouver
 - on utilise ici la logique temporelle **Safety Logic**
- ❹ Des **assertions** pour décrire les hypothèses sous lesquels on raisonne

⁴**Remarque** : La logique utilisée est exprimable en Lustre !

Vérification formelle en Lustre

Pour faire de la vérification formelle, il faut :

- ❶ Un **modèle du programme** compréhensible par le prouveur utilisé
 - Ici modèle Lustre = modèle exprimable sous forme d'**automate**
- ❷ Un **prouveur** pour vérifier les propriétés logiques
 - on utilise ici le Model Checker **Lesar**
- ❸ Une **logique**⁴ pour décrire les propriétés à prouver
 - on utilise ici la logique temporelle **Safety Logic**
- ❹ Des **assertions** pour décrire les hypothèses sous lesquels on raisonne

⁴**Remarque** : La logique utilisée est exprimable en Lustre !

La notion d'assertion

Les assertions permettent au concepteur d'écrire des hypothèses sur l'environnement extérieur et/ou sur le programme lui-même.

Ceci permet :

- d'optimiser la compilation
- de vérifier des propriétés sous conditions
- de simplifier la conception des programmes

La notion d'assertion

Les assertions permettent au concepteur d'écrire des hypothèses sur l'environnement extérieur et/ou sur le programme lui-même.

Ceci permet :

- d'optimiser la compilation
- de vérifier des propriétés sous conditions
- de simplifier la conception des programmes

La notion d'assertion

Les assertions permettent au concepteur d'écrire des hypothèses sur l'environnement extérieur et/ou sur le programme lui-même.

Ceci permet :

- d'optimiser la compilation
- de vérifier des propriétés sous conditions
- de simplifier la conception des programmes

La notion d'assertion

Les assertions permettent au concepteur d'écrire des hypothèses sur l'environnement extérieur et/ou sur le programme lui-même.

Ceci permet :

- d'optimiser la compilation
- de vérifier des propriétés sous conditions
- de simplifier la conception des programmes

La notion d'assertion

Les assertions permettent au concepteur d'écrire des hypothèses sur l'environnement extérieur et/ou sur le programme lui-même.

Ceci permet :

- d'optimiser la compilation
- de vérifier des propriétés sous conditions
- de simplifier la conception des programmes

`assert(Exp_bool)` : vrai à chaque instant, quel que soit le chemin d'exécution du système

Exemple

Exemple

```
assert(not(X and Y));
```

Affirme que les flots X et Y ne doivent jamais être vrai simultanément

```
assert(true -> not(X and pre(X)));
```

Affirme que les flot booléen X ne transporte jamais deux valeurs vraies consécutives

Notion d'automate : Rappel

- Un **automate** est un **graphe orienté** permettant de décrire l'ensemble des états possibles⁵ d'un système ainsi que son comportement dynamique.
- Un **programme Lustre** définit un **automate fini** car son espace mémoire est borné (Pas d'allocations dynamiques).

⁵Ici= ensemble des combinaisons de valeurs possibles des variables du programme

Notion d'automate : Rappel

- Un **automate** est un **graphe orienté** permettant de décrire l'ensemble des états possibles⁵ d'un système ainsi que son comportement dynamique.
- Un **programme Lustre** définit un **automate fini** car son espace mémoire est borné (Pas d'allocations dynamiques).

⁵Ici= ensemble des combinaisons de valeurs possibles des variables du programme

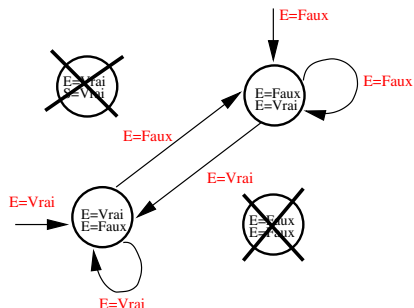
Exemple : espace d'états



```
node notE(E :bool)
  returns(S :bool);
let
  S = not E;
tel
```

Passage code Lustre/ Automate

Exemple : espace d'états

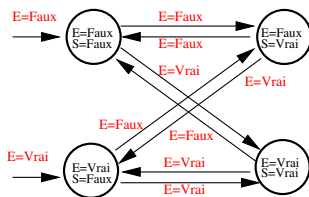


```

node notE(E :bool)
  returns(S :bool);
let
  S = not E;
tel
  
```

Passage code Lustre/ Automate

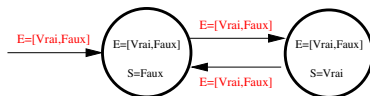
Exemple : Un oscillateur et son automate à actions



```
node osc(E :bool)
  returns(S :bool);
let
  S = false -> not pre(S);
tel
```

Un oscillateur et son automate complet

Exemple : un oscillateur

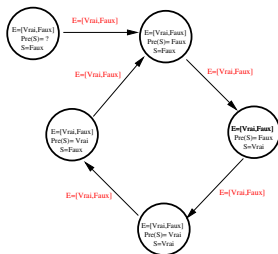


```

node osc(E :bool)
  returns(S :bool);
let
  S = false -> not pre(S);
tel
    
```

Code Lustre et son *automate à actions*

- But : réaliser un oscillateur deux fois plus lent... Et jouer dans les **pre** pour inclure l'histoire du programme courant.



```

node osc2(E :bool)
  returns(S :bool);
let
  S =not(false -> pre(false -> pre(S)));
tel
    
```

*Code Lustre et son **automate à actions***

Safety Logic

En vérification formelle Lustre, la logique permettant de décrire les formules à vérifier est **exprimable en langage Lustre**.

Il s'agit de la logique temporelle **SL** pour **Safety Logic**.

Définition : Safety Logic

C'est une logique temporelle à temps qualitatif (notion d'ordre entre les événements), capable d'exprimer des propriétés dites de **sûreté**

Définition : Propriété de sûreté

Une *propriété de sûreté* énonce que, sous certaines conditions, quelque chose *ne se produira jamais*.

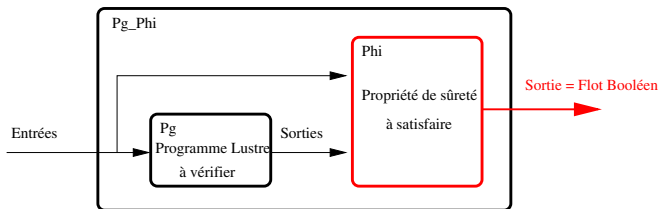
Remarque : Il existe des logiques temporelles plus expressives (CTL, LTL ...) permettant d'exprimer des propriétés plus complexe : *propriétés de vivacité, d'équité, absence de blocage, atteignabilité...*

Expression des propriétés à prouver

Une **propriété Lustre** sera donc exprimée par un nœud définissant une relation logique entre :

- des flots d'entrées de types *bool*
- et un unique flot de sortie de type *bool*

Schéma général de la vérification en Lustre



Fonctionnement d'un Model Checker

- **Pg** : Programme à vérifier
- **Phi** : Observateur de propriétés. Sa sortie doit toujours valoir **true**
- **Pg_Phi** : Programme à mettre dans l'outil de vérification (Model Checker Lesar).

Exemple

```
node Pg(E :int) returns (S :bool);
let
  S = if (E=0) then not(pre(S)) else (E>0);
tel
```

```
node Observateur_Phi(E :int, S :bool) returns (B :bool);
let
  assert(E=0);
  B = true  $\rightarrow$  (S xor pre(S));
tel
```

```
node Pg_Phi(E :int) returns (B :bool);
let
  B = Observateur_Phi(E, Pg(E));
tel
```

Opérateur temporel **never**

A chaque instant, le noeud **never** renvoie VRAI si et seulement si le flot d'entrée n'a jamais été vrai au cours de l'exécution.

```
node never(X :bool) returns (P :bool);
let
    P = if X then false else (true -> pre(P));
tel
```

| temps | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
|-------|------|------|------|------|------|------|------|-----|
| X | faux | faux | faux | vrai | faux | faux | vrai | ... |
| P | vrai | vrai | vrai | faux | faux | faux | faux | ... |

Opérateur temporel **never**

A chaque instant, le noeud **never** renvoie VRAI si et seulement si le flot d'entrée n'a jamais été vrai au cours de l'exécution.

```
node never(X :bool) returns (P :bool);
let
    P = if X then false else (true -> pre(P));
tel
```

| temps | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
|-------|------|------|------|------|------|------|------|-----|
| X | faux | faux | faux | vrai | faux | faux | vrai | ... |
| P | vrai | vrai | vrai | faux | faux | faux | faux | ... |

Opérateur temporel **never**

A chaque instant, le noeud **never** renvoie VRAI si et seulement si le flot d'entrée n'a jamais été vrai au cours de l'exécution.

```
node never(X :bool) returns (P :bool);
let
    P = if X then false else (true -> pre(P));
tel
```

| temps | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
|-------|------|------|------|------|------|------|------|-----|
| X | faux | faux | faux | vrai | faux | faux | vrai | ... |
| P | vrai | vrai | vrai | faux | faux | faux | faux | ... |

Opérateur temporel **since**

L'opérateur **since** renvoie VRAI à chaque instant si et seulement si aucune occurrence de X n'est apparu, ou bien si une occurrence de X est apparu alors elle a été suivi dans le futur d'une occurrence de Y.

```
node since(X,Y : bool) returns (P :bool);  
let  
  P= if X then Y  
    else if Y then true  
    else (true -> pre(P));  
tel
```

Opérateur temporel **since**

L'opérateur **since** renvoie VRAI à chaque instant si et seulement si aucune occurrence de X n'est apparu, ou bien si une occurrence de X est apparu alors elle a été suivi dans le futur d'une occurrence de Y.

```
node since(X,Y : bool) returns (P :bool);  
let  
  P= if X then Y  
    else if Y then true  
      else (true -> pre(P));  
tel
```

Opérateur temporel **since** : Exemple

```

node since(X,Y : bool) returns (P :bool);
let
  P= if X then Y
      else if Y then true
          else (true -> pre(P));
tel
    
```

| temps | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|-------|------|------|------|------|------|------|------|------|-----|
| X | faux | faux | vrai | faux | faux | faux | vrai | vrai | ... |
| P | vrai | vrai | faux | faux | vrai | vrai | faux | vrai | ... |

Opérateur temporel **once_B_from_A_to_C** : utilisation des opérateurs *since* et *never*

- Soit la propriété suivante :
“Toute occurrence de A doit être suivie par une occurrence de B avant la prochaine occurrence de C”
- La même propriété exprimée au passé :
“A chaque occurrence de C, soit A ne s’est jamais produit, soit A s’est produit et B s’est produit depuis la dernière occurrence de A”

Opérateur temporel **once_B_from_A_to_C** : utilisation des opérateurs *since* et *never*

- Soit la propriété suivante :
“Toute occurrence de A doit être suivie par une occurrence de B avant la prochaine occurrence de C”
- La même propriété exprimée au passé :
“A chaque occurrence de C, soit A ne s’est jamais produit, soit A s’est produit et B s’est produit depuis la dernière occurrence de A”

Opérateur temporel **once_B_from_A_to_C** : utilisation des opérateurs *since* et *never*

- Soit la propriété suivante :
“Toute occurrence de A doit être suivie par une occurrence de B avant la prochaine occurrence de C”
- La même propriété exprimée au passé :
“A chaque occurrence de C, soit A ne s’est jamais produit, soit A s’est produit et B s’est produit depuis la dernière occurrence de A”

```
node once_B_from_A_to_C(A, B, C : bool) returns (P :bool);
let
    P=if C then (never(A) or since(A,B)) else true;
tel
```


Table des Matières

- 1 Introduction
- 2 Introduction : Techniques Formelles
- 3 Rappel : Programmation Temps Réel classique
- 4 Programmation Réactive
- 5 LUSTRE : les principes de base
- 6 LUSTRE : détail du langage
- 7 Vérification formelle en Lustre
- 8 Flots et Horloges en pratique**
 - Rappel sur les Horloges
 - Programmation avec horloge
- 9 Exemple complet : le chronomètre STOP_WATCH

Rappel : Pourquoi utiliser des horloges ?

● Motivation :

- Obtenir une notion de « contrôle » des flot de données ;
- Pouvoir exprimer que certaines parties du programme seront exécutées moins souvent ;
- S'affranchir des primitives de synchronisation offertes par un noyau temps réel.

● Solution :

- On associe aux flots de données des horloges, rythmant la séquence de valeurs (le flot) et par voie de cause à effet les opérateurs...

● Mise œuvre :

- Utilisation de l'opérateur **when** pour associer une horloge à un flot et de l'opérateur **current** pour rendre continu un flot semi-continu.
- Rappel : par défaut, un flot possède une horloge : l'horloge de base...

Sur- et Sous- échantillonnage

| | | | | | | | |
|------------------------|------|-------|-------|------|------|-------|------|
| X | 4 | 1 | -3 | 0 | 2 | 7 | 8 |
| H | True | False | False | True | True | False | True |
| Y = X when H | 4 | | | 0 | 2 | | 8 |
| Z = current (Y) | 4 | 4 | 4 | 0 | 2 | 2 | 8 |

Cadence d'un noeud

L'horloge de l'instance d'un nœud
 =
 horloge de ses entrées effectives

● CONSEQUENCE :

- Si les entrées d'un nœud sont « sous-échantillonnées » alors cela forcera l'ensemble du nœud à tourner plus lentement.

● REMARQUE :

- « sous-échantillonner » les entrées ou « sous-échantillonner » les sorties produit des résultats différents.

| C | True | True | False | False | True | False | True |
|-------------------------------|------|------|-------|-------|------|-------|------|
| Count((r,true) when C) | 1 | 2 | - | - | 3 | - | 4 |
| Count((r,true)) when C | 1 | 2 | - | - | 5 | - | 7 |

Clock Checking

Un flot étant toujours associé à une horloge, et récursivement, une horloge étant un flot, un compilateur Lustre réalise à la fois :

- des opérations de **vérification de types sur les flots** (*Type Checking*);
- des opérations de **vérification des horloges de flots** (*Clock Checking*);

Dans la pratique : deux horloges sont égales si et seulement si il s'agit du même flot.

Voici quelques unes des règles appliquées :

$$\text{Clk}(\text{exp when } C) = C \quad \equiv \quad \text{Clk}(\text{exp}) = \text{Clk}(C)$$

$$\text{Clk}(\text{current}(\text{exp})) = \text{Clk}(\text{Clk}(\text{exp}))$$

Pour tout opérateur OP :

$$\text{Clk}(\text{OP}(E_1, E_2, E_3)) = C \quad \equiv \quad \text{Clk}(E_1) = \text{Clk}(E_2) = C$$

Séquencement

- **Hypothèse du synchronisme fort :**
 - Revient à considérer que toutes les exécutions sont instantannées ;
 - Ide : Cette hypothèse permet d'assurer qu'une exécution se termine durant un cycle d'horloge ;
- **Conséquence : Déterminisme**
 - aucun nouvel évènement ne peut perturber le bon déroulement de l'exécution d'un processus (Déterminisme)

Que fait-on si les opérateurs d'un n a une durée d'exécution supérieur à un cycle d'horloge ???

Séquencement

Solution :

- Décomposer l'opération d'un nœud ;
- ET exécuter les résultats de la décomposition en séquence.

Définition : Séquencement

Opération qui permet d'exécuter les opérations d'un nœud en une séquence d'opérations issues de la décomposition de l'opération principale. Ceci permet :

- permet de s'assurer que l'hypothèse du synchronisme est valide !
- Si l'hypothèse du synchronisme n'est pas valide : Le comportement du modèle Lustre ne correspondra pas au « monde réel ». Par conséquent toutes les propriétés vérifiées avec Lesar sont potentiellement fausses.

Synchronisation par condition

Synchronisation par condition

Il s'agit d'exécuter un nœud si une condition particulière B est satisfaite.

Ceci permet de rythmer l'exécution d'un nœud afin qu'il se synchronise avec d'autres nœuds.

Synchronisation par condition : conduct

Dans les versions récentes de Lustre, l'opérateur **conduct** facilite ce type de synchronisation par condition.

$$Y = \text{conduct "OP"} (B, X, \text{INIT})$$

Avec :

- **OP** est un opérateur (ide. un nœud) de signature $\text{OP} : T \Rightarrow T'$
- **B** est un flot Booléen ;
- **INIT** un flot de type T' ;
- **X** sont les entrées du nœud **OP** ;

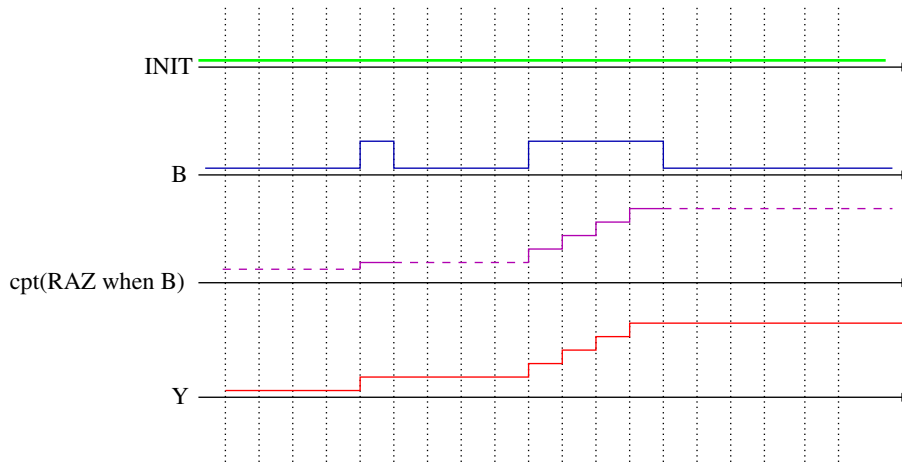
Formellement, cela donne :

INIT_n si quelque soit $i < n$, $B_i = \text{false}$

$Y = \text{Op}(X_n)$ si $B_n = \text{true}$

$Y_n - 1$ si $B_n = \text{false}$ et si il existe $i < n$ tel que $B_i = \text{true}$

Synchronisation par condition : conduct



$Y = \text{conduct} \ll \text{cpt} \gg (B, \text{RAZ}, 0)$

Synchronisation par condition : conduct

```
node Conduct_Op (B : bool,  
                 X : type(X),  
                 INIT : type Op(X))  
  returns (S : Type(Op(X)))  
let  
  
  S = if B  
    then current (Op(X when B))  
    else Init — pre(S);  
  
tel
```

Table des Matières

- 1 Introduction
- 2 Introduction : Techniques Formelles
- 3 Rappel : Programmation Temps Réel classique
- 4 Programmation Réactive
- 5 LUSTRE : les principes de base
- 6 LUSTRE : détail du langage
- 7 Vérification formelle en Lustre
- 8 Flots et Horloges en pratique
- 9 Exemple complet : le chronomètre STOP_WATCH

STOP_WATCH : Cahier des charges

Le but est de spécifier le comportement d'un chronomètre simplifié. Le chronomètre dispose de 3 entrées de trois boutons :

- **on_off** : permet d'activer et de désactiver le chronomètre ;
- **reset** : remet le chronomètre à zéro quand le chronomètre est désactivé ;
- **freeze** : gèle ou degèle l'affichage quand le chronomètre est actif.

Prototype du nœud :

```
node STOP_WATCH(on_off, reset, freeze : bool)
  returns (time : int )
```

Travail préliminaire : l'opérateur COUNT

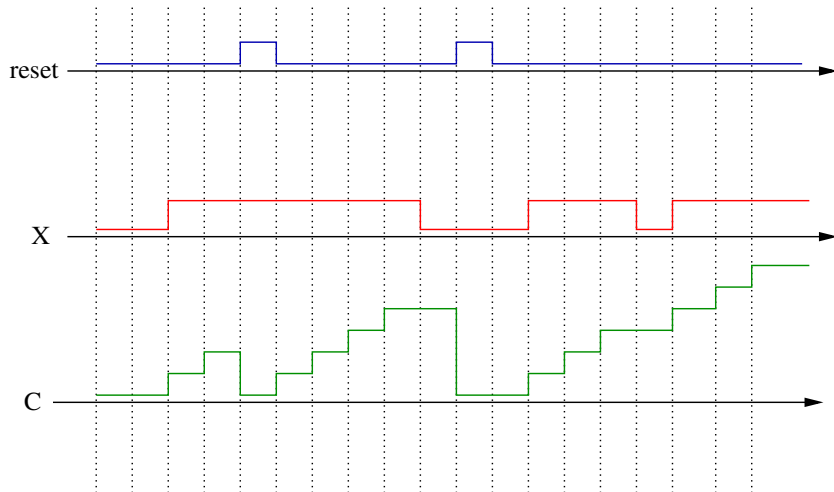
Prototype :

```
node COUNT(reset, X : bool) returns (C : int) ;
```

Spécification :

- Remise à 0 si reset, sinon il est incrémenter si X ;
- Tout se passe comme si C valait 0 à l'origine.

Travail préliminaire : l'opérateur COUNT



Travail préliminaire : l'opérateur COUNT

```
node COUNT(reset, X : bool) returns (C : int);
let
  C = if reset
      then 0
      else if X
            then (0 - pre(C)) + 1
            else (0 - pre(C));
tel
```


Travail préliminaire : l'opérateur SWITCH

Prototype du nœud SWITCH :

```
node SWITCH (on, off : bool) returns (s : bool);
```

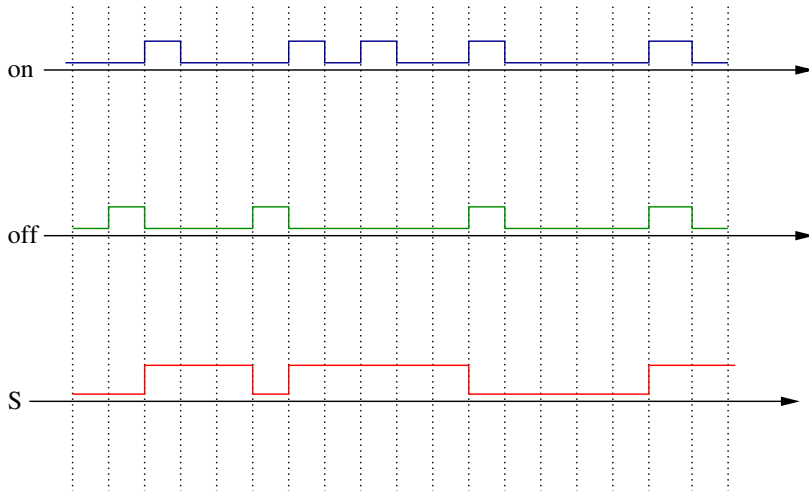
Spécifications :

- Passe de False à True si on ;
- Passe de True à False si off ;
- Tout se comporte comme si S était False à l'origine ;
- Doit fonctionner correctement même si off et on sont les mêmes.

Hypothèses sur l'environnement extérieur :

- On et Off ne sont jamais à True deux fois de suite... Cette hypothèse simule un appui court sur un interrupteur, par exemple...

Travail préliminaire : l'opérateur SWITCH



Travail préliminaire : l'opérateur SWITCH

```
node SWITCH ( on, off : bool ) returns ( S : bool );

assert( not(on and pre(on)) )
assert( not(off and pre(off)) )

let

    S = if (false -> pre(S))
        then not off
        else on;

tel
```

QUESTION :

Supposons que les hypothèses sur l'environnement extérieur ne soit pas vérifié (appui long sur les boutons de l'interrupteur)... Comment s'y prendre pour toute fois obtenir le même comportement ?

STOP_WATCH : Cahier des charges

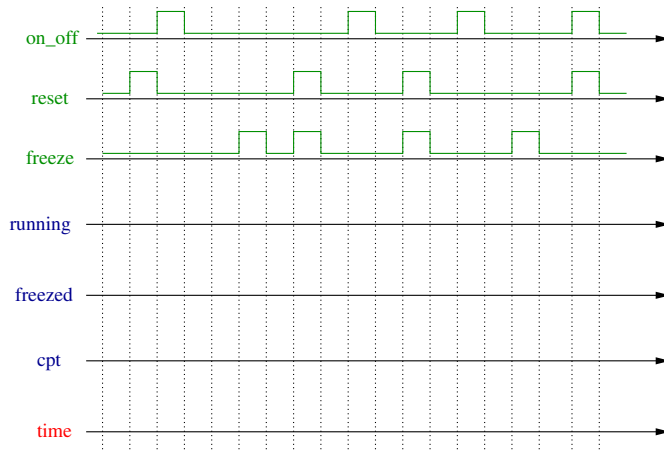
Le but est de spécifier le comportement d'un chronomètre simplifié. Le chronomètre dispose de 3 entrées de trois boutons :

- **on_off** : permet d'activer et de désactiver le chronomètre ;
- **reset** : remet le chronomètre à zéro quand le chronomètre est désactivé ;
- **freeze** : gèle ou degèle l'affichage quand le chronomètre est actif.

Prototype du nœud :

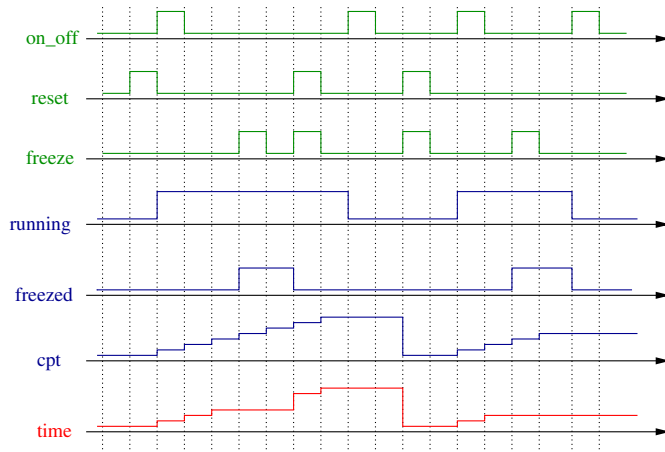
```
node STOP_WATCH(on_off, reset, freeze : bool)
  returns (time : int )
```

STOP_WATCH : Chronogrammes



Exécution du noeud STOP WATCH

STOP_WATCH : Chronogrammes



Exécution du noeud STOP WATCH

STOP_WATCH

```
node STOP_WATCH (on_off, reset, freeze : bool)
    returns (time : int);

    assert(true -> not(on_off and pre(on_off)))
    assert(true -> not(reset and pre(reset)))
    assert(true -> not(freeze and pre(freeze)))

    var running, freezed : bool , cpt : int;

    let

        running = SWITCH (on_off, on_off);

        freezed = SWITCH( freeze and running,
                          freeze or on_off);

        cpt = count( reset and not running, running);

        time = if freezed
                then ( 0 -> pre(time))
                else cpt;

    tel
```

STOP_WATCH : Vérification

Nous cherchons à vérifier la propriété de sûreté suivante : “ Il n’est pas possible de remettre à zéro le compteur en cours d’exécution ”

Pour cela, nous ajouterons une hypothèse supplémentaire au nœud principal :

```
assert(on_off = true -> false)
```

ide : Le chronomètre est activé à l’instant initial et reste en exécution ad vitam eternam...

Nous construisons le noeud permettant d’observer cette propriété :

```
node observateur_prop(on_off, reset, freeze : bool, S : int)
  returns (P : bool)
let
  P= true -> if (S=0) then false else pre(P);
tel
```


STOP_WATCH : Vérification /2

Nous construisons ensuite le noeud de vérification :

```
node VERIF(on_off,reset,freeze : bool)
    returns (B : bool);

let
    B = observateur_prop(on_off, reset, freeze,
        STOP_WATCH(on_off, reset, freeze));
tel
```

Il n'y a plus qu'à lancer le nœud VERIF dans le prouveur **LESAR**.

FIN

Annexes

- 10 LUSTRE : Génération de code embarqué
- 11 Langage réactif synchrone : ESTEREL
- 12 Commandes unix pour Lustre et Lesar

Schéma descriptif de la génération de code embarqué

Programme Lustre

```
node foo(...)
tel
...
let
```

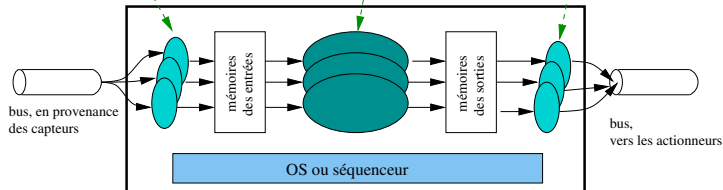
Compilation
qualifiée

Fonctions C ou ADA

```
void foo();
+ fonctions de lecture des entrées en mémoire
+ fonction d'écriture des sorties mémoires
```

Fonctions C d'acquisition des entrées
sur les bus, les capteurs
→ Produites manuellement

Fonctions C de production des sorties
sur les bus, vers les actionneurs
→ Produites manuellement



ESTEREL : présentation

ESTEREL a été développé en 1991 au laboratoire de recherche INRIA-Sophia.

Comme Lustre, il s'agit d'un **langage réactif**, particulièrement adapté aux systèmes de contrôles/commandes.

ESTEREL est un **langage textuel et impératif** de haut niveau permettant la spécification de système (création de modèles).

Contrairement à Lustre :

- ESTEREL adopte une **vision événementielle** ;
- ESTEREL adopte une **approche synchrone** ;

ESTEREL : présentation

ESTEREL a été développé en 1991 au laboratoire de recherche INRIA-Sophia.

Comme Lustre, il s'agit d'un **langage réactif**, particulièrement adapté aux systèmes de contrôles/commandes.

ESTEREL est un **langage textuel et impératif** de haut niveau permettant la spécification de système (création de modèles).

Contrairement à Lustre :

- ESTEREL adopte une **vision événementielle** ;
- ESTEREL adopte une **approche synchrone** ;

ESTEREL : présentation

ESTEREL a été développé en 1991 au laboratoire de recherche INRIA-Sophia.

Comme Lustre, il s'agit d'un **langage réactif**, particulièrement adapté aux systèmes de contrôles/commandes.

ESTEREL est un **langage textuel et impératif** de haut niveau permettant la spécification de système (création de modèles).

Contrairement à Lustre :

- ESTEREL adopte une **vision événementielle** ;
- ESTEREL adopte une **approche synchrone** ;

ESTEREL : présentation

ESTEREL a été développé en 1991 au laboratoire de recherche INRIA-Sophia.

Comme Lustre, il s'agit d'un **langage réactif**, particulièrement adapté aux systèmes de contrôles/commandes.

ESTEREL est un **langage textuel et impératif** de haut niveau permettant la spécification de système (création de modèles).

Contrairement à Lustre :

- ESTEREL adopte une **vision événementielle** ;
- ESTEREL adopte une approche **synchrone** ;

ESTEREL : présentation

ESTEREL a été développé en 1991 au laboratoire de recherche INRIA-Sophia.

Comme Lustre, il s'agit d'un **langage réactif**, particulièrement adapté aux systèmes de contrôles/commandes.

ESTEREL est un **langage textuel et impératif** de haut niveau permettant la spécification de système (création de modèles).

Contrairement à Lustre :

- ESTEREL adopte une **vision évènementielle** ;
- ESTEREL adopte une approche **synchrone** ;

ESTEREL : présentation

Les qualificatifs associés au langage ESTEREL sont les suivants :

- **Réactivité** : ESTEREL propose des instructions réactives qui prennent en compte les occurrences sporadiques (non cycliques) d'évènements ;
- **Atomicité des réactions** : l'hypothèse d'instantanéité des réactions implique leur atomicité. Dans le cas d'une implémentation réelle, il est nécessaire de prévoir des mécanismes pour garantir cette atomicité ;
- **Diffusion instantanée des signaux** : l'entité de base des programmes ESTEREL est le signal, outil de synchronisation et de communication ;
- **Déterminisme** : cette propriété est assurée par la compilation des programmes ESTEREL en des automates à états finis.

Le langage ESTEREL, est un langage au style impératif. Il possède des instructions ou primitives que l'on peut classer dans trois catégories :

- instructions classiques de programmation de haut niveau ;
- instructions réactives ;
- instructions temporelles.

ESTEREL : présentation

Les qualificatifs associés au langage ESTEREL sont les suivants :

- **Réactivité** : ESTEREL propose des instructions réactives qui prennent en compte les occurrences sporadiques (non cycliques) d'évènements ;
- **Atomicité des réactions** : l'hypothèse d'instantanéité des réactions implique leur atomicité. Dans le cas d'une implémentation réelle, il est nécessaire de prévoir des mécanismes pour garantir cette atomicité ;
- **Diffusion instantanée des signaux** : l'entité de base des programmes ESTEREL est le signal, outil de synchronisation et de communication ;
- **Déterminisme** : cette propriété est assurée par la compilation des programmes ESTEREL en des automates à états finis.

Le langage ESTEREL, est un langage au style impératif. Il possède des instructions ou primitives que l'on peut classer dans trois catégories :

- instructions classiques de programmation de haut niveau ;
- instructions réactives ;
- instructions temporelles.

ESTEREL : présentation

Les qualificatifs associés au langage ESTEREL sont les suivants :

- **Réactivité** : ESTEREL propose des instructions réactives qui prennent en compte les occurrences sporadiques (non cycliques) d'évènements ;
- **Atomicité des réactions** : l'hypothèse d'instantanéité des réactions implique leur atomicité. Dans le cas d'une implémentation réelle, il est nécessaire de prévoir des mécanismes pour garantir cette atomicité ;
- **Diffusion instantanée des signaux** : l'entité de base des programmes ESTEREL est le signal, outil de synchronisation et de communication ;
- **Déterminisme** : cette propriété est assurée par la compilation des programmes ESTEREL en des automates à états finis.

Le langage ESTEREL, est un langage au style impératif. Il possède des instructions ou primitives que l'on peut classer dans trois catégories :

- instructions classiques de programmation de haut niveau ;
- instructions réactives ;
- instructions temporelles.

ESTEREL : présentation

Les qualificatifs associés au langage ESTEREL sont les suivants :

- **Réactivité** : ESTEREL propose des instructions réactives qui prennent en compte les occurrences sporadiques (non cycliques) d'évènements ;
- **Atomicité des réactions** : l'hypothèse d'instantanéité des réactions implique leur atomicité. Dans le cas d'une implémentation réelle, il est nécessaire de prévoir des mécanismes pour garantir cette atomicité ;
- **Diffusion instantanée des signaux** : l'entité de base des programmes ESTEREL est le signal, outil de synchronisation et de communication ;
- **Déterminisme** : cette propriété est assurée par la compilation des programmes ESTEREL en des automates à états finis.

Le langage ESTEREL, est un langage au style impératif. Il possède des instructions ou primitives que l'on peut classer dans trois catégories :

- instructions classiques de programmation de haut niveau ;
- instructions réactives ;
- instructions temporelles.

ESTEREL : présentation

Les qualificatifs associés au langage ESTEREL sont les suivants :

- **Réactivité** : ESTEREL propose des instructions réactives qui prennent en compte les occurrences sporadiques (non cycliques) d'évènements ;
- **Atomicité des réactions** : l'hypothèse d'instantanéité des réactions implique leur atomicité. Dans le cas d'une implémentation réelle, il est nécessaire de prévoir des mécanismes pour garantir cette atomicité ;
- **Diffusion instantanée des signaux** : l'entité de base des programmes ESTEREL est le signal, outil de synchronisation et de communication ;
- **Déterminisme** : cette propriété est assurée par la compilation des programmes ESTEREL en des automates à états finis.

Le langage ESTEREL, est un langage au style impératif. Il possède des instructions ou primitives que l'on peut classer dans trois catégories :

- instructions classiques de programmation de haut niveau ;
- instructions réactives ;
- instructions temporelles.

ESTEREL : présentation

Les qualificatifs associés au langage ESTEREL sont les suivants :

- **Réactivité** : ESTEREL propose des instructions réactives qui prennent en compte les occurrences sporadiques (non cycliques) d'évènements ;
- **Atomicité des réactions** : l'hypothèse d'instantanéité des réactions implique leur atomicité. Dans le cas d'une implémentation réelle, il est nécessaire de prévoir des mécanismes pour garantir cette atomicité ;
- **Diffusion instantanée des signaux** : l'entité de base des programmes ESTEREL est le signal, outil de synchronisation et de communication ;
- **Déterminisme** : cette propriété est assurée par la compilation des programmes ESTEREL en des automates à états finis.

Le langage ESTEREL, est un langage au style impératif. Il possède des instructions ou primitives que l'on peut classer dans trois catégories :

- instructions classiques de programmation de haut niveau ;
- instructions réactives ;
- instructions temporelles.

ESTEREL : présentation

Les qualificatifs associés au langage ESTEREL sont les suivants :

- **Réactivité** : ESTEREL propose des instructions réactives qui prennent en compte les occurrences sporadiques (non cycliques) d'évènements ;
- **Atomicité des réactions** : l'hypothèse d'instantanéité des réactions implique leur atomicité. Dans le cas d'une implémentation réelle, il est nécessaire de prévoir des mécanismes pour garantir cette atomicité ;
- **Diffusion instantanée des signaux** : l'entité de base des programmes ESTEREL est le signal, outil de synchronisation et de communication ;
- **Déterminisme** : cette propriété est assurée par la compilation des programmes ESTEREL en des automates à états finis.

Le langage ESTEREL, est un langage au style impératif. Il possède des instructions ou primitives que l'on peut classer dans trois catégories :

- instructions classiques de programmation de haut niveau ;
- instructions réactives ;
- instructions temporelles.

ESTEREL : présentation

Les qualificatifs associés au langage ESTEREL sont les suivants :

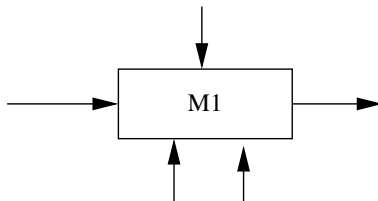
- **Réactivité** : ESTEREL propose des instructions réactives qui prennent en compte les occurrences sporadiques (non cycliques) d'évènements ;
- **Atomicité des réactions** : l'hypothèse d'instantanéité des réactions implique leur atomicité. Dans le cas d'une implémentation réelle, il est nécessaire de prévoir des mécanismes pour garantir cette atomicité ;
- **Diffusion instantanée des signaux** : l'entité de base des programmes ESTEREL est le signal, outil de synchronisation et de communication ;
- **Déterminisme** : cette propriété est assurée par la compilation des programmes ESTEREL en des automates à états finis.

Le langage ESTEREL, est un langage au style impératif. Il possède des instructions ou primitives que l'on peut classer dans trois catégories :

- instructions classiques de programmation de haut niveau ;
- instructions réactives ;
- instructions temporelles.

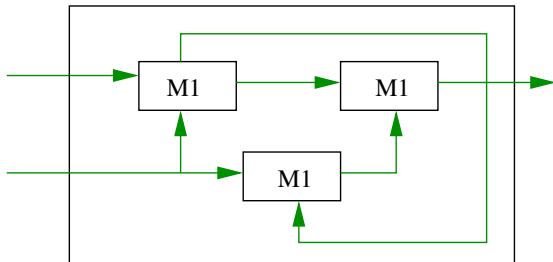
ESTEREL : L'unité de base du langage

Module :



Programme :

ensemble de modules
communicants de
façon synchrone



ESTEREL : Instructions classiques

- déclaration de signaux d'entrée : `input ident`
- déclaration de signaux de sortie : `output ident`
- déclaration de signaux internes :

`signal ident in
[...]
end`
- déclaration de variables internes :

`var ident in
[...]
end`
- acquisition capteurs : `sensor ident`
- affectation : `:=`
- constructeur de séquence : `;`
- opérateur de composition parallèle : `||`
- instruction qui ne fait rien : `nothing`

ESTEREL : Instructions classiques /2

- constructeur d'itération :

```
loop
[... ]
end loop
```

- constructeur d'alternative :

```
if cond then
[... ]
else
[... ]
end
```

- structure de traitement d'exception :

```
trap ident in
[... ]
handle ident do
[... ]
end
```

- unité de programmation :

```
module
[... ]
.
```

ESTEREL : Instructions réactive /légende

[: début d'une instruction
] : fin d'une instruction
↑ E : occurrence de l'évènement E
<instr> : instruction
<occ> : occurrence d'un évènement

ESTEREL : Instructions réactive /1

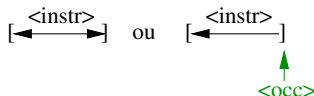
halt : ne fait rien mais ne se termine pas



emit S : émission d'un évènement
emis dans tout l'environnement...

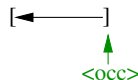


do $\langle \text{instr} \rangle$
watching $\langle \text{occ} \rangle$

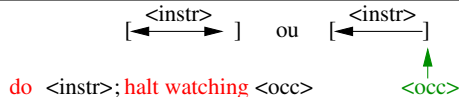


await $\langle \text{occ} \rangle$: attente de l'occurrence strictement future

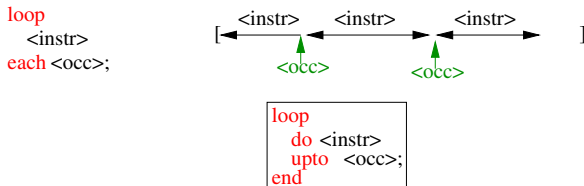
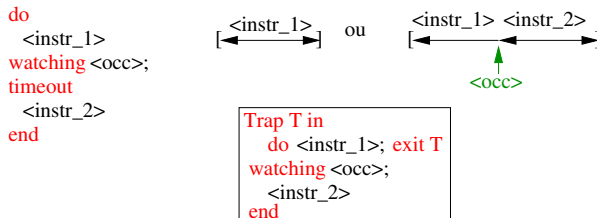
do halt watching



do $\langle \text{instr} \rangle$
upto $\langle \text{occ} \rangle$

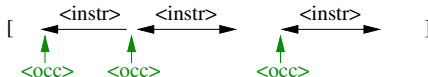


ESTEREL : Instructions réactive /2



ESTEREL : Instructions réactive /3

every <occ> do
 <instr>
 end every



await <occ>;
 loop
 <instr>;
 each <occ>

emit S



sustain S



loop
 emit S;
 each tick

ESTEREL : Exemple de module générique

```

module <mod_id> :

    - partie déclarative
    input <sig_dcl_list>;
    output <sig_dcl_list>;
    sensor <sig_dcl_list>;
    type <type_dcl>
    procedure <proc_dcl>;

    signal <sig_dcl_list> in
        var <var_dcl_list> in
            - partie impérative

                <corps du module>

        end var
    end signal
end module
    
```

ESTEREL : Un exemple simple

Spécification : Enfoncer un bouton et déclencher une action dans un délai d'une seconde, sinon déclencher l'alarme.

```
module EXEMPLE :  
  input BOUTON, SECONDE ;  
  output ACTION, ALARME ;  
  trap FINI in  
    do  
      do  
        halt  
        watching BOUTON ;  
        emit ACTION ;  
        exit FINI  
      watching SECONDE ;  
      emit ALARME ;  
    end
```

ESTEREL : Exemple de diffusion instantannée

```

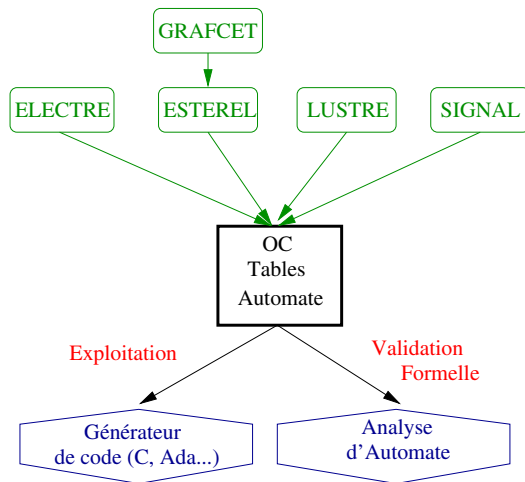
module EXEMPLE :
  emit S ;

  present T then
    emit U
  end

  present S then
    emit T
  end
    
```

REMARQUE : les signaux S, T, U sont synchrones.

ESTEREL : Environnement de programmation



Mémo : Utilisation de Lustre

Un programme Lustre possède l'extension **.lus**. Nous noterons *prog* pour désigner le nom d'un programme et *NŒUD* pour désigner le nom d'un noeud défini dans ce même programme.

Simulation d'un noeud :

Lustre dispose d'une interface de simulation (Luciole), basé sur un interpréteur de programmes, permettant de visualiser l'évolution des flots de données d'un noeud.

luciole prog.lus NŒUD

Compilation d'un noeud :

La commande **lustre** permet d'obtenir un fichier **NŒUD.oc**. Il s'agit d'un code objet écrit dans un formalisme OC commun à Esterel et Lustre. Il s'agit d'une description du code Lustre sous forme d'**automates** non nécessairementy minimal.

lustre prog.lus NŒUD

Mémo 2 : Utilisation de Lustre

Génération de code C à partir de .oc :

`poc NŒUD.oc`

ou

`ec2c NŒUD.oc`

Minimalisation de l'Automate :










Il est possible de rendre minimal l'automaté généré \Rightarrow on diminue son nombre d'états. :

`Ocmin NŒUD.oc -c`

Le compilateur Lustre est capable de produire directement un automate minimal en utilisant l'option demand.

`lustre prog.lus NŒUD -demand -v`

Bibliographie

-  Béatrice BERARD, Michel BIDOIT, François LAROUSSINIE, Antoine PETIT, Philippe SCHNOEBELEN,
Vérification de logiciels, Techniques et outils du model-checking
Vuibert Informatique , 1999
-  René DAVID, Karim NOUR, Christophe RAFFALLI,
Introduction à la logique : Théorie de la démonstration
Dunod, 2003
-  N. HALBWACHS, P. CASPI, P. RAYMOND, D. PILAUD,
The synchronous dataflow programming language LUSTRE
IMAG/LGI - Grenoble; VERILOG - Grenoble
-  N. HALBWACHS, P. RAYMOND,
A tutorial of Lustre
2002
- 
Lustre Language Reference Manual
1997
-  Nil GEISWEILLER,
Langage synchrone LUSTRE
ONERA, cours DESS-CAMSI, INSA, 2005
-  Frédéric BONIOL,
Outils et méthodes pour le temps réel : l'approche synchrone
ONERA, cours
-  F. COTTET, E. GROLLEAU,
Informatique Industrielle
ENSMA, cours
-  F. COTTET, Y. AIT-AMEUR,
Les systèmes informatiques temps réel : langages réactifs
ENSMA, cours