Travaux Pratiques : Langage LUSTRE

ENSA -2006

Cortier Alexandre
ONERA (Office nationale d'Etudes et de Recherches Aérospatiales)
email : alexandre.cortier@cert.fr
email : alexandre.cortier@laposte.net

10 novembre 2006

Table des matières

0.1 Exercices : Quelques noeuds simples

Question : Ecrire un nœud `Edge()`, qui prend en entrée un flot de booléen et un flot de sortie booléen qui renvoie vrai à chaque fois que le flot d'entrée passe de `false` à `true`.

Question : Ecrire un nœud `Compteur()`, qui prend en entrée un flot de booléen `reset`, qui renvoie un flot de sortie d'entiers et qui incrémente de 1 sa sortie à chaque instant. Le compteur est réinitialiser quand `reset` vaut `true`.

Question : Ecrire un nœud `Osc()` qui génère le flot de sortie : `(true, false, true, false, ...)`

Question : Ecrire un nœud `Compteur2()` qui se comporte comme le nœud `Compteur()` mais qui est deux fois plus lent. Utiliser les noeuds `Osc()`, `Compteur()` et les opérateurs `when` et `current`.

Question : Ecrire le nœud `Osc2()` qui génère le flot de sortie : `(true ,true ,false ,false , true, true ,false ,false ...)`.

0.2 Exercice : Notion d'Observateur

Les observateurs sont des programmes particuliers utilisés pour exprimer des propriétés ou des hypothèses. Leur caractéristique principale est d'avoir une seule sortie booléenne (qu'on nomme par convention ok).

Cette sortie doit rester vraie aussi longtemps que les entrées de l'observateur satisfont la propriété. Par abus de langage, on dit que l'observateur ?implante ? la propriété.

Ces observateurs peuvent être utilisés pour vérifier une propriétés `Phi` sur un programme `Pg`. Si nous souhaitons prouver cela sans utiliser l'outil graphique **xlesar** nous devons construire un noeud `Pg_Phi` et appeler l'outil **lesar** sur ce noeud. La figure 1 montre la manière dont on construit le programme de vérification `Pg_Phi`.

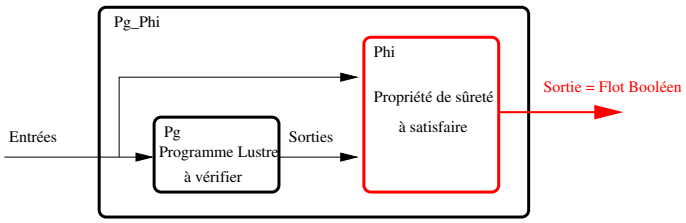

FIG. 1 –

Comme nous pouvons le constater, le noeud observateur `Phi` "implémentant" la propriété à vérifier prend en entrée : (1) les entrées du programme à vérifier `Pg` (2) les sorties du programme `Pg`. Le programme `Pg_Phi` de vérification appelle les noeuds `Pg` et `Phi`. Ce programme possède une seule sortie booléenne qui doit être vraie à tout instant.

Question : Ecrire la propriété qui énonce : ‘‘*Quand la sortie de* `osc2()` *passe de* `false` *à* `true` *alors, à l'instant suivant la sortie est encore vrai*”. Vérifier cette propriétés avec le prouveur lesar (et non pas l'outil graphique **xlesar** !) en utilisant le principe d'observateur vu à la section précédente.

Nous allons maintenant réaliser dans ce qui suit un noeud observateur permettant "d'implémenter" la propriété de sûreté suivante : *un événement "X" doit impérativement apparaître au moins une fois entre un événement "A" et un événement "B"*.. Il s'agit en quelque sorte de vérifier le bon ordonnancement d'un programme. Pour spécifier ce noeud, nous commençons par définir des noeuds intermédiaires : `implies`, `never` et `since`.

Question : Ecrire le noeud `implies(A,B:bool) returns(AimpliesB:bool);` qui implémente l'implication logique ordinaire.

Question : Ecrire le noeud `never(B:bool) returns(neverB:bool);`. Ce noeud retourne `true` aussi longtemps que son entrée n'a jamais été égale à `true`. Si l'entrée a été égale à `true` une fois, alors la sortie restera `false` pour toujours.

Question : Ecrire le noeud `since(X,Y:bool) returns(XsinceY:bool);`. La sortie de ce noeud vaut `true` si et seulement si : (1) soit la seconde entrée n'a jamais pris la valeur `true` (2) ou bien la première entrée a été `true` au moins une fois depuis la dernière valeur `true` de la seconde entrée.

Question : Ecrire un observateur `once_from_to(X, A, B : bool) returns (ok : bool)` implémentant la propriété de sûreté : *un événement "X" doit impérativement apparaître au moins une fois entre un événement "A" et un événement "B"*.. Vous utiliserez les noeuds définis précédemment.

0.3 Exercice : Portillon dans le métro

Pour éviter l'ouverture/fermeture répétée des portillons de sortie du métro, la RATP a déployé à St Lazare un mécanisme de détection des sorties des usagers.

La porte reste ouverte en permanence, deux cellules photo-sensibles (notées A et B) contrôlent le passage des usagers.

Le passage s'effectue de A vers B. Tout passage de B vers A doit déclencher la fermeture du portillon, représentée par une variable booléenne alarm.

Question : Coder en Lustre le noeud Portillon, il aura pour spécification :

```
node Portillon (A, B: bool) returns (alarm: bool);
```

On fera l'hypothèse que A et B (en utilisant le mot clef `assert`) ne peuvent être simultanément vrais, du fait des contraintes physiques du système.

Question : Simuler et vérifier ce noeud.

0.4 Exercice complet : le chronomètre STOP_WATCH

0.4.1 Cahier des charges

Le but est de spécifier le comportement d'un chronomètre simplifié. Le chronomètre dispose 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.

Nous chercherons à valider la propriété de sûreté suivante : *" Il n'est pas possible de remettre à zéro le compteur en cours d'exécution"*.

Afin de modéliser le comportement du chronomètre, nous allons tout d'abord définir des opérateurs intermédiaires (ide. des noeuds Lustre). Utiliser le simulateur Luciole à chaque étape pour vérifier que le noeud construit se comporte comme vous le souhaitez.

0.4.2 L'opérateur COUNT

Question : Réaliser un opérateur `COUNT(reset, X : bool) returns (C: int)` dont la spécification est la suivante :
- la sortie `C` est remise à zéro 0 si reset, sinon il est incrémenter si `X` ;
- Tout se passe comme si `C` valait 0 à l'origine.

0.4.3 L'opérateur SWITCH

Question : Réaliser un opérateur `SWITCH (on, off : bool) returns (S : bool)` dont la spécification est la suivante :
- La sortie `S` passe de `false` à `true` si `on` ;
- La sortie `S` passe de `true` à `false` si `off` ;
- Tout se comporte comme si `S` était `false` à l'origine ;
- Tout doit fonctionner correctement même si `off` et `on` sont les mêmes.

0.4.4 L'opérateur STOP_WATCH

Question : Réaliser le noeud `STOP_WATCH(on_off, reset, freeze : bool) returns (time : int)` en utilisant les opérateurs définis précédemment.

Question : Vérifier la propriété de sûreté suivante :*" Il n'est pas possible de remettre à zéro le compteur en cours d'exécution"*. Penser à utiliser une hypothèse sur l'environnement extérieur afin de réduire l'automate de vérification.

0.5 Exercice complet : Contrôleur de feux de voiture

Une voiture dispose de trois types de lampes : veilleuses, codes et phares. Le conducteur dispose d'une manette qui dispose de plusieurs degrés de liberté.

On souhaite décrire en Lustre le module de contrôle des feux d'une voiture. L'utilisateur entre ses ordres via une manette et une série d'interrupteurs, agissant sur les phares de la voiture.

0.5.1 Fonctionnement du contrôleur

Voici la description du contrôleur de phare :

1. La manette peut être tournée dans le sens direct (TD) ou indirect (TI) ;
2. A partir d'une situation initiale où tout est éteint, TD allume les veilleuses, un second TD éteint les veilleuses et allume les codes ;
3. Lorsqu'on est en codes ou en phares, TI les éteint et rallume les veilleuses, un second TI éteint tout ;
4. Le fait de tirer la manette vers l'avant (CP) permet de commuter entre codes et phares ; lorsqu'on est en codes, CP éteint les codes et allume les phares, un second CP éteint les phares et rallume les codes ;
5. Le conducteur ne peut pas simultanément tourner et tirer la manette.

0.5.2 Choix des variables d'entrée et de sortie du contrôleur

Pour les variables d'entrée, on prendra trois variables booléennes TD, TI et CP, vraies aux instants où l'action correspondante est effectuée par le conducteur.

Pour les sorties, on choisit trois flots qui représentent l'état des lampes : veilleuses (resp. codes, phares) est vrai tant que les veilleuses (resp. les codes, les phares) sont allumées, et est faux tant qu'elles sont éteintes.

0.5.3 Question

Écrire et simuler un programme Lustre qui décrit le noyau du contrôleur.

Annexe A

Rappel sur le langage Lustre

A.1 Lustre : Vocabulaire et positionnement du langage

Lustre est **un langage de haut niveau** $\Rightarrow$ pas de problèmes fins d'implémentation concrète du système. Par exemple, et contrairement à la programmation classique comme le C, pas de gestion de pointeurs....

Lustre est **un langage réactif** permettant la modélisation de systèmes temps réel. Plus en détail, nous pouvons dire que Lustre est **un langage synchrone** reposant sur le *paradigme du synchronisme fort* et dispose d'une *sémantique fonctionnelle à flots de données* .

Lustre suit une **approche algébrique** : le modèle décrit en Lustre repose sur un ensemble d'équations entre flots de données. Afin de définir un système **déterministe**, certaines contraintes sont imposées sur la structure de l'ensemble des équations (ex : non circularité, ...).

Lustre est une Technique Formelle lorsqu'il est associé à son *Model-Checker Lesar* (système de preuve). Lustre est donc le langage de modélisation des systèmes temps réels basé sur un cadre mathématique. Le système de preuve exploite cette modélisation formelle décrite en Lustre et la définition de propriétés, exprimées dans une logique adéquate (Safety Logic pour Lesar), afin de prouver que le modèle du système se comporte comme on l'attend : c'est à dire que la structure du modèle engendre les propriétés attendues...

A partir du modèle Lustre , il est possible de **générer du code source C**. Ce code source peut alors être intégré sur un système d'exploitation temps réel et permettre la gestion concrète d'un système de commandes (ou système temps réel en général).

A.2 Généralité

Lustre est un langage déclaratif (ide. algébrique) fonctionnel à flots de données suivant le paradigme du synchronisme fort. Ceci implique que :

1. Un **flot = séquence de valeurs** (finie ou infinie) ;
2. on considère le temps comme discret : **temps = succession d'instants** ;
3. hypothèse lié au synchronisme fort : le **temps d'exécution des fonctions est nul** ;
4. conséquence du synchronisme fort : à un instant donné, les flots de sorties sont définies instantanément en fonction des flots d'entrée et internes.

Un programme Lustre est appelé **nœud** (**node**). La structure d'un programme Lustre est donnée en figure A.1.

const name = val **;**

node nom($E_1 : Type_{E_1}$; ...) **returns** ($S_1 : Type_{S_1}$) **;**
var $V_1 : Type_{V_1}$; ... **;**
let
  assert(Exp_1) **;**
  Eq_1 **;**
    ...
  Eq_j **;**
    ...
tel

FIG. A.1 – Structure d'un programme Lustre

Quelques Remarques sur la méthodologie de programmation en Lustre :

1. les équations n'ont pas d'ordre ;
2. la circularité dans la définition des flots est prohibée ;
3. on doit toujours déclarer au moins un flot d'entrée, même si celui-ci n'est pas utilisé pour définir le(s) flot(s) de sortie ;
4. prendre garde à la définition des équations de flots. Rappelons qu'un flot est l'association d'une séquence de valeurs rythmée par une horloge. Une équation Lustre est bien typée si et seulement si :
   (a) le type de la séquence de valeurs entre membre droit et gauche de l'équation est correcte (ex : flot d'entiers = flot d'entiers...)
   (b) les flots du membre droit et gauche sont rythmés par la même horloge !

   Ceci limitera un grand nombre d'erreurs de compilation...

A.3 Données et opérateurs d'un programme Lustre

1. **Les types de données** : `bool`, `int`, `real`, `array`
2. **Opérateurs classiques** :
   – opérateurs arithmétiques : `+`, `-`, `*`, `/`, `div`, `mod`
   – opérateurs booléen : `and`,`or`,`xor`,`not`
   – opérateurs de comparaison : `=`, `<`, `<=`, `>`, `>=`, `<>`
   – opérateur de contrôle : `if then else`
3. **Opérateurs temporels** :
   – `pre` : précédent
   – $->$: suivi de
   – `when` : sous-échantillonnage
   – `current` : sur-échantillonnage

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)	*nil*	X_1	X_2	X_3	X_4	X_5	X_6	X_7	...
Y −> 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	*nil*	X_2	X_2	X_4	X_4	X_4	X_7	X_8	...
pre(Z)	–	*nil*	–	X_2	–	–	X_4	X_7	...
0 −> pre(Z)	–	0	–	X_2	–	–	X_4	X_7	...

FIG. A.2 – Mémo sur les opérateurs temporels

Annexe B

Tutoriel : outils Lustre

B.1 Mise en place

Vous pouvez télécharger Lustre (pour plates-formes GNU/Linux, x86) à l'URL suivante : http ://www-verimag.imag.fr/ raymond/edu/distrib/linux/index.html

Une fois l'archive téléchargée, décompressez la à l'aide de la commande :

```
tar zxvf lustre-v4-II.f-linux.tgz.gz
```

L'archive se décompresse dans le répertoire `PWD/lustre-v4-II.f-linux`.

Pour compléter l'installation, il faut positionner la variable d'environnement `LUSTRE_INSTALL` à *PWD/lustre-v4-II.f-linux* et ajouter `LUSTRE_INSTALL/bin` à la variable d'environnement `PATH`.

Sous BASH, cela implique de taper les commandes suivantes dans le shell que vous utiliserez, ou dans le fichier de configuration correspondant :

```
export LUSTRE_INSTALL=$PWD/lustre-v4-II.f-linux
export PATH=$LUSTRE_INSTALL/bin:$PATH
```

Sous TCSH, cela implique de taper les commandes suivantes :

```
setenv LUSTRE_INSTALL "${PWD}/lustre-v4-II.f-linux"
setenv PATH "${LUSTRE_INSTALL}/bin:${PATH}"
```

B.2 Utilisation des outils Lustre

B.2.1 Ecriture du programme edge

Dans ce premier exercice, on va analyser l'exemple fourni en cours : `edge`.

Ce programme détecte un front montant, c'est à dire le passage d'une variable booléenne de la valeur false à la valeur true. Nous en rappelons la spécification :

```
node EDGE (b : bool) returns (edge : bool);
let
   edge = false -> b and not pre b;
tel
```

Dans l'expression qui définit EDGE, on retrouve plusieurs opérateur Lustre dont nous rappelons la signification :

- Le "et logique" (and, $\wedge$) et la négation (not,$\neg$), la constante booléenne "faux" (mot-clé `false`). Les autres opérateurs sont particuliers à Lustre :
- La flèche (->) permet de distinguer la valeur initiale du flot (en partie gauche) des autres valeurs (en partie droite). La première valeur du flot edge est donc false, les valeurs successives sont définies par l'expression "b and not pre b".
- L'opérateur pre (pour précédent) permet de faire référence au "passé" : la valeur à l'instant précédent.

Au final, le noeud EDGE définit le flot modélisé par l'équation suivante :

$$\begin{array}{|l} EDGE(1) = faux \\ \forall t > 1, EDGE(t) = b(t) \wedge \neg b(t-1) \end{array}$$

Question 1 : Écrire le noeud `EDGE` dans un fichier edge.lus

B.2.2 Simulation

Dans cette section, nous nous intéressons à la simulation de programmes Lustre à l'aide du simulateur graphique **luciole**.Luciole requiert le nom du fichier `.lus` à lire, et le nom du noeud à simuler.

```
luciole edge.lus EDGE
```

Cette commande ouvre une fenêtre de simulation. Elle présente une série de boutons correspondant aux entrées/sorties du noeud.Dans le cas du noeud `EDGE`, on dispose d'un bouton pour l'entrée `b` et une "lampe" pour la sortie edge (Note : une lampe est un label qui s'affiche en rouge quand edge est vrai, et en gris quand edge est faux). Par défaut, avec cette interface, cliquer sur le bouton "b" provoque un cycle de calcul avec b vrai, cliquer sur le bouton "Step" provoque un cycle de calcul avec b faux. Le résultat est renvoyé sur la lampe edge.

FIG. B.1 – Commande : luciole edge.lus EDGE

Chronogrammes : La commande "Tools ! sim2chro" ouvre l'outil de visualisation de chronogramme. L'évolution des variables b et edge est alors automatiquement visualisée sous forme de chronogramme, indexé par les entrées du noeud..

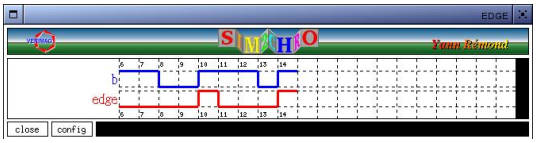

FIG. B.2 – Chronogramme d'exécution du noeud EDGE.

Modes auto-step et modes compose : Par défaut, l'interface d'exécution de *luciole* est en mode ?auto-step ?, c'est-à-dire que le noeud est activé dès qu'on active sur une entrée booléenne ou sur Step. Avec la commande "Clocks ! Compose", on passe en mode compose : les entrées booléennes deviennent des interrupteurs qu'on peut activer/désactiver sans provoquer un pas de calcul. Le pas de calcul est déclenché en appuyant sur le bouton Step.

Ce mode est nécessaire pour des programmes qui ont plusieurs entrées, pour pouvoir composer des états où plusieurs entrées sont vraies simultanément.

Horloge temps-réel : Utiliser le bouton *Step* peut devenir gênant, on peut alors utiliser une horloge "temps-réel".

Le menu "Clocks ! Real time clock" permet d'activer/désactiver le mode temps-réel. Dans ce mode, le pas de calcul est automatiquement déclenché à intervalles réguliers. La période de l'horloge peut être modifiée avec "Clocks ! Change period". Elle est exprimée en milli-secondes.

Attention ! L'aspect temps-réel est relatif et dépend de la plate-forme utilisée. On utilise un système multi-tâches multi-utilisateur qui n'offre aucune garantie temps-réel. En pratique, si la machine est un peu trop chargée, le simulateur aura du mal à soutenir une période inférieure à quelques ms.

B.2.3 Compilation en C

La compilation d'un programme Lustre se déroule en plusieurs phases :

– La pré-compilation transforme un programme Lustre en programme "Lustre noyau", aussi appelé code expansé. Cette phase est réalisée par la commande :

```
lus2ec edge.lus EDGE
```

Elle produit un fichier EDGE.ec, qui dans ce cas précis est pratiquement très proche du programme initial edge.lus.

– La compilation proprement dite transforme un programme ec en programme C. La commande est :

```
ec2c EDGE.ec -v
```

L'option -v permet de passer en mode "verbeux" : des informations additionnelles sur le déroulement de la compilation sont affichées (Note : tous les outils Lustre disposent d'une option -v).
Le résultat de la compilation est un fichier EDGE.c qui contient le noyau réactif du programme, ainsi que le fichier EDGE.h qui contient les informations nécessaires à l'utilisation du noyau.

– La compilation en C ne produit que le noyau réactif du système, c'est normalement à l'utilisateur d'écrire un programme principal qui se charge de l'acquisition des entrées et de la visualisation des sorties. Le compilateur ec2c propose une option -loop qui produit, en plus du noyau réactif, un programme principal standard. La commande :

```
ec2c EDGE.ec -loop -v
```

produit `EDGE.h`, `EDGE.c` plus un fichier `EDGE_loop.c` qui contient une fonction main standard. Pour des programmes simples (comme edge) ce programme principal peut être utilisé tel-quel.
– Pour obtenir un exécutable, il faut finalement utiliser un compilateur C, par exemple GCC sur GNU/Linux. La commande :

```
gcc EDGE.c EDGE_loop.c -o EDGE
```

compile les deux fichiers C, et produit l'exécutable EDGE. Le programme principal standard est en fait très rudimentaire : l'utilisateur doit taper au clavier, une par une, les entrées du programme.
– Lux permet d'enchaîner toute ces phases automatiquement. Il gère en particulier la phase de liaison avec les différentes bibliothèques nécessaires à la construction de l'exécutable. La commande :

```
lux edge.lus EDGE
```

enchaîne automatiquement toutes les phases et produit un exécutable EDGE.

B.2.4 Compilation en automate

Une solution alternative pour la compilation d'exécutable consiste à générer le noyau réactif du noeud Lustre sous la forme d'un automate à états finis :
– La phase de pré-compilation est la même :

```
lus2ec edge.lus EDGE
```

produit le fichier `EDGE.ec`. La commande :

```
ec2oc EDGE.ec
```

produit un automate, dans un format particulier appelé oc (dans un fichier `EDGE.oc`). On génère le code C correspondant avec la commande :

```
poc EDGE.oc
```

On obtient alors, comme avec `ec2c` un fichier `EDGE.c` et un fichier `EDGE.h`.
– La commande `poc` dispose aussi d'une option `-loop`. On peut donc enchaîner les commandes suivante pour obtenir un programme exécutable edge :

```
poc EDGE.oc -loop
gcc EDGE.c EDGE_loop.c -o EDGE
```

Cette méthode de génération de code est assez proche de la précédente, à ceci près qu'un automate fournit un modèle pertinent pour raisonner sur les exécutions possibles du programme, et ainsi analyser son espace d'états.

Si on dispose de l'outil autograph (1), on peut visualiser assez simplement l'automate, et voir s'il correspond bien à ce qu'on désirait. Pour cela, on peut utiliser lus2atg :

```
lus2atg edge.lus EDGE
```

Cette commande produit un fichier `edge.atg` directement exploitable par autograph. On tape alors la commande :

```
atg EDGE.atg
```

qui ouvre l'explorateur d'automate. L'exploration se fait à la souris, choisir ?Placing | Explore ? puis interagir avec le diagramme pour faire apparaitre les différents états. On obtient alors le graphique suivant :

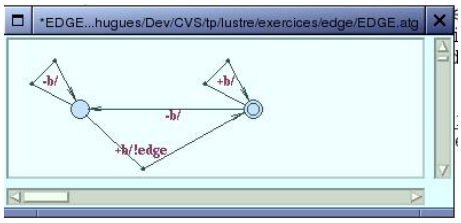

FIG. B.3 – L'automate du noeud edge

Sur cette figure, on remarque les choses suivantes :

- Dans l'état initial (double cercle), si b est vrai, on reste dans cet état, et si b est faux on passe à l'état suivant.
- Dans le deuxième état, tant que b est faux on "boucle", et si b est vrai on "émet" edge (c'est-à-dire que edge est vrai), et on retourne dans l'état initial.

B.2.5 Vérification de propriétés

Lustre dispose d'un outil de vérification formelle de propriétés. L'utilisateur exprime un ensemble de propriétés que doit vérifier le programme, et l'outil indique si ces propriétés sont satisfaites, ou fournit un contre-exemple.

On utilise ici l'outil **xlesar** : une interface graphique du vérificateur lesar. Pour lancer l'outil, taper : xlesar.

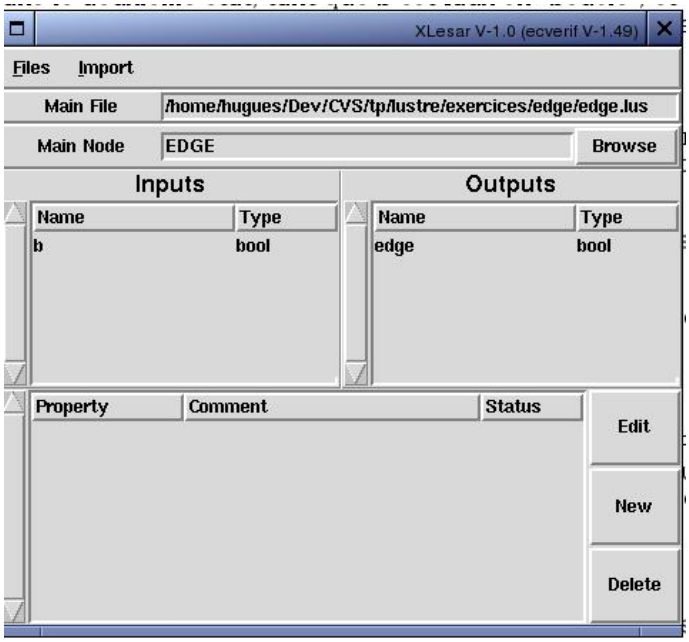

FIG. B.4 – Interface de vérification

Sélectionner le fichier et le noeud à analyser (commande `Browse`). Les entrées/sorties du noeud sont affichées.

La partie basse de la fenêtre permet de rentrer les différentes propriétés que l'on souhaite vérifier (commandes New , puis sur Edit). Cette dernière commande ouvre une fenêtre d'édition de propriété. La partie gauche est l'éditeur de propriété proprement dit, la partie droite permet de lancer la vérification.

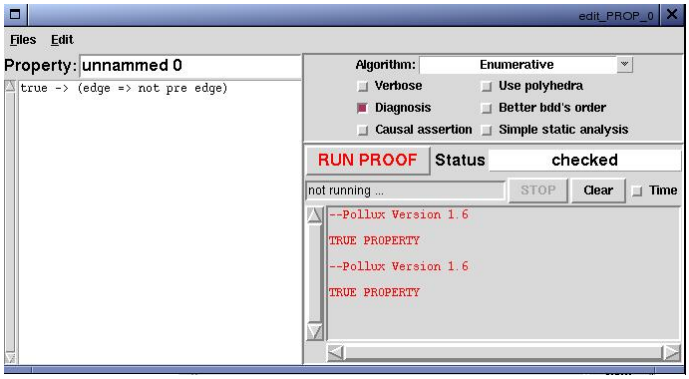

FIG. B.5 – L'outil **xlesar** en action

Une propriété triviale : Le point important est que la propriété doit être exprimée sous la forme d'une expression booléenne Lustre. Par défaut, cette définition est `true`, et est donc vraie. On peut tout de même essayer de lancer le vérificateur en appuyant sur ?RUN PROOF ?. On obtiendra naturellement (dans la fenêtre de dialogue) le résultat "`TRUE PROPERTY`".

Une propriété vraie : On va maintenant exprimer et vérifier une propriété plus complexe. On cherche à évaluer la propriété suivante : *La sortie EDGE ne peut pas être vraie à deux instants consécutifs*.

Pour utiliser le vérificateur, on doit tout d'abord traduire cette propriété en une expression Lustre. On va simplement dire que, initialement, la propriété est toujours vérifiée, puis à chaque instant, si `edge` est vraie alors `edge` était faux à l'instant précédent. En utilisant l'opérateur "implication logique" (=>), cela donne :

```
true -> (edge => not pre edge)
```

On peut alors lancer le prouveur, en utilisant l'option ?Verbose ? pour avoir des informations sur le déroulement de la preuve. On peut essayer les différents algorithmes proposés :
Enumerative , Symbolic forward et Symbolic backward. On doit évidemment obtenir la même réponse pour chaque algorithme.

Complexité de la preuve : Grâce au mode verbeux, on peut se faire une idée de la "complexité" de la preuve : avec l'algorithme énumératif, le nombre d'états et de transitions sont une bonne mesure de la complexité de la propriété (4 états et 8 transitions pour cet exemple). En symbolique, c'est le nombre d'itérations qui mesure la complexité (2 pas de calculs en mode "forward" et en mode "backward").

Un autre exemple de propriété vraie est edge ne peut être vrai que si b est vrai, ce qu'on traduit simplement par : `edge => b`.

Cette propriété est beaucoup plus évidente que la précédente : elle est, pour ainsi dire, écrite dans le programme. On peut d'ailleurs observer cette simplicité en regardant la complexité de la preuve : 2 états et 2 transitions en énumératif, 2 pas de calcul en "forward", et 0 pas de calcul en mode "backward".

Une propriété fausse : On va maintenant voir ce qui se passe pour une propriété fausse, par exemple :

"si b est vrai, alors edge est vrai"

Ce qu'on traduit par `b => edge`. Si on n'utilise aucune option, le prouveur se contente de répondre `FALSE PROPERTY`. Il faut utiliser l'option Diagnosis pour obtenir un contre exemple :

```
DIAGNOSIS:
--- TRANSITION 1 ---
b
```

L'interprétation est la suivante : dès la première réaction du programme, si b est vrai, la propriété n'est pas satisfaite. Ce résultat est attendu : la sortie est toujours fausse à l'instant initial. Il faut donc modifier la propriété :

"Sauf à l'instant initial, si b est vrai, alors edge est vrai."

Traduite en Lustre, cette propriété devient : `true -> (b => edge)`.

Là encore on va obtenir un résultat négatif, avec un nouveau contre-exemple :

```
DIAGNOSIS:
--- TRANSITION 1 ---
b
--- TRANSITION 2 ---
b
```

qui stipule que la propriété est fausse si b est vrai deux fois de suite. Mieux vaut alors se rentre à l'évidence : la propriété est bien totalement fausse.

Annexe C

Description des outils utilisés pour le TP

Un programme Lustre peut être édité avec un éditeur de texte classique (emacs, textedit,...).

C.1 Compilation du source Lustre et génération de l'automate

Le compilateur Lustre prend en entrée un fichier *nom.lus* qui représente le programme principal *nœud* et produit un fichier *nœud.oc*. On utilise la commande suivante pour lancer la compilation :

lustre <*nom.lus*> <*nœud*> [<*options*>]

Les principales options sont :

- **-default** : les options par défaut.
- **-v** : mode verbeux, affiche l'état de la compilation.
- **-o** <*file*> : le fichier de sortie sera *file.oc*
- **-const** : optimise le calcul des constantes.
- **-data** : l'algorithme de construction de l'automate tient compte des données.
- **-demand** : l'algorithme de construction de l'automate tient compte des sorties attendues.

C.2 Minimalisation de l'automate

Pour minimiser l'automate produit par le compilateur *nom.oc* en *nom_min.oc*, on utilise :

ocmin[<*options*>] <*nom.oc*>

Avec les options :

- **-o** <*file*> : le fichier de sortie sera *file.oc*
- **-v** : mode verbeux.

Cet outil procède à un test d'équivalence d'états dans l'automate *nom.oc*

C.3 Simulation graphique

Pour faire une simulation graphique de l'automate correspondant à un nœud, on utilise l'utilitaire **Luciole**, qu'on lance avec la commande :

luciole <*prog.lus*> <*nœud*>

L'interface (C.3) est composée de :

1. **La fenêtre principale** qui représente les *entrées et les sorties du nœud* via un ensemble de boutons. Le menu *clocks* permet de choisir parmi différents modes de simulation :
   (a) **auto-step** : une seule entrée peut être modifiée à chaque pas de l'horloge de base. Chaque click sur un bouton de variable d'entrée exécute un cycle du noeud.
   (b) **compose** : plusieurs entrées peuvent être modifiées à chaque pas de l'horloge de base. Les boutons des variables d'entrée deviennent des *check-boxes*. Il faut alors utiliser le bouton step pour exécuter un cycle.
   (c) **Real Time Clock** : les cycles sont exécutés suivant l'horloge de base dont on peut modifier la période.
2. Une fenêtre (*menu Tools → sim2chro X11*), qui affiche les *chronogrammes des flots d'entrée et de sortie* pendant la simulation.

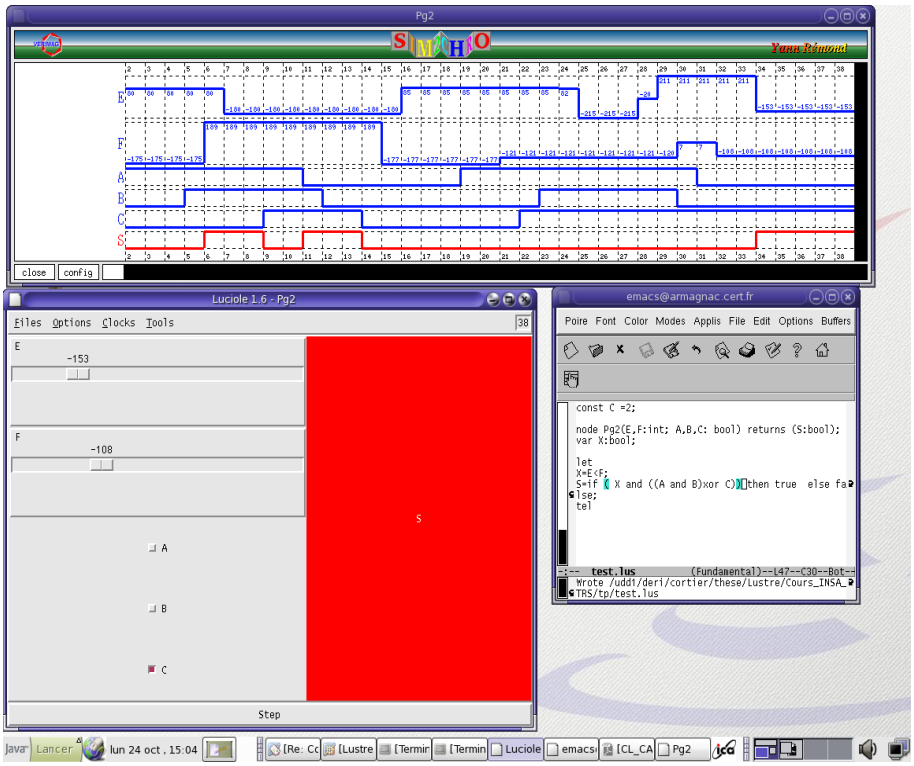

FIG. C.1 – L'interface de simulation de Luciole.

C.4 Génération de code C

Pour générer un fichier de code C (nom.c) à partir de l'automate (nom.oc) on utilise :

poc <*nom.oc*> *-loop*

gcc <*nom.c*> <*nom_loop.c*> -o *nom*

Le fichier *nom.c* contient la fonction de transition de l'automate, le fichier *nom_loop.c* contient la boucle principale qui appelle la fonction de transition. Le résultat se présente comme un filtre UNIX prenant à chaque cycle les valeurs des entrées sur **stdin** et affichant les sorties **stdout**.

On peut aussi utiliser le script suivant (si gcc est correctement installé, à tester...) qui prend un noeud et génère directement un exécutable :

lux <*nom.lus*>

Une fois le code C généré, il faut interfacer les fonctions issues du code Lustre avec l'environnement extérieur. Outre les fonctions définies par les noeuds du programme Lustre, des fonctions de lecture/écriture des entrées/sorties mémoires sont générées. L'interfaçage consiste à produire " manuellement " les fonctions d'acquisition des entrées (sur les bus, les capteurs . . .) et de production des sorties (sur les bus vers les actionneurs . . .). Le process de génération et d'interfaçage du code Lustre est schématisé en figure C.4

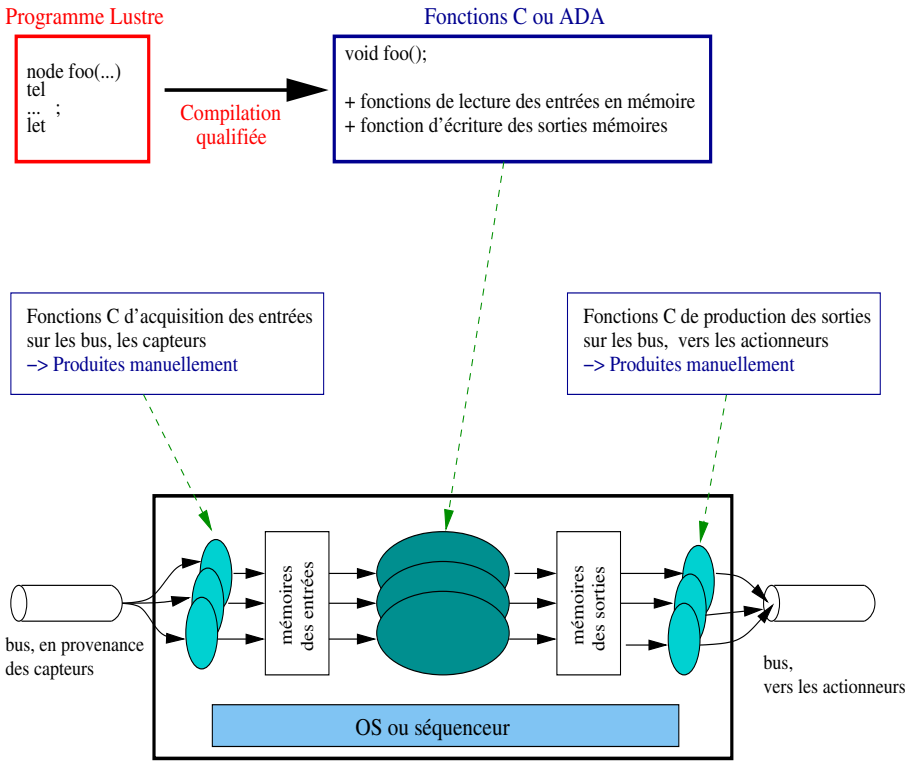

FIG. C.2 – Génération de code c et implantation.

C.5 Vérification formelle

Pour vérifier une propriété sur un nœud, il faut définir un nœud de vérification, constitué du produit parallèle du nœud à vérifier et d'un nœud observateur de la propriété à vérifier (C.5). Le nœud de vérification doit avoir un unique flot de sortie booléen, qui indique à chaque cycle si la propriété est vérifiée ou non en fonction des sorties et des entrées du nœud testé. On utilise la commande :

lesar <*nom.lus*> <*nœud*> [<*options*>]

Les principales options sont :

- **-v** : mode verbeux.
- **-diag** : affiche diagnostic de vérification.
- **-dbg** : mode debug.
- **-enum** : utilise une méthode de résolution énumérative.
- **-forward** : utilise un algorithme de résolution symbolique avant.
- **-backward** : utilise un algorithme de résolution symbolique arrière.
- **-poly** : utilise une librairie adaptée pour la représentation des propriétés impliquant des nombres entiers (systèmes de contraintes linéaires seulement) ;

L'outil Lesar cherche à prouver que le flot booléen représentant la propriété reste *vrai* quelle que soit l'exécution du système par la technique du **Model-Checking** (vérification exhaustive sur modèle).

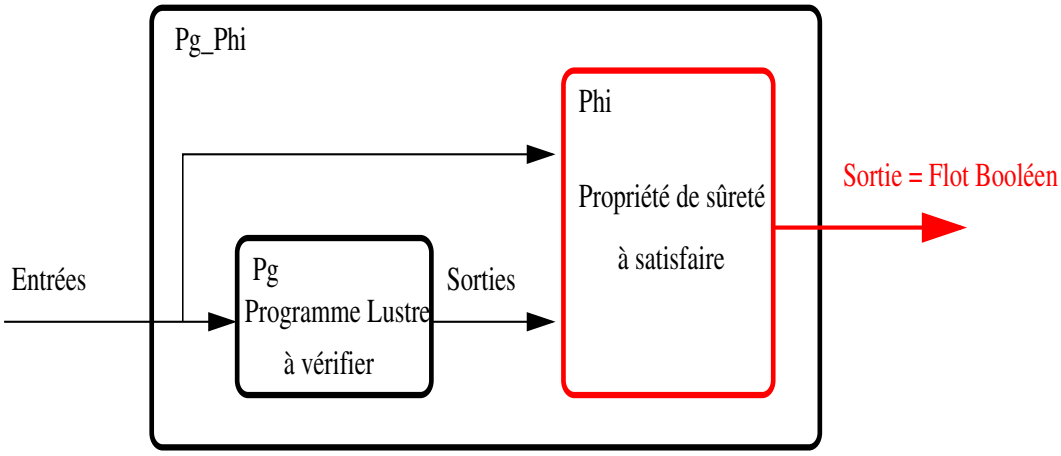

FIG. C.3 – Schéma général de vérification en Lustre.