Mdl vers Lustre vers Osek

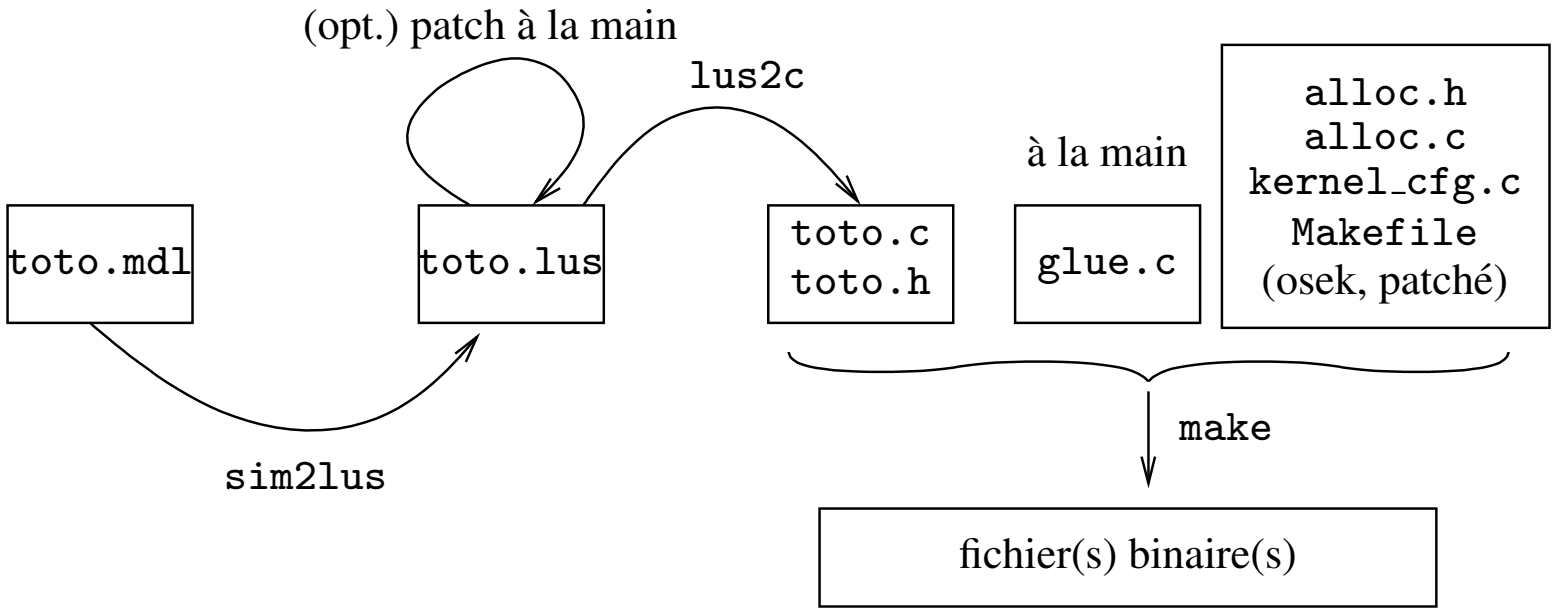
Pascal Raymond

Verimag/CNRS

3A SLE

Sommaire

Schéma général



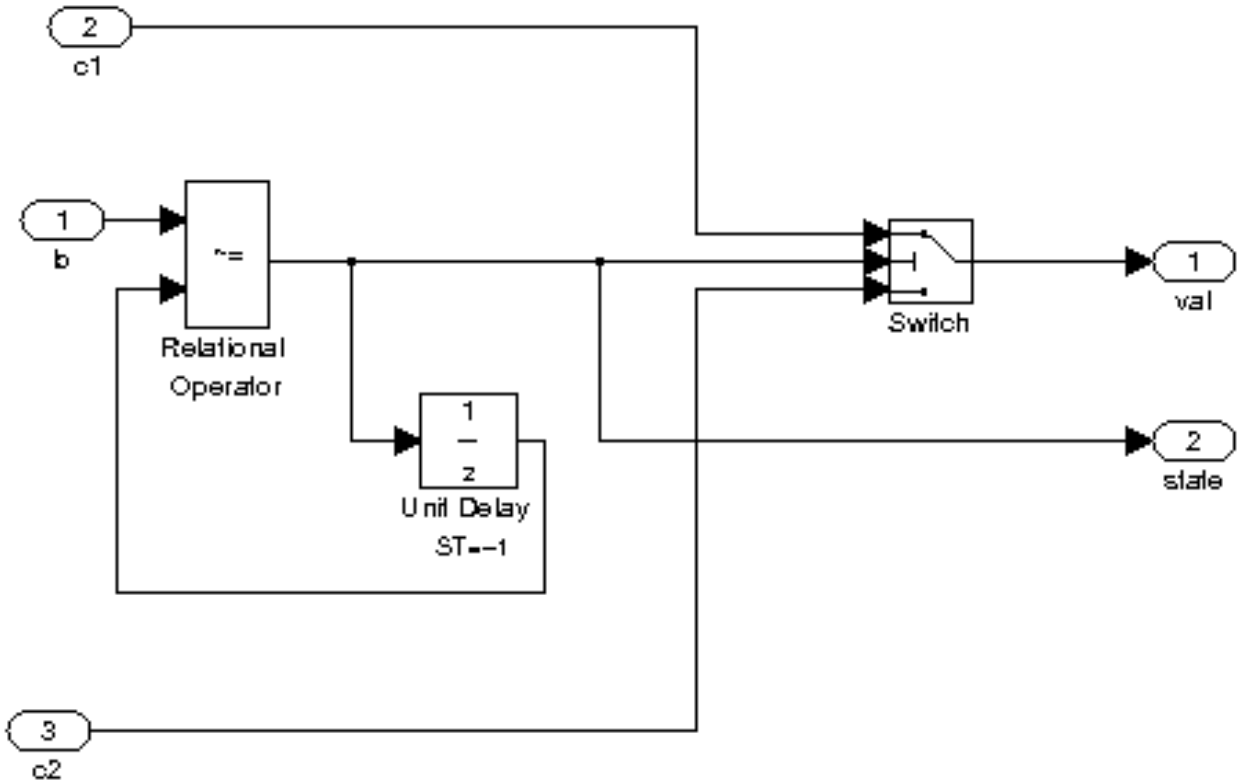
Sommaire

Spécifications

- on utilise un capteur mécanique en 3 (noté b)
- on utilise deux capteurs de lumières en 1 et 2 (notés c1 et c2)
- on affiche ”c1 = <valeur du capteur c1>” ou bien ”c2 = <valeur du capteur c2> :
  - par défaut c1,
  - on change chaque fois qu’on appuie sur b

Le modèle simulink

- deux entrées entières c1 et c2, une entrée logique b
- une sortie logique ”state” et une sortie entière ”val”



Contraintes sur le modèle

Le traducteur Simulink to Lustre est par nature très contraignant :

- le programme doit être tout seul dans un fichier, `testlight.mdl`
- les paramètres de simu doivent être *Fixed-step/discrete*
- les types de données doivent être renseignés (boolean, et int dans ce cas)

Sommaire

Le résultat sur l'exemple

La commande `sim2lus testlight.mdl` produit le fichier `testlight.lus` (légérement simplifié pour tenir sur une page) :

```
node testlight (b:bool; c1:int; c2:int)
returns (val:int; state:bool);
var Relational_Operator:bool;
  Unit_Delay:bool;
let
  Relational_Operator = b <> Unit_Delay;
  val = if Relational_Operator then c1 else c2 ;
  Unit_Delay = false -> pre Relational_Operator ;
  state = Relational_Operator ;
tel
```

Lustre vers C

La commande `lus2c testlight.lus testlight` produit :

- `testlight.c`
- `testlight.h`

Ce qu'il y a dedans :

- c'est *conceptuellement* une classe (au sens objet) bien que programmé en C de base
- on a un type (abstrait) `testlight_ctx`
- plus des ≪ méthodes ≫, qu'on détaille ...

Gestion des instances (1)

Le compilateur propose 3 méthodes pour gérer les instances :

- Méthode dynamique (`-ctx-heap`) :
  - ressemble le plus à ce qui se fait pour les langages objets
  - les instances doivent être créées explicitement dans le tas (heap)
    (ressemble au « new » des langages oo)
  - toutes les méthodes prennent en paramètre l'instance sur laquelle elles s'appliquent
    (équivalent du pointeur « this » des langages oo).

En particulier : on peut utiliser plusieurs instances d'un même nœud.

Gestion des instances(2)

- Méthode globale (`-ctx-global`) :
  - Pas de méthode « new », pas besoin de heap.
  - Les instances doivent être déclarées dans la zone globale.
  - ne change rien pour les méthodes : elle continuent à prendre un paramètre « this ».

Permet toujours d'avoir plusieurs instances d'un même nœud.
Pas besoin de tas, suffisant (et + efficace) pour l'embarqué.

Gestion des instances(3)

- Méthode statique (`-ctx-static`) :
  - Une unique instance par nœud.
  - Allouée statiquement dans la zone globale.
  - Pas de paramètre d'instance (de « this ») toutes les méthodes travialle directement dans la mémoire statique.

Autorise exactement une instance par nom de nœud.
Mais simplifie beaucoup l'utilisation...
Certainement suffisant pour ce qu'on veut faire

Initialisation et activation

Initialisation

- On considère ici le mode `-ctx-static`, il n'y a donc rien à faire pour allouer l'instance : elle existe déjà.
- Il faut par contre l'initialiser avec la procédure :
  `void testlight_init();`

Procédure step

`void testlight_step();`

- réalise un pas de calcul (une réaction)
- doit être intégrée dans un programme principal qui l'active au rythme voulu
- pas de paramètre : le passage se fait via *des appels de fonction*

Passage des paramètres

Fonctions d'entrée

- une pour chaque entrée, convention pour le nom, ex. :

```
void testlight_I_b(_boolean);
void testlight_I_c1(_integer);
void testlight_I_c2(_integer);
```

- définie dans `testlight.c`
- doivent être appelées par l'utilisateur (typiquement avant chaque appel du step)

Passage des paramètres

Fonctions de sorties

- une pour chaque sortie, convention pour le nom, ex. :

```
extern void testlight_O_val(_integer);
extern void testlight_O_state(_boolean);
```

- appelées dans la méthode step
- doivent être définies par l'utilisateur

Sommaire

Choix d'implémentation

Il faut faire des choix :

- on décide d'implémenter le programme par un tâche périodique, de période 50 ms (i.e. 20 Hz) sur le modèle de `modele1tache`
- on décide de l'association entrées-sorties informatiques/entrées-sorties physiques, par exemple :
  - entrées `c1` et `c2` associées à des capteurs de lumière branchés sur les ports 1 et 2,
  - entrées `b` associé à un capteur mécanique branché sur le port 3,
  - sorties reflétées par affichage sur l'écran LCD.

Initialisations OSEK

Certains *devices* nécessitent une initialisation, à mettre dans la procédure `ecrobot_device_initialize` :

```
void ecrobot_device_initialize() {
   ecrobot_set_light_sensor_active(NXT_PORT_S1);
   ecrobot_set_light_sensor_active(NXT_PORT_S2);
}
```

If faut aussi les ≪ éteindre ≫ proprement :

```
void ecrobot_device_terminate() {
   ecrobot_set_light_sensor_inactive(NXT_PORT_S1);
   ecrobot_set_light_sensor_inactive(NXT_PORT_S2);
}
```

Initialisations Lustre

Il faut initialiser le contexte du nœud, ça se fait par convention dans une procédure `usr_int` :

```
void usr_init(void) {
  // Initialisation du/des contextes lustre
  testlight_init();
}
```

Procédures de sorties

Un simple affichage sur le LCD : on ne sait pas dans quel ordre seront appelées ces fonctions dans le step.

```
void testlight_0_state(_boolean b){
   /* affiche sur la 1ere ligne */
   display_goto_xy(0,0);
   if (b) display_string("Capteur 1");
   else display_string("Capteur 2");
}
void testlight_0_val(_integer i){
   /* affiche sur la 2eme ligne */
   display_goto_xy(0,1);
   display_int(i);
}
```

Corps de la tâche

Essentiellement : positionnement des entrées et appel du step Lustre.

```
TASK(Task1){
/* Positionnement des entrées */
   testlight_I_c1(ecrobot_get_light_sensor(NXT_PORT_S1));
   testlight_I_c2(ecrobot_get_light_sensor(NXT_PORT_S2));
   testlight_I_b(ecrobot_get_touch_sensor(NXT_PORT_S3));
/* Appel du step */
   testlight_step();
/* Raffraichit le LCD */
   display_update();
/* Finalisation tâche obligatoire */
   TerminateTask();
}
```

Autres adaptations

- Il faut adapter le fichier `kernel_cfg.c` pour choisir la période.
- L'utilisation d'un opérateur Simulink (e.g. `truc`) se traduit par l'appel d'une procédure/macro (e.g. `ext_truc_call`) qu'il faudra définir en C