Programmation des systèmes réactifs Esterel (Partie 1)

EPU Electronique - Option GSE

9 octobre 2007

1 Environnement de programmation

Les manipulations se font sur la plate-forme Estérel Studio 5.3 (esterel-technologies) tournant sous Windows. Dans la suite ES = Esterel Studio.

Lancer l'application et vérifier la version (Help du menu). Cette version est liée à une licence[1].

Récupérer sur le site d'enseignement l'archive td3St.zip et l'extraire.

1.1 Découverte de l'environnement

Elle va se faire à l'aide de l'exemple ABRO. La documentation du langage peut se consulter en ligne (Help du menu). Les programmes sources sont disponibles sous TD3/src.

Mise en place

1. Créer un projet. `>File>New` ouvre un browser. Taper ABRO. Il est créé un fichier ABRO.etp.
2. Copier le fichier ABRO.strl dans le même répertoire que le projet.
3. Clic droit sur l'icône Model dans la fenêtre projet (à gauche). Sélectionner `Insert a file` et proposer ABRO.strl.
4. Sauver.
5. Pour éditer un fichier Esterel, cliquer sur son icône dans la fenêtre projet.

Configuration

Remarque : les commandes sont accessibles à partir du menu, comme décrit dans le texte. Il existe des icônes attachées à ces commandes. Il suffit ensuite de cliquer sur ces icônes directement dans les barres d'outils. Il y a également des raccoucis clavier (que je ne connais pas !)..

1. `>Project>Active configuration`.
2. Dans l'onglet General, renseigner la case Main Module. Ici mettre `ABRO`.
3. Dans l'onglet Code Gen, vérifier que le langage cible est `C ANSI`.
4. Fermer le dialogue par OK.
5. Sauver l'ensemble.

[1] Pour l'usage *académique*, on peut récupérer une version plus ancienne du compilateur Esterel (v5.92) tournant sous Linux ou sous Cygwin. Visiter le site `http://www.esterel-technologies.com`

Simulation interactive

1. `>Project>Check model`
2. Vérifier les messages dans la fenêtre de log située au bas de la fenêtre ES. Corriger éventuellement les erreurs.
3. Lancer la simulation interactive: `>Project>Simulate`.
4. La partie basse de la fenêtre ES est remplacée par deux sous-fenêtres: celle de gauche contient les entrées, celle de droite divers signaux que l'on peut choisir par un onglet. La fenêtre qui contenait le code source a également changé. Ici il s'agit du même module, mais avec des couleurs. Ce n'est plus une fenêtre d'édition, mais une fenêtre de simulation. On va y voir les exécutions avec remontée dans le code source.
5. Explorer diverses commandes. En particulier un clic droit sur le nom d'un signal dans la partie inférieure ouvre une fenêtre "pop-up". Chercher dans la notice les explications.
6. Lancer la simulation. `Tick` dans le menu de la fenêtre inférieure gauche permet de faire une réaction. On rend présent un signal en cliquant sur ce signal (d'entrée). Il devient rouge pour indiquer sa présence (bleu = absence).
7. Exécuter un scénario significatif (contenant les situations intéressantes à explorer). On peut y inclure des `Reset` qui renvoient à l'état initial. Observer les changements de couleur sur les signaux de sortie et à l'intérieur même du programme (les instructions en rouge marquent les endroits où réside le contrôle). Ceci est très utile en présence de parallélisme.
8. Enregistrer le script qui vient d'être exécuté (icône disquette). Fichier abro.esi par exemple (extension .esi à forcer).
9. Sortir de la simulation (Icône point d'exclamation barré).
10. Relancer la simulation. Rappeler le scénario enregistré : `>Simulation>Load scenario`, puis double clic sur abro.esi.
11. Rejouer le scénario en le faisant avancer en mode manuel (`>Simulation>Step forward`). On peut revenir en arrière. On peut agir sur les commandes. On peut éventuellement enregistrer le nouveau scénario.
12. Visualiser (icône Timewaves) et analyser les chronogrammes. Remarque: ce chronogramme est visualisé par GTKware, un logiciel libre.
13. Arrêter la simulation.
14. Fermer le projet.

2 Exemples simples

Après cet apprentissage à l'environnement (certaines fonctionnalités dont le test et la validation seront étudiées ultérieurement), nous allons construire des programmes élémentaires en Esterel. Le but est de faire découvrir les principales caractéristiques du langage.

Nous allons étudier les variantes d'une horloge. À chaque réception d'une impulsion (le signal pur d'entrée `pulse`), l'horloge émet alternativement `tic` et `tac`, deux signaux purs de sortie.

2.1 Simple horloge

```
// tictac0.strl
// basic tic-tac : never ending tic -- tac
// Charles Andre
// october 2006
```

```
module tictac0:
input pulse;
output tic, tac;
loop
  await pulse; emit tic;
  await pulse; emit tac
end loop
end module
```

Compiler ce programme et le simuler. Visualiser le programme source et observer les évolutions du programme. Expliquer le comportement obtenu. Expliquer pourquoi le premier `pulse` n'a apparemment pas d'effet.

2.2 Avortement

Le système précédent peut tenir compagnie, mais on risque de se lasser. Nous rajoutons un signal `hush` destiné à arrêter le tic-tac.

2.2.1 Préemption forte

Dès que le signal `hush` est présent on *doit* faire avorter le système. Le programme `tictac1.strl` est une solution.

```
// tictac1.strl
// basic tic-tac with strong abortion
// Charles Andre
// october 2006

module tictac1:
input pulse, hush;
output tic, tac;
abort
  loop
    await pulse; emit tic;
    await pulse; emit tac
  end loop
when hush
end module
```

Observer les modifications par rapport au programme initial. Compiler et simuler.

Ce programme aurait pu également s'écrire :

```
// tictac2.strl
// basic tic-tac with strong abortion
//       use of the run statement
// Charles Andre
// october 2006
// reuse of a module

module tictac0:
input pulse;
output tic, tac;
loop
  await pulse; emit tic;
  await pulse; emit tac
end loop
```

```
end module

// main
main module tictac2:
input pulse, hush;
output tic, tac;
abort
  run tictac0
when hush
end module
```

Récupérer ce programme (fichier `tictac2.strl`), le compiler et observer ses évolutions.

Vérifier (par observation des simulations) que `tictac1` et `tictac2` ont même comportement. On a tout intérêt à enregistrer un scénario et le jouer sur les deux solutions exécutées en parallèle. Cette technique n'a pas valeur de preuve. Nous verrons à la question suivante une technique de vérification formelle.

Remarque : On peut également définir le module `tictac0` dans le fichier `tictac0.strl`, définir le module `tictac2` dans un fichier `tictac2b.strl`, compiler avec édition de lien de ces deux fichiers. (Il faut dans ce cas charger les différents fichiers dans le projet). Dans la suite nous utilsons cette technique qui permet une meilleures réutilisation des modules.

2.2.2 Préemption faible

Cette fois, lors de l'occurrence de `hush`, on laisse au module préempté la possibilité d'exprimer ses "dernières volontés".

```
// tictac3.strl
// basic tic-tac with weak abortion
// Charles Andre
// october 2006
// use tictac0;strl

module tictac3:
input pulse, hush;
output tic, tac;
 weak abort
   run tictac0
 when hush
end module
```

Compiler le programme (fichier `tictac3.strl`) et observer ses évolutions. Bien noter la différence avec la préemption forte.

Le weak abort est une instruction dérivée. Le programme `tictac3b` est équivalent mais utilise un trap.

```
// tictac3b.strl
// basic tic-tac with weak abortion specifed by a trap
// Charles Andre
// october 2006
// uses tictac0.strl

module tictac3b:
input pulse, hush;
output tic, tac;
```

```
trap Purr in
    run tictac0
    ||
    await hush; exit Purr
end trap
end module
```

Compiler le programme (fichier `tictac3b.strl`) et observer ses évolutions. Visualisez les *exceptions* (Onglet Trap dans la fenêtre d'observation).

2.2.3 Preuve

Prouver à l'aide d'un des outils de vérification intégrés que tictac3 et tictac3b sont équivalents. Pour cette question, utiliser le model checker BDD et construire un observateur.

2.3 Signaux valués

On va demander à l'horloge de donner l'heure[2].

2.3.1 Affichage des secondes

Dans un premier temps on se limite aux secondes. À chaque `pulse` le compteur augmente de 1 (modulo 60). À chaque `tic` on affiche la valeur du compteur (c'est un peu curieux pour une horloge, mais c'est le choix pour cet exercice).

On utilise une variable `second : unsigned` et un signal de sortie `display_s : unsigned` (fichier `tictac4.strl`).

```
// tictac4.strl
// Charles Andre
// October 2006
// makes use of valued signals

// a M-modulo counter
module counterMod:
 input top;
 output display:unsigned;
 constant M:unsigned;
 var cmpt:unsigned := 0 in
   every top do
     cmpt := (cmpt + 1) mod M;
     emit display(cmpt)
   end every
 end var
end module

module tictac4:
 input pulse;
 output tic, tac;
 output display_s: unsigned;
 signal currentTime: unsigned in
   run tictac0
 ||
```

[2] C'est une requête tout à fait raisonnable, voire judicieuse.

```
    run counterMod [
          constant 60/M;
          signal pulse/top, currentTime/display
          ]
  ||
    every tic do
      emit display_s(?currentTime)
    end every
  end signal
end module
```

Compiler et simuler `tictac4`. Utiliser les possibilités de visualiser les variables.

- Analyser le fonctionnement du module `counter`. Bien noter qu'en ESTEREL une variable *ne peut jamais* être utilisée dans des branches parallèles.
- Noter la déclaration de signaux locaux. À quoi sert le signal local `currentTime` ?
- Visualiser les variables (Onglet `Variables` de la fenêtre d'observation) ainsi que les signaux locaux.
- Quel est l'intérêt d'instancier `counterMod` avec renommage de signaux ?
- Étudier les synchronisations entre les trois branches parallèles.