

Programmation des systèmes réactifs

Esterel (Partie 1)

ESINSA - Option GSE

27 septembre 2005

1 Environnement de programmation

Les manipulations se font sur la plate-forme Estérel Studio 5.2 (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 licence¹.

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 .../TD/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 raccourcis clavier (que je ne connais pas !).

1. >Project>Active configuration.
2. Si la version ES est la 5.0 (en ES v5.2 il n'y a plus qu'un compilateur): Dans l'onglet "Général" choisir Esterel comme Project language. Ignorer l'avertissement. Pour ce projet les programmes acceptés seront conformes à ESTEREL v5.
3. Renseigner la case Main Module. Ici mettre ABRO.
4. Dans l'onglet Code Gen, vérifier que le langage cible est C ANSI.
5. Fermer le dialogue par OK.
6. Sauver l'ensemble.

¹Pour 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.

Tracé de chronogrammes

1. Relancer la simulation.
2. >View>Dump Control... pour activer le mode collecte de chronogrammes.
3. Une nouvelle sous-fenêtre intitulée Waveform apparaît.
4. Cliquer sur le bouton Edit du fichier de configuration. Un panneau de choix s'affiche. Voir l'aide en ligne pour les paramètres. Mettre Tick duration à 2.
5. Sauver le fichier de configuration en cliquant sur la disquette. Nommer le `ABRO.cfg`, par exemple. Refermer la fenêtre de dialogue. Le nom du fichier de configuration apparaît alors dans la case correspondante.
6. Cliquer sur l'icône dossier à droite de la case Output File. Donner un nom à ce fichier qui va recevoir les traces de simulation. Par exemple `abro.vcd`. Le format vcd est commun à plusieurs logiciels de CAD circuits.
7. Activer l'enregistrement en appuyant sur Start.
8. Réaliser une simulation interactive (une vingtaine de pas).
9. Arrêter la simulation (Stop). Le scénario peut être enregistré au format esi, comme vu précédemment. Il est également disponible sous forme de chronogramme.
10. Visualiser (commande View) et analyser les chronogrammes. Remarque: ce chronogramme est visualisé par GTKware, un logiciel libre.

11. Arrêter la simulation.
12. 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 29, 1996
```

```
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 29, 1996
```

```
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 module copy (run statement)
% Charles Andre
% october 29, 1996

```

```

module tictac0:

```

```

  input pulse;
  output tic, tac;

```

```

  loop
    await pulse; emit tic;
    await pulse; emit tac
  end loop

```

```

end module

```

```

module tictac2:

```

```

  input pulse, hush;
  output tic, tac;

```

```

  abort
    run tictac0
  when hush

```

```

end module

```

Taper (ou 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 exécutés 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).

2.2.2 Prémption 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 29, 1996
% rev oct 23, 1997
```

```
module tictac0:
```

```
input pulse;
output tic, tac;
```

```
loop
  await pulse; emit tic;
  await pulse; emit tac
end loop
```

```
end module
```

```
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 sous XES. Bien noter la différence avec la préemption forte.

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

```
% tictac3.strl
% basic tic-tac with weak abortion
% Charles Andre
% october 29, 1996
% rev oct 23, 1997
```

```
module tictac0:
```

```
input pulse;
output tic, tac;
```

```
loop
  await pulse; emit tic;
  await pulse; emit tac
end loop
```

end module

module tictac3 :

input pulse , hush ;

output tic , tac ;

trap Purr **in**

run tictac0

||

await hush ; **exit** Purr

end trap

end module

Compiler le programme (fichier `tictac3b.str1`) et observer ses évolutions. Visualisez les *exceptions* (Menu Windows, item Traps).

2.2.3 Preuve

Prouver à l'aide d'un des outils de vérification intégrés que `tictac3` et `tictac3b` sont équivalents.

Pour cette question, il faut compiler le programme ESTEREL avec pour langage cible "Software circuit". Quels sont les fichiers créés par cette opération ?

2.3 Signaux valués

On va demander à l'horloge de donner l'heure².

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.

On utilise une variable `second : integer` et un signal de sortie `display_s : integer` (fichier `tictac4.str1`).

% tictac4.str1

%

% basic tic-tac with strong abortion

% use of module copy (run statement)

%

% displays the currentTime at each tic

% the currentTime is incremented by 1 at each pulse

%

% Charles Andre

% october 29, 1996

module tictac0 :

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

```

input pulse;
output tic , tac;

loop
    await pulse; emit tic;
    await pulse; emit tac
end loop

```

```

end module

```

```

% a modulo-60 counter
module counter:
    input top;
    output display: integer;

    var cmpt:=0 : integer in
        every top do
            cmpt := cmpt + 1;
            if cmpt = 60 then
                cmpt := 0
            end if;
            emit display(cmpt)
        end every
    end var
end module

```

```

module tictac4:

    input pulse;
    output tic , tac;
    output display_s: integer;

    signal currentTime: integer in
        run tictac0
        ||
        run counter [ 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.
- Visualiser l'arbre du programme (Menu Windows, item Program Tree) ainsi que le source des divers modules.
- Noter la déclaration de signaux locaux. À quoi sert le signal local currentTime?
- Visualiser les variables (Menu Windows, item Variables) ainsi que les signaux locaux (Menu Windows, item Locals).
- Quel est l'intérêt d'instancier counter avec renommage de signaux?

- Étudier les synchronisations entre les trois branches parallèles.

Exercice 1 : Réécrire ce programme de façon plus simple (en n'essayant pas de réutiliser des modules, ni de renommer des signaux).

Exercice 2 : Remplacer la variable `cmpt` par un signal dans le module `counter`. Vérifier le fonctionnement de la solution modifiée.

Exercice 3 : Réécrire ce programme pour qu'il prenne en compte une éventuelle occurrence initiale de `pulse`.

Exercice 4 : Réécrire ce programme en intégrant la préemption forte par le signal `hush`.