Validation des applications communicantes embarquées

Durée : 3h
Documents : tous documents autorisés

Veuillez respecter les notations introduites dans l'énoncé.
Les parties 1 et 2 sont indépendantes, veuillez les rédiger sur des copies séparées.
Des explications pertinentes en français seront **très** appréciées, et il est indispensable de **justifier** vos réponses.

---

1 - Modélisation transactionnelle et SystemC (10 points)

On considère une application qui utilise notamment deux micro-processeurs. Les deux processeurs n'utilisent pas de mémoire partagée, mais se serviront seulement d'un composant dédié « FIFO » qui permettra d'envoyer des données d'un processeur à l'autre. La spécification de ce composant est fournie en annexe.

Le processeur 1 va envoyer des données à la FIFO jusqu'à ce que celle-ci soit pleine. Lorsque la FIFO est pleine, le processeur se met en attente jusqu'à ce qu'une case se libère dans la FIFO.

De son côté, le processeur 2 va lire des données dans la FIFO jusqu'à ce que celle-ci soit vide, et lorsque c'est le cas, va se mettre en attente jusqu'à ce qu'une nouvelle donnée soit disponible.

Note : chaque processeur dispose d'une unique entrée interruption nommée `irq`, active sur front montant. Les autres entrées/sorties (processeur et mémoire) sont les mêmes que pour les composants vus en TP.

▷ **Question 1 (2 points) :**

En utilisant les conventions graphiques vues en cours, faire un schéma d'ensemble de la plateforme (mettre une légende pour indiquer la signification de chaque symbole). Expliquer comment intégrer le composant FIFO et comment l'utiliser, de façon concise mais précise.

Pour la question suivante, on suppose fournie une classe template FIFO implémentant une file (de taille 1024) en C++:

```
template <typename DATA_TYPE>
class FIFO {
public:
        FIFO();
        /*! Return 0 if put was successful,
          and 1 if the put failed (because of full FIFO) */
        int put(DATA_TYPE d);
        /*! Return 0 if get was successful,
          and 1 if the get failed (because of an empty FIFO) */
        int get(DATA_TYPE & d);
private: // ...
};
```

▷ **Question 2 (5 points) :**

Écrire en SystemC (fichier TLM_FIFO.h et fichier TLM_FIFO.cpp) la classe TLM_FIFO implémentant la spécification donnée en annexe.
Vous utiliserez les mêmes éléments de la bibliothèque TLM que ceux vus en cours et en TPs. On supposera l'existence de deux types `ADDRESS_TYPE` et `DATA_TYPE`, ce dernier étant codé sur 32 bits. Enfin, vous considérerez que les sorties d'interruptions sont à `false` en début de simulation (pas d'initialisation nécessaire).
Lorsqu'une transaction lève une erreur (écriture dans une file pleine, ou lecture dans une file vide), on renverra la valeur `tlm::TLM_GENERIC_ERROR_RESPONSE` à la place de `tlm::TLM_OK_RESPONSE` depuis le composant cible, et l'initiateur fera le nécessaire pour continuer la simulation.

On souhaite maintenant écrire en langage C le logiciel embarqué sur le processeur **1**. Pour simplifier, le processeur 1 va simplement envoyer les nombres 0, 1, 2, ..., 4096 au processeur 2 via le composant FIFO (i.e. on va générer les données avec une simple boucle « for »).

On suppose l'existence des fonctions suivantes :

- `void sync_wait()` : attend une interruption (pour simplifier, on suppose ici qu'il n'y a qu'une source d'interruptions)
- `void write_mem(a, d)` : écrit la donnée `d` à l'adresse `a` sur le bus auquel est relié le processeur,
- `int read_mem(a)` : lit une donnée à l'adresse `a` sur le bus auquel est relié le processeur, et la retourne.

Enfin, on supposera que la fonction `int main()` est la fonction principale. Le port esclave (cible) du composant Mailbox est à l'adresse `0x1000` sur le bus principal.

▷ **Question 3 (3 points) :**

Expliquez ce que doit faire le processeur 1 pour produire des données en boucle et les communiquer au processeur 2. Ce code doit prendre en compte le cas où la FIFO est vide (et donc l'écriture sur le composant TLM_FIFO renvoie une erreur).
Écrire le logiciel embarqué du processeur 1 : donner le contenu de la fonction principale. Vous introduirez les variables qui vous semblent nécessaires.

2 - Validation (10 points)

Exercice 1 : extraction de modèle et model-checking (6 points)

Considérons le programme suivant. L'ensemble des fonctions et procédures fournies décrit un système à états. Le programme principal permet de simuler ce système. La fonction `init` définit l'ensemble des états initiaux. La fonction `Observ` définit l'ensemble des états d'erreur. On suppose qu'il existe une fonction `random` sans paramètre qui rend un booléen aléatoire.

```
procedure Truc is
   X : Natural ;
   type State is record  A, B, La, Lb, F : Boolean ; end record ;
   S1 : State ;

   function Trans (X : Integer ; R : Boolean ; S : State)  return State is
      SS : State := S ;
   begin
      SS.lb := S.B ; SS.LA := S.A ; SS.F := False ;
      if X > 212 and R then SS.A := not S.A  ; end if ;
      if X >= 42  and not R then SS.B := not S.B  ; end if ;
      return SS ;
   end ;

   function Init  (S : State) return Boolean is
      -- si init est vrai, etat initial
   begin return  not S.A and not S.B and S.F ; end ;

   function Observ (S : State) return Boolean is
      -- si observ est faux, etat d'erreur
   begin return ((not S.F and (S.LA = S.A and S.LB /= S.B)) or  S.F) ; end ;

begin
   S1.A := False ;   S1.B := False ;   S1.F := False ;
   while (True) loop
      Put ("Current State = ") ; Put (S1) ;
      Put ("Input X = ") ; Get (X) ;
      S1 := Trans (X, Random, S1);
   end loop;
end Truc ;
```

L'état du système, tel qu'il est affiché en début de boucle, est décrit par les 5 booléens du record `S1`. Chaque tour de boucle correspond à une transition. La seule entrée explicite est un entier `X`. Le système est non déterministe à cause de l'appel à la fonction `random`. On va travailler sur ce système en appliquant le principe dit de "*predicate abstraction*" vu en cours. On choisit un ensemble de prédicats P_1, P_2, ... P_n sur `X`. Chaque configuration de valeurs des prédicats (par exemple P_1 and P_2 and ... and not P_n), définit un sous-ensemble des entiers (il y en a donc 2^n). Tous les entiers d'un ensemble défini comme cela ont le même effet sur les transitions du système.

▷ **Question 4 (3 points) :**

— Indiquer quels sont les prédicats sur `X` qu'il faut prendre en compte.
— Proposer un codage des sous-ensembles d'entiers correspondants, par des booléens (en nombre minimal).
— Combien y a-t-il d'états, et combien de transitions faut-il envisager, à partir de chaque état ?
— Donner la formule de la relation de transition qui doit lier la valeur d'un état à la valeur d'un état successeur (expliquez surtout comment elle est construite ; on ne demande pas de faire des simplifications booléennes)
— Donner la formule des états initiaux, et la formule des états d'erreur (comme on les écrit pour réaliser du model-checking symbolique).

On s'intéresse maintenant à la propriété "*si la valeur de l'entier lu est toujours dans l'intervalle* $[42, 212]$*, alors on n'atteint jamais d'état d'erreur*".

▷ **Question 5 (1 point) :**

— Est-ce une propriété de safety (sûreté) ou de liveness (vivacité) ? Comment l'hypothèse sur l'entier lu se traduit-elle avec votre codage des prédicats ? Justifiez.
— Est-ce qu'une technique de model-checking booléen peut permettre de prouver cette propriété, exactement ou de manière approchée ? Justifiez.

▷ **Question 6 (1 point) :**

— Dessiner la première étape d'un algorithme de model-checking énumératif en avant (ensemble des états initiaux, ensemble des états accessibles en une transition depuis les états initiaux).
— Cela permet-il de déterminer si la propriété définie ci-dessus est vraie ?

On s'intéresse maintenant au model-checking symbolique sur ce système.

▷ **Question 7 (1 point) :**

Exprimer la formule des états accessibles en une transition, en partant de la formule des états initiaux et de la formule de la relation de transition. Justifier.

Exercice 2 : interprétation abstraite (4 points)

Considérons le programme de la figure 1 et son graphe de contrôle (Figure 2).

```
T : array (1..4) of integer ;
b : boolean ; i: integer ;
i := 1 ;
while i <= 4 loop
   -- point XX
   T[i] := i+1 ; get (b) ;
   if b then i := i+1 ;
   else i := i+2 ; end if ;
end loop ;
```

Figure 1: Un programme impératif...

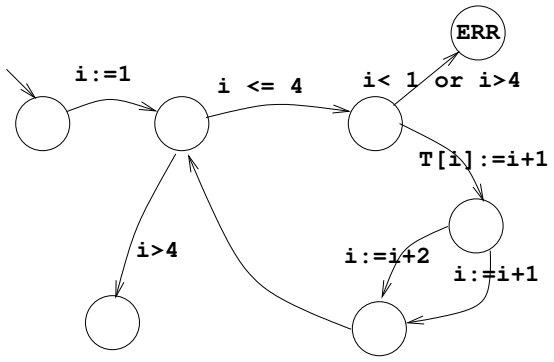

Figure 2: ...et son graphe de contrôle

L'instruction `get(b)` a été abstraite : elle donne simplement du non déterminisme dans le corps de boucle, puisqu'il est impossible de connaître la valeur lue. Le graphe correspond à ce que pourrait produire un compilateur de langage évolué comme Ada : avant chaque instruction qui accède au tableau, un test sur les bornes est réalisé, et le programme génère une exception au cas où les bornes sont violées. Ici, le modèle indique qu'on va dans un état d'erreur.

▷ **Question 8 (2.5 points) :**

— Dans les exécutions réelles du programme, quelles sont les valeurs possibles de `i` au point `XX` ?
— Y a-t-il dans ce programme un bug d'accès au tableau ?
— Appliquer une interprétation abstraite avec des intervalles à bornes entières, pour montrer que l'état d'erreur est inatteignable depuis l'état initial. Indiquer précisément les intervalles associés à chaque état, à chaque étape de l'itération.
— Expliquer pourquoi l'analyse se termine.

On change le programme, la nouvelle consition de boucle est : `while i /= 4`.

▷ **Question 9 (1.5 points) :**

Expliquer ce qui change dans l'analyse ; y a-t-il un bug d'accès au tableau dans le programme modifié, et si oui l'analyse par intervalles est-elle capable de le trouver ?

Annexe
FIFO Documentation

Le composant FIFO est un module esclave *Advanced Microcontroller Bus Architecture* (AMBA) destiné à être connecté à un bus haute-performance *Advanced High-performance Bus* (AHB).

Fonctionnalités

Le composant FIFO fournit la logique et les entrées/sorties nécessaires à la synchronisation de deux modules maîtres pour un échange unidirectionnel maître 1 vers maître 2 de données sur 32 bits.

Le composant contient une file (FIFO, pour « First In — First Out ») de taille fixe (1024 cases de 32 bits chacune). Lorsque la file est pleine, une écriture provoque une erreur, et lorsque la file est vide, une lecture provoque une erreur.

Entrées/Sorties

Le composant FIFO possède les entrées/sorties suivantes :

- Interface esclave compatible AMBA
- Sortie d'interruption `int_sender` pour le composant maître 1 (émetteur)
- Sortie d'interruption `int_receiver` pour le composant maître 2 (récepteur)

Fonctionnement interne

Initialement les deux sorties d'interruption du composant FIFO sont à 0 (ou **`false`**). Lors d'une écriture de valeur sur le registre `PUT_GET`, une interruption sur `int_receiver` est émise (passage à 1 ou **`true`**) pour signifier qu'au moins une donnée est disponible dans la FIFO. La sortie d'interruption `int_sender` est réinitialisée (à 0 ou **`false`**) et la valeur est stockée dans la FIFO.

Lors d'une lecture de valeur sur le registre `PUT_GET`, une interruption sur `int_sender` est émise (passage à 1) pour signifier qu'au moins une case libre est disponible dans la FIFO. La sortie d'interruption `int_receiver` est réinitialisée (à 0) et première valeur contenue dans la FIFO est renvoyée.

Récapitulatif des registres

Adresse relative	Type	Taille	Valeur initiale	Nom	Description
0x00	Lecture/Écriture	32 bits	0x00000000	PUT_GET	Lecture/Écriture dans la file.

Remarque sur le registre PUT_GET

Le registre PUT_GET n'est en réalité pas un registre de données ordinaire. Une écriture déclenche une écriture (action « put ») dans une file FIFO, et une lecture lit (en enlève) la première valeur de la file (action « get »).