Validation des systèmes embarqués

Telecom 3^e année

Examen — Janvier 2007

Consignes :

- Durée : 3h.
- Tous documents autorisés.
- Le barème est donné à titre indicatif.
- On attend des réponses courtes et pertinentes, inutile de recopier le cours.
- Les schémas brouillons seront pénalisés.
- Les deux parties sont indépendantes, merci de les rédiger sur des copies séparées. Pensez à lire le sujet en entier avant de commencer à répondre.

1 Méthodes de validation automatique (10 pts)

1.1 Systèmes booléens et model-checking

On considère un système réactif très simple muni de deux lampes `L1` et `L2` qui peuvent être allumées ou éteintes ; elles sont initialement éteintes. Ce système a trois entrées deux à deux exclusives (une seule peut-être vraie à chaque pas) : `inv` inverse l'état de chaque lampe ; `tt` allume tout (si une lampe est déjà allumée, cela n'a pas d'effet sur elle) ; `a1` allume la lampe 1 (si elle est déjà allumée, cela n'a pas d'effet).

Question 1 (1 point)
Dessiner le graphe complet des états accessibles depuis l'état initial.

On s'intéresse maintenant à la propriété "il n'est pas possible d'atteindre un état où la lampe 2 est allumée et la lampe 1 éteinte".

Question 2 (1 point)
(a) *Dire si c'est une propriété de sûreté (safety) ou de vivacité (liveness).*
(b) *Est-elle vraie pour le système décrit ci-dessus ? si non, donner un contre-exemple, c'est-à-dire un comportement du système qui mène à un état où la propriété est fausse.*
(c) *Si on développe les ensembles d'états accessibles "en avant" à partir de l'état initial, avec une méthode de calcul symbolique, combien d'itérations sont nécessaires pour déterminer si la propriété est vraie ou non ?*

On s'intéresse maintenant au codage de ce système de manière à réaliser des calculs de type model-checking symbolique. On considère tout d'abord un codage en relations, c'est-à-dire avec des variables `L1`, `L2`, `L1'`, `L2'`, `inv`, `a1`, `tt`.

Question 3 (1.5 points)
(a) *Donner l'ensemble des n-uplets de valeurs des variables* `L1`, `L2`, `L1'`, `L2'`, `inv`, `a1`, `tt` *qui code le système.*
(b) *Expliquer comment construire la formule booléenne globale sur ces mêmes variables, qui code tout le système.*

Le système étant déterministe, on peut envisager un codage en fonctions, pour économiser les variables `L1'`, `L2'`. Cela revient à considérer que `L1' = f1 (L1, L2, Inv, a1, tt)` et `L2' = f2 (L1, L2, Inv, a1, tt)`.

Question 4 (1.5 points)
Donner les formules booléennes qui définissent `f1` *et* `f2`*. Expliquez comment vous les construisez.*

L'état initial est `INIT` $= \overline{\texttt{L1}}.\overline{\texttt{L2}}$. On cherche à calculer la formule `ONESTEP` qui décrit l'ensemble des états accessibles en 1 pas depuis cet état initial.

Question 5 (2 points)
(a) *Expliquez comment vous construisez la formule* `ONESTEP` *à partir des formules* `f1`, `f2` *et* `INIT`*.*
(b) *En vous inspirant de la première question, donnez la formule de l'ensemble de tous les états accessibles (en un nombre quelconque de pas).*

1.2 Systèmes non booléens et interprétation abstraite

On considère ici des programmes qui manipulent une structure de données appelée Image. C'est un tableau carré de booléens, à n colonnes et n lignes. Quand une case contient 1, le point correspondant de l'image est allumé.
Notons $V = ([1, n] \times [1, n]) \longrightarrow \{0, 1\}$ l'ensemble des valeurs possibles d'une telle Image (en notant les booléens 1 pour "vrai" et 0 pour "faux"). Les opérations possibles sur une image I sont données ci-dessous. `Init_half`, `Aj` et `Ret` sont non-déterministes.

- `Init_half` : après cette opération, la moitié des points de l'image sont allumés.
- `Inv`, inversion du marquage : toute case qui contenait 1 contient maintenant 0, et inversement.
- `Aj`, ajout d'un point à l'image : une case qui contenait 0 contient maintenant 1. Si l'image était déjà complète (toutes les cases contenaient 1), cette opération n'a pas d'effet.
- `Ret`, retrait d'un point à l'image : une case qui contenait 1 contient maintenant 0. Si l'image était déjà vide (toutes les cases contenaient 0), cette opération n'a pas d'effet.
- Test de remplissage : la fonction `is_half` permet de demander si l'image I a exactement la moitié des points allumés.

Un exemple de programme qui travaille sur une telle structure de données est donné ci-dessous. On s'intéresse à propriété de sûreté suivante : "le point de programme noté `xx` est inaccessible depuis l'état initial".

```
-- programme exemple
I : image
init_half (I)
while is_half (I) loop
      Aj (I) ; Inv (I) ; Ret (I) ; Inv (I) ;
end loop ;
-- point xx
```

On propose l'abstraction suivante : on abstrait les ensembles d'états possibles d'une variable IMAGE par une information parmi les valeurs suivantes :

- `H` : Exactement la moitié des points sont allumés
- `M` : Strictement plus de la moitié des points sont allumés
- `L` : Strictement moins de la moitié des points sont allumés
- `HM` : Plus de la moitié, ou la moitié, des points sont allumés
- `HL` : Moins de la moitié, ou la moitié, des points sont allumés
- `notH` : Un nombre différent de la moitié des points sont allumés

Question 6 (3 points)

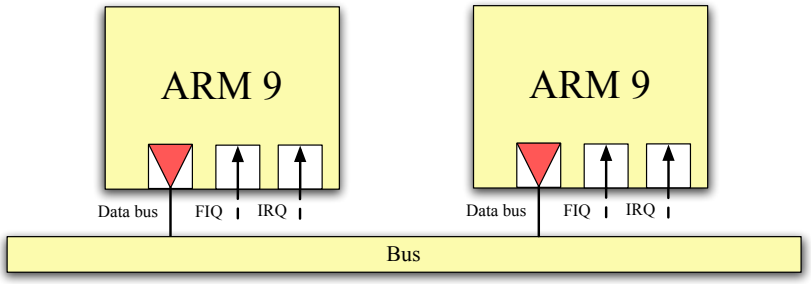
(a) *Construire l'automate interprété qui correspond au programme ci-dessus, en distinguant les transitions qui portent des conditions et les transitions qui portent des opérations sur la structure de données.*

(b) *En partant de l'état initial, associez à chaque état de l'automate une information abstraite d'après l'idée ci-dessus. Il s'agit de propager au mieux l'information abstraite sur les états de cet automate, en tenant compte des transitions qui portent des conditions, et des transitions qui portent des modifications de* `I`*. Justifiez soigneusement chaque étape, en particulier ce qui se passe quand on boucle. Que peut-on en déduire pour l'accessibilité de l'état* `xx` *?*

2 Modélisation transactionnelle en SystemC (10 pts)

2.1 Questions de cours

Question 7 (1 point) *Soit la plate-forme (incomplète) suivante :*



Comment vous y prendriez-vous pour communiquer une valeur (par exemple un entier) d'un processeur à l'autre ? Quel(s) composant(s) faudrait-t-il éventuellement ajouter ?

On a vu en cours deux manières d'exécuter du logiciel embarqué sur une plate-forme TLM : via un simulateur de jeu d'instruction (« ISS »), ou via un emballage natif (« native wrapper »).

Question 8 (1 point) *Dans quel(s) cas doit-on utiliser un compilateur croisé (c'est-à-dire un compilateur générant du code pour une plate-forme différente de celle sur laquelle il tourne) ?*

Question 9 (1 point) *Dans le cas de l'emballage natif, tous les accès au bus sont-ils modélisés ? Si non, lesquels ne le sont pas ?*

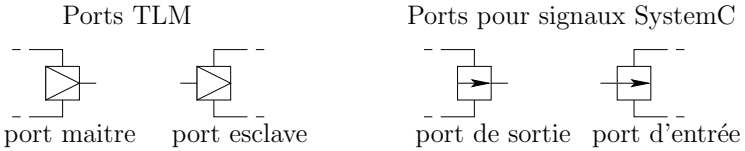
2.2 Problème : composant de compression JPEG matériel

2.2.1 Partie matérielle

On considère une plate-forme matérielle contenant :
- Un processeur
- Une RAM
- Un module de compression d'image JPEG (que nous appellerons CJPEG), dont une spécification est donnée en annexe.

Ces trois modules sont connectés à un bus (32 bits), et le module de compression est relié au processeur directement via un signal d'interruption. On modélisera ce bus par un canal TLM BASIC, avec pour simplifier comme type d'adresses et comme type de données le type `int`, qu'on suppose codé sur 32 bits.

Question 10 (1 point) *En utilisant les conventions vues en cours et rappelées partiellement ci-dessous, donnez une vue graphique de la plate-forme.*



On dispose d'une bibliothèque simple (en langage C, appelable directement depuis C++) de compression/décompression JPEG fournissant les fonctions et structures de données suivantes :

```
/* Un pixel est représenté par un ``int'', qu'on suppose codé sur 4
   octets */
typedef int pixel;

/* structure décrivant une image JPEG. Les données sont dans le
 * tableau d'octets "buffer", qui est de taille "size"
 */
struct jpeg_image {
        int size;
        unsigned char * buffer;
};

/* Compresse une image de taille "source_size_x"*"source_size_y",
 * contenue à l'adresse "source_buffer".
 * Alloue la mémoire pour le résultat, qui sera stocké dans
 * "dest_buffer"
 */
void jpeg_compress(pixel source_buffer[],
                   int source_size_x,
                   int source_size_y,
                   jpeg_image * dest_img);

/* Décompresse une image "source_img". Le résultat est de taille
 * "dest_size_x" * "dest_size_y", et est stoqué dans le tableau
 * "dest_buffer" (alloué par la fonction).
 */
void jpeg_uncompress(jpeg_image source_img,
                     pixel *dest_buffer[],
                     int * dest_size_x,
                     int * dest_size_y);
```

L'interface du composant CJPEG ainsi qu'un squelette d'implémentation sont donnés : Fichier `CJPEG.h` :

```
1  #include "systemc.h"
2  #include "basic_target_port.h"
3  #include "basic_initiator_port.h"
4  #include "basic_slave_base.h"
5  #include "common.h"
6
7  using namespace basic_protocol;
8
9  struct CJPEG : sc_module, basic_slave_base<int, int> {
10         basic_target_port<int, int> target_port;
11         basic_initiator_port<int, int> initiator_port;
12         sc_out<bool> irq;
13         SC_CTOR(CJPEG);
14         basic_status read(const int &a, int &d);
15         basic_status write(const int &a, const int &d);
16         void do_compress();
17         sc_event do_compress_event;
18         int size_x, size_y, start_addr;
```

```
19          int work_mem_start_addr, work_mem_size;
20          int res_start_addr,      res_size;
21  };
22
```

Fichier `CJPEG.cpp` :

```
1   #include "CJPEG.h"
2
3   CJPEG::CJPEG(sc_module_name name)
4     : sc_module(name),
5       target_port("master_port")
6   {
7           target_port.bind(*this);
8           SC_THREAD(do_compress);
9           sensitive << do_compress_event;
10  }
11
12  void CJPEG::do_compress() {
13          while(true) {
14                  wait(do_compress_event);
15                  pixel uncompressed[] = /* ... */;
16                  jpeg_image compressed;
17                  // ...
18                  jpeg_compress(/* ... */);
19                  // ...
20                  res_start_addr = /* ... */;
21                  res_size = /* ... */;
22                  // ...
23          }
24  }
25
26  basic_status CJPEG::write(const int &a, const int &d) {
27          switch (a) {
28          case 0x00:
29                  // ...
30                  break;
31          // ...
32          case 0x14:
33                  do_compress_event.notify();
34                  break;
35          default:
36                  printf("Invalid address 0x%08x\n", a);
37                  return ERROR;
38          }
39          return SUCCESS;
40  }
41
42  basic_status CJPEG::read(const int &a, int &d) {
43          switch (a) {
44          case 0x00:
45                  // ...
46                  break;
47          // ...
48          default:
49                  printf("Invalid address 0x%08lx\n", a);
```

```
50                  return ERROR;
51          }
52          return SUCCESS;
53  }
```

Question 11 (3.5 points) *Donnez le code des cas suivants de l'instruction* `switch` *dans le corps des fonctions* `read` *et* `write` *du composant CJPEG :*

SIZE_X *(adresse relative 0x00)*

RES_START_ADDR *(adresse relative 0x18)*

Donnez également le code de la méthode `CJPEG::do_compress()`*. On suppose que la compression d'une image prend 100 milisecondes.*
Attention : le composant devra aller chercher les données et stocker le résultat dans le composant RAM.
Pour simplifier, on supposera que le composant ne reçoit pas de nouvelle requête pendant un traitement.

2.2.2 Partie logicielle

On souhaite maintenant écrire en langage C le logiciel embarqué sur le processeur (portable sur la vraie puce et la plate-forme TLM). On suppose l'existence des fonctions vues dans le cours : `void write_mem(a, d)`, `int read_mem(a)`, `void sync()`, et `void sync_wait()`. On suppose les constantes suivantes définies :

```
// Adresse de départ pour les composants cibles
const int ram_start_addr = /* ... */;
const int cjpeg_start_addr = /* ... */

// Mémoire de travail du composant CJPEG
// dans le composant RAM.
const int work_mem_start_addr = /* ... */;
const int work_mem_size = /* ... */;
```

Question 12 (1 point) *Écrire une fonction* `init_cjpeg();` *qui positionne les registres* `WORK_MEM_START_ADDR` *et* `WORK_MEM_SIZE`*.*

Question 13 (1.5 points) *Écrire le corps de la fonction suivante :*

```
/* Meme specification que jpeg_compress() ci-dessus.
 * Cette fonction sera incluse dans le logiciel embarque, et utilise
 * le composant CJPEG comme accélérateur matériel.
 */
void jpeg_compress_hard(int source_buffer /* adresse de l'image
                                             dans le composant RAM */,
                        int source_size_x,
                        int source_size_y,
                        int & result /* adresse du résultat (jpeg)
                                        dans le composant RAM */,
                        int & size);
```

On s'autorise à utiliser le fait que le composant CJPEG est le seul à pouvoir envoyer des interruptions. On s'autorise également à renvoyer dans "result" une adresse de la mémoire de travail du composant CJPEG dans la RAM.

Annexe

CJPEG : Module de compression JPEG Documentation

Le composant CJPEG 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 CJPEG permet l'interfaçage de plusieurs périphériques générant des interruptions avec un bloc maître ne gérant qu'une seule entrée d'interruption (généralement un processeur).

Entrées/Sorties

Le composant CJPEG possède les entrées/sorties suivantes :

- Interface esclave compatible AMBA
- Interface maître compatible AMBA (pour permettre au composant d'accéder à la RAM)
- Une sortie d'interruption `int`

Les interruptions sont considérées actives sur front montant.

Utilisation du composant

Dans un premier temps, le composant doit être initialisé en spécifiant une zone de mémoire de travail avec `WORK_MEM_START_ADDR` et `WORK_MEM_SIZE`. Par la suite, le composant pourra écrire n'importe où dans cette zone mémoire.

Pour compresser une image, l'image doit être placée en mémoire, les spécifications de l'image source doivent être fournies via `SIZE_X`, `SIZE_Y` et `START_ADDR`, puis le décodage est lancé en écrivant la valeur 0x1 dans `CONTROL`.

Lorsque la compression est terminée, le signal d'interruption passe à 1. L'image décodée est alors disponible dans la mémoire à l'adresse `RES_START_ADDR`, sa taille (en octets) est `RES_SIZE`.

Récapitulatif des registres

Tous les registres sont sur 32 bits.

Légende pour le type de registre :

RW Lecture/écriture,

R Lecture seule,

W Écriture seule.

Adresse relative	Type	Valeur initiale	Nom	Description
0x00	RW	0	SIZE_X	Taille horizontale
0x04	RW	0	SIZE_Y	Taille verticale
0x08	RW	0	START_ADDR	Adresse de début de l'image à compresser en mémoire
0x0C	RW	0	WORK_MEM_START_ADDR	Adresse du début de la zone de travail du composant. Cette adresse doit être fournie par l'utilisateur, et pointer sur une zone de mémoire libre.
0x10	RW	0	WORK_MEM_SIZE	Taille de la zone de mémoire de travail (cf. WORK_MEM_START_ADDR).
0x14	W	0	CONTROL	Registre de controle : écrire 0x1 pour commencer la compression. Commencer la compression positionne le signal d'interruption à 0.
0x18	R	0	RES_START_ADDR	Adresse du résultat. Le composant CJPEG place le résultat dans sa mémoire de travail (cf. WORK_MEM_START_ADDR).
0x1C	R	0	RES_SIZE	Taille du résultat (cf. RES_START_ADDR).

Hypothèse simplificatrice

Pour simplifier, on suppose que le composant disposera toujours d'assez de mémoire (`WORK_MEM_SIZE` assez grand) pour faire le traitement.