Travail pratique #2 — INF8541 (gr. 20)

Automne 2001

Date de remise: Jeudi, 13 décembre, 16h00. Tout travail remis *après* l'heure indiquée sera considéré en retard (pénalité de 10 % par jour).
Aucun travail ne sera accepté après lundi, 17 décembre, 10h00.

1 Introduction : motivation du problème

Un domaine important, encore en plein essor, de l'informatique et du génie logiciel est celui des méthodes formelles de spécification et vérification (voir le cours MGL7160 *Méthodes formelles et semi-formelles*). Parmi les méthodes les plus prometteuses, on retrouve les approches basées sur la vérification de modèles (*model-checking*). L'intérêt d'une approche de vérification de modèles est que la vérification, une fois les spécifications appropriées développées, peut être faite de façon *automatique* :

1. Dans un premier temps, on décrit le comportement (abstrait) d'un système à l'aide d'une approche opérationnelle, par ex., algèbre de processus, systèmes de transition, automates, etc.
2. Ensuite, on décrit les *propriétés* qui devraient être satisfaites par ce système, par ex., à l'aide de diverses logiques (modale ou temporelle).
3. Finalement, à l'aide d'outils appropriés, on vérifie que le comportement décrit *satisfait* bien les propriétés désirées.

Pour effectuer de telles vérifications de façon complète et sûre, il est important de pouvoir explorer l'ensemble de tous les états possibles pouvant être atteints par le système, c'est-à-dire, de pouvoir explorer l'ensemble des comportements possibles du système.

Un petit exemple

La figure 1 présente une spécification du comportement (abstrait) d'une machine distributrice de biscuits et muffins. Cette spécification est écrite dans le langage `LOTOS` [1, 5]. Les éléments `p25, p100, ..., m25` dénotent des actions possibles de la machine. La spécification indique les séquences d'action possibles (opérateur "`;`"), les choix (opérateur "`[]`"), etc.

Une telle spécification correspond en fait à un automate fini. La figure 2 présente une description textuelle de l'automate correspondant :

- Chaque état possible de la machine est identifié par un numéro unique.
- La première ligne indique l'état initial de la machine.
- Chacune des lignes subséquentes indique une transition possible. Par exemple, la transition "`(0, "P100", 1)`" signifie que lorsque l'automate est dans l'état `0` et qu'un message "`P100`" est reçu — lorsqu'une action "`P100`" est exécutée —, alors une transition vers l'état `1` est effectuée. On dit que l'état `0` est l'état source (`source_state`) alors que `1` est l'état cible (`target_state`).

La figure 3, quant à elle, présente une version graphique du même automate.

Étant donné un description du comportement d'un automate, il est alors possible, par exemple en utilisant l'outil CADP [2], de vérifier que la machine distributrice ainsi spécifiée satisfait diverses propriétés. Par exemple, les propriétés suivantes sont satisfaites :

- `["P100"](["P100"] false and ["P25"] false)` : après avoir déposé une pièce de 1.00 $, il n'est plus possible d'insérer une nouvelle pièce.
- `<"P25"> true and <"P100"> true` : au départ, on peut mettre aussi bien une pièce de 0.25 $ qu'une pièce de 1.00 $.

```
SPECIFICATION distributrice[p25,p100,b_bisc,b_muff,bisc,muff,m25] : NOEXIT

LIBRARY TYPES ENDLIB

BEHAVIOUR
  Distributrice[p25,p100,b_bisc,b_muff,bisc,muff,m25]
WHERE
PROCESS Distributrice
        [p25, p100, b_bisc, b_muff, bisc, muff, m25]: NOEXIT :=
  p100;
    (b_muff;
       muff;
       Distributrice[p25, p100, b_bisc, b_muff, bisc, muff, m25]
     □
     b_bisc;
       bisc;
       m25;
       Distributrice[p25, p100, b_bisc, b_muff, bisc, muff, m25]
    )

  □

  p25;
    p25;
    p25;
    (b_bisc;
       bisc;
       Distributrice[p25, p100, b_bisc, b_muff, bisc, muff, m25]
     □
     p25;
       b_muff;
       muff;
       Distributrice[p25, p100, b_bisc, b_muff, bisc, muff, m25]
    )
ENDPROC

ENDSPEC
```

Figure 1: Une spécification LOTOS d'une machine distributrice

```
0
(0, "P100", 1)
(0, "P25", 2)
(1, "B_MUFF", 3)
(1, "B_BISC", 4)
(2, "P25", 5)
(3, "MUFF", 0)
(4, "BISC", 6)
(5, "P25", 7)
(6, "M25", 0)
(7, "B_BISC", 8)
(7, "P25", 9)
(8, "BISC", 0)
(9, "B_MUFF", 10)
(10, "MUFF", 0)
```

Figure 2: L'automate associé à la machine distributrice : version textuelle

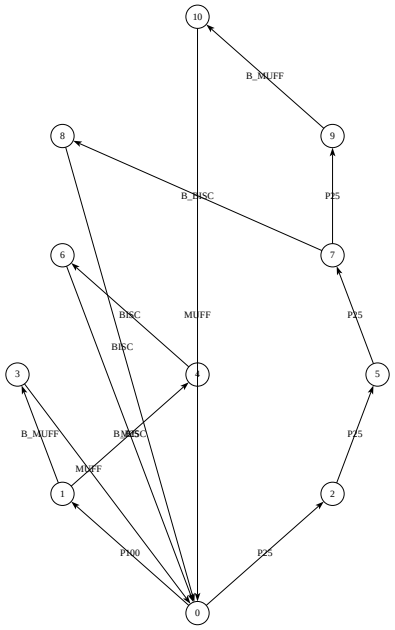

Figure 3: L'automate associé à la machine distributrice : version graphique

Des propriétés plus complexes peuvent évidemment être exprimées, mais elles sont alors plus difficiles à comprendre, à moins d'avoir suivi le cours MGL7160 ...

La principale difficulté de la vérification automatique de modèles est liée à ce qu'on appelle le problème de *l'explosion des états* (*state explosion*) : dès que le comportement du système devient relativement complexe, le nombre d'états de l'automate associé devient extrêmement élevé et croît *très* rapidement. Sur une machine uni-processeur, de nombreux systèmes ne peuvent donc pas être analysés à l'aide de cette approche, à cause du manque d'espace mémoire. Une solution possible est alors d'utiliser des multi-ordinateurs avec mémoires distribuées, de façon à augmenter l'espace mémoire disponible, donc la taille de l'espace des états pouvant être traité par le vérificateur de modèles.

2 Le problème à résoudre

Pour ce travail, à réaliser en Threaded-C sur la machine Earthquake, vous devez mettre en oeuvre un programme qui permettra, à partir d'une description *implicite* de la fonction de transition d'un automate, de distribuer l'ensemble des états et transitions sur les différents noeuds de la machine. Rappelons que la fonction de transition d'un automate permet, à partir d'un état source et d'une action (étiquette), de déterminer l'état destination (cible). Un fichier `automata.h`, et sa mise en oeuvre `automata.c`, définiront la fonction de transition que vous devrez utiliser pour explorer et générer l'automate et son graphe d'états. Plus précisément, la fonction de transition ainsi définie permet, à partir d'un état source, d'obtenir un itérateur (type `Transition_iterator`) permettant ensuite d'obtenir, de façon incrémentale et itérative, chacune des transitions sortant de l'état source, chaque transition étant elle-même définie par un état source (type `State`), une étiquette dénotant l'action conduisant à la transition (type `Label`), et un état destination (type `State`). Une autre fonction (`start_state()`) permet aussi d'obtenir l'état initial.

Par exemple, et de façon plus spécifique, étant donné la spécification `distributrice.lotos` présentée

```
#ifndef AUTOMATA_INCLUDED
#define AUTOMATA_INCLUDED

/* Basic auxiliary types. */
typedef int   State;
typedef char *Label;
typedef int   Bool;
typedef void *Element;

/* The operations in the libcii library require elements to be non 0. */
#define State_to_Element(s) ((void *)(s+1))
#define Element_to_State(s) ((((int)s)-1))

/* Creation of the automata. */
void build_automata();

/* Start state */
State start_state();

/* Transition. */
typedef struct Transition *Transition;

Transition mk_transition( State source_state, Label label, State target_state );
State      source_state ( Transition t );
Label      label        ( Transition t );
State      target_state ( Transition t );

/* Transition_iterator */
typedef struct Transition_iterator *Transition_iterator;

Transition_iterator Transition_iterator_new( State state );
Bool                has_more_transitions   ( Transition_iterator ti );
Transition          next_transition        ( Transition_iterator ti );

#endif
```

Figure 4: Interface du fichier `automata.h`

plus haut, un programme `distributrice.c` sera produit et vous sera fourni (je m'occuperai de vous fournir un certain nombre de tels programmes de tests, pour diverses spécifications). Ce programme, incorporé dans le fichier `automata.c` (par l'intermédiaire du fichier auxiliaire `build-automata.c` — il ne contient qu'une ligne de la forme "`#include "distributrice.c"`") que je vous fournirai aussi, définira un module réalisant l'interface présentée à la figure 4.

2.1 L'algorithme de distribution des transitions

Essentiellement, pour ce travail, il s'agit d'explorer et de générer l'ensemble des états et transitions à partir de l'état initial et de la fonction de transition définie par les opérations du fichier `automata.h`. Dans le cadre de ce travail, il suffira que chaque processeur imprime sur `stdout` (donc dans `etc.$$.i`) chacune des transitions qui lui est associée, en utilisant la même notation que celle de la figure 2. Notez toutefois que seul le processeur `0`, qui *initie* la distribution des états et des transitions, doit imprimer l'état initial (l'état initial est unique pour l'ensemble de l'automate).

L'idée générale du processus de distribution des états et transitions est la suivante : une fonction de dispersion associe à chaque état (nombre entier non-négatif) un numéro de processeur ; une transition "`(s1, e, s2)`" est alors traitée (ici, simplement imprimée sur `stdout`) par le processeur chargé de traiter l'état `s2`. La logique générale du traitement (en ignorant le problème de la terminaison) est présentée à la figure 5, en utilisant la notation d'Andrews et les opérations exportées par le fichier `automata.h` (figure 4). Notez que seul le pseudo-code pour le processeur `0` est indiqué ; les `N-1` autres processeurs exécutent le même code, sauf pour la partie qui traite l'état initial.

La distribution des états se fera en utilisant diverses fonctions de dispersion (spécifiées, dans le module `hash-state`, par `HASH1`, `HASH2` ou `HASH3` et la fonction `init_hash()` et utilisées par l'intermédiaire des fonctions `hash_state()`). C'est vous qui devrez définir diverses fonctions de distribution et voir l'impact du choix de la fonction sur l'efficacité du programme résultant. Notez que l'on suppose que la fonction `hash_state()` retourne un entier non-négatif inférieur à la valeur spécifiée lors de l'appel à `init_hash()`. L'interface des opérations associées est présentée à la figure 6. Ce fichier ne doit pas être modifié; toutefois, vous devez modifier et compléter le fichier `hash-state.c` (fonctions `hash1`, `hash2`, `hash3`).

2.2 Détection de la terminaison

La partie difficile du problème de distribution du graphe d'états est celle consistant à déterminer à quel moment tous les états ont été visités. À tout moment après qu'un processeur a terminé d'explorer l'ensemble des transitions disponibles sur son canal, il est possible qu'il reçoive d'autres messages lui indiquant de nouvelles transitions à explorer. En d'autres mots, l'ensemble du travail est terminé uniquement lorsque *tous* les processus ont terminé *et* lorsqu'il n'y a plus aucun messages en transit dans les canaux. Diverses façons de détecter ces conditions seront abordées en classe (problème de la détection distribuée de la terminaison).

2.3 Tamponnage (*buffering*)

Un autre élément clé pour assurer l'efficacité de votre solution est l'utilisation de *tampons* (*buffers*) de façon à réduire les communications. En d'autres mots, dans votre version Threaded-C, il faut éviter que la transmission de chacune des transmissions donne lieu à une communication inter-processeur. Il faut plutôt faire en sorte d'accumuler un certain nombre d'items (transitions) et, lorsque suffisamment d'items sont prêts à être transmis, alors une véritable communication peut être effectuée. Évidemment, il faut faire attention d'éviter les blocages qui pourraient survenir : si un processus s'apprête à terminer parce qu'il n'y a plus rien sur son canal d'entrée, il doit alors s'assurer de vider ses propres canaux de sortie, c'est-à-dire, il doit vider ses tampons. Évidemment, je vous suggère d'utiliser une approche incrémentale de développement de votre solution : faite fonctionner le tout avec des tampons de taille 1 (donc sans véritable tamponnage ;), puis, lorsque tout semble fonctionner, introduisez l'optimisation du tamponnage.

```
chan canal[N](Transition);

process Distributeur[0] {
  Set_T visited;

  init_hash( HASH_1, NUM_NODES );
  build_automata();
  visited = Set_new( TAILLE_INITIALE, NULL, NULL );

  # Debut de la partie specifique au processus 0.
  State s0 = start_state();
  printf( "%d\n", s0 );
  Set_put( visited, State_to_Element(s0) );
  Transition_iterator ti = Transition_iterator_new(s0);
  while( has_more_transitions(ti) ) {
    Transition t = next_transition(ti);
    send canal[hash_state(target_state(t))](t);
  }
  FREE(ti);
  # Fin de la partie specifique au processus 0.

  while( TRUE ) {
    Transition t0;

    receive canal[0](t0);
    if ( !Set_member(visited, target_state(t0)) ) {
      # Premiere visite de l'etat cible de la transition t0.
      printd( "(%d, \"%s\", %d)\n", source_state(t0), label(s0), target_state(t0) );
      Set_put( visited, State_to_Element(target_state(t0)) );
      Transition_iterator ti = Transition_iterator_new(target_state(t0));
      while( has_more_transitions(ti) ) {
         Transition t = next_transition(ti);
         send canal[hash_state(target_state(t))](t);
      }
      FREE(ti);
    }
}

process Distributeur[i = 1 to N] {
  # Semblable a Distributeur[0], mais sans traitement de l'etat initial.
  ...
}
```

Figure 5: Algorithme de distribution des transitions (notation d'Andrews)

```
#ifndef HASH_INCLUDED
#define HASH_INCLUDED

typedef int State;

typedef enum {HASH_1, HASH_2, HASH_3} Hash_function;

void init_hash( Hash_function h, int max_hash );
/*
 * requires:
 *   1 <= max_hash
 */

int  hash_state( State s );
/*
 * ensures:
 *   0 <= hash(s) < max_hash
 */


#endif
```

Figure 6: Interface pour les fonctions de dispersion (`hash-state.h`)

3 Les éléments que vous devrez utiliser

Pour développer votre solution au TP, vous *devrez* utiliser les éléments suivants, qui sont disponibles dans mon compte Earthquake (`~tremblay/tp2`) :

- Les fichiers `automata.h` et `automata.c`, qui permettent de définir la fonction de transition des automates.
- Divers fichiers de tests, par ex., `distributrice.c`, ..., etc. Chacun de ces fichiers, qui définira la fonction `build_automata()`, sera inclus dans le fichier `automata.c` par l'intermédiaire du fichier `build-automata.c`. Un script `mk-build-automata` permettra d'automatiser cette tâche et de compiler le programme résultant. Ce script nécessite deux arguments :
  1. Un fichier (de test) mettant en oeuvre la fonction `build-automata()` (je vous fournirai de tels fichiers de test).
  2. Un fichier utilisant `automata.h` pour parcourir l'automate généré par `build-automata()`. C'est ce fichier que vous devez définir et qui devra être écrit en Threaded-C.
- Les modules de la bibliothèque `cii` : C. Hanson [3] a défini et mis en oeuvre une bibliothèque contenant diverses structures de données (par ex., ensembles, listes, tables, etc.) qui seront grandement utiles pour ce travail. En fait, comme vous pourrez le constater en regardant le code des fichiers `automata.[hc]` et des fichiers de test (par ex., `distributrice.c`), les fonctions de transition sont définies à l'aide des types et opérations de cette librairie. J'expliquerai en classe certains de ces modules (dont les fichiers `*.h` sont dans le répertoire `tp2/cii`).
- Les fichiers `hash-state.[hc]` qui définissent, de façon partielle, diverses fonctions (`HASH{1,2,3}`) de dispersion sur les états, permettant de distribuer les transitions entre les différents processeurs, fonctions que vous aurez à compléter (fichier `hash-state.c`).

4 Preuves de bon fonctionnement

Pour valider votre module, vous devrez utiliser les jeux d'essai que je mettrai à votre disposition, sous forme de programmes définissant la procédure `build_automata()`, par ex., `distributrice.c`. Je vous indiquerai plus en détails, ultérieurement, comment vérifier que les bons automates sont générés.

5 Ce que vous devrez remettre

1. Une copie papier du code de votre programme, de même que les principaux résultats d'exécution montrant le bon fonctionnement du module.
2. Un fichier, transmis par courriel, contenant votre programme (y compris le fichier définissant les fonctions de dispersion).
3. Un rapport présentant les résultats de diverses expériences visant à déterminer l'impact du choix de la fonction de dispersion, de même que de l'effet du tamponnage, sur les performances du programme.
4. Un bref rapport écrit (manuel *technique*) expliquant les grandes lignes de la mise en oeuvre de votre programme : comment le programme fonctionne-t-il? quelles sont les limites et contraintes? est-ce que tous les jeux d'essai fonctionnent correctement? quelles difficultés avez-vous rencontrées? etc.

A Quelques fichiers de la librairie `cii` de Hanson

La librairie de Hanson définit de nombreuses structures de données : ensembles, listes, séquences, tables, etc. Les interfaces de trois de ces modules sont présentées plus bas. Un point à souligner est que ces modules sont définis de façon générique à l'aide de l'approche C classique, c'est-à-dire, en manipulant des pointeur de type "`void *`". Lorsqu'un tel type est utilisé, on peut aussi transmettre des entiers `int`, comme cela est fait pour les états de l'automate. Toutefois, une des contraintes de l'utilisation de pointeur "`void *`" est que cela implique est qu'un élément manipulé par une collection de la librairie ne peut pas être un pointeur `NULL` (de valeur 0). Or, dans le cas des états, il est possible qu'un état soit nul (état initial). C'est pour cette raison que j'ai introduit (Fig. 4) les macros `State_to_Element` et `Element_to_State`, qui assurent qu'un état est converti de façon unique en un élément pouvant être manipulé par les collections de Hanson.

A.1 `set.h`

La figure 7 présente l'interface pour le module `set` (ensembles).

A.2 `list.h`

La figure 8 présente l'interface pour le module `list` (listes linéaires simples).

A.3 `table.h`

La figure 9 présente l'interface pour le module `table` (tables, dictionnaires).

Références

[1] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[2] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In *ICSE '92*, pages 246–259, 1992.

[3] D.R. Hanson. *C Interfaces and Implementations — Techniques for Creating Reusable Software.* Addison-Wesley Professionnal Computing Series. Addison-Wesley, Reading, MA, 1997.

[4] D.B. Skillicorn and D. Talia. *Programming Languages for Parallel Processing.* IEEE Computer Society Press, 1995.

[5] K.J. Turner. *Using formal description techniques: an introduction to Estelle, LOTOS, and SDL.* J. Wiley & Sons, 1993. [En réserve: QA76.6U85].

```
#ifndef SET_INCLUDED
#define SET_INCLUDED
#define T Set_T
typedef struct T *T;

extern T      Set_new    (int hint,
                          int cmp(const void *x, const void *y),
                          unsigned hash(const void *x));
extern void   Set_free   (T *set);
extern int    Set_length (T set);
extern int    Set_member (T set, const void *member);
extern void   Set_put    (T set, const void *member);
extern void  *Set_remove (T set, const void *member);
extern void   Set_map    (T set,
                          void apply(const void *member, void *cl),
                          void *cl);
extern void **Set_toArray(T set, void *end);
extern T      Set_union  (T s, T t);
extern T      Set_inter  (T s, T t);
extern T      Set_minus  (T s, T t);
extern T      Set_diff   (T s, T t);

#undef T
#endif
```

Figure 7: Fichier d'interface pour les ensembles `set.h`

```
#ifndef LIST_INCLUDED
#define LIST_INCLUDED
#define T List_T
typedef struct T *T;
struct T {
    T rest;
    void *first;
};

extern T      List_append (T list, T tail);
extern T      List_copy   (T list);
extern T      List_list   (void *x, ...);
extern T      List_pop    (T list, void **x);
extern T      List_push   (T list, void *x);
extern T      List_reverse(T list);
extern int    List_length (T list);
extern void   List_free   (T *list);
extern void   List_map    (T list,
                           void apply(void **x, void *cl),
                           void *cl);
extern void **List_toArray(T list, void *end);

#undef T
#endif
```

Figure 8: Fichier d'interface pour les listes `list.h`

```
#ifndef TABLE_INCLUDED
#define TABLE_INCLUDED
#define T Table_T
typedef struct T *T;
extern T     Table_new     (int hint,
                            int cmp(const void *x, const void *y),
                            unsigned hash(const void *key));
extern void  Table_free    (T *table);
extern int   Table_length  (T table);
extern void *Table_put     (T table, const void *key, void *value);
extern void *Table_get     (T table, const void *key);
extern void *Table_remove  (T table, const void *key);
extern void  Table_map     (T table,
                            void apply(const void *key, void **value, void *cl),
                            void *cl);
extern void **Table_toArray(T table, void *end);

#undef T
#endif
```

Figure 9: Fichier d'interface pour les tables (dictionnaires) `table.h`