Formale Grundlagen der Informatik III: Lab 2 (CBMC)

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2014/2015
Deadline zur Abgabe: Dienstag, der 13.01.2015 - 23:59 Uhr

Im Rahmen dieses Labs müssen Sie Code und Assertions in der Sprache **C** und **PROMELA** schreiben und zum Testen **CBMC** bzw. **SPIN** verwenden. Sollten Sie CBMC noch nicht installiert haben, beachten Sie bitte die Anleitung, die in Moodle unter dem Punkt **0 - Organisation** zu finden ist.
Zur Abgabe laden Sie bitte alle erstellten Codes gesammelt in einem .zip Archiv vor der angegebenen Deadline über das Moodle Portal der Veranstaltung hoch.

Über alle **3** Labs können sie eine maximale Menge von **100** Punkten erlangen. Durch das vorliegende Lab können Sie **35** Punkte erzielen.

Informationen zur Abgabe: Bitte benennen Sie die Dateien ihrer Abgabe sinnvoll, damit diese problemlos den jeweiligen Aufgaben zugeordnet werden können. Geben Sie bitte in den Kopf jeder .c/.pml Datei (in Form eines Kommentars) und jeder Textabgabe die Mitglieder Ihrer Lab-Gruppe sowie die Gruppennummer an. Für Textabgaben verwenden Sie bitte ein reines Textformat (.txt).
Fassen Sie abschließend alle Daten ihrer Abgabe in einem .zip Archiv zusammen und bennenen Sie dieses nach folgender Konvention: lab2-l**xx**.zip, wobei **xx** durch Ihre jeweilige Gruppennummer zu ersetzen ist.

Hinweis: *Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Weiter Informationen finden Sie unter: https://www.informatik.tu-darmstadt.de/de/sonstiges/plagiarismus/*

Aufgabe 1 Modellierung (PROMELA) - Brücken Puzzle (10 P.)

Adam, Bianca, Christine und Dave befinden sich in einer dunklen Nacht auf einer Seite einer langen und schmalen Brücke. Sie wollen alle die Brücke überqueren, sind jedoch nur im Besitz einer einzigen Taschenlampe. Es können so nur maximal zwei Personen zur gleichen Zeit mit der Taschenlampe die Brücke überqueren. Ohne die Taschenlampe ist eine Überquerung aus Gründen der Sicherheit nicht möglich! Jede Person läuft mit einer anderen Geschwindigkeit. Wenn zwei Personen zusammen laufen, so wird immer die Geschwindigkeit der langsameren Person gewählt. Adam ist am schnellsten und benötigt zur Überquerung lediglich eine Minute. Bianca benötigt zwei Minuten, Christine 5 Minuten und Dave sogar 10 Minuten.

a) Schreiben Sie ein PROMELA Modell für das beschriebene Szenario. Es soll nicht-deterministisch gewählt werden, wer als nächstes die Brücke überquert und die Gesamtzeit soll entsprechend hochgezählt werden. Sie können die Zeit als ein byte modellieren.

b) Finden Sie den schnellsten Weg alle vier Personen auf die andere Seite der Brücke zu bringen. Identifizieren Sie den schnellsten Weg indem Sie LTL Eigenschaften formulieren. Beschreiben Sie Ihren verwendeten Ansatz kurz.

c) Nachdem Adam, Bianca, Christine und Dave Ihre Erledigungen auf der anderen Seite der Brücke erledigt haben müssen Sie die Brücke erneut überqueren. Kurz vor der Ankunft an der Brücke hat sich Bianca ihren Fuß vertreten und benötigt nun 3 Minuten zur Überquerung der Brücke. Modifizieren Sie Ihr Modell entsprechend und formulieren Sie neue LTL Eigenschaften um den schnellsten Weg zu identifizieren. Vergleichen Sie die zwei Lösungen. Sind diese gleich oder unterschiedlich. Beschreiben Sie Ihre Erkenntnisse kurz.

Wichtige Hinweise:

- Ihr Modell darf keinerlei Hinweise auf die Lösungen geben!
- Akzeptieren Sie nicht direkt jedes Ergebnis, welches Sie von SPIN geliefert bekommen. Es kann sein, dass Sie eine Kleinigkeit vergessen haben und daher ein falsches Ergebnis erhalten. Prüfen Sie daher Ihre Antwort auf Plausibilität.

Aufgabe 2 Sortieralgorithmen (15 P.)

In der aktuellen Aufgabe wollen wir uns mit Model-Checking im Bereich der Sortierverfahren befassen.

```
void sort(int *array, size) {
   int i,j;
   for(i=1; i<=size; i++) {
      for( j=i; array[j-1] > array[i] && j > 0; j-- )
         array[j] = array[j-1];
      array[j]=array[i];}
}
#define SIZE 10
void test () {
  int array[SIZE] = {5, 9, 1, 4, 13, 99, 2, 1, 7, 6};
  int size = sizeof(array) / sizeof(double);
  sort (array, size);
}
```

a) Bestimmen Sie zu obigem Sortieralgorithmus die Komplexität für den worst, best und average case. Nutzen Sie zur Angabe der Komplexität die O-Notation.

b) Schreiben Sie eine oder mehrere Assertions für CBMC um die korrekte Funktionsweise des Sortierverfahrens nachzuweisen (das Sortierverfahren arbeitet korrekt, wenn alle Elemente aufsteigend sortiert sind).
Sollten Sie während der Verifikation Fehler in obigem Algorithmus entdecken, so korrigieren Sie diesen entsprechend. Geben Sie bitte bei Änderungen jeweils einen kurzen Kommentar in Ihrem Quellcode an.

c) Wir wollen nun einen effizienteren Algorithmus für Sortierungen implementieren und verifizieren. Hierfür wählen wir **Heapsort**, welches Ihnen vermutlich aus GdI 2 noch ein Begriff sein sollte.
Die Funktionsweise ist in dem folgenden Pseudocode noch einmal kurz abgebildet:

```
function HEAPSORT(a, count)
    HEAPIFY(a, count)
    end ← count - 1
    while end > 0 do
        SWAP(a[end], a[0])
        end ← end - 1
        SIFTDOWN(a, 0, end)
    end while
end function
function HEAPIFY(a, count)
    start ← floor ((count - 2) / 2)
    while start ≥ 0 do
        SIFTDOWN(a, start, count - 1)
        start ← start - 1
    end while
end function
function SIFTDOWN(a, start, end)
    root ← start
    while root * 2 + 1 ≤ end do
        child ← root * 2 + 1
        swap ← root
        if a[swap] < a[child] then
            swap ← child
        end if
        if child+1 ≤ end ∧ a[swap] < a[child+1] then
            swap ← child + 1
        end if
        if swap = root then return
```

```
        else
            SWAP(a[root], a[swap])
            root ← swap
        end if
    end while
end function
```

1. Schreiben Sie zuerst eine Funktion *swap*, welche zwei Elemente eines Arrays vertauscht. Verifizieren Sie anschließend die korrekte Funktionsweise dieser Funktion durch CBMC.
2. Schreiben Sie anschließend die Funktionen *siftdown* und *heapify*. Verifizieren Sie auch deren korrekte Funktionsweise.
3. Komplettieren Sie nun Ihre Implementierung durch die Funktion heapsort. Verifizieren Sie anschließend mit Ihren bereits in b) geschriebenen Assertions die korrekte Sortierung Ihrer Implementierung.
   Prüfen Sie auch mittels CBMC Ihr System auf Safety Eigenschaften und korrigieren Sie Fehler, falls in Ihrem Code Fehler auftraten.

d) Erweitern Sie die beiden vorherigen Algorithmen so, dass diese statt einfacher Zahlen auf den folgenden Datentypen arbeiten:

```
struct kvpair{
  int key;
  char value[8];
};
```

Passen Sie auch Ihre Assertions entsprechend an.

e) Schreiben Sie nun eine Eigenschaft um die Stabilität der zwei Algorithmen zu verifizieren. Prüfen Sie anschließend mittels CBMC Ihre Implementierungen auf Stabilität.
Stabilität: Im Bereich von Sortierverfahren bedeutet Stabilität, dass beim Vorkommen von mehreren Datensätzen mit dem gleichen Sortierschlüssel deren Reihenfolge vor und nach der Sortierung gleich bleibt.

f) Wir haben nun zwei Eigenschaften für Sortieralgorithmen formuliert. Betrachten Sie nun den folgenden weiteren Algorithmus und versuchen Sie diesen auf ihre formulierten Eigenschaften zu verifizieren. Implementieren Sie den Pseudocode hierfür zuerst in C.

```
function MYSTICSORT(array L, i = 0, j = length(L)-1)
    if L[j] < L[i] then
        L[i] ↔ L[j]
    end if
    if (j - i + 1) > 2 then
        t ← (j - i + 1) / 3
        MYSTICSORT(L, i , j-t)
        MYSTICSORT(L, i+t, j )
        MYSTICSORT(L, i , j-t)
    end if
    return L
end function
```

g) Vergleichen Sie die Komplexität der drei Algorithmen, die in dieser Aufgabe zum Einsatz kamen und stellen Sie diese gegenüber (beim Algorithmus *mysticsort* reicht eine grobe Schätzung aus). Geben Sie auch Ihre Beobachtungen zur Stabilität an.

Aufgabe 3 Hamilton Graph (10 P.)

In der Vorlesung wurden bereits kurz Hamilton Pfade eingeführt. In der folgenden Aufgabe wollen wir CBMC nutzen um das Hamilton Pfad Problem für einen vorgegebenen Graphen zu lösen. Schreiben Sie daher ein C Programm bzw. eine Funktion, welche/s:

a) die Repräsentation eines Graphen in Form eines zweidimensionalen Arrays, welches die Transitionen kodiert, sowie die Zahl der Knoten übergeben bekommt

b) einen Pfad durch den Graphen rät

c) und prüft ob der Pfad ein Hamilton Pfad ist

Wichtige Hinweise:

- Ihr Modell soll Graphen von beliebiger Größe lösen können!
- Lösen Sie die Aufgabe so, dass CBMC einen Assertion error meldet, wenn ein Hamilton Pfad existiert.

Anbei ein exemplarisches Transitionenarray, welches Sie zum Testen verwenden können:

```
int graph [5][5] =
{
{0, 1, 0, 0, 1},
{1, 0, 1, 1, 0},
{0, 1, 0, 1, 0},
{0, 1, 0, 0, 1},
{1, 0, 0, 1, 0}
} ;
```