Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Florian Franzmann Martin Hoffmann Tobias Klaus

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
http://www4.cs.fau.de

17. Juni 2014

Überblick

1 ACSL

2 Aufgabenstellung

ANSI/ISO C Specification Language (ACSL)

- Beschreibungssprache für C-Programme
- Annotationen für C-Funktionen
- „Design by Contract“
- Vor-/Nachbedingungen, Invarianten, Varianten
  $\leadsto$ wie in AuD ☺
- verifizierbar mit `frama-c`

Sprachelemente

- Annotationen als Kommentare im Quellcode
- `/*@ ... */`
- Aussagenlogik wie in C (`&& || !`)
- außerdem Implikation `==>`, Äquivalenz `<==>`, ...
- Quantorenlogik `\forall`, `\exists`

```
\forall integer k; 0 <= k < n ==> a[k] == k * k;
```

- mathematische Typen

  `integer` ganze Zahlen $\rightsquigarrow \mathbb{Z}$
  `real` reelle Zahlen $\rightsquigarrow \mathbb{R}$
  `boolean` `\true` und `\false`

- C-Datentypen

Wahrheitswerte

- Wahrheitswerte in C: `int` $\begin{cases} \text{falsch} & \text{wenn } 0 \\ \text{wahr} & \text{sonst} \end{cases}$
- das muß in ACSL dann auch entsprechend geprüft werden
  - Gesamter Wertebereich muss erfasst werden
  - ⤳ `x == 0` bzw. `x != 0`, nicht etwa `x == 1`
- Alternative: `enum` verwenden und auf `x == true` bzw. `x == false` testen

bool.h

```
1 typedef enum _bool {
2   false = 0,
3    true = 1
4 } bool;
```

Mathematische Funktionen

Mathematische Funktionen

```
1  integer \min(integer x, integer y);
2  integer \max(integer x, integer y);
3
4  real \min(real x, real y);
5  real \max(real x, real y);
6
7  real \pow(real x, real y);
8  real \sqrt(real x);
9
10 real \sin(real x);
11 real \cos(real x);
```

und noch viele andere $\rightsquigarrow$ Frama-C-ACSL-Handbuch

für die Übungsaufgabe nicht relevant

Bestandteile von Verträgen

`requires` Vorbedingung, muß *beim Funktionsaufruf* erfüllt sein

`assigns` Beschränkung von Schreibzugriffen
wenn nicht vorhanden
$\Rightarrow$ *keine Einschränkung*
$\leadsto$ sollte immer angegeben werden!

`ensures` Nachbedingung
$\leadsto$ muß *nach Rückkehr* der Funktion erfüllt sein

`behavior` Fallunterscheidung

`\result` Rückgabewert der Funktion

Verträge

Schnittstelleneigenschaften

```
1  /*@
2     requires x >= 0;        // Vorbedingung
3     assigns \nothing;       // Funktion weisst nichts zu
4     ensures \result >= 0; // Nachbedingung
5     ensures \result * \result <= x;
6     ensures x < (\result + 1)
7                 * (\result + 1);
8   */
9  int32_t sqrt(int32_t x);
```

```
1
2  /*@
3     requires \valid(p);           // p muss gültiger Zeiger sein
4     assigns *p;                   // Wert von p wird zugewiesen
5     ensures *p == \old(*p) + 1; // old greift auf Wert vor Aufruf zu
6   */
7   void incstar(int *p);
```

Verhalten

Fallunterscheidungen

```
/*@ ...
   behavior p_changed:
     assumes n > 0;      // wird bei n > 0 aktiv
     requires \valid(p);
     assigns *p;
     ensures *p == n;
   behavior q_changed:
     assumes n <= 0;     // wird bei n ≤ 0 aktiv
     requires \valid(q);
     assigns *q;
     ensures *q == n;
 */
void f(int n, int *p, int *q) {
  if (n > 0) { *p = n; }
  else       { *q = n; }
}
```

Vollständigkeit von Verhalten I

complete

```
1  /*@
2     requires R;
3     behavior b1:
4       assumes A1;
5       ...
6     behavior bn:
7       assumes An;
8       ...
9     ....
10    complete behaviors b1, ..., bn;
11  */
```

- Dann gilt: `R ==> (A1 || ... || An)`

⇝ mindestens ein Verhalten trifft zu

- Kurzform: `complete behaviors;`

Vollständigkeit von Verhalten II

disjoint

```
/*@
   disjoint behaviors b1, ..., bn;
 */
```

- Dann gilt: `R ==> !(A1 && ... && An)`

⇝ nicht alle Verhalten gleichzeitig, mindestens zwei disjunkte

- Kurzform: `disjoint behaviors;`

Prädikate – Kapselung von Logik

- DRY
- Wie Funktionen, nur für aussagenlogische Ausdrücke ...

```
1  /*@ ...
2     ensures \result >= 0;
3     ensures \result * \result <= x;
4     ensures x < (\result + 1) * (\result + 1);
5   */
6  int32_t sqrt(int32_t x);
```

```
1  /*@ predicate is_sqrt(integer input, integer output)
2          = output >= 0
3            && output * output <= input
4            && input < (output + 1) * (output + 1);
5   */
6  /*@ ...
7     ensures is_sqrt(x, \result);
8   */
9  int32_t sqrt(int32_t x);
```

Korrektheitsbeweis für Schleifen

Invariante ändert sich *nicht* von Schleifendurchlauf zu Schleifendurchlauf

Variante wird in jedem Schleifendurchlauf *dekrementiert*,
- hat nichtnegativen Startwert
- ⤳ Terminierungsbeweis

`assigns` Wie gehabt, Aussagen über Seiteneffekte

`I` ⤳ Invariante

- `while(c) s;` `I` gelte für Seiteneffekte von `c` gefolgt von `s`
- `for(init; c; step) s;`
  `I` gelte für Seiteneffekte von `c` gefolgt von `s` gefolgt von `step`
- `do s while(c);` `I` gelte für Seiteneffekte von `s` gefolgt von `c`

Schleifenannotationen

Varianten, Invarianten, Seiteneffekte

```
1  /*@
2     loop invariant 0 <= i <= n;
3     loop invariant \forall integer k; 0 <= k < i
4                    ==> b[k] == a[n - 1 - k];
5     loop   variant n - i;
6     loop assigns i, b[0..i - 1];
7   */
8  for (size_t i = 0; i < n; ++i) {
9    b[i] = a[n - 1 - i];
10 }
```

Suchpfad anpassen

- notwendige Werkzeuge im CIP-Pool in `/local`
- nur auf x86-64-Maschinen, also nicht auf den Sun-Ray-Server

⇒ folgende Zeilen in `.bash_profile` notwendig

.bash_profile

```
1 for package in alt-ergo-0.95.2 why-2.34 why3-0.83 beagle-0.7.7 \
2   cvc3-2.4.1 cvc4-1.3 eprover-1.8 gappa-1.1.1 metis-2.3 metitarski \
3   spass-3.7 verit-201310d z3-4.3.1 frama-c-neon; do
4   addpackage ${package}
5 done
```

- erst dann werden die Werkzeuge durch die Shell gefunden!

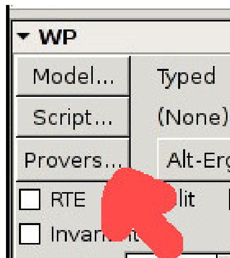
Frama-C per Makefile

Einmal ausführen, damit Frama-C seine Werkzeuge findet

```
1 % why3config --detect
```

Frama-C starten

```
1 % make frama-c-gui
```



Lesenswertes zu ACSL/jessie/frama-c

- http://frama-c.com
- http://frama-c.com/download/user-manual-Neon-20140301.pdf
- http://frama-c.com/download/acsl-implementation-Neon-20140301.pdf

Table of Contents

1 ACSL

2 Aufgabenstellung

Aufgabenstellung

1. Ampel-Programm
   - implementieren
   - und verifizieren

   ⇝ für jede Funktion mindestens ein nicht-triviales `assigns`, `ensures` und `requires`
2. `lower_bound()` verifizieren
   - für die Schleife: Invarianten, Variante und `assigns`-Aussage

Knackpunkt

- Vor-/Nachbedingungen finden
  - Vorbedingung so schwach wie möglich
  - Nachbedingung so stark wie möglich
- leider nicht automatisierbar ☹

Funktionalität von `lower_bound()`

- Vorbedingung: Eingabe-Array ist aufsteigend sortiert
- Binärsuche auf Array
  - Suchraum wird in jedem Schritt halbiert
  - falls mittleres Element größer als Vergleichswert $\leadsto$ suche links weiter
  - sonst suche rechts davon
  - wiederhole bis Mächtigkeit der Suchmenge 1

`lower_bound()` liefert größten Index zurück, für den gilt

Alle Array-Elemente mit niedrigerem Index kleiner als übergebener Wert

Implementierung `lower_bound()`

```
1  size_t lower_bound(const int *a, size_t n, int val)
2  {
3    size_t left = 0, right = n, middle = 0;
4    while (left < right) {
5      middle = left + (right - left) / 2;
6      if (a[middle] < val) {
7        left = middle + 1;
8      } else {
9        right = middle;
10     }
11   }
12   return left;
13 }
```

```
Wert a[k] | 1   4   5   8   9 11 14   18   21   22   23
Index k   | 0   1   2   3   4  5  6    7    8    9   10
                                /
lower_bound(a, 11, 10) == 5
```

Hilfestellung zur lower_bound()-Funktion I

- Annotationen zu reverse_copy() ähnlich zu lower_bound()

Formale Spezifikation reverse_copy()

```
1  /*@
2     predicate IsValidRange(int *a, integer n) =
3           (0 <= n) && \valid(a+(0..n - 1));
4   */
5  /*@
6     requires IsValidRange(a, n);
7     requires IsValidRange(b, n);
8     assigns b[0..(n - 1)];
9     ensures \forall integer i; 0 <= i < n
10                  ==> b[i] == a[n - 1 - i];
11  */
12 void reverse_copy(const int *a, size_t n, int *b);
```

Hilfestellung zur `lower_bound()`-Funktion II

Implementierung `reverse_copy()`

```
1  void reverse_copy(const int *a, size_t n, int *b) {
2    /*@
3       loop invariant 0 <= i <= n;
4       loop invariant \forall integer k; 0 <= k < i
5                       ==> b[k] == a[n - 1 - k];
6       loop   variant n - i;
7       loop assigns i, b[0..i - 1];
8     */
9     for (size_t i = 0; i < n; ++i) {
10      b[i] = a[n - 1 - i];
11    }
12 }
```

Fragen?