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

1. Juli 2014

Überblick

Astrée [1]

- Ziel: Nachweis der Abwesenheit von Laufzeitfehlern
- findet *alle* potentiellen Laufzeitfehler
- leider auch *falsch-positive*
  ⇝ wegen Gödelschem Unvollständigkeitstheorem (1931)

  Jedes hinreichend mächtige formale System ist entweder widersprüchlich oder unvollständig.

Programm ist korrekt, wenn

- Astrée keine Alarme meldet
- oder für alle Alarme nachgewiesen, dass falsch-positiv

Astrée weist nach

- Überschreitung von Array-Grenzen
- Ganzzahldivision durch Null
- ungültige Dereferenzierung, arithmetische Überläufe
- ungültige Gleitkommaoperationen
- dass Code nicht erreichbar ist
- Lesezugriff auf nicht initialisierte Variabeln
- Verletzung benutzerdefinierter Zusicherungen
  ⇝ `assert()`

Einschränkungen

- Rekursionen prinzipiell erlaubt, werden aber nicht analysiert
  - ⇝ für Rekursionsergebnis werden keine Einschränkungen ermittelt
- Auswertungsreihenfolge in C nicht vollständig spezifiziert
  - ⇝ *eine* bestimmte Ordnung wird angenommen
  - stimmt nicht notwendigerweise mit Compiler überein
  - optionale Warnung durch Astrée
- Funktionen der Standard-C-Bibliothek werden nicht erkannt
  - ⇝ Stubs anlegen
- dynamischer Speicher nicht erlaubt
  - ⇝ kein `malloc()`
  - keine Einschränkung im sicherheitskritischen Echtzeitbereich

Semantik

Astrée nimmt an, dass folgende semantische Regeln gelten:

1. der C99-Standard
2. implementierungsabhängiges Verhalten
   - Größe von Datentypen
   - Gleitkommastandard
   - ...
3. benutzerdefinierte Einschränkungen
   - z. B. ob statische Variablen mit 0 initialisiert werden
4. außerdem *benutzerspezifizierte Zusicherungen*
   - ⇝ und da wird es interessant ☺

Benutzerdefinierte Zusicherungen

`__ASTREE_known_fact((B))`

- Analyzer warnt, falls `B` *nie wahr werden kann*

`assert((B))`

- alternativ auch `__ASTREE_assert((B))`
- Analyzer erzeugt Alarm, falls `B` *nicht immer wahr ist*
- `B` kann nicht von der Form `e1 ? e2 : e3` sein

- Analyzer nimmt danach an, dass `B` wahr ist
- `B` muss seiteneffektfrei sein
- *die doppelten Klammern sind wichtig!*

Beispiel

```
1  #include <astree.h> // Astree-Makros ggf. abschalten
2
3  float filter(Alpha_State *s, float val) {
4    __ASTREE_known_fact((val == val));
5    __ASTREE_known_fact((-10.0f < val && val < 10.0f));
6    __ASTREE_known_fact((s->val == s->val));
7    __ASTREE_known_fact((FLT_MIN < s->val
8                         && s->val < FLT_MAX));
9    __ASTREE_assert((0 < s->alpha));
10   __ASTREE_assert((s->alpha < 1));
11
12   float residual = val - s->val;
13   s->val = s->val + s->alpha * residual;
14
15   __ASTREE_assert((s->val == s->val));
16   // ...
17   return s->val;
18 }
```

Stubs

`__ASTREE_modify((V1, ..., Vn))`

- zeigt an, dass Variablen `V1` bis `Vn` unbekannten Wert haben

⇝ braucht man um Stubs zu bauen

Beispiel

```
1  #ifdef __ASTREE__
2  __ASTREE_modify((x));
3  __ASTREE_known_fact((x >= 0));
4  #else
5  // ... Implementierung
6  #endif
```

Synchrone Programme

- Viele Echtzeitsysteme Endlosschleifenbasiert ☹

⇝ Astrée modelliert dies

`__ASTREE_max_clock((N))`

beschränkt Schleifendurchläufe

`__ASTREE_wait_for_clock(())`

wartet auf nächsten Tick

```
1  __ASTREE_max_clock((10)); // sonst evt. keine Schleifeninvariante
2  void main(void) {
3    while (1) {
4      // 1. 'volatile'-Werte lesen
5      // 2. Zustand und Ausgabe berechnen
6      // 3. Ausgabe schreiben
7      __ASTREE_wait_for_clock(()); // auf naechsten Clock-Tick warten
8    }
9  }
```

Asynchrone Programme

- Astrée modelliert auch asynchrone Ausführung von Aufgaben
- Keine Annahmen über Scheduler oder Prioritäten
- `automatic-shared-variables` muss auf `yes` stehen

```
1  int x, y;
2  volatile int s;
3
4  void t1(void) {
5    x = 1; s = 1; x = 0;
6  }
7
8  void t2(void) {
9    if (s > 0) {
10     y = -1;
11   } else {
12     y = 1;
13   }
14 }
15
16 void main(void) {
17   x = y; // synchroner Teil
18   __ASTREE_asynchronous_loop((t1(), t2()));
19 }
```

volatile

`__ASTREE_volatile_input((V))`

- zeigt an, dass `V` sich jederzeit ändern kann

⇝ modelliert Eingabe von außen

`__ASTREE_volatile_input((Vp, r))`

- `p` ist Pfad in der Variablen,
  z. B. `V.a[3-4].b` ⇝ Variable `V`, Arrayelemente `a[3]` und `a[4]`, Struct-Element `b`
  - `[i]` ⇝ Element `i`
  - `[i-j]` ⇝ Elemente `i` bis `j`
  - `[]` ⇝ alle Elemente
- `r` schränkt Wertebereich ein `[i, j]` ⇝ von `i` bis `j`

Analyse untersuchen

`__ASTREE_analysis_log(())`

- gibt Zustand der Analyse an dieser Stelle aus

`__ASTREE_log_vars((V1, ..., Vn))`

- zeigt Zustand der Analyse in Bezug auf einzelne Variabeln an

`__ASTREE_print(("text"))`

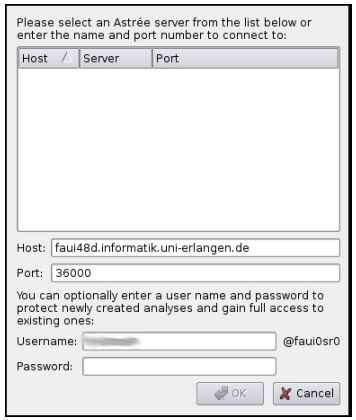
- gibt Text aus

Astrée verwenden

- Astrée im CIP:
  `% /proj/i4ezs/tools/astree-b217166/bin/astreec`
- Anmeldung mit Benutzername und Passwort
  - ⇝ Passwort wird bei der ersten Anmeldung festgelegt
    - Wir wissen nicht, wie man es nachträglich ändert
  - ⇝ gut merken ☺
- Dokumentation unter
  `/proj/i4ezs/tools/astree-b217166/share/astree_c/help`

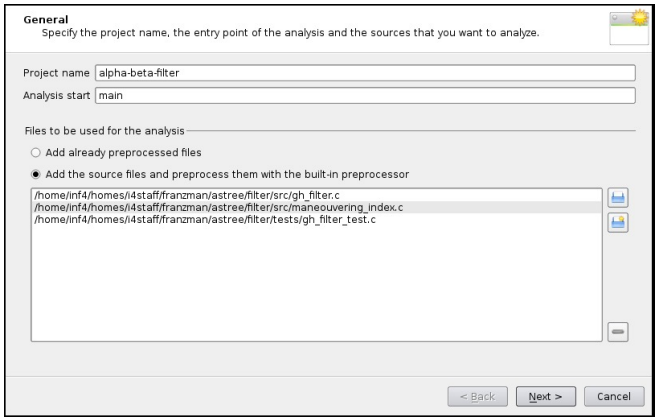
Anmelden

Host	faui48*f*.informatik.uni-erlangen.de
Port	36000
Username	nach Belieben
Password	bei der ersten Anmeldung festlegen und *gut merken*



Projekt anlegen

- Quelldateien zum Projekt hinzufügen



Präprozessoreinstellungen

- Include-Pfade festlegen

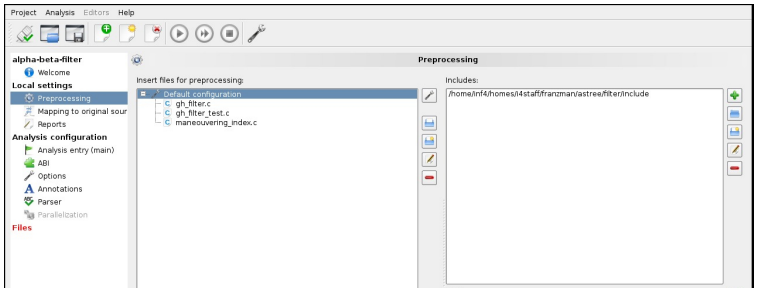


Table of Contents

Aufgabenstellung

- einfaches Filter implementieren
- Korrektheit der Implementierung nachweisen
  ⇝ Astrée
- zunächst mit Gleitkommaarithmetik
  ⇝ `float`
- dann für einen 8 Bit-Mikrocontroller mit Festkommaarithmetik

Festkommaarithmetik – Q-Notation

- kaum ein Mikrocontroller hat eine Gleitkommaeinheit
  ⇝ Festkommaarithmethik mit Ganzzahlen
- Zahlenformat häufig in Q-Notation [7] angegeben
- Q$m.n$ ⇝ Festkommazahl mit
  - m Bit vor dem Komma
  - n nach dem Komma
  - und einem Vorzeichenbit
  - Wertebereich: $[-2^m, 2^m - 2^{-n}]$
  - Auflösung: 2^{-n}

Implementierung als Integer

⇝ welches Q-Format Verwendung findet, ist dem Anwendungsprogrammierer überlassen

Q-Notation – Beziehung zu Gleitkommazahlen

von Gleitkomma nach Qm.n

1. Multiplikation mit 2^n
2. Runden auf die nächste Ganzzahl

von Qm.n nach Gleitkomma

1. Umwandlung in Gleitkommazahl $\rightsquigarrow$ cast
2. Multiplikation mit 2^{-n}

Operationen – Addition/Subtraktion

- Addition und Subtraktion wie bei Ganzzahlen

Addition

```
1 int8_t      a = ...;
2 int8_t      b = ...;
3 int8_t result = a + b;
```

Subtraktion

```
1 int8_t      a = ...;
2 int8_t      b = ...;
3 int8_t result = a - b;
```

Operationen – Multiplikation/Division

- braucht Zwischenergebnis von doppelter Bitbreite

Multiplikation

```
1 #define K   (1 << (n - 1))
2 int8_t     a = ...;
3 int8_t     b = ...;
4 int16_t temp = (int16_t) a * (int16_t) b;
5         temp += K;
6 int8_t result= temp >> n;
```

Division

```
1 int8_t      a = ...;
2 int8_t      b = ...;
3 int16_t  temp = (int16_t) a << n;
4         temp += b / 2;
5 int8_t result = temp / b;
```

Größe von Datentypen bestimmen I

```
% cat test.c
1  #include <stddef.h>
2
3  static size_t short_size      = sizeof(short);
4  static size_t ushort_size     = sizeof(unsigned short);
5  static size_t int_size        = sizeof(int);
6  static size_t uint_size       = sizeof(unsigned int);
7  static size_t long_size       = sizeof(long);
8  static size_t ulong_size      = sizeof(unsigned long);
9  static size_t longlong_size   = sizeof(long long);
10 static size_t float_size      = sizeof(float);
11 static size_t double_size     = sizeof(double);
12 static size_t longdouble_size = sizeof(long double);
13 static size_t ptr_size        = sizeof(void *);
14 static size_t fptr_size       = sizeof(void (*)(void));
15
16 void main(void) {}
```

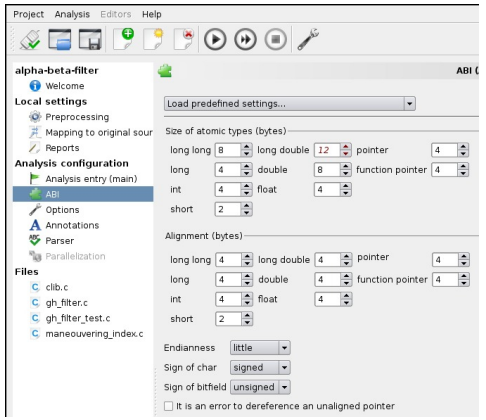
Größe von Datentypen bestimmen II

```
% avr-gcc -O0 -S -c test.c && cat test.s
```

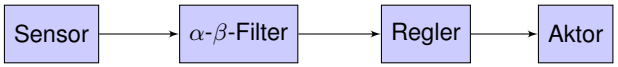
```
1  ...
2  short_size:
3    .word 2 ← short zwei Byte lang
4  .subsection  2
5    .align 2
6    .type  ushort_size,@object
7    .size  ushort_size,4
8  ...
```


ABI

- ABI festlegen



α-β-Filter [3]

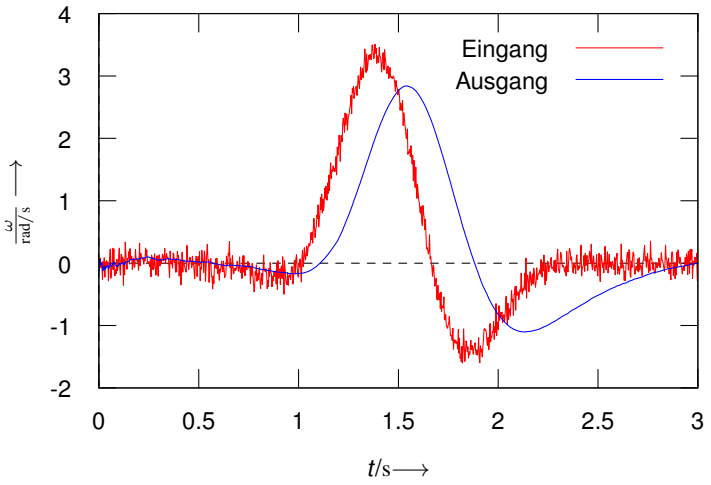


- Rauschunterdrückungsfilter
- geeignet zur Schätzung von physikalischen Größen
  - mit Ableitung $\neq 0$
  - z. B. Position eines Flugzeugs, Lagewinkel ...
  - im I4Copter
    - liefern Sensoren häufiger Werte als diese später verarbeitet werden

    $\rightsquigarrow$ α-β-Filter für *Ratenwandlung* verwendet

    $\rightsquigarrow$ nutzt gewonnene Information vollständig

Beispiel α-β-Filterung



Filteralgorithmus

- wird für jeden Messwert ausgeführt
- $y[\kappa]$: Eingabewert für Abtastschritt κ
- $\hat{x}[\kappa]$: Schätzung der Messgröße zum Abtastschritt κ
- T Abtastintervall, α, β Filterparameter
- Initialisierung: z. B. $\hat{x}[0] = \hat{\dot{x}}[0] = 0$

$$r[\kappa] = y[\kappa] - \hat{x}[\kappa - 1] \quad \rightsquigarrow \text{Schätzfehler} \quad (1)$$

$$\hat{\dot{x}}[\kappa] = \hat{\dot{x}}[\kappa - 1] + \frac{\beta}{T} \cdot r[\kappa] \quad \rightsquigarrow \text{1. Ableitung} \quad (2)$$

$$\hat{x}[\kappa] = \hat{x}[\kappa - 1] + T \cdot \hat{\dot{x}}[\kappa] + \alpha \cdot r[\kappa] \quad \rightsquigarrow \text{Schätzwert} \quad (3)$$

- sinnvolle Werte für α und β?
  - Literatur beschreibt viele Verfahren $\rightsquigarrow$ hier beispielhaft nur eines [5, 4]

Filterparameter

T Abtastintervall
$\rightsquigarrow$ in welchem Zeitabstand gemessen wird

σ_w^2 Prozessvarianz
$\rightsquigarrow$ wie lebhaft der gemessene Prozess ist

σ_v^2 Rauschvarianz
$\rightsquigarrow$ wie verrauscht das Signal ist

$$\lambda = \frac{\sigma_w T^2}{\sigma_v} \quad \rightsquigarrow \text{Tracking Index} \quad (4)$$

$$\theta = \frac{4 + \lambda - \sqrt{8\lambda + \lambda^2}}{4} \quad \rightsquigarrow \text{Dämpfungsparameter} \quad (5)$$

$$\alpha = 1 - \theta^2 \quad \rightsquigarrow \text{Gewicht für Wert} \quad (6)$$

$$\beta = 2(2 - \alpha) - 4\sqrt{1 - \alpha} \quad \rightsquigarrow \text{Gewicht für Ableitung} \quad (7)$$

Initialisierungsphase

- Zu Beginn hat das Filter keinen gültigen Zustand
- $\rightsquigarrow$ Einschwingphase, in der das Filter „lernt“
- n startet bei 0 und wächst in jeder Runde um 1

$$\alpha_n = \frac{2(2n+1)}{(n+2)(n+1)} \quad (8)$$

$$\beta_n = \frac{6}{(n+2)(n+1)} \quad (9)$$

- Einschwingphase endet, wenn $\alpha_n < \alpha$
- Wird der Filterzustand im Betrieb ungültig, wird eine neue Einschwingphase eingeleitet

Korrektheitsbedingung

- Filter ist nur dann korrekt, wenn es auch *stabil* ist
  - $\rightsquigarrow$ für wertbegrenzte Eingabe erfolgt wertbegrenzte Ausgabe [6]
  - $\rightsquigarrow$ für Eingabe 0 geht der Filterausgang asymptotisch gegen 0
- α-β-Filter stabil, wenn gilt

$$0 < \alpha \leq 1 \quad (10)$$

$$0 < \beta \leq 2 \quad (11)$$

$$0 < 4 - 2\alpha - \beta \quad (12)$$

- Außerdem: laut [2] Rauschunterdrückung nur dann, wenn

$$0 < \beta < 1 \quad (13)$$

- Stabilität muss auch in der Initialisierungsphase gegeben sein!

Sinnvolle Wertebereiche

- Erfahrungen mit dem I4Copter haben gezeigt, dass sich die Parameter in folgenden Bereichen bewegen:

  Abtastintervall $T \in (0 \ldots 1]$
  Prozessvarianz $\sigma_w^2 \in [0.5 \ldots 30.0]$
  Rauschvarianz $\sigma_v^2 \in \left[10^{-3} \ldots 10^{-1}\right]$
  Wert $y\,[\kappa] \in [-10 \ldots 10]$

$\rightsquigarrow$ Korrektheit mindestens für diese Wertebereiche zeigen!

Literatur I

[1] AbsInt Angewandte Informatik GmbH.
The Static Analyzer Astrée, April 2012.

[2] C. Frank Asquith.
Weight selection in first-order linear filters.
Technical report, Army Intertial Guidance and Control Laboratory Center, Redstone Arsenal, Alabama, 1969.

[3] Eli Brookner.
Tracking and Kalman Filtering Made Easy.
Wiley-Interscience, 1st edition, 4 1998.

[4] E. Gray, J. and W. Murray.
A derivation of an analytic expression for the tracking index for the alpha-beta-gamma filter.
IEEE Trans. on Aerospace and Electronic Systems, 29:1064–1065, 1993.

[5] Paul R. Kalata.
The tracking index: A generalized parameter for α-β and α-β-γ target trackers.
IEEE *Transactions on Aerospace and Electronic Systems*, AES-20(2):174–181, mar 1984.

Literatur II

[6] Richard G. Lyons.
Understanding Digital Signal Processing.
Prentice Hall, 3rd edition, 11 2010.

[7] Erick L. Oberstar.
Fixed-point representation & fractional math.
Technical report, Oberstar Consulting, August 2007.

Fragen?