Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Phillip Raffeck, Florian Schmaus, Simon Schuster

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

Sommersemester 2019

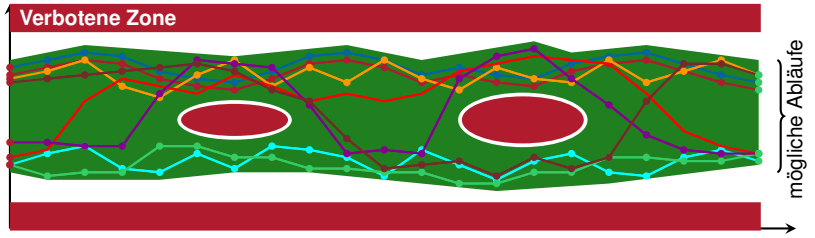
Erinnerung: Evaluation der Lehrveranstaltung

- TANs werden in der Übung verteilt
- $\geq 70\,\%$ Rückläuferquote $\rightsquigarrow$ Belohnung: **Semesterabschlussgrillen**
- Termin der Evaluation **?**

Überblick

Überblick

Abstrakte Interpretation



- Abstrakte Interpretation (engl. *abstract interpretation*)
  - Betrachtet eine abstrakte Semantik (engl. *abstract semantics*)
    - Sie umfasst alle Fälle der konkreten Programmsemantik
  - Ist die abstrakte Semantik sicher ⇒ konkrete Semantik ist sicher

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
- Unerreichbarer Code
- Lesezugriff auf nicht initialisierte Variabeln
- Verletzung benutzerdefinierter Zusicherungen
  ⇝ `assert()`

Einschränkungen

- Rekursion prinzipiell erlaubt, wird 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
  ⇝ mitgelieferte Stubs nutzen
- 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 nimmt an, dass `B` gegeben ist
- Analyzer warnt, falls `B` *nie wahr werden kann*
- `__ASTREE_known_range((V, [a; b]))` ⇝ Wertebereich

`assert((B))`/`__ASTREE_assert((B))`

- Analyzer erzeugt Alarm, falls `B` *nicht immer wahr ist*
- `B` kann nicht von der Form `e1 ? e2 : e3` sein
- `__ASTREE_global_assert(())` ⇝ gesamtes Programm

- `B` muss seiteneffektfrei sein
- *Doppelte Klammerung ist 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)); // known_fact(!isnan(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((0.0f < s->alpha));
10   __ASTREE_assert((s->alpha < 1.0f));
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: Emulation von Sensoren

Beispiel

```
1 #ifdef __ASTREE__
2 __ASTREE_modify((x; full_range));
3 __ASTREE_modify((x; [10, 20]))
4 __ASTREE_known_fact((x >= 0));
5 #else
6 // ... Implementierung
7 #endif
```

Schleifen ausrollen

- Astrée beschreibt abstrakte Semantik
- Frage: Wie viele Schleifendurchgänge betrachten?
- ⇝ Astrée versucht Aufwand zu vermeiden
- ⇝ Schleifen/Verzweigungen werden nicht ausgerollt
- Konsequenz:

Beispiel

```
1 unsigned int i = 0;
2 unsigned int j = 20;
3 while (j > 0) { --j; ++i; }
```

- Astrée kann nicht zeigen, daß die Schleife terminiert (☞ Satz von Rice, 1953)
- ⇝ Annahme für weitere Analyse: `i` läuft über

Lösung:

```
1 __ASTREE_unroll((30))
2 while (j > 0) { --j; ++i; }
```

Verzweigungen

- Dito bei Verzweigungen
- Astrée betrachtet normalerweise nur den schlimmsten Fall aller Zweige
- ⇝ Pessimistisches Ergebnis
- Falls Betrachtung der unterschiedlichen Pfade erforderlich:
- Lösung: Analyse vorübergehend aufspalten:

Verzweigungsanalyse

```
1 __ASTREE_partition_control
2 if (...) { ... }
3 else { ... }
4 ...
5 __ASTREE_partition_merge(());
```

- Auch für Schleifen, `switch` und Funktionsaufrufe
- ⇝ Blick ins Handbuch: es gibt noch weitere Tricks

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

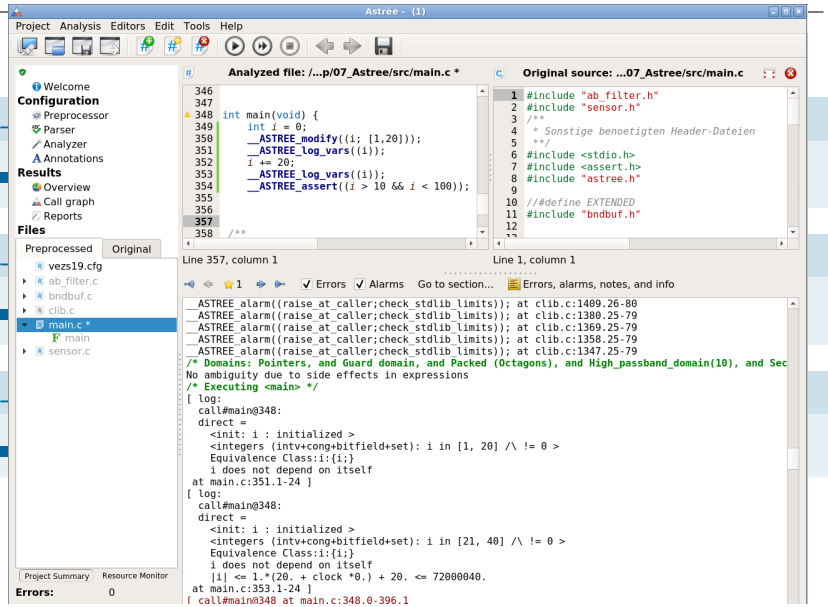
`__ASTREE_log_vars((V1, ..., Vn))`

- Zeigt Zustand der Analyse in Bezug auf einzelne Variabeln an

`__ASTREE_print(("text"))`

- Gibt Text aus

Analyse untersuchen



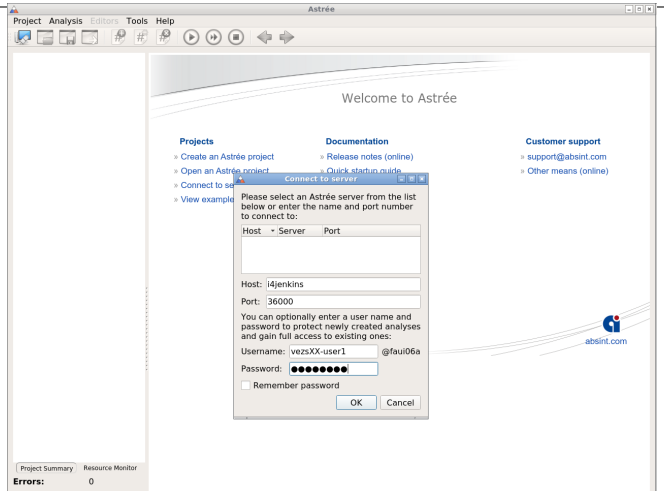
Astrée verwenden

- Astrée im CIP:
  `% /proj/i4ezs/tools/astree_c/bin/astreec`
- Anmeldung mit Benutzername und Passwort

⇝ Passwort wird bei der ersten Anmeldung festgelegt

- Dokumentation
  - HTML: `/proj/i4ezs/tools/astree_c/share/astree_c/share/html/index.html`
  - PDF: `/proj/i4ezs/tools/astree_c/help/astreec_QuickStart.pdf`

Anmelden

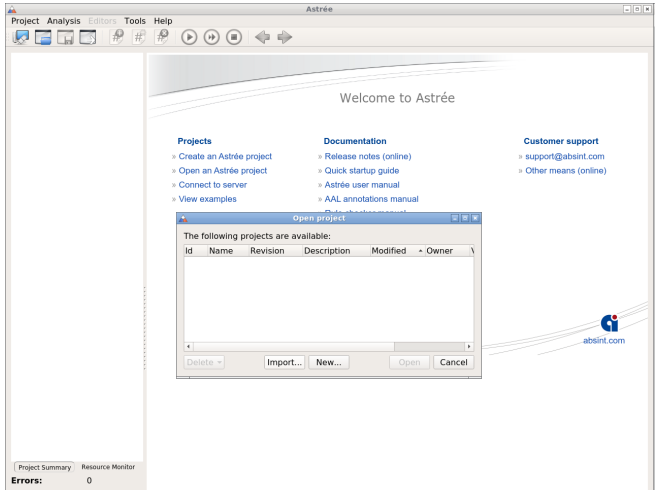


Host i4jenkins.cs.fau.de

Port 36000

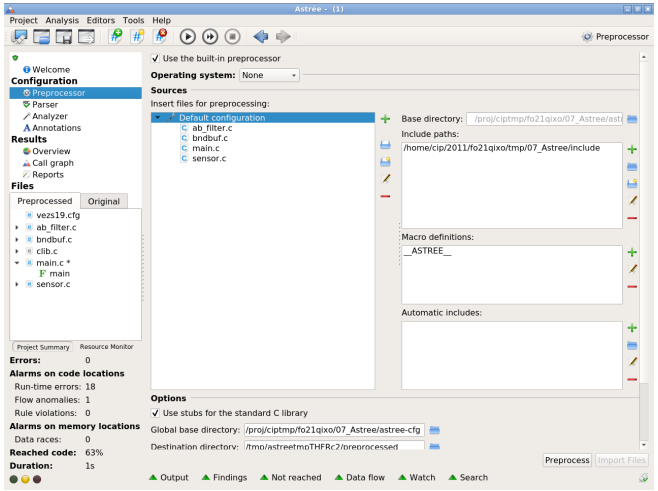
Beliebigen Benutzername/Passwort wählen.

Projekt anlegen



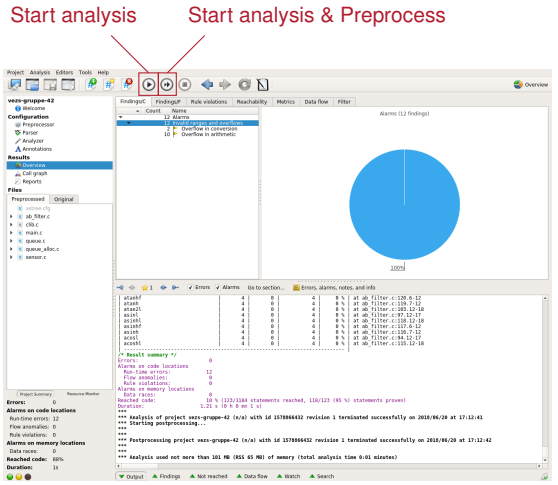
- „Import...“
- Projekt aus Vorgabedatei `astree-cfg/vezs19.dax` importieren

Quelldateien konfigurieren



- Quelltextdateien und Includepfade definieren
- Haken bei „Empty destination directory before preprocessing“ setzen

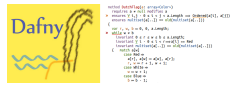
Analyse starten



Überblick

Programme zur Softwareverifikation

- Astrée
- Frama-C
- VeriFast
- Dafny
- SeaHorn
- Coq
- Isabelle/HOL
- Agda
- Idris
- ...



WP-Kalkül [4]

- Bestimmt die schwächste notwendige Vorbedingung $wp(S, Q)$
  - Für ein gegebenes imperatives Programmsegment S
  - Um die ebenfalls gegebene Nachbedingung Q sicherzustellen
  - Dieser Sachverhalt wird beschrieben durch: $P \Rightarrow wp(S, Q)$
    - Lässt sich die schwächste notwendige Vorbedingung $wp(S, Q)$ aus der gegeben Vorbedingung P folgern?
- Das WP-Kalkül ist eine Rückwärtsanalyse
  - Sie beginnt mit der Nachbedingung und durchläuft das Programmsegment in umgekehrter Reihenfolge
  - „Sozusagen“ umgekehrter Einsatz der Regeln des Hoare-Kalküls
- Jeder Anweisung wird eine Prädikattransformation zugewiesen
  - Abbildung: Nachbedingung $\mapsto$ notwendige schwächste Vorbedingung

  → Eine rückwärtige symbolische Ausführung des Programmsegments
- Frama-C setzt WP-Kalkül ein: Nachweis der Schritte erfolgt über verschiedene Theorembeweiser und Löser: Alt-Ergo, Coq, Why3, Z3, ...

Beispiel: WP-Kalkül Maximumsfunktion

WP : ? $(a > b \Rightarrow a \geq b \wedge a \geq a) \wedge (\neg(a > b) \Rightarrow b \geq b \wedge b \geq a)$

$\Leftrightarrow$ $(a > b \Rightarrow a \geq b) \wedge (a \leq b \Rightarrow b \geq a)$

$\Leftrightarrow$ $\top$ // *Gilt ohne Vorbedingung*

`if (a > b)`

$a > b \Rightarrow a \geq b \wedge a \geq a$

`result = a;`

$a > b \Rightarrow result \geq b \wedge result \geq a$

`else`

$\neg(a > b) \Rightarrow b \geq b \wedge b \geq a$

`result = b;`

$\neg(a > b) \Rightarrow result \geq b \wedge result \geq a$

$result \geq b \wedge result \geq a$

`return result ;`

$\backslash res \geq b \wedge \backslash res \geq a$: Q

Auswertungsreihenfolge

$wp(\texttt{if b then S1 else S2}, Q) \equiv (b \Rightarrow wp(S1, Q)) \wedge (\neg b \Rightarrow wp(S2, Q))$

$wp(\texttt{x = y}, Q) \equiv Q[x/y]$

$wp(\texttt{x = y}, Q) \equiv Q[x/y]$

$wp(\texttt{return x}, Q) \equiv Q[\backslash res/x]$

Frama-C: Syntax

- erlaubt Nachweise funktionaler Eigenschaften mittels wp-Kalkül
  - Prädiaktenlogik erster Stufe
    - ∀: `\forall`
    - ∃: `\exists`
    - ⇒: Implikation, `==>`
    - Die aus C bekannten aussagenlogischen Verknüpfungen: `!`, `&&`, `||`, `==`, `!=`, …
- Vor-/Nachbedingungen als Kommentare direkt im C-Code vor der betreffenden Funktion
  - `//@` oder `/*@ */`
- Vorbedingung: `requires`
- Nachbedingung: `ensures`
- Variablen, die durch die Funktion verändert werden: `assigns`
  - inklusive Arraybereiche: `a[start..end]`
  - Spezialfall `\nothing`
- Ergebnis des Funktionsaufrufs: `\result`

Beispiel: Maximumsfunktion

P : wahr

S :
```
int maximum(int a,int b) {
  int result = INT_MIN;

  if(a > b)
    result = a;
  else
    result = b;

  return INT_MAX;
}
```

Q : result $\geq a \wedge$ result $\geq b \wedge$ (result $== a \vee$ result $== b$)

```
/*@
  assigns \nothing;

  ensures \result >= a;
  ensures \result >= b;
  ensures \result == a || \result == b;
*/
int maximum(int a, int b) {
  int result = INT_MIN;
  if (a > b) {
    result = a;
  } else {
    result = b;
  }
  return result;
}
```

Frama-C (II): Speicherspezifikation

```
// Swap array[0] with pointer other, do not modify the rest of array

/*@ requires \valid(array) && \valid(other);
    requires \base_addr(array) != \base_addr(other);

    assigns array[0], *other;

    ensures array[1 .. (length - 1)] == \old(array[1 .. (length - 1)]);
    ensures *other    == \old(array[0]);
    ensures array[0] == \old(*other); */
void swap_first_element(int *array, size_t length, int *other) {
  int tmp  = array[0];
  array[0] = *other;
  *other   = tmp;
}
```

- Übergebene Zeiger sind gültig: `\valid(ptr)`
- Speicherobjekte hinter Zeigern überlappen sich nicht:
  `\base_addr(ptr1) != \base_addr(ptr2)`
- `assigns`: Nur bestimmte (außen sichtbare) Variablen werden verändert
- `ensures`: Zusammenhänge zwischen Werten vor (`\old(var)`) und nach (`var`) der Ausführung anfordern

Schleifen

Gesucht: $wp($`while (B) S;`$, Q)$

- Wann ist die Berechnung korrekt?
  - Stellt Q her $\rightsquigarrow$ Schleifen**invariante**
  - Terminiert $\rightsquigarrow$ Schleifen**variante**
- Schleifenvariante: Streng monoton fallender, *nichtnegativer* Ausdruck
  In Frama-C: `//@ loop variant length - i;`
- Schleifeninvariante: Ausdruck, der vor der Schleife als nach jedem Durchlauf des Schleifenkörpers B gilt
  In Frama-C: `//@ loop invariant i % 2 == 0;`
- Ferner: Schleife überschreibt nur bestimmte Werte
  In Frama-C: `//@ loop assign i, j, *ptr;`
- Beispiel:

```
int i = 0;
while(i < 30) { i++; }
```

  - Schleifenvariante: $30 - i \geq 0$
  - Schleifeninvariante: $0 <= i <= 30$

Die Schleifenvariante gilt vor, während und nach der Schleife

Nach der Schleife ist die Schleifenbedingung falsch

$\rightsquigarrow \neg(i < 30) \wedge (0 \leq i \leq 30) \rightsquigarrow i == 30$

Beispiel: Maximum in Liste finden

$P : a \neq \text{NULL} \wedge \text{length} > 0$

S :
```
int findMax(int *a, size_t length) {
  int max = a[0];
  for (size_t i = 0; i < length; i++) {
    if (a[i] > max) {
      max = a[i];
    }
  }
```

$Q : \forall k.k \geq 0 \wedge k < \text{length} \Rightarrow \text{a[k]} \leq \text{result}$

$\exists k.k \geq 0 \wedge k < \text{length} \Rightarrow \text{a[k]} == \text{result}$

```
/*@ requires length > 0;
    requires \valid(a + (0..length-1));
    assigns \nothing;
    ensures \forall int k;
        0 <= k < length ==> \result >= a[k];
    ensures \exists int k;
        0 <= k < length && \result == a[k];
 */
int findMax(int *a, int length) {
  int max = a[0];
    int max_idx = 0;
    /*@ loop invariant \forall int k;
                0 <= k < i ==> max >= a[k];
        loop invariant a[max_idx] == max;
        loop invariant 0 <= i <= length;
        loop invariant 0 <= max_idx < length;
        loop assigns i, max_idx, max;
    */
    for (int i = 0; i < length; i++) {
      if (a[i] > max) {
        max = a[i];
        max_idx = i;
      }
    }
    return max;
}
```


Frama-C (III): Prädikate

- „Wiederverwendbare“ aussagenlogische Formulierung
  ⇝ logische Prädikate: `predicate`
- ⇝ „Funktionen“ auf ACSL-Ebene, erleichtern Formulierung von Bedingungen
- Beispiel: Maximum einer Sequenz

```
//@ predicate is_max_of_seq(int max, int *seq, int start, int end) =
      \forall int i; start <= i < end ==> max >= seq[i];
/*@ ...
    ensures is_max_of_seq(\result, a, (int)0, length);
  ...
 */
int findMax(int *a, int length) {
  ...
    /*@ loop invariant is_max_of_seq(max, a, (int)0, i);
        ...
    */
  for (int i = 0; i < length; i++) {
    ...
  }
  return max;
}
```


Frama-C: Datenstrukturen beschreiben

- Invarianten über der Datenstruktur definieren
- Jede Methode:
  - Setzt Gültigkeit der Invarianten voraus
  - Erhält Sie nach Ausführung
- ⇝ Bei Kapselung: Korrektheit der Datenstruktur sichergestellt
- Beispiel: Konto

```
struct account {
  int balance;
  int credit_limit;
};
```

```
/*@ requires \valid(a);
    ensures  valid_account(a); */
void init(struct account *a) {
  a->balance = 0; a->credit_limit = 0;
}
```

- Invariante: Kreditrahmen wird nie verletzt:

```
/*@ predicate valid_account(struct account *a) =
      \valid(a)                       // Gültiger Zeiger
      && a->balance >= a->credit_limit; // Kreditrahmen nicht verletzt
 */
```

- Alle Methoden erfordern und erhalten die Invariante:

```
/*@ requires valid_account(a);
    requires amount > 0;
    ensures  valid_account(a); */
bool withdraw(struct account *a, int amount) {
  /* ... */
  return true;
}
```

```
/*@ requires valid_account(a);
    requires amount > 0;
    ensures  valid_account(a); */
void deposit(struct account *a, int amount) {
  /* ... */
}
```

⇝ Solange das Konto nur per `withdraw` und `deposit` modifiziert wird, kann der Kreditrahmen nie verletzt werden

Frama-C: Behaviors

- Oft gelten Nachbedingungen nur für bestimmte Eingabewerte

```
/*@ requires i != INT_MIN;
    ensures i < 0  ==> \result == -i;
    ensures i >= 0 ==> \result ==  i; */
int abs(int i) {
  if (i >= 0) return  i;
  else        return -i;
}
```

- Behaviors beschreiben Verhalten in bestimmten Kontexten:
  - `assumes`: Voraussetzungen, die das Verhalten aktiviert
  - `ensures`: Nachbedingung, die die Funktion dann einhält

Eingabe ist negativ

```
behavior negative:
  assumes i < 0;
  ensures \result == -i;
```

Eingabe ist positiv

```
behavior positive:
  assumes i >= 0;
  ensures \result == i;
```

  - `complete behaviors`: Die Beschreibung ist vollständig
    Behaviors beschreiben das Verhalten für alle Aufrufkontexte
  - `disjoint behaviors`: Die beschriebenen Verhalten überlappen sich nicht

Frama-C: Beispiel Behaviors

```c
#include <limits.h>

/*@ requires i != INT_MIN;
    behavior negative:
      assumes i < 0;
      ensures \result == -i;
    behavior positive:
      assumes i >= 0;
      ensures \result == i;
    complete behaviors;
    disjoint behaviors; */
int abs(int i) {
  if (i >= 0) return  i;
  else        return -i;
}
```

Frama-C: Fallstricke

- Die Frama-C-Gui bietet keinen Editor an!
- Reihenfolge der Annotationen z.T. relevant. Empfehlung:

Funktionen	*Schleifen*
requires	loop invariants
assigns	loop assigns
ensures	loop variants

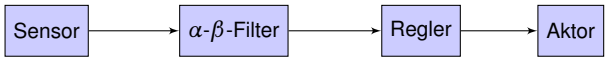
- `assert()` wird als nicht terminierend angenommen
  ⇝ Frama-C assert verwenden: `//@ assert x == 1;`
- Mehrere Annotationen immer in einen gemeinsamen Block `/*@ */`
- Weitere Informationen:
  - Frauenhofer Fokus: ACSL-By-Example:
    `https://github.com/fraunhoferfokus/acsl-by-example`
  - A. Blanchard: Introduction to C program proof with Frama-C and its WP plugin:
    `https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf`

Überblick

Aufgabenstellung

- Sensorstubs implementieren
- Implementierung eines Ringpuffers zur Verwaltung der Sensorwerte
- System erstellen:
  - Sensoren und Filter initialisieren
  - Sensorwerte abrufen
  - Sensorwerte filtern
- Abwesenheit von Laufzeitfehlern nachweisen
  ⇝ Astrée
- Numerische Stabilität des Filters nachweisen
  ⇝ Astrée
- Funktionale Korrektheit des Ringpuffers nachweisen
  ⇝ Frama-C

α-β-Filter [3]

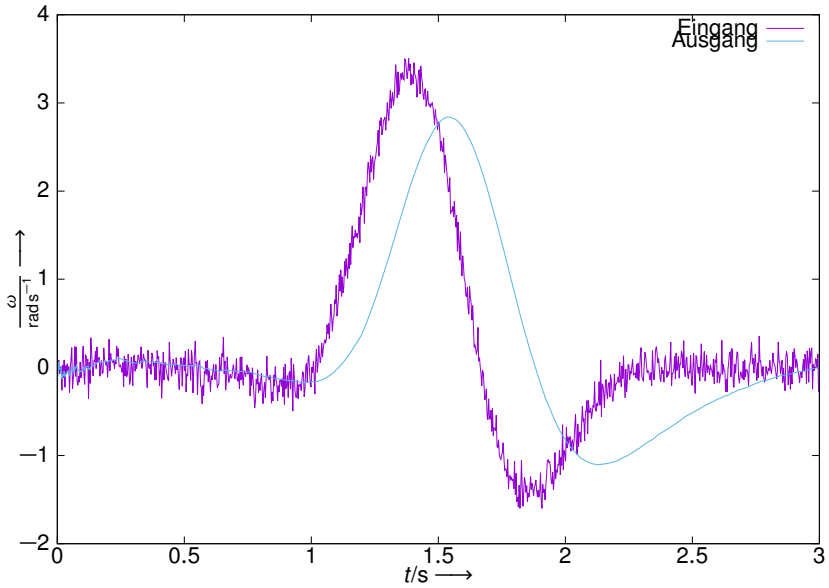


- Rauschunterdrückungsfilter
- Geeignet zur Schätzung von physikalischen Größen
  - mit Ableitung $\neq 0$
  - z. B. Position eines Flugzeugs, Lagewinkel ...
  - im I4Copter
    - liefern Sensoren häufiger Werte als diese später verarbeitet werden

  $\rightsquigarrow$ 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 [6, 5]

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
  ⇝ Einschwingphase, in der das Filter „lernt“
- n startet bei 1 und wächst in jeder Runde um 1

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

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

- 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
  ⇝ für wertbegrenzte Eingabe erfolgt wertbegrenzte Ausgabe [7]
  ⇝ für Eingabe 0 geht der Filterausgang asymptotisch gegen 0
- α-β-Filter stabil, wenn gilt

$$0 < \alpha \leq 1 \tag{10}$$

$$0 < \beta \leq 2 \tag{11}$$

$$0 < 4 - 2\alpha - \beta \tag{12}$$

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

$$0 < \beta < 1 \tag{13}$$

- 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 $y[\kappa] \in [-10 \ldots 10]$

⇝ Korrektheit mindestens für diese Wertebereiche zeigen!

Ringpuffer – Implementierung

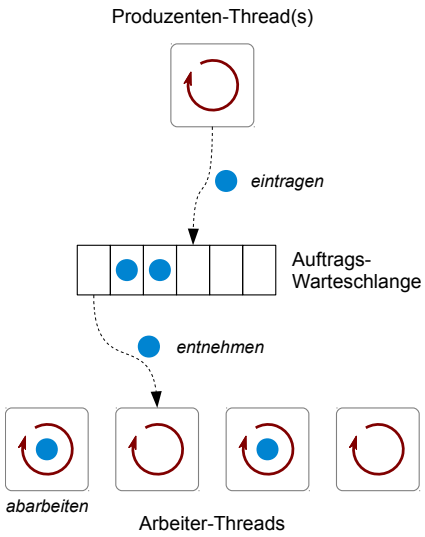
- Beispiel für dynamisch angelegten Ringpuffer

```
1 struct BndBuf{
2   int* buf; // array of <size>
3   int size;
4   int read_pos;
5   int write_pos;
6 };
```

- „Füllstandsanzeige”: Speichern von
  - Belegten Plätzen
  - Verfügbaren Plätzen
- Mögliche Signalisierung falls
  - Plätze frei wurden
  - Plätze belegt wurde

Ringpuffer – Anwendungsszenario

- Verwaltung einer begrenzten Anzahl an Ressourcen
- Beispiel: Thread-Pool
- Feste Menge von Arbeiter-Threads:
  - laufen endlos
  - erhalten Aufträge zur Abarbeitung
- Verteilen der Aufträge mittels zentraler, synchronisierter Warteschlange (z. B. Ringpuffer)
- Vorteil: kein ständiges Erzeugen + Zerstören von Threads für Aufträge



Literatur I

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

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

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

Edsger W. Dijkstra.
Guarded commands, nondeterminacy and formal derivation of programs.
Communications of the ACM, 18(8):453–457, August 1975.

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.

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

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