Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Abstrakte Interpretation & Verifikation

Tobias Klaus, Florian Schmaus, Peter Wägemann

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

27. Juni 2016

Überblick

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
- Unerreichbarer Code
- 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
  ⇝ 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 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

- Analyzer nimmt danach an, dass `B` wahr ist
- `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));
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_known_fact((x >= 0));
4 #else
5 // ... Implementierung
6 #endif
```

Schleifen ausrollen

- 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 Zweig

⇝ Pessimistisches Ergebnis

- 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

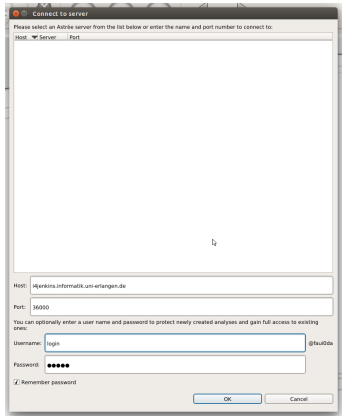
`__ASTREE_print (("text"))`

- Gibt Text aus

Astrée verwenden

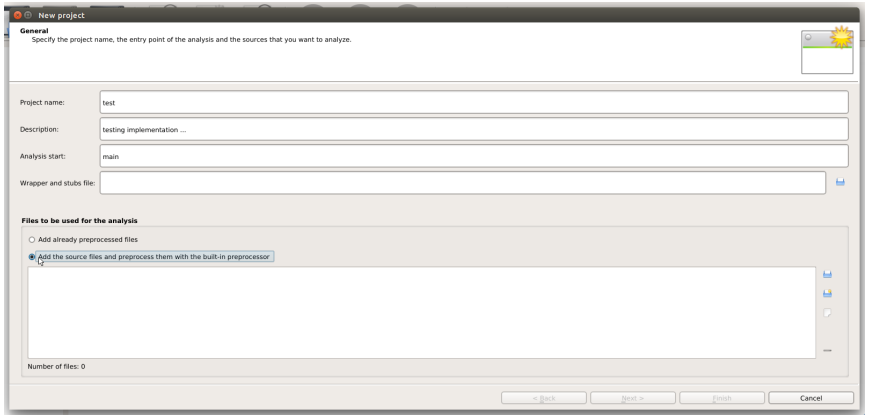
- Astrée im CIP:
  `% /proj/i4ezs/tools/astree_c/bin/astreec`
- Anmeldung mit Benutzername und Passwort
  ⇝ Passwort wird bei der ersten Anmeldung festgelegt
- Dokumentation
  - HTML: `/proj/i4ezs/tools/astree_c/share/astree_c/share/html/index.html`
  - PDF: `/proj/i4ezs/tools/astree_c/help/astreec_QuickStart.pdf`

Anmelden



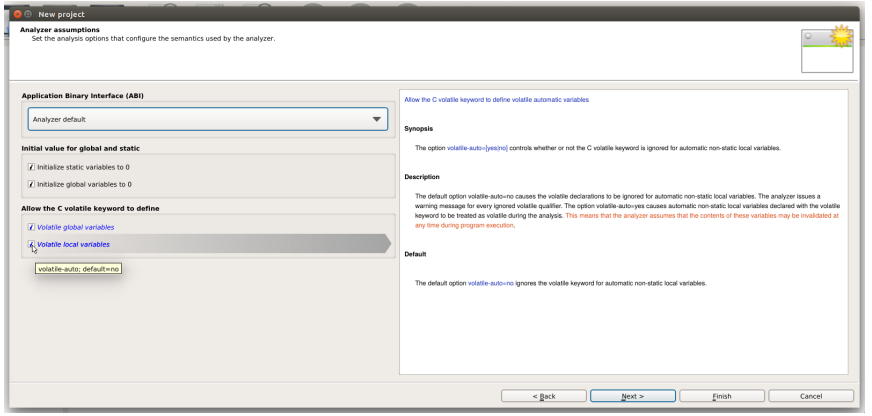
Host	i4jenkins.cs.fau.de
Port	36000
Username	nach Belieben
Passwort	bei der ersten Anmeldung festlegen und *gut merken*

Projekt anlegen



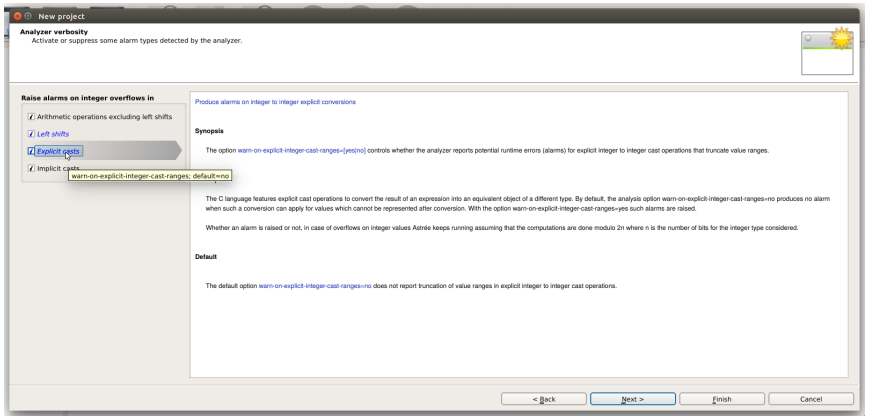
- Quelldateien zum Projekt hinzufügen

Analysen konfigurieren I



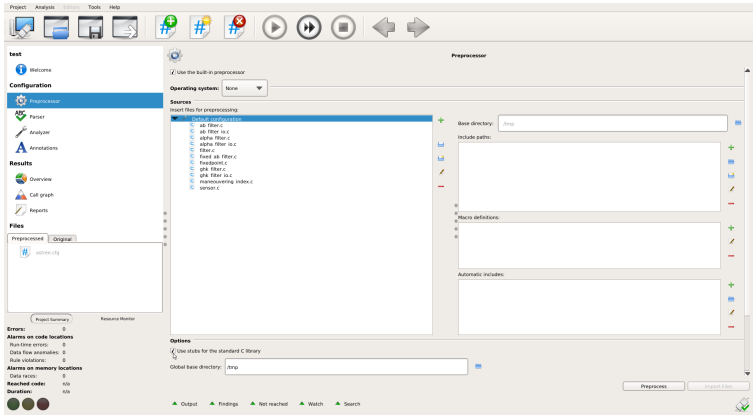
- *Volatile global variables*
- *Volatile local variables*

Analysen konfigurieren I



- *Left shift*
- *Explicit casts*

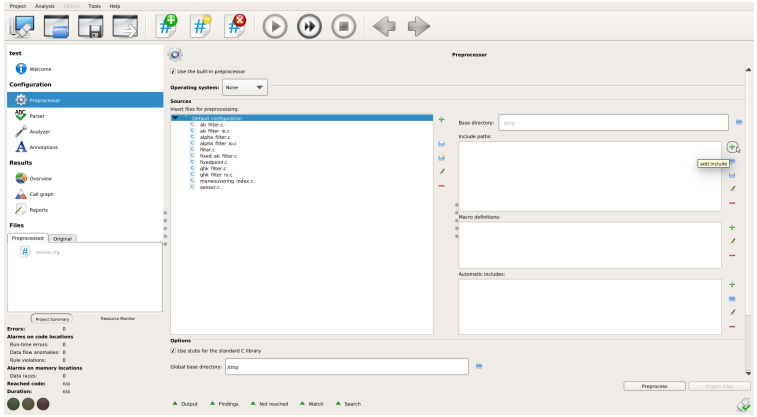
Libc Stubs



- Libc soll nicht mit analysiert werden:
  Use stubs for the standard C library

☞ Stubs für libc verwenden

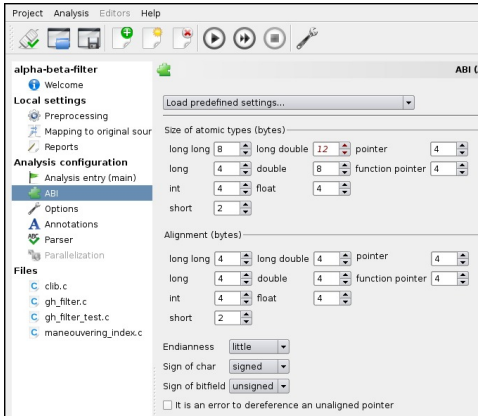
Präprozessoreinstellungen



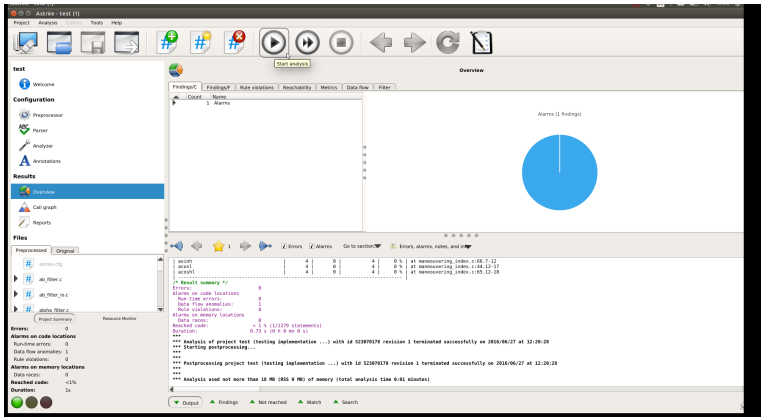
- Include-Pfade festlegen
- __ASTREE__ definieren

ABI

- ABI festlegen



Analyse starten

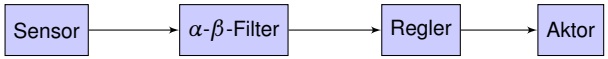


Aufgabenstellung

- Erweiterung eurer Warteschlange für Jobverwaltung
- Ersetzen nicht verifizierbarer Funktionen
  - Astrée kann kein `malloc`
- Sensorstubs implementieren
- Einfaches Filter implementieren
- System erstellen:
  - Sensoren und Filter initialisieren
  - Arbeitsaufträge erstellen
  - Arbeitsaufträge in Warteschlange einfügen
  - Arbeitsaufträge nach Priorität ausführen
- Korrektheit der Implementierung nachweisen

  ⇝ Astrée

Wiederholung: α-β-Filter [3]

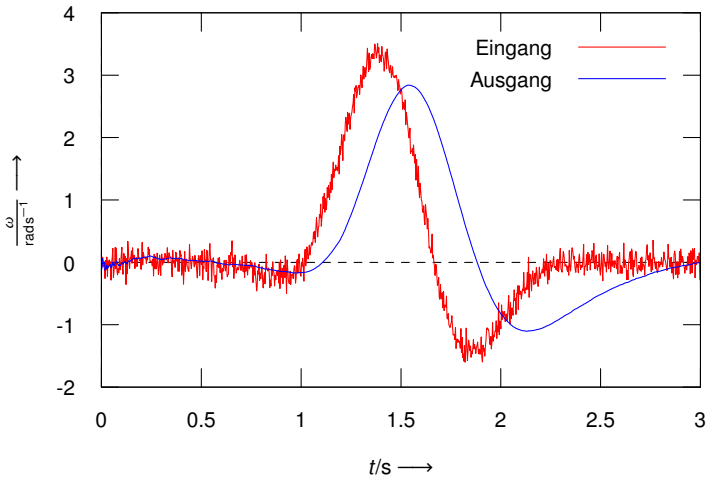


- Rauschunterdrückungsfilter
- Geeignet zur Schätzung von physikalischen Größen
  - mit Ableitung $\neq 0$
  - z. B. Position eines Flugzeugs, Lagewinkel . . .
  - im I4Copter
    - liefern Sensoren häufiger Werte als diese später verarbeitet werden

    ⇝ α-β-Filter für *Ratenwandlung* verwendet

    ⇝ 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 1 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

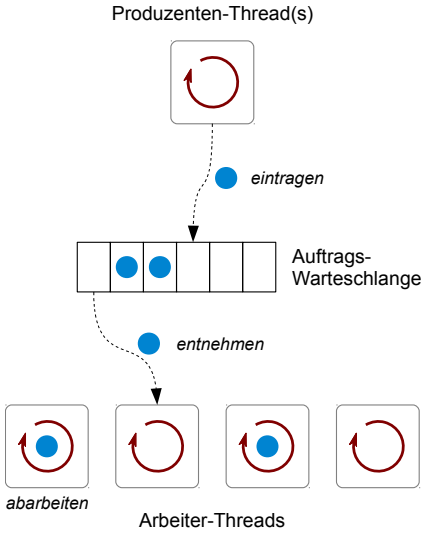
- Erfahrungen mit dem I4Copter haben gezeigt, dass sich die Parameter in folgenden Bereichen bewegen:

  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!

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



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
- ☞ Nicht-blockierende Implementierung problemlos

Überblick

Funktionszeiger

- Prioritätswarteschlange soll nun Jobs verwalten
- Nicht mehr Ganzzahlen, sondern Arbeitsaufträge
- C erlaubt es, Zeiger auf Funktionen anzulegen

```
1 typedef void(*Job)(void);
```

- Zeiger auf eine Funktion, die
  - keinen Rückgabewert hat
  - keine Parameter übernimmt

Verwendung

```
1 void job_function(void) { ...; }
2 void f_pointer_test(Job job) { job(); }
3
4 int main(void) {
5   f_pointer_test(job_function);
6   return 0;
7 }
```

Bitoperationen

- Einfacher Speicherallokator soll implementiert werden
- Freier Speicher soll durch eine Bitmaske verwaltet werden

⇝ notwendige Operationen

```
1  void set_bit(unsigned int *x, unsigned int bit_number) {
2    *x = *x | (1U << bit_number);
3  }
4
5  void reset_bit(unsigned int *x, unsigned int bit_number) {
6    *x = *x & ~(1U << bit_number);
7  }
8
9  bool is_set(unsigned int x, unsigned int bit_number) {
10   return 1U == ((x >> bit_number) & 1U);
11 }
```

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.


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.


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