

Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Abstrakte Interpretation

Phillip Raffeck, Florian Schmaus, Simon Schuster

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

Wintersemester 2020

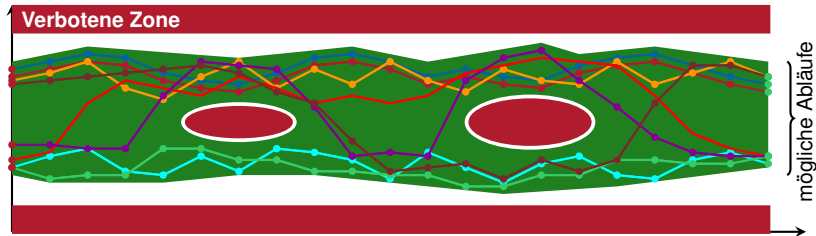


- Ziel: *bewiesenermaßen* korrektes Verhalten von Programmen
 - bisher: nichtfunktionale Eigenschaften
 - jetzt: semantische, *funktionale* Eigenschaften
- Werkzeuge: Astrée & Frama-C
- Grundvoraussetzung: Regeln zum logischen Schließen
 - In unserer Konfiguration (an sich mächtiger):
 - Astrée: Abstrakte Interpretation über der Intervallsemantik
 - Frama-C: wp-Kalkül
- Vorgehen: grobe *Beweisidee* zum Erreichen des Ziels
 - Astrée: Kurvendiskussion, Funktionalanalysis
 - Frama-C: Induktion über Datenstrukturen



- Wunsch: komplett automatisiertes Beweisen
 - *Gödelsches Unvollständigkeitstheorem* (1931)
„Jedes hinreichend mächtige formale System ist entweder widersprüchlich oder unvollständig.“
 - *Satz von Rice* (1953)
Es ist nicht möglich, eine beliebige nicht-triviale Eigenschaft der erzeugten Funktion einer Turing-Maschine algorithmisch zu entscheiden.
 - siehe einführende Erklärung zur Unentscheidbarkeit des Halteproblems [2]
 - Teilweise enormer menschlicher Aufwand nötig
 - Werkzeugunterstützung trotzdem sinnvoll
- Astrée & Frama-C





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

- Ziel: Nachweis der Abwesenheit von Laufzeitfehlern
- Findet *alle* potentiellen Laufzeitfehler
- Leider auch *falsch-positive*
→ **Gödelsches Unvollständigkeitstheorem**

Programm ist korrekt, wenn

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



- Überschreitung von Array-Grenzen
- Ganzzahldivision durch Null
- Ungültige Dereferenzierung, arithmetische Überläufe
- Ungültige Gleitkommaoperationen
- Unerreichbarer Code
- Lesezugriff auf nicht initialisierte Variablen
- Verletzung benutzerdefinierter Zusicherungen
 $\rightsquigarrow$ `assert()`



- 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



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]))  $\leadsto$  Wertebereich`

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

- Analyzer erzeugt Alarm, falls B *nicht immer wahr ist*
 - Analyzer nimmt danach an, dass B gilt
 - B kann nicht von der Form `e1 ? e2 : e3` sein
 - `__ASTREE_global_assert(( ))  $\leadsto$  gesamtes Programm`
 - `__ASTREE_check((B))  $\leadsto$  keine Annahme über B danach`
-
- B muss seiteneffektfrei sein
 - *Doppelte Klammerung ist wichtig!*



```
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 }
```



`__ASTREE_modify((V1, ..., Vn[;effect]))`

- Modelliert Veränderung der Variablen V1 bis Vn

~> Braucht man um Stubs zu bauen

- Beispiele
 - Emulation von Sensoren
 - Beschreibung des Verhaltens von Bibliotheksfunktionen
- Kein *effect* ~> kompletter Wertebereich (inklusive Nan, +/-Inf)

Beispiel

```
1 #ifdef __ASTREE__
2 __ASTREE_modify((x; full_range)); // alles außer NaN, +/-Inf
3 __ASTREE_modify((x; [10, 20])) // Einschränkung auf Intervall
4 #else
5 // ... Implementierung
6 #endif
```



Schleifen ausrollen

- Astrée beschreibt abstrakte Semantik
- Frage: Wie viele Schleifendurchgänge betrachten?
- ~> Astrée versucht Aufwand zu vermeiden
- ~> Schleifen werden (standardmäßig) einmal 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)
- ~> 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_last();
```

- Auch für Schleifen, `switch` und Funktionsaufrufe
- `__ASTREE_partition_merge` verschmilzt *alle* Partitionen
- Blick ins Handbuch: es gibt noch weitere Tricks



`__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



- Viele Echtzeitsysteme endlosschleifenbasiert
 - Allerdings durch andere Umstände begrenzt
 - 1kHz Ausführungsfrequenz, Neustart nach 7 Tagen $\leadsto$ 604,800,000 Ticks
 - `__ASTREE_max_clock((N))` legt Obergrenze für (virtuelle) Clock
 - `__ASTREE_wait_for_clock()` wartet auf nächsten Clock-Tick
- ```
1 __ASTREE_max_clock((10)); // maximale Anzahl Clock-Ticks
2 void main(void) {
3 int state_log = 0;
4 while (1) {
5 state_log = increment_state(state_log);
6 __ASTREE_wait_for_clock(); // auf naechsten Clock-Tick warten
7 }
8 }
```

⇒ Wert von `state_log` begrenzt



- 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 }
```



```
__ASTREE_analysis_log()
```

- Gibt Zustand der Analyse an dieser Stelle aus

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

- Zeigt Zustand der Analyse in Bezug auf einzelne Variablen an

```
__ASTREE_print("text")
```

- Gibt Text aus



# Analyse untersuchen

The screenshot displays the Astrée IDE interface. The top menu bar includes Project, Analysis, Editors, Edit, Tools, and Help. Below the menu is a toolbar with icons for file operations and execution. The left sidebar contains a 'Welcome' section, 'Configuration' (Preprocessor, Parser, Analyzer, Annotations), 'Results' (Overview, Call graph, Reports), and 'Files' (Preprocessed, Original). The 'Files' section shows a tree view with files like 'vezs19.cfg', 'ab\_filter.c', 'bndbuf.c', 'clib.c', 'main.c \*', 'main', and 'sensor.c'. The main window is split into two panes. The left pane shows the 'Analyzed file: /...p/07\_Astree/src/main.c \*' with line numbers 346 to 358. The right pane shows the 'Original source: ...07\_Astree/src/main.c' with line numbers 1 to 12. The bottom pane shows the analysis results for 'Line 357, column 1', including warnings about domains, pointers, and guard domains, and a summary of the execution of the 'main' function.

**Analyzed file: /...p/07\_Astree/src/main.c \***

```
346
347
348 int main(void) {
349 int i = 0;
350 _ASTREE_modify((i; [1,20]));
351 _ASTREE_log_vars((i));
352 i += 20;
353 _ASTREE_log_vars((i));
354 _ASTREE_assert((i > 10 && i < 100));
355
356
357 /**
358
```

**Original source: ...07\_Astree/src/main.c**

```
1 #include "ab_filter.h"
2 #include "sensor.h"
3 /**
4 * Sonstige benoetigten Header-Dateien
5 */
6 #include <stdio.h>
7 #include <assert.h>
8 #include "astree.h"
9
10 //#define EXTENDED
11 #include "bndbuf.h"
12
```

**Line 357, column 1**

Errors Alarms Go to section... Errors, alarms, notes, and info

**/\* Domains: Pointers, and Guard domain, and Packed (Octagons), and High\_passband\_domain(10), and Sec**

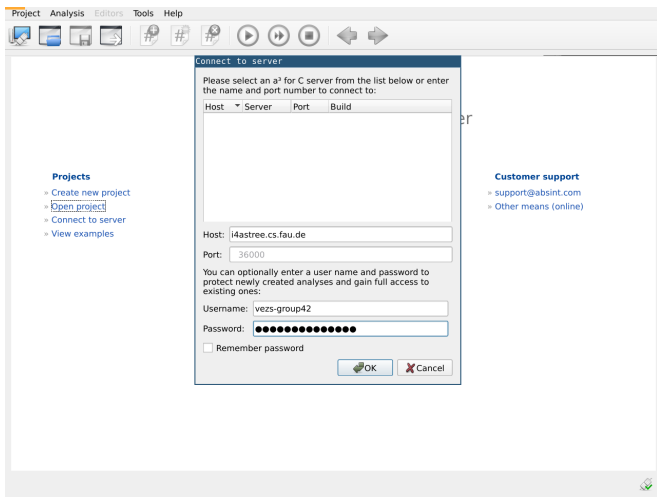
**No ambiguity due to side effects in expressions**

**/\* Executing <main> \*/**

```
[log:
call#main@348:
direct =
<init: i : initialized >
<integers (intv+cong+bitfield+set): i in [1, 20] /\ != 0 >
Equivalence Class:i:{i;}
i does not depend on itself
at main.c:351.1-24]
[log:
call#main@348:
direct =
<init: i : initialized >
<integers (intv+cong+bitfield+set): i in [21, 40] /\ != 0 >
Equivalence Class:i:{i;}
i does not depend on itself
|i| <= 1.*(20. + clock *0.) + 20. <= 72000040.
at main.c:353.1-24]
[call#main@348 at main.c:348.0-396.1
```

- Astrée im CIP:  
`% /proj/i4ezs/tools/astree_c/bin/a3c`
- Anmeldung mit Benutzername und Passwort  
→ Passwort wird bei der ersten Anmeldung festgelegt
- Dokumentation
  - PDF: `/proj/i4ezs/tools/astree_c/share/a3_c/help/a3c.pdf`





Host i4astree.cs.fau.de

Benutzer vezs-groupXX

Port 36000

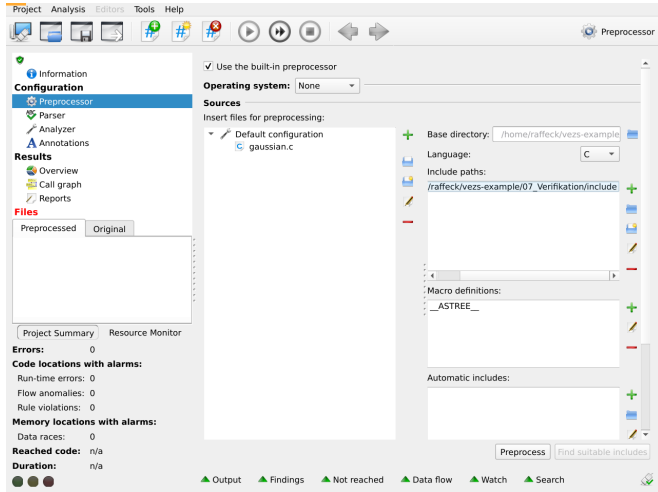
Passwort Beliebig wählbar





- 

# Quelldateien konfigurieren



## Quelltextdateien und Includepfade definieren

# Analyse starten

Start analysis

Start analysis & Preprocess

The screenshot shows the Astrée analysis interface. The toolbar at the top contains several icons. Two icons are highlighted with red boxes and red arrows pointing to them from the text labels above. The first icon, labeled 'Start analysis', is a play button. The second icon, labeled 'Start analysis & Preprocess', is a play button with a square in the center. The interface also shows a sidebar with 'Configuration' (Preprocessor, Parser, Analyzer, Annotations) and 'Results' (Overview, Call graph, Reports) sections. The 'Overview' section is selected, showing a pie chart titled 'Alarms (21 findings)'. The main window displays the 'Findings/C' tab, showing a table with 'Count' and 'Name' columns, and a value of '21 Alarms'. The bottom panel shows the 'Output' tab with a log of the analysis process, including the start of postprocessing, successful termination, and a result summary.

Project Analysis Editors Tools Help

Findings/C Findings/F Findings/Classification Rule violations Reachability Metrics Data file

Count Name  
21 Alarms

Alarms (21 findings)

\*\*\* Starting postprocessing...  
\*\*\*  
\*\*\* Postprocessing project with id 17 revision 1 terminated successfully on 2020/06/30 a  
\*\*\*  
\*\*\* Analysis used not more than 380 MB (RSS 195 MB) of memory (total analysis time 0:28 m  
\*\*\*  
/\* Result summary \*/  
Errors: 0  
Code locations with alarms  
Run-time errors: 19  
Flow anomalies: 0  
Rule violations: 1  
Memory locations with alarms  
Data races: 0  
Reached code: 5 % (98/1944 statements reached, 84/98 (85 %) statements p  
Duration: 28 s (28s)

Output Findings Not reached Data flow Watch Search



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



Rolf Wanka.  
Sachen gibt's, die gibt's gar nicht.

