

Verlässliche Echtzeitsysteme

Grundlagen der statischen Programmanalyse

Peter Ulbrich

Lehrstuhl für Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

<https://www4.cs.fau.de>

09. Juni 2016



- Bislang: Testen von Programmen
 - Konzepte, Verfahren, Metriken (s. Kapitel VIII)
 - ⚠ **Dynamische Codeanalyse** (Testen) **meist unzureichend!**

Program testing can be used to show the presence of bugs, but never to show their absence.

– Dijkstra, 1972

- ☞ Stichhaltige **Verifikation** funktionaler/nicht-funktionaler Eigenschaften?
 - Automatische **Extraktion** von (semantischen) Programmeigenschaften
 - Algorithmische **Analyse** der Programmsemantik
- Heute: **Statische Codeanalyse**





Allgemeine Fragestellung

- Terminiert die Programmausführung?
- Wie ist der Speicherverbrauch der Anwendung?
- Was sind die möglichen Ergebnisse der Ausführung?



Implementierungsspezifische Fragen

- **Generell:** Undefiniertes Verhalten in C/C++?
 - Fehlerhafte Zugriffe, Überschreitung von Array-Grenzen, hängende Zeiger, ...
 - Ausnahmen durch Division durch 0, Gleitkommaoperation-Fehler, ...
 - Typumwandlung, Ganzzahlüberlauf, ...
- Ausgang fallabhängig vorhersagbar oder ungewiss



Theoretische Betrachtung: Satz von Rice, 1953 [8]

- Eine beliebige nicht-triviale Eigenschaft eines Programms (einer Turing-vollständigen Sprache, [9]) ist algorithmisch unmöglich zu entscheiden
- Beispiel: $x = 17$; `if (TM(j)) x = 18;`, ist x konstant?

→ **Alle interessanten Fragen lassen sich nicht (exakt) beantworten!**





Approximative Beantwortung der Fragen

- Lösung praktischer Verifikationsprobleme ist **möglich**
- Ist ein Programm unter bestimmten Gesichtspunkten/Annahmen fehlerfrei?
- Neue Frage: Wie sicher ist die Abschätzung?

■ Vom dynamischen Testen zur statischen Analyse

- Automatische Extraktion von Programmeigenschaften
- Analysemethodik unter Zuhilfenahme von Approximationen

■ Was sind die **Grundlagen** abstrakter Interpretation?

- Betrachtung der **abstrakten Programmsemantik**
- Vereinfachung des entstehenden Zustandsraums
- Eine „informelle“ Sichtweise auf die Zusammenhänge



Grobes Verständnis für abstrakter Interpretation entwickeln!



- 1 Vom Testen zur Verifikation
 - Der Compiler als Analysewerkzeug
 - Der Heartbleed-Bug
 - Fehlersuche durch Instrumentierung
 - Fehlersuche durch statische Codeanalyse
 - Verfahren in der Übersicht
- 2 Abstraktion der Programmsemantik
 - Konkrete Programmsemantik
 - Sicherheitseigenschaften
 - Abstrakte Programmsemantik
- 3 Analyse & Vereinfachung
 - Sammelsemantiken
 - Präfixsemantiken
- 4 Zusammenfassung



```
1 unsigned short x;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Ausgabe des Übersetzers:

```
bash: gcc -Wall example.c  
warning: variable 'x' is uninitialized when used here  
while (x < 10000) {  
    ^
```



Der **Übersetzer** (engl. *compiler*) ist ein statisches Analyse-Werkzeug

- Neben der syntaktischen erfolgt hier auch eine semantische Prüfung
- Verschiedenste Analysen (Daten-, Kontrollfluss)
- Ausgabe als **Warnung** oder **Fehler**

→ Deckt (vorrangig) Fehler im definierten Verhalten auf



Der Übersetzer ist die **erste Verteidigungslinie**

- Es sollten immer alle Prüfungen aktiv sein (insbesondere -Wall)

⚠ Keine hinreichende Verifikation (KEIN: **it compiles, let's ship it!**)



Catastrophic is the right word. On the scale of 1 to 10, this is an 11.

– Bruce Schneier



Katastrophaler Fehler in OpenSSL 1.0.1 – 1.0.1f

- Erweiterung zur **periodischen Überwachung** (engl. *heartbeat*) in (D)TLS
- Eine beliebiger String (16 Bit Länge) dient als **Nutzlast** (engl. *payload*)
- Dieser wird von der Gegenstelle unverändert zurückgesendet



Ein Abgleich von **angegebener** und **tatsächlicher Länge** fand nicht statt



Folgen der fehlerhaften Implementierung

- Bei bekanntwerden waren ca. 17 % aller SSL-Dienste anfällig!
- Angreifer konnten wiederholt 64 KiB Speicher auslesen
- Inhalt: Zufällige Daten (Private Daten, Passwörter, Schlüssel, ...)

→ **All diese Daten gelten als kompromittiert!**



- ☞ **Address Sanitizer**¹ Plugin für gcc und Clang
 - Standard in aktuellen Versionen (-fsanitize=address)
 - Konstruktiver Ansatz ↪ Prüfungen zur Laufzeit

■ Strategie

- Identifikation bössartiger Operationen und Zugriffe
- Speicherzugriffe (Verwendung nach Freigabe), Ganzzahlüberläufe, ...

■ Umsetzung

- Instrumentierung des Programmcodes (Code und Daten) ↪ `assert()`
 - Benutzerspezifische Funktionen zur Behandlung (engl. *error hooks*)

→ Hohe Laufzeitkosten von ca. 200 %

⚠ Dies ist ein **fallspezifischer** und zudem **unsicherer** Ansatz!

¹<http://clang.lvm.org/docs/AddressSanitizer.html>

Clang-Analyzer-Plugin zur Aufdeckung des Heartbleed-Bugs

- Analyse-Durchlauf (engl. *analysis pass*) innerhalb von clang

■ Strategie

- Identifikation bössartiger Daten (Angreifer) ist das Problem
- Idee: Nur Netzwerkdaten werden beim Speicherzugriff zur Gefahr
- Lösung: Färbung von Datenfüßen `ntohl()²` → `memcpy()`

→ **Alarm** bei Absenz von weiteren **Plausibilitätsprüfungen** (engl. *sanitizer*)

■ Umsetzung

- Clang-Analyzer nutzt **symbolische Ausführung** (engl. *symbolic execution*) [6] zur Analyse von C/C++ Programmen
 - Pfadsensitive Analyse mit einem **Zustandsobjekt** (engl. *state object*) pro Pfad
 - Das Plugin befragt diese Objekte \rightsquigarrow Mögliche Wertebereiche für die Paketlänge

→ Spezifikation und Testbedingungen für die Bewertung der Paketlänge



Dies ist ein **fallspezifischer** und zudem **unsicherer** Ansatz!

²Transformation der **Byte-Reihenfolge** (engl. *endianness*) von Netzwerk zu Host.



Clang Heartbleed-Finder

```
if(fd != -1) {
```

1 Taking true branch →

```
    int size;  
    int res;  
  
    res = read(fd, &size, sizeof(int));  
  
    if(res == sizeof(int)) {
```

2 ← Taking true branch →

```
        size = ntohl(size);  
  
        if(size < sizeof(data_array)) {
```

3 ← Taking false branch →

```
            memcpy(buf, data_array, size);  
        }
```

```
        memcpy(buf, data_array, size);
```

4 ← Tainted, unconstrained value used in memcpy size

```
    }
```

- 1 Färbung des Datenflusses
- 2 Kontextsensitive Annahmen über Codepfade
- 3 Wertebereichsüberprüfungen
- 4 Plausibilitätsprüfung



Übersicht: Verifikationsverfahren

Abstraktion

Beispiel	<i>Testfall</i>	<i>Clang Address Sanitizer</i>	<i>Clang Static Analyser</i>	<i>Astrée</i>	<i>framaC, Isabelle</i>
Ebene	µ-Code	µ-Code + Instrumentierung	Zwischensprache / AST ¹	Abstrakte Programmsemantik	Abstrakte Semantik + Kontrakt
Technik	return()	assert()	Symbolische Ausführung / Erreichbarkeitsanalyse	Abstrakte Interpretation	Abstrakte Interpretation / Verifikation
Konkrete Maschine	CPU	CPU	Compiler	Model Checker	Model Checker
		dynamisch	statisch	unsicher	sicher



Es existieren verschiedenste Verfahren zur Programmverifikation

- Harte Klassifizierung ist schwierig (vgl. Redundanzarten IV/7)

■ Statisch versus dynamisch

- Nutzung der konkreten/abstrakten Programmsemantik (siehe Folien 15 ff)
- Konkrete Ausführung (Maschine) hängt jedoch von der Betrachtungsebene ab!

■ Sicher versus unsicher

- Vollständigkeit der Analyse (sicher \mapsto 100 %, siehe Folien 20 ff)
- Steht im Bezug zu einer bestimmten Spezifikation (z.B. C-Standard bei Astrée)

¹**Abstrakter Syntaxbaum** (engl. *abstract syntax tree*): Baumförmige Repräsentation der abstrakten Syntax eines Programmes. Typischerweise im Zuge der Übersetzung durch den Übersetzer aufgebaut und zur effizienten Verarbeitung genutzt.



- 1 Vom Testen zur Verifikation
 - Der Compiler als Analysewerkzeug
 - Der Heartbleed-Bug
 - Fehlersuche durch Instrumentierung
 - Fehlersuche durch statische Codeanalyse
 - Verfahren in der Übersicht
- 2 **Abstraktion der Programmsemantik**
 - **Konkrete Programmsemantik**
 - **Sicherheitseigenschaften**
 - **Abstrakte Programmsemantik**
- 3 Analyse & Vereinfachung
 - Sammelsemantiken
 - Präfixsemantiken
- 4 Zusammenfassung



Fehlersuche: Was kann hier alles schief gehen?

Die Gretchenfrage der Softwareentwicklung ...

```
1 unsigned int average(unsigned int *array,
2                       unsigned int size)
3 {
4     unsigned int temp = 0;
5
6     for(unsigned int i = 0; i < size; i++) {
7         temp += array[i];
8     }
9
10    return temp/size;
11 }
```

■ Wo könnte es hier klemmen?

- Ist der Zugriff auf Feld array in Zeile 7 korrekt?
- Kann die Addition in Zeile 7 überlaufen?
- Kann in Zeile 10 eine Division durch 0 auftreten?

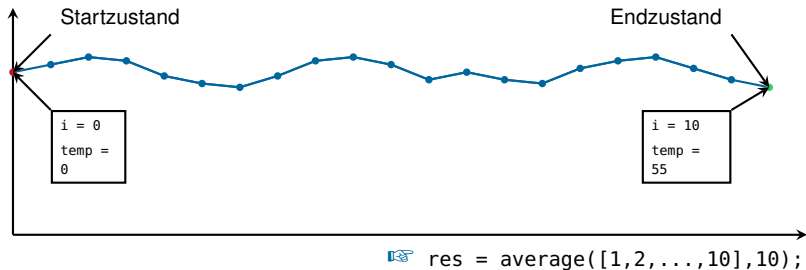
■ Wie findet man das heraus?



Schauen wir mal, wie sich das Programm verhält.



Das Verhalten zur Laufzeit ist entscheidend!



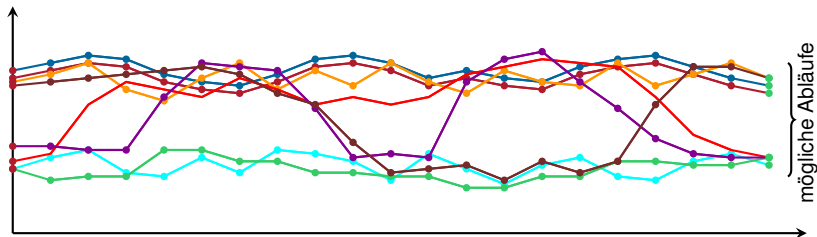
```
1 unsigned int average(uint *array,  
2                     uint size)  
3 {  
4     uint temp = 0;  
5  
6     for(uint i = 0;i < size;i++) {  
7         temp += array[i];  
8     }  
9  
10    return temp/size;  
11 }
```

i	temp
0	0
1	1
2	3
3	6
...	...
10	55



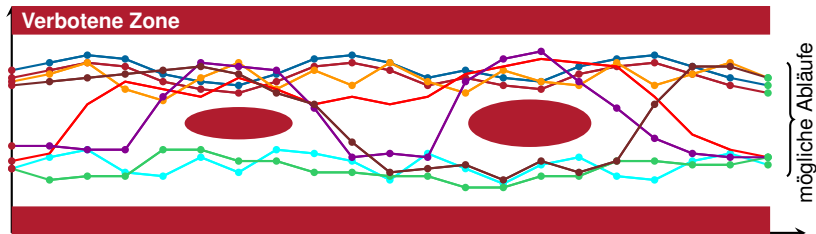
Konkrete Programmsemantik

Eine informelle Einführung in die Prinzipien abstrakter Interpretation [2]



- Die **konkrete Semantik** (engl. *concrete semantics*) beschreibt
 - Alle möglichen Ausführungen eines Programms
 - Unter allen möglichen Ausführungsbedingungen
 - Für unser Beispiel bedeutet dies:
 - 2^{32} verschieden große Felder, 2^{32} verschiedene Werte für jedes Element
- Sie beschreibt ein „unendliches“ mathematisches Objekt
 - Im Allgemeinen **nicht berechenbar** durch einen Algorithmus
 - Alle nicht-trivialen Fragestellungen sind **nicht entscheidbar**

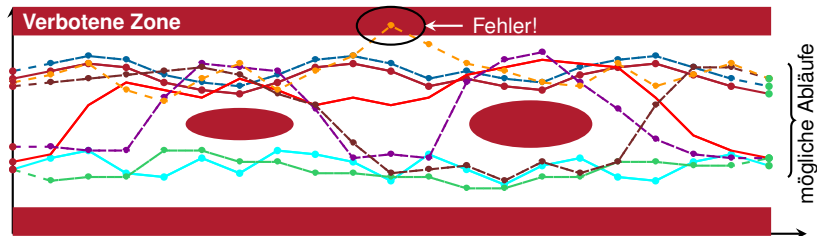




- **Sicherheitseigenschaften** (engl. *safety properties*) stellen sicher, dass keine **fehlerhaften/unerwünschten Zustände** eingenommen werden
- Ein **Sicherheitsnachweis** (engl. *safety proof*) garantiert, dass die konkrete Semantik nie eine **verbotene Zone** durchläuft
- ☞ Das ist ein **unentscheidbares Problem**
 - Die konkrete Programmsemantik ist nicht berechenbar

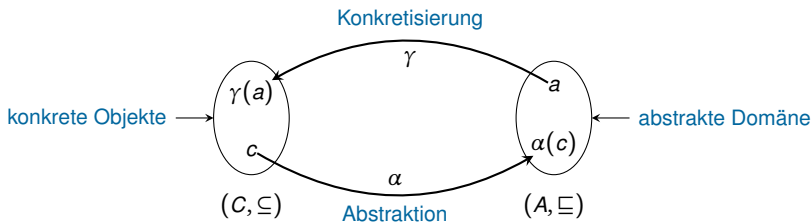


Testen: Das Problem der Möglichkeiten



- Testen betrachtet **nur eine Teilmenge** aller möglichen Ausführungen
 - ↪ Gut geeignet, um die **Existenz** von Defekten zu zeigen
 - ↪ Ungeeignet, um ihre **Abwesenheit** zu zeigen
 - Evtl. hat man die fehlerhafte Ausführung einfach nicht getestet
- Problem: **unzureichende Abdeckung** der konkreten Semantik





- Wähle eine **abstrakte Domäne** (engl. *abstract domain*)
 - Ersetzt die Menge konkreter Objekte S durch ihre Abstraktion $\alpha(S)$
 - Verschiedene Domänen unterscheiden sich hinsichtlich ihrer Präzision
 - Vorzeichen, **Intervalle**, Oktagon, Polyhedra, ...
- **Abstraktionsfunktion α** (engl. *abstraction function*)
 - Bildet die Menge konkrete Objekte auf ihre abstrakte Interpretation ab
- **Konkretisierungsfunktion γ** (engl. *concretization function*)
 - Bildet die Menge abstrakter Objekte auf konkrete Objekte ab



☞ Approximation von f durch die abstrakte Funktion f'

■ Häufig verwendet man **Galoisverbindungen** mit den Eigenschaften:

■ $(C, \sqsubseteq) \xrightleftharpoons[\alpha]{\gamma} (A, \sqsubseteq)$ und $\alpha(\gamma(a)) = a$ (Einbettung)

→ Konkretisierung gefolgt von Abstraktion impliziert keinen Präzisionsverlust

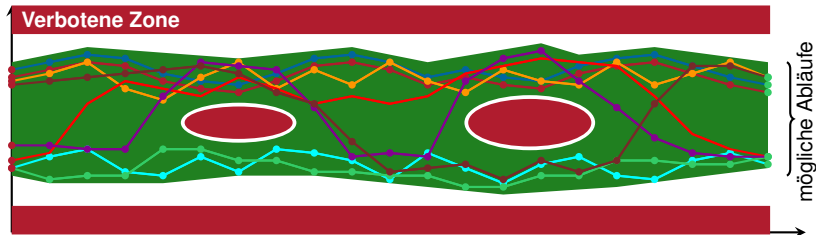
■ **Abstrakte Interpretation** nutzt diese Eigenschaften

- Statt die konkrete Funktion $f(c)$ zu berechnen
- Kann man sie annähern, indem
 - Man die abstrakte Funktion f' auf die Abstraktion $\alpha(c)$ anwendet
 - Und das Ergebnis $f'(\alpha(c))$ wieder konkretisiert

■ Beispiel: Die Einbettung der ganzen Zahlen (\mathbb{Z}) in die reellen Zahlen (\mathbb{R})

- Die abstrakte Funktion f' ist definiert als Abrundungsfunktion
- Eine ganze Zahl lässt sich ohne Präzisionsverlust konkretisieren





- **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

Formale Methoden sind abstrakte Interpretationen

Die abstrakte Semantik wird aber auf unterschiedliche Weise bestimmt

Model Checking

- Abstrakte Semantik wird explizit vom Nutzer angegeben
↳ endliche Beschreibung der konkreten Programmsemantik
 - Z.B. endliche Automaten, Aussagen- oder Prädikatenlogik
- Automatische Ableitung durch **statische Analyse**

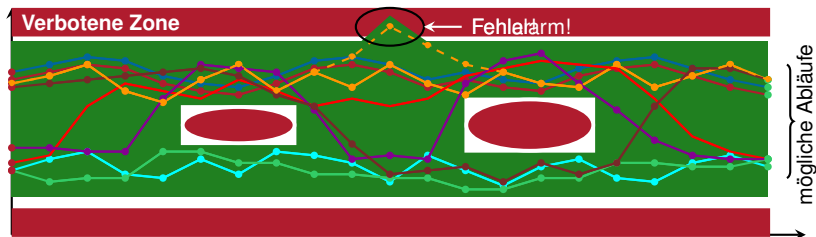
Deduktive Methoden

- Abstrakte Semantik wird durch Nachbedingungen beschrieben
- Nutzer gibt sie durch induktive Argumente an
 - Z.B. Vorbedingungen und Invarianten
- Automatische Ableitung durch **statische Analyse**

Statische Analyse

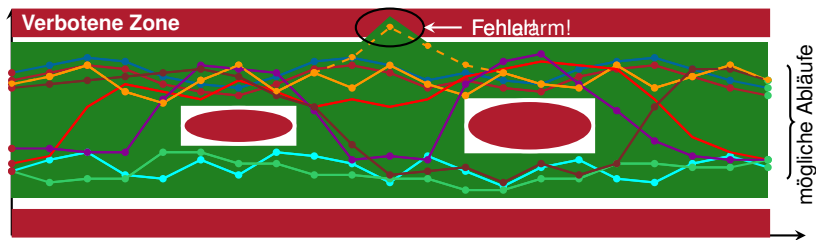
- Abstrakte Semantik wird ausgehend vom Quelltext bestimmt
 - Abbildung auf **vorab bestimmte, wohldefinierte Abstraktionen**
- Anpassungen (automatisch/durch den Nutzer) sind möglich





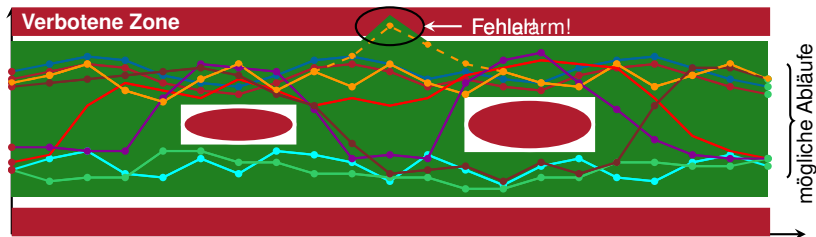
Vollständigkeit und Korrektheit: **Sicherheit** (engl. *soundness*)

- Keine potentieller Defekt darf übersehen werden
- ~> nur so kann die Abwesenheit von Defekten gezeigt werden
 - Ansonsten wäre gegenüber reinem Testen nichts gewonnen



Präzision

- Weitgehende Vermeidung von **Fehlalarmen** (engl. *false alarms*)
 - Synonyme englische Bezeichnung: **false positives**
- Ermöglicht erst eine vollkommen automatisierte Anwendung



Geringe Komplexität

- Berechnung der abstrakten Semantik in akzeptabler Laufzeit
 - Vermeidung der kombinatorischen Explosion des Zustandsraums

- 1 Vom Testen zur Verifikation
 - Der Compiler als Analysewerkzeug
 - Der Heartbleed-Bug
 - Fehlersuche durch Instrumentierung
 - Fehlersuche durch statische Codeanalyse
 - Verfahren in der Übersicht
- 2 Abstraktion der Programmsemantik
 - Konkrete Programmsemantik
 - Sicherheitseigenschaften
 - Abstrakte Programmsemantik
- 3 Analyse & Vereinfachung
 - Sammelsemantiken
 - Präfixsemantiken
- 4 Zusammenfassung



Reduktion des Zustandsraums ist unumgänglich!

 Fasse verschiedene Zustände geeignet zusammen

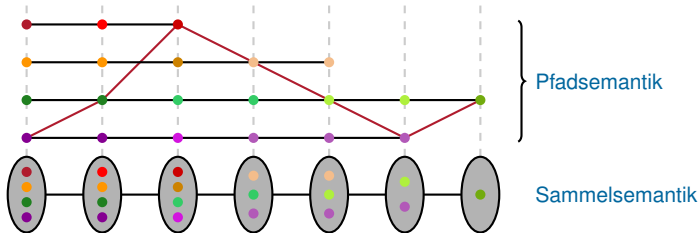
~> **Sammelsemantiken** (s. Folie 25 ff.)

 Betrachte nur den Anfang der Zustandshistorie

~> **Präfixsemantiken** (s. Folie 31 ff.)



Sammelsemantik (engl. *collecting semantics*)



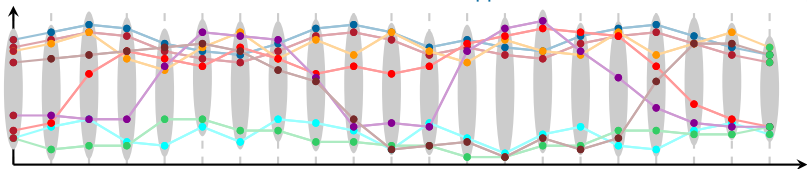
- Sammelt die Zustände aller Pfade an einem bestimmten Punkt
 - d. h. an einer bestimmten Programmanweisung
 - Aufgrund der Größe, wird sie i. d. R. approximiert
- Das ist eine **verlustbehaftete Abstraktion**
 - Beispiel: Existiert der rote Pfad?
 - Konkrete Semantik \mapsto **Nein**, Sammelsemantik \mapsto **???**

 Der **Laufzeitgewinn** wird durch **Unschärfe** erkauf!

- Das Ergebnis „**Weiß nicht ...**“ ist typisch für solche Methoden
- Und die Ursache vieler Vorbehalte ...

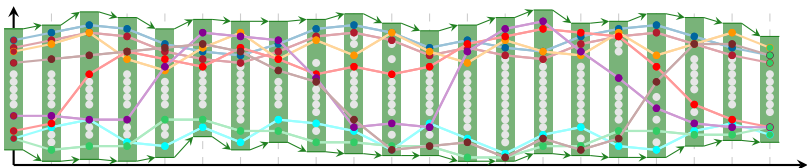


Als Approximation der Sammelsemantik [7]



- Die Sammelsemantik verwaltet Zustandsmengen
- ☞ die Intervallabstraktion nur ihre oberen und unteren Schranken
 - Die zu verwaltenden Daten werden dadurch beträchtliche reduziert
 - Allerdings wird auch die Präzision reduziert

~> bestimmte Zustände im approximierten Zustandsraum werden nicht erreicht



Beispiel: Intervallabstraktion für ein C-Programm

```
1 unsigned short x = 1;
2
3 while(x < 10000) {
4     x = x + 1;
5 }
6
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_5) \cap [10000, \infty]$

- Die Intervallabstraktion ist eine **manuell vorgegebene, abstrakte Interpretation** der Semantik der Programmiersprache C
 - C-Programme werden dann **automatisiert darauf abgebildet**
 - z. B. durch einen Übersetzer oder ein statisches Analysewerkzeug
 - Nur Elemente, die den Wertebereich von x betreffen, sind relevant
- Dies ist bereits eine **starke Vereinfachung**
 - Angenommen x wäre eingangs nicht bekannt
 - ~ es gäbe 10000 verschiedene Pfade durch den Zustandsraum
 - Nehme eine Schleifenobergrenze `unsigned short` y statt 10000 an
 - ~ es gäbe $\leq (2^{16})^2$ verschiedene Pfade durch den Zustandsraum



```
1 unsigned short x = 1;
2
3 while(x < 10000) {
4     x = x + 1;
5 }
6
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

- Approximation durch **chaotische Iteration** (engl. *chaotic iteration*)

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$

Iteration 2:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 2]$

Zeile 4 $x_4 = [2, 3]$

Zeile 7 $x_7 = \emptyset$



```
1 unsigned short x = 1;
2
3 while(x < 10000) {
4     x = x + 1;
5 }
6
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

- Approximation durch **chaotische Iteration** (engl. *chaotic iteration*)

Iteration 3:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 3]$

Zeile 4 $x_4 = [2, 4]$

Zeile 7 $x_7 = \emptyset$

viele, viele Iterationen später:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 9999]$

Zeile 4 $x_4 = [2, 10000]$

Zeile 7 $x_7 = [10000, 10000]$



```
1 unsigned short x = 1;
2
3 while(x < 10000) {
4     x = x + 1;
5 }
6
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \nabla x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \nabla x_4) \cap [10000, \infty]$

■ Approximation mit Hilfe eines Widening-Operators [7]

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$

Iteration 2:

Zeile 1 $x_1 = [1, 1]$

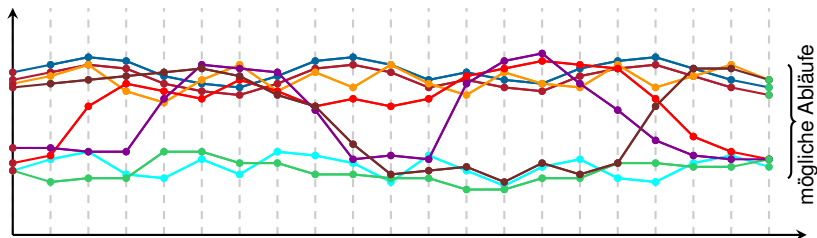
Zeile 3 $x_3 = [1, 9999]$

Zeile 4 $x_4 = [2, 10000]$

Zeile 7 $x_7 = [10000, 10000]$

■ Konvergenz in der 2. Iteration



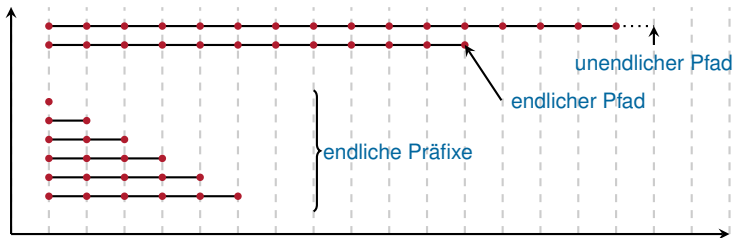


- Betrachte durch ein **Transitionssystem** beschriebene **Programmpfade**
 - Ausgehend von ausgezeichneten Startzuständen,
 - Beschreiben sie eine (unendliche) Abfolge von **Programmzuständen**,
 - Deren Reihenfolge durch die Übergangsrelation bestimmt wird.

~ die Gesamtheit dieser Programmpfade heißt **Pfadsemantik**

 - Wie die konkrete Programmsemantik ist sie **nicht berechenbar**.
- Reduktion der Komplexität durch **Abstraktion**
 - Unendliche Pfade \rightsquigarrow (endliche) **Pfadpräfixe**





- Pfadsemantiken enthalten alle endlichen und unendlichen Pfade
 - Pfadpräfixe enthalten nur die Anfänge dieser Pfade



das ist eine **verlustbehaftete Abstraktion**

- Beispiel: betrachte Worte der Sprache $a^n b$
 - Frage: Gibt es Worte mit unendlich vielen aufeinanderfolgenden 'a'?
 - Pfadsemantik: $\{a^n b | n \geq 0\} \mapsto$ **Nein**
 - Pfadpräfixe: $\{a^n | n \geq 0\} \cup \{a^n b | n \geq 0\} \mapsto$ **???**



- Menge der Prafixe ist rekursiv:

$$\begin{aligned} \text{Prafixe} = & \{x \mid x \text{ ist Startzustand}\} \cup \\ & \{x_1 \rightarrow^* x_2 \rightarrow x_3 \mid x_1 \rightarrow^* x_2 \in \text{Prafixe} \wedge x_2 \rightarrow x_3 \in \rightarrow\} \end{aligned}$$

- Zu losen ist die Fixpunktiteration $\text{Prafixe} = F(\text{Prafixe})$
 - ublicherweise besitzt diese Gleichung mehrere Losungen
 - ~> ordne die Losungen nach der **Teilmengenbeziehung** \subseteq
 - ~> wahle die kleinste Teilmenge als Losung
 - ~> **least fixpoint prefix trace semantics**
- Vereinfachungen ermoglichen **effektive, iterative Analysealgorithmen**
 - Vereinfachung im Sinne von Abstraktion bzw. Approximation
 - ~> man muss nur noch die Prafixe betrachten
 - Nicht mehr die vollstandigen (evtl. unendlichen) Pfade



- 1 Vom Testen zur Verifikation
 - Der Compiler als Analysewerkzeug
 - Der Heartbleed-Bug
 - Fehlersuche durch Instrumentierung
 - Fehlersuche durch statische Codeanalyse
 - Verfahren in der Übersicht
- 2 Abstraktion der Programmsemantik
 - Konkrete Programmsemantik
 - Sicherheitseigenschaften
 - Abstrakte Programmsemantik
- 3 Analyse & Vereinfachung
 - Sammelsemantiken
 - Präfixsemantiken
- 4 Zusammenfassung



Vom Test zur Verifikation

- Statische Codeanalyse erlaubt die Extraktion der Programmsemantik
- Verschiedene Abstufungen von Verifikationstechniken

Konkrete Programmsemantik ist nicht berechenbar

- Approximation durch eine **abstrakte Semantik**
 - Korrektheit der Approximation ist entscheidend
 - Nur so kann man einen **Sicherheitsnachweis** führen
 - Die Approximation muss präzise sein
 - Nur so kann man **Fehlalarme** vermeiden
 - Die Approximation darf nicht zu komplex sein
 - Nur so kann sie **effizient berechnet** werden

Transitionssystem beschreiben Programme

- **Pfadsemantiken** beschreiben die konkrete Programmsemantik
- Approximation durch **Pfadpräfixe** und **Sammelsemantik**
 - ↪ abstrakte Interpretation approximiert die Sammelsemantik



- [1] Cousot, P. :
Semantic foundations of program analysis.
In: *Program flow analysis: theory and applications* 10 (1981), S. 303–342

- [2] Cousot, P. :
Abstract Interpretation.
<http://web.mit.edu/16.399/www/>, 2005

- [3] Cousot, P. ; Cousot, R. :
Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.
In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*.
New York, NY, USA : ACM, 1977 (POPL '77), S. 238–252

- [4] Cousot, P. ; Cousot, R. :
Abstract interpretation frameworks.
In: *Journal of Logic and Computation* 2 (1992), Nr. 4, S. 511–547



- [5] Cousot, P. ; Cousot, R. :
Abstract Interpretation and Application to Logic Programs.
In: *Journal of Logic Programming* 13 (1992), Jul., Nr. 2-3, S. 103–179.
[http://dx.doi.org/10.1016/0743-1066\(92\)90030-7](http://dx.doi.org/10.1016/0743-1066(92)90030-7). –
DOI 10.1016/0743–1066(92)90030–7. –
ISSN 0743–1066
- [6] King, J. C.:
Symbolic execution and program testing.
In: *Communications of the ACM* 19 (1976), Nr. 7, S. 385–394
- [7] Midtgaard, J. :
Abstract Interpretation.
<http://www.cs.au.dk/~jmi/AbsInt/>, 2012
- [8] Rice, H. G.:
Classes of recursively enumerable sets and their decision problems.
In: *Transactions of the American Mathematical Society* 74 (1953), Nr. 2, S. 358–366
- [9] Turing, A. M.:
On computable numbers, with an application to the Entscheidungsproblem.
In: *Journal of Math* 58 (1936), Nr. 345-363, S. 5

