Verlässliche Echtzeitsysteme

Grundlagen der statischen Programmanalyse

Peter Ulbrich

Lehrstuhl für Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität Erlangen-Nürnberg

`https://www4.cs.fau.de`

KW25 2020

Wiederholung

- 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

Wiederholung

- 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

Wiederholung

- 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

Problem: Programmsemantik

Im Fokus: Undefiniertes Verhalten durch Laufzeitfehler

☞ **Allgemeine Fragestellung**

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

Problem: Programmsemantik

Im Fokus: Undefiniertes Verhalten durch Laufzeitfehler

☞ 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

Problem: Programmsemantik

Im Fokus: Undefiniertes Verhalten durch Laufzeitfehler

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 [9]

- Eine beliebige nicht-triviale Eigenschaft eines Programms (einer Turing-vollständigen Sprache, [10]) ist algorithmisch unmöglich zu entscheiden
- Beispiel: `x = 17; if (TM(j)) x = 18;`, ist `x` konstant?

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

Übersicht über die Vorlesung

☞ 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!

Gliederung

Statische Analyse durch den Übersetzer

Der nette Helfer aus der Nachbarschaft

```c
unsigned short x;

while(x < 10000) {
  x = x + 1;
}

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

Statische Analyse durch den Übersetzer

Der nette Helfer aus der Nachbarschaft

```c
unsigned short x;

while(x < 10000) {
  x = x + 1;
}

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)!

Beispiel: Der Heartbleed-Bug

> *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

Beispiel: Der Heartbleed-Bug

> *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!

Fehlersuche mittels Code-Instrumentierung

☞ Address Sanitizer[1] Plugin für gcc und Clang

- Standard in aktuellen Versionen (`-fsanitize=address`)
- Konstruktiver Ansatz ↦ Prüfungen zur Laufzeit

[1] http://clang.llvm.org/docs/AddressSanitizer.html

Fehlersuche mittels Code-Instrumentierung

☞ Address Sanitizer[1] Plugin für gcc und Clang

- Standard in aktuellen Versionen (`-fsanitize=address`)
- Konstruktiver Ansatz $\mapsto$ Prüfungen zur Laufzeit

- Strategie
  - Identifikation bösartiger Operationen und Zugriffe
  - Speicherzugriffe (Verwendung nach Freigabe), Ganzzahlüberläufe, ...
- Umsetzung
  - Instrumentierung des Programmcodes (Code und Daten) $\mapsto$ `assert()`
    - Benutzerspezifische Funktionen zur Behandlung (engl. *error hooks*)

→ Hohe Laufzeitkosten von ca. 170 %

[1] http://clang.llvm.org/docs/AddressSanitizer.html

Fehlersuche mittels Code-Instrumentierung

☞ Address Sanitizer[1] Plugin für gcc und Clang

- Standard in aktuellen Versionen (`-fsanitize=address`)
- Konstruktiver Ansatz $\mapsto$ Prüfungen zur Laufzeit

- Strategie
  - Identifikation bösartiger Operationen und Zugriffe
  - Speicherzugriffe (Verwendung nach Freigabe), Ganzzahlüberläufe, ...
- Umsetzung
  - Instrumentierung des Programmcodes (Code und Daten) $\mapsto$ `assert()`
    - Benutzerspezifische Funktionen zur Behandlung (engl. *error hooks*)

→ Hohe Laufzeitkosten von ca. 170 %

⚠ Dies ist ein fallspezifischer und zudem unsicherer Ansatz!

[1] http://clang.llvm.org/docs/AddressSanitizer.html

Fehlersuche mittels statischer Codeanalyse

☞ Clang-Analyzer-Plugin zur Aufdeckung des Heartbleed-Bugs

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

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

Fehlersuche mittels statischer Codeanalyse

☞ Clang-Analyzer-Plugin zur Aufdeckung des Heartbleed-Bugs

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

- Strategie
  - Identifikation bösartiger Daten (Angreifer) ist das Problem
  - Idee: Nur Netzwerkdaten werden beim Speicherzugriff zur Gefahr
  - Lösung: Färbung von Datenfüssen `ntohl()`[2] $\longrightarrow$ `memcopy()`

  → 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 $\leadsto$ Mögliche Wertebereiche für die Paketlänge

  → Spezifikation und Testbedingungen für die Bewertung der Paketlänge

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

Fehlersuche mittels statischer Codeanalyse

☞ Clang-Analyzer-Plugin zur Aufdeckung des Heartbleed-Bugs

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

- Strategie
  - Identifikation bösartiger Daten (Angreifer) ist das Problem
  - Idee: Nur Netzwerkdaten werden beim Speicherzugriff zur Gefahr
  - Lösung: Färbung von Datenfüssen `ntohl()`[2] ⟶ `memcopy()`
  - → 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 ⇝ Mögliche Wertebereiche für die Paketlänge
  - → Spezifikation und Testbedingungen für die Bewertung der Paketlänge

⚠ Dies ist ein fallspezifischer und zudem unsicherer Ansatz!

[2]Transformation 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	*Testfall*	*Clang Address Sanitizer*	*Clang Static Analyser*	*Astrée*	*framaC, Isabelle*
Ebene	µ-Code	µ-Code + Instrumentierung	Zwischensprache / AST	Abstrakte Programmsemantik	Abstrakte Semantik + Kontrakt
Technik	`return()`	`assert()`	*Symbolische Ausführung / Ereichbarkeitsanalyse*	*Abstrakte Interpretation*	*Abstrakte Interpretation / Verifkation*
Konkrete Maschine	CPU	CPU	Compiler	Analyzer	Prover

dynamisch | statisch — unsicher | sicher

- Es existieren verschiedenste Verfahren zur Programmverifikation
  - Harte Klassifizierung ist schwierig (vgl. Redundanzarten IV/8)

[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.

Übersicht: Verifikationsverfahren

Abstraktion →

Beispiel	*Testfall*	*Clang Address Sanitizer*	*Clang Static Analyser*	*Astrée*	*framaC, Isabelle*
Ebene	µ-Code	µ-Code + Instrumentierung	Zwischensprache / AST	Abstrakte Programmsemantik	Abstrakte Semantik + Kontrakt
Technik	`return()`	`assert()`	*Symbolische Ausführung / Ereichbarkeitsanalyse*	*Abstrakte Interpretation*	*Abstrakte Interpretation / Verifkation*
Konkrete Maschine	CPU	CPU	Compiler	Analyzer	Prover

dynamisch | statisch — unsicher | sicher

- ☞ Es existieren verschiedenste Verfahren zur Programmverifikation
  - Harte Klassifizierung ist schwierig (vgl. Redundanzarten IV/8)
- 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.

Gliederung

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?

Fehlersuche: Was kann hier alles schief gehen?

Die Gretchenfrage der Softwareentwicklung ...

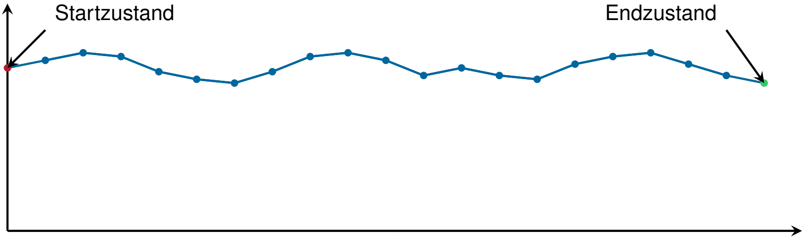
```
1  unsigned int average(unsigned int *array,
2                       unsigned int size)
3  {
4    unsigned int temp = 0;
5
6    for(unsigned int i = 0;i < size;i++) {
7      temp += array[i];
8    }
9
10   return temp/size;
11 }
```

- 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!

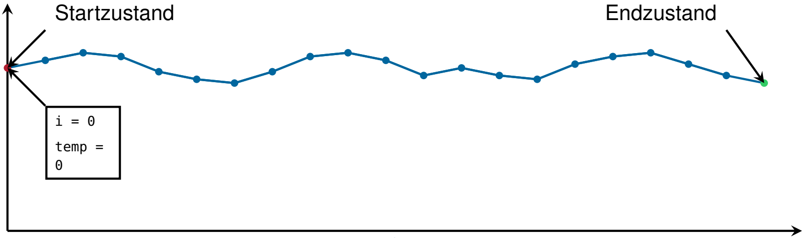


☞ `res = average([1,2,...,10],10);`

```
unsigned int average(uint *array,
                     uint size)
{
  uint temp = 0;

  for(uint i = 0;i < size;i++) {
    temp += array[i];
  }

  return temp/size;
}
```

i	temp

Das Verhalten zur Laufzeit ist entscheidend!

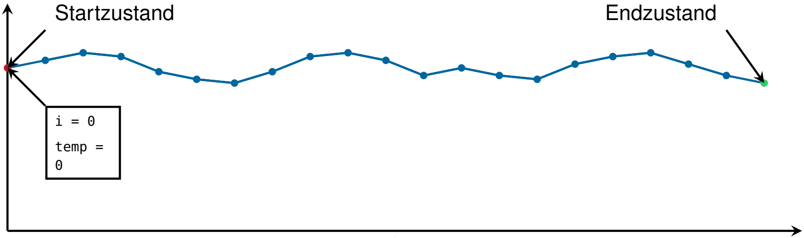


☞ `res = average([1,2,...,10],10);`

```
unsigned int average(uint *array,
                     uint size)
{
  uint temp = 0;

  for(uint i = 0;i < size;i++) {
    temp += array[i];
  }

  return temp/size;
}
```

i	temp
0	0

Das Verhalten zur Laufzeit ist entscheidend!

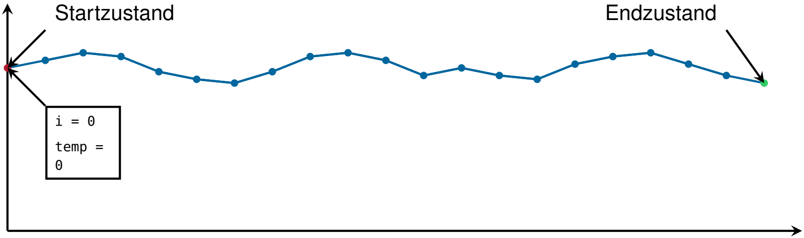


☞ `res = average([1,2,...,10],10);`

```
unsigned int average(uint *array,
                     uint size)
{
  uint temp = 0;

  for(uint i = 0;i < size;i++) {
    temp += array[i];
  }

  return temp/size;
}
```

i	temp
0	0
1	1

Das Verhalten zur Laufzeit ist entscheidend!

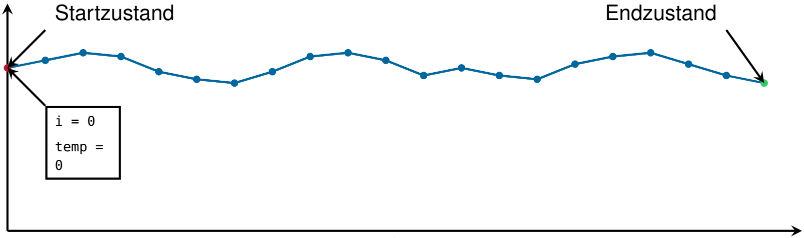


☞ `res = average([1,2,...,10],10);`

```
unsigned int average(uint *array,
                     uint size)
{
  uint temp = 0;

  for(uint i = 0;i < size;i++) {
    temp += array[i];
  }

  return temp/size;
}
```

i	temp
0	0
1	1
2	3

Das Verhalten zur Laufzeit ist entscheidend!

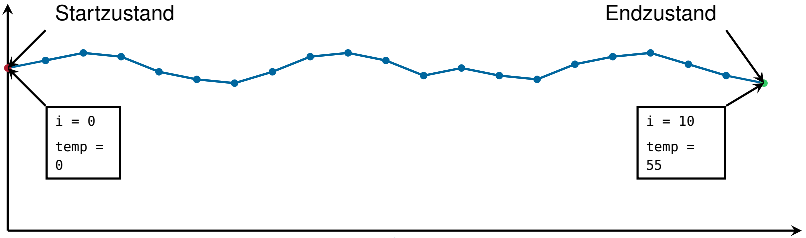


☞ `res = average([1,2,...,10],10);`

```
unsigned int average(uint *array,
                     uint size)
{
  uint temp = 0;

  for(uint i = 0;i < size;i++) {
    temp += array[i];
  }

  return temp/size;
}
```

i	temp
0	0
1	1
2	3
3	6

Das Verhalten zur Laufzeit ist entscheidend!



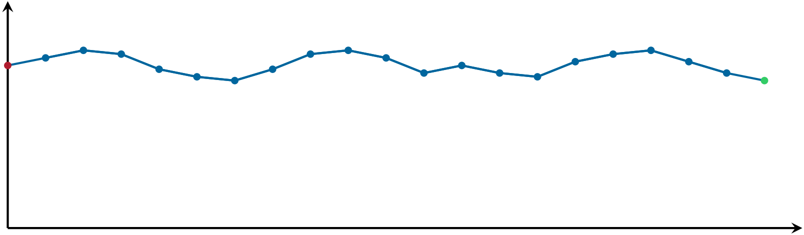
☞ `res = average([1,2,...,10],10);`

```
unsigned int average(uint *array,
                     uint size)
{
  uint temp = 0;

  for(uint i = 0;i < size;i++) {
    temp += array[i];
  }

  return temp/size;
}
```

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

Konkrete Programmsemantik

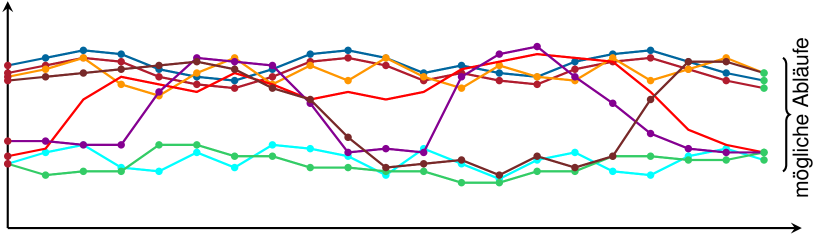
Eine informelle Einführung in die Prinzipien abstrakter Interpretation [2]



- Die konkrete Semantik (engl. *concrete semantics*) beschreibt
  - Alle möglichen Ausführungen eines Programms
  - Unter allen möglichen Ausführungsbedingungen

Konkrete Programmsemantik

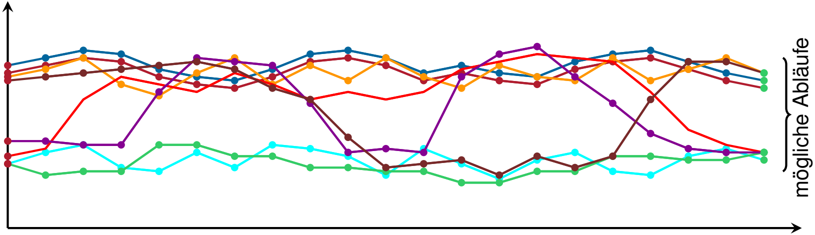
Eine informelle Einführung in die Prinzipien abstrakter Interpretation [2]



- 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

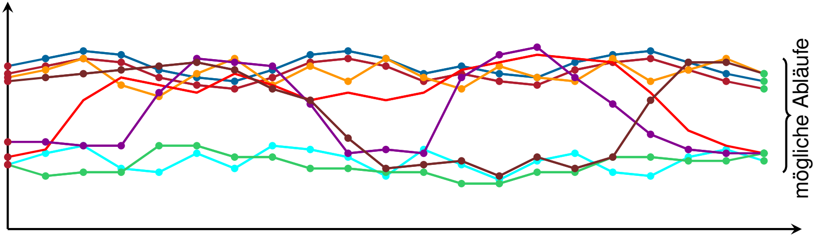
Konkrete Programmsemantik

Eine informelle Einführung in die Prinzipien abstrakter Interpretation [2]



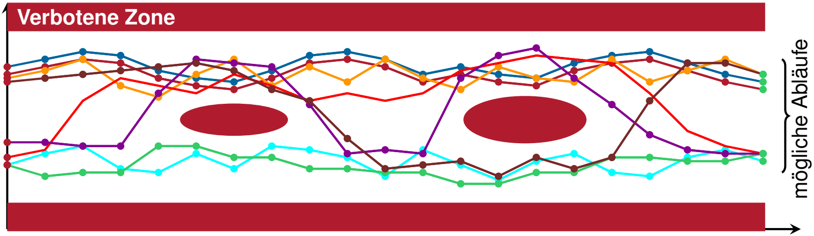
- Die konkrete Semantik (engl. *concrete semantics*) beschreibt
  - Alle möglichen Ausführungen eines Programms
  - Unter allen möglichen Ausführungsbedingungen
  - Für unser Beispiel bedeutet dies:
    - 2^{32} verschieden große Felder, 2^{32} verschiedene Werte für jedes Element
- Sie beschreibt ein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

Sicherheitseigenschaft



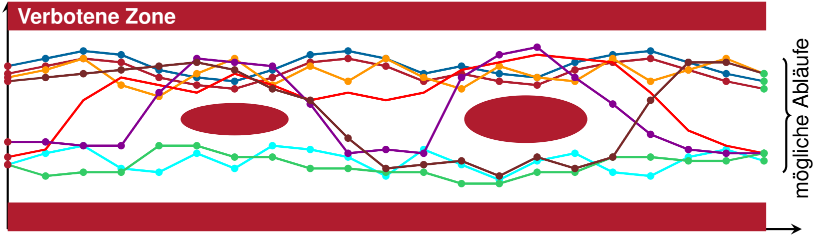
- Sicherheitseigenschaften (engl. *safety properties*) stellen sicher, dass keine fehlerhaften/unerwünschten Zustände eingenommen werden

Sicherheitseigenschaft



- 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

Sicherheitseigenschaft

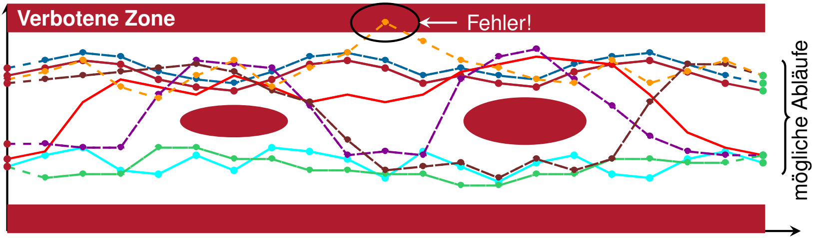


- Sicherheitseigenschaften (engl. *safety properties*) stellen sicher, dass keine fehlerhaften/unerwünschten Zustände eingenommen werden
- Ein Sicherheitsnachweis (engl. *safety proof*) garantiert, dass die konkrete Semantik nie eine verbotene Zone durchläuft

⚠ Das ist ein unentscheidbares Problem

- Die konkrete Programmsemantik ist nicht berechenbar

Testen: Das Problem der Möglichkeiten



- Testen betrachtet nur eine Teilmenge aller möglichen Ausführungen

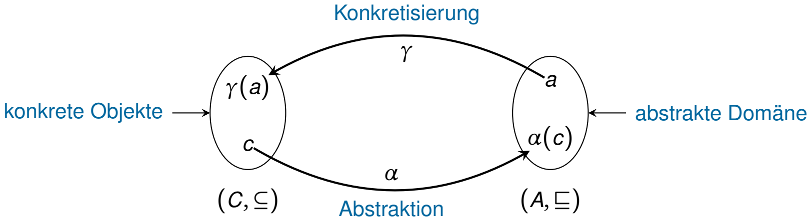
  → Gut geeignet, um die Existenz von Defekten zu zeigen

  → Ungeeignet, um ihre Abwesenheit zu zeigen

  - Evtl. hat man die fehlerhafte Ausführung einfach nicht getestet

- Problem: unzureichende Abdeckung der konkreten Semantik

Abstrakte Interpretation: Theoretisches Fundament



- 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'

Theoretisches Fundament $\rightsquigarrow$ Galoisverbindungen

☞ Approximation von f durch die abstrakte Funktion f'

- Häufig verwendet man Galoisverbindungen mit den Eigenschaften:
  - $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ und $\alpha(\gamma(a)) = a$ (Einbettung)

→ Konkretisierung gefolgt von Abstraktion impliziert keinen Präzisionsverlust

Theoretisches Fundament $\rightsquigarrow$ Galoisverbindungen

☞ Approximation von f durch die abstrakte Funktion f'

- Häufig verwendet man Galoisverbindungen mit den Eigenschaften:
  - $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ 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

Theoretisches Fundament $\leadsto$ Galoisverbindungen

☞ Approximation von f durch die abstrakte Funktion f'

- Häufig verwendet man Galoisverbindungen mit den Eigenschaften:
  - $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ und $\alpha(\gamma(a)) = a$ (Einbettung)

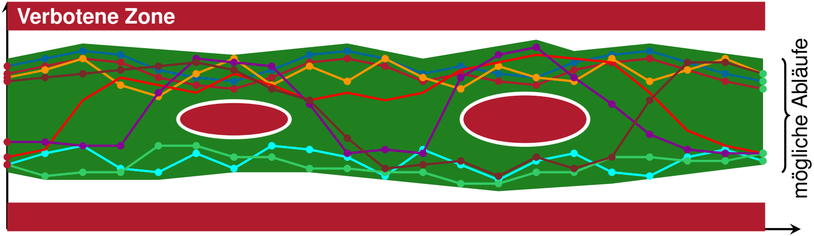
  → Konkretisierung gefolgt von Abstraktion impliziert keinen Präzisionsverlust

- 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



- 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

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

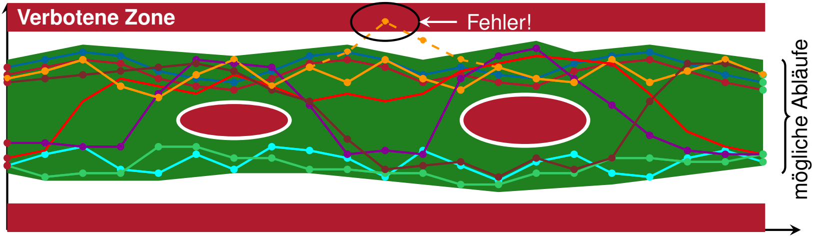
Deduktive Methoden

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

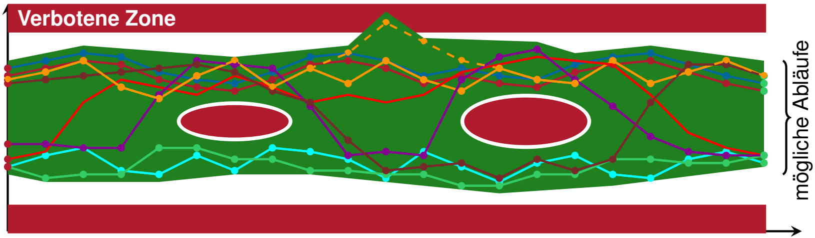
Statische Analyse

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

Eigenschaften abstrakter Semantiken



Eigenschaften abstrakter Semantiken



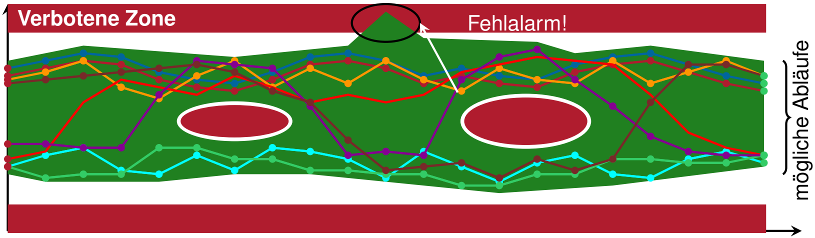
Vollständigkeit und Korrektheit (engl. *soundness*)

- Keine potentieller Defekt darf übersehen werden

⇝ nur so kann die Abwesenheit von Defekten gezeigt werden

  - Ansonsten wäre gegenüber reinem Testen nichts gewonnen

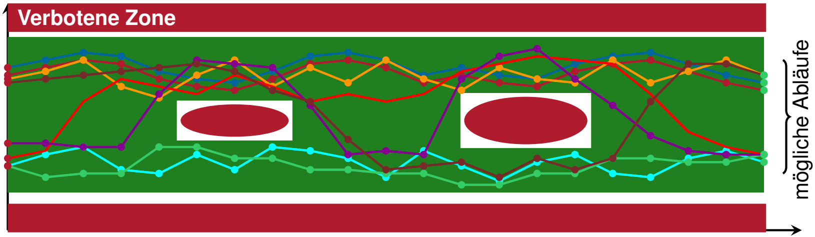
Eigenschaften abstrakter Semantiken



Präzision

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

Eigenschaften abstrakter Semantiken



Geringe Komplexität

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

Gliederung

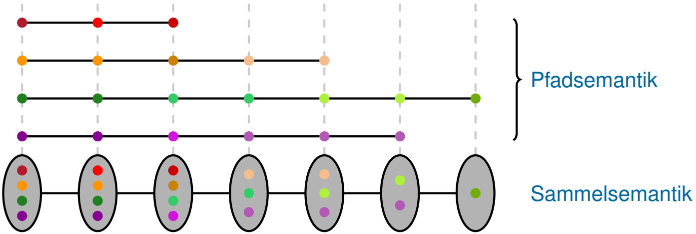
Problemstellung

⚠ Reduktion des Zustandsraums ist unumgänglich!

☞ Fasse verschiedene Zustände geeignet zusammen
→ Sammelsemantiken (s. Folie 25 ff.)

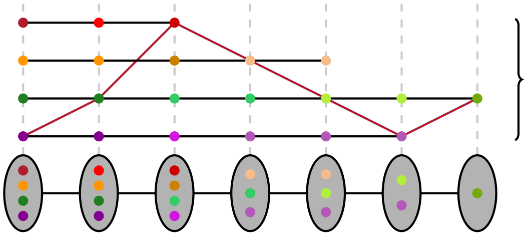
☞ Betrachte nur den Anfang der Zustandshistorie
→ Präfixsemantiken (s. Folie 31 ff.)

Sammelsemantik (engl. *collecting semantics*)



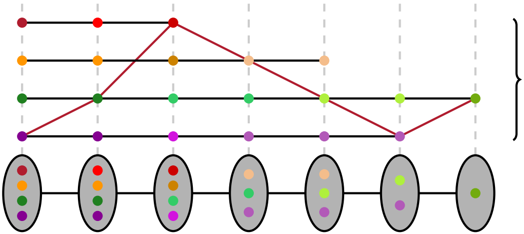
- Sammelt die Zustände aller Pfade an einem bestimmten Punkt
  - D.h. an einer bestimmten Programmanweisung
  - Aufgrund der Größe, wird sie i. d. R. approximiert

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 $\mapsto$???

Sammelsemantik (engl. *collecting semantics*)



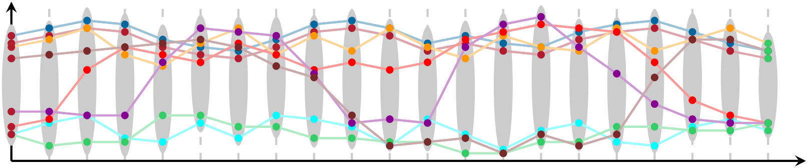
- Sammelt die Zustände aller Pfade an einem bestimmten Punkt
  - D.h. an einer bestimmten Programmanweisung
  - Aufgrund der Größe, wird sie i. d. R. approximiert
- Das ist eine verlustbehaftete Abstraktion
  - Beispiel: Existiert der rote Pfad?
    - Konkrete Semantik $\mapsto$ Nein, Sammelsemantik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

☞ Der Laufzeitgewinn wird durch Unschärfe erkauft!

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

Intervallabstraktion

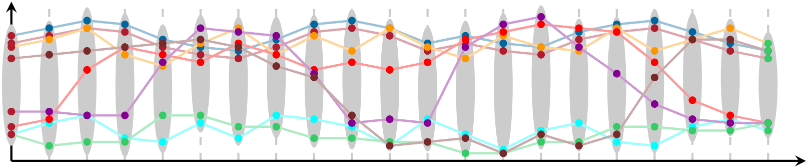
Als Approximation der Sammelsemantik [7]



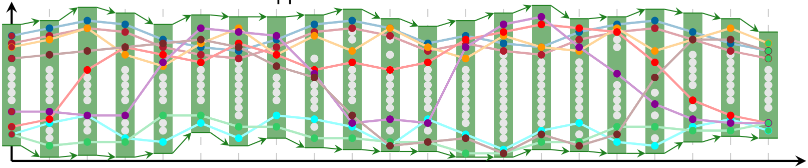
- Die Sammelsemantik verwaltet Zustandsmengen

Intervallabstraktion

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
    - $\leadsto$ Bestimmte Zustände im appromixierten Zustandsraum werden nicht erreicht



Beispiel: Intervallabstraktion für ein C-Programm

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

Intervallabstraktion liefert (am Ende der Zeile):

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

- 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
  - Bilden von Schnittmengen bei Erfüllung von Pfadbedingungen

Beispiel: Intervallabstraktion für ein C-Programm

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

Intervallabstraktion liefert (am Ende der Zeile):

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

- 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
  - Bilden von Schnittmengen bei Erfüllung von Pfadbedingungen
- 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

Beispiel: Intervallabstraktion für ein C-Programm (Forts.)

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

Beispiel: Intervallabstraktion für ein C-Programm (Forts.)

```
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 $x_1 = \emptyset$

Zeile 3 $x_3 = \emptyset$

Zeile 4 $x_4 = \emptyset$

Zeile 7 $x_7 = \emptyset$

Beispiel: Intervallabstraktion für ein C-Programm (Forts.)

```
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 $x_3 = \emptyset$

Zeile 4 $x_4 = \emptyset$

Zeile 7 $x_7 = \emptyset$

Beispiel: Intervallabstraktion für ein C-Programm (Forts.)

```
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 $x_4 = \emptyset$

Zeile 7 $x_7 = \emptyset$

Beispiel: Intervallabstraktion für ein C-Programm (Forts.)

```c
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 $x_7 = \emptyset$

Beispiel: Intervallabstraktion für ein C-Programm (Forts.)

```c
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 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$

Beispiel: Intervallabstraktion für ein C-Programm (Forts.)

```
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 $x_7 = \emptyset$

Beispiel: Intervallabstraktion (Forts.)

```
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*) [8]

Iteration 3:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 3]$

Zeile 4 $x_4 = [2, 3]$

Zeile 7 $x_7 = \emptyset$

Beispiel: Intervallabstraktion (Forts.)

```
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*) [8]

Iteration 3:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 3]$

Zeile 4 $x_4 = [2, 4]$

Zeile 7 $x_7 = \emptyset$

Beispiel: Intervallabstraktion (Forts.)

```
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*) [8]

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 $x_4 = [2, 9999]$

Zeile 7 $x_7 = \emptyset$

Beispiel: Intervallabstraktion (Forts.)

```
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*) [8]

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 $x_7 = \emptyset$

Beispiel: Intervallabstraktion (Forts.)

```
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*) [8]

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 $x_7 = [10000, 10000]$

Vereinfachung: Intervallabstraktion – mit Widening

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

Vereinfachung: Intervallabstraktion – mit Widening

```
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 [8]

Iteration 1:

Zeile 1 $x_1 = \emptyset$

Zeile 3 $x_3 = \emptyset$

Zeile 4 $x_4 = \emptyset$

Zeile 7 $x_7 = \emptyset$

Vereinfachung: Intervallabstraktion – mit Widening

```
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 [8]

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = \emptyset$

Zeile 4 $x_4 = \emptyset$

Zeile 7 $x_7 = \emptyset$

Vereinfachung: Intervallabstraktion – mit Widening

```
unsigned short x = 1;

while(x < 10000) {
  x = x + 1;
}

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 [8]

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = \emptyset$

Zeile 7 $x_7 = \emptyset$

Vereinfachung: Intervallabstraktion – mit Widening

```
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 [8]

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$

Vereinfachung: Intervallabstraktion – mit Widening

```
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 [8]

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 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$

Vereinfachung: Intervallabstraktion – mit Widening

```
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 $x_3 = (x_1 \triangledown x_4) \cap [-\infty, 9999]$

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

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

- Approximation mit Hilfe eines Widening-Operators [8]

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 $x_7 = \emptyset$

Vereinfachung: Intervallabstraktion – mit Widening

```
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 [8]

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 $x_7 = [10000, 10000]$

Vereinfachung: Intervallabstraktion – mit Widening

```
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 [8]

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 $x_1 = [1,1]$

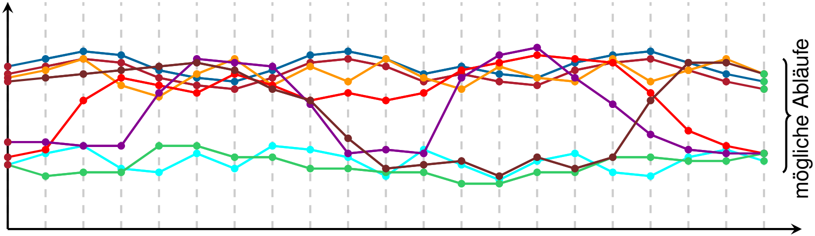
Zeile 3 $x_3 = [1,9999]$

Zeile 4 $x_4 = [2,10000]$

Zeile 7 $x_7 = [10000,10000]$

- ☞ Konvergenz in der 2. Iteration

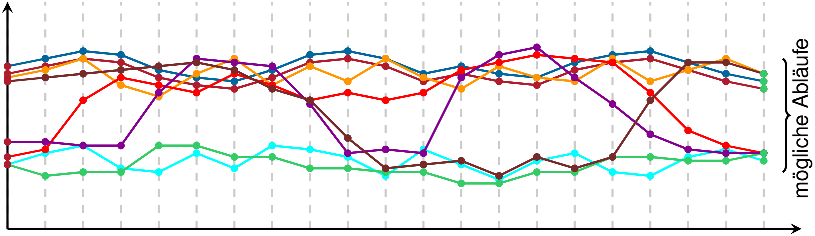
Pfadsemantik



- 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.

Pfadsemantik

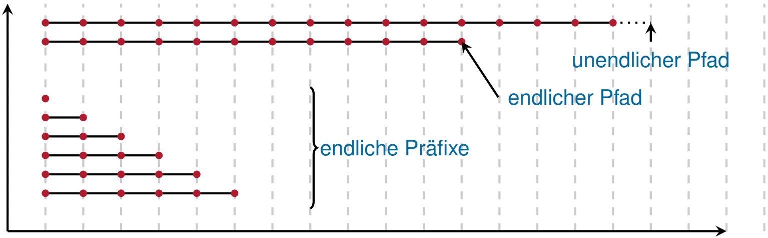


- Betrachte durch ein Transitionssystem beschriebene Programmpfade
  - Ausgehend von ausgezeichneten Startzuständen,
  - Beschreiben sie eine (unendliche) Abfolge von Programmzuständen,
  - Deren Reihenfolge durch die Übergangsrelation bestimmt wird.

  → Die Gesamtheit dieser Programmpfade heißt Pfadsemantik
    - Wie die konkrete Programmsemantik ist sie nicht berechenbar.

- Reduktion der Komplexität durch Abstraktion
  - Unendliche Pfade $\rightsquigarrow$ (endliche) Pfadpräfixe

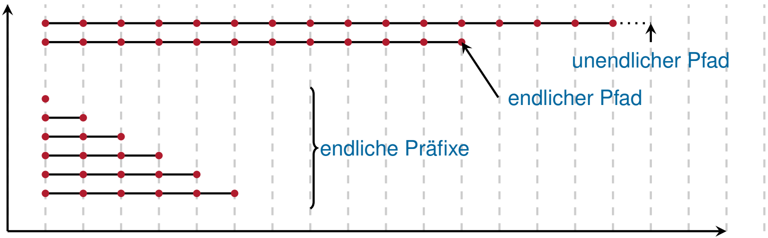
Pfadpräfixe



☞ Pfadsemantiken enthalten alle endlichen und unendlichen Pfade

- Pfadpräfixe enthalten nur die Anfänge dieser Pfade

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: $\{a^n | n \geq 0\} \cup \{a^n b | n \geq 0\} \mapsto$???

Gliederung

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

Literaturverzeichnis

[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

Literaturverzeichnis (Forts.)

[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`. –
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://janmidtgaard.dk/aiws15/`, 2015

[8] Nielson, F. ; Nielson, H. R. ; Hankin, C. :
Principles of program analysis.
Springer, 2015

[9] 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

Literaturverzeichnis (Forts.)

[10] Turing, A. M.:
On computable numbers, with an application to the Entscheidungsproblem.
In: *Journal of Math* 58 (1936), Nr. 345-363, S. 5