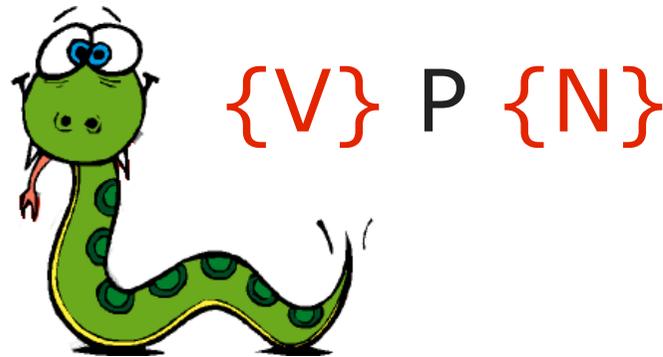


Algorithmen und Programmieren II

Programmverifikation



SS 2012

Prof. Dr. Margarita Esponda

Assertions oder Zusicherungen

Das Konzept der Zusicherungen kommt aus dem Gebiet der **Programmverifikation** und **Softwarequalität**.

Assertions sind Anweisungen, die mit Hilfe eines logischen Ausdrucks Bedingungen während der Ausführung eines Programms überprüfen.

Beim **Assertions** geht der Programmierer davon aus, dass diese immer erfüllt sind, überprüft aber trotzdem diese Behauptungen, um zu garantieren, dass Programme immer korrekt ablaufen.

Diese Art der Programmierung nennt man ***defensives Programmieren***.

Design by Contract (DBC)

Zusicherungen sind kein neues Konzept.

Assertions kommen aus dem DBC-Konzept, der seit Jahren in mehreren Programmiersprachen eingeführt ist.

Die Idee ist, dass die Software nur nach einem bestimmten Vertrag (**Spezifikation**) zwischen Softwareentwickler und Kunden entworfen wird.

Fast alle wichtigen modernen Programmiersprachen unterstützen das DBC-Konzept

Beispiel:

C, C++, C#, Java, Pearl, PHP, Python, usw.

Design by Contract (DBC)

Nach dem DBC-Konzept gibt es drei Sorten von Assertions:

Vorbedingungen (Preconditions)

Bedingungen, die vor der Ausführung einer Funktion oder Methode erfüllt werden sollten.

Nachbedingungen (Postconditions)

Bedingungen, die nach der Ausführung einer Funktion oder Methode gelten sollten.

Invariant (Invariante)

Bedingungen, die innerhalb eines Programmsegments immer wahr sein müssen. Zum Beispiel innerhalb einer `for`-Schleife, Funktion oder Klasse.

Motivation

Es wird ständig wichtiger aus finanziellen Gründen Fehler in Programmen zu finden.

Mit Assertions versucht der Softwareentwickler, Fehler so früh wie möglich zu beseitigen.

Nach Qusay H. Mahmoud, sahen im Jahr 2005 die Fehlerkorrekturkosten, je nachdem in welcher Phase der Softwareentwicklung die Fehler beseitigt wurden, wie folgt aus.

\$1	Definition der Anforderungen
\$5	Softwaredesign
\$10	Implementierung der Software
\$20	Testphase
\$200	nach der Lieferung des Software

assert-Anweisung in Python

Syntax:

```
assert  $e_1, e_2, \dots, e_n$ 
```

Beispiel:

```
myList = [1,2]
```

```
assert len(myList)>0
print(myList.pop())
```

```
assert len(myList)>0
print(myList.pop())
```

```
assert len(myList)>0
myList.pop()
```

```
>>>
```

```
2
```

```
1
```

Traceback (most recent call last):

File "assert_statement.py", line 11, in <module>

```
assert len(myList)>0
```

AssertionError

```
>>>
```

Programmverifikation

Imperative Programmiersprachen

Motivation

Wie können wir wissen, dass ein Programm korrekt funktioniert?

- Viel testen hilft, ist aber kein Beweis!
- Sicherheitskritische Softwarebereiche können sich keine Fehler erlauben

Beispiele:

- Steuerungssysteme für Medizingeräte oder Flugzeuge
- Überwachungssysteme für Verkehr (z.B. Zug-Verkehr)
- Software für Weltraumprojekte
- usw.

Programmverifikation

Imperative Programmiersprachen

Ziel

Grundsätzliches Verständnis von Verifikationsmethoden zu erwerben und mit Hilfe dieser Denkweise eine Verbesserung in der Entwicklung von Software zu erreichen.

Programmverifikation ist leider nicht immer einsetzbar und das systematische Testen bleibt als einzige Alternative zur Analyse der Korrektheit eines Programms.

Um ein Programm zu verifizieren, brauchen wir einen mathematischen Formalismus.

Zwei Formalismen

Hoare-Kalkül



Sir Charles Antony Hoare

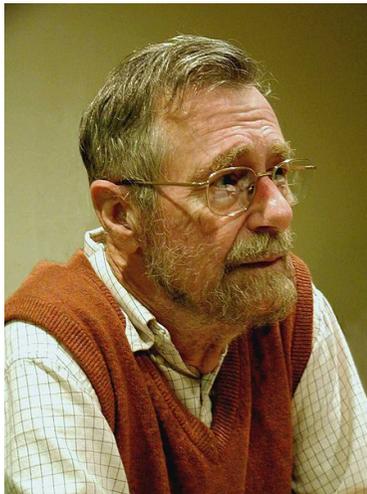
- 1930
- University of Oxford, Microsoft Research Cambridge
- 1980 Turing Award
- Einige wichtige Leistungen sind:
 - CSP (Communicating Sequential Processes)
 - Monitor-Konzept
 - Hoare-Kalkül
 - usw.

Die Verifikation eines Programms besteht aus zwei Schritten:

- Beweis der partiellen Korrektheit
- Beweis der Terminierung

Zwei Formalismen

wp-Kalkül



Edsger Wybe Dijkstra

- 1930
- 1972. Turing Award
- Dijkstra's Algorithms
- Semaphore-Konzept
- Guarded Command Language
- Die Methode der schwächsten Vorbedingung
wp-Kalkül
- *usw.*

Die Methode der schwächsten Vorbedingung. *weakest precondition (wp)*

- Die totale Korrektheit wird in einem einzigen Arbeitsgang bewiesen

Hoare-Kalkül

Grundlegende Definitionen:

Definition von Programmformeln

$\{P\} S \{Q\}$ wird als Programmformel bezeichnet, wenn S ein Programm ist und P und Q Prädikate (über Variablen und Konstanten von S) sind.

Gültigkeit von Programmformeln

Eine Programmformel $\{P\} S \{Q\}$ ist gültig, wenn vor Ausführung von S , das Prädikat P wahr ist und nach Beendigung von S , das Prädikat Q wahr ist, gilt.

Hoare-Kalkül

Programmformeln können als

Programmspezifikation für Programme benutzt werden, wenn diese erst geschrieben werden sollen. *DBC Design By Contract* oder als Programmdokumentation, um die Wirkung eines Programms formal zu beschreiben

Probleme:

- es existieren beliebig viele Formeln für ein Programm
- die Formeln können zu ungenau oder zu schwach sein

Beispiel:

{True} S {True} gilt immer für alle Programme **S**

Axiomatische Semantik

für einfache Anweisungen

Leeranweisungsaxiom

Die leere Anweisung hat keine Auswirkung auf den Zustand irgendeiner Programmvariablen und wird durch folgendes Axiom definiert:

Für jedes Prädikat P ist $\{P\} \{\}$ eine gültige Programmformel.

Axiomatische Semantik

für einfache Anweisungen

Zuweisungsaxiom

Sei $x = \mathit{expr}$ eine Zuweisung (in C, C++, Java, Python, usw.), die den Wert der Variablen x durch den Wert des Ausdrucks expr ersetzt, dann wird die Semantik durch das Zuweisungsaxiom wie folgt definiert.

Für jedes Prädikat P ist $\{P \langle x \leftarrow \mathit{expr} \rangle\} x = \mathit{expr} \{P\}$ eine gültige Programmformel.

Zuweisungsaxiom

Beispiel:

Folgende Programmformel

$$\{ z > 0 \wedge y - 1 > 0 \vee z == 0 \wedge y - 1 \geq 0 \}$$

$$y = y - 1$$

$$\{ z > 0 \wedge y > 0 \vee z == 0 \wedge y \geq 0 \}$$

ist gültig, weil nach Anwendung der Substitution $y \leftarrow y - 1$ in der Nachbedingung diese vor der Ausführung der Zuweisung gültig ist und somit die Nachbedingung auch.

Zuweisungsaxiom

In den Programmformeln verwenden wir ($==$) für die logischen Vergleichsoperationen innerhalb von Prädikaten, um diese von dem Zuweisungsoperator ($=$) innerhalb von Programmen zu unterscheiden.

Beispiel:

Folgende Programmformeln sind gültig:

$$\{z > 0\} \quad x = 0 \quad \{z > x\}$$

$$\{x == 1\} \quad x = x - 1 \quad \{x == 0\}$$

$$\{y \geq 1 \wedge y \leq 5\} \quad x = y + 1 \quad \{x \geq 2 \wedge y \leq 5\}$$

Das Hoare-Kalkül

Das Hoare-Kalkül stellt eine Reihe von **Ableitungsregeln** zur Verfügung, die, den Nachweis **partieller** oder **totaler Korrektheit** ermöglichen.

Die Ableitungsregeln haben folgende **allgemeine Form**:

$$\frac{H_1, \dots, H_n}{H}$$

und bedeuten, dass, **wenn** H_1 bis H_n **erfüllt sind**, **dann** auch H **erfüllt ist**.

Manche Regeln haben zusätzliche Bedingungen, die festlegen, wann sie angewendet werden dürfen.

Ableitungsregeln

Sequenzregel

Besteht eine Sequenz aus den Anweisungen S_1 und S_2 ,
 und sind die Programmformeln $\{P\} S_1 \{R\}$ und $\{R\} S_2$
 $\{Q\}$ gültig, dann ist auch $\{P\} S_1, S_2 \{Q\}$ gültig.

$$\frac{\{P\} S_1 \{R\}, \{R\} S_2 \{Q\}}{\{P\} S_1, S_2 \{Q\}}$$

"," als Verallgemeinerung
 der Trennzeichen
 zwischen Anweisungen

Ableitungsregeln

Sequenzregel

C, C++, Java, usw.

$$\{P\} S_1 \{R\}, \{R\} S_2 \{Q\}$$

$$\{P\} S_1; S_2 \{Q\}$$

Python

$$\{P\} S_1 \{R\}, \{R\} S_2 \{Q\}$$

$$\{P\} S_1$$

$$S_2 \{Q\}$$

Induktiv lässt sich diese Regel von zwei Anweisungen auf endlich viele aufeinanderfolgende Anweisungen erweitern.

Sequenzregel

Beispiel:

Nehmen wir an, wir möchten die Gültigkeit folgender Programmformel beweisen

$$\begin{array}{l}
 \{ x > y \} \\
 y = x - y \\
 y = x + y \\
 \{ x < y \} \quad \dots \dots \dots \quad (1)
 \end{array}$$

Nach der Sequenzregel ist die Programmformel gültig, wenn wir eine Bedingung **R** finden können, so dass die beiden Programmformeln

$$\begin{array}{l}
 \{x>y\} \ y = x - y \ \{R\} \quad \dots \dots \dots \quad (2) \\
 \text{und} \ \{R\} \ y = x+y \ \{x<y\} \quad \dots \dots \dots \quad (3)
 \end{array}$$

gültig sind.

Sequenzregel

$$\{x > y\} \quad y = x - y \quad \{R\} \quad \dots \dots \dots \quad (2)$$

$$\{R\} \quad y = x + y \quad \{x < y\} \quad \dots \dots \dots \quad (3)$$

Wenn wir das Zuweisungsaxiom auf (3) anwenden, erhalten wir als Vorbedingung $R \equiv x < x + y$,

$$\{R \equiv x < x + y\} \quad y = x + y \quad \{x < y\} \quad \dots \dots \quad (3)$$

und wenn wir wiederum das Prädikat R in (2) einsetzen, erhalten wir mit dem Zuweisungsaxiom die Vorbedingung $x < x + x - y$, die äquivalent zu $x > y$ ist.

$$\{x > y\} \equiv \{x < x + (x - y)\} \quad y = x - y \quad \{R \equiv x < x + y\} \quad \dots \quad (2)$$

Regel für Blockanweisungen

Eine Anweisungsfolge S kann durch **begin** und **end** in Pascal oder **geschweifte Klammern** in C oder Java eingeschlossen werden, ohne dass sich die Semantik von S ändert.

Pascal

$$\frac{\{P\} \ S \ \{Q\}}{\{P\} \ \text{begin } S \ \text{end } \{Q\}}$$

C, C++, Java, usw.

$$\frac{\{P\} \ S \ \{Q\}}{\{P\} \ \{S\} \ \{Q\}}$$

Ableitungsregeln

Bedingungsregel

wenn wir folgende Programmformel betrachten

```

{P}
  if Bedingung:
    S1
  else:
    S2
{Q}
    
```

müssen wir zwischen zwei Fällen unterscheiden

1. Fall *Bedingung* == *True*

dann ist $\{P \wedge \text{Bedingung}\} S_1 \{Q\}$ eine gültige Formel

2. Fall *Bedingung* == *False*

dann ist $\{P \wedge \neg \text{Bedingung}\} S_2 \{Q\}$ eine gültige Formel

Ableitungsregeln

Bedingungsregel bzw.
Inferenz-Schema für Fallunterscheidung

$$\{P \wedge B\} S_1 \{Q\}, \{P \wedge \neg B\} S_2 \{Q\}$$

$$\{P\} \text{if } (B) S_1 \text{ else } S_2 \{Q\}$$

C, C++, Java, usw.

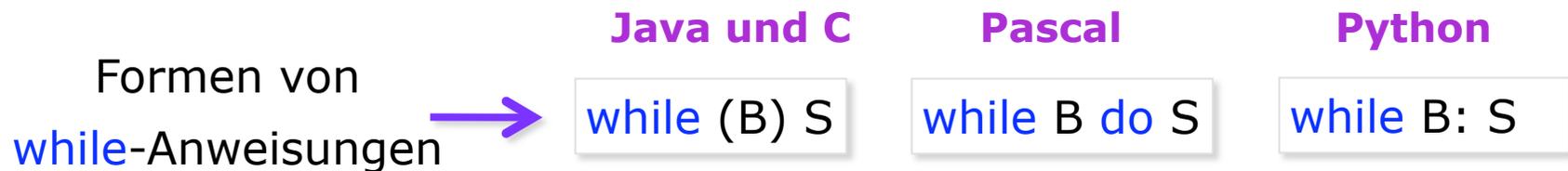
Python

$$\{P \wedge B\} S_1 \{Q\}, \{P \wedge \neg B\} S_2 \{Q\}$$

```
{P}
if B:
    S1
else:
    S2
{Q}
```

Ableitungsregeln

while-Regel



Nehmen wir an, dass $\{P \wedge B\} S \{P\}$ eine gültige Programmformel für den Schleifen-Block S ist

wenn B nach einem Schleifen-Durchlauf noch erfüllt wird, dann gilt die Vorbedingung $\{P \wedge B\}$

wenn B nicht mehr erfüllt wird, dann gibt es keinen Schleifen-Durchlauf mehr und es gilt $\{P \wedge \neg B\}$

while-Regel

$$\frac{\{P \wedge B\} S \{P\}}{\{P\} \text{ while } B: S \{P \wedge \neg B\}}$$

Die Vorbedingung P ist nach jedem Schleifen-Durchlauf wahr.

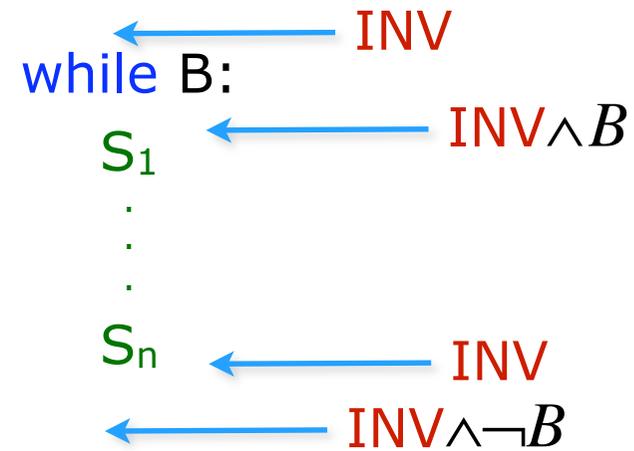
P wird als **Schleifen-Invariante** bezeichnet.

Die Festlegung von Schleifen-Invarianten spielt eine zentrale Rolle in der Korrektheitsanalyse von Programmen.

Aus der Gültigkeit von $\{P \wedge B\} S \{P\}$ kann nicht abgeleitet werden, dass jemals $\neg B$ erfüllt wird, d.h. wir wissen nicht, ob die Schleife terminiert.

while-Regel

$$\frac{\{INV \wedge B\} S \{INV\}}{\{INV\} \text{while } B: S \{P \wedge \neg INV\}}$$



while-Regel

Beispiel:

Nehmen wir an, wir möchten die Gültigkeit folgender Programmformel beweisen:

```

{ x ≥ 0 }
while x ≠ 0:
    x = x - 1
{ x == 0 }
    
```

wir wählen $P \equiv x \geq 0$ und wissen, dass $B \equiv x \neq 0$.

Weil $P \wedge \neg B \Leftrightarrow x == 0$, können wir unsere Programmformel wie folgt schreiben:

```

{ P }
while x ≠ 0:
    x = x - 1
{ P ∧ ¬B }
    
```

while-Regel

... weiter mit dem Beispiel:

Nach der **while**-Regel ist unsere Programmformel gültig,
wenn

$$\{P \wedge B\} \quad x = x - 1 \quad \{P\}$$

mit anderen Worten, wir müssen nur die Gültigkeit folgender
Programmformel beweisen

$$\{x \geq 0 \wedge x \neq 0\} \quad x = x - 1 \quad \{x \geq 0\}$$

Diese ergibt sich aus dem **Zuweisungsaxiom**

$$(x \geq 0 \wedge x \neq 0) \Rightarrow x \geq 1 \Rightarrow \{x - 1 \geq 0\} \quad x = x - 1 \quad \{x \geq 0\}$$

mit $P \equiv x \geq 0$ (Schleifeninvariante)

```
{ x ≥ 0 }
while x != 0:
    x = x - 1
{ x == 0 }      ✓
```

Ableitungsregeln

do-while-Regel

Pascal

$$\{P\} S \{Q\}, \{Q \wedge \neg B\} \Rightarrow \{P\}$$

$$\{P\} \text{ repeat } S \text{ until } B; \{P \wedge B\}$$

C und Java

$$\{P\} S \{Q\}, \{Q \wedge B\} \Rightarrow \{P\}$$

$$\{P\} \text{ do } S \text{ while } (B); \{P \wedge \neg B\}$$

Konsequenz-Regeln

Regel der Schwächeren Nachbedingung

Wenn nach der Ausführung einer Anweisungsfolge **S** die Bedingung **R** erfüllt ist, dann ist auch jede Nachbedingung **Q** erfüllt, die aus **R** impliziert werden kann.

$$\{P\} S \{R\}, R \Rightarrow Q$$

$$\{P\} S \{Q\}$$

Konsequenz-Regeln

Regel der **stärkeren Vorbedingung**

Wenn $\{R\} S \{Q\}$ eine gültige Programmformel ist, dann ist für jede Bedingung P , die R impliziert, auch die Programmformel $\{P\} S \{Q\}$ gültig.

$$\frac{P \Rightarrow R, \{R\} S \{Q\}}{\{P\} S \{Q\}}$$

Konsequenz-Regeln

Beispiel:

Nehmen wir an, dass folgende Programmformel gilt:

$$\{x > 0 \wedge y > 0\} \text{ S } \{z + a \cdot y == x \cdot y \wedge a == 0\}$$

aber

$$(z + a \cdot y == x \cdot y) \wedge a == 0 \Rightarrow z == x \cdot y$$

dann folgt aus der Regel der schwächeren Nachbedingung

$$\{x > 0 \wedge y > 0\} \text{ S } \{z == x \cdot y\}$$

ist eine gültige Programmformel.

Anwendung der Konsequenz-Regeln

Wenn wir die Regel der schwächeren Nachbedingungen zweimal verwenden, können wir folgendes Inferenz-Schema aus der Bedingungsregel ableiten:

Python

$$\{P \wedge B\} S_1 \{Q\}, \{P \wedge \neg B\} S_2 \{R\}$$

```

    {P}
    if B:
        S1
    else:
        S2
    {Q ∨ R}
    
```

weil

$$Q \Rightarrow Q \vee R$$

und

$$R \Rightarrow Q \vee R$$

und in beiden oberen Programmformeln angewendet werden kann.