Verlässliche Systeme

9. Kapitel
Verifikation von Software

Prof. Matthias Werner
Professur Betriebssysteme

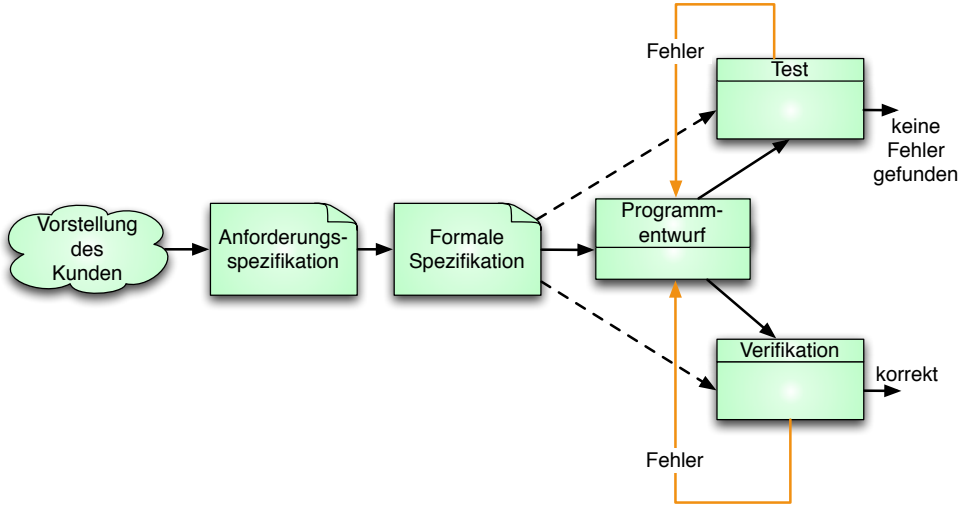
9.1 Grundsätzliches
Fehlertoleranz vs. Fehlerintoleranz

- Bisher haben wir Ausfälle der **Hardware** betrachtet
- **Jetzt**: Betrachten Software

- Wiederholung:
  - **Fehlerintoleranz** ➡ Fehler vermeiden
  - **Fehlertoleranz** ➡ Existenz von Fehlern akzeptieren und negative Auswirkungen verhindern/reduzieren

- Zwei Basisansätze der Fehlerintoleranz bei Software
  - **Testen**
  - **Verifikation**

Verifikation und Test in der SW-Entwicklung



Testen vs. Verifikation

Testen

- Für einige möglichst gut ausgewählte Testdaten wird das Programm ausgeführt und beobachtet, ob das gewünschte Ergebnis ermittelt wird
- Nicht alle möglichen Kombinationen von Eingabedaten können getestet werden
- Keine Gewissheit über die Korrektheit des Programms für die noch nicht getesteten Eingabedaten
- **Aber**: auch Anforderungsfehler/ Spezifikationsfehler können evtl. gefunden werden

Verifikation

- liefert einen mathematischen, formalen Beweis, dass das Programm der Spezifikation gehorcht
- Anforderungsfehler können nicht gefunden werden
  - Sie werden aber in der Praxis oft doch gefunden, weil das formale Vorgehen präzisere Spezifikationen erzwingt
- Verifikation sollte im Entwurf und der Implementierung mit eingeplant sein – nachträgliche Verifikation ist meist unmöglich

- Wir betrachten hier Verifikation

Semantiken

- Um über die Korrektheit von Programmen zu urteilen, muss die **Programmsemantik** formalisiert werden
- Gemeinhin sind drei Ansätze bekannt:
  - **Denotationelle Semantik**
    Semantik eines Programms wird durch eine Funktion zugewiesen; Nutzung von **Domainen** (Bereichstheorie)
  - **Operationelle Semantik**
    Semantik durch Betrachtung von Zustandsübergängen; jeder Schritt wird durch ein **Paar** (Vorzustand, Folgezustand) beschrieben
  - **Axiomatische Semantik**
    Beschreibung der Semantik durch ihre **logischen Eigenschaften**, wobei im Allgemeinen nur einige Eigenschaften betrachtet werden
- Wir betrachten axiomatische Semantik

Zusicherungen

- **Ziel**: Ableitung eines Nachzustandes aus einem Vorzustand
- Einen exakten Vor- und Nach**zustand** zu spezifizieren ist meist schwierig
- ➡ Spezifikation einer **Menge** von Zuständen über **Bedingungen** (*assertions*) an die Zustände
- Diese Bedingungen können unterschiedlich stark sein

```
// { y > 1 }
x= y*y;
// { x >= 0 }
```

- Schwächere Nachbedingung

```
// { y > 1 }
x= y*y;
// { x > y > 1 }
```

- Stärkere Nachbedingung

Verifikation

- Verifikation ist der formale, d.h. mathematische Beweis, dass das Programm die Spezifikation erfüllt, dass es korrekt ist

Partielle Korrektheit

Ein Programm(teil) ist **partiell korrekt**, wenn es von einem spezifizierten Vorzustand (Q) die Abarbeitung beginnt, bei einer eventuellen Terminierung einen spezifizierten Endzustand (R) erreicht.

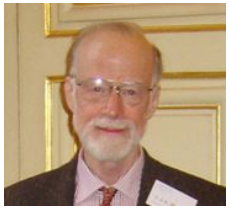
Totale Korrektheit

Ein Programm(teil) ist **total korrekt**, wenn es partiell korrekt ist und terminiert.

9.2 Hoare-Kalkül

C.A.R. Hoare

Sir Charles Antony Hoare
(besser bekannt als: Tony Hoare)



- ⋆ 1934
- Informatikprofessor, University of Oxford, später Microsoft Research Cambridge
- 1980: Turing Award
- 2011: John-von-Neumann-Medaille
- Bedeutende Leistungen:
  - Konzept des Monitors
  - **Hoare-Kalkül** (auch: Floyd-Hoare-Kalkül)
  - CSP (Communicating Sequential Processes)

Hoare-Kalkül

- Formale Präzisierung des Korrektheitsbegriffs
- Regeln für die Korrektheit einzelner Anweisungstypen
- **Vorwärtsanalyse** von imperativen Programmen
  - Ableiten von Aussagen über das Programm aus den Anweisungen heraus in Vorwärtsrichtung
  - Unterschied: Rückwärtsableiten von Aussagen (z.B. für imperative Sprachen: wp-Kalkül von Dijkstra)
- **Hoare-Tripel**: $\{Q\}P\{R\}$
  - Bedeutung: Wenn **vor** Ausführung von P die Zusicherung Q galt, dann muss **nach** Ausführung von P die Zusicherung R gelten
  - Manchmal[1] andere Syntax: $Q\{P\}R$
- Ohne Bezug auf ein konkretes Programm P: $\{Q\}.\{R\}$ ➡ Spezifikation
  - Aufgabe des Programmierers, ein Programm P zu schreiben, das der Spezifikation genügt

[1] z.B. in Hoares Originalartikel

Anwendung und Einschränkungen

- **Anwendung**
  - Zerlegung eines Programms $\{Q\}P\{R\}$ in Sequenz von Programmfragmente $\{Q_i\}P_i\{R_i\}$ dekomponiert werden
- **Theoretische Einschränkungen**
  - **Achtung:** Das Hoare-Tripel sagt **nichts** über P/R aus, wenn Q **nicht** gilt!
- **Praktische Einschränkungen**
  - Verifikation (mit Hoare-Kalkül) gelingt i.d.R. nur für kleine Programme
  - Praktischer Einsatz beschränkt sich daher auf (sicherheits)kritische Teile komplexer Programmsysteme, von denen das Funktionieren des restlichen System essentiell abhängt. (Trusted Computing Base)

Verifikationsregeln

Verifikationsregeln für imperative Programme

- **Imperative Programme**
  - Bestehen aus linearen Kontrollstrukturen
  - Korrektheit: Ergibt sich aufgrund der Korrektheit der Teilstrukturen
  - Ein komplexes Programm kann schrittweise durch korrektes Zusammensetzen aus einfacheren Strukturen verifiziert werden (Beweisdekomposition)
- **Verifikationsregeln**
  - Zuweisungs-Axiom
  - Sequenz-Regel (Kompositionsregel)
  - Konsequenz-Regel (Verstärkung und Abschwächung)
  - If-Else-Regel (Verzweigungsregel)
  - While-Regel (Schleifenregel)
  - Methodenaufrufregel

Schreibweise

- Allgemeines Format von Regeln

$$\frac{\text{Voraussetzung}}{\text{Aussage}}$$

Aus der Voraussetzung folgt die Aussage (Schlussfolgerung)

- Eine Aussage, die immer gilt (ohne Voraussetzung) heißt **Axiom**

$$\frac{}{\text{Aussage}}$$

Zuweisungsaxiom

$$\overline{\{R(x \leftarrow e)\}\ x = e;\ \{R\}}$$

- $\{R(x \leftarrow e)\}$: Die Vorbedingung entsteht dadurch, dass man die Nachbedingung hernimmt und jedes Vorkommen der Variablen x durch den Ausdruck e ersetzt
- Dadurch erhält man zu einer Nachbedingung die passende Vorbedingung
- **Beispiel**:

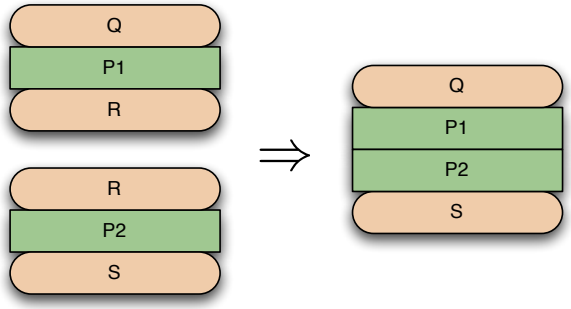
```
// { x+1 = a }
x= x+1;
// { x = a }
```

- Wenn nach der Zuweisung „$x = a$“ gilt, dann muss vor der Zuweisung „$x + 1 = a$“ bzw. „$x = a - 1$“ gegolten haben

Kompositionsregel (Sequenzregel)

$$\frac{\{Q\}\ P1\ \{R\} \qquad \{R\}\ P2\ \{S\}}{\{Q\}\ P1; P2\ \{S\}}$$

- Diese Regel wird benötigt, um aus den Aussagen zu einzelnen Programmteilen Aussagen über zusammengesetzte Programmteile zu gewinnen



Beispiel: Kompositionsregel

$$\frac{\{Q\}\ P1\ \{R\} \qquad \{R\}\ P2\ \{S\}}{\{Q\}\ P1; P2\ \{S\}}$$

```
// { x+1 = a }
x= x+1;
// { x = a }
```

```
// { x = a }
x= 2*x;
// { x = 2*a }
```

➡

backgroundcolor

```
// { x+1 = a }
x= x+1;
x= 2*x;
// { x = 2*a }
```


Beispiel: Swap

- Beweis eines Programmteils, das die Werte zweier Variablen vertauscht

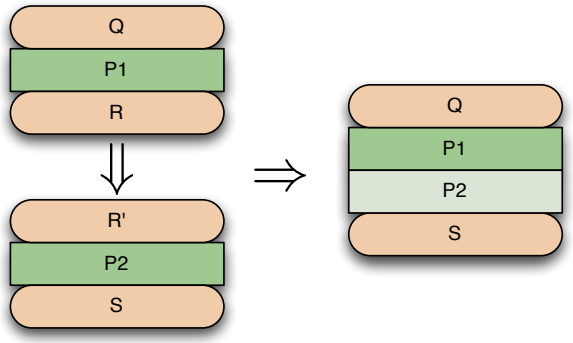
```
1  // PRE: { y=a ∧ x=b }
2  // { y=a ∧ x=b }
3  int z=x;
4  // { y=a ∧ z=b }
5  // { y=a ∧ z=b }
6  x=y;
7  // { x=a ∧ z=b }
8  // { x=a ∧ z=b }
9  y=z;
10 // POST: { x=a ∧ y=b }
```

Identität
Zuweisungsregel
Identität
Zuweisungsregel
Identität
Zuweisungsregel
Komposition
Komposition

- Das Beweisen geht nicht immer so einfach
- Vor- und Nachbedingungen müssen oft angepasst werden
- Verzweigungen und Schleifen treten auf

Anpassung von Vor- und Nachbedingungen

- Bei der Anwendung der Kompositionsregel stellt man häufig fest, dass die Nachbedingung R von $P1$ nicht mit der Vorbedingung R' von $P2$ übereinstimmt
- Kann man jedoch R' aus R ableiten ($R \Rightarrow R'$), dann ist die Schlussfolgerungskette wieder geschlossen



Konsequenzregel

$$\frac{Q \Rightarrow Q', \quad \{Q'\}\ P\ \{R'\}, \quad R' \Rightarrow R}{\{Q\}\ P\ \{R\}}$$

- Die Konsequenzregel gibt es, um die Übergänge zu beweisen, d.h. um Vor- und Nachbedingungen anzupassen
- Folgerungen verschiedener Art sein:
  - syntaktisch
  - logisch
  - arithmetisch

Beispiel für Konsequenzregel

$$\frac{Q \Rightarrow Q', \quad \{Q'\}\ P\ \{R'\}, \quad R' \Rightarrow R}{\{Q\}\ P\ \{R\}}$$

```
1  // { true }
        ⇓                    logische Folgerung
2  // { a=a }
        ⇓                    arithmetische Folgerung
3  // { a*a=a*a }
4  x=a*a                 // Zuweisungsregel
5  // { x=a*a }
        ⇓                    arithmetische Folgerung
6  // { x ≥ 0 }
```


Verzweigungsregel (If-Else-Regel)

$$\frac{\{B \wedge Q\}\ P1\ \{R\} \quad \{\neg B \wedge Q\}\ P2\ \{R\}}{\{Q\}\ if\ (B)\ P1\ else\ P2;\ \{R\}}$$

- Beide Alternativen **müssen** die gleiche Nachbedingung haben

```
1        // { true }
2  if (a<0)
3      // { a<0 ∧ true }
4      // { -a = |a| }
5    abs = -a;
6      // { abs = |a| }
7  else
8      // { a≥ 0 ∧ true }
9      // { a = |a| }
10   abs = a;
11     // { abs = |a| }
12       // { abs = |a| }
```

Verifikation von Schleifen

- Betrachten Beispiel: Berechne $x = \sum_{i=0}^{n-1} i$

```
    // { true }
    // { 0=0  ∧ 1=1 }
  int x= 0;
  int k= 1;
    // {Q: x=0 ∧ k=1 }
  while (k!=n) {
    x= x+k;
    k= k+1;
  }   // {R: x= 1+2+..+(n-1) }
```

- Wie kommt man von Vorbedingung Q zur Nachbedingung R?
- Muss man jeden einzelnen Schleifendurchlauf beweisen?
- Oder reicht es, den Schleifenrumpf einmal zu beweisen?

Verifikation von Schleifen (Forts.)

- Um die Schleife in eine einfache Anweisungssequenz zu zerlegen, könnten wir sie „ausrollen“:

```
            // { 0=0 }
int x= 0;
            // { x=0 } { x+1=0+1 }
x= x+1;
            // { x=0+1}  { x+2=0+1+2 }
x= x+2;
            // { x=0+1+2}  { x+3=0+1+2+3 }
...
            ...
            //{ x+(n-1)=0+1+2+3+...+(n-1) }
x= x+(n-1);
            // { x=0+1+2+...+(n-1)}
```

- **Idee**: einzelne Zusicherungen für jeden Durchlauf zu einem Ausdruck zusammenfassen
- Ausdruck gilt bei jedem Schleifendurchlauf ➡ **Invariante**

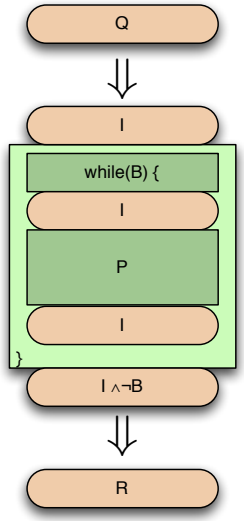
Schleifenregel

$$\frac{\{I \wedge B\}\, P\, \{I\}}{\{I\}\ while(B)\, P;\ \{I \wedge \neg B\}}$$

- Regel nutzt While-Schleife für den Korrektheitsbeweis
  - For-Schleifen können in äquivalente while-Schleifen umgeformt werden
- Keine Beachtung von Break- und Continue-Anweisungen
- Wesentlich: Konstruktion der Invarianten I, die **vor**, **während** und **nach** der Schleife gelten muss
- Aussage der Schleifenregel:
  Wenn vor jeder Schleifeniteration die Invariante I und die Schleifenbedingung B und nach jeder Iteration die Invariante gilt, so ist die Schleife korrekt und nach Beendigung der Schleife gilt die Schleifenbedingung nicht mehr

Schleife

- Die allgemeine Einsatz der Schleife: $\{Q\}\ while\,(B)\,P;\ \{R\}$



- Die Schleife hat also Vor- und Nachbedingungen, die zu der Invarianten „passen“ müssen

Anmerkung

Wir betrachten hier nur While-Schleifen. Andere Schleifen (repeat...until, for) lassen sich abbilden.

Vorgehen

- Folgende Schritte werden durchgeführt:
  1. Finde eine geeignete Schleifeninvariante I
  2. Weise nach, dass die Schleifeninvariante aus der Vorbedingung Q der Schleifenanweisung folgt:
     $$Q \Rightarrow I$$
  3. Beweise die Invarianz der Invariante:
     $$\{I \wedge B\} P \{I\}$$
  4. Zeige, dass die Invariante stark genug ist, die Nachbedingung zu erzwingen
     $$I \wedge \neg B \Rightarrow R$$

Schleife: Beispiel

- Gegeben sei folgendes Java-Programmstück[2]

```
1  // Calculates the square of a
2  // nonnegative integer number a
3  // {Q: 0 ≤ a}
4      y= 0;
5      x= 0;
6      while (y != a) {
7          x= x+2*y+1;
8          y= y+1;
9      }
10 // {R: x = a^2}
```

[2] Beispiel ist aus OHLBACH und EISINGER

Schritt 1: Invariante

```
1  // Calculates the square of a
2  // nonnegative integer number a
3  // {Q: 0 ≤a}
4      y= 0;
5      x= 0;
6      while (y != a) {   ⟵ I
7          x= x+2*y+1;    ⟵ I
8          y= y+1;
9      }                  ⟵ I
10 // {R: x = a^2}        ⟵ I
```

- Die Invariante I muss gelten:

⟸ vor der Schleife
⟸ vor jedem Schleifenrumpf

⟸ nach jedem Schleifenrumpf
⟸ nach der Schleife

- Identifikation der Invariante
  - Durch Analyse der Schleife: y wird schrittweise erhöht und die Schleife abgebrochen wird, wenn y den Wert a erreicht
  - Außerdem bekannt: bei jedem Durchlauf gilt (nach Zeile 7) $x = (y+1)^2$ und somit nach Zeile 8: $x = y^2$
  - Wählen daher als Schleifeninvariante: $y \leq a \wedge x = y^2$

Schritt 2: Folgerung aus Vorbedingung

```
1  // Calculates the square of a
2  // nonnegative integer number a
3  // {Q: 0 ≤a}
4      y= 0;
5      x= 0;
6      while (y != a) {
7          x= x+2*y+1;
8          y= y+1;
9      }
10 // {R: x = a^2}
```

- Aufgrund der Vorbedingung des Programmstücks $\{0 \leq a\}$ und der beiden Zuweisungen (Zeile 4 und 5) findet man als Vorbedingung vor Zeile 6:
  $$(0 \leq a) \wedge (y = 0) \wedge (x = 0)$$
- Daraus lässt sich offensichtlich die Schleifeninvariante ableiten:
  $$(0 \leq a) \wedge (y = 0) \wedge (x = 0) \Rightarrow (y \leq a) \wedge (x = y^2)$$

Schritt 3: Invarianz gilt

- Zu zeigen: $\{I \wedge B\}\ P\ \{I\}$
  also:

```
// {I ∧B: y ≤a ∧x=y 2∧y ≠a}
7         x= x+2*y+1;
8         y= y+1;
// {I: y ≤a ∧x=y 2}
```

- Dies kann durch zweimalige Anwendung der Zuweisungsregel gezeigt werden:

```
// {I ∧B: y ≤a ∧x=y 2∧y ≠a}
// {Q5: y+1 ≤a ∧x+2*y+1 = (y+1)2 }
7       x= x+2*y+1;
// {Q6: y+1 ≤a ∧x=(y+1)2 }
8       y= y+1;
// {I: y ≤a ∧x=y 2}
```

Schritt 3: Invarianz gilt (Forts.)

- Jetzt kann gezeigt werden, dass die Gültigkeit Invariante **nach** dem Schleifenkörper aus ihrer Gültigkeit **vor** dem Schleifenkörper folgt

```
        // {I ∧B: y ≤a ∧x=y 2∧y ≠a}
// {Q5: y+1 ≤a ∧x+2*y+1 = (y+1)2 }
7       x= x+2*y+1;
// {Q6: y+1 ≤a ∧x=(y+1)2 }
8       y= y+1;
// {I: y ≤a ∧x=y 2}
```

- Noch zu zeigen:
  1. Aus $y \leq a$ und $y \neq a$ folgt $y < a$ und damit $y + 1 \leq a$
  2. Aus $x = y^2$ folgt $x^2 + 2y + 1 = y^2 + 2y + 1 = (y + 1)^2$

Schritte 4: Nachbedingung

- Zu zeigen: Die Invariante
  - $(y \leq a) \wedge (x = y^2)$

  und
- die Negation der Schleifenbedingung
  - $\neg(y \neq a)$
- erzwingen die Nachbedingung

- D.h., aus $(y \leq a) \wedge (x = y^2) \wedge \neg(y \neq a)$ folgt $(x = y^2) \wedge (y = a)$ und folglich $x = a^2$ □

Gesamter Beweis

	// $\{Q : 0 \leq a\}$	Vorbedingung
	// $\{0 \leq a \wedge 0 = 0 \wedge 0 = 0\}$	logische Folgerung
4	y = 0;	
	// $\{0 \leq a \wedge y = 0 \wedge 0 = 0\}$	Zuweisungsregel
5	x = 0;	
	// $\{0 \leq a \wedge y = 0 \wedge x = 0\}$	Zuweisungsregel
	// $\{I : y \leq a \wedge x = y^2\}$	arithmetische Folgerung
6	`while (y!=a) {`	
	// $\{y \leq a \wedge x = y^2 \wedge y \neq a\}$	Schleifenregel
	// $\{y < a \wedge x = y^2\}$	logische Folgerung
	// $\{(y + 1) \leq a \wedge x + 2y + 1 = (y + 1)^2\}$	arithmetische Folgerung
7	x= x+2*y+1;	
	// $\{(y + 1) \leq a \wedge x = (y + 1)^2\}$	Zuweisungsregel
8	y= y+1;	
	// $\{I : y \leq a \wedge x = y^2\}$	Zuweisungsregel
9	}	
	// $\{y \leq a \wedge x = y^2 \wedge \neg(y \neq a)\}$	Schleifenregel
	// $\{R : x = a^2\}$	logische Folgerung

Unterprogramme

- Unterprogramme sind beweistechnisch schwierig
- Betrachten einfachen Fall: Methode mit einem Wert- und einem Referenzparameter (genauer: *call by value return*)

backgroundcolor,

```
void p(int i, Object o){
  // {Q}
  P
  // {R}
}
```

$$\frac{\{Q\}\, P\, \{R\}}{\{Q[i \leftarrow e, o \leftarrow r]\}\ p(e,r);\ \{R[o \leftarrow r]\}}$$

- Interpretation: beim Aufruf werden die formalen Parameter i und o durch die aktuellen Parameter e und r ersetzt
- Diese Substitution überträgt sich dann auf die Zusicherungen
- Bei Objekten (Referen!z) kann der Wert des aktuellen Parameters durch die Methode verändert werden

Terminierung

- Bisher: partielle Korrektheit wird bewiesen
- Für **totale** Korrektheit muss noch **Terminierung** bewiesen werden

Kritisch bei

- **Rekursion**
  - Die Rekursion muss nach einer endlichen Anzahl abbrechen, d.h. die Rekursionsbasis erreichen
- **Schleifen**
  - Die Schleifenbedingung muss nach einer endlichen Anzahl von Iterationen false werden
  - Es muss sichergestellt sein, dass auch der Schleifenrumpf terminiert (in jeder Iteration)

Terminierungsfunktion

- Zum Beweis der Terminerung einer Schleife muss eine Terminierungsfunktion τ angeben werden:
  $$\tau : V \to \mathbb{N}$$
  - V ist eine Teilmenge der Ausdrücke über die Variablenwerte der Schleife
- Die Terminierungsfunktion muss folgende Eigenschaften besitzen:
  - Ihre Werte sind natürliche Zahlen (einschließlich 0)
  - Jede Ausführung des Schleifenrumpfs verringert ihren Wert (streng monoton fallend)
  - Die Schleifenbedingung ist false, wenn $\tau = 0$

➡ τ ist obere Schranke für die noch ausstehende Anzahl von Schleifendurchläufen

Terminierungsregel

$$\frac{\{\tau = k\} P \{\tau < k\}\ \ \{\tau = 0\} \Rightarrow \neg B\ \ P \text{ terminiert}}{while(B)\, P;\ \{\text{Terminierung}\}}$$

Es ist also zu zeigen:

1. **Streng monotones Fallen von τ**
2. **Die Implikation der Nichterfüllung der Schleifenbedingung B bei niedrigsten τ**
3. **Die Terminierung des Rumpfs P**

Sind diese drei Eigenschaften gezeigt, so terminiert die Schleife

Beispiel: Quadratzahl

```
1 // {Q: 0 ≤a}
2     y= 0;
3     x= 0;
4     while (y != a) {
5         x= x+2*y+1;
6         y= y+1;
7       }
8 // {R: x = a^2 }
```

Wähle als Terminierungsfunktion $\tau = a - y$:

1. τ wird in jedem Durchgang dekrementiert, da y inkrementiert wird und a konstant bleibt.
2. Wenn $\tau = 0$ folgt $y = a$, d.h. die Schleifenbedingung $y \neq a$ ist falsch
3. Schleifenrumph enthält keine Schleifen/Rekursionen/Gotos... ➡ Terminierung trivial

Terminierung bei Rekursion

- Wie bei Schleifen wird eine Terminierungsfunktion τ konstruiert, die mit zunehmender Rekursionstiefe kleiner wird
- Es muss gelten:
  1. Die Werte von sind natürliche Zahlen (inkl. 0)
  2. Bei jedem Aufruf der Methode wird der Wert von echt kleiner
  3. Abbruch bei (spätestens) $\tau = 0$ erzwungen
- **Beispiel**: Fibonacci-Zahlen

```
1 public static int fib(int n){
2    if (n<3) return 1;
3    else return (fib(n-1)+fib(n-2));
4 }
```

- In diesem Fall können wir als Terminierungsfunktion $\tau = n$ wählen

Probleme bei Terminierung

- Betrachten folgendes Beispiel:

```
1 // { x = 4 }
2 while ( x == Summe zweier Primzahlen) x+= 2;
3 // { x > 4 }
```

- Terminiert dieses Programm?
- Goldbachsche Vermutung: Bis heute[1] nicht allgemein bewiesen, aber bis $\approx 10^{18}$ geprüft

Beweise für Terminierung sind nicht immer möglich!

- Diese Aussage ist wiederum bewiesen (➡ **Halteproblem**)

[1] Dezember 2014

9.3 Diskussion

- Beweise sind mindestens so kompliziert wie die Programme selbst, so dass auch ein Beweis Fehler enthalten kann
- Durch Einsatz von Werkzeugen kann die Qualität der Verifikation erheblich gesteigert und der Aufwand erheblich reduziert werden
- Andere Kalküle lassen sich z.T. besser formalisieren (z.B. λ-Kalkül ➡ **funktionale Programmierung**)
- Einige Eigenschaften, die zur fehlerhaften Diensterfüllung führen können, lassen sich mit dem Hoare-Kalkül nicht beweisen:
  - Nebenläufigkeit
  - Ressourcenverbrauch
  - Zeitverhalten

Werkzeuge

- Fortschritte in letzten Jahren
- **Aber:** Keine vollautomatischen Beweiser ➡ maschinelle Unterstützung beim Beweis
- Bekanntestes Tool: Isabelle/HOL
  - Sehr(!) komplex
  - Meta-Tool, Sprache/Logik muss beigebracht werden
- Betrachten **Dafny**
  - Microsoft Research, `http://rise4fun.com/Dafny`
  - Eigene Sprache, nahe an C#/Pascal
  - Erweitert um Spezifikationskonstrukte
  - **Beispiel:** Terminierung

```
function fib (n: int): int
  decreases n;
{
  if n<2 then n else fib(n-2) + fib(n-1)
}
```

```
> Dafny program verifier finished with 1 verified, 0 errors
```