Einführung in die Informatik II Zusicherungskalkül

Prof. Bernd Brügge, Ph.D
Dr. Christian Herzog
Institut für Informatik
Technische Universität München

Sommersemester 2001

11. - 13. Juni 2001

Überblick

- Zusicherungskalkül: formale Beweismethode für Programmeigenschaften
- Zuweisungsaxiom
- Inferenzregeln für
  - bedingte Anweisungen
  - Abschwächungen von Bedingungen
  - while-Schleifen
  - Terminierung für Schleifen
- formale Beweise
- Methode der eingestreuten Zusicherung
- partielle vs. totale Korrektheit
- ein großes Beispiel: Verifikation von bubbleSort
- Vor- und Nachteile des Zusicherungskalküls

Ziel der Vorlesung

- Sie verstehen das Konzept des Zusicherungskalküls.
- Sie können Eigenschaften kleiner Programme formal beweisen.
- Sie können die Methode der eingestreuten Zusicherungen anwenden..
- Sie verstehen, wie man Verträge des Spezifikationsmodells mit Hilfe des Zusicherungskalküls verifizieren kann.
- Sie können Vor- und Nachteile der Verifikation von Verträgen gegenüber der Kontrolle der Einhaltung von Verträgen durch Ausnahmen abschätzen.

Wiederholung: Vertragsbrüche

- Ursachen für Vertragsbrüche:
  - Entwurfsfehler
  - Benutzungsfehler
  - Implementierungfehler
- Was machen wir bei Vertragsbrüchen?
  - Wir leben mit ihnen (während der Laufzeit)
    - z.B. mit Hilfe von Ausnahmen
  - Wir verhindern sie (während der Entwicklung)
    - z.B. durch Programmverifikation
- In dieser Vorlesung konzentrieren wir uns auf die Vermeidung von Implementierungsfehlern durch Verifikation

Verifikation bei imperativen Programme

- In Info1 haben wir nur Eigenschaften *funktionaler* Programme bewiesen (partielle Korrektheit, Terminierung).
- Wesentlich neuer Aspekt bei der imperativen Programmierung ist der *Programmzustand*.

Erinnerung an Info1: **Der Zustand eines Programmes ist die Menge der Werte seiner Variablen.**

- Wenn wir Eigenschaften imperativer Programme beweisen wollen, dann wollen wir in der Regel nachweisen, dass das Programm einen gegebenen **Anfangszustand** in einen gewünschten **Endzustand** überführt.
- Anfangs- und Endzustand werden durch **Prädikate** beschrieben:
  - der Anfangszustand wird durch die **Vorbedingung**,
  - der Endzustand durch die **Nachbedingung** charakterisiert.
- Im sog. *Zusicherungskalkül* wird die Wirkung eines Programmes dadurch beschrieben, wie sich das Programm auf die Gültigkeit von Prädikaten auswirkt.

Beispiel: Wirkung der Zuweisung

- **Fragestellung**:
  - Wie wirkt sich der Zustandsübergang durch eine Zuweisung auf die Gültigkeit von Prädikaten aus?
- Beispiel einer Zuweisung: **x = 7;**
  - Da im Zustand davor $7 > 5$ gilt, gilt nachher auch: $x > 5$
  - Wenn im Zustand vorher $y + z < 7$ gilt, dann danach: $y + z < x$
  - salopp formuliert: *was im Zustand vor der Zuweisung für 7 gilt, gilt nachher für x.*
- Allgemeine Form der Zuweisung: **x = A;**
  - x ist dabei eine Variable, A ist ein Ausdruck.
  - salopp formuliert: *was im Zustand vor der Zuweisung für A gilt, gilt nachher für x*

Beschreibung der Wirkung der Zuweisung

- 1. Versuch, die Wirkung von **x = A;** formal zu beschreiben:
  - ist Q ein Prädikat, das im Zustand vor der Ausführung von **x = A;** gilt.
  - Dann muss man zur Charakterisierung des Zustandes nach der Zuweisung in Q jedes Auftreten von A durch x ersetzen.
  - leider haben wir dazu kein formales Mittel!
- 2. Versuch - diesmal anders herum:
  - ist R ein Prädikat, das im Zustand **nach** der Ausführung von **x = A;** gilt.
  - Dann muss man zur Charakterisierung des Zustandes **vor** der Zuweisung in R jedes Auftreten von x durch A ersetzen.
- Das bedeutet also:
  - Gilt im Zustand vor der Ausführung der Zuweisung **x = A;** das Prädikat **R[A/x]** ,
  - dann gilt im Zustand danach das Prädikat **R**

Der Zusicherungskalkül (C.A.R Hoare / R. Floyd)

- Im **Zusicherungskalkül** ist **{ R[A/x] } x = A { R }** eine wohlgeformte Formel.
- Der Zusicherungskalkül **ergänzt** die Prädikatenlogik um Formeln der Art **{ Q } S { R }**
  - Dabei sind **Q** und **R** Prädikate der Prädikatenlogik
  - **S** ist ein Programm(stück), z.B. eine Zuweisung oder der Rumpf einer Methode.
  - In den Prädikaten **Q** und **R** dürfen Programmvariablen aus S auftreten (als freie Variablen).
- Wenn die Formel **{ Q } S { R }** gilt, bedeutet das:
  - Wenn **S** in einem Zustand ausgeführt wird, in dem das Prädikat **Q** gilt, dann gilt im Zustand nach Ausführung von **S** das Prädikat **R**.
- **Q** und **R** sind Vor- und Nachbedingung von **S**.

Der Zusicherungskalkül

- Der Zusicherungskalkül ist eine Logik, die
  - die Prädikatenlogik (in den Vor- und Nachbedingungen Q und R),
  - den Zustand eines Programmes (durch Programmvariable als freie Variable der Prädikate)
  - und Programm-Code (im Programmstück S) verbindet.
- Im Zusicherungskalkül kann formal bewiesen werden, dass ein Programm seine Spezifikation erfüllt (Verifikation).
- Es kann z.B. bewiesen werden, dass die Implementierung einer Methode den OCL-Vertrag erfüllt.
- Für das formale Beweisverfahren müssen wir die Prädikatenlogik jedoch noch ergänzen
  - um **Axiome** der Art { Q } S { R }
  - und um **Inferenzregeln**, mit denen aus prädikatenlogischen Formeln und aus Formeln der Art { Q } S_i { R } andere Formeln der Art { Q } S { R } abgeleitet werden können.

Copyright 2001 Bernd Brügge, Christian Herzog

Induktive Vorgehensweise im Zusicherungskalkül

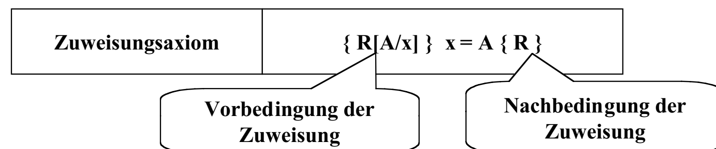
Zur Ableitung der Formel **{ Q } S { R }** gehen wir **induktiv** über den Aufbau von **S** vor:

- Wir betrachten **S** grundsätzlich als eine **Anweisung.**
- Dabei kann **S** eine
  - einfache Anweisung sein (z.B. Zuweisung) oder
  - eine zusammengesetzte Anweisung, die selbst wieder Anweisungen $\mathbf{S_i}$ enthält (z.B. eine bedingte Anweisung).
- Wir beschränken uns in dieser Vorlesung auf
  - Zuweisung
  - Anweisungsfolge
  - bedingte Anweisung und
  - while-Schleife.

Copyright 2001 Bernd Brügge, Christian Herzog

Ableitungen im Zusicherungskalkül

- Zuweisungen liefern im Zusicherungskalkül **Axiome**, also Formeln, die als wahr vorausgesetzt werden (siehe Info1).
  - Die Formel **{ R[A/x] } x = A { R }** wird für alle Prädikate R, für alle Programmvariablen x und für alle korrekten Ausdrücke A vom selben Typ wie x als wahr vorausgesetzt.

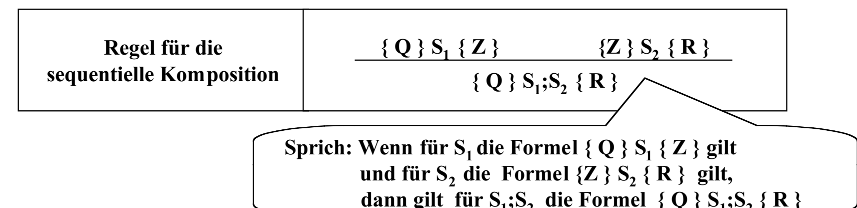


- Zusammengesetzte Anweisungen liefern **Regeln** (Inferenzregeln), wie aus der Gültigkeit von Formeln für einzelne Teil-Anweisungen auf die Gültigkeit der Formel für die Gesamtanweisung geschlossen werden kann.

Copyright 2001 Bernd Brügge, Christian Herzog

Regel für die Anweisungsfolge (sequentielle Komposition)

- S sei nun eine Folge aus zwei Anweisungen: $S_1;S_2$;
- Welche neue Inferenzregel würden wir zur Herleitung der Formel **{ Q }** $\mathbf{S_1;S_2}$ **{ R }** vorschlagen?
- Als Voraussetzungen bieten sich an:
  - eine Formel für S_1, die Q als Vorbedingung hat,
  - eine Formel für S_2, die R als Nachbedingung hat,
  - wobei die Nachbedingung für S_1 zugleich die Vorbedingung für S_2 ist.
- Im Zusicherungskalkül werden genau diese Überlegungen formalisiert:



Copyright 2001 Bernd Brügge, Christian Herzog

Beispiel: die swap-Anweisungsfolge

- Betrachten wir folgendes Programm, das den Inhalt zweier Variablen a und b vertauscht (swap):

  temp = a; a = b; b = temp;
- Wir wollen nun formal beweisen, dass nach Ausführung dieses Programmes b > a gilt, wenn voher a > b galt.
- Wir suchen also einen Beweis (eine Herleitung) für die Formel
  { a > b } temp = a; a = b; b = temp; { b > a } :
  - Zunächst suchen wir für die erste Anweisung temp = a; eine Formel mit passender Vorbedingung a > b.
  - Das Zuweisungsaxiom für temp = a; mit der Vorbedingung a > b lautet **{ a > b } temp = a; { temp > b }**, denn die Substitution (temp > b)[a/temp] liefert gerade a > b.

Erinnerung:

Zuweisungsaxiom	{ R[A/x] } x = A { R }

Beispiel: die swap-Anweisungsfolge (Fortsetzung)

- gesucht: **{ a > b } temp = a; a = b; b = temp; { b > a }**
- wir haben bereits: **{ a > b } temp = a; { temp > b }**
- Um die Kompositionsregel anwenden zu können, suchen wir nun für die zweite Anweisung a = b; eine Formel mit passender Vorbedingung temp > b:
  - Das Zuweisungsaxiom für a = b; mit der Vorbedingung temp > b lautet **{ temp > b } a = b; { temp > a }** , denn die Substitution (temp > a)[b/a] liefert gerade temp > b
- Aus **{ a > b } temp = a; { temp > b }**, **{ temp > b } a = b; { temp > a }** liefert nun die Regel für die sequentielle Komposition:
  { a > b } temp = a; a = b; { temp > a } .

Erinnerung:

Regel für die sequentielle Komposition	{ Q } S_1 { Z } {Z } S_2 { R } / { Q } S_1;S_2 { R }
Zuweisungsaxiom	{ R[A/x] } x = A { R }

Beispiel: die swap-Anweisungsfolge (Fortsetzung)

- gesucht: **{ a > b } temp = a; a = b; b = temp; { b > a }**
- wir haben bereits: **{ a > b } temp = a; a = b; { temp > a }**
- Um die Kompositionsregel noch einmal anwenden zu können, suchen wir nun für die dritte Anweisung b = temp; eine Formel mit passender Vorbedingung temp > a:
  - Das Zuweisungsaxiom für b = temp; mit der Vorbedingung temp > a lautet **{ temp > a } b = temp; { b > a }**, denn die Substitution (b > a)[temp/b] liefert gerade temp > a
- Aus **{ a > b } temp = a; a= b; { temp > a }** und **{ temp > a } b = temp; { b > a }** liefert nun wiederum die Regel für die sequentielle Komposition die gesuchte Formel:
  { a > b } temp = a; a = b; { temp > a } .

Erinnerung:

Regel für die sequentielle Komposition	{ Q } S_1 { Z } {Z } S_2 { R } / { Q } S_1;S_2 { R }
Zuweisungsaxiom	{ R[A/x] } x = A { R }

Ein Beweis für die swap-Anweisungsfolge

- fassen wir den Beweis für die swap-Anweisungsfolge noch einmal zusammen:

(1)	**{ a > b } temp = a; { temp > b }**	Zuweisungsaxiom
(2)	**{ temp > b } a = b; { temp > a }**	Zuweisungsaxiom
(3)	**{ a > b } temp = a; a= b; { temp > a }**	(1), (2) und Komp.-Regel
(4)	**{ temp > a } b = temp; { b > a }**	Zuweisungsaxiom
(5)	**{ a > b } temp = a; a = b; b = temp; { b > a }**	(3), (4) und Komp.-Regel

Erinnerung an Info1: **Ein Beweis ist eine endliche Zeichenkette von wohldefinierten Formeln, wobei jede Formel entweder ein Axiom ist oder durch Anwendung einer Inferenzregel abgeleitet worden ist.**

Inferenzregel zur Abschwächung von Bedingungen

- Oft liefern Axiome oder Regeln stärkere Nachbedingungen, als man benötigt, oder verlangen schwächere Vorbedingungen, als zur Verfügung stehen.
- Unter den Voraussetzungen **{ Q } S_1 { x > 10 }** und **{ x > 0 } S_2 { R }** kann beispielsweise die Regel zur sequentiellen Komposition nicht angewandt werden, da die Nachbedingung der ersten Formel nicht mit der Vorbedingung der zweiten Formel übereinstimmt.
- Hier stellt der Zusicherungskalkül eine Inferenzregel zur Verfügung, die es erlaubt, eine stärkere Vorbedingung als nötig zu verwenden und die Nachbedingung abzuschwächen:

Regel zur Abschwächung von Bedingungen	$\dfrac{Q \rightarrow P \qquad \{P\}\ S\ \{T\} \qquad T \rightarrow R}{\{Q\}\ S\ \{R\}}$

- Wegen $x > 10 \rightarrow x > 0$ folgt im obigen Beispiel wegen dieser Regel aus **{ Q } S_1 { x > 10 }** auch **{ Q } S_1 { x > 0 }** und mit **{ x > 0 } S_2 { R }** und der Komp.-Regel auch **{ Q } S_1;S_1 {R }**

Inferenzregel für die bedingte Anweisung

- Sei nun S die bedingte Anweisung if (B) S_1; else S_2 ;
- Welche weitere Inferenzregel wird für die Formel **{ Q } if (B) S_1; else S_2 { R }** benötigt?
- Wir unterscheiden zwei Fälle:
- In Zuständen, in denen B gilt, wird S_1 ausgeführt.
  - S_1 muss also unter der Vorbedingung $Q \wedge B$ die Nachbedingung R erreichen können: **{ $Q \wedge B$ } S_1 { R }**
- In Zuständen, in denen B nicht gilt, wird S_2 ausgeführt.
  - S_2 muss also unter der Vorbedingung $Q \wedge \neg B$ die Nachbedingung R erreichen können: **{ $Q \wedge \neg B$ } S_2 { R }**
- Im Zusicherungskalkül werden auch diese Überlegungen wieder formalisiert:

Regel für die bedingte Anweisung	$\dfrac{\{Q \wedge B\}\ S_1\ \{R\} \qquad \{Q \wedge \neg B\}\ S_2\ \{R\}}{\{Q\}\ \text{if (B) } S_1\text{; else } S_2\ \{R\}}$

Speziell: Inferenzregel für einseitige bedingte Anweisung

- Betrachten wir nun die einseitige bedingte Anweisung if (B) S;
- Gesucht ist wieder eine Regel, die **{ Q } if (B) S { R }** liefert.
- Auch hier unterscheiden wir wieder zwei Fälle:
- In Zuständen, in denen B gilt, wird S ausgeführt.
  - S muss also unter der Vorbedingung $Q \wedge B$ die Nachbedingung R erreichen können: **{ $Q \wedge B$ } S { R }**
- In Zuständen, in denen B nicht gilt, wird gar keine Anweisung ausgeführt.
  - R muss also direkt aus $Q \wedge \neg B$ folgen: $\mathbf{Q \wedge \neg B \rightarrow R}$
- Im Zusicherungskalkül:

Regel für die einseitige bedingte Anweisung	$\dfrac{\{Q \wedge B\}\ S\ \{R\} \qquad Q \wedge \neg B \rightarrow R}{\{Q\}\ \text{if (B) } S\ \{R\}}$

Erweiterung des swap-Beispiels

- Wir wollen unser swap-Beispiel zu einem „richtigen" Programm erweitern:

  if (a > b) {temp = a; a = b; b = temp;}

  Achtung bei der Schreibweise:
  - Bitte die geschweiften Klammern der Java-Syntax nicht mit denen des Zusicherungskalküls verwechseln!
  - Ebenso bedeutet das Zeichen = im Java-Teil der Formel die Zuweisung, in den Prädikatenteilen der Formel die Gleichheit
- Wir wollen nun formal beweisen, dass nach Ausführung dieses Programmes immer $a \leq b$ gilt. In diesem Fall wird also keine Vorbedingung gestellt.
- Für die Formalisierung verwenden wir die schwächste aller möglichen Vorbedingungen, nämlich true:
  { true } if (a > b) {temp = a; a = b; b = temp;} ***{ a ≤ b }***
- Für den Beweis haben wir die ersten Schritte bereits gezeigt.

Ein Beweis für das erweiterte swap-Programm

Zu beweisen: $\{true\}$ if (a > b) {temp = a; a = b; b = temp;} $\{a \le b\}$

(1)	$\{a > b\}$ temp = a; $\{temp > b\}$	Zuweisungsaxiom
(2)	$\{temp > b\}$ a = b; $\{temp > a\}$	Zuweisungsaxiom
(3)	$\{a > b\}$ temp = a; a = b; $\{temp > a\}$	(1), (2) und Komp.-Regel
(4)	$\{temp > a\}$ b = temp; $\{b > a\}$	Zuweisungsaxiom
(5)	$\{a > b\}$ temp = a; a = b; b = temp; $\{b > a\}$	(3), (4) und Komp.-Regel
(6)	$true \wedge (a > b) \rightarrow a > b$	Arithmetik
(7)	$b > a \rightarrow a \le b$	Arithmetik
(8)	$\{true \wedge (a > b)\}$ temp = a; a = b; b = temp; $\{a \le b\}$	(5)-(7) und Abschw.-Regel
(9)	$true \wedge \neg(a > b) \rightarrow a \le b$	Arithmetik
(10)	$\{true\}$ if (a > b) {temp = a; a = b; b = temp;} $\{a \le b\}$	(8), (9) und Regel einseitig bedingte Anw.

q.e.d

Regel für die einseitige bedingte Anweisung	$\dfrac{\{Q \wedge B\}\ S\ \{R\} \qquad Q \wedge \neg B \rightarrow R}{\{Q\}\ \text{if (B) S}\ \{R\}}$

Beweis in der Form „eingestreuter Zusicherungen“

- Der Beweis wird übersichtlicher, wenn man Vor- und Nachbedingungen als Kommentare direkt in den Programm-Code schreibt:

```
                         //   true
if  (a > b) {
                         //   a > b
    temp = a;            //   temp > b
    a = b;               //   temp > a
    b = temp;            //   b > a
}
                         //   a <= b
```

- Diese Methode nennt man **eingestreute Zusicherungen** oder **annotiertes Programm**.
- Sie gibt dem Zusicherungskalkül seinen Namen.
- Ein annotiertes Programm ist **korrekt annotiert**, wenn sich die Zusicherungen aus den Axiomen und Regeln ergeben.

Inferenzregel für die while-Schleife

- Betrachten wir nun noch die Schleife while (B) S;
- Der Rumpf der Schleife, S, wird nur ausgeführt, wenn die Bedingung B erfüllt ist. B kann also in die Vorbedingung von S aufgenommen werden.
- Falls B nach Ausführung von S wieder gilt, kann S ein weiteres Mal ausgeführt werden. Die Nachbedingung von S sollte also - zusammen mit B - auch Vorbedingung von S sein.
  - Die Voraussetzung an S kann also durch eine Formel der Art **$\{I \wedge B\}\ S\ \{I\}$** festgelegt werden.
- Die Schleife terminiert nur, wenn die Bedingung B nicht mehr erfüllt ist. $\neg B$ kann also in die Nachbedingung der Schleife aufgenommen werden.
- Da I zur Vor- und Nachbedingung von S gehört, gehört es auch zur Vor- und Nachbedingung der Schleife:

Regel für die while-Schleife	$\dfrac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \text{while (B) S}\ \{I \wedge \neg B\}}$

I wird die „Schleifen-Invariante“ genannt.

Bemerkungen zur Schleifen-Invariante I

Regel für die while-Schleife	$\dfrac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \text{while (B) S}\ \{I \wedge \neg B\}}$

- Die Vor- und Nachbedingung der Schleife unterscheiden sich nur durch die Zusicherung, dass am Ende der Schleife die Bedingung B nicht mehr gilt.
- Um mit diesem Mittel die Wirkung der Schleife so genau zu beschreiben, dass daraus weitere Aussagen abgeleitet werden können, muss I sorgfältig gewählt und auf B abgestimmt werden.
- Während in den anderen Fällen die Wahl der Vor- und Nachbedingung eher ein technisches Problem ist, fließen in I die Idee und das Verständnis für den implementierten Algorithmus ein.
- Die Wahl der Schleifen-Invarianten ist die schwierigste Aufgabe bei der Verifikation mit dem Zusicherungskalkül.
- Nur wer den Algorithmus entwirft, kann in der Regel die Schleifen-Invarianten angeben. Implementierung und Verifikation (zumindestens ausführliche Annotation) müssen also Hand in Hand laufen.

Beispiel: Spezifikation der Methode mult2()

- **Aufgabenstellung**:
  - Eine Klasse MathDienste soll eine Methode mult2() bereitstellen, die für nicht negative Parameter x das Doppelte von x als Ergebnis liefert.
- **OCL-Modell**:

```
MathDienste::mult2(int x):int
pre:  0 <= x
post: result = 2*x
```

- **Spezifikation der Schnittstelle von mult2 in Java und Javadoc**:

```
class MathDienste {
   /**
    * @pre  0 <= x
    * @post result = 2*x
    */
   public int mult2 (int x) {...}
}
```


Eine Implementierung von mult2() in Java

- In der folgenden Implementierung der Methode mult2() wird das Ergebnis ohne Verwendung der Multiplikation * berechnet:

```
class MathDienste {
   /**
    * @pre  0 <= x
    * @post result = 2*x
    */
   public int mult2 (int x) {
      int i,z;

      i = 0;
      z = 0;
      while (i < x) {
         z = z + 2;
         i = i + 1;
      }

      return z;
   }
}
```


Verifikation der Methode mult2()

- Wir beweisen mit dem Zusicherungskalkül, dass die Methode mult2() die im OCL-Vertrag bzw. in Javadoc gegebene Spezifikation erfüllt.

```
class MathDienste {
   /**
    * @pre  0 >= 0
    * @post result = 2*x
    */
   public int mult2 (int x) {
      int i,z;

      i = 0;
      z = 0;
      while (i < x) {
         z = z + 2;
         i = i + 1;
      }

      return z;
   }
}
```

- Zunächst tragen wir die Vorbedingung als Zusicherung vor Beginn des Rumpfes ein.

Verifikation der Methode mult2()

- Wir beweisen mit dem Zusicherungskalkül, dass die Methode mult2() die im OCL-Vertrag bzw. in Javadoc gegebene Spezifikation erfüllt.

```
class MathDienste {
   /**
    * @pre  0 <= x
    * @post result = 2*x
    */
   public int mult2 (int x) {
      int i,z;
            // x >= 0
      i = 0;
      z = 0;
      while (i < x) {
         z = z + 2;
         i = i + 1;
      }

      return z;
   }
}
```

- Zunächst tragen wir die Vorbedingung als Zusicherung vor Beginn des Rumpfes ein.
- Das „result" in der Nachbedingung entspricht dem Ausdruck, der mittels „return" von der Methode zurückgegeben wird.
- Der Ausdruck z muss also vor der return-Anweisung denselben Wert haben, wie 2*x
- Wir tragen auch diese Zusicherung in das Programm ein.

Verifikation der Methode mult2()

- Wir beweisen mit dem Zusicherungskalkül, dass die Methode mult2() die im OCL-Vertrag bzw. in Javadoc gegebene Spezifikation erfüllt.

```
class MathDienste {
   /**
    * @pre  0 <= x
    * @post result = 2*x
    */
   public int mult2 (int x) {
      int i,z;
            // x >= 0
      i = 0;
      z = 0;
      while (i < x) {
         z = z + 2;
         i = i + 1;
      }
            // z = 2*x
      return z;
   }
}
```

- Zunächst tragen wir die Vorbedingung als Zusicherung vor Beginn des Rumpfes ein.
- Das „result" in der Nachbedingung entspricht dem Ausdruck, der mittels „return" von der Methode zurückgegeben wird.
- Der Ausdruck z muss also vor der return-Anweisung denselben Wert haben, wie 2*x
- Wir tragen auch diese Zusicherung in das Programm ein.

Verifikation der Methode mult2()

- Wir beweisen mit dem Zusicherungskalkül, dass die Methode mult2() die im OCL-Vertrag bzw. in Javadoc gegebene Spezifikation erfüllt.

```
class MathDienste {
   /**
    * @pre  0 <= x
    * @post result = 2*x
    */
   public int mult2 (int x) {
      int i,z;
            // x >= 0            { Q }
      i = 0;
      z = 0;
      while (i < x) {              S
         z = z + 2;
         i = i + 1;
      }
            // z = 2*x           { R }
      return z;
   }
}
```

- Nun müssen wir die Gültigkeit einer Formel der Form **{ Q } S { R }** nachweisen.
- Dabei ist **Q** die Vorbedingung **x >= 0**
- **S** ist der Rumpf der Methode (ohne return).
- **R** ist die Nachbedingung **z = 2*x**
- Wir beschränken uns im folgenden auf diesen markierten Teil des Methodenrumpfes.
- Unsere Methode ist die der Annotation.

Verifikation der Methode mult2(): Annotation

```
      // 0<=x
i = 0;
z = 0;
while (i < x) {
   z = z + 2;
   i = i + 1;
}
      // z=2*x
```

Regel für die while-Schleife	$\dfrac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \text{while (B) S}\ \{ I \wedge \neg B \}}$

- Zunächst entscheiden wir uns für eine Schleifen-Invariante **I,** um die Schleifen-Regel anwenden zu können.
- **I** soll die Entwurfsidee der Schleife enthalten.
  - Diese Idee ist: **z = 2*i**
- Außerdem muss aus **I ∧¬B** die Nachbedingung **z = 2*x** folgen.
  - Dies gelingt, wenn **I** auch die Bedingung **i <= x** enthält.
- Wir wählen also für **I** das Prädikat **i <= x ∧ z = 2*i**
- Gemäß der Schleifen-Regel tragen wir nun **I** als Annotation
  - vor Beginn der Schleife

Verifikation der Methode mult2(): Annotation

```
      // 0<=x
i = 0;
z = 0;
      // i<=x and z=2*i
while (i < x) {
   z = z + 2;
   i = i + 1;
}
      // z=2*x
```

Regel für die while-Schleife	$\dfrac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \text{while (B) S}\ \{ I \wedge \neg B \}}$

- Zunächst entscheiden wir uns für eine Schleifen-Invariante **I,** um die Schleifen-Regel anwenden zu können.
- **I** soll die Entwurfsidee der Schleife enthalten.
  - Diese Idee ist: **z = 2*i**
- Außerdem muss aus **I ∧¬B** die Nachbedingung **z = 2*x** folgen.
  - Dies gelingt, wenn **I** auch die Bedingung **i <= x** enthält.
- Wir wählen also für **I** das Prädikat **i <= x ∧ z = 2*i**
- Gemäß der Schleifen-Regel tragen wir nun **I** als Annotation
  - vor Beginn der Schleife
  - und am Ende des Schleifen-Rumpfes ein.

Verifikation der Methode mult2(): Annotation

```
        // 0<=x
i = 0;
z = 0;
        // i<=x and z=2*i
while (i < x) {
   z = z + 2;
   i = i + 1;
        // i<=x and z=2*i
}
        // z=2*x
```

Regel für die while-Schleife	$\frac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \text{while (B) S}\ \{ I \wedge \neg B \}}$

- Zunächst entscheiden wir uns für eine Schleifen-Invariante **I,** um die Schleifen-Regel anwenden zu können.
- **I** soll die Entwurfsidee der Schleife enthalten.
  - Diese Idee ist: **z = 2*i**
- Außerdem muss aus $I \wedge \neg B$ die Nachbedingung **z = 2*x** folgen.
  - Dies gelingt, wenn **I** auch die Bedingung **i <= x** enthält.
- Wir wählen also für **I** das Prädikat **i <= x ∧ z = 2*i**
- Gemäß der Schleifen-Regel tragen wir nun **I** als Annotation
  - vor Beginn der Schleife
  - und am Ende des Schleifen-Rumpfes ein.
- Zu Beginn des Schleifen-Rumpfes gilt $I \wedge B$,

Verifikation der Methode mult2(): Annotation

```
        // 0<=x
i = 0;
z = 0;
        // i<=x and z=2*i
while (i < x) {
        // i<=x and z=2*i and i<x
   z = z + 2;
   i = i + 1;
        // i<=x and z=2*i
}
        // z=2*x
```

Regel für die while-Schleife	$\frac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \text{while (B) S}\ \{ I \wedge \neg B \}}$

- Zunächst entscheiden wir uns für eine Schleifen-Invariante **I,** um die Schleifen-Regel anwenden zu können.
- **I** soll die Entwurfsidee der Schleife enthalten.
  - Diese Idee ist: **z = 2*i**
- Außerdem muss aus $I \wedge \neg B$ die Nachbedingung **z = 2*x** folgen.
  - Dies gelingt, wenn **I** auch die Bedingung **i <= x** enthält.
- Wir wählen also für **I** das Prädikat **i <= x ∧ z = 2*i**
- Gemäß der Schleifen-Regel tragen wir nun **I** als Annotation
  - vor Beginn der Schleife
  - und am Ende des Schleifen-Rumpfes ein.
- Zu Beginn des Schleifen-Rumpfes gilt $I \wedge B$,
- am Ende der Schleife $I \wedge \neg B$

Verifikation der Methode mult2(): Annotation

```
        // 0<=x
i = 0;
z = 0;
        // i<=x and z=2*i
while (i < x) {
        // i<=x and z=2*i and i<x
   z = z + 2;
   i = i + 1;
        // i<=x and z=2*i
}       // i<=x and z=2*i and i>=x
        // z=2*x
```

Regel für die while-Schleife	$\frac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \text{while (B) S}\ \{ I \wedge \neg B \}}$

- Dann entscheiden wir uns für eine Schleifen-Invariante **I,** um die Schleifen-Regel anwenden zu können.
- **I** soll die Entwurfsidee der Schleife enthalten.
  - Diese Idee ist: **z = 2*i**
- Außerdem muss aus $I \wedge \neg B$ die Nachbedingung **z = 2*x** folgen.
  - Dies gelingt, wenn **I** auch die Bedingung **i <= x** enthält.
- Wir wählen also für **I** das Prädikat **i <= x ∧ z = 2*i**
- Gemäß der Schleifen-Regel tragen wir nun **I** als Annotation
  - vor Beginn der Schleife
  - und am Ende des Schleifen-Rumpfes ein.
- Zu Beginn des Schleifen-Rumpfes gilt $I \wedge B$,
- am Ende der Schleife $I \wedge \neg B$

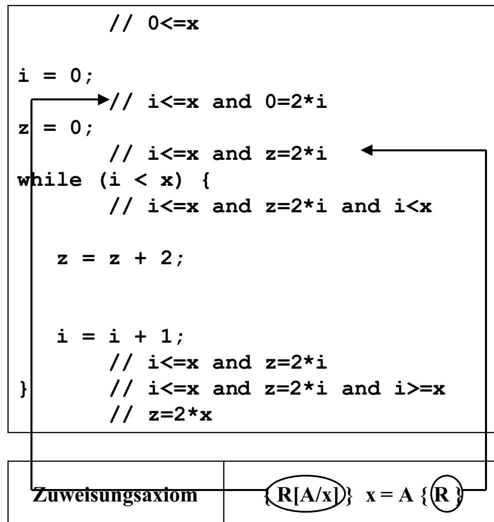
Verifikation der Methode mult2(): Annotation

```
        // 0<=x
i = 0;
z = 0;
        // i<=x and z=2*i
while (i < x) {
        // i<=x and z=2*i and i<x
   z = z + 2;
   i = i + 1;
        // i<=x and z=2*i
}       // i<=x and z=2*i and i>=x
        // z=2*x
```

Zuweisungsaxiom	{ R[A/x] } x = A {R}

- In dieser Situation nun überlegen wir, wie wir über die Zuweisung **z = 0;** die Zusicherung direkt vor der Schleife erreichen.
- Im Zuweisungsaxiom ist dies gerade die Nachbedingung **R**.
- Die Zusicherung vor der Zuweisung muss also entstehen, indem wir in **R** die Variable **z** durch **0** ersetzen.

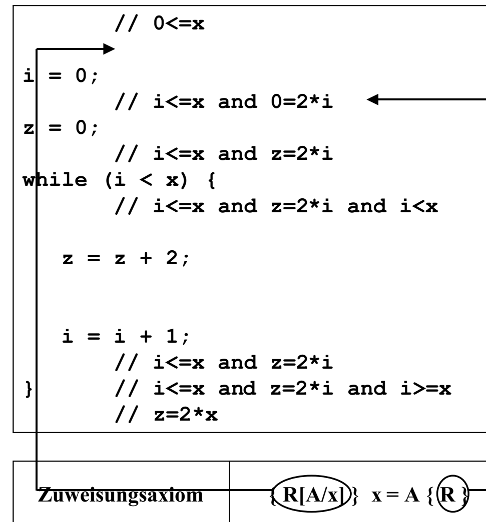
Verifikation der Methode mult2(): Annotation



- In dieser Situation nun überlegen wir, wie wir über die Zuweisung **z = 0;** die Zusicherung direkt vor der Schleife erreichen.
- Im Zuweisungsaxiom ist dies gerade die Nachbedingung **R**.
- Die Zusicherung vor der Zuweisung muss also entstehen, indem wir in **R** die Variable **z** durch **0** ersetzen.

Copyright 2001 Bernd Brügge, Christian Herzog

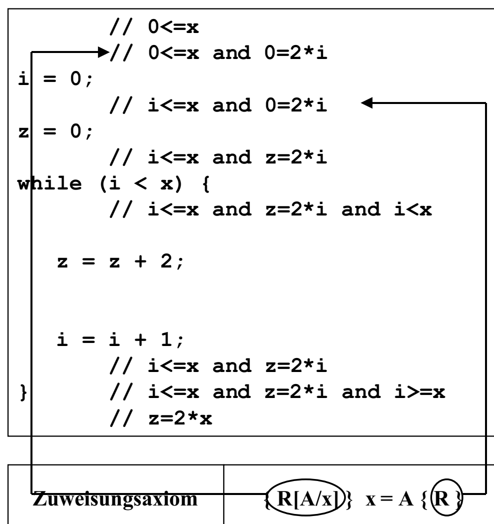
Verifikation der Methode mult2(): Annotation



- In dieser Situation nun überlegen wir, wie wir über die Zuweisung **z = 0;** die Zusicherung direkt vor der Schleife erreichen.
- Im Zuweisungsaxiom ist dies gerade die Nachbedingung **R**.
- Die Zusicherung vor der Zuweisung muss also entstehen, indem wir in **R** die Variable **z** durch **0** ersetzen.
- Analog gehen wir nun auch bei der Zuweisung **i = 0;** vor

Copyright 2001 Bernd Brügge, Christian Herzog

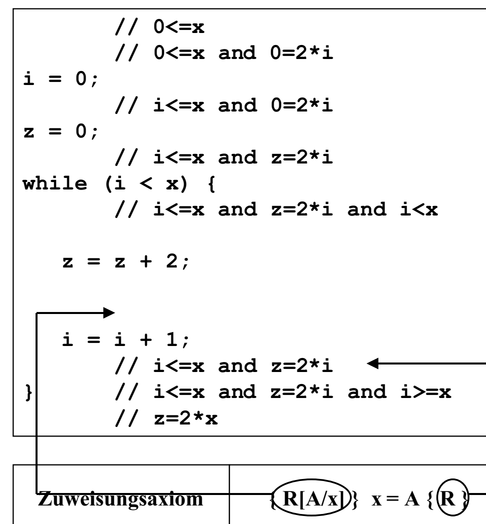
Verifikation der Methode mult2(): Annotation



- In dieser Situation nun überlegen wir, wie wir über die Zuweisung **z = 0;** die Zusicherung direkt vor der Schleife erreichen.
- Im Zuweisungsaxiom ist dies gerade die Nachbedingung **R**.
- Die Zusicherung vor der Zuweisung muss also entstehen, indem wir in **R** die Variable **z** durch **0** ersetzen.
- Analog gehen wir nun auch bei der Zuweisung **i = 0;** vor

Copyright 2001 Bernd Brügge, Christian Herzog

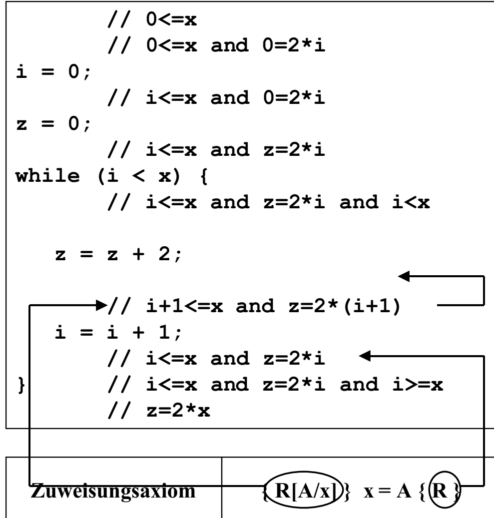
Verifikation der Methode mult2(): Annotation



- Nun machen wir weiter mit dem Rumpf der Schleife.
- Auch hier gehen wir von unten nach oben vor.
- Die Schritte bei der Zuweisung **i = i+1;** sind uns jetzt schon vertraut.

Copyright 2001 Bernd Brügge, Christian Herzog

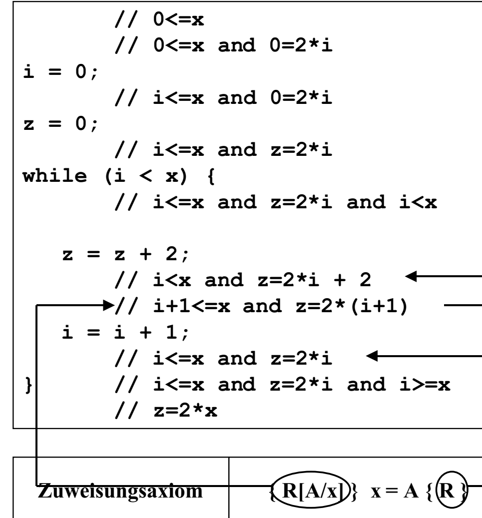
Verifikation der Methode mult2(): Annotation



- Nun machen wir weiter mit dem Rumpf der Schleife.
- Auch hier gehen wir von unten nach oben vor.
- Die Schritte bei der Zuweisung **i = i+1;** sind uns jetzt schon vertraut.
- Jetzt fügen wir noch einen Schritt ein, der das Prädikat äquivalent umformt.

Copyright 2001 Bernd Brügge, Christian Herzog

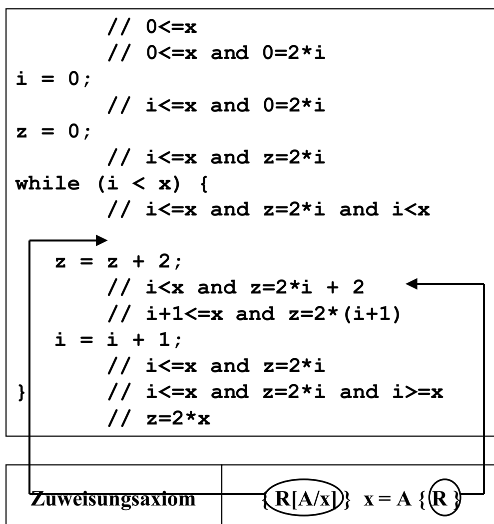
Verifikation der Methode mult2(): Annotation



- Nun machen wir weiter mit dem Rumpf der Schleife.
- Auch hier gehen wir von unten nach oben vor.
- Die Schritte bei der Zuweisung **i = i+1;** sind uns jetzt schon vertraut.
- Jetzt fügen wir noch einen Schritt ein, der das Prädikat äquivalent umformt.

Copyright 2001 Bernd Brügge, Christian Herzog

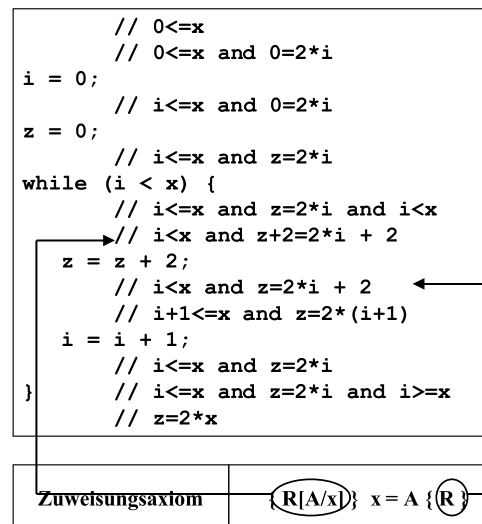
Verifikation der Methode mult2(): Annotation



- Nun machen wir weiter mit dem Rumpf der Schleife.
- Auch hier gehen wir von unten nach oben vor.
- Die Schritte bei der Zuweisung **i = i+1;** sind uns jetzt schon vertraut.
- Jetzt fügen wir noch einen Schritt ein, der das Prädikat äquivalent umformt.
- Der letzte Schritt bezieht sich auf die Zuweisung **z = z+2;**

Copyright 2001 Bernd Brügge, Christian Herzog

Verifikation der Methode mult2(): Annotation



- Nun machen wir weiter mit dem Rumpf der Schleife.
- Auch hier gehen wir von unten nach oben vor.
- Die Schritte bei der Zuweisung **i = i+1;** sind uns jetzt schon vertraut.
- Jetzt fügen wir noch einen Schritt ein, der das Prädikat äquivalent umformt.
- Der letzte Schritt bezieht sich auf die Zuweisung **z = z+2;**
- Das Programmstück ist nun vollständig annotiert.

Copyright 2001 Bernd Brügge, Christian Herzog

mult2(): Korrektheit der Annotation

```
       // 0<=x
       // 0<=x and 0=2*0
i = 0;
       // i<=x and 0=2*i
z = 0;
       // i<=x and z=2*i
while (i < x) {
       // i<=x and z=2*i and i<x
       // i<x and z+2=2*i + 2
   z = z + 2;
       // i<x and z=2*i + 2
       // i+1<=x and z=2*(i+1)
   i = i + 1;
       // i<=x and z=2*i          }
// i<=x and z=2*i and i>=x
       // z=2*x
```

- Nun müssen wir in einem zweiten Durchgang nachweisen, dass das Programm **korrekt** annotiert ist.
- Dazu zeigen wir, dass jeder Übergang von einer Zusicherung zur nächsten den Axiomen und Regeln des Zusicherungskalküls folgt.

mult2(): Korrektheit der Annotation

```
       // 0<=x              (1)
       // 0<=x and 0=2*0  (2)
i = 0;
       // i<=x and 0=2*i
z = 0;
       // i<=x and z=2*i
while (i < x) {
       // i<=x and z=2*i and i<x
       // i<x and z+2=2*i + 2
   z = z + 2;
       // i<x and z=2*i + 2
       // i+1<=x and z=2*(i+1)
   i = i + 1;
       // i<=x and z=2*i          }
// i<=x and z=2*i and i>=x
       // z=2*x
```

- Der Übergang von (1) nach (2) ist eine äquivalente Umformung des Prädikats.
- Die Abschwächungsregel würde hier sogar einen Übergang zu einem schwächeren Prädikat erlauben.

mult2(): Korrektheit der Annotation

```
       // 0<=x
       // 0<=x and 0=2*0  (2)
i = 0;
       // i<=x and 0=2*i  (3)
z = 0;
       // i<=x and z=2*i
while (i < x) {
       // i<=x and z=2*i and i<x
       // i<x and z+2=2*i + 2
   z = z + 2;
       // i<x and z=2*i + 2
       // i+1<=x and z=2*(i+1)
   i = i + 1;
       // i<=x and z=2*i          }
// i<=x and z=2*i and i>=x
       // z=2*x
```

- Der Übergang von (2) nach (3) folgt dem Zuweisungsaxiom:
- Substituiert man in (3) die Variable i durch den Ausdruck 0, so erhält man das Prädikat (2).

Zuweisungsaxiom	{ R[A/x] } x = A { R }

mult2(): Korrektheit der Annotation

```
       // 0<=x
       // 0<=x and 0=2*0
i = 0;
       // i<=x and 0=2*i  (3)
z = 0;
       // i<=x and z=2*i  (4)
while (i < x) {
       // i<=x and z=2*i and i<x
       // i<x and z+2=2*i + 2
   z = z + 2;
       // i<x and z=2*i + 2
       // i+1<=x and z=2*(i+1)
   i = i + 1;
       // i<=x and z=2*i          }
// i<=x and z=2*i and i>=x
       // z=2*x
```

- Auch der Übergang von (3) nach (4) folgt dem Zuweisungsaxiom:
- Substituiert man in (4) die Variable z durch den Ausdruck 0, so erhält man das Prädikat (3).
- Die Kompositions-Regel stellt sicher, dass (3) als Nachbedingung der Zuweisung **i = 0;** zugleich als Vorbedingung der Zuweisung **z = 0;** verwendet werden darf.

Zuweisungsaxiom	{ R[A/x] } x = A { R }

mult2(): Korrektheit der Annotation

```
       // 0<=x
       // 0<=x and 0=2*0
i = 0;
       // i<=x and 0=2*i
z = 0;
    (4)// i<=x and z=2*i
while (i < x) {
    (5)// i<=x and z=2*i and i<x
       // i<x and z+2=2*i + 2
   z = z + 2;
       // i<x and z=2*i + 2
       // i+1<=x and z=2*(i+1)
   i = i + 1;
       // i<=x and z=2*i          }
// i<=x and z=2*i and i>=x
       // z=2*x
```

- Nach der Kompositions-Regel ist (4) als Nachbedingung der Zuweisung **z = 0;** zugleich Vorbedingung für die while-Schleife.
- Folgt man der Regel für Schleifen, so muss (4) also die Rolle der Invarianten I spielen.
- Am Beginn des Rumpfes darf deshalb (4) $\wedge$ B vorausgesetzt werden, das ist gerade Prädikat (5).

Regel für die while-Schleife	$\dfrac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \text{while (B) S}\ \{ I \wedge \neg B \}}$

mult2(): Korrektheit der Annotation

```
       // 0<=x
       // 0<=x and 0=2*0
i = 0;
       // i<=x and 0=2*i
z = 0;
       // i<=x and z=2*i
while (i < x) {
    (5)// i<=x and z=2*i and i<x
    (6)// i<x and z+2=2*i + 2
   z = z + 2;
       // i<x and z=2*i + 2
       // i+1<=x and z=2*(i+1)
   i = i + 1;
       // i<=x and z=2*i          }
// i<=x and z=2*i and i>=x
       // z=2*x
```

- Der Übergang von (5) nach (6) ist eine äquivalente Umformung des Prädikats.

mult2(): Korrektheit der Annotation

```
       // 0<=x
       // 0<=x and 0=2*0
i = 0;
       // i<=x and 0=2*i
z = 0;
       // i<=x and z=2*i
while (i < x) {
       // i<=x and z=2*i and i<x
    (6)// i<x and z+2=2*i + 2
   z = z + 2;
    (7)// i<x and z=2*i + 2
       // i+1<=x and z=2*(i+1)
   i = i + 1;
       // i<=x and z=2*i          }
// i<=x and z=2*i and i>=x
       // z=2*x
```

- Der Übergang von (6) nach (7) folgt wieder dem Zuweisungsaxiom:
- Substituiert man in (7) die Variable z durch den Ausdruck z+2, so erhält man das Prädikat (6).

Zuweisungsaxiom	{ R[A/x] } x = A { R }

mult2(): Korrektheit der Annotation

```
       // 0<=x
       // 0<=x and 0=2*0
i = 0;
       // i<=x and 0=2*i
z = 0;
       // i<=x and z=2*i
while (i < x) {
       // i<=x and z=2*i and i<x
       // i<x and z+2=2*i + 2
   z = z + 2;
    (7)// i<x and z=2*i + 2
    (8)// i+1<=x and z=2*(i+1)
   i = i + 1;
       // i<=x and z=2*i          }
// i<=x and z=2*i and i>=x
       // z=2*x
```

- Der Übergang von (7) nach (8) ist eine äquivalente Umformung des Prädikats.

mult2(): Korrektheit der Annotation

```
       // 0<=x
       // 0<=x and 0=2*0
i = 0;
       // i<=x and 0=2*i
z = 0;
       // i<=x and z=2*i
while (i < x) {
       // i<=x and z=2*i and i<x
       // i<x and z+2=2*i + 2
   z = z + 2;
       // i<x and z=2*i + 2
    (8)// i+1<=x and z=2*(i+1)
   i = i + 1;
    (9)// i<=x and z=2*i          }
// i<=x and z=2*i and i>=x
       // z=2*x
```

- Der Übergang von (8) nach (9) folgt wieder der Kompositions-Regel und dem Zuweisungsaxiom:
- Substituiert man in (9) die Variable i durch den Ausdruck i+1, so erhält man das Prädikat (8).

Zuweisungsaxiom	{ R[A/x] } x = A { R }

mult2(): Korrektheit der Annotation

```
       // 0<=x
       // 0<=x and 0=2*0
i = 0;
       // i<=x and 0=2*i
z = 0;
       // i<=x and z=2*i
while (i < x) {
       // i<=x and z=2*i and i<x
       // i<x and z+2=2*i + 2
   z = z + 2;
       // i<x and z=2*i + 2
       // i+1<=x and z=2*(i+1)
   i = i + 1;
    (9)// i<=x and z=2*i          }
(10)// i<=x and z=2*i and i>=x
       // z=2*x
```

- Der Übergang von (9) nach (10) folgt der Regel für Schleifen:
- Prädikat (9) ist die Invariante I am Ende des Rumpfes.
- Damit ist die Voraussetzung für die Schleifen-Regel nachgewiesen.
- Nach der Schleife gilt deshalb (9) $\wedge \neg$B, das ist gerade Prädikat (10).

Regel für die while-Schleife	$\dfrac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \text{while (B) S}\ \{I \wedge \neg B \}}$

mult2(): Korrektheit der Annotation

```
       // 0<=x
       // 0<=x and 0=2*0
i = 0;
       // i<=x and 0=2*i
z = 0;
       // i<=x and z=2*i
while (i < x) {
       // i<=x and z=2*i and i<x
       // i<x and z+2=2*i + 2
   z = z + 2;
       // i<x and z=2*i + 2
       // i+1<=x and z=2*(i+1)
   i = i + 1;
       // i<=x and z=2*i          }
(10)// i<=x and z=2*i and i>=x
   (11)// z=2*x
```

- Der Übergang von (10) nach (11) ist wieder Umformung nach der Abschwächungsregel.

q.e.d

Zusammenfassung der Vorgehensweise bei der Verifikation der Methode mult2()

- Aus dem in OCL bzw. Javadoc spezifizierten Vertrag für mult2() haben wir die Prädikate Q und R gewonnen, die wir in die Formel **{ Q } S { R }** einsetzten.
  - S ist dabei der Methodenrumpf
- Danach wurde der Programmcode mit Zusicherungen annotiert:
  - Erster Schritt war die Suche nach der Schleifen-Invariante I.
  - Hatte man I gefunden, so gab die Schleifen-Regel an, wo die Annotationen I, I $\wedge$ B bzw. I $\wedge\neg$B einzufügen sind.
  - Die weiteren Annotationen richteten sich i.w. nach dem Zuweisungsaxiom:
    - von unten nach oben wurden jeweils die durch Substitution entstandenen Annotationen eingefügt.
- In einem zweiten Durchlauf von oben nach unten wurden alle Annotationen auf Korrektheit überprüft.
  - Dabei wurde meist das vorherige Vorgehen nach Regeln oder Axiomen noch einmal bestätigt.

Partielle versus totale Korrektheit

- Die Gültigkeit der Formel **{ Q } S { R }** bedeutet:
  - wenn S in einen Zustand ausgeführt wird, in dem das Prädikat Q gilt, dann gilt nach der Ausführung von S das Prädikat R.
- Der Zustand, in dem R gilt, wird aber nur erreicht, wenn die Ausführung von S **terminiert**.

Regel für die while-Schleife	$\dfrac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \textbf{while}\ (B)\ S\ \{ I \wedge \neg B \}}$

- Insbesondere bei der Schleifen-Regel gilt: Die Nachbedingung $I \wedge \neg B$ wird nur erreicht, wenn die Schleife terminiert.
- Die Schleifen-Regel eignet sich nur zum Nachweis der **partiellen Korrektheit**.
- Die Terminierung der Schleife (**totale Korrektheit)** muss zusätzlich nachgewiesen werden.

Copyright 2001 Bernd Brügge, Christian Herzog

Nachweis der Terminierung einer Schleife

Regel für die while-Schleife	$\dfrac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \textbf{while}\ (B)\ S\ \{ I \wedge \neg B \}}$

- Für die partielle Korrektheit muss für den Schleifenrumpf S gelten: $\{ I \wedge B \}\ S\ \{ I \}$
- Für den Nachweis der Terminierung der Schleife führen wir nun noch einen ganzzahligen Ausdruck A ein:
- Falls für ganze Zahlen a gilt
  - $\{ I \wedge B \wedge (A = a+1) \}\ S\ \{ I \wedge (A \leq a) \}$
    (der Wert von A nimmt also bei jedem Schleifendurchlauf ab)

  und
  - $I \wedge (A \leq 0) \rightarrow \neg B$
    (nur wenn A einen positiven Wert hat, kann B weiter gelten)

  dann terminiert die Schleife.
- Der Ausdruck A spielt die Rolle der Abstiegsfunktion beim Nachweis der Terminierung rekursiver Aufrufe (vgl. Info I).

Copyright 2001 Bernd Brügge, Christian Herzog

Beispiel für die Terminierung einer Schleife

- $\{ I \wedge B \wedge (A = a+1) \}\ S\ \{ I \wedge (A \leq a) \}$ und $I \wedge (A \leq 0) \rightarrow \neg B$

```
      // i<=x and z=2*i
while (i < x) {
       // i<=x and z=2*i and i<x
       // i<x and z+2=2*i + 2
 z = z + 2;
       // i<x and z=2*i + 2
       // i+1<=x and z=2*(i+1)
 i = i + 1;
       // i<=x and z=2*i
}      // i<=x and z=2*i and i>=x
```

- Im Beispiel ist der Bereich markiert, in dem der Nachweis für $\{ I \wedge B \}\ S\ \{ I \}$ geführt wird.
- Wählen wir nun für **A** den Ausdruck **x - i**
- Dann gilt: $I \wedge (x - i \leq 0) \rightarrow \neg(i < x)$
- $\{ I \wedge B \wedge (x - i = a+1) \}\ S\ \{ I \wedge (x - i \leq a) \}$ lässt sich nachweisen:

```
       // i<=x and z=2*i and i<x and x-i = a+1
       // i<x and z+2=2*i + 2 and x-i = a+1
 z = z + 2;
       // i<x and z=2*i + 2 and x-i = a+1
       // i+1<=x and z=2*(i+1) and x-(i+1) = a
 i = i + 1;
       // i<=x and z=2*i and x-i = a
       // i<=x and z=2*i and x-i <= a
```

Copyright 2001 Bernd Brügge, Christian Herzog

Der BubbleSort-Algorithmus aus Info 1

```
/**
  * Ziel: Sortiere die Werte von arr in aufsteigender Reihenfolge
  * Vorbedingung: arr ist nicht null.
  * Nachbedingung: Die Werte arr[0]...arr[arr.length-1] sind in aufsteigender
  * Reihenfolge sortiert.
  */
public void bubbleSort(int[] arr) {
    int temp;                                        // Temporäre Variable für Tausch
    for (int pass = 1; pass < arr.length; pass++)   // Für jede Runde
      for (int pair = 1; pair < arr.length; pair++)  // Für jedes Paar
        if (arr[pair-1] > arr[pair]) {              / / Vergleiche die Werte
          temp = arr[pair-1];                       //  und vertausche, falls
          arr[pair-1] = arr[pair];                  / /  notwendig
          arr[pair] = temp;
        } // if
} // bubbleSort( )
```

- Dies ist die Methode bubbleSort aus Info I (Folie 10/27).

Copyright 2001 Bernd Brügge, Christian Herzog

Der BubbleSort-Algorithmus - optimiert

```
/**
  * Ziel: Sortiere  die  Werte von arr in aufsteigender Reihenfolge
  * Vorbedingung: arr ist nicht null.
  * Nachbedingung:  Die Werte arr[0]…arr[arr.length-1] sind in aufsteigender
  * Reihenfolge sortiert.
  */
public void bubbleSort(int[]  arr) {
      int temp;                                     // Temporäre Variable für Tausch
      for (int pass = 1; pass < arr.length; pass++)    //  Für jede Runde
         for  (int pair = 1; pair < arr.length-pass+1; pair++)  //  Für jedes Paar
            if  (arr[pair-1]  > arr[pair]) {          / /  Vergleiche die Werte
               temp = arr[pair-1];                  //    und vertausche, falls
               arr[pair-1]  = arr[pair];            / /   notwendig
               arr[pair] =  temp;
            } //  if
}  // bubbleSort( )
```

- Dies ist derselbe Algorithmus, in leicht optimierter Version
  - einige unnötige Vergleiche werden vermieden

Der BubbleSort-Algorithmus - optimiert


```
/**
  * Ziel: Sortiere  die  Werte von arr in aufsteigender Reihenfolge
  * Vorbedingung: arr ist nicht null.
  * Nachbedingung:  Die Werte arr[0]…arr[arr.length-1] sind in aufsteigender
  * Reihenfolge sortiert.
  */
public void bubbleSort(int[]  arr) {
      int temp;                                     // Temporäre Variable für Tausch
      for (int pass = 1; pass < arr.length; pass++)    //  Für jede Runde
         for  (int pair = 1; pair < arr.length-pass+1; pair++)  //  Für jedes Paar
            if  (arr[pair-1]  > arr[pair]) {          / /  Vergleiche die Werte
               temp = arr[pair-1];                  //    und vertausche, falls
               arr[pair-1]  = arr[pair];            / /   notwendig
               arr[pair] =  temp;
            } //  if
}  // bubbleSort( )
```


- Vor- und Nachbedingungen sind informell formuliert.
- Um die Korrektheit des Algorithmus zu beweisen, müssen wir die Prädikate formalisieren.

Ein Vertrag für den BubbleSort-Algorithmus

```
/**
  * Ziel: Sortiere  die  Werte von arr in aufsteigender Reihenfolge
  * Vorbedingung: arr ist nicht null.
  * @post Sequence  (  0..arr.length-2  )->forall ( i |  arr[i]  <= arr[i+1]  )
  *
  */
public void bubbleSort(int[]  arr) {
      int temp;                                     // Temporäre Variable für Tausch
      for (int pass = 1; pass < arr.length; pass++)    //  Für jede Runde
         for  (int pair = 1; pair < arr.length-pass+1; pair++)  //  Für jedes Paar
            if  (arr[pair-1]  > arr[pair]) {          / /  Vergleiche die Werte
               temp = arr[pair-1];                  //    und vertausche, falls
               arr[pair-1]  = arr[pair];            / /   notwendig
               arr[pair] =  temp;
            } //  if
}  // bubbleSort( )
```

- Für die Formulierung der Nachbedingung verwenden wir OCL.
- Sequence (m..n) beschreibt in OCL die Sequenz der Zahlen von m bis n.

Ein Vertrag für den BubbleSort-Algorithmus

```
/**
  * Ziel: Sortiere  die  Werte von arr in aufsteigender Reihenfolge
  * @pre   true
  * @post Sequence  (  0..arr.length-2  )->forall ( i |  arr[i]  <= arr[i+1]  )
  *
  */
public void bubbleSort(int[]  arr) {
      int temp;                                     // Temporäre Variable für Tausch
      for (int pass = 1; pass < arr.length; pass++)    //  Für jede Runde
         for  (int pair = 1; pair < arr.length-pass+1; pair++)  //  Für jedes Paar
            if  (arr[pair-1]  > arr[pair]) {          / /  Vergleiche die Werte
               temp = arr[pair-1];                  //    und vertausche, falls
               arr[pair-1]  = arr[pair];            / /   notwendig
               arr[pair] =  temp;
            } //  if
}  // bubbleSort( )
```

- die Vorbedingung ersetzen wir durch true.
  - Dass arr nicht null ist, setzen wir stillschweigend voraus.
  - Wir müssten den Kalkül noch weiter formalisieren, um auch diese Voraussetzung explizit zu machen.

BubbleSort: Syntaktische Umformungen

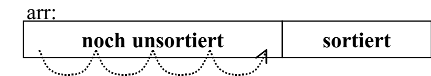
```
    // true
pass = 1;
while (pass < n) {
  pair = 1;
  while (pair < n-pass+1) {
    if (arr[pair-1] > arr[pair]) {
      temp = arr[pair-1];
      arr[pair-1] = arr[pair];
      arr[pair] = temp;
    }
    pair = pair + 1;
  }
  pass = pass + 1
}
//  Sequence(0..n-2)->forall(i|arr[i] <= arr[i+1])
```

- Dies ist der Rumpf der Methode bubbleSort, wobei
  - die for-Schleifen durch äquivalente while-Schleifen ersetzt sind,
  - pair++ und pass++ durch entsprechende Zuweisungen ersetzt sind,
  - Vor- und Nachbedingung als Zusicherungen angegeben sind,
  - arr.length mit n abgekürzt ist.

BubbleSort: Suche nach den Schleifen-Invarianten

```
    // true
pass = 1;

while (pass < n) {

  pair = 1;

  while (pair < n-pass+1) {

    if (arr[pair-1] > arr[pair]) {
      temp = arr[pair-1];
      arr[pair-1] = arr[pair];
      arr[pair] = temp;
    }
    pair = pair + 1;

  }
  pass = pass + 1

}
//  Sequence(0..n-2)->
//ff   forall(i|arr[i] <= arr[i+1])
```

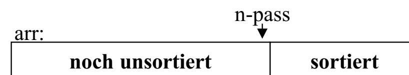
- Um die Schleifen-Invarianten zu finden müssen wir die Idee beim Entwurf des Algorithmus kennen:



- Bei jedem Durchlauf durch die äußere Schleife wandert das größte Element aus dem noch nicht sortierten linken Teil der Reihung nach rechts und vergrößert dort den bereits sortierten Teil.
- Vor und nach jedem Durchlauf durch die äußere Schleife gilt also:
  - Der rechte Teil ist sortiert.
  - Kein Element im linken Teil ist größer als eines im rechten Teil.
  - Die Trennstelle ist beim Index n-pass.

BubbleSort: Invariante der äußeren Schleife:

```
    // true
pass = 1;

while (pass < n) {

  pair = 1;

  while (pair < n-pass+1) {

    if (arr[pair-1] > arr[pair]) {
      temp = arr[pair-1];
      arr[pair-1] = arr[pair];
      arr[pair] = temp;
    }
    pair = pair + 1;

  }
  pass = pass + 1

}
//  Sequence(0..n-2)->
//ff   forall(i|arr[i] <= arr[i+1])
```



- Wir haben also als Invariante für die äußere Schleife zunächst:
  - Rechts von n-pass ist arr sortiert

  und
  - kein Element links von n-pass ist größer als eines rechts davon.
- Etwas formaler:
  - für alle i aus [0, n-2] gilt: $i > n\text{-}pass \rightarrow arr[i] \leq arr[i+1]$

  und
  - für alle i und j aus [0, n-1] gilt: $i \leq n\text{-}pass \wedge j > n\text{-}pass \rightarrow arr[i] \leq arr[j]$

BubbleSort: Invariante der äußeren Schleife:

```
    // true
pass = 1;

while (pass < n) {

  pair = 1;

  while (pair < n-pass+1) {

    if (arr[pair-1] > arr[pair]) {
      temp = arr[pair-1];
      arr[pair-1] = arr[pair];
      arr[pair] = temp;
    }
    pair = pair + 1;

  }
  pass = pass + 1

}
//  Sequence(0..n-2)->
//ff   forall(i|arr[i] <= arr[i+1])
```

- Wir haben also als Invariante für die äußere Schleife zunächst:
  - für alle i aus [0, n-2] gilt: $i > n\text{-}pass \rightarrow arr[i] \leq arr[i+1]$

  und
  - für alle i und j aus [0, n-1] gilt: $i \leq n\text{-}pass \wedge j > n\text{-}pass \rightarrow arr[i] \leq arr[j]$
- Hinzu kommt noch (um nach der Schleife pass=n ableiten zu können)

  und $pass \leq n$
- Dies nun formal in OCL-Notation:

```
Sequence(0..n-2)->forall(i| i>n-pass implies arr[i]<=arr[i+1] )
and  Sequence(0..n-1)->forall(i,j| i<=n-pass and j>n-pass
                                    implies arr[i]<=arr[j] )
and  pass<=n
```

BubbleSort: Invariante der äußeren Schleife:

```
    // true
pass = 1;

while (pass < n) {

  pair = 1;

  while (pair < n-pass+1) {

    if (arr[pair-1] > arr[pair]) {
      temp = arr[pair-1];
      arr[pair-1] = arr[pair];
      arr[pair] = temp;
    }
    pair = pair + 1;

  }
  pass = pass + 1

}
//  Sequence(0..n-2)->
//ff   forall(i|arr[i] <= arr[i+1])
```

- Wir tragen nun im Programm die Zusicherungen ein, die sich aus der Schleifenregel ergeben.
- Um es einigermaßen übersichtlich zu gestalten, kürzen wir die Invariante für die äußere Schleife mit I1 ab.
- Für die Nachbedingung schreiben wir kürzer SORTIERT.

```
Sequence(0..n-2)->forall(i| i>n-pass implies arr[i]<=arr[i+1] )
and  Sequence(0..n-1)->forall(i,j| i<=n-pass and j>n-pass
                                   implies arr[i]<=arr[j] )
and  pass<=n
```


BubbleSort: Annotation analog Regel für äußeres while

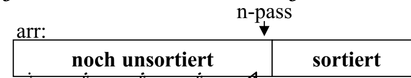
```
    // true
pass = 1;
    // I1
while (pass < n) {
    // I1 and pass<n
  pair = 1;

  while (pair < n-pass+1) {

    if (arr[pair-1] > arr[pair]) {
      temp = arr[pair-1];
      arr[pair-1] = arr[pair];
      arr[pair] = temp;
    }
    pair = pair + 1;

  }
  pass = pass + 1
    // I1
}   // I1 and pass>=n
    // SORTIERT
```

- Wir tragen nun im Programm die Zusicherungen ein, die sich aus der Schleifenregel ergeben.
- Um es einigermaßen übersichtlich zu gestalten, kürzen wir die Invariante für die äußere Schleife mit I1 ab.
- Für die Nachbedingung schreiben wir kürzer SORTIERT.

```
Sequence(0..n-2)->forall(i| i>n-pass implies arr[i]<=arr[i+1] )
and  Sequence(0..n-1)->forall(i,j| i<=n-pass and j>n-pass
                                   implies arr[i]<=arr[j] )
and  pass<=n
```


BubbleSort: Suche der Invariante für die innere Schleife

```
    // true
pass = 1;
    // I1
while (pass < n) {
    // I1 and pass<n
  pair = 1;

  while (pair < n-pass+1) {

    if (arr[pair-1] > arr[pair]) {
      temp = arr[pair-1];
      arr[pair-1] = arr[pair];
      arr[pair] = temp;
    }
    pair = pair + 1;

  }
  pass = pass + 1
    // I1
}   // I1 and pass>=n
    // SORTIERT
```



- Die Idee beim Entwurf des Algorithmus war, in der inneren Schleife den linken unsortierten Teil zu durchlaufen, und das bis dahin gefundene größte Element (bubble) jeweils mitzunehmen.
- Zum Durchlauf wird der Index pair verwendet.
- Invariante für innere Schleife:
  - kein Element links von pair ist größer als arr[pair-1]

  und
  - pair wird nicht größer als n-pass+1

  und (nicht vergessen)
  - es gilt I1

BubbleSort: Invariante für die innere Schleife

```
    // true
pass = 1;
    // I1
while (pass < n) {
    // I1 and pass<n
  pair = 1;

  while (pair < n-pass+1) {

    if (arr[pair-1] > arr[pair]) {
      temp = arr[pair-1];
      arr[pair-1] = arr[pair];
      arr[pair] = temp;
    }
    pair = pair + 1;

  }
  pass = pass + 1
    // I1
}   // I1 and pass>=n
    // SORTIERT
```

- Invariante für innere Schleife:
  - kein Element links von pair ist größer als arr[pair-1]

  und

  pair wird nicht größer als n-pass+1

  und (nicht vergessen)
  - es gilt I1
- Formaler:
  - I1

  und
  - für alle i aus [0, pair-2] gilt: $arr[i] \leq arr[pair-1]$

  und
  - $pair \leq n\text{-}pass+1$
- OCL-Notation:

```
I1  and  Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
    and  pair <= n-pass+1
```

BubbleSort: Invariante für die innere Schleife

```
    // true
pass = 1;
    // I1
while (pass < n) {
    // I1 and pass<n
  pair = 1;

  while (pair < n-pass+1) {

    if (arr[pair-1] > arr[pair]) {
      temp = arr[pair-1];
      arr[pair-1] = arr[pair];
      arr[pair] = temp;
    }
    pair = pair + 1;

  }
  pass = pass + 1
    // I1
}   // I1 and pass>=n
    // SORTIERT
```

- Nun wollen wir das Programm um die Zusicherungen für die Invariante der inneren Schleife ergänzen.
- Dazu kürzen wir diese Invariante ab mit I1 and I2

- OCL-ähnlich:

```
I1  and  Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
    and  pair <= n-pass+1
```

Copyright 2001 Bernd Brügge, Christian Herzog

BubbleSort: Annotation für innere Schleife

```
    // true
pass = 1;
    // I1
while (pass < n) {
    // I1 and pass<n
  pair = 1;
    // I1 and I2
  while (pair < n-pass+1) {
    // I1 and I2 and pair<n-pass-1
    if (arr[pair-1] > arr[pair]) {
      temp = arr[pair-1];
      arr[pair-1] = arr[pair];
      arr[pair] = temp;
    }
    pair = pair + 1;
    // I1 and I2
  } // I1 and I2 and pair>=n-pass+1
  pass = pass + 1
    // I1
}   // I1 and pass>=n
    // SORTIERT
```

- Nun wollen wir einen formalen Beweis für die Korrektheit von bubbleSort angeben.
- Allerdings müssen wir uns dabei auf ein Teilproblem beschränken.
- Wir wählen den Kern des Algorithmus, den Rumpf der inneren Schleife.
- Wir zeigen, dass der Rumpf I2 invariant lässt.

Copyright 2001 Bernd Brügge, Christian Herzog

BubbleSort: Beschränkung auf ein Teilproblem

```
    // true
pass = 1;
    // I1
while (pass < n) {
    // I1 and pass<n
  pair = 1;
    // I1 and I2
  while (pair < n-pass+1) {
    // I1 and I2 and pair<n-pass-1
    if (arr[pair-1] > arr[pair]) {
      temp = arr[pair-1];
      arr[pair-1] = arr[pair];
      arr[pair] = temp;
    }
    pair = pair + 1;
    // I1 and I2
  } // I1 and I2 and pair>=n-pass+1
  pass = pass + 1
    // I1
}   // I1 and pass>=n
    // SORTIERT
```

- Zum Beweis der Methode bubbleSort werden wir uns nun nur auf ein Teilproblem beschränken.
- Allerdings müssen wir uns dabei auf ein Teilproblem beschränken.
- Wir wählen den Kern des Algorithmus, den Rumpf der inneren Schleife.
- Wir zeigen, dass der Rumpf I2 invariant lässt.

Copyright 2001 Bernd Brügge, Christian Herzog

BubbleSort: Das Teilproblem

```
    // I2
if (arr[pair-1] > arr[pair]) {
  temp = arr[pair-1];
  arr[pair-1] = arr[pair];
  arr[pair] = temp;
}
pair = pair + 1;
    // I2
```

- Die ausführliche Schreibweise ergibt:

```
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
 //ff       and pair <= n-pass+1        (durchgestrichen)
if (arr[pair-1] > arr[pair]) {
  temp = arr[pair-1];
  arr[pair-1] = arr[pair];
  arr[pair] = temp;
}
pair = pair + 1;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
 //ff       and pair<=n-pass+1          (durchgestrichen)
```

- Um Platz auf den Folien zu sparen, werden wir dabei auch dieses Problem noch einmal verkleinern

Copyright 2001 Bernd Brügge, Christian Herzog

BubbleSort: Beweis des Teilproblems

```
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
if (arr[pair-1] > arr[pair]) {


  temp = arr[pair-1];


  arr[pair-1] = arr[pair];


  arr[pair] = temp;



}



pair = pair + 1;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
```

- Wir annotieren nun dieses Programm Schritt für Schritt
- Jeden Schritt begründen wir durch Regeln oder Axiome
- dadurch erhalten wir einen Beweis

BubbleSort: Beweis des Teilproblems

```
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
if (arr[pair-1] > arr[pair]) {
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
 //ff                       and arr[pair-1]>arr[pair]
  temp = arr[pair-1];


  arr[pair-1] = arr[pair];


  arr[pair] = temp;



}



pair = pair + 1;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
```

- Begründung: Voraussetzung der Regel für einseitig bedingte Anweisung

BubbleSort: Beweis des Teilproblems

```
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
if (arr[pair-1] > arr[pair]) {
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
 //ff                       and arr[pair-1]>arr[pair]
  temp = arr[pair-1];
 // Sequence(0..pair-2)->forall( i | arr[i]<=temp )
 //ff                       and temp>arr[pair]
  arr[pair-1] = arr[pair];


  arr[pair] = temp;



}



pair = pair + 1;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
```

- Begründung: Umkehrung der Substitution [arr[pair-1]/temp]
  gemäß Zuweisungsaxiom

BubbleSort: Beweis des Teilproblems

```
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
if (arr[pair-1] > arr[pair]) {
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
 //ff                       and arr[pair-1]>arr[pair]
  temp = arr[pair-1];
 // Sequence(0..pair-2)->forall( i | arr[i]<=temp )
 //ff                       and temp>arr[pair]
  arr[pair-1] = arr[pair];
 // Sequence(0..pair-2)->forall( i | arr[i]<=temp )
 //ff                       and temp>arr[pair-1]
  arr[pair] = temp;



}



pair = pair + 1;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
```

- Begründung: Umkehrung der Substitution [arr[pair]/arr[pair-1]]
  gemäß Zuweisungsaxiom

BubbleSort: Beweis des Teilproblems

```
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
if (arr[pair-1] > arr[pair]) {
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
 //ff                       and arr[pair-1]>arr[pair]
  temp = arr[pair-1];
 // Sequence(0..pair-2)->forall( i | arr[i]<=temp )
 //ff                       and temp>arr[pair]
  arr[pair-1] = arr[pair];
 // Sequence(0..pair-2)->forall( i | arr[i]<=temp )
 //ff                       and temp>arr[pair-1]
  arr[pair] = temp;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair] )
 //ff                       and arr[pair]>arr[pair-1]

}


pair = pair + 1;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
```

- Begründung: Umkehrung der Substitution [temp/arr[pair]]
  gemäß Zuweisungsaxiom

BubbleSort: Beweis des Teilproblems

```
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
if (arr[pair-1] > arr[pair]) {
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
 //ff                       and arr[pair-1]>arr[pair]
  temp = arr[pair-1];
 // Sequence(0..pair-2)->forall( i | arr[i]<=temp )
 //ff                       and temp>arr[pair]
  arr[pair-1] = arr[pair];
 // Sequence(0..pair-2)->forall( i | arr[i]<=temp )
 //ff                       and temp>arr[pair-1]
  arr[pair] = temp;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair] )
 //ff                       and arr[pair]>arr[pair-1]
 // Sequence(0..pair-1)->forall( i | arr[i]<=arr[pair] )
}


pair = pair + 1;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
```

- Begründung: Abschwächung (>= statt >) und Umformung des All-Quantors

BubbleSort: Beweis des Teilproblems

```
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )            Q
if (arr[pair-1] > arr[pair]) {
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )            Q ∧ B
 //ff                       and arr[pair-1]>arr[pair]
  temp = arr[pair-1];                                                  S
 // Sequence(0..pair-2)->forall( i | arr[i]<=temp )
 //ff
  arr[pair-1] = arr[pair
 // Sequence(0..pair-2)-
 //ff
  arr[pair] = temp;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair] )
 //ff                       and arr[pair]>arr[pair-1]
 // Sequence(0..pair-1)->forall( i | arr[i]<=arr[pair] )              R
}
 // Sequence(0..pair-1)->forall( i | arr[i]<=arr[pair] )              R

pair = pair + 1;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
```

Regel für die einseitige bedingte Anweisung	$\dfrac{\{ Q \wedge B \}\ S\ \{ R \} \qquad Q \wedge \neg B \rightarrow R}{\{ Q \}\ \text{if } (B)\ S\ \{ R \}}$

- Begründung: Anwendung der Regel für einseitig bedingte Anweisung

```
 Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )        Q∧¬ B → R
        and arr[pair-1]<=arr[pair]
 implies ( Sequence(0..pair-1)->forall( i | arr[i]<=arr[pair] ) )
```


BubbleSort: Beweis des Teilproblems

```
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
if (arr[pair-1] > arr[pair]) {
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
 //ff                       and arr[pair-1]>arr[pair]
  temp = arr[pair-1];
 // Sequence(0..pair-2)->forall( i | arr[i]<=temp )
 //ff                       and temp>arr[pair]
  arr[pair-1] = arr[pair];
 // Sequence(0..pair-2)->forall( i | arr[i]<=temp )
 //ff                       and temp>arr[pair-1]
  arr[pair] = temp;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair] )
 //ff                       and arr[pair]>arr[pair-1]
 // Sequence(0..pair-1)->forall( i | arr[i]<=arr[pair] )
}
 // Sequence(0..pair-1)->forall( i | arr[i]<=arr[pair] )
 // Sequence(0..pair+1-2)->forall( i | arr[i]<=arr[pair+1-1] )
pair = pair + 1;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
```

- Begründung: Umformung (Vorbereitung für Zuweisungsaxiom)

BubbleSort: Beweis des Teilproblems

```
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
if (arr[pair-1] > arr[pair]) {
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
 //ff                      and arr[pair-1]>arr[pair]
  temp = arr[pair-1];
 // Sequence(0..pair-2)->forall( i | arr[i]<=temp )
 //ff                      and temp>arr[pair]
  arr[pair-1] = arr[pair];
 // Sequence(0..pair-2)->forall( i | arr[i]<=temp )
 //ff                      and temp>arr[pair-1]
  arr[pair] = temp;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair] )
 //ff                      and arr[pair]>arr[pair-1]
 // Sequence(0..pair-1)->forall( i | arr[i]<=arr[pair] )
}
 // Sequence(0..pair-1)->forall( i | arr[i]<=arr[pair] )
 // Sequence(0..pair+1-2)->forall( i | arr[i]<=arr[pair+1-1] )
pair = pair + 1;
 // Sequence(0..pair-2)->forall( i | arr[i]<=arr[pair-1] )
```

- Begründung: Umkehrung der Substitution [pair+1/pair] gemäß Zuweisungsaxiom

q.e.d

Zusammenfassung Zusicherungskalkül

- Der Zusicherungskalkül ermöglicht formale Beweise (Verifikation) von Programmeigenschaften,
  - z.B. Einhaltung von Verträgen durch Methoden.
- Programmzustände werden durch Prädikate charakterisiert, Programmeigenschaften durch Formeln der Art {Q} S {R}.
- Die Axiome und Inferenzregeln ergeben sich aus dem induktiven Aufbau des Programmes.
- Das Programm wird um Zusicherungen ergänzt (annotiert), deren Gültigkeit
  - sich aus den Axiomen und Regeln ergibt (in der Phase des Annotierens) bzw.
  - mit den Axiomen und Regeln abgeleitet werden kann.
- Um geeignete Zusicherungen angeben zu können, benötigt man die Kenntnis über die Idee beim Algorithmen-Entwurf.
- Implementierung und Verifikation müssen also Hand in Hand erfolgen.

Nachteile des Zusicherungskalküls

- Die Zusicherungen werden sehr schnell sehr groß;
  - z.B. werden die Invarianten von verschachtelten Schleifen in der innersten Schleife alle mit **and** verknüpft.
- Der Ansatz ist global:
  - der Beweis eines Teilprogrammes kann nicht isoliert von den Zusicherungen der umgebenden Programmkonstrukte erfolgen.
- Kleine Änderungen am Programm machen den Beweis ungültig;
  - sie machen oft große Änderungen an den Zusicherungen erforderlich.
- Große Programme können in der Praxis nicht vollständig verifiziert werden.

Überwachung von Verträgen: Zusicherungskalkül vs. Ausnahmen

	Zusicherungskalkül	**Ausnahmen**
gut geeignet für	die Garantie von **Nachbedingungen** bei • sicherheitskritischen • nicht umfangreichen • oft verwendeten Methoden	die Überprüfung von **Vorbedingung** bei unsicheren • Benutzern • Umgebungen
zusätzlicher Aufwand während der Laufzeit	keiner	vorhanden
zusätzlicher Aufwand während der Entwicklung	sehr groß	vorhanden
Aufwand bei Änderung der Implementierung	sehr groß	sehr gering
Abhängigkeit von der Implementierung	sehr groß	sehr gering
Einsatz in der Praxis	nur für Teilbereiche	ja