# Informatik I - Programmierung
# Globalübung
# 08.11.2005

# Hoare-Kalkül

**Carsten Kern**

**Fachgruppe Informatik**
**RWTH Aachen**

© T. Weiler und C. Kern, RWTH Aachen

# Ariane Flug 501

So hätte es aussehen sollen

… aber das geschah tatsächlich

# Ariane 5

- Die Ariane 5 ist das jüngste Modell der Trägerrakete Ariane der europäischen Weltraumorganisation ESA.
- Mit 4 Satelliten beladen, sollte die Ariane 501 am 4. Juni 1996 ihren Jungfernflug starten. Dieser endete ca. 40 Sekunden nach dem Start mit einer Explosion der Trägerrakete
- 250 Mio. DM Startkosten
- 850 Mio. DM Cluster-Satelliten
- 600 Mio. DM für nachfolgende Verbesserungen
- Verdienstausfall für 2 bis 3 Jahre

(Quelle: http://wwwzenger.informatik.tu-muenchen.de/lehre/seminare/semsoft/unterlagen_02/ariane/website/Ariane.Htm und http://www-aix.gsi.de/~giese/swr/ariane5.html )

# Was war passiert?

- 37 Sekunden nach Zünden der Rakete erreichte Ariane 5 in 3700 m Flughöhe eine Horizontal-Geschwindigkeit von 32768.0 (interne Einheiten).
- Dieser Wert lag etwa fünfmal höher als bei Ariane 4. Die Umwandlung einer 64 bit Gleitkommazahl in eine 16 bit Festkommazahl führte zu einem Überlauf, der jedoch nicht abgefangen wurde.
- Nur bei 3 von 7 Variablen wurde ein Überlauf geprüft - für die anderen 4 Variablen existierten Beweise, dass die Werte klein genug bleiben würden (Ariane 4).
- Diese Beweise galten jedoch nicht für die Ariane 5 und wurden dafür auch gar nicht nachvollzogen.

(Quelle: http://wwwzenger.informatik.tu-muenchen.de/lehre/seminare/semsoft/unterlagen_02/ariane/website/Ariane.Htm und http://www-aix.gsi.de/~giese/swr/ariane5.html )

# Software und Fehler

- **Wunsch: Fehlerfreie Software**
- **Realität: Software enthält Fehler**
- **Garantiert fehlerfreie Software bislang nicht realisierbar**

## Qualitätssicherung

**Analytische Maßnahmen**

Ergebnisse kontrollieren
Erkannte Fehler korrigieren

**Beispiele**:
Review
Test
Verifikation

**Konstruktive / Organisatorische Maßnahmen**

Fehler verhindern
Fehler vermeiden

**Beispiele**:
Code-Richtlinien
Verwendung von Standards

# Verifikation - 1

- **Spezifikation: Angabe, *was* ein Programm tun soll**
  - natürliche Sprache
  - grafische Sprachen (UML, ...)
  - logische Sprachen (Z, VDM, ...)

- **Testen: Überprüfung für endlich viele Eingaben**

  ⇒ keine 100% Sicherheit

- **Verifikation: Mathematischer Beweis der Korrektheit**
  - Terminierung: Hält Programm immer an?
  - Partielle Korrektheit: Falls Programm anhält, erfüllt es Spezifikation?
  - Totale Korrektheit: Terminierung & Partielle Korrektheit

  ⇒ Semantik der Programmiersprache

# Verifikation - 2

## Programm P

```
i = n;
res = 1;
while (i > 1) {
   res = res * i;
   i = i - 1;
}
```

- **Spezifikation:**
  Programm berechnet (in **`res`**) Fakultät von **`n`**
- **Terminierung:**
  Programm hält an, weil **`i`** in jedem Schleifendurchlauf kleiner wird
- **Partielle Korrektheit:**
  Nach Ausführung ist **`res`** = **`n`**!
- **Totale Korrektheit**

Wie beweist man so etwas ?

⇒ Verifikation nötig bei sicherheitskritischen Anwendungen

⇒ hilft für Programmentwurf und Programmierstil

# Partielle Korrektheit: Hoare-Kalkül

- **Spezifikation (zur partiellen Korrektheit)**

  $\langle \varphi \rangle$ **P** $\langle \psi \rangle$

  Wenn vor Ausführung von P Vorbedingung $\varphi$ gilt

  und Ausführung von P terminiert,

  dann gilt hinterher Nachbedingung $\psi$.

- **Bsp:** `<n >= 0>` P `<res = n!>`

- Partielle Korrektheit ist *semantische* Aussage

  Hoare-Kalkül: 7 *syntaktische* Regeln zur Herleitung von Korrektheitsaussagen

# Zuweisungsregel

$$\frac{}{\langle \varphi[x/t] \rangle \; x = t; \; \langle \varphi \rangle}$$

**x** ist Variable, **t** ist Ausdruck (ohne Seiteneffekte),
$\varphi$ **[x/t]** ist $\varphi$ mit allen **x** ersetzt durch **t**

**Bsp:** **<5 = 5>  x = 5;  <x = 5>**

**<5 = 5>**

**x = 5;**

**<x = 5>**

# Sequenzregel

$$\frac{\langle \varphi \rangle\ P\ \langle \psi \rangle \qquad \langle \psi \rangle\ Q\ \langle \beta \rangle}{\langle \varphi \rangle\ P\ Q\ \langle \beta \rangle}$$

**Bsp:** <true>

```
        x = 5;
        res = x * x + 6;
```

<res = 31>

<true>
<5 = 5>

x = 5;

<x = 5>
<x * x + 6 = 31>

res = x * x + 6;

<res = 31>

# Konsequenzregel 1 (Stärkere Vorbedingung)

$$\frac{\langle \varphi \rangle\ P\ \langle \psi \rangle \qquad \alpha \Rightarrow \varphi}{\langle \alpha \rangle\ P\ \langle \psi \rangle}$$

**Bsp:** `<true>  x = 5;  <x = 5>`, denn:

$$\frac{\texttt{<5 = 5>}\ \texttt{x = 5;}\ \texttt{<x = 5>} \qquad \texttt{true} \Rightarrow \texttt{5 = 5}}{\texttt{<true>}\ \ \texttt{x = 5;}\ \ \texttt{<x = 5>}}$$

```
<true>
<5 = 5>
x = 5;
<x = 5>
```

# Konsequenzregel 2 (Schwächere Nachbedg.)

$$\frac{\langle\varphi\rangle\ P\ \langle\psi\rangle \qquad \psi \Rightarrow \beta}{\langle\varphi\rangle\ P\ \langle\beta\rangle}$$

**Bsp:** $\langle true\rangle\ \ x = 5;\ \ \langle x \geq 5\rangle$, denn:

$$\frac{\langle true\rangle\ x = 5;\ \langle x = 5\rangle \qquad x = 5 \Rightarrow x \geq 5}{\langle true\rangle\ \ x = 5;\ \ \langle x \geq 5\rangle}$$

$\langle true\rangle$
$\langle 5 = 5\rangle$
`x = 5;`
$\langle x = 5\rangle$
$\langle x \geq 5\rangle$

# Bedingungsregel 1

$$\frac{\langle \varphi \wedge B \rangle \; P \; \langle \psi \rangle \qquad \varphi \wedge \neg B \Rightarrow \psi}{\langle \varphi \rangle \; \texttt{if (B) \{P\}} \; \langle \psi \rangle}$$

**Bsp:** $\langle true \rangle$

```
        res = y;
        if (x > y) res = x;
```

$\langle res = max(x,y) \rangle$

denn: $\langle res = y \wedge x > y \rangle$

$\langle x = max(x,y) \rangle$

```
        res = x;
```

$\langle res = max(x,y) \rangle$

und $\langle res = y \wedge \neg x > y \rangle$

$\Rightarrow \langle res = max(x,y) \rangle$

$\langle true \rangle$

$\langle y = y \rangle$

```
res = y;
```

$\langle res = y \rangle$

```
if (x > y) res = x;
```

$\langle res = max(x,y) \rangle$

# Bedingungsregel 2

$$\frac{\langle \varphi \wedge B \rangle\ P\ \langle \psi \rangle \qquad \langle \varphi \wedge \neg B \rangle\ Q\ \langle \psi \rangle}{\langle \varphi \rangle\ \texttt{if (B) \{P\} else \{Q\}}\ \langle \psi \rangle}$$

**Bsp:** <true>

```
if (x < 0)
    res = -x;
else
    res = x;
```

<res = |x|>

denn: <x < 0>
<-x = |x|>

```
res = -x;
```

<res = |x|>

und < ¬ x < 0>
<x = |x|>

```
res = x;
```

<res = |x|>

# Schleifenregel (1)

$$\frac{\langle \varphi \wedge B \rangle \; P \; \langle \varphi \rangle}{\langle \varphi \rangle \; \texttt{while (B) \{P\}} \; \langle \varphi \wedge \neg B \rangle}$$

**Bsp.: (Berechnung der Fakultät von n)**

```
<n >= 0>
i = n;
res = 1;
while (i > 1) {
  res = res * i;
  i = i - 1;
}
<res = n!>
```

# Schleifenregel (2)

$$\frac{\langle \varphi \wedge B \rangle \ P \ \langle \varphi \rangle}{\langle \varphi \rangle \ \texttt{while (B) \{P\}} \ \langle \varphi \wedge \neg B \rangle}$$

**Zu zeigen:**

```
<n >= 0>
i = n;
res = 1;
< φ >
while (i > 1) {
  < φ ∧ (i > 1)>
  res = res * i;
  i = i - 1;
  < φ >
}
< φ ∧ ¬(i > 1)>
<res = n!>
```

**Problem**:
Wie finden wir die Schleifeninvariante $\varphi$ **?**

**Vorgehen:**
Betrachte Wert der verwendeten Variablen (hier für n = 4)

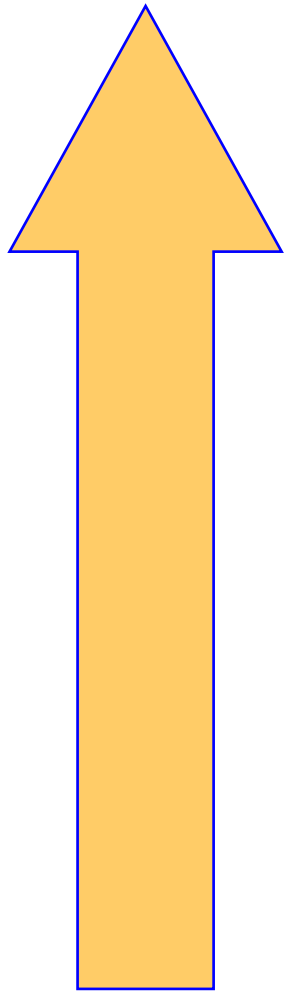
| res | i | n |
|---|---|---|
| 1 | 4 | 4 |
| 4 | 3 | 4 |
| 12 | 2 | 4 |
| 24 | 1 | 4 |

$\varphi \equiv (res * i! = n!) \wedge (i >= 0)$

# Schleifenregel (3)

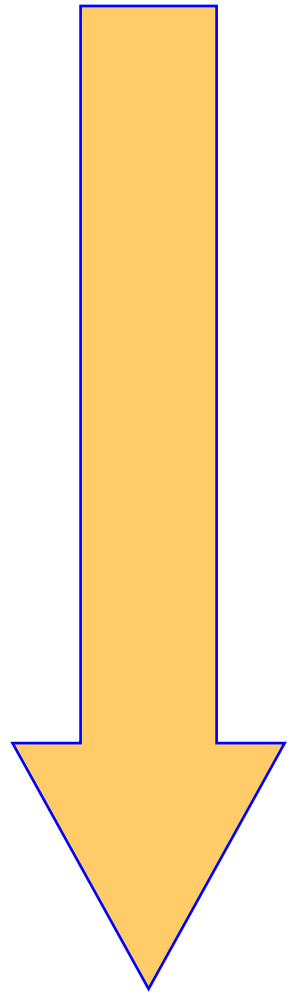
```
<n >= 0>
<(1 * n! = n!) ∧ (n >= 0)>
i = n;
<(1 * i! = n!) ∧ (i >= 0)>
res = 1;
<(res * i! = n!) ∧ (i >= 0)>
while (i > 1) {
  <(res * i! = n!) ∧ (i >= 0) ∧ (i > 1)>
  <(res * i! = n!) ∧ (i >= 1)>
  <((res * i) * (i-1)! = n!) ∧ ((i-1) >= 0)>
  res = res * i;
  <(res * (i-1)! = n!) ∧ ((i-1) >= 0)>
  i = i - 1;
  <(res * i! = n!) ∧ (i >= 0)>
}
<(res * i! = n!) ∧ (i >= 0) ∧ ¬(i > 1)>
<(res * i! = n!) ∧ (i >= 0) ∧ (i <= 1)>
<res = n!>
```

Zu zeigen: rot
Beweisschritte: blau



# Schleifenregel (4)

```
<n >= 0>
<n >= 0 ∧ n = n>
i = n;
<i >= 0 ∧ i = n>
<i >= 0 ∧ i = n ∧ 1 = 1>
res = 1;
<i >= 0 ∧ i = n ∧ res = 1>
<(res * i! = n!) ∧ (i >= 0)>
while (i > 1) {
  <(res * i! = n!) ∧ (i >= 0) ∧ (i > 1)>
  <(res * i! = n!) ∧ (i > 0)>
  <((res * i) * (i-1)! = n!) ∧ (i > 0)>
  res = res * i;
  <(res * (i-1)! = n!) ∧ (i > 0)>
  <(res * (i-1)! = n!) ∧ (i - 1 >= 0)>
  i = i - 1;
  <(res * i! = n!) ∧ (i >= 0)>
}
<(res * i! = n!) ∧ (i >= 0) ∧ ¬(i > 1)>
<(res * i! = n!) ∧ (i >= 0) ∧ (i <= 1)>
<res = n!>
```



# Potenzierung (Algorithmus)

**Eingabe:** b, e mit e >= 0

**Ausgabe:** erg = b ^ e

```
int bas = b;
int exp = e;
int erg = 1;
while (exp > 0) {
   if (exp % 2 == 1) {
      erg = erg * bas;
   }
   bas = bas * bas;
   exp = exp / 2;
}
```

Ganzzahlige Division in JAVA, falls Dividend und Divisor (wie hier) int-Werte sind.

# Potenzierung (Partielle Korrektheit - 1)

```
<e >= 0>
int bas = b;
int exp = e;
int erg = 1;
<φ>
while (exp > 0) {
   <φ ∧ (exp > 0)>
   if (exp % 2 == 1) {
      erg = erg * bas;
   }
   bas = bas * bas;
   exp = exp / 2;
   <φ>
}
<φ ∧ ¬ (exp > 0)>
<erg = b ^ e>
```

| b | e | bas | exp | erg |
|---|---|---|---|---|
| 2 | 9 | 2 | 9 | 1 |
| 2 | 9 | 4 | 4 | 2 |
| 2 | 9 | 16 | 2 | 2 |
| 2 | 9 | 256 | 1 | 2 |
| 2 | 9 | 65536 | 0 | 512 |

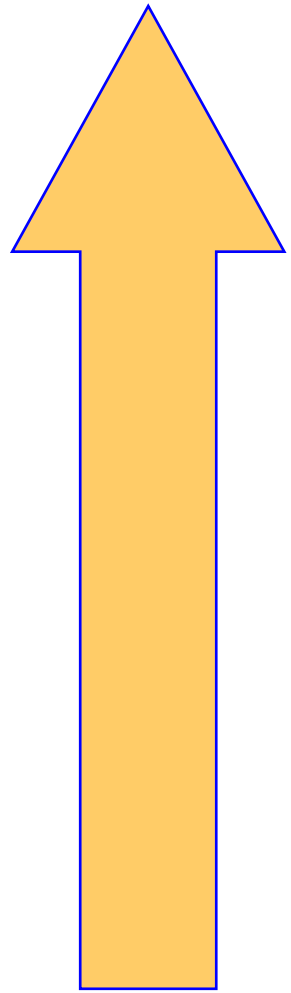
Schleifeninvariante:

$\varphi \equiv b \hat{} e = bas \hat{} exp * erg \wedge exp >= 0$

# Potenzierung (Partielle Korrektheit – 2)

**Zu zeigen:**

```
<e >= 0>
int bas = b;
int exp = e;
int erg = 1;
<b ^ e = bas ^ exp * erg ∧ exp >= 0>
while (exp > 0) {
   <b ^ e = bas ^ exp * erg ∧ exp >= 0 ∧ (exp > 0)>
   if (exp % 2 == 1) {
      erg = erg * bas;
   }
   bas = bas * bas;
   exp = exp / 2;
   <b ^ e = bas ^ exp * erg ∧ exp >= 0>
}
<b ^ e = bas ^ exp * erg ∧ exp >= 0 ∧ ¬(exp > 0)>
<erg = b ^ e>
```



# Potenzierung (Partielle Korrektheit – 3)

```
<b ^ e = bas ^ exp  * erg ∧ exp >= 0>
while (exp > 0) {
   <b ^ e = bas ^ exp  * erg ∧ exp >= 0 ∧ (exp > 0)>
   if (exp % 2 == 1) {
      erg = erg * bas;
   }
   <b ^ e = (bas * bas) ^ (exp DIV 2)  * erg ∧ exp >= 0>
   bas = bas * bas;
   <b ^ e = bas ^ (exp DIV 2)  * erg ∧ exp >= 0>

   exp = exp / 2;
   <b ^ e = bas ^ exp  * erg ∧ exp >= 0>
}
<b ^ e = bas ^ exp  * erg ∧ exp >= 0 ∧ ¬(exp > 0)>
```

# Potenzierung (IF - Anweisung)

$$\frac{\langle \varphi \wedge B \rangle\ P\ \langle \psi \rangle \qquad \varphi \wedge \neg B \Rightarrow \psi}{\langle \varphi \rangle\ \texttt{if (B) \{P\}}\ \langle \psi \rangle}$$

```
<b ^ e = bas ^ exp * erg ∧ (exp >= 0)>
if (exp % 2 == 1) {
   erg = erg * bas;
}
<b ^ e = (bas * bas) ^ (exp DIV 2) * erg ∧ exp >= 0>
```

**Zu zeigen also:**

```
<b ^ e = bas ^ exp * erg ∧ (exp >= 0) ∧ (exp % 2 = 1)>
erg = erg * bas;
<b ^ e = (bas * bas) ^ (exp DIV 2) * erg ∧ (exp >= 0)>
```

**und**

```
<b ^ e = bas ^ exp * erg ∧ (exp >= 0) ∧ ¬(exp % 2 = 1)>
⇒ <b ^ e = (bas * bas) ^ (exp DIV 2) * erg ∧ (exp >= 0)>
```

# IF – Anweisung (Bedingung erfüllt)

```
<b ^ e = bas ^ exp * erg ∧ (exp >= 0) ∧ (exp % 2 = 1)>
<b ^ e = bas ^ (2 * exp DIV 2) * (erg * bas) ∧ (exp >= 0)>
<b ^ e = (bas ^ 2) ^ (exp DIV 2) * (erg * bas) ∧ (exp >= 0)>
erg = erg * bas;
<b ^ e = (bas * bas) ^ (exp DIV 2) * erg ∧ (exp >= 0)>
```

Da exp % 2 = 1, also exp ungerade, gilt:

bas ^ exp = bas ^ (2 * exp DIV 2) * bas

Es gilt:

bas ^ (2 * exp DIV 2) = (bas ^ 2) ^ (exp DIV 2)

# IF – Anweisung (Bedingung nicht erfüllt)

<b ^ e = bas ^ exp * erg ∧ (exp >= 0) ∧ ¬(exp % 2 = 1)>

<b ^ e = bas ^ (2 * exp DIV 2) * erg ∧ (exp >= 0)>

<b ^ e = (bas ^ 2) ^ (exp DIV 2) * erg ∧ (exp >= 0)>

<b ^ e = (bas * bas) ^ (exp DIV 2) * erg ∧ (exp >= 0) >

Da exp % 2 = 0, also exp gerade, gilt:

bas ^ exp = bas ^ (2 * exp DIV 2)

Es gilt:

bas ^ (2 * exp DIV 2) = (bas ^ 2) ^ (exp DIV 2)

# Potenzierung (Partielle Korrektheit – 4)

```
<e >= 0>
int bas = b;
int exp = e;
int erg = 1;
<b ^ e = bas ^ exp * erg ∧ exp >= 0>
while (exp > 0) {
   <b ^ e = bas ^ exp * erg ∧ exp >= 0 ∧ (exp > 0)>
   if (exp % 2 == 1) {
      erg = erg * bas;
   }
   bas = bas * bas;
   exp = exp / 2;
   <b ^ e = bas ^ exp * erg ∧ exp >= 0>
}
<b ^ e = bas ^ exp * erg ∧ exp >= 0 ∧ ¬(exp > 0)>
<erg = b ^ e>
```

# Potenzierung (Partielle Korrektheit – 5)

**Noch zu zeigen:**

**<e >= 0>**

**<b ^ e = b ^ e * 1 ∧ e >= 0>**

```
int bas = b;
```

**<b ^ e = bas ^ e * 1 ∧ e >= 0>**

```
int exp = e;
```

**<b ^ e = bas ^ exp * 1 ∧ exp >= 0>**

```
int erg = 1;
```

**<b ^ e = bas ^ exp * erg ∧ exp >= 0>**

**q.e.d**

# Hinweise

- **Die Schleifeninvariante zu finden, ist ein kreativer Prozess**
  - Tabelle
  - Verständnis des Algorithmus
  - Erfahrung
- **In den einzelnen Schritten möglichst nicht umformen bzw. ausrechnen**
- **Die Beweisrichtung ist von „oben“ nach „unten“ (vorwärts)**
- **Die Beweisfindung wird oft erleichtert, wenn man von „unten“ nach „oben“ vorgeht (rückwärts)**

# Prof. Tony Hoare

- **Full name: Sir Charles Antony Richard Hoare,**
  **born: 11. Jan 1934**
- **Elliot Brothers Ltd., London (1960 – 1966)**
- **Professor of Computing Science at Queen's University, Belfast (1968 – 1977)**
- **Professor of Computation and Head of the Programming Research Group, Oxford (1977-1999)**
- **Senior Researcher with Microsoft Research in Cambridge (since 1999)**

(Quelle: http://research.microsoft.com/~thoare/)