

Verlässliche Echtzeitsysteme

Verifikation funktionaler Eigenschaften – Design by Contract

Peter Ulbrich

Lehrstuhl für Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

<https://www4.cs.fau.de>

06. Juli 2017



Übersicht der heutigen Vorlesung

- Verifikation **funktionaler** Eigenschaften: **Design-by-Contract**
 - Grundlage: Zusagen in Form von **Vor- und Nachbedingungen**
 - Wie beschreibt man diese **Verträge**?
 - Wie leitet man daraus Korrektheitsaussagen ab? \rightsquigarrow **Hoare- / WP-Kalkül**

- Beschreibung von Verträgen mit Hilfe von **Annotationen**
 - Beispielsweise durch eine Erweiterung der Programmiersprache
 - Überprüft mithilfe eines Verifikationswerkzeugs

- 👉 Entwickeln eines **groben Verständnisses**
 - Wieso ist formale Verifikation immer noch ein Problem?
 - Umsetzung geht weit über den Umfang der Vorlesung/Übung



- 1 Grundlagen
- 2 Formale Spezifikation
 - Hoare-Kalkül
 - WP-Kalkül
- 3 Praktische Überlegungen
 - Zusicherungen
 - Funktionale Verifikation in Astree
- 4 Zusammenfassung



- Diese Programm enthält diverse Fehler ...
 - Division durch 0, undefinierte Speicherzugriffe, Ganzzahlüberlauf

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

 Abstrakte Interpretation deckt diese Defekte auf

- Intervallanalyse erfasst z.B.
 - Den Wert 0 für size ...
 - Oder den möglichen Überlauf von temp



```
1  unsigned int average(unsigned int[16] array) {
2      unsigned long long temp = 0;
3
4      for(unsigned int i = 0; i < 16; i++) {
5          temp += array[i];
6      }
7
8      return temp/20;
9  }
```



Wir können diese Fehler beheben!

- Zumindest für Spezialfälle ist dies offensichtlich



```
1 unsigned int average(unsigned int[16] array) {
2     unsigned long long temp = 0;
3
4     for(unsigned int i = 0; i < 16; i++) {
5         temp += array[i];
6     }
7
8     return temp/20;
9 }
```



Wir können diese Fehler beheben!

- Zumindes für Spezialfälle ist dies offensichtlich



Aber: Ist diese Implementierung korrekt?

- Mit Sicherheit nicht \rightsquigarrow sie liefert einen vollkommen falschen Wert
- Wir müssen beschreiben, was wir von `average()` erwarten!





Berechnet Durchschnittswert aller Elemente des Felds a r r a y

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





Berechnet Durchschnittswert aller Elemente des Felds `array`

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

■ **Nachbedingung** der Funktion `average()`

- Sie wird durch die Implementierung der Funktion garantiert
- Aufrufer von `average()` kann diese Nachbedingung **ausnutzen**



Beschreibung der Programmsemantik

Gemäß der funktionalen Spezifikation der Funktion



Berechnet Durchschnittswert aller Elemente des Felds array



Annahmen des Entwicklers und **Forderungen** an den **Aufrufer** (engl. *caller*)

- Feld array hat **genau** size **korrekt initialisierte** Elemente
- Summe aller Elemente passt in temp \mapsto `sum(array, size) <= ULONG_MAX`

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

■ **Nachbedingung** der Funktion `average()`

- Sie wird durch die Implementierung der Funktion garantiert
- Aufrufer von `average()` kann diese Nachbedingung **ausnutzen**



Beschreibung der Programmsemantik

Gemäß der funktionalen Spezifikation der Funktion



Berechnet Durchschnittswert aller Elemente des Felds array



Annahmen des Entwicklers und **Forderungen** an den **Aufrufer** (engl. *caller*)

- Feld array hat **genau** size **korrekt initialisierte** Elemente
- Summe aller Elemente passt in temp \mapsto `sum(array, size) <= ULONG_MAX`

■ **Vorbedingungen** der Funktion `average()`

- Aufrufer von `average` muss diese **sicherstellen**

→ Die Implementierung der Funktion kann sie ausnutzen

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

■ **Nachbedingung** der Funktion `average()`

- Sie wird durch die Implementierung der Funktion garantiert

→ Aufrufer von `average()` kann diese Nachbedingung **ausnutzen**



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



temp ist nie größer als die Summe der Elemente in array

- $\sum \text{array}[0..i] \geq \text{temp}, \forall i < \text{size}$



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

 temp ist nie größer als die Summe der Elemente in array

- $\sum \text{array}[0..i] \geq \text{temp}, \forall i < \text{size}$



Invarianten der Funktion average ()

- Bedingungen die ihre Gültigkeit während der (gesamten) Ausführung behalten
- Von großem Nutzen für die Beweisführung!



Invarianten – Ein illustratives Beispiel



Das MU-Puzzle¹ [8]:

- **Annahme:** Es gibt drei Symbole: M, I, U, **Startpunkt:** MI
- In jedem Schritt ist eine der folgenden **Regeln** anzuwenden:

- 1 $xI \rightarrow xIU$: Füge U an, falls String auf I endet (MI \rightarrow MIU)
- 2 $Mx \rightarrow Mxx$: Verdopple die Zeichenkette nach M (MIU \rightarrow MIUIU)
- 3 $xIIIy \rightarrow xUy$: Ersetze III durch U (MUIIIU \rightarrow MUUU)
- 4 $xUUy \rightarrow xy$: Lösche beliebige UU (MUUU \rightarrow MU)

¹Video der Vorlesung: <http://ocw.mit.edu/high-school/humanities-and-social-sciences/godel-escher-bach/video-lectures/lecture-1-video/>



Invarianten – Ein illustratives Beispiel



Das MU-Puzzle¹ [8]:

- **Annahme:** Es gibt drei Symbole: M, I, U, **Startpunkt:** MI
- In jedem Schritt ist eine der folgenden **Regeln** anzuwenden:

- 1 $xI \rightarrow xIU$: Füge U an, falls String auf I endet (MI \rightarrow MIU)
- 2 $Mx \rightarrow Mxx$: Verdopple die Zeichenkette nach M (MIU \rightarrow MIUIU)
- 3 $xIIIy \rightarrow xUy$: Ersetze III durch U (MUIIIU \rightarrow MUUU)
- 4 $xUUy \rightarrow xy$: Lösche beliebige UU (MUUU \rightarrow MU)

- **Frage:** Ist die Zeichenkette MI überführbar in MU?

¹Video der Vorlesung: <http://ocw.mit.edu/high-school/humanities-and-social-sciences/godel-escher-bach/video-lectures/lecture-1-video/>



Invarianten – Ein illustratives Beispiel



Das MU-Puzzle¹ [8]:

- **Annahme:** Es gibt drei Symbole: M, I, U, **Startpunkt:** MI
- In jedem Schritt ist eine der folgenden **Regeln** anzuwenden:

- 1 $xI \rightarrow xIU$: Füge U an, falls String auf I endet (MI \rightarrow MIU)
- 2 $Mx \rightarrow Mxx$: Verdopple die Zeichenkette nach M (MIU \rightarrow MIUIU)
- 3 $xIIIy \rightarrow xUy$: Ersetze III durch U (MUIIIU \rightarrow MUUU)
- 4 $xUUy \rightarrow xy$: Lösche beliebige UU (MUUU \rightarrow MU)

- **Frage:** Ist die Zeichenkette MI überführbar in MU?

 Der formale (logische) Beweis ist ein kniffliges Problem

- Vor- und Nachbedingungen sind nicht ausreichend
 - Abstrakte Interpretation versagt hier ebenfalls (vgl. ab-Problem VIII/32)
- Manuelle Ableitung oft der einzige (mühsame) Ausweg!

¹Video der Vorlesung: <http://ocw.mit.edu/high-school/humanities-and-social-sciences/godel-escher-bach/video-lectures/lecture-1-video/>



Invarianten – Ein illustratives Beispiel



Das MU-Puzzle¹ [8]:

- **Annahme:** Es gibt drei Symbole: M, I, U, **Startpunkt:** MI
- In jedem Schritt ist eine der folgenden **Regeln** anzuwenden:

- 1 $xI \rightarrow xIU$: Füge U an, falls String auf I endet ($MI \rightarrow MIU$)
- 2 $Mx \rightarrow Mxx$: Verdopple die Zeichenkette nach M ($MIU \rightarrow MIUIU$)
- 3 $xIIIy \rightarrow xUy$: Ersetze III durch U ($MUIIIU \rightarrow MUUU$)
- 4 $xUUy \rightarrow xy$: Lösche beliebige UU ($MUUU \rightarrow MU$)

- **Frage:** Ist die Zeichenkette MI überführbar in MU?



Der formale (logische) Beweis ist ein kniffliges Problem

- Vor- und Nachbedingungen sind nicht ausreichend
 - Abstrakte Interpretation versagt hier ebenfalls (vgl. ab-Problem VIII/32)
- Manuelle Ableitung oft der einzige (mühsame) Ausweg!



Invariante: Die Zahl der Is ist kein Vielfaches von 3 \leadsto **NEIN!**

¹Video der Vorlesung: <http://ocw.mit.edu/high-school/humanities-and-social-sciences/godel-escher-bach/video-lectures/lecture-1-video/>





Zusicherungen (engl. *assertions*)

- Regeln das Verhältnis zwischen **Aufrufer** und **Gerufenem** (engl. *callee*)



Formales System zur Beschreibung einer Funktion



Zusicherungen (engl. *assertions*)

- Regeln das Verhältnis zwischen **Aufrufer** und **Gerufenem** (engl. *callee*)



P

Vorbedingungen (engl. *preconditions*)

- Werden vom **Aufrufer** erfüllt, in der **Funktion** genutzt



Formales System zur Beschreibung einer Funktion

-  **Zusicherungen** (engl. *assertions*)
 - Regeln das Verhältnis zwischen **Aufrufer** und **Gerufenem** (engl. *callee*)
- P** **Vorbedingungen** (engl. *preconditions*)
 - Werden vom **Aufrufer erfüllt**, in der **Funktion genutzt**
- Q** **Nachbedingungen** (engl. *postconditions*)
 - Werden vom **Gerufenem erfüllt**, vom **Aufrufer genutzt**
-  Unter der Bedingung, dass die Vorbedingungen gelten



Formales System zur Beschreibung einer Funktion



Zusicherungen (engl. *assertions*)

- Regeln das Verhältnis zwischen **Aufrufer** und **Gerufenem** (engl. *callee*)

P **Vorbedingungen** (engl. *preconditions*)

- Werden vom **Aufrufer erfüllt**, in der **Funktion genutzt**

Q **Nachbedingungen** (engl. *postconditions*)

- Werden vom **Gerufenem erfüllt**, vom **Aufrufer genutzt**

⚠ Unter der Bedingung, dass die Vorbedingungen gelten

I **Invarianten** (engl. *invariants*)

- Gelten sowohl **vor** als auch **nach** dem Funktionsaufruf

⚠ Eine zwischenzeitliche Verletzung innerhalb der Prozedur wird toleriert



Formales System zur Beschreibung einer Funktion



Zusicherungen (engl. *assertions*)

- Regeln das Verhältnis zwischen **Aufrufer** und **Gerufenem** (engl. *callee*)

P **Vorbedingungen** (engl. *preconditions*)

- Werden vom **Aufrufer erfüllt**, in der **Funktion genutzt**

Q **Nachbedingungen** (engl. *postconditions*)

- Werden vom **Gerufenem erfüllt**, vom **Aufrufer genutzt**

⚠ Unter der Bedingung, dass die Vorbedingungen gelten

I **Invarianten** (engl. *invariants*)

- Gelten sowohl **vor** als auch **nach** dem Funktionsaufruf

⚠ Eine zwischenzeitliche Verletzung innerhalb der Prozedur wird toleriert

S **Anweisungen** (engl. *statements*) \mapsto Programmsegment

- Beschreiben die **Implementierung** der Funktion

⚠ Formal beschrieben oder durch statische Analyse ermittelt



Formales System zur Beschreibung einer Funktion



Zusicherungen (engl. *assertions*)

- Regeln das Verhältnis zwischen **Aufrufer** und **Gerufenem** (engl. *callee*)

P **Vorbedingungen** (engl. *preconditions*)

- Werden vom **Aufrufer erfüllt**, in der **Funktion genutzt**

Q **Nachbedingungen** (engl. *postconditions*)

- Werden vom **Gerufenem erfüllt**, vom **Aufrufer genutzt**

⚠ Unter der Bedingung, dass die Vorbedingungen gelten

I **Invarianten** (engl. *invariants*)

- Gelten sowohl **vor** als auch **nach** dem Funktionsaufruf

⚠ Eine zwischenzeitliche Verletzung innerhalb der Prozedur wird toleriert

S **Anweisungen** (engl. *statements*) \mapsto Programmsegment

- Beschreiben die **Implementierung** der Funktion

⚠ Formal beschrieben oder durch statische Analyse ermittelt



Ableitbarkeit zeigt die formale Korrektheit der Funktion

- $P \wedge I \wedge S \Rightarrow Q \wedge I$



Überprüfung der Zusicherungen?

- Das Programmsegment S implementiert eine Transformation zwischen der Vorbedingung P und der Nachbedingung Q
 - Entsprechende Transformationen existieren für alle Programmkonstrukte
 - Zuweisungen, Sequenzen, Verzweigungen, Schleifen, Funktionsaufrufe, ...
- **Aufgabe:** Zusammenbringen von P , S und Q



Überprüfung der Zusicherungen?

- Das Programmsegment S implementiert eine Transformation zwischen der Vorbedingung P und der Nachbedingung Q
 - Entsprechende Transformationen existieren für alle Programmkonstrukte
 - Zuweisungen, Sequenzen, Verzweigungen, Schleifen, Funktionsaufrufe, ...
- **Aufgabe:** Zusammenbringen von P , S und Q

☞ Prädikattransformation (engl. *predicate transformer semantics*)

- Stellt Strategien bereit, um Hoare-Triple $\{P\} S \{Q\}$ zu beweisen
- Eine **Vorwärtsanalyse** liefert die **stärkste Nachbedingung** (engl. *strongest postcondition*) $sp(S, P)$
 - $\{P\} S \{Q\}$ gilt, genau dann wenn $sp(S, P) \Rightarrow Q$ wahr ist
- Eine **Rückwärtsanalyse** liefert die **schwächste Vorbedingung** (engl. *weakest precondition*) $wp(S, Q)$
 - $\{P\} S \{Q\}$ gilt, genau dann wenn $P \Rightarrow wp(S, Q)$ wahr ist

☞ Basiert auf dem **Hoare-** (siehe 12 ff) / **WP-Kalkül** (siehe 24 ff)

- Beschreibt die (formale) **Funktionssemantik** eines Programms



1 Grundlagen

2 Formale Spezifikation

- Hoare-Kalkül
- WP-Kalkül

3 Praktische Überlegungen

- Zusicherungen
- Funktionale Verifikation in Astree

4 Zusammenfassung



Sir Charles Anthony Richard (C.A.R.) Hoare

Ein Informatik-Pionier: Leben und Wirken



- 1934 geboren in Colombo, Sri Lanka
- ab 1956 Studium in Oxford und Moskau
- ab 1960 Elliot Brothers
- 1968 Habilitation an der Queen's University of Belfast
- ab 1977 Professor für Informatik (Oxford)

Auszeichnungen (Auszug)

- 1980 Turing Award
- 2000 Kyoto-Preis
- 2007 Friedrich L. Bauer Preis
- 2010 John-von-Neumann-Medaille

bekannte Werke (Auszug)

- Quicksort-Algorithmus [5]
- Hoare-Kalkül [6]
- Communicating Sequential Processes [7]



- Ein **formales System**, um Aussagen zur Korrektheit von Programmen zu treffen, die in imperativen Programmiersprachen verfasst sind.
- Das Hoare-Kalkül umfasst **Axiome** ...
 - Leere Anweisungen
 - Zuweisungen
- ... und **Ableitungsregeln** (bzw. **Inferenzregeln**)
 - Sequenzen (bzw. Komposition) von Anweisungen
 - Auswahlen von Anweisungen
 - Iterationen von Anweisungen und
 - Konsequenz



Ist **nicht vollständig** und bezieht sich nur auf die **partielle Korrektheit**



Definition von Zusicherungen

- Zusicherungen werden als Formeln der **Prädikatenlogik** beschrieben

☞ Üblicherweise definiert als sogenannte **Hoare-Triple**:

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

- P ist die Vorbedingung, Q die Nachbedingung, S ein Programmsegment
- P und Q werden als Formeln der Prädikatenlogik beschrieben
- Bedeutung: Falls P vor der Ausführung von S gilt, gilt Q danach
 - Dies setzt voraus, **dass S terminiert**
- Sonst ist keine Aussage über den folgenden Programmzustand möglich



Partielle Korrektheit: Die Terminierung muss gesondert bewiesen werden

- Man verwendet $\{P\} S \{falsch\}$ um auszudrücken, dass S nicht terminiert
- Nachweis grundsätzlich möglich, jedoch nicht Gegenstand der Vorlesung



```
S:int maximum(int a,int b) {  
    int result = INT_MIN;  
  
    if(a > b)  
        result = a;  
    else  
        result = b;  
  
    return result;  
}
```

- Das Programmsegment S ist die Implementierung der Funktion



Beispiel: Maximum-Funktion

P : wahr

```
S: int maximum(int a, int b) {  
    int result = INT_MIN;  
  
    if(a > b)  
        result = a;  
    else  
        result = b;  
  
    return result;  
}
```

- Das **Programmsegment** S ist die Implementierung der Funktion
- **Vorbedingung** P : **wahr**
 - Die Implementierung stellt keine Anforderungen an die Parameter



Beispiel: Maximum-Funktion

P : wahr

```
S:int maximum(int a,int b) {  
    int result = INT_MIN;  
  
    if(a > b)  
        result = a;  
    else  
        result = b;  
  
    return result;  
}
```

Q : $result \geq a \wedge result \geq b$

- Das **Programmsegment** S ist die Implementierung der Funktion
- **Vorbedingung** P : **wahr**
 - Die Implementierung stellt keine Anforderungen an die Parameter
- **Nachbedingung** Q : $result \geq a \wedge result \geq b$
 - „Offensichtliche“ Eigenschaft des zu berechnenden Ergebnisses



Beispiel: Maximum-Funktion

P : wahr

```
S:int maximum(int a,int b) {  
    int result = INT_MIN;  
  
    if(a > b)  
        result = a;  
    else  
        result = b;  
  
    return result;  
}
```

Q : $result \geq a \wedge result \geq b$

- Das **Programmsegment** S ist die Implementierung der Funktion
- **Vorbedingung** P : **wahr**
 - Die Implementierung stellt keine Anforderungen an die Parameter
- **Nachbedingung** Q : $result \geq a \wedge result \geq b$
 - „Offensichtliche“ Eigenschaft des zu berechnenden Ergebnisses
 - Wie man dieses Ergebnis bestimmt, ist hier nicht von Belang



- Leere Anweisung **skip**

$$\frac{}{\{P\}\mathbf{skip}\{P\}}$$

- Die leere Anweisung verändert den Programmzustand nicht
- Falls P vor **skip** gilt, gilt es auch danach

■ Leere Anweisung **skip**

$$\frac{}{\{P\}\mathbf{skip}\{P\}}$$

- Die leere Anweisung verändert den Programmzustand nicht
- Falls P vor **skip** gilt, gilt es auch danach

■ Zuweisung $\mathbf{x = y}$

$$\frac{}{\{P[y/x]\}\mathbf{x = y}\{P\}}$$

- $P[y/x] \rightsquigarrow$ jedes Auftreten von x in P wird durch y ersetzt
- was nach der Zuweisung für x gilt, galt vor der Zuweisung für y
- Beispiel: $\{y > 100\}\mathbf{x = y};\{x > 100\}$

$P : y > 100$

$S : \mathbf{x = y};$

$Q : x > 100$



- Für lineare Kompositionen $S_1; S_2$ zweier Segmente S_1 und S_2

$$\frac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

- Falls S_1 die Vorbedingung für S_2 erzeugt, können sie verkettet werden
- Im Anschluss an S_2 hat dessen Nachbedingung R Bestand



Sequenzregel

- Für lineare Kompositionen $S_1; S_2$ zweier Segmente S_1 und S_2

$$\frac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

- Falls S_1 die Vorbedingung für S_2 erzeugt, können sie verkettet werden
 - Im Anschluss an S_2 hat dessen Nachbedingung R Bestand
- Beispiel:

$$\frac{\{y + 1 = 43\}x = y + 1; \{x = 43\} \quad \{x = 43\}z = x; \{z = 43\}}{\{y + 1 = 43\}x = y + 1; z = x; \{z = 43\}}$$

$$P : y + 1 = 43$$

$$S_1 : x = y + 1;$$

$$Q : x = 43$$

$$Q : x = 43$$

$$S_2 : z = x;$$

$$R : z = 43$$

⊢

$$P : y + 1 = 43$$

$$S_1 : x = y + 1;$$

$$S_2 : z = x;$$

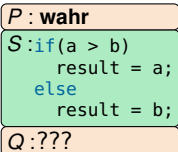
$$Q : z = 43$$



Auswahlregel

Wie behandelt man Verzweigungen in if-else-Anweisungen?

- Zwei alternative Programmsegmente S_1 und S_2



Auswahlregel

Wie behandelt man Verzweigungen in if-else-Anweisungen?

- Zwei alternative Programmsegmente S_1 und S_2
 - Diese werden durch eine Bedingung B unterschieden

```
P: wahr  
S: if(a > b)  
    result = a;  
    else  
    result = b;  
Q: ???
```



```
S0: if(a > b)
```

```
S1:  result = a;  
     else
```

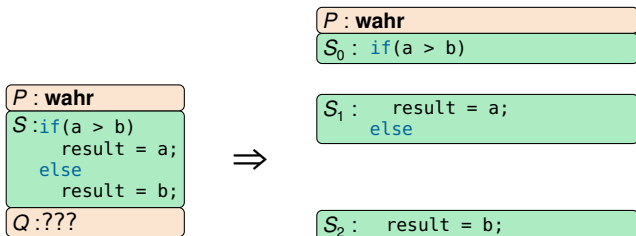
```
S2:  result = b;
```



Auswahlregel

Wie behandelt man Verzweigungen in if-else-Anweisungen?

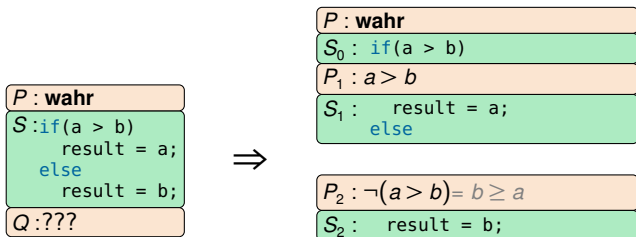
- Zwei **alternative Programmsegmente** S_1 und S_2
 - Diese werden durch eine **Bedingung** B unterschieden
 - Eingangs gilt in beiden Zweigen die Vorbedingung P



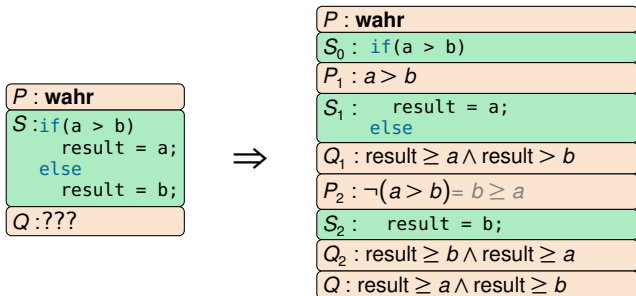
Auswahlregel

Wie behandelt man Verzweigungen in if-else-Anweisungen?

- Zwei **alternative Programmsegmente** S_1 und S_2
 - Diese werden durch eine **Bedingung** B unterschieden
 - Eingangs gilt in beiden Zweigen die Vorbedingung P
 - P und B sind die Basis für die Vorbedingungen für S_1 und S_2
 - $P_1 = P \wedge B$ und $P_2 = P \wedge \neg B$



- Zwei **alternative Programmsegmente** S_1 und S_2
 - Diese werden durch eine **Bedingung** B unterschieden
 - Eingangs gilt in beiden Zweigen die Vorbedingung P
 - P und B sind die Basis für die Vorbedingungen für S_1 und S_2
 - $P_1 = P \wedge B$ und $P_2 = P \wedge \neg B$
 - Die Nachbedingung setzt sich aus denen für S_1 und S_2 zusammen



- Die Nachbedingungen Q_1 und Q_2 für S_1 und S_2 lassen sich mit den hier vorgestellten Regeln in Abhängigkeit von P_1 und P_2 ableiten
 - Ermöglicht eine Vorgehensweise nach dem Schema **Divide & Conquer**
 - Zerlege komplexer Programmsegmente betrachte sie einzeln

- Auswahlregel:

$$\frac{\{P \wedge B\} S_1 \{Q\} \quad \{P \wedge \neg B\} S_2 \{Q\}}{\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \{Q\}}$$



- Wir möchten das Maximum über ein Feld aus Ganzzahlen bilden!
 - Ohne **Iteration** ist dies bei einer unbekanntem Feldgröße nicht möglich
 - Rekursion wäre natürlich eine Lösung, die ohne Iteration auskommt
 - Sie ist jedoch mit denselben Problemen behaftet ...

```
1 int maximum_array(int *array, int size) {
2     int result = INT_MIN;
3
4     for(int i = 0; i < size; i++)
5         result = maximum(array[i], result);
6
7     return result;
8 }
```



Iterationsregel

- Wir möchten das Maximum über ein Feld aus Ganzzahlen bilden!
 - Ohne **Iteration** ist dies bei einer unbekanntem Feldgröße nicht möglich
 - Rekursion wäre natürlich eine Lösung, die ohne Iteration auskommt
 - Sie ist jedoch mit denselben Problemen behaftet ...

```
1 int maximum_array(int *array, int size) {  
2     int result = INT_MIN;  
3  
4     for(int i = 0; i < size; i++)  
5         result = maximum(array[i], result);  
6  
7     return result;  
8 }
```

- Iterationsregel:

$$\frac{\{I \wedge B\} S \{I\}}{\{I\} \mathbf{while} B \mathbf{do} S \mathbf{done} \{I \wedge \neg B\}}$$

- B ist die **Laufbedingung** der Schleife, I ihre **Schleifeninvariante**
 - I gilt **vor**, **während** und **nach** der Ausführung der Schleife
 - Ein geeignetes I ist **manuell zu wählen** (Kunst!)



Iterationsregel – Schleifeninvariante

```
S0 : int result = INT_MIN;
```

```
S1 : for(int i = 0; i < size; i++)
```

```
S2 : result = maximum(array[i], result);
```

- Wo gilt die Schleifeninvariante *I*?



Iterationsregel – Schleifeninvariante

```
S0 : int result = INT_MIN;
```

```
P1 : I
```

```
S1 : for(int i = 0; i < size; i++)
```

```
S2 : result = maximum(array[i], result);
```

- Wo gilt die Schleifeninvariante *I*?
 - Vor der Ausführung der Schleife



```
 $S_0$  : int result = INT_MIN;
```

```
 $P_1$  : !
```

```
 $S_1$  : for(int i = 0; i < size; i++)
```

```
 $P_2$  : !
```

```
 $S_2$  : result = maximum(array[i], result);
```

```
 $Q_2$  : !
```

- **Wo gilt die Schleifeninvariante I ?**
 - Vor der Ausführung der Schleife
 - Vor und nach Ausführung des Schleifenrumpfes



```
S0 : int result = INT_MIN;
```

```
P1 : I
```

```
S1 : for(int i = 0; i < size; i++)
```

```
P2 : I
```

```
S2 : result = maximum(array[i], result);
```

```
Q2 : I
```

```
Q3 : I
```

- **Wo gilt die Schleifeninvariante I?**
 - Vor der Ausführung der Schleife
 - Vor und nach Ausführung des Schleifenrumpfes
 - Nach Beendigung der Schleife



```
S0 : int result = INT_MIN;
```

```
P1 : I
```

```
S1 : for(int i = 0; i < size; i++)
```

```
P2 : I
```

```
S2 : result = maximum(array[i], result);
```

```
Q2 : I
```

```
Q3 : I
```

- Wie lautet die Schleifeninvariante *I*?
 - Eine explizit sichtbare **Laufvariable** hilft bei ihrer Formulierung
 - `result` enthält immer den größten, bereits betrachteten Wert



S_0 : `int result = INT_MIN;`

P_1 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j]$

S_1 : `for(int i = 0; i < size; i++)`

P_2 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j]$

S_2 : `result = maximum(array[i], result);`

Q_2 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j]$

Q_3 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j]$

■ Wie lautet die Schleifeninvariante I ?

- Eine explizit sichtbare **Laufvariable** hilft bei ihrer Formulierung
- `result` enthält immer den größten, bereits betrachteten Wert
- Schleifenbedingung $I = \forall 0 \leq j < i : \text{result} \geq \text{array}[j]$



Iterationsregel – Laufbedingung

S_0 : `int result = INT_MIN;`

P_1 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j]$

S_1 : `for(int i = 0; i < size; i++)`

P_2 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j]$

S_2 : `result = maximum(array[i], result);`

Q_2 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j]$

Q_3 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j]$

- Wie lautet die Laufbedingung B der Schleife und wo gilt sie?



Iterationsregel – Laufbedingung

S_0 : `int result = INT_MIN;`

P_1 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j] \wedge B$

S_1 : `for(int i = 0; i < size; i++)`

P_2 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j] \wedge B$

S_2 : `result = maximum(array[i], result);`

Q_2 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j]$

Q_3 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j] \wedge \neg B$

- Wie lautet die Laufbedingung B der Schleife und wo gilt sie?
 - Sie gilt **vor** der Ausführung des Schleifenrumpfs
 - Sie gilt **nicht** mehr nach der Schleife



Iterationsregel – Laufbedingung

S_0 : `int result = INT_MIN;`

P_1 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j] \wedge i = 0$

S_1 : `for(int i = 0; i < size; i++)`

P_2 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j] \wedge i < \text{size}$

S_2 : `result = maximum(array[i], result);`

Q_2 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j]$

Q_3 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j] \wedge i \geq \text{size}$

- Wie lautet die Laufbedingung B der Schleife und wo gilt sie?
 - Sie gilt **vor** der Ausführung des Schleifenrumpfs
 - Sie gilt **nicht** mehr nach der Schleife
 - Sie lässt sich direkt aus der `for`-Anweisung ablesen $\rightsquigarrow B = i < \text{size}$



Iterationsregel – Verknüpfung

P : wahr

S_0 : `int result = INT_MIN;`

Q_1 : `result = INT_MIN`



P_1 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j] \wedge i = 0$

S_1 : `for(int i = 0; i < size; i++)`

P_2 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j] \wedge i < \text{size}$

S_2 : `result = maximum(array[i], result);`

Q_2 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j]$

Q_3 : $\forall 0 \leq j < i : \text{result} \geq \text{array}[j] \wedge i \geq \text{size}$



Q : $\forall 0 \leq j < \text{size} : \text{result} \geq \text{array}[j]$

- Verknüpfung mithilfe der Sequenzregel (Folie 17)
 - I folgt aus der Vorbedingung P
 - Q folgt aus dem Abbruchkriterium der Schleife $I \wedge \neg B$



- **Vorgehen** beim Anwenden der Iterationsregel



■ Vorgehen beim Anwenden der Iterationsregel

1 Finde eine geeignete Schleifeninvariante I

- Häufig dient der zu berechnene **mathematische Term** als Invariante
- Die **Laufvariable** ist eine weitere Konstruktionshilfe
- Hilfreich ist dessen **geschlossene Darstellung**, falls sie existiert
- z. B. iterative Bestimmung der Fakultät, Fibonacci-Zahlen, ...



■ Vorgehen beim Anwenden der Iterationsregel

1 Finde eine geeignete Schleifeninvariante I

- Häufig dient der zu berechnene **mathematische Term** als Invariante
- Die **Laufvariable** ist eine weitere Konstruktionshilfe
- Hilfreich ist dessen **geschlossene Darstellung**, falls sie existiert
- z. B. iterative Bestimmung der Fakultät, Fibonacci-Zahlen, ...

2 Weise nach, dass I aus der Vorbedingung P folgt: $P \Rightarrow I$

- Im wesentlichen eine Anwendung der **Konsequenzregel** (s. Folie 23)



■ Vorgehen beim Anwenden der Iterationsregel

1 Finde eine geeignete Schleifeninvariante I

- Häufig dient der zu berechnene **mathematische Term** als Invariante
- Die **Laufvariable** ist eine weitere Konstruktionshilfe
- Hilfreich ist dessen **geschlossene Darstellung**, falls sie existiert
- z. B. iterative Bestimmung der Fakultät, Fibonacci-Zahlen, ...

2 Weise nach, dass I aus der Vorbedingung P folgt: $P \Rightarrow I$

- Im wesentlichen eine Anwendung der **Konsequenzregel** (s. Folie 23)

3 Zeige die Invarianz der Invariante: $\{P \wedge I\}S\{I\}$

- **Vollständige Induktion**, falls der Wertebereich der Laufvariable geeignet ist



■ Vorgehen beim Anwenden der Iterationsregel

1 Finde eine geeignete Schleifeninvariante I

- Häufig dient der zu berechnene **mathematische Term** als Invariante
- Die **Laufvariable** ist eine weitere Konstruktionshilfe
- Hilfreich ist dessen **geschlossene Darstellung**, falls sie existiert
- z. B. iterative Bestimmung der Fakultät, Fibonacci-Zahlen, ...

2 Weise nach, dass I aus der Vorbedingung P folgt: $P \Rightarrow I$

- Im wesentlichen eine Anwendung der **Konsequenzregel** (s. Folie 23)

3 Zeige die Invarianz der Invariante: $\{P \wedge I\} S \{I\}$

- **Vollständige Induktion**, falls der Wertebereich der Laufvariable geeignet ist

4 Beweise, dass die Invariante die Nachbedingung impliziert: $I \wedge \neg B \Rightarrow Q$

- Im wesentlichen eine Anwendung der **Konsequenzregel** (s. Folie 23)



Konsequenzregel

- Manchmal ist eine Anpassung der Vor-/Nachbedingung erforderlich
 - z. B. aus technischen Gründen, falls die Vorbedingung $P = \mathbf{wahr}$ ist
 - Ansonsten lässt sich keine sinnvolle Beweiskette aufbauen
- Formalisiert wird dies durch die **Konsequenzregel**

$$\frac{P' \Rightarrow P \quad \{P\}S\{Q\} \quad Q \Rightarrow Q'}{\{P'\}S\{Q'\}}$$

- P' ist eine **Verstärkung** der Vorbedingung P
 - Verstärkungen sind z. B. das Hinzufügen konjunktiv verknüpfter Terme, ...
- Q' ist eine **Abschwächung** der Nachbedingung Q
 - Abschwächungen sind invertierte Verstärkungen
- Die allgemeine Iterationsregel ist eine Anwendung hiervon

$$\frac{P \Rightarrow I \quad \{I\} \mathbf{while\ } B \mathbf{ do\ } S \mathbf{ done\ } \{I \wedge \neg B\} \quad I \wedge \neg B \Rightarrow Q}{\{P\} \mathbf{while\ } B \mathbf{ do\ } S \mathbf{ done\ } \{Q\}}$$





(©Hamilton Richards 2002)

1930 geboren in Rotterdam

ab 1948 Studium an der Universität Leiden

ab 1962 Mathematikprofessor in Eindhoven

ab 1973 *Research Fellow* der Burroughs Corporation

ab 1984 Informatikprofessor in Austin, Texas

1999 Emeritierung

2002 verstorben in Nuenen

Auszeichnungen (Auszug)

1972 Turing Award

1982 Computer Pioneer Award

2002 Dijkstra-Preis

bekannte Werke (Auszug)

■ Dijkstra-Algorithmus [1]

■ Semaphore [4]

■ „GOTO considered harmful“ [2]

- Bestimmt die **schwächste notwendige Vorbedingung** $wp(S, Q)$
 - Für ein gegebenes **imperatives Programmsegment** S
 - Um die ebenfalls gegebene Nachbedingung Q sicherzustellen
 - Dieser Sachverhalt wird beschrieben durch: $P \Rightarrow wp(S, Q)$
 - Lässt sich die schwächste notwendige Vorbedingung $wp(S, Q)$ aus der gegebenen Vorbedingung P folgern?

- Bestimmt die **schwächste notwendige Vorbedingung** $wp(S, Q)$
 - Für ein gegebenes **imperatives Programmsegment** S
 - Um die ebenfalls gegebene Nachbedingung Q sicherzustellen
 - Dieser Sachverhalt wird beschrieben durch: $P \Rightarrow wp(S, Q)$
 - Lässt sich die schwächste notwendige Vorbedingung $wp(S, Q)$ aus der gegebenen Vorbedingung P folgern?
- Das WP-Kalkül ist eine **Rückwärtsanalyse**
 - Sie beginnt mit der Nachbedingung und durchläuft das Programmsegment in umgekehrter Reihenfolge
 - „Sozusagen“ umgekehrter Einsatz der Regeln des Hoare-Kalküls

- Bestimmt die **schwächste notwendige Vorbedingung** $wp(S, Q)$
 - Für ein gegebenes **imperatives Programmsegment** S
 - Um die ebenfalls gegebene Nachbedingung Q sicherzustellen
 - Dieser Sachverhalt wird beschrieben durch: $P \Rightarrow wp(S, Q)$
 - Lässt sich die schwächste notwendige Vorbedingung $wp(S, Q)$ aus der gegebenen Vorbedingung P folgern?
- Das WP-Kalkül ist eine **Rückwärtsanalyse**
 - Sie beginnt mit der Nachbedingung und durchläuft das Programmsegment in umgekehrter Reihenfolge
 - „Sozusagen“ umgekehrter Einsatz der Regeln des Hoare-Kalküls
- Jeder Anweisung wird eine **Prädikattransformation** zugewiesen
 - Abbildung: Nachbedingung \mapsto notwendige schwächste Vorbedingung

→ Eine rückwärtige **symbolisch Ausführung** des Programmsegments



- Axiome für die Anweisungen **skip** und **abort**

$$wp(\mathbf{skip}, Q) = \mathbf{wahr}$$

$$wp(\mathbf{abort}, Q) = \mathbf{falsch}$$

- **skip** ist die leere Anweisung, **abort** schlägt immer fehl

- Zuweisungsaxiom

$$wp(x = y, Q) = Q[x/y]$$

- In der Nachbedingung ersetzt man alle freien Vorkommen von x durch y
 - Dualität von WP-Kalkül und Hoare-Kalkül ist offensichtlich
 - Im Hoare-Kalkül (s. Folie 16) wird y in der Vorbedingung durch x ersetzt

- Sequenzregel

$$wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$$

- Die schwächste Vorbedingung $wp(S_2, Q)$ dient als Nachbedingung für S_1
 - Auch hier ist die Verwandtschaft zum Hoare-Kalkül unverkennbar
 - Dort war $sp(S_1, P)$ die Vorbedingung für S_2 (s. Folie 17)



- Betrachte erneut das Beispiel von Folie 15
 - Diesmal in leicht abgewandelter Form

P : wahr

```
S: int maximum(int a, int b) {  
    int result = INT_MIN;  
  
    if(a > b)  
        result = a;  
    else  
        result = b;  
  
    return INT_MAX;  
}
```

Q : $result \geq a \wedge result \geq b$

- Betrachte erneut das Beispiel von Folie 15
 - Diesmal in leicht abgewandelter Form

```
P : wahr  
S : int maximum(int a, int b) {  
    int result = INT_MIN;  
  
    if(a > b)  
        result = a;  
    else  
        result = b;  
  
    return INT_MAX;  
}  
Q : result ≥ a ∧ result ≥ b
```

- Die Nachbedingung wird ohne Zweifel erfüllt ...
 - ... im Sinne des Erfinders ist dies jedoch bestimmt nicht

- Betrachte erneut das Beispiel von Folie 15
 - Diesmal in leicht abgewandelter Form

```
P : wahr
S : int maximum(int a, int b) {
    int result = INT_MIN;

    if(a > b)
        result = a;
    else
        result = b;

    return INT_MAX;
}
Q : result ≥ a ∧ result ≥ b
```

- Die Nachbedingung wird ohne Zweifel erfüllt ...
 - ... im Sinne des Erfinders ist dies jedoch bestimmt nicht
- ☞ Die Nachbedingung ist **nicht stark genug**, sie ist **unvollständig**
 - **Frage**: Wann ist eine Nachbedingung vollständig?
 - **Frage**: Wie vollständig kann bzw. darf eine Nachbedingung sein?
 - Eine Frage, die sich nicht eindeutig und allgemein klären lässt



- Manches lässt sich mit Prädikatenlogik nicht gut beschreiben
 - Zeitliche Abfolgen: vor Funktion `foo()` muss `bar()` aufgerufen werden
 - Explizite Modellierung über Signalvariablen wird notwendig
 - Nebenläufigkeit und Synchronisation, Zeitschranken, ...



- Manches lässt sich mit Prädikatenlogik nicht gut beschreiben
 - Zeitliche Abfolgen: vor Funktion `foo()` muss `bar()` aufgerufen werden
 - Explizite Modellierung über Signalvariablen wird notwendig
 - Nebenläufigkeit und Synchronisation, Zeitschranken, ...
- Prädikatenlogische Ausdrücke werden sehr schnell sehr komplex
 - Es kommen implizit Bedingungen durch die C-Semantik hinzu
 - Wertebereiche, Funktionsaufrufe, Parametersemantik, Zeigerarithmetik, ...
 - ... etwaige Fehlermeldungen sind sehr schwer zu lesen



- Manches lässt sich mit Prädikatenlogik nicht gut beschreiben
 - Zeitliche Abfolgen: vor Funktion `foo()` muss `bar()` aufgerufen werden
 - Explizite Modellierung über Signalvariablen wird notwendig
 - Nebenläufigkeit und Synchronisation, Zeitschranken, ...
- Prädikatenlogische Ausdrücke werden sehr schnell sehr komplex
 - Es kommen implizit Bedingungen durch die C-Semantik hinzu
 - Wertebereiche, Funktionsaufrufe, Parametersemantik, Zeigerarithmetik, ...
 - ... etwaige Fehlermeldungen sind sehr schwer zu lesen
- Hier und heute wurden **nur partielle Korrektheitsbeweise** betrachtet!
 - **Terminierungsbeweise** müssen separat erbracht werden!
 - Solche Terminierungsbeweise sind mitunter **sehr schwierig!**



- 1 Grundlagen
- 2 Formale Spezifikation
 - Hoare-Kalkül
 - WP-Kalkül
- 3 **Praktische Überlegungen**
 - **Zusicherungen**
 - **Funktionale Verifikation in Astreé**
- 4 Zusammenfassung



Implementierung durch Zusicherungen \mapsto assert()

```
1 unsigned int average(unsigned int *array,  
2                     unsigned int size) {  
3     unsigned long long temp = 0;  
4  
5  
6     for(unsigned int i = 0;i < size;i++) {  
7  
8         temp += array[i];  
9     }  
10  
11     unsigned int result = temp/size;  
12  
13  
14     return result;  
15 }
```



Implementierung durch Zusicherungen \mapsto `assert()`

```
1 unsigned int average(unsigned int *array,  
2                     unsigned int size) {  
3     unsigned long long temp = 0;  
4     assert(size > 0);  
5  
6     for(unsigned int i = 0; i < size; i++) {  
7  
8         temp += array[i];  
9     }  
10  
11     unsigned int result = temp/size;  
12  
13  
14     return result;  
15 }
```




Vorbedingungen lassen sich durch `assert`-Anweisungen prüfen



Implementierung durch Zusicherungen \mapsto assert()

```
1 unsigned int average(unsigned int *array,  
2                     unsigned int size) {  
3     unsigned long long temp = 0;  
4     assert(size > 0);  
5  
6     for(unsigned int i = 0; i < size; i++) {  
7         assert(temp <= ULONG_MAX - array[i]);  
8         temp += array[i];  
9     }  
10  
11     unsigned int result = temp/size;  
12  
13  
14     return result;  
15 }
```

 **Vorbedingungen** lassen sich durch assert-Anweisungen prüfen

- Auch (**Schleifen**)invarianten lassen sich so handhaben



Implementierung durch Zusicherungen \mapsto assert()

```
1 unsigned int average(unsigned int *array,  
2                     unsigned int size) {  
3     unsigned long long temp = 0;  
4     assert(size > 0);  
5  
6     for(unsigned int i = 0; i < size; i++) {  
7         assert(temp <= ULONG_MAX - array[i]);  
8         temp += array[i];  
9     }  
10  
11     unsigned int result = temp/size;  
12     assert(result == average_2(array, size));  
13  
14     return result;  
15 }
```



Vorbedingungen lassen sich durch assert-Anweisungen prüfen

- Auch (**Schleifen**)**invarianten** lassen sich so handhaben



Problematisch sind vor allem **Nachbedingungen**

- Nachbedingungen werden **deklarativ beschrieben**
- In assert-Anweisung wird der Wert typischerweise **explizit konstruiert**
- Begrenzungen sind identisch zu klassischen Tests

→ Sinnvoll, um das Vorhandensein von Defekten zu demonstrieren!



- Astaré wurde entwickelt um **Laufzeitfehler** auszuschließen
 - Basierend auf Abstrakter Interpretation und Programmsemantik
 - Nutzt das Hoare-/WP-Kalkül **nicht** (ist nicht deklarativ)!
- Funktionale Verifikation ist somit **unvollständig**



- Astreé wurde entwickelt um **Laufzeitfehler** auszuschließen
 - Basierend auf Abstrakter Interpretation und Programmsemantik
 - Nutzt das Hoare-/WP-Kalkül **nicht** (ist nicht deklarativ)!
- Funktionale Verifikation ist somit **unvollständig**

```
1 __ASTREE_max_clock((65535)); // Schleifenobergrenze
2 while (1) {
3     __ASTREE_modify((input)); // Reset der Analyse von 'input'
4     __ASTREE_known_fact((input, [0,100])); // Vorbedingung 'input'
5
6     controller_step();
7
8     // Nachbedingung 'output'
9     __ASTREE_assert((0 <= output && output <= 2 * input));
10    __ASTREE_wait_for_clock();
11 }
```

- Funktionale Aspekte lassen sich dennoch in die Analyse einbeziehen
 - Mittels **Zusicherungen** und **Anwendungswissen** (vgl. Folie 30)
 - Der theoretische Hintergrund erleichtert auch hier die Suche!



- Astreé wurde entwickelt um **Laufzeitfehler** auszuschließen
 - Basierend auf Abstrakter Interpretation und Programmsemantik
 - Nutzt das Hoare-/WP-Kalkül **nicht** (ist nicht deklarativ)!
- Funktionale Verifikation ist somit **unvollständig**

```
1 __ASTREE_max_clock((65535)); // Schleifenobergrenze
2 while (1) {
3     __ASTREE_modify((input)); // Reset der Analyse von 'input'
4     __ASTREE_known_fact((input, [0,100])); // Vorbedingung 'input'
5
6     controller_step();
7
8     // Nachbedingung 'output'
9     __ASTREE_assert((0 <= output && output <= 2 * input));
10    __ASTREE_wait_for_clock();
11 }
```

- Funktionale Aspekte lassen sich dennoch in die Analyse einbeziehen
 - Mittels **Zusicherungen** und **Anwendungswissen** (vgl. Folie 30)
 - Der theoretische Hintergrund erleichtert auch hier die Suche!
- **Ein holistischer Verifikationsansatz erfordert weitere Werkzeuge**



```
1  int main()
2  {
3      unsigned int flag = 0;
4      float x=0.0, y=0.0;
5
6      for (unsigned int i = 0, i<10, i++) {
7          if (flag) {
8              x += x/y;
9          } else {
10             flag = 1; x = 1.0; y = 2.0;
11         }
12     }
13 }
```

- Pfadpräfixe (s. VIII/31 ff) abstrahieren von den Schleifendurchläufen
 - Der Schleifenrumpf wird im Extremfall auf einen Pfad reduziert
 - $i = [0, 9]$, $flag = [0, 1]$, $y = [0, 2.0]$



```
1  int main()
2  {
3      unsigned int flag = 0;
4      float x=0.0, y=0.0;
5
6      for (unsigned int i = 0, i<10, i++) {
7          if (flag) {
8              x += x/y;
9          } else {
10             flag = 1; x = 1.0; y = 2.0;
11         }
12     }
13 }
```

■ Pfadpräfixe (s. VIII/31 ff) abstrahieren von den Schleifendurchläufen

- Der Schleifenrumpf wird im Extremfall auf einen Pfad reduziert
- $i = [0, 9]$, $flag = [0, 1]$, $y = [0, 2.0]$



Ausrollen liefert zusätzliche Informationen

- Unterscheidung in ersten und zweiten Durchlauf verhindert den Fehlalarm
- **1) $i = 0$, $flag = 0$ 2) $i = [1, 9]$, $flag = 1$, $y = 2.0$**
- Erhöht jedoch die Kosten dramatisch (vgl. Pfadabdeckung VII/16)



Beispiel: Partitionierung

```
1  unsigned int foo(int cond) {  
2      if (cond) {  
3          x = 10;  
4          y = 5;  
5      } else {  
6          x = 20;  
7          y = 16;  
8      }  
9      return x - y;  
10 }
```

- Sammelsemantik (s. VIII/25 ff) fasst Pfade zusammen
 - Hier kann der Unterlauf nicht ausgeschlossen werden
- Rückgabewert = $[-6, 15]$



Beispiel: Partitionierung

```
1  unsigned int foo(int cond) {  
2      if (cond) {  
3          x = 10;  
4          y = 5;  
5      } else {  
6          x = 20;  
7          y = 16;  
8      }  
9      return x - y;  
10 }
```

■ Sammelsemantik (s. VIII/25 ff) fasst Pfade zusammen

- Hier kann der Unterlauf nicht ausgeschlossen werden

→ Rückgabewert = $[-6, 15]$



Selektive Partitionierung des Kontrollflusses

- Weist die Analyse an, Pfade getrennt zu verfolgen
- **cond**: Rückgabewert = 5, **!cond**: Rückgabewert = 4
- Wiederrum auf Kosten der Komplexität



- 1 Grundlagen
- 2 Formale Spezifikation
 - Hoare-Kalkül
 - WP-Kalkül
- 3 Praktische Überlegungen
 - Zusicherungen
 - Funktionale Verifikation in Astreé
- 4 Zusammenfassung



Funktionale Programmeigenschaften \mapsto Zusicherungen

- Vorbedingungen, Nachbedingungen und Invarianten
- Beschrieben durch Ausdrücke der Prädikatenlogik
- **Prädikamentransformation** \rightsquigarrow symbolische Ausführung
 - Bildet Semantik durch Transformation von Zusicherungen nach
 - **Strongest postcondition, weakest precondition**

Hoare-Kalkül \rightsquigarrow deduktive Ableitung von Nachbedingungen

- **Hoare-Tripel**, Axiome für leere Anweisungen und Zuweisungen
- **Ableitungsregeln** für Sequenzen, Verzweigungen und Iterationen
- **WP-Kalkül** \mapsto „Hoare-Kalkül rückwärts“
- Grenzen des Hoare- und WP-Kalküls

Astreé \rightsquigarrow Ein Verifikationswerkzeug

- Vorrangig zum Ausschluss von **Laufzeitfehlern** (Vollständig)
- Verifikation funktionaler Aspekte möglich (Unvollständig)
- \rightarrow Bottom-up Ansatz (im Gegensatz zu Frama-C, Ada Spark, ...)



- [1] Dijkstra, E. W.:
A note on two problems in connexion with graphs.
In: *Numerische Mathematik* 1 (1959), S. 269–271
- [2] Dijkstra, E. W.:
Letters to the editor: go to statement considered harmful.
In: *Communications of the ACM* 11 (1968), März, Nr. 3, S. 147–148.
<http://dx.doi.org/10.1145/362929.362947>. –
DOI 10.1145/362929.362947. –
ISSN 0001–0782
- [3] Dijkstra, E. W.:
Guarded commands, nondeterminacy and formal derivation of programs.
In: *Communications of the ACM* 18 (1975), Aug., Nr. 8, S. 453–457.
<http://dx.doi.org/10.1145/360933.360975>. –
DOI 10.1145/360933.360975. –
ISSN 0001–0782

- [4] Dijkstra, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Version: 1965.
<http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>.
Eindhoven, The Netherlands, 1965. –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)
- [5] Hoare, C. A. R.:
Algorithm 64: Quicksort.
In: *Communications of the ACM* 4 (1961), Jul., Nr. 7, S. 321–.
<http://dx.doi.org/10.1145/366622.366644>. –
DOI 10.1145/366622.366644. –
ISSN 0001–0782
- [6] Hoare, C. A. R.:
An axiomatic basis for computer programming.
In: *Communications of the ACM* 12 (1969), Okt., Nr. 10, S. 576–580.
<http://dx.doi.org/10.1145/363235.363259>. –
DOI 10.1145/363235.363259. –
ISSN 0001–0782

- [7] Hoare, C. :
Communicating Sequential Processes.
In: *Communications of the ACM* 21 (1978), Aug., Nr. 8, S. 666–677
- [8] Hofstadter, D. R.:
Gödel, Escher, Bach: An Eternal Golden Braid.
Basic Books, 1999. –
824 S. –
ISBN 978–0465026562

