

# Verlässliche Echtzeitsysteme

## WP-Kalkül

Fabian Scheler

Friedrich-Alexander-Universität Erlangen-Nürnberg  
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)  
[www4.informatik.uni-erlangen.de](http://www4.informatik.uni-erlangen.de)

05. Juni 2012



## Fragestellungen

- Wie überprüft man **benutzerdefinierte Korrektheitsbedingungen**?
  - Astrée [3] oder Polyspace sind spezifisch für die Programmiersprache
    - ~ die Bedingungen der Programmiersprache werden also eingehalten
    - ~ aber ist das Programm dann auch korrekt?
- ☞ Design-by-Contract: Angabe von Vor- und Nachbedingungen
  - Wie beschreibt man diese **Verträge**? ~ **Prädikatenlogik**
  - Wie leitet man daraus Korrektheitsaussagen ab? ~ **WP-Kalkül**
- ☞ Beschreibung von Verträgen mit Hilfe von **ACSL**
  - eine **Annotationssprache** für die Programmiersprache C
  - implementiert im Verifikationswerkzeug **Frama-C**



## Gliederung

- 1 Überblick
- 2 Problemstellung
- 3 Hoare-Kalkül
- 4 WP-Kalkül
- 5 Frama-C
- 6 Grenzen
- 7 Zusammenfassung



## Gliederung

- 1 Überblick
- 2 Problemstellung
- 3 Hoare-Kalkül
- 4 WP-Kalkül
- 5 Frama-C
- 6 Grenzen
- 7 Zusammenfassung



## Fehlersuche in C-Programmen

- Zur Erinnerung: 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



## Was der Entwickler wirklich will!

Frei nach der libjustdoit-Manier

- die Funktion average stellt Forderungen an den Aufrufer
    - das Feld array hat genau size korrekt initialisierte Elemente
      - insbesondere sind keine leeren Felder erlaubt ( $\text{size} > 0$ )
    - temp darf nicht überlaufen  $\Rightarrow \text{sum(array, size)} \leq \text{ULONG\_MAX}$
- ~ das sind die **Vorbedingungen**
  - der Aufrufer von average muss sie 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 }
```

- average liefert den Durchschnittswert aller Elemente des Felds array

~ das ist die **Nachbedingung**
  - sie wird durch die Implementierung der Funktion garantiert
  - ~ der Aufrufer von average kann diese Nachbedingung ausnutzen



## Ein korrekt(er)es C-Programm

- 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 }
```

- ✓ Division durch 0 ~ kann nicht mehr auftreten
- ✓ undefinierte Speicherzugriffe auf array ~ kann nicht mehr auftreten
- ✓ Ganzzahlüberlauf in der Variable temp ~ kann nicht mehr auftreten

☞ Aber: Ist diese Implementierung korrekt?

- mit Sicherheit nicht ~ sie liefert einen vollkommen falschen Wert

☞ Wir müssen beschreiben, was wir von average erwarten!



## Man ist vertraglich gebunden ...

- **Zusicherungen** (engl. *assertions*)
  - regeln das Verhältnis zwischen **Aufrufer** und **Prozedur**
- **Vorbedingungen** (engl. *preconditions*)  $P$ 
  - werden vom **Aufrufer** erfüllt, in der **Prozedur** genutzt
- **Nachbedingungen** (engl. *postconditions*)  $Q$ 
  - werden von der **Prozedur** erfüllt, vom **Aufrufer** genutzt
    - unter der Bedingung, dass die Vorbedingungen beim Prozederaufruf gelten
- **Invarianten** (engl. *invariants*)  $I$ 
  - gelten sowohl vor als auch nach dem Prozederaufruf
    - eine zwischenzeitliche Verletzung innerhalb der Prozedur wird toleriert
- salopp formuliert, heißt das:
  - Prozederaufrufe sind **Anweisungen** (engl. *statements*) ~ Bezeichnung  $S$ 
$$P \wedge I \wedge S \Rightarrow Q \wedge I$$
- „nimmt man Vorbedingungen, Invarianten und die Prozedur zusammen, kommt man bei den Nachbedingungen und den Invarianten heraus“



## Zusicherungen ... geht das einfach mit asserts?

- Vorbedingungen lassen sich durch assert-Anweisungen prüfen:

```
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 + array[i] <= ULONG_MAX);
8         temp += array[i];
9     }
10
11    unsigned int result = temp / size;
12    assert(result == average_2(array, size));
13
14    return result;
15 }
```

- auch (Schleifen)invarianten lassen sich so handhaben

problematisch sind vor allem **Nachbedingungen**

- Nachbedingungen werden **deklarativ** beschrieben
  - ↪ in einer assert-Anweisung wird der Vergleichswert aber explizit **konstruiert**
  - ↪ Begrenzungen sind identisch zu klassischen Tests
    - sinnvoll, um das Vorhandensein von Defekten zu demonstrieren, ...



## 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 [9]
- Hoare-Kalkül [10]
- Communicating Sequential Processes [11]

## Gliederung

1 Überblick

2 Problemstellung

3 Hoare-Kalkül

4 WP-Kalkül

5 Frama-C

6 Grenzen

7 Zusammenfassung



## Wie gibt man Zusicherungen an?

- Zusicherungen werden als Formeln der **Prädikatenlogik** beschrieben
- üblicherweise gibt man sie als sog. **Hoare-Triple** an:

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

- $P$  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  $\{P\} S \{\text{falsch}\}$  um auszudrücken, dass  $S$  nicht terminiert



## Wie gibt man Zusicherungen an? (Forts.)

Am Beispiel der Funktion `int maximum(int a,int b)`

```
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 ≥ a ∧ result ≥ b
```

- das Programmsegment ist die Implementierung der Funktion
- Vorbedingung  $P$  : **wahr**
  - ~ die Implementierung stellt keine Anforderungen an die Parameter
- Nachbedingung  $Q$  :  $\text{result} \geq a \wedge \text{result} \geq b$ 
  - ~ „offensichtliche“ Eigenschaft des zu berechnenden Ergebnisses
  - wie man dieses Ergebnis bestimmt, ist hier nicht von Belang



## Das Hoare-Kalkül

- ein **formales System** um Aussagen zur Korrektheit von Programmen zu treffen, die in imperativen Programmiersprachen verfasst sind
- es umfasst **Axiome** für ...
  - leere Anweisungen
  - Zuweisungen
- ... und **Ableitungsregeln** (bzw. **Interferenzregeln**) für
  - Sequenzen (bzw. Komposition) von Anweisungen
  - Auswählen von Anweisungen
  - Iterationen von Anweisungen und
  - Konsequenz
- ist **nicht vollständig** und bezieht sich nur auf die **partielle Korrektheit**
  - andernfalls würde diese eine Lösung des **Halteproblems** bedeuten
  - **Terminierung** ist daher gesondert nachzuweisen



## Wie überprüft man die Einhaltung der Zusicherungen?

- **Aufgabe:** Man muss „ $P$ ,  $S$  und  $Q$  zusammenbringen“!

Prädikattransformation (engl. *predicate transformer semantics*)

- das Programmsegment  $S$  implementiert eine Transformation zwischen der Vorbedingung  $P$  und der Nachbedingung  $Q$ 
  - entsprechende Transformationen existieren für alle Programmstrukturen
  - Zuweisungen, Sequenzen, Verzweigungen, Schleifen, Funktionsaufrufe, ...

- stellen Strategien bereit, um Hoare-Triple  $\{P\} S \{Q\}$  zu beweisen

- eine **Vorwärtsanalyse** liefert die **stärkste Nachbedingung**  $\text{sp}(S, P)$

- (engl. *strongest postcondition, sp*)

- $\{P\} S \{Q\}$  gilt, genau dann wenn  $\text{sp}(S, P) \Rightarrow Q$  wahr ist

- eine Rückwärtsanalyse liefert die **schwächste Vorbedingung**  $\text{wp}(S, Q)$

- (engl. *weakest precondition, wp*)

- $\{P\} S \{Q\}$  gilt, genau dann wenn  $P \Rightarrow \text{wp}(S, Q)$  wahr ist

- Prädikattransformation basiert auf dem **Hoare-Kalkül**

- beschreibt gewissermaßen die denotationelle Semantik eines Programms



## Axiome

- leere Anweisung **skip**

$$\overline{\{P\} \text{skip} \{P\}}$$

- die leere Anweisung verändert den Programmzustand nicht
  - ~ falls  $P$  vor **skip** gilt, gilt es auch danach

- Zuweisung  $x = y$

$$\overline{\{P[y/x]\}x = y \{P\}}$$

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

```
P : y > 100
S :   x = y ;
Q : x > 100
```



## 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:

$$\frac{\begin{array}{c} \{y + 1 = 43\}x = y + 1; \{x = 43\} \\ \{x = 43\}z = x; \{z = 43\} \end{array}}{\{y + 1 = 43\}x = y + 1; z = x; \{z = 43\}}$$

$\vdash$

|                    |
|--------------------|
| $P : y + 1 = 43$   |
| $S_1 : x = y + 1;$ |
| $Q : x = 43$       |

|                    |
|--------------------|
| $P : y + 1 = 43$   |
| $S_1 : x = y + 1;$ |
| $S_2 : z = x;$     |
| $Q : z = 43$       |

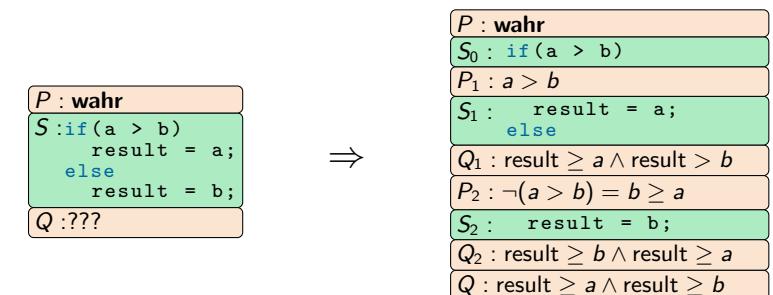


## Auswahlregel

Wie behandelt man Verzweigungen in if-else-Anweisungen?

- zwei **alternative Programmsegmente**  $S_1$  und  $S_2$

- diese werden durch eine **Bedingung**  $B$  unterschieden
- eingangs gilt in beiden Zweigen die Vorbedingung  $P$ 
  - sie und  $B$  ist 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



## Auswahlregel (Forts.)

- 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
- Ableitungsregel:

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



## Iterationsregel

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

```

1 int maximum(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 }
```

- Ableitungsregel:

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

- $B$  ist die **Laufbedingung** der Schleife,  $I$  ihre **Schleifeninvariante**
  - $I$  gilt **vor**, **während** und **nach** der Ausführung der Schleife
  - ein geeignetes  $I$  **manuell zu wählen** ist die Kunst
- partielle Korrektheit  $\leadsto$  anderweitiger Terminierungsbeweis notwendig!



## Iterationsregel (Forts.)

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

|                                                      |
|------------------------------------------------------|
| P <sub>1</sub> : I                                   |
| S <sub>1</sub> : for(int i = 0; i < size; i++)       |
| P <sub>2</sub> : I                                   |
| S <sub>2</sub> : result = maximum(array[i], result); |
| Q <sub>2</sub> : I                                   |
| Q <sub>3</sub> : I                                   |

## Iterationsregel (Forts.)

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

|                                                                |
|----------------------------------------------------------------|
| P <sub>1</sub> : $\forall 0 \leq j < i : result \geq array[j]$ |
| S <sub>1</sub> : for(int i = 0; i < size; i++)                 |
| P <sub>2</sub> : $\forall 0 \leq j < i : result \geq array[j]$ |
| S <sub>2</sub> : result = maximum(array[i], result);           |
| Q <sub>2</sub> : $\forall 0 \leq j < i : result \geq array[j]$ |
| Q <sub>3</sub> : $\forall 0 \leq j < i : result \geq array[j]$ |

- Wo gilt die Schleifeninvariante? Sie gilt
  - vor der Ausführung der Schleife
  - vor und nach Ausführung des Schleifenrumpfes
  - sowie nach Beendigung der Schleife



## Iterationsregel (Forts.)

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

|                                                                                   |
|-----------------------------------------------------------------------------------|
| P <sub>1</sub> : $\forall 0 \leq j < i : result \geq array[j]$                    |
| S <sub>1</sub> : for(int i = 0; i < size; i++)                                    |
| P <sub>2</sub> : $\forall 0 \leq j < i : result \geq array[j] \wedge i < size$    |
| S <sub>2</sub> : result = maximum(array[i], result);                              |
| Q <sub>2</sub> : $\forall 0 \leq j < i : result \geq array[j]$                    |
| Q <sub>3</sub> : $\forall 0 \leq j < i : result \geq array[j] \wedge i \geq size$ |

## Iterationsregel (Forts.)

|                                        |
|----------------------------------------|
| P : wahr                               |
| S <sub>0</sub> : int result = INT_MIN; |
| Q <sub>1</sub> : result = INT_MIN      |



|                                                                                   |
|-----------------------------------------------------------------------------------|
| P <sub>1</sub> : $\forall 0 \leq j < i : result \geq array[j]$                    |
| S <sub>1</sub> : for(int i = 0; i < size; i++)                                    |
| P <sub>2</sub> : $\forall 0 \leq j < i : result \geq array[j] \wedge i < size$    |
| S <sub>2</sub> : result = maximum(array[i], result);                              |
| Q <sub>2</sub> : $\forall 0 \leq j < i : result \geq array[j]$                    |
| Q <sub>3</sub> : $\forall 0 \leq j < i : result \geq array[j] \wedge i \geq size$ |



|                                                      |
|------------------------------------------------------|
| Q : $\forall 0 \leq i < size : result \geq array[i]$ |
|------------------------------------------------------|

- Wie lautet die Laufbedingung 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  $\sim B = i < size$



- Wie verknüpft man nun die Schleife mit dem Rest des Programms?
  - generalisierte Beschreibung der Schleife:  $\{P\} \text{while } B \text{ do } S \text{ done } \{Q\}$
  - I folgt aus der Vorbedingung P
  - Q folgt aus dem Abbruchkriterium der Schleife  $I \wedge \neg B$



## Iterationsregel (Forts.)

### Vorgehen beim Anwenden der Iterationsregel

#### 1 Finde eine geeignete Schleifeninvariante $I$

- häufig dient der zu berechnende **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 VI/23)

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

- **vollständiger 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 VI/23)



## Gliederung

- 1 Überblick
- 2 Problemstellung
- 3 Hoare-Kalkül
- 4 WP-Kalkül
- 5 Frama-C
- 6 Grenzen
- 7 Zusammenfassung



## Konsequenzregel

- manchmal ist eine Anpassung der Vor-/Nachbedingung erforderlich
  - z. B. aus technischen Gründen, falls die Vorbedingung  $P = \text{wahr}$  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

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



## Edger W. Dijkstra

Ein Informatik-Pionier: Leben und Wirken



(©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)      **bekannte Werke** (Auszug)  
1972 Turing Award      ■ Dijkstra-Algorithmus [5]  
1982 Computer Pioneer Award      ■ Semaphore [8]  
2002 Dijkstra-Preis      ■ „GOTO considered harmful“ [6]



## WP-Kalkül [7]

- 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



## Gliederung

- 1 Überblick
- 2 Problemstellung
- 3 Hoare-Kalkül
- 4 WP-Kalkül
- 5 Frama-C
- 6 Grenzen
- 7 Zusammenfassung



## Axiome und Sequenzregel

Die restlichen Regeln gleichen ebenfalls denen des Hoare-Kalküls

- Axiome für die Anweisungen **skip** und **abort** $wp(\text{skip}, Q) = \text{wahr}$        $wp(\text{abort}, Q) = \text{falsch}$ 
  - **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 VI/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 VI/17)

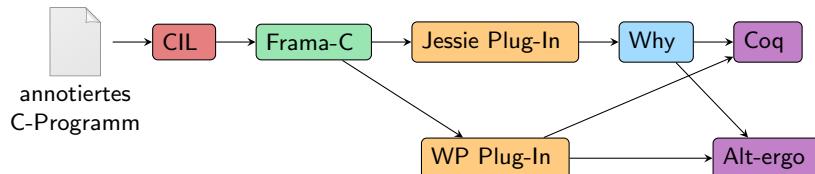


## Überblick

- Frama-C [2] ist ein **Rahmen** (engl. *framework*) für statische Analysen
  - Fokussierung auf die Programmiersprache C
  - stellt grundlegende Dienste für Analysen bereit
    - Analyse und Transformation von C-Programmen mithilfe des **abstrakten Syntaxbaums** (engl. *abstract syntax tree, AST*) basierend auf CIL [12]
    - Analyse und Transformation von **ACSL-Annotations** basierend auf CIL
    - Integration von Algorithmen und Verbänden für **abstrakte Interpretation**
    - Verwaltung der **Speicherzustände** von C-Programmen
- Analysen werden durch Plug-Ins implementiert
  - Intervallanalyse, Program Slicing, Metriken, **WP-Kalkül**, ...
  - Plug-Ins arbeiten **kooperativ**
    - sie können gegenseitig auf Ergebnisse zugreifen
    - ~ wähle das am besten geeignete Plug-In für die Verifikation einer Eigenschaft
    - ~ verwende diese Ergebnisse für die Verifikation anderer Eigenschaften
- **Austauschformat:** ANSI/ISO C Specification Language (ACSL) [1]
  - Annotationssprache um Zusicherungen für C-Programm zu beschreiben



## WP-Kalkül in Frama-C



- 1 alles fängt mit einem annotierten C-Programm an
- 2 dieses wird von in CIL für Frama-C in einen AST transformiert
- 3 die Plug-Ins Jessie und WP [4] implementieren das WP-Kalkül
  - Jessie konvertiert das C-Programm zunächst in ein Why-Programm
    - dieses bestimmt dann die schwächste Vorbedingung
    - und versucht sie mithilfe von Theorembeweisern zu verifizieren
  - das WP Plug-In hingegen erzeugt die schwächste Vorbedingung direkt
    - im Gegensatz zu Jessie lassen sich seine Ergebnisse weiterverwenden
    - es ist aber noch nicht so ausgereift wie Jessie

## Gliederung

- 1 Überblick
- 2 Problemstellung
- 3 Hoare-Kalkül
- 4 WP-Kalkül
- 5 Frama-C
- 6 Grenzen
- 7 Zusammenfassung



## Beispiel: int maximum(int a,int b)

```
/*@ ensures \result >= a && \result >= b; */
int maximum(int a,int b) {
    int result = INT_MIN;

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

    return result;
}
```

- ensures kennzeichnet einzuhaltende Nachbedingungen
  - das Ergebnis muss mindestens so groß sein wie a bzw. b
- /\*@ requires \valid(a) && \valid(b);
 ensures \result >= \*a && \result >= \*b; \*/
int maximum(int \*a,int \*b) {
 int result = INT\_MIN;

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

 return result;
}
- requires kennzeichnet geforderte Vorbedingungen
  - die Zeiger a bzw. b müssen auf gültige Speicherstellen deuten

## Grenzen des WP-Kalküls

- Betrachte erneut eines der Beispiele auf Folie VI/31
  - diesmal aber in leicht abgewandelter Form
- /\*@ ensures \result >= a && \result >= b; \*/
int maximum(int a,int b) {
 int result = INT\_MIN;

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

 return INT\_MAX;
}
- die Nachbedingung wird ohne Zweifel erfüllt
  - im Sinne des Erfinders ist dies aber 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, Zeitschränken, ...
- 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 aber sehr schwierig!
- unvollständige Hintergrundtheorie der prädikatenlogischen Formeln
  - sie beschreibt die Semantik der einzelnen Aussagen in diesen Formeln
    - z. B. prädikatenlogische Ausdrücke über linearen arithmetischen Termen



## Zusammenfassung

Funktionale Programmeigenschaften → Zusicherungen

- Vorbedingungen, Nachbedingungen und Invarianten
- beschrieben durch Ausdrücke der Prädikatenlogik

Prädikatentransformation → symbolische Ausführung

- bildet Semantik durch Transformation von Zusicherungen nach
- strongest postcondition, weakest precondition

Hoare-Kalkül → deduktive Ableitung von Nachbedingungen

- Hoare-Tripel, Axiome für leere Anweisungen und Zuweisungen
- Ableitungsregeln für Sequenzen, Verzweigungen und Iterationen
- Konsequenzregel passt Vor-/Nachbedingungen an

WP-Kalkül → „Hoare-Kalkül rückwärts“

- wird von Frama-C in den Plug-Ins WP und Jessie implementiert

Grenzen des WP-Kalküls



- 1 Überblick
- 2 Problemstellung
- 3 Hoare-Kalkül
- 4 WP-Kalkül
- 5 Frama-C
- 6 Grenzen
- 7 Zusammenfassung



## Literaturverzeichnis

- [1] BAUDIN, P. ; CUOQ, P. ; FILLIÂTRE, J. ; MARCHÉ, C. ; MONATE, B. ; MOY, Y. ; PREVOSTO, V. : *ACSL: ANSI/ISO C Specification Language. Preliminary Design (version 1.5)*. [http://frama-c.com/download/acsl\\_1.5.pdf](http://frama-c.com/download/acsl_1.5.pdf), 2011
- [2] CORRENSON, L. ; CUOQ, P. ; KIRCHNER, F. ; PREVOSTO, V. ; PUCCETTI, A. ; SIGNOLES, J. ; YAKOBOWSKI, B. : *Frama-C User Manual*. <http://frama-c.com/download/frama-c-user-manual.pdf>, 2011
- [3] COUSOT, P. ; COUSOT, R. ; FERET, J. ; MAUBORGNE, L. ; MINÉ, A. ; MONNIAUX, D. ; RIVAL, X. : *The ASTREE Analyzer*. In: SAGIV, S. (Hrsg.): *Proceedings 14th European Symposium on Programming (ESOP '05)* Bd. 3444. Heidelberg, Germany : Springer-Verlag, Apr. 2005 (Lecture Notes in Computer Science). – ISBN 3-540-25435-8, S. 21–30



## Literaturverzeichnis (Forts.)

- [4] CUOQ, P. ; MONATE, B. ; PACALET, A. ; PREVOSTO, V. :  
Functional dependencies of C functions via weakest pre-conditions.  
In: *International Journal on Software Tools for Technology Transfer* 13 (2011), S. 405–417.  
<http://dx.doi.org/10.1007/s10009-011-0192-z>. –  
DOI 10.1007/s10009-011-0192-z. –  
ISSN 1433–2779
- [5] DIJKSTRA, E. W.:  
A note on two problems in connexion with graphs.  
In: *Numerische Mathematik* 1 (1959), S. 269–271
- [6] 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



## Literaturverzeichnis (Forts.)

- [7] 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
- [8] 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)
- [9] 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



## Literaturverzeichnis (Forts.)

- [10] 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
- [11] HOARE, C.:  
Communicating Sequential Processes.  
In: *Communications of the ACM* 21 (1978), Aug., Nr. 8, S. 666–677
- [12] NECULA, G. C. ; MCPKEA, S. ; RAHUL, S. P. ; WEIMER, W. :  
CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs.  
In: HORSPOOL, R. N. (Hrsg.): *Proceedings of the 11th International Conference on Compiler Construction (CC '02)* Bd. 2304.  
Springer-Verlag : Springer-Verlag, Apr. 2002 (Lecture Notes in Computer Science).  
–  
ISBN 3–540–43369–4, S. 213–228

