Kapitel L:V

V. Erweiterungen und Anwendungen zur Logik

- Produktionsregelsysteme
- Inferenz für Produktionsregelsysteme
- Produktionsregelsysteme mit Negation
- Regeln mit Konfidenzen
- Nicht-monotones Schließen
- Logik und abstrakte Algebren
- Verifikation
- Verifikation mit dem Hoare-Kalkül
- Hoare-Regeln und partielle Korrektheit
- Terminierung

Bemerkungen:

- Literatur zu diesem Kapitel findet sich online:
  "'Semantics with Applications"' von Hanne Riis Nielson und Flemming Nielson.
- Eine neuere Version dieses Buches findet man bei Springer:
  "'Semantics with Applications: An Appetizer"' von Hanne Riis Nielson und Flemming Nielson.
  (siehe auch SpringerLink)
- Eine weitere gute Quelle, erschienen bei Springer:
  „The Science of Programming“ von David Gries (Springer 1981).

Verifikation

Software-Qualität

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

[C.A.R. Hoare, 1980]

Verifikation

Software-Qualität

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

[C.A.R. Hoare, 1980]

Software wird zur Benutzung freigegeben, nicht wenn sie nachweislich korrekt ist, sondern wenn die Häufigkeit, mit der neue Fehler entdeckt werden, auf ein für die Geschäftsleitung akzeptables Niveau gesunken ist.

[David L. Parnas (?)]

Verifikation

Software-Qualität

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

[C.A.R. Hoare, 1980]

Software wird zur Benutzung freigegeben, nicht wenn sie nachweislich korrekt ist, sondern wenn die Häufigkeit, mit der neue Fehler entdeckt werden, auf ein für die Geschäftsleitung akzeptables Niveau gesunken ist.

[David L. Parnas (?)]

Jedes sechste DV-Projekt wurde ohne jegliches Ergebnis abgebrochen, alle Projekte überzogen die Zeit- und Kostenrahmen um 100-200% und auf 100 ausgelieferte Programmzeilen kommen im Durchschnitt drei Fehler.

[Tom deMarco, 1991]

Verifikation

Anwendungsbedarf

- Informationssicherheit (Security)
  - Vertraulichkeit
  - Integrität
  - Authentizität
  - Nicht-Rückweisbarkeit (Signaturgesetz)

  Zertifizierung von IT-Systemen durch das Bundesamt für Sicherheit in der Informationstechnik:

  > Höhere Stufen der Vertrauenswürdigkeit erfordern formale Spezifikation und formale Verifikation.

  Beispiele
  - Home Banking
  - Geld- und Chipkarten

Verifikation

Anwendungsbedarf (Fortsetzung)

- Systemsicherheit (Safety)

  Software für sicherheitskritische Systeme ist formal zu spezifizieren und zu verifizieren.

  Beispiele:
  Eingebettete Systeme (Embedded Systems) als Regelungssysteme / reaktive Systeme unter Berücksichtigung von Realzeitaspekten in

  - Autos,
  - Flugzeugen,
  - Raumfahrzeugen,
  - Anlagensteuerungen.

Verifikation

Testen als Alternative

Tests können die *Anwesenheit* von Fehlern beweisen, aber nie die *Abwesenheit* von Fehlern (bei unendlich vielen möglichen Eingaben).

Klassifikation von Testverfahren:

- Schnittstellentest (Blackbox-Test)
  Die Ein- / Ausgaberelation wird auf Übereinstimmung mit der Spezifikation geprüft.

- Programmabhängiger Test (Whitebox-Test)
  Möglichst große Teile aller Pfade durch das Programm werden getestet. Eine möglichst große Überdeckung (des Programmcodes) ist erwünscht.

Verifikation

Testen als Alternative (Fortsetzung)

Systematische Auswahl von Testfällen:

- Schnittstellentest
  Pro spezifizierter Bedingung mindestens einen Testfall prüfen, Randbereiche (ggf. von beiden Seiten) prüfen, Maximal-, Minmalwerte nicht vergessen, eine genügend große Anzahl von Normalfällen prüfen.
- Überdeckungstest
  Erwünscht, aber kaum machbar ist eine Wegüberdeckung d.h. jeder Weg wird mindestens einmal durchlaufen.
  Auf jeden Fall nötig ist eine Anweisungsüberdeckung, d.h. jede Anweisung wird mindestens einmal durchlaufen.

Hauptproblem des Testens:

Kombinatorische Explosion der Möglichkeiten für Testfälle

Verifikation

Aufgaben bei der Softwareentwicklung

... unter anderem:

- Spezifikation
  Was soll die Software eigentlich leisten?
- Implementierung
  Wie soll etwas gemacht werden?
- Korrektheitsprüfung
  Tut das Programm auch das, was es soll?
- Komplexitätsuntersuchung
  Wie gut ist das Programm eigentlich (Zeit, Platz, Struktur)?

Verifikation

Korrektheitsprüfung: Softwaretest

Fehler

- syntaktische Fehler
- semantische Fehler
- Terminierungsfehler

Methoden

- Test
  Überprüfung der Korrektheit einer Implementierung für endlich viele Eingaben
  ➜ Dynamischer Test
- Verifikation
  Nachweis, dass eine Implementierung fehlerfrei das macht, was eine Spezifikation vorschreibt.
  ➜ Statischer Test

Verifikation

Algorithmus und Programm

Algorithmus:

Unter einem Algorithmus versteht man eine Verarbeitungsvorschrift, die so präzise formuliert ist, dass sie von einem mechanisch oder elektronisch arbeitenden Gerät durchgeführt werden kann. Aus der Präzision der sprachlichen Darstellung des Algorithmus muss die Abfolge der einzelnen Verarbeitungsschritte eindeutig hervorgehen. . . .

[Duden - Informatik]

Verifikation

Algorithmus und Programm (Fortsetzung)

Programm

Formulierung eines Algorithmus und der zugehörigen Datenbereiche in einer Programmiersprache.
Während Algorithmen relativ allgemein beschrieben werden können und an keine formellen Vorschriften gebunden sind, müssen Programme wesentlich konkreter sein:

- *Sie sind im exakt definierten und eindeutigen Formalismus einer Programmiersprache verfasst.*
- *Sie nehmen Bezug auf eine bestimmte Darstellung der verwendeten Daten.*
- *Sie sind auf einer Rechenanlage ausführbar.*

Ein und derselbe Algorithmus kann in verschiedenen Programmiersprachen formuliert werden; er bildet eine Abstraktion aller Programme, die ihn beschreiben.

[Duden - Informatik]

Verifikation

Arten von Programmierparadigmen

- imperativ (prozedural) z.B. Pascal, C

```
int fibonacci(int x) {
  if ( x == 0) return 1;
  if ( x == 1) return 1;
  return fibonacci(x-1) + fibonacci(x-2);
}
```

- funktional z.B. Lisp

```
(defun fibonacci (schrittzahl)
  (cond
    ((= schrittzahl 0) 1)
    ((= schrittzahl 1) 1)
    ((> schrittzahl 1)
       (+ (fibonacci (- schrittzahl 1))
          (fibonacci (- schrittzahl 2))))
  )
)
```

Verifikation

Programmierparadigmen (Fortsetzung)

- deklarativ z.B. Prolog

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(X,Y) :- X1 is X-1, fibonacci(X1,Y1),
                  X2 is X-2, fibonacci(X2,Y2),
                  Y is X1+X2.
```

- objektorientiert z.B. Java

```
class Fibonacci extends IntegerSequence
                implements Displayable {
    int computeMember(int x) {
        ...
    }
}
```

Verifikation

Programmierparadigmen (Fortsetzung)

- deklarativ z.B. Prolog

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(X,Y) :- X1 is X-1, fibonacci(X1,Y1),
                  X2 is X-2, fibonacci(X2,Y2),
                  Y is X1+X2.
```

- objektorientiert z.B. Java

```
class Fibonacci extends IntegerSequence
                implements Displayable {
    int computeMember(int x) {
        ...
    }
}
```

Eine Verifikation ist angepasst an Programmierparadigma

➜ Wir betrachten hier als Beispiel eine einfache imperative Programmiersprache.

Verifikation

Eine einfache imperative Programmiersprache

Variable

- Integer-Variable und Float-Variable
  Variable zur Aufnahme von Zahlenwerten beliebiger Größe/Genauigkeit

Konstanten

- Integer-Konstanten und Float-Konstanten
  Angabe fester Zahlenwerte

Verifikation

Eine einfache imperative Programmiersprache

Variable

- Integer-Variable und Float-Variable
  Variable zur Aufnahme von Zahlenwerten beliebiger Größe/Genauigkeit

Konstanten

- Integer-Konstanten und Float-Konstanten
  Angabe fester Zahlenwerte

Ausdrücke

- Arithmetic Expression
  Arithmetische Ausdrücke mit Zahlenwerten als Ergebnissen bestehend aus Konstanten, Variablen und Operatoren $(+, -, *, /, ...)$
- Boolean Expression
  Boolesche Ausdrücke bestehend aus arithmetische Ausdrücken mit Vergleichsoperatoren $(<, >, =, \leq, ...)$ und den Booleschen Konnektoren $(\texttt{and}, \texttt{or}, \texttt{not})$.

Verifikation

Eine einfache imperative Programmiersprache (Fortsetzung)

Elementare Anweisungen (Statements)

- Zuweisung

  <Variable> := <Arithmetic-Expression> ;

- Bedingte Anweisung
  - einseitig

    `if` (<Boolean-Expression>)
    `then` <Anweisung>

  - zweiseitig

    `if` (<Boolean-Expression>)
    `then` <Anweisung>
    `else` <Anweisung>

- Schleife

  `while` (<Boolean-Expression>) `do`
  <Anweisung>

Verifikation

Eine einfache imperative Programmiersprache (Fortsetzung)

Anweisungen und Programm

- Anweisungsblock

  ```
  begin
      <Anweisungsfolge>
  end
  ```

- Anweisungsfolge

  Eine Anweisungsfolge besteht aus einer endlichen Folge von Anweisungen. Eine Anweisungsfolge kann nicht leer sein.

- Anweisung

  Eine Anweisung ist eine Zuweisung, eine bedingte Anweisung, eine Schleife oder ein Anweisungsblock.

- Programm

  Ein Programm besteht aus einem Anweisungsblock.

Bemerkungen:

- Zur Vereinfachung verwenden wir eine Single-Entry-Single-Exit-Sprache: Es gibt nur jeweils eine Stelle, an der bei einem Programmaufruf die Ausführung beginnt und ebenso genau eine Stelle, an der die Programmausführung endet.
- Wir geben keine Deklaration in Form eines Prozedur- oder Funktionskopfes an und verzichten auf ein explizites Return-Statement.
- Der später vorgestellte Ansatz zur Verifikation läßt sich aber in einfacher Weise auf Prozedur- und Funktionsaufrufe in Expressions erweitern. Rekursive Funktionen verlangen jedoch ein ähnlich komplexes Vorgehen wie bei den Schleifen und ausserdem sollte man temporäre Variablen und die Sichtbarkeit von Variablen einführen.

Verifikation

Beispiel für ein Programm

Programm P:

```
begin
   a := x;
   b := y;
   while (a > 0) do
      begin
         a := a − 1;
         b := b + 1;
      end
end
```

Was leistet dieses Programm?

Verifikation

Einfache imperative Programmiersprache (Fortsetzung)

Vereinbarungen:

- Eingabe:
  Die Eingabewerte für ein Programm werden in speziellen Variablen übergeben. Diese Variablen werden durch das Programm *nicht* verändert.
- Initialisierung:
  Alle anderen Variablen enthalten einen beliebigen Wert, müssen also initialisiert werden. Außer den Eingabevariablen dürfen die Werte aller anderen Variablen überschrieben werden.
- Ausgabe:
  Das Ergebnis des Programmes wird in einer Variablen gespeichert, die keinen Eingabewert bereitstellt.
- Ausgabe:
  Das Ergebnis des Programmes steht in dieser Variablen auch nach dem Programmende zur Verfügung. Diese Variablen sind also *global*.

Verifikation

Beispiel für ein Programm

Programm P:

Eingabevariable x, y
Ausgabevariable b

```
begin
   a := x;
   b := y;
   while (a > 0) do
      begin
         a := a − 1;
         b := b + 1;
      end
end
```

Was leistet dieses Programm?

Verifikation

Semantik formaler Sprachen

- Übersetzersemantik
- Operationale Semantik
- Denotationelle Semantik
- Axiomatische Semantik

Verifikation

Semantik formaler Sprachen

- Übersetzersemantik

  Bedeutung wird vom Übersetzer (Compiler) durch Transformation in eine (einfache) Zielsprache bestimmt.

- Operationale Semantik

  Bedeutung wird durch Abläufe in einer abstrakten Maschine (Automat) bestimmt.

- Denotationelle Semantik

  Bedeutung wird durch eine Funktion $f : E \to A$ definiert, die die Eingangszustände E auf die Ausgangszustände A abbildet.

- Axiomatische Semantik

  Bedeutung wird durch Axiome zur Beschreibung von Voraussetzungen und Ergebnissen von Anweisungen bestimmt.

Verifikation mit dem Hoare-Kalkül

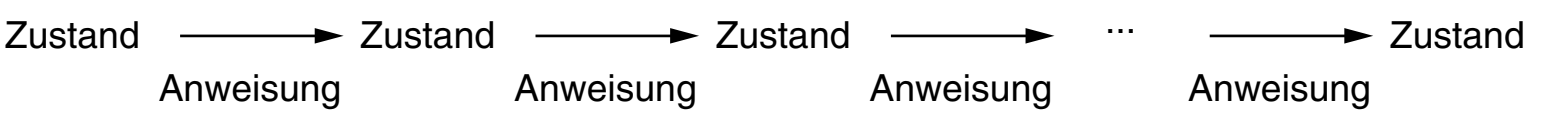
Der Hoare-Kalkül (auch Hoare-Logik) ist ein Formales System, entwickelt von dem britischen Informatiker C.A.R. Hoare und später verfeinert von Hoare und anderen Wissenschaftlern. Er wurde 1969 in einem Artikel mit dem Titel „An Axiomatic Basis for Computer Programming“ veröffentlicht. Der Zweck des Systems ist es, eine Menge von logischen Regeln zu liefern, die es erlauben, Aussagen über die Korrektheit von imperativen Computer-Programmen zu treffen und sich dabei der mathematischen Logik zu bedienen.

[Wikipedia, 2009]

Verifikation mit dem Hoare-Kalkül

Imperative Programmierung

- Zustandsorientierte Programmierung
- Zustand = aktuelle Belegung aller Variablen an einer Stelle im Programm (auf abstraktem Niveau, kein System-Stack, etc.)
- Ausführung von Anweisungen verändert Variablenbelegungen.
- Jede Anweisung kann einzeln betrachtet werden. (Anweisung $\neq$ Zeile)
- Aus der Beschreibung des Zustands vor einer Anweisung läßt sich der Zustand nach der Instruktion ableiten.



➜ Beschreibung von Zuständen durch sogenannte Zusicherungen.

➜ Zusicherungen abstrahieren so weit, dass alle Zustände erfasst sind, die an einer Stelle möglich sind. (Wichtig bei Schleifen.)

Verifikation mit dem Hoare-Kalkül

Annotation durch Zusicherungen

- *Zusicherungen sind (mathematische) Aussagen über die (Werte der) Variablen in einem Programm.*
- Zusicherungen enthalten Programmvariablen als freie Identifikatoren.
- Jede Zusicherung bezieht sich auf eine bestimmte Stelle in einem Programm. (Mögliche Stellen sind *nur* die Stellen vor oder nach Anweisungen.)
- Zusicherungen werden in geschweifte Klammern eingeschlossen. Beispiel: $\{x, y \in \mathbf{N}$ *und* $x \geq 5$ *und* $y > 2x\}$
- Man kann Zusicherungen als (formale) Kommentare in Programmen ansehen.
- *Zusicherungen sind gültig, falls sie in jedem an der entsprechenden Stelle während eines Programmablaufes möglicherweise auftretenden Zustand erfüllt sind (Floyd/Hoare).*

Verifikation mit dem Hoare-Kalkül

Spezifikation als spezielle Zusicherungen

Die Spezifikation eines Programms P besteht aus

1. einer Zusicherung V in den Eingabevariablen des Programms, die *Vorbedingung (precondition)* von P genannt wird,
2. einer Zusicherung N in den Ausgabevariablen des Programms, die *Nachbedingung (postcondition)* von P genannt wird.

- V soll die erlaubten Eingaben für das Programm beschreiben.
- N beschreibt, welches Ergebnis für diese Eingaben in welchen Ausgabevariablen berechnet werden soll.
- Man schreibt kurz: $\{V\}P\{N\}$

➜ Während die Gültigkeit von V als gegeben angenommen wird, wollen wir uns von der Gültigkeit von N überzeugen.

Verifikation mit dem Hoare-Kalkül

Beispiel für eine Spezifikation

P sei ein Programm zur Fakultätsberechnung.

Eingabevariable n
Ausgabevariable y

Spezifikation:

Programm P:

Vorbedingung $\{ n \in \mathbf{Z}$ *und* $n \geq 0\}$
Nachbedingung $\{ y = n!$ *und* $n \in \mathbf{Z}$ *und* $n \geq 0\}$

Verifikation mit dem Hoare-Kalkül

Verifikation auf Basis von Zusicherungen

- Die Spezifikation eines Programmes beschreibt
  - in der Vorbedingung eine Abstraktion des (für das Programm relevanten Teil des) Zustands vor der Programmausführung und
  - in der Nachbedingung eine Abstraktion des (für den Programmbenutzer relevanten Teil des) Zustands nach der Programmausführung.

Verifikation mit dem Hoare-Kalkül

Verifikation auf Basis von Zusicherungen

- Die Spezifikation eines Programmes beschreibt
  - in der Vorbedingung eine Abstraktion des (für das Programm relevanten Teil des) Zustands vor der Programmausführung und
  - in der Nachbedingung eine Abstraktion des (für den Programmbenutzer relevanten Teil des) Zustands nach der Programmausführung.

- Analog zur Spezifikation des gesamten Programmes können wir Spezifikationen einzelner Anweisungen betrachten.
  - Vorbedingung einer Anweisung
    ist eine Zusicherung vor der Ausführung der Anweisung.
  - Nachbedingung einer Anweisung
    ist eine Zusicherung für den aus der Vorbedingung durch Ausführung der Anweisung resultierenden Zustand.

Verifikation mit dem Hoare-Kalkül

Verifikation auf Basis von Zusicherungen

- Die Spezifikation eines Programmes beschreibt
  - in der Vorbedingung eine Abstraktion des (für das Programm relevanten Teil des) Zustands vor der Programmausführung und
  - in der Nachbedingung eine Abstraktion des (für den Programmbenutzer relevanten Teil des) Zustands nach der Programmausführung.

- Analog zur Spezifikation des gesamten Programmes können wir Spezifikationen einzelner Anweisungen betrachten.
  - Vorbedingung einer Anweisung
    ist eine Zusicherung vor der Ausführung der Anweisung.
  - Nachbedingung einer Anweisung
    ist eine Zusicherung für den aus der Vorbedingung durch Ausführung der Anweisung resultierenden Zustand.

➜ Bei der Verifikation eines Programmes mit einer vorgegebenen Spezifikation werden die Zusicherungen vor und nach jeder Anweisung des Programmes untersucht.

Verifikation mit dem Hoare-Kalkül

Verifikation auf Basis von Zusicherungen (Fortsetzung)

- Wie verändert eine Anweisung den Zustand und damit die Zusicherung?

  ➜ Für jeden Typ von Anweisungen gibt es eine eigene Verifikationsregel.

- Welcher Nachfolgezustand ergibt sich aus einem Zustand?
  Wie sah der Vorgängerzustand eines Zustands aus?

  ➜ Jede Anweisung wird einzeln verifiziert.

- Der Nachfolgezustand der einen Anweisung ist der Vorgängerzustand der nächsten Anweisung und damit die Nachbedingung der einen die Vorbedingung der nächsten Anweisung.

  ➜ Anweisungsblöcke werden durch zusammenpassende Einzelschritte verifiziert.

➜ Verifikation eines Programmes ist Beweis der Korrektheit des Programmes unter Verwendung von akzeptierten Verifikationsregeln für die verwendete Programmiersprache.

Verifikation mit dem Hoare-Kalkül

Definition 16 (Hoare-Formel)

Eine Hoare-Formel

$$\{V\}S\{N\}$$

besteht aus Zusicherungen V und N und einer Anweisungsfolge S. V heißt auch Vorbedingung, N Nachbedingung der Anweisungsfolge S.

Semantik einer Hoare-Formel:

> Für jeden (Ausgangs-)Zustand, für den vor Ausführung der Anweisung S die Zusicherung V gilt, gilt nach der Ausführung von S für den Folgezustand die Zusicherung N.

Beachte:
Die Anweisungsfolge S kann ein komplexes Programmstück, aber auch nur eine Anweisung sein.

➜ Woher kommen die benötigten Hoare-Formeln?

Verifikation mit dem Hoare-Kalkül

Verifikation auf Basis axiomatischer Semantik

Axiomatische Semantik:

> Die Semantik wird durch ein Axiom $\{V\}S\{N\}$ für jede ausführbare Anweisung S der verwendeten Programmiersprache definiert.
>
> [Hoare, 1969]

Verifikation mit dem Hoare-Kalkül

Verifikation auf Basis axiomatischer Semantik

Axiomatische Semantik:

> Die Semantik wird durch ein Axiom $\{V\}S\{N\}$ für jede ausführbare Anweisung S der verwendeten Programmiersprache definiert.
>
> [Hoare, 1969]

(Totale) Korrektheit

Der Hoare-Kalkül soll einen Nachweis liefern, dass für ein Programm P eine Spezifikation $\{V\}P\{N\}$ korrekt ist.

Erforderliche Teilbeweise:

- **Partielle Korrektheit**
  Wenn das Programm P terminiert, so transformiert P jeden Zustand, in dem V gültig ist, in einen Zustand, in dem N gültig ist.
- **Terminierung**
  Das Programm P terminiert für jeden Anfangszustand, in dem V gültig ist.

Verifikation mit dem Hoare-Kalkül

Beispiel für ein Programm

Programm P:

Vorbedingung $\{x, y \in \mathbf{N}\}$
Nachbedingung $\{b = x + y$ *und* $x, y \in \mathbf{N}\}$

```
begin
   a := x;
   b := y;
   while (a > 0) do
      begin
         a := a − 1;
         b := b + 1;
      end
end
```

➜ Wie können die nötigen Nachweise für die Korrektheit geführt werden?

Verifikation mit dem Hoare-Kalkül

Definition 17 (Hoare-Regel)

Eine Hoare-Regel ist ein Schema folgender Art

$$\frac{\begin{array}{c}\text{Voraussetzung 1}\\ \vdots \\ \text{Voraussetzung n}\end{array}}{\text{Schlussfolgerung}}$$

„Voraussetzung i“ ist entweder eine Hoare-Formel oder eine Formel der Art $\{V_1\} \Rightarrow \{V_2\}$, wobei V_1 und V_2 Zusicherungen sind und V_2 eine Folgerung von V_1 ist. „Schlussfolgerung“ ist wieder eine Hoare-Formel.

Hoare-Regeln können genutzt werden, um aus gegebenen Hoare-Formeln weitere Hoare-Formeln herzuleiten.

- ➜ Der Nachweis der partiellen Korrektheit eines Programmes P besteht aus der Angabe einer Herleitung der Hoare-Formel $\{V\}P\{N\}$ mit Hilfe von Hoare-Regeln, wobei V und N die Vor- und Nachbedingung aus der Spezifikation von P sind.

Verifikation mit dem Hoare-Kalkül

Definition 18 (Hoare-Kalkül)

Ein Hoare-Kalkül für eine (einfache) imperative Programmiersprache ist ein System von Regeln passend zu den Anweisungen der Programmiersprache, die für Programme P die Ableitung von Hoare-Formeln $\{V\}P\{N\}$ erlaubt.

Der Hoare-Kalkül kann genutzt werden, um bei gegebener Vorbedingung gültige Nachbedingungen zu ermitteln oder aber um die Vorbedingungen zu ermitteln, die für eine gewünschte Nachbedingung erforderlich ist.

Definition 19 (Herleitung im Hoare-Kalkül)

Eine Herleitung einer Hoare-Formel $\{V\}S\{N\}$ in einem Hoare-Kalkül ist eine Folge von Hoare-Formeln $\{V_1\}S_1\{N_1\}, \ldots, \{V_k\}S_k\{N_k\}$, deren letzte die herzuleitende Hoare-Formel ist, $\{V\}S\{N\} = \{V_k\}S_k\{N_k\}$ und für die jede Hoare-Formel $\{V_i\}S_i\{N_i\}$ mit Hilfe einer Regel des Kalküls unter Verwendung von nur den Hoare-Formeln $\{V_1\}S_1\{N_1\}, \ldots, \{V_{i-1}\}S_{i-1}\{N_{i-1}\}$ erzeugt wurde.

Verifikation mit dem Hoare-Kalkül

Kalkül für die einfache imperative Programmiersprache

Benötigte Hoare-Regeln:

- Regel für Zuweisungen
- Regel für bedingte Anweisungen
- Regel für Schleifen
- Regeln für Anweisungsfolgen und Anweisungsblöcke
- Abschwächungsregeln (unabhängig von konkreter Sprache)

Erzeugung von Anfangsformeln:

- Manche Regeln müssen ohne Voraussetzungen auskommen.

➜ Herleitungen sind baumartige Strukturen.
Wie lassen sich Herleitungen und Programme gemeinsam darstellen?

Verifikation mit dem Hoare-Kalkül

Verifikation

- Zusicherungen betreffen Zustände vor und nach Anweisungen eines Programmes.

  ➜ Zusicherungen werden unmittelbar in das Programm integriert.

- Hoare-Regeln spiegeln die Struktur von Anweisungen wider.

  ➜ Ein Programm kann nicht Zeile für Zeile, sondern nur der Struktur entsprechend bearbeitet werden.

- Abgeleitete Hoare-Formeln lassen den Nachweis der partiellen Korrektheit oder der Terminierung zu, im allgemeinen jedoch nicht beides gleichzeitig.

  ➜ Partielle Korrektheit und Terminierung von Schleifen werden mit zwei separaten Herleitungen mit dem Hoare-Kalkül gezeigt.

- Terminierung ist Voraussetzung in jeder Hoare-Formel.

Verifikation mit dem Hoare-Kalkül

Beipiel einer Verifikation: Addition zweier natürlicher Zahlen

Spezifikation

Vorbedingung $\{x, y \in \mathbf{N}\}$
Nachbedingung $\{b = x + y$ *und* $x, y \in \mathbf{N}\}$

Programm

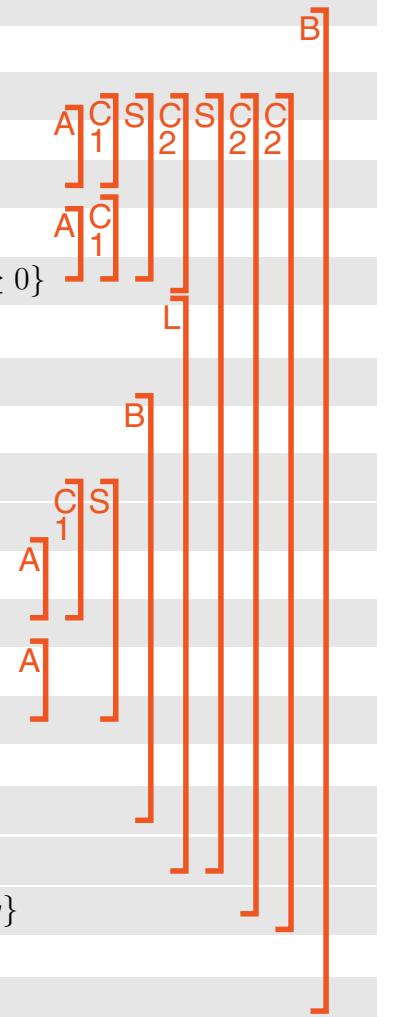
```
begin
   a := x;
   b := y;
   while (a > 0) do
      begin
         a := a − 1;
         b := b + 1;
      end
end
```

Verifikation mit dem Hoare-Kalkül

Beipiel einer Verifikation: Addition zweier natürlicher Zahlen (Fortsetzung)

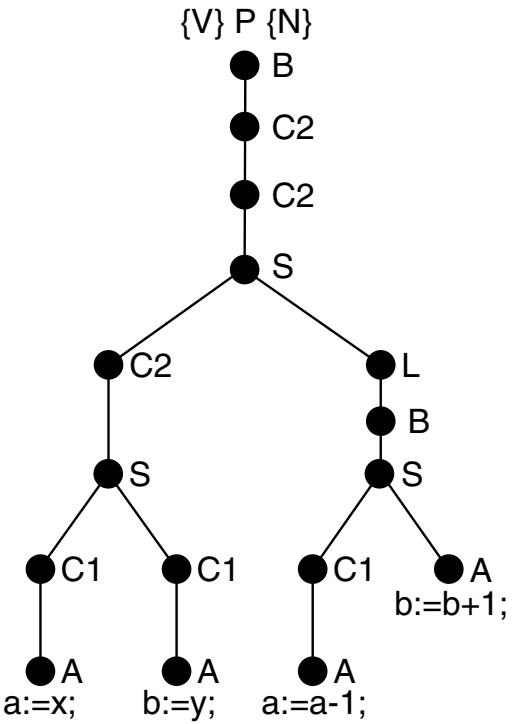
Partielle Korrektheit:

$\{x, y \in \mathbf{N}\}$
`begin`
  $\{x, y \in \mathbf{N}\} \Rightarrow \{x, y \in \mathbf{N}$ *und* $a = x\}$
  $a := x;$
  $\{x, y \in \mathbf{N}$ *und* $a = x\} \Rightarrow \{x, y \in \mathbf{N}$ *und* $a = x$ *und* $y = y\}$
  $b := y;$
  $\{x, y \in \mathbf{N}$ *und* $a = x$ *und* $b = y\} \Rightarrow \{x, y \in \mathbf{N}$ *und* $a + b = x + y$ *und* $a \geq 0\}$
  `while` $(a > 0)$ `do`
    $\{x, y \in \mathbf{N}$ *und* $a + b = x + y$ *und* $a \geq 0$ *und* $a > 0\}$
    `begin`
      $\{x, y \in \mathbf{N}$ *und* $a + b = x + y$ *und* $a \geq 0$ *und* $a > 0\}$
      $\Rightarrow \{x, y \in \mathbf{N}$ *und* $a - 1 + b + 1 = x + y$ *und* $a - 1 \geq 0\}$
      $a := a - 1;$
      $\{x, y \in \mathbf{N}$ *und* $a + b + 1 = x + y$ *und* $a \geq 0\}$
      $b := b + 1;$
      $\{x, y \in \mathbf{N}$ *und* $a + b = x + y$ *und* $a \geq 0\}$
    `end`
    $\{x, y \in \mathbf{N}$ *und* $a + b = x + y$ *und* $a \geq 0\}$
  $\{x, y \in \mathbf{N}$ *und* $a + b = x + y$ *und* $a \geq 0$ *und* $a \leq 0\}$
  $\Rightarrow \{x, y \in \mathbf{N}$ *und* $a + b = x + y$ *und* $a = 0\} \Rightarrow \{x, y \in \mathbf{N}$ *und* $b = x + y\}$
`end`
$\{x, y \in \mathbf{N}$ *und* $b = x + y\}$



Verifikation mit dem Hoare-Kalkül

Herleitung im Hoare-Kalkül als Baum



- Der Herleitung, die hinter der Verifikation steht, bildet eine Baumstruktur.
- Eine Verifikation eines Programmes kann zu großen Teilen sequentiell am Programm orientiert vorgenommen werden:
  - vorwärts: Hoare-Kalkül
  - rückwärts: Weakest Preconditions nach Dijkstra