Software-Qualitätsmanagement

Kernfach Angewandte Informatik

Sommersemester 2004

Prof. Dr. Hans-Gert Gräbe

Verifikationsregeln

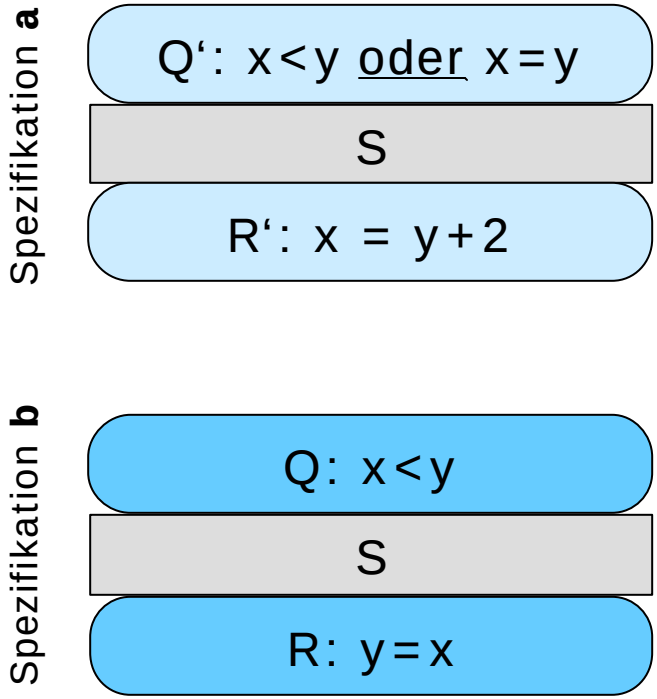
- **Voraussetzung**: Programm ist aus konditionierten Bausteinen modular zusammengesetzt
  - Korrektheit des gesamten Programms ergibt sich aus der korrekten Zusammensetzung korrekter Teilstrukturen
- **Vorgehen**: Komplexes Programm wird verifiziert durch schrittweises Zusammensetzen aus **verifizierten einfacheren Strukturen** nach wenigen einfachen **Verifikationsregeln**.
- Folgende Verifikationsregeln existieren:
  - Konsequenz-Regel,
  - Zuweisungs-Regel,
  - Sequenz-Regel,
  - **if**-Regel und
  - **while**-Regel

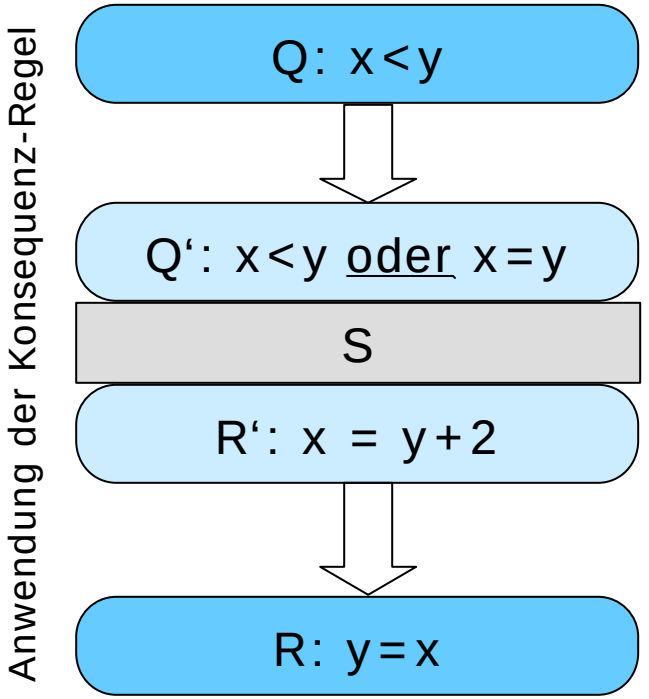
Konsequenz-Regel

Gilt **{Q‘} S {R‘}** und

- Q‘ wird durch Q ersetzt, wobei Q schärfer ist als Q‘.
- R‘ wird durch R ersetzt, wobei R schwächer ist als R‘.

so gilt auch **{Q} S {R}.**





3. Programmverifikation

- Geht man vorwärts durch ein Programm, so kann man Bedingungen abschwächen:
  - Hinzufügen eines Terms mit **oder**-Verknüpfung
  - Weglassen eines **und**-verknüpften Terms
  - ➔ Schwächere Bedingung
- Bei rückwärtiger Abarbeitung eines Programms, dürfen Bedingungen verschärft werden:
  - Hinzufügen eines Terms mit **und**-Verknüpfung
  - Weglassen eines **oder**-verknüpften Terms
  - ➔ Schärfere Bedingung

- Notation von Verifikationsregeln als Schlussregel:

$$\frac{\text{Voraussetzungen}}{\text{Schlussfolgerung}}$$

$$\frac{Q \Rightarrow Q',\ \{Q'\}\ S\ \{R'\},\ R' \Rightarrow R}{\{Q\}\ S\ \{R\}}$$

Zuweisungs-Regel

- Die Zuweisung x:=A verändert den Wert von x
  - Beispiel: { y+z = 25 } x:=y+z { x = 25 }
- Allgemeine Struktur: **{ R(A) } x:=A { R(x) }**
  - so zu verstehen: Hat man einen logischen Ausdruck R=R(x) mit der freien Variablen x und bildet Q::=R(A) durch Ersetzen dieser Variablen mit dem Ausdruck A, so ist die Aussage

    { Q } x:=A { R }

    wahr.
- Regel wird eingesetzt beim Rückwärtsarbeiten, um aus einer Nach- eine Vorbedingung abzuleiten:

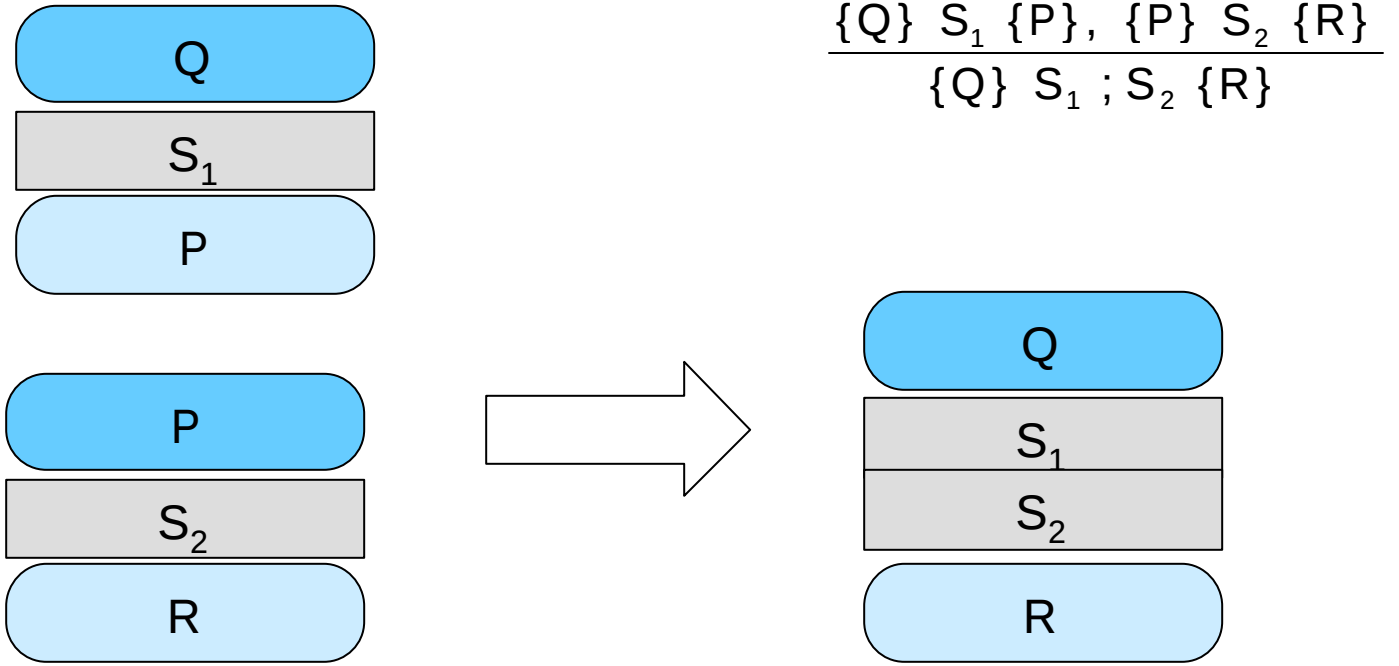
  Bsp.: { Q? } x := x+25 { x = 2y }

  { R(A) } x‘ = x+25 { R(x‘) : x‘ = 2y}

  ➔ Vorbedingung { Q : 2y = x + 25 }

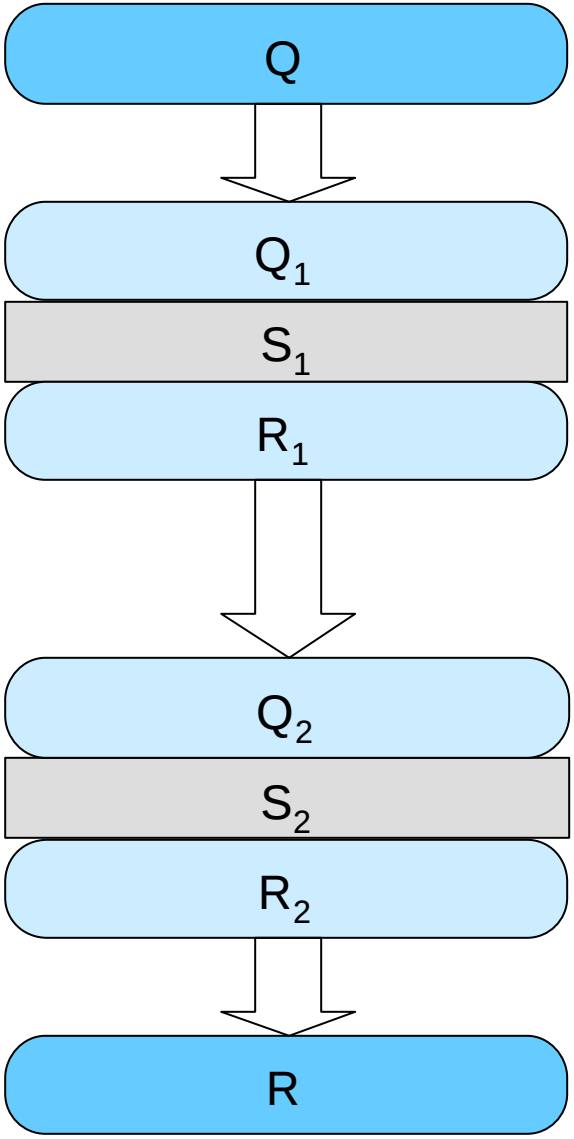
Sequenz-Regel

- Zwei Programmteile S_1 und S_2 können zusammengesetzt werden, wenn die Nachbedingung von S_1 gleich der Vorbedingung von S_2 ist.



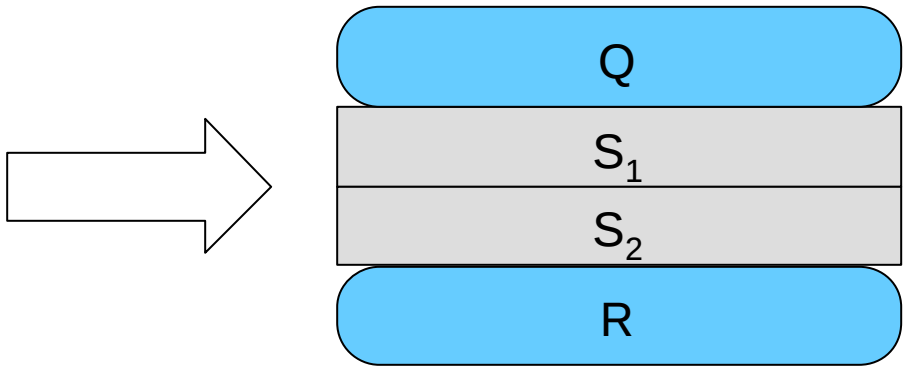
$$\frac{\{Q\}\ S_1\ \{P\},\ \{P\}\ S_2\ \{R\}}{\{Q\}\ S_1\ ;S_2\ \{R\}}$$

3. Programmverifikation



Die Sequenz-Regel kann mit Hilfe der Konsequenz-Regel noch verallgemeinert werden:
Es genügt, wenn die Nachbedingung von S_1 „schärfer“ ist als die Vorbedingung von S_2, um S_1 und S_2 zu einem Programmstück zusammenzusetzen.

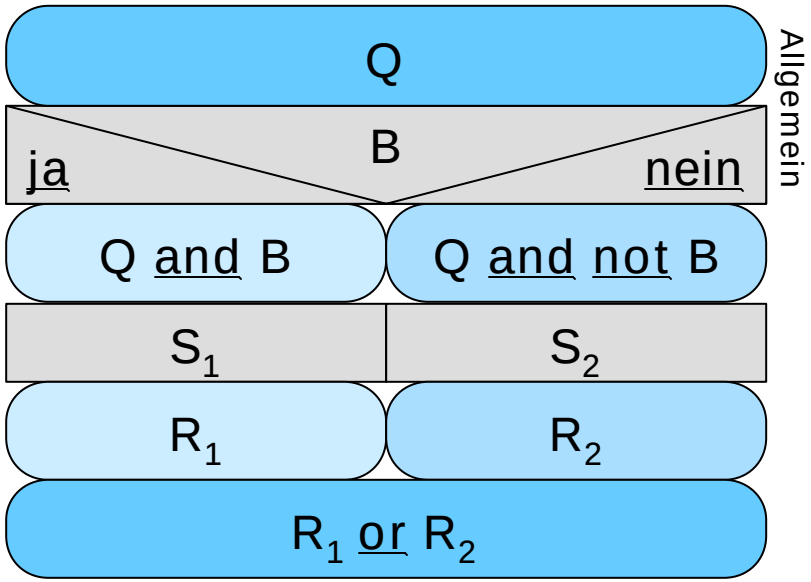
$$\frac{Q \Rightarrow Q_1, \{Q_1\}\, S_1\, \{R_1\}, R_1 \Rightarrow Q_2, \{Q_2\}\, S_2\, \{R_2\}, R_2 \Rightarrow R}{\{Q\}\, S_1\, ; S_2\, \{R\}}$$

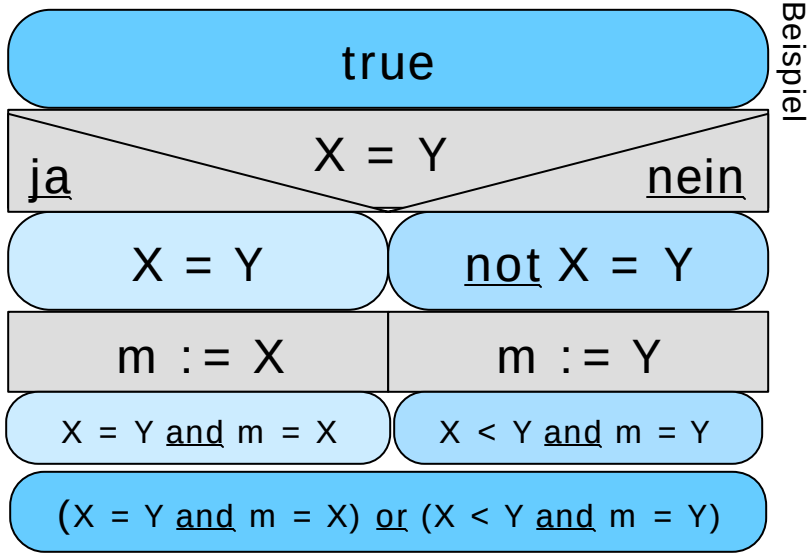


if-Regel

- Gibt an, unter welchen Voraussetzungen zwei Programmstücke S_1 und S_2 und eine Bedingung B zu einer zweiseitigen Auswahl mit der Vorbedingung Q und der Nachbedingung R zusammengesetzt werden können.

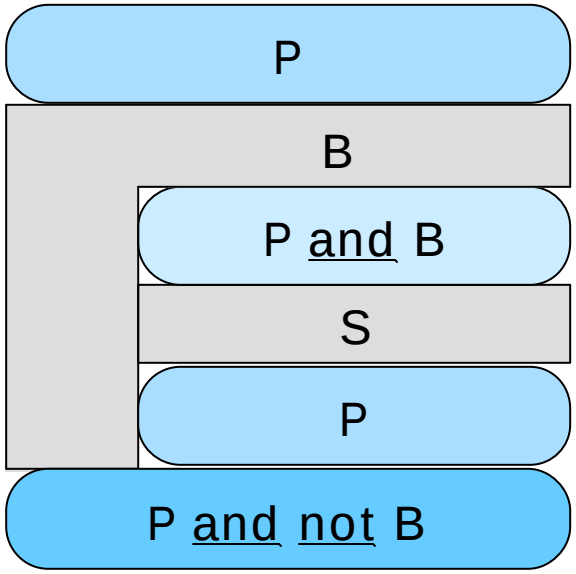
$$\frac{\{Q \text{ and } B\}\ S_1\ \{R\},\ \{Q \text{ and not } B\}\ S_2\ \{R\}}{\{Q\}\ \text{if } B \text{ then } S_1 \text{ else } S_2\ \{R\}}$$





while-Regel

- Bei der Verifikation von Schleifen spielt eine invariante Zusicherung **P**, die **Schleifeninvariante** eine entscheidende Rolle.
- Die Invariante gilt vor der Schleife und nach dem Schleifenrumpf.



$$\frac{\{P \text{ and } B\}\ S\ \{P\}}{\{P\}\ \text{while } B \text{ do } S\ \{P \text{ and not } B\}}$$

Diese Regel beweist nur **partiell** die Korrektheit der Schleife, denn die **Termination** wird durch P nicht garantiert.

Zum Beweis der Termination einer Schleife

- Wiederholungsbedingung B muss irgendwann falsch sein.
- Prüfung der Termination mit Hilfe einer **Terminationsfunktion t**.
  - **Idee**: Die Terminationsfunktion

    t : Programmzustände -> **Z**

    ist nach unten beschränkt **und** wird in jedem Schleifendurchlauf kleiner.

Formale Formulierung der Bedingungen für t:

1. { P and B and t = T } S { P and t<T } (T ist freie Variable)
2. P and B ➔ t = 0

- **Variation**: Kettenbedingung auf Halbordnungen
  - Beispiel: Termordnungen auf dem Term-Monoid $T=T(x_1,...x_n)$
  - es reicht die Kettenbedingung statt Beschränktheit

Konditionierungsregel für Schleifen

Bei gegebener Invariante **P** und Terminationsfunktion **t** muss eine **while**-Schleife folgende Punkte erfüllen:

1. Die Invariante P muss während der Initialisierung der Schleife gesichert werden:
   - {Q} init {P}
2. P bleibt im Schleifenrumpf S invariant, t wird bei jedem Ausführen des Schleifenrumpfes verringert.
   {P and B and t = T} S {P and t < T}
3. t ist vor jedem Ausführen des Schleifenrumpfes nicht negativ.
   P and B ➔ t = 0
4. Die Nachbedingung R ist eine Folge der Schleifeninvariante.
   P and not B ➔ R

Vorgehensweise, falls P und t bekannt

1. {Q} init {P} für die Schleifeninitialisierung verifizieren.
2. Suchen einer geeigneten Wiederholungsbedingung B, so dass nach Ende der Schleife die Nachbedingung R gilt

   P and not B ➔ R

3. Suche einer Terminationsfunktion t, für welche

   P and B ➔ t = 0

   und

   {P and B and t = T} S {P and t < T}

   gilt.

Entwickeln einer Schleife durch Weglassen einer Bedingung

Gegeben sei eine Spezifikation {Q} . {R: U and V}

1. R wird aufgeteilt in {R = P and not B}: P = U, B = not V
   - Die Invariante P ergibt sich durch Weglassen einer Bedingung.
   - Die weggelassene Bedingung {not V} wird zur Abbruchbedingung.
2. Initialisierung der Invarianten P durch ein Programmstück

   {Q} init {P}
3. Entwicklung eines Schleifenrumpfes mit der Spezifikation

   {P and B} S {P}
4. Hinzufügen der Terminationsbedingung **t**
   - **t** ergibt sich häufig aus dem Vergleich der Initialisierung mit der Abbruchbedingung {not B}.

3. Programmverifikation

Beispiel: *int y = isqrt(int x) mit y= ësqrt(x)û*

$\{Q: x \geq 0\}$ und $\{R: y \geq 0 \text{ and } y^2 \leq x \text{ and } x < (y+1)^2\}$

Aufteilen von R: $\{P: y \geq 0 \text{ and } x \geq y^2\}$ und $\{B: x \geq (y+1)^2\}$

Initialisierung: $\{Q: x \geq 0\}\ y:=0\ \{P\}$

Schleife: $\{P \text{ and } B\}\ y:=y+1\ \{P\}$

$\{y \geq 0 \text{ and } x \geq (y+1)^2\}\ y'=y+1\ \{y' \geq 0 \text{ and } x \geq y'^2\}$

Terminationsfunktion: **t** := x – y

$\{P \text{ and } B \text{ and } t=T\}\ y:=y+1\ \{P \text{ and } t<T\}$

3. Programmverifikation

Java-Implementierung:

```
int isqrt(int x) {
    int y=0;
    while ((y+1)*(y+1) <= x)
        y=y+1;
    return y;
}
```

oder nach leichter Optimierung

```
int isqrt(int x) {
    int y=1;
    while (y*y <= x)
        y=y+1;
    return (y-1);
}
```

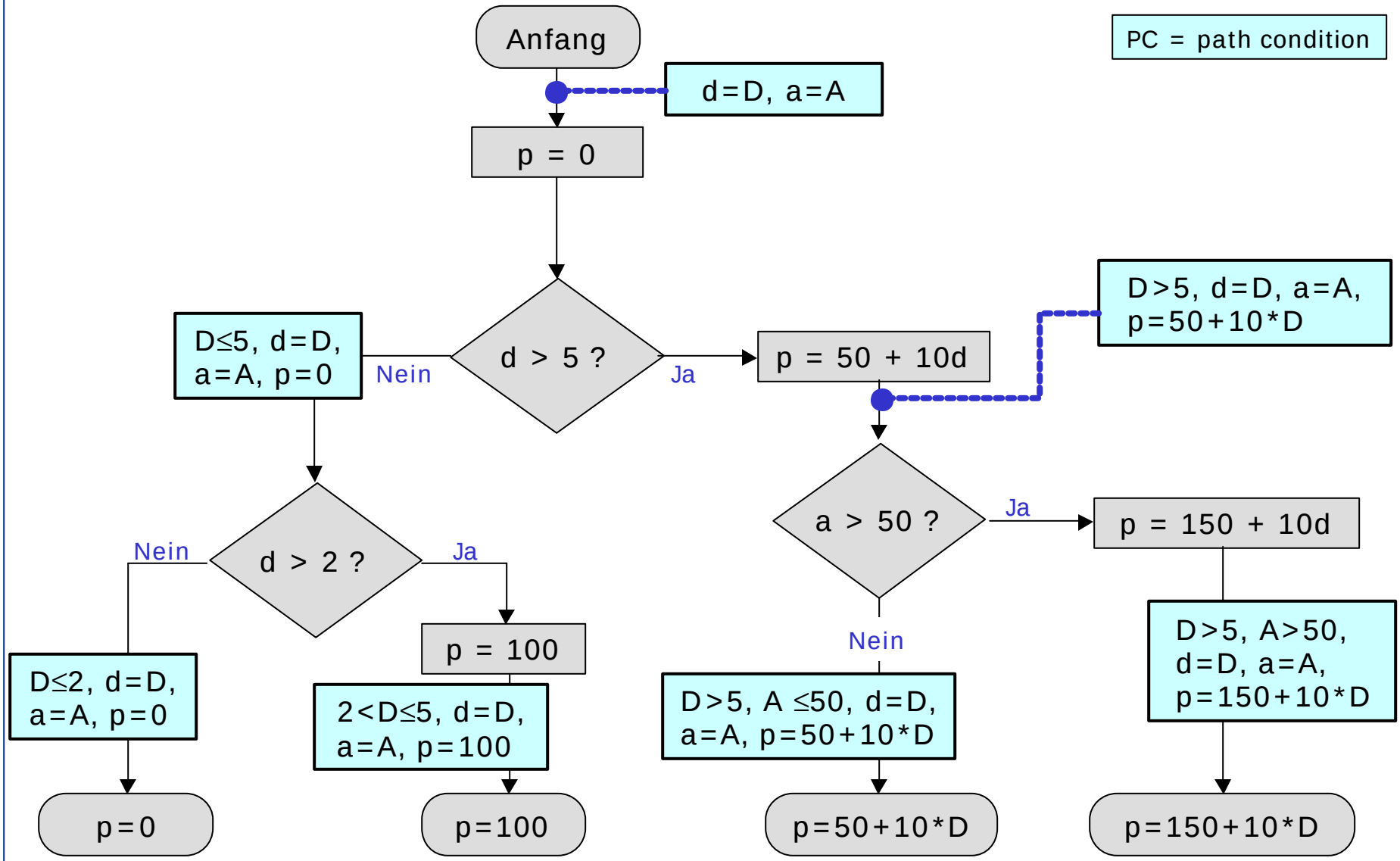
Symbolisches Testen: Überblick

- **Idee**: Die Eingabeparameter des Programms werden mit symbolischen Variablen belegt und längs aller möglicher Kontrollflüsse alle möglichen **Zwischenergebnisse** und **Konditionen** in symbolischer Form bestimmt.
  - Methode ist besonders gut geeignet, wenn sich die an Verzweigungspunkten gültigen Kombinationen boolescher Bedingungen vereinfachen lassen und Zwischenergebnisse arithmetischer Natur sind.
- **Methode hat Beweiskraft** im mathematischen Sinn, wenn die symbolischen Parameter durch alle denkbaren konkreten Parameterwerte ersetzt werden können.
  - etwa darf das Zwischenergebnis y/x nicht ohne die Kondition x!=0 auftreten.
- **Schleifen** lassen sich in diesem Ansatz nur bedingt abbilden.

Beispiel

```
int berechnePraemie(int Dienstjahre, int Alter) {
    Praemie = 0;
    if (Dienstjahre > 5) {
        Praemie = 50 + 10 * Dienstjahre;
        if (Alter > 50) Praemie = Praemie + 100;
    }
    else if (Dienstjahre > 2) Praemie = 100;
    return Praemie;
}
```

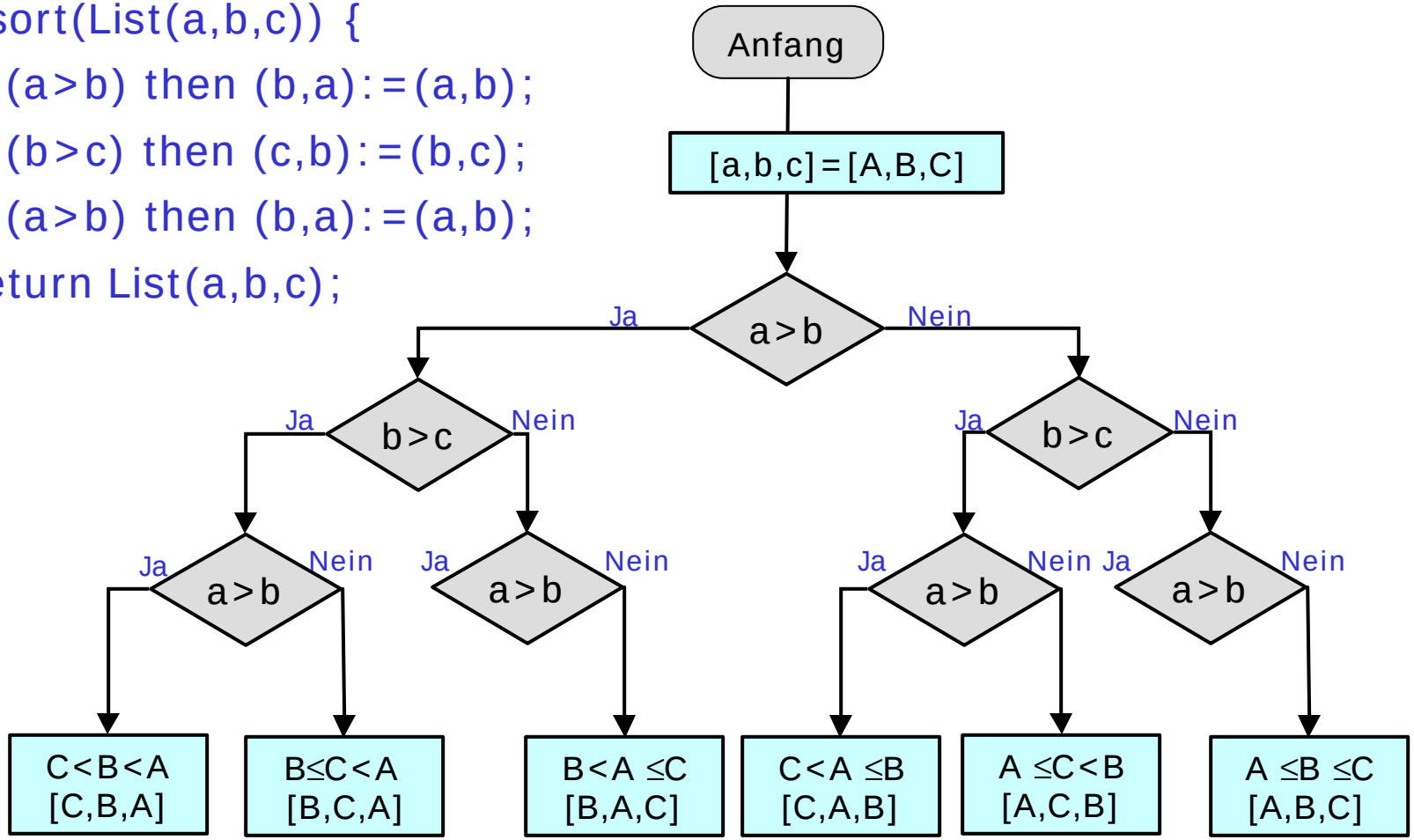
4. Symbolisches Testen



4. Symbolisches Testen

Beispiel

```
List sort(List(a,b,c)) {
    if (a>b) then (b,a):=(a,b);
    if (b>c) then (c,b):=(b,c);
    if (a>b) then (b,a):=(a,b);
    return List(a,b,c);
}
```



1. Einführung

Ansatz: Qualität von Systemkomponenten besteht nicht nur in deren **funktioneller Qualität** (Q.-Z. Funktionalität und Effizienz; Fokus der bisher besprochenen Qualitätssicherungs-Methoden Test und Verifikation), sondern auch in der **Qualität des Quellcodes** selbst (Q.-Z. Änderbarkeit, Übertragbarkeit sowie teilweise Benutzbarkeit).

Relevante Parameter:

- sinnvolle **Granularität** der Komponenten längs **funktioneller Grenzen**.
- sinnvolle **Schnittstellengestaltung** für die Zusammenarbeit der Komponenten untereinander.

Kann in quantitativen Parametern der **Bindung** (innerhalb einer Komponente) und **Kopplung** (zwischen Komponenten) erfasst werden.

Bindung und Kopplung

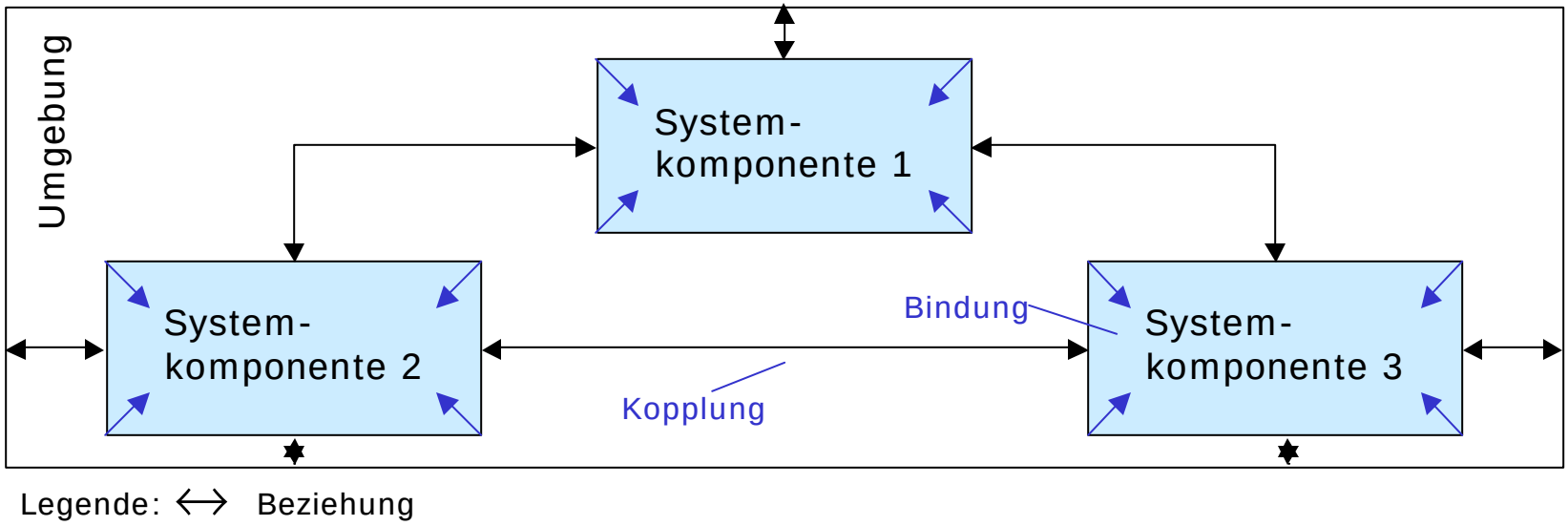
Die Bindung einer Systemkomponente und die Kopplung der Systemkomponenten untereinander bestimmen die Struktur eines Software-Systems.

Bindung (*cohesion*) ist ein qualitatives Maß für die Kompaktheit einer Systemkomponente. Es werden dazu die Beziehungen zwischen den Elementen einer Systemkomponente betrachtet.

Kopplung (*coupling*) ist ein qualitatives Maß für die Schnittstellen zwischen den Systemkomponenten. Es werden der Kopplungs-mechanismus, die Schnittstellenbreite und die Art der Kommunikation betrachtet.

2. Bindung und Kopplung

- Je stärker die Bindungen der Systemkomponenten im Vergleich zu den Kopplungen, desto ausgeprägter ist die Struktur und Modularität eines Systems.
- Die Forderung nach Einfachheit wird erfüllt, wenn die Kopplungen minimiert und die Bindungen maximiert werden.
- Für **Komponenten** spielt deren Bindung eine qualitätsrelevante Rolle, für **Systeme** die Kopplung zwischen den Komponenten.



2. Bindung und Kopplung

Bindung von Prozeduren und Funktionen

- Gute Bindung liegt vor, wenn nur solche Elemente zu einer Einheit zusammengefasst werden, die auch zusammen gehören.
- Bindung von Funktionen wird nur qualitativ erfasst.

[Stevens 81] unterscheidet folgende Bindungsarten :

1. zufällige Bindung
2. logische Bindung
3. zeitliche Bindung
4. prozedurale Bindung
5. kommunikative Bindung
6. sequentielle Bindung
7. funktionale Bindung

schwache Bindung

starke Bindung

2. Bindung und Kopplung

Ziel: Erreichen einer funktionalen Bindung.

- Alle Elemente sind an der Verwirklichung einer einzigen, abgeschlossenen Funktion beteiligt.
- Komplexe Funktionen werden realisiert, indem importierte Funktionen verwendet werden, die selbst funktional gebunden sind.

- **Kennzeichen** einer funktionalen Bindung:
  - Alle Elemente tragen dazu bei, ein einzelnes spezifisches Ziel zu erreichen.
  - Es gibt keine überflüssigen Elemente.
  - Die Aufgabe kann mit genau einem Verb und genau einem Objekt beschrieben werden.
  - Austausch gegen anderes Element, welches denselben Zweck erfüllt, leicht möglich.
  - Hohe Kontextunabhängigkeit, d.h. einfache Beziehungen zur Umwelt.

2. Bindung und Kopplung

Vorteile einer funktionalen Bindung:

- Hohe Kontextunabhängigkeit (die Bindungen befinden sich innerhalb der Prozedur, nicht zwischen Prozeduren).
- Geringe Fehleranfälligkeit bei Änderungen,
- Hoher Grad der Wiederverwendbarkeit,
- Leichte Erweiterbarkeit und Wartbarkeit, da sich Änderungen auf isolierte, kleine Teile beschränken.

- Konzept der Bindung verallgemeinert die (konzeptuellen) Regeln für „guten Code“ zu Regeln für „guten Software-Entwurf“.
- Die Bindungsart einer Prozedur lässt sich nicht automatisch ermitteln, sondern nur durch manuelle Prüfmethoden.
- Diese Untersuchungen sind noch im experimentellen Stadium und haben heute weitgehend informellen (damit aber nicht weniger bindenden) Charakter.

Bindung von Datenabstraktionen / Klassen

Beschreibt das Zusammenwirken verschiedener Funktionen, welche derselben Datenabstraktion oder Klasse zuzuordnen sind.

- Voraussetzung: Alle Methoden sind funktional gebunden

Gute Bindung einer Klasse (*model cohesion*) liegt vor, wenn

- sie ein einzelnes semantisch bedeutungsvolles Konzept repräsentiert,
- die Klasse keine verborgenen Klassen enthält und
- keine Operationen enthält, die an andere Klassen delegiert werden können.

Wird in der Literatur auch als Kohärenz bezeichnet.

Für Klassen ist weiter die Bindung innerhalb von Vererbungs-strukturen wesentlich.

Informale Bindung

[Myers 78] fordert für abstrakte Datenobjekte informale Bindung.

- liegt vor, wenn mehrere, in sich abgeschlossene, funktional gebundene Zugriffsoperatoren, die zu einer Datenabstraktion gehören, auf einer einzigen Datenstruktur operieren.
- **Idee**: hinter der gemeinsamen Funktionalität liegt auch ein gemeinsames Datenmodell

Merkmale:

- Unterstützt das Geheimnisprinzip, d.h. die Datenstruktur gehört nur zu einer Datenabstraktion,
- Änderungen der Datenstruktur tangieren nur eine Datenabstraktion,
- Problem der Vermischung von Zugriffsoperationen, da alle auf derselben Datenstruktur operieren.

Bindung in Vererbungsstrukturen

- Die ganze Vererbungshierarchie muss untersucht werden.
- **Starke Vererbungsbindung** liegt vor, wenn die Hierarchie eine Generalisierungs-/Spezialisierungshierarchie im Sinne der konzeptuellen Modellierung ist.
- **Schwache Vererbungsbindung** liegt vor, wenn die Hierarchie nur zum "*code sharing*" verwendet wird.
- Das **Ziel** jeder neu definierten Unterklasse muss sein, ein einzelnes semantisches Konzept auszudrücken.