Praxis des Programmierens

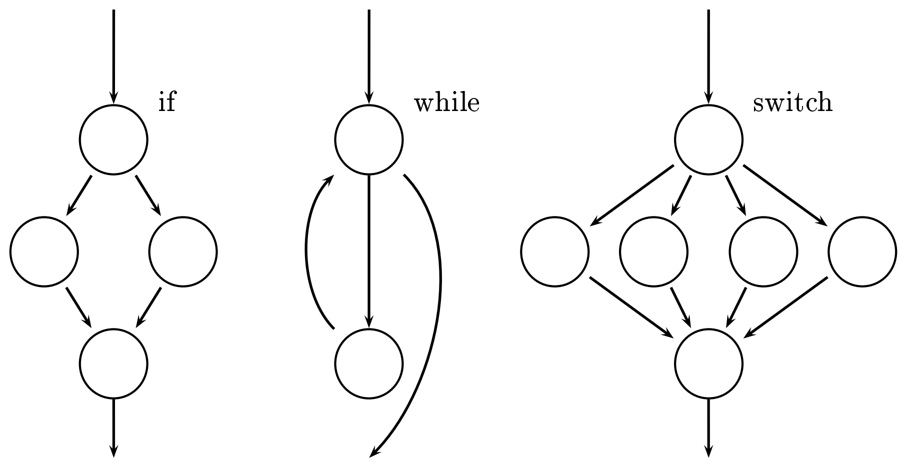
Vorlesung 13.07.1999

Uwe Waldmann

7 Verifikation und Validierung (Fortsetzung)

7.$(n-1)$ Dynamische Verifikation – Beispiel: Pfad-Testen

Im Gegensatz zum funktionalen (oder Black-Box-) Testen wird beim strukturellen (oder White-Box-) Testen die Auswahl der Testdaten durch Kenntnisse über die Implementierung beeinflußt. Ein Beispiel für eine White-Box-Teststrategie ist Pfad-Testen. Hier ist das Ziel, sicherzustellen, daß jede Anweisung innerhalb der Komponente beim Test mindestens einmal ausgeführt wird. Das heißt insbesondere, daß jede bedingte Anweisung sowohl mit wahrer als auch mit falscher Bedingung durchlaufen werden muß. Dazu ermittelt man zunächst einen Programmflußgraphen für die Komponente, indem man die Kontrollstrukturen durch Teilgraphen ersetzt:

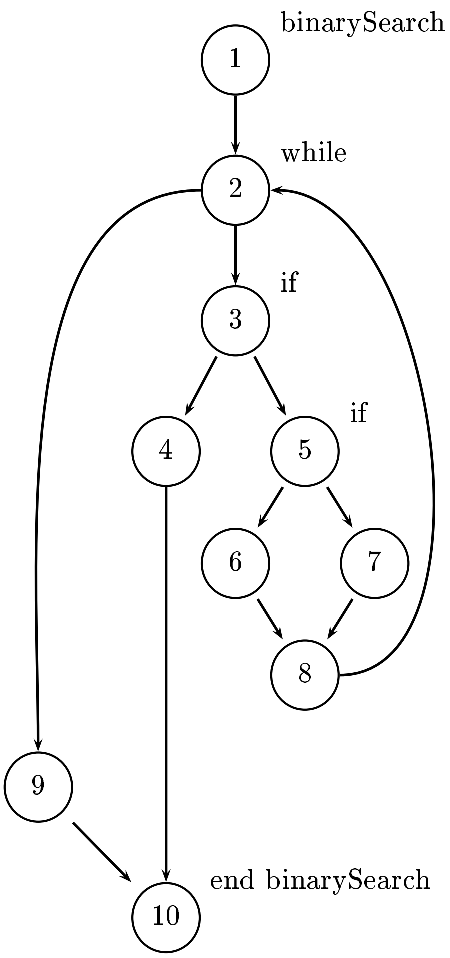


Dann sucht man eine Menge von Pfaden durch diesen Graphen, die jede Kante mindestens einmal abdeckt.

Beispiel: Gegeben sei ein Integer-Array $a[]$, dessen Elemente aufsteigend sortiert sind, also $j < k \Rightarrow a[j] < a[k]$. Die Methode *binarySearch* liefert in logarithmischer Zeit zu jedem *key* denjenigen Index i, für den $a[i] = key$ gilt, oder -1, falls kein solches i existiert:

```
static int binarySearch(int key, int[] a) {
  int bot = 0;                  // falls ein i existiert
  int top = a.length - 1;       //   mit a[i] == key, dann
                                //   gilt bot <= i <= top.
  int mid;                      // Mitte des Intervalls.
  while (bot <= top) {
    mid = (top + bot) / 2;      // Division mit Abrundung.
    if (a[mid] == key) {
      return mid;               // a[mid] == key, also
                                //   gesuchter Index == mid.
    } else if (a[mid] < key) {
      bot = mid + 1;            // a[mid] < key, also
                                //   gesuchter Index > mid.
    } else {
      top = mid - 1;            // a[mid] > key, also
                                //   gesuchter Index < mid.
    }
  }
  return -1;                    // Schleife verlassen,
                                //   da bot > top,
                                //   also existiert kein i
                                //   mit a[i] == key.
}
```

Für *binarySearch* erhalten wir den folgenden Programmflußgraphen:



Eine Menge von Pfaden durch diesen Graphen, die jede Kante mindestens einmal abdeckt, ist zum Beispiel

1–2–9–10

1–2–3–5–6–8–2–3–4–10

1–2–3–5–7–8–2–3–4–10

Dies ist nicht die kleinste mögliche Menge. Eine andere, kleinere Menge ist

1–2–3–5–6–8–2–3–4–10

1–2–3–5–7–8–2–9–10

Wenn wir für jeden dieser Pfade eine Testeingabe haben, bei der genau dieser Pfad abgearbeitet wird, dann wird jede Anweisung innerhalb von *binarySearch* beim Test mindestens einmal ausgeführt. (Aufgabe: Gib für jeden dieser Pfade eine Testeingabe an.)

7.n Statische Verifikation – Beispiel: Hoare-Kalkül

Das Ziel formaler Verifikationsmethoden ist es, einen formalen (mathematischen) Beweis zu liefern, daß ein Programm eine Spezifikation erfüllt, oder daß zwei Programme äquivalent sind. Eine der bekanntesten Methoden zur formalen Verifikation wurde von Hoare entwickelt. Der Hoare-Kalkül beschäftigt sich mit Aussagen der Form

$$P \ \{ A \} \ Q$$

wobei A eine Anweisung oder Anweisungsfolge[1] ist und P und Q logische Formeln (Zusicherungen, Assertions) sind, die Belegungen von Programmvariablen während des Programmlaufs beschreiben. Solch eine Aussage ist zu lesen als „wenn vor Ausführung von A die Zusicherung P gilt und A terminiert, dann gilt danach Q“.[2] Man nennt P Vorbedingung und Q Nachbedingung der Aussage und bezeichnet $P \ \{ A \} \ Q$ auch als *Hoare-Tripel.*

Offensichtlich können Hoare-Tripel mehr oder weniger interessant sein. Beispielsweise ist die Aussage $3 \neq 5 \ \{ \mathit{Prog} \} \ 3 \neq 5$ richtig, aber völlig uninteressant. Ziel ist es, zu Aussagen $P \ \{ \mathit{Prog} \} \ Q$ zu kommen, bei denen Q eine zu beweisende Eigenschaft des Programms *Prog* ist und P die Voraussetzungen beschreibt, unter denen das Programm aufgerufen werden darf. Bei dem oben beschriebenen Programm *binarySearch* möchte man zum Beispiel zeigen, daß, wenn das Array $a[]$ sortiert ist, der gesuchte Index oder -1 zurückgegeben wird. (Die Vorbedingung kann auch true sein, d. h., Q gilt nach Ablauf von *Prog* unbedingt.)

Verschiedene Rechenregeln erlauben es, Hoare-Tripel herzuleiten.

Leere Anweisung $\quad P \ \{ ; \} \ P$

Wenn die Zusicherung P vor der leeren Anweisung gilt, dann gilt sie auch danach.

Zuweisung $\quad P[x \leftarrow e] \ \{ x := e \} \ P$

Wenn vor der Zuweisung $x := e$ die Zusicherung P gilt, in der x durch e ersetzt wurde, dann gilt P danach.[3]

[1] Wir verwenden hier Algol-Syntax für die Programmfragmente, um zusätzliche Verwirrung durch die geschweiften Klammern von Java zu vermeiden.

[2] Die Terminierung muß also noch zusätzlich bewiesen werden. Es gibt eine Variante der obigen Notation, nämlich $\{P\} \ A \ \{Q\}$; diese hat die Bedeutung „wenn vor Ausführung von A die Zusicherung P gilt, dann terminiert A und danach gilt Q“.

[3] Variablen, die in P durch einen Quantor gebunden sind (z. B. das x in $\forall x : a[x] > 0$), werden nicht ersetzt.

Beispiel: $y+1>0 \; \{ \, x := y+1 \, \} \; x>0$:
Hier ist P die Zusicherung $x>0$. Wenn vor der Zuweisung $x := y+1$ die Zusicherung $y+1>0$ gilt (das ist, was man aus P erhält, wenn man x durch $y+1$ ersetzt), dann gilt nach der Zuweisung $x>0$.

Noch ein (triviales) Beispiel: $3=3 \; \{ \, z := 3 \, \} \; z=3$:
Wenn vor der Zuweisung $z := 3$ die Zusicherung $3=3$ gilt (was offensichtlich immer der Fall ist), dann gilt danach $z=3$.

Sequenz $$\frac{P \; \{ \, A \, \} \; Q \qquad Q \; \{ \, B \, \} \; R}{P \; \{ \, A; B \, \} \; R}$$

Wenn $P \; \{ \, A \, \} \; Q$ und $Q \; \{ \, B \, \} \; R$ gelten, dann können wir daraus $P \; \{ \, A; B \, \} \; R$ schließen (d, h., wenn vor der Anweisungssequenz $A; B$ die Zusicherung P gilt, dann gilt danach R).

Nachbedingung schwächen $$\frac{P \; \{ \, A \, \} \; Q \qquad Q \Rightarrow Q'}{P \; \{ \, A \, \} \; Q'}$$

Wenn $P \; \{ \, A \, \} \; Q$ gilt und Q' aus Q folgt, dann können wir daraus $P \; \{ \, A \, \} \; Q'$ schließen.

Vorbedingung stärken $$\frac{P' \Rightarrow P \qquad P \; \{ \, A \, \} \; Q}{P' \; \{ \, A \, \} \; Q}$$

Wenn P aus P' folgt und $P \; \{ \, A \, \} \; Q$ gilt, dann können wir daraus $P' \; \{ \, A \, \} \; Q$ schließen.

Auf den ersten Blick erscheinen diese beiden Regeln etwas unmotiviert. Welchen Nutzen soll es haben, erst eine stärkere Formel herzuleiten, um daraus dann eine stärkere Formel zu schließen? Eine Antwort auf diese Frage gibt es nach der folgenden Herleitungsregel für bedingte Anweisungen.

Bedingte Anweisung $$\frac{P \wedge B \; \{ \, T \, \} \; Q \qquad P \wedge \neg B \; \{ \, E \, \} \; Q}{P \; \{ \, \text{if } B \text{ then } T \text{ else } E \, \} \; Q}$$

Hier sehen wir, daß in den beiden Prämissen die gleiche Nachbedingung stehen muß, damit die Regel anwendbar ist. Sollten die Nachbedingungen verschieden sein, muß man gegebenenfalls mit Hilfe der Regel *Nachbedingung schwächen* nachhelfen.

Beispiel: Sei A die bedingte Anweisung

$$\text{if } x>0 \text{ then } y := x \text{ else } y := -x.$$

Wir wollen zeigen, daß (ohne irgendwelche Vorbedingungen) nach dieser Anweisung $y = |x|$ gilt, oder symbolisch, true $\{ A \}$ $y = |x|$. Nach der Regel für bedingte Anweisungen können wir dieses Hoare-Tripel aus den beiden Tripeln $x > 0 \{ y := x \}$ $y = |x|$ und $\neg x > 0 \{ y := -x \}$ $y = |x|$ herleiten:

$$\frac{x > 0 \; \{ y := x \} \; y = |x| \qquad \neg x > 0 \; \{ y := -x \} \; y = |x|}{\text{true} \; \{ A \} \; y = |x|}$$

Wie können wir die beiden Prämissen dieser Regel beweisen? Betrachten wir die zweite der beiden Prämissen (der Beweis für die erste erfolgt analog). Um hier die Regel für Zuweisungen anwenden zu können, müssen wir erst dafür sorgen, daß sich Vor- und Nachbedingung nur durch die Ersetzung von y durch $-x$ unterscheiden. Das heißt, wir zeigen zunächst eine stärkere Aussage, aus der wir $\neg x > 0 \{ y := -x \}$ $y = |x|$ schließen können, nämlich

$$-x = -x \wedge \neg x > 0 \; \{ y := -x \} \; y = -x \wedge \neg x > 0.$$

Diese Aussage folgt offenbar aus der Regel für Zuweisungen (die Vorbedingung ergibt sich aus der Nachbedingung, wenn man y durch $-x$ ersetzt). Da der erste Teil der Vorbedingung trivial ist, ist die Vorbedingung äquivalent zu $\neg x > 0$. Andererseits impliziert die Nachbedingung, wie man leicht sieht, $y = |x|$. Mittels der Regel *Nachbedingung schwächen* erhält man daher

$$\frac{\neg x > 0 \; \{ y := -x \} \; y = -x \wedge \neg x > 0 \qquad y = -x \wedge \neg x > 0 \Rightarrow y = |x|}{\neg x > 0 \; \{ y := -x \} \; y = |x|}$$

was genau der fehlende Beweisschritt ist.

Die letzte Regel des Hoare-Kalküls, die wir hier besprechen, ist die Regel für while-Schleifen:

While $$\frac{I \wedge B \; \{ L \} \; I}{I \; \{ \text{while } B \text{ do } L \} \; I \wedge \neg B}$$

Die Zusicherung I, die in dieser Regel auftaucht, nennt man Schleifeninvariante. Im Gegensatz zu den bisherigen Regeln, die man weitgehend mechanisch anwenden kann, erfordert das Finden einer sinnvollen Schleifeninvariante meist einige Kreativität. Es reicht nämlich nicht aus, ein I zu finden, für das $I \wedge B \{ L \} I$ gilt. Sinnvoll ist die Schleifeninvariante nur dann, wenn wir außerdem zeigen können, daß die Vorbedingung I vor der Schleife gilt, und wenn wir aus der Nachbedingung $I \wedge \neg B$ die gewünschte Eigenschaft der Schleife schließen können. Meist ist es einfacher, die Schleifeninvariante mit dem Programm zusammen zu entwickeln, als nachträglich eine Invariante zu suchen.

Als Beispiel betrachten wir hier das Programm *binarySearch* aus dem letzten Abschnitt. Bevor wir für die while-Schleife dieses Programms eine Schleifeninvariante angeben können, müssen wir aber das Programm zunächst leicht abändern. Der Hoare-Kalkül gilt nämlich nur für while-Schleifen, die auf genau eine Weise – bei Verletzung der Schleifenbedingung – verlassen werden können. Um das außerplanmäßige *return* aus dem Schleifenrumpf heraus zu eliminieren, führen wir eine boolesche Variable *found* ein:

```
static int binarySearch(int key, int[] a) {
  int bot = 0;                    // falls ein i existiert
  int top = a.length - 1;         //   mit a[i] == key, dann
                                  //   gilt bot <= i <= top.
  int mid;                        // Mitte des Intervalls.
  boolean found = false;          // Index gefunden?
  while (!found && bot<=top) {
    mid = (top + bot) / 2;        // Division mit Abrundung.
    if (a[mid] == key) {
      found = true;               // a[mid] == key, also
                                  //   gesuchter Index == mid.
    } else if (a[mid] < key) {
      bot = mid + 1;              // a[mid] < key, also
                                  //   gesuchter Index > mid.
    } else {
      top = mid - 1;              // a[mid] > key, also
                                  //   gesuchter Index < mid.
    }
  }
  if (found) {
    return mid;                   // Schleife verlassen,
                                  //   da Index gefunden.
  } else {
    return -1;                    // Schleife verlassen,
                                  //   da bot > top.
  }
}
```

Eine mögliche Schleifeninvariante I für die while-Schleife dieses Programms sieht wie folgt aus:

$$
\begin{aligned}
&a[\,] \text{ sortiert}\\
&\quad \wedge \Big((\mathit{found} \wedge a[\mathit{mid}] = \mathit{key})\\
&\qquad \vee (\neg \mathit{found} \wedge \forall i\colon (0 \le i < \mathit{bot} \vee \mathit{top} < i \le a.\mathit{length} - 1) \Rightarrow a[i] \ne \mathit{key}) \Big)
\end{aligned}
$$

Wie man durch Nachrechnen leicht zeigt, ist I tatsächlich eine Schleifeninvariante, es gilt also $I \wedge B$ $\{$ *Schleifenrumpf* $\}$ I. Außerdem ist I sinnvoll: Zum einen ist I zu Beginn des Algorithmus stets erfüllt: Da $\mathit{found} = \text{false}$, $\mathit{bot} = 0$ und $\mathit{top} = a.\mathit{length} - 1$, reduziert sich I zu „$a[\,]$ sortiert“, und das ist gerade die Voraussetzung für *binarySearch*. Zum zweiten gilt nach Verlassen der Schleife die Nachbedingung $I \wedge \neg B$, und das ist äquivalent zu

$$
\begin{aligned}
&a[\,] \text{ sortiert}\\
&\quad \wedge \Big((\mathit{found} \wedge a[\mathit{mid}] = \mathit{key})\\
&\qquad \vee (\neg \mathit{found} \wedge \forall i\colon (0 \le i \le a.\mathit{length} - 1) \Rightarrow a[i] \ne \mathit{key}) \Big)
\end{aligned}
$$

Wir können damit nun leicht zeigen, daß die if-Anweisung nach der while-Schleife das richtige Ergebnis liefert: mid, falls $\mathit{found} = \text{true}$, und -1 sonst.

— *Ende der Vorlesung* —