Unterabschnitt 3.3.3

Testen und Verifikation

Testen und Verifikation

Ein zentraler Teil der Software-Entwicklung besteht darin zu prüfen, ob die entwickelte Software auch den gestellten Anforderungen entspricht.

Überblick:

- Einführende Bemerkungen zur Qualitätssicherung
- Testen
- Verifikation von Programmeigenschaften

Qualitätssicherung: Kleine Einführung

Bei der Qualitätssicherung in der Softwareentwicklung spielen zwei Fragen eine zentrale Rolle:

- Wird das richtige System entwickelt?
- Wird das System richtig entwickelt?

Qualitätssicherung: Kleine Einführung (2)

Validation hat die Beantwortung der ersten Frage zum Ziel.
Beispielsweise ist zu klären, ob

- die benutzten Anforderungen die Vorstellungen des Auftraggebers richtig wiedergeben,
- die Anforderungen von allen Beteiligten gleich interpretiert werden,
- Unterspezifizierte Aspekte richtig konkretisiert wurden.

Qualitätssicherung: Kleine Einführung (3)

Validation prüft also die Übereinstimmung von Software mit den Vorstellungen der Auftraggeber/ Benutzer bzw. mit der Systemumgebung, in der die Software eingesetzt wird.

Unter ***Verifikation*** verstehen wir den Nachweis, dass Software bestimmte, explizit beschriebene Eigenschaften besitzt.

Beispiele:

- Mit Testen kann man prüfen, ob ein Programm zu gegebenen Eingaben die erwarteten Ausgaben liefert (Beschreibung: Testfälle).
- Mittels mathematischen Beweisen kann man zeigen, dass ein Programm für alle Eingaben ein bestimmtes Verhalten besitzt (Beschreibung: boolesche Ausdrücke, logische Formeln).
- Nachweis, dass ein Programm einen gegebenen Entwurf und die darin festgelegten Eigenschaften besitzt (Entwurfsbeschreibung).

Bemerkungen:

- Liegen die beschriebenen Eigenschaften in einer formalen Sprache vor, kann die Verifikation automatisiert werden.
- Zu prüfende Eigenschaften bei Funktionen:
  - Terminierung für zulässige Parameter
  - Verhalten wie im Entwurf festgelegt
- Grundsätzlich können sich Validation und Verifikation auf alle Phasen der Softwareentwicklung beziehen.
- Wir betrachten im Folgenden Testen und Verifikation durch Beweis anhand einfacher Beispiele im Kontext der funktionalen Programmierung. Eine systematischere Betrachtung von Aspekten der Qualitätssicherung ist Gegenstand von SE 2.

Begriffsklärung: (Testen)

Testen bedeutet die Ausführung eines Programms oder Programmteils mit bestimmten Eingabedaten.

Testen kann sowohl zur Validation als auch zur Verifikation dienen.

Bei funktionalen Programmen bezieht sich Testen überwiegend auf Eingabe- und Ausgabeverhalten von Funktionen.

Wir betrachten:

- Testen mit Testfällen
- Testen durch dynamisches Prüfen

Testen mit Testfällen

- Beschreibe das “Soll-Verhalten” der Software durch eine (endliche) Menge von Eingabe-Ausgabe-Paaren.
- Prüfe, ob die Software zu den Eingaben der Testfälle die entsprechenden Ausgaben liefert.

Beispiel:

Testen der folgenden Funktionsdeklaration:

```
fac :: Int -> Int
-- Berechnet fuer n in [0..12] die Fakultaet

fac 0 = 1
fac n = if  0<n || n <= 12
        then n * fac (n-1)
        else undefined
```

Testfälle:

0	→	1
12	→	479001600
13	→	Fehler: undefiniert/unzulaessiger Parameter
-1	→	Fehler: undefiniert/unzulaessiger Parameter
-1073741824	→	Fehler: undefiniert/unzulaessiger Parameter

Beobachtetes Verhalten:

0	→	1
12	→	479001600
13	→	1932053504
-1	→	*** Exception: stack overflow
-1073741824	→	*** Exception: stack overflow

Bemerkung:

- Das Verhalten von Funktionen mit unendlichem Argumentbereich kann durch Testen nur teilweise verifiziert werden. Testen kann im Allg. nicht die Abwesenheit von Fehlern zeigen.
- Wichtig ist die Auswahl der Testfälle. Sie sollten die “relevanten” Argumentbereiche abdecken.

Testen durch dynamisches Prüfen

- Beschreibe Eigenschaften von Zwischen- oder Ergebniswerten mit den Mitteln der Programmiersprache (meist boolsche Ausdrücke); d.h. implementiere Prüfprädikate.
- Rufe die Prüfprädikate an den dafür vorgesehenen Stellen im Programm auf.
- Lasse die Prüfprädikate in der Testphase des zu testenden Programms auswerten.

  Bei negativem Prüfergebnis muss ein Fehler erzeugt werden.

Anders als beim Testen mit Testfällen wird also das Verhalten des Programms an bestimmten Stellen automatisch während der Auswertung geprüft.

Prüfungen:

1. Prüfung der Zulässigkeit von Parametern beim Aufruf
2. Prüfung durch Ergebniskontrolle

Bemerkung:
Viele moderne Programmiersprachen bieten spezielle Sprachkonstrukte für das Testen durch dynamische Prüfung an.

Verifikation durch Beweis

Verifikation im engeren Sinne meint meist Verifikation durch Beweis. Hier verwenden wir den Begriff in diesem engeren Sinne.

Im Gegensatz zum Testen erlaubt Verifikation (durch Beweis) die Korrektheit zu zeigen, d.h. insbesondere die Abwesenheit von Fehlern.

Wir betrachten hier nur Programmverifikation, d.h. den Nachweis, dass ein Programm eine spezifizierte Eigenschaft besitzt.

Verifikation durch Beweis (2)

Die Spezifikation kommt üblicherweise aus dem Entwurf bzw. den Anforderungen.

Zwei zentrale Eigenschaften:

- Programm liefert die richtigen Ergebnisse, wenn es terminiert (***partielle Korrektheit***).
- Programm terminiert für die zulässigen Eingaben.

Beide Eigenschaften zusammen ergeben ***totale Korrektheit***.

Beispiele: (Spezifikation)

Spezifikation:
Eine Funktion `ggt` soll implementiert werden.
Für $m,\ n$ mit $m \geq 0,\ n \geq 0,\ m, n$ nicht beide null, soll gelten:

$$ggt\ \ m\ \ n = \max \{\ k \mid k \text{ teilt } m \text{ und } n\ \}$$

Beispiel: (Implementierung)

Euklidscher Algorithmus:

```
ggt :: Integer -> Integer -> Integer
-- m, n >= 0, nicht beide gleich 0

ggt m n = if m==0 then n else ggt (n `mod` m) m
```

Beispiel: (Verifikation von ggt)

Vorüberlegung:
Für $n \geq 0$, $m > 0$ gilt:

k teilt m und $n \quad \Leftrightarrow \quad k$ teilt m und k teilt $(n \bmod m)$

Induktion über den Parameterbereich:
Wir zeigen:

a) ggt ist korrekt für $m = 0$ und beliebiges n.

b) Vorausgesetzt: *ggt* ist korrekt für alle Paare (k, n) mit $k \leq m$ und n beliebig;
dann auch für alle Paare $(m + 1, n)$ mit n beliebig.

Beispiel: (Verifikation von ggt) (2)

- Ad (a) – Induktionsanfang:

```
  ggt 0 n
=
  n
=
  max{ k | k teilt n}
=
  max{ k | k teilt 0 und n}
```

Beispiel: (Verifikation von ggt) (3)

- Ad (b) – Induktionsschritt:

  Voraussetzung: Sei m gegeben.
  Für alle Paare (k, n) mit $k \leq m$ gilt: `ggt` ist korrekt für (k, n)

  Zeige: Für alle n gilt: ggt ist korrekt für $(m + 1, n)$!

```
    ggt (m+1) n
=    (* Deklaration von ggt *)
    ggt (n mod (m+1)) (m+1)
=    (* n mod (m+1) ≤ m und Induktionsvoraussetzung*)
    max { k |  k teilt (n mod (m+1)) und (m+1) }
=    (*Vorueberlegung*)
    max { k |  k teilt (m+1) und n}
QED.
```

Bemerkung:

So wie Testen Testfälle oder Prüfprädikate voraussetzt, so benötigt Verifikation mit Beweis eine Spezifikation oder andere Beschreibung der zu zeigenden Eigenschaften.

Äquivalente Funktionsdeklarationen:

Wir haben gesehen, dass unterschiedliche Funktionsdeklarationen das gleiche Ein- und Ausgabeverhalten haben können.

Zum Beispiel kann eine Deklaration in einer “aufwendigeren” Rekursionsformen einfacher zu lesen sein, aber eine entsprechende lineare oder repetitive Funktionsdeklaration performanter sein.

Transformiert man die eine in die andere Form ist es wichtig, die Äquivalenz zu zeigen.

Bemerkung:

- Semantisch äquivalente Programme können große Unterschiede bei der Effizienz aufweisen.
- Bedeutungserhaltende Transformationen spielen in der Programmoptimierung und dem Refactoring von Software eine wichtige Rolle.

Korrektheit von Transformationen:
Wir transformieren die rekursive Funktionsdeklarationen in einfachere Deklarationen und zeigen Äquivalenz.

Beispiel: (linear → repetitiv)

Wir betten die Fakultätsfunktion

```
fac :: Integer -> Integer
fac n  =  if n == 0 then 1 else n * fac (n-1)
```

in eine Funktion mit einem weiteren Argument ein, das Zwischenergebnisse aufsammelt:

```
facrep :: Integer -> Integer -> Integer
facrep n res =  if n==0  then  res
                         else  facrep (n-1) (res*n)
```

Damit lässt sich die Fakultät auch definieren als:

```
fac1 n = facrep n 1
```

Dadurch wurde eine lineare Rekursion in eine repetitive Form gebracht.

Lemma:

Für die obigen Deklarationen von `fac` und `facrep` gilt:

$\forall n$ mit $n \geq 0$, r mit $r \geq 0$: $\quad (fac\ \ n) * r = facrep\ \ n\ \ r$

insbesondere: $\forall n$ mit $n \geq 0$: $\quad fac\ \ n = fac1\ \ n$

Beweis: mittels Parameterinduktion nach *n*

Induktionsanfang:

Zu zeigen: $\forall r$ mit $r \geq 0 : \quad (fac\ \ 0) * r = facrep\ \ 0\ \ r$

```
   (fac 0) * r
=                  (* Deklaration von fac *)
   (if 0==0 then 1 else 0 * (fac (0-1))) * r
=                  (* Ausdrucksauswertung *)
   r
=                  (* Ausdrucksauswertung *)
   if 0==0 then r else facrep (0-1) (r*0)
=                  (* Deklaration von facrep *)
   facrep 0 r
```

Beweis: (Forts.)

Induktionsschritt: $k \to k+1$

Induktionsvoraussetzung:

Für $k \geq 0$: $\forall r$ mit $r \geq 0$: `(fac k)*r =facrep k r`

Zu zeigen: `fac(k+1)* r = facrep (k+1)r`

```
   fac(k+1) * r

=                  (* Deklaration von fac *)

   (if k+1==0 then 1 else (k+1) * fac (k+1-1)) * r

=                  (* Ausdrucksumformung *)

  (k+1) * (fac k) *r
```

Beweis: (Forts.)

```
   (k+1) * (fac k) *r
=     (* Kommutativitaet+Assoziativitaet der Multipl. *)

   ((fac k)* r) * (k+1)
=     (* Induktionsvoraussetzung *)

   (facrep k r) * (k+1))
=     (* Ausdrucksumformung *)

    if  k+1==0  then r else facrep k (r*(k+1))
=     (* Deklaration von facrep *)

    facrep (k+1) r
```

Beispiel: (kaskadenartig → linear)

Tranformation der Fibonacci-Funktion `fib` in eine lineare Form.

Idee:
Führe zwei zusätzliche Parameter ein, in denen ausgehend von 0 und 1 die Zwischenergebnisse berechnet werden.

Es soll also gelten:

```
fib1 n  =  fibemb n 0 1
```

Wir definieren `fibemb` zu:

```
fibemb 0 letzt res = 0
fibemb 1 letzt res = res
fibemb n letzt res = fibemb (n-1) res (letzt+res)
```

Beweis: mittels Parameterinduktion nach n.

Terminierung

Zentrale Eigenschaft einer Funktionsdeklaration ist, dass ihre Anwendung auf die *zulässigen* Parameter terminiert.

Diese Eigenschaft gilt für alle nicht-rekursiven Funktionsdeklarationen, die sich nur auf terminierende Funktionen abstützen.

Bei rekursiven Funktionsdeklarationen muss die Terminierung nachgewiesen werden.

Idee:
Die Parameter sollten bei jedem rekursiven Aufruf “kleiner” werden.

Beispiele: ("kleiner werdende" Parameter)

```
1. foldrplus [ ]     =  0
   foldrplus (x:xs) =  x + foldrplus xs

2. einfuegen [] z ix      =  [z]
   einfuegen xs z 0       =  z:xs
   einfuegen (x:xs) z ix =  einfuegen xs z (ix-1)

3. foo m n = if m<n then m `div` n else foo (m-n) n
```

Definition: (Ordnung)

Eine Teilmenge R von $M \times N$ heißt eine (binäre) ***Relation***.

Gilt $M = N$, dann nennt man R ***homogen***.

Eine homogene Relation heißt:

- ***reflexiv***, wenn für alle $x \in M$ gilt: $(x, x) \in R$
- ***antisymmetrisch***, wenn für alle $x, y \in M$ gilt:
  wenn $(x, y) \in R$ und $(y, x) \in R$, dann $x = y$
- ***transitiv***, wenn für alle $x, y, z \in M$ gilt:
  wenn $(x, y) \in R$ und $(y, z) \in R$, dann $(x, z) \in R$

Definition: (Ordnung) (2)

Eine reflexive, antisymmetrische und transitive homogene Relation auf $M \times M$ heißt eine (partielle) ***Ordnungsrelation***.

Eine Menge M mit einer Ordnungsrelation R heißt eine (partielle) ***Ordnung***.

Meist benutzt man Infixoperatoren wie $\leq$ (oder $\subseteq$) zur Darstellung der Relation und schreibt

$$x \leq y \quad \text{statt} \quad (x, y) \in R$$

Definition: (Kette, noethersche Ordnung)

Sei $(M, \leq)$ eine Ordnung. Eine Folge $\varphi : \mathbb{N} \to M$ heißt eine (abzählbar unendliche) ***aufsteigende*** Kette, wenn für alle $i \in \mathbb{N}$ gilt:

$$\varphi(i) \leq \varphi(i+1)$$

(absteigende Kette: entsprechend).

Eine Kette φ wird ***stationär***, falls es ein $j \in \mathbb{N}$ gibt, so dass

$$\varphi(j) = \varphi(j+k) \text{ für alle } k \in \mathbb{N}$$

Definition: (Kette, noethersche Ordnung) (2)

Sei N eine Teilmenge von M. $x \in N$ heißt:

- ***größtes*** Element von N, wenn $\forall y \in N$ gilt: $y \leq x$.
- ***kleinstes*** Element von N, wenn $\forall y \in N$ gilt: $x \leq y$.
- ***maximales*** Element von N, wenn $\forall y \in N$ gilt: $x \leq y$ impliziert $x = y$.
- ***minimales*** Element von N, wenn $\forall y \in N$ gilt: $y \leq x$ impliziert $x = y$.

Eine Ordnung $(M, \leq)$ heißt ***noethersch***, wenn jede nicht-leere Teilmenge von M ein minimales Element besitzt.

Lemma:

Eine Ordnung ist genau dann noethersch, wenn jede absteigende Kette stationär wird.

Beweis: (siehe Theorievorlesung)

Terminierungskriterium:

Sei $f : S \to T$ eine rekursive Funktionsdeklaration mit formalem Parameter n und sei P die Menge der zulässigen Parameter von f. Jede Anwendung von f auf Elemente von P terminiert,

- wenn es eine noethersche Ordnung $(M, \leq)$
- und eine Abb. $\delta : P \to M$ gibt,
- so dass für jede rekursive Anwendung $f(G(n))$ im Rumpf der Deklaration gilt:

  i) $G(n)$ ist ein zulässiger Parameter, d.h. $G(n) \in P$.

  ii) Die aktuellen Parameter werden echt kleiner, d.h.

$$\delta(G(n)) < \delta(n)$$

Bemerkung:

- Da die aktuellen Parameter nur endlich oft echt kleiner werden können (und dann stationär werden), garantiert das obige Kriterium die Terminierung.
- Um die Terminierung nachzuweisen, muss man also eine geeignete noethersche Ordnung und eine geeignete Abbildung δ finden.
- Ist der Argumentbereich bereits noethersch geordnet, kann δ selbstverständlich auch die Identität sein.

Beispiele: (Terminierungsbeweis)

```
1. foldrplus xs  =
     if  null xs then 0
                 else (head xs) + foldrplus (tail xs)
```

P ist die Menge aller *endlichen* Listen über `Integer`.

Noethersche Ordnung $(\mathbb{N}, \leq)$.

Als δ wähle die Funktion länge (Länge einer Liste).

Zu zeigen:

i) `tail xs` ist ein zulässiger Parameter: ok.

ii) `länge (tail xs) < länge xs`: ok.

Beispiele: (Terminierungsbeweis) (2)

```
2. einfuegen ([],z,ix)    =  [z]
   einfuegen (xs,z,0)     =  z:xs
   einfuegen (x:xs,z,ix)  =  einfuegen (xs,z,ix-1)
```

P ist die Menge aller Tripel aus `(a,[a],Int)`.

Noethersche Ordnung $(\mathbb{N}, \leq)$.

Als δ wähle die Funktion `längefst`, die `länge` auf die erste Komponente anwendet.

Zu zeigen:

i) `(xs,z,ix-1)` ist ein zulässiger Parameter : ok.

ii) `längefst(xs,z,ix-1) < längefst(x:xs,z,ix)` : ok.

Beispiele: (Terminierungsbeweis) (3)

Bemerkung:

Hätte man stattdessen für δ die Selektion auf die dritte Komponente gewählt, hätte man Terminierung nur für eine kleinere Menge zulässiger Parameter zeigen können, nämlich z.B. für Parametertripel `(xl,el,ix)` mit `ix` $\geq$ `0`.

Beispiele: (Terminierungsbeweis) (4)

```
3. foo(m,n) =
      if  m<n then m `div` n else foo (m-n,n)
```

P ist die Menge aller Paare (m, n) aus `(Integer,Integer)` mit ($m < n$ und $n \neq 0$) oder $n > 0$.

Noethersche Ordnung $(\mathbb{N}, \leq)$.

$\delta : Z \times Z \to \mathbb{N}$ mit

$$\delta(m,n) = \begin{cases} 0 & , \text{ falls } m < n \\ m - n + 1 & , \text{ falls } m \geq n \end{cases}$$

Beispiele: (Terminierungsbeweis) (5)

Zu zeigen:

i) $(m - n, n)$ ist ein zulässiger Parameter: ok.

ii) Unter der Voraussetzung $m \geq n$ und $n > 0$:

1. Fall: $m - n \geq n$:

   $\delta(m - n, n) = m - n - n + 1 < m - n + 1 = \delta(m, n)$

2. Fall: $m - n < n$:

   $\delta(m - n, n) = 0 < m - n + 1 = \delta(m, n)$

Bemerkung:

- Terminierungsbeweise sind bei der Entwicklung von Qualitätssoftware sehr wichtig, und zwar unabhängig vom verwendeten Modellierungs- bzw. Programmierparadigma.
- Es sollte zur Routine der Softwareentwicklung, gehören, den zulässigen Parameterbereich festzulegen und dafür Terminierung zu zeigen.