2.9 Funktionen/Funktionen testen

2.9.1 Warum testen?

Fast überall wird dem Testen von Programmen und Programmbestandteilen großer Wert zugemessen. Das hat mehrere Gründe:

a) Kaum ein Programm funktioniert auf Anhieb, wenn vorher nichts getestet wurde.

b) Selbst wenn es funktioniert, stecken meistens noch mehrere versteckte Fehler darin. Wenn der Anwender sie findet, hat das Konsequenzen.

c) Bei der Entwicklung kann es passieren, dass gutgemeinte Korrekturen oder Erweiterungen neue Fehler erzeugen: Das Programm funktioniert plötzlich nicht mehr.

Es gibt zwei Möglichkeiten, Fehler aufzuspüren:

a) Ein Systemtest testet das gesamte Programm (oder System, falls Hardware dazugehört).

b) Ein Komponententest testet Teile des Programms, z. B. eine Funktion.

Der Systemtest setzt meistens auf der Oberfläche auf, die der Kunde benutzt, er benutzt deshalb oft eine eigene Sprache. Der Komponententest dagegen findet auf der Ebene der Programmierung statt deshalb auch in derselben Programmiersprachen, z. B. C. Komponententests haben den weiteren Vorteil, dass sie versteckte Fehler finden können; außerdem finden sie Fehler sehr früh und sparen damit Zeit (und Kosten).

2.9.2 Systemtest – ein Beispiel

Das folgende Programm soll den Ersatzwiderstand der gemischten Schaltung nach Abbildung 1, also `R1||(R2+(R3||R4))` berechnen und ausgeben (`gemischt0.c`):

```c
#include <stdio.h>

double parallel(double rx, double ry)
{
    double rz;
    rz=rx*ry/(rx+ry);
    return rz;
}

int main(void)
{
    double r1, r2, r3, r4, rers;
    printf("Gemischte Schaltung R1||(R2+(R3||R4)):\n");
    printf("R1 in Ohm: "); scanf("%lf", &r1); while(getchar()!='\n'){}
    printf("R2 in Ohm: "); scanf("%lf", &r2); while(getchar()!='\n'){}
    printf("R3 in Ohm: "); scanf("%lf", &r3); while(getchar()!='\n'){}
    printf("R4 in Ohm: "); scanf("%lf", &r4); while(getchar()!='\n'){}
    rers=parallel(r1,r2+parallel(r3,r4));
    printf("Rers = %lf Ohm\n" , rers);
    return 0;
}
```

Ein Systemtest sieht nun so aus, dass man verschiedene Werte eingibt und nachsieht, ob das Ergebnis stimmt. Jeden Durchlauf mit einer neuen Wertekombination nennt man einen *Testfall*.

Terminal

```
schueler@debian964:~$ a.out
Gemischte Schaltung R1||(R2+(R3||R4)):
Eingabe R1 in Ohm: 100
Eingabe R2 in Ohm: 100
Eingabe R3 in Ohm: 100
Eingabe R4 in Ohm: 100
Rers = 60.000000 Ohm
schueler@debian964:~$ a.out
Gemischte Schaltung R1||(R2+(R3||R4)):
Eingabe R1 in Ohm: 100
Eingabe R2 in Ohm: 100
Eingabe R3 in Ohm: 0
Eingabe R4 in Ohm: 0
Rers = -nan Ohm
```

Hier ist offensichtlich ein Fehler vorhanden. Aber wo? Dazu schaut man sich die betroffenen Funktionen an, also `main()` und `parallel()`.

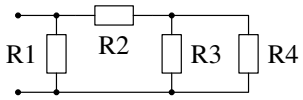

Abbildung 1: Gemischte Schaltung (π-Filter)

2.9.3 Komponententest mit getrenntem Test-Programm

Man kann die Funktion `parallel` aus dem alten Quelltext in ein neues Test-Programm kopieren; dieses Test-Programm dient nur dazu, die Funktion zu überprüfen.

Das ist vergleichbar mit einem Test-Aufbau, der ein elektrisches Modul prüfen soll. So ein Test-Aufbau enthält oft einen Generator, das Modul und ein Messgerät, das die Ausgangsgrößen des Moduls misst . Genauso ist es auch hier (`testparallel1.c`):

```
1  #include <stdio.h>
2
3  double parallel(double rx, double ry)
4  {
5      double rz;
6      rz=rx*ry/(rx+ry);
7      return rz;
8  }
9
10 int main(void)
11 {
12     double rx, ry, rz;
13     printf("Rx eingeben: ");
14     scanf("%lf", &rx);
15     printf("Ry eingeben: ");
16     scanf("%lf", &ry);
17     rz=parallel(rx,ry);
18     printf("Rz=%lg\n", rz);
19     return 0;
20 }
```

- Zeile 3–7: Hier ist das Testobjekt
- Zeile 13–16: Der Tester gibt die **Eingabe**daten auf der Tastatur ein
- Zeile 17: Aufruf des Testobjekts (**Verarbeitung** der Daten)
- Zeile 18: Der Tester erhält die **Ausgabe**daten auf dem Bildschirm

Eine Testsitzung sieht jetzt z. B. so aus:

Terminal

```
schueler@debian964:~$ gcc testparallel1.c
schueler@debian964:~$ a.out
Rx eingeben: 500
Ry eingeben: 500
Rz=250
schueler@debian964:~$ a.out
Rx eingeben: 5
Ry eingeben: 15
Rz=3.75
schueler@debian964:~$ a.out
Rx eingeben: 100
Ry eingeben: 0
Rz=0
schueler@debian964:~$ a.out
Rx eingeben: 0
Ry eingeben: 0
Rz=-nan
```

Und schon wurde ein Fehler gefunden (falls man aufpasst).

Diese Art des Komponententests hat ihre Vorteile:

- Der Tester kann die Ergebnisse ansehen und intuitiv immer neue Grenzen der Funktion austesten, und dies so lange, bis er die ersten Fehler gefunden hat.

Sie hat aber auch Nachteile:

- Jeder Testfall wird von Hand eingegeben und überprüft. Das erfordert Zeit.
- Der Tester kann Fehler übersehen.

2.9.4 Komponententest automatisieren

Deshalb sollen die Tests automatisiert werden. Dazu überlegt man sich zu jedem Testfall das gewünschte Ergebnis. Nur dann, wenn das errechnete Ergebnis nicht mit dem Wunsch übereinstimmt, folgt eine Ausgabe (`testparallel2.c`):

```
1  #include <stdio.h>
2
3  double parallel(double rx, double ry)
4  {
5      double rz;
6      rz=rx*ry/(rx+ry);
7      return rz;
8  }
9
10 int main(void)
11 {
```

```
12      if(parallel(500, 500)!=250) printf("Fehler 1!\n");
13      if(parallel(5,15)!=3.75)    printf("Fehler 2!\n");
14      if(parallel(100, 0)!=0)     printf("Fehler 3!\n");
15      if(parallel(0,0)!=0)        printf("Fehler 4!\n");
16      return 0;
17  }
```

Jetzt ist die Testsitzung viel kürzer:

Terminal

```
schueler@debian964:~$ gcc testparallel2.c
schueler@debian964:~$ a.out
Fehler 4!
```

2.9.5 Komponententest im bestehenden Programm

Die vorgestellte Art des Komponententests hat noch einige Nachteile:

- Für jede zu testende Komponente (Funktion) muss ein eigenes Test-Programm geschrieben werden.
- Die zu testende Komponente (Funktion) muss irgendwie in das Test-Programm hineinkommen. Es bietet sich an, dafür die `#include`-Direktive des C-Präprozessors zu verwenden.
- Das Test-Programm muss von Hand gestartet werden. Bei Änderungen an der Funktion könnte man vergessen, es erneut zu testen. Vielleicht hat man ja neue Fehler in die Funktion eingebaut ...

Deshalb werden die Test jetzt in das bestehede Programm (gemischte Schaltung) eingebaut. Schon zu Beginn der Funktion `main()` sollte man die Funktion `parallel()` testen. Am besten packt man alle Tests, die sich auf `parallel()` beziehen, in eine eigene kleine Funktion (damit sie nicht stören). Diese Testfunktion nennen wir z. B. `ftest_parallel()` und rufen sie zum Beginn von `main()` auf (`gemischt1.c`):

```
1   #include <stdio.h>
2
3   double parallel(double rx, double ry)
4   {
5       double rz;
6       rz=rx*ry/(rx+ry);
7       return rz;
8   }
9
10  void ftest_parallel(void)
11  {
12      if(parallel(500, 500)!=250) printf("Fehler 1!\n");
13      if(parallel(5,15)!=3.75)    printf("Fehler 2!\n");
14      if(parallel(100, 0)!=0)     printf("Fehler 3!\n");
15      if(parallel(0,0)!=0)        printf("Fehler 4!\n");
16  }
17
18  int main(void)
19  {
20      double r1, r2, r3, r4, rers;
21      ftest_parallel();
22
23      printf("Gemischte Schaltung R1||(R2+(R3||R4)):\n");
```

```
24      printf("R1 in Ohm: "); scanf("%lf", &r1); while(getchar()!='\n'){}
25      printf("R2 in Ohm: "); scanf("%lf", &r2); while(getchar()!='\n'){}
26      printf("R3 in Ohm: "); scanf("%lf", &r3); while(getchar()!='\n'){}
27      printf("R4 in Ohm: "); scanf("%lf", &r4); while(getchar()!='\n'){}
28      rers=parallel(r1,r2+parallel(r3,r4));
29      printf("Rers = %lf Ohm\n" , rers);
30      return 0;
31  }
```

Und wieder ist die Suche erfolgreich:

Terminal

```
schueler@debian964:~$ gcc gemischt1.c
schueler@debian964:~$ a.out
Fehler 4!
Gemischte Schaltung R1||(R2+(R3||R4)):
Eingabe R1 in Ohm: ...
```

2.9.6 Zusicherung mit `assert()`

Anstelle die Fehler durchzunummerieren, kann man auch die Nummer der Quelltextzeile ausgeben (siehe `src/gemischt2.c`):

```
1      if(parallel(500, 500)!=250) printf("Fehler in Zeile %d!\n", __LINE__);
```

Terminal

```
schueler@debian964:~$ gcc gemischt2.c
schueler@debian964:~$ a.out
Fehler in Zeile 15!
Gemischte Schaltung R1||(R2+(R3||R4)):
Eingabe R1 in Ohm: ...
```

Allerdings sind die `printf()`-Zeilen jetzt sehr lang und wiederholen sich. Deshalb gibt es ein Makro mit dem Namen `assert()`. Ein Makro ist eine Ersetzung mit `define`, hat aber Parameter wie eine Funktion. Man kann ein Makro deshalb wie eine Funktion anwenden. Das Makro `assert()` übernimmt jetzt also die Ausgabe von `printf()`.

Aber `assert()` kann noch mehr. *assert* heißt nämlich „stelle sicher“. Wenn der Wert in der Klammer TRUE entspricht (also ungleich null ist), dann ist alles in Ordnung und das Programm läuft weiter.

Steht dagegen in der Klammer der Wert FALSE (also null), liegt ein Fehler vor und es passieren zwei Dinge:

- Eine Fehlermeldung wird ausgegeben.
- Das Programm wird sofort abgebrochen.

Man spricht daher von einer *Zusicherung*: **Nach** der `assert()`-Zeile läuft das Program nur dann noch, falls der Wert in der Klammer TRUE war.

Hier ein Beispiel (siehe `src/assert_beispiel.c`):

```
1  #include <assert.h>
2  #include <stdio.h>
3  int berechne_kehrwert(int x)
4  {
5      int y;
6      assert(x!=0);
```

```
7     y=1/x;
8     return y;
9  }
10 int main(void)
11 {
12    int a=0, b;
13    b=berechne_kehrwert(a);
14    printf("%i\n", b);
15    return 0;
16 }
```

- Zeile 1: <assert.h> muss eingebunden werden, damit man assert benutzen kann.
- Zeile 6: Hier wird sichergestellt, dass x niemals null sein kann. Falls doch, wird das Programm sofort abgebrochen.
- Zeile 7: Wird nur erreicht, falls x nicht null ist. Somit ist ausgeschlossen, dass man hier durch null dividiert.

Terminal

```
schueler@debian964:~$ gcc assert_beispiel.c
schueler@debian964:~$ a.out
a.out: assert_beispiel.c:6: berechne_kehrwert: Assertion `x!=0' failed.
Abgebrochen
```

Netterweise werden also der Name des Programms, der Name der Quelltextdatei, die Zeilennummer, der Name der Funktion und der Inhalt der Zusicherung ausgegeben. Ein weiterer Vorteil von assert: Mit #define NDEBUG zum Beginn des Quelltextes kann man alle assert-Zeilen abschalten und das Programm damit schneller machen.

2.9.7 Komponententest mit assert()

Und nun zurück zur Parallelschaltung: Mit assert stellen wir sicher, dass alle Testfälle funktionieren (siehe src/gemischt3.c):

```
1 void ftest_parallel(void)
2 {
3     assert(parallel(500, 500) == 250);
4     assert(parallel(5, 15) == 3.75);
5     assert(parallel(100, 0) == 0);
6     assert(parallel(0,0) == 0);
7 }
```

Man beachte, dass die Logik nun positiv ist: In den Klammern von assert stehen lauter wahre Aussagen. Sichergestellt wird also, dass parallel() richtig funktioniert. Der bekannte Fehler wird jetzt so dargestellt:

Terminal

```
schueler@debian964:~$ gcc gemischt3.c
schueler@debian964:~$ a.out
a.out: gemischt3.c:15: ftest_parallel: Assertion `parallel(0,0)==0' failed.
Abgebrochen
```

2.9.8 Auf der Fehlersuche

Jetzt braucht man nur noch den Fehler zu beseitigen. Offensichtlich führt der Fall $x = y = 0$ in Zeile 6 zur Zuweisung `rz=0.0/0.0;`

Jetzt kann man diesen Fall entweder mit einer `if`-Abfrage vor Zeile 6 abfangen oder eine andere Formel verwenden:

```
1   rz=1.0/(1.0/rx+1.0/ry); // gemischt4.c, Zeile 6
```

Und da der Komponententest im Programm integriert ist, kann man gleich überprüfen, ob die Funktion ...

- beim vierten Testfall jetzt das richtige Ergebnis liefert (das war das Ziel),
- bei den bisherigen Testfällen immer noch funktioniert (man spricht dann von einem *Regressionstest*) – das ist nicht selbstverständlich.

Terminal

```
schueler@debian964:~$ gcc gemischt4.c
schueler@debian964:~$ a.out
Gemischte Schaltung R1||(R2+(R3||R4)):
Eingabe R1 in Ohm: ...
```